Mesh Tools for Automatic Generation of Finite-Element Code in Two Dimensions*

Eugene C. Gartland, Jr.† Jiahua Que†

January, 1996

Abstract

This report describes a set of tools for creating, refining, searching, and visualizing unstructured finite-element meshes for general regions in two dimensions. These were developed in support of a project concerned with using symbolic and numeric techniques to automatically generate code for the finite-element analysis of problems in liquid crystal physics and other areas of applied computational science. Included are descriptions of the tools and data structures, C and Fortran source listings, and examples.

1 Introduction

This report describes a set of tools for creating, refining, searching, and visualizing unstructured finite-element meshes for general regions in two dimensions. The tools include C, Fortran-77, and Matlab programs, files, and data structures. They were developed in support of a project—involving the authors, Paul Wang (Kent State University), and Naveen Sharma (Xerox Corporation)—concerned with using symbolic and numeric techniques to automatically generate code for the finite-element analysis of problems in liquid crystal physics. This project grows out of earlier work by Wang and

*This research was partially supported by the National Science Foundation Science and Technology Center on Advanced Liquid Crystalline Optical Materials (ALCOM) under grant DMR 89-20147, by NSF grant DMS 93-10733, and by the Research Challenge Program of the Ohio Board of Regents.

†Department of Mathematics and Computer Science, Kent State University, P.O. Box 5190, Kent, Ohio 44242-0001 (gartland@mcs.kent.edu, jque@mcs.kent.edu).

Sharma on using auto-code-generation techniques for problems in structural mechanics ([10]).

Though widely used in practice, the data structures and tools for manipulating finite-element meshes are not standardized or readily available in the public domain. Every general finite-element package contains such tools; however their form and nature is driven by the particular package, implementation, and history of development. The mesh-tool packages we found were either too expensive (usually contained as a subset of a large, general, commercial software package, like NASTRAN [9] or MODULEF [5], [6]), too elaborate (like GEOMPACK [7] or Kaskade [4]), or too particular (like PLTMG [1], which is restricted to type-one triangular elements). We have opted to develop our own collection of simple tools, acknowledging that we are "reinventing the wheel."

We have taken the approach of starting from a given, coarse, initial mesh of a general two-dimensional region using isoparametric triangles or rectangles. The initial mesh (element node indices and node coordinates) is supplied in a simple ASCII file format; it can be generated by an individual using an editor or by an initial-mesh-generating program (of which we supply a few simple ones). The region is determined by a finite number of boundary segments, on which points can be determined from a user-provided Fortran-or-C-coded parameterization routine. Thus multiply-connected regions are allowed. All subsequent meshes are generated as refinements of the initial one. For our first implementation, a uniform refinement process is provided; although all the routines were developed in a more general way to enable local refinement, at some future point.

We have striven to make the package as clean and simple as possible, developing the minimum capabilities necessary to accommodate the needs of the larger project. However we have built in generality, flexibility, and extensibility where it seemed possible to do this without adding undo complexity and inefficiency. We have adopted the philosophy of mixing C and Fortran-77, using Fortran wherever possible but using C where necessary or appropriate. The basic mesh tree is a C data structure, and so all the routines that need to deal with it directly are necessarily in C. The preference for Fortran (where possible) is driven by the facts that it is the main language of the existing PIER code-generating system we are building upon and that there still seems to be (at this point) some advantages to Fortran for large-scale scientific computing—this mostly has to do with compiler optimization of numerical computations, software available on the new parallel computers (like the T3D at the Ohio Supercomputer Center), and the

like—see [8].

We have also attempted to make the programs and package as portable as possible, using ANSI-standard coding and public-domain software as much as possible. This goal has been difficult to satisfy completely.

2 Mesh and mesh-tree data structure

The basic mesh and mesh tree are designed to handle isoparametric triangles and quadrilaterals of arbitrary polynomial degree. Our notation and terminology generally conforms to that of Ciarlet [3] (the standard mathematical reference) and Becker, Carey and Oden [2] (a compatible introduction). Local node ordering is assumed to be consistent with that on the master element, with vertices being listed first and in counter-clockwise order (see [3, §2.2] and [2, §5.4]).

Sufficient links and pointers are included to facilitate navigation of the tree and to allow for non-uniform refinement, depths, and levels, anticipating some use of multi-level computing in actual applications. A small number of basic "tree navigation" routines are provided and are used by some of the other routines (such as in refinement, searching, interpolation, and producing graphical output). Also allowed is the mixing of different element types and degrees (for p-version finite-element applications); although the present refinement routine is only constructed to generate "children" of the same type and degree as their "parent."

The "include" file, which contains a description of the basic "element" data structure, is described below.

2.1 Include file: `element.h`

Constants:

`NEG`	constant -1
`TWOPI`	constant 2π
`MAX_NODES`	constant 9

Macros:

`MOD(x,y)`	the least residue of x modulo y

Data structure:

`element`	basic data structure for an individual element

Components:

`int elt`	*shape of the element* `elt` $= 1$ triangle `elt` $= 2$ rectangle For root, if `elt` $= 0$ the tree is of mix-shaped elements, otherwise `elt` denotes the element shape for all element in the tree.
`int type`	*type of the element* triangle: `type` $= 1$ three nodes: the three vertices `type` $= 2$ six nodes: the three vertices and edge midpoints rectangle: `type` $= 1$ four nodes: the four vertices `type` $= 2$ nine nodes: the four vertices and edge midpoints plus the center point of the rectangle For root, if `elt` $\neq 0$ `type` will denote the element type for all element in the tree.
`int global_no`	*global number of the element* This is assigned to the element when it is produced. This number is the global order of the element in its element tree. The `global_no` of root is 0. The largest `global_no` + 1 is the total number of elements in the element tree.
`int local_no`	*order of the element among its siblings* For r siblings, the `local_no` is an integer from $0, 1, \ldots, r-1$. The `local_no` of root is the current deepest level in the tree.
`int tree_order`	*order of the element when the mesh-tree is navigated* If the mesh-tree is updated, this component may be changed.
`int depth`	*depth of the element in the element tree* Root has `depth` 0. Each other element has `depth` one more than that of its parent.

`int level`	*a number reflecting the history of the refinement process* An element is of `level` k if it is produced during the kth refinement. The `level` of root is 0. The `level` of all initial mesh elements is 1.
`int status`	*general purpose status flag*
`int num_node`	*number of nodes in the element* The `num_node` of root is the total number of nodes in the tree.
`int num_sub_ele`	*number of sub-elements (or children) of the element*
`int num_nbr`	*number of neighbors of this element* This is the same as the number of the element's edges. The `num_nbr` of root is the total number of elements (all levels) in the tree.
`int *node`	*an array of global node index numbers*
`element **sub_ele`	*array of pointers to sub-elements (i.e., children)*
`element **nbr`	*an array of pointers to neighbor elements* A neighbor element is an adjacent element in terms of spatial placement in the problem domain. `nbr` on boundary edge is always NULL pointer.
`element *parent`	*pointer to the parent of this element* The root element has a NULL parent. Every other element has a nontrivial parent from which it is produced during initialization or the refinement process.

Descriptions of the tree navigating routines, which are used to traverse a mesh tree in various ways, are listed below. All routines are in C.

2.2 Tree-navigation routines

There are five primitive routines to facilitate navigating the mesh tree in various ways. Also described is one generic/skeleton routine that uses these primitives to navigate the entire tree from a given starting point. This last routine is utilized by the save-file procedure, drawing routines, etc.

`tree_mv_down(` `int` *depth*, `element **`*element* `)`

Input:	*depth*	== desired depth for searching
	element	== element at which the search is started
Output:	*element*	== destination element

Search is started at *element* and continued downward (if possible) always following the leftmost branch to an element at a depth of *depth* or to an

element that is a leaf, which may be at a shallower depth.

`tree_mv_right`(`int` *depth*, `element **`*element*)

Input:	*depth*	== desired depth for searching
	element	== element at which the search is started
Output:	*element*	== destination element

Backtracking is started from *element* along the path to its ancestors. If the path is the rightmost one in the mesh-tree, backtracking will stop at root and return -1. Otherwise, the first path to its right will be found by locating the first ancestor of *element* on that path. Search is then started and continued downward (if possible) always following the leftmost branch to an element at a depth of *depth* or to an element that is a leaf, which may be at a shallower depth.

`elt_mv_sibling`(`element **`*element*)

Input:	*element*	== element at which the search is started
Output:	*element*	== destination element

Search is started from *element*. If *element* is the last one among its siblings, search will stop and return -1. Otherwise, its next sibling element will be found and passed to *element* as output.

`elt_mv_right`(`element **`*element*)

Input:	*element*	== element at which the search is started
Output:	*element*	== destination element

Backtracking is started from *element* along the path to its ancestors. If the path is the rightmost one in the mesh-tree, backtracking will stop at root and return -1. Otherwise, the first path to its right will be found by locating the first ancestor of *element* on that path. Its child on that path is then passed to *element* as output.

`elt_mv_next`(`element **`*element*)

Input:	*element*	== element at which the search is started
Output:	*element*	== destination element

Search is started from *element*. If *element* is not a leaf, the first child of it will be passed to *element* as an output. Otherwise, the mechanism of the above routine will be applied to find an element which will then be passed to *element* as an output or stop at root and return −1.

`tree_navigate`(`element **`*root*)

Input:	*root*	==	tree or subtree root at which the navigation is started

For procedures that require navigating the entire tree (like `tree_save_file()`, `tree_gen_array()`, `tree_gen_nbr()`, `tree_gen_order()` and `dw_data()` below), the basic algorithm described in Figure 1 applies.

Basic Algorithm

$ce \leftarrow root$
$k \leftarrow 0,\ count \leftarrow 0$

{ INITIAL OPERATIONS }

while `num_sub_ele` of $ce \neq 0$
    $count \leftarrow count + 1$
    `tree_mv_down`(k, ce)
    $k \leftarrow k + 1$

    { REPEATED OPERATIONS }

while `elt_mv_right`(ce) $\neq -1$
    $count \leftarrow count + 1$
    $k \leftarrow$ `depth` component of ce

    { REPEATED OPERATIONS }

    while `num_sub_ele` of $ce \neq 0$
        $count \leftarrow count + 1$
        $k \leftarrow k + 1$
        `tree_mv_down`(k, ce)

        { REPEATED OPERATIONS }

{ ENDING OPERATIONS }

Figure 1: Generic Tree-Navigation Routine

3 Save-file format and initial meshes

A mesh can be saved to or loaded from an ASCII disk file, and this is the most convenient way to create an initial coarse mesh. The format of the "save file" is described below, as are routines to save a mesh in this format and to load a mesh from such a file. A save file can be manually created, in an editor. We also provide some routines to generate initial meshes for eight standard test cases: a circle (of prescribed radius and center) meshed with type-one or type-two triangles or rectangles; and a rectangle (of prescribed width, height, and center) similarly meshed.

3.1 Save-file format

A mesh can be saved in a file by using a save-to-disk routine. The save file takes the following form:

1. General information (this may be an arbitrary number of rows headed by # in first column)
2. Size (total number of elements) of the tree (one row)
3. `elt` and `type` (one row)
4. Particular information used to retrieve the tree (number of rows = size)

There are $m+n+2$ columns of integers:

`depth`	`node`	`num_sub_ele`	`nbr`
1 column	m columns	1 column	n columns

$$\text{Here,}\quad m = \begin{cases} 3 & \text{if triangle, type one element tree} \\ 6 & \text{if triangle, type two element tree} \\ 4 & \text{if rectangle, type one element tree} \\ 9 & \text{if rectangle, type two element tree} \end{cases}$$

$$n = \begin{cases} 3 & \text{if triangle element tree} \\ 4 & \text{if rectangle element tree} \end{cases}$$

Example

The following is a simple save file which holds the basic infomation for initialization of a circle meshed with six equilaterel type-one triangular elements numbered counter-clockwise $1, \ldots, 6$.

```
#
# Circle with 6 type-1 triangles
#


 7
 1  1
 0  0  0  0  6  0   0  0
 1  0  1  2  0  6  -1  2
 1  0  2  3  0  1  -1  3
 1  0  3  4  0  2  -1  4
 1  0  4  5  0  3  -1  5
 1  0  5  6  0  4  -1  6
 1  0  6  1  0  5  -1  1
```

3.2 Save and load file routines

There are two main routines, `tree_save_file()` and `tree_load_file()`, to save and load a tree to and from a file. When a tree is loaded, the auxiliary routines `tree_retrieve()`, `tree_gen_array()`, and `tree_gen_nbr()` are used to form an identical tree structure.

The following routine will navigate an existing mesh tree rooted at *rt* and create a tree-file *file* for that tree in the format described above.

`void tree_save_file( element *`*rt*`, int `*size*`, char *`*file*` )`

Input:	*rt*	== root of the mesh-tree
	size	== size of global arrays t, x, y
	file	== name of tree-file to be generated

Algorithm

Based on the frame routine `tree_navigate()` given in Figure 1 (§2.2)

The { INITIAL OPERATIONS } are:

open $file$
$file \leftarrow$ arbitrary information
$file \leftarrow$ tree size
$file \leftarrow$ elt and type

The { REPEATED OPERATIONS } are:

$file \leftarrow$ components of ce :
`depth, node, num_sub_ele`, all `nbrs`' `tree_order`s, `global_no`

The { ENDING OPERATIONS } are:

close *file*

The following routine opens tree-file *file*, reads in the first data as the size (total number of elements) of the tree and then calls `tree_retrieve()` to generate the whole tree.

`void tree_load_file(` `char *`*file*, `Element **`*rt*)
Input: *file* == tree-file
Output: *rt* == root of retrieved tree

The following routine is a recursive function that will retrieve the whole mesh tree *rt* from a mesh tree file, which is already opened for reading and has the assigned file descriptor *fd*. It is called by `tree_load_file()`.

`void tree_retrieve(` `FILE *`*fd*, `element **`*rt*)
Input: *fd* == tree-file descriptor
Output: *rt* == root of retrieved tree

Algorithm

for current element
read integers from current line of *file*;
set its `depth`, `num_sub_ele`, `num_node`, `node`, `elt`, `type`, `global_no`, `tree_order`, `status` to corresponding integer;
for each `sub_ele`
set its components: `local_no`, `num_sub_ele`, `parent`.

The following routine will generate an element array for a mesh tree rooted at *rt* with size *size* and the `tree_order` component for all elements − the order in which the element is visited when the tree is navigated. The element array contains all elements in the mesh tree indexed in the order in which the tree is navigated. This array will be used to generate the `nbr` components for each element in the retrieved tree.

`void tree_gen_array( element *`*rt*`, int `*size*`, element **`*array* `)`

Input:	*rt*	== root of the mesh-tree
	size	== size of global arrays t, x, y
Output:	*array*	== tree element array generated

Algorithm

Based on the frame routine `tree_navigate()` given in Figure 1 (§2.2)

The { INITIAL OPERATIONS } are:

$array_0 \leftarrow ce$
`tree_order` of $rt \leftarrow count$

The { REPEATED OPERATIONS } are:

$array_{count} \leftarrow ce$
`tree_order` of $rt \leftarrow count$

No { ENDING OPERATIONS }

After calling `tree_retrieve_file()`, a retrieved tree rooted at *rt* is generated. However, the components for its elements are not yet complete. The following routine, `tree_gen_nbr()`, will be called to generate the `nbr` component for each element in the retrieved tree. In the tree-file *file*, there are several columns containing the order numbers of all neighbor elements for each element in the original tree. For the retrieved tree, `tree_gen_nbr()` will assign the appropriate elements to the `nbr` component of each element according to those numbers and by using the array *array*, an ordered element-array of the retrieved tree, created by `tree_gen_array()`.

The element array *array* of a retrieved tree must be generated before setting the `nbr` component to each element in the tree. The `tree_order`

entries in original tree-file are then used to indicate the indices in the element array for all neighbors of the element.

`void tree_gen_nbr(` `element *`*rt*, `element **`*array*, `char *`*file* `)`

Input:	*rt*	==	root of the tree retrieved from a tree-file
	array	==	tree element array of the retrieved tree
	file	==	the tree-file
Output:	*rt*	==	root of the retrieved tree with all `nbr` components properly set

Algorithm

Based on the frame routine `tree_navigate()` given in Figure 1 (§2.2)

The { INITIAL OPERATIONS } are:

open *file*
read *info*, *size*
read all integers from current line of *file*

The { REPEATED OPERATIONS } are:

read all integers from current line of *file*
set `nbr` components of *ce* by assigning proper *array* entries to them

The { ENDING OPERATIONS } are:

close *file*

4 Initialization routines

In this section, some routines used for initializing a given region are presented. The two simplest geometries for the boundary of the region are a *circle* and a *rectangle*. In each case, the region is divided into several elements with the same geometric properties. Associated boundary routines (necessary for drawing, mesh refinement, etc.) are also provided.

4.1 Initialization routines

`init_cir_tri1(` `float **`x, `float **`y, `float **`t, `element **`rt)

Input:	x, y	== global arrays of coordinate value x, y
	t	== global array of parameter t
	rt	== root of the element tree
Output:	rt	== root of the updated element tree
	t	== updated global array of parameter t
	x, y	== updated global arrays of coordinate value x, y

Generate an element tree of depth one. The root is a circle region. Information of this tree is already saved in a file. The routine first reads in this file (using `tree_load_file()`) and retrieves all components of the tree to make this region initialized. It then builds the x, y and t arrays, using the boundary routine `bdry_circle()` (see below). The center and radius of the circle are set by constant/parameter statements inside `init_cir_tri1()` and `bdry_circle()`, which must be consistent.

The routines `init_cir_tri2()`, `init_cir_rec1()`, `init_cir_rec2()`, `init_rec_tri1()`, `init_rec_tri2()`, `init_rec_rec1()`, `init_rec_rec2()`, `init_anl_tri1()`, `init_anl_tri2()`, `init_anl_rec1()`, `init_anl_rec2()` similarly generate type-one or type-two triangular or rectangular initial tessellation of a circle, rectangle or annulus.

4.2 Boundary routines

The user must have a "bdry" routine, which describes the geometry of the region and whose name is passed as an external procedure to routines that need to use it. This routine will be used by the refinement and drawing routines. The region may have any number of (interior or exterior) closed boundary segments. Boundary segments are parametered by parameter t. In the case of a region with hole(s), along the outer boundary, t will grow (starting at *tinit*) counter clockwise, while along any inner boundary, t will grow (starting at *tinit*) clockwise. This routine is in an "object oriented" style. All boundary-related data are encapsulated with a specific object. Which of them is extracted for further use is depending on the value of input switch. The routine will perform one of three tasks:

1. return total number of closed boundary segments.

2. give parameter range for a given boundary segment.
3. return $(x(t), y(t))$ coordinates of a boundary point on a specified segment.

`subroutine xy_bdry(` x, y, t, $flag$, $nseg$, $tinit$, $tfinal$)

Input:	t	== boundary parameter for the region
	$flag$	== flag for return value
	$nseg$	== boundary indicator
Output:	x, y	== coordinates of point on the boundary
	$tinit$	== initial value of t for boundary
	$tfinal$	== final value of t for boundary

This routine returns appropriate value(s) according to the values of $flag$ and $nseg$. When $flag = -1$, it will evaluate and return x, y coordinates of a given point on boundary segment $nseg$ from its boundary parameter value t. When $flag = 0$, it will return the total number of boundary segments set in $nseg$. When $flag = 1$, it will return $tinit$ and $tfinal$ of segment $nseg$, $nseg = 1, 2, 3, \cdots$. The name `xy_bdry` can be `bdry_circle`, `bdry_rectangle`, `bdry_annulus` in case of circle, rectangle or annulus boundary or other user specified name for particular application.

4.3 Examples

Figures of initializations in the twelve cases are illustrated below.

Circle domain

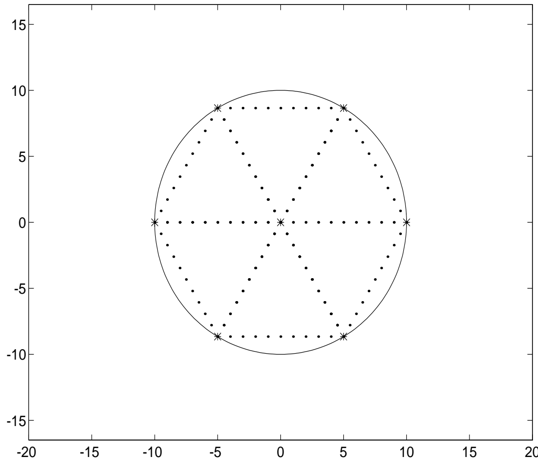


Fig. 4.1 type-one triangle

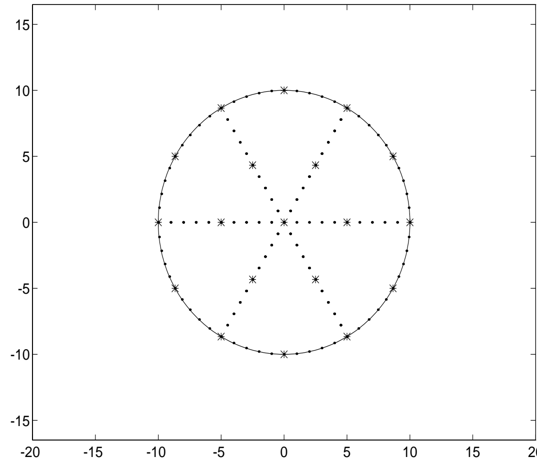


Fig. 4.2 type-two triangle

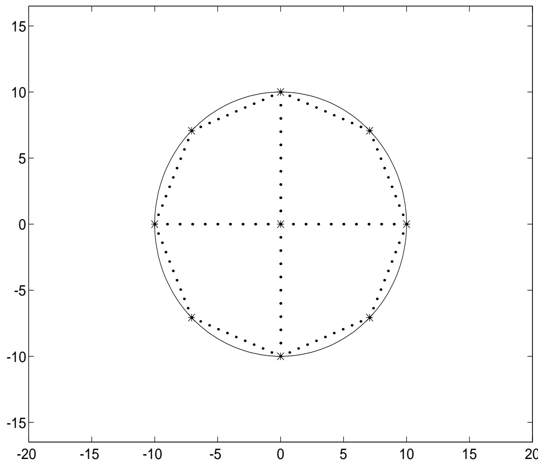


Fig. 4.3 type-one rectangle

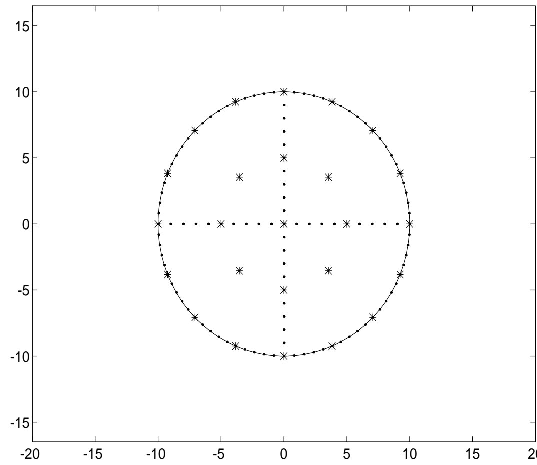


Fig. 4.4 type-two rectangle

Rectangle domain

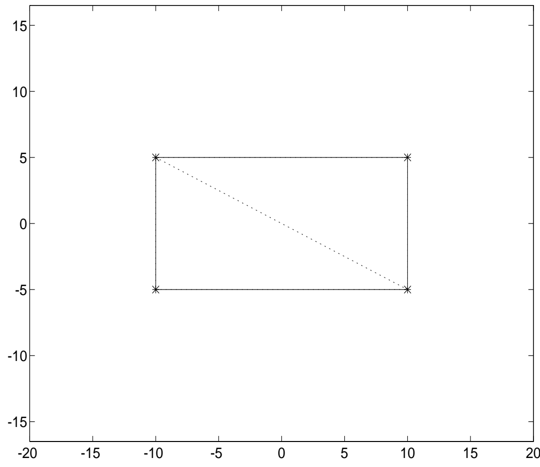


Fig. 4.5 type-one triangle

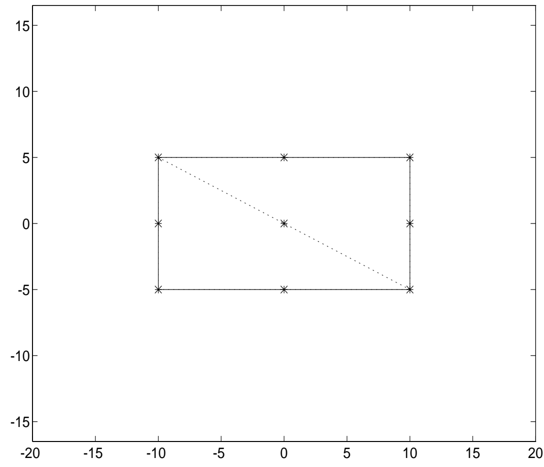


Fig. 4.6 type-two triangle

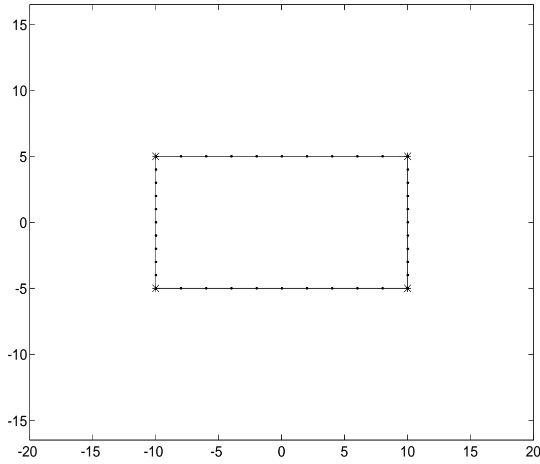


Fig. 4.7 type-one rectangle

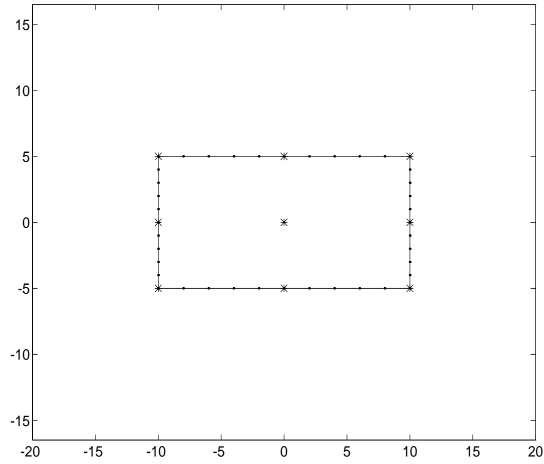


Fig. 4.8 type-two rectangle

Annulus domain

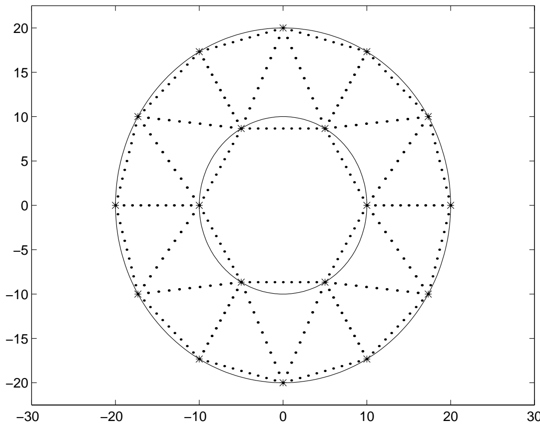


Fig. 4.9 type-one triangle

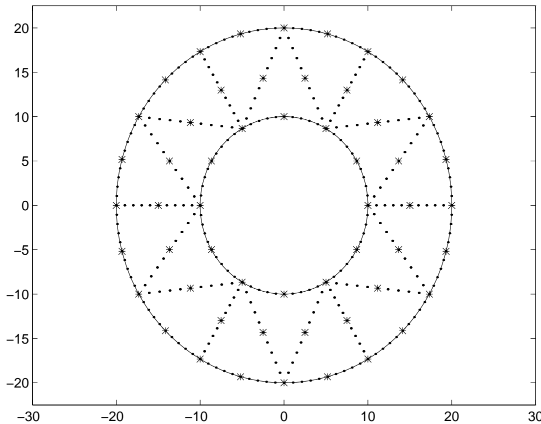


Fig. 4.10 type-two triangle

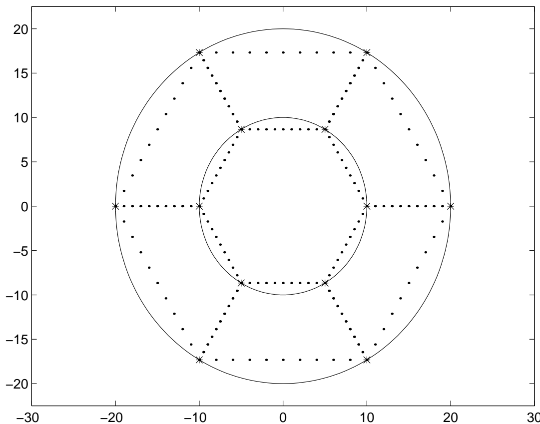


Fig. 4.11 type-one rectangle

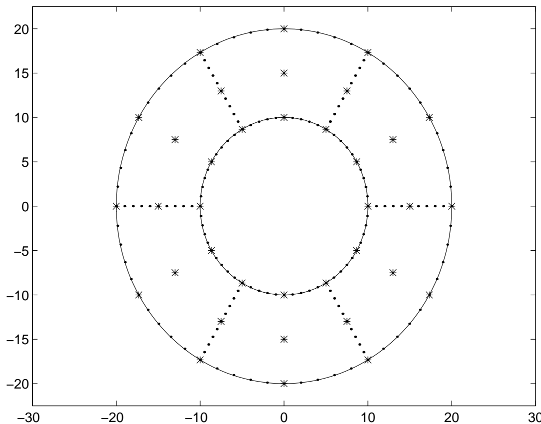


Fig. 4.12 type-two rectangle

5 Mesh drawing routines

Drawing routines are provided to visualize the generated meshes. These have various uses and were helpful in debugging some of the other routines (such as the initial-mesh and mesh-refinement routines). The main drawing routine, `dw_data()`, takes as inputs (1) a mesh tree (a pointer to the root of

a mesh tree), (2) `x[]`, `y[]`, `t[]` (arrays of nodal coordinates and parameter values), (3) an `xy_bdry()` external function (to generate boundary points $(x(t), y(t))$), and (4) a prescribed *level* (which gives the refinement level to be drawn). It outputs data files for a short Matlab plotting program, which does the actual plotting.

The routines and data files are described below. The basic mesh-navigation routines are used to visit each element at the prescribed level (or the maximum level, in case `level = 0`), outputting the data element by element. The resulting plots include the true boundary segments of the region (solid line), all the nodes (circles), and the boundaries of all the elements (dotted lines).

5.1 Drawing routines

The following routine will navigate the whole tree rooted at *root*. For each element visited, it calls the `elt_dat()` routine to generate the drawing data for this element.

`dw_data(` `float *`x, `float *`y, `int` *level*, `element *`rt)

Input:	x, y	== global arrays of coordinate value x, y
	level	== the `level` of elements which will be drawn
	rt	== the mesh-tree to be drawn
Output:	data generated for drawing are directly written into file *elt.dat*	

The following routine will take in the node array of an element and generates drawing points for each edge of the element. It proceeds by working from each edge of the associated master element, partitioning into equally spaced points, and then using the master element shape functions to map these points to true coordinates.

`subroutine elt_dat(` *nd*, *x_nd*, *y_nd*, *elt*, *type*, *psi*)

Input:	*nd*	== node number
	x_nd, *y_nd*	== for the element array of node coordinate value x, y
	psi	== work array for master element shape function values
	elt	== element shape
	type	== element type
Output:	data generated for drawing are directly written into file *elt.dat*	

Algorithm

Load (ξ_1, η_1), ..., (ξ_{nd}, η_{nd}) which are the coordinates of the nodes of the appropriate master element (nd is the number of nodes)

for each edge ab in the master element

$$\xi \leftarrow \xi_a + j\,(\xi_b - \xi_a)\,/\,step_number$$
$$\eta \leftarrow \eta_a + j\,(\eta_b - \eta_a)\,/\,step_number \qquad j = 0, \ldots, step_number$$

$$x_pt \leftarrow \sum_{j=1}^{nd} x_nd_j * \hat{\psi}_j(\xi, \eta)$$
$$y_pt \leftarrow \sum_{j=1}^{nd} y_nd_j * \hat{\psi}_j(\xi, \eta)$$

write x_pt, y_pt to *elt.dat*

Here, (x_pt, y_pt) is a plot point, x_nd, y_nd are the coordinate arrays of the nodes of the element which is being drawn. The master-element shape functions, $\hat{\psi}_j(\xi, \eta)$, are evaluated by the following Fortran subroutine. Formulas for these can be found in [2, §2.2]. They can also be discerned easily from the source file draw.f.

`subroutine psi_eval(`	*elt*, *type*, *xi*, *eta*, *psi*)	
Input:	*elt*	== element shape
	type	== element type
	xi, *eta*	== coordinates of a point
Output:	*psi*	== array of values determined by *xi*, *eta*

5.2 Data files and formats

Four data files are generated by the drawing routines above and are required by the matlab program to generate the actual plot.

info.dat: Drawing information

column 1	column 2	column 3
step_number on an edge	number of elements for given depth	number of nodes
column 4	column 5	
number of boundaries	number of points on boundary	

elt.dat: Coordinate arrays of drawing points on the edges of all elements for given depth.

column 1	column 2
x coordinate array	y coordinate array

bd.dat: Coordinate arrays of drawing points on the boundary (boundaries).

column 1	column 2
x coordinate array	y coordinate array

nod.dat: Coordinate arrays of all nodes in the mesh

column 1	column 2
x coordinate array	y coordinate array

Example:

Initializing the circle domain with center at (0,0) and radius 10 will produce the following data files:

info.dat					*elt.dat*		*bd.dat*		*nod.dat*	
11	18	7	1	200	0.00	0.00	10.00	0.00	0.00	0.00
					1.00	0.00	10.00	0.31	10.00	0.00
					2.00	0.00	9.98	0.63	5.00	8.66
					3.00	0.00	9.96	0.94	-5.00	8.66
					4.00	0.00	9.92	1.25	-10.00	0.00
					5.00	0.00	9.88	1.56	-5.00	-8.66
					6.00	0.00	9.82	1.87	5.00	8.66
					7.00	0.00	9.76	2.18		
					8.00	0.00	9.69	2.40		
					9.00	0.00	9.51	-3.09		
					10.00	0.00	⋮	⋮		
							-6.37	7.71		
					⋮	⋮	-6.61	7.50		
							-7.07	7.07		
					10.00	0.00	⋮	⋮		
					9.00	0.00	10.00	-0.31		
					8.00	0.00	10.00	0.00		
					⋮	⋮				
					0.00	0.00				

5.3 Matlab drawing routine: draw.m

The actual plot is created by the short matlab program draw.m.

Algorithm

Load *info.dat*, *elt.dat*, *bd.dat*, *nod.dat*
$N \leftarrow step_number + 1$, $M \leftarrow$ No. of elements, $L \leftarrow$ No. of nodes
$T \leftarrow$ No. of boundaries, $B \leftarrow$ number of points on boundaries

draw lines between every two succeeding points which data are read from all rows in *nod.dat*

for $j \leftarrow 1, \dots, M$
  $ilower \leftarrow (j - 1)\, N + 1$
  $iupper \leftarrow j\, N$

draw lines between every two succeeding points which data are read
from *ilower* to *iupper* rows in *elt.dat*

draw lines between every two succeeding points which data are read
from all rows in *bd.dat*
for $j \leftarrow 1, \ldots, T$
    ilower $\leftarrow (j - 1) B + 1$
    iupper $\leftarrow j B$
    draw lines between every two succeeding points which data are read
    from *ilower* to *iupper* rows in *bd.dat*

6 Searching and interpolation

A basic operation with such a geometric discretization as we are dealing with here is to find the element at the finest or some other specified level that contains a given point. Such a procedure is required, for example, to produce an approximate (interpolation) value to a solution at a point off the mesh. Because of the tree structure of the mesh, such a search is efficiently accomplished by working down through the refinement level from parents to sub-elements. The descriptions of these search and interpolation routines are listed below.

6.1 Search routines

Main routine here is `find_closure()`, which will find the *elt* on the finest mesh-level which contains a given *rt*; auxiliary routines `is_leaf()`, `angle()` and `is_contained()` are used.

`int is_leaf( element **`*element*)
Input: *element* == element in mesh-tree

If the *element* is a leaf of the mesh tree, then return 1, else return 0.

`float angle( float` x_p, `float` y_p, `float` x_1, `float` y_1, `float` x_2, `float` y_2)
Input: x_p, y_p == coordinates of point
x_1, y_1, x_2, y_2 == coordinates of endpoints of side

This routine computes the inner product of vector formed by (x_1, y_1), (x_p, y_p) and the normal of the vector formed by (x_1, y_1), (x_2, y_2).

`int is_contained(` `float` x_p, `float` y_p, `float` $x[\,]$, `float` $y[\,]$,
`element *`*element*, `int` n)

Input:	x_p, y_p	== coordinates of point
	x, y	== coordinate arrays of element nodes
	element	== element in mesh-tree
	n	== number of nodes

This routine determines whether the point (x_p, y_p) is located in the element *element* or not by computing n (n is the number of edges of *element*) inner products of the vectors formed by (x_i, y_i), (x_p, y_p) and the normal of the vectors formed by (x_i, y_i), (x_{i+1}, y_{i+1}), $i = 0, \ldots, n$, with $x_{n+1} = x_0,\ y_{n+1} = y_0$ by using `angle()`. If the point is in the element, return 1, otherwise return 0. It is assumed that the point is located inside the region, i.e., not beyond the boundary of the mesh.

`void find_closure(` `float` x_p, `float` y_p, `float *`x, `float *`y,
`element *`*element*, `element **`*elt*)

Input:	x_p, y_p	== coordinates of point
	x, y	== coordinate arrays of element nodes
	element	== element in mesh-tree
Output:	*elt*	== leaf

Search is started at *element*. If the point (x_p, y_p) is located in *element*, this routine will recursively search all sub-elements of *element* until an element which is a leaf and contains the point. If the point is not in *element*, search will start at the element to the right of *element*.

6.2 Interpolation

$<$ not yet implemented $>$

7 Mesh refinement

Mesh refinement is a process to refine all or part of the elements of a tree to form an updated mesh-tree. This process is based on operations on individual elements. The term *red refine* refers to the basic refinement operation whereby triangular elements are divided into four similar triangular elements by connecting the midpoints on the three sides and rectangular elements are divided into four similar rectangular elements by connecting the two pairs of points which are midpoints on opposite sides. The term *green refine* refers to the complementary refinement operation typically done to maintain element compatibility in the mesh: a triangular element is divided into two parts by connecting one of its vertices with the midpoint on the side opposite. An element list is given before the refinement process. If the list is empty, overall refinement is assumed; otherwise only those elements that are on the list will be red refined. Green refinement is done on neighbors of red refined element in order to guarantee a "legal" finite element mesh. The routines for tree refinement and individual element refinement are listed below.

7.1 Refinement routines

`refine( float **`x, `float **`y, `float **`t, `element *`rt,
`int` elt, `int` $type$, `struct list` lst, `void` $(*xy_bdry)$)

Input:	x, y	== global array of coordinate
	t	== global array of parameter t
	rt	== root of the mesh-tree
	elt	== shape of elements in the mesh-tree
	$type$	== type of the element
	lst	== list of elements to be refined
	xy_bdry	== user-defined boundary function
Output:	x, y	== updated global arrays of coordinate
	t	== updated global array of parameter t

Navigates all leaves in the mesh-tree and processes each individual element with one of the `t1_fulldiv()`, `t2_fulldiv()`, `r1_fulldiv()`, or `r2_fulldiv()` routines according to the shape and type of the elements in the mesh-tree and the region concerned.

`t1_fulldiv( float **`x, `float **`y, `float **`t, int $*array_size$, `element **`rt,
`element *`ce, `struct list` lst, `void` $(*xy_bdry)$)

Input:	x, y	== global array of coordinate values
	t	== global array of parameter t
	$array_size$	== size of the global arrays
	rt	== root of the element tree
	ce	== element visited
	lst	== list of elements to be fully divided
	xy_bdry	== user-defined boundary function
Output:	x, y	== updated global arrays of coordinate
	t	== updated global array of parameter t
	$array_size$	== updated size of the global arrays

Visits the type one triangle element *ce*. If *ce* is on the element list *lst* or *lst* is empty, red refines *ce*. For any neighbor of *ce* which is not divided, if it is in *lst* or *lst* is empty leaves it untouched, otherwise green refines it.

Algorithm

if list *lst* is not empty and element *ce* is not in *lst* terminate:

red refine *ce* by constructing *sub_ele*[0], ... , *sub_ele*[3] of element *ce*, all of triangle type 1;

update element *ce*:

> set `num_sub_ele` to 4;
> set `node`[0] for all sub-elements of *ce* which are not the central sub-element;
> set *nbr*[1] of these same sub-elements of *ce* to `sub_ele`[3] (the central sub-element of *ce*);
> set `nbr`[i] of the central sub-element to `sub_ele`[i], $i = 0, 1, 2$;

for each edge of *ce*

examine *ce*'s neighbor element on the edge;

if there is no neighbor, i.e., the edge is a part of the boundary:

create a new node which is the midpoint on the edge by calling boundary evaluating routines;
increase the size of global node array by 1.

if there is a neighbor element *ele*:

if *ele* is already red refined:

set *new_node* to the existing node which is the midpoint of
the edge since it is a node in *ele*;
set neighbor relations between adjacent sub-elements of *ce* and *ele*;

if *ele* is not red refined:

if *lst* is not empty and *ele* is not in *lst*:

green refine *ele*;
total number of elements is increased by 1;
set *num_sub_ele* of *ele* to 2;
set parent-child relations for *ele* and its sub-elements;
set neighbor relations between the adjacent elements of *ce* and *ele*.

set the node *new_node* as a node of the two sub-elements of *ce* on the edge;

set parent-child relations for *ce* and its sub-elements;

set neighbor relations for sub-elements of *ce*;

The routine `t2_fulldiv()` proceeds similarly, for type-two triangles. The routines `r1_fulldiv()` and `r2_fulldiv()` refine type-one and type-two rectangles in an analogous manner; they differ from the triangle refinement routines in that they do not perform green refinement of neighbors.

7.2 Examples

Some examples of refinement are given below.

Circle domain

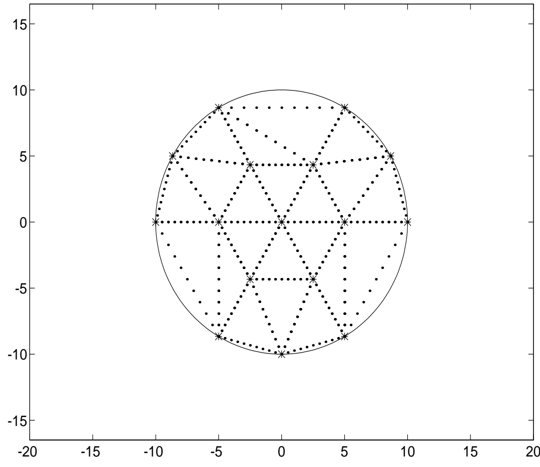


Fig. 7.1 type-one triangle

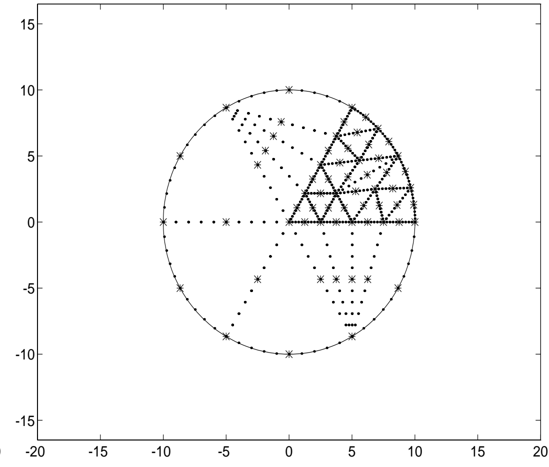


Fig. 7.2 type-two triangle

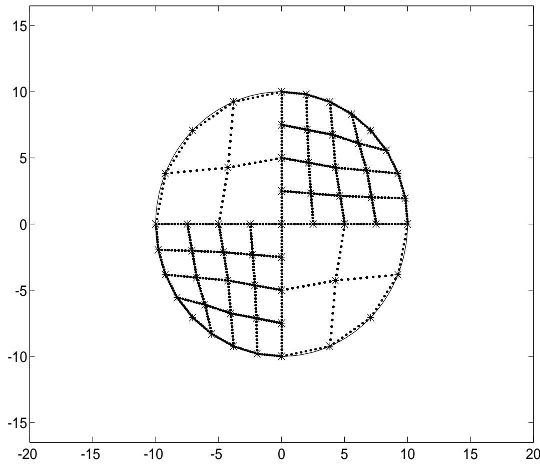


Fig. 7.3 type-one rectangle

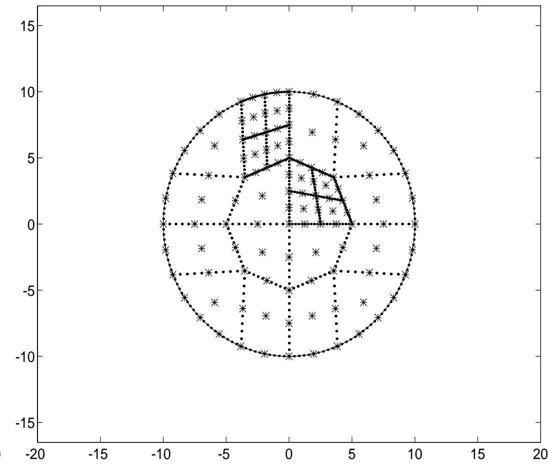


Fig. 7.4 type-two rectangle

Rectangle domain

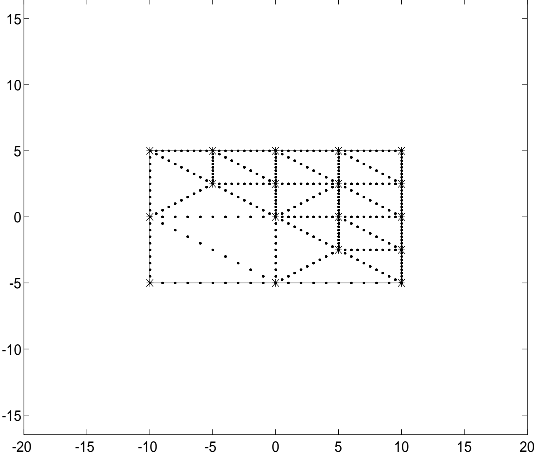


Fig. 7.5 type-one triangle

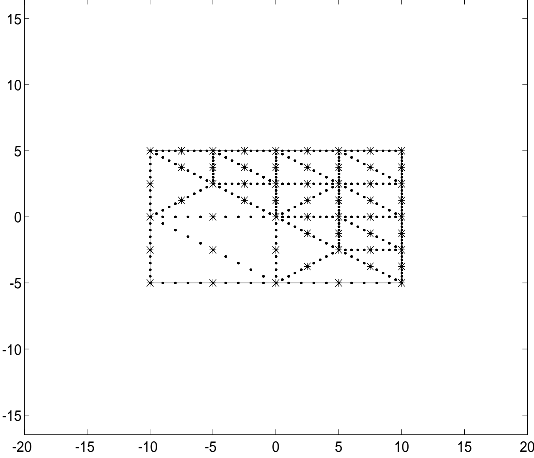
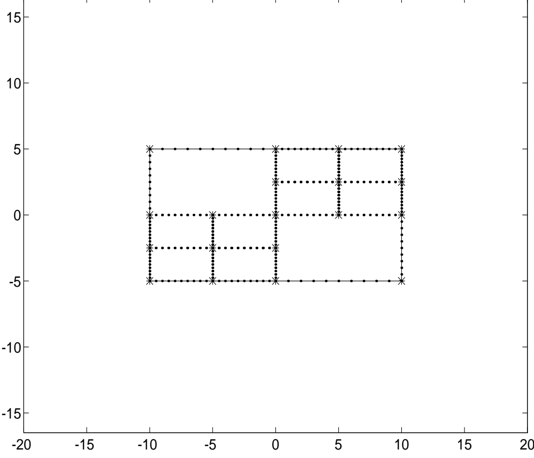


Fig. 7.6 type-two triangle



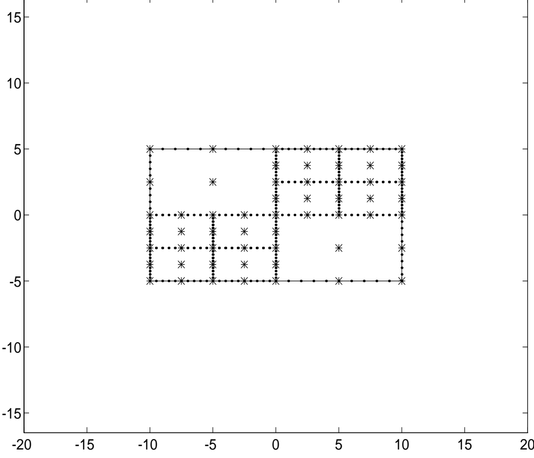


Fig. 7.7 type-one rectangle

Fig. 7.8 type-two rectangle

8 Makefiles and preprocessing

The followings are sample makefiles for SUN and HP

SUN SPARC :

```
# HUMMVEE version
# Note: We use gcc here since the only ANSI C compiler we have for Sparc
#       is public domain gnu C (gcc). After compiling a Fortran file with f77
#       the name of any function in that file, say fun, will be converted to
#       _fun_ in the corresponding object file. However, after compiling a C
#       file with gcc, the name of any function, say also fun, will be converted
#       to _fun in the corresponding object file. So, if a function name appears
#       in both Fortran and C files, we redefine the function name (using the C
#       preprocessor) by appending a _ to the function name in C file before
#       compiling. This makes the function name identical when all object files
#       are linked together. The following CFLAG1, CFLAG2, CFLAG3 are
#       defined to serve this purpose.

.SUFFIXES : .o .f .c

CC = gcc
FF = f77
CFLAGS = -ansi -c
FFLAGS = -c
HFILE = element.h
OBJS_CORE = loadsave.o navigate.o refine.o f_draw.o c_draw.o
OBJS_USER = f_test.o testfci.o xy_bdry.o init.o ref_info.o
CFLAG1 = -Dtest_fci=test_fci_ -Ddata_file=data_file_ -Ddw_data=dw_data_
CFLAG2 = -Dbdry_circle=bdry_circle_ -Dbdry_rectangle=bdry_rectangle_
         -Dbdry_annulus=bdry_annulus_
CFLAG3 = -Ddw_data=dw_data_ -Delt_dat=elt_dat_

all : f_test
.f.o :
	${FF} ${FFLAGS} $*.f
.c.o :
	${CC} ${CFLAGS} $*.c
loadsave.o: loadsave.c ${HFILE}
```

```
navigate.o: navigate.c ${HFILE}
testfci.o : testfci.c ${HFILE}
        ${CC} ${CFLAGS} ${CFLAG1} testfci.c
init.o : init.c ${HFILE}
        ${CC} ${CFLAGS} ${CFLAG2} init.c
c_draw.o : c_draw.c ${HFILE}
        ${CC} ${CFLAGS} ${CFLAG3} c_dw.c
refine.o : refine.c ${HFILE}
        ${CC} ${CFLAGS} ${CFLAG2} refine.c

f_test : ${OBJS_CORE} ${OBJS_USER}
        ${FF} ${OBJS_CORE} ${OBJS_USER} -o f_test -lm

clean :
        rm -f *.o *.dat
```

HP :

```
# CONDOR version
# Note: Option Aa for cc refers to ANSI mode. When option A is used, by the
#       request of the preprocessor cpp, the macro _INCLUDE_XOPEN_SOURCE
#       should be defined

.SUFFIXES : .o .c .f

CC = cc
FF = f77
CFLAGS = -Aa -c -D_INCLUDE_XOPEN_SOURCE
FFLAGS = -c
HFILE = element.h
OBJS_CORE = loadsave.o navigate.o refine.o f_draw.o c_draw.o
OBJS_USER = f_test.o testfci.o xy_bdry.o init.o ref_info.o

.f.o :
        ${FF} ${FFLAGS} $*.f
.c.o :
        ${CC} ${CFLAGS} $*.c
```

```
loadsave.o: ${HFILE}
navigate.o: ${HFILE}
testfci.o : ${HFILE}
init.o : ${HFILE}
c_draw.o : ${HFILE}
refine.o : ${HFILE}

all : f_test

f_test : ${OBJS_CORE} ${OBJS_USER}
	${FF} ${OBJS_CORE} ${OBJS_USER} -o f_test -lm
clean :
	rm -f *.o *.dat
```

Acknowledgment

The authors are grateful to Professor Paul S. Wang (Director of Research, Institute for Computational Mathematics, Department of Mathematics and Computer Science, Kent State University) and Dr. Naveen Sharma (Xerox Corp.) for their collaboration, support, and helpful discussions.

References

[1] Randolph E. Bank. *PLTMG, A Software Package for Solving Elliptic Partial Differential Equations: Users' Guide 7.0*, volume 15 of *Frontiers in Applied Mathematics*. SIAM, Philadelphia, 1994.

[2] Eric B. Becker, Graham F. Carey, and J. Tinsley Oden. *Finite Elements, An Introduction*, volume I of *The Texas Finite Element Series*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[3] Philippe G. Ciarlet. *The Finite Element Method for Elliptic Problems*. North-Holland Publishing Company, Amsterdam, New York, Oxford, 1978.

[4] Bodo Erdman, Jens Lang, and Rainer Roitzsch. *KASKADE Manual version 2.0*. Konrad-Zuse-Zentrum, Berlin, 1993.

[5] M. Bernadou et al. *A Modular Library of Finite Elements*. INRIA, 1985.

[6] P.L. George. *Mesh Generation and Modification*. INRIA, 1991. MODULEF User Guide number 3.

[7] Barry Joe. Geompack – a software package for the generation of meshes using geometric algorithms. *Adv. Eng. Software*, 13:325–331, 1991.

[8] Tom Macdonald. C versus fortran-77 for scientific programming. *Scientific Programming*, 1:99–114, 1992.

[9] MacNeal-Schwendler Corporation. *Introduction to MSC/NASTRAN Version 67*.

[10] Naveen Sharma. *Synthesis of Sequential and Parallel Programs for Finite Element Analysis*. PhD thesis, Kent State University, 1992.