Chapter 13: The Finite Element Method

13.1: A NONTECHNICAL OVERVIEW OF THE FINITE ELEMENT METHOD

The **Finite Element Method** (**FEM**) is actually a large collection of numerical methods for solving PDEs. It was first devised as a numerical tool by mathematician Richard Courant[1] in a 1943 paper on torsion problems [Cou-43], and is based on analogous techniques and principles to those developed in the early twentieth century for one-dimensional boundary value problems, as were presented in Section 10.5. The method was extensively elaborated on during the 1950s and 1960s by engineers as a practical approach for solving various PDEs in structural engineering. In the 1960s and 1970s, mathematicians worked to give the method a solid theoretical basis and extended it as a tool for solving many different PDE problems. Active research on this method continues today and it has become the most commonly used numerical method for partial differential equations. As a very pertinent example, in MATLAB's PDE Toolbox, all of the programs for solving PDEs use FEMs. Writing even somewhat general programs for FEMs is a very complicated task. Our goal in this chapter will be to explain the method and to write some programs to implement it in several specific instances. This should be sufficient for readers needing to delve deeper into FEMs to be able to extend the programs into more general ones. MATLAB's PDE Symbolic Toolbox programs are open to its users to read and modify. So, in principle, after reading this chapter, readers could modify some of the FEM programs in MATLAB's toolbox to suit their exact needs (if not already met).

---

[1] Richard Courant was an exceptionally influential mathematician. He grew up in Germany and had a rather difficult childhood, having to work to help support his family while going to school. He eventually entered the Univerisity of Breslau (now Wroclaw, Poland) as an undergraduate and was lured to major in mathematics by the exciting lectures in his classes. He went on to Göttingen for his graduate studies where he worked with Hilbert and later became a professor there. His education was interrupted by military service for Germany in WWI, where he developed an effective electronic communications system that was implemented for the troops. Despite the important contributions he was making to the University of Göttingen, not to mention his important military service to his country, when the Nazis came to power in the early 1930s, he was forced to resign his professorship. The Nazis had decreed that any "non-Aryan" civil servant was to be terminated and having one Jewish grandparent was sufficient to make someone "non-Aryan." There was supposed to be an exemption for individuals who gave Germany military service in WWI, but despite ardent efforts on the part of the university to keep him, Courant was still "retired." He subsequently accepted an offer at New York University. The transition was very difficult for him. Coming from a world-renowned institute and having been surrounded by top-notch mathematicians, when he got to NYU, he found his colleagues to be very weak and the students likewise poor. He made use, however, of his extensive contacts and was able to hire a large group of strong new faculty. Today, NYU's mathematics department, also known as the *Courant Institute*, is considered by many as the premiere applied mathematics institute in the world.

Figure 13.1: Richard Courant (1888–1972), German mathematician.

The FEM methods basically split up the domain of the problem into small pieces, called **elements**, that have simple structure. There are many different ways to perform such decompositions and the geometry certainly changes with the dimension of the space. A common approach for two-dimensional domains is to **triangulate** the domain into small triangles. The **triangulation** must be done in such a way that whenever two triangles touch, they will have either an entire common edge (and thus two common vertices) or just a common vertex. The reason for this is that the FEM approximate solution for the PDE will be made up of separate "pieces" on the various elements and they need to connect up (interpolate) together in a neat fashion. When a domain has a curved boundary, the sizes of the triangles can be made small enough so that the triangulation approximates the domain reasonably well. An example of such a triangulation is shown in Figure 13.2. In three dimensions, tetrahedra (pyramids) are commonly used; but the process is still known as triangulation.

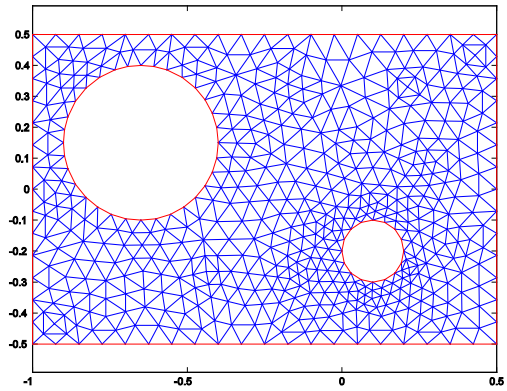


FIGURE 13.2: A triangulation of a planar domain consisting of a rectangle with two circles deleted. The circular boundary portions are thus approximated by polygons (as shown). This triangulation was created using MATLAB's PDE Toolbox.

Even in two dimensions, of course, there are numerous ways to perform such a triangulation. We point out some important features of the triangulation in Figure 13.2. First, notice that the triangles in the mesh all seem to be roughly the same size. This uniformity is not necessary, but for a general problem on a given domain it is usually the best generic triangulation scheme. Another important property is that none of the sidelengths of any triangle in the mesh is much shorter than the other two sides of the same triangle. Another way to describe this property is that the area of the inscribed circle of any of these triangles is not much

smaller than the area of the triangle. This feature is essential in order that the finite element method be stable.

Just writing a good program to create such generic triangulations is already an arduous task. It must be thought out how the geometry of the original domain should be inputted (as a matrix) and then the triangles must be created and stored, usually by their vertices. Additionally, the vertices of the triangulation will need to be numbered and it will be helpful for the numbering to be done in such a way that the numbers of the three vertices of any triangle are reasonably close.

Like finite difference methods, finite element methods will discretize the PDE into a linear system. The nature of the discretization, however, is very different for FEMs. Mathematically, the PDE is first converted to a so-called **variational problem**. This is usually done in one of two ways: the **Rayleigh-Ritz method**, where the solution of the PDE is recast as the solution of a certain minimization problem among a large class of functions, or the **Galerkin method**, where the solution is recast as a certain unique representing function. Although different in philosophy, the two approaches often turn out to be equivalent. With either method, the approximate solution is found by restricting attention to a certain finite-dimensional space of so-called **admissible functions** determined by **basis functions** corresponding to each of the elements. Even with the type of elements being specified, there are numerous choices for the basis functions. The simplest choice would be constant functions, but these do not blend together well. The next simplest choice would be to have linear functions on each element. For two-dimensional domains with triangulations, this type of basis function turns out to be quite effective. Over each element, the graph of such a basis function will be the triangular portion of a plane (sitting over the two-dimensional triangle). Since three points determine a plane, these basis functions will be flexible enough to accommodate specifying values at the three vertices of their triangle, and mesh triangles that have common vertices or edges will have their graphs coinciding at common points. The resulting approximating functions will thus be continuous over the original domain, but in general will not be differentiable at common edges or vertices of different triangles. More complicated spline-type basis functions can be used to overcome this differentiability problem, but, of course, the limited benefits of using such more complicated basis functions would have to be weighed against the resulting increase in technical difficulties. For most applications on two-dimensional domains with triangular elements, such linear basis functions are sufficient and most commonly used. MATLAB's PDE Toolbox, for example, uses such basis functions.[2]

[2] The programs in MATLAB's PDE toolbox are designed only to handle PDEs with two space variables, and so, for example, they cannot solve three-dimensional elliptic problems such as steady-state heat distributions and structure of materials. The programs can, however, accommodate a time variable in hyperbolic or parabolic equations. The reason for this is that FEMs for parabolic and hyperbolic problems invoke finite difference schemes for the time derivatives and thus can accommodate two space variables in addition to the time variable.

Focusing only on (linear combinations of) the basis functions, the FEM will solve a linear system to determine the best candidate to solve the variational problem (Rayleigh-Ritz or Galerkin) and this will be the approximation to the actual solution. This approximation turns out to be simply the projection of the actual solution onto the finite-dimensional space of admissible functions or, informally, the best admissible function for solving the variational problem. Figure 13.3 shows the FEM solution for the following PDE problem on the domain Ω of Figure 13.2:

$$\begin{cases} \Delta u = 0 \text{ on } \Omega, \quad u = u(x, y) \\ \partial u / \partial n = 0, \text{ on outer rectangle} \\ u = 40, \text{ on small circle, } u = 500, \text{ on large circle} \end{cases} . \tag{1}$$

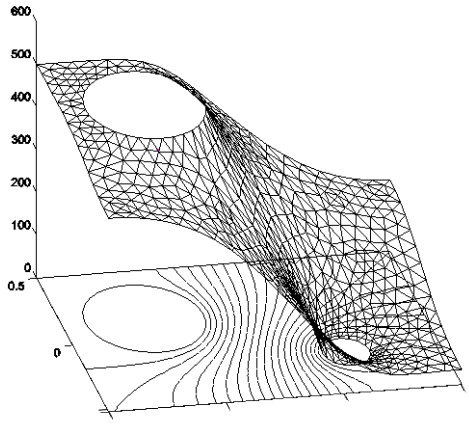


FIGURE 13.3: A FEM solution of the steady-state heat problem (1) on the domain and triangulation as pictured in Figure 13.2. Contour lines have been added. This solution was created using MATLAB's PDE Toolbox.

The problem can be thought of as the steady-state heat distribution on the rectangular region Ω (from the Laplace equation $\Delta u = 0$). The first boundary condition on the edge of the outer rectangle: $\partial u / \partial n = 0$, is a Neumann boundary condition (n denotes the unit outward normal vector for the domain), stating that heat does not flow out of or into the rectangle (i.e., the boundary is insulated). The two constant temperatures on the interior circular boundaries are Dirichlet boundary conditions specifying certain temperatures that are fixed on each. The problem may be thought of as a basic version of the cooling of a nuclear reactor within some enclosed region (the rectangle); the large very hot circle denoting the reactor and the small circle denoting the cooling source (usually a stream of fresh water).

In the triangulation process, it is not always efficient to make the triangles be all of essentially the same size. Indeed, at places where the solution varies

drastically, smaller triangles should be used and in areas of small variation larger ones can be and should be used. More triangles entail more work so we should use very small ones only where they are needed. Of course, not knowing the solution ahead of time can make it difficult to predict where the solution will be varying wildly. Sharp corners or curves in the domain, as well as areas where the coefficients of the PDE (if variable) rapidly change, are usually problem areas. There are more sophisticated so-called **adaptive methods** of the FEM that iteratively take into account all available information so as to refine the elements accordingly in a way that aims to reach the best possible accuracy with specified constraints (such as operating time, number of triangles, etc.). Figure 13.4 shows such an adaptive triangulation for the boundary value problem (1). Triangulation of domains is an art!

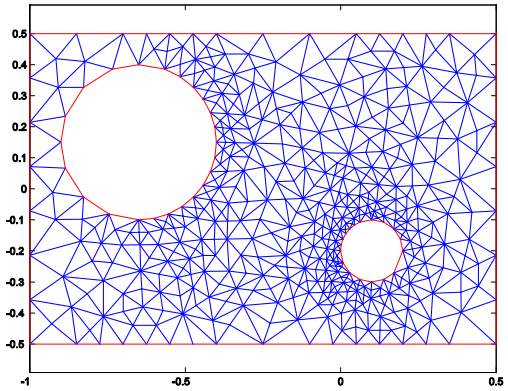


FIGURE 13.4: A triangulation of the planar domain of Figure 13.2 that was obtained using an adaptive FEM to solve the steady-state heat problem (1). Compare with the triangulation of Figure 13.2 and, in particular, notice how the triangles closer to the boundaries of the circles (facing inward) are much smaller than the farther away triangles. This triangulation was created using MATLAB's PDE Toolbox.

An outline for the rest of this chapter is as follows: We will be focusing on linear elliptic boundary value problems on planar domains. Our FEM will use piecewise linear basis functions on triangulations of the domains. Section 13.2 will introduce some practical techniques for producing effective triangulations of planar domains and explain how to construct and manipulate basis functions. Section 13.3 will explain the complete program of using a FEM to solve quite general boundary value problems on arbitrary planar domains and boundary conditions. The most time-consuming step of the FEM is the construction of the linear system whose solution will give the values of the approximate solution. This process, known as "the assembly process," is broken up into an element-by-element computation involving the calculation of certain double integrals and (depending on the boundary data) line integrals. We will demonstrate with examples that it is not efficient to use MATLAB's integration tools in the assembly process. Indeed, if the elements are sufficiently small (depending on the

coefficients of the problem), it turns out to be perfectly adequate to use some simple numerical quadrature formulas (for triangles and line segments). This will allow us to attain essentially the same accuracy as with the more elaborate integrators but at a very small fraction of the time. In numerical differential equations, much can be learned from experimentation, and this chapter provides numerous opportunities in this area. The committed reader can gain a great deal by exploring some of the more advanced topics that are introduced in the exercises.

13.2: TWO-DIMENSIONAL MESH GENERATION AND BASIS FUNCTIONS

Theoretically, the finite element method for two-dimensional problems shares many common threads with the one-dimensional Rayleigh-Ritz methods introduced in Section 10.5. It would behoove the reader to glance over that section periodically as he or she proceeds to work through this and the next section. The major practical difference is in the geometry of the two-dimensional elements and basis functions versus the very simple one-dimensional elements. We will restrict our focus in the text of this chapter to triangular elements and piecewise linear basis functions, although some of the exercises will delve into other sorts of elements and basis functions. In this section we show how triangulations can be created and give some convenient methods for constructing, storing, and manipulating corresponding basis functions.

The two main advantages of the FEM over finite difference methods are its ease in dealing with more complicated domains than simply rectangular ones, and its flexibility in dealing with many sorts of boundary conditions. To illustrate construction of the basis functions, we use the simple triangulation of the hexagonal domain shown in Figure 13.5.

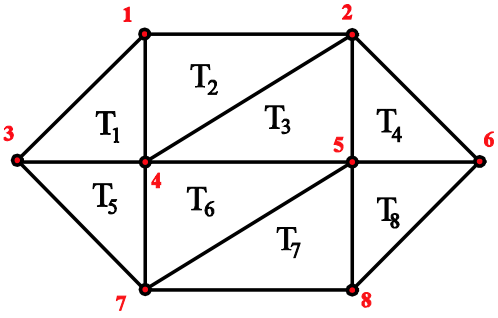


FIGURE 13.5: A simple triangulation of a hexagonal domain. The eight nodes are labeled in red and the eight triangles are also labeled. The ordering is somewhat arbitrary. It is just a coincidence that the number of nodes coincides with the number of triangles. At this point we left out *x*- and *y*-coordinates so as to emphasize the element numbering.

The **nodes** in a triangulation are simply the vertices of the triangles. As in the one-dimensional method, each node gives rise to a basis function that takes on the value 1 at its corresponding node and zero at all other nodes. Piecewise linear

functions work well here since a linear function is completely determined on a triangle once its values are specified on the three vertices. Furthermore, two linear functions so determined on triangles that share an edge will agree on the common edge. The resulting basis function will be the unique piecewise linear function on the hexagon having the property that it is linear on each element and takes on the value 1 at its associated node and 0 at all other nodes. In this context the basis functions are sometimes known as **pyramid functions**. The pyramid function for the node #4 of Figure 13.5 is illustrated in Figure 13.6.

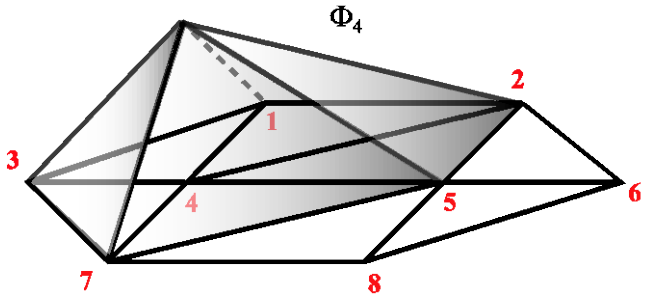


FIGURE 13.6: A graph of the piecewise linear basis or pyramid function $\Phi_4 = \Phi_4(x, y)$ for node #4 in the triangulation of the hexagonal domain of Figure 13.5. The function takes on the value 1 at node #4, zero at all other nodes, and is linear on each triangle. Thus, on the unshaded triangles, the pyramid function is identically zero.

To get formulas for the pyramid functions, we need to introduce coordinates. For the purpose of an example, we assign coordinates as follows: node #3 will be put at the origin (0,0), node #4 will have coordinates (1,0), and the coordinates of nodes #1 and #7 are (1,1), and (1,–1), respectively. The corresponding coordinates of nodes #2, #5, and #8 are obtained by adding 3/2 to the first coordinates of the last three, and node #6 has coordinates (7/2,0). Knowing these coordinates, all of the information of the triangulation can be represented by the following two matrices, *N*(odes) and *T*(riangles):

$$N = \begin{bmatrix} 1 & 1 \\ 2.5 & 1 \\ 0 & 0 \\ 1 & 0 \\ 2.5 & 0 \\ 3.5 & 0 \\ 1 & -1 \\ 2.5 & -1 \end{bmatrix}, \quad T = \begin{bmatrix} 1 & 3 & 4 \\ 1 & 2 & 4 \\ 2 & 4 & 5 \\ 2 & 5 & 6 \\ 3 & 4 & 7 \\ 4 & 5 & 7 \\ 5 & 7 & 8 \\ 5 & 6 & 8 \end{bmatrix}.$$

The eight rows of *N* give the coordinates of the corresponding numbered nodes, the first column entry gives the *x*-coordinates, and the second column entry the *y*-coordinates. The eight rows of *T* give the node numbers of the corresponding

numbered triangles, in order (see Figure 13.5). Such matrices will be needed in writing programs for the FEM.

EXAMPLE 13.1: Write down a formula for the basis function $\Phi_4 = \Phi_4(x, y)$ shown in Figure 13.6.

SOLUTION: From its piecewise linearity, on each of the eight triangles $T_\ell\,(1 \le \ell \le 8)$, $\Phi_4(x, y)$ will be a linear function and so can be written as

$$\Phi_4(x, y) = a_\ell{}^4 x + b_\ell{}^4 y + c_\ell{}^4 = a_\ell x + b_\ell y + c_\ell\,, \quad (x, y) \in T_\ell\,, \tag{2}$$

where $a_\ell{}^4 = a_\ell$, $b_\ell{}^4 = b_\ell$, $c_\ell{}^4 = c_\ell$ are real constants to be determined.[3] We now fix an index ℓ and let the three nodes of T_ℓ be denoted by (x_r, y_r), (x_s, y_s), and (x_t, y_t) where $t = 4$. The graph of such a linear function $z = \Phi_4(x, y)$ is a plane in three-dimensional space determined by the three nodal values $\Phi_4(x_r, y_r) = 0$, $\Phi_4(x_s, y_s) = 0$, and $\Phi_4(x_t, y_t) = 1$. These nodal equations may be expressed as the following linear system:

$$\begin{cases} a_\ell x_r + b_\ell y_r + c_\ell = 0 \\ a_\ell x_s + b_\ell y_s + c_\ell = 0 \\ a_\ell x_t + b_\ell y_t + c_\ell = 1 \end{cases}. \tag{3}$$

Putting (3) in matrix notation gives:

$$MA = Z, \text{ where M} = \begin{bmatrix} x_r & y_r & 1 \\ x_s & y_s & 1 \\ x_t & y_t & 1 \end{bmatrix}, \; A = \begin{bmatrix} a_\ell \\ b_\ell \\ c_\ell \end{bmatrix}, \text{ and } \; Z = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}. \tag{4}$$

Geometrically, since three noncollinear points determine a unique plane, it follows that the linear system (3)/(4) will have a unique solution as long as the three nodes are not collinear. This is certainly the case for any triangulation. We mention one further important point that the system will be well conditioned provided that the area of the triangle T_ℓ is not much greater than that of the inscribed circle. This is a quantitive way of saying that the three nodes of T_ℓ should not be close to being collinear (convince yourself of this!). This can be analytically verified using the following explicit formulas for the determinant of M and M^{-1}:

$$|\det(M)| = 2 \cdot \text{area}(T_\ell), \tag{5}$$

[3] The superscipts, although technically necessary, can be omitted in this example since there is only one basis function under consideration.

$$M^{-1} = \frac{1}{\det(M)} \begin{bmatrix} (y_s - y_t) & (y_t - y_r) & (y_r - y_s) \\ (x_t - x_s) & (x_r - x_t) & (x_s - x_r) \\ (x_s y_t - x_t y_s) & (x_t y_r - x_r y_t) & (x_r y_s - x_s y_r) \end{bmatrix}. \quad (6)$$

For a proof of the interesting equation (5), the reader is referred to Exercise 22. Using (5), equation (6) can be proved by a direct (albeit tedious) verification. Equations (5) and (6) plainly show that the matrix is well conditioned as long as the triangle is not too long and thin. Apart from this, the explicit formula (6) is useful to build into large-scale FEM programs where such matrices need to be inverted large numbers of times in constructing the basis functions. We continue with this example in a way that will help us later when we need to write general programs. We begin by entering the node matrix *N* and the triangle matrix *T* into our MATLAB session:

```
>> N=[1 1;5/2 1;0 0;1 0;5/2 0;7/2 0;1 -1;2.5 -1];
>> T=[1 3 4;1 2 4;2 4 5;2 5 6;3 4 7;4 5 7;5 7 8;5 6 8];
```

Since Φ_4 vanishes over triangles #4, #7, and #8, the coefficients *a*, *b*, *c* are all zero for these triangles. The following loop will give us what we need and is easily modified to function in general FEM routines. It will store the needed coefficients of Φ_4 on the remaining triangles in a four-column matrix *A*: The first column gives the triangle number; the remaining three give the corresponding coefficients of Φ_4 as in (2). Since the coefficients are all fractions, we display the output in rational format.

```
>> format rat, counter=1;
>> for L=1:8
if ismember(4,T(L,:))==1
%checks to see if 4 is a node of triangle #L
%if yes, next two commands reorder the vector T(L,:) to
%construct a vector "nv" of length 3
%of nodes of triangle #L with 4 appearing last
  index=find(T(L,:)==4);
  nv=[T(L,1:2) 4]; nv(index)=T(L,3);
  xr=N(nv(1),1); yr=N(nv(1),2); xs=N(nv(2),1); ys=N(nv(2),2);
  xt=N(nv(3),1);yt=N(nv(3),2);
  M=[xr yr 1;xs ys 1;xt yt 1]; %matrix M of (4)
Minv=[ys-yt yt-yr yr-ys; xt-xs xr-xt xs-xr; xs*yt-xt*ys xt*yr-xr*yt
xr*ys-xs*yr]/det(M);
  % inverse matrix M from (6)
  abccoeff=Minv*[0;0;1]; %coefficents of basis function on triangle#L
  A(counter,:)=[L abccoeff'];
  counter=counter+1;
end
end
>> A
→A =          1      1      -1      0
              2      0      -1      1
              3    -2/3      0     5/3
              5      1      1      0
              6    -2/3      1     5/3
```

From this matrix, we can write down the explicit formula for Φ_4:

$$\Phi_4(x,y)=\begin{cases} x-y, & \text{if } (x,y)\in T_1,\\ -y+1, & \text{if } (x,y)\in T_2,\\ -\frac{2}{3}x+\frac{5}{3}, & \text{if } (x,y)\in T_3,\\ x+y, & \text{if } (x,y)\in T_5,\\ -\frac{2}{3}x+y+\frac{5}{3}, & \text{if } (x,y)\in T_6,\\ 0, & \text{otherwise.}\end{cases}$$

The reader should verify that these formulas indeed possess the required values at the nodes and hence (by linearity) on each element.

EXERCISE FOR THE READER 13.1: Find formulas for Φ_3 and Φ_5 analogous to that found for Φ_4 in the above example.

The careful reader may have realized that we can further cut our computation time down in the solution of the system (4) if we always agree to set it up so that (x_t, y_t) is the vertex on which the value of the local basis function equals 1. Since the inverse of the coefficient matrix *M* of (4) is explicitly known (6), the matrix product *MZ* will simply be the third column of M^{-1} so that in the notation of (4), we have (using (5)):

$$\begin{bmatrix} a_\ell \\ b_\ell \\ c_\ell \end{bmatrix} = \frac{1}{2\cdot \text{area}(T_\ell)} \begin{bmatrix} y_r - y_s \\ x_s - x_r \\ x_r y_s - x_s y_r \end{bmatrix}. \tag{6a}$$

Up to now, most of our plots for functions of two variables have been over rectangular domains. Thus it will be important for us to learn how to get MATLAB to plot functions, such as the above basis functions, that are piecewise linear and continuous on a set of triangular finite elements. Such functions are determined entirely by their nodal values. MATLAB can accommodate us quite nicely for this task with the following command:

`trimesh(T,x,y,z,C)` →	Given a 3-column matrix `T` whose rows are node numbers for a triangulation, vectors `x` and `y` of the coordinates of the numbered nodes, and corresponding `z` coordinates of a piecewise linear function on the nodes, this command will produce a plot of the resulting piecewise linear function. The last argument `C` is an optional *rgb* vector that can be used to specify color (see Section 7.2). The default edge coloring is proportional to the edge height as in Chapter 11. The vector `z` can be omitted to produce a two-dimensional plot of the triangulation.

We are nicely set up to have MATLAB construct a plot of Φ_4.

```
>> x=N(:,1); y=N(:,2);
>> z=zeros(8,1); z(4)=1;
>> trimesh(T,x,y,z)
>> hidden off %allows hidden edges to appear
```

The resulting MATLAB plot is shown in Figure 13.7. Since there are only two heights for the edges, the coloring is not very elaborate. With finer triangulations and more complicated functions, the resulting trimesh plots can be quite useful and attractive, as the one shown in Figure 13.2.

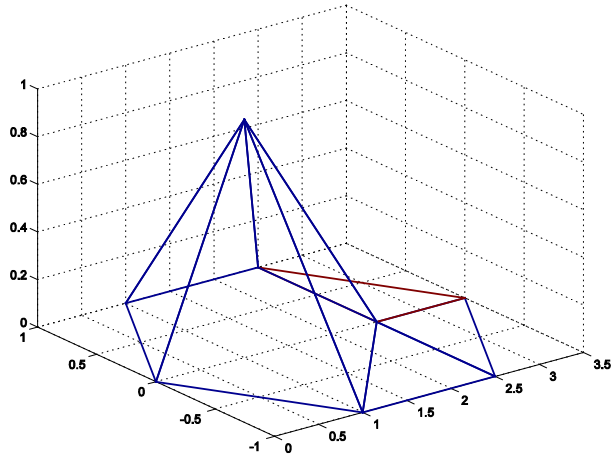


FIGURE 13.7: MATLAB's graphical rendition of the basis function $\Phi_4(x, y)$ of Example 13.1.

Given a triangulation of a domain and any function or data f defined on the nodes $N_1, N_2, \cdots, N_m$, the **finite element interpolant** of this function/data is given by:

$$\sum_{j=1}^{n} f(N_j)\Phi_{N_j}(x, y).$$

We point out that the graph of this interpolant is most easily obtained by simply using the trimesh command directly on the triangle matrix and corresponding values of f; the calculations for the basis functions are not necessary. This will not be the case for more general elements (see Exercise 26).

We stress that in the determination of the hat functions Φ_j,

Φ_j on each element (triangle). On each element T_ℓ,

$\Phi_j(x, y) = a_\ell^j x + b_\ell^j y + c_\ell^j$, x, y, and

1 T_ℓ

Φ_j. Although these local basis functions are quite natural and have simple formulas, there is another local basis that often has theoretical advantages. To simplify notation, we fix an element $T = T_\ell$, and denote its three vertices by

$v_1, v_2,$ and v_3 (the exact ordering is unimportant, but let's assume they are numbered in counterclockwise order). The corresponding three **standard local basis** functions $\phi_1, \phi_2,$ and ϕ_3 are the linear functions determined (exactly) by the following conditions:

$$\phi_i(v_j) = \delta_{ij} \equiv \begin{cases} 1, & \text{if } i = j, \\ 0, & \text{if } i \neq j. \end{cases} \tag{7}$$

(The symbol δ_{ij} is called the **Kronecker delta symbol**.) It was described earlier how each of these functions can be expressed using the original local basis functions. In terms of these local basis functions, any linear function $\phi(x, y)$ can be conveniently expressed as:

$$\phi(x, y) = \phi(v_1)\phi_1(x, y) + \phi(v_2)\phi_2(x, y) + \phi(v_3)\phi_3(x, y). \tag{8}$$

(*Proof:* Both sides are linear functions that agree at the three noncollinear points v_1, v_2 and v_3 and so must be identical.) Each basis function Φ is simply made up of pieces of corresponding elements containing the node associated with Φ, and each of these pieces is a linear combination (8) of the above local basis functions for its element. The previous example thus gave efficient strategies for computing all of the local basis functions as well as the corresponding basis functions.

EXERCISE FOR THE READER 13.2: (a) Explain why piecewise linear basis functions could not be used if rectangular elements (with sides parallel to the axes) were used in place of triangular ones.
(b) Give an example of a simple type of basis function that could be used for such rectangular elements. Make sure that your construction will ensure that any given basis function will be continuous across element edges.

As mentioned Section 13.1, triangulation is an art, and as such there has been a notable amount of research in the development of efficient and effective triangulation and mesh generation schemes. One particularly successful and often used method in this area is that of the **Delaunay triangulation**, relative to a given finite set of points in the plane. This triangulation will result in a set of triangles whose vertices coincide with the given finite set (of nodes) and with the further property that the circumcircle of each triangle in the collection contains only nodes that are vertices of that triangle. This condition favors well-rounded triangles over thin ones, which are better for the FEM.

Definitions: Suppose that we have a set of distinct points $P = \{p_1, p_2, \cdots, p_n\}$ in the plane $\mathbb{R}^2$. The **Delaunay triangulation** relative to P consists of all triangles connecting three noncollinear points $p_i, p_j, p_k \in P$ with the property that there

exists a point $a \in \mathbb{R}^2$ which lies equidistant to each of the points p_i, p_j, p_k and closer to these three than to any other point $p_\ell \in P,\ \ell \neq i, j, k$.

Figure 13.8 illustrates the Delaunay triangulation for a very small set of four points. It can be shown that the Delaunay triangulation of a finite set of points will always be a triangulation for the **convex hull**[4] of this set. The Delaunay triangulation has the important property that the minimum angle of any of its triangles is as large as possible for any triangulation of the same set of points (see Section 1.2 of [Ede-01]). This makes the Delaunay triangulation very suitable for the FEM. There is a dual notion of the Delaunay triangulation which leads to an equivalent formulation. We give the relevant definitions:

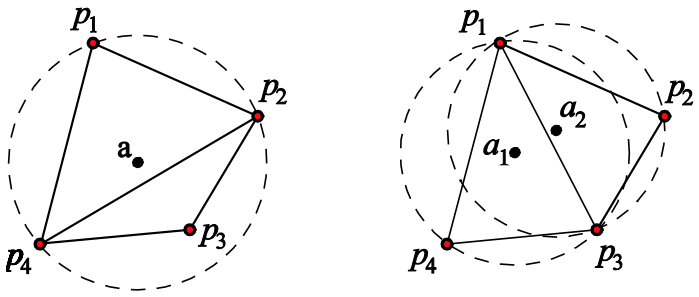


FIGURE 13.8: Two triangulations are shown for the same 4-point set $\{p_1, p_2, p_3, p_4\}$. (a) (left) The first one violates the Delaunay condition since p_3 lies in the circumcircle of the larger triangle. (b) (right) The second gives the Delaunay triangulation. Circles and centers are drawn in to show the validity of the condition.

Definitions: Suppose that we have a set of distinct points $P = \{p_1, p_2, \cdots, p_n\}$ in the plane $\mathbb{R}^2$ (write $p_i = (x_i, y_i)$). Relative to this set, for each $p_i (1 \leq i \leq n)$ we define the **Voronoi region** $V(p_i)$ as:

$$V(p_i) = \{p \in \mathbb{R}^2 : | p - p_i | \leq | p - p_\ell | \text{ for each } p_\ell \in P,\ \ell \neq i\}. \tag{9}$$

[4] The convex hull of a set of points is the smallest convex set which contains each of the points. There is a degenerate case in which some four of the points lie on a common circle (with no other points inside this circle). Here no three of the points will lie any closer to the center of the circle than the fourth. In algorithms for the Delaunay triangulation what is usually done in degenerate cases is that one of the points is slightly perturbed (moved). Since the area of a circle is zero, the probability that a fourth point will lie on the circle determined by three points is zero. In degenerate cases "the" Delaunay triangulation is not unique. The whole subject of triangulation and more general mesh generation has become quite an important discipline in itself. Good references are Chapter 13 of [Ros-00] and [Ede-01].

Here absolute values denote the Euclidean distance (this coincides with the 2-norm introduced in Chapter 7). The **Voronoi diagram** for P is simply an illustration of the totality of all of the Voronoi regions.

It is not difficult to show that each Voronoi region is a convex set which, if bounded, is a polygon (Exercise 27). In words, the Voronoi region $V(p_i)$ is simply the set of all points in the plane whose closest element of the set P is p_i. If a school district wished to minimize bussing times and costs, and if the points p_i represented locations of schools, the Voronoi region of a given school would roughly include all households whose children would be sent to that school.

The duality result states that two points $p_i, p_j \in P$ are joined by an edge in the Delaunay triangulation if and only if their Voronoi regions $V(p_i)$, $V(p_j)$ share a common edge. The Ukrainian mathematician Georges Voronoi was the first to introduce his concept in 1908 [Vor-08]. Subsequently, Russian mathematician Boris Delaunay introduced his triangulation in a 1934 paper [Del-34] that he dedicated to Voronoi. These concepts have numerous applications; details of the rich and interesting history can be found in the book [OkBoSu-92], which contains over 600 references. Construction of the Delaunay triangulation for a given set of n points in the plane has been the focus of much research. The first algorithms that were discovered worked in $O(n^4)$ time, but modern refined algorithms perform in $O(n \log n)$ time. Some survey articles of this area are: [SuDr-95] and [BeEp-92], see also Chapter 13 of [Ros-00]. We will make use of MATLAB's built-in functions that will perform both of the Delaunay triangulation and the Voronoi diagrams, so the task of triangulations will thus be reduced to the more simple problem of node deployment.

We proceed now to introduce the relevant MATLAB functions:

`tri = delaunay(x,y)` →	If `x` and `y` are vectors of the same length n giving the coordinates of n (noncollinear) points in the plane, this command will output an $n \times 3$ matrix `tri` whose rows contain the indices (rel. to the `x` and `y` vectors) of the triangles in the Delaunay triangulation.
`voronoi(x,y)` →	If `x` and `y` are as above, this command will result in a graphic of the Voronoi diagram[5] for the set of points corresponding to `x` and `y`.

Once created, the Delaunay triangulation can be used, just like any other triangulation for the FEM. To view the Delaunay triangulation, we could use the above `trimesh` command. We illustrate by having MATLAB compute both

[5] Actually, the `voronoi` command will show only the bounded Voronoi regions (i.e., those that have finite areas). There is an easy way to get MATLAB to show all of the regions; see Exercise for the Reader 13.2.

objects for the set of node values that we used for the diagram in Figure 13.5. The following commands result in the plot shown in Figure 13.9(a).

```
>> N=[1 1;5/2 1;0 0;1 0;5/2 0;7/2 0;1 -1;2.5 -1];
>> x=N(:,1); y=N(:,2);
>> tri=delaunay(x,y), trimesh(tri,x,y)
>> axis([-1 4.5 -1.5 1.5]), hold on
>> plot(x,y,'ro')
```

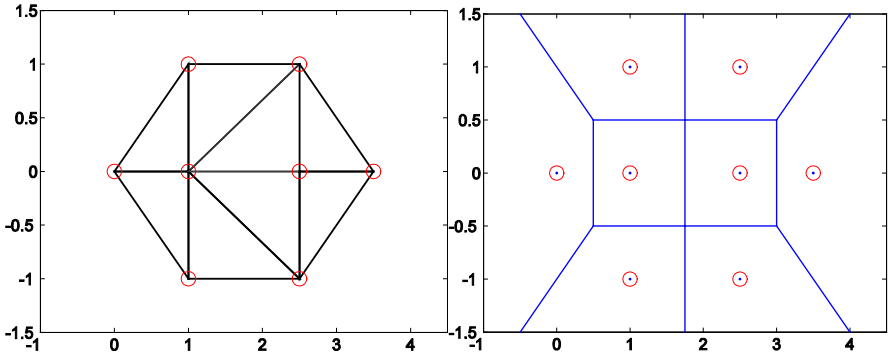


FIGURE 13.9: (a) (left) MATLAB plot of the Delaunay triangulation of the set of data points indicated by circles. (b) (right) MATLAB plot of the dual Voronoi diagram for the same set of data points.

The Voronoi diagram in Figure 13.9(b) was obtained using the minor modification of the MATLAB's `voronoi` program that appears in the following exercise for the reader.

EXERCISE FOR THE READER 13.3: (a) Write an M-file called `voronoiall(x,y)` that will function just like MATLAB's `voronoi`, except that it will show the unbounded Voronoi regions (not all of them, of course) with a reasonable axis view. (b) Use your program to recreate the plot of Figure 13.9(b).

Some comments are in order. First notice that the Delaunay triangulation that MATLAB gave us coincides with the one we used previously. Also notice that this example demonstrates that the Delaunay triangulation is not unique (so it really should not have been called "the" Delaunay triangulation). Indeed, the two diagonal edges in the center could have been reversed (i.e., reflect the triangulation horizontally) to result in another triangulation that also meets the Delaunay criteria, or the duality theorem's criterion. (The reader should convince himself or herself of these assertions.) The Voronoi regions are of course uniquely defined and so the Voronoi diagram is unique.

Having the `delaunay` function to work with makes it a lot easier to do a triangulation; we need only specify the node points. This should be done in a way that will give rise to a Delaunay triangulation whose triangles do not get too thin.

A good general rule is to deploy node points in more or less squarelike configurations. The sizes of adjacent squares should not change too abruptly. Of course, when approaching the boundary, special care must be exercised. For boundary value problems, nodes need to be put on the boundary as well. It is also possible to increase the density of nodes in certain parts of the domain in regions where coefficients of the PDE are more active (oscillatory). The next example will create three different triangulations for the same domain, a disk.

EXAMPLE 13.2: Let Ω denote the unit disk $\left\{ p = (x, y) \in \mathbb{R}^2 : \|p\|_2 \le 1 \right\}$. [6] Use MATLAB to create and plot three different triangulations of Ω each having between 1000 and 2000 nodes for each of the three requirements:
(a) The nodes are more or less uniformly distributed.
(b) The density of the nodes increases as $\|p\|_2$ increases, i.e., as we approach the boundary.
(c) The distribution of the nodes increases near the boundary point (1,0).
NOTE: We left the exact number of nodes somewhat flexible since we want to stress node deployment schemes and do not wish to be distracted with trying to use a precise number of nodes.

SOLUTION: Part (a): We will give two different strategies for this part.

Method 1: We use a squarelike configuration for the nodes. For the most part, this will be quite a simple scheme, but near the boundary circle $\|p\|_2 = 1$, things get a bit awkward. The square $S = \left\{ p = (x, y) \in \mathbb{R}^2 : -1 \le x, y \le 1 \right\}$ includes the disk Ω as its inscribed circle. Since the ratio of the areas of Ω to S is $\pi \cdot 1^2 / (2 \cdot 2) = \pi / 4 = 0.785...$, it follows that if we uniformly distribute a large number of nodes in the interior of S, roughly 78.5% of them will be in the interior of Ω. Since it is a simple matter to uniformly distribute any (square) number of nodes in S, we will begin by uniformly distributing about 2000 nodes in S, and let δ denote the square side length that is used. Of these nodes we will keep all of them that lie inside of Ω but at a distance of at least $\delta / 2$ from the boundary circle $\|p\|_2 = 1$. Then we add on a set of nodes on the boundary circle, which are uniformly spaced with gaps about equal to δ. The total amount of nodes thus constructed for Ω will be (well) over 1000 and certainly less than 2000.

To begin, since $\sqrt{2000} = 44.721...$
$N_0 = 45^2 = 2025$ interior nodes in S. The horizontal and vertical gaps should be

[6] We are using here the norm notation from Chapter 7: $\|p\|_2 = \|(x, y)\|_2 \equiv \sqrt{x^2 + y^2}$ is the 2-norm which is simply the (planar) Euclidean distance from $p = (x, y)$ to the origin $(0,0)$.

$\delta = 2/(45+1)$. The following MATLAB commands will create these nodes, and store them in two vectors `x0`, `y0`.

```
>> delta =2/46; counter=1;
for i=1:45
for j=1:45
  x0(counter)=-1+i*delta; y0(counter)=-1+j*delta;
  counter=counter+1;
end
end
```

Next, from these two vectors we extract those components corresponding to points which lie within the slightly smaller circle $\|p\|_2 = 1-\delta/2$; the newly created vectors will be labeled as `x` and `y`.

```
>> counter=1;
>> for i=1:2025
    if norm([x0(i) y0(i)],2) < 1-delta/2
      x(counter)=x0(i); y(counter)=y0(i);
      counter=counter+1;
    end
end
```

Finally, since $2\pi/\delta = 144.5133...$, we tack onto the existing vectors `x` and `y` an additional 145 entries corresponding to 145 equally spaced points on the unit circle $\|p\|_2 = 1$. Figure 13.10(a) shows a plot of this node set.

```
>> for i=1:145
    x(i+1597)=cos(2*pi/145*i); y(i+1597)=sin(2*pi/145*i);
end
>> plot(x,y,'bo'), axis('equal')
```

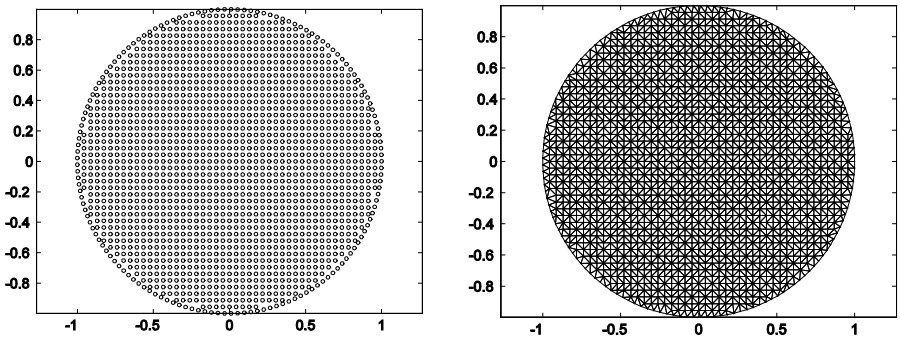


FIGURE 13.10: (a) (left) Grid of nodes from Method 1 of Example 13.2(a), essentially a square pattern except near the boundary. There are 1742 nodes. (b) (right) A corresponding Delaunay triangulation that has 3337 triangles.

The corresponding Delaunay triangulation will result from the following MATLAB commands and is shown in Figure 13.10b.

```
>> tri=delaunay(x,y); trimesh(tri,x,y), axis('equal')
```

Method 2: Here we will deploy nodes on circles of increasing radii. The gaps between nodes on a given circle and the gaps between radii of adjacent circles of deployment should be all about equal (uniformity). The final circle will be the boundary of $\Omega: \|p\|_2 = 1$. The only mathematical preliminaries are to decide how many circles to deploy. Letting δ denote the common gap size, since the radii of the circles increase steadily from 0 to 1, the average radius will be about 1/2, which means the average circumference will be about $2\pi(1/2) = \pi$. Thus the average number of nodes on a circle will be π/δ. Likewise, the number of circles of deployment is about $1/\delta$, so that the total number of nodes will be, roughly, $(\pi/\delta)\cdot(1/\delta) = \pi/\delta^2$. Setting this equal to 1800, say (we want it close to 2000, but to insure the actual number of nodes remains under 2000 we play it a bit safe), and solving for delta gives $\delta = \sqrt{\pi/1800} = 0.04177...$. We may now turn the node deployment over to MATLAB with this scheme:

```
>> delta=sqrt(pi/1800); x(1)=0;  y(1)=0;
>> nodecount=1; ncirc=floor(1/delta); minrad=1/ncirc;
>> for i=1:ncirc
    rad=i*minrad;
    nnodes=floor(2*pi*rad/delta);
    anglegap=2*pi/nnodes;
    for k=1:nnodes
      x(nodecount+1)=rad*cos(k*anglegap);
      y(nodecount+1)=rad*sin(k*anglegap);
      nodecount = nodecount+1;
    end
end
```

The plotting of the nodes and then the Delaunay triangulation is done just as in Method 1 above; the results are shown in Figure 13.11.

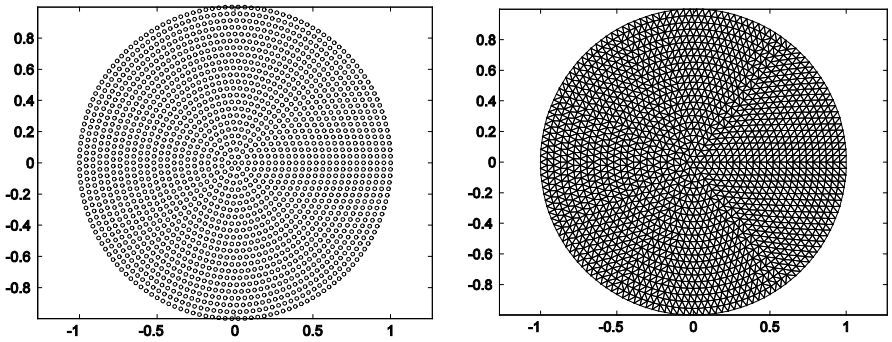


FIGURE 13.11: (a) (left) Grid of nodes from Method 2 of Example 13.2(a). There are 1887 nodes. (b) (right) A corresponding Delaunay triangulation that has 3438 triangles. Both the node distribution as well as the Delaunay triangulation take on an aesthetically more appealing pattern than those of Method 1, since this method better respected the symmetry of the domain.

Part (b): The requirement is rather vague. We will use a deployment scheme similar to that of Method 2 in part (a). The new difficulty is that there will need to

be more circles of nodes of larger radii so it will take more work to estimate the total number of nodes. Such an estimate will depend first on how we plan to distribute the radii for the circles of nodes. Here is (but) one scheme. We start off with a single node at the origin (0,0). Then we move to the circle $\|p\|_2 = 1/2 = \text{rad}(1) = 1 - 1/2$ and deploy 8 (equally spaced) nodes on this circle. Our next circle is $\|p\|_2 = 3/4 = \text{rad}(2) = 1 - 1/4$ on which we deploy $2 \cdot 8 = 16$ nodes. After this we deploy $2 \cdot 16 = 32$ nodes on the circle. We continue this pattern, so that at the nth circle of deployment will be $\|p\|_2 = \text{rad}(n) = 1 - 1/2^n$ on which we will deploy 2^{n+2} nodes. This will continue until the number of remaining nodes is still greater than the number of most recently installed nodes (on the last circle of deployment). The final step will be to put all of the remaining nodes on the unit circle $\|p\|_2 = 1$. This plan will create exactly 2000 nodes. Here now is the MATLAB code needed to create such a set of nodes.

```
>> xb(1)=0; yb(1)=0; rnodes=1999; %remaining nodes
>> newnodes=8; %nodes to be added on next circle
>> radcount=1; %counter for circles
>> oldnodes=1; %number of nodes already deployed
>> while newnodes < rnodes/2
  rad = 1 - 2^(-radcount);
  for i=1:newnodes
    xb(oldnodes+i)=rad*cos(2*pi*i/newnodes);
    yb(oldnodes+i)=rad*sin(2*pi*i/newnodes);
  end
  oldnodes=oldnodes + newnodes; %update oldnodes
  rnodes = rnodes - newnodes; %update rnodes
  radcount=radcount+1; %update radcount
  newnodes = 2*newnodes; %update newnodes
end  % next deploy remaining nodes on boundary
>> for i=1:rnodes
xb(oldnodes+i)=cos(2*pi*i/rnodes); yb(oldnodes+i)=sin(2*pi*i/rnodes);
end
```

The plotting of the nodes and creation of the Delaunay triangulation is obtained in the same fashion as in part (a). The results are shown in Figure 13.12.

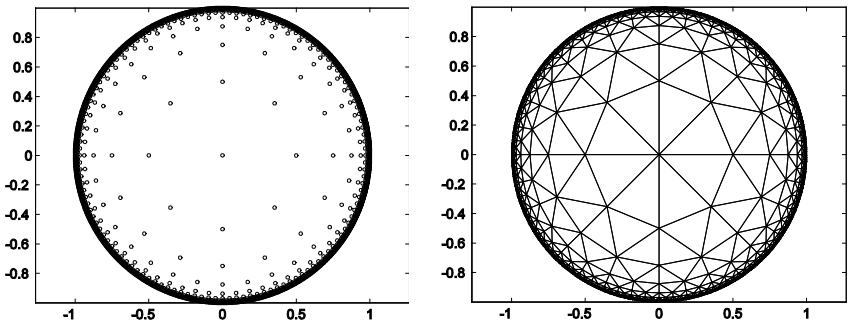


FIGURE 13.12: (a) (left) Grid of nodes for Example 13.2(b). There are 2000 nodes. (b) (right) A corresponding Delaunay triangulation that has 3015 triangles. Such a triangulation is useful for BVPs which are particularly sensitive to boundary data.

Part (c): The way we will deploy nodes is to first partition Ω into subsets determined by the regions between pairs of circles centered at (1,0). For each positive integer *n*, we define the following subset $\Omega_n \subseteq \Omega$:

$$\Omega_n = \{(x,y) \in \Omega : 1/2^n < \text{dist}((x,y),(1,0)) < 2 \cdot (1/2^n)\}.$$

In words, Ω_n is just the portion of points inside Ω that lie in between the two concentric circles with center (1,0), and radii $1/2^n$ and $2 \cdot (1/2^n)$. A typical such region is illustrated in Figure 13.13. The idea will be to deploy roughly a fixed number of nodes (we will use about 100) in each of these regions, up to a certain value of the index. We will also need to put some nodes in the remaining part of Ω (which actually can be written as Ω_0); here we again put roughly the same number, about 100 nodes. To decide how to put the nodes on the boundary circle of Ω as well as in the interior of the domains Ω_n, we use the following estimates. From Figure 13.13, it is clear that Ω_n is always contained in the half annulus (enclosed between the two dotted circles in the figure), and consequently,

$$\text{Area}(\Omega_n) \le \tfrac{1}{2}[\pi \cdot (2 \cdot 2^{-n})^2 - \pi \cdot (2^{-n})^2] = \tfrac{3\pi}{2} 2^{-2n}.$$

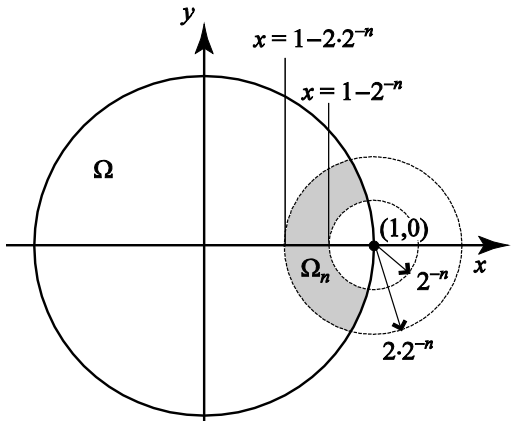


FIGURE 13.13: Illustration of a typical region Ω_n (shaded) for the triangulation scheme of part (c) of Example 13.2. Such regions are useful for general triangulation schemes when it is desired to have large finer meshes near a special point of the domain.

Also, the estimate becomes increasingly accurate as *n* gets larger. Now, if we deploy a square grid of nodes with (horizontal = vertical) grid spacing = *s* in the interior Ω_n, each node would give rise to a square of area s^2 inside of Ω_n (to be specific, let's say that the node gets associated with the square of side length *s* having the node as its lower left vertex). Thus, if we were to put a grid of 100 such nodes in the interior of Ω_n, the area bound above would yield the following estimate for *s*:

$$100 \cdot s^2 \le \text{Area}(\Omega_n) \le \tfrac{3\pi}{2} 2^{-2n} \Rightarrow s \le \sqrt{3\pi}\, 2^{-n} / (10\sqrt{2}).$$

We use this as a scheme for the horizontal/vertical grid gap to put between nodes that lie inside each Ω_n. The actual number of nodes on each deployment will be less than 100 because the above estimates are inequalities. Since we will essentially be placing 100 nodes at each iteration on Ω_n (and the corresponding portion of the boundary circle adjacent to Ω_n) starting with $n = 0$, it follows that we should let n run up to about 15 in this scheme. For deploying nodes on the boundary circle that lie adjacent to Ω_n, we also use s as the gap (this time the circular arclength gap) between boundary nodes. Since the boundary circle has radius one, angles are equal to the corresponding boundary arclengths. We will create the nodes using nested loops. On each iteration for n (master loop), the loops will first create and store the corresponding value of s, determined by using an equality in the above inequality for s.

Next, a double loop will be set up that will run through a horizontal and vertical grid that will cover the domain Ω_n and have (horizontal = vertical) grid gap = s. For this part, note (again from Figure 13.13) that the domain Ω_n is always contained within the rectangle: $R_n = \{(x, y) : 1 - 2 \cdot 2^{-n} \le x \le 1, -2 \cdot 2^{-n} \le y \le 2 \cdot 2^{-n}\}$ Grid points lying in the interior of Ω_n are added as nodes. Once this double nested loop has been executed and interior nodes have been added, the same master loop will then move on to install nodes on the two portions of the boundary of Ω_n that lie on the unit circle (= boundary of the main domain Ω). We will need to compute the angles (made from (0,0) to the positive x-axis) of the two endpoints of the top boundary arc of Ω_n on the unit circle. (The node deployment on the bottom symmetric boundary arc can be gotten by simply negating the y-coordinates of the nodes in the upper arc.) It is easily shown using the law of cosines that these two angles θ_1 and θ_2 (which technically should be denoted by $\theta_{1,n}$ and $\theta_{2,n}$ to indicate their dependence on Ω_n) satisfy: $\cos(\theta_1) = 1 - 2/2^{2n}$ and $\cos(\theta_2) = 1 - 2^{-2n}/2$. The following MATLAB code is an implementation of the scheme described above.

```
>> n=0; nodecount=1;
>> while n<16
  s=sqrt(3*pi/2)/10/2^n; hgrid=2/s/2^n; vgrid=4/s/2^n;
%these will be sufficient horizontal and vertical grid counts to
%create a rectangular grid (with gap size =s) that will cover the
%domain Omega_n
  for i=1:hgrid
  for j=1:vgrid
   xnew=1-2/2^n+i*s; ynew=-2/2^n+j*s;
   pij = [xnew ynew]; p=[1 0];
 if norm(pij,2)<1-s/2 & norm(pij-p,2)<2/2^n & norm(pij-p,2)>1/2^n+s/2
%The three conditions here check to see if the node should be added.
%The first says that the node should be in the unit circle (with a
%safe distance to the boundary to prevent interior nodes from getting
%too close to boundary nodes which will be added later).  The second
```

```
%and third state that the distance from the node to the special
%boundary point (1,0) should be between the two required radii.  The
%last condition has a safety term added to the lower bound to prevent
%nodes from successive iterations from getting too close.
   x(nodecount)=xnew; y(nodecount)=ynew; nodecount=nodecount+1;
  end; end; end
%The next part of the loop puts nodes on the boundary.
 theta1=acos(1-2/2^(2*n)); theta2=acos(1-2^(-2*n)/2);
 if n==0, theta1=theta1-s; end
 for theta = theta1:-s:(theta2+s/2)
  x(nodecount)=cos(theta); y(nodecount)=sin(theta);
  x(nodecount+1)=cos(theta); y(nodecount+1)=-sin(theta);
  nodecount=nodecount+2;
 end
 n=n+1;
end
%We need to put a node at the special unsymmetric point (-1,0).
x(nodecount)=-1; y(nodecount)=0; nodecount=nodecount+1;
%Finally we put nodes in the portion of the domain between (1,0)
%and the last Omega_n, and then on the boundary.
%We need first to bump n back down one unit.
n=n-1;
for i=1:hgrid
for j=1:vgrid
  xnew=1-2/2^n+i*s; ynew=-2/2^n+j*s;
  pij = [xnew ynew]; p=[1 0];
  if norm(pij,2)<1-s/2 & norm(pij-p,2)< 1/2^n
   x(nodecount)=xnew; y(nodecount)=ynew; nodecount=nodecount+1;
  end; end; end
for theta = -theta2:s:theta2
  x(nodecount)=cos(theta); y(nodecount)=sin(theta);
  nodecount=nodecount+1;
end
```

Plotting of the nodes as well as the corresponding triangulation is accomplished exactly as it was done in the above two parts. The results are shown in Figures 13.14 and 13.15.

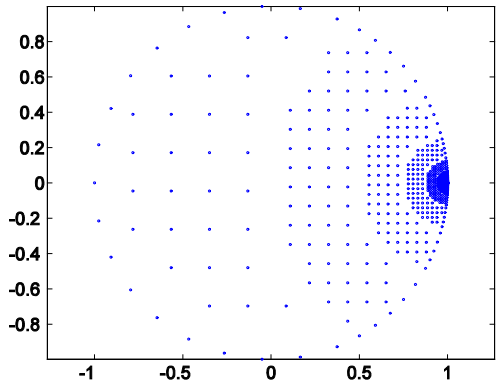


FIGURE 13.14: Node distribution from the solution of part (c) of Example 13.2. The 1457 nodes are constructed in clusters with each cluster getting its grid gap size cut in half as we move in towards the special boundary point (1,0). The exercises will examine some related schemes for this domain where there is a smoother transition in gap sizes of nodes as we progress toward (0,1).

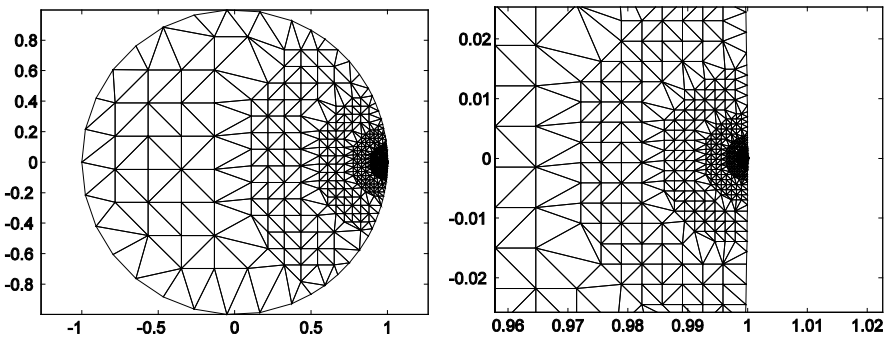


FIGURE 13.15: (a) (left) The Delaunay triangulation corresponding to the network of nodes of Figure 13.14 that has 2733 triangles. (b) (right) A $20\times$ magnification of the triangulation of (a) near the point of focus (1,0).

On the node sets that were constructed in the last example, the Delaunay triangulation worked quite nicely because the domain Ω was convex. In general, the Delaunay triangulation of a set of nodes will triangulate the convex hull of this set. Delaunay triangulation can still be used to triangulate a nonconvex domain Ω. This is usually done either by breaking up the domain into convex pieces, triangulating each piece, and merging these triangulations or simply by triangulating the convex hull and deleting triangles that are not part of the domain.[7] With either strategy, some sort of (global) reindexing will be necessary when constructing the final triangulation. Our next example will illustrate the latter strategy and the exercise for the reader which follows will require also the former strategy.

EXAMPLE 13.3: Let Ω denote the annular domain $\left\{p=(x,y)\in\mathbb{R}^2 : 1\le\|p\|_2 \le 2\right\}$. Use MATLAB to create and plot a triangulation of Ω having between 200 and 400 nodes that are more or less uniformly distributed.

SOLUTION: We use a node deployment strategy that is based on that of Method 2 of part (a) of the last example, distributing nodes on concentric circles starting at $\|p\|_2 = 1$ $\|p\|_2 = 2.$ δ denote the (approximate) common gap size between nodes (and the circles of node deployment), the average radius will be (roughly) 3/2, so that the average circumference will be $2\pi(3/2) = 3\pi.$ The average number of nodes on a circle of deployment will thus be $3\pi/\delta$ and the number of such circles will be (roughly) $1/\delta.$ This gives the

[7] Of course, the convex hull of a set of nodes for a domain will not always coincide with the domain even when the domain is convex (e.g., a disk) just as the triangulation will not coincide with the domain. But if the mesh is finer these approximations will become indistinguishable from the true objects.

following approximation for the total number of nodes: $(3\pi/\delta)\cdot(1/\delta) = 3\pi/\delta^2$. Setting this equal to 350 and solving for delta gives us a good value to use: $\delta = 0.164097....$

```
>> delta=sqrt(3*pi/350);
nodecount=1; ncirc=floor(1/delta); radgap=1/ncirc;

for i=0:ncirc
  rad=1+i*radgap; nnodes=floor(2*pi*rad/delta); anglegap=2*pi/nnodes;
  for k=1:nnodes
  x(nodecount)=rad*cos(k*anglegap); y(nodecount)=rad*sin(k*anglegap);
   nodecount = nodecount+1;
  end
end
>> tri=delaunay(x,y); trimesh(tri,x,y), axis('equal')
>> size(x)
→ans = 1   399
```

We are just under the desired upper bound on the number of nodes (if we had gone over, we could just increase δ a bit and try again. The resulting plot of the triangulation is shown in Figure 13.16(a).

Locating and deleting the unwanted triangles in Figure 13.16(a) would be an arduous task. The problem could be greatly simplified if we were to throw in an extra "ghost node" at (0,0). This is done by simply entering

```
>> x(400)=0; y(400)=0;
>> tri=delaunay(x,y); trimesh(tri,x,y), axis('equal')
```

The resulting triangulation shown in Figure 13.16(b) is much easier to work with. The triangles that need to get deleted are simply those that have (0,0) (node #400) as one of their vertices. So we simply delete from the triangulation matrix all rows that contain the entry 400. It is very simple to tell MATLAB how to do this, after checking there are 722 triangles (elements). We will make use of the following "set difference" command:

`c = setdiff(a,b)` →	If `a` and `b` are vectors, the output of this "set difference" command will be another vector `c` whose elements consist of the different values of `a` that do not occur in `b`.

Here is a simple example:

```
>> setdiff([3 1 2 3], [2]) →ans = 1    3
```

Now back to our problem; the following series of commands will produce the final triangulation of Figure 13.16(c).

```
>> badelcount=1;
for ell=1:722
  if ismember(400,tri(ell,:))
   badel(badelcount)=ell;
   badelcount=badelcount+1;
  end; end
```

```
>> tri=tri(setdiff(1:722,badel),:);
>> x=x(1:399); y=y(1:399);
>> trimesh(tri,x,y), axis('equal')
```

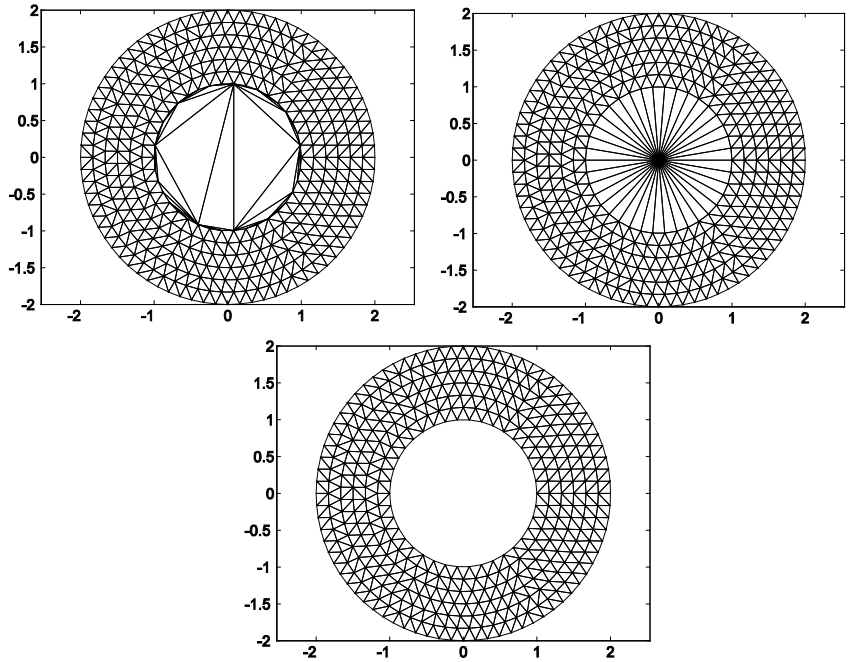


FIGURE 13.16: (a) (upper left) The Delaunay triangulation obtained from a set of nodes in the annular domain $\left\{p=(x,y)\in\mathbb{R}^2 : 1\le \|p\|_2 \le 2\right\}$ of Example 13.3. (b) (upper right) The Delaunay triangulation for the same set with one additional "ghost node" added at (0,0). (c) Resulting triangulation for the annulus.

The schemes introduced in the previous examples can be combined in various fashions to give a decent collection of strategies for triangulation of planar domains that will be sufficient for our purposes. The topic of mesh generation has been receiving a great deal of attention beginning in the 1990s. We will see later in this chapter that boundary value problems on domains with obtuse ($>\pi$) interior angles (see Figure 13.17 for two examples of such domains) usually require special attention with the numerical methods at corresponding boundary points. The next exercise for the reader asks the reader to construct suitable triangulations for such domains.

EXERCISE FOR THE READER 13.4: Using a scheme similar to that of the solution of part (c) of Example 13.2, get MATLAB to create and plot triangulations each having between 500 and 1000 nodes for the two domains illustrated in Figure 13.17(a), (b), respectively. In each, arrange it so that the distribution of nodes increases near the special boundary point p

illustration. For the domain of part (a), take the exterior angle to be 60°; for the one of part (b), make up your own coordinates and dimensions.

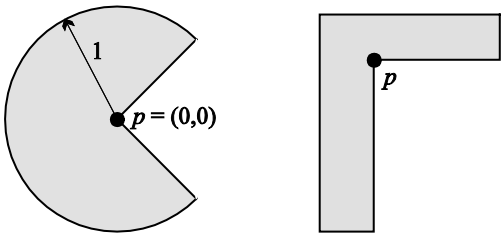
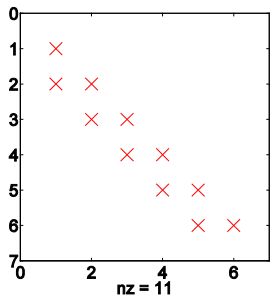


FIGURE 13.17: (a), (b) A pair of domains possessing a boundary point p with an obtuse interior angle. Such boundary points usually require extra care when boundary value problems on them are solved numerically.

In each of the triangulations done above, we tried to create them to meet most of the desired properties that were mentioned at the beginning of the chapter. There is one notable exception, however, that we did not even contemplate in our constructions. Namely, we made no efforts to arrange that the node numbers for any given triangle in the triangulations were reasonably close. (The constructions were already quite complicated without this and the Delaunay triangulation program left parts of the construction out of our control). The reason why this is a desirable property to have is that the resulting stiffness matrix will be banded (and hence sparse and easier to deal with). To get a rough idea of the relative numbering of the nodes, recall that MATLAB's function `spy`, introduced in Chapter 11, can gives us a "graph" of the nonzero entries of a matrix. For convenience, we give here a quick example reviewing its syntax.

```
>> d=ones(1,6); b=2*d(1:5);
>> A=diag(d)+diag(b,-1)

A =     1   0   0   0   0   0
        2   1   0   0   0   0
        0   2   1   0   0   0
        0   0   2   1   0   0
        0   0   0   2   1   0
        0   0   0   0   2   1
```




```
>> spy(A, 'rx') %mark nonzero entries with red
>> %              x's; or use spy(A)
```

FIGURE 13.18: A simple spy plot of a banded 6×6 matrix. The locations of nonzero entries are indicated by x's. The total number of nonzero entries (nz = 11) is indicated below the graph. The spy command is a useful tool for obtaining a quick understanding of the structure of a matrix, and, in particular, allows for quick detection of sparse and banded matrices.

The triangulation that we created in the solution of Example 13.2(c) had 1457 nodes. The corresponding stiffness matrix A would thus be 1457×1457. As in the one-dimensional FEM, we will see in the next section that the a_{ij} entry (corresponding to node numbers #*i* and #*j*) of the stiffness matrix will be given by a certain integral involving products of (gradients of) the corresponding basis functions Φ_i and Φ_j. Throughout the text of this chapter, we will be restricting our attention to piecewise linear basis functions, and for such a basis function, say Φ_i, it will be zero except on those elements that have node #*i* as one of their vertices. It follows that a_{ij} will be zero unless nodes #*i* and #*j* are both vertices of the same triangle. In the following example, we will use this fact to find out all possible nonzero entries of the stiffness matrix, draw a spy diagram, and list the total number of possible nonzero entries. The way we will form this matrix is to simply put a positive integer at all entries that are possibly nonzero.

EXAMPLE 13.4: Let A denote the 1457×1457 stiffness matrix for the triangulation obtained in Example 13.2(c) and with piecewise linear basis functions. Using the information above, construct a matrix M that will have positive integer entries where the corresponding entries of the stiffness matrix are zero, and zero entries where the stiffness matrix has zero entries.

SOLUTION: The way we will construct M will be similar to the so-called "assembly" method that we will use in the next section to build stiffness matrices. The construction will proceed element by element. More precisely, we begin with M being a 1457×1457 matrix of zeros. We then run down the list of triangles/elements (all 2733 of them), and for each one we change the corresponding entry of the of the matrix M to equal 1 (these are the only entries of that stiffness matrix A that could be nonzero). If an element is represented by the three vertices: [i j k], the entries we will bump up by one for this element would be the following nine entries $a_{\alpha\beta}$ where α and β run through i, j, and k. This is a much more efficient scheme rather than constructing the nonzero elements directly (in which case for each one a search would need to be done over all elements to see if the corresponding pair of nodes share a common element).

Assuming the matrix `tri` obtained in the last example (for the Delaunay triangulation of the nodes in part (c)), the following commands will "assemble" a suitable matrix M, and then create a `spy` diagram of it (and hence also of the stiffness matrix). The spy diagram is shown in Figure 13.19.

```
>> M=zeros(1457); for c=1:2733
  E=tri(c,:);
  for i=1:3
  for j=1:3
   M(E(i),E(j))=M(E(i),E(j))+1;
  end
  end
end
>> spy(M,'b+') %or use spy(M) to use default '.' markers
```

```
>> 9835/1457^2  →ans = 0.0046
```

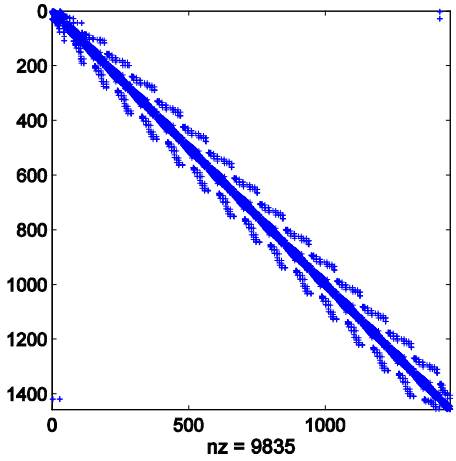


FIGURE 13.19: A spy diagram of the stiffness matrix for Example 13.4. The possible nonzero entries account for only 0.46% of all of the entries, so this stiffness matrix is sparse as stiffness matrices usually are. The fuzzy patterns (top and bottom) correspond to the boundary nodes being added after the interior nodes. The number of such patterns is the number of master iterations in the node construction.

The last ratio is simply that of the nonzero entries to the total entries of M. Thus at most 0.46% of the entries in the stiffness matrix can be nonzero, and the matrix is indeed sparse.

Can the reader explain the isolated four markers in the upper-right and lower-left of Figure 13.19?

EXERCISES 13.2

1. (a) For the hexagonal domain of Figure 13.5 with node coordinates as given by the matrix N following Figure 13.6, deploy between 1000 and 2000 nodes more or less uniformly throughout the domain and boundary and plot the nodal configuration. You should of course include the boundary nodes of Figure 13.5 but not necessarily the interior nodes (#4, #5).
   (b) Create a corresponding Delaunay triangulation and plot.

2. Repeat parts (a) and (b) of Exercise 1, but this time let the nodes increase in density as one moves toward the boundary.

3. Repeat parts (a) and (b) of Exercise 1, but this time let the nodes increase in density as one approaches the exterior node #6.

4. Letting Ω denote the unit disk $\left\{p=(x,y)\in\mathbb{R}^2 : \|p\|_2 \le 1\right\}$ of Example 13.2, use MATLAB to create and plot a triangulation of Ω having between 1000 and 2000 nodes for which the nodes increase in density near the segment $-\pi/4 \le \theta \le \pi/4$ on the boundary, that is, near the (smaller) circular arc connecting the points $(\sqrt{2}/2, \pm\sqrt{2}/2)$ on the boundary. Let the node distribution

elsewhere in the disk be more or less uniform.
Suggestion: Use the solution to part (c) of Example 13.2 for some relevant ideas. First deploy some nodes in a circle centered at (0,0), say $\|p\|_2 \le 0.5$, then use a loop to deploy nodes in the annuli $A_n = \{p : 1-2^n \le \|p\|_2 \le 1-2^{n+1}\}$, n = 1, 2, …. As an indicator of closeness to circular arc (for points (x,y)) in A_n use the inequality $\tan(y/x) \le 1 + 2 \cdot 2^n$.

5. Repeat Exercise 4 with the modification that node distribution should increase near the boundary. For the portion of the boundary complementary to $-\pi/4 \le \theta \le \pi/4$, make the rate of increase in node density to be roughly 10% compared to the rate of increase near the special portion.

6. (a) Write an M-file, call it `[x y tri]=circtri(angle1, angle2, maxnodes)` that will do the following. The input variables `angle1` and `angle2` denote two angles on the unit circle such that $0 \le \text{angle2} - \text{angle1} < 2\pi$. The M-file will create a set of nodes, stored in the output variables `x` and `y` for the triangulation of the unit disk $\{p = (x,y) \in \mathbb{R}^2 : \|p\|_2 \le 1\}$ in a way analogous to the one explained in Exercise 4 (for the special case `angle1` = $-\pi/4$ and `angle2` = $\pi/4$) but with the total number of nodes deployed being between the input variable `maxnodes` and half of this variable. Thus `maxnodes` should be a positive integer, at least equal to, say, 20. The final output variable `tri` will be a three-column matrix corresponding to the Delaunay triangulation of the node set. Note that the syntax includes the possibility that angle1 = angle2, in which case a triangulation similar to that done in Example 13.2(c) is required.
(b) Use your program to redo Exercise 4.
(c) Run your program, and plot the nodes and resulting triangulations for each of the following sets of input variables:
(i) angle1 = $\pi/2$, angle2 = $5\pi/6$, maxnodes = 500
(ii) angle1 = $-\pi$, angle2 = $\pi/2$, maxnodes = 1500
(iii) angle1 = $7\pi/6$, angle2 = $11\pi/6$, maxnodes = 1200

7. (a) Write an M-file, call it `[x y tri]=circtri2(angle1, angle2, maxnodes, r)`, that has the same syntax as that explained in Exercise 6, except that there is an additional input variable `r` which is to be a positive number less than 1 and the triangulation will be performed as explained in Exercise 5 (for the special case angle1 = $-\pi/4$ and angle2 = $\pi/4$, and r = 0.1). The parameter `r` will denote the relative density that nodes are increasing as we near the complementary arc compared to when we near the arc angle1 < θ < angle2.
(b) Use your program to redo Exercise 5.
(c) Run your program, and plot the nodes and resulting triangulations for each of the following sets of input variables:
(i) angle1 = $\pi/2$, angle2 = $5\pi/6$, maxnodes = 500, r = 0.25
(ii) angle1 = $-\pi$, angle2 = $\pi/2$, maxnodes = 1500, r = 0.05
(iii) angle1 = $7\pi/6$, angle2 = $11\pi/6$, maxnodes = 1200, r = 0.025.

8. (*Triangulating General Convex Polygons*) (a) Write an M-file, call it `[x y tri]=unipolytri(xv, yv, maxnodes)`, that will do the following. The input variables `xv` and `yv` denote the vectors corresponding to the x- and y-coordinates of vertices of a convex polygon which are assumed to be ordered in counterclockwise fashion around the boundary. The first vertex should also be the last vertex (to close the polygon). The last variable `maxnodes` denotes a positive integer, say, at least 20. The program will create a set of nodes for a triangulation of the polygon and its boundary that will be stored in the output variables x and y. The nodes are to be configured in a square pattern (cf., Method 1 of the solution to Example 13.2(a)) throughout the polygon and its boundary. The number of nodes deployed should be somewhere between `maxnodes` and half of this number. The third output

variable `tri` denotes a 3-column matrix corresponding to the Delaunay triangulation for the node set which is constructed.
(b) Use your program to redo Exercise 1.
(c) Run your program, and plot the nodes and resulting triangulations to obtain triangulations for each of the following convex polygons using between 200 and 400 nodes for each.
(i) The rectangle with vertices $(\pm 1, \pm 10)$.
(ii) The triangle with vertices (0,0), (1,0), (0,8).
(iii) A regular octagon unit sidelength.
(iv) The septagon with vertices (0,0), (2,0), (16,1), (16,4), (13,5),(11,4), (1,3).
Suggestion: One way to view a convex polygon is that its set of points can be described as the intersection of all points in the plane which simultaneously lie on the correct side of each of its edges. Each such edge requirement can be written mathematically in the form $ax + by \le c$. Set up a grid of about maxnodes nodes in the rectangle $R = \{(x, y) : \min(xv) \le x \le \min(xv), \min(yv) \le y \le \min(yv)\}$, then use each of the edge requirements (put in form $ax + by < c$ to save boundary points for later) to decide with a loop which of these points should be interior points. Finally put nodes on the boundary with appropriate density. Make sure that there are no interior nodes that are too close to the boundary. That the number of nodes put in the polygon will satisfy the required bounds follows from the fact that the area of the polygon is at least half the area of the rectangle *R* (why?).

NOTE: (*Triangulating General Polygons*) Since any polygon can be decomposed into convex pieces, the program `unipolytri` of Exercise 8 can be used to essentially uniformly triangulate general polygons. For example, the polygon of Figure 13.17(b) is not convex but can be written as a union of two (convex) rectangles R_1 and R_2, that have corresponding areas A_1 and A_2. (There are a couple of ways to do this.) Suppose we wish to triangulate the region using somewhere between 500 and 1000 nodes. We could run the program `unipolytri` on R_1 using maxnodes to be about $1000A_1/(A_1 + A_2)$ and then on R_2 using maxnodes to be about $1000A_2/(A_1 + A_2)$. The ratios attempt at allocating an appropriate number of nodes to each piece. We can pretty much juxtapose these two node sets to arrive at a node set for the original polygon (after reindexing and deleting some nodes at the common interior interface boundary). This idea, and its extensions to greater numbers of convex pieces, is explored in the following three exercises. In particular, these exercises require the reader to have completed Exercise 8(a).

9. Use the idea of the above note to redo Exercise for the Reader 13.4(b).

10. Use the idea of the above note to triangulate, using between 400 and 800 nodes, the decagon that has the following vertices: $(\pm 2, 0)$, $(\pm 1, \pm 1)$, $(\pm 2, \pm 2)$. Plot both the node diagram as well as the triangulation.

11. Consider the symmetric (nonconvex) polygon consisting of the rectangle with vertices: $(\pm 1, -1)$, $(\pm 1, 0)$ with two (left and right) triangles with vertices: $(\pm 1, 1)$, $(\pm 1, 0)$, $(0, 0)$ joined on top.
(a) Apply the method in the above note to triangulate this region by splitting it into the left and right halves (which are each convex). Display the node configuration and corresponding triangulation.
(b) Apply the method in the above note to triangulate this region by splitting it into the following three convex pieces: the bottom rectangle, and the two triangles. Display the node configuration and corresponding triangulation. Does the density appear uniform? If not, explain, and adjust the ratios of node densities to correct the problem.

The next three exercises will involve triangulations of the domains having domains with curved boundaries illustrated in Figure 13.20.

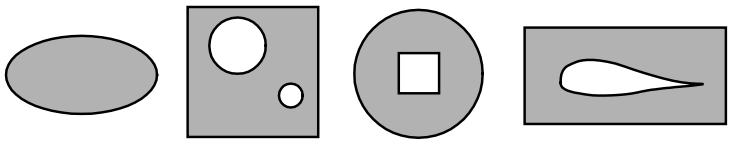


FIGURE 13.20: Four domains with curved boundary portions: (from left) (a) An ellipse, (b) a square with two circular holes, (c) a disk with a square hole, (d) an airfoil removed from a rectangle.

12. Let the elliptical region Ω of Figure 13.20(a) have equation (for its boundary): $x^2 + 4y^2 = 4$. Use MATLAB to create and plot triangulations of Ω having between 400 and 800 nodes and with the following additional properties:
(a) The nodes are more or less uniformly distributed with essentially a square grid (as in Method 1 of part (a) in the solution of Example 13.2).
(b) The nodes are deployed on concentric ellipses of the same eccentricity as the boundary ellipse (cf. Method 2 of part (a) in the solution of Example 13.2).
(c) The nodes are deployed in concentric ellipses (as in part (b)) but the density increases as we near the boundary (cf. part (b) in the solution of Example 13.2).
(d) The density of the nodes increases as we approach the interior point $(x, y) = (1, 0)$ and such that between 20 and 30 nodes are deployed on the boundary (cf. Method 2 of part (a) in the solution of Example 13.2).

13. Let the region Ω of Figure 13.20(b) be specified as follows: The square (outside) boundary has equations: $x = 0, 2$, $y = 0, 2$ and the removed circles have the following centers and radii: upper left circle: center = (0.5, 1.5), radius = 0.25; lower right circle: center = (1.5, 0.5), radius = 0.1. Use MATLAB to create and plot triangulations of Ω having between 400 and 800 nodes and with the following additional properties:
(a) The nodes are, more or less, uniformly distributed with essentially a square grid (cf. Method 1 of part (a) in the solution of Example 13.2).
(b) The density of the nodes increases as we near each of the two interior circle boundary portions and such that the square boundary has between 20 and 30 nodes.

14. Let the region Ω of Figure 13.20(c) be specified as follows: The outside circle has: center = (0, 0) and radius = 2; the inside square has equations: $x = \pm 1, y = \pm 1$. Use MATLAB to create and plot triangulations of Ω having between 400 and 800 nodes and with the following additional properties:
(a) The nodes are more or less uniformly distributed with essentially a square grid (cf. Method 1 of part (a) in the solution of Example 13.2).
(b) The nodes are deployed on concentric circles (to the outer boundary circle) and more or less uniformly distributed.
(c) The density of the nodes increases as we near any of the four corner points on the inside square boundary and the outside circle will have between 20 and 30 nodes.

15. Let Ω denote the region of Figure 13.20(d). (a) Use MATLAB to plot an airfoil (the inside boundary of Ω) by setting up a set of points on the boundary of the foil. Then enter (as different vectors) the four vertices of an appropriate rectangle for the outer boundary of Ω. Your foil does not have to be identical with the one in the figure, but should more or less resemble it. We are more concerned here with triangulations rather than aerodynamics. Use MATLAB to create and plot triangulations of Ω having between 400 and 800 nodes and with the following additional properties:
(b) The nodes are more or less uniformly distributed with essentially a square grid (cf. Method 1 of part (a) in the solution of Example 13.2).
(c) The density of the nodes increases as we near the airfoil (inside) part of the boundary and the outside rectangle will have between 20 and 30 nodes. See Figure 13.21 for examples of

some related triangulations.

Suggestions: To find appropriate x and y vectors for the foil, it is probably easiest to copy the figure down on graph paper and record a set of ordered (so it will plot correctly) vertices on the foil that is dense enough so as to render a decent plot. A more elaborate scheme would be to build up the boundary in terms of piecewise cubic splines whose derivatives match up on the interfaces (see the end of Section 10.5). MATLAB has a useful command for some of the tasks of this problem called `inpolygon` that will test if a given point lies within a given polygon:

`test = inpolygon(x,y, xpoly, ypoly)` →	If `xpoly` and `ypoly` are the x- and y-coordinates of a set of vertices defining a polygon and `x` and `y` are the coordinates of any point in the plane, the output `test` will be 1 if the point (x,y) lies inside or on the boundary of the polygon and 0 otherwise. If `x` and `y` are vectors for a set of points, the output `test` will be a corresponding vector of 1's and/or 0's.

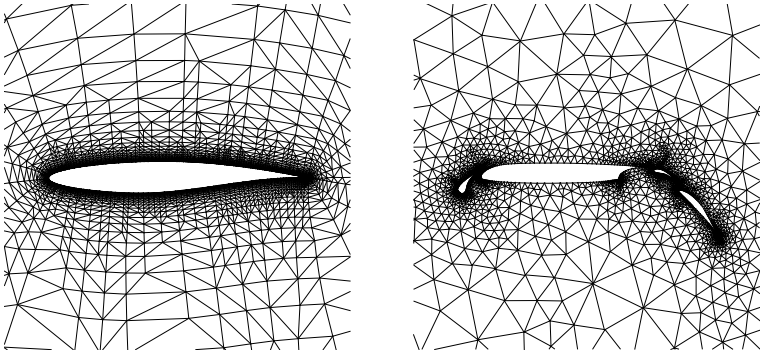


FIGURE 13.21: Two triangulations of airfoils. (a) (left) A single component airfoil similar to that in Figure 13.20(d). The triangulation is structured using lines normal to the surface (with increasing density as we near the boundary). (b) A more complex airfoil with flaps. The triangulation is done in a way that the node density increases as we near crucial portions of the configuration.[8]

NOTE: (*Rectangular Elements*) For domains whose boundaries are made up of only vertical and horizontal segments, rectangular elements are often a popular choice for the FEM. A typical rectangular element is illustrated in Figure 13.22(a). If we use just the four vertices as the nodes of each rectangular element, then each local basis function has four degrees of freedom, so linear functions (whose graphs are planes) are no longer permissible to use as local basis functions. Popular choices for basis functions in this case are piecewise bilinear functions:

$$axy + bx + cy + d.$$

These functions reduce to linear functions on any of the four edges of the rectangle so that continuity is assured across boundaries when elements are put together. Note that this would not be the case if the element were an arbitrary quadrilateral (if not all four sides are parallel to one of the axes). The next four exercises look more closely into rectangular elements.

[8] These two triangulations were created by Tim Barth (at the NASA Ames Research Center) and we thank him for his kind permission to include them in this text. Such triangulations of airfoils coupled with the FEM are used to model aerodynamics and design space and air vessels.

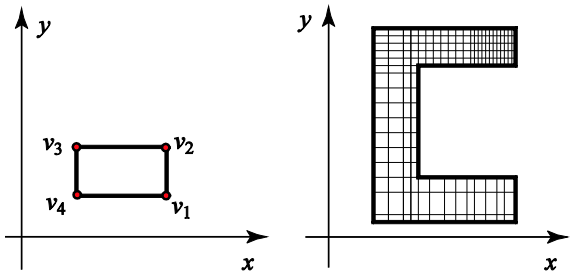


FIGURE 13.22: (a) (left) Illustration of a typical rectangular element with its four nodes consisting of its vertices. (b) (right) Tessellation of a domain into rectangular elements.

16. For the domain in Figure 13.22(b), let the outer vertices be (1,1), (7,1), (7,3), (3,3), (3,6), (7,6), (7,8), and (1,8). Tessellate the domain with square elements having unit sidelength (so there should be 30 elements).
(a) Write down a formula for the basis function $\Phi_{(2,2)}(x,y)$ corresponding to the interior node (2,2).
(b) Use MATLAB to draw a three-dimensional graph of this basis function.
(c) Repeat parts (a) and (b) for the basis function $\Phi_{(1,1)}(x,y)$ corresponding to the interior node (1,1).
(d) Are these basis functions differentiable (smooth) across all edges of adjacent elements? (It was already pointed out that they are continuous across edges, and this should be evidenced from the graphs.)

17. Let the domain in Figure 13.22(b) have the vertices and tessellation of the last exercise. On this domain, consider the following function:

$$f(x,y)=\begin{cases}[(x-1)/2]^2, & \text{if } y\le 3,\\ [(x-1)/2]^2(2/3)\,|\,y-9/2\,|, & \text{if } 3\le y\le 9/2,\\ -[(x-1)/2]^2(2/3)\,|\,y-9/2\,|, & \text{if } 9/2\le y\le 6,\\ -[(x-1)/2]^2, & \text{if } y\ge 6,\end{cases}$$

(a) Use MATLAB to draw a three-dimensional graph of this function.
(b) Use MATLAB to draw a three-dimensional graph of the finite element interpolant to this function using the basis functions (Exercise 16) for the square elements of the tessellation. Note that this approximation is simply the function:

$$f(1,1)\Phi_{(1,1)}(x,y)+f(1,2)\Phi_{(1,2)}(x,y)+\cdots,$$

where each term of the sum corresponds to a node of the tessellation.
(c) Create and plot a corresponding approximation to $f(x,y)$ that arises from the triangulation of the domain using 60 triangles, each square element giving rise to two triangular elements via the diagonal from lower left to upper right.
(d) Repeat part (b) except this time use squares of sidelength 1/4 in the tessellation. (So there will be 16 times as many elements.)
Suggestion: In parts (b) and (d), use the `meshgrid` command for each element and use the `hold on` command.

18. The **standard rectangular element** has vertices $(\pm 1,\pm 1)$. (a) Show that the corresponding four local basis functions (viz. (7)) are given by the following formulas (the ordering of the nodes is as in Figure 13.22a):

$$\rho_3(x,y)=(1/4)(1-x)(1+y),\ \rho_2(x,y)=(1/4)(1+x)(1+y),$$
$$\rho_4(x,y)=(1/4)(1-x)(1-y),\ \rho_1(x,y)=(1/4)(1+x)(1-y).$$

(The local basis function ρ_i corresponds to the vertex v_i and they are written with the same

orientation as the vertices appear in the element.)
(b) Use MATLAB to draw three-dimensional graphs of each of these four local basis functions.

19. (a) Find formulas, as in Exercise 18, for the four local basis functions for a general rectangular element with vertices: (a,b), $(a+h,b)$, $(a+h,b+k)$, $(a,b+k)$. Your formulas will depend, of course, on the parameters a, b, h, and k.
(b) Find an affine mapping $(x,y)=F(\tilde{x},\tilde{y})$ that carries the standard rectangular element of Exercise 18 (thought of as lying in the $\tilde{x}\tilde{y}$-plane) onto the general rectangular element of part (a) (thought of as lying in the xy-plane). In matrix form the mapping can be written as $\begin{bmatrix} x \\ y \end{bmatrix} = A\begin{bmatrix} \tilde{x} \\ \tilde{y} \end{bmatrix} + v$, where A is a 2×2 matrix and v is a 2×1 vector. How is the determinant of the matrix A related to the areas of the two rectangular elements?
Suggestion: For part (a), try first by trial and error for some simple specific parameters; let them get more general and look for patterns. For example, you might start with $a = -1$, $h = 2$, $b = -1$, $k = 1$. These parameters are very close to those of the standard element with one difference (h is 2 instead of 1). Next try changing h to 3, keeping all else fixed. Then use $a = -1$, $h = 1$, $b = -1$, $k = 2$; finally change a and b to other values, etc. Alternatively, part (a) can be done quite elegantly using part (b). See Exercise 23 for the relevant idea.

20. We define a planar domain Ω to be *horizontally blocklike* if it has the following form:

$$\Omega = \{(x,y):\ a \le x \le b,\ 0 \le y \le \lambda(x)\},$$

where $\lambda(x)$ is a positive-valued step function on $a \le x \le b$, i.e., there is a partition of $a \le x \le b$: $a = a_0 < a_1 < a_2 < \cdots < a_{n+1} = b$ into $n+1$ subintervals $I_i = [a_{i-1}, a_i]$ $(1 \le i \le n+1)$ and corresponding positive numbers c_i $(1 \le i \le n+1)$, such that $\lambda(x)$ can be written as: $\lambda(x) = c_i \Leftrightarrow x \in I_i$ $(1 \le i \le n+1)$. A typical horizontally blocklike domain is illustrated in Figure 13.23.

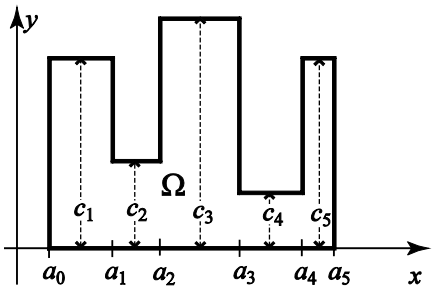


FIGURE 13.23: Illustration of a typical horizontally blocklike domain Ω for Exercise 20.

(a) Write an M-file, `[x y nodes elems]=recttess_hbd_basic(a,c,h)`, that will perform a basic rectangular tessellation of a horizontally blocklike domain in the following fashion. The input parameters are firstly two vectors `a` and `c` that contain the defining parameters of the horizontally blocklike domain to be tessellated. It is assumed of course that `c` has one less component than `a`, the components of `a` are increasing, and the components of `c` are positive (otherwise they would not define a horizontally blocklike domain).
The final input variable, `h`, will be the (approximate) sidelength of each of the rectangles used in the tessellation. More specifically, each of the elements (rectangles) in the tessellation should have its length (l) and width (w) lying within the interval: $\frac{1}{2}h \le w,l \le 2\cdot h$. The tessellation will be a basic one in the sense that it will be completely determined by a single set of horizontal and vertical grid lines. (Note: This is not the case for the tessellation of Figure 13.22(b), since different sets of vertical lines are used for the grids in the upper and lower passages.) The first

two output variables `x` and `y` are vectors of the values of the corresponding vertical lines and horizontal lines defining the tessellation. The third output variable, `nodes`, is a 2-column matrix giving all of the nodes of the tessellation. The fourth and final output variable, `elems`, is a 4-column matrix giving the node numbers of the each of the elements, where the ordering of the elements starts at the lower left, moves all the way up, then back down to the bottom to the next element on the right, and so on.
(b) Run your program using the following sets of input variables:
(i) a = [1 3 4 7 9 10], c = [8 4 12 2 10], h = 3,
(ii) a = [1 3 4 7 9 10], c = [8 4 12 2 10], h = 1,
(iii) a = [1 3 4 7 9 10], c = [8 4 12 2 10], h = 0.13,
and for each of these for which a tessellation is created, plot the tessellation. For (iii), your plot should look like the one shown in Figure 13.24.

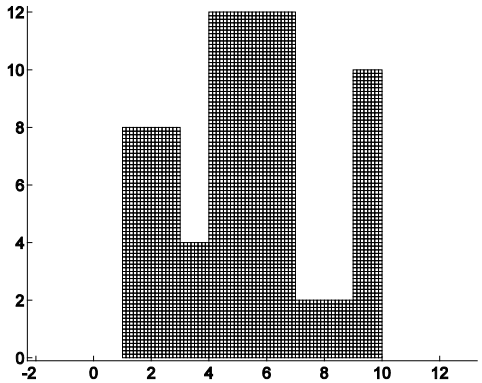


FIGURE 13.24: A tessellation of a horizontally blocklike domain obtained using a program of Exercise 16 using the input data of part (b)(iii). There are 4694 nodes and 4432 rectangular elements.

Suggestions: Part (a): First use the `sort` command to create a vector `cvals` of the values of `c` (in increasing order), with zero appended as the smallest value. Now find the minimum gaps occurring in the vectors `cvals` and `a`. The inputted `sidelength` should not be too small relative to the smaller of these two gaps. If the `sidelength` exceeds, say, twice this minimum gap value, have the function exit with an error flag (and no tessellation). Now move on to defining a vector `x` for the vertical gridlines of the tessellation. Use a loop, running through each of the gaps determined from the values of `a`. If the size of a certain gap is less than twice the `sidelength`, let `x` simply contain the values of `a` at the ends of this gap (no interior grid values); otherwise, use $k - 1$ interior and equally spaced gridlines within the gap, where `k=ceil(gap/sidelength)`. (You need to verify that this will result in elements having horizontal sidelengths within the desired bounds.) In a similar fashion, define a vector `y` for the horizontal gridlines of the tessellation. Next, use the vectors `a`, `c`, `x`, and `y` to define the matrix `nodes`. This can be done with a double loop, but note that you will need to set it up so the larger `c` value is used in cases where x(i) lies on an interface of two blocks. Finally use the vectors `x`, `y` and the matrix `nodes` to create the matrix `elems` of elements. You should set things up so that for a given element (row of the `elems` matrix) the nodes progress, say counterclockwise, around the element. With this being done, plotting of the tessellations (part (b)) can easily be accomplished using the following simple loop:

```
hold on, [e1  e2]=size(elems);
for i=1:e1
R=nodes(elems(i,:),:);
xr=R(:,1); xr(5)=xr(1);yr=R(:,2); yr(5)=yr(1);
plot(xr,yr)
end
```

MATLAB's `find` command can be useful for many parts of this program.

21. (a) Referring to Exercise 20, formulate the definition of the corresponding concept of a *vertically blocklike domain*.
(b) Write an M-file:

```
[x y nodes elems]=recttess_vbd_basic(a,c,sidelength)
```

that will perform a basic rectangular tessellation of a vertically blocklike domain in a similar syntax and fashion to the program of part (a) of Exercise 16. Here, the input variable `a` is an increasing vector corresponding to the *y*-values of endpoints of the blocks, and the vector `c` (length one less than `y`) gives the corresponding horizontal lengths of the blocks.
(c) Run your program using the following sets of input variables:
(i) a = [2 3 5 6 8 10], c = [6 8 7 4 2], h = 3,
(ii) a = [2 3 5 6 8 10], c = [6 8 7 4 2], h = 1,
(iii) a = [2 3 5 6 8 10], c = [6 8 7 4 2], h = 0.13.
Suggesions: Refer to those of Exercise 20 for ideas. If the reader has already completed Exercise 20(a), the current program could invoke that of Exercise 20 along with a rotation of axes (viz. Section 7.2). After all, a vertically blocklike domain is simply a rotation of a horizontally blocklike domain and vice versa. The same goes for corresponding tessellations.

22. Prove identity (5) equating the area of a triangle in the plane with vertices: (x_r, y_r), (x_s, y_s), and (x_t, y_t) to half the absolute value of determinant of the matrix

$$\mathrm{M}=\begin{bmatrix} x_r & y_r & 1 \\ x_s & y_s & 1 \\ x_t & y_t & 1 \end{bmatrix}.$$

Suggestion: First use some properties of determinants from Chapter 7 to observe that the determinant will not change if a constant X is added to all of the x-coordinates (first column) and/or a constant Y is added to all the y-coordinates (second column). The corresponding effect on the triangle is simply a shift, leaving the area unchanged. Thus, we may assume that the triangle lies in the first quadrant. Furthermore, reduce to a configuration such as that shown in Figure 13.25. Express the area of the triangle shown as the difference of the sum of the areas of the two trapezoids between the top two edges of the triangle and the x-axis, less the area of the trapezoid between the bottom edge of the triangle and the x-axis. Compare this expression with the det(M). Recall that the area of a trapezoid with base b and heights h_1, h_2 equals $(b/2)(h_1 + h_2)$.

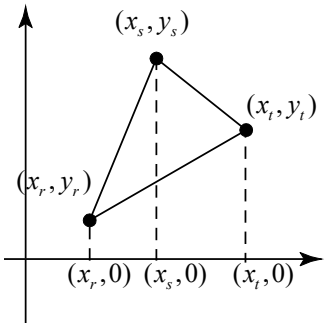


FIGURE 13.25: Geometric diagram for the proof in Exercise 22.

23. To gain a deeper understanding of elements, it is often convenient to work with a so-called **standard element**, which is essentially equivalent to all elements. For our triangular elements, with three nodes at the vertices, we will use the standard element $\tilde{T}$ that has vertices $v_1 = (1, 0)$,

$v_2 = (0,1)$, and $v_3 = (0,0)$. This standard element is illustrated in Figure 13.26.

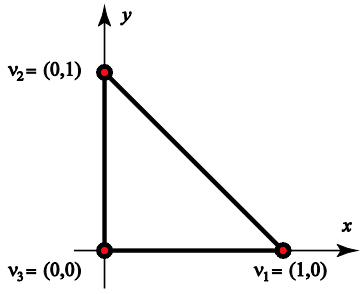


FIGURE 13.26: Illustration of the standard element for all triangular elements with three nodes at the vertices.

(a) Show that the standard local basis functions (viz. (7)) for the standard element of Figure 13.26 are given by:

$$\phi_1(x,y) = x,\ \phi_2(x,y) = y,\ \text{and}\ \phi_3(x,y) = 1 - x - y.$$

(b) For any triangular element T with specified vertices $v_1 = (x_r, y_r)$, $v_2 = (x_s, y_s)$, and $v_3 = (x_t, y_t)$ (labeled in counterclockwise order), show that the following affine mapping $(x,y) = F(\tilde{x},\tilde{y})$ (see Section 7.2) will transform the standard basis element $\tilde{T}$ onto T and map corresponding nodes onto one another:

$$\begin{aligned} x &= (x_r - x_t)\tilde{x} + (x_s - x_t)\tilde{y} + x_t, \\ y &= (y_r - y_t)\tilde{x} + (y_s - y_t)\tilde{y} + y_t, \end{aligned} \quad \text{or} \quad \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x_r - x_t & x_s - x_t \\ y_r - y_t & y_s - y_t \end{bmatrix} \begin{bmatrix} \tilde{x} \\ \tilde{y} \end{bmatrix} + \begin{bmatrix} x_t \\ y_t \end{bmatrix}.$$

For clarity, we have used two different sets of coordinates, $(\tilde{x},\tilde{y})$ for the coordinates of the plane for $\tilde{T}$ and (x,y) for the coordinates of the plane of T. The action of this affine transformation is illustrated in Figure 13.27.

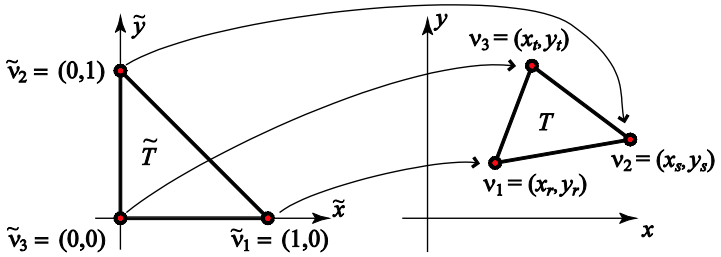


FIGURE 13.27: Illustration of the action of the affine mapping of Exercise 23(b) that takes the standard element $\tilde{T}$ onto an arbitrary element T.

(c) Discover and then prove a relationship for the determinant of the 2×2 matrix of the affine transformation of part (b) and the areas of the two elements T, $\tilde{T}$. If you are not sure of the relationship, do some experiments using MATLAB. For the proof, cf., Exercise 22.

(d) Writing $A = \begin{bmatrix} x_r - x_t & x_s - x_t \\ y_r - y_t & y_s - y_t \end{bmatrix}$ for the matrix of the affine transformation of part (b) observe first that the inverse affine mapping is given by: $(\tilde{x}, \tilde{y}) = F^{-1}(x, y) = A^{-1} \cdot \left(\begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} x_t \\ y_t \end{bmatrix} \right)$, and show that the standard local basis elements for T are related to those (of part (a)) for $\tilde{T}$ as follows:

$$\phi_i(x, y) = \tilde{\phi}_i(F^{-1}(x, y)), \quad i = 1, 2, 3.$$

Note: Here we have used the notational convention of part (b), so that the $\tilde{\phi}_i$'s are the standard local basis elements corresponding to $\tilde{T}$, while ϕ_i are those corresponding to T. To prove this relation one simply needs to observe that both sides are linear functions of (x,y) and compare them on the vertices v_1, v_2, v_3.

24. (*Quadratic Basis Functions on Triangular Elements*) For some BVPs it is desirable to use basis functions Ψ_j which are piecewise quadratic rather than the piecewise linear basis functions that were used in the text. Thus, on each element T, such a basis function will have its general formula written as:

$$\Psi_j(x, y) = ax^2 + bxy + cy^2 + dx + ey + f,$$

where, in order to simplify notation, we have omitted subscripts and superscripts on the six coefficients a, b, c, d, e, f. Since we now have six local basis functions (for each term), we will need to correspondingly have six nodes on each element in order that the coefficients be uniquely determined. A very natural (and as it turns out effective) way to do this is to put three extra nodes on the midpoints of each of the edges of the elements. The corresponding standard element (see Exercise 23), is shown in Figure 13.28.

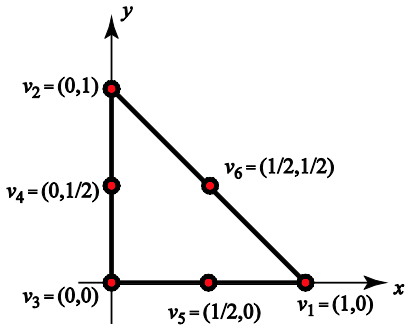


FIGURE 13.28: The standard triangular element with six nodes for piecewise quadratic FEM. The three additional nodes from the piecewise linear standard elements are placed at the midpoints of the segments, the numbering is as before for the vertex nodes, while the midpoint nodes are numbered counterclockwise in order of the opposite vertices.

(a) Show that (by analogy with (7)) the corresponding standard local basis functions for the standard element of Figure 13.26 are given by:

$$\begin{aligned}
\psi_1(x, y) &= \phi_1(x, y) \cdot (2\phi_1(x, y) - 1), \\
\psi_2(x, y) &= \phi_2(x, y) \cdot (2\phi_2(x, y) - 1), \\
\psi_3(x, y) &= \phi_3(x, y) \cdot (2\phi_3(x, y) - 1), \\
\psi_4(x, y) &= 4\phi_2(x, y) \cdot \phi_3(x, y), \\
\psi_5(x, y) &= 4\phi_1(x, y) \cdot \phi_3(x, y), \\
\psi_6(x, y) &= 4\phi_1(x, y) \cdot \phi_2(x, y),
\end{aligned}$$

where the ϕ_j's are the piecewise linear standard local basis functions of Exercise 23(a).

(b) Do the six identities of part (a) continue to remain valid when the local basis functions correspond to an arbitrary element?

(c) Show that the affine mapping $(x,y) = F(\tilde{x},\tilde{y})$ of Exercise 23(b) maps the standard triangular element of Figure 13.28 (viewed in the $\tilde{x}\tilde{y}$-plane) onto the corresponding triangular element with midpoint nodes (viewed in the xy-plane) such that the node correspondence is maintained.

(d) Now letting $\tilde{\psi}_j(x,y)$ $j = 1,...,6,$ denote the standard local basis functions of part (a) and $\psi_j(x,y)$ denote the corresponding local basis functions for an arbitrary element, prove that $\psi_i(x,y) = \tilde{\psi}_i(F^{-1}(x,y))$, $i = 1,...,6,$ where F is the affine mapping of part (c).

25. (*Quadratic Basis Functions on Triangular Elements, Cont.*) Let $\psi(x,y) = ax^2 + bxy + cy^2 + dx + ey + f$ be a quadratic function on a triangular element T with six nodes (vertices and midpoints). Assume that $\psi(x,y) = 0$ on an entire line segment of T that is opposite to vertex v_j. Prove that $\psi(x,y)$ can be factored as $\psi(x,y) = \phi_j(x,y) \cdot \phi(x,y)$ where $\phi_j(x,y)$ is the standard linear local basis function for the vertex v_j and $\phi(x,y)$ is some other linear function $\phi(x,y) = ax + by + c$.

Suggestion: First prove this for the case of the standard element and j = 2, then use affine mapping (see Exercises 23 and 24) to translate your proof to work for a general triangular element.

26. (*Quadratic Basis Functions on Triangular Elements, Cont.*) (a) Use MATLAB to create three-dimensional plots of each of the standard local basis functions $\psi_j (j = 1,...,6)$ for the standard element of Figure 13.28. The graphs of two of these functions are roughly depicted in Figure 13.29.

Suggestion: One way to get high-resolution plots over such a triangular region is to triangulate it into much smaller elements and then use the `trimesh`.

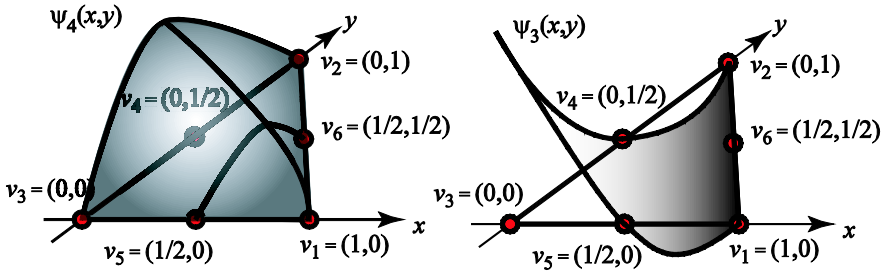


FIGURE 13.29: Graphical illustration of two of the quadratic local basis functions for the standard triangular element of Figure 13.28.

(b) Write down a formula for the (nonlocal) basis function $\Psi_4 = \Psi_4(x,y)$ corresponding to the interior node #4 of the triangulation of Figure 13.4 using the nodal parameters given below Figure 13.5.

Suggestion: The midpoint nodes will need to be numbered. This can be done systematically as in Example 13.1, but the linear systems will of course be larger.

(c) Use MATLAB to plot the (nonlocal) basis function $\Psi_4 = \Psi_4(x,y)$.

(d) Repeat parts (b) and (c) for the midpoint node between the numbered nodes 4 and 5.

27. Given any finite set $P = \{p_1, p_2, \cdots, p_n\}$ of distinct points in the plane, show that each of the corresponding Voronoi boxes $V(p_i)$ is a convex set.

Suggestion: Observe that a Voronoi box is an intersection of half-planes.

28. As mentioned in the text, when a triangulation is created for a given domain to use in the FEM, it is usually desirable to have the angles of each of the elements not get too small. In this exercise you will be creating an M-file that will be able to perform a check for this on a given triangulation and locate any "problem elements."
(a) Write an M-file, called `theta = minangle(v1, v2, v3)` whose three input variables `v1`, `v2`, `v3` are 2×1 matrices giving the coordinates of the three vertices of a triangle in the *xy*-plane, and whose output `theta` is the smallest (interior) angle of this triangle, measured in degrees.
(b) Write an M-file called `theta = minanglemesh(x,y,tri)` whose inputs are two vectors `x`, `y` of the same size giving the coordinates of the nodes of a triangulation, and `tri`, a 3-column matrix having as its rows the node numbers of the elements in the triangulation. The output `theta` will be the minimum angle (measured in degrees) of any angle of any element of the triangulation. Run this program on the triangulations of Figure 13.5 (with parameters given in the matrices *N* and *T* preceding Example 13.1), as well as each of the triangulations created in Example 13.2 of the unit disk.
(c) Write an M-file called `[badelems, thetas] = minanglemesh(x,y,tri,tol)` that, along with the input variables of part (b), has the additional input variable `tol` that will be a positive number denoting the smallest desired angle (measured in degrees) to be tolerated in a triangulation. There are two output variables: `badelems`, which will give the element numbers (corresponding to row numbers of `tri`) whose minimum angles are less than `tol`, and `thetas`, which is a vector of the same size as `badelems` gives the corresponding offending minimum angles of the bad elements. Additonally, a graphic will be produced that will graph only the elements corresponding to `badelmens`. This will allow for appropriate measures to be taken to modify the triangulation, if necessary. Run this program on the triangulations of Figure 13.5 (with parameters given in the matrices *N* and *T* preceding Example 13.1), as well as each of the triangulations created in Example 13.2 of the unit disk, using three different values for `tol` for each: The first one chosen so that there are no offending elements, the second chosen so that there are a few offending elements (if possible), and the third chosen so there are a lot of offending elements.

13.3: THE FINITE ELEMENT METHOD FOR ELLIPTIC PDE'S

In this section we will present versions of the FEM for solving the following general type of BVP on a domain $\Omega \subset \mathbb{R}^2$

$$\begin{cases} \text{(PDE)} & -\nabla\bullet(p\nabla u) + qu = f & \text{on } \Omega \\ \text{(BCs)} & u = g & \text{on } \Gamma_1 \\ & \vec{n}\bullet\nabla u + ru = h & \text{on } \Gamma_2 \end{cases}. \tag{10}$$

The data functions: p, q, f, g, r, h are allowed to be functions of (*x*,*y*), defined on their respective indicated sets. The boundary $\partial\Omega$ is decomposed into the portions Γ_1 and Γ_2, $\partial\Omega = \Gamma_1 \cup \Gamma_2$. On the first portion Γ_1 there are Dirichlet boundary conditions $u = g$, and on the complementary portion Γ_2 we are assuming

generalized Neumann boundary or **Robin** boundary conditions: $\vec{n}\bullet(p\nabla u)+ru=h.$ Here $\vec{n}=\vec{n}(x,y)$ denotes the outward unit normal vector defined on the $\partial\Omega$, and $\nabla u=(\partial u/\partial x,\partial u/\partial y)$ is the gradient of $u(x,y)$. Thus, from multivariable calculus, the dot product $\vec{n}\bullet\nabla u(x,y)$ is just the partial derivative of u in the direction of the outward pointing normal vector $\vec{n}=\vec{n}(x,y)$ at any point (x,y) on the boundary. If $r(x,y)\equiv 0,$ the BCs on Γ_2 generalize the usual Neumann boundary conditions.[9] We allow for the possibility that either $\Gamma_1=\partial\Omega$ (so $\Gamma_2=\varnothing$) and the boundary conditions are purely of Dirichlet form, or that $\Gamma_2=\partial\Omega$ with boundary conditions being entirely of Robin form. The PDE in (10) is written in the so-called **divergence form**. This is the most general form for linear elliptic PDEs on which the standard FEM is applicable, and indeed this is the most general elliptic PDE to which MATLAB's symbolic toolbox is applicable. A great many elliptic boundary value problems can be expressed in the form (10). Sometimes, it will be convenient for us to write the PDE in (10) in expanded form:

$$-\partial/\partial x[pu_x]-\partial/\partial y[pu_y]+qu=f \qquad \text{on } \Omega.$$

The reason for the negative signs will become clear once the FEM is introduced. This PDE is the natural generalization to two space variables of the ODEs that were considered in Section 10.5. We begin by outlining the FEM for the BVP (10) in the case of purely Dirichlet boundary conditions (i.e., $\Gamma_2=\varnothing$). The FEM will look quite similar to the one-dimensional version presented in Section 10.5. The proofs of the underlying results will not be included in this text. They share many common elements with the one-dimensional theory presented in Section 10.5, but for technical reasons, the higher dimensional analogues require some more advanced mathematical machinery (including, for example, some elements of Sobolev spaces). The interested reader can consult one of the following references: [Cia-02], [AxBa-84], [StFi-73], or [Joh-87] for more details on the theory.

In cases of purely Dirichlet boundary conditions and when the data the BVP (10) satisfy: p,q,f are piecewise continuous on Ω, along with the first partial

---

[9] Let us briefly review the physical significance of the three types of BCs in the context of a steady-state heat distribution BVP (a prototypical BVP). The Dirichlet boundary condition $u=g$ means that (on the portion of the boundary where the condition holds) the boundary is being maintained (by some coolant or heat reservoir) at a specified temperature. The Neumann boundary condition $\vec{n}\bullet\nabla u=0$ means that the boundary is insulated (no heat loss or transfer). The Robin boundary condition (after dividing through by p, which will always be assumed positive): $\vec{n}\bullet\nabla u+ru=h$ when written in the form $\vec{n}\bullet\nabla u=-r(u-\tilde{h})$ looks like the usual Newton's law of cooling where the net heat transfer (out of the region) is proportional to the difference of the inside temperature (u) and the outside temperature ($\tilde{h}$).

derivatives of p and q, g is piecewise continuous on $\partial\Omega$ and $p(x,y) > 0, q(x,y) \geq 0$, the BVP can be shown to be equivalent to the minimization problem:

Minimize the functional:

$$F[u] = \iint_{\Omega} [\tfrac{1}{2} p u_x^{\,2} + \tfrac{1}{2} p u_y^{\,2} + \tfrac{1}{2} q u^2 - f u]\, dx dy\,, \tag{11}$$

over the following set of admissible functions:

$$\mathcal{A} = \{ v : \Omega \to \mathbb{R} : v(x) \text{ is continuous}, v'(x) \text{ is piecewise continuous and bounded, and } v(x,y) = g(x,y) \text{ on } \partial\Omega \}. \tag{12}$$

The concept of piecewise continuity on Ω (or $\partial\Omega$) simply means that the domain (or boundary) can be broken up into finitely many elements (arcs) on each of which the given function reduces to a continuous function.

Analogous to the one-dimensional method presented in Section 10.5, the FEM will solve a corresponding finite-dimensional minimization problem where the functional $F[u]$ of (11) is kept the same, but the set of admissible functions is reduced to an approximating smaller set that is determined by the basis functions of the triangulation. Thus we will be looking for minimizers of the functional F among functions of the form $v = \sum_{i=1}^{m} c_i \Phi_i$, where the $\Phi_i = \Phi_i(x,y)$ are the basis functions. The basis functions corresponding to nodes on the boundary will have their coefficients determined by the Dirichlet boundary conditions; it is the remaining coefficients (corresponding to interior nodes) that need to be determined. We now briefly outline the FEM for BVPs with purely Dirichlet BCs. We follow this outline with some additional details and then give examples.

FEM FOR THE BVP (10) IN CASE OF PURELY DIRICHLET BC'S ($\Gamma_2 = \varnothing$):

Step #1: Decompose the domain into elements, and represent the set of nodes and elements using matrices. Separate the nodes N_i into the internal nodes: $N_1, N_2, \cdots, N_n$ (that lie in Ω), and the boundary nodes $N_{n+1}, N_{n+2}, \cdots, N_m$ (that lie on $\partial\Omega$). Denote the basis function Φ_{N_i} corresponding to node N_i simply by Φ_i.

Step #2: Use the Dirichlet BCs $u(x,y) = g(x,y)$ on $\partial\Omega$ to determine the coefficients of the boundary node basis functions of an admissible function: $v = \sum_{i=1}^{m} c_i \Phi_i$, i.e., $c_i = g(N_i)$ for each $i = n+1, n+2, \cdots, m$.

Step #3: Assemble the $n \times n$ stiffness matrix A and load vector b needed to determine the remaining coefficients $c_1, c_2, \cdots, c_n$ that work to solve the discrete minimization problem corresponding to the BVP.
Step #4: Solve the stiffness equation $Ac = b$, and obtain the FEM solution

$$v = \sum_{i=1}^{m} c_i \Phi_i .$$

The first step was examined in detail in the last section for triangular elements with piecewise linear basis functions. Such elements and basis functions are the ones that will be used exclusively in the text of this section. The exercises will consider some other sorts of elements and/or basis functions. Step #2 is rather clear. Step #3 will be accomplished by a so-called **assembly** technique where the entries of the stiffness matrix and load vectors are built by looking at the contributions of each element.

If we substitute the expression $v = \sum_{i=1}^{m} c_i \Phi_i$ for u into the functional $F[u]$, and then differentiate with respect to c_k (under the integral sign), we arrive at the following equation (Exercise 17):

$$\frac{\partial}{\partial c_k} F\left[\sum_{i=1}^{m} c_i \Phi_i\right] = \iint_{\Omega} [p \sum_{i=1}^{m} c_i \partial_x(\Phi_i)\partial_x(\Phi_k) + p \sum_{i=1}^{m} c_i \partial_y(\Phi_i)\partial_y(\Phi_k) + q \sum_{i=1}^{m} c_i \Phi_i \Phi_k - f \Phi_k] dxdy. \tag{13}$$

Keeping in mind that the values of c_k for $k > n$ will have been computed in Step #2, since we seek a critical point of F, we set the above equations equal to zero for $1 \le k \le n$ to obtain the following $n \times n$ linear system for the unknown coefficients:

$$Ac = b, \tag{14}$$

where the c represents the (column) vector of the unknown (internal node) coefficients: $c = [c_1 \;\; c_2 \;\; \cdots \;\; c_n]'$. The entries of the stiffness matrix $A = [a_{ij}]$ are given by (Exercise 17):

$$a_{ij} = \iint_{\Omega} [p \nabla \Phi_i \bullet \nabla \Phi_j + q \Phi_i \Phi_j] dxdy \;\; (1 \le i, j \le n), \tag{15}$$

and the entries of the load vector $b = [b_j]$ are given by:

$$b_j = \iint_{\Omega} f \Phi_j \, dxdy - \sum_{s=n+1}^{m} \iint_{\Omega} [p \nabla \Phi_s \bullet \nabla \Phi_j + q \Phi_s \Phi_j] dxdy \;\; (1 \le j \le n). \tag{16}$$

We point out that the coefficients c_s ($s > n$) are known from Step #2. Note that from (15) (since the dot product is commutative: $\vec{v} \bullet \vec{w} = \vec{w} \bullet \vec{v}$) it follows that $a_{ij} = a_{ji}$, i.e., the stiffness matrix is a symmetric matrix.

Keeping in mind that each of the basis functions is made up of its linear "pieces" on each of the elements, it is more efficient to compute the stiffness entries a_{ij} and load entries b_j by running through each of the elements and adding up contributions. Assuming that the nodes and elements have been stored in a 2-column matrix *N* and a 3-column matrix *E*, respectively (as in the last section, but then we labeled the element matrix as *T*), we now outline the assembly process:

ASSEMBLY PROCESS FOR THE FEM FOR (10) IN CASE OF PURELY DIRICHELT BC'S ($\Gamma_2 = \varnothing$):

Step #1: Initialize $n \times n$ stiffness matrix A and $n \times 1$ load vector b with all zero entries.
Step #2: Let ℓ run from 1 to L = the number of elements (= number of rows of the matrix E whose ℓth row gives the node numbers of the ℓth element T_ℓ). For each index ℓ, we create the 3×3 element stiffness matrix $A^\ell = [a^\ell_{\alpha\beta}]$ ($1 \le \alpha, \beta \le 3$) for the element T_ℓ and the corresponding 3×1 element load vector $b^\ell = [b^\ell_\alpha]$ by restricting the integrals in formulas (15) and (16) from Ω to T_ℓ:

$$a^\ell_{\alpha\beta} = \iint_{T_\ell} [p \nabla \Phi_{i_\alpha} \bullet \nabla \Phi_{i_\beta} + q \Phi_{i_\alpha} \Phi_{i_\beta}] dxdy \quad (1 \le \alpha, \beta \le 3), \tag{15$^\ell$}$$

and

$$b^\ell_\alpha = \iint_{T_\ell} f \Phi_{i_\alpha} dxdy - \sum_{s=n+1}^{m} c_s \iint_{T_\ell} [p \nabla \Phi_s \bullet \nabla \Phi_{i_\alpha} + q \Phi_s \Phi_{i_\alpha}] dxdy \quad (1 \le \alpha \le 3). \tag{16$^\ell$}$$

(Here, the index i_α denotes the global node number corresponding to the αth vertex of T_ℓ, i.e., $i_\alpha = E(\ell, \alpha)$, whereas the local node number α for a vertex of T_ℓ is just the corresponding column number of the index α in the ℓth row of the element matrix E.) We then transplant these contributions into the appropriate places of the (global) stiffness matrix and load vector:

$$A(E(\ell,\alpha), E(\ell,\beta)) = A(E(\ell,\alpha), E(\ell,\beta)) + a^\ell_{\alpha\beta} \quad (1 \le \alpha, \beta \le 3), \tag{17}$$

and

$$b(E(\ell,\alpha)) = b(E(\ell,\alpha)) + b^\ell_\alpha \quad (1 \le \alpha \le 3). \tag{18}$$

We point out that formulas (15^{ℓ}) and (16^{ℓ}) need only be carried out when the indices α and/or β correspond to interior nodes.[10] Also, the integrands in summation of (16^{ℓ}) will vanish on the element T_{ℓ} unless the corresponding exterior node (number s) is a vertex of T_{ℓ}.

We turn now to a simple example involving the Poisson PDE and constant (Dirichlet) boundary conditions on the hexagonal domain of the last section with only eight triangular elements (Figure 13.5). MATLAB will be able to help us with general multiple integrals, and we will explain how this can be done after this introductory example. The integrals that will need to get done in the course of this example will be simple enough to do by hand, and all will be evaluated using the results of the following exercise for the reader:

EXERCISE FOR THE READER 13.5: Let T denote any (convex) triangle in the plane having vertices $v_1 = (x_r, y_r)$, $v_2 = (x_s, y_s)$, and $v_3 = (x_t, y_t)$ (Figure 13.30), and let $\phi = \phi_3$ denote the local basis function for T corresponding to the vertex v_3, i.e., $\phi(x, y)$ is the linear function determined by the equations $\phi(v_i) = \delta_{i3}$ ($i = 1, 2, 3$). Establish the following formulas:

(a) The gradient vector $\vec{\nabla}\phi$ points in the direction of the altitude $\vec{a}$ and has magnitude $\| \nabla\phi \| = 1 / \| \vec{a} \|$.

(b) $\iint_T \phi(x, y)dxdy = \frac{1}{3}\text{Area}(T)$.

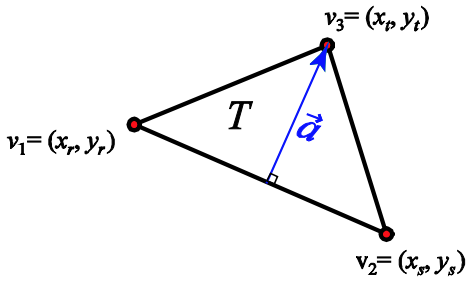


FIGURE 13.30: A typical (convex) triangular element whose local basis function $\phi = \phi_3$ is analyzed in Exercise for the Reader 13.5. Since the element is convex, the (blue) altitude vector $\vec{a}$ shown will lie inside the triangle.

EXAMPLE 13.5: Let Ω be the hexagonal domain of Figure 13.5 with eight nodes (as labeled) given by: #1: (1,1), #2: (2.5,1), #3: (0,0), #4: (1,0), #5:

[10] Otherwise the entries are meaningless. Thus, technically, the element stiffness matrices A^{ℓ} (load vectors b^{ℓ}) will not be complete 3×3 (3×1) matrices in cases where the element T_{ℓ} has some of its vertices on the boundary (in Example 13.5, this will be the case for all of the elements).

(2.5,0), #6: (3.5,0), #7: (1, −1), and #8: (2.5, −1). Consider the following Poisson BVP for this domain:

$$\begin{cases}(\text{PDE}) & -\Delta u = f(x,y) & \text{on } \Omega \\ (\text{BC}) & u = 1 & \text{on } \partial\Omega\text{'} \end{cases}$$

where the "load" $f(x,y)$, is given by:

$$f(x,y) = \begin{cases} 0, & \text{if } x \le 2.5, \\ -1, & \text{if } x > 2.5. \end{cases}$$

Using the triangulation of Figure 13.5 and the corresponding piecewise linear basis functions of the last section, apply the FEM to solve this BVP.

SOLUTION: In this problem the BC is purely Dirichlet, so we may follow the above procedure.

The numbering of the nodes in Figure 13.5 has one drawback in that it does not conform to our current notation where the interior nodes are numbered first. We could redo the numbering to conform but instead will work around the numbering that was already set up. The corresponding matrices *N*(odes) and *E*(lements) are reproduced here:

$$N = \begin{bmatrix} 1 & 1 \\ 2.5 & 1 \\ 0 & 0 \\ 1 & 0 \\ 2.5 & 0 \\ 3.5 & 0 \\ 1 & -1 \\ 2.5 & -1 \end{bmatrix}, \quad E = \begin{bmatrix} 1 & 3 & 4 \\ 1 & 2 & 4 \\ 2 & 4 & 5 \\ 2 & 5 & 6 \\ 3 & 4 & 7 \\ 4 & 5 & 7 \\ 5 & 7 & 8 \\ 5 & 6 & 8 \end{bmatrix}.$$

Keep in mind that there are $m = 8$ nodes here of which $n = 2$ are interior (nodes #4 and #5). Thus an admissible function (for the FEM) $v = \sum_{i=1}^{8} c_i \Phi_i$ will have all but two of the coefficients (c_4, c_5) determined by the Dirichlet boundary conditions. Since (in the notation of (10)) $g(x,y) = 1$, we have that $c_i = g(N_i) = 1$ for $i \neq 4,5$, and so the FEM solution will have form: $v = c_4\Phi_4 + c_5\Phi_5 + \sum_{s \neq 4,5} \Phi_s$ and the rest of the problem is to compute these remaining two coefficients.

We are now at the assembly stage of the FEM. Note that since (in the notation of (10)), $p = 1$ and $q = 0$, and $c_s = g(N_s) = 1$ ($s \neq 4,5$), equations (15^{ℓ}) and (16^{ℓ}) simplify to:

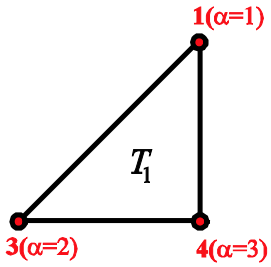
$$a_{\alpha\beta}^{\ell} = \iint_{T_\ell} \nabla\Phi_{i_\alpha} \bullet \nabla\Phi_{i_\beta}\, dxdy \quad (1 \le \alpha, \beta \le 3),$$

and

$$b_{\alpha}^{\ell} = \iint_{T_\ell} f\Phi_{i_\alpha}\, dxdy - \sum_{s \ne 4,5} \iint_{T_\ell} \nabla\Phi_s \bullet \nabla\Phi_{i_\alpha}\, dxdy \quad (1 \le \alpha \le 3),$$

respectively (we have incorporated the change needed to accommodate the node numbering scheme).

We initialize a 2×2 stiffness matrix A of zeros and the corresponding 2×1 initial load vector b and pass now to a detailed calculation of the first iteration of the assembly loop: $\ell = 1$ corresponding to the first element T_1 of Figure 13.5. Figure 13.31 shows this element and its corresponding element stiffness matrix A^1.



$$A^1 = \begin{bmatrix} a_{11}^1 & a_{12}^1 & a_{13}^1 \\ a_{21}^1 & a_{22}^1 & a_{23}^1 \\ a_{31}^1 & a_{32}^1 & a_{33}^1 \end{bmatrix} \begin{matrix} \leftarrow N_1 \\ \leftarrow N_3 \\ \leftarrow N_4 \end{matrix}$$
$$\begin{matrix} \uparrow & \uparrow & \uparrow \\ N_1 & N_3 & N_4 \end{matrix}$$

FIGURE 13.31: (a) (left) Illustration of the first element T_1 of Figure 13.5 with the global node numbers (from Figure 13.5) as well as the local node numbers from the matrix T. (b) (right) The corresponding element stiffness matrix A^1 along with a labeling of the corresponding nodes.

Of the nodes for T_1, only node #4 ($\alpha = 3$) is an interior node so we need only compute the single entry:

$$a_{33}^1 = \iint_{T_1} \nabla\Phi_4 \bullet \nabla\Phi_4 dxdy.$$

From the formula obtained in Example 13.1 for Φ_4, we know that on T_1, $\Phi_4(x, y) = x - y$, so that $\nabla\Phi_4 = (1, -1)$ (this also follows from the preceding exercise for the reader), and $\nabla\Phi_4 \bullet \nabla\Phi_4 = 2$. Consequently,

$$a_{33}^1 = \iint_{T_1} \nabla\Phi_4 \bullet \nabla\Phi_4 dxdy = \iint_{T_1} 2\, dxdy = 2 \cdot \text{Area}(T_1) = 2 \cdot (1/2) = 1.$$

Similarly, we have only to compute the single load entry:

$$b_3^1 = \iint_{T_1} f\Phi_4\, dxdy - \sum_{s=1,3} \iint_{T_1} \nabla\Phi_s \bullet \nabla\Phi_4 dxdy.$$

Since the load $f(x,y)$ vanishes throughout T^1, only the latter two integrals need to be computed. Both integrands are constants and so the integrals can be simply evaluated as the preceding one. We need the gradients of Φ_1 and of Φ_3 on T^1. Using part (a) of the preceding exercise for the reader, we compute $\nabla\Phi_1 = (0,1)$ and $\nabla\Phi_3 = (-1,0)$ and so the corresponding dot products with $\nabla\Phi_4 = (1,-1)$ are both -1. Hence,

$$b_3^1 = -\iint_{T_1} \nabla\Phi_1 \bullet \nabla\Phi_4 dxdy - \iint_{T_1} \nabla\Phi_3 \bullet \nabla\Phi_4 dxdy = -\big(-\text{Area}(T_1) - \text{Area}(T_1)\big) = 1.$$

The just-computed entries $a_{33}^1 = 1$, $b_3^1 = 1$ need to be transplanted to update the appropriate entries of the stiffness matrix and load vector *b*:

$$A = \begin{matrix} \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{matrix} \leftarrow N_4 \\ \leftarrow N_5 \end{matrix} \\ \begin{matrix} \uparrow & \uparrow \\ N_4 & N_5 \end{matrix} \end{matrix} \qquad b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \begin{matrix} \leftarrow N_4 \\ \leftarrow N_5 \end{matrix}$$

Since the local index $\alpha = 3$ corresponds to the internal node N_4, the corresponding index for the (global) stiffness matrix and load vector is 1, and we update: $a_{11} = a_{11} + a_{33}^1 = 0 + 1 = 1$, and $b_1 = b_1 + b_3^1 = 0 + 1 = 1$. In summary, after the first iteration of the assembly process ($\ell = 1$), our updated stiffness matrix and load vector are as follows:

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

The treatment for the next iteration $\ell = 2$ is quite similar since the element T_2 also has one interior node (#4) and two boundary nodes (#1, #2). To prepare for the computations, we note that $\text{Area}(T_2) = 3/4$ and on T_2:

$$\nabla\Phi_1 = (-2/3, 1), \quad \nabla\Phi_2 = (2/3, 0), \quad \nabla\Phi_4 = (0, -1).$$

We have used Exercise for the Reader 13.5. Actually, with less work, the needed gradient vectors here and in all other computations of this example can be gleaned from the explicit formula for Φ_4 obtained in Example 13.1 by comparing relevant triangles.

From the second row of the element matrix *E*, we see that the three vertices of T_2, nodes #1, #2, and #4, have local node numbers $\alpha = 1,\ 2,\ \text{and}\ 3,$ respectively, so that the node correspondence for the element stiffness matrix A^2 is as follows:

$$A^2 = \begin{bmatrix} a_{11}^2 & a_{12}^2 & a_{13}^2 \\ a_{21}^2 & a_{22}^2 & a_{23}^2 \\ a_{31}^2 & a_{32}^2 & a_{33}^2 \end{bmatrix} \begin{matrix} \leftarrow N_1 \\ \leftarrow N_2 \\ \leftarrow N_4 \end{matrix}$$
$$\begin{matrix} \uparrow & \uparrow & \uparrow \\ N_1 & N_2 & N_4 \end{matrix}$$

Since only node N_4 is internal, we need only compute the entry a_{33}^2 and the corresponding element load vector entry b_3^2, and since $f(x, y)$ again vanishes on T_2, these computations can be carried out just as before, using the above gradients and area:

$$a_{33}^2 = \iint_{T_2} \nabla\Phi_4 \bullet \nabla\Phi_4 \, dxdy = \iint_{T_2} 1 \, dxdy = \text{Area}(\text{T}_2) = 3/4,$$

$$b_3^2 = -\iint_{T_2} \nabla\Phi_1 \bullet \nabla\Phi_4 \, dxdy - \iint_{T_2} \nabla\Phi_2 \bullet \nabla\Phi_4 \, dxdy = -\big(-\text{Area}(T_2) + 0\big) = 3/4.$$

Transplanting these results into the appropriate places in the stiffness matrix and load vector results in the following updates:

$$A = \begin{bmatrix} 1+3/4 & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} 7/4 & 0 \\ 0 & 0 \end{bmatrix}, \quad b = \begin{bmatrix} 1+3/4 \\ 0 \end{bmatrix} = \begin{bmatrix} 7/4 \\ 0 \end{bmatrix}.$$

Proceeding now to $\ell = 3,$ the situation is a bit different in that the element T_3 has two internal nodes. This will mean that we will need to compute a total of six entries (four for the element stiffness matrix A^3 and two for the corresponding element load vector b^3). We obtain, as before, the area $\text{Area}(T_3) = 3/4,$ and the gradient vectors on T_3,

$$\nabla\Phi_2 = (0, 1), \quad \nabla\Phi_4 = (-2/3, 0), \quad \nabla\Phi_5 = (2/3, -1).$$

From the third row of the element matrix *E*, we see that the three vertices of T_3: nodes #2, #4, and #5 have local node numbers $\alpha = 1,\ 2,\ \text{and}\ 3,$ respectively, so that the node correspondence for the element stiffness matrix A^3 is as follows:

$$A^3 = \begin{bmatrix} a_{11}^3 & a_{12}^3 & a_{13}^3 \\ a_{21}^3 & a_{22}^3 & a_{23}^3 \\ a_{31}^3 & a_{32}^3 & a_{33}^3 \end{bmatrix} \begin{matrix} \leftarrow N_2 \\ \leftarrow N_4 \\ \leftarrow N_5 \end{matrix}$$
$$\begin{matrix} \uparrow & \uparrow & \uparrow \\ N_2 & N_4 & N_5 \end{matrix}$$

The computations of the needed entries of A^3 and b^3 are now done just as before. We briefly summarize them:

$$a_{22}^3 = \iint_{T_3} \nabla\Phi_4 \bullet \nabla\Phi_4 \, dxdy = \frac{4}{9} \cdot \frac{3}{4} = 1/3, \quad a_{23}^3 = a_{32}^3 = \iint_{T_3} \nabla\Phi_4 \bullet \nabla\Phi_5 \, dxdy = -\frac{4}{9} \cdot \frac{3}{4} = -1/3,$$

$$a_{33}^3 = \iint_{T_3} \nabla\Phi_5 \bullet \nabla\Phi_5 \, dxdy = \frac{13}{9} \cdot \frac{3}{4} = 13/12, \quad b_2^3 = -\iint_{T_3} \nabla\Phi_2 \bullet \nabla\Phi_4 \, dxdy = 0,$$

and $b_3^3 = -\iint_{T_3} \nabla\Phi_2 \bullet \nabla\Phi_5 \, dxdy = -\big(-\text{Area}(T_3)\big) = 3/4.$

Transplanting these results into the appropriate places in the stiffness matrix and load vector results in the following updates:

$$A = \begin{bmatrix} 7/4 + 1/3 & 0 - 1/3 \\ 0 - 1/3 & 0 + 13/12 \end{bmatrix} = \begin{bmatrix} 25/12 & -1/3 \\ -1/3 & 13/12 \end{bmatrix}, \quad b = \begin{bmatrix} 7/4 + 0 \\ 0 + 3/4 \end{bmatrix} = \begin{bmatrix} 7/4 \\ 3/4 \end{bmatrix}.$$

In the next iteration, $\ell = 4$ and $f(x, y)$ no longer vanishes on the element. Since $f(x, y)$ is constant throughout T_4, however, we will still be able to use Exercise for the Reader 13.5 to evaluate the new integral that arises. The nodes of T_4: N_2, N_5, N_6 have local node numbers (from the fourth row of E) $\alpha = 1, 2, 3$, respectively. The needed element area is $\text{Area}(T_4) = 1/2,$ and the gradient vectors on T_4:

$$\nabla\Phi_2 = (0, 1), \quad \nabla\Phi_5 = (-1, -1), \quad \nabla\Phi_6 = (1, 0).$$

As only one of the nodes is internal, we have only two entries to compute:

$$a_{22}^4 = \iint_{T_4} \nabla\Phi_5 \bullet \nabla\Phi_5 \, dxdy = \iint_{T_2} 2 \, dxdy = 1, \quad \text{and}$$

$$b_2^4 = \iint_{T_4} f\Phi_5 \, dxdy - \iint_{T_4} \nabla\Phi_2 \bullet \nabla\Phi_5 dxdy - \iint_{T_4} \nabla\Phi_6 \bullet \nabla\Phi_5 dxdy$$
$$= -\tfrac{1}{3}\text{Area}(T_4) - (-\text{Area}(T_4) - \text{Area}(T_4)) = 5/6.$$

(In the last calculation we use Exercise for the Reader 13.5(b).) The updated stiffness matrix and load vectors now become:

$$A=\begin{bmatrix}25/12 & -1/3\\ -1/3 & 13/12+1\end{bmatrix}=\begin{bmatrix}25/12 & -1/3\\ -1/3 & 25/12\end{bmatrix},\quad b=\begin{bmatrix}7/4\\ 3/4+5/6\end{bmatrix}=\begin{bmatrix}7/4\\ 19/12\end{bmatrix}.$$

Each of the remaining four iterations is done almost identically to one of the four that has just been done. We summarize each remaining iteration only by the stiffness matrix and load vector updates:

$$\ell=5:\quad A=\begin{bmatrix}25/12+1 & -1/3\\ -1/3 & 25/12\end{bmatrix}=\begin{bmatrix}37/12 & -1/3\\ -1/3 & 25/12\end{bmatrix},\quad b=\begin{bmatrix}7/4+1\\ 19/12\end{bmatrix}=\begin{bmatrix}11/4\\ 19/12\end{bmatrix}$$

$\ell=6:$

$$A=\begin{bmatrix}37/12+13/12 & -1/3-1/3\\ -1/3-1/3 & 25/12+1/3\end{bmatrix}=\begin{bmatrix}25/6 & -2/3\\ -2/3 & 29/12\end{bmatrix},\ b=\begin{bmatrix}11/4+3/4\\ 19/12\end{bmatrix}=\begin{bmatrix}15/4\\ 19/12\end{bmatrix}$$

$$\ell=7:\quad A=\begin{bmatrix}25/6 & -2/3\\ -2/3 & 29/12+3/4\end{bmatrix}=\begin{bmatrix}25/6 & -2/3\\ -2/3 & 19/6\end{bmatrix},\quad b=\begin{bmatrix}15/4\\ 19/12+3/4\end{bmatrix}=\begin{bmatrix}15/4\\ 7/3\end{bmatrix}$$

and finally,

$$\ell=8:\quad A=\begin{bmatrix}25/6 & -2/3\\ -2/3 & 19/6+1\end{bmatrix}=\begin{bmatrix}25/6 & -2/3\\ -2/3 & 25/6\end{bmatrix},\quad b=\begin{bmatrix}15/4\\ 7/3+5/6\end{bmatrix}=\begin{bmatrix}15/4\\ 19/6\end{bmatrix}.$$

With the stiffness matrix and load vector now "assembled," the remaining coefficients c_4, c_5 are simply the solutions of the linear system:

$$Ac=b \Leftrightarrow \begin{bmatrix}25/6 & -2/3\\ -2/3 & 25/6\end{bmatrix}\begin{bmatrix}c_4\\ c_5\end{bmatrix}=\begin{bmatrix}15/4\\ 19/6\end{bmatrix}\quad\Rightarrow\quad\begin{bmatrix}c_4\\ c_5\end{bmatrix}=\begin{bmatrix}1277/1218\\ 565/609\end{bmatrix}.$$

With this small system (solved on MATLAB) exact arithmetic was feasible. The FEM solution $v=c_4\Phi_4+c_5\Phi_5$ can now be plotted quite easily using the `trimesh` command as in the last section. We need to make sure we have the node matrix *N* and the element matrix *E* stored, and then assign the values for c_4, c_5 to nodes #4, #5 and values of one for the remaining nodes (from the Dirichlet BCs):

```
>> N=[1 1;5/2 1;0 0;1 0;5/2 0;7/2 0;1 -1;2.5 -1];
>> E=[1 3 4;1 2 4;2 4 5;2 5 6;3 4 7;4 5 7;5 7 8;5 6 8];
>> x=N(:,1); y=N(:,2);
>> z=ones(8,1); z(4)= 1277/1218; z(5)= 565/609;
>> trimesh(E,x,y,z)
>> hidden off, xlabel('x-values'), ylabel('y-values')
```

The resulting plot is shown in Figure 13.32.

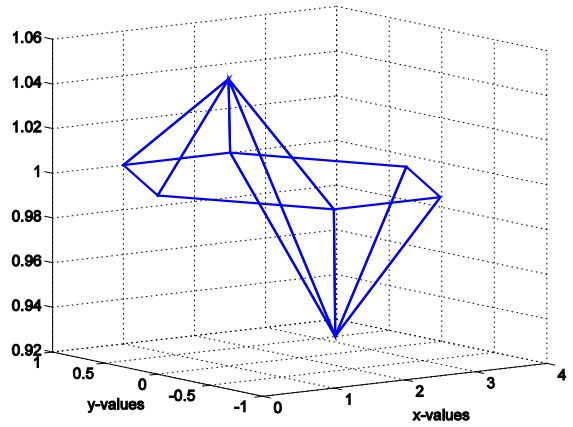


FIGURE 13.32: Plot of our first FEM solution to the BVP of Example 13.5. Only 8 elements and 2 internal nodes were used, so the plot is rather coarse.

EXERCISE FOR THE READER 13.6: If in the BVP of Example 13.5 we change the BC to $u \equiv 2$ on $\partial\Omega$, but leave all else the same, how would the exact solution of this modified problem compare with that of the original? Perform the FEM on this modified problem (with the same triangulation) and compare the numerical solution with that of the original problem.

The resolution used in the last example was made deliberately coarse so that we could focus on the various facets of the FEM. We now move on to apply the FEM to a problem with a much more elaborate triangulation of the domain. The added complexity will force us to write some MATLAB loops to make the FEM feasible. The BVP we choose, the Laplace equation with Dirichlet boundary conditions on the unit disk, is rather special in that an explicit solution is available. We will thus be able to compare our FEM solution with the exact solution. Such examples are important as an aid for creating and testing production-level FEM codes. We state as a theorem this beautifully explicit result due to Poisson.[11]

[11] After his secondary education, Siméon-Denis Poisson went to work as a surgeon's apprentice with an uncle in Fontainbleau, a small city not far from Paris. His lack of coordination forced him to abandon his pursuit of this profession and he subsequently went to the local École Central for undergraduate studies in search of a new career. His mathematical ability was noticed by his instructors who encouraged him to take the entrance exams at the premiere École Polytechnique in Paris. Despite his relatively minor training, he placed at the very top and was admitted in 1798. His talents were quickly noticed and further cultivated by his teachers Laplace and Lagrange. Although his lack of manual dexterity precluded him from doing well in certain subjects (such as descriptive geometry), he excelled in subjects where drawing diagrams was not needed and at age 18 wrote a seminal memoir on finite differences which was well received. After graduation from École Polytechnique he was offered a position there, a rare honor which he accepted. He spent the remainder of his career there and led a very productive life of contributions both to mathematics and physics. He cared deeply for mathematics and for maintaining the quality and sanctity of the École Polytechnique. He was able to stop a group of politically active students at the École from publishing a lampooning attack on Napoleon's leadership, fearing that this could do harm to the École. He was elected to the physics section of the prestigious national Institute (a corresponding position in the mathematics section was

Figure 13.33: Siméon-Denis Poisson (1781–1840), French mathematician.

THEOREM 13.1: (*Poisson's Integral Formula*) Suppose that $f(\theta)$ is a continuous function (given in polar coordinates) on the circle $x^2 + y^2 = R^2$ (θ is the polar coordinate angle). If Ω is the disk inside this circle, $\Omega = \left\{ p = (x, y) \in \mathbb{R}^2 : \|p\|_2 < R \right\}$, then the solution of the Dirichlet problem:

$$\begin{cases} (\text{PDE}) & \Delta u = 0 & \text{on } \Omega \\ (\text{BC}) & u(R, \theta) = g(\theta) & \text{on } \partial\Omega \end{cases} \tag{19}$$

is unique and is given by:

$$u(r, \theta) = \frac{R^2 - r^2}{2\pi} \int_0^{2\pi} \frac{g(\phi) d\phi}{R^2 - 2Rr\cos(\theta - \phi) + r^2} \tag{20}$$

Here, (r, θ) denotes the polar coordinates of any point inside Ω, ($r < R$).

We omit the proof of this result (an enlightening one can be found in Section 4.6 in the textbook [Ahl-79]). The result and proof actually extends to higher dimensions; see Section 7.5 in [Zau-89] for the three-dimensional analogue. It turns out as well that the result remains valid for more general boundary data $f(\theta)$. For example, if $f(\theta)$ is only piecewise continuous, then (20) will still solve the Dirichlet problem (19), and the solution will be continuous at all points on $\Omega \cup \partial\Omega = \left\{ p = (x, y) \in \mathbb{R}^2 : \|p\|_2 \le R \right\}$ except at those points on the boundary at which $f(\theta)$ is discontinuous (see again Section 4.6 in the textbook [Ahl-79]). This beautiful formula is one very rare instance where a general BVP has an explicit and practical solution. Recall that solutions of the Laplace PDE in (19) are called harmonic functions (Chapter 11). The BVP (19) can be viewed, for example, as finding the steady-state heat distribution of a circular plate whose temperature on the boundary is maintained with a certain known distribution ($f(\theta)$).

---

not available; due to the limit set on membership a death of a member had to occur for a new slot to open). His name permeates many areas of mathematics and physics, which apart from differential equations (Poisson bracket and integral formulas), include probability (Poisson distribution), harmonic analysis (Poisson summation formula), and elasticity (Poisson's ratio). During his career he wrote over 300 research papers, but he was known never to work on more than one project at a time. He was extremely methodical and well-organized; if an idea for a new project would cross his mind while working on one paper he would write a brief note about it and place it in his wallet. After finishing one paper, he would then pull out all of the notes from his wallet to decide on the best topic for his next project.

We will be able to use (20) to get MATLAB to run through a sufficiently fine set of nodes in the disk to obtain a plot of the exact solution. The nodes could be chosen to be those used in a FEM approximation so that the errors of the FEM solution could be examined. All of this will be done in Example 13.7.[12]

Example 13.5 was intentionally set up so as to avoid the problem of having to numerically integrate functions of two variables. In more general examples, we will need to show how to deal with such integrals. MATLAB has an integrator to perform double integrals in floating point arithmetic. Such integrals can be time consuming depending on the oscillatory behavior of the integrand. Triangulations can be made finer in parts of the domain where the data functions have larger variations, and thus the integrals become less difficult to evaluate numerically. In practice, however, rather than using general integration programs or symbolic integrators, well-known quadrature approximations are employed. Such approximation schemes take advantage of the special structure of elements to approximate an integral over an element by a certain weighted average among certain special points of the element. We will present both approaches below. The first method will be to use MATLAB's numerical integrator. To facilitate general codes, we will appeal to some of the Symbolic Toolbox capabilities. The second method will utilize special quadrature formulas. The performance accuracy and times of both approaches will be compared and contrasted with an example where the exact solution can be obtained (and in which the FEM integrals will be quite simple). After presenting both methods, we will discuss some of the underlying theory. Particular readers may wish to cover only one method. Readers who do not have access to or wish to avoid using the Symbolic Toolbox may wish either to skip Method 1, or to be prepared to recode those parts of it which appeal to symbolic functionality. In our numerical example (as we will see below), Method 2 ran about 200 times faster than Method 1 and gave the same quality of results. Such results are typical and this is why we recommend Method

---

[12] As an aside, we point out here some related facts. A celebrated result in the theory of complex variables (which can be found in [Ahl-79], the classic treatise on the subject) known as the Riemann mapping theorem, states that any simply connected planar domain $D \subset_{\neq} \mathbb{R}^2$ can be mapped conformally onto the unit disk $U = \left\{ p = (x, y) \in \mathbb{R}^2 : \|p\|_2 < 1 \right\}$. Simply connected means roughly that the domain has no holes inside, i.e., if γ is any closed path in the domain, then the interior of γ contains only points in the domain; see [Ahl-79] for more details. A conformal mapping is a one-to-one function (of two variables) F such that $F(D) = U$. Conformal mappings have the property that they preserve angles and have many beautiful properties (see [Ahl-79]). One particularly useful property of conformal mappings is that they preserve harmonic mappings, i.e., if $u(x, y)$ is a harmonic function on the domain U and $F : D \to U$ is a conformal mapping, then $v = u(F(x, y))$ is a harmonic function on D. This result means that for any simply connected domain in the plane, there is a corresponding Poisson integral formula for solutions of the Dirichlet problem gotten by changing variables to the disk. This is quite a satisfying and complete result, theoretically, at least. The practical problem for a given simply connected domain thus reduces to computing explicity a function which conformally maps it to the disk U. This problem has been extensively studied and there are many situations where the mappings have been found. This approach has led to numerous applications to physical BVPs involving the Laplace equation, including also steady-state fluid flow, and electrostatics. See [BrChSi-03] for more on conformal mapping with an emphasis on applications.

2. We include Method 1 only for comparison purposes; for readers interested in practical codes, it may be skipped altogether.

NUMERICAL APPROXIMATION TO DOUBLE INTEGRALS— METHOD 1: USING MATLAB's NUMERICAL INTEGRATOR `dblquad`:

MATLAB's numerical integrator for double integrals has a syntax that requires the integration to be performed over a rectangle. We explain its functionality below and then show how it can be adapted to perform integrations over more general regions.

`dblquad(fun,xmin,xmax, ymin,ymax)`→	Assume that `fun` is an inline function of x and y.[13] This command will numerically compute the integral $\int_{x\,\min}^{x\,\max}\int_{y\,\min}^{y\,\max} \text{fun}(x,y)dydx,$ using a double iteration with the single variable function integrator `quad` and with a default tolerance for error being 1e-6. As with `quad`, the syntax of `dblquad` requires that we make the integrand `fun(x,y)` able to input a vector argument for (the first variable) `x` and return a vector of the same size.
`dblquad(fun,xmin,xmax, ymin,ymax, tol,@quadl,p1,p2,...)` →	Optional extra inputs: `tol` allows specification of an error tolerance, `@quadl` specifies that the more refined `quadl` integrator be used in the iterations, the last inputs `p1`, `p2`, … represent numerical values to assign in case `fun` depends on additional parameters: `fun = fun(x,y,p1,p2,…)`.

The following simple example will illustrate the syntax requirement on `fun`:

To evaluate the integral of x^2y^2 over the rectangle $R=[0,2]\times[1,2]$:

$$\int_R x^2y^2\,dxdy = \int_0^2\int_1^2 x^2y^2\,dydx,$$

we could simply enter:

```
>> dblquad(inline('x.^2.*y.^2','x', 'y'), 0, 2, 1, 2)   →ans = 6.2222
```

The vector syntax requirement on the first variable `x` is automatically satisfied since this variable appears in the single term for the integrand. If, however, we wanted (for testing purposes) to compute the area of the rectangle R, the corresponding command:

```
>> dblquad(inline('1','x', 'y'), 0, 2, 1, 2)
```

[13] As usual, if instead "fun" has been stored as an M-file, it should be written with single quotes: `dblquad('fun',...)` or preceded with the "@" symbol: `dblquad(@fun,...)`.

gives a series of error messages:

```
??? Index exceeds matrix dimensions.

Error in ==> C:\MATLAB6p5\toolbox\matlab\funfun\quad.m
On line 67  ==> if ~isfinite(y(7))
(more…)
```

The syntax can be adjusted accordingly as follows:

```
>> dblquad(inline('1*ones(size(x))','x', 'y'), 0, 2, 1, 2)
→ans = 2
```

which (as we know) gives the correct answer. A similar syntax note was pointed out in Chapter 3 for `quad`.

In order to use `dblquad` to integrate over regions other than rectangles the following identity will be useful:

$$\begin{aligned}\int_T \text{fun}(x,y)\,dxdy &\equiv \int_{\text{xmin}}^{\text{xmax}} \int_{\text{ylow}(x)}^{\text{ytop}(x)} \text{fun}(x,y)\,dydx \\ &= \int_{\text{xmin}}^{\text{xmax}} \int_{0}^{1} \text{fun}(x, \text{ylow}(x) + u(\text{ytop}(x) - \text{ylow}(x)))[\text{ytop}(x) - \text{ylow}(x)]\,dudx,\end{aligned} \tag{21}$$

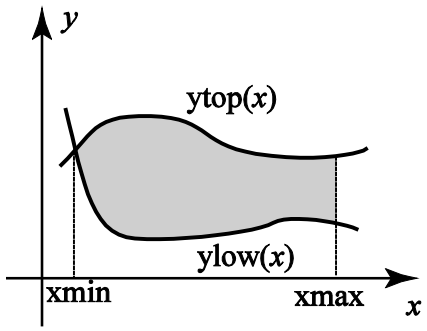


FIGURE 13.34: Illustration of a typical planar region on which integrals can be computed using (21).

Here, the region T need not be a triangle, but rather any region in the plane bounded below by the curve $\text{ylow}(x)$ and above by the curve $\text{ytop}(x)$ and over the range [xmin, xmax]; see Figure 13.34.

The identity (21) is easily established by a simple variable substitution; see Exercise 20. Using this identity, we may use `dblquad` to compute any double integral. Since all of our integrals in the text proper of this section will be over triangles, the next example will present some more or less typical evaluations of double integrals over triangles.

EXAMPLE 13.6: Let the triangle T of Figure 13.30 have the following vertices: $v_1 = (1,3)$, $v_2 = (5,1)$, and $v_3 = (4,6)$. Use MATLAB's `dblquad` to numerically compute the following integrals:

(a) $\int_T 2xy^2 dxdy$

(b) $\int_T \sin(xy\sqrt{y})dxdy$

In each, decrease the tolerance or change to `quadl`, as needed, until the answers agree to four decimals.

SOLUTION: We need first to express the integrals as double integrals. Letting x be the outer integration variable, the x-range of T is $1 \le x \le 5$. Over this range, the lower function `ylow` of x will be the line segment from v_1 to v_2 (see Figure 13.30). Writing this line segment as a function of x yields: $\text{ylow}(x) = -\frac{1}{2}x + \frac{7}{2}$. The corresponding upper function `ytop` of x splits up into two formulas determined by the two segments $v_1\ v_3$ and $v_3\ v_2$. Writing each of these segments as a function of x yields the following formula for `ytop`: $\text{ytop}(x) = \begin{cases} x+2, & \text{if } x \le 4 \\ -5x+26, & \text{if } x > 4 \end{cases}$.

Part (a): Using the above functions, we can rewrite the integral in the following iterated form:

$$\int_T 2xy^2 dxdy = \int_1^5 \int_{\text{ylow}(x)}^{\text{ytop}(x)} 2xy^2 dydx = \int_1^4 \int_{\text{ylow}(x)}^{x+2} 2xy^2 dydx + \int_4^5 \int_{\text{ylow}(x)}^{26-5x} 2xy^2 dydx.$$

The latter form is a more convenient one to implement on MATLAB. The code given below is written in a way that will make it easy to adapt to handle the general computation of such integrals and to this end it is more convenient to use some Symbolic Toolbox capabilities.

```
>> syms x y u
>> ylow = -.5*x+3.5; ytop1=x+2; ytop2=-5*x+26;
>> fun=2*x*y^2;
>> ynew1=ylow+u*(ytop1-ylow);
>> funprep1=subs(fun,y,ynew1)*(ytop1-ylow);
>> ynew2=ylow+u*(ytop2-ylow);
>> funprep2=subs(fun,y,ynew2)*(ytop2-ylow);
>> funnew1=vectorize(inline([char(funprep1),'*ones(size(u))'],...
                                        'u','x'));
>> %we needed to convert the symbolic expression back into a
>> %character string for construction of an inline function.
>> funnew2=vectorize(inline([char(funprep2),'*ones(size(u))'],...
                                        'u','x'));
>> dblquad(funnew1,0,1,1,4)+ dblquad(funnew2,0,1,4,5)→ans = 724.8000
```

Using a smaller tolerance (than the default 10^{-6}) gives the same result:

```
>> dblquad(funnew1,0,1,1,4,1e-7)+ dblquad(funnew2,0,1,4,5,1e-7)
→ans =724.8000
```

Part (b) Implementing the same strategy, we obtain:

```
>> fun=sin(sin(x)*y);
>> funprep1=subs(fun,y,ynew1)*(ytop1-ylow);
>> funprep2=subs(fun,y,ynew2)*(ytop2-ylow);
>>
funnew1=vectorize(inline([char(funprep1),'*ones(size(u))'],'u','x'))
>>
funnew2=vectorize(inline([char(funprep2),'*ones(size(u))'],'u','x'))
>> dblquad(funnew1,0,1,1,4)+ dblquad(funnew2,0,1,4,5)
→ ans = 0.1397
```

There is agreement when we reduce the tolerance as above.

The numerical integration(s) of part (b), unlike that in part (a), took a noticeable amount of time. This is due to the fact that the integrand in part (b) is very oscillatory over the domain. In general, double integrals can take a lot of work to evaluate effectively since, if the integrals cannot be done explicitly, any method basically has to iterate evaluations of a one-variable integral on numerous slices (the number goes up when more accuracy is desired). When performing the FEM to solve a given BVP, the triangulation can and should be done so as to use smaller elements in areas of high oscillation of the given data. This will assure that the integrals that arise in the assembly process will be numerically quite tame and easy to compute. MATLAB's symbolic integrator `int` can also be used to evaluate double integrals, and although the syntax is a bit simpler than for `dblquad`, the M-files we introduce below will help to make `dblquad` more convenient to use. Also, the extra computing time needed for `int` to attempt to find exact antiderivatives, which is usually not possible in general, is not worth the occasional extra precision in the answers.

To save on having to go through the above complicated syntax each time a numerical integral is encountered, we give here an M-file that is essentially a user-friendly version of `dblquad`. It is a simple modification of the code employed in the last example.

PROGRAM 13.1: User-friendly M-file for numerically computing double integrals over planar regions bounded between two functions of *x*, as in Figure 13.34. Integrand `fun` is entered as a function of the symbolic variables *x* and *y*.

```
function nint= quad2d(fun,xmin,xmax,ylow,ytop)
% numerically computes a double integral of a function 'fun' on a
% region over the interval minx<x<maxx and between the functions of
% x: ylow<y<ytop.
% INPUTS:  fun = a function of the symbolic variables x and y
% minx = minimum x-value for region
% maxx = maximum x-value for region
% ylow = function of symbolic variable for lower boundary of region
% ytop = function of symbolic variable for upper boundary of region
```

```
% OUTPUT:  nint = numerical approximation of integral using the
% integrator 'dblquad' in conjunction with the default settings.
% x and y should be declared symbolic variables before this M-file is
% used.
syms u x y
ynew=ylow+u*(ytop-ylow);
funprep=subs(fun,y,ynew)*(ytop-ylow);
funnew=vectorize(inline([char(funprep),'*ones(size(u))'],'u','x'));
nint = dblquad(funnew,0,1,xmin,xmax);
```

EXERCISE FOR THE READER 13.7: Use the above program to numerically compute the following double integrals:

(a) $\int_S xy^2 dxdy$, where S is the circular sector $\{(r,\theta): 0 \le r \le 1, 0 \le \theta \le \pi/4\}$.

(b) $\int_U \exp(1-x^2-2y^2) dxdy$, where U is the region enclosed between the curves $y = e^x$, $y = x^2 - 1$ and $y = 0$.

EXERCISE FOR THE READER 13.8: (a) Write an M-file, `integ= triangquad2d (fun,v1,v1,v3)` whose inputs are a function of x,y (written as a symbolic expression), and three 2×1 matrices `v1`, `v2`, `v3` which are vertices (in any order) of triangle in the xy-plane. If we denote this triangle by T, the output `integ` will be the numerical integral $\int_T \text{fun}(x,y)\, dxdy$, computed with `quad2d` as in Example 13.6.

(b) Use your function in Part (a) to reevaluate the integrals of Example 13.6, and also to compute the following integrals in which T_1 is the triangle with vertices (0,0), (6,0), (12,2), and T_2 is the triangle with vertices (1,3), (3,2), and (2,5).

(i) $\int_{T_1} 1\, dxdy = 6$, (ii) $\int_{T_2} 1\, dxdy = 5/2$, (iii) $\int_{T_1} 2x^2\, dxdy = 504$, and

(iv) $\int_{T_2} \sin(x^2)\, dxdy \approx -0.2998$

Suggestions: Branch your program off into two cases: Either the triangle has a vertical side or the three x-coordinates of the vertices are distinct. Draw lots of pictures of triangles as you are proceeding.

EXAMPLE 13.7: Consider the Dirichlet problem (19) on the unit disk $\Omega = \{(x,y): x^2 + y^2 < 1\}$,

$$\begin{cases} (\text{PDE}) & \Delta u = 0 & \text{on } \Omega \\ (\text{BC}) & u(1,\theta) = g(\theta) & \text{on } \partial\Omega \end{cases},$$

(we put $R = 1$ in (19)), where $g(\theta) = \begin{cases} 2\theta^2, & \text{if } 0 \le \theta \le 2 \\ 8, & \text{if } 2 < \theta \le 3 \\ 0, & \text{if } 3 < \theta \le 2\pi \end{cases}$.

(a) Use the FEM with a triangulation of the disk involving between 50 and 100 nodes deployed on circles of increasing radii but more or less uniformly (as in Method 2 of the solution to Example 13.2(a) of the last section) to solve this BVP and plot the FEM solution.
(b) Use the Poisson integral formula (20) to numerically compute the exact solution at each of the nodes in part (a), and plot it. Compare with the plot obtained in part (a), and compute the maximum error (at the interior nodes).
(c) Repeat both parts (a) and (b), this time using between 500 and 1000 nodes.

SOLUTION: Part (a): The triangulation can be done in exactly the same fashion as was done in Method 2 of part (a) of the solution of Example 13.2 (simply change the value of `delta=sqrt(pi/90);` everything else is the same). The code is thus omitted here; the nodes were stored in vectors `x` and `y` and the triangulation in the matrix `tri`. The triangulation is shown in Figure 13.35.

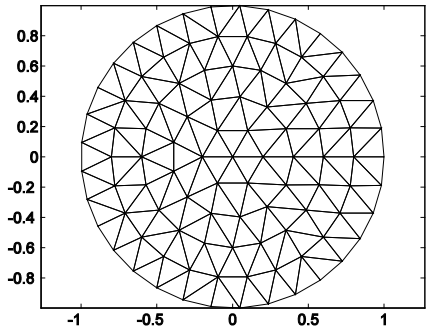


FIGURE 13.35: Triangulation for the FEM solution of Example 13.7(a). There are 99 nodes and 163 triangular elements.

By the way in which the nodes were created, the numbering scheme conforms to that of the procedure outline (the boundary nodes are indexed last). In the notation of the procedure, $m = 99$ (= total number of nodes), as seen by entering `size(x)`. We can use a simple MATLAB loop to compute n (= number of interior nodes):

```
>> n=1;
>> while x(n)^2+y(n)^2<1-eps
     n=n+1;
end
>> n=n-1  →n = 66
```

(Note: We used `eps` (= machine epsilon) to safeguard the inequality from roundoff errors.) Thus there are $n = 66$ interior nodes.

We now use the boundary data to assign the corresponding coefficients c_i ($i > n$) of the basis functions for thefor the FEM solution $v = \sum_{i=1}^{m} c_i \Phi_i$. To facilitate this,

we will create an M-file for the boundary data function $g(\theta)$. Since the function will eventually need to be integrated (in part (b) when we use the Poisson integral formula), and the function is defined by cases, we will implement the special vector construction for this M-file that was explained in Chapter 4:

```
function y = EX13_7_bdydata(x)
for i = 1:length(x)
if (0<=x(i))&(x(i)<=2)
     y(i)=2*x(i)^2;
elseif (2<x(i))&(x(i)<=3)
     y(i)=8;
else
     y(i)=0;
end
end
```

Now, since the boundary data function is a function of the angle θ, and the nodes are stored as ordered pairs of *xy*-coordinates, in order to use this function to assign the node coefficients, we must compute and input the corresponding angles for each node. MATLAB has the following built-in functions for such coordinate changes:

`[th,r]=cart2pol(x,y)` →	If `(x,y)` denote the cartesian coordinates of a point in the plane, the output `[th,r]` will be the corresponding polar coordinates, where the angle `th` is chosen in the interval $(-\pi,\pi]$, and the radius `r` is nonnegative.
`[x,y]=pol2cart(r,th)` →	Inputs a set of polar coordinates (`r`,`th`) and outputs the corresponding cartesian coordinates.

The following loop will now store the boundary node coefficients:

```
for i=67:99
  th=cart2pol(x(i),y(i));
  if th<0, th=th+2*pi; end
  %need to ensure th is in domain of boundary data function
  c(i)=EX13_7_bdydata(th);
end
```

We are now ready to move on to the assembly process. We first observe that since (in the notation of (10)), $q \equiv 0$, $f \equiv 0$ and $p \equiv 1$, equations (15^{ℓ}) and (16^{ℓ}) simplify to:

$$a_{\alpha\beta}^{\ell} = \iint_{T_{\ell}} \nabla\Phi_{i_\alpha} \bullet \nabla\Phi_{i_\beta}\, dxdy \quad (1 \le \alpha, \beta \le 3),$$

and

$$b_{\alpha}^{\ell} = -\sum_{s=n+1}^{m} c_s \iint_{T_{\ell}} \nabla\Phi_{s} \bullet \nabla\Phi_{i_\alpha}\, dxdy \quad (1 \le \alpha \le 3),$$

respectively. Also, of the 33 possible indices s in the b_{α}^{ℓ} formulas, only those (at most two) corresponding to boundary nodes of the element T_{ℓ} need to be

considered. Since each gradient appearing in the above integrals is of a linear function on an element, the integrands are all constants, and so the corresponding integrals will be simply the constant times the area of the underlying element. We will use the M-file of Exercise for the Reader 13.8 to evaluate each of these integrals (within a loop).

We will make use of MATLAB's `setdiff` built-in function, which was introduced in the last section, but with an optional second output variable.

`[d,ind] = setdiff(a,b)` →	The first output variable was explained in the last section. The optional second output variable will be the indices of `a` which produce the vector `d`.

Here is a brief usage example:

```
>> a = [1 2 3]; b = [2 4];
>> d = setdiff(a,b)    →d = 1    3
>> [d,ind] = setdiff(a,b);  ind   →ind =  1    3
>> a = [3 2 1];
>> [d,ind] = setdiff(a,b)   →d =  1    3,   ind = 3    1
```

As usual, we first initialize the $n \times n$ ($n = 66$) stiffness matrix A of zeros and the corresponding $n \times 1$ initial load vector b and create a program that will completely perform the assembly. Here is the complete code for the assembly process for Example 13.7.

```
>> N=[x' y'];
>> E=tri;
>> n=66; m=99; syms x y
>> A=zeros(n); b=zeros(n,1);
>> [L cL]=size(E);
>> for ell=1:L
  nodes=E(ell,:);
  intnodes=nodes(find(nodes<=n));
  bdynodes=nodes(find(nodes>n));
  %find gradients [a b] of local basis functions
  % ax + by +c; distinguish between int node
  %local basis functions and bdy node local basis
  %functions

  for i=1:length(intnodes)
   xyt=N(intnodes(i),:); %main node for local basis function
   onodes=setdiff(nodes,intnodes(i));
   %two other nodes (w/ zero values) for local basis function
   xyr=N(onodes(1),:);
   xys=N(onodes(2),:);
   M=[xyr 1;xys 1;xyt 1]; %matrix M of (4)
   abccoeff=[xyr(2)-xys(2); xys(1)-xyr(1); xyr(1)*xys(2)-...
   xys(1)*xyr(2)]/det(M);%coefficents of basis function on triangle#L
   %See formula (6a)
   intgrad(i,:)=abccoeff(1:2)';
  end

  for j=1:length(bdynodes)
   xyt=N(bdynodes(j),:); %main node for local basis function
```

```
   onodes=setdiff(nodes,bdynodes(j));%two other nodes
   % (w/ zero values) for local basis   function
   xyr=N(onodes(1),:);
   xys=N(onodes(2),:);
   M=[xyr 1;xys 1;xyt 1]; %matrix M of (4)
   abccoeff=[xyr(2)-xys(2); xys(1)-xyr(1); xyr(1)*xys(2)-...
   xys(1)*xyr(2)]/det(M); %coefficents of basis function on
triangle#L
   bdygrad(j,:)=abccoeff(1:2)';
  end

  %update stiffness matrix
  for i1=1:length(intnodes)
  for i2=1:length(intnodes)
   fun = sym(intgrad(i1,:)*intgrad(i2,:)'); %integrand for (15ell)
   integ=triangquad2d(fun,xyt,xyr,xys);
   A(intnodes(i1),intnodes(i2))=A(intnodes(i1),intnodes(i2))+integ;
  end
  end

  %update load vector
  for i=1:length(intnodes)
  for j=1:length(bdynodes)
   fun = sym(intgrad(i,:)*bdygrad(j,:)'); %integrand for part of
(16ell)
   integ=triangquad2d(fun,xyt,xyr,xys);
   b(intnodes(i))=b(intnodes(i))-c(bdynodes(j))*integ;
  end
  end
end
sol=A\b;
c(1:n)=sol';
```

The result is now easily plotted using the `trimesh` function of the last section:

```
>> x=N(:,1); y=N(:,2);
>> trimesh(E,x,y,c), xlabel('x-values'), ylabel('y-values')
>> hidden off
```

The resulting plot is shown in Figure 13.36.

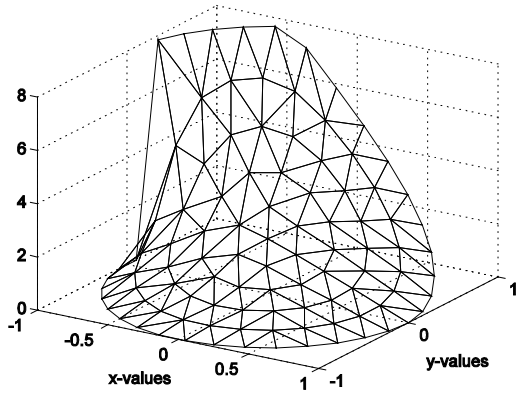


FIGURE 13.36: Plot of the FEM solution of the Dirichlet problem of Example 13.7.

Part (b): The following simple loop will implement the Poisson integral formula (20) to determine the value of the exact solution at each of the interior nodes c_i ($i \le n$). As has been the convention thus far, we continue to use MATLAB's numerical integrators for one-dimensional integrals; in this case we use `quadl`. We will leave the already assigned values at the boundary nodes c_i ($i > n$). We first create and store an M-file for the integrand in the Poisson integral formula (20) using the boundary data of the current example. Since this function will be integrated (with `quadl`) we will need to construct it as shown in Chapter 4 so that it will appropriately handle vector inputs.

```
function y = EX13_7_poisson(phi,r,th)
for i = 1:length(phi)
if (0<=phi(i))&(phi(i)<=2)
     y(i)=2*phi(i)^2*(1-r^2)/2/pi/(1-2*r*cos(th-phi(i))+r^2);
elseif (2< phi(i))&( phi(i)<=3)
     y(i)=8*(1-r^2)/2/pi/(1-2*r*cos(th-phi(i))+r^2);
else
     y(i)=0;
end
end
```

```
>> cp=c; %initialize node values for Poisson integral method.
>> for i=1:n
[th, r]=cart2pol(N(i,1),N(i,2)); %polar coors for node #i
cp(i)= quadl(@EX13_7_poisson,0,3,[],[],r,th);
%since integrand vanishes on (3, 2*pi] we can reduce the interval of
%integration.
end
```

The plot of the exact solution just obtained[14] will be quite similar to that of our FEM approximation in Figure 13.36. The resulting error plot is now easily obtained by the following commands, and the plot is shown in Figure 13.37.

```
>> trimesh(E,x,y,abs(c-cp))
>> hidden off
>> xlabel('x-values'), ylabel('y-values')
```

Part (c): The code in parts (a) and (b) is written in a way so that just one small change in one line of the code is required to do part (c). In the creation of the nodes (as in Method 2 of in the solution of Example 13.2(a)) we only need to change the paratmer δ to $\sqrt{\pi/900}$. The resulting node set contains m = 897 nodes of which the first n = 791 are interior nodes, and the Delaunay triangulation contains 1686 elements. The main loop took close to an hour on the author's computer. See Figure 13.38 for plots of the FEM solution and error.

---

[14] Of course, the Poisson integral formula, as mentioned, is exact. The only errors will be the errors that arise from the numerical integration. By default, the accuracy goal will have error < 1e-6, and the integrand is well-behaved so such errors will not be relevant for our present comparison purposes. In case they do become relevant (with a much finer mesh, say), we could always set a new accuracy goal for `quadl`.

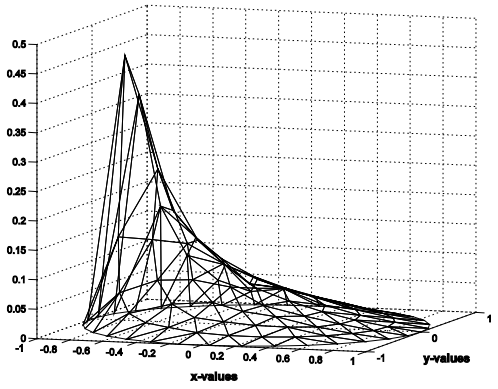


FIGURE 13.37: Plot of the error of the FEM solution of Example 13.7, obtained by comparing it to the exact solution over the same grid from the Poisson integral formula.[15]

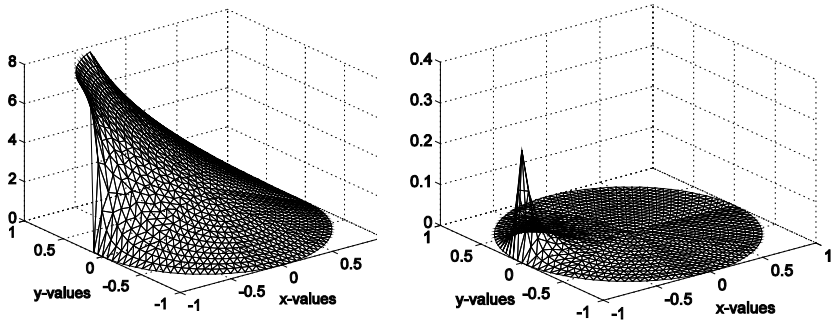


FIGURE 13.38: (a) (left) Plot of the FEM solution of the Dirichlet problem of Example 13.7(b). There are 897 nodes and 1686 triangular elements. (b) (right) Plot of the corresponding error. Notice that in the solution there is one distinguished element (near $(x, y) = (\cos(3), \sin(3))$) whose z range stretches all the way from 0 to the maximum value of 8. This is inevitable since the boundary data has a jump discontinuity from $z = 8$ to $z = 0$ at $(x, y) = (\cos(3), \sin(3))$. The error, although quite small over most of the domain, has a distinguished spike near $(x, y) = (\cos(3), \sin(3))$.

Looking at the errors of the last two solutions, we would be led to conjecture that better FEM solutions could be obtained (for the same numbers of nodes) if we were to concentrate more nodes near the boundary point $(x, y) = (\cos(3), \sin(3))$ at which there is a jump discontinuity. The next exercise for the reader will explore this. This example also motivates the concept of adaptive methods for FEM. One scheme for such a method begins with a more or less uniform node distribution

[15] More preciselely, this plot is the difference between the FEM solution and the piecewise linear interpolant of the exact solution.

(and triangulation) and computes the corresponding FEM solution of a BVP. For each element, the *z*-stretch (the difference of maximum and minimum *z*-values of FEM solution over just the element) is recorded and those for which this stretch is in, say the largest 10% or exceeds a certain numerical value (this can be adjusted) are flagged. In the vicinity of such elements, extra nodes are added and a new mesh is created. This is iterated a certain number of times (which can be adjusted) or until the maximum *z*-streches fall below a certain prescribed value (which also can be adjusted). Such an adaptive scheme will be addressed in the exercises of this section.

EXERCISE FOR THE READER 13.9: Use the FEM with a triangulation of the disk involving roughly 100 nodes deployed in a way so that more nodes are used near $(x, y) = (\cos(3), \sin(3))$ to solve the BVP Example 13.7. Can you triangulate in such a way that the maximum error is smaller than that obtained in the solution of part (b) of Example 13.7 (when 897 nodes were used)? Plot the error (as computed above using the Poisson integral formula).
Suggestion: Try several different schemes with the main goal being to minimize the maximum total error (i.e., the *z*-height of the error graph). The node sets are small enough so that CPU time will not hinder multiple experiments.

NUMERICAL APPROXIMATION TO DOUBLE INTEGRALS— METHOD 2: APPROXIMATION QUADRATURE FORMULAS (RECOMMENDED):

Suppose that T is a region in the plane. A so-called **Gauss quadrature** formula for approximation of general integrals over T takes the form:

$$\int_T f(x, y)dxdy \approx w_1 f(\xi_1) + w_2 f(\xi_2) + \cdots + w_n f(\xi_n), \tag{22}$$

where the **weights** $w_1, w_2, \cdots, w_n$ are specified real numbers and the **sampling points** $\xi_1 = (x_1, y_1),\ \xi_2 = (x_2, y_2),\ \xi_1 = (x_1, y_1),\ \xi_2 = (x_2, y_2),\ \cdots, \xi_n = (x_n, y_n)$ are specified points in T. In general, these formulas are developed with the goal that they be exact for polynomials (in two variables) up to a specified degree. If such a formula was exact for polynomials of degree up to p, Taylor's theorem in two variables could then be used to show that if the integrand has continuous partial derivatives up to order $p + 1$ then the error of the approximation (22) is $O(h^{p+1})$, where h is the diameter of T (Exercise 28). For each sampling point there are three degrees of freedom (the weight, and the coordinates of the sampling point). For example, when T is a triangle with vertices V_1, V_2, and V_3 it can be shown (Exercise 29) that the following formula is exact for any polynomial of degree at most one:

$$\int_T f(x, y)dxdy \approx \frac{\text{Area}(T)}{3}\{f(V_1) + f(V_2) + f(V_3)\}. \tag{23}$$

This may be interpreted as a two-dimensional generalization of the trapezoidal rule. With the same number of sample points we can do better: If we choose them to be the midpoints of the edges of the triangle, rather than the vertices, we arrive at the following formula that turns out to be exact for polynomials of degree at most 2:

$$\int_T f(x,y)dxdy \approx \frac{\text{Area}(T)}{3}\left\{f([V_1+V_2]/2)+f([V_1+V_3]/2)+f([V_2+V_3]/2)\right\}. \quad (24)$$

For a brief but enlightening introduction on how such formulas are derived, see Section 5.2 of [ZiMo-83]. More details can be found in the article [Cow-73]; see also [Kry-62].

EXERCISE FOR THE READER 13.10: (a) Write an M-file for the Gaussian quadrature formula (24) having the following syntax:

```
int = gaussianintapprox(f, V1, V2, V3)
```

The input variables are: `f`, an inline function or an M-file, and `V1`, `V2`, and `V3`, the vertices of a triangle in the plane (listed as row vectors of length two). The output `int` is a number corresponding to the integral approximation of (24).
(b) Run through the MATLAB codes of part (c) of Example 13.7 on your own computer, and take note of the time it takes for the main finite element part of the code (after the triangulation). Then rewrite this part of the code to use the M-file of part (a) of this exercise in place of `dblquad`, and compare the resulting error and runtime.

Example 13.7 gives a nice demonstration of how refinements of the mesh will reduce the errors of the FEM approximations of the actual solution. In general, if the data for the BVP (10) satisfy: p, q, and f are piecewise continuous on Ω, the first partial derivatives of p q, and g are piecewise continuous on Γ_1, r and h are piecewise continuous on Γ_2 and $p(x,y) \geq p_0 > 0$, $q(x,y) \geq 0$, then it can be shown that with the above FEM scheme (as well as the one below for more general boundary conditions), the error of the FEM approximation is of order δ, where δ is the maximum diameter of any of the (triangular) elements. This result can be roughly expressed by the following inequality:

$$\|u - \hat{u}\| \leq C\delta. \quad (25)$$

Here u represents the exact solution of the BVP, $\hat{u}$ is the FEM solution (corresponding to a triangular mesh with h defined as above), and C is a constant that depends on the problem but not on δ. The norm on the left can be any of several norms to measure the errors. The order of the errors can be upgraded from δ to higher powers of δ by using basis functions that are locally polynomial of higher degree (some examples of such elements were given in the exercises of the

previous section). To give more specific results would require a deeper theoretical discussion involving some functional analysis. We refer the interested reader to Chapter 4 of [Joh-87]; see also [Cia-02] and [StFi-73]. We caution the reader that the situation of the Dirichlet problem on a disk for the Laplace PDE is very atypical in that an explicit solution is available (Theorem 13.1). The next two exercises for the reader will involve slight variations of this PDE; the first one deals with the same type of BVP but on another domain, while the second deals with a slightly different PDE (Poisson's) on the disk. It may come as a surprise that for such mild variations, no explicit solution techniques are known.

From our experience with both methods of numerical quadrature applied to the same problem of Example 13.7, we see that in any FEM program, the amount of time devoted to numerical quadrature is a crucial consideration. The relevant theorem on how numerical quadrature schemes affect the order of convergence of an FEM depends on the maximum degree of general polynomials for which the quadrature scheme integrates exactly. Stated roughly, if the error of an FEM approximation is of order δ^k, i.e., $\|u - \hat{u}\| \le C\delta^k$ (cf. (25)), and if the Gaussian quadrature formula (of form (22)) used is exact for all polynomials (in x and y) of degree at most $2k - 2$, then the same order of accuracy (perhaps with a different value of C) will hold for the solution resulting from the FEM with the numerical quadrature formula being used. For details, see Chapter IV of [CiLi-89]; see also Section 4.3 of [StFi-73]. In our case, $k = 1$, so that this theorem really only requires that constant functions be integrated exactly by the numerical quadrature formula being used. Nonetheless, the extra precision of our quadrature formula (24) comes at very little cost and will imrove the accuracy of our method. In the following two exercises for the reader, this quadrature formula is to be invoked in the FEM.

EXERCISE FOR THE READER 13.11: Using the grid of Example 13.3, apply the FEM to solve the following Dirichlet problem on the annulus $\Omega = \{(x, y) : 1 \le x^2 + y^2 \le 4\}$:

$$\begin{cases} (\text{PDE}) & \Delta u = 0 & \text{on } \Omega \\ (\text{BC}) & u \equiv 2 & \text{on } x^2 + y^2 = 1 \\ & u(2, \theta) = \cos(2\theta) & \text{on } x^2 + y^2 = 4 \end{cases}.$$

Plot your FEM solution and indicate the number of nodes, internal nodes, and elements. Your solution plot should look like that in Figure 13.39(a).

EXERCISE FOR THE READER 13.12: (a) Use the FEM to solve the following Poisson-Dirichlet problem on the unit disk $\Omega = \{(x, y) : x^2 + y^2 \le 1\}$ with the triangulation of Method 2 of the solution to Example 13.2(a)

$$\begin{cases} (PDE) & \Delta u = f & \text{on } \Omega \\ (BC) & u = 0 & \text{on } \partial\Omega \end{cases},$$

where the load $f(x,y)$ equals 100 on the (small) disk of radius r = 0.125 and center (x,y) = (0,0.5), and zero elsewhere. Plot your FEM solution and indicate the number of nodes, internal nodes, and elements. Your solution plot should look like that in Figure 13.39(b).

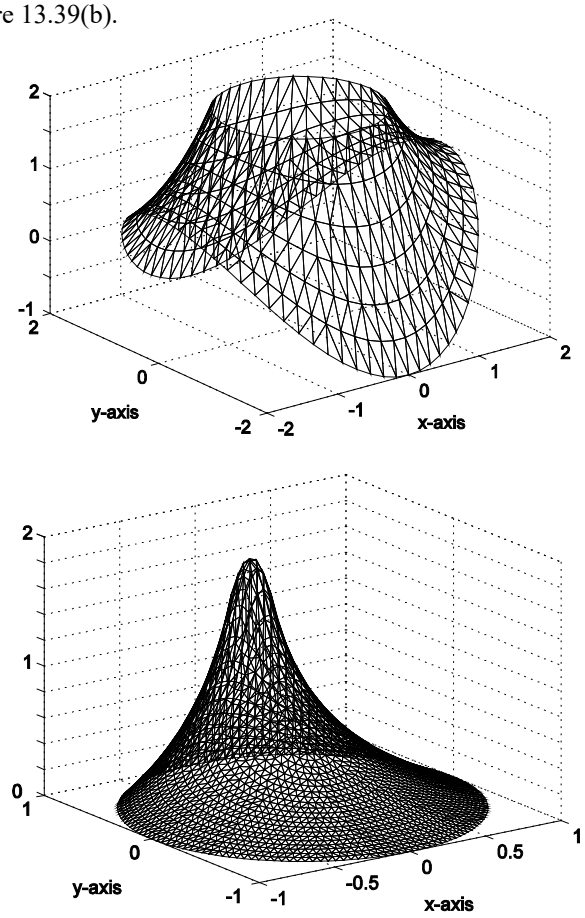


FIGURE 13.39: (a) (top) Plot of the FEM solution to the BVP of Exercise for the Reader 13.11. (b) (bottom) Plot of the FEM solution to the BVP of Exercise for the Reader 13.12(a).

(b) The triangulation of part (a) was rather uniform and had 1795 nodes. In this part we try to work with a smaller number of nodes but deploy them in a strategy that concentrates more of them near where the inhomogeneity $f(x,y)$ has most of its action. Construct a triangulation using between 500 to 1000 nodes and distributed in the four subregions of Figure 13.40(a) as follows: Roughly 50% of the nodes are to be deployed in Ω_1, 25% in Ω_2, 15% in Ω_3, and only 10% in Ω_4. Each of these regions is simply the intersection of the whole domain with the insides of the circles with center (0, ½) having radii: 1/4, 1/2, 1, and 3/2,

respectively. The distribution should be more or less uniform in each subregion. Obtain and plot the FEM solution. Your solution should look something like the one shown in Figure 13.40(c).

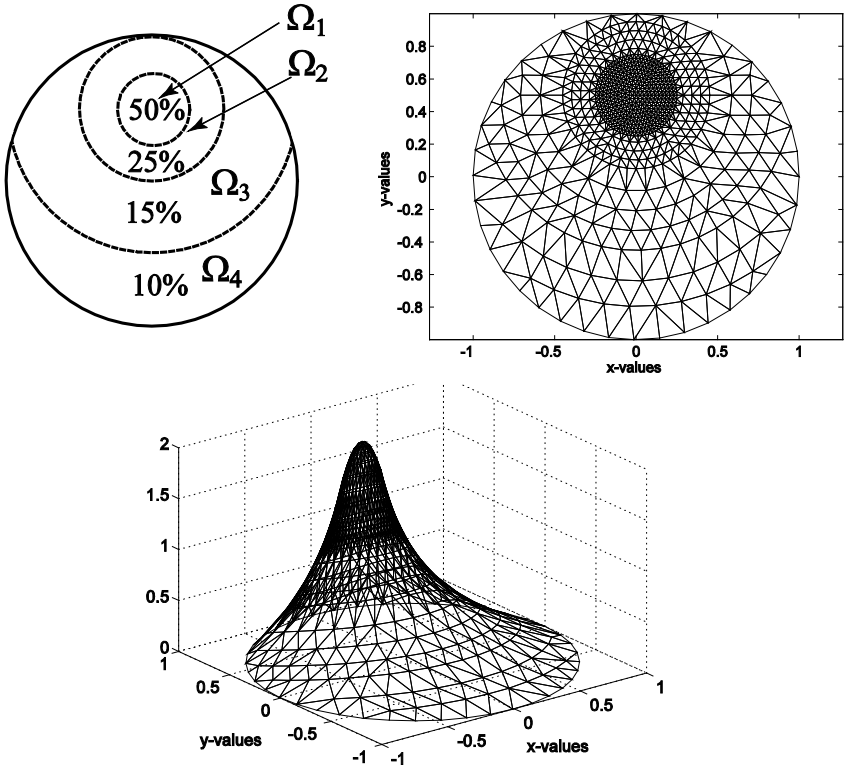


FIGURE 13.40: (a) (left) Diagram for node deployment strategy of part (b) in Exercise for the Reader 13.12. The unit disk Ω is split up into four subregions: $\Omega_1, \Omega_2, \Omega_3,$ and Ω_4. (b) (top right) A corresponding triangulation. (c) (bottom) The corresponding FEM solution; it appears graphically indistinguishable from the one obtained in part (a).

We now move on to describe the FEM for the general case of the BVP (10):

$$\begin{cases} \text{(PDE)} & -\nabla\bullet(p\nabla u) + qu = f & \text{on } \Omega \\ \text{(BCs)} & u = g & \text{on } \Gamma_1 \\ & \vec{n}\bullet\nabla u + ru = h & \text{on } \Gamma_2 \end{cases}.$$

Under the assumptions indicated in the theoretical discussion earlier in this section, this BVP can be shown to be equivalent to the following minimization problem:

Minimize the functional:

$$F[u] = \iint_{\Omega} [\tfrac{1}{2} p u_x^{\,2} + \tfrac{1}{2} p u_y^{\,2} + \tfrac{1}{2} q u^2 - f u]\, dxdy + \int_{\Gamma_2} [\tfrac{1}{2} r u^2 - h u]\, ds, \tag{26}$$

over the following set of admissible functions:

$$\mathcal{A} = \{ v : \Omega \to \mathbb{R} : v(x) \text{ is continuous, } v'(x) \text{ is piecewise continuous and bounded, and } v(x,y) = g(x,y) \text{ on } \Gamma_1 \}. \tag{27}$$

Note that the class of admissible functions requires only the Dirichlet boundary conditions (on Γ_1). The Robin boundary conditions (on Γ_2), are accounted for in the functional (26) and will be automatically satisfied by the solution.

Analogous to the one-dimensional method presented in Section 10.5, the FEM will solve a corresponding finite-dimensional minimization problem where the functional $F[u]$ of (26) is kept the same, but the set of admissible functions is reduced to an approximating smaller set determined by the basis functions of the triangulation. Thus we will be looking for minimizers of the functional F among functions of the form $v = \sum_{i=1}^{m} c_i \Phi_i$, where the $\Phi_i = \Phi_i(x,y)$ are the basis functions. The basis functions corresponding to nodes on the boundary portion Γ_1 will have their coefficients determined by the Dirichlet boundary conditions; it is the remaining coefficients (corresponding to interior nodes and nodes on the boundary portion Γ_2) that need to be determined. We now briefly outline the FEM for this general BVP. We follow this outline with some additional details and then give examples.

FEM FOR THE BVP (10)—GENERAL CASE:

Step #1: Decompose the domain into elements, and represent the set of nodes and elements using matrices. Separate the nodes N_i into the internal nodes and non-Dirichlet boundary nodes: $N_1, N_2, \cdots, N_n$ (that lie in $\Omega \cup \Gamma_2$), and the Dirichlet boundary nodes $N_{n+1}, N_{n+2}, \cdots, N_m$ (that lie on Γ_1). Denote the basis function Φ_{N_i} corresponding to node N_i simply by Φ_i. It is important that nodes be placed at all endpoints (interfaces) of Γ_1/Γ_2 and that these endpoints be counted as Dirichlet boundary nodes (i.e., grouped with those in Γ_1).

Step #2: Use the Dirichlet BCs $u(x,y) = g(x,y)$ on Γ_1 to determine the coefficients of the Dirichlet boundary node basis functions of an admissible function: $v = \sum_{i=1}^{m} c_i \Phi_i$, i.e., $c_i = g(N_i)$ for each $i = n+1, n+2, \cdots, m$.

Step #3: Assemble the $n \times n$ stiffness matrix A and load vector b needed to determine the remaining coefficients $c_1, c_2, \cdots, c_n$ which work to solve the discrete minimization problem corresponding to the BVP.
Step #4: Solve the stiffness equation $Ac = b$, and obtain the FEM solution

$$v = \sum_{i=1}^{m} c_i \Phi_i .$$

As before, the coefficients $c_1, c_2, \cdots, c_n$ will eventually be determined as the solution vector $c = [c_1 \;\; c_2 \;\; \cdots \;\; c_n]'$ of a linear system (14) $Ac = b$. The stiffness matrix A and load vector b will, in general, have entries given as follows:

$$a_{ij} = \iint_{\Omega} [p \nabla \Phi_i \bullet \nabla \Phi_j + q \Phi_i \Phi_j] dx dy + \int_{\Gamma_2} r \Phi_i \Phi_j \, ds \;\; (1 \le i, j \le n), \tag{28}$$

and

$$\begin{aligned} b_j = & \iint_{\Omega} f \Phi_j \, dx dy + \int_{\Gamma_2} h \Phi_j \, ds \\ & - \sum_{s=n+1}^{m} c_s \left\{ \iint_{\Omega} [p \nabla \Phi_s \bullet \nabla \Phi_j + q \Phi_s \Phi_j] dx dy + \int_{\Gamma_2} r \Phi_s \Phi_j \, ds \right\} \;\; (1 \le j \le n). \end{aligned} \tag{29}$$

In these formulas the integrals over Γ_2 are with respect to arclength (i.e., positively oriented line integrals). These can be derived in a similar fashion to what was done in the purely Dirichlet BC case (see the development of (13)). As before, we observe that the stiffness matrix is a symmetric matrix. The time-consuming part of the FEM is still the assembly process. The mechanics are as in the purely Dirichlet case (just replace (15^{ℓ}) and (16^{ℓ}) with their analogs for (28) and (29)).

The assembly process can be coded much like the way we did it for Example 13.7. The only new feature here is the presence of the line integrals. Before entering into the MATLAB code, we give a brief outline of how such line integrals can be evaluated. We show how to numerically evaluate integrals of the form $\int_{\Gamma_2 \cap T_\ell} F \, ds$, where F is any function on Γ_2 in the setting of an assembly code. Any such integral can be broken up into a sum of corresponding integrals over line segments. Let L denote a typical such line segment, connecting nodes N_1 and N_2 of T_ℓ. Letting $\vec{v} = N_2 - N_1$, we can write:

$$\int_L F \, ds = \int_0^1 F(N_1 + s\vec{v}) \, \| \vec{v} \| \, ds, \tag{30}$$

from the definition of line integrals. Such integrals could be done with MATLAB's `quad` (or `quadl`)—but in a general FEM code, it would be awkward

to combine the M-files for the integrand "*F*" with the needed change in variables unless we resort to symbolic variables. We avoid this dilemma and maintain consistency with the recommended method for approximating double integrals by invoking the following numerical quadrature approximation for ordinary integrals:

$$\int_0^1 f(x)dx \approx (1/6)\{f(0)+4f(1/2)+f(1)\}. \tag{31}$$

This formula, known as the **Newton-Coates formula** with three equally spaced points, is exact for polynomials up to degree three (Exercise 24). For more on such one-dimensional quadrature formulas, see Chapter 5 of [ZiMo-83], or see any good book on numerical analysis. What is most pertinent is that the accuracy of this approximation makes it feasible to use in the FEM; the underlying theory can be found in the references mentioned above. Combining (30) and (31) yields the following quadrature approximation, which is easily incorporated in FEM codes:

$$\int_L F\,ds \approx (\| N_1 - N_2 \| /6)\{F(N_1)+4F([N_1+N_2]/2)+F(N_2)\}, \tag{32}$$

EXERCISE FOR THE READER 13.13: (a) Write an M-file `lineint = bdyintapprox(fun, tri, redges)` that works as follows: The inputs will be `fun`, an inline (or M-file) function of the variables `x` and `y`, a 3×2 matrix `tri` of nodes of a triangle in the plane, and a 2-column matrix `redges`, possibly empty ([ ]). The rows of `redges` consist of the corresponding increasing node indices (from 1 to 3 corresponding to their row in `tri`) of nodes that are endpoints of segments of the triangle that are part of the "Robin" boundary (for an underlying FEM problem). Thus the rows of `redges` can include only the following three vectors: [1 2], [1 3], and [2 3]. The output, `lineint`, will be the approximation of the corresponding line integral of `fun` over the Robin segments of the triangle, by using formula (32).
(b) Test the accuracy of your M-file in computing the following line integrals over the indicated edge sets of the triangle with vertices $N_1=(0,0)$, $N_2=(2,0)$, $N_3=(0,3)$, and then on the triangle with vertices $N_1=(0,0)$, $N_2=(.2,0)$, $N_3=(0,.3)$. In the notation used below, ε_{ij} denotes the edge of this triangle joining N_i to N_j $(i\neq j)$. The line integrals are given below for the larger triangle:

$$\int_{\varepsilon_{12}\cup\varepsilon_{23}} 4\,ds = 8+4\sqrt{13}, \qquad \int_{\varepsilon_{12}\cup\varepsilon_{13}} \cos(\pi x/4+\pi y/2)\,ds = 2/\pi.$$

In our next example, we will solve a BVP over an odd-shaped region. The problem is carefully constructed so that the exact solution will be available for comparison purposes. In Exercise for the Reader 13.14, the reader will be asked to solve another such problem on the same region for which an exact solution is not available.

EXAMPLE 13.8: Use the finite element method to solve the following mixed BVP over the parabolically shaped domain $\Omega = \{(x, y) : 0 \le x \le 10,\ 0 \le y \le x(10 - x)\}$:

$$\begin{cases} \text{(PDE)} & -\Delta u = -1/25 & \text{on } \Omega \\ \text{(BCs)} & u = 0 & \text{on } x\text{-axis} \\ & \vec{n} \bullet \nabla u = \dfrac{y}{25(101 - 40x + 4x^2)^{1/2}} & \text{on } y = x(10 - x) \end{cases}.$$

(a) Use first a triangulation with between 300 and 500 nodes that are more or less uniformly distributed. Compare with the exact solution $u(x, y) = y^2 / 50$.

(b) Repeat part (a) this time using a similar triangulation with between 1000 and 2000 nodes.

Before we begin to solve this example, we leave the reader to perform the following:

EXERCISE FOR THE READER 13.14: Verify that the exact solution provided really solves the BVP in the above example.

SOLUTION TO EXAMPLE 13.8: Part (a): To decide on the linear gap distance between nodes, we first find the area of Ω:

$$\text{area}(\Omega) = \int_0^{10} x(10 - x)dx = [5x^2 - x^3 / 3]_0^{10} = 500/3$$

If we place the nodes in small square configurations (cf. Method 1 of Example 13.2(a)), then, roughly, each node would account for an area δ^2. Thus, if m denotes the number of nodes we use, then we would have (approximately) $m\delta^2 \approx \text{area}(\Omega)$, the left side being a bit larger due to boundary nodes. This gives the estimate

$$\delta \approx \sqrt{\text{area}(\Omega)/m}, \tag{33}$$

for the gap size we should use if we want to deploy m nodes. This formula can be used in the creation of squarelike nodegrids on any two-dimensional region with smooth boundary curves. Using (33) with the above area and $m = 350$, we arrive at the value $\delta = \sqrt{500/3/350} \approx 0.6901...$.

We begin by using MATLAB to deploy nodes in the interior of Ω, maintaining a safe distance close to δ away from the boundary and placing them in a square grid configuration with sidelength δ:

```
>> bdyf= inline('x.*(10-x)');
```

```
>> delta=sqrt(500/3/350);
>> nodecount=1;
>> for i=1:10/delta
for j=1:bdyf(i*delta)/delta
  xtemp=i*delta; ytemp=j*delta;
  if (bdyf(xtemp-delta/2)>ytemp)&...
  (bdyf(xtemp)>ytemp+delta/2)&(bdyf(xtemp+delta/2)>ytemp)
%These conditions assure that the parabolic portion of the boundary
%does not get too close to the candidate (xtemp, ytemp) for an
%internal node.
   x(nodecount)=xtemp; y(nodecount)=ytemp;
   nodecount = nodecount+1;
  end
end
end
```

We would like to assign the boundary nodes in such a way that the distance gap between nodes is approximately δ. This is quite simple to do on the straight portion of the boundary. For the curved portion, we now introduce a general method to accomplish such node deployment. Recall, the arclength formula for the graph of a function $f(x)$ over an interval $[a,b]$: $L=\int_a^b \sqrt{1+[f'(x)]^2}\,dx$. Now the parabolic boundary graph function has $f'(x)=10-2x$ so that the largest that the integrand $\sqrt{1+[f'(x)]^2}$ will be over [0, 10] is $\sqrt{101}\approx 10$. (This maximum change in arclength occurs near the endpoints where the parabola is most steep.) Since we will place a node on the parabola at x = 0, y = 0 (call this the "most recent node"), and then continue advancing x by $\delta/3\cdot 10$ (so the corresponding arclength of the parabola will advance by no more than about $\delta/3$), as soon as the arclength from the most recent node exceeds δ, we create a new node. Since we will place a node also at (10,0), we will place a safeguard to prevent the nodes on the parabola from getting too close to this one. The code below is set up so that the Dirichlet nodes are indexed last.

```
>> arcint = inline('sqrt(101-40*x+4*x.^2)');
>> xref1=0; xref2=delta/30; cumlen=quad(arcint,xref1,xref2);
>> while xref1<10
  while cumlen<delta
   xref2=xref2+delta/30;;
   cumlen=quad(arcint,xref1,xref2);
  end
  if xref2<10-delta/40
   x(nodecount)=xref2; y(nodecount)=bdyf(xref2);
   nodecount = nodecount+1;
  end
  xref1=xref2; xref2=xref2+delta/30;
  cumlen=quad(arcint,xref1,xref2);
end
if quad(arcint,xref1,10)>delta/2
  nodecount = nodecount+1;
end
>> x(nodecount)=10; y(nodecount)=0;
>> nodecount = nodecount+1;
```

```
>> %finally put nodes on interior of horizontal segment
>> num = floor(10/delta); delta2=10/num; xref=10-delta2;
>> while xref>delta2/4
  x(nodecount)=xref; y(nodecount)=0;
  nodecount = nodecount+1; xref=xref-delta2;
end
>> x(nodecount)=0; y(nodecount)=0; %last node
>> nodecount = nodecount+1; tri=delaunay(x,y);
>> trimesh(tri,x,y), axis('equal') %Plots the triangulation
```

The triangulation is shown in Figure 13.41(a). From the way the nodes were constructed, the boundary nodes come after the interior nodes and the first boundary node is on the parabolic portion of the boundary. We can thus find the key indices by:

```
>> nint=min(find(abs(y-bdyf(x))<10*eps))-1 %number of interior nodes
→nint = 307
>> n=find(x==10&y==0)-1 %number of interior/Robin nodes
→n = 373
>> m=length(x) %number of nodes
→m = 388
>> size(tri)
→ans =  693     3  (So there are 693 elements.)
```

We give special names to the node numbers of the endpoints of the segment (interface with Robin/Dirichlet nodes):

```
>> dir1 = m;%node (0,0)
>> dir2 = nint+1; %node (10,0)
```

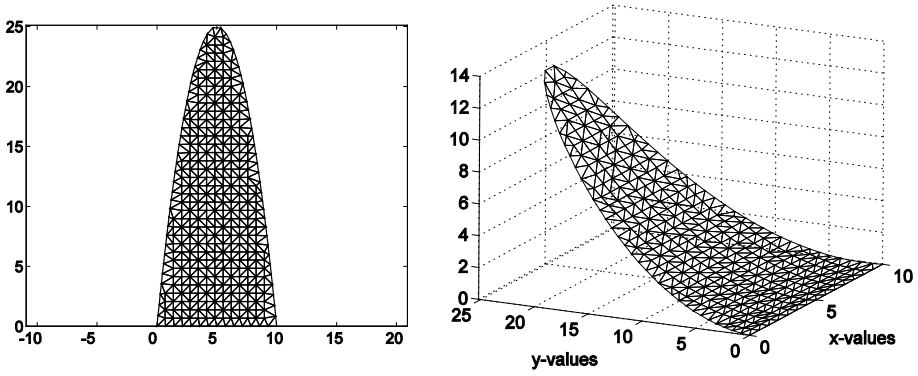


FIGURE 13.41: (a) (left) The triangulation of the parabolic region for the BVP of Example 13.8(a). There are 388 nodes and 693 elements. With this resolution the curved boundary is rather well represented by the element boundaries, except near the top where the curvature of the parabola is most extreme. (b) (right) The FEM solution of Example 13.8(a). The exact solution is graphically indistinguishable, the maximum relative error being less than 1%.

From the key node indices found above, we conclude:

Interior nodes: 1:307
Robin nodes (on interior of parabola): 308: 373
Dirichlet nodes (on line segment): 374:388

Notice we have created nodes at the interfaces of the two boundary portions (Dirichlet meets Robin) and these interface nodes will be assigned the Dirichlet conditions, as required. Since the Dirichlet boundary conditions are zero, simply creating a 388-length vector `c` of zeros will take care of assigning the Dirichlet nodes their appropriate values:

```
>> c= zeros(m,1);
```

The crucial index here is $n = 373$, the number of interior nodes added to the number of Robin boundary nodes; this is how many coefficients need to be determined. Since, in (10), we have $p \equiv 1$, $q \equiv 0$, $f = -1/25$, $g \equiv 0$, $r \equiv 0$, and since $c_s = 0$ $(s > n)$, the element matrix analogues of (28) and (29) (cf. (15^{ℓ}) and (16^{ℓ})) are as follows:

$$a_{\alpha\beta}^{\ell} = \iint_{T_\ell} [\nabla \Phi_{i_\alpha} \bullet \nabla \Phi_{i_\beta}] \, dxdy \quad (1 \le \alpha, \beta \le 3),$$

and

$$b_{\alpha}^{\ell} = (-1/25) \iint_{T_\ell} (\Phi_{i_\alpha}) \, dxdy + \int_{\Gamma_2 \cap T_\ell} h(x,y) \Phi_{i_\alpha} \, ds \quad (1 \le \alpha \le 3),$$

where $h(x,y) = \dfrac{y}{25(101 - 40x + 4x^2)^{1/2}}$.

For each element index ℓ, these coefficients need to be computed only when the nodes i_α and/or i_β are interior or Robin nodes (i.e., $i_\alpha, i_\beta \le n \equiv 373$) corresponding to vertices of the corresponding element. The assembly code will invoke the M-file `gaussianintapprox` of Exercise for the Reader 13.10 for approximating the double integrals, and the M-file `robinbdyint` of Exercise for the Reader 13.13 for numerically evaluating line integrals. Here is the assembly code:

```
N=[x' y']; E=tri; A=zeros(n); b=zeros(n,1); [L cL]=size(E);
for ell=1:L
  nodes=E(ell,:); %global node indices of element
  intnodes=nodes(find(nodes<=n)); %global interior/Robin node indices
%find coefficients [a b c] of local basis functions
% ax + by +c; for  int/robin nodes
  for i=1:length(intnodes)
   xyt=N(intnodes(i),:); %main node for local basis function
   onodes=setdiff(nodes,intnodes(i));
%global indices for two other nodes (w/ zero values) for local basis
%function
   xyr=N(onodes(1),:);
```

```
   xys=N(onodes(2),:);
   M=[xyr 1;xys 1;xyt 1]; %matrix M of (4)
%local basis function coefficients using (6B)
   abccoeff=[xyr(2)-xys(2);xys(1)-xyr(1);xyr(1)*xys(2)-
xys(1)*xyr(2)]/...
   det(M);
   intgrad(i,:)=abccoeff(1:2)';
   abc(i,:)=abccoeff';
end

% determine if there are any Robin edges
marker=0; %will change to 1 if there are Robin edges.
roblocind=find(nodes==dir1|nodes==dir2|(nodes<=n& nodes >=(nint+1)));
%local indices of nodes for possible robin edges
if length(roblocind)>1
  elemnodes = N(nodes,:);
%now find robin edges and make a 2 column matrix out of their local
%indices.
  rnodes=nodes(roblocind); %global indices of robin nodes
  count=1;
  for k=(nint+1):n
   if ismember(k,rnodes) & ismember(k+1,rnodes)
    robedges(count,:)=[find(nodes==k)  find(nodes==k+1)];
    count=count+1; marker =1;
  end
end
end

%update stiffness matrix
for i1=1:length(intnodes)
for i2=1:length(intnodes)
  if intnodes(i1)>=intnodes(i2) %to save some computation, we use
  %symmetry the stiffness matrix.
  fun1 = num2str(intgrad(i1,:)*intgrad(i2,:)',10);
  %integrand a-integral
  fun=inline(fun1,'x', 'y');
  integ=gaussianintapprox(fun,xyt,xyr,xys);
A(intnodes(i1),intnodes(i2))=A(intnodes(i1),intnodes(i2))+integ;
end
end
end

%update load vector
for i1=1:length(intnodes)
  ai1 = num2str(abc(i1,1),10);
  bi1 = num2str(abc(i1,2),10);
  ci1 = num2str(abc(i1,3),10);
  fun=inline([ai1,'*x+',bi1, '*y+', ci1],'x','y');
  integ=-1/25*gaussianintapprox(fun,xyt,xyr,xys);
  b(intnodes(i1))=b(intnodes(i1))+integ;
  %now add Robin portion, if applicable
  %robin edges were computed above
  if marker==1
prod=inline(['y./(25.*sqrt(101-
40.*x+4.*x.^2))','*(',ai1,'*x+',bi1,...
 '*y+', ci1,')'],'x','y');
b(intnodes(i1))=b(intnodes(i1))+bdyintapprox(prod, elemnodes,...
robedges);
end
```

```
end
clear roblocind rnodes robedges
end
A=A+A'-A.*eye(n); %Use symmetry to fill in remaining entries of A.


sol=A\b; c(1:n)=sol'; c(n+1:m)=0;

%The result is now easily plotted using the 'trimesh' function of the
%last section:

x=N(:,1); y=N(:,2);
trimesh(E,x,y,c), hidden off
xlabel('x-axis'), ylabel('y-axis')
```

The following commands will plot the error using the exact solution provided. The result is shown in Figure 13.42(a).

```
cexact = zeros(m,1);
for i=1:length(x),cexact(i)=y(i)^2/50; end
trimesh(E,x,y,abs(c-cexact))
```

Part (b) is done in exactly the same fashion. In fact, the above code is designed in such a way that only the second line (defining delta) in the node deployment needs to be adjusted (change 350 to 1800). With this change, the above code will produce a numerical solution with error plot shown in Figure 13.42(b).[16]

The various examples done so far contain all of the necessary techniques needed to apply the FEM to general BVPs of form (10). The next two exercises for the reader contain two more examples.

EXERCISE FOR THE READER 13.15: Consider the following steady-state heat distribution problem on the parabolic region $\Omega = \{(x, y) : 0 \le x \le 10,\ 0 \le y \le x(x-10)\}$ of the preceding example:

$$\begin{cases} (\text{PDE}) & -\Delta u = f & \text{on } \Omega \\ (\text{BCs}) & u = 0 & \text{on } x\text{-axis} \\ & \vec{n} \bullet \nabla u + 2u = 40 & \text{on } y = x(10-x) \end{cases},$$

where the source function is given by: $f(x, y) = \begin{cases} 200, & \text{if } 4 \le x \le 6 \text{ and } 10 \le y \le 15 \\ 0, & \text{otherwise.} \end{cases}$

The region Ω along with the boundary conditions and the support of the source function f is illustrated in Figure 13.43(a).

(a) Compute and plot the numerical solution of this BVP using the triangulation of the solution to part (a) of Example 13.8.

---

[16] For the convenience of the reader, the entire MATLAB codes for this example (and other longer examples and exercises for the reader of this chapter) are included as downloadable text files on the ftp site for this book (see the preface for the URL of this ftp site). These codes can easily be pasted directly into the MATLAB window, and they can be modified to solve other FEM problems.

(b) Compute and plot the numerical solution of this BVP using the triangulation of the solution to Part (b) of Example 13.8. Your plot should look like the one in Figure 13.43(b).

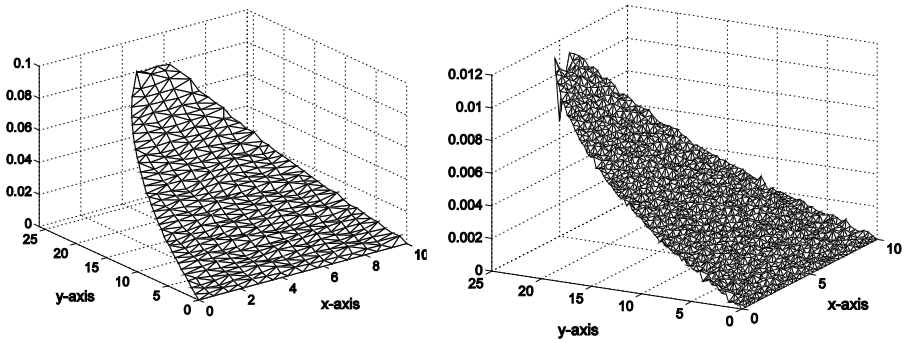


FIGURE 13.42: Error plots for the FEM solution of Example 13.8. (a) (left) Using the triangulation of part (a), which had 693 elements, the actual error was less than 1%. (b) (right) Using the triangulation of part (b), which had 3587 elements, the actual error was less than 0.1%.

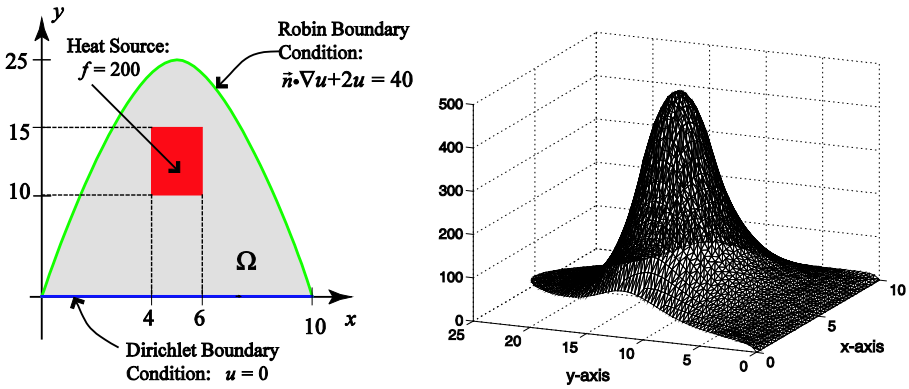


FIGURE 13.43: (a) (left) Illustration of the domain and boundary conditions for the steady-state heat distribution problem of Exercise for the Reader 13.15. The parabolically shaped plate has an internal rectangular heat source (temperature = 200) shown by a dark rectangle. The bottom (flat) edge is maintained at temperature 0 and the curved part of the boundary has a Robin boundary condition. (b) (right) An FEM solution of this problem.

EXERCISE FOR THE READER 13.16: (a) Contruct a squarelike grid and then a corresponding triangulation with between 2000 and 3000 nodes for the domain of Figure 13.44(a).
(b) Use the FEM with your triangulation to solve the Laplace problem $\Delta u = 0$ on this domain with boundary conditions as shown in Figure 13.44(a) and then plot your solution. Your plot should look like the one shown in Figure 13.43(b).

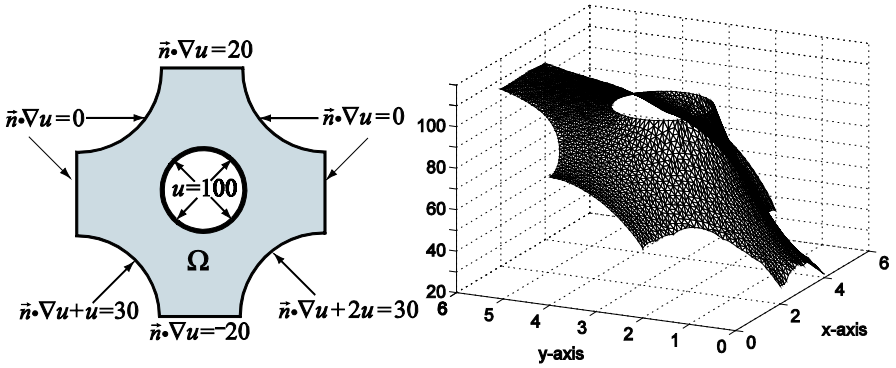


FIGURE 13.44: (a) (left) Illustration of the domain and boundary conditions for the BVP problem of Exercise for the Reader 13.16. The circular (inner) boundary portion has Dirichlet boundary conditions; the remaining (outer) boundary portions have the indicated Neumann or Robin conditions. (b) (right) Plot of the FEM solution for the Laplace problem having the indicated boundary data of (a). The triangulation used had 2655 nodes and 5024 elements.

EXERCISES 13.3

1. (a) Using the exact method of Example 13.5 (with Exercise for the Reader 13.4), solve the following BVP on the same hexagonal domain and triangulation of that example:

$$\begin{cases}(\text{PDE}) & -\Delta u = 0 & \text{on } \Omega \\ (\text{BC}) & u = x + y & \text{on } \partial\Omega\text{'}\end{cases}$$

and plot the resulting numerical solution.
(b) Check that $u(x,y) = x + y$ is the exact solution of the BVP; compare the numerical solution with this exact solution.

2. (a) Using the exact method of Example 13.5 (with Exercise for the Reader 13.4), solve the following BVP on the same hexagonal domain and triangulation of that example:

$$\begin{cases}(\text{PDE}) & -\Delta u = 0 & \text{on } \Omega \\ (\text{BC}) & u = x^2 + y^2 & \text{on } \partial\Omega\text{'}\end{cases}$$

and plot the resulting numerical solution.
(b) Check that $u(x,y) = x^2 + y^2$ is the exact solution of the BVP; compare the numerical solution with this exact solution.

3. (a) Using the exact method of Example 13.4 (with Exercise for the Reader 13.4), apply the FEM on the triangular domain Ω of Figure 13.45 with the triangulation shown there to solve the following BVP:

$$\begin{cases}(\text{PDE}) & -\Delta u = 2 & \text{on } \Omega \\ (\text{BC}) & u = x & \text{on } \partial\Omega\text{'}\end{cases}$$

The vertical/horizontal distance between adjacent nodes is one, and node #7 has coordinates (0,0) (so, for example, node #1 is at (0,3) and node #10 is at (3,0)). Plot the resulting numerical solution.
(b) Solve this problem again with the FEM but this time use the Gauss quadrature formula (24) (or the `gaussianintapprox` M-file) to evaluate integrals. Compare with the solution obtained in part (a);

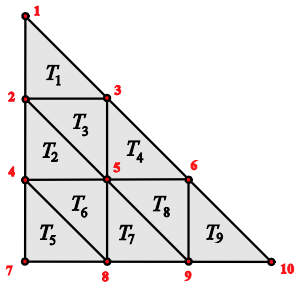


FIGURE 13.45: Triangular domain with basic triangulation for Exercise 3.

comment on any discrepancies or lack thereof.
(c) Re-solve the problem this time using MATLAB's `dblquad` to evaluate all double integrals. Compare with the solution obtained in part (a).

4. Repeat all parts of Exercise 3 for the following BVP on the domain Ω of Figure 13.45 described there.

$$\begin{cases} \text{(PDE)} & -\Delta u = f(x,y) & \text{on } \Omega \\ \text{(BC)} & u = (x+y)^2 & \text{on } \partial\Omega \end{cases}, \quad \text{where } f(x,y) = \begin{cases} 0, & \text{if } y \le 1, \\ 1, & \text{if } 1 < y \le 2 \\ 2, & \text{if } 2 < y \end{cases}.$$

5. Using the same triangulation on the hexagonal domain of Example 13.5, use the FEM to solve the following BVP:

$$\begin{cases} \text{(PDE)} & -\Delta u + u = 3 & \text{on } \Omega \\ \text{(BC)} & u = x + y & \text{on } \partial\Omega \end{cases}.$$

Note: Since the quadrature formula (24) is exact for polynomials up to second degree, it can be used to exactly evaluate all of the integrals that arise.

6. Using the same triangulation on the triangular domain of Exercise 3, use the FEM to solve the following BVP:

$$\begin{cases} \text{(PDE)} & -\Delta u + u = f(x,y) & \text{on } \Omega \\ \text{(BC)} & u = x^2 & \text{on } \partial\Omega \end{cases}, \quad \text{where } f(x,y) = \begin{cases} 0, & \text{if } x \le 1, \\ -2, & \text{if } x > 1 \end{cases}.$$

Note: Since the quadrature formula (24) is exact for polynomials up to second degree, it can be used to exactly evaluate all of the integrals that arise.

7. Consider the Dirichlet problem (19) on the unit disk $\Omega = \{(x,y) : x^2 + y^2 < 1\}$:

$$\begin{cases} \text{(PDE)} & \Delta u = 0 & \text{on } \Omega \\ \text{(BC)} & u(1,\theta) = \cos^2(\theta) & \text{on } \partial\Omega \end{cases}.$$

(a) Use the FEM with the triangulation of part (c) of Example 13.7 to compute the numerical solution of the problem, performing the double integrals as in Example 13.7. Keep track of the time needed to perform the main assembly (using `tic…toc`). Use Poisson's integral formula (Theorem 13.1) to compute the "exact solution" to this problem at the nodes of the triangulation and plot solution and the error of the FEM solution obtained.
(b) Repeat part (a), but this time use the Gauss quadrature formula (24) (or the `gaussianintapprox` M-file) to compute the double integrals in the FEM. Compare and contrast the FEM numerical solutions of parts (a) and (b).
(c) Use the FEM as in part (b) to find the numerical solution of this problem using a triangulation of the circle having between 3000 and 4000 nodes. Plot the error against the corresponding "exact solution" from Poisson's integral formula.
(d) Repeat each of the above parts for the Dirichlet problem identical to the above but with the boundary condition being changed to $u(1,\theta) = \sin^2(\theta/2)$.

8. Consider the following Dirichlet problem (19) on the disk $\Omega = \{(x,y) : (x-1)^2 + (y-3)^2 < 5\}$:

$$\begin{cases} \text{(PDE)} & \Delta u = 0 & \text{on } \Omega \\ \text{(BC)} & u(x,y) = \ln x + 2y & \text{on } \partial\Omega \end{cases}.$$

(a) Use the FEM with a triangulation having between 500 and 1000 nodes to compute the numerical solution of the problem, performing the double integrals as in Example 13.7. Keep track of the time needed to perform the main assembly (using `tic…toc`). Use Poisson's integral formula (Theorem 13.1) to compute the "exact solution" to this problem at the nodes of the triangulation and plot solution and the error of the FEM solution obtained.
(b) Repeat part (a), but this time use the Gauss quadrature formula (24) (or the `gaussianintapprox` M-file) to compute the double integrals in the FEM. Compare and contrast the FEM numerical solutions of parts (a) and (b).
(c) Use the FEM as in part (b) to find the numerical solution of this problem using a

triangulation of the circle having between 3000 and 4000 nodes. Plot the error against the corresponding "exact solution" from Poisson's integral formula.
(d) Repeat each of the above parts for the Dirichlet problem identical to the above but with (i) the boundary condition being changed to $u(x,y) = 2x + e^y$ and then (ii) with the PDE changed to $-\nabla\bullet(e^{x+y}u) + u = y$ but all else as in the original problem.

9. Consider the following Robin problem for the Laplacian on the unit disk $\Omega = \{(x,y): x^2 + y^2 < 1\}$:

$$\begin{cases} \text{(PDE)} & \Delta u = 0 & \text{on } \Omega \\ \text{(BC)} & \vec{n}\bullet\nabla u + u = 3 & \text{on } \partial\Omega \end{cases}.$$

(a) Use the FEM to solve this problem using a triangulation having between 500 to 1000 nodes and plot your numerical solution.
(b) Create a triangulation having between 1500 and 2000 nodes containing the node set of your triangulation of part (a). Re-solve the BVP with the FEM on this triangulation. Plot the new solution, compare it with that of part (a), and finally plot the difference of the two solutions on the common node set.
(c) Create a triangulation having between 3000 and 3500 nodes containing the node set of your triangulation of part (b). Re-solve the BVP with the FEM on this triangulation. Plot the new solution, compare it with that of part (b), and finally plot the difference of the two solutions on the common node set.
(d) Repeat each of parts (a) through (c) for the BVP with the same Robin boundary conditions of the above problem, but with the PDE changed to:
(i) $\nabla\bullet(e^{x+y}u) = 0$, (ii) $\nabla\bullet(e^{x+y}u) = 3$, (iii) $\nabla\bullet(e^{x+y}u) = -3$, (iv) $-\nabla\bullet(e^{x+y}u) + u = 3$.

10. Consider the following BVP on the annulus $\Omega = \{(x,y): 1 \le x^2 + y^2 \le 4\}$ of Exercise for the Reader 13.11:

$$\begin{cases} \text{(PDE)} & \Delta u = e^{x^2/2} & \text{on } \Omega \\ \text{(BCs)} & \vec{n}\bullet\nabla u \equiv 10 & \text{on } x^2 + y^2 = 1 \\ & u(2,\theta) = 50 & \text{on } x^2 + y^2 = 4 \end{cases}.$$

(a) Use the FEM to solve this problem using a triangulation having 500 to 1000 nodes and plot your numerical solution.
(b) Create a triangulation having between 1500 and 2000 nodes containing the node set of your triangulation of part (a). Re-solve the BVP with the FEM on this triangulation. Plot the new solution, compare it with that of part (a), and finally plot the difference of the two solutions on the common node set.
(c) Create a triangulation having between 3000 and 3500 nodes containing the node set of your triangulation of part (b). Re-solve the BVP with the FEM on this triangulation. Plot the new solution, compare it with that of Part (b), and finally plot the difference of the two solutions on the common node set.
(d) Repeat each of parts (a) through (c) for the BVP with the same Robin boundary conditions of the above problem, but with the PDE changed to:
(i) $\nabla\bullet(e^{x+y}u) = 0$, (ii) $\nabla\bullet(e^{x+y}u) = 3$, (iii) $\nabla\bullet(e^{x+y}u) = -3$, (iv) $\nabla\bullet(e^{x+y}u) + u = 3$.

11. This exercise will use the FEM to solve the heat problem (1)

$$\begin{cases} \Delta u = 0 \text{ on } \Omega, \quad u = u(x,y) \\ \partial u/\partial n = 0, \text{ on outer rectangle} \\ \text{u=40, on small circle, u=500, on large circle} \end{cases}$$

from the introductory section. Take the domain (see Figure 13.2) to be the rectangle: $-1 < x < 0.5$, $-0.5 < y < 0.5$ with the following two disks deleted: larger circle: center = (–0.65, 0.15), radius = 0.25 and smaller circle: center = (0.1, –0.2), radius = 0.1. In each case you are to plot your results.
(a) First use a triangulation with between 300 and 500 nodes, more or less uniformly spaced.

(b) Repeat part (a) using a triangulation with between 1500 and 2000 nodes.
(c) (i) Repeat parts (a) and (b) on the BVP gotten from (1) by changing the BC on the outer rectangle to be $\partial u/\partial n + u = 40$, but keeping all else the same. (ii) Do this again using instead the BC on the larger circle to be $\partial u/\partial n + 4u = 40$. (iii) Repeat using instead the BC on the larger circle to be $\partial u/\partial n + u = 80$.
(d) (i) Repeat parts (a) and (b) on the BVP gotten from (1) by changing the PDE to be $\Delta u = f(x, y)$, where $f(x, y) = -100$ on the circle with center center = (0.3, 0.25), radius = 0.1, and $f(x,y) = 0$ elsewhere (but keeping all else the same). (ii) Do this again but change the PDE to $\Delta u + 2u = f(x, y)$.

12. (*Comparison of the FEM and the Finite Difference Method for a Certain Mixed BVP*) (a) Use the FEM to solve the BVP of Exercise for the Reader 11.8. For the triangulation, let the node set correspond to that in part (a) of that exercise for the reader, i.e., nodes are uniformly spaced in a squarelike grid with horizontal and vertical gap size equaling h = 0.05. Let MATLAB's `delaunay` produce the actual triangulation once you create the node set. Plot your solution and compare it with Figure 11.23(a). Produce also a contour plot for your FEM solution and compare it with Figure 11.23(b).
(b) Repeat part (a), but using the finer grid with horizontal/vertical gap size $h = 0.02$.

13. Repeat both parts of Exercise 11 on the BVP with the same boundary conditions but with the PDE changed from the Laplace equation to $-\nabla\cdot([x^2 + y^2 + 1]u) + u = \cos(xy)$.

14. (*Determination of Maximum Tolerable Heat*) Consider the domain of Figure 13.46:

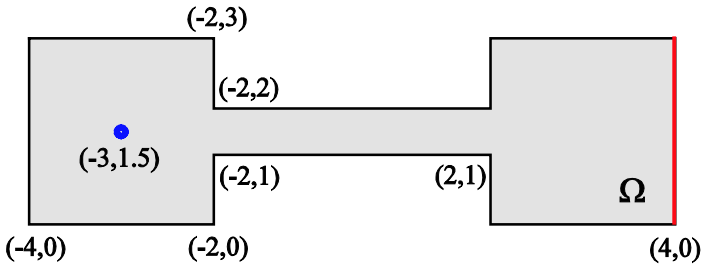


FIGURE 13.46: A domain consisting of two squares joined by a rectangular neck.

(a) In this domain, an observer at location (-3, 1.5) (left side) cannot tolerate a temperature greater than 50. All edges except for the right edge are kept insultated $\vec{n}\cdot\nabla u = 0$ while the right edge will be maintained at a certain temperature $u = T_{\text{hot}}$. What is the maximum value of T_{hot} so the observer's requirement is met? Try to get your answer accurate to at least two decimals. For the PDE in the domain use the basic Laplace equation $\Delta u = 0$.
(b) How would the answer in part (a) change if the rectangular length were to be doubled in length?
(c) How would the answer in part (a) change if the rectangular length were to have only half of its height?
(d) How would the answer in part (a) change if the square on the right were to have its sidelength doubled (but the left square is still kept the same)?
(e) How would the answer in part (a) change if the hot edge of the square on the right were the top edge rather than the right edge?

15. Let Ω be the domain shown in Figure 13.47, with the deleted disk having center (2.5, 2.5) and radius, 0.75.
(a) Create triangulation of Ω having between 300 and 400 (essentially equally spaced) nodes.
(b) Create a triangulation of Ω having between 1500 and 2000 nodes.
(c) Use the FEM with the triangulation of part (a) to solve the following BVP on Ω :

$$\begin{cases} \Delta u = 0 \text{ on } \Omega \\ \vec{n}\bullet\nabla u = 10, \text{ on triangle .} \\ \text{u=100, on circle} \end{cases}$$

Plot your result and then repeat with the triangulation of part (b).

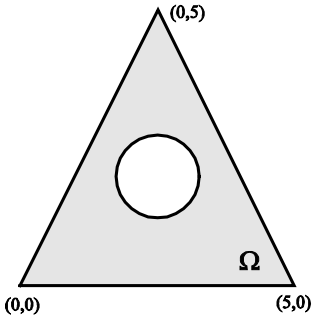


FIGURE 13.47: Triangular domain with basic triangulation for Exercise 3.

(d) Repeat part (c) on the modified BVP gotten by changing the PDE to be (i), $-\Delta u = x^2/2,$ but keeping all else the same; and then to (ii) $-\Delta u + e^{x/2}u = x^2/2.$

16. (a) Triangulate the domain of Figure 13.48 using between 400 and 800 nodes, more or less equally spaced.
(b) Repeat part (a) but this time use between 2000 and 2500 nodes.
(c) Use the FEM and the triangulation of part (a) to solve the heat problem on the domain of Figure 13.48 governed by the Laplace equation $\Delta u = 0$ and the boundary conditions shown in the figure. Plot your numerical solution.
(d) Repeat part (c), this time using the triangulation of part (b).
(e) Repeat both parts (c) and (d) on the modified BVP gotten by changing the boundary conditions on Γ_1^2 and Γ_1^3 to be $\vec{n}\bullet u = 0$ (insulated), but keeping all else the same.
(f) Repeat both parts (c) and (d) on the modified BVP gotten by changing the boundary conditions on Γ_1^2 and Γ_1^3 to be the Robin conditions: $\vec{n}\bullet u = 20$ and $\vec{n}\bullet u = -20,$ respectively, but keeping all else the same.

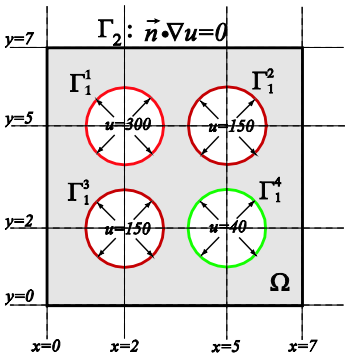


FIGURE 13.48: Boundary conditions for the heat problem of Exercise 11. The outer square boundary Γ_2 is insulated, while the four circular inner boundary portions Γ_1^i, $1 \le i \le 4$, are each maintained at the indicated temperatures.

(g) Repeat both parts (c) and (d) on the modified BVP gotten by changing the boundary conditions on Γ_1^2 and Γ_1^3 to be the Robin conditions: $\vec{n}\bullet u + u = 20$ and $\vec{n}\bullet u + u = 20,$ respectively, but keeping all else the same.
(h) Repeat both parts (c) and (d) on the modified BVP gotten by changing the PDE to be $\nabla\bullet(([2^{x/2} + y]u) + u = 10,$ but keeping all else the same.

17. Let Ω be the domain between the x-axis and the graph of $y = e^x$ from $x = 0$ to $x = 4$.
(a) For the function $u(x, y) = \sin(x/(y+1)) + x^2 y/25,$ determine functions $g(x)$, $f(x,y)$ and $h(x,y)$ so that $u(x,y)$ solves the following BVP:

$$\begin{cases} \text{(PDE)} & -\Delta u + u = f(x, y) & \text{on } \Omega \\ \text{(BCs)} & u = g(x) \text{ on } x\text{-axis}; & \vec{n}\bullet\nabla u + u = h(x, y) \quad \text{on curved portion of } \partial\Omega \end{cases}.$$

(b) Construct a triangulation of Ω having between 300 and 500 nodes. Compute the corresponding FEM solution and use the exact solution to plot the error.
(c) Repeat part (b) this time using between 1500 and 2000 nodes.

18. Write an M-file, `integ=quadint(fun,v1,v2,v3, v4)`, whose inputs are `fun` an inline function of *x*,*y*, and four 2×1 matrices `v1`, `v2`, `v3`, `v4` that are vertices (in any order) of quadrilateral (four sided polygon) in the *xy*-plane. If we denote this quadrilateral by Q, the output `integ` should be the numerical integral $\int_Q \text{fun}(x,y)\,dxdy$, computed using `dblquad`, MATLAB's numerical integrator.

19. Derive formulas (13) through (18) for the FEM for BVPs with purely Dirichlet BCs.

20. (a) Establish the integral formula (21) for general planar regions of Figure 13.34.
(b) Derive a similar integration formula for regions between functions of y.
Suggestion: For part (a), in the last integral, make the following substitution $y = y\text{low}(x) + u(y\text{top}(x) - y\text{low}(x))$.

21. Suppose that a Gauss quadrature formula (22):
$$\int_T f(x,y)dxdy \approx w_1 f(\xi_1) + w_2 f(\xi_2) + \cdots + w_n f(\xi_n)$$
is exact for polynomials of degree up to p. Use Taylor's theorem in two variables to show that if the integrand has continuous partial derivatives up to order $p + 1$, then the error of the approximation (22) is $O(h^{p+1})$, where h is the diameter of the triangle T.

22. Show that the Gauss quadrature formula (23):
$$\int_T f(x,y)dxdy \approx \frac{\text{Area}(T)}{3}\{f(V_1) + f(V_2) + f(V_3)\}$$
is exact for linear (first-degree) polynomials, but not for quadratic (second degree) polynomials. Here, T is a triangle and V_1, V_2, V_3 are its vertices.

23. Show that the Gauss quadrature formula (24):
$$\int_T f(x,y)dxdy \approx \frac{\text{Area}(T)}{3}\{f([V_1+V_2]/2) + f([V_1+V_3]/2) + f([V_2+V_3]/2)\},$$
is exact for quadratic (second-degree) polynomials. Here, T is a triangle and V_1, V_2, V_3 are its vertices.
Suggestion: First work with the standard triangle with vertices (0,0), (1,0), and (0,1). You need only verify it for the basis polynomials and use of linearity. Once this is done use affine maps to get the result for general triangles (see Exercise 19 of the previous section).

24. Show that the Newton-Coates quadrature formula (30):
$$\int_0^1 f(x)dx \approx (1/6)\{f(0) + 4f(1/2) + f(1)\}$$
is exact when $f(x)$ is polynomial of degree at most three.

NOTE: The next four exercises will introduce the reader to some refinement and adaptive implementations of the FEM. These are based on the simple refinement scheme of splitting an element into four similar elements by introducing a new node at the midpoint of each edge; see Figure 13.49.

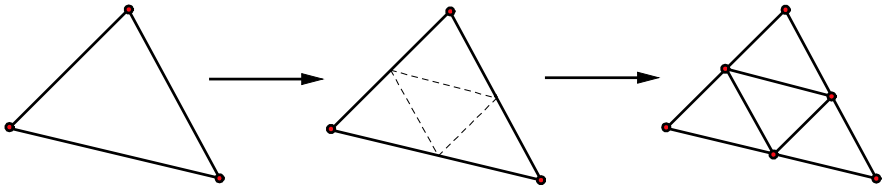


FIGURE 13.49: A refinement scheme for triangular meshes that is easily programmed into FEM routines.

25. (*M-file for Automatic Mesh Refinement*) The scheme of Figure 13.49 gives rise to a very natural refinement scheme that can be repeated for any number of iterations. Simply start off with any triangulation $\mathscr{A}_0$ of a given domain. For the first refinement $\mathscr{A}_1$ of $\mathscr{A}_0$, the node set will be the node set of $\mathscr{A}_0$, along with the midpoints of all element edges. Each element of $\mathscr{A}_0$ gives rise to four elements of $\mathscr{A}_1$ as in Figure 13.49. This procedure can be iterated to get a sequence of successively finer triangulations $\mathscr{A}_0$, $\mathscr{A}_1$, $\mathscr{A}_2$, …. We point out two very nice properties: (i) the node set of $\mathscr{A}_n$ is contained in the node set of $\mathscr{A}_{n+1}$ (making it simple to compare FEM solutions on successive triangulations), and (ii) the minimum angle of any element of $\mathscr{A}_{n+1}$ equals the minimum angle of any element of $\mathscr{A}_n$ (this keeps control of the eccentricity of elements, which is important for the FEM).
(a) Write an M-file that will perform the above refinement and with the following syntax:
`[newnodes, newtri] = meshrefine(nodes, tri)`
The input variable `nodes` is a two-column matrix of x- and y-coordinates of a given triangulation of a planar domain and `tri` is the corresponding three-column matrix of node numbers of the elements of the triangulation. (As usual, the node numbers are the rows of the nodes as they appear in the `nodes` matrix.) The output variables: `newnodes` and `newtri` are the corresponding matrices for the refined partition.
(b) With $\mathscr{A}_0$ being the triangulation of the hexagonal domain of Example 13.1 (see Figure 13.5), apply your M-file to construct and plot the next three successive triangulations: $\mathscr{A}_1$, $\mathscr{A}_2$ and $\mathscr{A}_3$.
(c) With $\mathscr{A}_0$ being the triangulation of the annular domain of Example 13.3 (see Figure 13.16(c)), apply your M-file to construct and plot the next three successive triangulations: $\mathscr{A}_1$, and $\mathscr{A}_2$.
(d) Comment on the performance of this refinement scheme for domains with polygonal boundaries (as in part (b)) versus domains with curved boundaries (as in part (c)). Can you suggest any modifications to help the above scheme better represent boundaries in cases of curved domains? Property (i) should still be maintained, and (ii) should be "essentially maintained" in that the minimum angle of any element of $\mathscr{A}_n$ should not be too much smaller than the minimum angle of all elements of $\mathscr{A}_0$. For any such ideas, build them into a modified M-file and experiment on some domains.

26. (*Examples of FEM with Mesh Refinement*) (a) For the BVP of Example 13.5, set up MATLAB code to perform the FEM starting with the triangulation of that example, and then after refining the triangulation (as in Exercise 25), re-solving the problem on the new triangulation and looking at the absolute value of the difference of the new FEM solution with the previous FEM solution (on the previous grid). Continue to iterate this process until the absolute value of the difference is less than 1e-4 or the FEM calculations take more than a few minutes, whichever happens first. Plot the successive FEM solutions as well as the difference graphs.
(b) Repeat the instructions of part (a) on the BVP of Exercise 2; compare the final FEM

solution with the exact solution given there.
(c) Repeat the instructions of part (a) on the BVP of Exercise 3 (using the initial triangulation given there).

27. (*An Adaptive Scheme for the FEM*) This exercise develops an example of an adaptive scheme for the FEM. General adaptive schemes recursively solve a BVP with the FEM (starting with any triangulation of the domain) and then attempt to locate those elements where the error of the FEM solution is greatest. The mesh is next refined in a way that puts more nodes near the elements that were identified in the error estimation. This process is then iterated until some stopping criterion (a sufficiently small estimated error or difference in successive FEM approximate solutions) allows an exit. Here is a basic outline of one such scheme:
(i) Start with any triangulation of a domain and solve the given boundary value problem with the finite element method.
(ii) For each element, note its *oscillation* (= max value − min value of computed solution on three vertices).

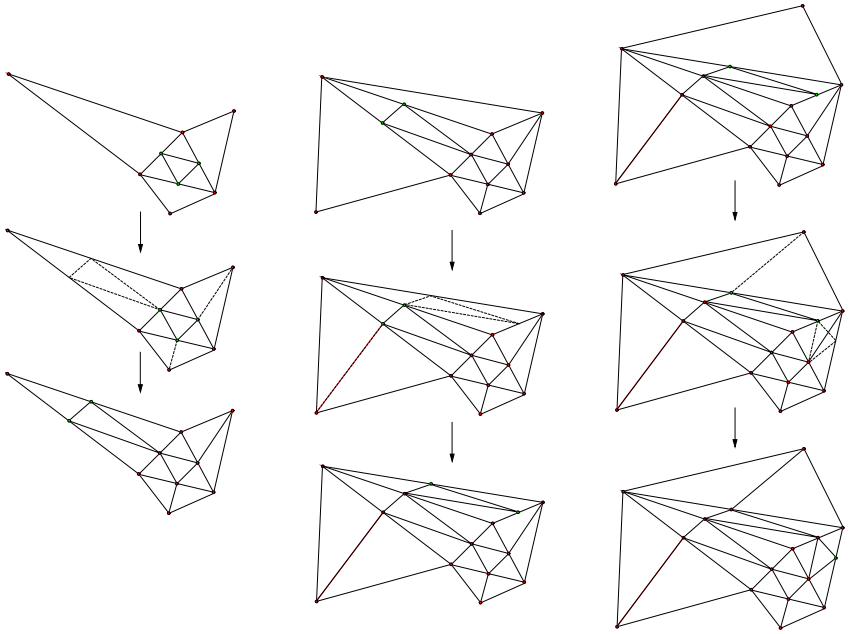


FIGURE 13.50: Illustration of adaptive mesh refinement scheme of Exercise 27. (a) (left) Step 1, (b) (middle) Step 2, and (c) (right) contingency plan for Step 3.

(iii) Flag those elements whose oscillations are "large" (with respect so some specified indicator, say more than double of the average).[17]
(iv) Refine the mesh accordingly so that each element flagged in (iii) gets split into three similar (triangular) elements as in Figure 13.49. Adjacent elements need to be refined accordingly so no hanging nodes remain. The two requirements are that the original node set is contained in the refined node set and no angle of any element gets too small (eccentricity requirement).
For definiteness, let us say that in (iii) the flagging criterion for elements is that the maximum

[17] We are using a rather basic error indicator. More sophisticated error indicators can be developed using advanced techniques of Sobolev spaces; see, for example, [CiLi-89], [Cia-02], or the classical reference [StFi-73] for details on such methods.

oscillation is more than double the average of all of the oscillations. In (iv) let us say that the eccentricity requirement stipulates that the minimum angle of any refinement cannot be less than 1/3 of the minimum angle, $\theta_{\min}$, of the original triangulation.

Balancing these two requirements makes the refinement scheme a delicate task. This sort of a scheme can be accomplished by iteratively applying a series of refinements that attempt (based on the two constraints) to isolate the "hanging nodes." We give an outline for such a scheme:

OUTLINE FOR ADAPTIVE MESH REFINEMENT SCHEME:
Step 1: After refining the flagged elements as in (iv), the new nodes introduced need to mesh into the next triangulation. Until they do become vertices of all adjancent elements, they will be referred to as "hanging nodes." Examine all neighboring elements of the flagged elements that were just refined; see Figure 13.50(a). If possible, we would like to contain the spread of green ("hanging nodes") but the problem is that we do not want any of the triangles to have very small angles. For each of the three neighbor triangles, if half the angle of the node opposite the hanging node is not too small ($< \theta_{\min}/3$), then simply split it into two triangles by joining the hanging node of the first triangle to the opposite node of the neighbor triangle (Figure 13.50(a) has two such triangles[18]). Otherwise, we are forced to refine the neighbor triangle as in (iv), but this introduces two new hanging (green) nodes. (Figure 13.50(a) has one of these).
Step 2: If Step 1 introduced any new hanging (green) nodes (as it did in Figure 13.50(a)), look at the neighboring triangles and try to contain the hanging nodes as in Step 1. We may again introduce hanging nodes. (Figure 13.50(b) illustrates this). We continue to iterate this step until there are no longer any hanging nodes. There is one contingency we need to mention (if a neighboring triangle runs into another that was already refined), this is illustrated in Figure 13.50(c); below we explain what to do in such situations.
Contingency plan for Step 3: Figure 13.50(c) illustrates what to do if a neighbor triangle runs into one that was already refined. We do not refine any triangle twice (this will give some control on the convergence of the algorithm and prevent the possibility of an infinite loop). Instead, we revert to the original refinement (three subtriangles instead of two) to take care of the internal green node; see Figure 13.50(c) .
(a) Write a MATLAB program that will perform the above adaptive scheme on the BVP and initial triangulation of Example 13.5. What happens when you run this program? Repeat, but now change the flagging criterion in (iii) to be that the oscillation of the FEM solution over an element exceeds 1/10. Repeat with 1/10 replaced by 1/100. Plot each refined mesh as well as the final FEM solution.
(b) Repeat the instructions of part (a) on the BVP of Exercise 2; compare the final FEM solution with the exact solution given there.
(c) Repeat the instructions of part (a) on the BVP of Exercise 3 (using the initial triangulation given there).
(d) Do you have any ideas for an alternative mesh refinement scheme (satisfying the two constraints mentioned above)?

28. (*Obtuse Angles in the Domain Are Sometimes Problematic for the FEM*) Engineers have known for some time, and mathematicians subsequently confirmed theoretically, that obtuse corners in the domain of a BVP can often slow down the convergence of the FEM near the boundary points with obtuse angles (see Section 8.1 of [StFi-73] or Section 5.6 of [AxBa-84]) . Simple examples of domains with such obtuse angles are shown in Figure 13.51(b), (c). In general, the larger the obtuse angle, the greater the possible problems with the FEM. The extreme case is with an interior angle of 2π physically corresponding to a crack, fissure, or material interface in the domain; see Figure 13.51(c). This exercise will investigate such phenomena and explore stategies to mitigate problems that might arise. We will examine a certain Dirichlet problem for

---

[18] Note that at the first iteration, this could not occur with the stated eccentricity requirement since bisecting any of the original angles would result in angles at least as large as $\theta_{\min}/3$; so this pathology in the figure could only occur in later iterations. In particular, for the first refinement, all hanging nodes could be isolated in Step 1.

the Laplace equation on such a domain where the exact solution is known. For any angle ω, where $0 < \omega \le 2\pi$, we let Ω_ω denote the subdomain of all points in the unit square $-1 \le x, y \le 1$ whose polar coordinates (r,θ) satisfy $0 < \theta < \omega$. Thus the domains of Figure 13.51 are all examples of such domains. In particular, the domain in Figure 13.51(b) is $\Omega_{3\pi/2}$ and that of Figure 13.51(c) is $\Omega_{2\pi}$.

(a) Show that on any such domain Ω_ω the function (given in polar coordinates) $u(r,\theta) = r^{\pi/\omega}\sin(\pi\theta/\omega)$ is harmonic (i.e., satisfies the Laplace equation $\Delta u = 0$) and vanishes on the *angular edges* (i.e., the rays of the angle emanating from the point *O*; see Figure 13.51.).

(b) For each of the domains in Figure 13.51 (for the one in Figure 13.51(a) use $\omega = 2\pi/3$), apply the FEM to solve BVP consisting of the Laplace equation with the boundary conditions *u* = 0 on the angular edges of the boundary and $u(r,\theta) = r^{\pi/\omega}\sin(\pi\theta/\omega)$ on the remaining portion of the boundary. Of course, you will need to convert the latter boundary conditions into cartesian (*x*,*y*) coordinates. Start off with the corresponding triangulations shown in Figure 13.52. Then apply the algorithm of Exercise 25 to successively refine the triangulations and re-solve with the FEM. For each triangulation, plot the exact error using the exact solution in part (a). Go through three refinements for each domain.

(c) Repeat part (b) for each of the three BVPs given there, but this time using the adaptive scheme of Exercise 27 in place of the refinement scheme of Exercise 25.

(d) Using the special form of the triangulations given, can you think of a more convenient refinement scheme for this problem? Make up a reasonable one and test it out for several iterations comparing with the exact solution at each step.

Suggestion: An elegant and illuminating way to do part (a) is to derive the Laplace operator in polar coordinates to be: $u_{xx} + u_{yy} = u_{rr} + (1/r)u_r + (1/r^2)u_{\theta\theta}$.

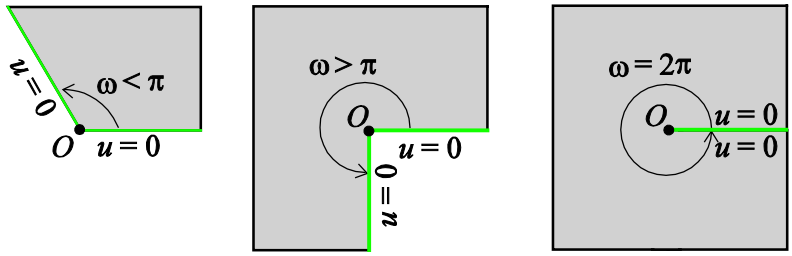


FIGURE 13.51: Simple examples of domains with different sorts of angles at a boundary point *O*. (a) (left) In general acute angles do not pose any problems for the FEM. (b) (middle) Obtuse angles can sometimes lead to slower convergence of the FEM. (c) (right) The larger the obtuse angle, the greater the potential difficulty. The extreme case is the slit domain. The indicated homogeneous Dirichlet boundary conditions on the angular edges is relevant for Exercise 28.

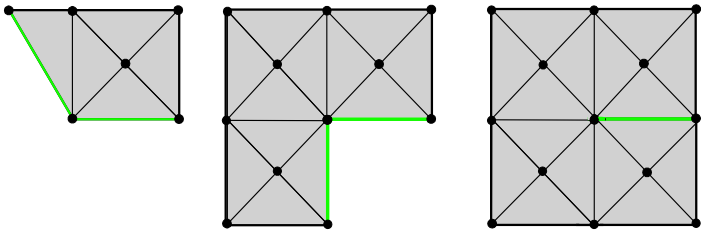


FIGURE 13.52: Initial triangulation for the domains of Figure 13.51 for Exercise 28.

29. Suppose that the FEM of this section is used to compute the solution of a BVP of form (10) whose exact solution is known to be a linear function $u(x, y) = ax + by + c$. Assume the integrals are all computed exactly and that the domain and triangulation are such that the boundary of the domain consists entirely of edges of the triangulation. Will the FEM solution always coincide with the exact solution? Either explain whether this is true or, if you are unable to do so, perform a series of numerical experiments to test this hypothesis.
Note: Since the basis functions are piecewise linear, this seems to be the most general type of solutions that the FEM might be able to produce exactly. An example of such a BVP and triangulation is given in Exercise 1.

CHAPTER 13: THE FINITE ELEMENT METHOD

EFR 13.1: For Φ_3, the code given in Example 13.1 just needs a small modification to accommodate the change of node. Indeed, in the for loop, the three modified lines are: `if ismember(3,T(L,:))==1,index=find(T(L,:)==3);nv=[T(L,1:2)3]; nv(index)=T(L,3);`. The new output of the modified loop is then the following matrix A:

```
→A =    1    -1    0    1
        5    -1    0    1
```

From this we can write:

$$\Phi_3(x,y)=\begin{cases} -x+1, & \text{if } (x,y)\in T_1,\\ -x+1, & \text{if } (x,y)\in T_5,\\ 0, & \text{otherwise.}\end{cases}$$

In a similar fashion, we find that:

$$\Phi_4(x,y)=\begin{cases} \frac{2}{3}x-y-\frac{2}{3}, & \text{if } (x,y)\in T_3, \quad -x-y+\frac{7}{2}, \quad \text{if } (x,y)\in T_4,\\ \frac{2}{3}x-\frac{2}{3}, & \text{if } (x,y)\in T_6, \quad y+1, \quad \text{if } (x,y)\in T_7,\\ -x+y+\frac{7}{2}, & \text{if } (x,y)\in T_8, \quad 0, \quad \text{otherwise.}\end{cases}$$

EFR 13.2: (a) As a quadrilateral has four vertices and a linear function in x and y has only three parameters (see equation (2)), linear functions are not versatile enough to accommodate arbitrarily specifying four numerical values at the vertices of a quadrilateral.

(b) A generic function of x and y having four parameters is a so-called bilinear function: $axy+bx+cy+d$, these are often used for quadrilateral elements. In the special case where the elements are rectangles parallel to the axis, the matter is further discussed in some of the exercises at the end of Section 13.2.

EFR 13.3: (a) The M-file is boxed below:

```
function voronoiall(x,y)
% M-file for EFR 13.3
% inputs:  two vectors x and y of the same size giving, respectively,
% the x- and y-coordinates of a set of distinct points in the plane;
% outputs:  none, but a graphic will be produced of the Voronoi
% regions corresponding to the point set in the plane,
% including the unbounded regions
n=length(x);
xbar = sum(x)/n; ybar = sum(y)/n; %centroid of points
md = max(sqrt(x-xbar).^2 + sqrt(y-ybar).^2);
%maximum distance of points to centroid
mdx = max(abs(x-xbar)); mdy = max(abs(y-ybar));
%max x- and y- distances to averages
% We create additional points that lie in a circle of radius 3md
% about (xbar, ybar).  We  deploy them with angular gaps of 1 degree,
% this will be suitable for all practical purposes.
xnew=x; ynew=y;
for k = 1:360
    xnew(n+k)=xbar+3*md*cos(k*pi/180);
    ynew(n+k)=ybar+3*md*sin(k*pi/180);
end
voronoi(xnew,ynew)
axis([min(x)-mdx/2 max(x)+mdx/2  min(y)-mdy/2  max(y)+mdy/2])
```

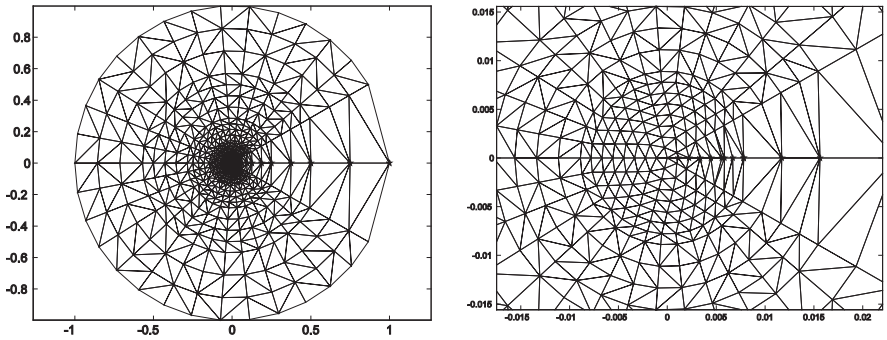
(b) With this program, we can easily re-create Figure 13.9(b):

```
>> N=[1 1;5/2 1;0 0;1 0;5/2 0;7/2 0;1 -1;2.5 -1];x=N(:,1); y=N(:,2);
>> voronoiall(x,y)
```

EFR 13.4: (a) Although the scheme we used in the solution of part (c) of Example 13.2 can be adapted for this triangulation, we will introduce a slightly different approach. Specifically, we will take advantage of the fact that intersections of circles (centered at (0,0)) all have the same boundary angles. At each iteration, we will deploy nodes on circles of equally spaced radii in the annular sector domains: $\Omega_n = \{(x, y) \in \Omega : 1/2^n < \text{dist}((x, y),(0,0)) < 2 \cdot (1/2^n)\}$ for n = 1, 2, … By the special shape of Ω, we get the following exact formula for the area of Ω_n: $\text{Area}(\Omega_n) = \frac{1}{2} \cdot \frac{5\pi}{3}[(2 \cdot 2^{-n})^2 - (2^{-n})^2] = \frac{5\pi}{2} 2^{-2n}$. Thus, if we were to deploy (approximately) 100 nodes in Ω_n with a uniform grid, the gap size s should (approximately) satisfy: $100s^2 = \frac{5\pi}{2} 2^{-2n}$ or $s = \sqrt{\pi/40} \cdot 2^{-n}$. We use this for the gaps between radii, and, on average, arrange for a similar gap size between adjacent nodes on a given circle of deployment. Since the domain is not convex, we will use additional ghost nodes (as in Example 13.3) to help us detect and remove unwanted elements.

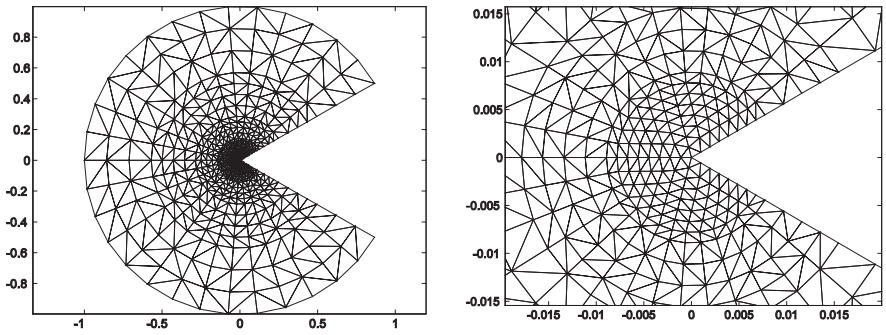
```
%Script for EFR 13.4a
count=1;
for n=1:7
s=sqrt(pi/40)/2^n;
len = 5*pi/2/2^n; %avg. arclength of node circular arc in Omega_n
nnodes= ceil(len/s); %number of nodes to put on each circular arc
ncirc = ceil(1/2^n/s);
%number of circlular arcs w/ to put nodes on Omega_n
rads = linspace(2/2^n, 1/2^n+s/2, ncirc);
%radii of circular arcs with nodes
angles = linspace(pi/6, 11*pi/6, nnodes); %angles for node deployment

%deploy nodes:
for r=rads
    for theta = angles
        x(count)=r*cos(theta); y(count)=r*sin(theta); count=count+1;
    end
end
end
%the final portion takes a slightly different approach since we want
%to deploy nodes throughout the whole sector (not just the annulus).
%We will thus want the circles of deployment to have radii all the
%way down to s(gap size), but on the smaller circles we should deploy
%less nodes
n=8; s=sqrt(pi/40)/2^n;
len = 5*pi/2/2^n;
%avg. arclength of node circular arc in Omega_n-outer circles
nnodes= ceil(len/s);
%number of nodes to put on each outer circular arc
rads = linspace(2/2^n, 0, ceil(2/2^n/s));
%radii of circular arcs with nodes
angles = linspace(pi/6, 11*pi/6, nnodes); %angles for node deployment

%delploy nodes
for r=rads
    for theta = linspace(pi/6, 11*pi/6, ceil(len/s*r/(2/2^n)))
        x(count)=r*cos(theta); y(count)=r*sin(theta); count=count+1;
    end
end

% Put in extra ghost nodes to detect bad elements
% There are several ways to do this, we will deploy them in a
% sufficient pattern on the positive x-axis.
```

```
nnodes=count-1; %number of nodes (=932)
for k=0:7
  x(count)=1/2^k; y(count)=0; count=count+1;
  x(count)=.75/2^k; y(count)=0; count=count+1;
end
for k=rads
    if k>0
     x(count)=k; y(count)=0; count=count+1;
 end
end
tri = delaunay(x,y);
%The following two commands will plot the triangulation containing
%the ghost nodes, the latter indicated by pentacles.  This plot and a
%magnification are shown in the %figure below.
```



By the way that the ghost nodes were deployed, the unwanted elements are precisely those that have a ghost node as one of their vertices. The remaining code will search and destroy these elements, it is modeled after that of Example 13.3. The final triangulation and a zoomed view are shown in the two figures below.

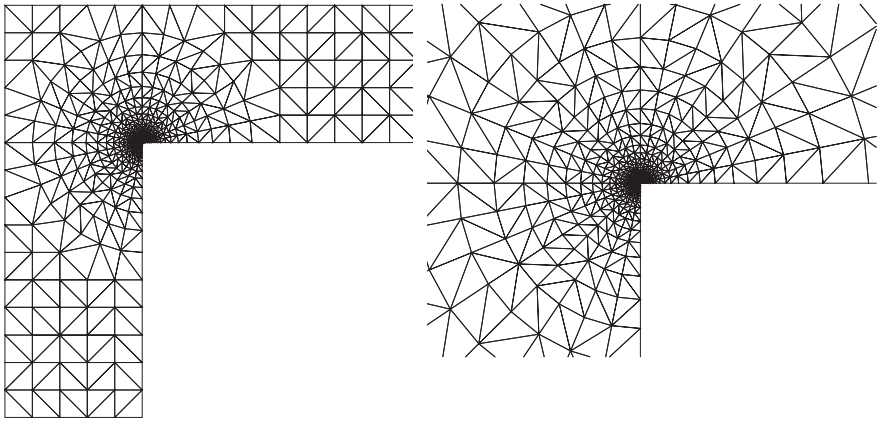
```
plot(x(nnodes+1:count-1), y(nnodes+1:count-1), 'rp')
hold on, trimesh(tri,x,y), axis('equal')
%>> size(tri)
%ans =
%       1876           3
badelcount=1;
for ell=1:1876
    if max(ismember(nnodes+1:count-1, tri(ell,:)))
        badel(badelcount)=ell;
        badelcount=badelcount+1;
    end
end
clf
tri=tri(setdiff(1:1876,badel),:);
x=x(1:nnodes); y=y(1:nnodes);
trimesh(tri,x(1:nnodes),y(1:nnodes)), axis('equal')
```



(b) We take the vertices of the domain to be: (0, 0), (2,0), (2,1), (–1, 1), (–1, –2), and (0, –2). The code below presents yet another variation of node deployment schemes. The crucial part (Stage 2 in the code below) is the deployment of nodes inside the circle with center (0, 0) and radius 0.8. We put an equal number of nodes (13) on each such circle. Because of the exponential decay of the radii, the gaps between radii remain close to the arclength gaps on the corresponding circles. The code below uses ghost nodes and they can be viewed by executing the code up to the line with the first `trimesh` command (as in part (a)). We show only a figure of the final triangulation along with a zoomed view (without axes).

```
%Script for EFR 13.4b
%We deploy the nodes in three stages
%Stage 1:  Outside the circle of radius 1, center (0,0), squarelike
%grid with gap size s = 0.2
%We can do boundary and interior nodes together:
count=1;
for xt=-1:.2:2
    for yt=-2:.2:1
        pt=[xt yt]; %test point
        if norm(pt,2)>.8+.1 & ~(xt>0 & yt<0)
      %these conditions ensure the test point is in the domain and a
      %safe distance from the boundary of the outer circle of Stage 2
            x(count)=xt; y(count)=yt; count=count+1;
        end
    end
end
%Stage 2:  Put nodes on concentric circles with exponential decay of
%radii
angles=0:pi/16:3*pi/2;
%this vector of angles will not change in the loop
for k=1:40
    r=.8^k;
    for theta = angles
     x(count)=r*cos(theta); y(count)=r*sin(theta); count=count+1;
     if k==0 & (x(count-1)<-.95|y(count-1)>.95)
     count=count-1; end %discard points too close to domain boundary
    end
end
%Stage 3:  Put nodes on the inside of the last circle of Stage 2
gap=3*pi/4*r/13;
%approx. gap size gotton by dividing arclength of last circle
%by number of nodes that were put on it
xvec=linspace(-r,r,2*ceil(r/gap)+1); yvec=xvec;
for xt=xvec
    for yt=yvec
     pt=[xt yt]; %test point
        if norm(pt,2)<=r-gap/2 & ~(xt>0 & yt<0)
```

```
          %these conditions ensure the test point is in the domain
          %and a safe distance from the boundary of the circle
            x(count)=xt; y(count)=yt; count=count+1;
        end
    end
end
%plot(x,y,'rp')
tri = delaunay(x,y);
hold on, trimesh(tri,x,y), axis('equal')

% Now we put in extra ghost nodes to detect bad elements
% There are several ways to do this, we will deploy them in a
% sufficient pattern on the ray theta = - pi/4
nnodes=count-1; %number of nodes
x(count)=1; y(count)=-1; count=count+1;
for k=0:40
x(count)=.8^k*cos(-pi/4); y(count)=.8^k*sin(-pi/4); count=count+1;
end
for k=r:-gap:gap
x(count)=k*cos(-pi/4); y(count)=k*sin(-pi/4); count=count+1;
end
x(count)=gap/2*cos(-pi/4); y(count)=gap/2*sin(-pi/4); count=count+1;
tri = delaunay(x,y);
clf, plot(x(nnodes+1:count-1), y(nnodes+1:count-1), 'rp')
hold on, trimesh(tri,x,y), axis('equal')
size(tri)
%ans =
%       2406           3
badelcount=1;
for ell=1:2406
    if max(ismember(nnodes+1:count-1, tri(ell,:)))
        badel(badelcount)=ell;
        badelcount=badelcount+1;
    end
end
clf
tri=tri(setdiff(1:2406,badel),:);
x=x(1:nnodes); y=y(1:nnodes);
trimesh(tri,x,y), axis('equal'),  axis off
```



EFR 13.5: (a) In multivariable calculus, it is proved that the gradient vector of a function of two variables points in the direction in which the partial derivative is maximum and has magnitude equal to this maximum partial derivative. Also, the gradient is perpendicular to the direction in which the partial derivatives are zero. Since ϕ is a linear function, its gradient is a constant vector. Since the function ϕ is zero on the line joining v_1 and v_2, it follows that the gradient must be perpendicular to this side of T and therefore it must be parallel to $\vec{a}$ (the opposite direction would have negative partial derivative). The magnitude of the partial derivative, since the function is linear, can be gotten by taking the difference quotient of the values of ϕ at the tip and tail of $\vec{a}$ over the length of the vector $\vec{a}$ and this completes the proof of (a).
(b) The integral will be unchanged if we perform a rotation change of variables (which has Jacobian = 1), so we may assume that the line joining v_1 and v_2 is the x-axis. Write $v_1 = (a,0)$, $v_2 = (b,0)$ and let c denote the x-coordinate of v_3. Thus $a \le c \le b$ and $\phi(x,y) = y / \| \vec{a} \|$. Assume first that $a < c < b$. The height of the triangle at any value of $x \in [a,b]$ is given by:

$$h(x) = \begin{cases} \dfrac{\| \vec{a} \|}{c} \cdot \left(\dfrac{x-a}{c-a} \right), & \text{if } x \le c, \\ \| \vec{a} \| \cdot \left(1 - \dfrac{x-c}{b-c} \right), & \text{if } x > c. \end{cases}.$$

Thus we may compute:

$$\iint_T \phi(x,y)\,dxdy = \int_a^b \int_0^{h(x)} \frac{y}{\| \vec{a} \|}\,dydx = \int_a^b \frac{h(x)^2}{2}\,dx = \frac{\| \vec{a} \|}{2} \int_a^c \left(\frac{x-a}{c-a} \right)^2 dx + \frac{\| \vec{a} \|}{2} \int_c^b \left(1 - \frac{x-c}{b-c} \right)^2 dx.$$

These two integrals are easily done by u-substitution. In the first one, we let $u = \dfrac{x-a}{c-a}$, so $du = \dfrac{dx}{c-a}$ and the integral becomes: $\displaystyle\int_a^c \left(\frac{x-a}{c-a} \right)^2 dx = (c-a) \int_0^1 u^2 du = \frac{1}{3}(c-a)$. Similarly, the second integral is $\dfrac{1}{3}(b-c)$; combining these gives $\displaystyle\iint_T \phi(x,y)\,dxdy = \frac{1}{3} \frac{\| \vec{a} \|}{2} (b-a) = \tfrac{1}{3} \text{Area}(T)$, as asserted. In the remaining case that $c = a$ or $c = b$ (so the triangle is a right triangle), the function $h(x)$ can be written as a single formula and the above proof simplifies.

EFR 13.6: If u_{old} denotes the exact solution of the BVP of Example 13.5, we let $u_{\text{new}} \equiv u_{\text{old}} + 1$. Certainly u_{new} satisfies the PDE $-\Delta u = f(x,y)$ since u_{old} does. Also, since $u_{\text{old}} \equiv 1$ on the boundary, we get that $u_{\text{new}} \equiv 2$ on the boundary. Thus u_{new} will solve the modified BVP. Since the coefficients of the stiffness matrix (see (15^{ℓ})) do not depend on the boundary values, this matrix A will be the same for both problems. The only change will be in the load vector coefficients; now (16^{ℓ}) takes on the following form:

$$b_\alpha^\ell = \iint_{T_\ell} f \Phi_{i_\alpha}\,dxdy - 2 \sum_{s \ne 4,5} \iint_{T_\ell} \nabla \Phi_s \bullet \nabla \Phi_{i_\alpha}\,dxdy \quad (1 \le \alpha \le 3).$$

The only difference from the example is the presence of the factor of 2. Since the computations leading to the load vector b parallel very closely those of Example 13.5, we simply summarize the element-by-element updates of the vector b:

$$\ell=1:\ b=\begin{bmatrix}2\\0\end{bmatrix},\ \ell=2:\ b=\begin{bmatrix}2+3/2\\0\end{bmatrix}=\begin{bmatrix}7/2\\0\end{bmatrix},\ \ell=3:\ b=\begin{bmatrix}7/2\\0+3/2\end{bmatrix}=\begin{bmatrix}7/2\\3/2\end{bmatrix},$$

$$\ell=4:\ b=\begin{bmatrix}7/2\\3/2+11/6\end{bmatrix}=\begin{bmatrix}7/2\\10/3\end{bmatrix},\ \ell=5:\ b=\begin{bmatrix}7/2+2\\10/3\end{bmatrix}=\begin{bmatrix}11/2\\10/3\end{bmatrix},\ \ell=6:\ b=\begin{bmatrix}11/2+3/2\\10/3\end{bmatrix}=\begin{bmatrix}7\\10/3\end{bmatrix},$$

$$\ell=7:\ b=\begin{bmatrix}7\\10/3+3/2\end{bmatrix}=\begin{bmatrix}7\\29/6\end{bmatrix},\ \ell=8:\ b=\begin{bmatrix}7\\29/6+11/6\end{bmatrix}=\begin{bmatrix}7\\20/3\end{bmatrix}.$$

If we solve the resulting matrix equation $Ax = b$; we get (to 4 decimals) $x(1) = 1.9869$, and $x(2) = 1.9179$. Comparing with the solutions of the original system: $x(1) = 0.9278$ and $x(2) = 1.0484$, we see that the numerical solutions are somewhat close, but definitely do not differ by one. With finer triangulations, the exact relationship would be made more apparent.

EFR 13.7: (a) We first draw a picture of the region S on which the integration is to take place, and realize it lying between two functions of x; see the left figure below. It is convenient to break the integral up into two pieces since the top function experiences a formula change at $x = \cos(\pi/4)$.

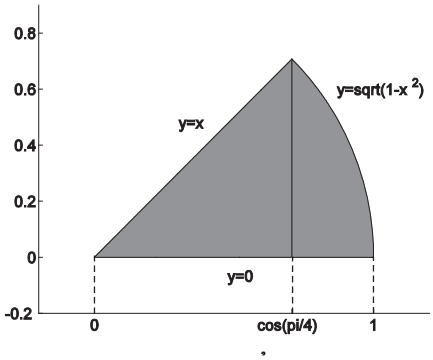
```
>> syms x y
>> Int_A = quad2d(x*y^2,0, cos(pi/4), 0, x)+quad2d(x*y^2,cos(pi/4),
1, 0, sqrt(1-x^2))
```

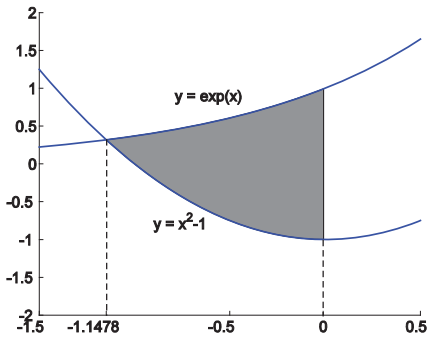
→ Int_A = 0.0236 (This is the numerical approximation to the first integral in "format short.")

(b) The picture, shown on the right below, shows that the curves intersect when x is negative. We first find this intersection point:

```
>> xmin = fzero(inline('x^2-1-exp(x)'), -1)
```
→ xmin =-1.1478

```
>> Int_B = quad2d(exp(1-x^2-2*y^2),xmin, 0, x^2-1, exp(x))
```
→Int_B = 2.0661





Note: Both plots were created in MATLAB using the built-in function `patch`. The syntax is as follows:

`patch([x xrev], [flow ftop_rev], [r g b])` →	This command will produce a graphic of a shaded region between two functions. Here `x` is a vector of x-coordinates for an interval on which the corresponding vectors `flow` and `ftop_rev` represent two functions. The function represented by `flow` has its graph lying below the one represented by `ftop_rev`. The syntax requires also as input the vector `xrev` that is the vector `x` taken in reverse order. The vector `ftop_rev` (representing the top function) correspondingly needs to be inputted in reverse order. The final input is a 3×1 rgb vector of numbers between 0

	and 1 that will determine the color of the patch.

As an example, we give the code used to create the second plot:

```
>> x=xmin:.01:0;
>> for i=1:length(x), xrev(i)=x(length(x)+1-i); end
>> patch([ x xrev], [x.^2-1  exp(xrev)], [.5 .5 .5]), hold on
>> t = -1.5:.01:.5;  plot(t,t.^2-1,'b'), plot(t,exp(t),'b'),
>> axis([-1.5 .5 -2 2])
>> gtext('y = exp(x)') %use mouse to place text on graphic window
>> gtext('y = x^2-1') %use mouse to place text on graphic window
```

Some embellishments were done to the graph using menu options on the graphics window.

EFR 13.8: (a) The M-file is boxed below:

```
function integ=triangquad2d(fun,v1,v2,v3)
% M-file for EFR 13.8.  This function will integrate a function
% of two variables x and y over a triangle T in the plane.
% It uses the M-file 'quad2d' of Program 13.1
% Input variables:  fun = a symbolic expression (using one or both of
% the symbolic variables x and y,  v1, v2, and v3:  three length 2
% vectors giving the vertices of the triangle T in the plane.
% NOTE: Before this program is used, x and y should be declared as
% symbolic variables.syms x y u
vys = [v1(2) v2(2) v3(2)];
vxs = [v1(1) v2(1) v3(1)];
minx=min(vxs); maxx=max(vxs);
miny=min(vys);
minyind =find(vys==miny);
minxind =find(vxs==minx);
maxxind =find(vxs==maxx);
if length(minxind)==2|length(maxxind)==2 %triangle has a vertical
side
    if length(minxind)==2
        vertx=minx;
        vertymax=max(vys(minxind));
        vertymin=min(vys(minxind));
    else
        vertx=maxx;
        vertymax=max(vys(maxxind));
        vertymin=min(vys(maxxind));
    end
    thirdind = find(vxs~= vertx);
    topslope=sym((vys(thirdind)-vertymax)/(vxs(thirdind)-vertx));
    botslope=sym((vys(thirdind)-vertymin)/(vxs(thirdind)-vertx));
    ytop=topslope*(x-vxs(thirdind))+vys(thirdind);
    ylow=botslope*(x-vxs(thirdind))+vys(thirdind);
    integ = quad2d(fun,minx,maxx,ylow,ytop);
else %no vertical sides so vertices have 3 different x coordinates
    midind = find(vxs>minx& vxs<maxx); %index of middle vertex
    longslope=sym((vys(maxxind)-vys(minxind))/(maxx-minx));
    ylong = longslope*(x-vxs(minxind))+vys(minxind);
    if vys(midind)>subs(ylong,x,vxs(midind));
    %long edge lies below mid vertex
     topleftslope = sym((vys(midind)-vys(minxind))/(vxs(midind)-...
                    vxs(minxind)));
     toprgtslope = sym((vys(midind)-vys(maxxind))/(vxs(midind)- ...
                    vxs(maxxind)));
     ytopleft = topleftslope*(x-vxs(midind))+vys(midind);
     ytoprgt = toprgtslope*(x-vxs(midind))+vys(midind);
     integ = quad2d(fun,minx, vxs(midind),ylong,ytopleft)+...
```

```
          quad2d(fun,vxs(midind),maxx,ylong,ytoprgt);
   else %long edge lies above mid vertex
     botleftslope = sym((vys(midind)-vys(minxind))/(vxs(midind)- ...
                        vxs(minxind)));
     botrgtslope = sym((vys(midind)-vys(maxxind))/(vxs(midind)- ...
                       vxs(maxxind)));
     ybotleft = botleftslope*(x-vxs(midind))+vys(midind);
     ybotrgt = botrgtslope*(x-vxs(midind))+vys(midind);
     integ = quad2d(fun,minx,vxs(midind),ybotleft,ylong)+...
         quad2d(fun,vxs(midind),maxx,ybotrgt,ylong);
   end
end
```

(b) To recalculate the integrals of Example 13.5, the following commands will suffice and result in the same outputs that were obtained in the example:

```
>> syms x y
>> v1 = [1 3]; v2 = [5 1]; v3 = [4 6]; %Triangle of Example 13.5
>> triangquad2d(2*x*y^2,v1,v2,v3) →ans = 724.8000
>> triangquad2d(sin(x*y*sqrt(y)),v1,v2,v3) →ans = 0.1397
```

The remaining integrals can be done in the same swift fashion. We store separately the vertices of the two triangles *T*1 and *T*2:

```
>> v1 = [0 0]; v2 = [6 0]; v3 = [12 2]; %Triangle T1 of EFR 13.8
>> V1 = [1 3]; V2 = [3 2]; V3 = [2 5]; %Triangle T2 of EFR 13.8
>> int_1 = triangquad2d(1,v1,v2,v3) →int_1 = 6
>> int_2 = triangquad2d(1,V1,V2,V3) → int_2 = 2.5000
>> int_3 = triangquad2d(2*x^2,v1,v2,v3) → int_3 = 504
>> int_4 = triangquad2d(sin(x^2),V1,V2,V3) → int_4 = -0.2998
```

These numerical calculations are all in agreement with the exact answers that were provided.

EFR 13.9: We will use modification of the method used in part (c) of Example 13.2. In that example, a similar node deployment was required on the same domain, except that there we wanted more nodes to be focused near the boundary point (1,0) and here we want the focus area to be the boundary point (cos(3), sin(3)). What we will do is very slightly modify the node deployment code of the example (since here we want less nodes) and then simply rotate the node set by an angle of $\theta = 3$ (using the rotation transformation of Section 7.2). The rotation idea is quite a natural one; it could be circumvented, but then we would need a more serious modification of the code of the example. Since the codes are long, we indicate only the changes needed for the present problem.

Referring to the notations of the solution of part (c) of Example 13.2, in the determination of the gap size s to use in the region Ω_n, we will use roughly 10 nodes (rather than 100) per such region, so s should satisfy $10 \cdot s^2 \le \text{Area}(\Omega_n) \le \frac{3\pi}{2} 2^{-2n}$ or $s \le \sqrt{3\pi/20} \cdot 2^{-n}$. If we then run through 8 iterations (n runs from 0 to 7) of deploying nodes just as in the example, we see that the nodes are a bit sparse in the first two regions. To mitigate this, we make s a bit smaller in the first two iterations. If we replace the first three lines of the node deployment code of the example with the following four lines (and run the rest of the code), we will arrive at a triangulation that looks quite appropriate.

```
>> n=0; nodecount=1;
>> while n<8
s=sqrt(3*pi/20)/2^n;
if  n==0, s = s/3; elseif n ==1, s=s/2; end
```

This node set should now be rotated by an angle of $\theta = 3$, and this is done using the rotation matrix of Section 7.2:

```
>> Rot=[cos(3)  -sin(3); sin(3) cos(3)]*[x; y];
>> xn = Rot(1,:); yn = Rot(2,:);  %newly rotated nodes for desired
triangulation.
```

Now, in order to be able to more easily use the assembly code of Example 13.7, we should reorder the nodes so that the boundary nodes appear last. This is accomplished with the following commands:

```
bdyind = find(xn.^2 + yn.^2 > 1 - 10*eps);
size(xn), size(bdyind)  →  1   123,   1    38
```

```
intind = setdiff(1:123, bdyind); %these are the indices of interior
nodes
xn = [xn(intind) xn(bdyind)]; %reordered x-coordinates of nodes
yn = [yn(intind) yn(bdyind)]; %reordered x-coordinates of nodes
tri=delaunay(xn,yn);
trimesh(tri,xn,yn), axis('equal') %triangulation is shown below left
```

We now store the boundary values:

```
for i=86:123
th=cart2pol(x(i),y(i));
if th<0, th=th+2*pi; end
%need to ensure th is in domain of boundary data function
c(i)=ex_13_7_bdydata(th);
end
```

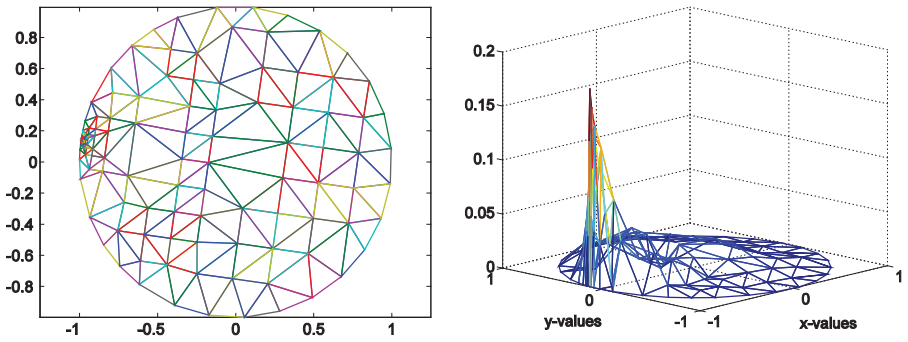
The assembly code of the example will now work very well in this situation; we need only change the first three lines as follows:

```
N=[xn' yn'];
E=tri;
n=85; m=123; syms x y
```

When this and the rest of the code is run, we will have created the numerical solution's values stored as a vector `c`. The exact solution's values can be created just as in the example (the code is verbatim) and stored as a vector `cp`. This being done, the following command will plot the error of the numerical solution; the plot is shown on the right.

```
>> trimesh(E,xn,yn,abs(c-cp))
```

Notice that the maximum error is seen to be smaller than that obtained in part (b) of the example (cf. Figure 13.38b), using a lot less nodes but a more appropriate node deployment strategy.



EFR 13.10: (a) The M-file is boxed below:

```
function int = gaussianintapprox(f,V1,V2,V3)
% M-file for numerically approximating integral of a function f(x,y)
% over a triangle in the plane with vertices V1, V2, V3
% Approximation is done using the Gaussian quadrature formula (24)
% of Chapter 13.
% Input Variables:  f = an inline function or an M-file of the
% integrand specified as a function of two variables: x and y
% V1, V2, V3 length 2 row vectors containing coordinates of the
% vertices of the triangle.  Output variable: int = approximation
A=feval(f,(V1(1)+V2(1))/2,(V1(2)+V2(2))/2);
B=feval(f,(V1(1)+V3(1))/2,(V1(2)+V3(2))/2);
C=feval(f,(V2(1)+V3(1))/2,(V2(2)+V3(2))/2);
M=[V1 1;V2 1; V3 1];
```

```
area=abs(det(M))/2;  %See formula (5) of Chapter 13
int=area*(A+B+C)/3;
```

(b) After creating and storing the triangulation for part (c) of Example 13.7, and then the boundary values (just as was done in the example), the first three lines of the assembly code should read as follows:

```
>> N=[x' y'];
>> E=tri;
>> A=zeros(n); b=zeros(n,1);
```

(Same as before, except now we do not need symbolic variables.) The rest of the assembly code only needs changing in the two places where the numerical integrator `triangquad2d` was used. To save space, we include the relevant modified passages here; the ftp site for this book includes a file for the complete code.

```
%update stiffness matrix
for i1=1:length(intnodes)
for i2=1:length(intnodes)
fun1 = num2str(intgrad(i1,:)*intgrad(i2,:)',10); %integrand for
(15ell)
fun=inline(fun1,'x', 'y');
integ=gaussianintapprox(fun,xyt,xyr,xys);
A(intnodes(i1),intnodes(i2))=A(intnodes(i1),intnodes(i2))+integ;
end
end
%update load vector
for i=1:length(intnodes)
for j=1:length(bdynodes)
fun1 = num2str(intgrad(i,:)*bdygrad(j,:)',10); %integrand for (16ell)
fun=inline(fun1,'x', 'y');
integ=gaussianintapprox(fun,xyt,xyr,xys);
b(intnodes(i))=b(intnodes(i))-c(bdynodes(j))*integ;
end
end
```

Whereas the original code took about an hour to run (on the author's computer), the modified assembly code took only a few seconds. Moreover, an examination of the error plot (against the exact Poisson solution) shows the errors of the two methods to be about the same.

EFR 13.11: For completeness, we include a full code for the FEM. Assume that the node set (vectors x and y) have been constructed as in Example 13.3. Although from the construction it is clear that the nodes on the inner circle came first and those on the outer circle came last, before we triangulate, we give a code that will automatically reindex so that the interior nodes precede the boundary nodes:

```
m = length(x); %m = total number of nodes.
cnt1=1; cnt2=1;
for i=1:m
if norm([x(i) y(i)],2)<1+4*eps %tests if node is on inner circle
bdy1(cnt1)=i; cnt1=cnt1+1;
elseif norm([x(i) y(i)],2)>2-4*eps %tests if node is on outer circle
bdy2(cnt2)=i; cnt2=cnt2+1;
end
end
n=m-length(bdy1)-length(bdy2); % n = total number of interior nodes
xnew=[x(setdiff(1:m, union(bdy1, bdy2))) x(bdy1) x(bdy2)]; x=xnew;
ynew=[y(setdiff(1:m, union(bdy1, bdy2))) y(bdy1) y(bdy2)]; y=ynew;
```

Next, we form the Delaunay triangulation using the code of Example 13.3. This being done, we assign the boundary values:

```
c(n+1:n+length(bdy1))=2;
for i=n+length(bdy1)+1:m
th=cart2pol(x(i),y(i));
c(i)=cos(2*th);
end
```

The remaining code below will perform the FEM and create the plot of the numerical solution (Figure 13.39). We use Method 2 (Gaussian quadrature for the integrals). We include the code for completeness only, but because of the way we have prepared things, the remaining code is identical to that of the preceding EFR (if we had written it out there):

```
N=[x' y'];
E=tri;
A=zeros(n); b=zeros(n,1);
[L cL]=size(E);
for ell=1:L
nodes=E(ell,:);
bdynodes=nodes(find(nodes>n));
intnodes=setdiff(nodes,bdynodes);

%find gradients [a b] of local basis functions
% ax + by +c; distinguish between int node
%local basis functions and bdy node local basis
%functions

for i=1:length(intnodes)
xyt=N(intnodes(i),:); %main node for local basis function
onodes=setdiff(nodes,intnodes(i));
%two other nodes (w/ zero values) for local basis function
xyr=N(onodes(1),:);
xys=N(onodes(2),:);
M=[xyr 1;xys 1;xyt 1]; %matrix M of (4)
abccoeff=[xyr(2)-xys(2); xys(1)-xyr(1); xyr(1)*xys(2)-...
xys(1)*xyr(2)]/det(M); %coefficients of basis function on triangle#L,
see formula (6a)
intgrad(i,:)=abccoeff(1:2)';
end

for j=1:length(bdynodes)
xyt=N(bdynodes(j),:); %main node for local basis function
onodes=setdiff(nodes,bdynodes(j));%two other nodes (w/ zero values)
for local basis function
xyr=N(onodes(1),:);
xys=N(onodes(2),:);
M=[xyr 1;xys 1;xyt 1]; %matrix M of (4)
abccoeff=[xyr(2)-xys(2); xys(1)-xyr(1); xyr(1)*xys(2)-...
xys(1)*xyr(2)]/det(M); %coefficents of basis function on triangle#L,
see formula (6a)
bdygrad(j,:)=abccoeff(1:2)';
end
%update stiffness matrix
for i1=1:length(intnodes)
for i2=1:length(intnodes)
fun1 = num2str(intgrad(i1,:)*intgrad(i2,:)',10); %integrand for
(15ell)
fun=inline(fun1,'x', 'y');
integ=gaussianintapprox(fun,xyt,xyr,xys);
A(intnodes(i1),intnodes(i2))=A(intnodes(i1),intnodes(i2))+integ;
end
end

%update load vector
for i=1:length(intnodes)
for j=1:length(bdynodes)
fun1 = num2str(intgrad(i,:)*bdygrad(j,:)',10); %integrand for (16ell)
```

```
fun=inline(fun1,'x', 'y');
integ=gaussianintapprox(fun,xyt,xyr,xys);
b(intnodes(i))=b(intnodes(i))-c(bdynodes(j))*integ;
end
end

end
sol=A\b;
c(1:n)=sol';
>> trimesh(tri,x,y,c)
>> xlabel('x-axis'), ylabel('y-axis')
```

EFR 13.12: (a) Rerun the triangulation code of the solution of Example 13.2(a). The construction was done in a way that the boundary nodes came last. So we will be able to adapt the assembly code of EFR 13.11 quite simply. (The only change will be in dealing with the load vector, because of the presence of the inhomogeneity function.)

```
>> m=length(x); %number of nodes
>> n = min(find(x.^2+y.^2>1-10*eps))-1;  %number of interior nodes
```

Now we easily modify the (boxed) code of the preceding EFR to work for the present situation. There are some changes here due to the fact that the boundary data is now all zero, but we do have a nonzero inhomogeneity fucntion $f(x,y)$, and thus (16^{ℓ}) takes on the following more simple form: $b_{\alpha}^{\ell} = \iint_{T_{\ell}} f\Phi_{i_{\alpha}}\, dxdy \quad (1 \le \alpha \le 3)$. Thus, the "updating the load vector" portion should be replaced by:

```
%update load vector
for i1=1:length(intnodes)
xyt=N(intnodes(i),:); %main node for local basis function
onodes=setdiff(nodes,intnodes(i));
%two other nodes (w/ zero values) for local basis function
xyr=N(onodes(1),:);
xys=N(onodes(2),:);
M=[xyr 1;xys 1;xyt 1]; %matrix M of (4)
abccoeff=[xyr(2)-xys(2); xys(1)-xyr(1); xyr(1)*xys(2)-...
xys(1)*xyr(2)]/det(M); %coefficients of basis function on triangle#L,
                       %see formula (6a)
%since we cannot mix M-file and inline functions to input into
%another M-file, we recode the gaussianintapprox M-file
atemp=num2str(abccoeff(1),10); btemp=num2str(abccoeff(2),10);
ctemp=num2str(abccoeff(3),10);
phixy=inline([atemp, '*x+', btemp, '*y+',ctemp],'x','y');
Atemp=feval(@EFR13_12f,(xyt(1)+xyr(1))/2,(xyt(2)+xyr(2))/2)*...
feval(phixy,(xyt(1)+xyr(1))/2,(xyt(2)+xyr(2))/2);
Btemp=feval(@EFR13_12f,(xyt(1)+xys(1))/2,(xyt(2)+xys(2))/2)*...
feval(phixy,(xyt(1)+xys(1))/2,(xyt(2)+xys(2))/2);
Ctemp=feval(@EFR13_12f,(xyr(1)+xys(1))/2,(xyr(2)+xys(2))/2)*...
feval(phixy,(xyr(1)+xys(1))/2,(xyr(2)+xys(2))/2);
M=[xyr(1) xyr(2) 1;xys(1) xys(2) 1; xyt(1) xyt(2) 1];
area=abs(det(M))/2;
integ=area*(Atemp+Btemp+Ctemp)/3;
b(intnodes(i1))=b(intnodes(i1))+integ;
end
```

Also, the loop portion of the assembly code commencing with "`for j=1:length(bdynodes)`" can be deleted since the boundary node gradients that it creates will not be needed in (16^{ℓ}). With these modifications, the code will produce the numerical solution of Figure 13.39(b), once an M-file for the inhomogeneity function is created (due to the cases in its definition, an inline construction is not feasible):

```
function z = EFR13_12f(x,y)
```

```
if norm([x y]-[0 .5],2)<.25
z=20;
else
z=0;
end
```

(b) The assembly instructions are exactly as in part (a), after we have created the node set and triangulation according to the specifications. The following code will create such a triangulation:

```
% node deployment, use concentric circles centered at (0, 1/2)
% except for on the boundary
% Step 1 inside Omega1 (small circle) has 50% of nodes
% d1=common gap size
% avg radius = 1/8, avg. circumf= pi/4,
% avg no. of nodes on circ = pi/4/d1
% number of circles = 1/4/d1
% setting 50% of 800 = [pi/4/d1][1/4/d1] gives
d1=sqrt(pi/16/400);
x(1)=0; y(1)=.5;
nodecount=1; ncirc=floor(1/4/d1); minrad=1/4/ncirc;
for i=1:ncirc, rad=i*minrad; nnodes=floor(2*pi*rad/d1);
anglegap=2*pi/nnodes;
for k=1:nnodes
  x(nodecount+1)=rad*cos(k*anglegap);
  y(nodecount+1)=rad*sin(k*anglegap)+.5;
  nodecount = nodecount+1;
end
end

% step 2:  inside annulus Omega2 has 25% of nodes
% d2=common gap size
% avg radius = 3/8, avg circumf = 3pi/4,
% avg no of nodes on circ = 3pi/4/d2
% number of circles = 1/4/d2
d2=sqrt(3*pi/16/200); ncirc=floor(1/4/d2);minrad=1/4+(d1+d2)/2;
%blend interface
for i=1:ncirc
  rad=minrad + (i-1)*d2; nnodes=floor(2*pi*rad/d2);
  anglegap=2*pi/nnodes;
  for k=1:nnodes
    x(nodecount+1)=rad*cos(k*anglegap);
    y(nodecount+1)=rad*sin(k*anglegap)+.5;
    nodecount = nodecount+1;
end
end

% step 3:  inside region Omega3 has 15% of nodes
% d3 = common gap size
% avg radius = 3/4, avg arclength (approx)= (2pi +pi)/2*3/4=9pi/8
% number of circles = 1/2/d3
d3=sqrt(9*pi/16/120); ncirc=floor(1/2/d3); minrad=1/2+(d2+d3)/2;
%blend interface
for i=1:ncirc
  rad=minrad + (i-1)*d3; nnodes=floor(2*pi*rad/d3);
  anglegap=2*pi/nnodes;
   for k=1:nnodes
    xtest=rad*cos(k*anglegap); ytest=rad*sin(k*anglegap)+.5;
    if norm([xtest ytest],2)<1-d3/2 %don't put nodes too close to bdy
 x(nodecount+1)=xtest; y(nodecount+1)=ytest; nodecount = nodecount+1;
    end
   end
```

```
end

% step 4: inside region Omega4 has 10% of nodes
% d4 = common gap size
% avg radius = 5/4, avg (approx) arclength =  5pi/4
% number of circles = 1/2/d4
d4=sqrt(5*pi/8/80); ncirc=floor(1/2/d4); minrad=1+(d3+d4)/2; %blend
interface
for i=1:ncirc
  rad=minrad + (i-1)*d4; nnodes=floor(2*pi*rad/d4);
  anglegap=2*pi/nnodes;
  for k=1:nnodes
    xtest=rad*cos(k*anglegap); ytest=rad*sin(k*anglegap)+.5;
    if norm([xtest ytest],2)<1-d4/2 %don't put nodes too close to bdy
 x(nodecount+1)=xtest; y(nodecount+1)=ytest; nodecount = nodecount+1;
    end
  end
end

% step 5: put nodes on boundary
% if bdy point is touches Omega3 use d3 spacing
% otherwise use d4 spacing
theta=0;
while theta<2*pi-d4
  x(nodecount+1)=cos(theta); y(nodecount+1)=sin(theta);
  nodecount = nodecount+1;
  if norm([cos(theta) sin(theta)]-[0 .5], 2)<.5
    theta=theta+d3;
  else
    theta=theta+d4;
  end
end
```

EFR 13.13: (a) The M-file is boxed below:

```
function lineint = bdyintapprox(fun, tri, redges)
% function M-file for EFR 13.13
% inputs will be 'fun', an inline function (or M-file) of vars x, y; % a matrix 'tri' of nodes of
a triangle in the plane, and a 2-column % matrix 'redges', possibly empty ([ ]), containing, as
rows,  the
% corresponding node indices (from 1 to 3 indicating nodes
% by their row in 'tri') of nodes which are endpoints of segments of
% the triangle which are part of the 'Robin' boundary (for an
% underlying FEM problem).  Thus the rows of 'redges' can include
% only the following three vectors:  [1 2], [1 3], and [2 3].  (Or
% permutations of these.) The output, 'lineint' will be the the
% Newton-Coates approx. ((31) of Chapter 13) line integral of 'fun'
% over the Robin segments of the triangle.
lineint=0;
[rn cn] = size(redges); %rn = number of Robin edges
if rn == 0
    return
end
for i=1:rn
    nodes = redges(i,:);
    N1=tri(nodes(1),:); N2=tri(nodes(2),:);
    N1x=N1(1); N1y=N1(2); N2x=N2(1); N2y=N2(2);
    vec = N2-N1;

approx=norm(vec,2)/6*(feval(fun,N1x,N1y)+4*feval(fun,(N1x+N2x)/2,(N1y+N2y)/2)+feval(fun,N2x,N2y));
```

```
    lineint=lineint+approx;
end
```

(b)

```
>> tri1 = [0 0;2 0;0 3]; tri2=tri1/10;
>> f1 = inline('4','x','y'); f2=inline('cos(pi*x/4+pi*y/2)','x','y');
>> redges1 = [1 2;2 3]; redges2 = [1 2; 1 3];
>> Int1=bdyintapprox(f1,tri1,redges1)   →Int1 = 22.4222
>> abs(Int1-8-4*sqrt(13)) →ans = 1.7764e-015  (Error for first approximation)
>> bdyintapprox(f1,tri2,redges1)   →ans =  2.2422
>> Int2=bdyintapprox(f2,tri1,redges2)   →Int2 =  0.3619 (Error for first
approximation)
>> abs(Int2-2/pi)   →ans = 0.4882
```

It is not surprising that the error for the first integration was as small as machine precision, since the method is exact for polynomials of degree up to three and we are integrating a constant function. A similar accuracy would hold for the integral over the smaller triangle. The second integration had a very large error and this was due to the fact that the integrand experiences a lot of variation on the edges. A similarly large error (although a bit smaller relatively) would occur if we looked at the integral of the second function over the smaller triangle (the error would be 0.1263). When we utilize this integrator in our FEM codes, we can use a fine enough partition (in the portions of the boundary where the data has more variation) to prevent such problems.

EFR 13.14: The PDE and the Dirichlet portion of the BCs are plainly satisfied, so we have only to check the Neumann BC on the parabolic portion of the boundary. A tangent vector to a point on the parabola $y = \varphi(x) \equiv x(10 - x)$ is given by $\vec{\tau}(x) = (d/dx)(x, \varphi(x)) = (1,\ 10 - 2x)$. Since this tangent vector has positive x-component, an outward-pointing normal vector can be obtained from it by rotating $\vec{\tau}(x)$ by an angle of $\pi/2$ (see Section 7.2). Dividing this vector by its Euclidean norm (see Section 7.6) gives the outward pointing unit normal vector: $\vec{n} = \vec{n}(x) = (2x - 10,1)/\left\|(2x - 10,1)\right\|_2 = \dfrac{(2x-10,1)}{\sqrt{4x^2 - 40x + 101}}$. Taking the dot product with the gradient of u $\nabla u(x, y) = (0, y/25)$ of the exact solution given produces the stated Neumann BC.

EFR 13.15: The triangulations for this problem have been already done and can simply be imported. The main task is to set up the assembly process. In the notation of (10), we have: $p \equiv 1,\ q \equiv 0,\ g \equiv 0,\ r = 2$ (on Γ_2), $h = 40$ (on Γ_2), $f(x)$ is as specified. Thus, $c_s = 0$ $(s > n)$ and the element matrix analogues of (28) and (29) (cf., (15$^\ell$) and (16$^\ell$) become:

$$a_{\alpha\beta}^{\ell} = \iint_{T_\ell} [\nabla\Phi_{i_\alpha} \bullet \nabla\Phi_{i_\beta}]\, dxdy + 2 \int_{\Gamma_2 \cap T_\ell} \Phi_{i_\alpha}\Phi_{i_\beta}\, ds \quad (1 \le \alpha, \beta \le 3), \text{ and}$$

$$b_{\alpha}^{\ell} = \iint_{T_\ell} f(x, y)\Phi_{i_\alpha}\, dxdy + 40 \int_{\Gamma_2 \cap T_\ell} \Phi_{i_\alpha}\, ds \quad (1 \le \alpha \le 3).$$

This is just a bit more involved than the assembly equations for Example 13.8, since in the former there were no line integrals in the first (stiffness matrix coefficient) equations. Nonetheless, the assembly code of the example can be easily adapted to fill our present needs. We first need to store an M-file for the inhomogeneity function *f*(*x*,*y*):

```
function z = EFR13_15f(x,y)
if x>=4 & x<=6 & y>=10 & y<=15
z=200;
else
z=0;
end
```

Before running the assembly code below, we assume that the triangulation code of Example 13.8(a) has been run. In particular, the following variables have been created: `nint` = the number of interior nodes, `n` = the number of interior/Robin nodes, `m` = the number of nodes, `dir1` = `m` = node index for (0,0), and `dir2` = `nint + 1` = the node index for (10,0). As in the example, in the first part of the

code, we need not compute gradients of basis functions corresponding to Dirichlet nodes, since the Dirichlet boundary values are all zero. The only new technical issue here is that in the computation of the load coefficients (in the first integral), since it is awkward to mix inline functions and M-files into a single function, we choose to simply recode the `gaussianintapprox` M-file (which is a rather short code).

```
N=[x' y']; E=tri; A=zeros(n); b=zeros(n,1); [L cL]=size(E);
for ell=1:L
nodes=E(ell,:); %global node indices of element
percent=100*ell/L %optional percent meter will show progression.
intnodes=nodes(find(nodes<=n)); %global interior/Robin node indices
%find coefficients [a b c] of local basis functions
% ax + by +c; for  int/robin nodes
for i=1:length(intnodes)
xyt=N(intnodes(i),:); %main node for local basis function
onodes=setdiff(nodes,intnodes(i));
%global indices for two other nodes (w/ zero values) for local basis
function
xyr=N(onodes(1),:); xys=N(onodes(2),:);
M=[xyr 1;xys 1;xyt 1]; %matrix M of (4)
%local basis function coefficients using (6B)
abccoeff=[xyr(2)-xys(2); xys(1)-xyr(1); xyr(1)*xys(2)-...
          xys(1)*xyr(2)]/det(M);
intgrad(i,:)=abccoeff(1:2)'; abc(i,:)=abccoeff';
end

% determine if there are any Robin edges
marker=0; %will change to 1 if there are Robin edges.
roblocind=find(nodes==dir1|nodes==dir2|(nodes<=n & ...
                nodes >=(nint+1)));
%local indices of nodes for possible robin edges
if length(roblocind>1
elemnodes = N(nodes,:);

%now find robin edges and make a 2 column matrix out of their local
%indices.
rnodes=nodes(roblocind); %global indices of robin nodes
count=1;
for k=(nint+1):(n-1)
if ismember(k,rnodes) & ismember(k+1,rnodes)
robedges(count,:)=[find(nodes==k)  find(nodes==k+1)]; count=count+1;
marker =1;
end
end
end

%update stiffness matrix
for i1=1:length(intnodes)
for i2=1:length(intnodes)
if intnodes(i1)>=intnodes(i2)
%to save some computation, we use symmetry of the stiffness matrix.
fun1 = num2str(intgrad(i1,:)*intgrad(i2,:)',10);
           %integrand for (15ell)
fun=inline(fun1,'x', 'y'); integ=gaussianintapprox(fun,xyt,xyr,xys);
A(intnodes(i1),intnodes(i2))=A(intnodes(i1),intnodes(i2))+integ;

%now add Robin portion, if applicable
%robin edges were computed above
if marker==1
ai1 = num2str(abc(i1,1),10); ai2 = num2str(abc(i2,1),10);
bi1 = num2str(abc(i1,2),10); bi2 = num2str(abc(i2,2),10);
```

```
ci1 = num2str(abc(i1,3),10); ci2 = num2str(abc(i2,3),10);
prod=inline(['2*(',ai1,'*x+',bi1, '*y+', ci1,')* ...
      (',ai2,'*x+',bi2, '*y+',ci2,')'],'x','y');
A(intnodes(i1),intnodes(i2))=A(intnodes(i1),intnodes(i2)) ...
            +bdyintapprox(prod,elemnodes, robedges);
end
end
end
end

%update load vector
for i1=1:length(intnodes)
ai1 = num2str(abc(i1,1),10); bi1 = num2str(abc(i1,2),10);
ci1 = num2str(abc(i1,3),10);
phi=inline([ai1,'*x+',bi1, '*y+', ci1],'x','y');

%since we cannot mix M-file and inline functions to input into
%another M-file, we basically must recode the gaussianintapprox M-
%file
Atemp=feval(@EFR13_15f,(xyt(1)+xyr(1))/2,(xyt(2)+xyr(2))/2)*...
      feval(phi,(xyt(1)+xyr(1))/2,(xyt(2)+xyr(2))/2);
Btemp=feval(@EFR13_15f,(xyt(1)+xys(1))/2,(xyt(2)+xys(2))/2)*...
      feval(phi,(xyt(1)+xys(1))/2,(xyt(2)+xys(2))/2);
Ctemp=feval(@EFR13_15f,(xyr(1)+xys(1))/2,(xyr(2)+xys(2))/2)*...
      feval(phi,(xyr(1)+xys(1))/2,(xyr(2)+xys(2))/2);
M=[xyr(1) xyr(2) 1;xys(1) xys(2) 1; xyt(1) xyt(2) 1];
area=abs(det(M))/2;
integ=area*(Atemp+Btemp+Ctemp)/3;
b(intnodes(i1))=b(intnodes(i1))+integ;

%now add Robin portion, if applicable
%robin edges were computed above
if marker==1
prod=inline(['40*(',ai1,'*x+',bi1, '*y+', ci1,')'],'x','y');
b(intnodes(i1))=b(intnodes(i1))+ ...
       bdyintapprox(prod, elemnodes, robedges);
end
end
clear roblocind rnodes robedges
end
A=A+A'-A.*eye(n); %Use symmetry to fill in remaining entries of A.

sol=A\b;
c(1:n)=sol';
c(n+1:m)=0;

%The result is now easily plotted using the 'trimesh' function of the
%last section:

x=N(:,1); y=N(:,2);
trimesh(E,x,y,c)
hidden off
xlabel('x-axis'), ylabel('y-axis')
```

The above code will produce a plot of the FEM solution.

EFR 13.16: In the notation of (10), we have: $p \equiv 1$, $q \equiv 0$, $f \equiv 0$, $g \equiv 100$ (on Γ_1), $r = 0, 1,$ or 2 (on Γ_2), and $h = 0, 20,$ or 30 (on Γ_2). The element matrix analogues of (28) and (29) (cf., (15^{ℓ}) and (16^{ℓ})) thus become:

$$a_{\alpha\beta}^{\ell} = \iint_{T_\ell} [\nabla\Phi_{i_\alpha} \bullet \nabla\Phi_{i_\beta}]\, dxdy + r \int_{\Gamma_2 \cap T_\ell} \Phi_{i_\alpha}\Phi_{i_\beta}\, ds \quad (1 \le \alpha, \beta \le 3), \text{ and}$$

$$b_{\alpha}^{\ell} = \iint_{T_\ell} f(x,y)\Phi_{i_\alpha}\, dxdy + 40 \int_{\Gamma_2 \cap T_\ell} \Phi_{i_\alpha}\, ds \quad -$$

$$\sum_{s=n+1}^{m} 100 \left[\iint_{T_\ell} [\nabla\Phi_s \bullet \nabla\Phi_{i_\alpha}]\, dxdy + r \int_{\Gamma_2 \cap T_\ell} \Phi_s\Phi_{i_\alpha}\, ds \right] \quad (1 \le \alpha \le 3).$$

The triangulation is new but can be accomplished with the various techniques that we have developed so far. Here is the complete annotated code for our construction. The code also introduces some special variables used to store important node numbers corresponding to the eight corner nodes on the boundary.

```
%Mesh Generation
A =36-pi*(4+1); %area of region
delta = sqrt(A/2500); count = 1;


%place interior nodes first
for i=1:ceil(6/delta), for j=1:ceil(6/delta)
xt=i*delta; yt=j*delta; xy=[xt yt];
if norm(xy,2)>2+delta/2 & norm(xy-[6 0],2)>2+delta/2 & ...
   norm(xy-[6 6],2)>2+delta/2 &norm(xy-[0 6],2)>2+delta/2 ...
   & norm(xy-[3 3],2)>1+delta/2 & xt<6-delta/2 & yt<6-delta/2
x(count)=xt; y(count)=yt; count=count+1;
end, end, end
nint=count-1; %number of interior nodes

%now deploy boundary nodes; we will group them according to their
%boundary conditions; as usual, the Robin nodes precede the Dirichlet
%nodes.  At the corners there is some ambiguity since the normal
%vector is undefined.  We make some conventions that Robin
%conditions take precedence over Neumann conditions, and for Neumann
%conditions at an interface, we simply average the values of the
%normal derivative values.

%Helpful Auxilliary Vectors:
v1=linspace(2,4,2/delta); lenv1=length(v1);
thetaout=linspace(0,pi/2,pi/delta); %node angular gaps for big
quarter circles
lenthout=length(thetaout);
thetain=linspace(0,2*pi,2*pi/delta); %node angular gaps for smaller
interior cirlce
lenthin=length(thetain);

%Neumann conditions with zero boundary values:
for i=2:lenv1 %east
x(count)=6; y(count)=v1(i); count=count+1;
end
for i=2:lenthout %northeast
x(count)=6+2*cos(-pi/2-thetaout(i)); y(count)=6+2*sin(-pi/2-...
    thetaout(i)); count=count+1;
end
toprightindex=count-1

for i=2:lenv1 %north
x(count)=6-v1(i); y(count)=6; count=count+1;
end
```

```
topleftindex=count-1

for i=2:lenthout %northwest
x(count)=2*cos(-thetaout(i)); y(count)=6+2*sin(-thetaout(i));
count=count+1;
end

for i=2:lenv1-1 %west
x(count)=0; y(count)=6-v1(i); count=count+1;
end
lastwestind=count-1
firstsouthind=count

for i=2:lenv1-1 %south
x(count)=v1(i); y(count)=0; count=count+1;
end
lastsouthind=count-1

%Now we move on to the two Robin portions
firstswind=count
for i=1:lenthout %southwest
x(count)=2*cos(thetaout(i)); y(count)=2*sin(thetaout(i));
count=count+1;
end
lastswind=count-1
firstseind=count

for i=1:lenthout %southeast
x(count)=6+2*cos(pi/2+thetaout(i)); y(count)=2*sin(pi/2+thetaout(i));
count=count+1;
end
n=count-1 %number of interior and Robin nodes
lastseind=n;

% finally put in the Dirichlet nodes
for i=1:lenthin
x(count)=3+cos(thetain(i)); y(count)=3+sin(thetain(i));
count=count+1;
end
m=count-1   %number of nodes

%ASIDE:  Enter these commands to plot the nodes
%plot(x(1:nint),y(1:nint),'b.'), axis('equal')
%hold on
%plot(x(nint:m),y(nint:m),'rp'), axis('equal')

%Since the domain is not convex (in 5 spots) we will use the
%technique of Example 13.3 of introducing 5 ghost nodes that will
%yield a triangulation from which it will be easier to delete the
%unwanted triangles

x(m+1)=3; y(m+1)=3;
x(m+2)=5; y(m+2)=1;
x(m+3)=5; y(m+3)=5;
x(m+4)=1; y(m+4)=5;
x(m+5)=1; y(m+5)=1;
tri=delaunay(x,y);
trimesh(tri,x,y,'LineWidth', 1.2), axis('equal')
%Plots the triangulation
```

```
axis('equal')
%Now we need to delete all elements which have a node with index in
%the range m+1 to m+5.
size(tri)  %ans =5224  3,  so there are 5224 elements
badelcount=1;
for ell=1:5224
if sum(ismember(m+1:m+5, tri(ell,:)))>0
badel(badelcount)=ell;
badelcount=badelcount+1;
end
end

tri=tri(setdiff(1:5224,badel),:);
x=x(1:m); y=y(1:m);
trimesh(tri,x,y), axis('equal')
```

To facilitate writing the assembly code, we store the following M-files for the functions *r* and *h*:

```
function z=h_EFR13_16(x,y)
%Inhomogeneity function for
Neumann/Robin BC of %EFR13.16.
if y>2+eps & y<6-eps
    z=0;
elseif y>=6-eps, z=20;
elseif y<eps, z=-20;
elseif (y>=eps & y<=2+eps)|[x y]==[2
0]|[x y]==[4 0]
    z=30;
end
if y==0 & (x==2|x==4), z=5; end
if y==2, z=15; end
if y==6 % (x==2|x==4), z=5; end
```

```
function r=r_EFR13_16(x,y)
%u-coefficient function
for %Neumann/Robin BC of
%EFR13.16.
if (y>=0&y<=2) & x<=2
    r=1;
elseif (y>=0&y<=2) & x>=4
    r=2;
else
    r=0;
end
```

The assembly code is long, but it can be done by combining elements of the others we have developed so far. For space considerations we will refer the complete assembly code to the FTP site for the book (see beginning of this appendix for the URL.)