```
> restart;
  with(LinearAlgebra):
  with(ListTools):
  with(plots):
```

The finite element method: application to 2D PDEs

The purpose of this worksheet is to describe how to use finite element methods to solve partial differential equations of the form

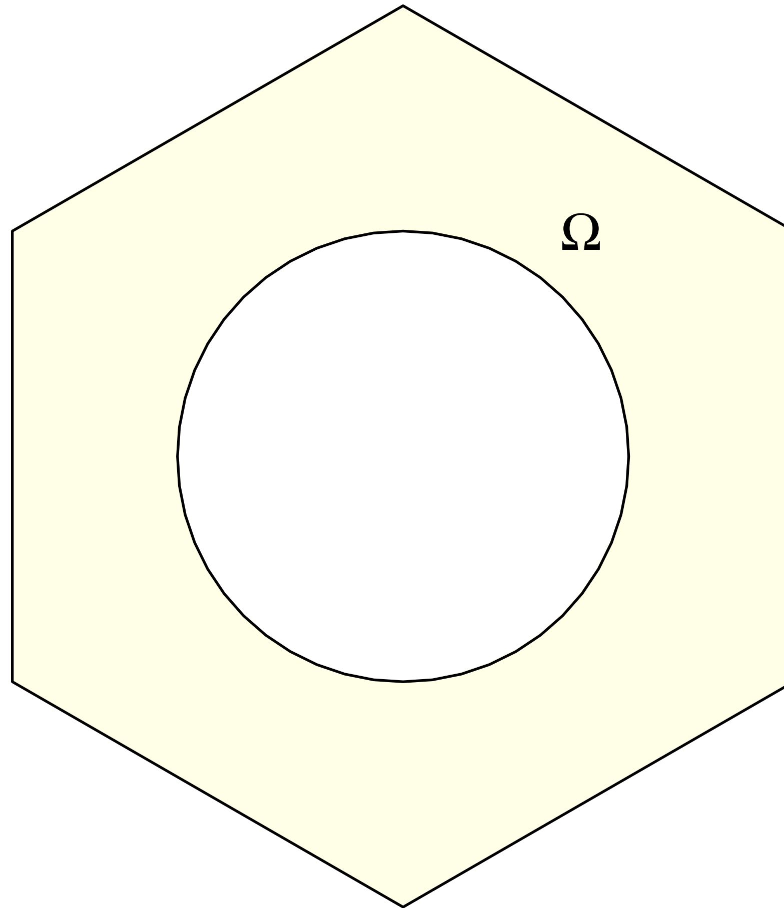
$$\rho \frac{\partial^2 u}{\partial t^2} + \lambda \frac{\partial u}{\partial t} = \nabla^2 u - R, \quad u = u(t, x, y),$$

for $(x, y) \in \Omega$. Here, $\{\rho, \lambda, R\}$ are all known functions the spatial coordinates (x, y), but not time t. The main motivation for employing the finite element method is that it can be applied to problems on arbitrary domains Ω. For the purposes of this worksheet we will take Ω to be the region in-between two polar curves $r_1(\theta)$ and $r_2(\theta)$. Here is an example of the region when the inner boundary is a circle and the outer boundary is a regular polygon with m vertices:

```
> m := 6:
  r := [theta -> 1/2 , theta -> cos(Pi/m)/cos(theta - 2*Pi/m*floor((m*theta + Pi)/(2*Pi)))];
  n := 4*(2*m):
  theta := i -> 2*Pi*(i-1)/n:
  data := Reverse([seq([seq([r[j](theta(i))*cos(theta(i)),r[j](theta(i))*sin(theta(i))],i=1..n)
  ],j=1..2)]):
  p1 := polygonplot(data,filled=true,color=["LightYellow",white],axes=none,scaling=constrained)
  :
  p2 := textplot([.4,.5,Omega],font=[TIMES,ROMAN,16]):
  display([p1,p2],title=typeset("An example of the computational domain ",Omega),view=[-1.1.
  .1.1,-1.1..1.1]);
```

$$r := \left[\theta \to \frac{1}{2}, \theta \to \frac{\cos\left(\frac{\pi}{m}\right)}{\cos\left(\theta - \frac{2\,\pi\,\mathrm{floor}\left(\frac{1}{2}\,\frac{m\,\theta + \pi}{\pi}\right)}{m}\right)}\right]$$

An example of the computational domain Ω



We will assume Dirichlet boundary conditions on the inner and outer boundaries of the form

$$u\left(\partial\Omega_{\text{inner}}\right) = g_1(\theta), \quad u\left(\partial\Omega_{\text{outer}}\right) = g_2(\theta).$$

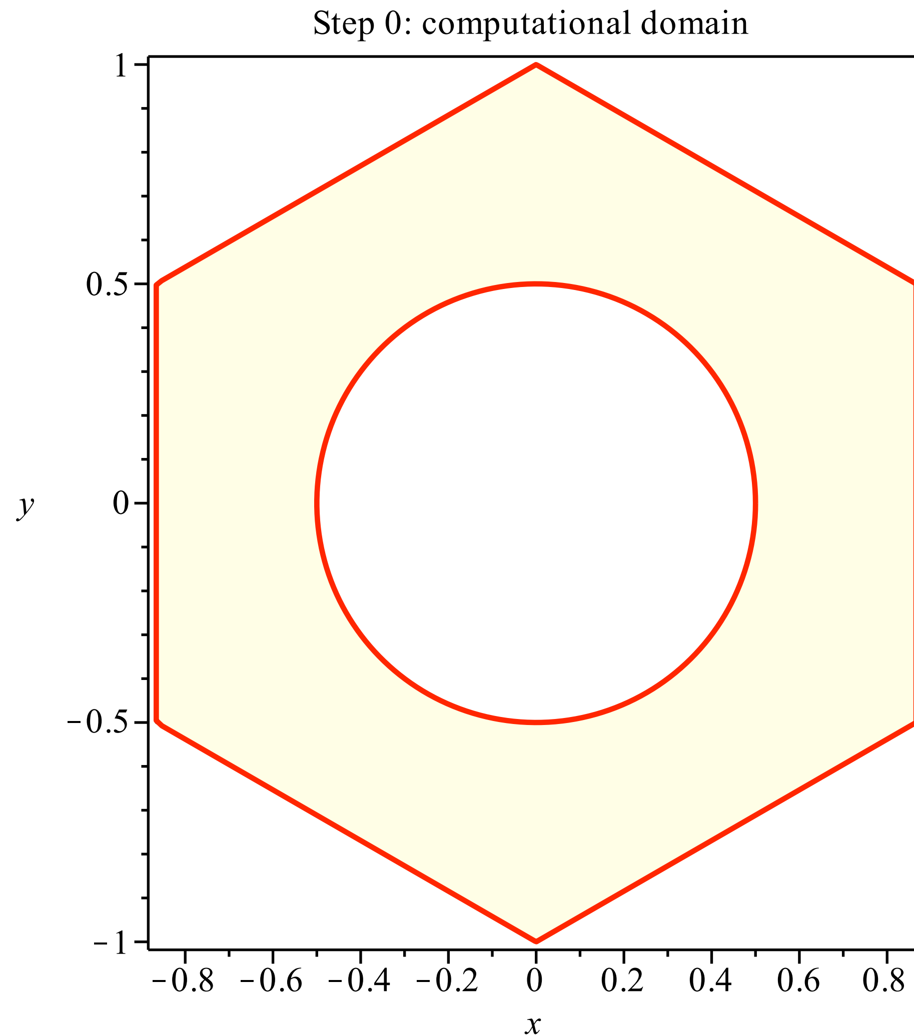
The main steps in constructing the finite element solution are:

1. Covering the computational domain Ω with a mesh of triangles.
2. Expressing the unknown function $u(t, x, y)$ as a linear superposition of piecewise continous basis functions $\phi_i(x, y)$ defined on these triangles: $u(t, x, y) = \Sigma_i\, a_i(t)\phi_i(x, y)$. Some of the expansion coefficients $a_i(t)$ will be fixed by boundary conditions, while the other ones are what we are trying to determine.
3. Deriving the "weak form" of the original PDE by multiplying it by a basis function and integrating over Ω, and then iterating over all the basis functions with undetermined amplitudes. The result will be a system of linear ODEs to solve for the unknowns coefficients $a_i(t)$ (if at least one of ρ or λ are non-vanishing), or a linear system of algebraic equations for $a_i(t) = a_i$ (if $\rho = \lambda = 0$).

Triangulating the domain Ω

The first step in constructing our finite element solution of the PDE is triangulating the domain Ω. That is, we place a set of nodes on Ω which comprise the vetricies of a collection of triangles that cover the entire region. Constructing such triangulations for arbitrary regions is an interesting problems in its own right, with numerous subtle issues, but the general problem is beyond the scope of this document. We will content ourselves with a simple algorithms that is only applicable when Ω is the region in-between $r_1(\theta)$ and $r_2(\theta)$. The method is illustrated by the following movie:

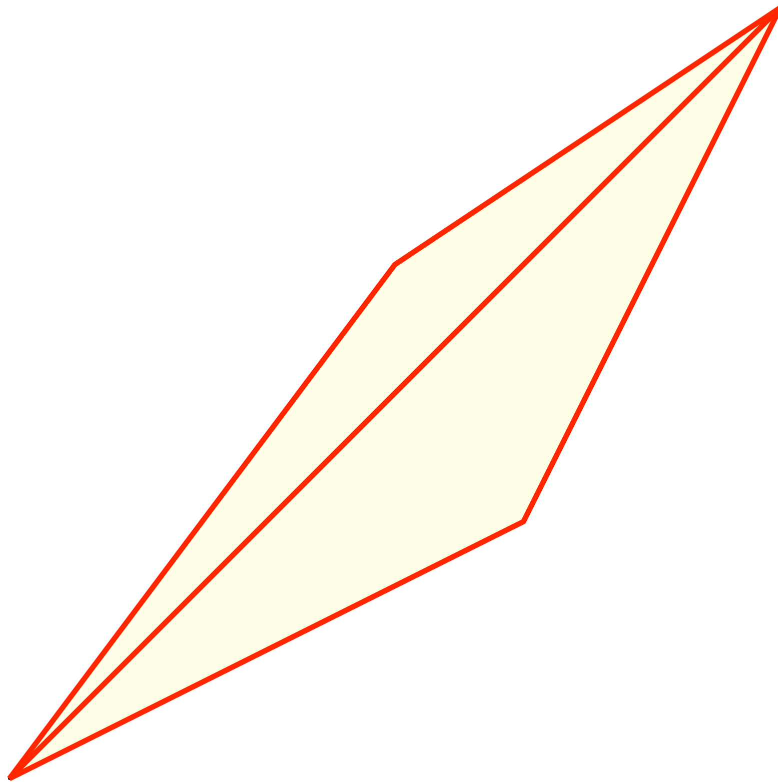
```
> shells := 6:
  Labels := ["computational domain","divide into concentric shells","partition into
  quadrilaterals","form triangles","fix skinny triangles"]:
  R := (theta,j) -> r[1](theta) + (j-1)/(shells-1)*(r[2](theta)-r[1](theta)):
  p[1] := plot([seq([r[j](theta)*cos(theta),r[j](theta)*sin(theta),theta=0..2*Pi],j=1..2)],
  color=red,axes=boxed,thickness=2,labels=[x,y]):
  p[2] := plot([seq([R(theta,j)*cos(theta),R(theta,j)*sin(theta),theta=0..2*Pi],j=1..
  shells-1)],color=red,axes=boxed):
  p[3] := plot([seq([seq([r[k](theta(i))*cos(theta(i)),r[k](theta(i))*sin(theta(i))],k=1..2)
  ],i=1..n)],color=red):
  p[4] := plot([seq(seq([seq([R(theta(i+k),k+l-1)*cos(theta(i+k)),R(theta(i+k),k+l-1)*sin
  (theta(i+k))],k=1..2)],i=1..n),l=1..shells-1)],color=red):
  display([seq(display([p1,seq(p[j],j=1..i)],title=typeset("Step ",i-1,": ",Labels[i])),i=1.
  .4)],insequence=true,scaling=constrained,axes=boxed);
```

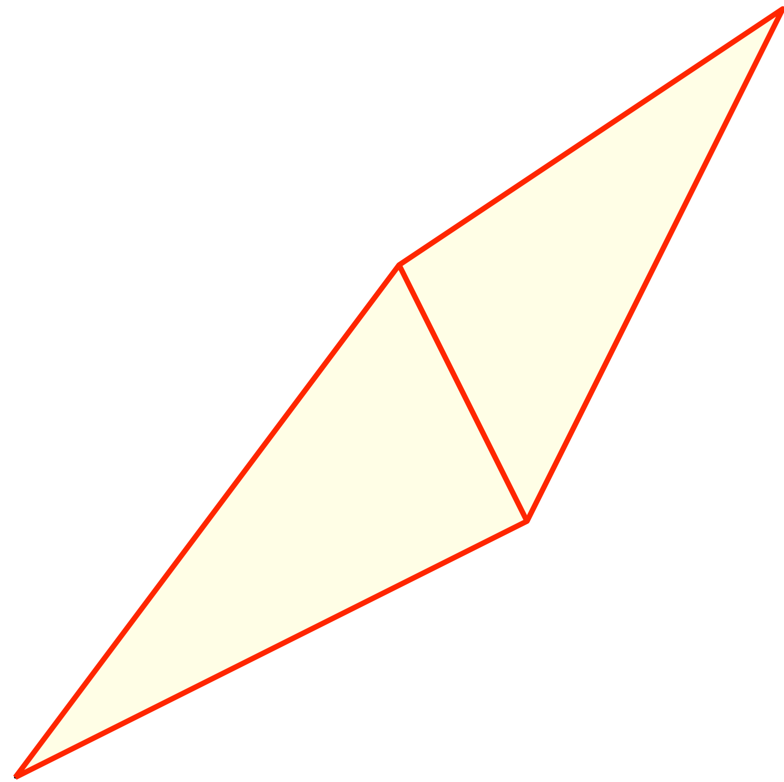


The individual steps in the procedure are illustrated by each frame of the movie. We begin with the domain Ω we are trying to triangulate (step 0). We draw a series of concentric shells between the inner and outer boundaries that are a simple linear interpolation between $r_1(\theta)$ and $r_2(\theta)$ given by the mapping **R(theta,j)** (step 1). We arrange things such that the total number of shells is given by **shells**, including the boundary curves. Then, we draw **n** radial lines between the two boundaries, which has the effect of subdividing Ω into a set of quadrilaterals (step 2). Finally, we subdivide each of these quadrilaterals into a pair or triangles (step 3). This is not actually the final step because the triangluar mesh of step 3 may contain some undesirable elements: long skinny triangles. This will be an issue when we start

integrating functions like the source $f(x, y)$ over triangles using quadrature approximation rules (see below). The problem can be ameliorated by performing a "flip" procedure as illustrated in the following plot:

```
> data := [[0,0],[1,1/2],[3/2,3/2],[3/4,1]]:
  q0 := plot([op(data),[0,0]],color=red,thickness=2):
  q1 := polygonplot(data,color="LightYellow",axes=none):
  q2 := plot([[0,0],[3/2,3/2]],color=red,thickness=2):
  q3 := plot([[1,1/2],[3/4,1]],color=red,thickness=2):
  q4 := display([q0,q1,q2],title="Before flip"):
  q5 := display([q0,q1,q3],title="After flip"):
  display(Array([q4,q5]),scaling=constrained);
```





The algorithm is to take two skinny triangles sharing a common edge, and convert them into two more well-proportioned triangles by replacing the shared edge with one connecting the previously unshared verticies. Here is a procedure that constructs the mesh given the boundary curves, **n**, and **shells**, using this flipping technique when necessary:

```
> GenerateMesh := proc(r,n,shells,g)
    local R, COUNT, i, j, N, nodes, triangles, p, d, x, gnat, element, Gamma, BoundaryData:
```

```
R := (theta,i) -> r[1](theta) + (i-1)/(shells-1)*(r[2](theta)-r[1](theta));
COUNT := 0:
for i from 1 to shells do:
   for j from 1 to n do:
       COUNT := COUNT + 1:
       nodes[COUNT] := evalf([R((j-1)/n*2*Pi,i)*cos((j-1)/n*2*Pi),R((j-1)/n*2*Pi,i)*sin
((j-1)/n*2*Pi)]):
   od:
od:
nodes := convert(nodes,list):
N[1] := nops(nodes);
COUNT := 0:
for i from 1 to shells-1 do:
    for j from 1 to n do:
        p[1] := (i-1)*n + j:
        if (j<>n) then p[2] := p[1] + 1 else p[2] := (i-1)*n + 1 fi:
        p[3] := p[1]+n;
        p[4] := p[2]+n;
        p := convert(p,list):
        x := map(u->Vector(nodes[u]),p):
        d[1] := (x[1]-x[3]).(x[1]-x[3]):
        d[2] := (x[2]-x[4]).(x[2]-x[4]):
        if (d[1]<d[2]) then:
          triangles[COUNT+1] := [p[1],p[3],p[2]]:
          triangles[COUNT+2] := [p[3],p[4],p[2]]:
        else:
          triangles[COUNT+1] := [p[1],p[4],p[2]]:
          triangles[COUNT+2] := [p[3],p[4],p[1]]:
        fi:
        COUNT := COUNT+2:
    od:
od:
triangles := convert(triangles,list):
N[2] := nops(triangles);
for i from 1 to N[1] do:
   COUNT := 0:
   for j from 1 to N[2] do:
        gnat := convert(triangles[j],set) intersect {i}:
        if (gnat<>{}) then:
          COUNT := COUNT + 1:
          element[i][COUNT] := j:
```

```
            fi:
        od:
        element[i] := convert(element[i],list):
    od:
    element := convert(element,list):
    Gamma[0] := [seq(i,i=n+1..(shells-1)*n)];
    Gamma[1] := [seq(i,i=1..n)];
    Gamma[2] := [seq(i,i=(shells-1)*n+1..shells*n)];
    BoundaryData[1] := evalf([seq(g[1]((j-1)/n*2*Pi),j=1..n)]);
    BoundaryData[2] := evalf([seq(g[2]((j-1)/n*2*Pi),j=1..n)]);
    [nodes,triangles,element,Gamma,BoundaryData]:

end proc:
```

The ouput of the code is a list of five objects used to describe the mesh:

1. The first quantity **nodes** is a list of the *xy*-coordinates of each node in the mesh. We will label each node by its position in this list; i.e., the ith element in **nodes** will be the coordinates of what we call the ith node. We usually denote the number of nodes by N_1.
2. The second object **triangles** is a list the triangles making up the mesh. Each element is a list of three integers representing the vertex nodes of the triangle as labelled in **nodes**. This list also serves to label each triangle; i.e., we call the triangle corresponding to the ith element of **triangles** the ith triangle. We usually denote the number of triangles by N_2.
3. The third object **element** output by **GenerateMesh** is a list of *N1* sub-lists. The ith sub-list gives all the triangles (as enumerated in **triangles**) containing the ith node (as enumerated in **nodes**).
4. The fourth object is a remember table **Class** consisting of three entries: **Gamma[0]** is a list of all the nodes that do not lie on the boundary of Ω, **Gamma[1]** is a list with all the nodes on the interior boundary, and **Gamma[2]** is a list with all the nodes on the outer boundary (again, all nodes are identified by their position in **nodes**).
5. The fifth object **BoundaryData** is a remember table containing two lists representing the boundary values of the field *u* on each of the nodes in **Gamma[1]** and **Gamma[2]**, respectively.

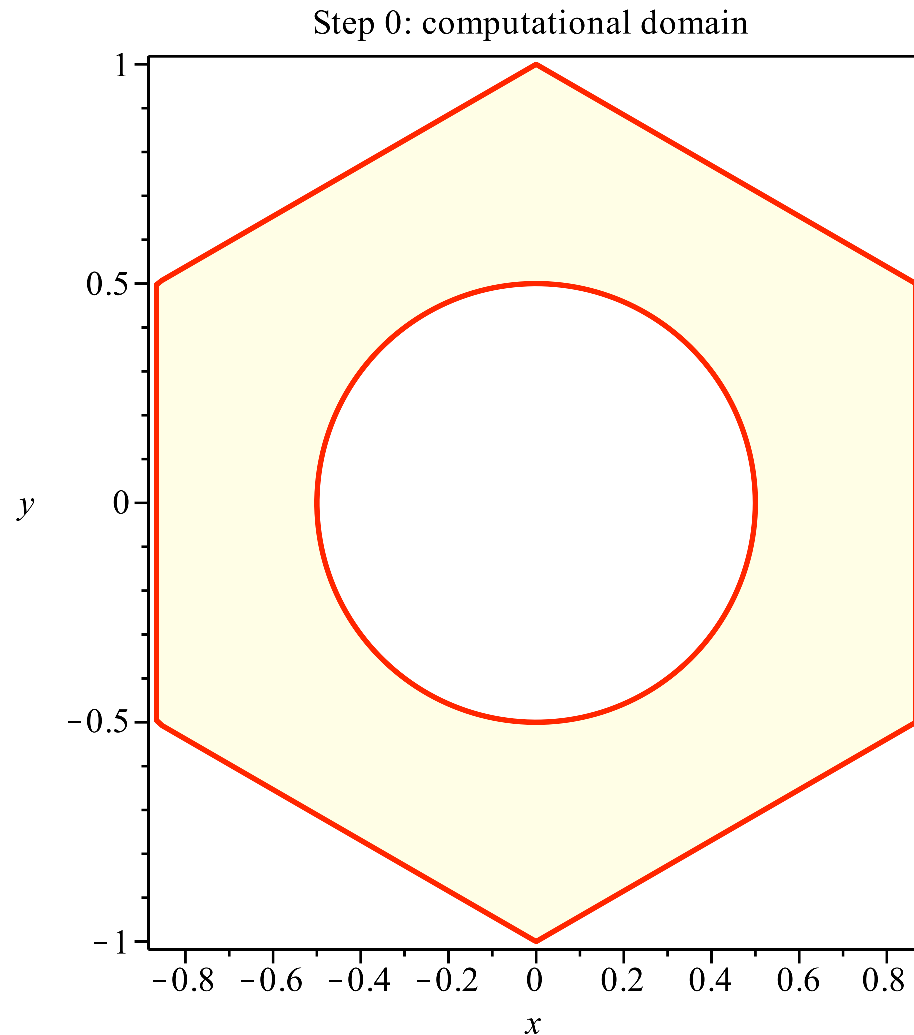
We also define a procedure **DrawMesh** that draws a picture of the mesh given **nodes** and **triangles**:

```
> DrawMesh2D := proc(nodes,trianges):
    plot(map(x->[nodes[x[1]],nodes[x[2]],nodes[x[3]],nodes[x[1]]],triangles),axes=boxed,
  color=red,scaling=constrained,labels=[x,y]):
  end proc:
```

The **GenerateMesh** procedure allows us to add another frame to the mesh generation move from above:

```
> g := [theta->0,theta->0]:
  Mesh := GenerateMesh(r,n,shells,g):
```

```
nodes := Mesh[1]:
triangles := Mesh[2]:
element := Mesh[3]:
Gamma := Mesh[4]:
BoundaryData := Mesh[5]:
p[5] := DrawMesh2D(nodes,triangles):
display([seq(display([p1,seq(p[j],j=1..i)],title=typeset("Step ",i-1,": ",Labels[i])),i=1.
.4),display([p1,seq(p[j],j=1..3),p[5]],title=typeset("Step ",4,": ",Labels[5]))],
insequence=true,scaling=constrained,axes=boxed);
```



Basis functions

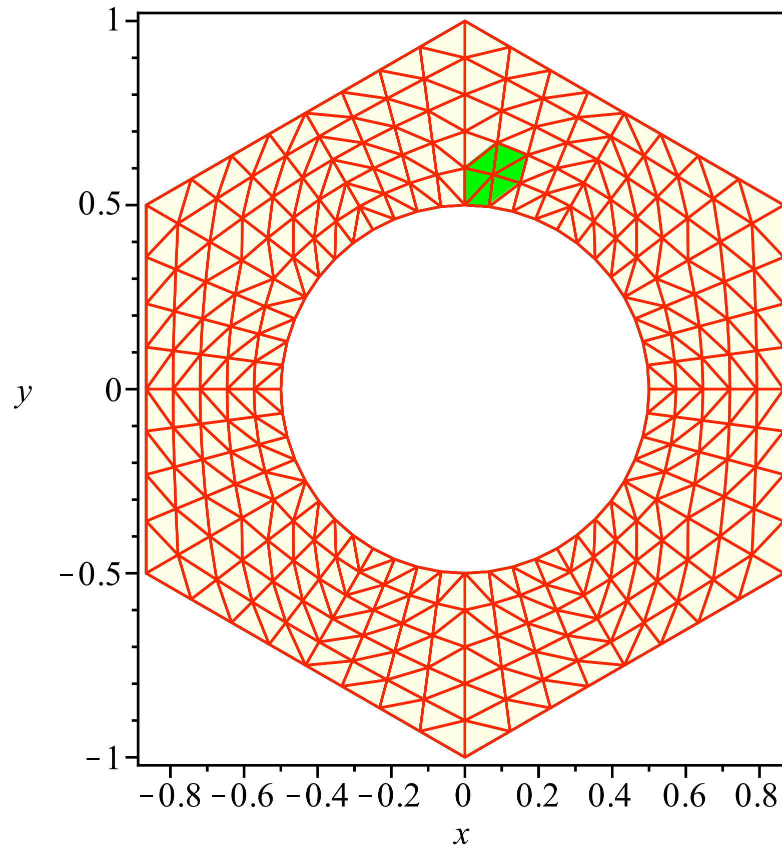
We wish to express the unknown function $u(x, y)$ as a superposition of basis functions $\phi_i(x, y)$. The particular choice of basis is very important and essentially defines the type of finite element method we are using. We will use *piecewise planar* or *piecewise linear* basis functions. These are defined to be continuous linear functions of (x, y) such that $\phi_i(x, y) = 1$ if (x, y) is the position of the ith node and

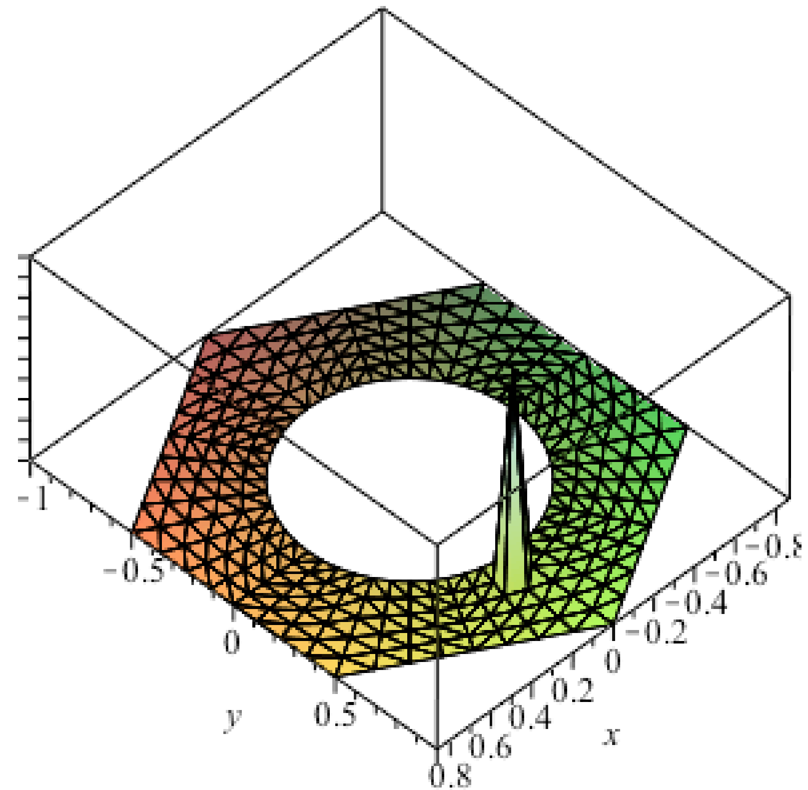
$\phi_i(x, y) = 0$ if (x, y) is the position of the jth node $(i \neq j)$. The definition is easier to see in pictures, which we generate using the following procedures:

```
> # This procedure colours in the region in the xy plane where the i^th basis function is non-zero
  DrawElement2D := proc(i,nodes,triangles,element):
      display(map(j->polygonplot(map(k->nodes[k],triangles[j]),color=green),element[i]));
  end proc:

  # This procedure plots the i^th basis function in 3 dimensions
  DrawElement3D := proc(k,nodes,triangles)
      local N, data, Data, i:

      N[1] := nops(nodes):
      N[2] := nops(triangles):
      data := Vector(1..N[1],datatype=float):
      data[k] := 1;
      for i from 1 to N[2] do;
         Data[i] := Matrix(map(u->[op(nodes[u]),data[u]],triangles[i]),datatype=float[8]);
      od:
      Data := convert(Data,list):
      polygonplot3d(Data,scaling=constrained,axes=boxed);

  end proc:
```

Here are a pair of plots visualizing the nature of the kth basis function. The plot on the left indicates the portion of Ω for which $\phi_k(x, y) \neq 0$, while the plot on the right is the graph of $\phi_k(x, y)$.

```
> k := 60:
  q1 := display([DrawMesh2D(nodes,triangles),DrawElement2D(k,nodes,triangles,element),p1],
  title=typeset("Region where ",phi[k](x,y)<>0," in ",Omega)):
  q2 := display(DrawElement3D(k,nodes,triangles),title=typeset("The ", '''k''' = k," basis
  function"),labels=[x,y,phi[k](x,y)]):
  display(Array([q1,q2]));
```





Finally, for purposes of book-keeping we need to split our basis functions into two categories: those with central nodes in the interior of the domain and those with central nodes on the boundary. Recall that Γ_0 was a list containing all the interior nodes. Let us denote the number of interior nodes by *N3* and let $\Phi_A(x, y)$ represent the basis function corresponding to the Ath node in Γ_0. Also recall that Γ_1 and Γ_2 are list containing nodes on the inner and outer boundary. Let's form another list $\Gamma_3 = \Gamma_1 \cup \Gamma_2$ containing all of the boundary nodes, and let *N4* be

the number of elements in that list. We denote the basis function centered on the Ath node of Γ_3 by $\Upsilon_A(x, y)$.

Weak form of the PDE

We now derive the weak form of the PDE

$$\rho \frac{\partial^2 u}{\partial t^2} + \lambda \frac{\partial u}{\partial t} = \nabla^2 u - R.$$

To do this, we need to multiply the PDE by an interior basis function Φ_A and integrate over Ω. Making use of one of Green's identities

$$\int_\Omega v\, \nabla^2 u \, \mathrm{d}A = \int_{\partial\Omega} v\, (\mathbf{n} \cdot \nabla u)\, \mathrm{d}s - \int_\Omega \nabla v \cdot \nabla u \, \mathrm{d}A,$$

we obtain the weak form of the PDE:

$$\int_\Omega \Phi_A\, \rho \frac{\partial^2 u}{\partial t^2}\, \mathrm{d}A + \int_\Omega \Phi_A\, \lambda \frac{\partial u}{\partial t}\, \mathrm{d}A = \int_{\partial\Omega} \Phi_A\, (\mathbf{n} \cdot \nabla u)\, \mathrm{d}s - \int_\Omega \nabla \Phi_A \cdot \nabla u \, \mathrm{d}A - \int_\Omega \Phi_A\, R \, \mathrm{d}A.$$

Since $\Phi_A(x, y)$ is a basis function centered on an interior node, it vanishes identically on the boundary. Hence, the first term on the righthand side is equal to zero. Now, we write the PDE solution as a superposition of sums over the interior and boundary nodes:

$$u(t, x, y) = \sum_{B=1}^{N3} a_B(t) \Phi_B(x, y) + \sum_{B=1}^{N4} b_B\, \Upsilon_B(x, y).$$

The coefficients of the boundary nodes are fixed by the boundary conditions: b_B will just be the value of $g_1(\theta)$ or $g_2(\theta)$ evaluated at the Bth node in the Γ_3 list (depending on whether it is an inner or outer node). Substituting this decomposition into the weak form of the PDE yields:

$$M\frac{\mathrm{d}^2\mathbf{a}}{\mathrm{d}t^2} + C\frac{\mathrm{d}\mathbf{a}}{\mathrm{d}t} + K\mathbf{a} = \mathbf{f}, \qquad \mathbf{a} = \begin{bmatrix} a_1(t) \\ \vdots \\ a_{N3}(t) \end{bmatrix}$$

$$M = \begin{bmatrix} m_{1,1} & \cdots & m_{1,N3} \\ \vdots & & \vdots \\ m_{N3,1} & \cdots & m_{N3,N3} \end{bmatrix}, \quad C = \begin{bmatrix} c_{1,1} & \cdots & c_{1,N3} \\ \vdots & & \vdots \\ c_{N3,1} & \cdots & c_{N3,N3} \end{bmatrix}, \quad K = \begin{bmatrix} k_{1,1} & \cdots & k_{1,N3} \\ \vdots & & \vdots \\ k_{N3,1} & \cdots & k_{N3,N3} \end{bmatrix}, \quad \boldsymbol{f} = \begin{bmatrix} F_1 \\ \vdots \\ F_{N3} \end{bmatrix},$$

where:

$$m_{A,B} = \int_\Omega \Phi_A \,\rho\, \Phi_B \,\mathrm{d}A, \quad c_{A,B} = \int_\Omega \Phi_A \,\lambda\, \Phi_B \,\mathrm{d}A, \quad k_{A,B} = \int_\Omega \nabla\Phi_A \cdot \nabla\Phi_B \,\mathrm{d}A,$$

$$F_A = -\sum_{B=1}^{N4} b_B \int_\Omega \nabla\Phi_A \cdot \nabla\Upsilon_B \,\mathrm{d}A - \int_\Omega \Phi_A R \,\mathrm{d}A$$

Finite element methods were first used extensively in structural engineering, so the names of the above objects are inherited from that field: M is the mass matrix, C is the damping matrix, K is the stiffness matrix, and $\mathbf{f}$ is the applied force. The three matricies are symmetric. The following two sections describe how to calcuate $\{M, C, K, \mathbf{f}\}$.

Integration on triangles: quadrature formulae (under construction)

In this section, we discuss how to calculate the integrals in the entires of the mass, damping, stiffness and force matrices/vector. Some important features of these integrals follow from the fact that the basis functions are only non-zero on the triangles containing the central nodes. From this we deduce that:

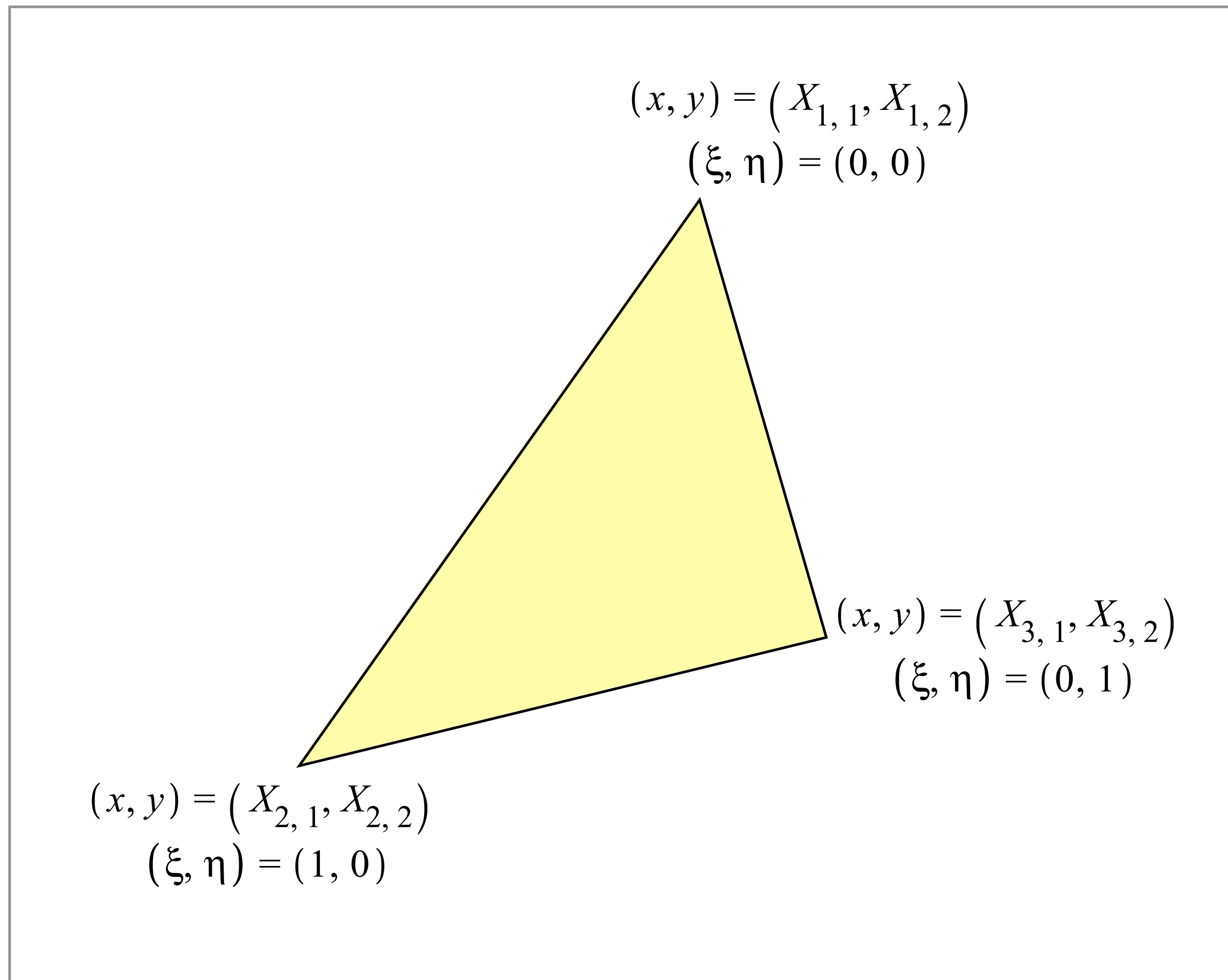
- If a given integral involves two distinct basis functions, the integral will only be non-zero if the central nodes of the basis functions are adjacent to one-another. Furthermore, if the nodes are adjacent the integrand will only be non-zero on triangles that contain both nodes. For the kind of mesh we have constructed, two adjacent nodes will share at most two triangles. (Have a look at the mesh again.)
- If a given integral involves only one basis function, or perhaps one basis function squared (such as in $m_{A,A}$), the region of integration can

be taken to be a sum of integrals over each triangle containing the central node.

In either case, the integrals reduce to a sum of integrals on triangles of the form

$$L = \int_{\triangle} Q(x, y)\, \mathrm{d}A.$$

Here is a scketch of what one of these integration domains might look like:



While it is possible to do these integrals directly in the (x, y) plane, it is a lot easier to change to the so-called "fundamental" coordinates (ξ, η), where the vertices of the triangle reside on points $\{(0,0), (1,0), (1,0)\}$ as labelled in the plot. In such a coordinate system, the integrals we want to calculation will be in the easier-to-handle form

$$L = \int_0^1 \int_0^{1-\xi} Q(x(\xi,\eta), y(\xi,\eta))\, J \,d\eta\, d\xi = \int_0^1 \int_0^{1-\xi} q(\xi,\eta)\, J\, d\eta\, d\xi, \qquad J = \text{abs}\left(\det\begin{bmatrix} \partial x/\partial\xi & \partial x/\partial\eta \\ \partial y/\partial\xi & \partial y/\partial\eta \end{bmatrix}\right)$$

Here, J is the Jacobian of the coordinate transformation (technically, the absolute value of the determinant of the transformation) and $q(\xi,\eta) = Q(x(\xi,\eta), y(\xi,\eta))$ is just the original integrand written in the new (ξ,η) coordinates. For simplicity, we demand that the coordinate tranformation be linear:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{bmatrix}\begin{bmatrix} \xi \\ \eta \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}, \qquad J = |a_{1,1} a_{2,2} - a_{1,2} a_{2,1}|.$$

To find the $a_{i,j}$ and b_i coefficents, we need to enforce that the image of $(x, y) = (X_{1,1}, X_{1,2})$ under the transformation be $(\xi,\eta) = (0, 0)$, the image of $(x, y) = (X_{2,1}, X_{2,2})$ be $(\xi,\eta) = (1, 0)$, and $(x, y) = (X_{3,1}, X_{3,2})$ be $(\xi,\eta) = (0, 1)$. Here is some code that solves for the coefficients assuming arbitary vertex coordinates of the original triangle:

```
> tr := [seq(x[i] = add(a[i,j]*xi[j],j=1..2)+b[i],i=1..2)]:
  Xi := <<0,1,0>|<0,0,1>>:
  sol := solve([seq(seq(subs(seq([xi[i]=Xi[j,i],x[i]=X[j,i]],i=1..2),tr[k]),k=1..2),j=1..3)
  ],{seq(seq(a[i,j],i=1..2),j=1..2),b[1],b[2]});
  J := subs(sol,abs(a[1,1]*a[2,2]-a[1,2]*a[2,1]));
```

$$sol := \{a_{1,1} = X_{2,1} - X_{1,1}, a_{1,2} = X_{3,1} - X_{1,1}, a_{2,1} = X_{2,2} - X_{1,2}, a_{2,2} = X_{3,2} - X_{1,2}, b_1 = X_{1,1}, b_2 = X_{1,2}\}$$

$$J := |(X_{2,1} - X_{1,1})(X_{3,2} - X_{1,2}) - (X_{3,1} - X_{1,1})(X_{2,2} - X_{1,2})| \qquad \textbf{(4.1}$$

Now, note that the area of the original triangle can be calculated by embedding the triangle in the *xy*-plane in 3D space, and calculating $\frac{1}{2}|\mathbf{u} \times \mathbf{v}|$, where **u** and **v** are vectors describing two edges of the triangle:

```
> for i from 1 to 3 do:
      x[i] := <X[i,1],X[i,2],0>;
  od:
```

```
u,v := x[2]-x[1],x[3]-x[1];
w := CrossProduct(u,v);
area = abs(w[3])/2;
```

$$u, v := \begin{bmatrix} X_{2,1} - X_{1,1} \\ X_{2,2} - X_{1,2} \\ 0 \end{bmatrix}, \begin{bmatrix} X_{3,1} - X_{1,1} \\ X_{3,2} - X_{1,2} \\ 0 \end{bmatrix}$$

$$w := \begin{bmatrix} 0 \\ 0 \\ (X_{2,1} - X_{1,1})(X_{3,2} - X_{1,2}) - (X_{3,1} - X_{1,1})(X_{2,2} - X_{1,2}) \end{bmatrix}$$

$$area = \frac{1}{2} \left| (X_{2,1} - X_{1,1})(X_{3,2} - X_{1,2}) - (X_{3,1} - X_{1,1})(X_{2,2} - X_{1,2}) \right| \qquad \textbf{(4.2}$$

Comparing **(4.1)** and **(4.2)**, we see the Jacobian is just 2 × the area of the original triangle. Hence, our integral is

$$L = 2 \times \text{area} \times \int_0^1 \int_0^{1-\xi} q(\xi, \eta)\, d\eta\, d\xi.$$

Now, we can still elect to calculate this directly, but it is more common to evaluates it using a so-called quadrature rules which involve replacing the integrand with a constant value. We will employ a midpoint quadrature rule that involves the approximation

$$q(\xi, \eta) \approx \frac{1}{3}\left[q\left(0, \frac{1}{2}\right) + q\left(\frac{1}{2}, 0\right) + q\left(\frac{1}{2}, \frac{1}{2}\right) \right].$$

That is, we replace the function q with the average of its values at the midpoints of the edges of the triangular integration domain. Once we do this, the remaining integral is trivial:

```
> L[quadrature] := 2*area*Int(Int(1/3*(q(1/2,0)+q(0,1/2)+q(1/2,1/2)),eta=0..1-xi),xi=0..1);
  L[quadrature] := factor(convert(L[quadrature],int));
```

$$L_{quadrature} := 2\, area \left(\int_0^1 \int_0^{1-\xi} \left(\frac{1}{3} q\left(\frac{1}{2}, 0\right) + \frac{1}{3} q\left(0, \frac{1}{2}\right) + \frac{1}{3} q\left(\frac{1}{2}, \frac{1}{2}\right) \right) d\eta\, d\xi \right)$$

$$L_{quadrature} := \frac{1}{3}\, area \left(q\left(\frac{1}{2}, 0\right) + q\left(0, \frac{1}{2}\right) + q\left(\frac{1}{2}, \frac{1}{2}\right) \right) \quad (4.3$$

Written in the original (x, y) coordinates, the midpoint quadrature rule is hence:

$$L \approx \frac{\text{area}}{3}\left[Q(\mathbf{p}_1) + Q(\mathbf{p}_2) + Q(\mathbf{p}_3)\right],$$

where the $\mathbf{p}_i$'s are the midpoints of each edge of the triangle.

We now present procedure to calculate the various triangular integrals we are confronted with when constructing the matrices $\{M, C, K, \mathbf{f}\}$. To actually use the midpoint rule for the integrals, we will need the areas of each triangle in the mesh. This procedure returns a list of the areas, which are calulated by embedding the triangle in the xy-plane in 3D space, and calculating $\frac{1}{2}|\mathbf{u} \times \mathbf{v}|$, where $\mathbf{u}$ and $\mathbf{v}$ are vectors describing two edges of the triangle:

```
> CalculateAreas := proc(nodes,triangles)
      local N, i, X, u, v, areas:
      N[2] := nops(triangles):
      for i from 1 to N[2] do:
         X := Matrix(map(x->nodes[x],triangles[i]));
         u := X[1,1..2]-X[3,1..2];
         v := X[2,1..2]-X[3,1..2];
         areas[i] := abs(Determinant(Matrix([[u],[v]])))/2;
      od:
      convert(areas,list):
   end proc:

   areas := CalculateAreas(nodes,triangles):
```

The following three procedures (`ElevatedTriangle`, `gradient`, `quadrature[1]`) are used for the calculation of integrals of the form

$$J = \int_{\triangle} \nabla\phi_i \cdot \nabla\phi_j \, \mathrm{d}A.$$

This procedure returns the 3-dimensional coordinates (x, y, z) of the nodes in the kth triangle assuming the ith node is at $z = 1$ and the other nodes are at $z = 0$ (it will be used to calculate the gradient of the basis functions):

```
> ElevatedTriangle := proc(i,k,nodes,triangles) local j, X:
```

```
        for j from 1 to 3 do:
            if (triangles[k][j]=i) then:
                X[j] := [op(nodes[triangles[k][j]]),1]:
            else:
                X[j] := [op(nodes[triangles[k][j]]),0]:
            fi:
        od:
        Matrix([X[1],X[2],X[3]]):
    end proc:
```

This procedure calculates the gradient of the ith basis function ($\nabla\phi_i$) in the kth triangle (it makes use of the **ElevatedTriangle** procedure above):

```
> Plane := unapply(solve({seq(X[i,3] = a*X[i,1]+b*X[i,2]+c,i=1..3)},{a,b,c}),X,a,b,c):

  gradient := proc(i,k,nodes,triangles) local X, ans, a, b, c:
     X := ElevatedTriangle(i,k,nodes,triangles);
     #ans := solve({seq(X[j,3] = a*X[j,1]+b*X[j,2]+c,j=1..3)},{a,b,c});
     ans := Plane(X,a,b,c);
     subs(ans,<a,b>);
  end proc:
```

This procedure calculates the integral of $\nabla\phi_i \cdot \nabla\phi_j$ over the kth triangle (this is exact since the gradients are constant over a triangle). (These type of integrals will occur in the calculation of $\{K, \mathbf{f}\}$.)

```
> quadrature[1] := proc(i,j,k,nodes,triangles,areas):
     gradient(i,k,nodes,triangles).gradient(j,k,nodes,triangles)*areas[k]:
  end proc:
```

This procedure uses the midpoint quadrature rule to calculate the integral of $\phi_i\, Q\, \phi_j$ over the kth triangle assuming $i \neq j$. (These type of integrals will occur in the calculation of the off-diagonal elements of $\{M, C\}$.)

```
> quadrature[2] := proc(i,j,k,Q,nodes,triangles,areas)
      local X, populated, n, m, p, ans:
      X := Matrix(map(x->nodes[x],triangles[k])):
      populated := map(x->evalb(x=i or x=j),triangles[k]);
      for n from 1 to 3 do:
         if (n=3) then m:=1 else m:=n+1 fi:
         if (populated[n] and populated[m]) then:
             p := convert((X[n,1..2]+X[m,1..2])/2,list):
             ans := 1/4*areas[k]/3*Q(op(p)):
         fi:
      od:
      ans:
```

```
end proc:
```

This procedure uses the midpoint quadrature rule to calculate the integral of $\phi_i\ Q\ \phi_i$ over the kth triangle. (These type of integrals will occur in the calculation of the diagonal elements of $\{M, C\}$.)

```
> quadrature[3] := proc(i,k,Q,nodes,triangles,areas)
      local X, populated, n, m, p, ans:
      X := Matrix(map(x->nodes[x],triangles[k])):
      populated := map(x->evalb(x=i),triangles[k]);
      ans := 0:
      for n from 1 to 3 do:
         if (n=3) then m:=1 else m:=n+1 fi:
         if (populated[n] or populated[m]) then:
             p := convert((X[n,1..2]+X[m,1..2])/2,list):
             ans := ans + 1/4*areas[k]/3*Q(op(p)):
         fi:
      od:
      ans:
  end proc:
```

This procedure uses the midpoint quadrature rule to calculate the integral of $\phi_i\ R$ over the kth triangle. (These type of integrals will occur in the calculation of **f**.)

```
> quadrature[4] := proc(i,k,R,nodes,triangles,areas)
      local X, populated, n, m, p, ans:
      X := Matrix(map(x->nodes[x],triangles[k])):
      populated := map(x->evalb(x=i),triangles[k]);
      ans := 0:
      for n from 1 to 3 do:
         if (n=3) then m:=1 else m:=n+1 fi:
         if (populated[n] or populated[m]) then:
             p := convert((X[n,1..2]+X[m,1..2])/2,list):
             ans := ans + 1/2*areas[k]/3*R(op(p)):
         fi:
      od:
      ans:
  end proc:
```

Assembling the matrices

With the quadrature formulae of the previous section, we can now describe how to assemble the matrices $\{M, C, K, \mathbf{f}\}$. Here is an overview of the algorithm:

1. Construct a loop over all the interior nodes (i.e., elements of Γ_0). The loop is indexed by $A = 1 \ldots N3$.
2. For each node in the loop (which we call the "central A node"), calculate the diagonal elements of the mass, damping and stiffness matrices (i.e., $\{k_{AA}, c_{AA}, m_{AA}\}$), as well as the $-\int_\Omega \Phi_A R \, \mathrm{d}A$ contribution to F_A. Each of these involves integration over all the triangles adjacent to the central A node, which are given by the **element** list.
3. Find all the nearest neighbour nodes to the central A node, and determine whether they are boundary or interior nodes.
4. If a neighbour is an interior node with position B in the Γ_0 list , calculate its contribution to the off-diagonal elements of the mass, damping and stiffness matrices (i.e., $\{k_{AB}, c_{AB}, m_{AB}\}$). Each of these involves integration over the two triangles shared by the central A node and the neighbour B node.
5. If a neighbour is a boundary node, calculate its contibution to F_A via the term $-\sum_{B=1}^{N4} b_B \int_\Omega \nabla\Phi_A \cdot \nabla\Upsilon_B \, \mathrm{d}A$. Again, the integral will only have support on the two triangles shared by the central and boundary node.
6. Move on to the next interior node.

This algorithm is implemented in **AssembleMatrices** below. That procedure will make use of two other procedures **NodeTest** and **Neighbours**, which we now describe: The **NodeTest** procedure takes the ith node and determines if it is in Γ_0, Γ_1, or Γ_2. It returns a list of two integers $[n, m]$, where n indicates which list contains the node (i.e., $i \in \Gamma_n$), and m is the position of the node in that list.

```
> NodeTest := proc(i,Gamma) local j, n:
     for j from 2 to 0 by -1 do:
         n := Search(i,Gamma[j]):
         if (n<>0) then break fi:
     od:
     [j,n]:
  end proc:
```

The **Neighbours** procedure gives us some detailed information about the nodes immediately adjacent to the ith node. The output is an $n \times 3$ array, where n is the number of nearest neighbours to the ith node. The $[j, 1]$ element of this array is the position of the jth neighbou in the **nodes** list. The $[j, 2]$ element of the array is the output of **NodeTest** on the jth neighbour; i.e., it tells us if that neighbour is on the inner boundary, outer boundary, or interior of the mesh and where it can be found in the **Gamma** lists. Finally, the $[j, 3]$ element is a list of triangle that contain both the ith node and its jth neighbour. Note that if both the ith node and j^{th} neighbour lie on a common boundary, they will share one triangle. Otherwise, they will share two triangles.

```
> Neighbours := proc(i,triangles,element,Gamma)
```

```
    local Triangles, neighbours, n ,m, data, j, shared, k:
    Triangles := element[i];
    neighbours := convert(convert(map(u->op(triangles[u]),element[i]),set) minus {i},list)
;
    n := nops(neighbours);
    m := nops(Triangles);
    data := Array(1..n,1..3):
    for j from 1 to n do:
       data[j,1] := neighbours[j]:
       data[j,2] := NodeTest(data[j,1],Gamma):
       shared := {}:
       for k from 1 to m do:
          if (Search(data[j,1],triangles[Triangles[k]]) <> 0) then:
             shared := shared union {Triangles[k]}:
          fi:
       od:
       data[j,3] := convert(shared,list):
    od:
    data;
end proc:
```

Finally, here is the procedure that actually puts together the matrices using the algoirthm described above:

```
> AssembleMatrices := proc(nodes,triangles,element,Gamma,BoundaryData,areas,rho,lambda,R)
    local N, M, C, K, f, A, i, neighbours, n, j, tri, k, jj, B, b:
    N[3] := nops(Gamma[0]):
    M := Matrix(N[3],N[3],datatype=float,shape=symmetric):
    C := Matrix(N[3],N[3],datatype=float,shape=symmetric):
    K := Matrix(N[3],N[3],datatype=float,shape=symmetric):
    f := Vector(N[3],datatype=float):
    # this is step 1 in the alogorithm (cycling over all interior nodes)
    for A from 1 to N[3] do:
        i := Gamma[0][A]:
        # this is step 2 in the algorithm (filling in diagonal elements and R contribution to the force vector)
        n := nops(element[i]):
        for j from 1 to n do:
            M[A,A] := M[A,A] + quadrature[3](i,element[i][j],rho,nodes,triangles,areas):
            C[A,A] := C[A,A] + quadrature[3](i,element[i][j],lambda,nodes,triangles,areas)
:
            K[A,A] := K[A,A] + quadrature[1](i,i,element[i][j],nodes,triangles,areas):
            f[A] := f[A] - quadrature[4](i,element[i][j],R,nodes,triangles,areas):
        od:
```

```
                # this is step 3 in the algorithm (finding and classifying neighbouring nodes)
                neighbours := Neighbours(i,triangles,element,Gamma):
                n := rhs(ArrayDims(neighbours)[1]);
                # this loop over neighbouring nodes implements steps 4 and 5
                for j from 1 to n do:
                    tri[1] := neighbours[j,3][1]:
                    tri[2] := neighbours[j,3][2]:
                    jj := neighbours[j,1]:
                    if (neighbours[j,2][1]=0) then:
                        # this is step 4 (only executed for neighbours that are interior nodes, fills in off-diagonal elements)
                        B := neighbours[j,2][2]:
                        if (K[A,B]=0) then:
                           M[A,B] := add(quadrature[2](i,jj,tri[k],rho,nodes,triangles,areas),k=1.
.2):
                           C[A,B] := add(quadrature[2](i,jj,tri[k],lambda,nodes,triangles,areas),
k=1..2):
                           K[A,B] := add(quadrature[1](i,jj,tri[k],nodes,triangles,areas),k=1..2):
                        fi:
                    else:
                        # this is step 5 (only executed for neighbours that are boundary nodes, calculate boundary contribution to the
force vector)
                        b := BoundaryData[neighbours[j,2][1]][neighbours[j,2][2]]:
                        f[A] := f[A] - b*add(quadrature[1](i,jj,tri[k],nodes,triangles,areas),k=1.
.2):
                    fi:
                od:
            od:
            [M,C,K,f]:
end proc:
```

Elliptic boundary value problems

In this section, we set $\rho = \lambda = 0$ so that the PDE to solve is Poisson's equation in 2 dimensions:

$$\nabla^2 u = R.$$

In this case, the matrix equation we need to solve for the amplitudes $\mathbf{a}$ of the basis functions is

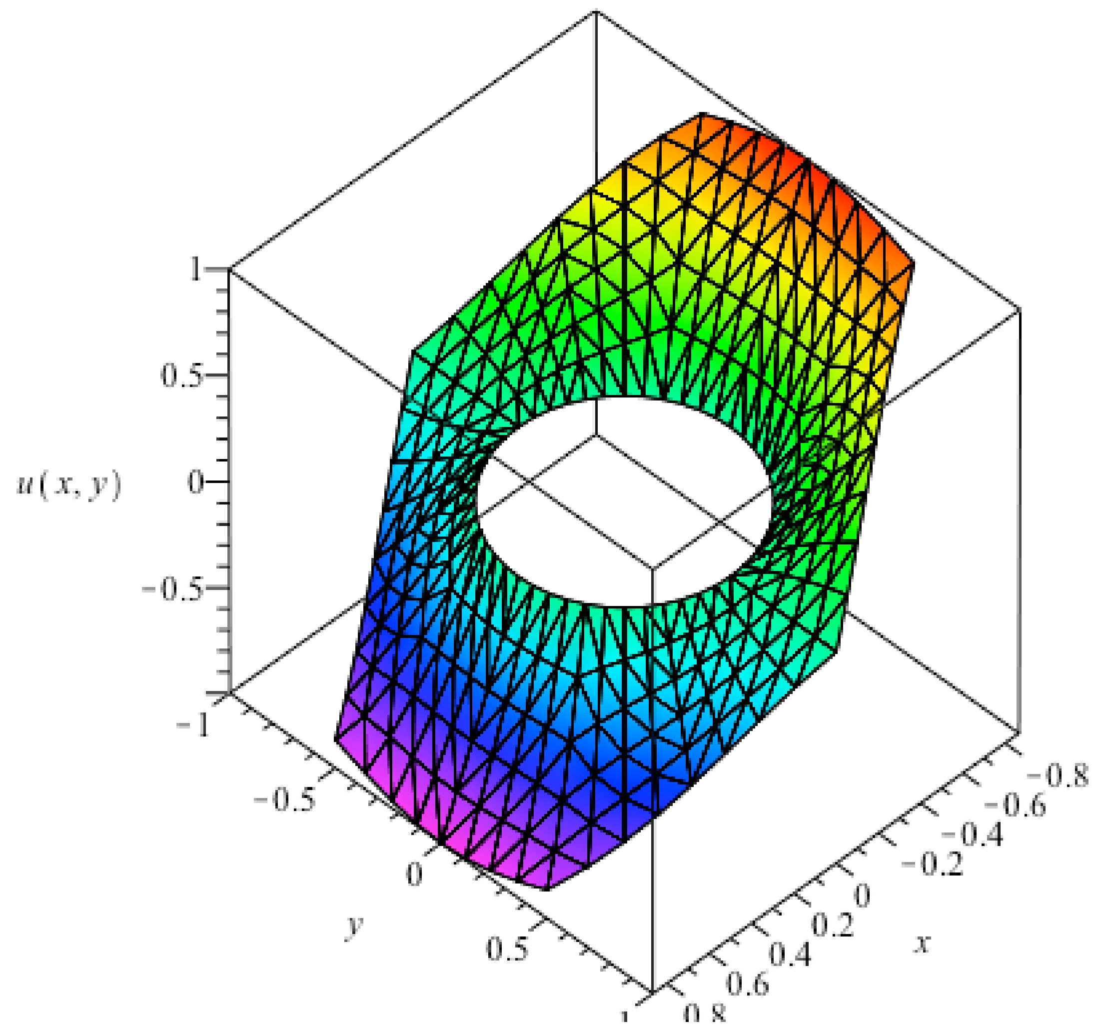
$$K\mathbf{a} = \mathbf{f}.$$

This procedure generates the finite element solution in the region Ω between two polar curves $r_1(\theta)$ and $r_2(\theta)$, with boundary data for u given by $g_1(\theta)$ and $g_2(\theta)$, respectively. As above, **n** is the number of angular gridlines in the mesh and **shells** is the number of concentric shells in the mesh. The output is a 3D plot of the solution.

```
> EllipticSolver := proc(r,n,shells,g,R)
     local rho, lambda, Mesh, nodes, triangles, element, Gamma,
           BoundaryData, areas, Matrices, K, f, a, data, i, Data:

     rho := (x,y) -> 0:
     lambda := (x,y) -> 0:
     Mesh := GenerateMesh(r,n,shells,g):
     nodes := Mesh[1]:
     triangles := Mesh[2]:
     element := Mesh[3]:
     Gamma := Mesh[4]:
     BoundaryData := Mesh[5]:
     areas := CalculateAreas(nodes,triangles):
     Matrices := AssembleMatrices(nodes,triangles,element,Gamma,BoundaryData,areas,rho,
  lambda,R):
     K := Matrices[3]:
     f := Matrices[4]:
     a := convert(LinearSolve(K,f),list):
     data := [op(BoundaryData[1]),op(a),op(BoundaryData[2])]:
     for i from 1 to nops(triangles) do;
        Data[i] := Matrix(map(u->[op(nodes[u]),data[u]],triangles[i]),datatype=float[8]);
     od:
     Data := convert(Data,list):
     polygonplot3d(Data,shading=zhue,scaling=constrained,axes=boxed,labels=[x,y,u(x,y)]);

  end proc:
```

In this example, we set $R = 0$ so we are actually solving Laplace's equation $\nabla^2 u = 0$:

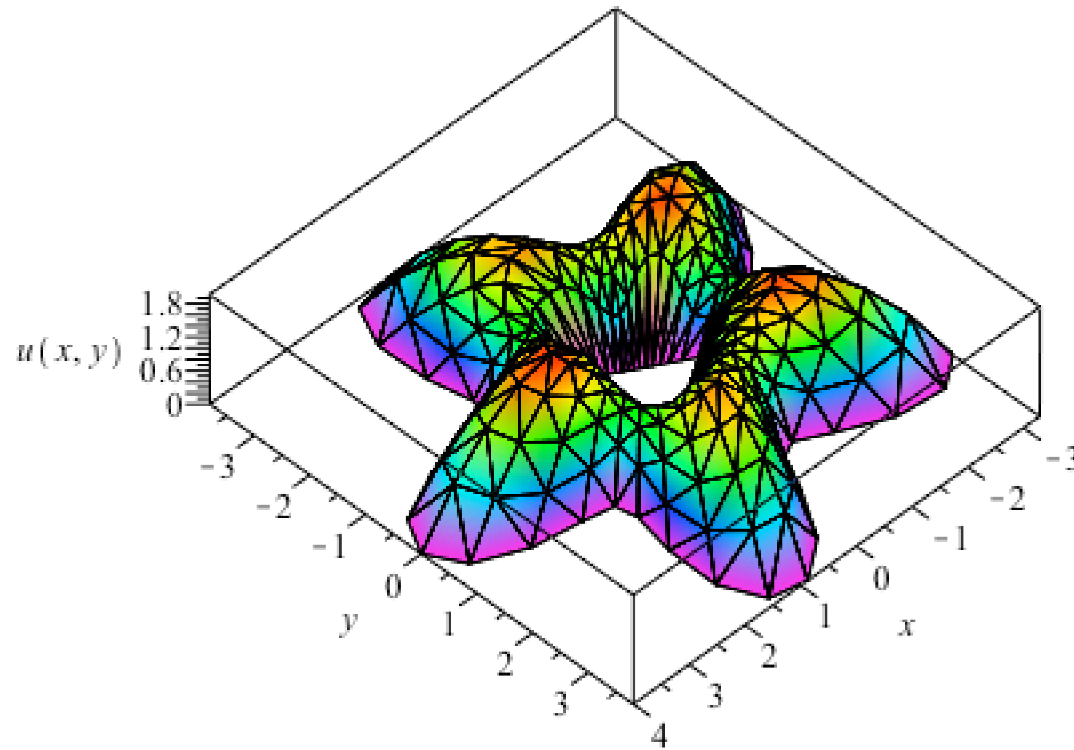
```
> m := 6:
  r := [theta -> 1/2 , theta -> cos(Pi/m)/cos(theta - 2*Pi/m*floor((m*theta + Pi)/(2*Pi)))]:
  n := 4*(2*m):
  shells := 6:
  g := [theta -> 0, theta->-cos(theta)]:
  R := (x,y) -> 0:
  EllipticSolver(r,n,shells,g,R);
```



This example gives the solution of Poisson's equation $\nabla^2 u = -4$:

```
> m := 3:
  r := [theta -> cos(Pi/m)/cos(theta - 2*Pi/m*floor((m*theta + Pi)/(2*Pi))) , theta -> 3+cos
  (5*theta)]:
  n := 7*(2*m):
  shells := 7:
  g := [theta -> 0, theta -> 0]:
  R := (x,y) -> -4:
```

```
EllipticSolver(r,n,shells,g,R);
```



Parabolic initial value problems

In this section, we will concentrate on case when ρ = 0; i.e., the equation we want to solve is:

$$\lambda \frac{\partial u}{\partial t} = \nabla^2 u - R.$$

In this case, the matrix equation we need to solve for the amplitudes **a** of the basis functions is

$$C \frac{d\mathbf{a}}{dt} + K\mathbf{a} = \mathbf{f}.$$

This can be re-arranged to give:

$$\frac{d\mathbf{a}}{dt} = A\mathbf{a} + \mathbf{g}, \quad A = -C^{-1}K, \quad \xi = C^{-1}\mathbf{f}.$$

We solve this matrix ODE by introducing a time lattice $t_i = ih$, where h is the stepsize. We write $\mathbf{a}_i = \mathbf{a}(t_i)$ and use a trapezoidal stencil to generate the numeric solution:

$$\left(I - \frac{h}{2}A\right)\mathbf{a}_{i+1} = \left(I + \frac{h}{2}A\right)\mathbf{a}_i + h\xi.$$

The following procedure generates the finite element solution in the region Ω between two polar curves $r_1(\theta)$ and $r_2(\theta)$, with boundary data for u given by $g_1(\theta)$ and $g_2(\theta)$, respectively. As above, **n** is the number of angular gridlines in the mesh and **shells** is the number of concentric shells in the mesh. Initial data for u is given by $G = G(x, y)$, **N** is the number of timesteps, and **t_max** is the end time of the simulation (we start the simulation at $t = 0$). The output is a movie of the finite element solution.

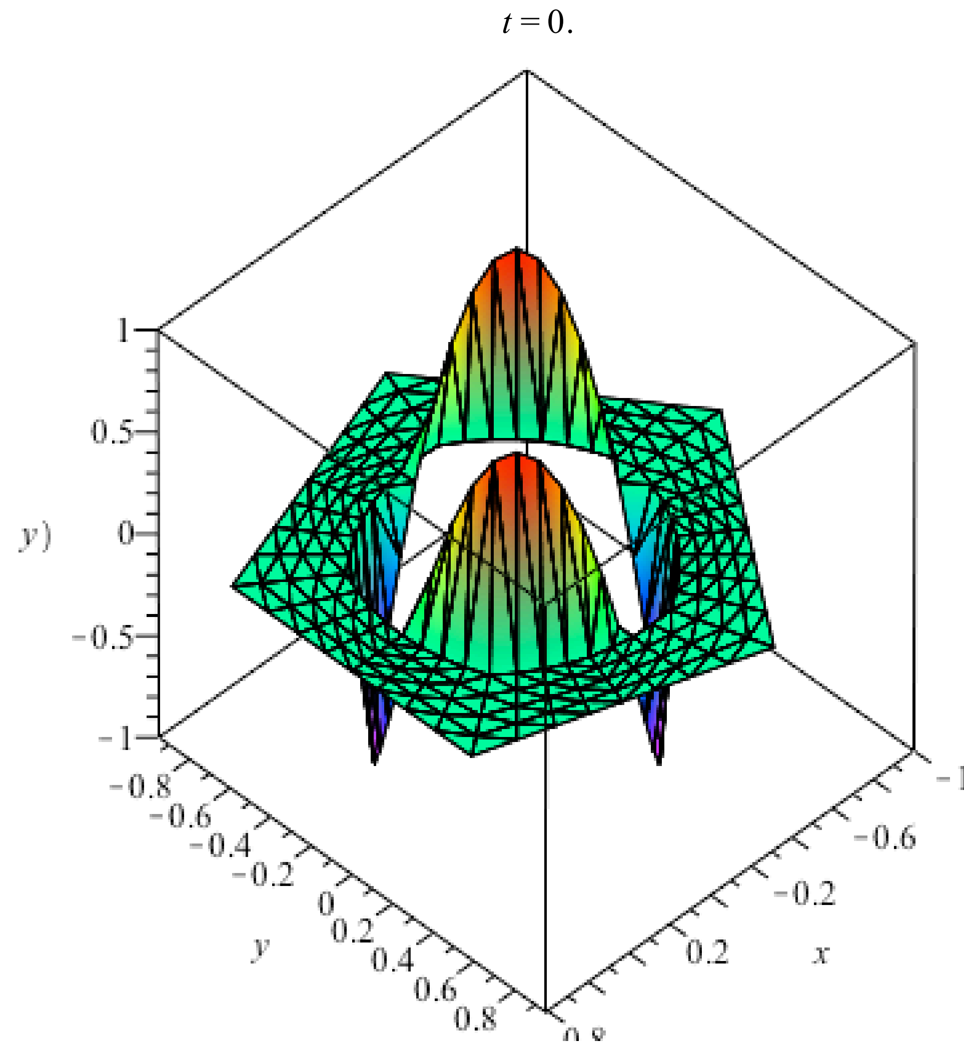
```
> ParabolicSolver := proc(N,t_max,r,n,shells,R,lambda,g,G)
    local rho, Mesh, nodes, triangles, element, Gamma, T, C, p, Cinv,
          BoundaryData, areas, Matrices, K, f, a, i, A, xi, h, zeta:
    T := j -> j/N*t_max:
    rho := (x,y) -> 0:
    Mesh := GenerateMesh(r,n,shells,g):
    nodes := Mesh[1]:
    triangles := Mesh[2]:
    element := Mesh[3]:
    Gamma := Mesh[4]:
    BoundaryData := Mesh[5]:
    areas := CalculateAreas(nodes,triangles):
```

```
   Matrices := AssembleMatrices(nodes,triangles,element,Gamma,BoundaryData,areas,rho,
lambda,R):
   C := Matrices[2]:
   K := Matrices[3]:
   f := Matrices[4]:
   a := convert(evalf(map(u->G(op(nodes[u])),Gamma[0])),Vector):
   p[0] := frame(a,BoundaryData,nodes,triangles,T(0)):
   Cinv := C^(-1):
   A := -Cinv.K:
   xi := Cinv.f:
   h := evalf(T(1)-T(0));
   zeta[1] := 1 - h*A/2:
   zeta[2] := 1 + h*A/2:
   for i from 1 to N do:
      a := LinearSolve(zeta[1],zeta[2].a+h*xi):
      p[i] := frame(a,BoundaryData,nodes,triangles,T(i)):
   od:
   display(convert(p,list),insequence=true):
end proc:

frame := proc(a,BoundaryData,nodes,triangles,T)
   local data, i, Data:
   data := [op(BoundaryData[1]),op(convert(a,list)),op(BoundaryData[2])]:
   for i from 1 to nops(triangles) do;
      Data[i] := Matrix(map(u->[op(nodes[u]),data[u]],triangles[i]),datatype=float[8]);
   od:
   Data := convert(Data,list):
   polygonplot3d(Data,shading=zhue,scaling=constrained,axes=boxed,
                 labels=[x,y,u(t,x,y)],title=typeset(t=evalf[4](T)));
end proc:
```

Here is an example of the output of this procedure as applied to the simple heat equation $\partial_t u = \nabla^2 u$, where u represents the temperature in some body. In this example, the interior of the object and outer boundary have zero initial temperature, while the inner boundary has a non-trivial thermal profile.

```
> N := 50:
  t_max := 0.1:
  m := 5:
  r := [theta -> 1/2,theta -> 1*(cos(Pi/m)/cos(theta - 2*Pi/m*floor((m*theta + Pi)/(2*Pi))))
  ]:
  n := 4*(2*m):
  shells := 6:
```

```
R := (x,y) -> 0:
g := [theta -> sin(2*theta), theta -> 0]:
G := (x,y) -> 0:
lambda := (x,y) -> 1:
ParabolicSolver(N,t_max,r,n,shells,R,lambda,g,G);
```



Hyperbolic initial value problems

In this section, we concentrate on the complete PDE:

$$\rho \frac{\partial^2 u}{\partial t^2} + \lambda \frac{\partial u}{\partial t} = \nabla^2 u - R.$$

In this case, the matrix ODE to solve is

$$M \frac{\mathrm{d}^2 \mathbf{a}}{\mathrm{d}t^2} + C \frac{\mathrm{d}\mathbf{a}}{\mathrm{d}t} + K\mathbf{a} = \mathbf{f}.$$

We convert this into a set of coupled first order ODEs via the definition $\mathbf{b} = \frac{\mathrm{d}\mathbf{a}}{\mathrm{d}t}$:

$$\frac{\mathrm{d}\mathbf{a}}{\mathrm{d}t} = \mathbf{b}, \qquad \frac{\mathrm{d}\mathbf{b}}{\mathrm{d}t} = -M^{-1}K\mathbf{a} - M^{-1}C\mathbf{b} + M^{-1}\mathbf{f},$$

This can in turn be represented by a single matrix ODE:

$$\frac{\mathrm{d}\mathbf{B}}{\mathrm{d}t} = A\mathbf{B} + \xi, \quad \mathbf{B} = \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \end{bmatrix}, \quad A = \begin{bmatrix} 0 & I \\ -M^{-1}K & -M^{-1}C \end{bmatrix}, \quad \xi = \begin{bmatrix} 0 \\ M^{-1}\mathbf{f} \end{bmatrix}.$$

We will solve this with the same type of trapezoidal stencil that we used in the last section; i.e., we write $\mathbf{B}_i = \mathbf{B}(t_i)$ with $t_{i+1} - t_i = h$ and use a trapezoidal stencil to generate the numeric solution:

$$\left(I - \frac{h}{2}A\right)\mathbf{B}_{i+1} = \left(I + \frac{h}{2}A\right)\mathbf{B}_i + h\mathbf{g}.$$

The following procedure generates the finite element solution in the region Ω between two polar curves $r_1(\theta)$ and $r_2(\theta)$, with boundary data for u given by $g_1(\theta)$ and $g_2(\theta)$, respectively. As above, **n** is the number of angular gridlines in the mesh and **shells** is the number of concentric shells in the mesh. Initial data for u and $\partial_t u$ are given by $G = G(x, y)$ and $H = H(x, y)$, respectively, **N** is the number of

timesteps, and `t_max` is the end time of the simulation (we start the simulation at $t = 0$). The output is a movie of the finite element solution

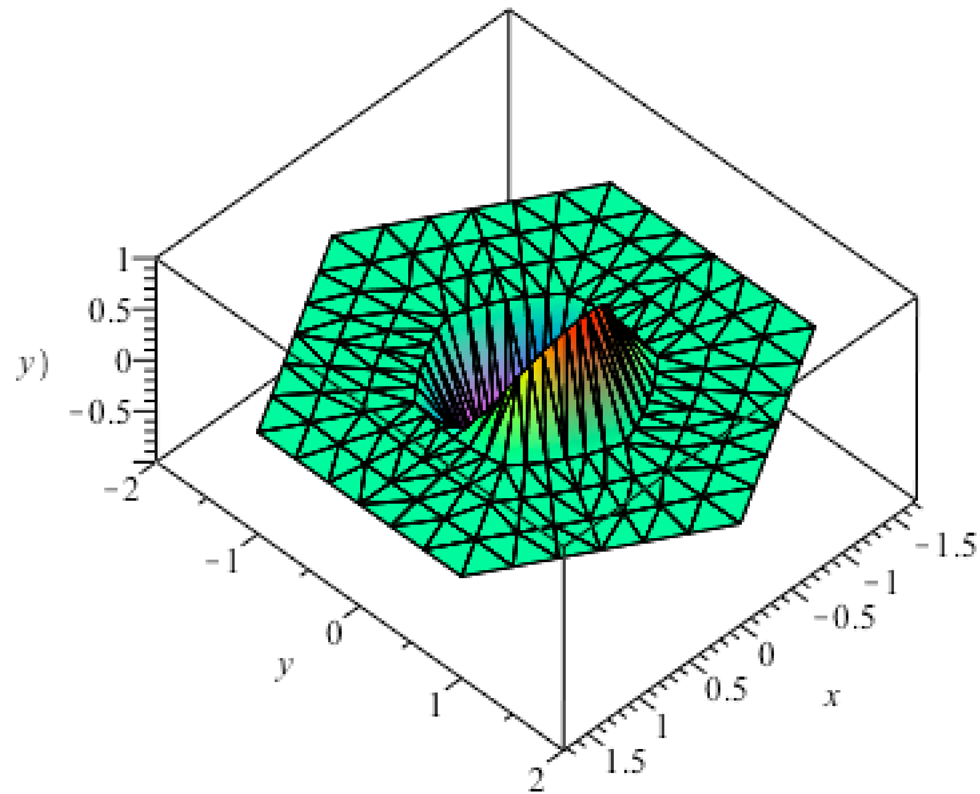
```
> HyperbolicSolver := proc(N,t_max,r,n,shells,R,lambda,rho,g,G,H)
    local Mesh, nodes, triangles, element, Gamma, T, C, p, Cinv, M, b,
          BoundaryData, areas, Matrices, K, f, a, i, A, xi, h, zeta, B,
          N3, Minv:
    T := j -> j/N*t_max:
    Mesh := GenerateMesh(r,n,shells,g):
    nodes := Mesh[1]:
    triangles := Mesh[2]:
    element := Mesh[3]:
    Gamma := Mesh[4]:
    BoundaryData := Mesh[5]:
    areas := CalculateAreas(nodes,triangles):
    Matrices := AssembleMatrices(nodes,triangles,element,Gamma,BoundaryData,areas,rho,
 lambda,R):
    M := Matrices[1]:
    C := Matrices[2]:
    K := Matrices[3]:
    f := Matrices[4]:
    a := convert(evalf(map(u->G(op(nodes[u])),Gamma[0])),Vector):
    b := convert(evalf(map(u->H(op(nodes[u])),Gamma[0])),Vector):
    B := Vector([a,b]):
    N3 := nops(Gamma[0]):
    p[0] := frame(B[1..N3],BoundaryData,nodes,triangles,T(0)):
    Minv := M^(-1):
    A := Matrix([[ZeroMatrix(N3,N3),IdentityMatrix(N3,N3)],[-Minv.K,-Minv.C]]):
    xi := Vector([ZeroMatrix(N3,1),Minv.f]):
    h := evalf(T(1)-T(0));
    zeta[1] := 1 - h*A/2:
    zeta[2] := 1 + h*A/2:
    for i from 1 to N do:
       B := LinearSolve(zeta[1],zeta[2].B+h*xi):
       p[i] := frame(B[1..N3],BoundaryData,nodes,triangles,T(i)):
    od:
    display(convert(p,list),insequence=true):
 end proc:
```

Here is some example output for the wave equation $\partial_t^2 u = \nabla^2 u$.

```
> N := 100:
  t_max := 8:
  m := 6:
```

```
r := [theta -> 1/2,theta -> 2*(cos(Pi/m)/cos(theta - 2*Pi/m*floor((m*theta + Pi)/(2*Pi))))
]:
n := 3*(2*m):
shells := 5:
R := (x,y) -> 0:
g := [theta -> sin(theta), theta -> 0]:
G := (x,y) -> 0:
H := (x,y) -> 0:
lambda := (x,y) -> 0:
rho := (x,y) -> 1:
HyperbolicSolver(N,t_max,r,n,shells,R,lambda,rho,g,G,H);
```

$t = 0.$



Here is another wave equation example:

```
> N := 200:
  t_max := 20:
  m := 6:
  r := [theta -> 1/2,theta -> 2+cos(theta)/2]:
  n := 2*(2*m):
  shells := 5:
  R := (x,y) -> 0:
```

```
g := [theta -> 0, theta -> 0]:
G := (x,y) -> 0:
H := (x,y) -> 1:
lambda := (x,y) -> 0:
rho := (x,y) -> 1:
HyperbolicSolver(N,t_max,r,n,shells,R,lambda,rho,g,G,H);
```

$t = 0.$