3 Problem Set 3

Problem 1.
(a) So we seek to write a function that can generate meshes and output

$$\text{p, t, e = pmesh(pv,hmax,nref)}$$

We are given the functions **delaunay(p)**, **all_edges(t)**, **boundary_nodes(t)**, **tplot(p, t)**, and **inpolygon(p, pv)**. Following the instructions, we must first create a function that divides the boundaries into nodes with a distance $\leq h_{max}$. Julia has a convenient function for this

$$\text{ceil(a/b)}$$

which outputs a value $\leq b$ which satisfies $a\%b = 0$. Thus, we can do this using the distance formula and appropriate iterate around the boundary lines.

```
function div_poly(polys_xy0,hmax)
    x0=polys_xy0[1,1]
    y0=polys_xy0[1,2]
    ret = [x0 y0]
    for i = 2:size(polys_xy0)[1]
        #println(polys[i,:])
        x=polys_xy0[i,1]
        y=polys_xy0[i,2]
        len=sqrt((x-x0)^2+(y-y0)^2)
        n=ceil(len/hmax)
        for j=1:n-1
          nx=(x0*(n-j)+x*j)/n
          ny=(y0*(n-j)+y*j)/n
          #println(nx,",",ny)
          ret = [ret; nx ny]
        end
        ret = [ret; x y]
        x0=x
        y0=y
    end
    return ret
end
```

This takes in the matrix pv of the polygon vertices, and the distance h_{max} we seek to make the node spacing. Next, we define a couple functions for the following parts, namely **centroid** for determining which triangles are outside the polygon, **area** for determining which triangle to add another node, **circumcenter** for determining where to add said node, and **get_new_tri** to omit the triangles that are outside of the domain. Firstly, we can find the centroid by using

$$O_x = \frac{1}{3}(A_x + B_x + C_x)$$

with A,B,C being the 3 vertices of the triangle and doing the same for the y coordinate respectively. Quite straightforwardly, this yields

```
function centroids(points, tri)
    ret = reshape([],0,2)
    for i = 1:size(tri)[1]
        nx = (points[tri[i,1],1]+points[tri[i,2],1]+points[tri[i,3],1])/3
        ny = (points[tri[i,1],2]+points[tri[i,2],2]+points[tri[i,3],2])/3
        ret = [ret; nx ny]
```

```
    end
    return ret
end
```

Next, we define the area function to find which triangle needs to have an additional node added. Note that we really only need the index of the triangle with the largest area, the actual areas do not matter in our situation. This can be calculated using the shoelace formula

$$a = \frac{1}{2}(A_x(B_y - C_y) + B_x(C_y - A_y) + C_x(A_y - B_y))$$

and constantly keeping a parameter on whichever triangle has the greatest area. Likewise, this is written by

```
function area(points,tri)
   ret = []
   max = 0.0
   maxindex = -1
   for i = 1:size(tri)[1]
       na = (points[tri[i,1],1]*(points[tri[i,2],2]-points[tri[i,3],2])
           + points[tri[i,2],1]*(points[tri[i,3],2]-points[tri[i,1],2])
           + points[tri[i,3],1]*(points[tri[i,1],2]-points[tri[i,2],2]))/2
       ret = [ret; na]
       if na>max
           max=na
           maxindex=i
       end
   end
   return ret, maxindex, max
end
```

Lastly, to find the circumcenter, we simply need one value, unlike the functions area and centroid where we indexed through all the triangles, we can determine the centroids through manipulation of their midpoints and then finding the intersection of the 2 bisectors. This is shown by

```
function circumcenter(p, t, it)
    ct = t[it,:]
    dp1 = p[ct[2], :] - p[ct[1], :]
    dp2 = p[ct[3], :] - p[ct[1], :]

    mid1 = (p[ct[2], :] + p[ct[1], :])/2
    mid2 = (p[ct[3], :] + p[ct[1], :])/2

    rhs = mid2-mid1
    s = [-dp1[2] dp2[2] ; dp1[1] -dp2[1]]\rhs
    cpc = mid1 + s[1] * [-dp1[2], dp1[1]]

    return cpc
end
```

where *it* represents the index of the triangle that we are targeting in any certain scenario. Finally, to omit the triangles that are determined to be outside of the polygon, we can simply copy the matrix over using the function

```
function get_new_tri(inside,t)
    t_new=zeros(Int64,0,3)
    for i = 1:size(t)[1]
        if inside[i]
```

```
            t_new = [t_new;t[i,:]']
        end
    end
    return t_new
end
```

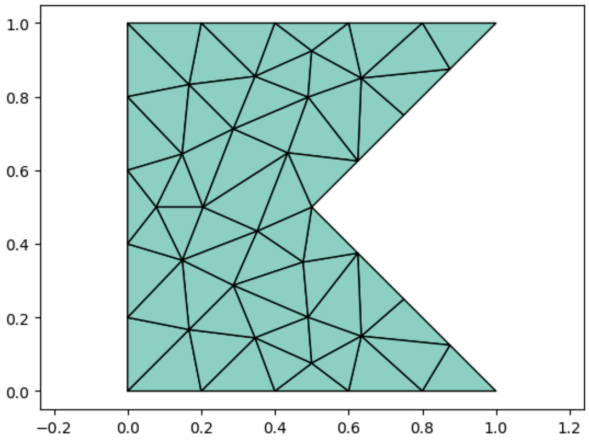
where inside is the vector outputted by the inpolygon function as given by the meshutilities. To put this all together, we run the first iteration of delaunay outside the loop, as it has to take into account the division of the boundary segments.

```
pvd = div_poly(pv,hmax)
t = delaunay(pvd)

centers = centroids(pvd,t)
inside = inpolygon(centers,pv)
t = get_new_tri(inside,t)
areas, maxindex, max = area(pvd,t)
```

Here, we have the established boundary nodes in pvd, and then identify and omit the triangles which are outside the boundary using the inpolygon function given to us. After this is initialized, we can simply run this in a loop until all areas are under $h_{max}^2/2$. This is expressed by

```
 while (max > 0.5*hmax.^2)
        added = circumcenter(pvd,t,maxindex)
        pvd = [pvd; added']
        t = delaunay(pvd)
        centers = centroids(pvd,t)

        inside = inpolygon(centers,pv)
        t = get_new_tri(inside,t)
        areas, maxindex, max = area(pvd,t)
end
```

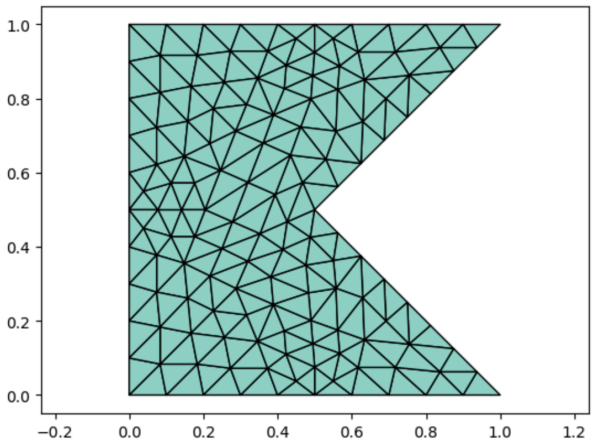
which simply keeps on re triangulating with the circumcenter node added of the largest triangle. From this, we can plot the figure given in step g, as shown below.



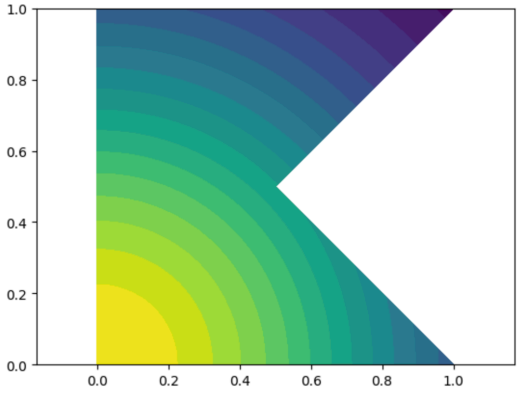
Lastly, we need to do the refinement. This can be accomplished by simply finding the average of the x y nodes on each of the edges called in the all_edges function. Note that we still have to run through which triangles are inside the polygon to omit those outside the boundary.

```
for i = 1:nref
        edges = all_edges(t)[1]
        for n = 1:size(edges)[1]
            node = (pvd[edges[n,1],:] + pvd[edges[n,2],:])/2.0
            pvd = [pvd;node']
        end
        t = delaunay(pvd)
        centers = centroids(pvd,t)
        inside = inpolygon(centers,pv)
        t = get_new_tri(inside,t)
end
```

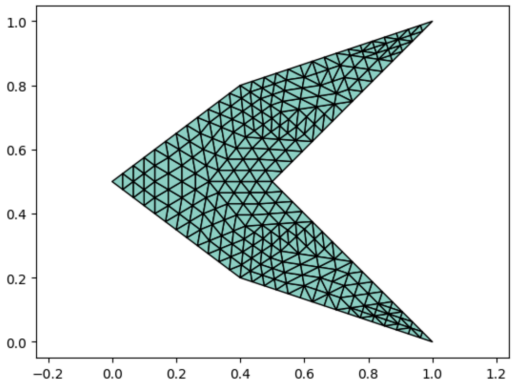
hence these 3 bits of code can be combined to make the function pmesh. A quick test of the given example in the problem set confirms this works, we get the following result



Quick test of some other polygons confirms this works reliably, and can plot functions.



(a) Plotting Function



(b) Different Polygon

Problem 2.

For this problem, we seek to propagate the solution of the Eikonal equations inwards. Using the discretization that is provided in the slides, as we only have 2 dimensions i and j rather than the 3 used in the slides, we can use the upwind method of the form

$$\phi_{ijk}^{n+1} = \phi_{ijk}^{n} - \Delta t \left(\max(F, 0) \nabla_{ijk}^{+} + \min(F, 0) \nabla_{ijk}^{-} \right)$$

Where

$$\nabla^+_{ijk} = [\max(D^{-x}\phi^n_{ijk}, 0)^2 + \min(D^{+x}\phi^n_{ijk}, 0)^2 + \max(D^{-y}\phi^n_{ijk}, 0)^2 + \min(D^{+y}\phi^n_{ijk}, 0)^2]$$

$$\nabla^-_{ijk} = [\min(D^{-x}\phi^n_{ijk}, 0)^2 + \max(D^{+x}\phi^n_{ijk}, 0)^2 + \min(D^{-y}\phi^n_{ijk}, 0)^2 + \max(D^{+y}\phi^n_{ijk}, 0)^2]$$

First defining the the four forward and backward step approximations we can quite straightforwardly get the functions

```
function D_mx(phi,i,j,h)
  if i>1
    return (phi[i-1,j]-phi[i,j])/(-h)
  else
    return 0.0
  end
end
function D_px(phi,i,j,h)
  return (phi[i+1,j]-phi[i,j])/(h)
end
function D_my(phi,i,j,h)
  if j>1
    return (phi[i,j-1]-phi[i,j])/(-h)
  else
    return 0.0
  end
end
function D_py(phi,i,j,h)
  return (phi[i,j+1]-phi[i,j])/(h)
end
```

to evaluate the discretization for the upwinded step for ϕ we can see that only ∇^+ or ∇^- is evaluated since of the max, min function. This can be evaluated through the code

```
if fv>0.0
      lap_p=sqrt(max(D_mx(phi,i,j,h),0)^2+min(D_px(phi,i,j,h),0)
      ^2+max(D_my(phi,i,j,h),0)^2+min(D_py(phi,i,j,h),0)^2)
      phi_n1[i,j]=phi[i,j]-dt*(fv*lap_p-1.0)
    else
      lap_m=sqrt(min(D_mx(phi,i,j,h),0)^2+max(D_px(phi,i,j,h),0)
      ^2+min(D_my(phi,i,j,h),0)^2+max(D_py(phi,i,j,h),0)^2)
      phi_n1[i,j]=phi[i,j]-dt*(fv*lap_m-1.0)
end
```

thus, we can use

```
  N=size(phi)[1]
  nz=spzeros(Int,N,N)
  phi_n1=spzeros(N,N)
  (I,J,V)=findnz(phi)
  #println(nz)
  for (i,j,v) in zip(I,J,V)
    #global nz
    nz[i,j]=1
    if i>1
      nz[i-1,j]=1
      if j>1
```

```
        nz[i-1,j-1]=1
      end
      if j<N
        nz[i-1,j+1]=1
      end
    end
    if i<N
      nz[i+1,j]=1
      if j>1
        nz[i+1,j-1]=1
      end
      if j<N
        nz[i+1,j+1]=1
      end
    end
    if j>1
      nz[i,j-1]=1
    end
    if j<N
      nz[i,j+1]=1
    end
  end
  (I,J,V)=findnz(nz)
```

to calculate the iterative step using the level set method. To actually iterate this along time, we can define a timestep and appropriately write

```
h = 1/100
N = ceil(Int,1.0/h)
phi=spzeros(N,N)
phi[floor(Int,0.2/h),floor(Int,0.2/h)]=1.0
phi[floor(Int,0.2/h),floor(Int,0.2/h)]=0.0
function F(x,y)
    if y < 50
        return 0.5
    else
        return 1
    end
end
dt=0.0001
for k=1:500
  global phi, F, dt, h
  phi=cal_phi_n1(phi,F,dt,h)
  println("t=",k*dt)
  #println(phi)
  println("-------------")
end
```

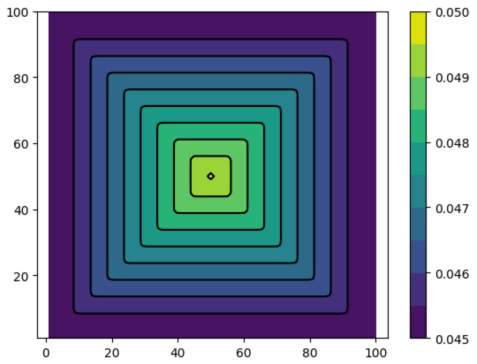
From the equation provided, we can solve

$$\vec{r} = n \times dt$$
$$\vec{r} = \frac{\nabla \phi}{|\nabla \phi|}$$

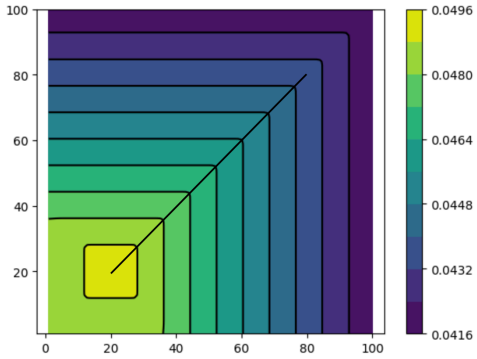
and using numerical methods,

$$\frac{\frac{\phi_{i+1,j}-\phi_{i-1,j}}{2h}, \frac{\phi_{i+1,j+1}-0}{2h}}{\sqrt{(\frac{\phi_{i+1,j}-\phi_{i-1,j}}{2h})^2 + (\frac{\phi_{i+1,j+1}-0}{2h})^2}} \times dt$$

For instance, if we just use the center point .5 .5 in the unit square, we can see that the solution propagates, giving a contour graph of the form



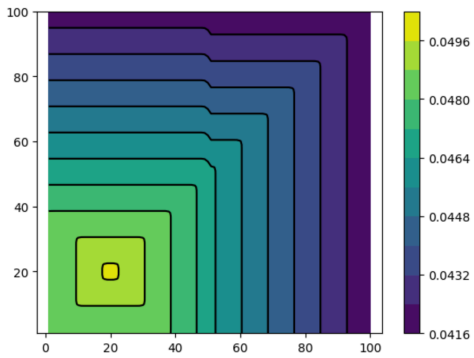
Following this, we can use the test case provided to get the following for a departure of 0.2 0.2 and an arrival of 0.8 0.8



Some of the other cases notes that the values of F are changing, which can be accounted for simply by introducing a function for F, for instance

```
function F(x,y)
    return 1 - 0.9 * (cos(4pi*x)*(2.71828)^(-10*((x-.5)^2+ (y-.5)^2)))
end
```

or whatever is appropriate for F. For the piece wise function of F, we can see that this results in a staggered contour around 0.5, as the speed values are different around this boundary.



4 Problem Set 4

Problem 1.
(a) To get the Galerkin formulation, we first multiply each side of the original equation by a dummy function $v(x)$ and then take find the integral along the bounds.

$$u''''(x) = f(x) = 480x - 120$$

$$\int_\Omega u''''(x)v(x)dx = \int_\Omega f(x)v(x)dx$$

Integrating by parts 2 times and plugging in the domain that is specified in the problem yields

$$v(1)u'''(1) - v(0)u'''(0) - v'(1)u''(1) + v'(0)u''(0) - \int_0^1 u''(x)v''(x)dx = \int_0^1 f(x)v(x)dx$$

Since $v(0) = v'(0) = v'(1) = v(1) = 0$, all the leftmost terms outside the integral evaluate to 0, and we get the Galerkin formula desired in the problem,

$$\int_0^1 u''(x)v''(x)dx = \int_0^1 f(x)v(x)dx$$

(b) The basis function can be found by a manipulation of the conditions that it needs to fulfill, specifically given the nodes that it has the different elements on K on, we have

$$\phi_i(x) = \frac{(x - x_{i+1})^2(a(x - x_i) + 1)}{(x - x_{i+1})^2}$$

Expanding for the function a, we have

$$\phi_1(x) = \frac{(x - x_{i-1})^2\left(\frac{2(x-x_i)}{x_{i-1}-x_i} + 1\right)}{(x_i - x_{i-1})^2} = -16x^3 + 12x^2$$

as we are solving for the range between 0 and 0.5 for x in this specific basis function component.

$$\phi_2(x) = \frac{(x - x_{i+1})^2\left(\frac{2(x-x_i)}{x_{i+1}-x_i} + 1\right)}{(x_i - x_{i+1})^2} = 16x^3 - 36x^2 + 24x - 4$$

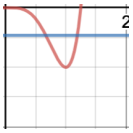
as we are solving for the range between 0.5 and 1 for x in this specific basis function component. ϕ_1 and ϕ_2 denote the 2 basis function components.
(c) Evaluating the functions

$$a_{ij} = \int_0^1 \phi_i''\phi_j''dx$$

we get

$$\begin{bmatrix} -8 & 19.2 \\ 19.2 & -8 \end{bmatrix}\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} 1/2 \\ 1/2 \end{bmatrix}$$

which can then be solved for the numerical solutions.



Problem 2.
We want to find a function that can solve the equation

$$-\nabla^2 u(x,y) = 1$$

on the domain that is provided in each of the three test cases, which will be triangulated through the pmesh function established in the previous homework. As shown by the finite element notes, are Galerkin form is given by

$$\int_\Omega \nabla u_h \cdot \nabla v dx = \int_\Omega f v dx + \oint_\Gamma g v ds$$

However, as all our Neumann conditions are of the form

$$n \cdot \nabla u = 0$$

we won't have the last term shown here. Considering a single triangular element T^k, with vertices x_1^k, x_2^k, x_3^k we have the linear basis functions of the form

$$\varphi_\alpha^k = c_\alpha^k + c_{x,\alpha}^k x + c_{y,\alpha}^k y$$

This can be solved as a linear system given by

$$\begin{pmatrix} 1 & x_1^k & y_1^k \\ 1 & x_2^k & y_2^k \\ 1 & x_3^k & y_3^k \end{pmatrix} \begin{pmatrix} c_\alpha^k \\ c_{x,\alpha}^k \\ c_{y,\alpha}^k \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

and replacing the RHS with respective columns of the 3x3 identity matrix to get a total of 9 coefficients. We can solve this through introducing simple evaluations for both the RHS and LHS of this equation, writing it into a function of the form

```
function coefficients(RHS, p, t, k)
    LHS = [1 p[t[k,1],1] p[t[k,1],2]; 1 p[t[k,2],1] p[t[k,2],2]; 1 p[t[k,3],1] p[t[k,3],2]]
    coeff = LHS\RHS'
    return coeff'
end
```

which returns the 9 coefficients in a 3x3 matrix for ease of indexing. The elementary matrix as referenced in the notes then becomes

$$A_{\alpha\beta}^k = \int_{T^k} \frac{\partial \varphi_\alpha^k}{\partial x}\frac{\partial \varphi_\beta^k}{\partial x} + \frac{\partial \varphi_\alpha^k}{\partial y}\frac{\partial \varphi_\beta^k}{\partial y} dx = \text{Area}^k (c_{x,\alpha}^k c_{x,\beta}^k + c_{y,\alpha}^k c_{y,\beta}^k)$$

Using the previously established shoelace formula that was inside the pmesh function, we can get the area of the triangle in question and the coefficients can be taken by the 3x3 matrix established above. Disregarding the area, this value is given by

```
function element(coeffmatrix, a, b)
    return coeffmatrix[a,2]*coeffmatrix[b,2] + coeffmatrix[a,3]*coeffmatrix[b,3]
end
```

In order to implement this, we have to effectively run the stamping method for each of the triangles that we have in the mesh given by the array t from the pmesh output. After establishing the necesary coefficients, we can use

```
A[t[k,:],t[k,:]] = A[t[k,:],t[k,:]]+insert
#aa = areas[k]/3.0
b[t[k,:]] = b[t[k,:]].+ areas[k]/3.0
```

to "stamp" the values of the local matrices into the global one. This results in a established code of

```
areas = triarea(p,t)
n = size(p)[1]
A = spzeros(n, n); b = zeros(n);
for k = 1:size(t)[1]
    cm = [coefficients([1 0 0] , p, t, k) ;
          coefficients([0 1 0] , p, t, k) ;
          coefficients([0 0 1] , p, t, k) ]
    insert = [element(cm, 1, 1) element(cm, 1, 2) element(cm, 1, 3);
              element(cm, 2, 1) element(cm, 2, 2) element(cm, 2, 3);
              element(cm, 3, 1) element(cm, 3, 2) element(cm, 3, 3)]
    insert *=  areas[k]

    A[t[k,:],t[k,:]] = A[t[k,:],t[k,:]]+insert
    #aa = areas[k]/3.0
    b[t[k,:]] = b[t[k,:]].+ areas[k]/3.0
end
```

After this, we establish the Dirichelt boundary conditions which are given along the nodes which are specified by the vector e that was outputted from the pmesh function. We go along each node that is expressed in this vector, and set $u_{h,i} = 0$ which can be accomplished by the code

```
for k = 1:size(e)[1]
    i=e[k]
    A[i,:].=0.0
    A[i,i]=1.0
    b[i]=0.0
end
dropzeros!(A)
```

Combining this all we get the following code for the **fempoi** function as a whole

```
using SparseArrays
using LinearAlgebra
function coefficients(RHS, p, t, k)
    LHS = [1 p[t[k,1],1] p[t[k,1],2]; 1 p[t[k,2],1] p[t[k,2],2]; 1 p[t[k,3],1] p[t[k,3],2]]
    coeff = LHS\RHS'
    return coeff'
end

function element(coeffmatrix, a, b)
    return coeffmatrix[a,2]*coeffmatrix[b,2] + coeffmatrix[a,3]*coeffmatrix[b,3]
end

function fempoi(p, t, e)
    areas = triarea(p,t)
    n = size(p)[1]
    A = spzeros(n, n); b = zeros(n);
    for k = 1:size(t)[1]
        cm = [coefficients([1 0 0] , p, t, k) ;
              coefficients([0 1 0] , p, t, k) ;
              coefficients([0 0 1] , p, t, k) ]
        insert = [element(cm, 1, 1) element(cm, 1, 2) element(cm, 1, 3);
                  element(cm, 2, 1) element(cm, 2, 2) element(cm, 2, 3);
                  element(cm, 3, 1) element(cm, 3, 2) element(cm, 3, 3)]
```

```
        insert *=  areas[k]

        A[t[k,:],t[k,:]] = A[t[k,:],t[k,:]]+insert
        #aa = areas[k]/3.0
        b[t[k,:]] = b[t[k,:]].+ areas[k]/3.0
    end
    for k = 1:size(e)[1]
        i=e[k]
        A[i,:].=0.0
        A[i,i]=1.0
        b[i]=0.0
    end
    dropzeros!(A)
    println(b)
    println(A)

    return A \ b
end
```

Using the 3 test cases that are provided on the problem sheet, we get roughly the same answers as shown below.

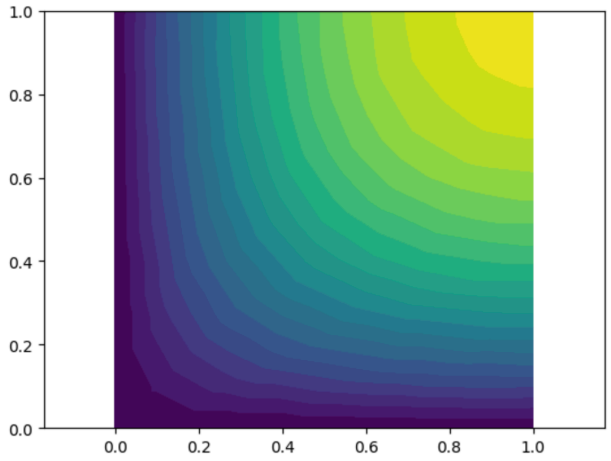


Figure 6: square with left/bottom Dirichlet

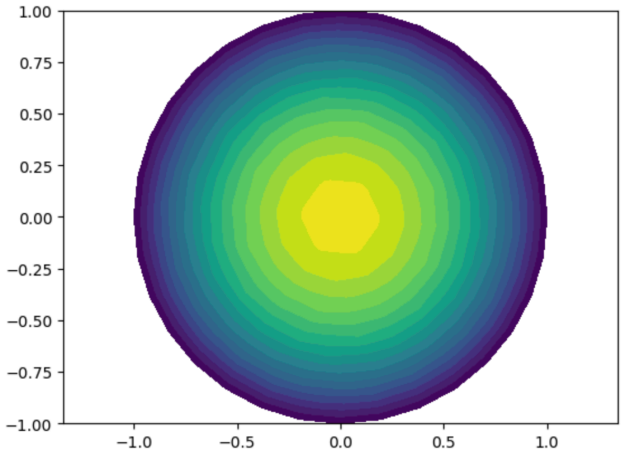


Figure 7: circle with Dirichlet

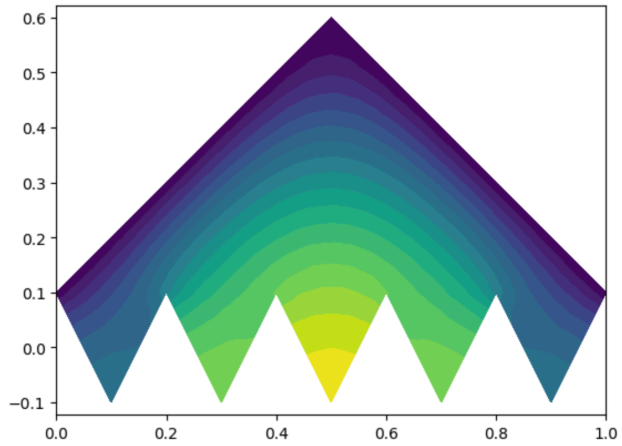


Figure 8: polygon with mix of both

Problem 3.
For the error function, the general formula for this we use the max norm of the difference between the u values for the 2 nodes. We can do this simply because the respective specifications are simply adding on to the end of the matrix. This results in simply comparing the truncated matrix of the higher nref solution with the lower ones. The basic formula for max norm can be seen by

```
errorarray = maximum(abs.(umax[1:size(uprox)[1]] - uprox))
```

Repeating this for each of the iterations from 0 to nref, we get to output an array with the error vector.

```
function poiconv(pv, hmax, nrefmax)
   pmax, tmax, emax = pmesh(pv, hmax, nrefmax)
   umax = fempoi(pmax, tmax, emax)
   errors = zeros(nrefmax)
   for ncur = 0:(nrefmax-1)
       pprox, tprox, eprox = pmesh(pv, hmax, ncur)
       uprox = fempoi(pprox, tprox, eprox)
       errorarray = maximum(abs.(umax[1:size(uprox)[1]] - uprox))
       #errors = [errors ; errorarray]
```

```
        errors[ncur+1]=errorarray
    end
    return errors
end
```

Using the function provided by the problem set

```
hmax = 0.15
pv_square = Float64[0 0; 1 0; 1 1; 0 1; 0 0]
pv_polygon = Float64[0 0; 1 0; .5 .5; 1 1; 0 1; 0 0]

errors_square = poiconv(pv_square, hmax, 3)
errors_polygon = poiconv(pv_polygon, hmax, 3)
errors = [errors_square errors_polygon]

error2 = errors[1, :]


clf()
loglog(hmax ./ [1,2,4], errors)
rates = @. log2(errors[end-1,:]) - log2(errors[end,:])
```

we can see the convergence plots are given by

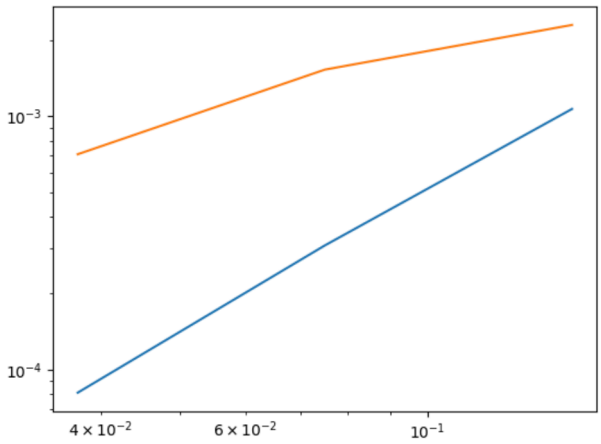


Figure 9: convergence plots

And get the rates of $1.930788150433111, 1.1096771919045025$ as the output as specified by the problem statement.

5 Problem Set 5

Problem 1.
(a) We want to find a Galerkin formulation for the given system, with the 3 boundary Neumann conditions. To do this, we integrate on the respective boundaries and multiplying by a test function v, we get

$$\int_\Omega -\nabla^2 u_h v dx = \int_\Omega k^2 u_h v dx$$

$$\int_\Omega -\nabla^2 u_h v dx = \int_\Omega k^2 u_h v dx + \oint_{\Gamma_{out}} (-iku_h) v ds + \oint_{\Gamma_{in}} (2ik - iku_h) v ds$$

Thus, the Galerkin formulation is given by: Find a function $u_h \in V_h$ that satisfies

$$\int_\Omega \nabla u_h \cdot \nabla v dx = \int_\Omega k^2 u_h v dx + \oint_{\Gamma_{out}} (-iku_h) v ds + \oint_{\Gamma_{in}} (2ik - iku_h) v ds$$

(b) We replace $u_h = \sum_{j=1}^N u_j \varphi_j$ and $v = \varphi_i(x)$ yields

$$\int_\Omega [\sum_{j=1}^N u_j \nabla \varphi_j] \cdot \nabla \varphi_i(x) dx = \int_\Omega k^2 [\sum_{j=1}^N u_j \varphi_j] \varphi_i dx - \oint_{\Gamma_{out}} ik [\sum_{j=1}^N u_j \varphi_j] \varphi_i ds + \oint_{\Gamma_{in}} (2ik - ik[\sum_{j=1}^N u_j \varphi_j]) \varphi_i ds$$

Factoring out the u vector with a rearrangement of terms and switching the summation and integral gives

$$\sum_{j=1}^N u_j \left(\int_\Omega \nabla \varphi_i \cdot \nabla \varphi_j dx - k^2 \int_\Omega \varphi_i \varphi_j dx + ik \left(\oint_{\Gamma_{out}} \varphi_i \varphi_j ds + \oint_{\Gamma_{in}} \varphi_i \varphi_j ds \right) \right) = 2ik \oint_{\Gamma_{in}} \varphi_i ds$$

thus, this gives the discretization of the form $\mathbf{Au} = \mathbf{b}$ where $A = K - k^2 M + ik(B_{out} + B_{in})$ and $\mathbf{b} = 2ik\mathbf{b}_{in}$

$$K = \int_\Omega \nabla \varphi_i \cdot \nabla \varphi_j dx \quad M = \int_\Omega \varphi_i \varphi_j dx \qquad B_{out} = \oint_{\Gamma_{out}} \varphi_i \varphi_j ds \qquad B_{in} = \oint_{\Gamma_{in}} \varphi_i \varphi_j ds \qquad \mathbf{b}_{in} = \oint_{\Gamma_{in}} \varphi_i ds$$

which can be seen in the discretization above.
(c) The transmit identity can be shown by $u^H B_{out} u$ and expanding

$$H(u) = \int_{\Gamma_{out}} |u|^2 ds$$

$$H(u) = u^H B_{out} u = u_i u_j \oint_{\Gamma_{out}} \varphi_i \varphi_j ds = \oint_{\Gamma_{out}} (u_i \varphi_i)(u_j \varphi_j) ds = \oint_{\Gamma_{out}} |u|^2 ds$$

as the hermitian transpose can be rearranged into u.

Problem 2.
(a) Quite straightforwardly, we can see that the solution $u(x, y) = e^{-ikx}$ works since

$$-\nabla^2 (e^{-ikx}) - k^2 (e^{-ikx}) = 0$$

All the boundary conditions hold aswell as $u' = -ike^{-ikx}$, using the appropriate normal vectors gives

$$[0, \pm 1][-ike^{-ikx}, 0] = 0$$

$$[1, 0][-ike^{-ikx}, 0] = -iku$$

$$[-1, 0][-ike^{-ikx}, 0] = 2ik - iku$$

for the boundary conditions on $\Gamma_{wall}, \Gamma_{out}, \Gamma_{in}$ the first 2 evidently hold for all x, while the last one holds for specifically the $x = 0$ boundary on Γ_{in}, as $ike^{-ik(0)} + ike^{-ik(0)} = 2ik$

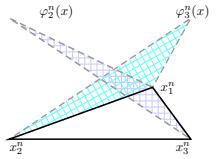
(b) Given the appropriate boundary conditions stated by the problem, we can use the **all_edges** function to find the appropriate boundary edges whose nodes are on the given boundaries. This is accomplished by looping through all the edges and testing whether it is a vertical edge on $x = 0$ for in, $x = 5$ for out, and all others are classified as wall

```
function waveguide_edges(p,t)
    elist, bedgeindex = all_edges(t)
    bnode = boundary_nodes(t)
    ein = []
    eout = []
    ewall = []
    for i = 1:size(bedgeindex)[1]
        if p[elist[bedgeindex[i],1],1] == 0 && p[elist[bedgeindex[i],2],1] == 0
            ein = [ein; bedgeindex[i]]
        elseif p[elist[bedgeindex[i],1],1] == 5 && p[elist[bedgeindex[i],2],1] == 5
            eout = [eout; bedgeindex[i]]
        else
            ewall = [ewall; bedgeindex[i]]
        end
    end
    ein = elist[ein,:]
    eout = elist[eout,:]
    ewall = elist[ewall,:]
    return ceil.(Int,ein), ceil.(Int,eout), ceil.(Int,ewall)
end
```

(c) For K, we have the same formula from the PS4 FEM problem, as the dot product $\nabla\varphi_i \cdot \nabla\varphi_j$ can be shown to be expressed as

$$K_{ij}^n = \int_{T_n} c_{x,i}^n c_{x,j}^n + c_{y,i}^n c_{y,j}^n dx = \text{Area}^n (c_{x,i}^n c_{x,j}^n + c_{y,i}^n c_{y,j}^n)$$

for the basis functions φ_i and φ_j, and the triangle element T^n. Next, we can find the mass matrix M to be expressed by a product of 2 given basis functions i, j.



Following the quadrature expression that is given for quadratic elements, we can effectively evaluate the mass matrix M with similar methods that are used in problem four. Thus, we have the form

$$M_{ij}^n = \int_{T_n} \varphi_i \varphi_j d\mathbf{x} = \frac{\text{Area}^k}{3}(f(\frac{4x_1 + x_2 + x_3}{6}) + f(\frac{x_1 + 4x_2 + x_3}{6}) + f(\frac{x_1 + x_2 + 4x_3}{6}))$$

where $f = \varphi_i \varphi_j$. Each of the following line integrals are computed in similar ways. As we know that at the boundaries Γ_{in} and Γ_{out} that x is strictly constant, and y goes from 0 to 1, we can directly compute the following line integrals as $\varphi_i \varphi_j$ can be computed numerically. Thus we get a 2 by 2 matrix for each set of boundary point form an edge, which are then subsequently stamped onto their respective nodes.

$$\begin{aligned}
B_{out} &= \oint_{\Gamma_{out}} \varphi_i \varphi_j ds \\
&= \int_{y_i}^{y_j} (c_i^k + c_{x,i}^k x + c_{y,i}^k y)(c_j^k + c_{x,j}^k x + c_{y,j}^k y) dy \\
&= (c_i^k c_j^k) y + (c_i^k c_{x,j}^k + c_j^k c_{x,i}^k) xy + \frac{1}{2} y^2 (c_{y,i}^k c_j^k + c_{y,j}^k c_i^k) \\
&\quad + \frac{1}{2} y^2 x (c_{x,i}^k c_{y,j}^k + c_{x,j}^k c_{y,i}^k) + c_{x,j}^k c_{x,i}^k x^2 y + \frac{1}{3} y^3 c_{y,j}^k c_{y,i}^k \Big|_{y_i}^{y_j}
\end{aligned}$$

In this specific case, the x values would all be a constant above 0, as this is the Γ_{out} boundary, whereas the other in boundary would see the x values become 0. The element load vector is calculated in a similar way, just with only one basis function. Our coefficient matrix is given by

```
function coeffmx(RHS, p, t, k)
    nm = p[t[k,1:3],:]
    LHS = zeros(3,3)
    for n = 1:3
        LHS[n,:] = [1 nm[n,1] nm[n,2]]
    end
    return (LHS\RHS)
end
```

and the individual entries for the matrices K and M are given by the functions

```
function kentry(cmx, i, j, ak)
    return ak*(cmx[2,i]*cmx[2,j] + cmx[3,i]*cmx[3,j])
end
function mentry(cmx, i, j, ak, p, t, k)
    f(x,y) = (cmx[1,i] + cmx[2,i]*x + cmx[3,i]*y)*(cmx[1,j] + cmx[2,j]*x + cmx[3,j]*y)
    result = 0.0
    for n = 1:3
        base = (p[t[k,1],:] + p[t[k,2],:] + p[t[k,3],:])/6 + p[t[k,n],:]/2
        result = result + f(base[1],base[2])
    end
    result = (result*ak)/3
    return result
end
```

On the other hand, the line integrals are a bit more complicated to compute, as we need to index the two nodes i, j and the triangular element which they are both part of, k. This is done by introducing an additional function specifically for such indexing

```
function find_bnd(e,t)
  ind=intersect(hcat(getindex.(findall(x -> x==e[1], t),1)),hcat(getindex.(findall(x -> x==e[2],t),1)))
  i=0
  j=0
  for k=1:3
    if t[ind[1],k]==e[1]
      i=k
    elseif t[ind[1],k]==e[2]
      j=k
    end
  end
  return ind[1],i,j
end

function boundsint(p, t, e)
    k, i, j = find_bnd(e, t)
    cmx = coeffmx(Matrix{Float64}(I, 3, 3), p, t, k)
    cross = 0
    f(x,y) = cmx[1,i]*cmx[1,j]*y + x*y*(cmx[1,j]*cmx[2,i] + cmx[1,i]*cmx[2,j]) +
    (y^2*(cmx[3,i]*cmx[1,j] + cmx[3,j]*cmx[1,i]))/2 + (y^2*x*(cmx[2,i]*cmx[3,j] +
    cmx[2,j]*cmx[3,i]))/2 + cmx[2,i]*cmx[2,j]*x^2*y + (cmx[3,i]*cmx[3,j]*y^3)/3
    h(x,y) = cmx[1,i]*cmx[1,i]*y + x*y*(cmx[1,i]*cmx[2,i] + cmx[1,i]*cmx[2,i]) +
    (y^2*(cmx[3,i]*cmx[1,i] + cmx[3,i]*cmx[1,i]))/2 + (y^2*x*(cmx[2,i]*cmx[3,i] +
    cmx[2,i]*cmx[3,i]))/2 + cmx[2,i]*cmx[2,i]*x^2*y + (cmx[3,i]*cmx[3,i]*y^3)/3
    g(x,y) = cmx[1,j]*cmx[1,j]*y + x*y*(cmx[1,j]*cmx[2,j] + cmx[1,j]*cmx[2,j]) +
    (y^2*(cmx[3,j]*cmx[1,j] + cmx[3,j]*cmx[1,j]))/2 + (y^2*x*(cmx[2,j]*cmx[3,j] +
    cmx[2,j]*cmx[3,j]))/2 + cmx[2,j]*cmx[2,j]*x^2*y + (cmx[3,j]*cmx[3,j]*y^3)/3
    if p[t[k,i],2] > p[t[k,j],2]
        cross = f(p[t[k,i],1],p[t[k,i],2]) - f(p[t[k,i],1],p[t[k,j],2])
        id = h(p[t[k,i],1],p[t[k,i],2]) - h(p[t[k,i],1],p[t[k,j],2])
        jd = g(p[t[k,i],1],p[t[k,i],2]) - g(p[t[k,i],1],p[t[k,j],2])
    elseif p[t[k,i],2] < p[t[k,j],2]
        cross = f(p[t[k,i],1],p[t[k,j],2]) - f(p[t[k,i],1],p[t[k,i],2])
        id = h(p[t[k,i],1],p[t[k,j],2]) - h(p[t[k,i],1],p[t[k,i],2])
        jd = g(p[t[k,i],1],p[t[k,j],2]) - g(p[t[k,i],1],p[t[k,i],2])
```

```
    end
    return [id cross; cross jd]
end
```

which can be then combined by iterating through each element, creating our desired function, stamping each element independently

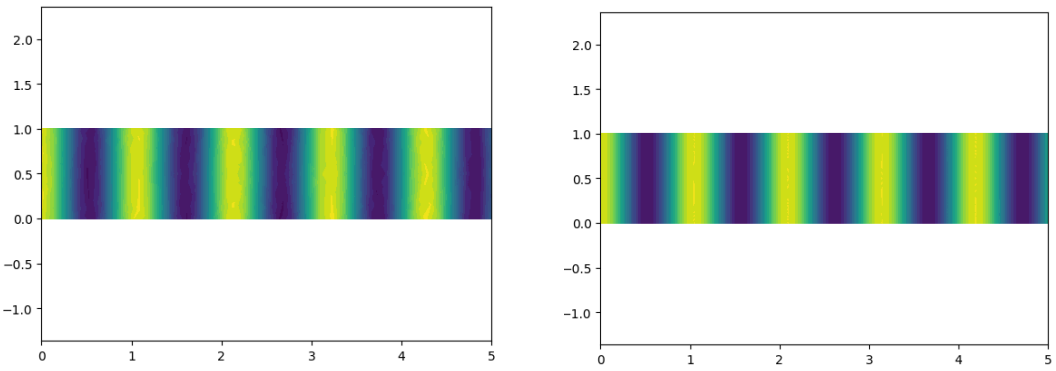
```
function femhelmholtz(p, t, ein, eout)
    areas = triarea(p,t)
    n = size(p)[1]
    K = spzeros(n,n); M = spzeros(n,n); Bin = spzeros(n,n); Bout = spzeros(n,n); bin = zeros(n);
    for kn = 1:size(t)[1]
        kinsert = zeros(3,3)
        minsert = zeros(3,3)
        cmx = coeffmx(Matrix{Float64}(I, 3, 3), p, t, kn)
        for i = 1:3, j = 1:3
            kinsert[i,j] = kentry(cmx,i,j,areas[kn])
            minsert[i,j] = mentry(cmx, i, j, areas[kn], p, t, kn)
        end
        K[t[kn,:],t[kn,:]] = K[t[kn,:],t[kn,:]]+kinsert
        M[t[kn,:],t[kn,:]] = M[t[kn,:],t[kn,:]]+minsert
    end
    for m = 1:size(ein)[1]
        einsert = boundsint(p,t,ein[m,:])
        Bin[ein[m,:],ein[m,:]] = Bin[ein[m,:],ein[m,:]] + einsert
    end
    for m = 1:size(eout)[1]
        einsert = boundsint(p,t,eout[m,:])
        Bout[eout[m,:],eout[m,:]] = Bout[eout[m,:],eout[m,:]] + einsert
    end
    for m = 1:size(ein)[1]
        binsert = loadvec(p,t,ein[m,:])
        bin[ein[m,:]] = bin[ein[m,:]] + binsert'
    end
    return K , M , Bin , Bout , bin
end
```

this yields the desired matrices.

(d) Through simply following the decomposition of A established in problem 1, we can use the function

```
function helmholtz(pv, hmax, nref, k)
    p, t, e = pmesh(pv, hmax, nref)
    ein, eout, ewall = waveguide_edges(p,t)
    K, M, Bin, Bout, bin = femhelmholtz(p, t, ein, eout)
    A = K-k^2*M+k*im*(Bout + Bin)
    B = bin*2*k*im
    u = A\B
    ur = real.(u)
    "tplot(p,t,ur)"
    return u
end
```

which yields the appropriate solutions, pictured to the left is nref = 1, the right is nref = 4



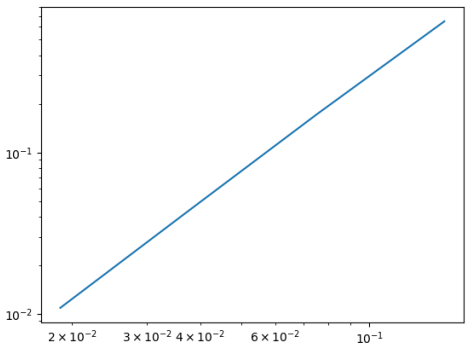
To find the errors and convergence rate we compare each iteration with running the highest refined mesh through the uexact function, yielding

```
function helmerrors(pv, hmax, nrefmax, k)
   pmax, tmax, emax = pmesh(pv, hmax, nrefmax)
    e=Base.MathConstants.e
    f(x,y) = e.^(-k*x*im)
   umax = f(pmax[:,1],1)
   errors = zeros(nrefmax)
   for ncur = 1:(nrefmax)
       "pprox, tprox, eprox = pmesh(pv, hmax, ncur)"
       uprox = helmholtz(pv, hmax, ncur, k)
       errorarray = maximum(abs.(umax[1:size(uprox)[1]] - uprox))
       #errors = [errors ; errorarray]
       errors[ncur]=errorarray
   end
   return errors
end
```

This gives us errors

$$[0.6488018138668635 \quad 0.17140607713178171 \quad 0.04336208511502913 \quad 0.010882557042220237]$$

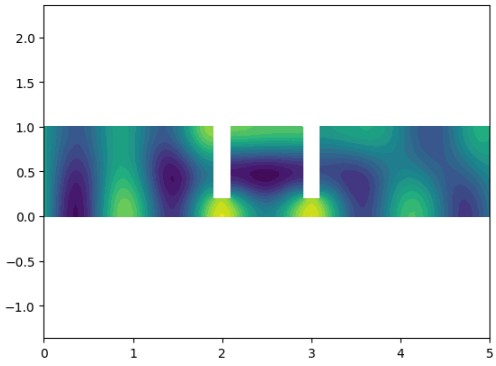
which can be seen in the plot



and yields a convergence rate of 1.9944165497829438.

Problem 3.

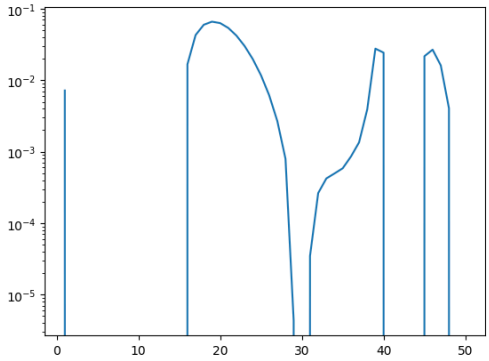
(a) Quite straightforwardly, this is done with the consideration of the coordinates of the new mesh. We can see an example solution in



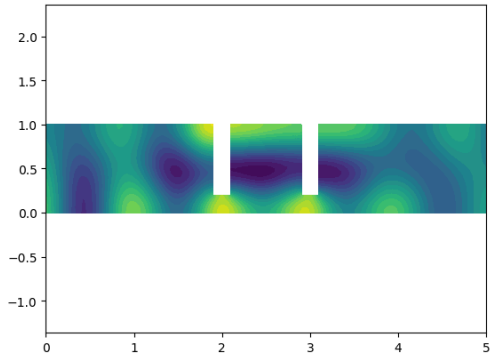
(b) We introduce code to iterate past each of the following steps, in the form of

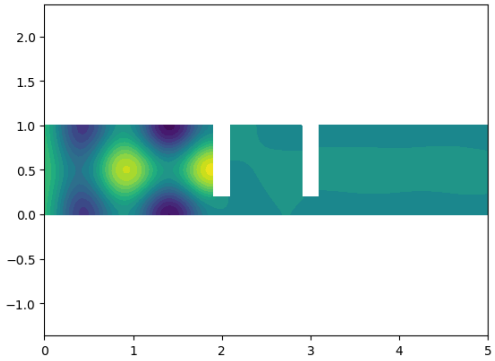
```
pv = [0.0 0.0; 5 0; 5 1; 3.1 1; 3.1 0.2; 2.9 0.2; 2.9 1; 2.1 1; 2.1 0.2; 1.9 0.2; 1.9 1; 0 1; 0 0]
Hmx = []
for n = 1:50
    k = 6+ 0.01*n
    u, ur, Bout = helmholtz(pv, 0.2, 2, k)
    H = conj(u')*Bout*u
    Hmx = [Hmx; H]
end
Hmx
```

running this data through a semilog plot yields



(c) The two solutions that represent the maximum and minimum of $H(u)$ correspond to the maximum and minimum of the magnitude of H, which is given by the k values at 13 and 36 for the least and highest, respectively.





Problem 4.
(a) First, we add the midpoints of each of the triangles, as specified here. It is rather straightforward to add the midpoints of each of the edges, as there is a function used in **pmesh** for this, but it would be much harder to find the appropriate indices to add to **t2**. Thus, I indexed a dictionary *mid_point* to store the midpoints of each edge, according to the appropriate key. This makes it rather straightforward to avoid duplicates, as all edges are unique, and can be easily appended to **t2** through iteration.

```
elist, bedges, emap = all_edges(t)
mid_point=Dict()
for k = 1:size(elist)[1]
    x=(p[elist[k,1],1]+p[elist[k,2],1])/2.0
    y=(p[elist[k,1],2]+p[elist[k,2],2])/2.0
    p=[p; x y]
    mid_point[elist[k,2],elist[k,1]]=size(p)[1]
    mid_point[elist[k,1],elist[k,2]]=size(p)[1]
end
t2=zeros(Int64,0,6)
for i = 1:size(t)[1]
   t12=mid_point[t[i,1],t[i,2]]
   t23=mid_point[t[i,2],t[i,3]]
   t31=mid_point[t[i,3],t[i,1]]
   t2=[t2; t[i,:]' t12 t23 t31]
end
```

To find the boundary nodes, we get the boundary nodes from the original mesh using the **boundary_nodes** function, and iterate through the respective keys in the *mid_point* structure to yield the additional midpoint boundary nodes.

```
e2 = boundary_nodes(t)
for i=1:size(bedges)[1]
   e=elist[bedges[i],:]
   md=mid_point[e[1],e[2]]
   e2=vcat(e2,md)
end
```

(b) In order to solve the Poisson equation for quadratic elements, we follow a similar outline as the linear Poisson equation solver, but with slight changes to use quadratics. For a given triangle T_k we have a 6x6 Vandermode matrix, as each basis function is given by

$$\varphi_i^k(x) = c_i^k + c_{i,x}^k x + c_{i,y}^k y + c_{i,xy}^k xy + c_{i,x^2}^k x^2 + c_{i,y^2}^k y^2$$

where every c denotes some constant. Thus, for any given triangular element, the coefficients are found by

$$\begin{pmatrix} 1 & x_1^k & y_1^k & x_1^k y_1^k & x_1^{k2} & y_1^{k2} \\ \cdot & & & & & \\ \cdot & & & & & \\ \cdot & & & & & \end{pmatrix} \begin{pmatrix} c_1^k & \cdot & \cdot & \cdot \\ c_{1,x}^k & & & \\ c_{1,y}^k & & & \\ c_{1,xy}^k & & & \\ c_{1,x^2}^k & & & \\ c_{1,y^2}^k & & & \end{pmatrix} = \begin{pmatrix} 1 & 0 & & \\ 0 & \cdot & & \\ & & \cdot & \\ & & & \cdot & 0 \\ & & & 0 & 1 \end{pmatrix}$$

and the corresponding coefficient matrix can be seen to model the 6 basis functions that satisfy δ_{ij} for the 6 points for each triangle. To find this result is quite straightforward - we introduce the function

```
function coeff2(RHS, p2, t2, k)
    nm = p2[t2[k,1:6],:]
    LHS = zeros(6,6)
    for n = 1:6
       LHS[n,:] = [1 nm[n,1] nm[n,2] nm[n,1]*nm[n,2] nm[n,1]^2 nm[n,2]^2]
    end
    return (LHS\RHS), LHS
end
```

Next, we need to establish a function that can evaluate the appropriate quadrature for this problem. The problem statement gives that

$$\begin{aligned}\int_{T_k} f(\mathbf{x})d\mathbf{x} &\approx \frac{A_k}{3}\sum_{i=1}^{3} f\left(\sum_{j=1}^{3}(\mathbf{x}_j/6+\delta_{ij}\mathbf{x}_j/2)\right) \\ &= \frac{A_k}{3}(f(\frac{4x_1+x_2+x_3}{6})+f(\frac{x_1+4x_2+x_3}{6})+f(\frac{x_1+x_2+4x_3}{6}))\end{aligned}$$

To implement this, we find f to be

$$\begin{aligned}A_{ij}^k &= \int_{T^k}\frac{\partial\varphi_i^k}{\partial x}\frac{\partial\varphi_j^k}{\partial x}+\frac{\partial\varphi_i^k}{\partial y}\frac{\partial\varphi_j^k}{\partial y}dx \\ &= \int_{T_k}\left((c_{i,x}^k+c_{i,xy}^k y+2xc_{i,x^2}^k)(c_{j,x}^k+c_{j,xy}^k y+2xc_{j,x^2}^k)+(c_{i,y}^k+c_{i,xy}^k x+2yc_{i,y^2}^k)(c_{j,y}^k+c_{j,xy}^k x+2yc_{j,y^2}^k)\right)d\mathbf{x} \\ &\approx \frac{A_k}{3}(f(\frac{4x_1+x_2+x_3}{6})+f(\frac{x_1+4x_2+x_3}{6})+f(\frac{x_1+x_2+4x_3}{6}))\end{aligned}$$

$$\begin{aligned}f = {} & x^2(c_{i,x^2}^k c_{j,x^2}^k+c_{i,xy}^k c_{j,xy}^k)+y^2(c_{i,y^2}^k c_{j,y^2}^k+c_{i,xy}^k c_{j,xy}^k) \\ & +xy(2c_{j,x^2}^k c_{i,xy}^k+2c_{i,x^2}^k c_{j,xy}^k+2c_{j,y^2}^k c_{i,xy}^k+2c_{i,y^2}^k c_{j,xy}^k)+x(2c_{i,x^2}^k c_{j,x}^k+2c_{j,x^2}^k c_{i,x}^k+c_{i,y}^k c_{j,xy}^k+c_{j,y}^k c_{i,xy}^k) \\ & +y(2c_{i,y^2}^k c_{j,y}^k+2c_{j,y^2}^k c_{i,y}^k+c_{i,x}^k c_{j,xy}^k+c_{j,x}^k c_{i,xy}^k)+c_{i,x}^k c_{j,x}^k+c_{i,y}^k c_{j,y}^k\end{aligned}$$

Please forgive any typos in f, intended to be the expansion of the function above. Thus, the function is as follows

```
function gquad(k, i, j, cmx, p2, t2, ak)
    f(x,y) = 4x^2*(cmx[5,i]*cmx[5,j]) + 2x*y*(cmx[5,i]*cmx[4,j] +
        cmx[5,j]*cmx[4,i]) + y^2*(cmx[4,i]*cmx[4,j]) + 2x*(cmx[5,i]*cmx[2,j] +
        cmx[5,j]*cmx[2,i]) + y*(cmx[2,i]*cmx[4,j] + cmx[2,j]*cmx[4,i]) +
        cmx[2,i]*cmx[2,j]
    g(x,y) = 4y^2*(cmx[6,i]*cmx[6,j]) + 2x*y*(cmx[6,i]*cmx[4,j] +
        cmx[6,j]*cmx[4,i]) + x^2*(cmx[4,i]*cmx[4,j]) + 2y*(cmx[6,i]*cmx[3,j] +
        cmx[6,j]*cmx[3,i]) + x*(cmx[3,i]*cmx[4,j] + cmx[3,j]*cmx[4,i]) +
        cmx[3,i]*cmx[3,j]

    result = 0
    for n = 1:3
        base = (p2[t2[k,1],:] + p2[t2[k,2],:] + p2[t2[k,3],:])/6
        num = base + p2[t2[k,n],:]/2
        result = result + f(num[1],num[2]) + g(num[1],num[2])
    end
    result = (result*ak)/3
    return result
end
```

Similarly, we need to do the same for

$$b = \int_{T_k}\varphi_i d\mathbf{x}$$

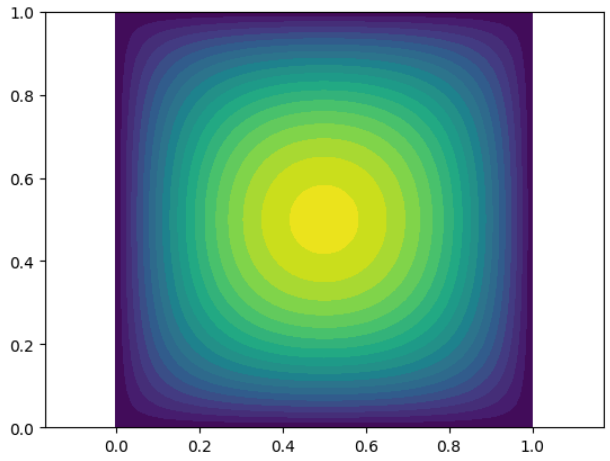
where we define a function **gquad2** which performs a similar function, shown by

```
function gquad2(k, i, cmx, p2, t2, ak)
    h(x,y) = cmx[1,i] + x*cmx[2,i] + y*cmx[3,i] + x*y*cmx[4,i] + x^2*cmx[5,i] + y^2*cmx[6,i]
    result = 0
    for n = 1:3
        base = (p2[t2[k,1],:] + p2[t2[k,2],:] + p2[t2[k,3],:])/6
        num = base + p2[t2[k,n],:]/2
        result = result + h(num[1],num[2])
    end
    result = (result*ak)/3
    return result
end
```

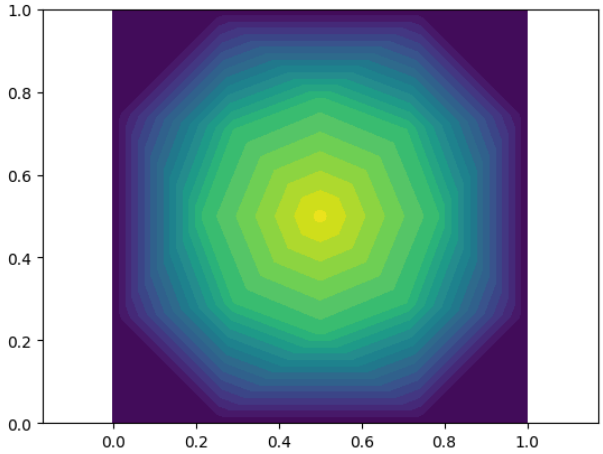
To assemble the **fempoi2**, we iterate past all the 36 different basis function i, j combinations, for each triangle element T_k, and then stamp the resulting matrix into the master matrix A, and similarly for b. The Dirichlet conditions are imposed in the same way as before, setting the appropriate elements in both stiffness matrices.

```
function fempoi2(p2, t2, e2)
    areas = triarea(p2,t2)
    n = size(p2)[1]
    A = spzeros(n,n); b = zeros(n);
    for k = 1:size(t2)[1]
        insert = zeros(6,6)
        insert2 = zeros(6)
        cmx, lhss = coeff2(Matrix{Float64}(I, 6, 6), p2, t2, k)
        for i = 1:6
            for j = 1:6
                insert[i,j] = gquad(k,i,j,cmx,p2,t2,areas[k])
            end
            insert2[i] = gquad2(k,i,cmx,p2,t2,areas[k])
        end
        A[t2[k,:],t2[k,:]] = A[t2[k,:],t2[k,:]]+insert
        b[t2[k,:]] = b[t2[k,:]]+ insert2
    end
    for k = 1:size(e2)[1]
        i=e2[k]
        A[i,:].=0.0
        A[i,i]=1.0
        b[i]=0.0
    end
    return A\b
end
```

Running this for the most refined case, $n = 4$ we get the following solution



and a less refined mesh can be seen by



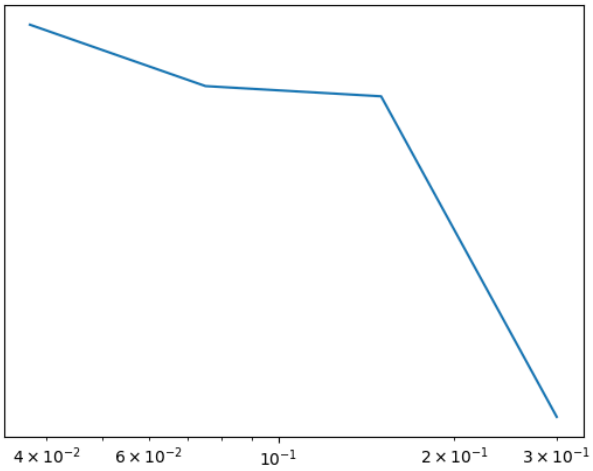
(d) Modifying the **poiconcv** code from the last problem set yields a simple way to find convergence plots.

```
function convtest(pv, hmax, nrefmax)
    p, t, e = pmesh(pv,hmax,nrefmax)
    p2, t2, e2 = p2mesh(p,t)
    umax = fempoi2(p2,t2,e2)
    errors = zeros(nrefmax)
    for ncur = 0:(nrefmax-1)
       pprox, tprox, eprox = pmesh(pv, hmax, ncur)
       pprox2, tprox2, eprox2 = p2mesh(pprox, tprox)
       uprox = fempoi2(pprox2, tprox2, eprox2)
       errorarray = maximum(abs.(umax[1:size(uprox)[1]] - uprox))
       #errors = [errors ; errorarray]
       errors[ncur+1]=errorarray
    end
    return errors
end
```

which can be implemented by running

```
errors = convtest(pv,hmax,nrefmax)
clf()
loglog(hmax ./ [1,2,4,8], errors)
rates = @. log2(errors[end-1,:]) - log2(errors[end,:])
```

This yields the figure



and a rate of 0.016159505751054848.

UC Berkeley Math 228B, Spring 2019: Problem Set 3

Due March 7

1. Write a Julia function with the syntax

   ```
   p, t, e = pmesh(pv, hmax, nref)
   ```

   which generates an unstructured triangular mesh of the polygon with vertices `pv`, with edge lengths approximately equal to $h_{\max}/2^{n_{\text{ref}}}$, using a simplified Delaunay refinement algorithm. The outputs are the node points `p` (N-by-2), the triangle indices `t` (T-by-3), and the indices of the boundary points `e`.

   (a) The 2-column matrix `pv` contains the vertices x_i, y_i of the original polygon, with the last point equal to the first (a closed polygon).

   (b) First, create node points along each polygon segment, such that all new segments have lengths $\le h_{\max}$ (but as close to $h_{\max}$ as possible). Make sure not to duplicate any nodes.

   (c) Triangulate the domain using the `delaunay` function in the mesh utilities.

   (d) Remove the triangles outside the domain (see the `inpolygon` command in the mesh utilities).

   (e) Find the triangle with largest area A. If $A > h_{\max}^2/2$, add the circumcenter of the triangle to the list of node points.

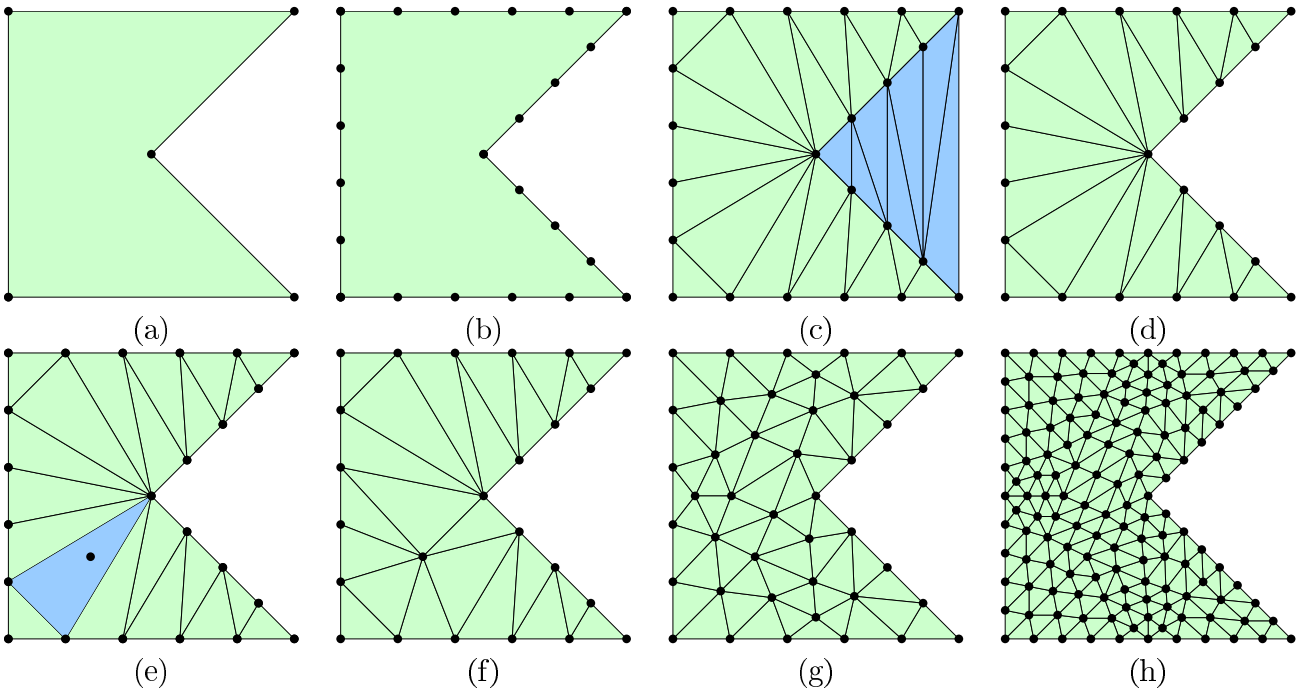
   (f) Retriangulate and remove outside triangles (steps (c)-(d)).

   (g) Repeat steps (e)-(f) until no triangle area $A > h_{\max}^2/2$.

   (h) Refine the mesh uniformly n_{ref} times. In each refinement, add the center of each mesh edge to the list of node points, and retriangulate. Again, make sure not to duplicate any nodes, using e.g. the command `unique(p, dims=1)`.

   Finally, find the nodes `e` on the boundary using the `boundary_nodes` function. The following commands create the example in the figures. Also make sure that the function works with other inputs, that is, other polygons, $h_{\max}$, and n_{ref}.

   ```
   pv = [0 0; 1 0; .5 .5; 1 1; 0 1; 0 0]
   p, t, e = pmesh(pv, 0.2, 1)
   tplot(p, t)
   ```



2. Consider the Eikonal equation for first arrivals/optimal path planning problems. The equation can be written

$$F(x,y)|\nabla\phi(x,y)| = 1,$$

where $F(x,y)$ is the speed function. We will use the solution $\phi(x,y)$ to determine the optimal path from the departure point (x_D, y_D) to the arrival point (x_A, y_A), where $\phi(x_A, y_A) = t$. The level set with value t of the function $\phi(x,y)$ gives the maximum distance away from our departure point that can be traveled in a time t. In addition, the optimal path between the departure point and the arrival point is determined by traveling in the normal direction of the level sets.

We will solve the Eikonal equation using a level set method and a time-stepping approach. The equation

$$\phi_t + F|\nabla\phi| = 1, \qquad \phi(x_D, y_D) = 0$$

is integrated in time until a steady-state is reached. This is not an efficient method for solving the Eikonal equation, but it will be sufficient for this problem set (and it illustrates how to solve more general time-dependent problems). If you are interested in more sophisticated solvers, feel free to implement the more efficient fast marching method instead.

(a) Write a computer code to solve the Eikonal equation on the unit square $x, y \in [0,1]$. Use the first-order upwinded scheme in space, and appropriate treatment of the boundaries.

(b) Write a computer code to find the optimal path between the departure/arrival point, by solving the ODE $d\boldsymbol{r}/dt = \boldsymbol{n}$, where $\boldsymbol{r}$ is the current position on the path and $\boldsymbol{n}$ is the normal vector.

(c) Run your codes with grid spacing $h = 1/100$ for the following cases and plot both the solutions (e.g. as contour curves of $\phi(x,y)$) and the optimal paths:

Case 1: $(x_D, y_D) = (0.2, 0.2)$, $(x_A, y_A) = (0.8, 0.8)$, $F(x,y) = 1$ (for testing).
Case 2: $(x_D, y_D) = (0.2, 0.2)$, $(x_A, y_A) = (0.8, 0.45)$, and

$$F(x,y) = \begin{cases} 1.0 & \text{if } y \geq 0.5, \\ 0.5 & \text{if } y < 0.5. \end{cases}$$

Case 3: $(x_D, y_D) = (0.2, 0.2)$, $(x_A, y_A) = (0.8, 0.8)$, and

$$F(x,y) = 1 - 0.9 \cdot \cos(4\pi x) \cdot e^{-10\,((x-.5)^2 + (y-0.5)^2)}$$

Case 4: Make up your own speed function F, only returning the values 0.01 and 1 but with a non-trivial optimal path.

Code Submission: Submit a zip-file on bCourses which contains a Julia file named `pmesh.jl` where all requested functions are defined (that is, `pmesh`), and any other supporting functions. Alternatively, submit a Jupyter notebook which will define this function when executed.

UC Berkeley Math 228B, Spring 2019: Problem Set 4

Due March 21

1. Consider the boundary value problem

$$u''''(x) = f(x) \equiv 480x - 120, \qquad \text{for } x \in (0,1) \tag{1}$$
$$u(0) = u'(0) = u(1) = u'(1) = 0 \tag{2}$$

   (a) Derive the following Galerkin formulation for the problem (1)-(2) on some appropriate function space V_h: Find $u_h \in V_h$ such that

$$\int_0^1 u_h''(x)v''(x)\,dx = \int_0^1 f(x)v(x)\,dx, \qquad \forall v \in V_h. \tag{3}$$

   (b) Define the triangulation $T_h = \{K_1, K_2\}$, where $K_1 = [0, \frac{1}{2}]$ and $K_2 = [\frac{1}{2}, 1]$, and the function space

$$V_h = \{v \in C^1([0,1]) : v\big|_K \in \mathbb{P}_3(K)\ \forall K \in T_h,\ v(0) = v'(0) = v(1) = v'(1) = 0\}. \tag{4}$$

   Find a basis $\{\varphi_i\}$ for V_h. *Hint*: Consider Hermite polynomials on each element.

   (c) Solve the Galerkin problem (3) using your basis functions. Plot the numerical solution $u_h(x)$ and the true solution $u(x)$.

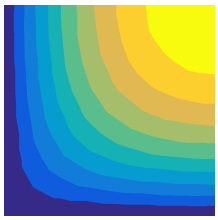
2. Implement a Julia function with the syntax

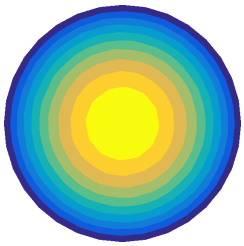
```
u = fempoi(p,t,e)
```

that solves Poissons's equation $-\nabla^2 u(x,y) = 1$ on the domain described by the unstructured triangular mesh `p,t`. The boundary conditions are homogeneous Neumann ($n \cdot \nabla u = 0$) except for the nodes in the array `e` which are homogeneous Dirichlet ($u = 0$).

Here are a few examples for testing the function:

```
# Square, Dirichlet left/bottom
pv = Float64[0 0; 1 0; 1 1; 0 1; 0 0]
p, t, e = pmesh(pv, 0.15, 0)
e = e[@. (p[e,1] < 1e-6) | (p[e,2] < 1e-6)]
u = fempoi(p, t, e)
tplot(p, t, u)

# Circle, all Dirichlet
n = 32; phi = 2pi*(0:n)/n
pv = [cos.(phi)  sin.(phi)]
p, t, e = pmesh(pv, 2pi/n, 0)
u = fempoi(p, t, e)
tplot(p, t, u)

# Generic polygon geometry, mixed Dirichlet/Neumann
x = 0:.1:1
y = 0.1*(-1).^(0:10)
pv = [x y; .5 .6; 0 .1]
p, t, e = pmesh(pv, 0.04, 0)
e = e[@. p[e,2] > (.6 - abs(p[e,1] - 0.5) - 1e-6)]
u = fempoi(p, t, e)
tplot(p, t, u)
```





Turn page ⟶

3. Implement a Julia function with the syntax

   ```
   errors = poiconv(pv, hmax, nrefmax)
   ```

   that solves the all-Dirichlet Poisson problem for the polygon `pv`, using the mesh parameters `hmax` and `nref = 0,1,...,nrefmax`. Consider the solution on the finest mesh the exact solution, and compute the max-norm of the errors at the nodes for all the other solutions (note that this is easy given how the meshes were refined – the common nodes appear first in each mesh). The output `errors` is a vector of length `nrefmax` containing all the errors.

   Test the function using the commands below, which makes a convergence plot and estimates the rates:

```
hmax = 0.15
pv_square = Float64[0 0; 1 0; 1 1; 0 1; 0 0]
pv_polygon = Float64[0 0; 1 0; .5 .5; 1 1; 0 1; 0 0]

errors_square = poiconv(pv_square, hmax, 3)
errors_polygon = poiconv(pv_polygon, hmax, 3)
errors = [errors_square errors_polygon]

clf()
loglog(hmax ./ [1,2,4], errors)
rates = @. log2(errors[end-1,:]) - log2(errors[end,:])
```

Code Submission: Submit a zip-file on bCourses which contains a Julia file named `fempoi.jl` where all requested functions are defined (that is, `fempoi` and `poiconv`), and any other supporting functions. Alternatively, submit a Jupyter notebook which will define these functions when executed.

UC Berkeley Math 228B, Spring 2019: Problem Set 5

Due April 11

In this problem set, you will study two extensions of the simple piece-wise linear Poisson solver `fempoi` from the previous problem set. First, you will extend the equations to solve the Helmholtz equation, for simulation of wave propagation in waveguides. Next, you will extend the Poisson solver to use quadratic elements instead of linear.

Time-harmonic Waveguide Simulations

1. Consider the following 2-D Helmholtz problem, for a given wave number k with normalized propagation velocity, and so-called Sommerfeld radiation conditions at the in/out boundaries:

$$-\nabla^2 u - k^2 u = 0, \qquad \text{in } \Omega, \tag{1}$$
$$\boldsymbol{n} \cdot \nabla u = 0, \qquad \text{on } \Gamma_{\text{wall}} \tag{2}$$
$$\boldsymbol{n} \cdot \nabla u + iku = 0, \qquad \text{on } \Gamma_{\text{out}} \tag{3}$$
$$\boldsymbol{n} \cdot \nabla u + iku = 2ik, \qquad \text{on } \Gamma_{\text{in}} \tag{4}$$

   Here, the domain boundary $\Gamma = \partial\Omega$ is decomposed into the three parts $\Gamma = \Gamma_{\text{wall}} \cup \Gamma_{\text{out}} \cup \Gamma_{\text{in}}$. For a computed solution u, we will also calculate its intensity at the output boundaries:

$$H(u) = \int_{\Gamma_{\text{out}}} |u|^2 \, ds, \tag{5}$$

   where $|\cdot|$ is the complex absolute value.

   (a) Derive a Galerkin finite element formulation for (1)-(4), for an appropriate space V_h of continuous piece-wise linear functions.

   (b) Show that the discretized system can be written in the form $A\boldsymbol{u} = \boldsymbol{b}$, where

$$A = K - k^2 M + ik(B_{\text{in}} + B_{\text{out}}) \quad \text{and} \quad \boldsymbol{b} = 2ik\boldsymbol{b}_{\text{in}} \tag{6}$$

   for real matrices $K, M, B_{\text{in}}, B_{\text{out}}$ and a vector $\boldsymbol{b}_{\text{in}}$, which do not depend on the wave number k. Give explicit expressions for the matrix/vector entries, involving the basis functions $\varphi_i(\boldsymbol{x})$ for the space V_h.

   (c) Show that the transmitted intensity (5) for a finite element solution $\boldsymbol{u}$ can be calculated as $H(\boldsymbol{u}) = \boldsymbol{u}^H B_{\text{out}} \boldsymbol{u}$.

Turn page $\longrightarrow$

2. For the implementation, consider a model test problem with wave number $k = 6$ and a straight channel domain of dimensions 5×1:

$$\Omega = \{0 \le x \le 5, 0 \le y \le 1\} \tag{7}$$
$$\Gamma_{\text{in}} = \{x = 0, 0 \le y \le 1\} \tag{8}$$
$$\Gamma_{\text{out}} = \{x = 5, 0 \le y \le 1\} \tag{9}$$
$$\Gamma_{\text{wall}} = \{0 \le x \le 5, y = 0 \text{ or } y = 1\} \tag{10}$$

   (a) Show that an exact solution to the Helmholtz problem (1)-(4) for the domain (7)-(10) is given by $u_{\text{exact}}(x, y) = e^{-ikx}$.

   (b) Write a function which for a given triangular mesh `p`, `t` identifies the boundary edges corresponding to the wall, the in, and the out boundaries, respectively:

```
ein, eout, ewall = waveguide_edges(p, t)
```

   Use the function `all_edges()` in the Mesh utilities notebook on the course web page to find all mesh edges, then assume that the in-boundary consists of all vertical edges with $x = 0$, and that the out-boundary consist of all vertical edges with $x = 5$.

   (c) Write a function that computes the matrices $K, M, B_{\text{in}}, B_{\text{out}}$ and the vector $\boldsymbol{b}_{\text{in}}$ for a given mesh:

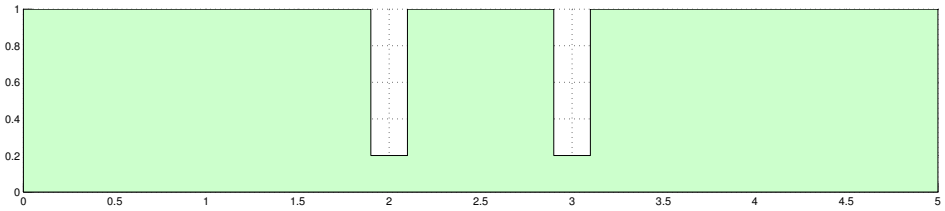
```
K, M, Bin, Bout, bin = femhelmholtz(p, t, ein, eout)
```

   (d) Solve the discretized problem on the meshes generated by

```
pv = [0 0; 5 0; 5 1; 0 1; 0 0]
p, t, e = pmesh(pv, 0.3, nref)
```

   where `nref` ranges from 1 to 4. Compute max-norm errors using the exact solution u_{exact}, plot errors vs. mesh size in a log-log plot, and estimate the order of convergence.

3. Finally, you will use your Helmholtz solver to compute a frequency response for a waveguide with two slits, see figure below. The waveguide is again of dimensions 5-by-1, and the slits are 0.2 units wide and 0.8 units deep, centered at $x = 2$ and $x = 3$.



   (a) Create a mesh for the domain using `pmesh`, with `hmax` $= 0.2$ and `nref` $= 2$.

   (b) To look for resonance phenomena around $k \approx 2\pi$, solve for a range of wave numbers k between $k = 6$ and $k = 6.5$, in steps of $\Delta k = 0.01$. For each k, solve the problem and calculate $H(\boldsymbol{u})$. Plot H vs. k in a semi-log plot.

   (c) Plot two of your solutions using `tplot` of the real part, for the wave numbers k corresponding to the smallest and the largest value of $H(\boldsymbol{u})$.

Turn page $\longrightarrow$