

### EXAMPLE 13.3:

Let  $\Omega$  denote the annular domain  $\{p = (x, y) \in \mathbb{R}^2 \leq \|p\|_2 \leq 2\}$ . Use MATLAB to create and plot a triangulation of  $\Omega$  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  $\|p\|_2 = 1$  and ending at  $\|p\|_2 = 2$ . Letting, as before,  $\delta$  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 following approximation for the total number of nodes:  $(3\pi/\delta) \cdot (1/\delta) = 3\pi/\delta$ . Setting this equal to 350 and solving for delta gives us a good value to use:  $\delta = 0.164097\dots$

```
>> 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  $\delta$  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:

$c = \text{setdiff}(a, b)$ $\rightarrow$	<p>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.</p>
--	---

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).

```
1| >> badelcount=1;
2| for ell=1:722
3|     if ismember(400,tri(ell,:))
4|         badel(badelcount)=ell;
5|         badelcount=badelcount+1;
6|     end
7| end
8| >> tri=tri(setdiff(1:722,badel),:);
9| >> x=x(1:399); y=y(1:399);
10| >> trimesh(tri,x,y), axis('equal')
```

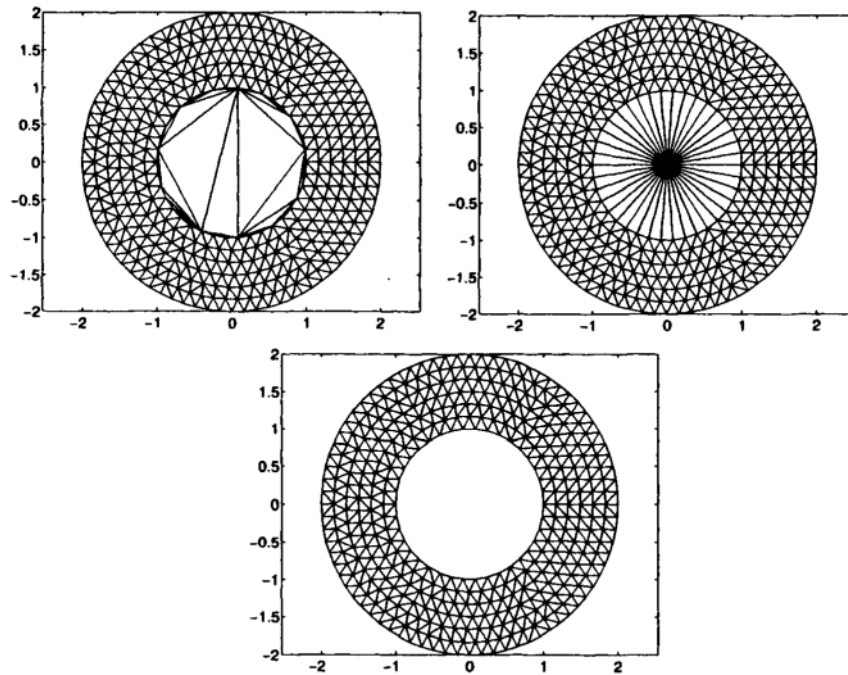


Figure 1: : (a) (upper left) The Delaunay triangulation obtained from a set of nodes in the annular domain  $\{p = (x,y) \in \mathbb{R}^2 \leq \|p\|_2 \leq 2\}$  of Example 13.3. (b) (upper right) The Delaunay triangulation for the same set with one additional "ghost node" added at  $(0,0)$ . (c) Resulting 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  $p$  indicated in the illustration. For the domain of part (a), take the exterior angle to be  $60^\circ$ ; for the one of part (b), make up your own coordinates and dimensions.

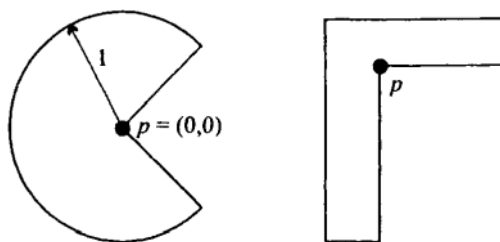


Figure 2: : (a), (b) A pair of domains possessing a boundary point  $p$  with an obtuse interior angle. Such boundary points usually require extra care when boundary value problems on them are solved numerically.

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 give 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)
```

```

      1  0  0  0  0  0
      2  1  0  0  0  0
A=    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)

```

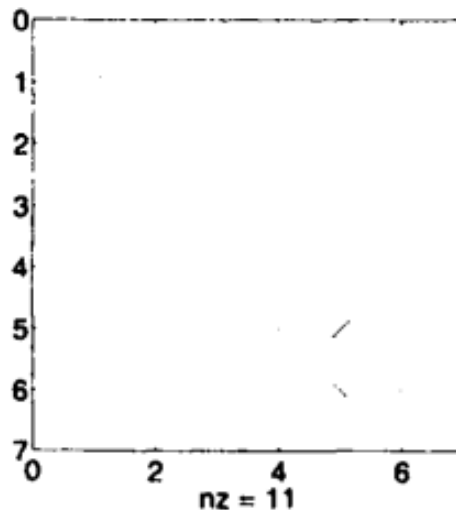


Figure 3: A simple spy plot of a banded 6x6 matrix. The locations of nonzero entries are indicated by x's. The total number of nonzero entries ( $nz = 11$ ) is indicated below the graph. The spy command is a useful tool for obtaining a quick understanding of the structure of a matrix, and, in particular, 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  $1457 \times 1457$ . 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  $\Phi_i$ , and  $\Phi_j$ . Throughout the text of this chapter, we will be restricting our attention to piecewise linear basis functions, and for such a basis function, say  $(\Phi_i)$ , 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  $1457 \times 1457$  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  $1457 \times 1457$  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  $\alpha$  and  $\beta$  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

```

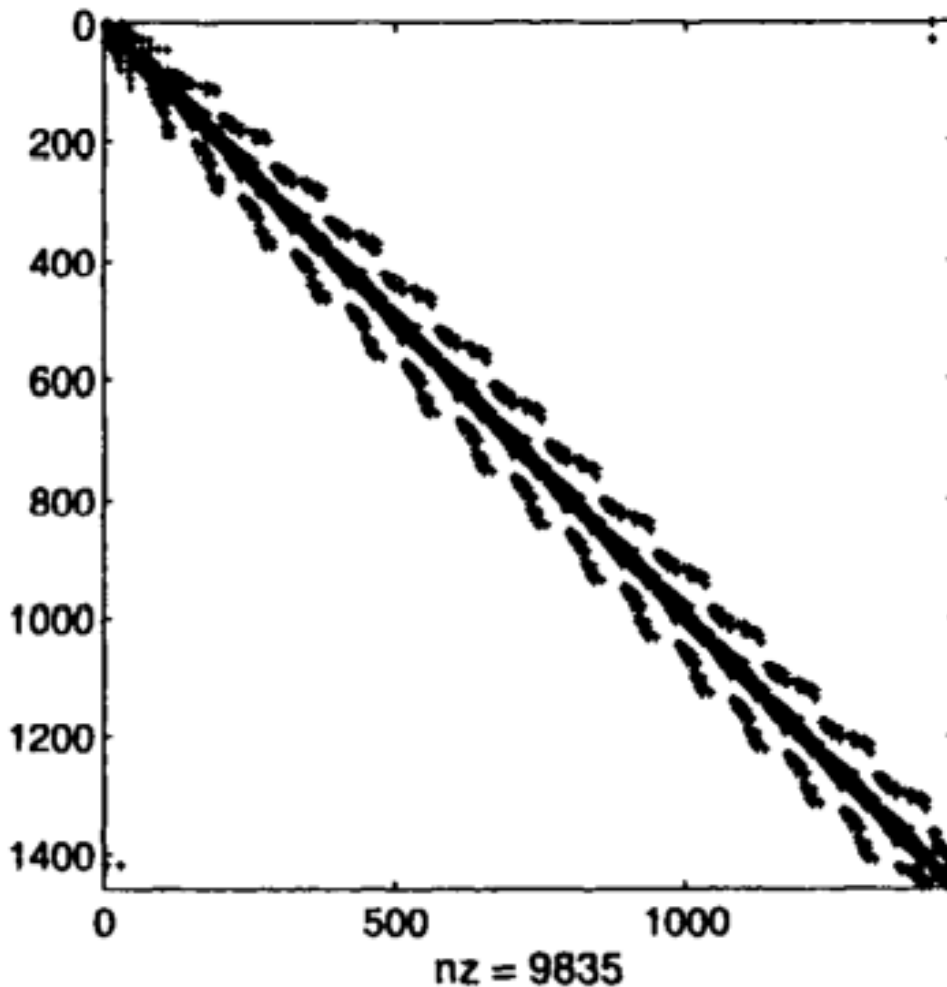


Figure 4: A spy diagram of the stiffness matrix for Example 13.4. The possible nonzero entries account for only 0.46% of all of the entries, so this stiffness matrix is sparse as stiffness matrices usually are. The fuzzy patterns (top and bottom) correspond to the boundary nodes being added after the 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  $\Omega$  denote the unit disk  $\{p = (x, y) \in \mathbb{R}^2 \leq \|p\|_2 \leq 2\}$  of Example 13.2, use MATLAB to create and plot a triangulation of  $\omega$  having between 1000 and 2000 nodes for which the nodes increase in density near the segment  $-\pi/4 \leq \theta \leq \pi/4$  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  $\|p\|_2 \leq 0.5$ , then use a loop to deploy nodes in the annuli  $A_n = \{p : 1 - 2^n \leq \|p\|_2 \leq 1 - 2^{n+1}\}$ ,  $n = 1, 2, \dots$ . As an indicator of closeness to circular arc (for points (xy)) in  $A_n$  use the inequality  $\tan(y/x) \leq 1 + 2 \cdot 2^n$ .

5. Repeat Exercise 4 with the modification that node distribution should increase near the boundary. For the portion of the boundary complementary to  $-\pi/4 \leq \theta \leq \pi/4$ , 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] = circctr i (angle1, angle2, maxnodes)` that will do the following. The input variables `angle1` and `angle2` denote two angles on the unit circle such that  $0 \leq \text{angle2} - \text{angle1} < 2/\pi$ . The M-file will create a set of nodes, stored in the output variables `x` and `y` for the triangularon of the unit disk  $p = \{(x,y) \in \mathbb{R}^2 : \|p\|_2 \leq 2\}$  in a way analogous to the one explained in Exercise 4 (for the special case  $\text{angle} = -\pi/4$  and  $\text{angle2} = \pi/4$  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 triangularon of the node set. Note that the syntax includes the possibility that `angle1 = angle2`, in which case a triangularon 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)  $\text{angle1} = \pi/2$ ,  $\text{angle2} = 5\pi/6$ ,  $\text{maxnodes} = 500$   
 (ii)  $\text{angle1} = -\pi$ ,  $\text{angle2} = \pi/2$ ,  $\text{maxnodes} = 1500$   
 (iii)  $\text{angle1} = 7\pi/6$ ,  $\text{angle2} = 11\pi/6$ ,  $\text{maxnodes} = 1200$
7. (a) Write an M-file, call it `[x y tri] = circctr i (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 triangularon will be performed as explained in Exercise 5 (for the special case  $\text{angle1} = -\pi/4$  and  $\text{angle2} = \pi/4$ , 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  $\text{angle1} < \theta < \text{angle2}$ .  
 (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)  $\text{angle1} = \pi/2$ ,  $\text{angle2} = 5\pi/6$ ,  $\text{maxnodes} = 500$ ,  $r=0.25$   
 (ii)  $\text{angle1} = -\pi$ ,  $\text{angle2} = \pi/2$ ,  $\text{maxnodes} = 1500$ ,  $r=0.25$   
 (iii)  $\text{angle1} = 7\pi/6$ ,  $\text{angle2} = 11\pi/6$ ,  $\text{maxnodes} = 1200$ ,  $r=0.025$
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 triangularon 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  $ax + by \leq c$ . Set up a grid of about `maxnodes` nodes in the rectangle  $R = \{(x,y) : \min(xv) \leq x \leq \max(xv), \min(yv) \leq y \leq \max(yv)\}$ , 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:  $(\pm 2, 0)$ ,  $(\pm 1, 1)$ ,  $(\pm 2, \pm 2)$ . Plot both the node diagram as well as the trianguloron.
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.

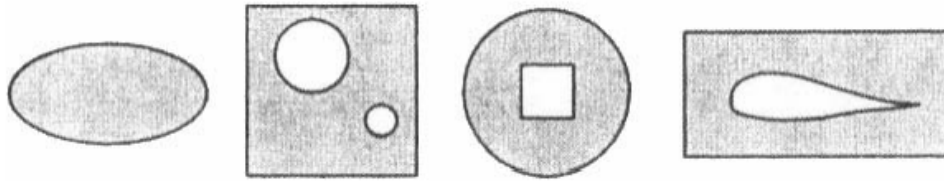


Figure 5: Four domains with curved boundary portions: (from left) (a) An ellipse, (b) a square with two circular holes, (c) a disk with a square hole, (d) an airfoil removed from a rectangle

12. Let the elliptical region  $\omega$  of Figure 13.20(a) have equation (for its boundary):  $x^2 + 4y^2 - 4$ . Use MATLAB to create and plot triangulations of  $\omega$  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  $\omega$  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  $\omega$  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  $\omega$  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  $\omega$  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  $\omega$  denote the region of Figure 13.20(d). (a) Use MATLAB to plot an airfoil (the inside boundary of  $\omega$ ) 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  $\omega$ . 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  $\omega$  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:

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

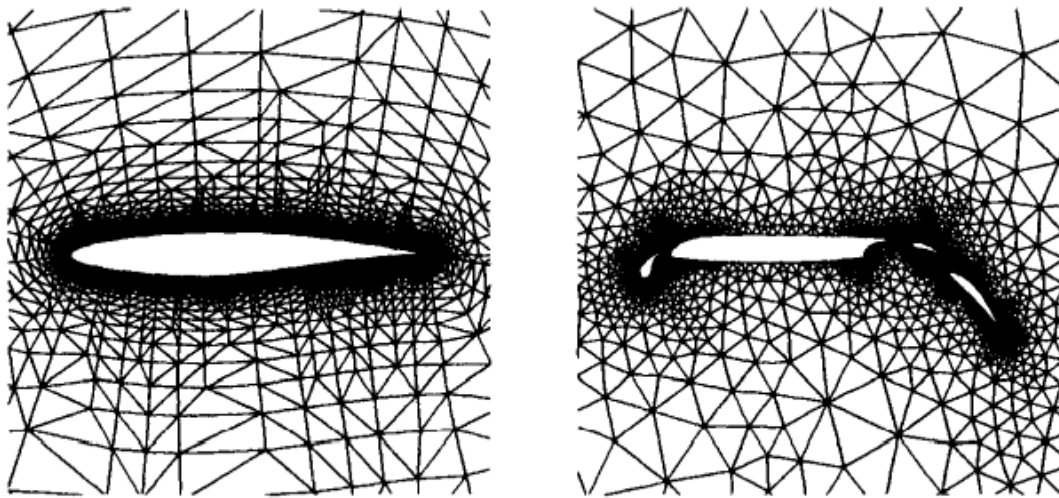


Figure 6: Two triangulations of airfoils, (a) (left) A single component airfoil similar to that in Figure 13.20(d). The triangulation is structured using lines normal to the surface (with increasing density as we near the boundary), (b) A more complex airfoil with flaps. The triangulation is done in a way that the node density increases as we near crucial portions of the configuration<sup>8</sup>

**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.

<sup>8</sup>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.

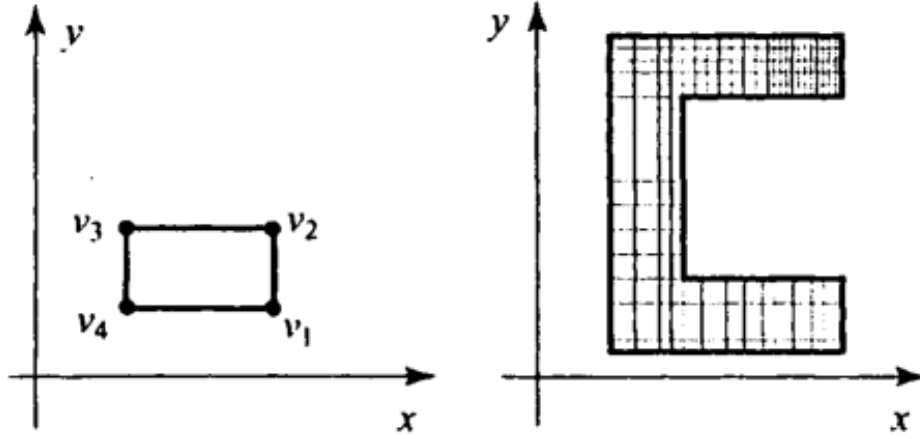


Figure 7: (a) (left) Illustration of a typical rectangular element with its four nodes consisting of its vertices, (b) (right) Tessellation of a domain into rectangular elements.

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).
- Write down a formula for the basis function  $\Theta_{(2,2)}(x,y)$  corresponding to the interior node  $(2,2)$ .
  - Use MATLAB to draw a three-dimensional graph of this basis function.
  - Repeat parts (a) and (b) for the basis function  $\Theta_{(1,1)}(x,y)$  corresponding to the interior node  $(1,1)$ .
  - 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:

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

- Use MATLAB to draw a three-dimensional graph of this function.
- 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:

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

where each term of the sum corresponds to a node of the tessellation.

- 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.
- 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):

$$\begin{aligned} \rho_3(x,y) &= (1/4)(1-x)(l+y), & \rho_2(x,y) &= (1/4)(1+x)(l+y) \\ \rho_4(x,y) &= (1/4)(1-x)(l-y), & \rho_1(x,y) &= (1/4)(1+x)(l-y) \end{aligned}$$

(The local basis function  $\rho_i$  corresponds to the vertex  $v_i$ , and they are written with the same orientation as the vertices appear in the element.)

- 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

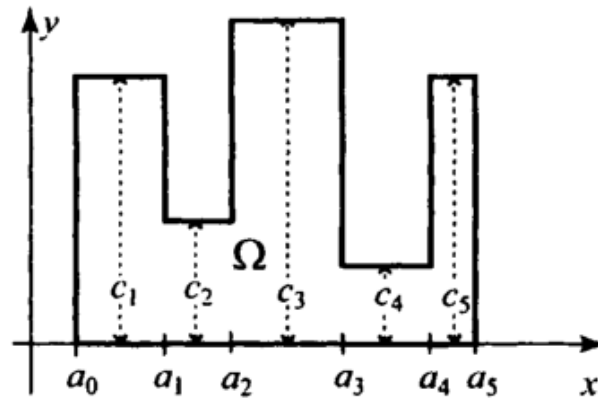


form the mapping can be written as  $\begin{bmatrix} x \\ y \end{bmatrix} = A \begin{bmatrix} \tilde{x} \\ \tilde{y} \end{bmatrix} + v$ , where  $A$  is a  $2 \times 2$  matrix and  $v$  is a  $2 \times 1$  vector. How is the determinant of the matrix  $A$  related to the areas of the two rectangular 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  $\Omega$  to be horizontally blocklike if it has the following form:

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

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

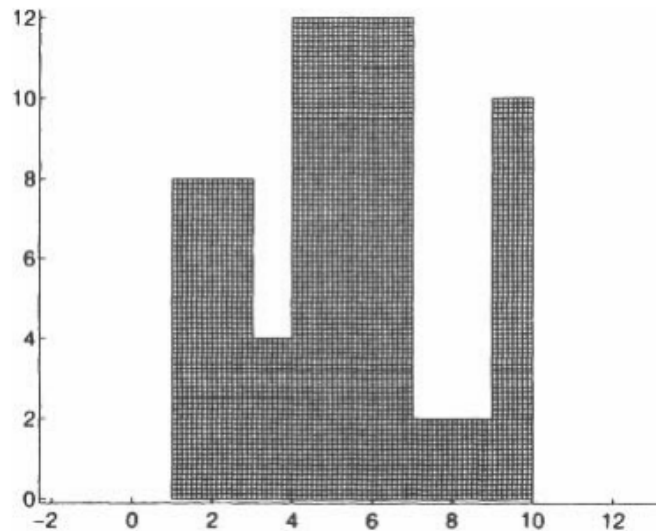


**FIGURE 13.23:** Illustration of a typical horizontally blocklike domain  $\Omega$  for Exercise 20.

(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:  $\frac{1}{2}h \leq w, l \leq 2 \cdot h$ . 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 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  $k = \text{ceil}(\text{gap}/\text{sidelength}) + 1$ . (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, [el e2]=size(elems) ;
for i=1:el
    R=nodes(elems(i, :), :);
    xr=R(:,1); yr(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 node s eleras]=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)  $a = [2 \ 3 \ 5 \ 6 \ 8 \ 10]$ ,  $c = [6 \ 8 \ 7 \ 4 \ 2]$ ,  $h = 0.13$ ,

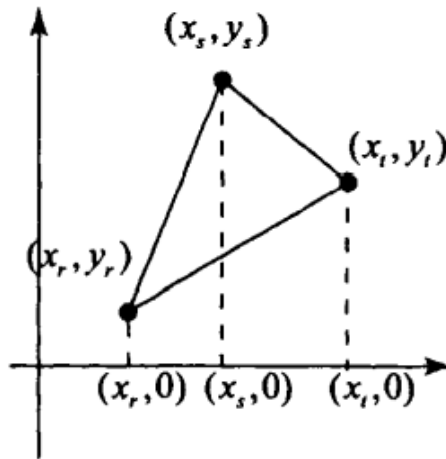
**Suggestions:** Refer to those of Exercise 20 for ideas. If the reader has already completed Exercise 20 (a), the current program could invoke that of Exercise 20 along with a rotation of axes (viz. Section 7.2). After all, a vertically blocklike domain is simply a rotation of a horizontally blocklike 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

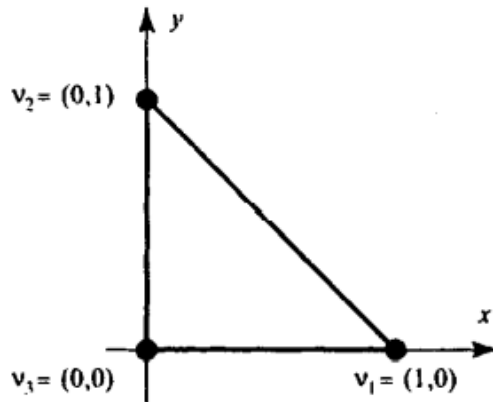
$$M = \begin{bmatrix} x_r & y_r & 1 \\ x_s & y_s & 1 \\ x_t & y_t & 1 \end{bmatrix}.$$

**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

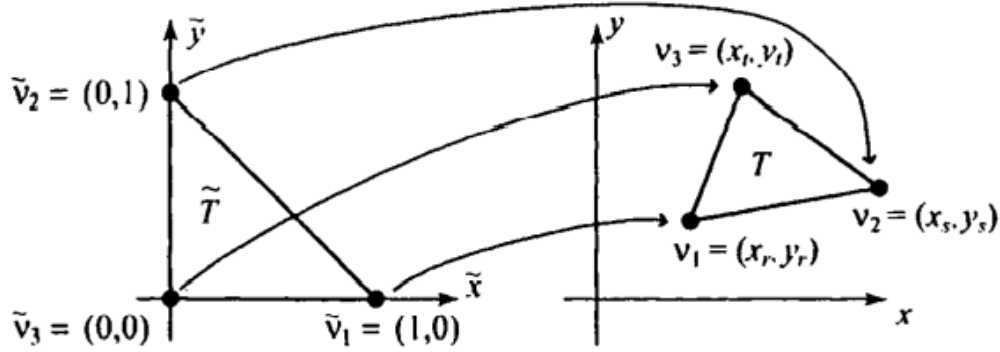
- (a) Show that the standard local basis functions (viz. (7)) for the standard element of Figure 13.26 are given by:

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

- (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  $T$  onto an arbitrary element  $T$ .

(c) Discover and then prove a relationship for the determinant of the  $2 \times 2$  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:

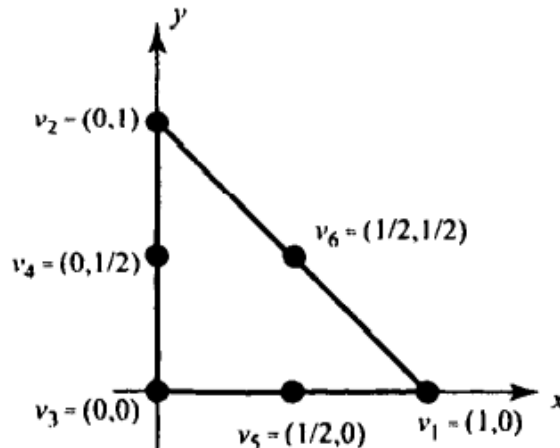
$$\phi_i(x, y) = \tilde{\phi}_i(F^{-1}(x, y)), i = 1, 2, 3.$$

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  $\phi$  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  $\Psi_j$  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  $\phi_j$ '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.

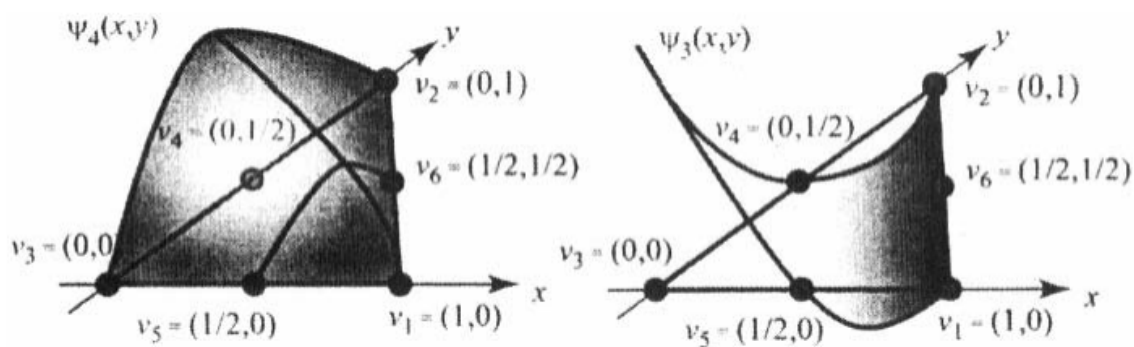
(d) Now letting  $\tilde{\psi}_j(x, y)$ ,  $j = 1, \dots, 6$ , denote the standard local basis functions of part (a) and  $\psi_j(x, y)$  denote the corresponding local basis functions for an arbitrary element, prove that  $\psi_i(x, y) = \tilde{\psi}_i(F^{-1}(x, y))$ ,  $i = 1, \dots, 6$ , where  $F$  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  $\psi_j$  ( $j = 1, \dots, 6$ ) 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  $P = \{p_1, p_2, \dots, p_n\}$  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  $2 \times 1$  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. Additionally, a graphic will be produced that will graph only the elements corresponding to `badelems`. 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.

## 0.1. 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  $\Omega \subset \mathbb{R}^2$

$$\begin{cases} (PDE) & -\nabla \cdot (p \nabla u) + qu = f & \text{on } \Omega \\ (BCs) & u = g & \text{on } \Gamma_1 \\ & \vec{n} \cdot \nabla u + ru = h & \text{on } \Gamma_2 \end{cases} \quad (1)$$

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  $\Gamma_1$  and  $\Gamma_2$ ,  $\partial\Omega = \Gamma_1 \cup \Gamma_2$ . On the first portion  $\Gamma_1$  there are Dirichlet boundary conditions  $u = g$ , and on the complementary portion  $\Gamma_2$  we are assuming generalized Neumann boundary or Robin boundary conditions:  $\vec{n} \cdot (p \nabla u) + ru = h$ . 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  $\vec{n} \cdot \nabla u(x, y)$  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  $\Gamma_2$  generalize the usual Neumann boundary conditions.<sup>9</sup> We allow for the possibility that either  $\Gamma_1 = \partial\Omega$  (so  $\Gamma_2 = \emptyset$ ) 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:

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

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.,  $\Gamma_2 = \emptyset$ ). 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  $\Omega$ , along with the first partial derivatives of  $p$  and  $q$ ,  $g$  is piecewise continuous on  $\partial\Omega$  and  $p(x, y) >$

<sup>9</sup>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  $\vec{n} \cdot \nabla u = 0$  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):  $\vec{n} \cdot \nabla u + ru = h$  when written in the form  $\vec{n} \cdot \nabla u = -r(u - \tilde{h})$  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 ( $\tilde{h}$ ).

$0, q(x, y) \geq 0$ , the BVP can be shown to be equivalent to the minimization problem:

Minimize the functional:

$$F[u] = \iint_{\Omega} \left[ \frac{1}{2} p u_x^2 + \frac{1}{2} p u_y^2 + \frac{1}{2} q u^2 - f u \right] dx dy \quad (2)$$

over the following set of admissible functions:

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

The concept of piecewise continuity on  $\Omega$  (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 follow this outline with some additional details and then give examples.

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

**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:**

$N_1, N_2, \dots, N_n$  (that lie in  $\Omega$ ), and the boundary nodes  $N_{n+1}, N_{n+2}, \dots, N_m$  (that lie on  $\partial\Omega$ ). Denote the basis function  $\Phi_{N_i}$  corresponding to node  $N_i$  simply by  $\Phi_i$ .

**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  $i = n+1, n+2, \dots, m$ .

**Step #3: Assemble the  $n \times n$  stiffness matrix  $A$  and load vector  $b$  needed to determine the remaining coefficients  $c_1, c_2, \dots, c_n$  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):

$$\begin{aligned} \frac{\partial}{\partial c_k} F \left[ \sum_{i=1}^m c_i \Phi_i \right] &= \iint_{\Omega} \left[ p \sum_{i=1}^m c_i \partial_x(\Phi_i) \partial_x(\Phi_k) + p \sum_{i=1}^m c_i \partial_y(\Phi_i) \partial_y(\Phi_k) \right. \\ &\quad \left. + q \sum_{i=1}^m c_i \Phi_i \Phi_k - f \Phi_k \right] dx dy \end{aligned} \quad (3)$$

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  $1 \leq k \leq n$  to obtain the following  $n \times n$  linear system for the unknown coefficients:

$$Ac = b, \quad (4)$$

where the  $c$  represents the (column) vector of the unknown (internal node) coefficients:

$$c = [c_1 \quad c_2 \quad \dots \quad c_n]^T. \quad (5)$$

The entries of the stiffness matrix  $A = [a_{ij}]$  are given by (Exercise 17):

$$a_{ij} = \iint_{\Omega} [p \nabla \Phi_i \cdot \nabla \Phi_j + q \Phi_i \Phi_j] dx dy \quad (1 \leq i, j \leq n) \quad (6)$$

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

$$b_j = \iint_{\Omega} f \Phi_j dx dy - \sum_{s=n+1}^m \iint_{\Omega} [p \nabla \Phi_s \cdot \nabla \Phi_j + q \Phi_s \Phi_j] dx dy \quad (1 \leq j \leq n). \quad (7)$$

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:  $\vec{v} \cdot \vec{w} = \vec{w} \cdot \vec{v}$ ) 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 DIRICHLET BC'S ( $\Gamma_2 = \emptyset$ ) :**

**Step #1: Initialize  $n \times n$  stiffness matrix  $A$  and  $n \times 1$  load vector  $b$  with all zero entries.**

**Step #2: Let  $\ell$  run from 1 to  $L =$  the number of elements (= number of rows of the matrix  $E$  whose  $\ell$  th row gives the node numbers of the  $\ell$  th element  $T_\ell$ ). For each index  $\ell$ , we create the  $3 \times 3$  element stiffness matrix  $A^\ell = [a_{\alpha\beta}^\ell]$  ( $1 \leq \alpha, \beta \leq 3$ ) for the element  $T_\ell$  and the corresponding  $3 \times 1$  element load vector  $b^\ell = \{b_\alpha^\ell\}$  by restricting the integrals in formulas (15) and (16) from  $\Omega$  to  $T_\ell$  :**

$$a_{\alpha\beta}^\ell = \iint_{T_\ell} [p \nabla \Phi_{i_\alpha} \cdot \nabla \Phi_{i_\beta} + q \Phi_{i_\alpha} \Phi_{i_\beta}] dx dy \quad (1 \leq \alpha, \beta \leq 3), \quad (8)$$

and

$$b_\alpha^\ell = \iint_{T_\ell} f \Phi_{i_\alpha} dx dy - \sum_{s=n+1}^m c_s \iint_{T_\ell} [p \nabla \Phi_s \cdot \nabla \Phi_{i_\alpha} + q \Phi_s \Phi_{i_\alpha}] dx dy \quad (1 \leq \alpha \leq 3) \quad (9)$$

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

$$A(E(\ell, \alpha), E(\ell, \beta)) = A(E(\ell, \alpha), E(\ell, \beta)) + a_{\alpha\beta}^\ell \quad (1 \leq \alpha, \beta \leq 3) \quad (10)$$

and

$$b(E(\ell, \alpha)) = b(E(\ell, \alpha)) + b_\alpha^\ell \quad (1 \leq \alpha \leq 3) \quad (11)$$

We point out that formulas (15') and (16') need only be carried out when the indices  $\alpha$  and/or  $\beta$  correspond to interior nodes..<sup>10</sup> Also, the integrands in summation of (16') will vanish on the element  $T$ , unless the corresponding exterior node (number  $s$ ) is a vertex of  $T_\ell$ .

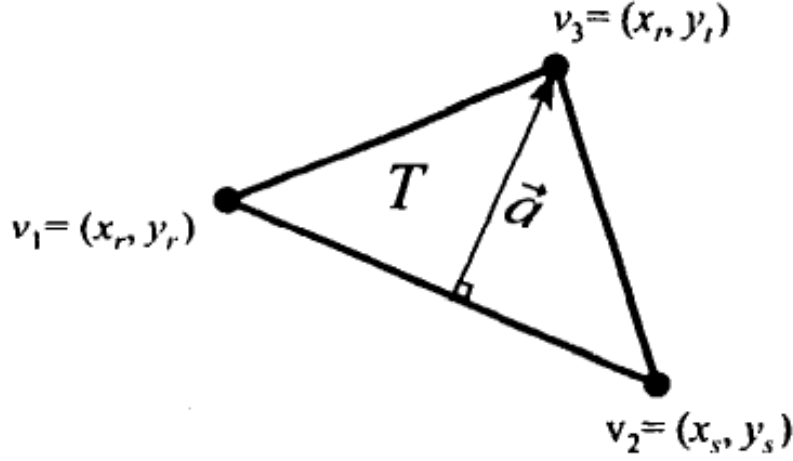
We turn now to a simple example involving the Poisson PDE and constant (Dirichlet) boundary conditions on the hexagonal domain of the last section with only eight triangular elements (Figure 13.5). MATLAB will be able to help us with general multiple integrals, and we will explain how this can be done 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)  $\iint_T \phi(x, y) dx dy = \frac{1}{3} \text{Area}(T)$ .

<sup>10</sup>Otherwise the entries are meaningless. Thus, technically, the element stiffness matrices  $A^\ell$  (load vectors  $b^\ell$ ) will not be complete  $3 \times 3$  ( $3 \times 1$ ) matrices in cases where the element  $T_\ell$  has some of its vertices on the boundary (in Example 13.5, this will be the case for all of the elements).





**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  $\Omega$  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: (2.5, 0), #6: (3.5, 0), #7: (1, -1), and #8: (2.5, -1). Consider the following Poisson BVP for this domain:

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

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

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

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') and (16') simplify to:

$$a_{\alpha\beta}^\ell = \iint_{T_\ell} \nabla \Phi_{i_\alpha} \cdot \nabla \Phi_{i_\beta} dx dy \quad (1 \leq \alpha, \beta \leq 3)$$

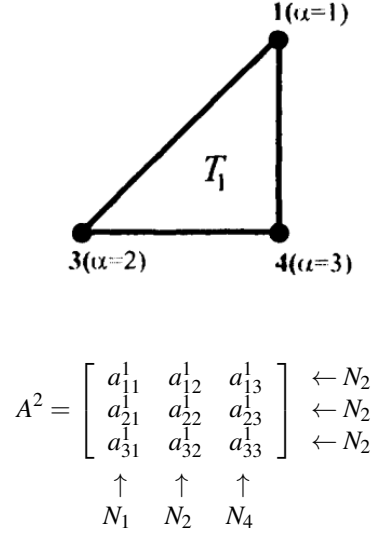
and

$$b_\alpha^\ell = \iint_{T_\ell} f \Phi_{i_\alpha} dx dy - \sum_{s \neq 4, 5} \iint_{T_\ell} \nabla \Phi_s \cdot \nabla \Phi_{i_\alpha} dx dy \quad (1 \leq \alpha \leq 3),$$

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

We initialize a  $2 \times 2$  stiffness matrix  $A$  of zeros and the corresponding  $2 \times 1$  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  $A'$ .



$$A^2 = \begin{bmatrix} a_{11}^1 & a_{12}^1 & a_{13}^1 \\ a_{21}^1 & a_{22}^1 & a_{23}^1 \\ a_{31}^1 & a_{32}^1 & a_{33}^1 \end{bmatrix} \begin{matrix} \leftarrow N_2 \\ \leftarrow N_2 \\ \leftarrow N_2 \end{matrix}$$

$$\begin{matrix} \uparrow & \uparrow & \uparrow \\ N_1 & N_2 & N_4 \end{matrix}$$

**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  $A'$  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:

$$a_{33}^1 = \iint_{T_1} \nabla \Phi_4 \cdot \nabla \Phi_4 dx dy$$

From the formula obtained in Example 13.1 for  $\Phi_4$ , 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  $\nabla \Phi_4 \cdot \nabla \Phi_4 = 2$ . Consequently,

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

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

$$b_3^1 = \iint_{T_1} f \Phi_4 dx dy - \sum_{x=1,3} \iint_{T_1} \nabla \Phi_5 \cdot \nabla \Phi_4 dx dy$$

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  $\Phi_1$  and of  $\Phi_3$  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,

$$b_3^1 = - \iint_{T_1} \nabla \Phi_1 \cdot \nabla \Phi_4 dx dy - \iint_{T_1} \nabla \Phi_3 \cdot \nabla \Phi_4 dx dy = -(-\text{Area}(T_1) - \text{Area}(T_1)) = 1$$

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  $b$ :

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{matrix} \leftarrow N_4 \\ \leftarrow N_5 \end{matrix}$$

$$\begin{matrix} \uparrow & \uparrow \\ N_4 & N_4 \end{matrix}$$

$$b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \begin{matrix} \leftarrow N_4 \\ \leftarrow N_5 \end{matrix}$$

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  $\Phi_4$  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  $\alpha = 1, 2$ , and  $3$ , respectively, so that the node correspondence for the element stiffness matrix  $A^2$  is as follows:

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

$$\begin{matrix} \uparrow & \uparrow & \uparrow \\ N_1 & N_2 & N_4 \end{matrix}$$

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:

$$a_{33}^2 = \iint_{T_2} \nabla\Phi_4 \cdot \nabla\Phi_4 dx dy = \iint_{T_2} 1 dx dy = \text{Area}(T_2) = 3/4,$$

$$b_3^2 = - \iint_{T_2} \nabla\Phi_1 \cdot \nabla\Phi_4 dx dy - \iint_{T_2} \nabla\Phi_2 \cdot \nabla\Phi_4 dx dy = -(-\text{Area}(T_2) + 0) = 3/4.$$

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  $\alpha = 1, 2$ , 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:

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

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

$$\text{and } b_3^3 = - \iint_{T_3} \nabla\Phi_2 \cdot \nabla\Phi_5 dx dy = -(-\text{Area}(T_3)) = 3/4$$

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:

$$\begin{aligned} a_{22}^4 &= \iint_{T_4} \nabla \Phi_5 \cdot \nabla \Phi_5 dx dy = \iint_{T_2} 2 dx dy = 1, \quad \text{and} \\ b_2^4 &= \iint_{T_4} f \Phi_5 dx dy - \iint_{T_4} \nabla \Phi_2 \cdot \nabla \Phi_5 dx dy - \iint_{T_4} \nabla \Phi_6 \cdot \nabla \Phi_5 dx dy \\ &= -\frac{1}{3} \text{Area}(T_4) - (-\text{Area}(T_4) - \text{Area}(T_4)) = 5/6. \end{aligned}$$

(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:

$$\begin{aligned} \ell = 5: \quad A &= \begin{bmatrix} 25/12 + 1 & -1/3 \\ -1/3 & 25/12 \end{bmatrix} = \begin{bmatrix} 37/12 & -1/3 \\ -1/3 & 25/12 \end{bmatrix}, \quad b = \begin{bmatrix} 7/4 + 1 \\ 19/12 \end{bmatrix} = \begin{bmatrix} 11/4 \\ 19/12 \end{bmatrix} \\ \ell = 6: \\ A &= \begin{bmatrix} 37/12 + 13/12 & -1/3 - 1/3 \\ -1/3 - 1/3 & 25/12 + 1/3 \end{bmatrix} = \begin{bmatrix} 25/6 & -2/3 \\ -2/3 & 29/12 \end{bmatrix}, \quad b = \begin{bmatrix} 11/4 + 3/4 \\ 19/12 \end{bmatrix} = \begin{bmatrix} 15/4 \\ 19/12 \end{bmatrix} \\ \ell = 7: A &= \begin{bmatrix} 25/6 & -2/3 \\ -2/3 & 29/12 + 3/4 \end{bmatrix} = \begin{bmatrix} 25/6 & -2/3 \\ -2/3 & 19/6 \end{bmatrix}, \quad b = \begin{bmatrix} 15/4 \\ 19/12 + 3/4 \end{bmatrix} = \begin{bmatrix} 15/4 \\ 7/3 \end{bmatrix} \end{aligned}$$

and finally,

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

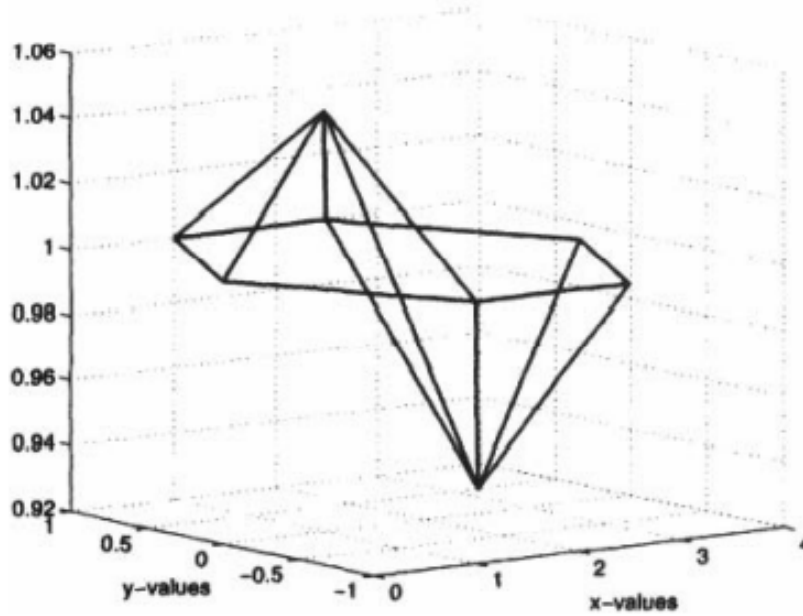
With the stiffness matrix and load vector now "assembled," the remaining coefficients  $c_4, c_5$  are simply the solutions of the linear system:

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

With this small system (solved on MATLAB) exact arithmetic was feasible. The FEM solution  $v = c_4 \Phi_4 + c_5 \Phi_3$  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<sup>11</sup>.



**Figure 13.33:** Simeon-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  $x^2 + y^2 = R^2$  ( $\theta$  is the polar coordinate angle). If  $\Omega$  is the disk inside this circle,  $\Omega = \{p = (x, y) \in \mathbb{R}^2 : \|p\|_2 < R\}$ ,

then the solution of the Dirichlet problem:

$$\begin{cases} \text{(PDE)} & \Delta u = 0 & \text{on } \Omega \\ \text{(BC)} & u(R, \theta) = g(\theta) & \text{on } \partial\Omega \end{cases} \quad (12)$$

is unique and is given by:

$$u(r, \theta) = \frac{R^2 - r^2}{2\pi} \int_0^{2\pi} \frac{g(\phi) d\phi}{R^2 - 2Rr \cos(\theta - \phi) + r^2} \quad (13)$$

Here,  $(r, \theta)$  denotes the polar coordinates of any point inside  $\Omega$ , ( $r < R$ ).

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  $\Omega \cup \partial\Omega = \{p = (x, y) \in \mathbb{R}^2 : \|p\|_2 \leq R\}$  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 ( $f(\theta)$ ). 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<sup>12</sup>

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 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.

<sup>11</sup>After his secondary education, Simeon-Denis Poisson went to work as a surgeon's apprentice with an uncle in Fontainebleau, 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 Ecole 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 Ecole 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 Ecole 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 Ecole Polytechnique. He was able to stop a group of politically active students at the Ecole from publishing a lampooning attack on Napoleon's leadership, fearing that this could do harm to the Ecole. He was elected to the physics section of the prestigious national Institute (a corresponding position in the mathematics section was 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.

<pre>dblquad(fun,xmin,xmax         ymin,ymax) -&gt;</pre>	<p>Assume that fun is an inline function of x and y<sup>13</sup></p> <p>This command will numerically compute the integral</p> $\int_{x_{\min}}^{x_{\max}} \int_{y_{\min}}^{y_{\max}} fun(x,y) dy dx$ <p>using a double iteration with the single variable function integrator quad and with a default tolerance for error being le-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.</p>
<pre>dblquad(fun,xmin,xmax         ymin,ymax,         tol,@quadl,pi,p2,...)         -&gt;</pre>	<p>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 pi, p2, ... represent numerical values to assign in case fun depends on additional parameters: "fun = fun(x,y,pl,p2,...) .</p>

The following simple example will illustrate the syntax requirement on fun

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

$$\int_R x^2 y^2 dx dy = \int_0^2 \int_1^2 x^2 y^2 dy dx,$$

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)
```

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
```

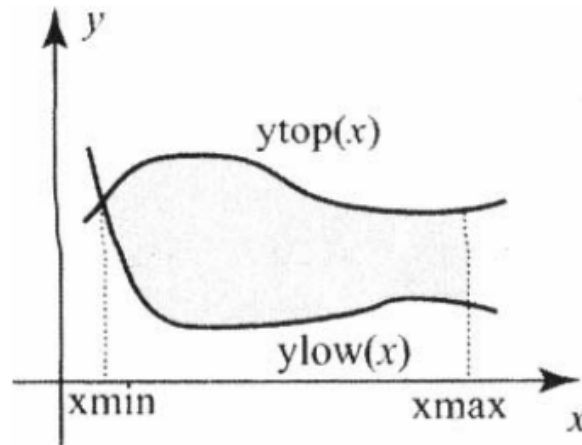
which (as we know) gives the correct answer. A similar syntax note was pointed out in Chapter 3 for quad.

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

$$\begin{aligned} \int_T fun(x,y) dx dy &\equiv \int_{x_{\min}}^{x_{\max}} \int_{y_{\text{low}}(x)}^{y_{\text{top}}(x)} fun(x,y) dy dx \\ &= \int_{x_{\min}}^{x_{\max}} \int_0^1 fun(x, y_{\text{low}}(x) + u(y_{\text{top}}(x) - y_{\text{low}}(x))) [y_{\text{top}}(x) - y_{\text{low}}(x)] du dx, \end{aligned} \quad (14)$$

<sup>12</sup>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  $D \subset \mathbf{R}^2$  can be mapped conformally onto the unit disk  $U = \{p = (x,y) \in \mathbf{R}^2 : \|p\|_2 < 1\}$ . Simply connected means roughly that the domain has no holes inside, i.e., if  $\gamma$  is any closed path in the domain, then the interior of  $\gamma$  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  $F : D \rightarrow U$  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 explicitly 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.

<sup>13</sup>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, ...).



**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  $y_{\text{low}}(x)$  and above by the curve  $y_{\text{top}}(x)$  and over the range  $[x_{\text{min}}, x_{\text{max}}]$ ; 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)  $\int_T 2xy^2 dx dy$

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

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  $1 \leq x \leq 5$ . Over this range, the lower function  $y_{\text{low}}$  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:  $y_{\text{low}}(x) = -\frac{1}{2}x + \frac{7}{2}$ . The corresponding upper function  $y_{\text{top}}$  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  $y$  top:  $y_{\text{top}}(x) = \begin{cases} x+2, & \text{if } x \leq 4 \\ -5x+26, & \text{if } x > 4 \end{cases}$

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

$$\int_T 2xy^2 dx dy = \int_1^5 \int_{y_{\text{low}}(x)}^{y_{\text{top}}(x)} 2xy^2 dy dx = \int_1^4 \int_{y_{\text{low}}(x)}^{x+2} 2xy^2 dy dx + \int_4^5 \int_{y_{\text{low}}(x)}^{-5x+26} 2xy^2 dy dx$$

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 xyu
>> ylow = -.5*x+3.5; ytop 1=x+2;   ytop 2=-5*x+26;
>> f un = 2*x*y^2;
>> ynew 1=y low +u*(y topl -y low) ;
>> funprepl=subs (fun, Y, ynew 1)*(y top 1-ylow);
>> ynew 2=ylow+u*(y top2 -y low) ;
>> funprep2=subs (fun, y, ynew 2)*(y top 2-y low );
>> funnew 1= vectorize (inline ([char( funprepl ), ones '(size(u))'],...
                                'u',x')) ;

>> %we needed to convert the symbolic expression back into a
>> %character string for construction of an inline function.
>> funnew Z= vectorize (inline ( [char (funprep 2),'* ones ( size (u) )'],... 'u',(x+)) ;
```



```
>>dblquad (funnew 1,0,1,1,4)+ dblquad (funnew 2,0,1,4,5 )
→ ans = 724.8000
```

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

```
>> dblquad ( funnew 1,0,1,1,4,1e-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);
>> funprepl = subs (fun, y,ynew1) * (ytop1-ylow) ;
>> funprep2=subs (fun,y, ynew 2)*(ytop2-ylow);
>>
funnew1 = vectorize( inline ( [char (funprepl), ' ' ones (size(u))'],u',x'))
>>
funnew2 =vectorize (inline ( [char( funprep 2),'* ones (size(u))'],u',x'))
>> dblquad (funnew 1,0,1,1,4)+ dblquad (funnew 2,0,1,4,5) →
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 userfriendly 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  $\min x < x < \max x$  and between the functions of
% $x$ :  $y_{low} < y < y_{top}$ .
% 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 variables for lower boundary of region
% ytop = function of symbolic variables for lower 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 uxy ynew=ylow+u* (ytop-ylow);
funprep=subs(fun, y, ynew) (ytop-ylow);
funnew=vectorize (inline ([char (funprep),*];
```

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

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

(b)  $\int_U \exp(1 - x^2 - 2y^2) dx dy$ , 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  $\mathbf{x}, \mathbf{y}$  (written as a symbolic expression), and three  $2 \times 1$  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  $\int_T \text{fun}(x,y) dx dy$ , 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)  $\int_{T_1} 1 dx dy = 6$ ,

(ii)  $\int_{T_2} 1 dx dy = 5/2$ ,

(iii)  $\int_{T_1} 2x^2 dx dy = 504$ , and

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

**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.

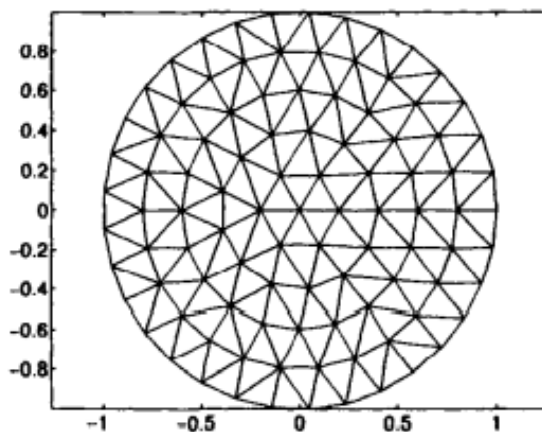
**EXAMPLE 13.7:** Consider the Dirichlet problem (19) on the unit disk  $\Omega = \{(x,y) : x^2 + y^2 < 1\}$

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

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

(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 the FEM solution  $v = \sum^m c_i \Phi_i$ . 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  $\theta$ , 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:

<code>[th,r]=cart2pol (x,y) -&gt;</code>	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.
<code>[x,y]=pol2cart (r,th) -&gt;</code>	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') and (16') simplify to:

$$a_{\alpha\beta}^l = \int_{T_l} \int \nabla \Phi_{i_\alpha} \cdot \nabla \Phi_{i_\beta} dx dy \quad (1 \leq \alpha, \beta \leq 3)$$

and

$$b_\alpha^l = - \sum_{s=n+1}^m c_s \int_{T_l} \int \nabla \Phi_s \cdot \nabla \Phi_{i_\alpha} dx dy \quad (1 \leq \alpha \leq 3)$$

respectively. Also, of the 33 possible indices  $s$  in the  $b_\alpha^l$  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.

<code>[d,ind] = setdiff(a,b) -&gt;</code>	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
» 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));
    bdyndnodes=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(bdyndnodes)
        xyt=N(bdyndnodes (j) , :); % main node for local basis function
        onodes=setdiff (nodes,bdyndnodes (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 il=1:length(intnodes)
        for i2=1:length(intnodes)
            fun = sym(intgrad(il,:)*intgrad(i2, :))' ; % integrand for (15ell)
            integ=triangquad2d(fun,xyt,xyr,xys);
            A(intnodes(il),intnodes(i2))=A(intnodes(il),intnodes(i2))+integ; end
        end

        % update load vector
        for i=1:length(intnodes)
            for j=1:length(bdyndnodes)
                fun = sym(intgrad (i, :) *bdygrad (j,:))' ; % integrand for part of (16ell)
                integ=triangquad2d(fun,xyt,xyr,xys);
```

```

        (intnodes(i))=b(intnodes(i))-c(bdynodes(j))*integ;
    end
end
end
sol=A/b
c(1:n)=sol';

```

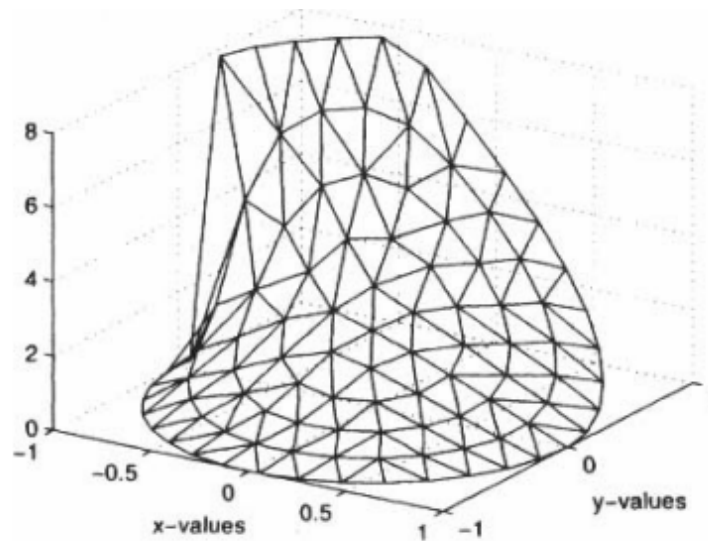
The result is now easily plotted using the `trimesh` function of the last section:

```

» 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  $c_i (i \leq n)$ . 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)=2*phi(i) 2*(1-r^2)/2/pi/(1-2*r*cos(th-phi(i))+r^2);
else
    y(i)=0;
end
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

```

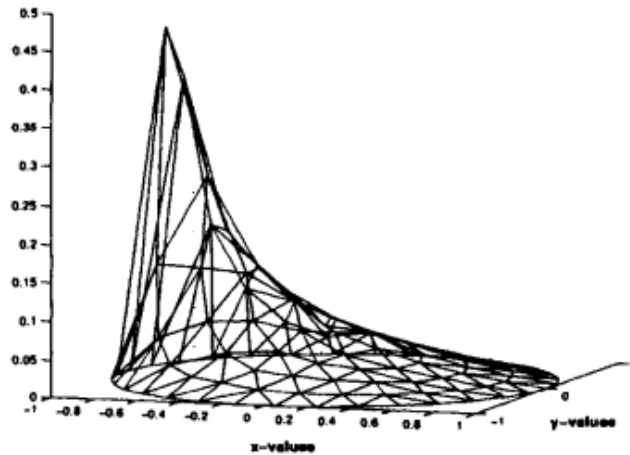
The plot of the exact solution just obtained<sup>14</sup> will be quite similar to that of our FEM approximation in Figure 13.36. The

<sup>14</sup>Of course, the Poisson integral formula, as mentioned, is exact. The only errors will be the errors that arise from the numerical integration. By

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 parameter  $\delta$  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.



**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<sup>15</sup>

(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  $z$ -stretches 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:

$$\int_T f(x,y) dx dy \approx w_1 f(\xi_1) + w_2 f(\xi_2) + \cdots + w_n f(\xi_n) \quad (15)$$

where the **weights**  $w_1, w_2, \dots, w_n$  are specified real numbers and the **sampling points**  $\xi_1 = (x_1, y_1)$ ,  $\xi_2 = (x_2, y_2)$ ,  $\xi_1 = (x_1, y_1)$ ,  $\xi_2 = (x_2, y_2), \dots, \xi_n = (x_n, y_n)$  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

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.

<sup>15</sup>More precisely, this plot is the difference between the FEM solution and the piecewise linear interpolant of the exact solution.

exact for any polynomial of degree at most one:

$$\int_T f(x,y) dx dy \approx \frac{\text{Area}(T)}{3} \{f(V_1) + f(V_2) + f(V_3)\} \quad (16)$$

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:

$$\int_T f(x,y) dx dy \approx \frac{\text{Area}(T)}{3} \{f([V_1 + V_2]/2) + f([V_1 + V_3]/2) + f([V_2 + V_3]/2)\} \quad (17)$$

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  $\Omega$ , the first partial derivatives of  $p, q$ , and  $g$  are piecewise continuous on  $\Gamma_1$ ,  $r$  and  $h$  are piecewise continuous on  $\Gamma_2$  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  $\delta$ , where  $\delta$  is the maximum diameter of any of the (triangular) elements. This result can be roughly expressed by the following inequality:

$$\|u - \hat{u}\| \leq C\delta \quad (18)$$

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  $\delta$ . The norm on the left can be any of several norms to measure the errors. The order of the errors can be upgraded from  $\delta$  to higher powers of  $\delta$  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  $\delta^k$ , i.e.,  $\|u - \hat{u}\| \leq C\delta^k$  (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 improve the accuracy of our method. In the following two exercises for the reader, this quadrature formula is to be invoked in the FEM.

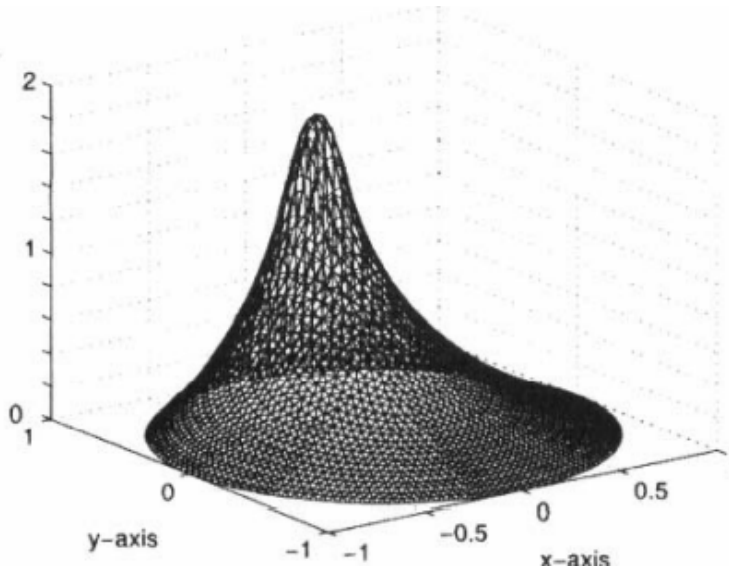
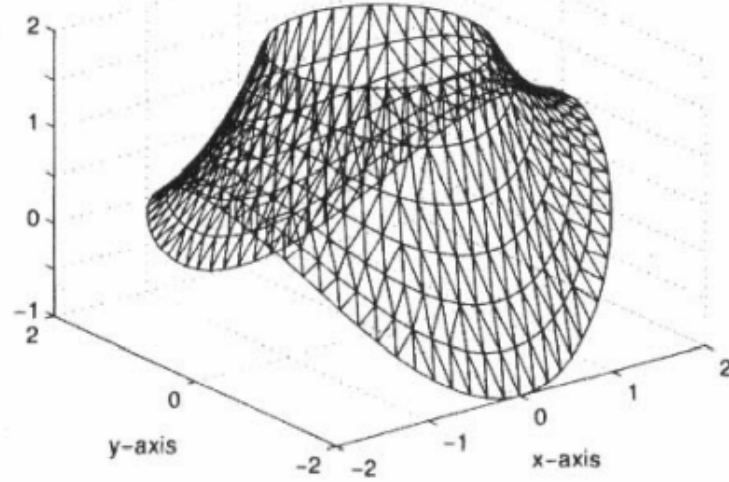
**EXERCISE FOR THE READER 13.11:** Using the grid of Example 13.3, apply the FEM to solve the following Dirichlet problem on the annulus  $\Omega = \{(x,y) : 1 \leq x^2 + y^2 \leq 4\}$ :

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

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

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

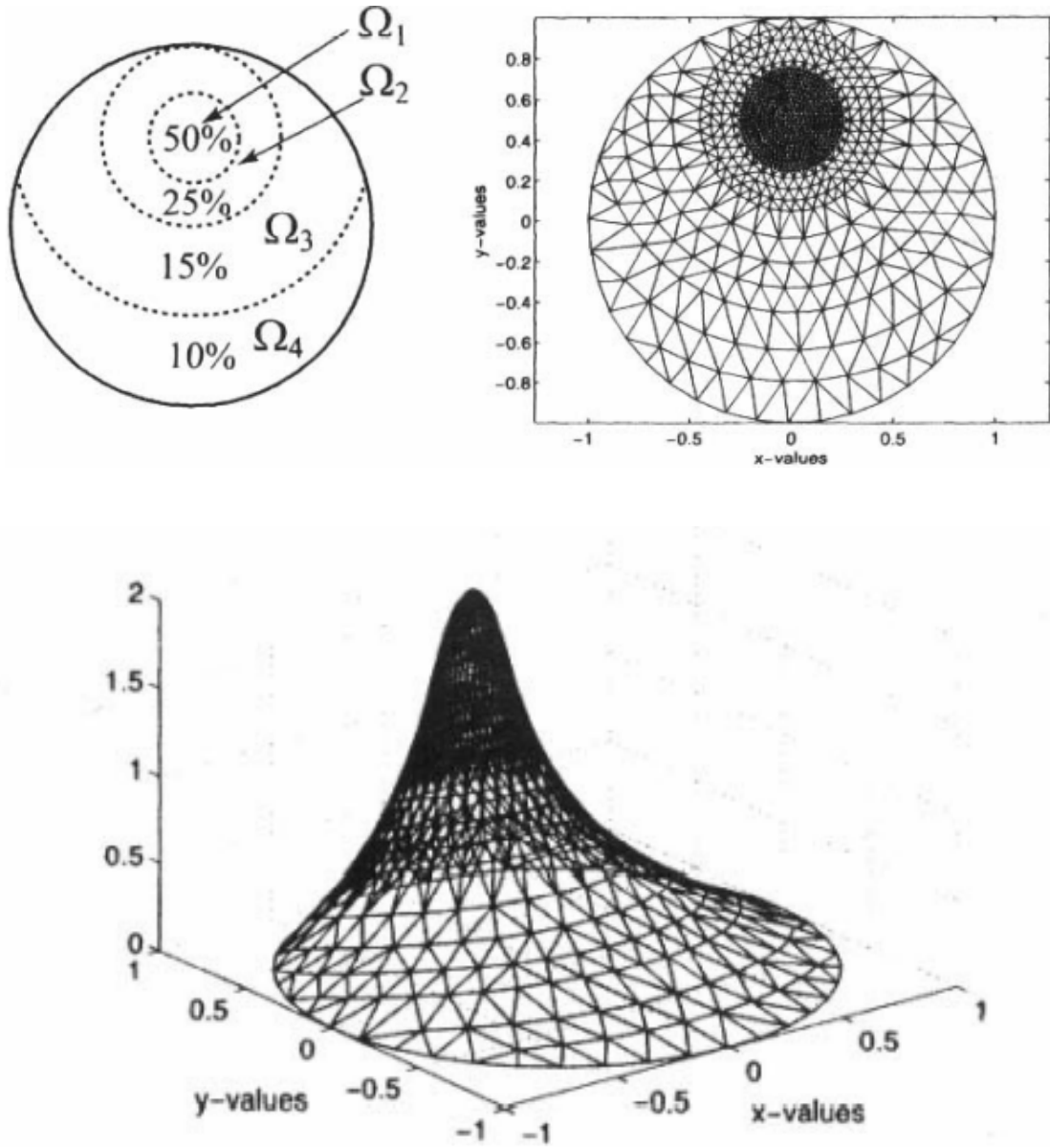
where the load  $f(x,y)$  equals 100 on the (small) disk of radius  $r = 0.125$  and center  $(x,y) = (0,0.5)$ , and zero elsewhere. Plot your FEM solution and indicate the number of nodes, internal nodes, and elements. Your solution plot should look like that in Figure 13.39(b).



**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  $\Omega_1$ , 25% in  $\Omega_2$ , 15% in  $\Omega_3$ , and only 10% in  $\Omega_4$ . Each of these regions is simply the intersection of the whole domain with the insides of the circles with center  $(0, 1/2)$  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  $\Omega$  is split up into four subregions:  $\Omega_1, \Omega_2, \Omega_3$ , and  $\Omega_4$ . (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):

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

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:

$$F[u] = \iint_{\Omega} \left[ \frac{1}{2} p u_x^2 + \frac{1}{2} p u_y^2 + \frac{1}{2} q u^2 - f u \right] dx dy + \int_{\Gamma_2} \left[ \frac{1}{2} r u^2 - h u \right] ds \quad (19)$$

over the following set of admissible functions:

$$A = \{v : \Omega \rightarrow \mathbb{R} : v(x) \text{ is continuous, } v'(x) \text{ is piecewise continuous and bounded, and } v(x, y) = g(x, y) \text{ on } \Gamma_1\}. \quad (20)$$

Note that the class of admissible functions requires only the Dirichlet boundary conditions (on  $\Gamma_1$ ). The Robin boundary

conditions (on  $\Gamma_2$ ), 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  $\Gamma_1$  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  $\Gamma_2$ ) 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:  $N_1, N_2, \dots, N_n$  (that lie in  $\Omega \cup \Gamma_2$ ), and the Dirichlet boundary nodes  $N_{n+1}, N_{n+2}, \dots, N_m$  (that lie on  $\Gamma_1$ ). Denote the basis function  $\Phi_{N_i}$  corresponding to node  $N_i$  simply by  $\Phi_i$ . It is important that nodes be placed at all endpoints (interfaces) of  $\Gamma_1/\Gamma_2$  and that these endpoints be counted as Dirichlet boundary nodes (i.e., grouped with those in  $\Gamma_1$ ).**

**Step #2: Use the Dirichlet BCs  $u(x, y) = g(x, y)$  on  $\Gamma_1$  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  $i = n+1, n+2, \dots, m$ .**

**Step #3: Assemble the  $n \times n$  stiffness matrix  $A$  and load vector  $b$  needed to determine the remaining coefficients  $c_1, c_2, \dots, c_n$  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  $c_1, c_2, \dots, c_n$  will eventually be determined as the solution vector  $c = [c_1 \ c_2 \ \dots \ c_n]'$  of a linear system (14)  $Ac = b$ . The stiffness matrix  $A$  and load vector  $b$  will, in general, have entries given as follows:

$$a_{ij} = \iint_{\Omega} [p \nabla \Phi_i \cdot \nabla \Phi_j + q \Phi_i \Phi_j] dx dy + \int_{\Gamma_2} r \Phi_i \Phi_j ds \quad (1 \leq i, j \leq n) \quad (21)$$

and

$$b_j = \iint_{\Omega} f \Phi_j dx dy + \int_{\Gamma_2} h \Phi_j ds - \sum_{s=n+1}^m c_s \left\{ \iint_{\Omega} [p \nabla \Phi_s \cdot \nabla \Phi_j + q \Phi_s \Phi_j] dx dy + \int_{\Gamma_2} r \Phi_s \Phi_j ds \right\} \quad (1 \leq j \leq n). \quad (22)$$

In these formulas the integrals over  $\Gamma_2$  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 timeconsuming part of the FEM is still the assembly process. The mechanics are as in the purely Dirichlet case (just replace (15<sup>ℓ</sup>) and (16<sup>ℓ</sup>) 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  $\int_{\Gamma_2 \cap r_1} F ds$ , where  $F$  is any function on  $\Gamma_2$  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  $T_\ell$ . Letting  $\vec{v} = N_2 - N_1$ , we can write:

$$\int_L F ds = \int_0^1 F(N_1 + s\vec{v}) \|\vec{v}\| ds \quad (23)$$

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:

$$\int_0^1 f(x) dx \approx (1/6) \{f(0) + 4f(1/2) + f(1)\} \quad (24)$$

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:

$$\int_L F ds \approx (\|N_1 - N_2\|/6) \{F(N_1) + 4F([N_1 + N_2]/2) + F(N_2)\} \quad (25)$$

**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  $3 \times 2$  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:  $\begin{bmatrix} 1 & 2 \end{bmatrix}$ ,  $\begin{bmatrix} 1 & 3 \end{bmatrix}$ , and  $\begin{bmatrix} 2 & 3 \end{bmatrix}$ . 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,  $\varepsilon_{ij}$  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:

$$\int_{\varepsilon_{12} \cup \varepsilon_{21}} 4ds = 8 + 4\sqrt{13}, \quad \int_{\varepsilon_{12} \cup \varepsilon_{13}} \cos(\pi x/4 + \pi y/2) ds = 2/\pi.$$

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  $\Omega = \{(x,y) : 0 \leq x \leq 10, 0 \leq y \leq x(10-x)\}$

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

(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  $\omega$ :

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

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  $\delta^2$ . 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

$$\delta \approx \sqrt{\text{area}(\Omega)/m} \quad (26)$$

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  $\delta = \sqrt{500/3/350} \approx 0.6901\dots$

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

```
» 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, y temp) 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  $\delta$ . 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  $\delta$ , 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 xrefK10
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
    (xnodecount)=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
```

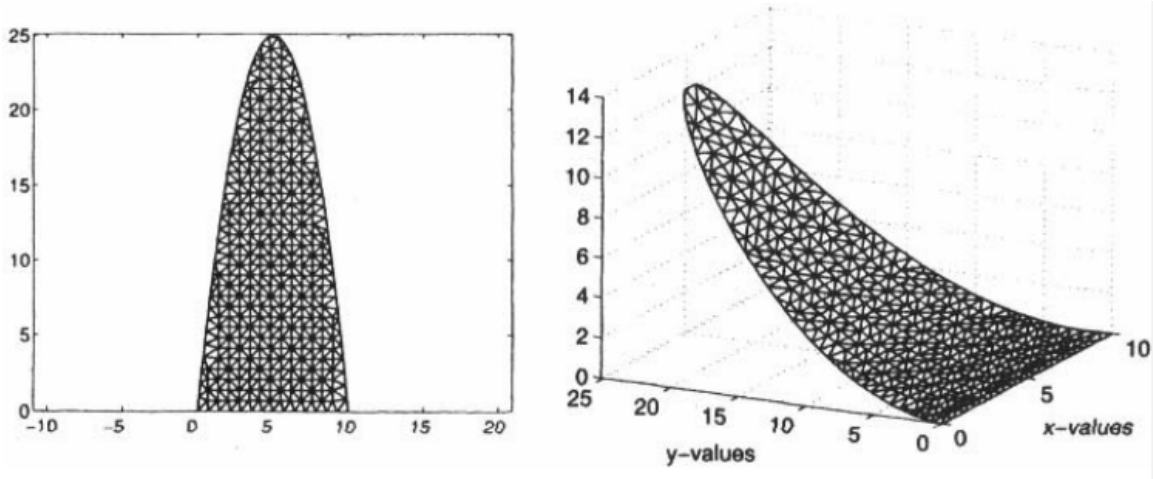
The triangulation is shown in Figure 13.41(a). From the way the nodes were constructed, the boundary nodes come after the interior nodes and the first boundary node is on the parabolic portion of the boundary. We can thus find the key indices by:

```
>> nint=min (f ind(abs (y-bdyf(x) )<10*eps))-1 % number of interior nodes
→ nint = 307 >> n=find(x==10&y==0)-1 % number of interior/Robin node-;;
→ 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):

```
>> dirl = 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.  $(15^\ell)$  and  $(16^\ell)$ ) are as follows:

$$a'_{\alpha\beta} = \iint_{T_i} [\nabla \Phi_{i_\alpha} \cdot \nabla \Phi_{i_\beta}] dx dy \quad (1 \leq \alpha, \beta \leq 3)$$

and

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

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

For each element index  $\ell$ , these coefficients need to be computed only when the nodes  $i_\alpha$  and/or  $i_\beta$  are interior or Robin nodes (i.e.,  $i_\alpha, i_\beta \leq n \equiv 373$ ) 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=find(nodes<=n); %global interior/Robin node indices% find coefficient
    [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) ' ;
        abe(i,:)=abccoeff;
end

determine if there are any Robin edges
marker=0; %will change to 1 if there are Robin edges.
roblocind=find(nodes==dirl|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,modes) & ismember(k+1, rnodes)
                        robedges(count,:)=find(nodes==k ) find(nodes==k+1)];
                        count=count+1; marker =1;
                end
        end
end
end

% update load vector
for il=1:length(intnodes)
        ail = num2str(abc(il, 1) , 10);
        bil = num2str(abc(il,2) ,10);
        cil = num2str(abc(il,3),10) ;
        fun=inline([ail,'*x+',bil, '*y+', cil],'x','y');
        integ=-1/25*gaussianintapprox(fun,xyt,xyr,xys);
        b(intnodes(il))=b(intnodes(il))+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) ) ','* (' ,ali,'*x+',bil,...)
        *y+, cil,')'], 'x','y');
        b(intnodes(il))=b(intnodes(il))+bdyintapprox(prod, elemnodes,...
        robedges);
        end
        end
        clear roblocind modes 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).<sup>16</sup>

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.

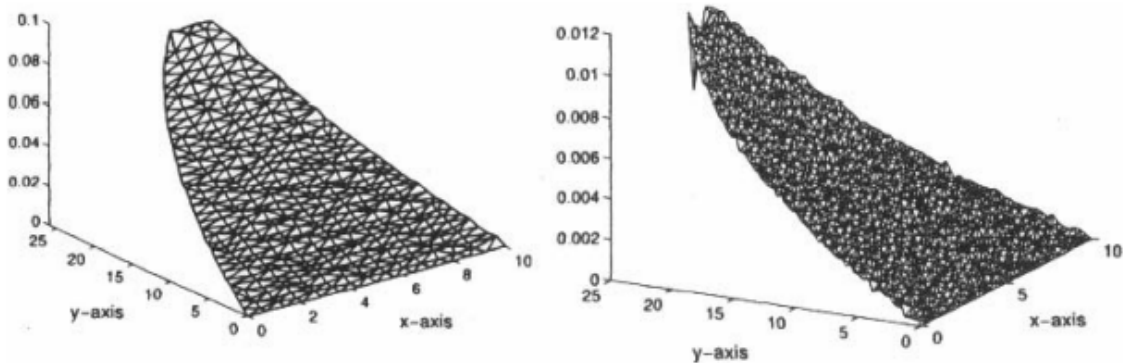
**EXERCISE FOR THE READER 13.15:** Consider the following steady-state heat distribution problem on the parabolic region  $\Omega = \{(x,y) : 0 \leq x \leq 10, 0 \leq y \leq x(x-10)\}$  of the preceding example:

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

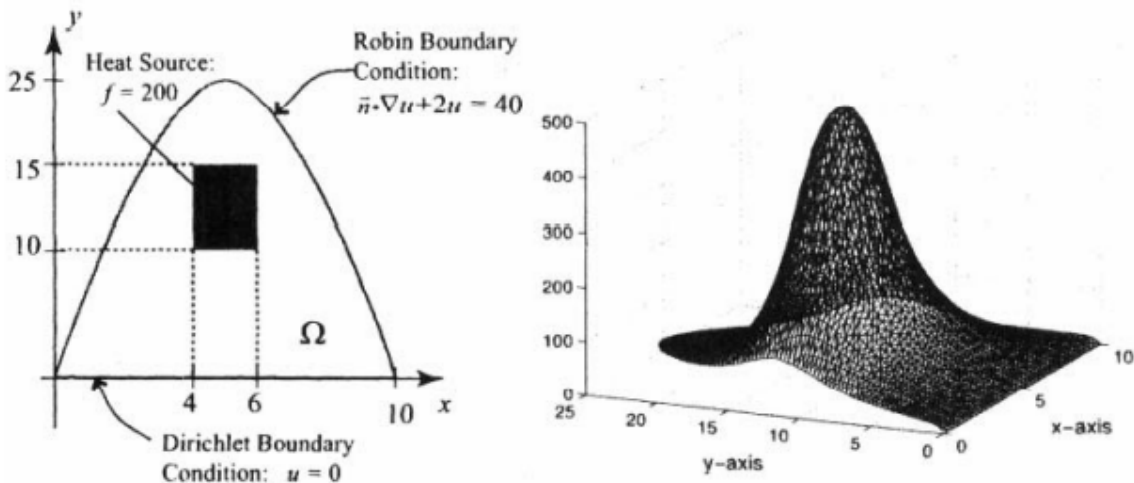
where the source function is given by:  $f(x,y) = \begin{cases} 200, & \text{if } 4 \leq x \leq 6 \text{ and } 10 \leq y \leq 15 \\ 0, & \text{otherwise.} \end{cases}$

The region  $\Omega$  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.

(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%.

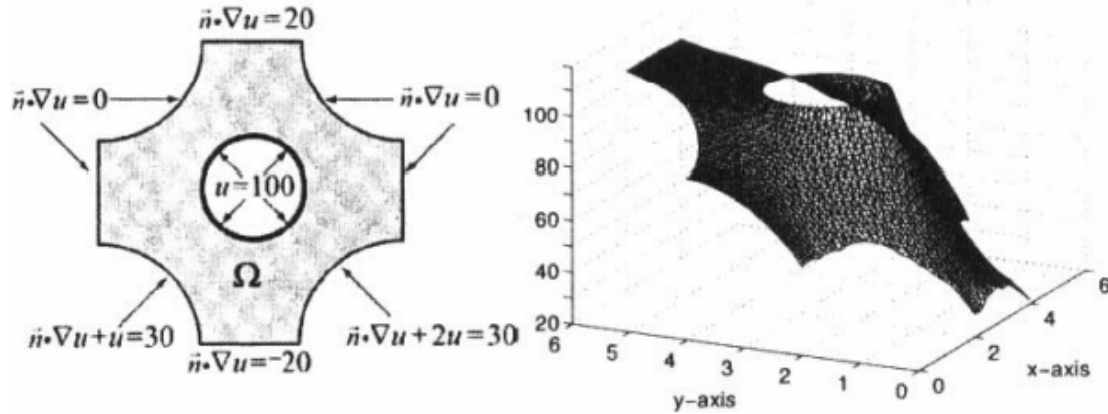


<sup>16</sup>For the convenience of the reader, the entire MATLAB codes for this example (and other longer examples and exercises for the reader of this chapter) are included as downloadable text files on the ftp site for this book (see the preface for the URL of this ftp site). These codes can easily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

**FIGURE 13.43:** 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) Construct 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:

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

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:

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

and plot the resulting numerical solution.

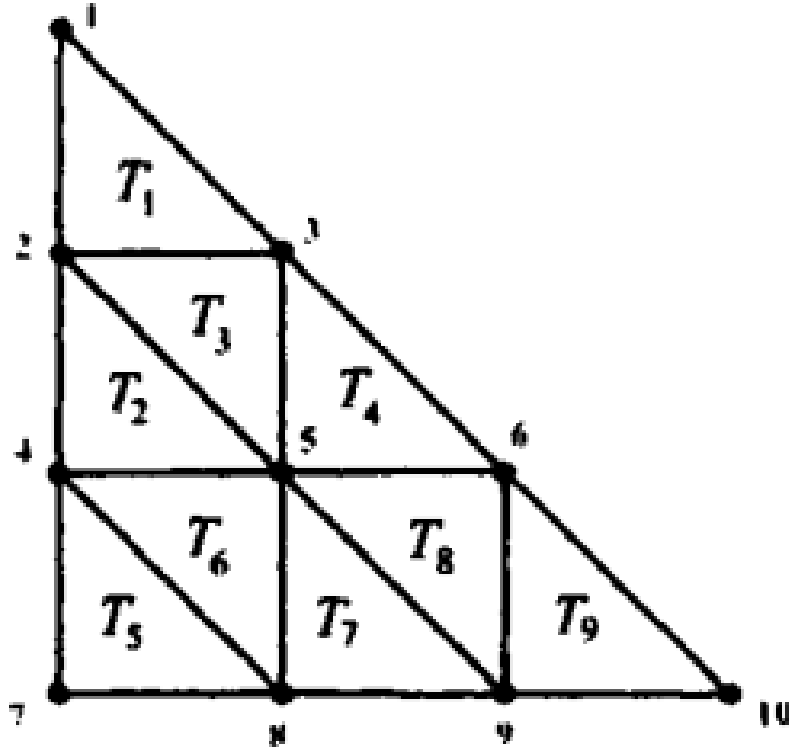
(b) Check that  $u(x, y) = x^2 + y^2$  is the exact solution of the BVP; compare the numerical solution with this exact solution.

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

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

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); 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).





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

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

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

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:

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

**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  $-\nabla \cdot (e^{x+y}u) + u = y$  but all else as in the original problem.

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

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

- (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)  $\nabla \cdot (e^{x+y}u) = 0$ , (ii)  $\nabla \cdot (e^{x+y}u) = 3$ , (iii)  $\nabla \cdot (e^{x+y}u) = -3$ , (iv)  $-\nabla \cdot (e^{x+y}u) + u = 3$

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

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

- (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)  $\nabla \cdot (e^{x+y}u) = 0$ , (ii)  $\nabla \cdot (e^{x+y}u) = 3$  (iii)  $\nabla \cdot (e^{x+y}u) = -3$  (iv)  $\nabla \cdot (e^{x+y}u) + u = 3$

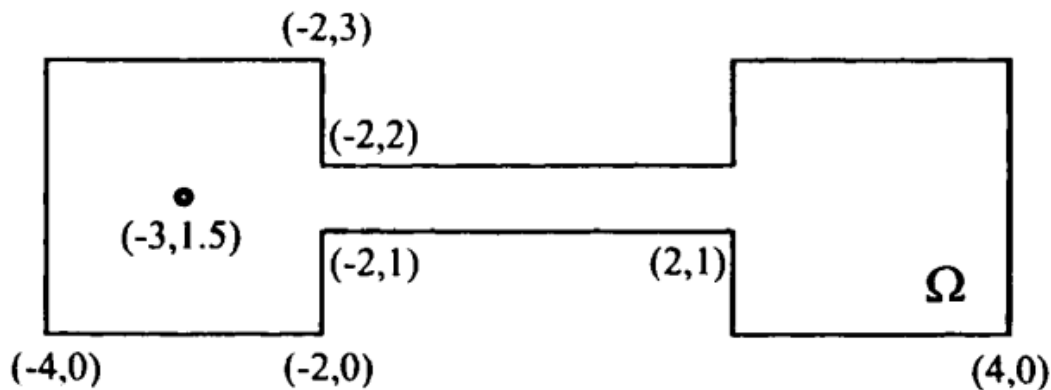
11. This exercise will use the FEM to solve the heat problem (1)

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

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  $(0.3, 0.25)$ , 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 `de launay` 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  $-\nabla \cdot ([x^2 + y^2 + 1] \nabla u) + u = \cos(xy)$ .
14. (*Determination of Maximum Tolerable Heat*) Consider the domain of Figure 13.46:

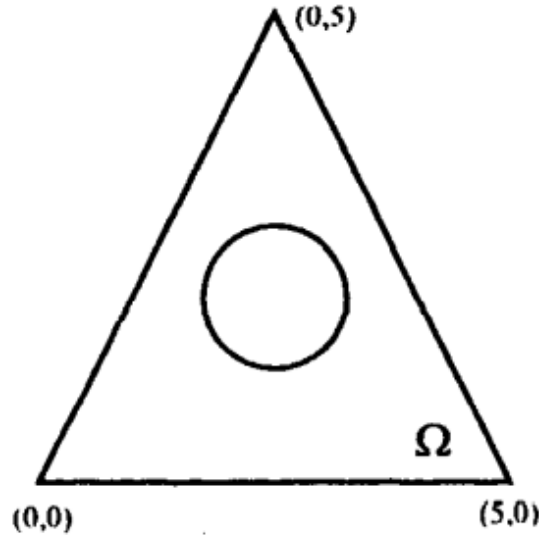


**FIGURE 13.46:** A domain consisting of two squares joined by a rectangular neck.

- (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 insulated  $\vec{n} \cdot \nabla u = 0$  while the right edge will be maintained at a certain temperature  $u = T_{\text{hot}}$ . What is the maximum value of  $T_{\text{hot}}$  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  $\Omega$  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  $\Omega$  having between 300 and 400 (essentially equally spaced) nodes.  
(b) Create a triangulation of  $\Omega$  having between 1500 and 2000 nodes.  
(c) Use the FEM with the triangulation of part (a) to solve the following BVP on  $\Omega$ :

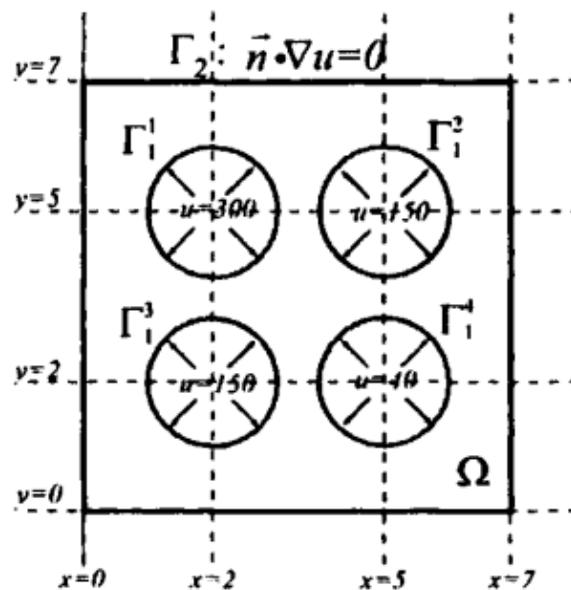
$$\begin{cases} \Delta u = 0 & \text{on } \Omega \\ \vec{n} \cdot \nabla u = 10, & \text{on triangle} \\ u = 100, & \text{on circle} \end{cases}$$

Plot your result and then repeat with the triangulation of part (b). (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)  $-\Delta u + e^{x/2}u = x^2/2$



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

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  $\Gamma_1^2$  and  $\Gamma_1^3$  to be  $\vec{n} \cdot u = 0$  (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  $\Gamma_1^2$  and  $\Gamma_1^3$  to be the Robin conditions:  $\vec{n} \cdot u = 20$  and  $\vec{n} \cdot u = -20$ , respectively, but keeping all else the same.
- (g) Repeat both parts (c) and (d) on the modified BVP gotten by changing the boundary conditions on  $\Gamma_1^2$  and  $\Gamma_1^3$  to be the Robin conditions:  $\vec{n} \cdot u + u = 20$  and  $\vec{n} \cdot u + u = 20$ , 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  $\nabla \cdot (([2^{x/2} + y] u) + u = 10$ , but keeping all else the same.



**FIGURE 13.48:** Boundary conditions for the heat problem of Exercise 11. The outer square boundary  $\Gamma_2$  is insulated, while the four circular inner boundary portions  $\Gamma_1^i$ ,  $1 \leq i \leq 4$ , are each maintained at the indicated temperatures.

17. Let  $\Omega$  be the domain between the  $x$ -axis and the graph of  $y = e^x$  from  $x = 0$  to  $x = 4$ . (a) For the function  $u(x, y) =$

$\sin(x/(y+1)) + x^2y/25$ , determine functions  $g(x)$ ,  $f(x,y)$  and  $h(x,y)$  so that  $u(x,y)$  solves the following BVP:

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

(b) Construct a triangular mesh of  $\Omega$  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  $2 \times 1$  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  $\int_Q f(x,y) dx dy$ , 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):

$$\int_T f(x,y) dx dy \approx w_1 f(\xi_1) + w_2 f(\xi_2) + \cdots + w_n f(\xi_n)$$

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  $A$  is the diameter of the triangle  $T$ .

22. Show that the Gauss quadrature formula (23):

$$\int_T f(x,y) dx dy \approx \frac{\text{Area}(T)}{3} \{f(V_1) + f(V_2) + f(V_3)\}$$

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):

$$\int_T f(x,y) dx dy \approx \frac{\text{Area}(T)}{3} \{f([V_1 + V_2]/2) + f([V_1 + V_3]/2) + f([V_2 + V_3]/2)\}$$

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.

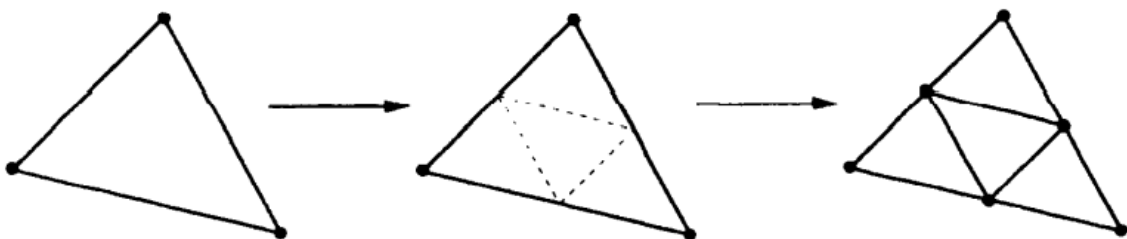
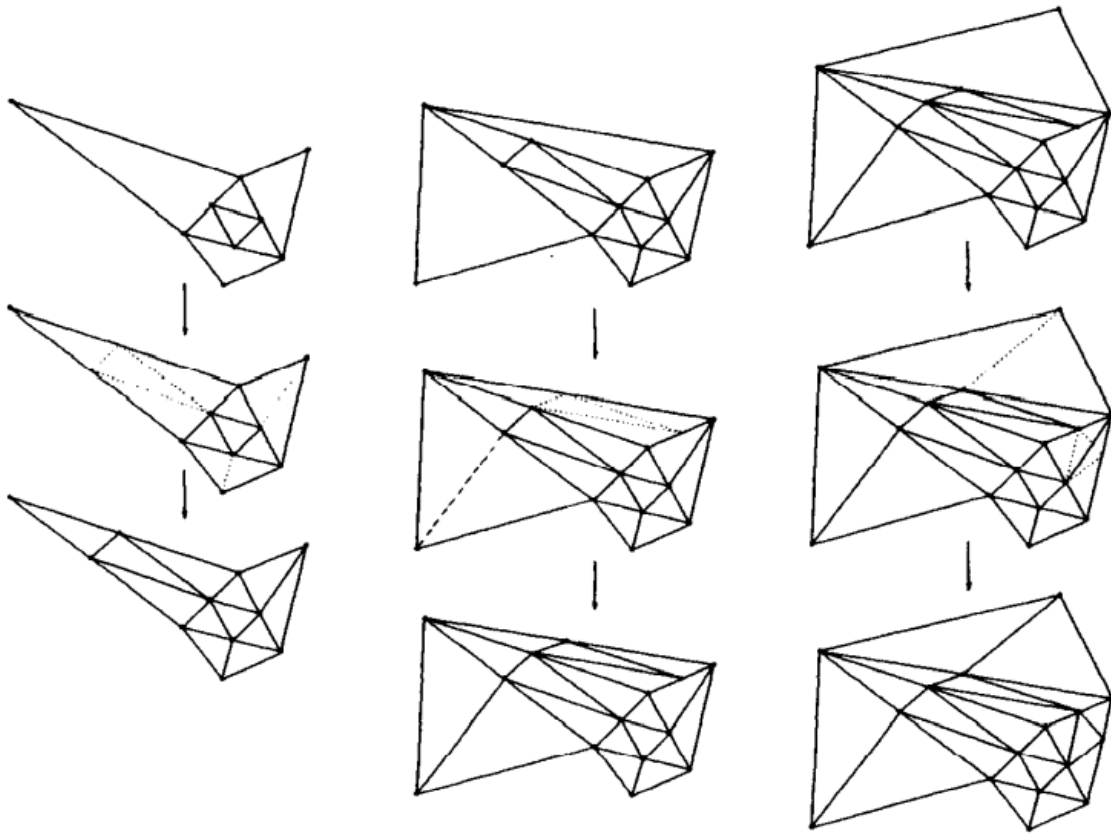


FIGURE 13.49: A refinement scheme for triangular meshes that is easily programmed into FEM routines.

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 triangularon  $A_0$  of a given domain. For the first refinement  $A_1$  of  $A_0$ , the node set will be the node set of  $A_0$ , along with the midpoints of all element edges. Each element of  $A_0$  gives rise to four elements of  $A_1$  as in Figure 13.49. This procedure can be iterated to get a sequence of successively finer triangulations  $A_0 A_1 A_2 \dots$ . We point out two very nice properties: (i) the node set of  $A_n$  is contained (i) the node set of  $A_{n+1}$  (making it simple to compare FEM solutions on successive triangulations), and (ii) the minimum angle of any element of  $A_{n+1}$  equals the minimum angle of any element of  $A_n$  (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 triangularon of a planar domain and `tri` is the corresponding three-column matrix of node numbers of the elements of the triangularon. (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  $A_0$  being the triangularon 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:  $A_1$ ,  $A_2$ , and  $A_3$  (c) With  $A_0$  being the triangularon 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:  $A_1$ , and  $A_2$  (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  $A_n$ , should not be too much smaller than the minimum angle of all elements of  $A_0$ . 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 (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 to some specified indicator, say more than double of the average)<sup>17</sup>

(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 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 adjacent 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<sup>18</sup>). 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

<sup>17</sup>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.

<sup>18</sup>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.

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).

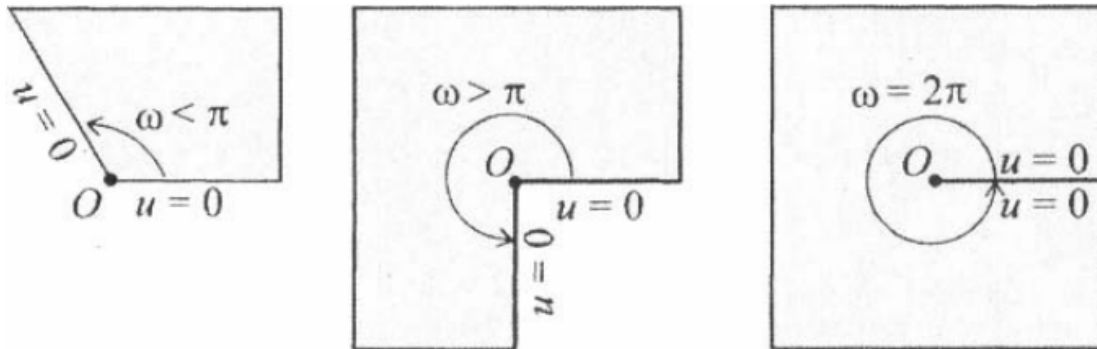
(a) Write a MATLAB program that will perform the above adaptive scheme on the BVP and initial triangulation of Example 13.5. What happens when you run this program? Repeat, but now change the flagging criterion in (iii) to be that the oscillation of the FEM solution over an element exceeds  $1/10$ . 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).

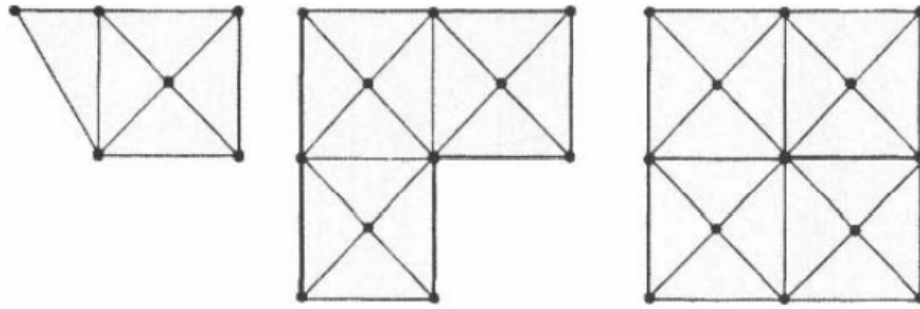
(d) Do you have any ideas for an alternative mesh refinement scheme (satisfying the two constraints mentioned above)?

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 comers 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  $2\pi$  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 strategies to mitigate problems that might arise. We will examine a certain Dirichlet problem for the Laplace equation on such a domain where the exact solution is known. For any angle  $\omega$ , where  $0 < \omega \leq 2\pi$ , we let  $\Omega_\omega$  denote the subdomain of all points in the unit square  $-1 \leq x, y \leq 1$  whose polar coordinates  $(r, \theta)$  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  $\Omega_\omega$  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 resolve 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.

# APPENDIX A: INTRODUCTION TO MATLAB'S SYMBOLIC TOOLBOX

## A.1: WHAT ARE SYMBOLIC COMPUTATIONS?

This appendix is meant as a quick reference for occasions in which exact mathematical calculations or manipulations are needed and are too arduous to expediently do by hand. Examples include the following:

1. Computing the (formula) for the derivative or antiderivative of a function
2. Simplifying or combining algebraic expressions
3. Computing a definite integral exactly and expressing the answer in terms of known functions and constants such as  $\pi, e, \sqrt{7}$  (if possible)
4. Finding analytical solutions of differential equations (if possible)
5. Solving algebraic or matrix equations exactly (if possible)

Such exact arithmetic computations are known collectively as **symbolic computations**. MATLAB is unable to perform symbolic computations but the Symbolic Math Toolbox is available (or included with the Student Version), which computing system. Thus, MATLAB has essentially subcontracted symbolic computations to MAPLE, and acts as a "middleman" so that it is not necessary to use two separate softwares while working on problems. Invoking such symbolic capabilities needs specific actions on the user's part, such as declaring certain variables to be symbolic variables. This is a safety device since symbolic calculations are usually much more expensive than the default floating point calculations and are usually not called for (see Chapter 5). It is important to point out that symbolic expressions are different data types than the other sorts of data types that MATLAB uses. Consequently, care needs to be taken when passing data from one type of data to the other. Moreover, most mathematical problems have answers that cannot be expressed in terms of well-known functions (e.g.,  $\ln(x)$ ,  $\sqrt{x}$ ,  $\arcsin(x)$ ) and/or constants (e.g.,  $e, \pi, \sqrt{2}$ ), and therefore cannot be solved symbolically.

There are also circumstances where the precision of MATLAB's floating point arithmetic is not good enough for a given computation and we might wish to work in more than the 15 (or so) significant digits that MATLAB uses as a default. As a middle ground between this and exact arithmetic, the Symbolic Toolbox also offers what is called variable precision arithmetic, where the user can specify how many significant digits to work with. We point out that there are a few special occasions where symbolic calculations have been used in the text.

The remainder of this appendix will present a brief survey of some of the functionality and features of the Symbolic Toolbox that will be useful for our needs. All of the MATLAB code and output given in a particular section results from a new MATLAB session having been started at the beginning of that section.

## A.2: ANALYTICAL MANIPULATIONS AND CALCULATIONS

To begin a symbolic calculation, we need to declare the relevant variables as symbolic. To declare  $x, y$  as symbolic variables we enter:

```
» syms x y
```

Let's now do a few algebraic manipulations. The basic algebra manipulation commands that MAPLE has are as follows: `expand`, `factor`, `simplify`; they work on algebraic expressions just as anyone who knows algebra would expect. The next examples will showcase their functionality. We point out that any new variable introduced whose formula depends on a symbolic variable will also be symbolic.

```
» p2 = (x+2*y)^2; , p4 = (x+2*y)^4;
» expand(p2) %Multiplies out the binomial product.
->ans = x^2+4*x*y+4*y^2
» expand(p4)
-> ans = x^4+8*x^3*y+24*x^2*y^2+32*x*y^3+16*y^4
» pretty(ans) %Puts the answer in a prettier form.
»
4      3      2      2      3      4
x  + 8 x y + 24 x y + 32 x y +16y
```

In general, for any sort of analytical expression `exp`, the command `expand(exp)` will use known analytical identities to try and rewrite `exp` in a form in which sums and products are expanded whenever possible.

```
» pretty (expand (tan (x+2*y) )) ->
```

$$\frac{\tan(x) + 2 \frac{\tan(y)}{1 - \tan(y)}}{1 - 2 - \tan(y)^2} \tan(x) \tan(y)$$

To clean up (simplify) any sort of analytical expression (involving powers, radicals, trig functions, exponential functions, logs, etc.), the `simplify` function is extremely useful. » `simplify(log(2*sin(x)^2+cos(2*x)))` ->ans=0

```
» h = x^6 - x^5 - 12*x^4 - 2*x^3 + 41*x^2 + 51*x + 18;
```

```
» pretty(factor(h))
```

```
->      2      3
(x+2 (x-3) (x+1))
```

This function will also factor positive integers into primes. This brings up an important point. MATLAB also has a function `factor` that (only) does this latter task. Due to the limitations of floating point arithmetic, MATLAB's version is more restrictive than MAPLE's; it is programmed to give an error if the input exceeds  $2^{32} \approx 4.2950e+009$ .

```
» factor(3^101-1)
```

```
??? Error using ==> factor
```

The maximum value of `n` allowed is  $2^{32}$

```
» factor(sym(3^101-1)) %declaring the integer input as symbolic
```

```
%brings forth the MAPLE version this command.
```

```
»ans = (2)^110*(43)*(47)*(89)*(6622026029)
```

Whereas the Student Version of MATLAB includes access to many of the Symbolic Toolbox commands that one might need to supplement MATLAB functionality, the complete Symbolic Toolbox (for MATLAB's professional version) includes unrestricted access to all of MAPLE's commands. All of the Symbolic Toolbox commands that we discuss in this Appendix are available with the Student Version. To learn more about additional Symbolic Toolbox commands available on the version of MATLAB that you are using, consult the Help menu.

The `factor` function is programmed to look only for real rational factors, so it will not perform factorizations such as  $x^2 - 3 = (x + \sqrt{3})(x - \sqrt{3})$  or  $x^2 + 1 = (x + i)(x - i)$ . Recall (Chapter 6) that it is not always possible to find explicit expressions for all roots/factors of a polynomial, but nevertheless, by the fundamental theorem of algebra, any degree  $n$  polynomial always has  $n$  roots (counted according to multiplicity) that can be real or complex numbers. In cases where it is possible, the `solve` command can find them for us; otherwise, it produces decimal approximations.

<code>solve (exp,var ) -&gt;</code>	If <code>exp</code> is a symbolic expression that involves the symbolic variable <code>var</code> , this command asks MAPLE to find all real and complex roots of the equation <code>exp=0</code> . In cases where they cannot be found exactly (symbolically), numerical (decimal) approximations are found. If there are additional symbolic variables, MAPLE solves for <code>var</code> in terms of them.
-------------------------------------	---

To solve the equation  $x^5 - 5x^4 + 8x^3 - 40x^2 + 16x - 80 = 0$ , we simply enter:

```
»solve (x^5 - 5*x^4 + 8*x^3 - 40*x^2 + 16*x - 80 )
```

```
» %shorter syntax if only one var
```

```
->ans = [2*i] [2i] [5]
```

```
[-2*i] [-2i]
```

The slightly perturbed polynomial equation  $x^5 - 5x^4 + 8x^3 - 40x^2 + 16x - 78 = 0$ , also has five different roots, but they cannot be expressed exactly, so MAPLE will give us numerical approximations, in its default 32 digits:

```
» solve (x^5 - 5*x^4 + 8*x^3 - 40*x^2 + 16*x - 78)
```

```
->ans = [ -.28237724125630031806612784925449e-1 -
2.1432362125064684675126753513414'i]
[ -.28237724125630031806612784925449e-1 +
2.1432362125064684675126753513414*i]
[ .29428740076409528006464576345708e-1 -
1.8429038593310837866143850920505*1]
[ .29428740076409528006464576345708e-1 +
1.8429038593310837866143850920505*1]
[ 4.9976179680984410076002964171595]
```

We can get the quadratic formula for the solutions of  $ax^2 + bx + c = 0$  with the following commands:

```
» syms a b c, solve(a*x^2+b*x+c,x)
-> ans = [ 1/2/a*(-b+(b^2-4*a*c)^(1/2))][1/2/a*(-b-(b^2-4*a*c)^(1/2))]
```

Similarly, the Tartaglia formulas for the three solutions of the general cubic  $ax^3 - bx^2 + cx + d = 0$ , could be obtained.

### A.3: CALCULUS

Table A.I summarizes the Symbolic Toolbox commands needed to perform the most common "clerical" tasks in calculus: differentiation and integration.

**TABLE A.I:** Differentiation and integration using the Symbolic Toolbox.

Assume that f has been stored as a symbolic function of symbolic variables: f(x) (or /(x,y,...), if we have a function of several variables.	
<code>diff(f,x) -&gt;</code>	Computes $f'(x) = \frac{df}{dx}$ ( or $\frac{\partial f}{\partial x}$ )
<code>diff(f,x) -&gt;</code>	Computes $f'(x) = \frac{d^2f}{dx^2}$ ( or $\frac{\partial^2 f}{\partial x^2}$ )
<code>int(f,x) -&gt;</code>	Calculates (if possible) an antiderivative of $f(x)$ : $\int f(x)dx$ (does not add on integration constant). If there are other variables, they are treated as constant parameters.
<code>int(f,x,a,b) -&gt;</code>	Calculates (exactly, if possible) the definite integral: $\int_a^b f(x)dx$ (does not add on integration constant). If there are other variables, they are treated as constant parameters.

$$\begin{array}{lll}
 \text{(a) } \frac{d}{dx} x^x & \text{(b) } \frac{\partial^3}{\partial x \partial y^2} \left( \frac{\cos(x+y^2+z^3)}{1+x^2+y^2} \right) & \text{(c) } \int \ln(x) dx \\
 \text{(d) } \int \sin(x^2) dx & \text{(e) } \int_0^1 \sin(x^2) dx & \text{(f) } \int_{-\infty}^{\infty} e^{-x^2} dx
 \end{array}$$

SOLUTION: Part (a):

```
» syms x y z
» diff(x^x)-> ans = x^x*(log(x)+1)
```

So the answer is  $x^x(\ln x + 1)$ .

Part (b):

```
» f=cos(x+y^2+z^3)/(1+x^2+y^2);
» pdf=diff(diff(f,y,2),x) -> pdf=
4*sin(x+y^2+z^3)*y^2/(x^2+1+y^2)+8*cos(x+y^2+z^3)*y^2/(x^2+1+y^2)^2*x
2*cos(x+y^2+z^3)/(x^2+1+y^2)+4*sin(x+y^2+z^3)/(x^2+1+y^2)^2*x+8*cos(x+y^2+z^3)*y^2/(x^
2+1+y^2)^2-32*sin(x+y^2+z^3)*y^2/(x^2+1+y^2)^3*x-8*sin(x+y^2+z^3)/x^2+1+y^2)^3*y^2-
48*cos(x+y^2+z^3)/(x^2+1+y^2)^4*x+2*sin(x+y^2+z^3)/(x^2+1+y^2)^2+8*cos(x+y^2+z^3)
/(x^2+1+y^2)^3*x
```

We shall refrain from putting this mess in usual mathematical notation, but we will do something else with it later (which is why we gave it a name).

```
Part(c):>> int(log(x)) -> ans = x*log(x)-x
```

```
Part (d):>> int(sin(x^2),x) -> ans = 1/2*2^(1/2)*pi^(1/2)*FresnelS(2^(1/2)/pi^(1/2)*x)
```

This answer to part (d) needs a bit of explanation. Most indefinite integrals cannot be expressed in terms of the elementary functions. Using some additional special functions (e.g., Bessel functions, hypergeometric functions, the error function, and the above Fresnel sine function), additional integrals can be computed (but still only relatively few); thus MAPLE has found an antiderivative for us, but for most practical purposes this answer by itself is not so interesting. A similar result turns up (by the fundamental theorem of calculus) for the corresponding definite integral.

Part (e): `>> int(sin(x*2),x,0,1) -> ans = 1/2*FresnelS(2^(1/2)/pi^(1/2))*2^(1/2)*pi^(1/2)`

The following commands show how to get a more useful decimal answer out of this or any answer to a symbolic computation:

<code>vpa (a, d) -&gt;</code>	If <i>a</i> is a symbolic answer representing a number and <i>d</i> is a nonnegative number, this command will convert the number <i>a</i> to decimal form with <i>d</i> significant digits, <i>vpa</i> stands for variable precision arithmetic. The default value is $d = 32^1$
<code>digits(d) vpa (a) -&gt;</code>	Has the same result as above, but now the default value of $d=32$ digits of MAPLE's arithmetic is reset to <i>d</i> in subsequent calculations.

`>> vpa (ans) -> ans = 31026830172338110180815242316540`

If we (for whatever reason) wanted to see the first 100 digits of  $\Pi$ , we could simply enter:

`>> vpa (pi, 100) -> ans = 3.14159265358979323846264338327950288419716939937510582 097494459`

Part (f): Improper integrals are done with the same syntax as proper integrals.

`>> int (exp(-x^2),x,-Inf,Inf) -> ans = pi^(1/2)`

Thus we get that  $\int_{-\infty}^{\infty} e^{-x^2} dx = \sqrt{\pi}$

Often, we need to evaluate a symbolic expression or substitute some of its variables with other variables or expressions. The following command `subs` is very useful in this respect:

<code>subs (S,old ,new) -&gt;</code>	If <i>S</i> is a symbolic expression, <i>old</i> is a symbolic variable appearing in <i>S</i> (or a vector of variables), <i>new</i> is a symbolic number or symbolic expression (or a vector of such things having the same size as <i>old</i> ), this command will produce the symbolic expression resulting from substituting in <i>S</i> each occurrence of <i>old</i> by the corresponding expression in <i>new</i> .
--------------------------------------	--

For example, suppose (in the setting of Example A.1) we wanted to compute

$$\left. \frac{\partial^3}{\partial x \partial y^2} \left( \frac{\cos(x+y^2+z^3)}{1+x^2+y^2} \right) \right|_{\substack{x=\pi \\ y=\pi/2 \\ z=0}}$$

From what we have already computed, we could simply enter:

`>> subs (pdf, [x y z], [pi pi/2 0]) -> ans = -0.2016`

Since all symbolic variables were substituted with nonsymbolic (ordinary MATLAB floating point) numbers, the result is now a regular MATLAB floating point number. To retain the accuracy of symbolic computation in the substitution, we could instead enter:

`>> exact=subs(pdf,[ x y z] , sym([p i pi/ 2 0])) ; %suppress messy output  
>> vpa(exact) %could specify more or less digits here .  
->ans = -.20163609585811087949860391144560`

Note that the main difference is that in the latter we declared the numbers to be symbolic (`exact`):

<code>fpn=double(sbn) -&gt;</code>	if <i>sbn</i> is a (MAPLE) symbolic number, this command creates a (MATLAB) floating point number <i>fpn</i> from it essentially by rounding it off to about 16 digits of accuracy.
<code>sbn=sym(fpn) -&gt;</code>	If <i>fpn</i> is a (MATLAB) floating point number, this command creates a (MAPLE) symbolic number <i>sbn</i> from it by treating it as an exact number.

The Symbolic Toolbox has a simple way for computing Taylor series:

<sup>1</sup>Thus, MAPLE uses approximately a 32-digit floating point arithmetic system in cases where exact answers are not possible. This is about double of what MATLAB uses and for many computations is overkill since large-scale calculations would proceed much more slowly. Thus, generally speaking, use of the Symbolic Toolbox should be limited to symbolic computations, except in the occasional instances where, say, the problem being solved is very ill-conditioned and roundoff errors run out of control with IEEE floating point arithmetic (see Chapter 5).

<code>taylor (&lt;fun&gt;, n, a ) -&gt;</code>	If <fun> is a symbolic expression representing a function of a (previously declared) symbolic variable (say x), n is a positive integer, and a is a real number, this command will produce the Taylor polynomial of the function centered at JC = a of order (degree at most) n-1 . The last input a is optional, the default value is a = 0.
--	---

**EXAMPLE A.2:** Obtain the 15th-order Taylor polynomial of  $f(x) = x^2 \tan(x^3)$  centered at  $x = 0$ .

**SOLUTION:**

```
» taylor (x^3*tan(x^2),16)->ans = x^5 + 1/3*x^9 + 2/15*x^13
```

In the notation of Chapter 2, we can thus write  $p_{15}(x) = x^5 + \frac{x^9}{3} + \frac{2x^{13}}{15}$ .

#### A.4: ORDINARY DIFFERENTIAL EQUATIONS<sup>2</sup>

Analytic (symbolic) solutions of ordinary differential equations and systems of them, if they exist, can be found using the `dsolve` function from the Symbolic Toolbox.<sup>3</sup> Since the function has many available features, we roughly indicate the possible syntaxes for its use and give examples of each.

<code>dsolve ('&lt;diff_eq&gt;') -&gt;</code>	Looks for the analytic general solution of the differential equation: <diff_eq>, in which first, second, third, etc. derivatives are denoted by D, D2, D3, etc., using the default independent variable t.
<code>dsolve ('&lt;diff_eq&gt;', 'var') -&gt; )</code>	Works as above but specifies the independent variable to be var .

**EXAMPLE A.3:** Find, if possible, analytic general solutions of the following ODEs:

- (a)  $y' = y^2 - 2y, y = y(t)$
- (b)  $u'' + 5u' - 6u = \cos(x), u = u(x)$
- (c)  $y'' = y^2 - 2y, y = y(t)$

**SOLUTION:** Part (a):

```
» y= dsolve ( 'Dy=y^2-2*y' )
» y=2/(1+2*exp(2*t)*C1)
```

So we have the general solution,  $y(t) = \frac{2}{1+2Ce^{2t}}$ , where C is an arbitrary constant. Note that the `dsolve` did not even require us to declare any symbolic variables. The `subs` function, however, does require symbolic variables. Thus, if we try to set C1 equal to zero in y directly, we get an error message. But by first declaring C1 as a symbolic variable, we get the intended result:

```
» subs(y,C1,0)
??? Undefined function or variable 'C1'

» syms C1
» subs(y,C1,0)
-> ans =2
```

Part (b): If we do not specifically declare x as the independent variable, x will be treated as a constant and we get an unintended solution of a more trivial differential equation. The second MATLAB code below gives us what we want.

```
» dsolve('D2u+5*Du-6*u=cos(x)')
» ans=-1/6*cos(x)+C1*exp(t)+C2*exp(-6*t)

» dsolve('D2u+5*Du-6*u=cos(x)', 'x')
-> ans = -7/74*cos(x)+5/74*sin(x)+C1*exp(x)+C2*exp(-6*x)
```

So we have the general solution:

$$u(x) = C_1 e^x + C_2 e^{-6x} - (7/74)\cos(x) + (5/74)\sin(x)$$

<sup>2</sup>Since this book does not assume that the reader has had any experience with differential equations, it is advised that those readers without such experience wait to read this subsection until they have started studying Part II of the book (ordinary differential equations).

<sup>3</sup>Although most ODEs (like indefinite integrals) do not have analytic solutions, this tool is occasionally useful when dealing with special well-known types of ODE which do have analytic solutions. The Symbolic Toolbox freely uses a collection of special functions when it looks for symbolic solutions

where  $C_1, C_2$  are arbitrary constants.

Part (c):

```
» dsolve('D2y=y^2-2*y', 'x')
-> Warning: Explicit solution could not be found; implicit solution returned.
> In C: MATLAB6p5 toolbox symbolic dsolve.m at line 292
->ans = [ 3*Int(1/(6*a^3-18*a^2+9*C1)^(1/2),a='..y)-x-C2=0,-3*Int(1/(6*a^3-18*a^2+9*
C1)^(1/2),a='..y)-x-C2=0]
```

Thus we see that, despite its simplicity (and similarity to the ODE in part (a)), the ODE of Part (c) does not have symbolic solutions.

The `dsolve` function can also solve initial and boundary value problems, the conditions need only be inserted as additional inputs after the DE:

<code>dsolve(' &lt;diff_ eq&gt;', ' cond1', ' cond2', . . . , ' var') -&gt;</code>	Syntax is as above but with additional inputs corresponding to auxiliary conditions (boundary or initial) which we would like the solution to satisfy.
--	--

**EXAMPLE A.4:** Solve the following ODE problems.

- (a)  $\begin{cases} y'(t) = 2ty \\ y(1) = 1 \end{cases}$   
 (b)  $\begin{cases} y'' + y = e^x \cos(x) \\ y(0) = 1, y'(\pi) = 0 \end{cases}$

SOLUTION:

Part (a):

```
» y = dsolve(Dy = 2*t*y, 'y(1)=1') -> y = 1/exp(1)*exp(t^2)
```

Thus we get the exact solution  $y(t) = e^{t^2-1}$

Part (b):

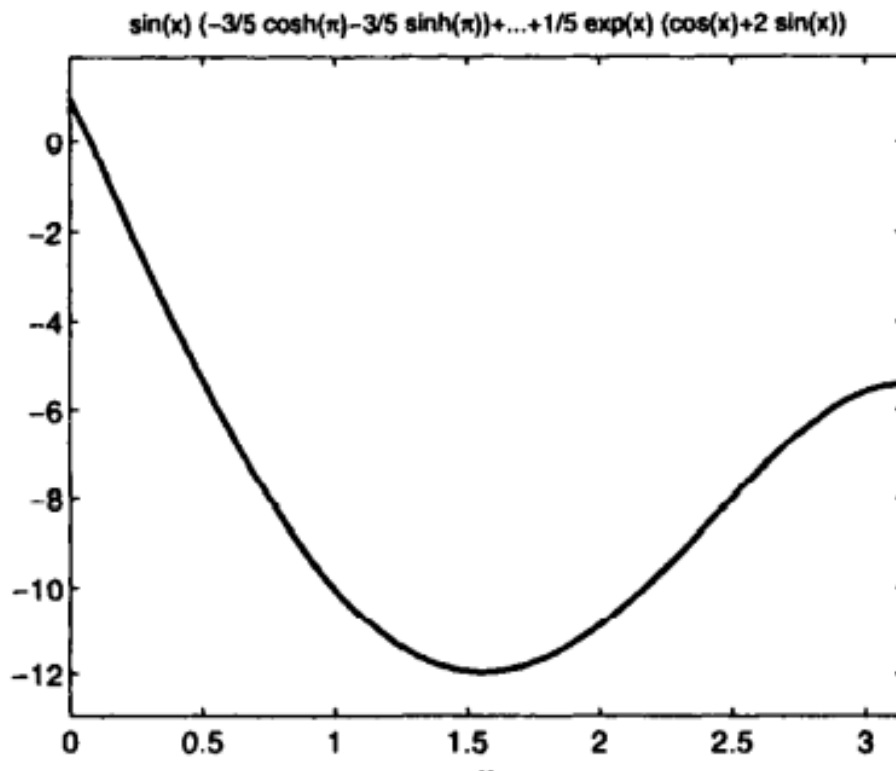
```
» y = dsolve('D2y+y=exp(x)*cos(x)', 'y(0)=1', 'Dy(pi)=0', 'x')
-> ans = -1/10*exp(x)*(sin(2*x)^2*cos(2*x))*cos(x) +
(1/5*(cos(x)+2*sin(x))*exp(x)*cos(x)+2/5*exp(x))*sin(x)+4/5*cos(x)+(-3/5*cosh(pi)-
3/5*sinh(pi))*sin(x)
```

To plot a symbolic function, we could use the `subs` command to create vectors of  $y$ -coordinates and plot using MATLAB as shown in Chapter 1. Alternatively, the Symbolic Toolbox supplies a function `ezplot` that will directly and painlessly plot a symbolic function of a single symbolic variable.

<code>ezplot(f, [a b]) -&gt;</code>	If $f$ represents a symbolic function of a single symbolic variable (say $x$ ), and $a < b$ are real numbers, this command will produce a plot of $f(x)$ over the interval $[a, b]$ .
-------------------------------------	---

With  $y$  still stored as the solution of the last boundary value problem, the following command will result in the plot shown in Figure A.1.

```
» ezplot(y, [0 pi])
```



**FIGURE A.1:** Plot of the solution of the boundary value problem of Example A.4(b).

The final useful feature of `dsolve` is that it can solve systems of ODE. The syntax is a natural extension of the previous codes:

```
dsolve ('<diff_eq1>', '<diff_eq2>', . . .
        'cond1', 'cond2', . . . , 'var') ->
```

Syntax is as above but with additional differential equations with other unknown functions and a listing of all additional conditions to be satisfied by the unknown functions.

**EXAMPLE A.5:** Solve the following linear first order system of ODEs:

$$\begin{cases} x'(t) = 3x + 2y + z, & x(0) = 1 \\ y'(t) = x - y + z, & y(0) = 2 \\ z'(t) = 2x + 2y + 2z, & z(0) = 3 \end{cases}.$$

**SOLUTION**

```
» [x,y,z]=dsolve('Dx=3*x+2*y+z','Dy=x-
y+z','Dz=2*(x+y+z)','x(0)=1','y(0)=2','z(0)=3')

-> x=4/41*(-328*exp(t)+369*exp(-1/2*(-3+41^(1/2))*t)-81*41^(1/2)*exp(-1/2*(-
3+41^(1/2))*t)+81*41^(1/2)*exp(1/2*(3+41^(1/2))*t)+369*exp(1/2*(3+41
^(1/2))*t))/(1+41^(1/2))

y=-2/41*(-738*exp(-1/2*(-3+41^(1/2))*t)-18*41^(1/2)*exp(-1/2*(-3+41^(1/2))*t)-
164*exp(t)+18*41^(1/2)*exp(1/2*(3+41^(1/2))*t)-738*exp(1/2*(3+41^(1/2))*t))/(1+41
^(1/2-1+41^(1/2)))

z=-4/41*(-369*exp(-1/2*(-3+41^(1/2))*t)+81*41^(1/2)*exp(-1/2*(-3+41^(1/2))*t)-
492*exp(1/2*(3+41^(1/2))*t)+81*41^(1/2)*exp(1/2*(3+41^(1/2))*t)-369*exp(1/2*(3+41^(1/2))*t))/(1+41^(1/2))/(-1+41^(1/2))
```

By themselves, these solutions do not appear to be very enlightening. But like any other symbolic functions, they can be manipulated and combined and vectors can be created from them using `subs`, so that much qualitative analysis, as is done in the text, can be performed.



## 0.2. APPENDIX B: SOLUTIONS TO ALL EXERCISES FOR THE READER

NOTE: All of the M-files of this appendix (like the M-files of the text) are downloadable as text files from the ftp site for this text:

[ftp://ftp.wiley.com/public/sci\\_tech\\_med/numerical\\_differential/](ftp://ftp.wiley.com/public/sci_tech_med/numerical_differential/)

Occasionally, for space considerations, we may refer a particular M-file to this site. Also, in cases where a long MATLAB command does not fit on a single line (in this appendix), it will be continued on the next line. In an actual MATLAB session, (long) compound commands should either be put on a single line, or three periods (...) should be entered after a line to hold off MATLAB's execution until the rest of the command is entered on subsequent lines and the ENTER key is pressed. The text explains these and other related concepts in greater detail.

### CHAPTER 1: MATLAB BASICS

**EFR 1.1:** `linspace(-2,3,11)`

**EFR 1.2:** `t = 0:.01:10*pi; x = 5*cos(t/5)+cos(2*t);  
y = 5*sin(t/5)+sin(3*t); plot(x,y), axis('equal')`

**EFR 1.3:** Simply run the code through MATLAB to see if you analyzed it correctly.

### CHAPTER 2: BASIC CONCEPTS OF NUMERICAL ANALYSIS WITH TAYLOR'S THEOREM

**EFR 2.1:**

```
x=-10:.05:10; y=cos(x); p2=1-x.^2/2; p4=1-x.^2/2+x.^4/gamma(5); p6=1-x.^2/2+x.^4/gamma(5)-x.^6/gamma(7); p8=1-x.^2/2+x.^4/gamma(5)-x.^6/gamma(7)+x.^8/gamma(9); p10=p8-x.^10/gamma(11); hold on; plot(x,p10,'k:'), axis([-2*pi 2*pi -1.5 1.5]), plot(x,p8,'c:'), plot(x,p6,'r-'), plot(x,p4,'k--'), plot(x,p2,'g')
```

**EFR 2.2:** Computing the first few derivatives of:

$$f(x) = x^{1/2}, f'(x) = \frac{1}{2}x^{-1/2}, f''(x) = -\frac{1 \cdot 1}{2 \cdot 2}x^{-3/2}, f'''(x) = \frac{1 \cdot 1 \cdot 3}{2 \cdot 2 \cdot 2}x^{-5/2},$$

$f^{(4)}(x) = -\frac{1 \cdot 1 \cdot 3 \cdot 5}{2 \cdot 2 \cdot 2 \cdot 2}x^{-7/2} \dots$  leads us to discover the general pattern:

$f^{(n)}(x) = (-1)^{n+1} \frac{1 \cdot 3 \cdot 5 \cdots (2[n-1]-1)}{2^n} x^{-(2n-1)/2}$  (for  $n \geq 2$ ). Applying Taylor's theorem (with  $a=16, x=17$ ), we estimate the error of this approximation:

$$|R_n(17)| = \left| \frac{f^{(n+1)}(c)}{(n+1)!} 1^{n+1} \right| = \left| \frac{1 \cdot 3 \cdot 5 \cdots (2n-1)}{2^{n+1}(n+1)!} c^{-(2n+1)/2} \right| \leq \frac{1 \cdot 3 \cdot 5 \cdots (2n-1)}{2^n(n+1)!} 16^{-(2n+1)/2} =$$

$\frac{1 \cdot 3 \cdot 5 \cdots (2n-1)}{2^n(n+1)! \cdot 4 \cdot 16^n} = \frac{1 \cdot 3 \cdot 5 \cdots (2n-1)}{2^{5n+2}(n+1)!}$ . We use MATLAB to find the smallest  $n$  for which this last expression is less than  $10^{10}$ ; then Taylor's theorem will assure us that the Taylor polynomial of this order will provide us with the desired approximation.

```
>> n=2; ErrorEst=1*3/gamma(n+2)/2^(5*n+2);  
>> while ErrorEst>1e-10, n=n+1; ErrorEst=ErrorEst*(2*n-1)/(n+1)/2^5;  
end  
>> n->n=7  
>> ErrorEst->ErrorEst = 2.4386e-011 %this checks out.
```

So  $p_7(17) = \sum_{k=0}^7 \frac{1}{k!} f^{(k)}(16) \cdot 1^k$  will give the desired approximation. We use MATLAB to perform and check it:

```
>> sum=16^(1/2)+16^(-1/2)/2; %first-order Taylor Polynomial  
term = 16^(-1/2)/2; %first-order term  
for k=2:7, term = -term*(2*(k-1)-1)/2/16/k; sum=sum+term; end, format  
long  
>> sum->sum = 4.12310562562925 (approximation)  
>>abs (sum-sqrt (17))->ans =1.1590e-011 %actual error excels goal
```

**EFR 2.3:** Using ordinary polynomial substitution, subtraction, and multiplication (and ignoring terms in the individual Maclaurin series that give rise to terms of order higher than 10), we use (9) and (10) to obtain: (a)  $\sin(x^2) - \cos(x^3) =$

$$\left( x^2 - \frac{(x^2)^3}{3!} + \frac{(x^2)^5}{5!} - \dots \right) - \left( 1 - \frac{(x^3)^2}{2!} + \dots \right) = -1 + x^2 - \left( \frac{1}{2!} - \frac{1}{3!} \right) x^6 + \frac{x^{10}}{5!} \dots$$

(b)  $\sin^2(x^2) = \left(x^2 - \frac{(x^2)^3}{3!} + \dots\right) \cdot \left(x^2 - \frac{(x^2)^3}{3!} + \dots\right) = x^4 - \frac{2}{3!}x^3 + \dots$  In each case,  $p_{10}(x)$  consists of all of the terms listed on the right-hand sides.

### CHAPTER 3: INTRODUCTION TO M-FILES

**EFR 3.1** : In the left box we give the stored M-file; in the right we give the subsequent MATLAB session.

<pre>% script file for EFR 3.1: listp2 power =2; while power &lt;= n     power     power=2*power; end</pre>	<pre>» n=5 ; listp 2 -&gt; power = 2, power = 4 » n=264;listp 2 -&gt; power = 2, power = 4, power = 8, power =16, power = 32, power = 64, power =128, power = 256, »n=2917;listp2 -&gt; power = 2, power = 4, power = 8, power =16, power = 32, power = 64, power =128, power = 256, power = 1024, power = 2048</pre>
---	---

Note: If we wanted the output to be just a single vector of the powers of 2, the following modified script would do the job:

```
% script file for EFR 3.1: Iistp2ver2
power =2; vector = [ ]; %start off with empty vector
while power <= n
    vector = [vector power];
    power=2*power;
end, vector
```

For example, with this file stored, if we enter » n=264; Iistp2ver2 , we get the following vector output:  
->vector = 2 4 8 16 32 64 128 256

**EFR 3.2:** With the boxed function M-file below saved, MATLAB will give the following outputs:

```
function f = fact(n)
% FACT f = fact(n) returns the factorial n! of a nonnegative integer
n
f=1;
for i=1:n
    f=f*i;
end
```

```
» fact(4) , fact (10), fact(0)
->ans = 24, 3628800, 1
```

**EFR 3.3** : At any (non-endpoint) maximum or minimum value  $y(x_0)$ , a differentiable function has its derivative equaling zero. This means that the tangent line is horizontal, so that for small values of a  $\Delta x = x - x_0$ ,  $\Delta y/\Delta x$  approaches zero. Thus, the y-variations are much smaller than the x-variations as x gets close to the critical point in question. This issue will be revisited in detail in Chapter 6.

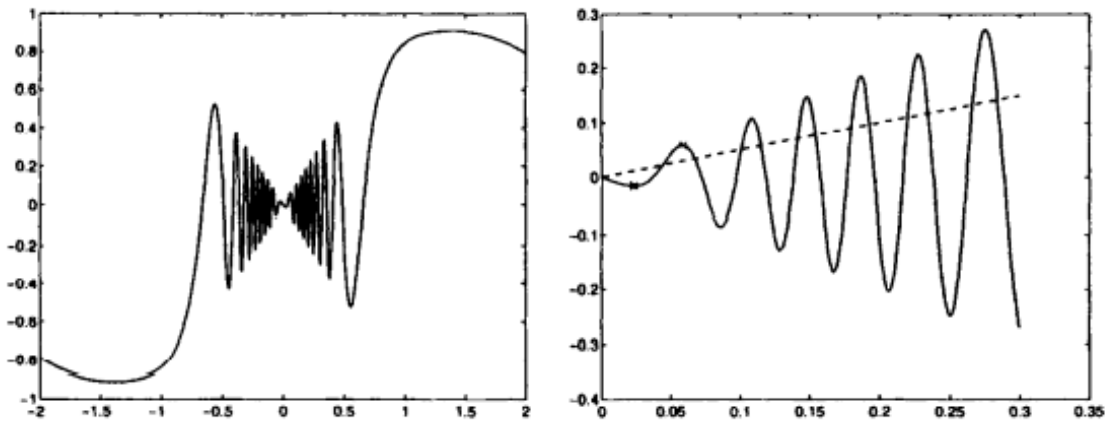
**EFR 3.4:** We have only considered the values of y at a discrete set of (equally spaced)  $x$ -values. It is possible for a function to oscillate wildly in intervals between sets of discrete points (think trig functions with large amplitudes). More analysis can be done to preclude such pathologies (e.g., checking to see that there are no other critical points).

**EFR 3.5** : The M-file for the function is straightforward:

```
function y = wiggly(x)
%Function M-file for the mathematical function of EFR 3.5
y=sin(exp(1./(x.^2 + 0.5).^2)).*sin(x);
```

```
(a)>> x=-2:.001:2 ; plot(x,wiggly(x) ) %plot is shown on left below
(b)>> quad (Gwiggly, 0,2 , 1e-5 ) ->ans = 1.03517910753379
(c) To better see what we are looking for, we create another plot of the
    function zoomed in near x = 0.
>> x=0:.001:.3 ; plot(x,wiggly(x) ) %plot is shown (w/ othe radditions)
on right below.
```

We seek the x-coordinates of the two points marked with "x's" in the figure below.



```
>>xmax=fminbnd('wiggly(x)',0,0.1,optimset('TolX',le-5))
->xmin =0.02289435851906
>>xmax=fminbnd('wiggly(x)',0,0.1,optimset('TolX',le-5))
->xmax =0.05909071987402
Red and green x's can now be added to the graph as follows: >> hold on,
plot (xmin, wiggly (xmin), 'rx'), plot (xmin, wiggly (xmin) , 'gx')
```

This also gives us a visual check that we found what we were looking for.)

(d) To get a rough idea of the location of the x-value we are searching for, we now add the graph of the line  $y = x/2$  (as a black dotted line): `>> plot (x, x/2 , ' k-)` From the graph, we see that the intersection point we are looking for is the one closest to the midpoint of `xmin` and `xmax`.

```
>> xcross=fzero('wiggly(x)-x/2',(xmin+xmax)/2 )
-> xcross =0.04479463640226
```

Let's do a quality check: `>> wiggly (xcross)-xcross/2`  
`->ans = 2.185751579730777e-016 (Very Good!)`

## CHAPTER 4 : PROGRAMMING IN MATLAB B

**EFR 4.1 :** Simply run the code through MATLAB to see if you analyzed it correctly.

**EFR 4.2:** (a) The M-file is boxed below:

```
function [ ] = sum2sq(n)
%M-file for EFR 4.2
for a=1:sqrt(n)
    b=sqrt(n-a^2); %solve n=a^2+b^2 for b
    if b==floor(b); %checks to see if b is integer
        fprintf('the integer %d can be written as the sum of squares
of %d and %d', n,a,b)
        return
    end
end
fprintf('the integer %d cannot be written as the sum of squares', n)
```

(b) We now perform the indicated program runs:

```
>> sum2sq (5) ->the integer 5 can be written as the sum of squares of 1 and 2
>> sum2sq (25) -> the integer 25 can be written as the sum of squares of 3 and 4
>> sum2sq (12233) -The integer 12233 can be written as the sum of squares of 28 and 107
```

(c) The following modification of the above M-file will be more suitable to solving this problem:

```
function flag = sum2sqb(n)
%M-file for EFR 4.2b
flag=0; %will change to 1 if n can be written as a^2+b^2
for a=1:sqrt(n)
    b=sqrt(n-a^2); %solve n=a^2+b^2 for b
    if b==floor(b); %checks to see if b is integer
        flag=1;
        return
    end
end
end
```

The program has output 1 if and only if  $n$  is expressible as a sum of squares; otherwise the output is zero. Now the following simple code will compute the desired integer  $n$ :

```
>> for n=99999:-1:1 , flag=sum2sqb(n);
if flag==0
```

```
fprintf('%d is the largest integer less than 100,000 not expressible
as a sum of squares',n)
break
end
end
```

->99999 is the largest integer less than 100,000 not expressible as a sum of squares (We did not have to go very far.)

(d) A minor modification to the above code will give us what we want; simply change the for loop to »for n=1001:1 : 99999 (and the wording in the fprintf statement).

We then find the integer to be 1001.

(e) The following code will determine what we are looking for:

```
>> for n=2:99999, flag=sum2sqb(n); if flag==0, count=count+1; end,
end
>> count
->count =75972
```

**Note:** Part (e) took only a few seconds. If the programs were written less efficiently, for example, if we had run a nested loop by letting a and b run separately between all integers from 0 to  $\sqrt{n}$  (or larger), some parts of this problem (notably, part (e)) could not be done in a reasonable amount of computer time.

**EFR 4.3:** (a) Before you run the indicated computations in a MATLAB session, try to figure out the output by hand. This will assure that you understand both the Collatz sequence generation process as well as the program. The reason for clearing the vector a at the end of the script is so that on subsequent runs, this vector will not start with old values from previous runs.

(b) The M-file is boxed below:

```
function n = collctr(an)
n=0;
while an ~= 1
    if ceil(an/2)==an/2 %tests if an is even
        an=an/2;
    else
        an=3*an+1;
    end
    n=n+1;
end
```

**EFR 4.4:** (a) The M-file is boxed below:

```
%raffledraw.m
%scriptfile for EFR 4.4
K = input('Enter number of players: ') ;
N=zeros(K,26); %this allows up to 26 characters for each players
%name.
n=input('Enter IN SINGLE QUOTES first player name: ') ;
len(l)=length(n);
N(1,1:len(l))=n;
W(1)=input('Enter weight of first player: ') ;
for i=2:K-1
    n=input('Enter IN SINGLE QUOTES next player name: ') ;
    len(i)=length(n);
    N(i,1:len(i))=n;
    W(i)=input('Enter weight of this player: ') ;
end
u=input('Enter IN SINGLE QUOTES last player name: ') \;
len(K)=length(n) ;
N(K, 1:len(K))=n;
W(K)=input('Enter weight of last player: ') ;
totW = sum(W); %total weight of all players (= # of raffle tickets)
%the next four commands are optional, they only add suspense and
%drama to the raffle drawing which the computer can do in lightning
%time
fprintf('\r \r RANDOM SELECTION PROCESS INITIATED \r \r ...')
pause %creates a 1 second pause
fprintf('C\r \r ...SHUFFLING \r \r')
pause(5) %creates a 5 second pause
%%%%%%%%%%%%%%
rand('state',sum(100*clock))
magic = floor(totW*rand); %this will be a random number between 0 and
%totW
count =W(1); %number of raffle tickets of player 1
if magic<=count
    fprintf('WINNER IS %s \r \r \ char(N(1,1:len(1)))')
    fprintf('CONGRATULATIONS %s!!!!!!!!!!!!!!', char(N(1,1:len (1))))
```

```

|   return
| else count = count + W(2); k=2;
|   while 1
|     if magic <=count
|       fprintf ('WINNER IS %s \r \r    % char (N (k, 1: len (k) )) )
|       fprintf('CONGRATULATIONS %s!!!!!!!!!!!!!!', char(N(k, 1:len(k))))
|       return
|     end
|     k=k+1; count = count +W(k);
|   end
| end

```

(b) We now perform the indicated program runs:

```

» raffledraw  Enter number of players: 4
Enter IN SINGLE QUOTES first player name: 'Alfredo '
Enter weight of first player: 4
Enter IN SINGLE QUOTES next player name: ' Denise '
Enter weight of this player: 2
Enter IN SINGLE QUOTES next player name: ' Sylvester '
Enter weight of this player: 2
Enter IN SINGLE QUOTES last player name: ' Laurie '
Enter weight of last player: 4

```

RANDOM SELECTION PROCESS INITIATED

```

... ..SHUFFLING....
->WINNER IS Laurie
->CONGRATULATIONS Laurie!!!!!!!!!!!!!!

```

On a second run the winner was Denise. If written correctly, and if this same raffledra w is run many times, it should turn out (from basic probability) that Alfredo and Laurie will each win roughly 4/12 or 33 1/3% of the time while Denise and Sylvester will win roughly 2/12 or 16 2/3% of the time.

---

## CHAPTER 5: FLOATING POINT ARITHMETIC AND ERROR ANALYSIS

**EFR 5.1:** For shorthand we write: FPA to mean "the floating point answer," EA to mean "the exact answer," E to mean the "error" = |FAP-EA|, and RE to mean "the relative error" = E/|EA|.

- (a) FPA = 0.023, EA = 0.0225, E = 0.0005, RE = 0.02222...  
 (b) FPA =  $370,000 \times .45 = 170,000$ , EA = 164990.2536, E = 5009.7464, RE = 0.030363...  
 (c) FPA =  $8000 \div 120 = 67$ , EA = 65.04878..., E = 1.9512195121..., RE = 0.029996...

**EFR 5.2:** (a) As in the solution of Example 5.3, since the terms are decreasing, we continue to compute partial sums (in 2-digit rounded floating point arithmetic) until the terms get sufficiently small so as to no longer have any effect on the accumulated sum.

$$\begin{aligned}
 S_1 &= 1, S_2 = S_1 + 1/2 = 1.5, S_3 = S_2 + 1/3 = 1.5 + .33 = 1.8, S_4 = S_3 + 1/4 = 1.8 + .25 = 2.1, \\
 S_5 &= S_4 + 1/5 = 2.1 + .2 = 2.3, S_6 = S_5 + 1/6 = 2.3 + .17 = 2.5, S_7 = S_6 + 1/7 = 2.5 + .14 = 2.6, \\
 S_8 &= S_7 + 1/8 = 2.6 + .13 = 2.7, S_9 = S_8 + 1/9 = 2.7 + .11 = 2.8, S_{10} = S_9 + 1/10 = 2.8 + .1 = 2.9.
 \end{aligned}$$

This pattern continues until we reach  $S_{20}$ : In each such partial sum,  $1/k$  contributes 0.1 to the cumulative sum. As soon as we reach  $S_{21}$ , the terms ( $1/21 = 0.048$ ) in floating point arithmetic become too small to have any effect on the cumulative sum so we have converged; thus the final answer is:  $2.9 + 10 \times .1 = 3.9$ .

(b)(i)  $x^2 = 100$ : Working in exact arithmetic, there are, of course, two solutions:  $x = \pm 10$ . These are also floating point solutions and any other floating point solutions will lie in some intervals about these two. Let's start with the floating point solution  $x = 10$ . In arithmetic of this problem, the next floating point number greater than 10 is 11 and (in floating point arithmetic)  $11^2 = 120$ , so there are no floating point solutions greater than 10. Similarly the floating point number immediately preceding 10 is 9.9 and (in floating point arithmetic)  $9.9^2 = 98$ , so there are no (positive) floating point solutions less than 10. Similarly, -10 is the only negative floating point solution. Thus there are exactly two floating point solutions (or more imprecisely: between 2 and 10 solutions).

(ii)  $8x^2 = x^5$ : In exact arithmetic, we would factor this  $x^5 - 8x^2 = x^2(x^3 - 8) = 0$  to get the real solutions:  $x = 0$  and  $x = 2$ . Because of underflow, near  $x = 0$ , we can get many (more than 10) floating point solutions. Indeed, since  $e=8$ , if  $|x| < 10^{-5}$ , then both sides of the equation will underflow to zero so we will have a solution. Any number of form  $\pm a \times 10^{-c}$ , where  $a$  and  $b$  are any digits ( $a \neq 0$ ) and  $c = 6, 7$ , or  $8$ , will thus be a floating point solution, so certainly there are more than 10 solutions. (How many are there exactly?)

**EFR 5.3:** (a) As in the solution to Example 5.4, we may assume that  $x \neq 0$  and write  $x = d_1 d_2 \cdots d_s d_{s+1} \cdots \times 10^e$ . Now, since we are using  $s$ -digit rounded arithmetic,  $fl(x)$  is the closer of the two numbers  $d_1 d_2 \cdots d_s \times 10^e$  and  $d_1 d_2 \cdots d_s \times 10^e + 10^{-s} \times 10^e$ . Since the gap between these two numbers has length  $10^{-s} \times 10^e$ , we may conclude that  $|x - fl(x)| \leq \frac{1}{2} \cdot 10^{-s} \times 10^e$ . On the other hand,  $|x| \geq .100 \cdots 0 \times 10^e = 10^{e-1}$ . Putting these two estimates together, we obtain the following estimate for the relative error:  $\left| \frac{x - fl(x)}{x} \right| \leq \frac{\frac{1}{2} \cdot 10^{-s} \times 10^e}{10^{e-1}} = \frac{1}{2} \cdot 10^{1-s}$ . Since equality is possible, we conclude that  $u = \frac{1}{2} \cdot 10^{1-s}$ , as asserted. The floating point numbers are the same whether we are using chopped or rounded arithmetic, so the gap from 1 to the next floating point number is still  $10^{1-s}$ , as explained in the solution of Example 5.4.

(b) If  $x = 0$ , we can put  $\delta = 0$ ; otherwise put  $\delta = [fl(x) - x]/x$ .

**EFR 5.4:** (a) Since  $N-i \leq N$  when  $i$  is nonnegative, we obtain from (6) that

$$\begin{aligned} |fl(S_N) - S_N| &\leq u[(N-1)a_1 + (N-1)a_2 + (N-2)a_3 + \cdots + 2a_{N-1} + a_N] \\ &\leq u[Na_1 + Na_2 + Na_3 + \cdots + Na_{N-1} + Na_N] = Nu \sum_{n=1}^N a_n. \end{aligned}$$

(b) Simply divide both sides of the inequality in (a) by  $\sum_{n=1}^N a_n$  obtain the inequality in (b).

**EFR 5.5:** From  $1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots = \frac{\pi}{4}$ , we can write  $\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \cdots = \sum_{n=0}^{\infty} (-1)^n a_n$ , where  $a_n = 4/(2n+1)$ . Letting  $S_N$  denote the partial sum  $\sum_{n=0}^N (-1)^n a_n$ , Leibniz's theorem tells us that  $\text{Error} = |\pi - S_N| \leq a_{N+1} = 4/(2N+3)$ . Since we want  $\text{Error} < 10^{-7}$ , we should take  $N$  large enough to satisfy  $4/(2N+3) < 10^{-7} \Rightarrow 2N+3 > 4 \cdot 10^7 \Rightarrow N > (4 \cdot 10^7 - 3)/2 = 19,999,998.5$ .

Letting  $N = 19,999,999$ , we get MATLAB to perform the summation of the corresponding terms in order of increasing magnitude:

```
>> format long
>> Sum=0; N=19999999;
>> for n=N:-1:0
Sum=Sum+(-1)^n*4/(2*n+1) ;
end
>> Sum
->Sum = 3.14159260358979 (approximation to pi)
>> abs(pi-Sum)
->ans = 4.999999969612645e-008 (exact error of approximation)
```

---

## CHAPTER 6: ROOTFINDING

**EFR 6.1:** The accuracy of the approximation  $x_7$  is actually better than what was guaranteed from (1). The actual accuracy is less than 0.001 (this can be shown by continuing with the bisection method to produce an approximation  $x_n$  with guaranteed accuracy less than 0.00001 (how large should  $n$  be?) and then estimating  $|x_7 - \text{root}| \leq |x_7 - x_n| + |x_n - \text{root}| \leq 9 \times 10^{-4} + 1 \times 10^{-5} < 0.001$ . So actually,  $|f(x_7)|$  is over 30 times as large as  $|x_7 - \text{root}|$ . This can be explained by estimating  $y'(\text{root}) \geq 30$  (do it graphically, for example). Thus, for small values of  $\Delta x \equiv x - \text{root}$ ,  $\Delta y/\Delta x$  gets larger than 30. This is why the  $y$ -variations turn out to be more than 30 times as large as the  $x$ -variations, when  $x$  gets close to the root.

**EFR 6.2:** (a) Since  $f(0) = 1 - 0 > 0$ ,  $f(\pi/2) = 0 - \pi/2 < 0$ , and  $f(x)$  is continuous, we know from the intermediate value theorem that  $f(x)$  has a root in  $[0, \pi/2]$ . Since  $f'(x) = \sin(x) - 1 < 0$  on  $(0, \pi/2)$ ,  $f(x)$  is strictly decreasing so it can have only one root on  $[0, \pi/2]$ . (b) It is easy to check that the first value of  $n$  for which  $\pi/(2 \cdot 2^n) (= (b-a)/2^n)$  is less than 0.01 is  $n = 8$ . Thus by (1), using  $x_0 = 0$ , it will be sufficient to run through  $n = 8$  iterations of the bisection method to arrive at an approximation  $x_8$  of the root that has the desired accuracy. We do this with the following MATLAB loop:

```
>> xn=0; an=0; bn=pi/2 ; n=0;
>> while e n<=8
xn=(an+bn)/2 ; n=n+1;
if f(x)==0, root = xn; return
elseif f(x)>0, an=xn; bn=bn;
else, an=an; bn=xn;
end
end
>> xn
->xn = 0.73937873976088
```

c) The following simple MATLAB loop will determine the smallest value of  $n$  for which  $\pi/(2 \cdot 2^n)$  will be less than  $10^{12}$  (by (1) this would be the smallest number of iterations in the bisection method for which we could be guaranteed the indicated accuracy). (This could certainly also be done using logs.)

```
>> while pi/2/2^n>=1e-12 , n=n+1; end
>> n
->n = 41
>> pi/2/2^41, pi/2/2^40 %we perform a check
```

->ans = 7.143154683921678e-013 (OK) 1.428630936784336e-012 (too big, so it checks!)

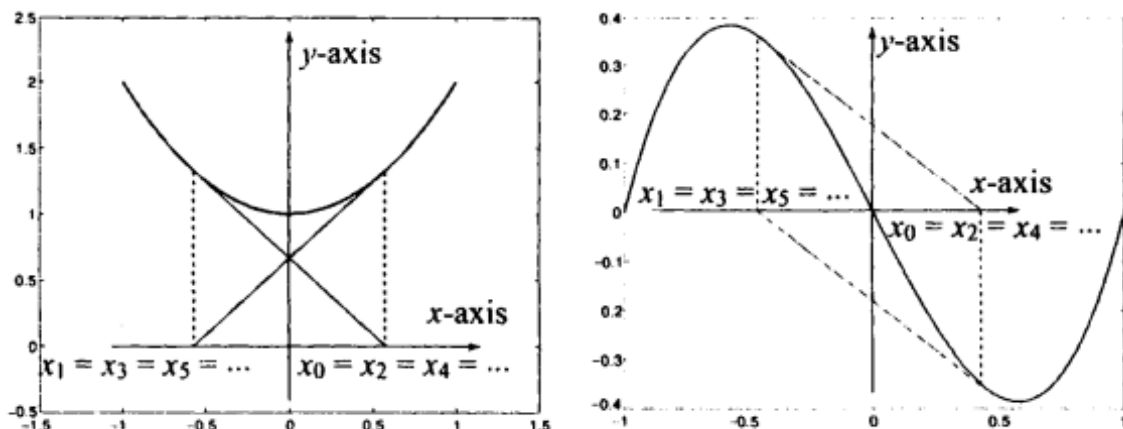
**EFR 6.3:** (a) The condition  $y_n \cdot y_a > 0$  mathematically translates to  $y_n$  and  $y_a$  having the same sign, so this is (mathematically) equivalent to our condition  $\text{sign}(y_n) == \text{sign}(y_a)$ .

(b) We are aiming for a root so at each iteration,  $y_n$  and  $y_a$  should be getting very small; thus their product  $y_n \cdot y_a$  will be getting smaller much faster (e.g., if both are about  $1e-175$ , then their product would be close to  $1e-350$  and this would underflow). Thus, with the modified loop we run the risk of a premature underflow destroying any further progress of the bisection method.

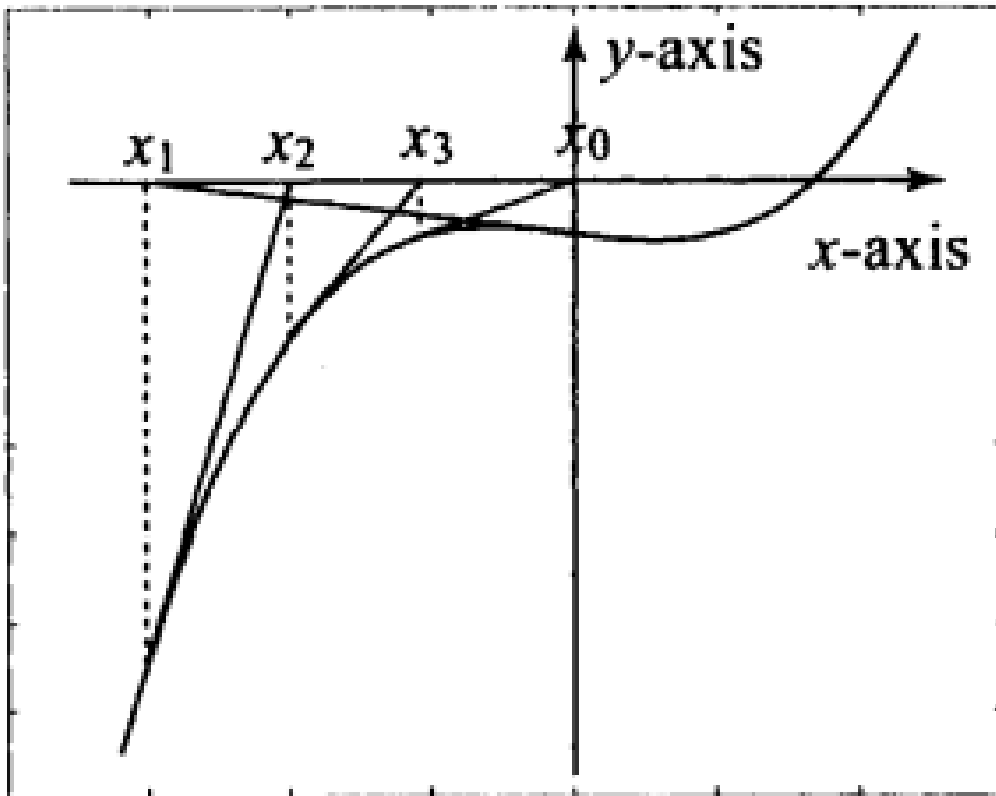
(c) Consider the function  $f(x) = (x + .015)^{101}$ , which certainly has a (unique) root  $x = -0.015$  and satisfies the requirements for using the bisection method. As soon as the interval containing  $x_n$  gets to within  $1e-2$  of the root, both  $y$ -values  $y_n$  and  $y_a$  would then be less than  $1e-200$ ; so their product would be less than  $1e-400$  and so would underflow to zero. This starts to occur already when  $n = 2$  ( $x_n = 0$ ), and causes the modified if-branch to default to the else-if option—taking the left half subinterval as the new interval. From this point on, all approximations will be less than  $-0.5$ , making it impossible to reach the 0.001 accuracy goal.

**EFR 6.4:** The distance from  $x$  to  $e$  is less than MATLAB's unit roundoff and the minimum gap between floating point numbers (see Example 5.4 and Exercise for the Reader 5.3). Thus MATLAB cannot distinguish between the two numbers  $x$  and  $e$ , and (in the notation of Chapter 5) we have  $\text{fl}(x) = \text{fl}(e) = e$  (since important numbers like  $e$  are built in to MATLAB as floating point numbers). As a result, when MATLAB evaluates  $\ln(x)$ , it really computes  $\ln(\text{fl}(x)) = \ln(e)$  and so gets zero.

**EFR 6.5:** (a) If we try to work with quadratic polynomials (parabolas), cycling cannot occur unless the parabola did not touch the  $x$ -axis (this is easy to convince oneself of with a picture and not hard to show rigorously). If we allow polynomials that do not have roots, then an example with a quadratic polynomial is possible, as shown in the left-hand figure below. For a specific example, we could take  $f(x) = x^2 + 1$ . For cycling as in the picture, we would want  $x_1 = x_0$ . Putting this into Newton's formula and solving (the resulting quadratic) gives  $x_0 = 1/\sqrt{3}$ . One can easily run a MATLAB program to see that this indeed produces the asserted cycling. To get an example of polynomial cycling with a polynomial that actually has a root we need to use at least a third-degree polynomial. Working with  $f(x) = x^3 - x = x(x-1)(x+1)$ , which has the three (equally spaced) roots  $x = 0, \pm 1$  the graph suggests that we can have a period-two cycling, so we put  $x_1 = x_0$  into Newton's formula. The resulting cubic equation is easily solved exactly (it factors) or with the Symbolic Toolbox (see Appendix A) or approximately using Newton's method. The solution  $x_0 = 1/\sqrt{5}$  produces the period-two cycling shown in the right-hand figure below, as can be checked by running Newton's method.



(b) On the right is an illustration of a period-four cycle in Newton's method. An explicit such example is furnished by  $f(x) = x^3 - x - 3$ . The calculations would be, of course, more elaborate than those of part (a); it turns out that  $x_0$  should be taken to be a bit less than zero. (More precisely, about  $-0.007446$ ; you may wish to run a couple of hundred iterations of Newton's method using this value for  $x_0$  to observe the cycling.) By contemplating the picture, it becomes clear that this function has cycles of any order. Just move JC0 closer to the right toward the location where  $f'(x)$  has a root.



**EFR 6.6 :** (a) The M-file is boxed below:

```
function [root, yval, niter] = secant(varfun, x0, x1, tol, nmax)
% input variables: varfun, x0, x1 tol, nmax
% output variables: root, yval, niter
% varfun = the string representing a mathematical function (built-in,
% M-file, or inline) , x0 and x1 = the two (different) initial
% approx.
% The program will perform the Secant method to approximate a root of
% varfun near x=x0 until either successive approximations differ by
% less than tol or nmax iterations have been completed, whichever
% comes first. If the tol and nmax variables are omitted default
% values of eps (approx. 10^
% (-16)) and 30 are used.
% We assign the default tolerance and maximum number of iterations if
% none are specified
if nargin < 4
tol=eps; nmax=50;
end

%we now initialize the iteration
xn=x0; xnnext=x1;

%finally we set up a loop to perform the approximations
for n=1:nmax
    yn=feval(varfun, xn); ynnext=feval(varfun, xnnext);
    if ynnext == 0
        fprintf('Exact root found\r')
        root = xnnext; yval = 0; niter=n;
        return
    end
    if yn == ynnext
        error('horizontal secant encountered, Secant method failed, try
changing x0, x1*)
    end
    newx=xnnext-feval(varfun, xnnext)*(xnnext-xn)/(feval(varfun,xnnext)-
feval(varfun, xn) );
    if abs(newx-xnnext)<tol
        fprintf('The secant method has converged\r')
        root = newx; yval = feval(varfun, root); niter=n;
        return
    elseif n==nmax
        fprintf('Maximum number of iterations reached\r')
        root = newx; yval = feval(varfun, root); niter=nmax
    end
end
```



```

|   return
|   end
|   xn=xnnext; xnnext=newx'
| end

```

(b) The syntax of this M-file is very close to that of newton:

```

>> f=inline('x^4-2') ; [r y n] = secant(f , 2,1.5 )
->The secant method has converged, r = 1.18920711500272,
y = -2.220446049250313e-016, n = 9
>> abs(r-2^(1/4)) ->ans = 0

```

In conclusion, the secant method took nine iterations and the approximate root (r) had residual which was essentially zero (in floating point arithmetic) and coincided with the exact answer  $\sqrt[4]{2}$  (in floating point arithmetic).

**EFR 6.7:** mean "the highest order of convergence,\* and AEC to mean "the asymptotic error constant." For each sequence, we determine these quantities if they exist:

(i) HOC = 1; AEC = 1 (linear convergence), (ii) HOC = 1, AEC = 1/2 (linear convergence), (iii) HOC = 3/2, AEC = 1, (iv) HOC = 2, AEC = 1 (quadratic convergence), (v) HOC does not exist. There is hyperconvergence for every order  $\alpha < 2$ , but the sequence does not have quadratic convergence

(b) The sequence  $e_n - e^{-3^n}$  has HOC = 3. In general,  $e_n = e^{-k^n}$  has HOC = k whenever it is a positive number.

**EFR 6.8:** Write  $f(x) = (x-r)^M h(x)$ , where M is the order of the root (and so  $h(r) \neq 0$ ). Differentiating, we see that the function  $F(x) = f(x)/f'(x)$  can be written as  $F(x) = (x-r)H(x)$ , where  $H(x) = h(x)/[Mh(x) + (x-r)h'(x)]$ . Since  $H(r) = 1/M \neq 0$ , we see that  $x = r$  is a simple root of  $F(x)$ . Since  $F'(x) \equiv [(f'(x))^2 - f(x)f''(x)]/(f'(x))^2$ , this method requires computing both  $f'(x)$  and  $f''(x)$ . The roundoff errors can also get quite serious. For example, if we are converging to a simple root, then in the iterative computations of  $F'(x_n) \equiv [(f'(x_n))^2 - f(x_n)f''(x_n)]/(f'(x_n))^2$ ,  $(f'(x_n))^2$  will be converging to a positive number, while  $f(x_n)f''(x_n)$  will be converging to zero. Thus, when these two numbers are subtracted roundoff errors can be insidious. With higher-order roots each of  $(f'(x_n))^2$  and  $f(x_n)f''(x_n)$  will be getting small very fast and can underflow to zero causing Newton's method to stop. If the root is a multiple root and the order is known not to be too high then this method performs reasonably well. If the order is known, however, the newtonmr method is a better choice.

## CHAPTER 7: MATRICES AND LINEAR SYSTEMS

**EFR 7.1:** Abbreviate the matrices in (1) by  $DE = P$  and write  $P = [p_{ij}]$ . Now, by definition,  $p_{ij} = (\text{ith row of } D) \cdot (\text{jth column of } E) = d_i \cdot e_{ij}$  (by diagonal form of  $D$ ). But by the diagonal form of  $E$ ,  $e_{ij}$  (and hence also  $p_{ij}$ ) is zero unless  $i = j$ , in which case  $e_{ij} = e_i$ . Thus  $p_{ij} = d_i \cdot e_i$  if  $i = j$ ; 0, if  $i \neq j$  and this is a restatement of (1)

```

function A=randint(n,m,k)
%generates an n by m matrix whose entries are random integers whose
%absolute values do not exceed k
A=zeros(n,m);
for i=1:n
    for j=1:m
        x=(2*k+1)*rand-k; %produces a random real number in (-k,k+1)
        A(i,j)=floor(x);
    end
end

```

(b) In the random experiments below, we print out the matrices only in the first trial

```
>> A=randint(6,6,9) ; B=randint(6,6,9) ; det(A*B), det(A*B)-det(A)*det(B)
```

```

→ A =      9   -5    2    0    7    5   → B =      7    0   -6    3    6   -9
      -1   -9    6   -1    2    6          3   -2    6    0    4   -1
          8    5   -6   -2    8    8          -4   -6   -6    3   -4    1
      -2    7   -8   -3    6   -9          -7    4   -2    7    7    2
      -7   -6   -6    2   -4   -6          0    8    6    3    6    3
      -9    5   -1    8   -1   -2          -3   -4   -3    1    4   -4

```

```
->det(A*B)=      -1.9436e+010      ->ans=      0
```

```
>> A=randint(6,6,9) ; B=randint(6,6,9) ; det(A*B), det(A*B)-
det(A)*det(B)
->ans = 6.8755e+009, 0
```

```
>> A=randint(6,6,9) ; B=randint(6,6,9) ; det(A*B), det(A*B)-
det(A)*det(B)
->ans = 8.6378e+010, 0
```

The last output 0 in each of the three experiments indicates that formula (4) checks.

```

(c) Here, because of their size, we do not print out any of the matrices.
>> A=randint(16,16,9) ; B=randint(16,16,9) ; det(A*B), det(A*B)-
det(A)*det(B)
->ans = -1.2268e+035, 18816e+021
>> A=randint(16,16,9) ; B=randint(16,16,9) ; det(A*B), det(A*B)-
det(A)*det(B)
->ans = 1.4841 e+035, -6.9913e+021
>> A=randint(16,16,9) ; B=randint(16,16,9) ; det(A*B), det(A*B)-
det(A)*det(B)
->ans = 3.3287e+035, ans = 7.0835e+021

```

The results in these three experiments are deceptive. In each, it appears that the left and right sides of (4) differ by something of magnitude 1021. This discrepancy is entirely due to roundoff errors! Indeed, in each trial, the value of the determinant of  $AB$  was on the order of 1035. Since MATLAB's (double precision IEEE) floating point arithmetic works with only about 15 significant digits, the much larger (3 5-digit) numbers appearing on the left and right sides of (4) have about the last 20 digits turned into unreliable "noise." This is why the discrepancies are so large (the extra digit lost came from roundoff errors in the internal computations of the determinants and the right side of (4)). Note that in part (b), the determinants of the smaller matrices in question had only about 10 significant digits, well within MATLAB's working precision.

**EFR 7.3:** Using the `fill` command as was done in the text to get the gray cat of Figure 7.3(b), you can get those other-colored cats by simply replacing the RGB vector for gray by the following: Orange  $\rightarrow \text{RGB} = [1 \ .5 \ 0]$ , Brown  $\rightarrow \text{RGB} = [.5 \ .25 \ 0]$ , Purple  $\rightarrow \text{RGB} = [5 \ 0 \ .5]$ . Since each of these colors can have varying shades, your answers may vary. Also, the naked eye may not be able to distinguish between colors arising from small perturbations of these vectors (say by .001 or even .005). The RGB vector representing MATLAB's cyan is  $\text{RGB} = [0 \ 1 \ 1]$ .

**EFR 7.4:** By property (10) (of linear transformations):  $L(\alpha P_1) = \alpha L(P_1)$ ; if we put  $\alpha = 0$ , we get that  $L(\vec{0}) = \vec{0}$  (where  $\vec{0}$  is the zero vector). But a shift transformation  $T_{V_0}(x,y) = (x,y) + V_0$  satisfies  $T_{V_0}(\vec{0}) = \vec{0} + V_0 = V_0$ . So the shift transformation  $T_{V_0}$  being linear would force  $V_0 = \vec{0}$ , which is not allowed in the definition of a shift transformation (since then  $T_{V_0}$  would then just be the identity transformation).

**EFR 7.5:** (a) As in the solution of Example 7.4, we individually multiply out the homogeneous coordinate transformation matrices (as per the instructions in the proof of Theorem 7.2) from right to left. The first transformation is the

shift with vector  $(1,0)$  with matrix:  $T_{(1,0)} \sim \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = H_1$ . After this we apply a scaling  $S$  whose matrix is given

by  $S \sim \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = H_2$ . The homogeneous coordinate matrix for the composition of these two transformations is:

$M = H_2 H_1 = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ . We assume (as in the text) that we have left in the graphics

window the first (white) cat of Figure 7.3(a) and that the CAT matrix  $A$  is still in our workspace. The following commands will now produce the new "fat CAT":

```

>> H1=[1 0 1;0 1 0; 0 0 1] ; H2=[2 0 0;0 1 0;0 0 1] ; M=H2*H1
>> AH=A; AH(3,:)=ones(1,10); %homogenize the CAT matrix
>> AH1=M*AH; % homogenized "fat CAT" matrix
>> hold on
>> plot(AH1(1,:), AH1(2,:), 'r')
>> axis([-2 10 -3 6]) % set wider axes to accommodate "fat CAT"
>> axis('equal')

```

The resulting plot is shown in the left-hand figure that follows. (b) Each of the four cats needs to first get rotated by its specified angle about the same point  $(1.5,1.5)$ . As in the solution to Example 7.4, these rotations can be accomplished by first shifting this point to  $(0,0)$  with the shift  $T_{(1.5,1.5)}$ , then performing the rotation, and finally shifting back with the inverse shift  $T_{(1.5,1.5)}$ . In homogeneous coordinates, the matrix representing this composition is (just like in the solution to Example 7.4):

$$M = \begin{bmatrix} 1 & 0 & 1.5 \\ 0 & 1 & 1.5 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -1.5 \\ 0 & 1 & -1.5 \\ 0 & 0 & 1 \end{bmatrix}$$

After this rotation, each cat gets shifted in the specified direction with  $T_{(\pm 1, \pm 1)}$ . For the colors of our cats let's use the following: black ( $\text{rgb} = [0 \ 0 \ 0]$ ), light gray ( $\text{rgb} = [.7 \ .7 \ .7]$ ), dark gray ( $\text{rgb} = [.3 \ .3 \ .3]$ ), and brown ( $\text{rgb} = [.5 \ .25 \ 0]$ ). The following commands will then plot those cats:

```

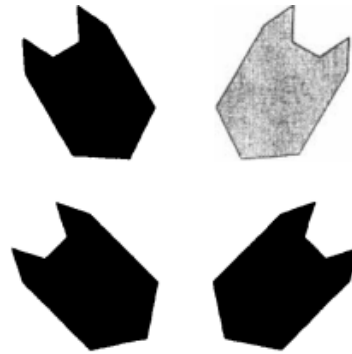
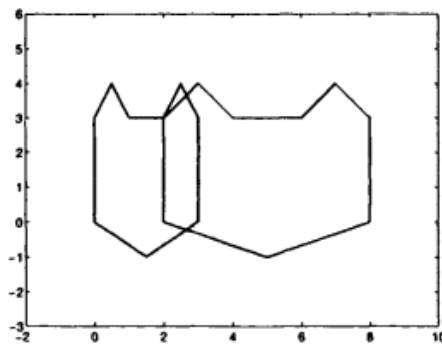
>> clf, hold on %prepare graphic window
>> %upper left cat, theta = pi/6 (30 deg), shift vector = (-3, 3)
>> c = cos(pi/6); s = sin(pi/6);

```

```

>> M=[1 0 1.5;0 1 1.5;0 0 1]*[c -s 0;s c 0;0 0 1]*[1 0 -1.5;0 1 -
1.5;0 0 1];
>> AUL=[1 0 -3;0 1 3;0 0 1]*M*AH;
>> fill(AUL(1,:), AUL(2,:), [0 0 0])
>> %upper right cat, theta = -pi/6 (-30 deg), shift vector = (3, 1)
>> c = cos(-pi/6); s = sin(-pi/6);
>> M=[1 0 1.5;0 1 1.5;0 0 1]*[c -s 0;s c 0;0 0 1]*[1 0 -1.5;0 1 -
1.5;0 0 1];
>> AUR=[1 0 1;0 1 1;0 0 1]*M*AH;
>> fill(AUR(1,:), AUR(2,:), [.7 .7 .7])
>> %lower left cat, theta = pi/4 (45 deg), shift vector = (-3, -3)
>> c = cos(pi/4); s = sin(pi/4);
>> M=[1 0 1.5;0 1 1.5;0 0 1]*[c -s 0;s c 0;0 0 1]*[1 0 -1.5;0 1 -
1.5;0 0 1];
>> ALL=[1 0 -3;0 1 -3;0 0 1]*M*AH;
>> fill(ALL(1,:), ALL(2,:), [.3 .3 .3])
>> %lower right cat, theta = -pi/4 (-45 deg), shift vector = (3, -3)
>> c = cos(-pi/4); s = sin(-pi/4);
>> M=[1 0 1.5;0 1 1.5;0 0 1]*[c -s 0;s c 0;0 0 1]*[1 0 -1.5;0 1 -
1.5;0 0 1];
>> ALR=[1 0 3;0 1 -3;0 0 1]*M*AH;
>> fill(ALR(1,:), ALR(2,:), [.5 .25 0])
>> axis('equal'), axis off %see graphic w/out distraction of axes

```



**EFR 7.6:** (a) This first M-file is quite straightforward and is boxed below.

```

function B=mkhom(A)
B=A;
[n m]=size(A);
B(3,:)=ones(1,m);

```

(b) This M-file is boxed below.

```

function Rh=rot(Ah,x0,y0,theta)
%viz. EFR 7.6; theta should be in radians
%inputs a 3 by n matrix of homogeneous vertex coordinates, xy
%coordinates of a point and an angle theta. Output is corresponding
%matrix of vertices rotated by angle theta about (x0,y0).

%first construct homogeneous coordinate matrix for shifting (x0,y0)
to (0,0)
S2=[1 0 -x0;0 1 -y0;0 0 1];
%next the rotation matrix at (0,0)
R=[cos(theta) -sin(theta) 0; sin(theta) cos(theta) 0;0 0 1];
%finally the shift back to (x0,y0)
SB=[1 0 x0;0 1 y0;0 0 1];
%now we can obtain the desired rotated vertices:
Rh=SB*R*S2*Ah;

```

**EFR 7.7:** (a) The main transformation that we need in this movie is vertical scaling. To help make the code for this exercise more modular, we first create, as in part (b) of the last EFR, a separate M-file for vertical scaling:

```

function Rh=vertscales(Ah,b,y0)
%inputs a 3 by n matrix of homogeneous vertex coordinates, a (pos.)
%numbers a for y- scales, and an optional arguments y0

%for center of scaling. Output is homogeneous coord. matrix of scaled
%vertices. default value of y0 is 0.

if nargin < 3
    y0=0;
end
%first construct homogeneous coordinate matrix for shifting y=y0 to
%y=0

```

```
SZ=[1 0 0;0 1 -y0; 0 0 1];
%next the scaling matrix at (0,0)
S=[1 0 0; 0 b 0;0 0 1];
%finally the shift back to y=0
SB=[1 0 x0;0 1 y0;0 0 1];
%now we can obtain the desired scaled vertices
Rh=SB*S*SZ*Ah;
```

Making use of the above M-file, the following script recreates the CAT movie of Example 7.4 using homogeneous coordinates:

```
%script for EFR 7.6(a): catmovieNol.m cat movie creation
%Basic CAT movie, where cat closes and reopens its eyes.
elf, counter=1;
A=[0 0 .5 1 2 2.5 3 3 1.5 0; ...
  0 3 4 3 3 4 3 0- 1 0]; %Basic CAT matrix
Ah = mkhom(A); %use the M-file from EFR 7.6
t=0:.02:2*pi; %creates time vector for parametric equations for eyes
xL=1+.4*cos(t); y=2+.4*sin(t); %creates circle for left eye
LE=mkhom([xL; y]); %homogeneous coordinates for left eye
xR=2+.4*cos(t); y=2+.4*sin(t); %creates circle for right eye
RE=mkhom([xR; y]); %homogeneous coordinates for right eye
xL=1+.15*cos(t); y=2+.15*sin(t); %creates circle for left pupil
LP=mkhom([xL; y]); %homogeneous coordinates for left pupil
xR=2+.15*cos(t); y=2+.15*sin(t); %creates circle for right pupil
RP=mkhom([xR; y]); %homogeneous coordinates for right pupil
for s=0:.2:2*pi
    factor = (cos(s)+1)/2;
    plot(A(1,:), A(2,:), 'k'), hold on
    axis([-2 5 -3 6]), axis('equal')
    LEtemp=vertscale(LE,factor,2); LPtemp=vertscale(LP,factor,2);
    REtemp=vertscale(RE,factor,2); RPtemp=vertscale(RP,factor,2);
    hold on
    filKLEtempd, :) , LEtemp(2, :) , 'y') , fill(REtemp(1, :) ,
    REtemp(2,:), 'y')
    filKLPTempd, :) , LPtemp(2, :) , 'k') , fill(RPtemp(1, :) ,
    RPtemp(2,:), 'k')
    M(:, counter) = getframe;
    hold off
    counter=counter+1;
end
```

(b) As in part (a), the following script M-file will make use of two supplementary M-files,  $AhR=reflx(Ah, x0)$  and  $AhS=shift(Ah, x0, y0)$ , that perform horizontal reflections and shifts in homogeneous coordinates, respectively. The syntaxes of these M-files are explained in Exercises 5 and 6 of this section. Their codes can be written in a fashion similar to the code `vertscale` but for completeness can be downloaded from the ftp site for this text (see the beginning of this appendix). They can be avoided by simply performing the homogeneous coordinate transformations directly, but at a cost of increasing the size of the M-file that we give:

```
%coolcatmovie.m: script for making coolcat movie matrix M of EFR 7.7
%act one: eyes shifting left/right
t=0:.02:2*pi; counter=1;
A=[0 0 .5 1 2 2.5 3 3 1.5 0; ...
  0 3 4 3 3 4 3 0- 1 0];
x=1+.4*cos(t); y=2+.4*sin(t); xp=1+.15*cos(t); yp=2+.15*sin(t);
LE=[x;y]; LEh=mkhom(LE); LP=[xp;yp]; LPh=mkhom(LP);
REh=reflx(LEh/ 1.5); RPh=reflx(LPh, 1.5);
LW=[.3 -1; .2 -.8]; LW2=[.25 -1.1; .25 -.6]; %left whiskers
LWh=mkhom(LW); LW2h=mkhom(LW2);
RWh=reflx(LWh, 1.5); RW2h=reflx(LW2h, 1.5); %reflect left whiskers
%to get right ones
M=[1 1.5 2; .25 -.25 .25]; Mh=mkhom(M); %matrix & homogenization of
%cats mouth
Mhrefl=refly(Mh, -.25); %homogeneous coordinates for frown
for n=0:(2*pi)/20:2*pi
    plot(A(1,:), A(2,:), 'k')
    axis([-2 5 -3 6]), axis('equal')
    hold on
    plot(LW(1,:), LW(2,:), 'k'), plot(LW2(1,:), LW2(2,:), 'k')
    plot(RWh(1,:), RWh(2,:), 'k')
    plot(RW2h(1,:), RW2h(2,:), 'k')
    plot(Mhrefl(1,:), Mhrefl(2,:), 'k')
    fill(UEU, :) , LE(2, :) , 'y') , fill(REh(1, :) , REh(2, :) , 'y')
    LPshft=shift(LPh, -.25*sin(n), 0); RPshft=shift(RPh, -.25*sin(n), 0);
    fill(LPshft(1, :) , LPshft(2, :) , 'k') , fill(RPshft(1, :) ,
    RPshft(2, :) , 'k')
    Mov(:, counter)=getframe;
    hold off
    counter = counter +1;
end
```

```

%act two: eyes shifting up/down
for n=0:(2*pi)/20:2*pi
plot(A(1,:), A(2,:), 'k')
axis([-2 5 -3 6]), axis('equal')
hold on
plot(LW(1,:), LW(2,:), 'k'), plot(LW2(1,:), LW2(2,:), 'k')
plot(RWh(1,:), RWh(2,:), 'k')
plot(RW2h(1,:), RW2h(2,:), 'k')
plot(Mhrefl(1,:), Mhrefl(2,:), 'k')
fill(LE(1,:), LE(2f:), 'y'), fill(REh(1f:), REh(2,:), 'y')
LPshft=shift(LPh, 0, .25*sin(n)); RPshft=shift(RPh, 0, .25*sin(n));
fill(LPshft(1,:), LPshft(2,:), 'k'), fill(RPshft(1,:),
RPshft(2,:), 'k')
Mov(:, counter)=getframe;
hold off
counter = counter +1;
end

%act three: whisker rotating up/down then smiling
for n=0:(2*pi)/10:2*pi
plot(A(1,:), A(2,:), 'k')
axis([-2 5 -3 6]), axis('equal')
hold on
fill(LE(1,:), LE(2,:), 'y'), fill(LP(1,:), LP(2f:), 'k')
fill(REh(1,:), REh(2,:), 'y'), fill(RPh(1,:), RPh(2,:), 'k')
LWrot=rot(LWh, .3, .2, -pi/6*sin(n)); LW2rot=rot(LW2h, .25, .25, -
pi/6*sin(n));
RWrot=reflx(LWrot, 1.5); RW2rot=reflx(LW2rot, 1.5);
plot(LWrot(1,:), LWrot(2,:), 'k'), plot(LW2rot(1,:), LW2rot(2,:), 'k')
plot(RWrot(1,:), RWrot(2,:), 'k'), plot(RW2rot(1f:), RW2rot(2,:), 'k')
if n == 2*pi
    plot(Mh(1,:), Mh(2,:), 'k')
    for n=1:10, L(:,n)=getframe; end
    Mov(:, counter: (counter+9))=L;
    break
else
    plot(Mhrefl(1,:), Mhrefl(2,:), 'k')
end
Mov(:, counter)=getframe;
hold off
counter = counter +1;
end
%THE END

```

**EFR 7.8:** (a) Certainly the zeroth generation consists of  $1 = 3^0$  triangles. Since the sidelength is one, and the triangle has each of its angles being  $\pi/3$ , its altitude must be  $\sin(\pi/3) = \sqrt{3}/2$ . Thus, the area of the single zeroth generation triangle is  $\sqrt{3}/4$ . Now, each time we pass to a new generation, each triangle splits into three (equilateral) triangles of half the length of the triangles of the current generation. Thus, by induction, the  $n$ th generation will have  $3^n$  equilateral triangles of sidelength  $1/2^n$  and hence each of these has area  $(1/2) \cdot 1/2^n \cdot [\sqrt{3}/2]/2^n = \sqrt{3}/4^{n+1}$ .

(b) From part (a), the  $n$ th generation of the Sierpinski carpet consists of  $3^n$  equilateral triangles each having area  $\sqrt{3}/4^{n+1}$ . Hence the total area of this  $m$ th generation is  $\sqrt{3}(3/4)^n/4$ . Since this expression goes to zero as  $n \rightarrow \infty$ , and since the Sierpinski carpet is contained in each of the generation sets, it follows that the area of the Sierpinski carpet must be zero.

**EFR 7.9:** (a) The  $2 \times 2$  matrices representing dilations:  $\begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix}$  ( $s > 0$ ), and reflections with respect to the  $x$ -axis:

$\begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$  or  $y$ -axis:  $\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$  are both diagonal matrices and thus commute with any other  $2 \times 2$  matrices; i.e., if

$D$  is any diagonal matrix and  $A$  is any other  $2 \times 2$  matrix, then  $AD = DA$ . In particular, these matrices commute with

each other and with the matrix representing a rotation through the angle  $\theta$ :  $\begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}$ . By composing rotations

and reflections, we can obtain transformations that will reflect about any line passing through  $(0,0)$ . Once we throw in translations, we can reflect about any line in the plane and (as we have already seen) rotate with any angle about any point in the plane. By the definition of similitudes, we now see that compositions of these general transformations can produce the most general similitudes. Translating into homogeneous coordinates (using the proof of Theorem 7.2) we see that the

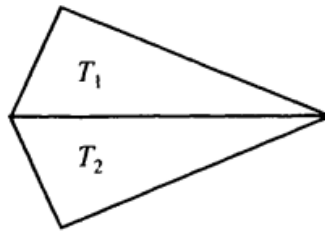
matrix for such a composition can be expressed as  $\begin{bmatrix} s \cos \theta & -s \sin \theta & x_0 \\ \pm s \sin \theta & \pm s \cos \theta & y_0 \\ 0 & 0 & 1 \end{bmatrix}$  where  $s$  now is allowed to be any nonzero

number. If the sign in the second row is negative, we have a reflection: If  $s > 0$ , it is a  $y$ -axis reflection; if  $s < 0$ , it is an  $x$ -axis reflection. (b) Let  $T_1$  and  $T_2$  be two similar triangles in the plane. Apply a dilation, if necessary, to  $T_1$  so that it has the same sidelengths as  $T_2$ . Next, apply a shift transformation to  $T_1$  so that a vertex gets shifted to a corresponding vertex of  $T_2$ , and then apply a rotation to  $T_1$  about this vertex so that a side of  $T_1$  transforms into a corresponding side of  $T_2$ .

At this point, either  $T_1$  and  $T_2$  are now the same triangle, or they are reflections of one another across the common side. A

final reflection about this line, if necessary, will thus complete the transformation of  $T_1$  into  $T_2$  by a similitude.

(c) It is clear that dilations, rotations, and shifts are essential. For an example to see why reflection is needed, simply take  $T_1$  to be any triangle with three different angles and  $T_2$  to be its reflection about one of the edges (see figure). It is clearly not possible to transform one of these two triangles into the other using any combination of dilations, rotations, and shifts.



**EFR 7.10:**(a) There will be only one generation; here are the outputs that were asked for (in format short):

```

A →      0      1.0000  2.0000  A1 →      0      0.5000  1.0000
          0      1.7321   0          0      0.8660   0
          1.0000  1.0000  1.0000      1.0000  1.0000  1.0000

A →      1.0000  1.5000  2.0000  A1 →      0.5000  1.0000  1.5000
          0      0.8660   0          0.8660  1.7321  0.8660
          1.0000  1.0000  1.0000      1.0000  1.0000  1.0000

A1([ 1  2 ],2) →      0.5000  A3([ 1  2 ],2) →      1.5000
                   0.8660                   0.8660

```

(b) Since the program calls on itself and does so more than once (as long as `niter` is greater than zero), placing a `hold off` anywhere in the program will cause graphics created on previous runs to be lost, so such a feature could not be incorporated into the program.

(c) Since we want the program to call on itself iteratively with different vertex sets, we really need to allow vertex sets to be inputted. Different vertex inputs are possible, but in order for the program to function effectively, they should be vertices of a triangle to which the similitudes in the program correspond, (e.g., any of the triangles in any generation of the Sierpinski gasket).

**EFR 7.11:** (a) S2,S1,S3,S2,S3,S2

b) We list the sequence of float points in nonhomogeneous coordinates and in format short : [0.5000 0.8660], [0.2500 0.4330], [1.1250 0.2165], [1.0625 0.9743], [1.5313 0.4871], [1.2656 1.1096].

(c) The program is designed to work for any triangle in the plane. The reader can check that the three similitudes are constructed in a way that uses midpoints of the triangle and the resulting diagram will look like that of Figure 7.15.

**EFR 7.12:** (a) As with `sgasket2`, the program `sgasket3` constructs future-generation triangles simply from the vertices and (computed) midpoints of the current-generation triangles. Thus, it can deal effectively with any triangle and produce Sierpinski-type fractal generations.

(b) For illustration purposes, the following trials were run on MATLAB's Version 5, so as to illustrate the flop count differences. The code is easily modified to work on newer versions of MATLAB by simply deleting the "flops" commands.

```

V1=[0 0]; V2=[1 sqrt(3)]; V3=[2 0]; '%vertices of an equilateral triangle
test=[1 3 6 8 10]

```

<pre> » for i=1:5 flops(0), tic, sgasket1(V1,V2,V3,test(i)), toe, flops end &gt; (ngen =1) elapsedjime = 0.0600, ans =191    (ngen =3) elapsedjime = 0.2500, ans =2243    (ngen =6) elapsedjime = 0.8510, ans =62264    (ngen =8) elapsedjime = 7.2310, ans =560900    (ngen =10) elapsed time = 65.4640, ans =5048624 </pre>	<pre> » for i=1:5 flops(0), tic, sgasket1(V1,V2,V3,test(i)), toe, flops end &gt; (ngen =1) elapsedjime = 0.1400, ans = 45    (ngen =3) elapsedjime = 0.1310, ans =369    (ngen =6) elapsedjime = 0.7210, ans =9846    (ngen =8) elapsedjime = 6.2990, ans =88578    (ngen =10) elapsedjime = 46.7260, ans =797166 </pre>
---	--

We remind the reader that the times will vary, depending on the machine being used and other processes being run. The above tests were run on a rather slow machine, so the resulting times are longer than typical.

**EFR 7.13:** The M-file is boxed below:

```
function []=snow(n)
S=[0 1 2 0;0 sqrt(3) 0 0];
index=1;
while index <=n
    len=length(S(1,:));
    for i = 1:(len-1)
        delta=S(:,i+1)-S(:,i);
        perp=[0 -1;1 0]*delta;
        T(:,4*(i-1)+D)=S(:,i);
        T(:,4*(i-1)+2)=S(:,i) + (1/3)*delta;
        T(:,4*(i-1)+3)=S(:,i) + (1/2)*delta-(1/3)*perp;
        T(:,4*(i-1)+4)=S(:,i) + (2/3)*delta;
        T(:,4*(i-1)+5)=S(:,i+1);
    end
    index=index+1;
    S=T;
end
plot (S(1,:),S(2,:)), axis('equal')
```

The outputs of snow (1), snow (2), and snow (6) are illustrated in Figures 7.17 and 7.18.

**EFR 7.14:** For any pair of nonparallel lines represented by a two-dimensional linear system:  $\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} e \\ f \end{bmatrix}$ , the coefficient matrix will have nonzero determinant  $\alpha = ad - bc$ . The lines are also represented by the equivalent system  $\begin{bmatrix} a/\alpha & b/\alpha \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} e/\alpha \\ f \end{bmatrix}$ , where now the coefficient matrix has determinant  $(a/\alpha)d - (b/\alpha)c = 1$ . This change simply amounts to dividing the first equation by  $\alpha$ .

**EFR 7.15:** (a) As in the solution of Example 7.7, the interpolation equations  $p(-2) = 4, p(1) = 3, p(2) = 5$ , and  $p(5) = -22$  (where  $p(x) = ax^3 + bx^2 + cx + d$ ) translate into the linear system:  $\begin{bmatrix} -8 & 4 & -2 & 1 \\ 1 & 1 & 1 & 1 \\ 8 & 4 & 2 & 1 \\ 125 & 25 & 5 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} 4 \\ 3 \\ 5 \\ -22 \end{bmatrix}$ . We solve this using left division, as in Method 1 of the solution of Example 7.7:

```
>> format long
>> A=[-8 4 -2 1;1 1 1 1;8 4 2 1;125 25 5 1]; b=[4 3 5 -22]';
>> X=A\b
->x= 0.47619047619048 (=a)
    1.05952380952381 (=b)
    2.15476190476190 (=c)
    0.26190476190476 (=d)
```

(b) As in part (a) and the solution of Example 7.7, we create the matrix A and vector b of the corresponding linear system:  $Ax = b$ . A loop will facilitate the construction of A:

```
>> xvals = -3:5; A = zeros(9) %initialize the 9 by 9 matrix A
>> for i =1:length(xvals)
A(i,:)=xvals(i). (8:-1:0);
end
>> b = [-14.5 -12 15.5 2 -22.5 -112 -224.5 318 3729.5]'
```

We next go through each of the three methods of solving the linear system that were introduced in the solution of Example 7.7. We are working on an older and slower computer with MATLAB Version 5, so we will have flop counts, but the times will be slower than typical. The code is easily modified to work on the new version of MATLAB by simply deleting the flop s commands. We show the output for x only for Method 1 (in format long) as the answers with the other two methods are essentially the same

**Method 1:**

» flops(0), tic, x=A/b, toe, flops	->x =	-0.00000000000000 0.00000000000000 0.50000000000000 -0.00000000000001 -6.00000000000000 -1.99999999999996 0.00000000000000 -17.00000000000003 2.00000000000000	-> elapsed_time = 0.1300 ->ans = 1125 (flops)
---------------------------------------	----------	--	--

**Method 2:**

» flops(0), tic, x=inv(A)*b, toe, flops	->elapsed_time = 0.3010 ->ans = 1935 (flops)
--	---

**Method 3:**

» Ab=A; Ab(:,10)=b; » flops(0), tic, rref(Ab), toe, flops	->elapsed_time = 3.3150 ->ans = 2175 (flops)
---	---

The size of this problem is small enough so that all three methods produce essentially the same vector  $x$ . The computation times and flop counts begin to demonstrate the relative efficiency of the three methods. Reading off the coefficients of the polynomial in order (from  $x$ ), we get (after taking into account machine precision and rounding):  $a = b = d = g = 0$ ,  $c = 1/2$ ,  $e = -6$ ,  $f = -2$ ,  $h = -17$ , and  $k = 2$ , so that the interpolating polynomial is given by  $p(x) = \frac{1}{2}x^6 - 6x^4 - 2x^3 - 17x + 2$ . It is readily checked that this function satisfies all of the interpolation requirements.

**EFR 7.16:** As in Example 7.8, for a fixed  $n$ , if we let  $x$  denote the exact solution, we then have  $b_n = H_n x = c(n)(1 \frac{1}{2} \frac{1}{3} \cdots \frac{1}{n-1} \frac{1}{n})'$ . In order for  $b_n$  to have all integer coordinates, we need to have  $c(n)$  be a multiple of each of the integers  $1, 2, 3, \dots, n$ . The smallest such  $c(n)$  is thus the least common multiple of these numbers. We can use MATLAB's `lcm(a, b)` to find the lcm of any set of integers with a loop. Here is how it would work to find  $c(n) = \text{lcm}(1, 2, \dots, n)$ :

```
» cn=1 %initialize
» for k=1:n, c(n)=lcm(cn, k), end
```

The remaining code for constructing the exact solution  $x$ , the numerical solution of Method 1, `x_meth1`, and the numerical solution of Method 2 `x_meth2` are just as in Example 7.9. The `flops` commands in these codes should be omitted if you are using Version 6 or later. Also, since these computations were run on an older machine, the elapsed times will be larger than what is typical (but their ratios should be reasonably consistent). The loop below will give us the data we need for both parts (a) and (b):

```
1>> for n=20:10:30
2|cn=1; %initialize
3|for k=1:nf c(n)=lcm(cn, k); end
4|x = zeros(n,1); x(1)=cn;
5|bn = hilb(n)*x;
6|flops(0), tic, x_meth1=hilb(n)\bn; toc, flops
7|flops(0), tic, x_meth2=inv(hilb(n))*bn; toc, flops
8|Pct_err_meth1=100*max(abs(x-x_meth1))/cn,
9|Pct_err_meth2=100*max(abs(x-x_meth2))/cn
10|end
```

Along with the expected output, we also got some warnings from MATLAB that the matrix is either singular or poorly conditioned (to be expected). The output is summarized in the following table:

	Computer Time: $n = 20 / n = 30$	Flop Count: $n = 20 / n = 30$	Percentage of Maximum Error: $n = 20 / n = 30$
Method 1:	0/0 seconds	10,339/27,481	0%/0%
Method 2:	0/0.01 seconds	20,312/63,509	512.5%/5400%

**Note:** The errors may vary depending on which version of MATLAB you are using. (c) The errors with Method 1 turn out to be undetectable as  $n$  runs well over 1000. The computation times become more of a problem than the errors. MATLAB's "left divide" is based on Gaussian elimination with partial pivoting. After we study this algorithm in the next section, the effectiveness of this algorithm on the problem at hand will become clear.

**EFR 7.17:**(a)&(b): The first two are in reduced row echelon form. The corresponding general solutions are as follows: (for  $M_1$ ):  $x_1 = 3, x_2 = 2$ ; (for  $M_2$ ):  $x_1 = 2s - 3t - 2, x_2 = s, x_3 = 5t + 1, x_4 = t$ , where  $s$  and  $t$  are any real numbers

»rref([ 1 3 2 0 3; 2 6 2 -8 4])	->ans	1 3 0 -8 1 0 0 1 4 1
---------------------------------	-------	-------------------------

(c) From the outputted reduced row echelon form, we obtain the following general solution of the first system:  $x_1 = 1 - 3s + 8f, x_2 = s, x_3 = 1 - 4t, x_4 = t$ , where  $s$  and  $t$  are any real numbers. Because of the arithmetic nature of the algorithm being used (as we will learn in the next section), it is often advantageous to work in `format rat` in cases where the linear system being solved is not too large and has integer or fraction coefficients. We do this for the second system:

**EFR 7.18:** (a) The algorithm for forward substitution:  $x_1 = b_1/a_{11}$ ,  $x_j = (b_j - \sum_{k=1}^{j-1} a_{jk}x_k)/a_{jj}$  (the first formula is redundant since the latter includes it as a special case) is easily translated into the following MATLAB code (cf. Program 7.4):



```
function x=fwdsubst(L,b)
%Solves the lower triangular system Lx=b by forward substitution
%Inputs: L = lower triangular matrix, b = column vector of same
%dimension
%Output: x = column vector (solution)
[n m]=size(L);
x(1)=b(1)/L(1,1);
```

<pre>for j=2: n x(j) = (b(j)-L(j,1:j-1)*x(1:j-1)')/L(j,j) ; end x=x';</pre>		
<pre>»L[1 2 3 4;0 2 3 4;0 0 3 4;0 0 0 4]'; » b=[4 3 2 1]'; » format rat » fwdsubst(L,b)</pre>	->ans	4 -5/2 -5/6 -5/12

**EFR 7.19** : The two M-files are boxed below:

```
function B=rowmult(A,i,c)
% Inputs: A = any matrix, i = any row index, c = any nonzero number
% Output: B = matrix resulting from A by replacing row i by this row
% multiplied by c.
[m,n]=size(A);
if i<1|i>m
error(*Invalid index')
end
B=A;
B(i, :)=c*A(i, :);

function B=rowcomb(A,i,j,c)
% Inputs: A = any matrix, i, j - row indices, c = a number
% Output: B = matrix resulting from A by adding to row j the number
% c times row i.
[m,n]=size(A);
if i<1|i>m|j<1|j>m
error('Invalid index')
end
if i=j
error('Invalid row operation')
end
B=A;
B(j, :)=c*A(i, :)+A(j, :);
```

**EFR 7.20:** If we use `gausslim` to solve the system of Example 7.13, we get the correct answer (with lightning speed) with a flop count of 104 (if you have access to Version 5). In the table below, we give the corresponding data for the linear systems of parts (a) and (b) of EFR 7.16 (compare with the table in the solution of that exercise):

Program	Computer Time:	Flop Count:	Percentage of Maximum Error:
	$n = 20/ n = 30$	$n = 20/ n = 30$	$n = 20/ n = 30$
7.6	0.03/0.06 seconds	9,906/31,201	0%/0%

We observe that the time is detectable, although it was not when we used MATLAB's "left divide". Similarly, if we solve the larger systems of part (c) of EFR 7.16, we still get 0% errors for large values of  $n$ , but the times needed for `gausselim` to do the job are much greater than they were for "left divide". MATLAB's "left divide" is perhaps its most important program. It is based on Gaussian elimination, but also relies on numerous other results and techniques from numerical linear algebra. A full description of "left divide" would be beyond the scope of this book; for the requisite mathematics, we refer to [GoVL-83].

**EFR 7.21:** Working just as in Example 7.14, but this time in rounded floating point arithmetic, the answers are as follows:  
(a)  $x_1 = 1, x_2 = .999$  and (b)  $x_1 = .001, x_2 = .999$ .

**EFR 7.22:** Looking at (28) we see that solving for  $x_j$  takes: 1 division +  $(n - j)$  multiplications +  $(n - j - 1)$  additions (if  $j < n$ ) + 1 subtraction (if  $j = n$ ). Summing from  $j = n$  to  $y = 1$ , we deduce that:

$$\text{Total multiplications/divisions} = \sum_{j=1}^n n - j + 1 = n^2 + n - n(n+1)/2 = (n^2 + n)/2,$$

Total additions/subtractions =  $\sum_{j=1}^{n-1} [n-j-1+1] = \sum_{j=1}^{n-1} [n-j] = \sum_{j=1}^{n-1} j = (n^2 - n)/2$ .

Adding gives the grand total of  $n^2$  flops, as asserted.

**EFR 7.23:** Here we let  $x = (x_1, x_2, \dots, x_n)$  denote any  $n$ -dimensional vector and  $\|x\|$  denote its max norm  $\|x\|_\infty = \max\{|x_1|, |x_2|, \dots, |x_n|\}$ . The first norm axiom (36A) is clear from the definition of the max norm. The second axiom (36B) is also immediate:  $\|cx\| = \max\{|cx_1|, |cx_2|, \dots, |cx_n|\} = |c| \max\{|x_1|, |x_2|, \dots, |x_n|\} = |c|\|x\|$ . Finally, the triangle inequality (36C) for the max norm readily follows from the ordinary triangle inequality for real numbers:

$$\begin{aligned}\|x+y\| &= \max\{|x_1+y_1|, |x_2+y_2|, \dots, |x_n+y_n|\} \\ &= \max\{|x_1|+|y_1|, |x_2|+|y_2|, \dots, |x_n|+|y_n|\} \leq \|x\| + \|y\|\end{aligned}$$

EFR 7.24: (a) We may assume that  $B \neq 0$ , since otherwise both sides of the inequality are zero. Using definition (38), we compute:

$$\begin{aligned}\|AB\| &= \max\left\{\frac{\|ABx\|}{\|x\|}, x \neq 0 \text{ (vector)}\right\} = \max\left\{\frac{\|A(Bx)\|}{\|x\|} \cdot \frac{\|Bx\|}{\|Bx\|}, Bx \neq 0 \text{ (vector)}\right\} \\ &= \max\left\{\frac{\|A(Bx)\|}{\|Bx\|} \cdot \frac{\|Bx\|}{\|x\|}, Bx \text{ (vector)}\right\} \leq \|A\|\|B\|\end{aligned}$$

(b) First note that for any vector  $x \neq 0$ , the vector  $y = Ax$  is also nonzero (since  $A$  is nonsingular), and  $A^{-1}y = x$ . Using this notation along with definition (38), we obtain:

$$\begin{aligned}\|A^{-1}\| &= \max\left\{\frac{\|A^{-1}y\|}{\|y\|}, y \neq 0 \text{ (vector)}\right\} = \left(\min\left\{\frac{\|y\|}{\|A^{-1}y\|}, y \neq 0 \text{ (vector)}\right\}\right)^{-1} \\ &= \min_{y=Ax} \left\{\frac{\|Ax\|}{\|x\|}, x \neq 0 \text{ (vector)}\right\}^{-1}.\end{aligned}$$

**EFR 7.25:** (a) We first store the matrix  $A$  with the following loop, and then ask MATLAB for its condition number:

```
>> norm(z , inf )->ans =8.7156e+004
At first glance, the accuracy looks quite decent. The warnings, however,
remove any guarantees that
Theorem 7.7 would otherwise allow us to have.
(d) >> z2=inv(A)*b; r2=b-A*z2; -> Warning: Matrix is close to singular or
badly
scaled. Results may be inaccurate. RCOND = 8.296438e-017.
>> errest2=cl*norm(r2 , inf)/norm(A, inf) ->errest2 = 2.3494
(e) As in Example 7.23, we solve the system symbolically and then get the
norms that we asked for:
>> S=sym(A); x=S\b ; x=double(x);
>> norm(x-z , inf) ->ans =3.0347e-005
>> norm(x-z2 , inf) ->ans =3.0347e-005
```

Thus, despite the warning we received, the numerical results are much more accurate than the estimates of Theorem 7.7 had indicated.

**EFR 7.26:** (a) Since  $\lambda I - A$  is a triangular matrix, Proposition 7.3 tells us that the determinant  $p_A(X) = \det(\lambda I - A)$  is simply the product of the diagonal entries:  $p_A(\lambda) = (\lambda - 2)^2(\lambda - 1)^2$ . Thus  $A$  has two eigenvalues:  $\lambda = 1, 2$ , each having algebraic multiplicity 2.

(b)» `[V, D] = eig([ 2 1 0 0; 0 2 0 0; 0 0 1 0; 0 0 0 1])`

->V=	1.0000 -1.0000 0 0	->D=	2 0 0 0
	0 0.0000 0 0		0 2 0 0
	0 0 1.0000 0		0 0 1 0
	0 0 0 1.0000		0 0 0 1

From the output of `eig`, we see that the eigenvalue  $\lambda = 1$  has two linearly independent eigenvectors:  $[0 \ 0 \ 1 \ 0]'$  and  $[0 \ 0 \ 0 \ 1]'$ , and so has geometric multiplicity 2, while the eigenvalue  $\lambda = 2$  has only one independent eigenvector  $[2 \ 0 \ 0 \ 0]'$ , and so has geometric multiplicity 1.

(c) From the way in which part (a) was done, we see that the eigenvalues of any triangular matrix are simply the diagonal entries (with repetitions indicating algebraic multiplicities).

**EFR 7.27:** (a) The M-file is boxed below:

```
function [x, k, cliff] = jacobi (A,b, x0, tol , kmax)
% performs the Jacob i iteration on the linear system Ax=b.
```

```
% Inputs: the coefficient matrix 'A' , the inhomogeneity (column)
% vector 'b' , the seed (column) vector 'x0' for the iteration
% process, the tolerance 'tol' which will cause the iteration to stop
% if the 2-norms of differences of successive iterates becomes
% smaller than 'tol' , and 'kmax' which is the maximum number of
% iterations to perform.
% Outputs: the final iterate 'x' , the number of iterations performed
% 'k' , and a vector 'diff' which records the 2-norms of successive
% differences of iterates.
% If any of the last three input variables are not specified , default
% values of x0= zero column vector , tol=1e-10 and kmax=100 are used .

%assign default input variables , as necessary
if nargin<3 , x0=zeros(size(b)) ; end
if nargin<4 , tol=1e-10 ; end
if nargin<5 , kmax=100; end
if min(abs(diag(A)))<eps
    error('Coefficient matrix has zero diagonal entries, iteration
cannot be performed.\r')
end

[n m]=size(A);
xold=x0 ;
k=1 ; diff=[] ;
while k<=kmax
    xnew=b;
    for i=1:n
        for j=1:n
            if j~=i
                xnew(i)=xnew(i)-A(i,j)*xold(j);
            end
        end
        xnew(i)=xnew(i)/A(i,i);
    end
    diff(k)=norm(xnew-xold,2);
    if diff(k)<tol
        fprintf('Jacobi iteration has converged in %d iterations.\r', k)
        x=xnew;
        return
    end
    k=k+1; xold=xnew;
end
fprintf('Jacobi iteration failed to converge.\r')
x=xnew;
```

```
(b)>> A=[3 1 -1;4 -10 1;2 1 5]; b=[-3 28 20]';
>> [x, k, diff] = jacobi(A,b,[0 0 0]',le-6) ;
-> Jacobi iteration has converged in 26 iterations.
>> norm(x-[1 -2 4]', 2)->ans = 3.9913e-007 (Error is in agreement with
Example 7.26.)
>> diff(2,6)->ans = 8.9241e-007 (Last successive difference is in
agreement with Example 7.26.)
>> [x, k, diff] = jacobi(A,b,[0 0 0]');
-> Jacobi iteration has converged in 41 iterations. (With default error
tolerance 1e-10)
```

**EFR 7.28:** (a) The M-file is boxed below:

```
function [x, k, diff] = sorit(A,b,omega, x0,tol,kmax)
% performs the SOR iteration on the linear system Ax=b.
% Inputs: the coefficient matrix 'A' , the inhomogeneity (column)
% vector 'b' , the relaxation parameter 'omega' , the seed (column)
% 'x0' for the iteration process, the tolerance 'tol' vector which
% will cause the iteration to stop if the 2-norms of successive
% iterates becomes smaller than 'tol', and 'kmax' which is the
% maximum number of iterations to perform.
% Outputs: the final iterate 'x' , the number of iterations performed
% 'k' , and a vector 'diff' which records the 2-norms of successive
% differences of iterates.
% If any of the last three input variables are not specified, default
% values of x0= zero column vector, tol=1e-10 and kmax=100 are used.

%assign default input variables, as necessary
if nargin<4, x0=zeros(size(b)); end
if nargin<5, tol=1e-10; end
if nargin<6, kmax=100; end

if min(abs(diag(A)))<eps
    error('Coefficient matrix has zero diagonal entries, iteration
cannot be performed.\r')
end

[n m]=size(A);
xold=x0;
k=1; diff=[];
```

```

while k<=kmax
    xnew~b;
    for i=1:n
        for j=1:n
            if j<i
                xnew(i)=xnew(i)-A(i,j)*xnew(j);
            elseif j>i
                xnew(i)=xnew(i)-A(i,j)*xold(j) ;
            end
        end
        xnew(i)=xnew(i)/A(i,i);
        xnew(i)=omega*xnew(i) + (1-omega)*xold(i);
    end
    diff(k)=norm(xnew-xold,2) ;
    if diff(k)<tol
        fprintf('SOR iteration has converged in %diterations\r,k)
        x=xnew;
        return
    end
    k=k+1; xold=xnew;
end
fprintf('*SOR iteration failed to converge.\r'
)
x=xnew;

```

(b) We set the relaxation parameter equal to 1 for SOR to reduce to Gauss-Seidel:

```

>> A=[3 1 -1/ 4 -10 1;2 1 5) ; b=[- 3 28 20]' ;
>>[x, k, diff ] = sorit(A,b,1 , [0 0 0] \le-6 )
-> SOR iteration has converged in 17 iterations
>> norm(x-[1 -2 4]',2)
->ans =1.4177e-007 (This agrees exactly with the error estimate of
    Example 7.27.)
>> (x, k, diff ] = sorit(A,b,.9,[ 0 0 0]',le-6) ;
-> SOR iteration has converged in 9 iterations

```

**EFR 7.29:** Below is the complete code needed to recreate Figure 7.41. After running this code, follow the instructions of the exercise to create the key.

```

>> jerr=1; n=1 ;
>> while jerr>=1e-6
x=jacobi(A,b,[0 0 0]',1e-7,n);
Jerr(n)=norm(x-[1 -2 4]', 2); jerr=Jerr(n); n=n + 1;
end
>> semilogy(1:n-1,Jerr,'bo-')
>> hold on

>> gserr=1; n=1;
>> while gserr>=1e-6
x=gaussseidel(A,b,[0 0 0]',1e-7,n);
GSerr(n)=norm(x-[1 -2 4]',2); gserr=GSerr(n); n=n+1;
end
>> semilogy(1:n-1,GSerr,'gp-')

>> sorerr=1; n=1;
>> while sorerr>=1e-6
x=sorit(A,b,0.9, [0 0 0]',1e-1,n);
SORerr(n)=norm(x-[1 -2 4]',2); sorerr=SORerr(n); n=n+1;
end
>> semilogy(1:n-1,SORerr, 'rx-')
>> xlabel('Number of iterations'), ylabel('Error')

```

**EFR 7.30:** (a) By writing out the matrix multiplication and observing repeated patterns we arrive at the following formula for the vector  $b \equiv Ax$  of size  $2500 \times 1$ . Introduce first the following two  $1 \times 50$  vectors  $b'$   $\bar{b}$ :

$$b' = [1 \ 4 \ -1 \ 4 \ -1 \ \dots \ 4 \ -1 \ 5],$$

$$\bar{b} = [0 \ 2 \ -2 \ 2 \ -2 \ \dots \ 2 \ -2 \ 3].$$

In terms of copies of these vectors, we can express  $b$  as the transpose of the following vector:

$$b = [b' \ \bar{b} \ \bar{b} \ \dots \ \bar{b} \ \bar{b} \ b'].$$

(b) We need first to store the matrix  $A$ . Because of its special form, this can be expeditiously accomplished using some loops and the `diag` command as follows:

```

>> x=ones(2500,1); x(2:2:2500,1)=2 ;
>> tic , A=4*eye(2500); toc
->elapsed_time =0.6090
>> vl=-1*ones (49,1); vl=[vl;0]; %seed vector for sub/super diagonals

```

```

function [x, k, diff] = sorsparsediad(diags, inds,b,omega,
x0, tol,kmax)
% performs the SOR iteration on the linear system Ax=b in cases where
% the n by n coefficient matrix A has entries only on a sparse set of
% diagonals.
% Inputs: The input variables are 'diags', an n by J matrix where
% eachcolumn consists of the entries of one of A's diagonals. The
% first column of diags is the main diagonal of A (even if all zeros),
% and 'inds', a 1 by n vector of the corresponding set of indices
% for the diagonals (index zero corresponds to the main diagonal).
% the relaxation paramter 'omega', the seed (column) vector 'x0' for
% the iteration process, the tolerance 'tol' which will cause the
% iteration to stop if the infinity-norms of successive iterates
% become smaller than 'tol', and 'kmax' which is the maximum number
% of iterations to perform.
% Outputs: the final iterate 'x', the number of iterations performed
% 'k', and a vector 'diff' which records the 2-norms of successive
% differences of iterates.
% If any of the last three input variables are not specified, default
% values of x0= zero column vector, tol=1e-10 and kmax=1000 are used.

%assign default input variables, as necessary
if nargin<5, x0=zeros(size(b)); end
if nargin<6, tol=1e-10; end
if nargin<7, kmax=1000; end

if min(abs(diags(:,1)))<eps
    error('Coefficient matrix has zero diagonal entries, iteration
cannot be performed.\r')
end

[n D]=size(diags);
xold=x0;
k=1; diff=[];

while k<=kmax
    xnew=b;
    for i=1:n
        for d=2:D %run thru non-main diagonals and scan for entries that
            effect xnew(i)
                ind=inds(d);
                if ind<0&i>-ind %diagonal below main and j<i case
                    aij=diags(i+ind,d);
                    xnew(i)=xnew(i)-aij*xnew(i+ind);
                elseif ind>0&i<=n-ind %diagonal above main and j>i case
                    aij=diags(i,d);
                    xnew(i)=xnew(i)-aij*xold(i + ind) ;
                end
            end
        end
    end
    diff=[diff; norm(xnew-xold)];
    xold=xnew;
    k=k+1;
end

```

```

        end
    end
    xnew(i)=xnew(i)/diags(i, 1) ;
    xnew(i)=omega*xnew(i)+(1-omega)*xold(i);
end
diff(k)=norm(xnew-xold, inf) ;
if diff(k)<tol
    fprintf('SOR iteration has converged in %d iterations\r', k)
    x=xnew;
    return
end
k=k+1; xold=xnew;
end
fprintf('SOR iteration failed to converge. \r')
x=xnew;

```

(b) In order to use this program, we must create the input matrix `diags` from the nontrivial diagonals of the matrix `A`. The needed vectors were constructed in the solution of EFR 7.30(b); we reproduce the relevant code:

```

>> vl=-1*ones(49,1); vl=[vl;0]; %seed vector for sub/super diagonals
secdiag=vl;
for i=1:49
    if i<49
        secdiag=[secdiag;vl];
    else
        secdiag=[secdiag;vl(1:49)];
    end
end

```

We now construct the columns of `diags` to be the nontrivial diagonals of `A` taken in the order of the vector:

```

>> inds=[ 0 1- 1 50 -50]
>> diags=zeros(2500,5) ;
>> diags(:,1)=4 ; diags(1:2499,[ 2 3])=[secdiag secdiag] ;
>> diags(1:2450 , [4 5])= [-ones(2450,1) -ones(2450,1)] ;

```

We will also need the vectors `x` and `b`; we assume they have been obtained (and entered in the workspace) in one of the ways shown in the solution of EFR 7.30. We now apply our new SOR program on this problem using the default tolerance:

```

>> tic
>> [xsor, k, diff]=sorsparsediag(diags, inds,b,2/(1+sin(pi/51)),
zeros(size(b))); toc
->SOR iteration has converged in 222 iterations
->elapsed_time = 0.6510
>> max(abs(xsor-x))
->ans = 6.1213e-010

```