Incompressible fluid flow and C^1 finite elements

Ana Maria Soane and Rouben Rostamian
Department of Mathematics and Statistics
UMBC

December 2007

Contents

Chapter 1

Problem specification

The user is expected to supply a *problem specification file* that contains the complete description of the problem to be solved and the desired post-processing actions. The file is expected to define the domain geometry, boundary conditions, and functions that specify boundary values and the forcing terms, all in standard C.

There is no restriction on the name of the file—it can have any name suitable for a C program—but we prefer to name it with a leading underscore, such as *_ell-shaped.c*, to make it stand out among other files in a directory listing.

A problems specification file consists of four distinct section:

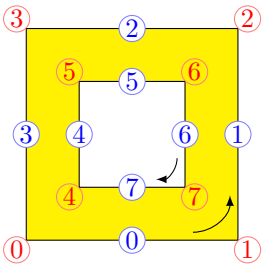
File headers These consist of obligatory headers that declare various data structures which are needed to capture user input.

Functions This section includes the definitions of all function needed to specify the problem such as boundary values and forcing terms.

Geometry and boundary value specifications The geometry of the domain is specified here. The section also includes the types of boundary condition and the function that generates the boundary data.

Trailing material The items specified above are invisible outside the scope of the file. In this trailing section we set up a function that provides a means of external access to the data in this file. The material in the trailing section is fixed, that is, it is independent of the problem.

In the following sections we provide examples to illustrate the process.



$$
\begin{aligned}
-\Delta u &= f(x,y) && \text{in } \Omega, \\
\frac{\partial u}{\partial \nu} &= 0 && \text{on edges 0 and 2,} \\
u &= g_1(x,y) && \text{on edge 1,} \\
u &= g_3(x,y) && \text{on edge 3,} \\
u &= 0 && \text{on edges 4, 5, 6, 7.}
\end{aligned}
$$

Figure 1.1: The domain Ω is the region between the two squares with vertices at $(\pm 2, \pm 2)$ and $(\pm 1, \pm 1)$. The diagram on the left shows the domain and the labeling of its vertices and edges. Arrows indicate the orientation of the boundaries. The Poisson boundary value problem is specified on the right. The notation $\partial u/\partial \nu$ expresses the derivative of the solution u in the direction of outward normal to the boundary.

1.1 Example: *_square_hole.c*

This section gives a complete problem specification file, *_square_hole.c*, for a Poisson boundary value problem in a domain Ω consisting of the region between two squares with vertices at $(\pm 2, \pm 2)$, $(\pm 1, \pm 1)$, as shown in Figure 1.1.

File contents:

⟨*_square_hole.c* 8a⟩≡
- ⟨*headers for problem specification* 8b⟩
- ⟨*functions associated with _square_hole.c* 9a⟩
- ⟨*geometry and boundary conditions for _square_hole.c* 9b⟩
- ⟨*the trailing material for _square_hole.c* 11b⟩

1.1.1 The headers section

The chunk ⟨*headers for problem specification* 8b⟩ should include the following minimal headers. This is the same for all problem specification files:

⟨*headers for problem specification* 8b⟩≡ (8a 12a)
```
#include "make_mesh.h"
```

The contents of these header files will be described later. The user may add any other headers and preprocessor macros as needed.

1.1.2 The function specification section

The chunk ⟨*functions associated with _square_hole.c* 9a⟩ specifies zero or more functions that describe the problem's boundary values and the forcing terms.

For instance the data functions for the Poisson problem stated above may be:

⟨*functions associated with _square_hole.c* 9a⟩≡ (8a)

```
static double f(double x, double y, void *params)
{
    return x*y;
}

static double g1(double x, double y, void *params)
{
    return 1 - y*y;
}

static double g3(double x, double y, void *params)
{
    return 1 - y*y*y;
}
```

All functions in a problem specification file should conform to the prototype declared in ⟨*generic function prototype* (never defined)⟩. The `params` argument in each of these function allows the caller to pass arbitrary parameters to the function, as needed.

1.1.3 The geometry and boundary condition specification section

The geometry of domain is specified in terms nodes, edges and holes, the details of which will be explained in the following paragraphs.

Specification of the nodes The user is responsible for numbering the domain's vertices. Nodes may be numbered in any order in consecutive non-negative integers beginning with zero. Each node is specified as a triplet `n, x, y` of numbers where the integer `n` is the node number and `x` and `y` are its Cartesian coordinates. The triplets are stored in an array of `NodeData` structures. The vertex data for the domain shown in Figure 1.1 is entered in the file *_square_hole.c* as:

⟨*geometry and boundary conditions for _square_hole.c* 9b⟩≡ (8a) 10▷

```
static NodeData nodes[] = {
    { 0,    -2.0, -2.0 },
    { 1,    +2.0, -2.0 },
    { 2,    +2.0, +2.0 },
    { 3,    -2.0, +2.0 },

    { 4,    -1.0, -1.0 },
    { 5,    -1.0, +1.0 },
    { 6,    +1.0, +1.0 },
    { 7,    +1.0, -1.0 },
```

```
};
```

`NodeData`, which is a `typedef` for `struct NodeData`, is declared in *make_mesh.h* which will be described later.

Remark. The array of node data can have any name; it need not be called `nodes`. Similarly, the arrays of edge and hole data, `edges` and `holes`, described in the following paragraphs, may be named as desired. In fact, a problem definition file may have several definitions of the `edges` array, for example, under different names, for specifying alternative boundary conditions.

Specification of the edges The user is responsible for numbering the edges of the polygons that define the domain's boundary. Edges may be numbered in any order in consecutive non-negative integers beginning with zero, just as for nodes.

The domain's boundary (or boundaries, if there are holes) are viewed as *oriented curves*. The positive direction along the outer boundary is counter-clockwise. The positive direction along the boundaries of holes, if any, is clockwise. This agrees with the orientation convention in Green's theorem in calculus. Each edge inherits an orientation from the boundary. Thus may speak of the edge's *first vertex* and *second vertex* in an unambiguous way.

An edge is specified in terms of five items, `n`, `node1`, `node2`, `bc_type`, `bc_func`, which will now describe in detail.

The first items, `n`, is the edge number which is assigned by the user as described above. The next two items, `node1` and `node2`, are the node numbers of the vertices that are connected by this edge. The ordering is significant here: `node1]` `is the edge's first vertex and [[node2` is the edge's second vertex.

The fourth item, `bc_type`, is an integer that indicates the *type* of the boundary condition applied to the edge. The list of supported boundary types is given in ⟨*boundary condition types* (never defined)⟩.

The fifth item, `bc_func`, is a pointer to a function that supplies the boundary values. The function should conform to the generic function prototype, `Func`, declared in ⟨*generic function prototype* (never defined)⟩. To specify a zero boundary value, `bc_func` may be set to `NULL`.

The edge data for the domain shown in Figure 1.1 is entered in the file *_square_hole.c* as:

⟨*geometry and boundary conditions for _square_hole.c* 9b⟩+≡ (8a) ◁9b 11a▷

```
static EdgeData edges[] = {
    { 0,    0, 1,   BC_NEUMANN,     NULL },
    { 1,    1, 2,   BC_DIRICHLET,   g1 },
    { 2,    2, 3,   BC_NEUMANN,     NULL },
    { 3,    3, 0,   BC_NEUMANN,     g3 },

    { 4,    4, 5,   BC_DIRICHLET,   NULL },
    { 5,    5, 6,   BC_DIRICHLET,   NULL },
    { 6,    6, 7,   BC_DIRICHLET,   NULL },
```

```
    { 7,    7, 4,   BC_DIRICHLET,   NULL },
};
```

Specification of the holes The locations of holes in the domain, if any, are specified in the array `holes` of type `HoleData`. A hole is specified by giving the Cartesian coordinates of an arbitrary point within the hole. The hole is completely determined by that point and the edges that surround it.

The domain in Figure 1.1 has a single hole, therefore the array `holes` consists of a single element:

⟨*geometry and boundary conditions for _square_hole.c* 9b⟩+≡ (8a) ◁10

```
static HoleData holes[] = {
    { 0.0, 0.0 },
};
```

Remark. If the domain has no holes, replace the definition of `holes` with:

```
static HoleData *holes = NULL;
```

1.1.4 The trailing material

The only externally visible object in the problem specification file is the function `get_fem` which meshes the domain according to the user specified data and returns a `fem` structure to the called. The `fem` structure encapsulates the complete definition of the boundary value problem, including the domain, mesh, and supplied boundary and forcing functions. The function `get_fem` takes a single argument, which specifies an upper bound to the largest triangle in the domain's triangulation.

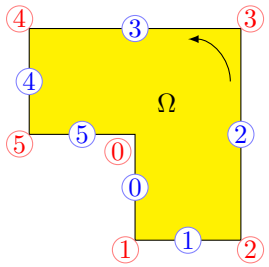
⟨*the trailing material for _square_hole.c* 11b⟩≡ (8a)

```
#define N(array) (sizeof(array) / sizeof((array)[0]))
Mesh *get_fem(double elem_max_area)
{
    return make_mesh(nodes, N(nodes), edges, N(edges), holes, N(holes),
            elem_max_area);
}
```

The preprocessor macro, `N`, is defined for convenience; it calculates the number of elements of a C array.

1.2 Example: *_ell-shape.c*

This section gives a complete problem specification file, *_ell-shape.c*, for a Poisson boundary value problem in an L-shaped domain Ω shown in Figure 1.2. The coordinates of all vertices are -1, 0 or $+1$.



$$
\begin{aligned}
-\Delta u &= f(x,y) && \text{in } \Omega, \\
u &= 0 && \text{on edges 0, 1, 5,} \\
u &= g_2(x,y) && \text{on edge 2,} \\
u &= g_4(x,y) && \text{on edge 4,} \\
\frac{\partial u}{\partial \nu} &= 0 && \text{on edge 3.}
\end{aligned}
$$

Figure 1.2: The domain Ω is the L-shaped domain shown on the left. The vertex coordinates are -1, 0 or $+1$.. The arrow indicates the positive direction along the boundary. The Poisson boundary value problem is specified on the right. The notation $\partial u/\partial \nu$ expresses the derivative of the solution u in the direction of outward normal to the boundary.

File contents:

⟨*_ell-shape.c* 12a⟩≡
⟨*headers for problem specification* 8b⟩
⟨*functions associated with _ell-shape.c* 12b⟩
⟨*geometry and boundary conditions for _ell-shape.c* 13a⟩
⟨*the trailing material for _ell-shape.c* 13d⟩

1.2.1 The headers section

The contents of the headers section is the same as in the previous example because the headers section is problem-independent.

1.2.2 The function specification section

Here we specify he functions f, g_2 and g_4 that enter in the description of the boundary value problem shown in Figure 1.2.

⟨*functions associated with _ell-shape.c* 12b⟩≡ (12a)

```
static double f(double x, double y, void *params)
{
    return x*y;
}

static double g2(double x, double y, void *params)
{
    return 1 - y*y;
}

static double g4(double x, double y, void *params)
{
```

```
    return y*(1-y);
}
```

1.2.3 The geometry and boundary condition specification section

Nodes:

⟨*geometry and boundary conditions for _ell-shape.c* 13a⟩≡ (12a) 13b▷

```
static NodeData nodes[] = {
    { 0,     0.0,  0.0 },
    { 1,     0.0, -1.0 },
    { 2,    +1.0, -1.0 },
    { 3,    +1.0, +1.0 },
    { 4,    -1.0, +1.0 },
    { 5,    -1.0,  0.0 },
};
```

Edges:

⟨*geometry and boundary conditions for _ell-shape.c* 13a⟩+≡ (12a) ◁13a 13c▷

```
static EdgeData edges[] = {
    { 0,    0, 1,   BC_DIRICHLET,   NULL },
    { 1,    1, 2,   BC_DIRICHLET,   NULL },
    { 2,    2, 3,   BC_DIRICHLET,   g2 },
    { 3,    3, 4,   BC_NEUMANN,     NULL },
    { 4,    4, 5,   BC_DIRICHLET,   g4 },
    { 5,    5, 0,   BC_DIRICHLET,   NULL },
};
```

Holes: Since Ω has no holes, we set `holes` to `NULL`:

⟨*geometry and boundary conditions for _ell-shape.c* 13a⟩+≡ (12a) ◁13b

```
static HoleData *holes = NULL;
```

1.2.4 The trailing material

The contents of the trailing material is similar to that in the previous example except for that the macro `N` does not apply to `holes` since `holes` is not a C array.

⟨*the trailing material for _ell-shape.c* 13d⟩≡ (12a)

```
#define N(array) (sizeof(array) / sizeof((array)[0]))
Mesh *get_fem(double elem_max_area)
{
```

```
    return make_mesh(nodes, N(nodes), edges, N(edges), holes, 0,
            elem_max_area);
}
```

1.3 Compiling into modules

The obvious way to use a problem specification file is to compile and link it with the rest of the program files that constitute the FEM solver. There is something inelegant about this, however. We shouldn't have to recompile or relink the program just to alternate between solving a problem A and a problem B. A better way is to keep the solver and the problem specification separate. We compile the solver once for all into a regular executable program. We compile the program specification file into a dynamically loadable shared library object which we call a *module*. The executable receives the name of the module as a command line argument and calls the dynamic linking loader, `dlopen()`, to load the module at run time. Any number of problems can be solved by the same executable, just by giving it different command line arguments.

The dynamic linking loader `dlopen()` is not a part of the standard C therefore our method is not portable across operating systems. It is defined, however, under the POSIX standard, therefore systems compliant with POSIX should be able to follow our approach. The alternative is to forgo dynamic loading and just compile (and recompile) the program specification file along with the rest of the program in the traditional way.

A further complication arises due the the variability among compilers of the method of making a shared library. We use *GNU gcc* to compile our programs. The command:

```
gcc -shared -fPIC -Wl,-soname,myprob.so -o myprob.so myprob.c
```

compiles the program specification file *myprob.c* and produces the module *myprob.so*. If the name of the executable is *fem*, then to solve the problem described in *myprob.c* we enter the command:

```
fem --module myprob.so --poisson-solve
```

See Chapter 8 for the description of all command line options.

Chapter 2

Meshing

The problem specification data which is supplied by the user in the file *__problem.c*, is passed to the function function `make_mesh()` in the file *make_mesh.c* which analyzes and transforms the data quite extensively. The final outcome is the construction of a `Fem` structure which contains the complete problem statement and the corresponding meshed domain. This structure is returned to `make_mesh()`'s caller. A sample meshed domain is shown in Figure B.1.

2.1 Data structures for capturing user input

Before examining `make_mesh()`, we describe the data structures that are used to hold user-supplied geometry and related information. These are:

2.1.1 The `NodeData` and `EdgeData` structures

The `NodeData` and `EdgeData` structures hold information about a domain's nodes and edges (and boundary conditions). The examples in Chapter 1 show how the user input is captured into these structures.

The `NodeData` and `EdgeData` structures cross-reference each other, therefore we issue a preliminary incomplete declaration for both:

⟨*data structures for capturing user input* 15a⟩≡ (17c) 15b▷
```
typedef struct NodeData NodeData;
typedef struct EdgeData EdgeData;
```

The complete `NodeData` structure is declared as:

⟨*data structures for capturing user input* 15a⟩+≡ (17c) ◁15a 16a▷
```
struct NodeData {
    int nodeno;
```

```
    double x;
    double y;
    EdgeData *e1;           /* preceding boundary edge */
    EdgeData *e2;           /* succeeding boundary edge */
    int reentrant;          /* 0,1,2,... for reentrant, -1 otherwise */
    List *satellite_nodes;  /* extra nodes around a reentrant vertex */
};
```

The first three members correspond to the triplet `n, x, y` noted in Section 1.1.3. The remaining members are for the program's internal use. The roles of those entries will be described further down.

The complete `EdgeData` structure is declared as:

⟨*data structures for capturing user input* 15a⟩+≡ (17c) ◁15b 16b▷

```
struct EdgeData {
    int edgeno;
    int node1;              /* number of first node */
    int node2;              /* number of second node */
    enum bc_types bc_type;  /* boundary condition type */
    Func bc_func;           /* boundary condition data */
    NodeData *n1;           /* pointer to first node */
    NodeData *n2;           /* pointer to second node */
    int ancestry;           /* ancestor (used for splitting edges) */
    int patch;              /* patch number of this edge */
};
```

The structure's first five members correspond to the the values of `n`, `node1`, `node2`, `bc_type`, `bc_func` described Section 1.1.3. The remaining members are for the program's internal use. The roles of those entries will be described further down. The prototype `Func` for functions that supply the boundary values is declared in ⟨*generic function prototype* (never defined)⟩.

2.1.2 The `HoleData` structure

The `HoleData` structure is declared as:

⟨*data structures for capturing user input* 15a⟩+≡ (17c) ◁16a 17a▷

```
typedef struct HoleData {
    double x;
    double y;
} HoleData;
```

The `x` and `y` members hold the coordinates of a point in a hole in the domain. See Section 1.1.3 for an example.

2.1.3 The `RegionData` structure

The domain over which we solve the PDE may be subdivided into subdomains which are called *regions* in the *Triangle* library. The domain is triangulated such that no triangle straddles a region boundary. We store region data in a `RegionData` structure:

⟨*data structures for capturing user input* 15a⟩+≡ (17c) ◁16b

```
typedef struct {
    double x;
    double y;
    int marker;
} RegionData;
```

2.1.4 Prototype of `make_mesh()`

As noted in the beginning of this chapter, `make_mesh()` is the only function with external linkage in *make_mesh.c*. We declare its prototype in *make_mesh.h* to make it known to the external callers.

⟨*prototype make_mesh* 17b⟩≡ (17c)

```
Mesh *make_mesh(
        NodeData *nodes, int nnodes,
        EdgeData *edges, int nedges,
        HoleData *holes, int nholes,
        double elem_max_area);
```

2.1.5 The file *make_mesh.h*

The file *make_mesh.h* contains the data structures described in the previous sections.

⟨*make_mesh.h* 17c⟩≡

```
#ifndef H_MAKE_MESH_H
#define H_MAKE_MESH_H

#include "fem.h"
#include "linked-list.h"

typedef struct triangulateio Triangle;

⟨data structures for capturing user input 15a⟩
⟨prototype make_mesh 17b⟩
```

```
#endif /* H_MAKE_MESH_H */
```

2.2 The function `make_mesh()`

The file *make_mesh.c* consists of a large number of function but only `make_mesh()` is visible to the outside. All others are declared `static` therefore have no external linkage. The outline of `make_mesh()` is:

⟨*function make_mesh* 18⟩≡ (36)

```
#ifdef DEBUG
static void dump_node_and_edge_lists(List *node_list, List *edge_list)
{
    List *p;

    for (p = node_list; p != NULL; p = p->rest) {
        NodeData *node = p->first;
        printf("node %d: (%g, %g)\n", node->nodeno, node->x, node->y);
    }

    for (p = edge_list; p != NULL; p = p->rest) {
        EdgeData *edge = p->first;
        printf("edge %d: (%d -> %d), ancestor=%d\n",
            edge->edgeno, edge->n1->nodeno, edge->n2->nodeno, edge->ancestry);
    }
}
#endif

Mesh *make_mesh(
    NodeData *nodes, int nnodes,
    EdgeData *edges, int nedges,
    HoleData *holes, int nholes,
    double elem_max_area)
{
    Mesh *mesh;
    Triangle *out;
    List *node_list = NULL;
    List *edge_list = NULL;
    List *region_list = NULL;
    int npatches;
    int i;

    ⟨sanity check 19a⟩
    ⟨complete node arrays 19b⟩
    ⟨complete edge arrays 21a⟩
    ⟨identify boundary patches 22a⟩
    ⟨convert node and edge arrays into linked lists 24b⟩
```

```
    ⟨localize reentrant vertices 25a⟩
    ⟨insert satellite nodes 29⟩
    ⟨sort node and edge lists 32a⟩
    /* dump_node_and_edge_lists(node_list, edge_list); */
    ⟨triangulate 32b⟩
    mesh = triangle_to_mesh(out, edges);
    mesh->nbseries = npatches;
    create_boundary_series(mesh);
    for (i = 0; i < npatches; i++)
        sort_bseries(mesh->bseries[i]);
    /* TEMPORARY */
    {
    for (i = 0; i < npatches; i++) {
        List *p;
        printf("Sorted bseries\n");
        for (p = mesh->bseries[i]; p != NULL; p = p->rest) {
            Edge *edge = p->first;
            printf("%d:(%d -> %d) ",
                    edge->edgeno,
                    edge->n1->nodeno,
                    edge->n2->nodeno);
        }
        putchar('\n');
    }
    }
    return mesh;
}
```

We will describe the details of the various chunks that make up this function in the following sections.

2.2.1 Sanity check

When the domain has no holes, the `holes` argument of `make_mesh()` should be `NULL` and the `nholes` argument should be zero. Here we check that this is entered correctly:

⟨*sanity check* 19a⟩≡ (18)

```
if ((holes == NULL && nholes != 0) || (holes != NULL && nholes == 0))
    ABORT("mesh_mesh() called with inconsistent 'holes' and 'nholes' arguments");
```

2.2.2 Completing the node arrays

The user's data in file *__problem.c* initializes each `NodeData` structure only partially. The chunk ⟨*complete node arrays* 19b⟩ fills in the remaining members by calling:

⟨*complete node arrays* 19b⟩≡ (18)

```
complete_node_array(nodes, nnodes, edges, nedges);
```

where:

⟨*complete_node_array()* 20a⟩≡ (36) 20b▷

```
static void complete_node_array(
        NodeData *nodes, int nnodes,
        EdgeData *edges, int nedges)
{
    int i;
```

Filling the `e1` and `e2` members. The members `e1` and `e2` of a `node` are pointers to the two boundary edges that straddle that node. The edge `e1` precedes edge `e2` as we move through the `node` along the boundary along its positive orientation.

To assign values to the pointers `e1` and `e2`, we walk through the `edges` array. From the user's supplied data, `edge->node1` gives the node number of the edge's first node. Then `node[edge->node1]` gives that node's `NodeData` structure. Then `node[edge->node1].e2` gives that node's pointer to the node's forward edge, which we set it to point to `edge`.

⟨*complete_node_array()* 20a⟩+≡ (36) ◁20a 20c▷

```
    for (i = 0; i < nedges; i++) {
        EdgeData *edge = &edges[i];
        nodes[edge->node1].e2 = edge;
        nodes[edge->node2].e1 = edge;
    }
```

Filling the `reentrant` member. The member `reentrant` of a `NodeData` structure is a flag to signal a reentrant vertex. Reentrant vertices are enumerated $0, 1, 2, \ldots$ and their `reentrant` members are set to these values. Non-reentrant nodes receive the default value of -1.

Our algorithm will add "satellite nodes" around each reentrant vertex and cut-off specially marked regions for special treatment. The member `nodes[i].satellite_nodes` of a `NodeData` structure is a head of a linked list of the node's satellite nodes. Here we initialize the `reentrant` and `satellite_nodes` members to their default values:

⟨*complete_node_array()* 20a⟩+≡ (36) ◁20b

```
    for (i = 0; i < nnodes; i++) {
        nodes[i].reentrant = -1;
        nodes[i].satellite_nodes = NULL;
    }
}
```

2.2.3 Completing the edge arrays

The user's data in file *__problem.c* initializes each `EdgeData` structure only partially. The chunk ⟨*complete edge arrays* 21a⟩ fills in the remaining members by calling:

⟨*complete edge arrays* 21a⟩≡ (18)
```
complete_edge_array(nodes, nnodes, edges, nedges);
```

where:

⟨*complete_edge_array()* 21b⟩≡ (36) 21c▷
```
static void complete_edge_array(
        NodeData *nodes, int nnodes,
        EdgeData *edges, int nedges)
{
    int i;
```

The members `n1` and `n2` of an `edge` are pointers to the `NodeData` structures of the edge's vertices. The pointer `n1` corresponds to the edge's first node and the pointer `n2` corresponds to the edge's second node.

Initial edges have their `edge->ancestry` members set to be their edge numbers. Edges that are obtained by by spliting an edge in two (see Section 2.2.7) will inherit their parent edge's `edge->ancestry` value.

⟨*complete_edge_array()* 21b⟩+≡ (36) ◁21b
```
    for (i = 0; i < nedges; i++) {
        EdgeData *edge = &edges[i];
        edge->n1 = &nodes[edge->node1];
        edge->n2 = &nodes[edge->node2];
        edge->ancestry = edge->edgeno;
    }
}
```

2.2.4 Grouping boundary patches

For the reasons that will become clear when we apply boundary data to our boundary value problem, we need to identify distinct *boundary patches* for a given problem. A *boundary patch* is a maximal connected subset of boundary edges that have same type of boundary condition. Let us define a few terms to clarify this definition.

- By *'a connected subset of boundary edges'* we mean a subset of boundary edges that forms a connected set in the plane, that is, the set cannot be split into two subsets with a positive distance between them.

  Examples: Two adjacent edges of a square form a connected set. The four edges of a square form a connected set. Two opposite edges of a square

do not form a connected set. The set of edges that form the inner and outer boundaries of an annulus is not a connected set.

- Currently our solver handles boundary condition of types Dirichlet or Neumann. Our solver distinguishes between a *homogenous boundary condtion*, that is where the boundary data is zero, and *nonhomogeneous boundary conditions*, that is where non-zero boudary data is specified. We say two edges *'have same type of boundary condition'* if all of the following conditions hold:

  1. The boundary condition types are either both Dirichle or both Neumann.
  2. Both boundary conditions are homogeneous or both boundary conditions are nonhomogeneous.

  In particular, if an edge with a homogeneous Dirichlet data is *not* of the same type as an edge with a non-homogeneous Dirichlet data.

- We say 'a connected subset of boundary edges that have same type of boundary condition' is *'maximal'* if no greater subset of edges exists with the same property.

  Examples: If homogeneous Dirichlet data is prescribed on three edges of a square and nonhomogeneous Dirichlet data is prescribed on the fourth edge, then the fourth edge is a maximal set and so is the set of the first three edges.

The task of identifying the boundary patches is performed by calling:

⟨*identify boundary patches* 22a⟩≡ (18)
```
npatches = identify_boundary_patches(edges, nedges);
```

which identifyes boundary patches and assignes a number to each, beginning with zero. Then it stores the patch number of each edge in the `patch` member of the edge's `EdgeData` structure. It performs this task by making a linked list of all boundary edges, then splitting it into several linked lists, one per boundary patch.

⟨*function identify_boundary_patches()* 22b⟩≡ (36) 23a▷
```
static int identify_boundary_patches(EdgeData *edges, int nedges)
{
    List *list_in = NULL;
    int npatches = 0;
    int i;

    /* make a linked list of all edges */
    for (i = 0; i < nedges; i++)
        list_in = List_push(list_in, &edges[i]);
```

Now split the linked list of edges into two linked lists. The linked list `list_out` is a maximal connected subset of boundary edges that have same type of boundary condition, therefore it forms a boundary patch. The linked list `list_rem` contains the remaining edges. Then we subject `list_rem` to the same process and repeat until `list_rem` is empty.

⟨*function identify_boundary_patches()* 22b⟩+≡ (36) ◁22b

```
    do {
        List *list_out = NULL;
        List *list_rem = NULL;
        split_edgelist(&list_in, &list_out, &list_rem);
        while (list_out != NULL) {
            EdgeData *edge;
            list_out = List_pop(list_out, (void **)&edge);
            edge->patch = npatches;
        }
        npatches++;
        if (list_rem == NULL)
            break;
        list_in = list_rem;
    } while (1);

    return npatches;
}
```

The function `split_edgelist()` handles the actual splitting of the linked list. If `list_in` is empty, then there is nothing to split, therefore it returns. Otherwise it removes the first link from `list_in` and calls itself recursively with the remaining list. Once the resulting `list_out` is determined, it calls the function `fits_in()` to check if the previously removed link is "of the same type" as those in `list_out`. If so, it pushes that link into `list_out` otherwise it pushes that link into `list_rem`.

⟨*function split_edgelist()* 23b⟩≡ (36)

```
static void split_edgelist(List **list_in, List **list_out, List **list_rem)
{
    void *first;

    if (*list_in == NULL)
        return;

    *list_in = List_pop(*list_in, &first);
    split_edgelist(list_in, list_out, list_rem);
    if (*list_out == NULL || fits_in(*list_out, first))
        *list_out = List_push(*list_out, first);
    else
        *list_rem = List_push(*list_rem, first);
}
```

Here is `fits_in()`. It returns 1 (true) if `edge` is in the same boundary patch as those in `list_out`. Otherwise it returns zero (false). We introduce several intermediate boolean variables in the computation to make the flow of logic transparent. This is detrimental to the function's efficiency because all intermediate variables are evaluated even though the outcome of the funtion may be predictable from the very first test. Nevertheless, we leave the intermediate variables in because in our view the gain in clarity overrides the gain in efficiency in this case.

⟨*function fits_in()* 24a⟩≡ (36)

```
static int fits_in(List *list_out, EdgeData *edge)
{
    List *p;

    for (p = list_out; p != NULL; p = p->rest) {
        EdgeData *ep = p->first;
        int same_bc_type = edge->bc_type == ep->bc_type;
        int both_hom     = edge->bc_func == NULL && ep->bc_func == NULL;
        int both_non_hom = edge->bc_func != NULL && ep->bc_func != NULL;
        int same_type    = same_bc_type && (both_hom || both_non_hom);
        int n1_fits      = edge->n1 == ep->n1 || edge->n1 == ep->n2;
        int n2_fits      = edge->n2 == ep->n1 || edge->n2 == ep->n2;
        if (same_type && (n1_fits || n2_fits))
            return 1;
    }
    return 0;
}
```

2.2.5 Converting node and edge arrays into linked lists

We are going to extend the user-supplied nodes and edges by adding additional nodes and edges near reentrant vertices. To facilitate the expansion, we embed the `nodes` and `edges` arrays into linked lists. The original arrays will not be used after this point.

⟨*convert node and edge arrays into linked lists* 24b⟩≡ (18)

```
for (i = 0; i < nnodes; i++)
    node_list = List_push(node_list, &nodes[i]);

for (i = 0; i < nedges; i++)
    edge_list = List_push(edge_list, &edges[i]);
```

2.2.6 Localizing reentrant vertices

Around every reentrant vertex we create a local neighborhood as follows. We divide the interior angle $\alpha > \pi$ of a reentrant corner by $\pi/3$ then round up the

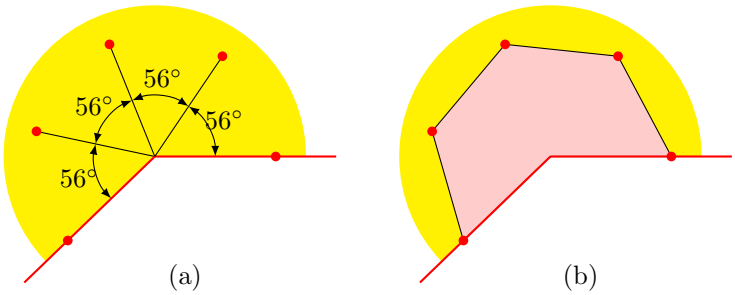


Figure 2.1: A reentrant angle of 224 degrees has been subdivided into four 56 degree angles by inserting five satellite nodes. This produces a local region near the reentrant corner which will be triangulated along with the rest of the domain.

result to an integer, let us say n. Then we divide α into n equal angles of size α/n each by inserting $n+1$ "satellite" nodes. This is best explained by referring to the diagram in Figure 2.1 where $\alpha = 224$ degrees, therefore $n = 4$. The line segments that connect the nodes delimit a *region* of the overall domain Ω.

Each region may be assigned a number (of type `double`) called the region's *attribute*. After triangulation, each triangle inherits its region's attribute. This makes it possible to tell to which region a triangle belongs. In our program, we enumerate the regions with integers 0, 1, 2, ..., and pass the number (cast to `double`) to *Triangle* as the region's attribute value.

The creation of satellite nodes and labeling of the local regions is done by calling:

⟨*localize reentrant vertices* 25a⟩≡ (18)
```
localize_reentrant_vertices(&node_list, &nnodes,
        /* &edge_list, &nedges, */ &region_list, elem_max_area);
```

where:

⟨*localize_reentrant_vertices* 25b⟩≡ (36) 26a▷
```
static void localize_reentrant_vertices(
        List **p_node_list, int *nnodes,
        /* List **p_edge_list, int *nedges, */
        List **region_list, double elem_max_area)
{
    List *node_list = *p_node_list;
    List *p;
    int reentrant_node_count = 0;
```

Identifying reentrant vertices. We walk over the list of nodes. For each node we identify its preceding and succeeding nodes. Then we form the vectors $\langle x_1, y_1\rangle$ and $\langle x_2, y_2\rangle$ which point from that node to its neighbors. It can be

shown that if $x_2y_1 - y_2x_1 < 0$, then the vertex is reentrant.

⟨*localize_reentrant_vertices* 25b⟩+≡ (36) ◁25b

```
for (p = node_list; p != NULL; p = p->rest) {
    NodeData *this_node = (NodeData *)p->first;
    EdgeData *prev_edge = this_node->e1;
    EdgeData *next_edge = this_node->e2;
    NodeData *prev_node = prev_edge->n1;
    NodeData *next_node = next_edge->n2;
    double x1 = prev_node->x - this_node->x;
    double y1 = prev_node->y - this_node->y;
    double x2 = next_node->x - this_node->x;
    double y2 = next_node->y - this_node->y;

    if (x2*y1-y2*x1 < 0)    /* reentrant corner */
        make_satellite_nodes(this_node, x1, y1, x2, y2, nnodes,
                region_list, &reentrant_node_count, elem_max_area);
}
}
```

The function `make_satellite_nodes`**.** The function `make_satellite_nodes`, called above, allocates and fills a `NodeData` structure for each of the satellite nodes of a given node. The structures are attached as a linked list to the node structure's `node->satellite_nodes` member. Here are the details:

⟨*make_satellite_nodes* 26b⟩≡ (36) 26c▷

```
static void make_satellite_nodes(NodeData *node, double x1, double y1,
        double x2, double y2, int *nnodes, List **region_list,
        int *reentrant_node_count, double elem_max_area)
{
```

First, we calculate the interior angle α of the reentrant corner, divide into $\pi/3$ radians (60 degrees), and round up the result to an integer n. Then divide the angle into n angle of size $t = \alpha/n$.

⟨*make_satellite_nodes* 26b⟩+≡ (36) ◁26b 26d▷

```
double alpha = 2.0*Pi + atan2(x2*y1-y2*x1, y2*y1+x1*x2);
int n = ceil(alpha/(Pi/3.0));
double t = alpha/n;
```

Next, find the length of the shorter of the two edges that emanate from the given node. Then compute r which will be the distance of the satellite nodes from the central node.

⟨*make_satellite_nodes* 26b⟩+≡ (36) ◁26c 27a▷

```
double len1 = sqrt(x1*x1 + y1*y1);
double len2 = sqrt(x2*x2 + y2*y2);
double min_len = MIN(len1, len2);
double r = MIN(0.25*min_len, sqrt(2*2*elem_max_area/sin(t)));
```

Then we find the unit vector pointing from the central node along the second edge and scale it by a factor of r to obtain a vector $\langle a, b\rangle$:

⟨*make_satellite_nodes* 26b⟩+≡ (36) ◁26d 27b▷

```
double a = x2 / len2 * r;
double b = y2 / len2 * r;
int j;
```

The satellite nodes will be placed at:

$$x_j = \hat{x} + a\cos jt - b\sin jt, \quad y_j = \hat{y} + a\sin jt + b\cos jt, \quad j = 0, 1, \ldots, n.$$

where $\langle \hat{x}, \hat{y}\rangle$ is the central node. This will define the central node's local neighborhood, as shown in Figure 2.1(b). The *Triangle* library needs to receive the coordinates of a point inside each local neighborhood. Additionally, it needs to receive the coordinates of a point outside of all local neighborhoods. If `*region_list` is `NULL`, we are dealing with the first reentrant node, therefore we take the opportunity to define the "outside point" at this time. Considering the sequence $\langle x_j, y_j\rangle$ defined above, the point with coordinates:

$$x = \hat{x} + 1.01(a\cos t - b\sin t), \quad y = \hat{y} + 1.01(a\sin t + b\cos t)$$

is a good candidate for the outside point.

⟨*make_satellite_nodes* 26b⟩+≡ (36) ◁27a 27c▷

```
RegionData *regiondata;

if (*region_list == NULL) {
    regiondata = xmalloc(sizeof *regiondata);
    regiondata->x = node->x + 1.01*(a*cos(t) - b*sin(t));
    regiondata->y = node->y + 1.01*(a*sin(t) + b*cos(t));
    regiondata->marker = -1;
    *region_list = List_push(*region_list, regiondata);
}
```

Similarly, the point with coordinates:

$$x = \hat{x} + 0.3(a\cos t - b\sin t), \quad y = \hat{y} + 0.3(a\sin t + b\cos t)$$

is a good candidate for the inside point.

⟨*make_satellite_nodes* 26b⟩+≡ (36) ◁27b 28a▷

```
node->reentrant = (*reentrant_node_count)++;
regiondata = xmalloc(sizeof *regiondata);
regiondata->x = node->x + 0.3*(a*cos(t) - b*sin(t));
regiondata->y = node->y + 0.3*(a*sin(t) + b*cos(t));
regiondata->marker = node->reentrant;
*region_list = List_push(*region_list, regiondata);
```

Finally, we create a `NodeData` for each satellite node and insert it in a linked list attached to `node->satellite_nodes`:

⟨*make_satellite_nodes* 26b⟩+≡ (36) ◁27c

```
    for (j = 0; j <= n; j++) {
        double x = node->x + a*cos(j*t) - b*sin(j*t);
        double y = node->y + a*sin(j*t) + b*cos(j*t);
        NodeData *new = xmalloc(sizeof *new);
        new->nodeno = (*nnodes)++;
        new->x = x;
        new->y = y;
        new->e1 = NULL;
        new->e2 = NULL;
        new->reentrant = -1;
        node->satellite_nodes = List_push(node->satellite_nodes, new);
    }
}
```

2.2.7 Merging satellite nodes with the original nodes

At this point, satellite nodes created in ⟨*make_satellite_nodes* 26b⟩ are being held in linked lists attached to the `satellite_nodes` member of the `NodeData` structure for each reentrant vertex. One of the goals in this section is to retrieve these nodes and merge them with the original set of nodes supplied by the user. Another goal is to identify and create new edges that connect the new nodes and merge them with the original set of edges supplied by the user.

Referring to Figure 2.1(b), we see that the introduction of five satellite nodes gives rise to four new edges that connect them. Additionally, the two satellite nodes that fall on the domain's boundary, split each of the two boundary edge into two. Therefore those two user-supplied edges need to be removed and replaced by four newly created edges.

The function `split_edge()`

To implement the latter observation, we introduce an auxiliary function, `split_edge()`, that takes an edge, say AB, and a node, say C, and replaces the edge AB by two edges AC and CB. In our application, the node C always will lie on the line segment AB, however the function `split_edge()` is more general and will work equally well when C is positioned arbitrarily relative to AB. See Figure 2.2. Since `split_edge()` changes the number of edges of the domain, it receives a pointer to the variable that holds the number of edges and updates it as needed.

⟨*split_edge* 28b⟩≡ (36)

```
static List *split_edge(List *edge_list, int *nedges,
        EdgeData *AB, NodeData *C)
{
    EdgeData *AC = xmalloc(sizeof *AC);
```

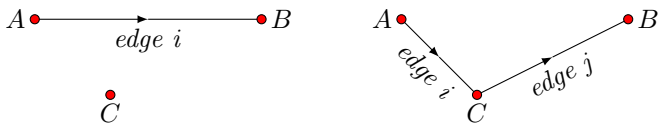


Figure 2.2: The function `split_edge()` takes an edge like AB and a node, like C, shown on the left, and replaces the edge AB with the edges AC and CB, shown on the right. The edge AC receives the same edge number as that of the removed edge AB. The edge CB receives a new edge number.

```
    EdgeData *CB = xmalloc(sizeof *CB);

    AC->edgeno = AB->edgeno;
    AC->n1 = AB->n1;
    AC->node1 = AC->n1->nodeno;
    AC->n2 = C;
    AC->node2 = AC->n2->nodeno;
    AC->bc_type = AB->bc_type;
    AC->bc_func = AB->bc_func;
    AC->ancestry = AB->ancestry;

    CB->edgeno = (*nedges)++;   /* new edge number */
    CB->n1 = C;
    CB->node1 = CB->n1->nodeno;
    CB->n2 = AB->n2;
    CB->node2 = CB->n2->nodeno;
    CB->bc_type = AB->bc_type;
    CB->bc_func = AB->bc_func;
    CB->ancestry = AB->ancestry;

    C->e1 = AC;
    C->e2 = CB;
    AC->n1->e2 = AC;
    CB->n2->e1 = CB;
    edge_list = List_remove(edge_list, AB, NULL);   /* CHECK FOR MEMORY LEAK */
    edge_list = List_push(edge_list, AC);
    edge_list = List_push(edge_list, CB);
    return edge_list;
}
```

The function `insert_satellite_nodes_and_edges()`

Now that we are done with `split_edge()`, we are ready to describe the chunk ⟨*insert satellite nodes* 29⟩. It is just one one function call:

⟨*insert satellite nodes* 29⟩≡ (18)

```
  insert_satellite_nodes_and_edges(&node_list,
```

```
        &nnodes, &edge_list, &nedges);
```

where:

⟨*insert_satellite_nodes_and_edges* 30a⟩≡ (36) 30b▷

```
static void insert_satellite_nodes_and_edges(
        List **p_node_list, int *nnodes,
        List **p_edge_list, int *nedges)
{
    List *node_list = *p_node_list;
    List *edge_list = *p_edge_list;
    List *p;
```

We walk over the list of nodes. If the `satellite_nodes` of a node structure is other than `NULL`, it has a linked list of satellite nodes hanging from it.

⟨*insert_satellite_nodes_and_edges* 30a⟩+≡ (36) ◁30a 30c▷

```
    for (p = node_list; p != NULL; p = p->rest) {
        NodeData *node = p->first;
        NodeData *saved = NULL;
        List *q = node->satellite_nodes;
        NodeData *sat_node;

        if (q == NULL)
            continue;
```

The first and last satellite nodes are special because they fall on the domain's boundary while the rest are internal to the domain. The boundary edges that receive stallelite nodes are split in two.

Satellite nodes are assembled in the linked list in the reverse order of their creation. Therefore the satellite node `q->first` splits the edge `node->e1`:

⟨*insert_satellite_nodes_and_edges* 30a⟩+≡ (36) ◁30b 30d▷

```
        sat_node = q->first;
        edge_list = split_edge(edge_list, nedges, node->e1, sat_node);
```

and the last satellite node splits the edge `node->e2`:

⟨*insert_satellite_nodes_and_edges* 30a⟩+≡ (36) ◁30c 30e▷

```
        q = List_reverse(q);
        sat_node = q->first;
        edge_list = split_edge(edge_list, nedges, node->e2, sat_node);
```

We go over the satellite nodes and merge them with the master list of nodes.

⟨*insert_satellite_nodes_and_edges* 30a⟩+≡ (36) ◁30d 31a▷

```
        while (q != NULL) {

            sat_node = q->first;

            node_list = List_push(node_list, sat_node);
```

At the same time, we handle the interior edges that connect the satellite nodes. If there are n satellite nodes, there are $n-1$ such edges. As we go through the linked list of satellite nodes, we save a pointer to the previously visited satellite node (except for the first node) and define an edge that connects the current node to the previously saved node.

⟨*insert_satellite_nodes_and_edges* 30a⟩+≡ (36) ◁30e

```
            if (saved != NULL) {
                EdgeData *edge = xmalloc(sizeof *edge);
                edge->edgeno = (*nedges)++;
                edge->n1 = saved;
                edge->node1 = edge->n1->nodeno;
                edge->n2 = sat_node;
                edge->node2 = edge->n2->nodeno;
                edge->bc_type = 0;  /* EXPLAIN */
                edge->bc_func = NULL;
                edge->ancestry = -10;   /* EXPLAIN */
                edge_list = List_push(edge_list, edge);
            }

            saved = sat_node;
            q = q->rest;
        }
        List_free(&(node->satellite_nodes));
    }
    *p_node_list = node_list;
    *p_edge_list = edge_list;
}
```

2.2.8 Sorting node and edge lists

Nodes in the expanded lists produced in ⟨*insert satellite nodes* 29⟩ are not in any particular order. The *Triangle* library expects to receive a list of nodes ordered by their node numbers, beginning with zero. Similarly with the list of edges. Therefore we apply the `List_sort()` function to put the lists of edges and nodes in order. A call to `List_sort()` needs to be supplied by a function that receives pointers to two nodes (or edges) and returns -1, 0, or $+1$ if the node (or edge) number of the first argument is lower, equal, or higher than that of the second.

For this purpose we introduce the functions:

⟨*functions for sorting node and edge lists* 31b⟩≡ (36)

```
static int cmp_nodes(const void *a, const void *b)
{
    NodeData *n1 = (NodeData *)a;
    NodeData *n2 = (NodeData *)b;
    if (n1->nodeno < n2->nodeno)
        return -1;
```

```
    else if (n1->nodeno == n2->nodeno)
        return 0;
    else
        return +1;
}

static int cmp_edges(const void *a, const void *b)
{
    EdgeData *e1 = (EdgeData *)a;
    EdgeData *e2 = (EdgeData *)b;
    if (e1->edgeno < e2->edgeno)
        return -1;
    else if (e1->edgeno == e2->edgeno)
        return 0;
    else
        return +1;
}

static void sort_node_and_edge_lists(
        List **node_list, List **edge_list)
{
    *node_list = List_sort(*node_list, cmp_nodes);
    *edge_list = List_sort(*edge_list, cmp_edges);
}
```

and then we call:

⟨*sort node and edge lists* 32a⟩≡ (18)

```
sort_node_and_edge_lists(&node_list, &edge_list);
```

2.2.9 Calling *Triangle* to perform triangulation

The chunk ⟨*triangulate* 32b⟩ passes the prepared lists of nodes and edges to the function `run_triangle()` which is a front end to the *Triangle* library which meshes the domain and returns its result in the variable `out` which is a pointer to a `struct triangulateio` structure which is declared in *Triangle*'s header file *triangle.h* and has been `typedef`'ed to `Triangle` in file *make_mesh.h*.

⟨*triangulate* 32b⟩≡ (18)

```
out = run_triangle(node_list, nnodes, edge_list, nedges,
        holes, nholes, region_list, elem_max_area);
```

where:

⟨*run_triangle* 32c⟩≡ (36) 33a▷

```
static Triangle *run_triangle(
        List *node_list, int nnodes,
        List *edge_list, int nedges,
        HoleData *holes, int nholes,
        List *region_list,
```

```
        double elem_max_area)
{
```

The structures pointed to by `in` and `out` will pass data to, and received data from, the *Triangle* library. The string `opts` will hold a string of options that control the details of the operation of *Triangle*.

⟨*run_triangle* 32c⟩+≡ (36) ◁32c 33b▷

```
    Triangle *in = xmalloc(sizeof *in);
    Triangle *out = xmalloc(sizeof *out);
    char opts[32];
    List *p;
    int i;
```

The input array '`double *in->pointlist`' holds the coordinates of the domain's vertices. The coordinates (x_i, y_i) of node number i are stored in the array's $2i$ and $2i + 1$ elements:

⟨*run_triangle* 32c⟩+≡ (36) ◁33a 34a▷

```
    in->numberofpoints = nnodes;
    MAKE_VECTOR(in->pointlist, in->numberofpoints * 2);

    for (p = node_list; p != NULL; p = p->rest) {
        NodeData *np = p->first;
        int i = np->nodeno;
        in->pointlist[2*i]   = np->x;
        in->pointlist[2*i+1] = np->y;
    }

    /* Point markers (needed?) */
    MAKE_VECTOR(in->pointmarkerlist, in->numberofpoints);
    for (i = 0; i < in->numberofpoints; i++)
        in->pointmarkerlist[i] = 555;   /* TEMPORARY */

    /* Point attributes (probably we won't have a use for this) */
    in->numberofpointattributes = 0;    /* TEMPORARY */
```

In *Triangle*'s jargon, *segments* are the input line segments that specify a domain's boundaries and regions. *Edges* are the output line segments that connects nodes.

The input array '`int *in->segmentlist`' holds the node numbers of a segment's first and second nodes. For edge number i, these are stored in in the array's $2i$ and $2i + 1$ elements.

Triangle associates with each segment a number called *the segment marker* whose values are stored in the array '`int *in->segmentmarkerlist`'. We store the ith edge's `ancestry` value plus 2 in `in->segmentmarkerlist[i`. The marker values will be inherited by the edge's sub-edges upon triangulation therefore we will know each sub-edge's ancessory by its marker value.

Remark. *Triangle* uses marker numbers 1 and 0 to indicate if an edge is or is not on the boundary. For instance, all internal edges have their markers set to 0. By adding 2 we are distinguishing our markers from defaults produced by *Triangle*.

⟨*run_triangle* 32c⟩+≡ (36) ◁33b 34b▷

```
in->numberofsegments = nedges;
MAKE_VECTOR(in->segmentlist, in->numberofsegments * 2);
MAKE_VECTOR(in->segmentmarkerlist, in->numberofsegments);
for (p = edge_list; p != NULL; p = p->rest) {
    EdgeData *ep = p->first;
    int i = ep->edgeno;
    in->segmentlist[2*i]     = ep->node1;
    in->segmentlist[2*i+1]   = ep->node2;
    in->segmentmarkerlist[i] = ep->ancestry + 2;
}
```

The input array '`double *in->regionlist`' holds information about the domain's *regions*. For region i, four data values are stored in the elements $4i$, $4i+1$, $4i+2$ and $4i+3$. The first two data values are the x and y coordinates of an of an arbitrary point within the region. The third data value is the region's *attribute* which will be inherited by all the triangles within that region. In this slot we store the `RegionData`'s `regiondata->marker` which has been assigned a number in ⟨*make satellite nodes* (never defined)⟩. Note that the The fourth data point is not used by FEM therefore we leave it unspecified.

⟨*run_triangle* 32c⟩+≡ (36) ◁34a 34c▷

```
in->numberofregions = List_length(region_list);
MAKE_VECTOR(in->regionlist, in->numberofregions * 4);
for (p = region_list, i = 0; p != NULL; p = p->rest, i++) {
    RegionData *regiondata = p->first;
    in->regionlist[4*i+0] = regiondata->x;
    in->regionlist[4*i+1] = regiondata->y;
    in->regionlist[4*i+2] = (double)regiondata->marker;
    /* in->regionlist[4*i+3] = unspecified; not used) */
}
```

The input array `double *in->holelist` holds the (x, y) coordinates of an arbitrary point inside each of the domain's holes.

⟨*run_triangle* 32c⟩+≡ (36) ◁34b 35a▷

```
in->numberofholes = nholes;
if (nholes != 0) {
    MAKE_VECTOR(in->holelist, in->numberofholes * 2);
    for (i = 0; i < nholes; i++) {
        in->holelist[2*i+0] = holes[i].x;
        in->holelist[2*i+1] = holes[i].y;
    }
}
```

The setup of the input structure is complete. Now we set up the output structure. Memory for arrays associated with the output will be allocated by *Triangle* provided that they are initialized to zero:

⟨*run_triangle* 32c⟩+≡ (36) ◁34c 35b▷

```
out->pointlist = NULL;
out->pointmarkerlist = NULL;
out->edgelist = NULL;
out->edgemarkerlist = NULL;
out->trianglelist = NULL;
out->triangleattributelist = NULL;

out->segmentlist = NULL;
out->segmentmarkerlist = NULL;
```

The actions of *Triangle* library's `triangulate()` function are controlled by a string argument which encodes the desired actions. Our encoded string contains:

Q Quiet operation; don't print out statistics.

z Node numbering begins with zero.

p The input data specifies a segment list.

A The input data specifies regional attributes.

j Discard duplicate vertices, if any.

e Produce edge list data.

q30 Accept no triangles with an angle less than 30 degrees.

a*x* Accept no triangles with an area greater than x.

⟨*run_triangle* 32c⟩+≡ (36) ◁35a

```
snprintf(opts, sizeof opts, "QzpAjeq30a%f", elem_max_area);
triangulate(opts, in, out, NULL);

free(in->pointlist);
free(in->pointmarkerlist);
free(in->segmentlist);
free(in->segmentmarkerlist);
free(in);

return out;
}
```

Remark. A pointer to `in->regionlist` is copied from *Triangle*'s input to its output. We will be referring to the `regionlist` later on, that's why there is no `free(in->regionlist)` in the code fragment above.

2.3 Function summary

We have completed the description of all functions that are involved in interpreting the user supplied data and passing them to *Triangle* to triangulate the domain. We group these function and the related preprocessor macros under a single chunk for later referencing:

⟨*functions defined in Chapter 2* 36⟩≡
```
#define MIN(x,y) ((x) < (y) ? (x) : (y))
#define Pi 3.14159265358979323846
```
⟨*function fits_in()* 24a⟩
⟨*function split_edgelist()* 23b⟩
⟨*function identify_boundary_patches()* 22b⟩
⟨*run_triangle* 32c⟩
⟨*functions for sorting node and edge lists* 31b⟩
⟨*split_edge* 28b⟩
⟨*insert_satellite_nodes_and_edges* 30a⟩
⟨*make_satellite_nodes* 26b⟩
⟨*localize_reentrant_vertices* 25b⟩
⟨*complete_node_array()* 20a⟩
⟨*complete_edge_array()* 21b⟩
⟨*function make_mesh* 18⟩

In the next chapter we will analyze *Triangle*'s output and cast its data into structures suitable for the rest of our work. That further processing of the triangulation data will introduce additional functions which we will group with the present function and write to file named *make_mesh.c*.

Chapter 3

Processing *Triangle*'s output

The purpose of the data structures `NodeData`, `EdgeData` and (others in the file *make_mesh.h*) is for preliminary processing of the user-supplied data up to the point where the data is passed to *Triangle*. Once the domain is meshed, those preliminary data structures are of no further use and are abandoned. In this chapter we will introduce a new set of data structures for holding mesh data then will transfer *Triangle*'s output to our data structures.

3.1 New data structures for mesh

The `Node` data structure. Nodes will be stored in `Node` data structures:

⟨*mesh data structures* 37a⟩≡ (45b) 38b▷

```
typedef struct {
    int nodeno;
    double x;
    double y;
} Node;
```

Here `nodeno` is the node's number, enumerated beginning with zero, and `x` and `y` are the node's Cartesian coordinates.

The `Edge` data structure. A boundary edge's boundary condition is specified by the boundary condition type and a a pointer to a function that generates the boundary values. The possible boundary condition types are:

⟨*boundary condition types* 37b⟩≡ (45b)

```
enum bc_types {
```

```
    BC_DIRICHLET,
    BC_NEUMANN
};
```

which correspond tho the traditional Dirichlet and Neumann types. The function that generates the boundary values should have a signature conforming the prototype:

⟨*generic function prototype* 38a⟩≡ (45b)
```
typedef double (*Func)(double x, double y, void *params);
```

The `params` argument in each of these function allows the caller to pass arbitrary parameters to the function, as needed.

Remark. The same function signature is used for specifying a PDE's forcing function.

We introduce a structure to hold boundary condition data:

⟨*mesh data structures* 37a⟩+≡ (45b) ◁37a 38c▷
```
typedef struct {
    int patch;
    enum bc_types bc_type;
    Func bc_func;
} BC;
```

And edge is strored in an `Edge` structure:

⟨*mesh data structures* 37a⟩+≡ (45b) ◁38b 38d▷
```
typedef struct {
    int edgeno;
    Node *n1;
    Node *n2;
    BC *bc;
} Edge;
```

Here `edgeno` is the edge's number, enumerated beginning with zero, `n1` and `n2` are pointers to `Node` structures of the edge's first and second nodes, respectively and `bc` is a pointer to a `BC` structure for boundary edges and `NULL` otherwise.

The `Elem` data structure. Triangles constructed by *Triangle*, which we call *elements*, will be stored in `Elem` data structures:

⟨*mesh data structures* 37a⟩+≡ (45b) ◁38c 39a▷
```
typedef struct {
    int elemno;
    Node *n[3];
    Edge *e[3];
    int marker;
    double **Q;
    double ***DQ;
```

```
    double ***D2Q;
} Elem;
```

Here `elemno` is the element's number, enumerated beginning with zero, `n` is an array of three pointers to the nodes at the element's vertices, `e` is an array of three pointers to the element's edges, and `marker` is the the region number that this element belongs to. If the element is in the local neighborhood of the reentrant vertex number i, then `marker` equals i, where $i = 0, 1, 3, \ldots$, otherwise `marker` equals -1. The `Q`, `DQ` and `D2Q` members point to arrays that hold data on the element's basis functions. These will be described in Section sec:computing-basis-functions.

The `Mesh` data structure. Upon return from *Triangle*, we know the number of nodes, edges and elements of the mesh. Therefore it is most efficient to store the data in arrays of fixed length rather than linked lists. The arrays are encapsulated in a `Mesh` structure:

⟨*mesh data structures* 37a⟩+≡ (45b) ◁38d 44b▷

```
typedef struct {
    int nnodes;
    int nedges;
    int nelems;
    int nbseries;
    Node *nodes;
    Edge *edges;
    Elem *elems;
    List **bseries;
} Mesh;
```

where the meanings of the members should be self-evident.

3.2 Reading *Triangle*'s output into the new data structures

The function `triangle_to_mesh()` receives a pointer to the *Triangle*'s output and extracts the information there into a `Mesh` data structures. It allocated the needed memory and returns a pointer to the `Mesh` structure.

⟨*function triangle_to_mesh* 39b⟩≡ (45c)

```
static Mesh *triangle_to_mesh(struct triangulateio *triangle_out, EdgeData *edges)
{
    Mesh *mesh;
    int i;

    mesh = xmalloc(sizeof *mesh);
    mesh->nnodes = triangle_out->numberofpoints;
    mesh->nedges = triangle_out->numberofedges;
```

```
    mesh->nelems = triangle_out->numberoftriangles;

    mesh->nodes = make_nodelist(mesh->nnodes);
    mesh->edges = make_edgelist(mesh->nedges);
    mesh->elems = make_elemlist(mesh->nelems);

    ⟨fill the array of nodes 40a⟩
    ⟨fill the array of edges 40b⟩
    ⟨fill the array of elems 40c⟩

    ⟨free triangle_out 42a⟩

    return mesh;
}
```

The `make_nodelist()`, `make_edgelist()` and `make_elemlist()` functions called above are simple memory allocation functions for the arrays of nodes, edges and elems. They will be described in Section 3.3. Once memories for these arrays are acquired, we may proceed to copy data from `triangle_out` into them.

⟨*fill the array of nodes* 40a⟩≡ (39b)

```
for (i = 0; i < mesh->nnodes; i++) {
    mesh->nodes[i].nodeno = i;
    mesh->nodes[i].x = triangle_out->pointlist[2*i];
    mesh->nodes[i].y = triangle_out->pointlist[2*i+1];
}
```

⟨*fill the array of edges* 40b⟩≡ (39b)

```
for (i = 0; i < mesh->nedges; i++) {
    mesh->edges[i].edgeno = i;
    mesh->edges[i].n1 = &mesh->nodes[triangle_out->edgelist[2*i]];
    mesh->edges[i].n2 = &mesh->nodes[triangle_out->edgelist[2*i+1]];
    if (triangle_out->edgemarkerlist[i] > 1) {
        int ancestor = triangle_out->edgemarkerlist[i] - 2;
        BC *bc = xmalloc(sizeof *bc);
        bc->bc_type = edges[ancestor].bc_type;
        bc->bc_func = edges[ancestor].bc_func;
        bc->patch = edges[ancestor].patch;
        mesh->edges[i].bc = bc;
    } else
        mesh->edges[i].bc = NULL;
}
```

⟨*fill the array of elems* 40c⟩≡ (39b)

```
for (i = 0; i < mesh->nelems; i++) {
    mesh->elems[i].elemno = i;
    mesh->elems[i].n[0] = &mesh->nodes[triangle_out->trianglelist[3*i]];
    mesh->elems[i].n[1] = &mesh->nodes[triangle_out->trianglelist[3*i+1]];
    mesh->elems[i].n[2] = &mesh->nodes[triangle_out->trianglelist[3*i+2]];
```

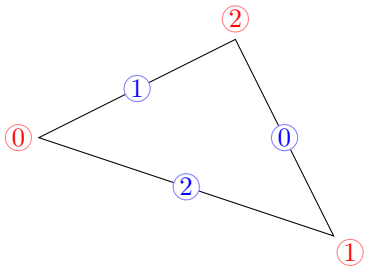


Figure 3.1: An element's edges are numbered according to their opposite vertices.

```
    mesh->elems[i].marker = (int)triangle_out->triangleattributelist[i];
    assign_elem_edges(mesh, &mesh->elems[i]);
}
```

The function `assign_elem_edges()` called in the chunk above fills the element's edge pointers, that is, its `Elem` structure's `e[0]`, `e[1]`, `e[2]` members. We enumerate an element's edges according to the vertex they face, that is, edge 0 is opposite vertex 0, edge 1 is opposite vertex 1, and edge 2 is opposite vertex 2. See Figure 3.1.

For function `assign_elem_edges()` receives a pointer to an element. For each of the element's three vertices it finds the node numbers `node1` and `node2` of the edge opposite the vertex. Then it scans the array of edges and finds the edge that connects those nodes. The element's `Elem` structure's edge pointer is set accordingly. Admittedly, this is not an efficient approach because it scans the array of edges $3n$ times where n is the number of triangles. It may be possible to replace this with a more clever approach if it turn out that efficiency is an issue.

⟨*function assign_elem_edges* 41⟩≡ (45c)

```
static void assign_elem_edges(Mesh *mesh, Elem *elem)
{
    int i, r;

    for (i = 0; i < 3; i++) {
        int j = (i+1)%3;
        int k = (i+2)%3;
        int node1 = elem->n[j]->nodeno;
        int node2 = elem->n[k]->nodeno;

        for (r = 0; r < mesh->nedges; r++) {
            Edge *edge = &mesh->edges[r];
            int m1 = edge->n1->nodeno;
            int m2 = edge->n2->nodeno;

            if ((m1 == node1 && m2 == node2)
```

```
                    || (m1 == node2 && m2 == node1)) {
                elem->e[i] = edge;
                break;
            }
        }
    }
}
```

Once data from *Triangle*'s output has been copied into a `Mesh` structure, *Triangle*'s data is no more needed, there we free the corresponding memory:

⟨*free triangle_out* 42a⟩≡ (39b)

```
free(triangle_out->pointlist);
free(triangle_out->pointmarkerlist);
free(triangle_out->edgelist);
free(triangle_out->edgemarkerlist);
free(triangle_out->segmentlist);
free(triangle_out->segmentmarkerlist);
free(triangle_out->trianglelist);
free(triangle_out->triangleattributelist);
free(triangle_out->regionlist);
free(triangle_out->holelist);
free(triangle_out);
```

3.3 Mesh constructor/destructor functions

Although allocating memory for the node, edge and elem arrays could have be done within the body of the function ⟨*function triangle_to_mesh* 39b⟩, we chose to separate them into auxiliary functions for greater flexibility. In fact, we define a function, `make_nodelist()` to make the array of nodes and another, `free_nodelist()` to deallocate the memory. Thus, if effect, we have a 'constructor' and a 'destructor' for the array of nodes:

⟨*mesh constructor/destructor functions* 42b⟩≡ (45c) 43a▷

```
static Node *make_nodelist(int nnodes)
{
    Node *nodelist;

    MAKE_VECTOR(nodelist, nnodes);

    /* later
    int i;
    for (i = 0; i < nnodes; i++)
        nodelist[i].ddpm = NULL;
    */

    return nodelist;
}
```

```
static void free_nodelist(Node *nodelist, int nnodes)
{
    if (nodelist == NULL)
        return;

    /* later
    int i;
    for (i = 0; i < nnodes; i++)
        FREE_MATRIX(nodelist[i].ddpm);
    */

    FREE_VECTOR(nodelist);
}
```

Similarly we define a constructor and destructor pair for the edge array:

⟨*mesh constructor/destructor functions* 42b⟩+≡ (45c) ◁42b 43b▷

```
static Edge *make_edgelist(int nedges)
{
    Edge *edgelist;

    MAKE_VECTOR(edgelist, nedges);

    return edgelist;
}

static void free_edgelist(Edge *edgelist, int nedges)
{
    FREE_VECTOR(edgelist);
}
```

And a constructor and destructor pair for the elem array:

⟨*mesh constructor/destructor functions* 42b⟩+≡ (45c) ◁43a

```
static Elem *make_elemlist(int nelems)
{
    Elem *elemlist;

    MAKE_VECTOR(elemlist, nelems);

    /* later
    for (i=0; i<nelems; i++) {
        elemlist[i].phi = NULL;
        elemlist[i].dphi = NULL;
        elemlist[i].d2phi = NULL;
        elemlist[i].k = NULL;
    }
    */

    return elemlist;
```

```
}

static void free_elemlist(Elem *elemlist, int nelems)
{
    if (elemlist == NULL)
        return;

    /* later
    for (i=0; i<nelems; i++) {
        FREE_VECTOR(elemlist[i].phi);
        FREE_VECTOR(elemlist[i].dphi);
        FREE_VECTOR(elemlist[i].d2phi);
        FREE_MATRIX(elemlist[i].k);
    }
    */

    FREE_VECTOR(elemlist);
}
```

Finally, we introduce a function to take care of freeing memory for a `Mesh` structure and all its members:

⟨*function free_mesh* 44a⟩≡ (45c)

```
void free_mesh(Mesh *mesh)
{
    if (mesh == NULL)
        return;

    free_nodelist(mesh->nodes, mesh->nnodes);
    free_edgelist(mesh->edges, mesh->nedges);
    free_elemlist(mesh->elems, mesh->nelems);
    free(mesh);
}
```

3.4 The file *fem.h*

Mesh data structures declared in this chapter are the central focus of most of FEM's functions. We encapsulate those declarations in a file *fem.h* for future reference. For future use, we add declarations for structures intended to represent points and vectors in two and three dimensions:

⟨*mesh data structures* 37a⟩+≡ (45b) ◁39a

```
typedef struct {
    double x;
    double y;
} Point2d, Vec2d;

typedef struct {
```

```
    double x;
    double y;
    double z;
} Point3d, Vec3d;
```

Furthermore, we add a prototype for the function `get_fem()` for external linking:

⟨*prototype get_fem* 45a⟩≡ (45b)
```
Mesh *get_fem(double elem_max_area);
```

⟨*fem.h* 45b⟩≡
```
#ifndef H_FEM_H
#define H_FEM_H

#include <stdlib.h>
#include "linked-list.h"

⟨generic function prototype 38a⟩
⟨boundary condition types 37b⟩
⟨mesh data structures 37a⟩
⟨prototype get_fem 45a⟩

#endif /* H_FEM_H */
```

3.5 The file *make_mesh.c*

The function defined in the current and the previous chapters interpret the user-supplied information, triangulate the domain, and produce a mesh structure. We lump the functions defined in these two chapter in a single file *make_mesh.c*.

⟨*make_mesh.c* 45c⟩≡
```
#include <stdio.h>
#include <math.h>
#include <triangle.h>
#include "xmalloc.h"
#include "make_mesh.h"
#include "linked-list.h"
#include "fem.h"
#include "array.h"
#include "abort.h"
#include "boundary.h"

⟨mesh constructor/destructor functions 42b⟩
⟨function free_mesh 44a⟩
⟨function assign_elem_edges 41⟩
⟨function triangle_to_mesh 39b⟩
⟨functions defined in Chapter 2 (never defined)⟩
```

Chapter 4

The Argyris element

4.1 Calculus on a triangle

Consider a general triangle with vertices at $\mathbf{v}_1 = \langle v_{11}, v_{12} \rangle$, $\mathbf{v}_2 = \langle v_{21}, v_{22} \rangle$, $\mathbf{v}_3 = \langle v_{31}, v_{32} \rangle$ ordered in the counter-clockwise direction. The triangle's *edge vectors* $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_3$ are defined by:

$$\mathbf{e}_1 = \mathbf{v}_3 - \mathbf{v}_2, \quad \mathbf{e}_2 = \mathbf{v}_1 - \mathbf{v}_3, \quad \mathbf{e}_3 = \mathbf{v}_2 - \mathbf{v}_1. \tag{4.1}$$

This is illustrated in Figure 4.1. Note that $\mathbf{e}_1 + \mathbf{e}_2 + \mathbf{e}_3 = \mathbf{0}$ always holds. Any two edge vectors determine the triangle's area, $\mathcal{A}$, which is given by:[1]

$$\mathcal{A} = \frac{1}{2}(e_{11}e_{22} - e_{21}e_{12}) = \frac{1}{2}(e_{21}e_{32} - e_{31}e_{22}) = \frac{1}{2}(e_{31}e_{12} - e_{11}e_{32}), \tag{4.2}$$

where e_{11}, e_{12}, etc., are the components of the edge vectors.

We will assume throughout, without further elaboration, that all triangles considered here are *non-degenerate*, that is, they have positive area.

A generic point $\mathbf{x} = \langle x_1, x_2 \rangle$ within the triangle is a unique convex combination of the vertices:

$$\mathbf{x} = \lambda_1 \mathbf{v}_1 + \lambda_2 \mathbf{v}_2 + \lambda_3 \mathbf{v}_3 \tag{4.3}$$

where $0 \leqslant \lambda_i \leqslant 1$ for $i = 1, 2, 3$, and $\lambda_1 + \lambda_2 + \lambda_3 = 1$. The numbers λ_1, λ_2, λ_3 are called the *barycentric coordinates* of the point $\mathbf{x}$. We write $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, \lambda_3)$ for short.

The transformation between $\mathbf{x}$ and $\boldsymbol{\lambda}$ may be viewed as a one-to-one mapping of the triangle $\mathcal{T}_2$ with vertices at $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_3$ in the plane and the triangle $\mathcal{T}_3$

[1] The classical intrinsic (coordinate-free) formula for a triangle's area is *Heron's formula*: $\mathcal{A} = \sqrt{p(p-a)(p-b)(p-c)}$ where $a = \|\mathbf{e}_1\|$, $b = \|\mathbf{e}_2\|$, $c = \|\mathbf{e}_3\|$, and $p = (a+b+c)/2$. An alternative intrinsic formula for the area is: $\mathcal{A}^2 = \frac{1}{4}(a^2b^2 + b^2c^2 + c^2a^2) - \frac{1}{16}(a^2 + b^2 + c^2)^2$. In our work we use the coordinate-based forms in (4.2) because they are easier to compute.

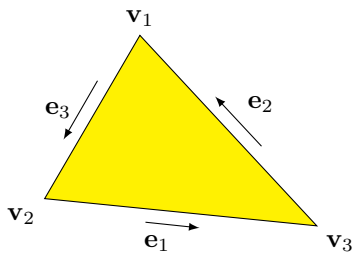


Figure 4.1: The vectors $\mathbf{e}_1$, $\mathbf{e}_2$, $\mathbf{e}_3$ shown are not faithful representations of their definitions given in (4.1). The vector $\mathbf{e}_1$, for instance, should extend from the vertex $\mathbf{v}_2$ to the vertex $\mathbf{v}_3$. The caricature rendition here is intended as a convenient mnemonic.

with vertices at $(1,0,0)$, $(0,1,0)$, $(0,0,1)$ in the $(\lambda_1,\lambda_2,\lambda_3)$ coordinate system in $\mathbf{R}^3$. See Figure 4.2. To emphasize the dependence of $\mathbf{x}$ on $\boldsymbol{\lambda}$ or vice versa, we write $\mathbf{x}(\boldsymbol{\lambda})$ and $\boldsymbol{\lambda}(\mathbf{x})$ when necessary.

The one-to-one nature of the mapping between triangles $\mathcal{T}_2$ and $\mathcal{T}_3$ may be made explicit by writing (4.3) in components in matrix form:

$$\begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix} = \begin{pmatrix} v_{11} & v_{21} & v_{31} \\ v_{12} & v_{22} & v_{32} \\ 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{pmatrix},$$

where the last row enforces the condition $\lambda_1 + \lambda_2 + \lambda_3 = 1$. It may be verified that the inverse of the mapping is:

$$\begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{pmatrix} = \frac{1}{\mathcal{J}} \begin{pmatrix} -e_{12} & e_{11} & v_{21}v_{32} - v_{31}v_{22} \\ -e_{22} & e_{21} & v_{31}v_{12} - v_{11}v_{32} \\ -e_{32} & e_{31} & v_{11}v_{22} - v_{21}v_{12} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix},$$

where $\mathcal{J} = 2\mathcal{A}$. Referring to (4.2) we have:

$$\mathcal{J} = e_{11}e_{22} - e_{21}e_{12} = e_{21}e_{32} - e_{31}e_{22} = e_{31}e_{12} - e_{11}e_{32}. \tag{4.4}$$

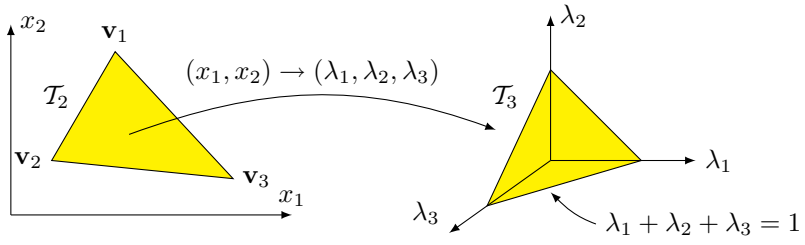


Figure 4.2: The triangle $\mathcal{T}_2$ in the (x_1, x_2) plane is mapped to the triangle $\mathcal{T}_3$ in the $(\lambda_1, \lambda_2, \lambda_3)$ space.

A slightly rearranged version of this equation is sometimes more useful:

$$\begin{pmatrix}\lambda_1\\ \lambda_2\\ \lambda_3\end{pmatrix} = \frac{1}{\mathcal{J}}\begin{pmatrix}-e_{12} & e_{11}\\ -e_{22} & e_{21}\\ -e_{32} & e_{31}\end{pmatrix}\begin{pmatrix}x_1\\ x_2\end{pmatrix} + \frac{1}{\mathcal{J}}\begin{pmatrix}v_{21}v_{32}-v_{31}v_{22}\\ v_{31}v_{12}-v_{11}v_{32}\\ v_{11}v_{22}-v_{21}v_{12}\end{pmatrix}. \tag{4.5}$$

For future reference we let:

$$A = \begin{pmatrix}v_{11} & v_{21} & v_{31}\\ v_{12} & v_{22} & v_{32}\\ 1 & 1 & 1\end{pmatrix}, \quad B = \frac{1}{\mathcal{J}}\begin{pmatrix}-e_{12} & e_{11}\\ -e_{22} & e_{21}\\ -e_{32} & e_{31}\end{pmatrix}, \tag{4.6}$$

and

$$\overline{B} = A^{-1} = \frac{1}{\mathcal{J}}\begin{pmatrix}-e_{12} & e_{11} & v_{21}v_{32}-v_{31}v_{22}\\ -e_{22} & e_{21} & v_{31}v_{12}-v_{11}v_{32}\\ -e_{32} & e_{31} & v_{11}v_{22}-v_{21}v_{12}\end{pmatrix}. \tag{4.7}$$

Equation (4.5) may be written in the compact form:

$$\boldsymbol{\lambda}(\mathbf{x}) = B\mathbf{x} + \mathbf{b} \tag{4.8}$$

from which it is evident that the 3×2 matrix B represents the derivative of the linear mapping $\mathbf{x} \to \boldsymbol{\lambda}(\mathbf{x})$. The matrix B will play a significant role in the rest of this chapter. It may be verified by direct calculation that:

$$B\,\mathbf{e}_1 = \begin{pmatrix}0\\ -1\\ 1\end{pmatrix}, \quad B\,\mathbf{e}_2 = \begin{pmatrix}1\\ 0\\ -1\end{pmatrix}, \quad B\,\mathbf{e}_3 = \begin{pmatrix}-1\\ 1\\ 0\end{pmatrix}. \tag{4.9}$$

4.2 Differentiation formulas

4.3 Notation for derivatives

Let X and Y be a vector spaces and let $f : X \to Y$. The function f is said to be *differentiable* at a point $\mathbf{x} \in X$ if there exists a linear operator $Df(\mathbf{x}) : X \to Y$ such that:

$$f(\mathbf{x}+\mathbf{a}) = f(\mathbf{x}) + Df(x)[a] + o(\mathbf{a}) \text{ for all } \mathbf{a} \in X,$$

where $Df(\mathbf{x})[\mathbf{a}]$ indicates the application of the operator $Df(\mathbf{x})$ to the vector $\mathbf{a}$, and where $o(\mathbf{a})$ is the usual "little o" notation which refers to a generic function such that $\lim_{\mathbf{a}\to 0} o(\mathbf{a})/\|\mathbf{a}\| = \mathbf{0}$. The operator $Df(\mathbf{x})$ is called *the derivative of f at* $\mathbf{x}$.

Similarly f is said to be *twice differentiable* at $\mathbf{x}$ if there exists a linear operator $D^2 f(\mathbf{x}) : X \times X \to Y$ such that

$$Df(\mathbf{x}+\mathbf{a}) = Df(\mathbf{x}) + D^2 f\big(\mathbf{x}\big)[\mathbf{a}] + o(\mathbf{a}), \text{ for all } \mathbf{a} \in X.$$

Consequently, if f is twice differentiable, then:

$$Df(\mathbf{x}+\mathbf{a})[\mathbf{b}] = Df(\mathbf{x})[\mathbf{b}] + \big(D^2 f(\mathbf{x})[\mathbf{a}]\big)[\mathbf{b}] + o(\mathbf{a}), \text{ for all } \mathbf{a}, \mathbf{b} \in X.$$

A basic theorem in calculus states that if f is twice differentiable at $\mathbf{x}$, then $\big(D^2 f(\mathbf{x})[\mathbf{a}]\big)[\mathbf{b}] = \big(D^2 f(\mathbf{x})[\mathbf{b}]\big)[\mathbf{a}]$ for all $\mathbf{a}, \mathbf{b} \in X$. On account of this symmetry, we write $D^2 f(x)[a, a]$ for the common value. The operator $D^2 f(\mathbf{x})$ is called *the second derivative of f at $\mathbf{x}$*.

4.4 Derivative formulas

A function $\hat{p}$ defined on the triangle $\mathcal{T}_2$ induces a function p on the triangle $\mathcal{T}_3$ through $\hat{p}(\mathbf{x}) = p\big(\boldsymbol{\lambda}(\mathbf{x})\big)$. When $\hat{p}$ is differentiable, the application of the chain rule of differentiation in conjunction with (4.8) gives:

$$D\hat{p}(\mathbf{x}) = Dp\big(\boldsymbol{\lambda}(\mathbf{x})\big)B, \quad \text{and} \quad D^2\hat{p}(\mathbf{x}) = B^T D^2 p\big(\boldsymbol{\lambda}(\mathbf{x})\big)B, \tag{4.10}$$

where B is defined in (4.6) and B^T is its transpose. By combining the relationships in (4.9) and (4.10) we obtain:

Lemma 1. *Let $\hat{p}$ be a differentiable function defined on the triangle $\mathcal{T}_2$ and let $p\big(\boldsymbol{\lambda}(\mathbf{x})\big) = \hat{p}(\mathbf{x})$. Then we have the following formulas for derivatives of $\hat{p}$ along the edge vectors:*

$$D\hat{p}(\mathbf{x})[\mathbf{e}_1] = \frac{\partial p}{\partial \lambda_3} - \frac{\partial p}{\partial \lambda_2}, \tag{4.11a}$$

$$D\hat{p}(\mathbf{x})[\mathbf{e}_2] = \frac{\partial p}{\partial \lambda_1} - \frac{\partial p}{\partial \lambda_3}, \tag{4.11b}$$

$$D\hat{p}(\mathbf{x})[\mathbf{e}_3] = \frac{\partial p}{\partial \lambda_2} - \frac{\partial p}{\partial \lambda_1}, \tag{4.11c}$$

$$D^2\hat{p}(\mathbf{x})[\mathbf{e}_1, \mathbf{e}_1] = \frac{\partial^2 p}{\partial \lambda_2^2} - 2\frac{\partial^2 p}{\partial \lambda_2 \partial \lambda_3} + \frac{\partial^2 p}{\partial \lambda_3^2}, \tag{4.11d}$$

$$D^2\hat{p}(\mathbf{x})[\mathbf{e}_2, \mathbf{e}_2] = \frac{\partial^2 p}{\partial \lambda_3^2} - 2\frac{\partial^2 p}{\partial \lambda_3 \partial \lambda_1} + \frac{\partial^2 p}{\partial \lambda_1^2}, \tag{4.11e}$$

$$D^2\hat{p}(\mathbf{x})[\mathbf{e}_3, \mathbf{e}_3] = \frac{\partial^2 p}{\partial \lambda_1^2} - 2\frac{\partial^2 p}{\partial \lambda_1 \partial \lambda_2} + \frac{\partial^2 p}{\partial \lambda_2^2}, \tag{4.11f}$$

$$D^2\hat{p}(\mathbf{x})[\mathbf{e}_1, \mathbf{e}_1] = -\frac{\partial^2 p}{\partial \lambda_1 \partial \lambda_2} + \frac{\partial^2 p}{\partial \lambda_2 \partial \lambda_3} + \frac{\partial^2 p}{\partial \lambda_3 \partial \lambda_1} - \frac{\partial^2 p}{\partial \lambda_3^2}, \tag{4.11g}$$

$$D^2\hat{p}(\mathbf{x})[\mathbf{e}_2, \mathbf{e}_2] = -\frac{\partial^2 p}{\partial \lambda_2 \partial \lambda_3} + \frac{\partial^2 p}{\partial \lambda_3 \partial \lambda_1} + \frac{\partial^2 p}{\partial \lambda_1 \partial \lambda_2} - \frac{\partial^2 p}{\partial \lambda_1^2}, \tag{4.11h}$$

$$D^2\hat{p}(\mathbf{x})[\mathbf{e}_3, \mathbf{e}_3] = -\frac{\partial^2 p}{\partial \lambda_3 \partial \lambda_1} + \frac{\partial^2 p}{\partial \lambda_1 \partial \lambda_2} + \frac{\partial^2 p}{\partial \lambda_2 \partial \lambda_3} - \frac{\partial^2 p}{\partial \lambda_2^2}, \tag{4.11i}$$

where the expressions on the right had sides are evaluated at $\boldsymbol{\lambda}(\mathbf{x})$.

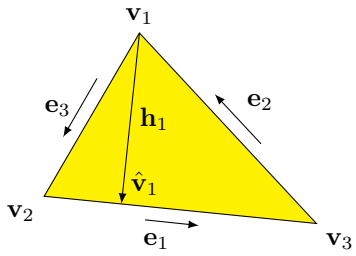


Figure 4.3: The altitude vector $\mathbf{h}_1$ equals $\mathbf{e}_3 + r\mathbf{e}_1$ for some r.

Proof. From (4.9) and (4.10) we have:

$$D\hat{p}(\mathbf{x})[\mathbf{e}_1] = \Big(Dp\big(\boldsymbol{\lambda}(\mathbf{x})\big)B\Big)[\mathbf{e}_1] = \big(Dp\big(\boldsymbol{\lambda}(\mathbf{x})\big)[B\mathbf{e}_1]$$
$$= Dp\big(\boldsymbol{\lambda}(\mathbf{x})\big)\begin{pmatrix}0\\-1\\1\end{pmatrix} = \frac{\partial p}{\partial \lambda_3} - \frac{\partial p}{\partial \lambda_1},$$

which verifies (4.11a). The others are derived in the same way. □

4.5 Derivative normal to the boundary

In a triangle with vertices $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_3$, let $\hat{\mathbf{v}}_1$, $\hat{\mathbf{v}}_2$, $\hat{\mathbf{v}}_3$ be the feet of altitudes dropped from the vertices $\mathbf{v}_1$, $\mathbf{v}_2$, $\mathbf{v}_3$. We define *altitude vectors* of the triangle by:

$$\mathbf{h}_1 = \hat{\mathbf{v}}_1 - \mathbf{v}_1, \qquad \mathbf{h}_2 = \hat{\mathbf{v}}_2 - \mathbf{v}_2, \qquad \mathbf{h}_3 = \hat{\mathbf{v}}_3 - \mathbf{v}_3.$$

The altitude vector $\mathbf{h}_1$ is shown in figure 4.3. The next lemma shows how to express altitude vectors as linear combination of edge vectors.

Lemma 2. *Altitude vectors can be expressed as linear combination of edge vectors as:*

$$\mathbf{h}_1 = \frac{1}{\|\mathbf{e}_1\|^2}\big((\mathbf{e}_3\cdot\mathbf{e}_1)\mathbf{e}_2 - (\mathbf{e}_1\cdot\mathbf{e}_2)\mathbf{e}_3\big), \tag{4.12a}$$

$$\mathbf{h}_2 = \frac{1}{\|\mathbf{e}_2\|^2}\big((\mathbf{e}_1\cdot\mathbf{e}_2)\mathbf{e}_3 - (\mathbf{e}_2\cdot\mathbf{e}_3)\mathbf{e}_1\big), \tag{4.12b}$$

$$\mathbf{h}_3 = \frac{1}{\|\mathbf{e}_3\|^2}\big((\mathbf{e}_2\cdot\mathbf{e}_3)\mathbf{e}_1 - (\mathbf{e}_3\cdot\mathbf{e}_1)\mathbf{e}_2\big). \tag{4.12c}$$

Proof. In Figure 4.3 we see that $\mathbf{h}_1 = \mathbf{e}_3 + r\mathbf{e}_1$ for some r. The orthogonality of $\mathbf{h}_1$ and $\mathbf{e}_1$ requires that $\mathbf{h}_1\cdot\mathbf{e}_1 = (\mathbf{e}_3 + r\mathbf{e}_1)\cdot\mathbf{e}_1 = 0$ whence $r = -(\mathbf{e}_3\cdot\mathbf{e}_1)/\|\mathbf{e}_1\|^2$.

We note that

$$1 - r = 1 + \frac{\mathbf{e}_3 \cdot \mathbf{e}_1}{\|\mathbf{e}_1\|^2} = \frac{1}{\|\mathbf{e}_1\|^2}(\mathbf{e}_1 \cdot \mathbf{e}_1 + \mathbf{e}_3 \cdot \mathbf{e}_1) = \frac{1}{\|\mathbf{e}_1\|^2}\mathbf{e}_1 \cdot (\mathbf{e}_1 + \mathbf{e}_3) = -\frac{\mathbf{e}_1 \cdot \mathbf{e}_2}{\|\mathbf{e}_1\|^2}$$

because $\mathbf{e}_1 + \mathbf{e}_2 + \mathbf{e}_3 = \mathbf{0}$. Therefore:

$$\mathbf{h}_1 = \mathbf{e}_3 + r\mathbf{e}_1 = \mathbf{e}_3 - r(\mathbf{e}_2 + \mathbf{e}_3) = -r\mathbf{e}_2 + (1 - r)\mathbf{e}_3 = \frac{\mathbf{e}_3 \cdot \mathbf{e}_1}{\|\mathbf{e}_1\|^2}\mathbf{e}_2 - \frac{\mathbf{e}_1 \cdot \mathbf{e}_2}{\|\mathbf{e}_1\|^2}\mathbf{e}_3,$$

which proves (4.12a). The others are obtained by cyclic rotation of symbols.

□

Remark. The length of the altitude vector $\mathbf{h}_1$ may be obtained by computing the dot product $\mathbf{h}_1 \cdot \mathbf{h}_1$ of the expression for $\mathbf{h}_1$ in (4.12a). However we obtain a more compact result by recalling that $\mathbf{h}_1 = \mathbf{e}_3 + r\mathbf{e}_1$, therefore:

$$\|\mathbf{h}_1\|^2 = \|\mathbf{e}_3\|^2 + r^2\|\mathbf{e}_1\|^2 + 2r\mathbf{e}_3 \cdot \mathbf{e}_1,$$

then substituting $r = -(\mathbf{e}_3 \cdot \mathbf{e}_1)/\|\mathbf{e}_1\|^2$, which leads to the simplified expression:

$$\|\mathbf{h}_1\|^2 = \frac{1}{\|\mathbf{e}_1\|^2}\big(\|\mathbf{e}_3\|^2\|\mathbf{e}_1\|^2 - (\mathbf{e}_3 \cdot \mathbf{e}_1)^2\big). \tag{4.13}$$

With hindsight, this could have been obtained by a direct application of the Pythagorean theorem to the triagnle with vertices $\mathbf{v}_1$, $\hat{\mathbf{v}}_1$, $\mathbf{v}_3$. Formulas for $\|\mathbf{h}_2\|$ and $\|\mathbf{h}_2\|$ are obtained by cyclic rotation of the indices in (4.13).

Lemma 3. *Let $\hat{p}$ be as in Lemma 1. Then we have the following formula for the derivative of $\hat{p}$ along the altitude vector $\mathbf{h}_1$:*

$$\|\mathbf{e}_1\|^2 D\hat{p}(x)[\mathbf{h}_1] = -\mathbf{e}_1 \cdot \Big(\frac{\partial p}{\partial \lambda_1}\,\mathbf{e}_1 + \frac{\partial p}{\partial \lambda_2}\,\mathbf{e}_2 + \frac{\partial p}{\partial \lambda_3}\,\mathbf{e}_3\Big). \tag{4.14}$$

Formulas for derivatives along the altitude vectors $\mathbf{h}_2$ and $\mathbf{h}_3$ are obtained by making the obvious changes.

Proof. By combining (4.9) and (4.12) we obtain:

$$\begin{aligned}\|\mathbf{e}_1\|^2 B\,\mathbf{h}_1 &= (\mathbf{e}_3 \cdot \mathbf{e}_1)\begin{pmatrix}1\\0\\-1\end{pmatrix} - (\mathbf{e}_1 \cdot \mathbf{e}_2)\begin{pmatrix}-1\\1\\0\end{pmatrix}\\ &= \begin{pmatrix}(\mathbf{e}_3 + \mathbf{e}_2) \cdot \mathbf{e}_1\\ -\mathbf{e}_1 \cdot \mathbf{e}_2\\ -\mathbf{e}_3 \cdot \mathbf{e}_1\end{pmatrix} = \begin{pmatrix}-\mathbf{e}_1 \cdot \mathbf{e}_1\\ -\mathbf{e}_1 \cdot \mathbf{e}_2\\ -\mathbf{e}_1 \cdot \mathbf{e}_3\end{pmatrix}.\end{aligned}$$

Then from (4.10):

$$\begin{aligned}\|\mathbf{e}_1\|^2 D\hat{p}(x)[\mathbf{h}_1] &= \|\mathbf{e}_1\|^2 Dp\big(\boldsymbol{\lambda}(\mathbf{x})\big)B\,\mathbf{h}_1 = Dp\big(\boldsymbol{\lambda}(\mathbf{x})\big)\begin{pmatrix}-\mathbf{e}_1 \cdot \mathbf{e}_1\\ -\mathbf{e}_1 \cdot \mathbf{e}_2\\ -\mathbf{e}_1 \cdot \mathbf{e}_3\end{pmatrix}\\ &= (-\mathbf{e}_1 \cdot \mathbf{e}_1)\frac{\partial p}{\partial \lambda_1} + (-\mathbf{e}_1 \cdot \mathbf{e}_2)\frac{\partial p}{\partial \lambda_2} + (-\mathbf{e}_1 \cdot \mathbf{e}_3)\frac{\partial p}{\partial \lambda_3},\end{aligned}$$

which is equivalent to (4.14). □

4.6 The Laplacian

Lemma 4. *Let $\hat{p}$ be as in Lemma 1. Then we have the following formula for the Laplacian of $\hat{p}$:*

$$\Delta p(\mathbf{x}) = \frac{1}{\mathcal{J}^2}\bigg(\|\mathbf{e}_1\|^2\frac{\partial^2 p}{\partial\lambda_1^2} + \|\mathbf{e}_2\|^2\frac{\partial^2 p}{\partial\lambda_2^2} + \|\mathbf{e}_3\|^2\frac{\partial^2 p}{\partial\lambda_3^2} + 2\mathbf{e}_1\cdot\mathbf{e}_2\frac{\partial^2 p}{\partial\lambda_1\partial\lambda_2} + 2\mathbf{e}_2\cdot\mathbf{e}_3\frac{\partial^2 p}{\partial\lambda_2\partial\lambda_3} + 2\mathbf{e}_3\cdot\mathbf{e}_1\frac{\partial^2 p}{\partial\lambda_3\partial\lambda_1}\bigg), \quad (4.15)$$

where $\mathcal{J}$ is defined in (4.4).

Proof. The differentiation formula in (4.10) gives the second derivative of $\hat{p}$ in terms of the second derivatives of p. The Laplacian of $\hat{p}$, $\Delta\hat{p}$, is the trace of $D^2\hat{p}$, therefore:

$$\Delta p(\mathbf{x}) = \operatorname{tr} D^2\hat{p}(\mathbf{x}) = \operatorname{tr}\Big(B^T D^2 p\big(\boldsymbol{\lambda}(\mathbf{x})\big)B\Big).$$

To simplify the result, let us note that $\operatorname{tr}\big(U^TV\big) = \operatorname{tr}\big(VU^T\big)$ for arbitrary (not necessarily square) $m\times n$ matrices U and V. Therefore for any $m\times n$ matrix B and $n\times n$ (square) matrix P we have:

$$\operatorname{tr} B^TPB = \operatorname{tr} B^T(PB) = \operatorname{tr}(PB)B^T = \operatorname{tr} P(BB^T) = \operatorname{tr} PF,$$

where we have let $F = BB^T$. Corresponding to the matrix B in (4.6) we have:

$$F = \frac{1}{\mathcal{J}^2}\begin{pmatrix}-e_{12} & e_{11}\\ -e_{22} & e_{21}\\ -e_{32} & e_{31}\end{pmatrix}\begin{pmatrix}-e_{12} & -e_{22} & -e_{32}\\ e_{11} & e_{21} & e_{31}\end{pmatrix} = \frac{1}{\mathcal{J}^2}\begin{pmatrix}\|\mathbf{e}_1\|^2 & \mathbf{e}_1\cdot\mathbf{e}_2 & \mathbf{e}_3\cdot\mathbf{e}_1\\ \mathbf{e}_1\cdot\mathbf{e}_2 & \|\mathbf{e}_2\|^2 & \mathbf{e}_2\cdot\mathbf{e}_3\\ \mathbf{e}_3\cdot\mathbf{e}_1 & \mathbf{e}_2\cdot\mathbf{e}_3 & \|\mathbf{e}_3\|^2\end{pmatrix}.$$

Therefore the equation for the Laplacian takes the form:

$$\Delta\hat{p}(\mathbf{x}) = \operatorname{tr} D^2\hat{p}(\mathbf{x}) = \operatorname{tr}\Big(D^2 p\big(\boldsymbol{\lambda}(\mathbf{x})\big)F\Big),$$

which expands to to the expressions given in (4.15). □

4.7 Integration over a triangle

Consider the triangle $\mathcal{T}_2$ in Figure 4.1. An arbitrary point $\mathbf{x}$ in $\mathcal{T}_2$ may be arrived at by moving from the vertex $\mathbf{v}_1$ in the direction of the edge vector $\mathbf{e}_3$ by a fraction, say t_1, of that edge length, then change course and move parallel to the edge vector $\mathbf{e}_1$ by a fraction, say t_2, of that edge length. Any point in the triangle may be reached this way with some $0\leqslant t_1\leqslant 1$ and $0\leqslant t_2\leqslant t_1$. This observation gives a two-parameter representation of the triangle in parameters t_1 and t_2:

$$\mathbf{x} = \mathbf{v}_1 + t_1\mathbf{e}_3 + t_2\mathbf{e}_1, \quad 0\leqslant t_2\leqslant t_1\leqslant 1.$$

At the same time, we know that $\mathbf{x}$ may be expressed in barycentric coordinates as in (4.3). Matching the two expressions for $\mathbf{x}$ we obtain a mapping from the (t_1, t_2) to $(\lambda_1, \lambda_2, \lambda_3)$ coordinates:

$$\lambda_1 = 1 - t_1, \quad \lambda_2 = t_1 - t_2, \quad \lambda_1 = t_2. \tag{4.16}$$

Consider the problem of integration of a function $\hat{p}$ defined on the triangle $\mathcal{T}_2$. As before, define the function p on $\mathcal{T}_3$ through $p\big(\boldsymbol{\lambda}(\mathbf{x})\big) = \hat{p}(\mathbf{x})$. Then we have the change of variables formula:

$$\int_{\mathcal{T}_2} \hat{p}(\mathbf{x})\, d\mathbf{x} = \mathcal{J} \int_0^1 \int_0^{t_1} p(1 - t_1, t_1 - t_2, t_2)\, dt_2\, dt_1, \tag{4.17}$$

where the multiplier $\mathcal{J}$ is the Jacobian of the mapping from the (t_1, t_2) domain to the (x_1, x_2) domain. Since the mapping is linear, the Jacobian is constant therefore it has been taken out of the integration. The double integral on the right hand is on a triangle of area $1/2$, therefore $\mathcal{J} = 2\mathcal{A}$ where $\mathcal{A}$ is the area of the triangle $\mathcal{T}_2$. Thus the factor $\mathcal{J}$ here coincides with that defined in (4.4).

4.8 The Argyris shape functions

Argyris elements are quintic polynomials defined on triangles designed to fit together as C^1 functions on a triangulated domain. In a sense, Argyris elements generalize to two dimensions what cubic splines do on interpolating over intervals in one dimension.

A quintic polynomial $\hat{p}(x, y)$ in two variables has 21 degrees of freedom thus a unique such polynomial is obtained by specifying 21 constraints. In an Argyris element the 21 constraints are:

- values of $\hat{p}$ at the triangle's vertices (3 constraints)
- the values of $\partial\hat{p}/\partial x$ and $\partial\hat{p}/\partial y$ at the vertices (6 constraints)
- the values of $\partial^2\hat{p}/\partial x^2$, $\partial^2\hat{p}/\partial x\partial y$ and $\partial^2\hat{p}/\partial y^2$ at the vertices (9 constraints)
- the values of derivatives $\partial\hat{p}/\partial\nu$ in the direction of unit outward normal vector $\boldsymbol{\nu}$, at midpoints of the triangle's edges (3 constraints).

We refer to the 21 data items that specify the constraints as *the Argyris data*.

There is some flexibility in specifying the first and second derivatives at the vertices. Instead of specifying $\partial\hat{p}/\partial x$ and $\partial\hat{p}/\partial y$ at a vertex, we may specify directional derivatives in the direction of the two edges that meet at the vertex. These are equivalent, because we can convert from one set to the other through a change of basis in $\mathbf{R}^2$. Similarly, instead of specifying $\partial^2\hat{p}/\partial x^2$, $\partial^2\hat{p}/\partial x\partial y$

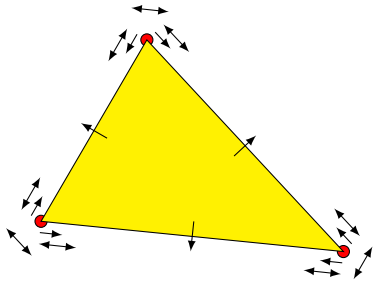


Figure 4.4: The Argyris triangular element is specified by the values of a function and its first and second derivatives at vertices and the values of derivatives normal to the edges at their midpoints—a total of 21 conditions. In this schematic drawing, single and double arrows indicate first and second derivatives.

and $\partial^2\hat{p}/\partial y^2$, we may specify second derivatives in the directions parallel to the triangle's three edges. Because of this equivalence, we refer to the 21 data items as *the Argyris data* in either case. Figure 4.4 shows a schematic representation of the 21 constraints.

Expressing partial derivatives in the x and y directions has the practical advantage that ultimately in our finite element formulation, it is those derivatives that constitute the problem's main unknowns. The disadvantage is that the conditions are not intrinsic to the triangle; they depend on how the triangle is situated relative to the coordinate axes. Consequently, the expressions obtained for the polynomial basis tend to be quite messy and inelegant.

Expressing partial derivatives as derivatives along the triangle's edges has the advantage that it gives an intrinsic description of the element and the expressions obtained for the polynomial basis are much simpler. The disadvantage is that elements expressed this way need to go through a interpretation stage to connect to x and y derivatives that are ultimately needed.

In our solver we use the intrinsic description of the Argyris element. The interpretation stage noted above results in some complexity in the code but it appears that because of the simpler expressions for the basis functions, the code is more efficient overall.

Thus let us assume that we are given the following Argyris data:

- Values of $\hat{p}$ at the vertices:

$$\hat{p}(\mathbf{v}_1), \quad \hat{p}(\mathbf{v}_2), \quad \hat{p}(\mathbf{v}_3). \tag{4.18a}$$

- Values of first derivatives at each vertex applied to the edge vectors of the

edges that meet at that vertex:

$$D\hat{p}(\mathbf{v}_1)[\mathbf{e}_2], \quad D\hat{p}(\mathbf{v}_1)[\mathbf{e}_3], \tag{4.18b}$$
$$D\hat{p}(\mathbf{v}_2)[\mathbf{e}_3], \quad D\hat{p}(\mathbf{v}_2)[\mathbf{e}_1], \tag{4.18c}$$
$$D\hat{p}(\mathbf{v}_3)[\mathbf{e}_1], \quad D\hat{p}(\mathbf{v}_3)[\mathbf{e}_2]. \tag{4.18d}$$

- Values of second derivatives at each vertex applied to edge vectors of the three edges:

$$D^2\hat{p}(\mathbf{v}_1)[\mathbf{e}_1,\mathbf{e}_1], \quad D^2\hat{p}(\mathbf{v}_1)[\mathbf{e}_2,\mathbf{e}_2], \quad D^2\hat{p}(\mathbf{v}_1)[\mathbf{e}_3,\mathbf{e}_3], \tag{4.18e}$$
$$D^2\hat{p}(\mathbf{v}_2)[\mathbf{e}_1,\mathbf{e}_1], \quad D^2\hat{p}(\mathbf{v}_2)[\mathbf{e}_2,\mathbf{e}_2], \quad D^2\hat{p}(\mathbf{v}_2)[\mathbf{e}_3,\mathbf{e}_3], \tag{4.18f}$$
$$D^2\hat{p}(\mathbf{v}_3)[\mathbf{e}_1,\mathbf{e}_1], \quad D^2\hat{p}(\mathbf{v}_3)[\mathbf{e}_2,\mathbf{e}_2], \quad D^2\hat{p}(\mathbf{v}_3)[\mathbf{e}_3,\mathbf{e}_3]. \tag{4.18g}$$

- Values of derivatives at midpoints of the each edge, applied to the altitude vector incident on that edge:

$$D\hat{p}(\mathbf{m}_1)[\mathbf{h}_1], \quad D\hat{p}(\mathbf{m}_2)[\mathbf{h}_2], \quad D\hat{p}(\mathbf{m}_3)[\mathbf{h}_3]. \tag{4.18h}$$

The 21 conditions in equations (4.18) determine a unique fifth degree polynomial in two variables on the triangle. For the purpose of finite element computations, however, it is more convenient to express the polynomial in barycentric coordinates. We apply the transformation rules in lemmas 1 and 3 to convert the data in the equation set (4.18) from Cartesian to barycentric coordinates.

To manage the complexity, instead of constructing a single fifth degree polynomial that satisfies the 21 conditions in (4.18), we construct a basis consisting 21 fifth degree polynomials, P_i, $i = 1, 2, \ldots, 21$, called the *Argyris shape functions*, and represent $\hat{p}$ as a linear combination of the P_i function with factors given in the Argyris data in equations (4.18). The ith polynomial, P_i, will produce 0 for all conditions in (4.18) except for the ith conditions, for which it will produce 1.

The construction of the 21 basis elements is quite tedious. We call on *Maple*—a software for symbolic computation—to help with the calculations. See Appendix A for the details of the the construction procedure.

4.9 The Argyris basis functions

The Argyris shape functions P_1 through P_{21} form a basis for the space of fifth degree polynomials on the triangle $\mathcal{T}_2$. Any fifth degree polynomial $\hat{p}$ may be expressed as a linear combination of them where the coefficients are the Argyris

data:

$$\begin{aligned}\hat{p}(\mathbf{x}) = \sum_{i=1}^{3} \quad & \hat{p}(\mathbf{v}_i)\, P_{6(i-1)+1}\big(\boldsymbol{\lambda}(\mathbf{x})\big) \\ & +D\hat{p}(\mathbf{v}_i)[\mathbf{e}_k]\, P_{6(i-1)+2}\big(\boldsymbol{\lambda}(\mathbf{x})\big) \\ & -D\hat{p}(\mathbf{v}_i)[\mathbf{e}_j]\, P_{6(i-1)+3}\big(\boldsymbol{\lambda}(\mathbf{x})\big) \\ & +D^2\hat{p}(\mathbf{v}_i)[\mathbf{e}_k,\mathbf{e}_k]\, P_{6(i-1)+4}\big(\boldsymbol{\lambda}(\mathbf{x})\big) \\ & +D^2\hat{p}(\mathbf{v}_i)[\mathbf{e}_j,\mathbf{e}_j]\, P_{6(i-1)+5}\big(\boldsymbol{\lambda}(\mathbf{x})\big) \\ & +D^2\hat{p}(\mathbf{v}_i)[\mathbf{e}_i,\mathbf{e}_i]\, P_{6(i-1)+6}\big(\boldsymbol{\lambda}(\mathbf{x})\big) \\ + \sum_{i=1}^{3} & D\hat{p}(\mathbf{m}_i)[\mathbf{h}_i]\, P_{18+i}\big(\boldsymbol{\lambda}(\mathbf{x})\big), \end{aligned} \tag{4.19}$$

where $j = (i \mod 3) + 1$ and $k = (i+1 \mod 3) + 1$.

Remark. The awkward appearance of the indices is due to the 1-based indexing which is the common practice in mathematics. If we change to 0-based indexing, so that vertices, edges, altitudes and midpoints are $\mathbf{v}_i$, $\mathbf{e}_i$, $\mathbf{h}_i$ and $\mathbf{m}_i$, $i = 0, 1, 2$, and the shape functions are P_i, $i = 0, 1, \ldots, 20$, then the expression for $\hat{p}(\mathbf{x})$ takes on a somewhat more transparent form:

$$\begin{aligned}\hat{p}(\mathbf{x}) = \sum_{i=0}^{2} \quad & \hat{p}(\mathbf{v}_i)\, P_{6i}\big(\boldsymbol{\lambda}(\mathbf{x})\big) \\ & +D\hat{p}(\mathbf{v}_i)[\mathbf{e}_k]\, P_{6i+1}\big(\boldsymbol{\lambda}(\mathbf{x})\big) \\ & -D\hat{p}(\mathbf{v}_i)[\mathbf{e}_j]\, P_{6i+2}\big(\boldsymbol{\lambda}(\mathbf{x})\big) \\ & +D^2\hat{p}(\mathbf{v}_i)[\mathbf{e}_k,\mathbf{e}_k]\, P_{6i+3}\big(\boldsymbol{\lambda}(\mathbf{x})\big) \\ & +D^2\hat{p}(\mathbf{v}_i)[\mathbf{e}_j,\mathbf{e}_j]\, P_{6i+4}\big(\boldsymbol{\lambda}(\mathbf{x})\big) \\ & +D^2\hat{p}(\mathbf{v}_i)[\mathbf{e}_i,\mathbf{e}_i]\, P_{6i+5}\big(\boldsymbol{\lambda}(\mathbf{x})\big) \\ + \sum_{i=0}^{2} & D\hat{p}(\mathbf{m}_i)[\mathbf{h}_i]\, P_{18+i}\big(\boldsymbol{\lambda}(\mathbf{x})\big), \end{aligned}$$

where $j = i+1 \mod 3$ and $k = i+2 \mod 3$. In our computer program we use 0-based indexing (which is the natural scheme in C) but in the documentation we will continue using the 1-based scheme to avoid unconventional mathematical notation for vertices as in $\mathbf{v}_0 = \langle v_{00}, v_{01} \rangle$.

Going back to (4.19), we expand the differentiation operators into their componets in the Cartesian coordinates. For example:

$$\begin{aligned} D\hat{p}(\mathbf{v}_i)[\mathbf{e}_k] &= \Big(e_{k1}\frac{\partial \hat{p}}{\partial x_1} + e_{k2}\frac{\partial \hat{p}}{\partial x_2} \Big)\Big|_{\mathbf{v}_i}, \\ D^2\hat{p}(\mathbf{v}_i)[\mathbf{e}_k,\mathbf{e}_k] &= \Big(e_{k1}^2\frac{\partial^2 \hat{p}}{\partial x_1^2} + 2e_{k1}e_{k2}\frac{\partial^2 \hat{p}}{\partial x_1 \partial x_2} + e_{k2}^2\frac{\partial^2 \hat{p}}{\partial x_2^2} \Big)\Big|_{\mathbf{v}_i}. \end{aligned}$$

We view the partial derivatives, such as $\partial\hat{p}/\partial x_i$ and $\partial^2\hat{p}/\partial x_i\partial x_j$, that appear in these expansions, as *global attributes of the nodes of the mesh* rather than local properties of the particular element. If several elements have a common vertex at a certain node, the elements share the values for the partial derivatives at that node.

Such "globalization" of partial derivatives to the mesh's nodes is straightforward because there is an unambiguous meaning to the x_1 and and x_2 directions at each node. The situation for the mid-edge derivatives, $D\hat{p}(\mathbf{m}_i)[\mathbf{h}_i]$, is more complicated. An element's outward normal at an edge's midpoint does not generalize to a well-defined direction for that edge from the mesh's global perspective because two elements that share an edge have outward normals pointing in opposite directions. In the left diagram in Figure 4.5 we see a two adjacent elements sharing an edge AB. On the right we see an exploed view of the two elements and the individual outward normals. We resolve the conflict between the opposing outward normals by resorting to the global node numbering of the mesh. Our construction of the mesh (see Chapter 2) assigns a "first node" and a "second node" to each edge of the mesh. This gives an unambiguous direction to an edge, independent of the surrounding elements. We rotate the edge counterclockwise by 90 degrees thus obtaining a well-defined normal direction. *We define the mid-edge unit normal to point in that direction.*

The right digagram in Figure 4.5 depicts a vector $\vec{AB}$ presumably pointing from the "first node" to the "second node", and the resulting mid-edge unit normal, $\boldsymbol{\nu}$. Thus if the altitude $\mathbf{h}_i$ dropped from the vertex $\mathbf{v}_i$ points in the direction of $\boldsymbol{\nu}$, then $D\hat{p}(\mathbf{m}_i)[\mathbf{h}_i]$ equals $\|\mathbf{h}_i\|\partial\hat{p}/\partial\boldsymbol{\nu}$. If the altitude $\mathbf{h}_i$ dropped from the vertex $\mathbf{v}_i$ points in the opposite direction of $\boldsymbol{\nu}$, then $D\hat{p}(\mathbf{m}_i)[\mathbf{h}_i]$ equals $-\|\mathbf{h}_i\|\partial\hat{p}/\partial\boldsymbol{\nu}$. In summary:

$$D\hat{p}(\mathbf{m}_i)[\mathbf{h}_i] = \text{sign}(\mathbf{h}_i \cdot \boldsymbol{\nu})\, \|\mathbf{h}_i\| \frac{\partial\hat{p}}{\partial\boldsymbol{\nu}}. \tag{4.20}$$

The length of the altitude vector is given in (4.13).

We replace the differentiation operators by partial derivatives in (4.19) and

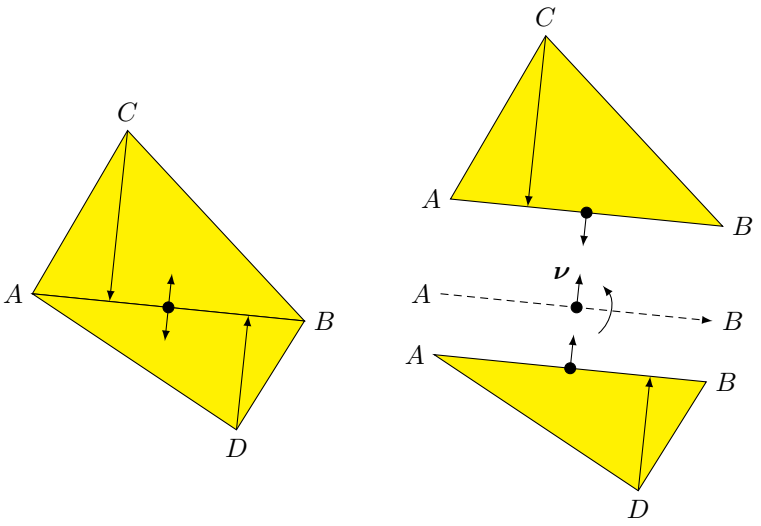


Figure 4.5: The diagram on the left shows two adjacent elements, ABC and ADB. The outward normal vectors on the common edge AB point in opposite directions. On the right we see an exploded view of he elements. The edge AB is viewed as a vector pointing from its "first node" to its "second node", these being attributes that are defined at the creation of the mesh. The edge unit normal, $\boldsymbol{\nu}$, is obtained by rotating the vector $\overrightarrow{AB}$ by 90 counterclockwise degrees then normalizing to unit length.

regroup:

$$\begin{aligned}\hat{p}(\mathbf{x}) = \sum_{i=1}^{3} \quad & P_{6(i-1)+1}\big(\boldsymbol{\lambda}(\mathbf{x})\big)\,\hat{p}(\mathbf{v}_i) \\ & + \Big(e_{k1}P_{6(i-1)+2}\big(\boldsymbol{\lambda}(\mathbf{x})\big) - e_{j1}P_{6(i-1)+3}\big(\boldsymbol{\lambda}(\mathbf{x})\big)\Big)\Big(\frac{\partial \hat{p}}{\partial x_1}\Big)\Big|_{\mathbf{v}_i} \\ & + \Big(e_{k2}P_{6(i-1)+2}\big(\boldsymbol{\lambda}(\mathbf{x})\big) - e_{j2}P_{6(i-1)+3}\big(\boldsymbol{\lambda}(\mathbf{x})\big)\Big)\Big(\frac{\partial \hat{p}}{\partial x_2}\Big)\Big|_{\mathbf{v}_i} \\ & + \Big(e_{k1}^2P_{6(i-1)+4}\big(\boldsymbol{\lambda}(\mathbf{x})\big) + e_{i1}^2P_{6(i-1)+5}\big(\boldsymbol{\lambda}(\mathbf{x})\big) \\ & \qquad + e_{j1}^2P_{6(i-1)+6}\big(\boldsymbol{\lambda}(\mathbf{x})\big)\Big)\Big(\frac{\partial^2 \hat{p}}{\partial x_1^2}\Big)\Big|_{\mathbf{v}_i} \\ & + \Big(e_{k2}^2P_{6(i-1)+4}\big(\boldsymbol{\lambda}(\mathbf{x})\big) + e_{i2}^2P_{6(i-1)+5}\big(\boldsymbol{\lambda}(\mathbf{x})\big) \\ & \qquad + e_{j2}^2P_{6(i-1)+6}\big(\boldsymbol{\lambda}(\mathbf{x})\big)\Big)\Big(\frac{\partial^2 \hat{p}}{\partial x_2^2}\Big)\Big|_{\mathbf{v}_i} \\ & + 2\Big(e_{k1}e_{k2}P_{6(i-1)+4}\big(\boldsymbol{\lambda}(\mathbf{x})\big) + e_{i1}e_{i2}P_{6(i-1)+5}\big(\boldsymbol{\lambda}(\mathbf{x})\big) \\ & \qquad + e_{j1}e_{j2}P_{6(i-1)+6}\big(\boldsymbol{\lambda}(\mathbf{x})\big)\Big)\Big(\frac{\partial^2 \hat{p}}{\partial x_1 \partial x_2}\Big)\Big|_{\mathbf{v}_i} \\ & + \sum_{i=1}^{3} \operatorname{sign}(\mathbf{h}_i \cdot \boldsymbol{\nu}_i) P_{18+i}\big(\boldsymbol{\lambda}(\mathbf{x})\big)\|\mathbf{h}_i\|\Big(\frac{\partial \hat{p}}{\partial \boldsymbol{\nu}_i}\Big)\Big|_{\mathbf{m}_i}, \end{aligned} \tag{4.21}$$

where $j = (i \mod 3) + 1$ and $k = (i+1 \mod 3) + 1$, as before, and $\boldsymbol{\nu}_i$ is the unit normal to the edge, as described earlier.

Equation (4.21) embodies the essence of finite element computations with the Argyris element—it expresses a function $\hat{p}(\mathbf{x})$ in terms of Argyris shape functions P_1 through P_{21} and coefficients that are the Argyris data.

It should be noted that in (4.21) $\hat{p}$ is not a straight sum of the shape functions P_i but of a certain linear combinations of them. This leads us to introduce *the 21 basis functions*:

$$\begin{aligned} Q_{6(i-1)+1}(\boldsymbol{\lambda}) &= P_{6(i-1)+1}(\boldsymbol{\lambda}), \\ Q_{6(i-1)+2}(\boldsymbol{\lambda}) &= e_{k1}P_{6(i-1)+2}(\boldsymbol{\lambda}) - e_{j1}P_{6(i-1)+3}(\boldsymbol{\lambda}), \\ Q_{6(i-1)+3}(\boldsymbol{\lambda}) &= e_{k2}P_{6(i-1)+2}(\boldsymbol{\lambda}) - e_{j2}P_{6(i-1)+3}(\boldsymbol{\lambda}), \\ Q_{6(i-1)+4}(\boldsymbol{\lambda}) &= e_{k1}^2P_{6(i-1)+4}(\boldsymbol{\lambda}) + e_{i1}^2P_{6(i-1)+5}(\boldsymbol{\lambda}) + e_{j1}^2P_{6(i-1)+6}(\boldsymbol{\lambda}), \\ Q_{6(i-1)+5}(\boldsymbol{\lambda}) &= e_{k2}^2P_{6(i-1)+4}(\boldsymbol{\lambda}) + e_{i2}^2P_{6(i-1)+5}(\boldsymbol{\lambda}) + e_{j2}^2P_{6(i-1)+6}(\boldsymbol{\lambda}), \\ Q_{6(i-1)+6}(\boldsymbol{\lambda}) &= 2\big(e_{k1}e_{k2}P_{6(i-1)+4}(\boldsymbol{\lambda}) + e_{i1}e_{i2}P_{6(i-1)+5}(\boldsymbol{\lambda}) \\ &\qquad + e_{j1}e_{j2}P_{6(i-1)+6}(\boldsymbol{\lambda})\big), \\ Q_{18+i}(\boldsymbol{\lambda}) &= \operatorname{sign}(\mathbf{h}_i \cdot \boldsymbol{\nu}_i)\|\mathbf{h}_i\|P_{19+i}(\boldsymbol{\lambda}), \end{aligned} \tag{4.22}$$

where $i = 1, 2, 3$ and $j = (i \mod 3) + 1$ and $k = (i+1 \mod 3) + 1$, as before.

Combining the representation in (4.21) with the definitions in (4.22), we express $\hat{p}$ as a linear combination of the basis functions:

$$\begin{aligned}\hat{p}(\mathbf{x}) = \sum_{i=1}^{3} \quad & Q_{6(i-1)+1}\big(\boldsymbol{\lambda}(\mathbf{x})\big)\, \hat{p}(\mathbf{v}_i) \\ & + Q_{6(i-1)+2}\big(\boldsymbol{\lambda}(\mathbf{x})\big) \left(\frac{\partial \hat{p}}{\partial x_1}\right)\Big|_{\mathbf{v}_i} \\ & + Q_{6(i-1)+3}\big(\boldsymbol{\lambda}(\mathbf{x})\big) \left(\frac{\partial \hat{p}}{\partial x_2}\right)\Big|_{\mathbf{v}_i} \\ & + Q_{6(i-1)+4}\big(\boldsymbol{\lambda}(\mathbf{x})\big) \left(\frac{\partial^2 \hat{p}}{\partial x_1^2}\right)\Big|_{\mathbf{v}_i} \\ & + Q_{6(i-1)+5}\big(\boldsymbol{\lambda}(\mathbf{x})\big) \left(\frac{\partial^2 \hat{p}}{\partial x_2^2}\right)\Big|_{\mathbf{v}_i} \\ & + Q_{6(i-1)+6}\big(\boldsymbol{\lambda}(\mathbf{x})\big) \left(\frac{\partial^2 \hat{p}}{\partial x_1 \partial x_2}\right)\Big|_{\mathbf{v}_i} \\ + \sum_{i=1}^{3} & Q_{18+i}\big(\boldsymbol{\lambda}(\mathbf{x})\big) \Big(\frac{\partial \hat{p}}{\partial \boldsymbol{\nu}_i}\Big)\Big|_{\mathbf{m}_i}. \end{aligned} \tag{4.23}$$

In our computations, we will need expressions for the first and second derivatives of $\hat{p}$. These are obtained by applying the general differentiation formulas in (4.10) and definition of B in (4.6) to each of the 21 terms in (4.23).

4.10 Computing the basis functions

The details of the construction of the Argyris shape functions P_i, $i = 1, \ldots, 21$, are given in Appendix A. The user interface to the shape functions is provided through the function `get_shape_data()` whose prototype is given in ⟨*prototype of get_shape_data* (never defined)⟩. It receives a pointer to an element, a pointer to a an array of barycentric coordinates, and a pointer to a pre-allocated array, `Pvals`, in which it calculates and stores the values of P_i and its derivatives at the specified points.

As we saw in Section 4.9, shape functions enter the calculations in the form of a certain combinations which we called Q_i. We introduce a function, `get_basis_data()`, which computes the values of the functions Q_i and their first and second derivatives (with respect to Cartesian coordinates!) for a given element. It receives an array `Point3d *points` of barycentric coordinates of prescribed points and a three-indexed array `double ***Q` in which the computed values are stored. The value of the dth derivative of Q_i at point m is stored in `Q[m][i][d]`. The d in 'dth derivative is one of the six mnemonic values:

⟨*x derivative codes* 61⟩≡ (65c)

```
enum x_derivs { d00, d10, d01, d20, d11, d02 };
```

where `d`ij indicates the $\partial^{i+j}/\partial x_1^i \partial x_2^j$ derivative of a function in the Cartesian (x_1, x_2) coordinates.

Here is the outline of `get_basis_data()`:

⟨*function get_basis_data()* 62a⟩≡ (65b)
```
void get_basis_data(Elem *elem, Point3d *points, int npoints, double ***Q)
{
    ⟨declarations for function get_basis_data() 62b⟩
    ⟨allocate memory for the Pvals and Qvals arrays 63a⟩
    ⟨calculate edge vectors 63b⟩
    ⟨compute matrix B 63c⟩
    ⟨call get_shape_data() to fill the Pvals array 63d⟩
    ⟨for each point... 64a⟩
        ⟨evaluate the 21 basis functions 64b⟩
        ⟨evaluate the x, y, xx, xy, yyy derivatives of the basis functions 64c⟩
    ⟨free memory for the Pvals and Qvals arrays 65a⟩
}
```

This is quite a large function. Although it is possible to break it into smaller functions, the large amount of shared data among its components would make the splitting somewhat unnatural. We describe the various parts of this function in the following subsections.

4.10.1 The declarations section

The function begins with:

⟨*declarations for function get_basis_data()* 62b⟩≡ (62a)
```
double ***Pvals;
double **Qvals;
enum which_deriv idx[3] = { D100, D010, D001 };
enum which_deriv idx2[3][3] = {
    { D200, D110, D101 },
    { D110, D020, D011 },
    { D101, D011, D002 },
};
double B[3][2];
Vec2d edge[3];
double J;
int i, m;
```

The derivative codes `D`xxx defined in ⟨*derivative codes* (never defined)⟩ are used as mnemonic devices to access any of the 10 derivatives of up to order two of a function expressed in barycentric coordinates. The `idx[]` and `idx2[]` arrays defined above establish a a mapping between raw indices and their mnemonic counterparts. For instance `idx2[2][3]` evaluates to `D011` indicating that the mnemonic for the derivative $\partial/\partial\lambda_2\partial\lambda_3$ is `D011`.

4.10.2 Setting up the working arrays

We allocate memory for the array `Pvals` to store the values of the 21 shape functions P_i and their derivatives in it. The value of the derivative of index j of P_i at point m will be stored in `Pvals[m][i][j]`. The index j of the derivatives goes from 0 to 9, corresponding to one zeroth derivative, three first derivatives and six second derivatives.

⟨*allocate memory for the Pvals and Qvals arrays* 63a⟩≡ (62a)
```
MAKE_3ARRAY(Pvals, npoints, 21, 10);
MAKE_2ARRAY(Qvals, 21, 10);
```

4.10.3 The main computational loop

Then we compute the components of the edge vectors for each edge:

⟨*calculate edge vectors* 63b⟩≡ (62a)
```
for (i = 0; i < 3; i++) {
    int j = (i + 1) % 3;
    int k = (i + 2) % 3;
    edge[i].x = elem->n[k]->x - elem->n[j]->x;
    edge[i].y = elem->n[k]->y - elem->n[j]->y;
}
```

and the 3×2 matrix B, defined in (4.6):

⟨*compute matrix B* 63c⟩≡ (62a)
```
J = edge[0].x * edge[1].y - edge[1].x * edge[0].y;

B[0][0] = - edge[0].y / J;
B[1][0] = - edge[1].y / J;
B[2][0] = - edge[2].y / J;

B[0][1] =   edge[0].x / J;
B[1][1] =   edge[1].x / J;
B[2][1] =   edge[2].x / J;
```

We complete our preparations by calling `get_shape_data]()` which returns the values of the 21 shape functions and their first and second derivatives in the `Pvals` array:

⟨*call get_shape_data() to fill the Pvals array* 63d⟩≡ (62a)
```
get_shape_data(elem, points, npoints, Pvals);
```

4.10.4 Computing the basis functions Q_i

At each of the `npoints` prescribed points we compute the basis functions Q_i according to (4.22):

⟨*for each point...* 64a⟩≡ (62a)

```
for (m = 0; m < npoints; m++) {
    int n, d;
```

⟨*evaluate the 21 basis functions* 64b⟩≡ (62a)

```
for (d = 0; d < 10; d++) {

    for (i = 0; i < 3; i++) {
        int j = (i + 1) % 3;
        int k = (i + 2) % 3;

        Qvals[6*i][d]
            = Pvals[m][6*i][d];
        Qvals[6*i+1][d]
            = edge[k].x * Pvals[m][6*i+1][d]
            - edge[j].x * Pvals[m][6*i+2][d];
        Qvals[6*i+2][d]
            = edge[k].y * Pvals[m][6*i+1][d]
            - edge[j].y * Pvals[m][6*i+2][d];
        Qvals[6*i+3][d]
            = edge[k].x * edge[k].x * Pvals[m][6*i+3][d]
            + edge[i].x * edge[i].x * Pvals[m][6*i+4][d]
            + edge[j].x * edge[j].x * Pvals[m][6*i+5][d];
        Qvals[6*i+4][d]
            = edge[k].y * edge[k].y * Pvals[m][6*i+3][d]
            + edge[i].y * edge[i].y * Pvals[m][6*i+4][d]
            + edge[j].y * edge[j].y * Pvals[m][6*i+5][d];
        Qvals[6*i+5][d] = 2.0 * (
              edge[k].x * edge[k].y * Pvals[m][6*i+3][d]
            + edge[i].x * edge[i].y * Pvals[m][6*i+4][d]
            + edge[j].x * edge[j].y * Pvals[m][6*i+5][d]);
    }

    for (i = 0; i < 3; i++) {
        int j = (i+1)%3;
        double ei2 = edge[i].x * edge[i].x + edge[i].y * edge[i].y;
        double ej2 = edge[j].x * edge[j].x + edge[j].y * edge[j].y;
        double eij = edge[i].x * edge[j].x + edge[i].y * edge[j].y;
        double h = sqrt((ei2*ej2 - eij*eij)/ei2);
        int sgn = (elem->e[i]->n1 == elem->n[j]) ? +1 : -1;
        Qvals[18+i][d] = Pvals[m][18+i][d] * h * sgn;
    }
}
```

⟨*evaluate the x, y, xx, xy, yyy derivatives of the basis functions* 64c⟩≡ (62a)

```
for (n = 0; n < 21; n++) {
    double T[2][2];
    double S[2];
    int i, j;   /* these shadow the outer i and j */
    int r, s;
```

```
    Q[m][n][d00] = Qvals[n][D000];

    for (j = 0; j < 2; j++) {
        S[j] = 0.0;
        for (r = 0; r < 3; r++)
            S[j] += Qvals[n][idx[r]] * B[r][j];
    }
    Q[m][n][d10] = S[0];
    Q[m][n][d01] = S[1];

    for (i = 0; i < 2; i++)
        for (j = i; j < 2; j++) {   /* T[i][j]'s upper triangle */
            T[i][j] = 0.0;
            for (r = 0; r < 3; r++)
                for (s = 0; s < 3; s++)
                    T[i][j] += Qvals[n][idx2[r][s]] * B[r][i] * B[s][j];
        }
    Q[m][n][d20] = T[0][0];
    Q[m][n][d11] = T[0][1];
    Q[m][n][d02] = T[1][1];
}
```

And finally free memory allocated to the `Pvals` and `Qvals` arrays:

⟨*free memory for the Pvals and Qvals arrays* 65a⟩≡ (62a)

```
}
FREE_3ARRAY(Pvals);
FREE_2ARRAY(Qvals);
```

4.11 A unit test

We write the function `get_basis_data()` described in the previous sections into a file *basis_functions.c*.

⟨*basis_functions.c* 65b⟩≡

```
#include <math.h>
#include "array.h"
#include "shape_functions.h"
#include "basis_functions.h"
⟨function get_basis_data() 62a⟩
⟨unit test for basis_functions.c 66⟩
```

⟨*basis_functions.h* 65c⟩≡

```
#ifndef H_BASIS_FUNCTIONS_H
#define H_BASIS_FUNCTIONS_H

#include "fem.h"
```

⟨*x derivative codes* 61⟩

```
void get_basis_data(Elem *elem, Point3d *points, int npoints, double ***Q);

#endif  /* H_BASIS_FUNCTIONS_H */
```

The computation performed in this `get_basis_data()` is in the core of our FEM calculations. To ensure its correctness and integrity, we have implemented a *Maple* script to evaluate the basis functions and their derivatives at a prescribed point a single element. We have included a unit test in the file *basis_functions.c* to evaluate the basis functions at the same point on that same element. The outputs of the *Maple* and C programs should be identical.

If using the *gcc* compiler, compile the unit test with:

```
gcc -Wall -std=c99 -pedantic basis_functions.c \
    shape_functions.o xmalloc.o -lm -DTEST
```

⟨*unit test for basis_functions.c* 66⟩≡ (65b)

```
#ifdef TEST

#include <stdio.h>

int main(void)
{
    double ***Q;
    int i;

    Node nodes[3] = {
        { 0, 0.0,  0.0 },
        { 1, 4.0, -1.2 },
        { 2, 1.1,  3.7 },
    };

    /* NOTE: Initialization with non-constant members requires a C99 compiler */
    Edge edges[3] = {
        { 0, &nodes[1], &nodes[2] },
        { 1, &nodes[2], &nodes[0] },
        { 2, &nodes[0], &nodes[1] },
    };

    Elem elems[1] = {
        {   0,
            { &nodes[0], &nodes[1], &nodes[2] },
            { &edges[0], &edges[1], &edges[2] },
        },
    };

    Point3d points[] = {
```

```
        { 0.32, 0.37, 1.0-0.32-0.37 },
    };

    MAKE_3ARRAY(Q, 1, 21, 6);

    get_basis_data(&elems[0], points, 1, Q);

    printf("lambda1 = %17.15g  lambda2 = %17.15g  lambda3 = %17.15g\n",
           points[0].x, points[0].y, points[0].z);

    printf("%2s %15s %15s %15s %15s %15s %15s\n",
           "i", "Q[i]", "Qx[i]", "Qy[i]", "Q_xx[i]", "Q_xy[i]", "Q_yy[i]");

    for (i = 0; i < 21; i++)
        printf("%2d %15.10f %15.10f %15.10f %15.10f %15.10f %15.10f\n",
               i,
               Q[0][i][d00],
               Q[0][i][d10],
               Q[0][i][d01],
               Q[0][i][d20],
               Q[0][i][d11],
               Q[0][i][d02]);

    return 0;
}

#endif /* TEST */
```

4.11.1 The *Maple* script

⟨*check-basis-funcs.maple* 67⟩≡

```
#  vi: ts=4 sw=4
#
# In this maple script we define a single element by prescribing
# its three vertices.  We prescribe an arbitrary point by its
# barycentric coordinates (lambda1, lambda2, lambda3), then
# evaluate the basis functions Q_i (not the shape functions P_i)
# and their x, y, xx, xy, yy derivatives at that point and print
# the results.
#
# To run, in Maple do:
# restart; read "check-basis-funcs.maple";
#
# 2007-07-04

kernelopts(printbytes=false):
with(LinearAlgebra):
Digits := 15;
```

```
# read the catalog of Argyris shapes:
read "catalog-all.maple":

# define the element's nodes:
nodes := [ <0.,0.>, <4.,-1.2>, <1.1,3.7> ]:

V[1] := nodes[1]:
V[2] := nodes[2]:
V[3] := nodes[3]:

E[1] := V[3] - V[2]:
E[2] := V[1] - V[3]:
E[3] := V[2] - V[1]:

A := < <V[1]|V[2]|V[3]>,<1|1|1> >:
B := A^(-1):

for i from 1 to 3 do
    j := i+1:
    k := i+2:
    if j > 3 then j := j - 3 end if:
    if k > 3 then k := k - 3 end if:

    Q[(6*(i-1)+1)] := P||(6*(i-1)+1);

    Q[(6*(i-1)+2)] :=
              E[k][1] * P||(6*(i-1)+2)
            - E[j][1] * P||(6*(i-1)+3);

    Q[(6*(i-1)+3)] :=
              E[k][2] * P||(6*(i-1)+2)
            - E[j][2] * P||(6*(i-1)+3);

    Q[(6*(i-1)+4)] :=
              E[k][1]^2 * P||(6*(i-1)+4)
            + E[i][1]^2 * P||(6*(i-1)+5)
            + E[j][1]^2 * P||(6*(i-1)+6);

    Q[(6*(i-1)+5)] :=
              E[k][2]^2 * P||(6*(i-1)+4)
            + E[i][2]^2 * P||(6*(i-1)+5)
            + E[j][2]^2 * P||(6*(i-1)+6);

    Q[(6*(i-1)+6)] := 2 * (
              E[k][1] * E[k][2] * P||(6*(i-1)+4)
            + E[i][1] * E[i][2] * P||(6*(i-1)+5)
            + E[j][1] * E[j][2] * P||(6*(i-1)+6) );

end do:
```

```
for i from 1 to 3 do
    j := i+1:
    if j > 3 then j := j - 3 end if:
    ei2 := E[i]^%T . E[i];
    ej2 := E[j]^%T . E[j];
    eij := E[i]^%T . E[j];
    h := sqrt((ei2*ej2 - eij^2)/ei2);
    Q[18+i] := h * P||(18+i);
end do:

# convert to (x,y) coords
z := B . <x, y, 1>:
for i from 1 to 21 do
    q[i] := unapply(Q[i](z[1],z[2],z[3], E[1], E[2], E[3]), [x,y]);
end do:

# Given point:
lambda1 := 0.32:
lambda2 := 0.37:
lambda3 := 1 - lambda1 - lambda2:

# find (x,y)
zz := A . < lambda1, lambda2, lambda3 >:
X := zz[1]:
Y := zz[2]:

printf("lambda1 = %17.15g  lambda2 = %17.15g  lambda3 = %17.15g\n",
        lambda1, lambda2, lambda3);

printf("%2s %15s %15s %15s %15s %15s %15s\n",
        "i", "Q[i]", "D[1](q[i])", "D[2](q[i])",
        "D[1,1](q[i])",
        "D[1,2](q[i])",
        "D[2,2](q[i])");

for i from 1 to 21 do
    printf("%2d %15.10f %15.10f %15.10f %15.10f %15.10f %15.10f\n",
        i-1,    # subtract 1 so that it matches the C output
        q[i](X, Y),
        D[1](q[i])(X, Y),
        D[2](q[i])(X, Y),
        D[1,1](q[i])(X, Y),
        D[1,2](q[i])(X, Y),
        D[2,2](q[i])(X, Y));
end do;
```

Chapter 5

Constructing functions from the Argyris data

In the previous chapeters we have seen how to define an mesh a domain, and how to build the Argyris basis functions on the elements. In this *interlude/intermezzo* chapter we assume that we are given an meshed domain and Argyris data on each element. We show how to construct the unique C^1 function on the domain corresponding to this data.

To be specific, let us look at an example. We define a domain, let's say Ω in $\mathbf{R}^2$ bounded by quadrilateral where the nodes are given in:

⟨*define a simple domain* 71a⟩≡ 71b▷
```
Node nodes[] = {
    { 0, 0.0,  0.0 },
    { 1, 1.0,  0.0 },
    { 2, 1.0,  1.0 },
    { 3, 0.0,  1.0 },
};
```

The mesh consists of two triangles produced by dividing the quadrilateral through one of its diagonals:

⟨*define a simple domain* 71a⟩+≡ ◁71a 72a▷
```
Edge edges[] = {
    { 0, &nodes[0], &nodes[1] },
    { 1, &nodes[1], &nodes[2] },
    { 2, &nodes[2], &nodes[3] },
    { 3, &nodes[3], &nodes[0] },
    { 4, &nodes[0], &nodes[2] },
};
```

The two elements are defined accordingly:

⟨*define a simple domain* 71a⟩+≡ ◁71b 72b▷

```
Elem elems[] = {
    { 0,    { &nodes[0], &nodes[1], &nodes[2] },
            { &edges[1], &edges[4], &edges[0] } },
    { 1,    { &nodes[0], &nodes[2], &nodes[3] },
            { &edges[2], &edges[3], &edges[4] } },
};
```

And finally define the mesh structure that encapsulates all of the above:

⟨*define a simple domain* 71a⟩+≡ ◁72a

```
Mesh mesh = {
    sizeof nodes  / sizeof *nodes,  /* number of nodes */
    sizeof edges  / sizeof *edges,  /* number of edges */
    sizeof elems  / sizeof *elems,  /* number of elementss */
    nodes,
    edges,
    elems
};
```

Now let us prescribe arbitrarily the following Argyris data on the mesh. Six values are specified at each node, which correspond to the value of the function and its x and y, xx, xy, yy derivatives, respectively. One value is specified at the midpoint of each edge, which corresponds to the value of the derivative in the direction perpendicular to the edge. Althogether in a mesh with $nnodes$ nodes and $nedges$ edges we will have $6nnodes + nedges$ values.

⟨*sample argyris data* 72c⟩≡

```
double argyris_data[] = {
    0.0, 0.0, 0.0, 0.0, 0.0, 0.0,   /* Node 0 */
    0.0, 0.0, 0.0, 0.0, 0.0, 0.0,   /* Node 1 */
    0.0, 0.0, 0.0, 0.0, 0.0, 0.0,   /* Node 2 */
    0.0, 0.0, 0.0, 0.0, 0.0, 0.0,   /* Node 3 */
    0.0,    /* Edge 0 midpoint */
    0.0,    /* Edge 1 midpoint */
    0.0,    /* Edge 2 midpoint */
    0.0,    /* Edge 3 midpoint */
    1.0,    /* Edge 4 midpoint */
};
```

The function `plot_in_geomview()` which we will describe later in this chapter, receives a pointer to the mesh structure and writes a file which may be read into *Geomview* to produce a plot of the function. The prototype of `plot_in_geomview()` is:

⟨*prototype plot_in_geomview* 72d⟩≡

```
void plot_in_geomview(Mesh *mesh, double *data, int n, enum x_derivs d);
```

where the last argument, which is one of the mnemonic codes defined in ⟨x

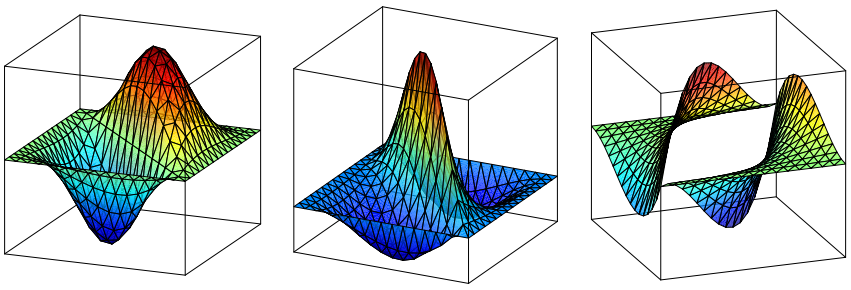


Figure 5.1: From left to right, the graphs of f, $\partial f/\partial x$ and $\partial^2 f/\partial x^2$, the latter showing clearly that the second derivative is discontinuous.

derivative codes (never defined)⟩, request drawing the graph of the specified derivative.

The plotting algorithm approximates the graph's smooth surface by a large number of flat triangles. The argment `n` shown in the prototype above controls the number of triangles within each element. To be precise, the graph over each element is approximated by n^2 triagles.

The graphs in Figure 5.1 were produced by:

```
plot_in_geomview(&mesh, argyris_data, 16, d00);
plot_in_geomview(&mesh, argyris_data, 16, d10);
plot_in_geomview(&mesh, argyris_data, 16, d20);
```

⟨*To be doucmented* 73⟩≡

```
#include <stdio.h>
#include <float.h>
#include "fem.h"
#include "array.h"
#include "abort.h"
#include "basis_functions.h"
#include "dump_mesh.h"

#define N(i,j) (offset + (i)*((i)+1)/2 + (j))       /* offset + i*(i+1)/2 + j */

typedef struct {
    double *node_data[3];
    double *edge_data[3];
} ArgyrisData;

#define REDHUE(s)       (huefunc((s)-23.0/64))
#define GREENHUE(s)     (huefunc((s) -7.0/64))
#define BLUEHUE(s)      (huefunc((s) +9.0/64))

/* This defines a continuous, piecewise-linear function defined on
   the entire real line.  Its graph is trapezoid-shaped and connects
```

```
   the points (0,0) (16/64,1), (33/64,1), (49/64,0).  The function
   is zero outside the support of the trapezoid.
   In combination with REDUE, GREENHUE, BLUEHUE macros defined above,
   produces an RGB scale used for our ZHUE rendering which coincides
   with Matlab's JET colormap.
*/
static float huefunc(float s)
{
    if (s < 0)
        return 0.0;
    else if (s < 16.0/64)
        return 4.0*s;
    else if (s < 33.0/64)
        return 1.0;
    else if (s < 49.0/64)
        return 1.0 - 4.0*(s-33.0/64);
    else
        return 0.0;
}


static void plot_elem_in_geomview(Elem *elem, ArgyrisData *data, int n,
        FILE *fp_nodes, FILE *fp_elems,
        Point3d *points, double ***Q, double *zmin, double *zmax, enum x_derivs d)
{
    Vec2d edge[3];
    int i, j, k;
    int offset = (n+1)*(n+2)/2 * elem->elemno;

    for (i = 0; i < 3; i++) {
        int j = (i + 1) % 3;
        int k = (i + 2) % 3;
        edge[i].x = elem->n[k]->x - elem->n[j]->x;
        edge[i].y = elem->n[k]->y - elem->n[j]->y;
    }

    /*
    We parametrize the triangle by paramters t and s, each going from
    0 to 1.  A generic point is given by x = u + t*c + s*a.
    The barycentric coordinates of the x are (1-t, t-s, s).  I found
    this in Maple:
    u := <u1,u2>;  v := <v1, v2>;  w := <w1,w2>;
    A := < <u|v|w>, <1|1|1> >;
    a := w-v; b := u-w; c := v-u;
    x := u + t*c + s*a;
    zz := A . <lambda1, lambda2, lambda3>;
    solve({ zz[1] = x[1], zz[2] = x[2], zz[3] = 1}, { lambda1, lambda2, lambda3 });

    For a partition of order n, we will have (n+1)*(n+2)/2 nodes and n^2 elems.
    */
```

```
printf("n = %d\n", n);

k = 0;
for (i = 0; i <= n; i++)
    for (j = 0; j <= i; j++) {
        double t = (double)i / n;
        double s = (double)j / n;
        points[k].x = 1.0 - t;      /* lambda1 */
        points[k].y = t - s;        /* lambda2 */
        points[k].z = s;        /* lambda3 */
        k++;
    }

printf("k = %d\n", k);


get_basis_data(elem, points, k, Q);

/* now evaluate function */
k = 0;
for (i = 0; i <= n; i++)
    for (j = 0; j <= i; j++) {
        double t = (double)i / n;
        double s = (double)j / n;
        double lambda[] = { 1.0 - t, t - s, s };
        double x = 0.0, y = 0.0, z = 0.0;
        int r, p;

        for (r = 0; r < 3; r++) {
            x += lambda[r] * elem->n[r]->x;
            y += lambda[r] * elem->n[r]->y;
        }

        for (r = 0; r < 3; r++) {
            for (p = 0; p < 6; p++)
                z += data->node_data[r][p] * Q[k][6*r+p][d];
            z += *(data->edge_data[r]) * Q[k][18+r][d];
        }

        if (z > *zmax)
            *zmax = z;
        else if (z < *zmin)
            *zmin = z;

        fprintf(fp_nodes, "%g %g %g\n", x, y, z);
        k++;
    }

for (i = 0; i < n; i++)
```

```
        for (j = 0; j <= i; j++) {
            fprintf(fp_elems, "%d %d %d\n", N(i,j), N(i+1,j), N(i+1,j+1));
            if (j + 1 <= i)
                fprintf(fp_elems, "%d %d %d\n", N(i,j), N(i+1,j+1), N(i,j+1));
            }
}

void plot_in_geomview(Mesh *mesh, double *data, int n, enum x_derivs d)
{
    FILE *fp_nodes;
    FILE *fp_elems;
    FILE *fp_out;
    double zmax, zmin;
    Point3d *points;
    double ***Q;
    int i;
    ArgyrisData argyris_data;
    double x, y, z;
    int n1, n2, n3;

    if ((fp_nodes = tmpfile()) == NULL)
        ABORT("unable to open temporary file\n");

    if ((fp_elems = tmpfile()) == NULL)
        ABORT("unable to open temporary file\n");

    if ((fp_out = fopen("zz.off", "w") )== NULL)
        ABORT("unable to open file zz.off for writing\n");

    /* working arrays */
    printf("length of vector 'points' = %d\n", (n+1)*(n+2)/2);
    MAKE_VECTOR(points, (n+1)*(n+2)/2);
    MAKE_3ARRAY(Q, (n+1)*(n+2)/2, 21, 6);

    zmin =  DBL_MAX;
    zmax = -DBL_MAX;

    for (i = 0; i < mesh->nelems; i++) {
        Elem *elem = &mesh->elems[i];
        argyris_data.node_data[0] = &data[6*elem->n[0]->nodeno];
        argyris_data.node_data[1] = &data[6*elem->n[1]->nodeno];
        argyris_data.node_data[2] = &data[6*elem->n[2]->nodeno];
        argyris_data.edge_data[0] = &data[6*mesh->nnodes + elem->e[0]->edgeno];
        argyris_data.edge_data[1] = &data[6*mesh->nnodes + elem->e[1]->edgeno];
        argyris_data.edge_data[2] = &data[6*mesh->nnodes + elem->e[2]->edgeno];
        plot_elem_in_geomview(elem, &argyris_data, n, fp_nodes, fp_elems, points,
                Q, &zmin, &zmax, d);
    }

    printf("zmin = %g, zmax = %g\n", zmin, zmax);
```

```
    /* preamble */
    fputs("{appearance {shading csmooth +edge}\n", fp_out);
    fputs("COFF\n", fp_out);    /* We have a [C]olor OFF file */
    fprintf(fp_out, "%d %d %d\n",
            (n+1)*(n+2)/2 * mesh->nelems,   /* number of nodes */
            n * n * mesh->nelems,       /* number of triangles */
            -1);                /* number of edges (not used) */

    /* The scaling here is temporary.  Here I am  scaling z so that the height of
     * the object changes to 1.  I need to compute the x and y ranges as well in order
     * to do a more meaningful scaling.
     */
    rewind(fp_nodes);
    while (fscanf(fp_nodes, "%lf %lf %lf", &x, &y, &z) == 3) {
        double s = (z - zmin)/(zmax - zmin);    /* color scale */
        double zc = 0.5*(zmax + zmin);
        double zh = zmax - zmin;
        double zscaled = zc + (z-zc)/zh;
        fprintf(fp_out, "%g %g %g %g, %g, %g, %g\n",
                x, y, zscaled,
                REDHUE(s), GREENHUE(s), BLUEHUE(s), 1.0);
    }

    rewind(fp_elems);
    while (fscanf(fp_elems, "%d %d %d", &n1, &n2, &n3) == 3)
        fprintf(fp_out, "3 %d %d %d\n", n1, n2, n3);

    /* closing */
    fputs("}\n", fp_out);

    /* done */
    fclose(fp_nodes);
    fclose(fp_elems);
    fclose(fp_out);
    FREE_VECTOR(points);
    FREE_3ARRAY(Q);
}

int main(void)
{
    ⟨sample argyrys data (never defined)⟩
    plot_in_geomview(&mesh, argyris_data, 16, d00);
}
```

Chapter 6

The mass, stiffness and bending matrices

With each element E we associate three 21×21 symmetric matrices called the *mass*, *stiffness* and *bending* matrices, and refer to them through the symbols M, K, L, respectily. Their entries are defined through:

$$m_{ij} = \int_E Q_i\big(\boldsymbol{\lambda}(\mathbf{x})\big)\, Q_j\big(\boldsymbol{\lambda}(\mathbf{x})\big)\, d\mathbf{x}, \tag{6.1}$$

$$k_{ij} = \int_E \nabla_{\mathbf{x}} Q_i\big(\boldsymbol{\lambda}(\mathbf{x})\big) \cdot \nabla_{\mathbf{x}} Q_j\big(\boldsymbol{\lambda}(\mathbf{x})\big)\, d\mathbf{x}, \tag{6.2}$$

$$l_{ij} = \int_E \Delta_{\mathbf{x}} Q_i\big(\boldsymbol{\lambda}(\mathbf{x})\big)\, \Delta_{\mathbf{x}} Q_j\big(\boldsymbol{\lambda}(\mathbf{x})\big)\, d\mathbf{x}, \tag{6.3}$$

where the basis functions Q_i are defined in (4.22) and where $\nabla_{\mathbf{x}}$ and $\Delta_{\mathbf{x}}$ are the gradient and Laplacian operators *which differentiate their arguments with respect to the* $\mathbf{x}$ *variable.*

Equation (4.22) gives the basis functions Q_i in terms of the shape functions P_i. We have explicit formulas (fifth degree polymials) for the shape functions, therefore we should be able to compute each of $3 \times 21 \times 22/2 = 693$ double integrals that define the entries in M, K, L. We have used *Maple* to evaluate the integrals and written the results as C statements into the files *pxp.incl*, *dpxdp.incl*, *d2pxd2p.incl* that correspond to the matrices M, K, and L, respectively.

In this chapter we will describe how to use these collections of statements to compute the three matrices.

6.1 The file *pxp.incl*

The shape function P_i are intrinsic to an element (triangle), that is, they are independent of the position of the triangle relative to the coordinate axes. Therefore for each i and j, the integral:

$$P_{ij} \equiv \int_E P_i\big(\boldsymbol{\lambda}(\mathbf{x})\big)\, P_j\big(\boldsymbol{\lambda}(\mathbf{x})\big)\, d\mathbf{x}, \tag{6.4}$$

is determined solely by the element's edge vectors **a**, **b**, **c**. Thus P_{ij} is a geometric property of the element.

We have cataloged formulas for all $21 \times 22/2 = 231$ instances of P_{ij} in the file *pxp.incl* expressed as C statements. Shown below is an extract consisting of several lines from the beginning and the end of the file.[1] The value of P_{ij} is encoded as `PxP[i][j]`:

⟨*the file pxp.incl* 80⟩≡

```
double x1 = ab/aa;
double x2 = bc/aa;
double x3 = ca/aa;
double x4 = ab/bb;
double x5 = bc/bb;
double x6 = ca/bb;
double x7 = ab/cc;
double x8 = bc/cc;
double x9 = ca/cc;
double PxP[21*22/2];

PxP[0][0] = x4 * x4 / 0.462e3 + x4 * x9 / 0.308e3
        + x9 * x9 / 0.462e3 - 0.41e2 / 0.2772e4 * x4
        - 0.41e2 / 0.2772e4 * x9 + 0.41e2 / 0.462e3;
PxP[1][0] = -x4 * x8 / 0.2640e4 - x8 * x9 / 0.1980e4
        - x4 / 0.840e3 + 0.41e2 / 0.23760e5 * x8
        - 0.3e1 / 0.1540e4 * x9 + 0.2021e4 / 0.166320e6;
PxP[1][1] = 0.7e1 / 0.59400e5 * x8 * x8 + x8 / 0.1100e4
        + 0.577e3 / 0.207900e6;
PxP[2][0] = -x4 * x5 / 0.1980e4 - x5 * x9 / 0.2640e4
        - 0.3e1 / 0.1540e4 * x4 + 0.41e2 / 0.23760e5 * x5
        - x9 / 0.840e3 + 0.2021e4 / 0.166320e6;
... 223 line deleted ...
PxP[20][17] = -x4 / 0.69300e5 - 0.1e1 / 0.20790e5;
PxP[20][18] = -x1 / 0.69300e5 - 0.1e1 / 0.20790e5;
PxP[20][19] = -0.1e1 / 0.69300e5;
PxP[20][20] = 0.32e2 / 0.51975e5;
```

The numerical coefficients entering these expressions are *not* truncated floating point approximations, despite what they may appear; they are *exact fractions*

[1] The file *pxp.incl* is not a self-contained *C translation unit* thus it cannot to be compiled on its own; it is inended to be '`#include`'ed in a regular C file. That is why its name does not have a normal `.c` extension.

which have been written in the C syntax. For instance, the last entry, `0.2021e4 / 0.166320e6`, stands for the *mathematically exact* fraction 2021/166320. There are no approximate or truncated numbers in the file *PxP.incl*.[2]

The notation `aa`, `ab` defined in the beginning of the file correspond to the following geometric quantities:

$$\begin{array}{lll} aa \equiv \mathbf{a}.\mathbf{a}, & bb \equiv \mathbf{b}.\mathbf{b}, & cc \equiv \mathbf{c}.\mathbf{c}, \\ ab \equiv \mathbf{a}.\mathbf{b}, & bc \equiv \mathbf{b}.\mathbf{c}, & ca \equiv \mathbf{c}.\mathbf{a}. \end{array} \tag{6.5}$$

6.2 Computing the mass, stiffness and bending matrices

The basis function, Q_i, defined in (4.22) depend on the relative position of the triangle with respect to the coordinate axes, therefore unlike the shape functions, P_i, they are not instrinsic propertypes of the element. Consequently, the functions Q_i, and the integrals in (6.1), (6.2), (6.3), cannot be computed and catalogued once for all; they need to be computed as needed for each element.

We see in (4.22) that each Q_i is a linear combination of as many as three shape functions P_i. We write this in symbolic form as:

$$Q_i = c_{i_1} P_{i_1} + c_{i_2} P_{i_2} + c_{i_3} P_{i_3} \tag{6.6}$$

where one or more coefficients may be zero. Then the integrand $Q_i Q_j$ in (6.1) may be expressed as a sum of up to nine terms $P_{i_p} P_{j_q}$ whose integrals may be looked up in the catalog in the file *pxp.incl*. Computing the mass matrix reduces to looking up values in the catalog and combining them with the proper coefficients.

In our program, the mass, stiffness and bending matrices are computed by calling the same function, `get_matrix()`. It receives a pointer to an element structure and a pointer to a preallocated memory for a 21×21 symmetric matrix, and one of the three flags:

⟨*flags for get_matrix()* 81a⟩≡ (87b)
```
enum which_matrix { MassMatrix, StiffnessMatrix, BendingMatrix };
```

and computes the requested matrix.

The function begins with calulating the triangle's edge vectors and the geometric quantities listed in (6.5):

⟨*function get_matrix()* 81b⟩≡ (87a) 82a▷
```
void get_matrix(Elem *elem, double **matrix, enum which_matrix which_matrix)
{
    double u1 = elem->n[0]->x;
```

[2]Of course the C compiler will convert these into its internal approximate floating point representation, as it does with any fraction.

```
double u2 = elem->n[0]->y;
double v1 = elem->n[1]->x;
double v2 = elem->n[1]->y;
double w1 = elem->n[2]->x;
double w2 = elem->n[2]->y;

double a1 = w1-v1;
double a2 = w2-v2;
double b1 = u1-w1;
double b2 = u2-w2;
double c1 = v1-u1;
double c2 = v2-u2;
double aa = a1*a1+a2*a2;
double bb = b1*b1+b2*b2;
double cc = c1*c1+c2*c2;
double ab = a1*b1+a2*b2;
double bc = b1*c1+b2*c2;
double ca = c1*a1+c2*a2;

double J = a1*b2-b1*a2; /* twice the area of the triangle */
```

The notation **a**, **b** and **c** for the edge vectors follows our previous usage. However for the purpose of the current computation, it is more convenient to have an index notation for edge vectors. So we let introduce the array `edge[]` of length 3, the elements of which are the edge vectors:

⟨*function get_matrix()* 81b⟩+≡ (87a) ◁81b 82b▷

```
Vec2d edge[] = {
    { a1, a2 },
    { b1, b2 },
    { c1, c2 },
};
```

To organize the calculations of the Q_i versus P_i, we introduce an array of structures that will hold the indices i_p and coefficients c_{i_p} (see (6.6)) of each Q_i:

⟨*function get_matrix()* 81b⟩+≡ (87a) ◁82a 83a▷

```
struct {
    int nterms;
    int idx[3];
    double coeff[3];
} Q[21];

int i, j;
```

We store the values of the integrals (**??**) in a lower triangular array, `PxP`, that we create by calling our `MAKE_LMATRIX` macro. Referring to the documentation of `MAKE_LMATRIX`, we see that the values of the matrix are stored in the row order, from top to bottom, in a contiguous array pointer to by `PxP[0]`. We give

a name to that array for ease of refefence:

⟨*function get_matrix()* 81b⟩+≡ (87a) ◁82b 83b▷

```
double **PxP;
double *pxp;
MAKE_LMATRIX(PxP, 21);
pxp = PxP[0];
```

The files *pxp.incl*, *dpxdp.incl*, *d2pxd2p.incl*, contain the values of the integrals all assigned as `pxp[...] = ....`

⟨*function get_matrix()* 81b⟩+≡ (87a) ◁83a 83c▷

```
switch (which_matrix) {
    case MassMatrix:
        #include "pxp.incl"
        break;
    case StiffnessMatrix:
        #include "dpxdp.incl"
        break;
    case BendingMatrix:
        #include "d2pxd2p.incl"
        break;
}
```

Remark. The switch statement above *is not* what it may seem in a superficial inspection. The three `#include` preprocessor directives are encountered at the preprocessing phase of the compilation therefore all three files, *pxp.incl*, *dpxdp.incl*, *d2pxd2p.incl*, are included in the translation unit, regardless of the surrounding `switch` statement. Each of the files defines the components of index 0 through 230 of an array named `pxp`. During the execution phase, one of the three sets of definitions is selected depending on the setting of the `which_matrix` flag.

We prepare for the computation of the integrals by populating the array of structures `Q[21]` defined above. `Q`'s first 6 entries correspond to the basis functions which have non-zero values at vertex 0. The next 6 entries correspond to the basis functions which have non-zero values at vertex 1 and the next 6 entries correspond to the basis functions which have non-zero values at vertex 2. Note that this analysis phase is independent of which of the mass, stiffness or bending matrices are to be computed.

⟨*function get_matrix()* 81b⟩+≡ (87a) ◁83b 84▷

```
for (i = 0; i < 3; i++) {
    int j = (i + 1) % 3;
    int k = (i + 2) % 3;

    Q[6*i].nterms = 1;
    Q[6*i].idx[0] = 6*i;
    Q[6*i].coeff[0] = 1.0;
```

```
    Q[6*i+1].nterms = 2;
    Q[6*i+1].idx[0] = 6*i+1;
    Q[6*i+1].idx[1] = 6*i+2;
    Q[6*i+1].coeff[0] =  edge[k].x;
    Q[6*i+1].coeff[1] = -edge[j].x;

    Q[6*i+2].nterms = 2;
    Q[6*i+2].idx[0] = 6*i+1;
    Q[6*i+2].idx[1] = 6*i+2;
    Q[6*i+2].coeff[0] =  edge[k].y;
    Q[6*i+2].coeff[1] = -edge[j].y;

    Q[6*i+3].nterms = 3;
    Q[6*i+3].idx[0] = 6*i+3;
    Q[6*i+3].idx[1] = 6*i+4;
    Q[6*i+3].idx[2] = 6*i+5;
    Q[6*i+3].coeff[0] = edge[k].x * edge[k].x;
    Q[6*i+3].coeff[1] = edge[i].x * edge[i].x;
    Q[6*i+3].coeff[2] = edge[j].x * edge[j].x;

    Q[6*i+4].nterms = 3;
    Q[6*i+4].idx[0] = 6*i+3;
    Q[6*i+4].idx[1] = 6*i+4;
    Q[6*i+4].idx[2] = 6*i+5;
    Q[6*i+4].coeff[0] = edge[k].y * edge[k].y;
    Q[6*i+4].coeff[1] = edge[i].y * edge[i].y;
    Q[6*i+4].coeff[2] = edge[j].y * edge[j].y;

    Q[6*i+5].nterms = 3;
    Q[6*i+5].idx[0] = 6*i+3;
    Q[6*i+5].idx[1] = 6*i+4;
    Q[6*i+5].idx[2] = 6*i+5;
    Q[6*i+5].coeff[0] = 2.0 * edge[k].x * edge[k].y;
    Q[6*i+5].coeff[1] = 2.0 * edge[i].x * edge[i].y;
    Q[6*i+5].coeff[2] = 2.0 * edge[j].x * edge[j].y;
}
```

The remaining 3 entries correspond to those basis functions that have non-zero normal derivatives at edge midpoints.

⟨*function get_matrix()* 81b⟩+≡ (87a) ◁83c 85a▷

```
for (i = 0; i < 3; i++) {
    int j = (i+1)%3;
    double ei2 = edge[i].x * edge[i].x + edge[i].y * edge[i].y;
    double ej2 = edge[j].x * edge[j].x + edge[j].y * edge[j].y;
    double eij = edge[i].x * edge[j].x + edge[i].y * edge[j].y;
    double h = sqrt((ei2*ej2 - eij*eij)/ei2);
    int sgn = (elem->e[i]->n1 == elem->n[j]) ? +1 : -1;
    Q[18+i].nterms = 1;
    Q[18+i].idx[0] = 18+i;
```

```
        Q[18+i].coeff[0] = sgn * h;
    }
```

Having computed the coefficients in (6.6), now we form the product Q_iQ_j through a doubly-nested for-loop then and substitite for the integrals of the shape functions from the catalog. Note that after computing the indices i_p and j_q, we read the value of the integral from $P_{i_pj_q}$ or $P_{j_qi_p}$, whichever corresponds to the entry in the lower triangle part of the matrix P_{ij}.

⟨*function get_matrix()* 81b⟩+≡ (87a) ◁84 85b▷

```
    for (i = 0; i < 21; i++)
        for (j = 0; j <= i; j++) {
            double sum = 0.0;
            int p, q;
            for (p = 0; p < Q[i].nterms; p++) {
                int ip = Q[i].idx[p];
                for (q = 0; q < Q[j].nterms; q++) {
                    int jq = Q[j].idx[q];
                    double myPP = (jq > ip) ? PxP[jq][ip] : PxP[ip][jq];
                    sum += Q[i].coeff[p] * Q[j].coeff[q] * myPP;
                }
            }
            switch (which_matrix) {
                case MassMatrix:
                    sum *= J;
                    break;
                case StiffnessMatrix:
                    sum /= J;
                    break;
                case BendingMatrix:
                    sum /= (J*J*J);
                    break;
            }
            matrix[i][j] = sum;
        }
```

Storage for the array PxP is no more needed so we free it:

⟨*function get_matrix()* 81b⟩+≡ (87a) ◁85a

```
    FREE_LMATRIX(PxP);
}
```

6.3 A unit test for get_matrix()

We create an element and pass it to get_matrix() to compute its mass mass, stiffness and bending matrices. Compile with:

```
gcc -Wall -std=c99 -pedantic mass-stiffness-bending.c xmalloc.o
```

`-lm -DTEST -DM=xxx`
where `xxx` is one of `MassMatrix`, `StiffnessMatrix` or `BendingMatrix`.

⟨*unit test for get_matrix()* 86⟩≡ (87a)

```
#ifdef TEST

#include <stdio.h>
#include "array.h"

int main(void)
{
    Node nodes[3] = {
        { 0, 0.0,  0.0 },
        { 1, 4.0, -1.2 },
        { 2, 1.1,  3.7 },
    };

    Edge edges[3] = {
        { 0, &nodes[1], &nodes[2] },
        { 1, &nodes[2], &nodes[0] },
        { 2, &nodes[0], &nodes[1] },
    };

    Elem elem = {
         0,
         { &nodes[0], &nodes[1], &nodes[2] },
         { &edges[0], &edges[1], &edges[2] },
    };

    double **matrix;

    int i, j;

    MAKE_LMATRIX(matrix, 21);

    get_matrix(&elem, matrix, M);

    for (i = 0; i < 21; i++) {
        printf("%2d  ", i);
        for (j = 0; j <= i; j++)
            printf("%8.4f ", matrix[i][j]);
        putchar('\n');
    }

    FREE_LMATRIX(matrix);

    return EXIT_SUCCESS;
}

#endif /* TEST */
```

⟨*mass-stiffness-bending.c* 87a⟩≡
```
#include <stdlib.h>
#include <math.h>
#include "array.h"
#include "mass-stiffness-bending.h"
⟨function get_matrix() 81b⟩
⟨unit test for get_matrix() 86⟩
```

⟨*mass-stiffness-bending.h* 87b⟩≡
```
#ifndef H_MASS_STIFFNESS_BENDING_H
#define H_MASS_STIFFNESS_BENDING_H

#include "fem.h"

⟨flags for get_matrix() 81a⟩

void get_matrix(Elem *elem, double **matrix, enum which_matrix which_matrix);

#endif /* H_MASS_STIFFNESS_BENDING_H */
```

6.4 Symbolic computation of the mass matrix

In this section we document the *Maple* script which was used to produce the symbolic expressions for the mass matrix which are encoded as a sequence of 231 C statements in the file *pxp.incl*. The reader who is not interested the method of derivation of these expressions may skip this section without loss of generality.

We will use the change of variables formula (4.17) in Section **??** to convert integration over an arbitrary triangle to integration over the triangle $0 \leqslant t_2 \leqslant t_1 \leqslant 1$

We begin with reading into *Maple* the symbolic expressions for the shape functions P_i:

⟨*mass-matrix.maple* 87c⟩≡ 88a▷
```
read "catalog-all.maple";
```

We define the three edge vectors. The *Maple* command `Vector(2, symbol='a'` defines a vector of length two whose components have they symbolic names a_1 and a_2. It would be convenient to refer to this vector with the symbol a in our calculation. However letting '`a := [Vector(2, symbol='a'`' is not allowed because then the symbol `a` will have conflicting meanings as the name of a vector and the name of its comonents. The trick to make this work is to declare the symbol `a` on the right hand side as something distinct from the symbol `a` on the left hand side. The command '`convert('a', 'local')`' creates a new symbol, named `a`, which is represented by the letter

'a' by internally is distinct from the anadonrned symbol `a`. Thus we let:

⟨*mass-matrix.maple* 87c⟩+≡ ◁87c 88b▷

```
a := Vector(2, symbol=convert('a', 'local'));
b := Vector(2, symbol=convert('b', 'local'));
c := Vector(2, symbol=convert('c', 'local'));
```

Remark. The Jacobian of the mapping, J, in the formula (4.17) equals twice the area of the triangle. With the edge vectors defined above, we have $J = c_1a_2 - c_2a_1$. However *in the following computation, and in the resulting catalog of mass matrix entries, we omit the factor* J. The user of the catalog is responsible for multiplying the entries by J.

We introduce the notation described in (6.5) to simplify the mass matrix entries:

⟨*mass-matrix.maple* 87c⟩+≡ ◁88a 88c▷

```
myrels1 := [
    a^%T . a = aa,
    b^%T . b = bb,
    c^%T . c = cc,
    a^%T . b = ab,
    b^%T . c = bc,
    c^%T . a = ca ];
```

Additional simplication is achieved through letting:

⟨*mass-matrix.maple* 87c⟩+≡ ◁88b 88d▷

```
myrels2 := [
    ab/aa=x1, bc/aa=x2, ca/aa=x3,
    ab/bb=x4, bc/bb=x5, ca/bb=x6,
    ab/cc=x7, bc/cc=x8, ca/cc=x9 ];
```

We designate a file in which to write the output and initialize it with some preamble material. Note that *Maple*'s I/O functions are very similar to those of C's.

⟨*mass-matrix.maple* 87c⟩+≡ ◁88c 89a▷

```
outfile := "/tmp/pxp.incl":
fp := fopen(outfile, WRITE):
fprintf(fp, "/* Do not edit!\n"):
fprintf(fp, " *\n"):
fprintf(fp, " * This C file was machine-generated by running\n"):
fprintf(fp, " * the script %s through Maple.\n", "mass-matrix.maple"):
fprintf(fp, " * Creation date: %s\n", StringTools[FormatTime]()):
fprintf(fp, " * Maple version: %s\n", kernelopts(version)):
fprintf(fp, "*/\n"):
fprintf(fp, "{ /* encloses everything in a block */\n"):
for tt in myrels2 do
    fprintf(fp, "double %a = %a;\n", rhs(tt), lhs(tt))
end do:
fclose(fp);
```

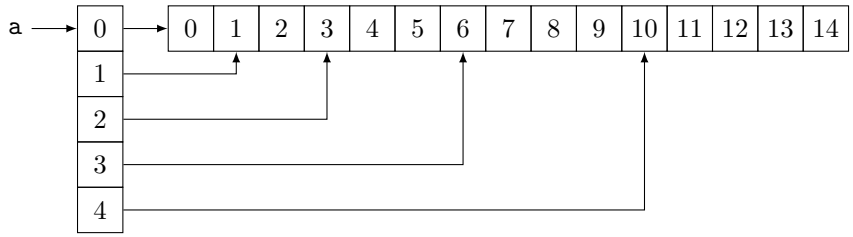


Figure 6.1: Caption here.

The for-loop over `myrels2` declares and defines 9 variables "`double x1 = ab/aa`', "`double x2 = bc/aa`', etc. These declarations, along with everything else in this *Maple*-generated file, are made into a C block by enclosing then with a pair of braces. This makes the code conformannt to the C89 standard which, unlike C99, does not allow mixing declarations with code.

The *Maple* program computes the lower triangle of the mass matrix and stores the entries, scanned in row order, from top to bottom, in an array `PP`, consistent with the storage scheme of used by the `MAKE_LMATRIX` macro. For the integration we use the change of variables formula (4.17), note however, that we do not include the J factor:

⟨*mass-matrix.maple* 87c⟩+≡ ◁88d 89b▷

```
vars := [seq(x||k, k=1..9)]:
PP := Vector(21*22/2):
k := 1:
for i from 1 to 21 do
    for j from 1 to i do
        int(int( P||i(1-t,t-s,s,a,b,c) * P||j(1-t,t-s,s,a,b,c),
                s=0..t), t=0..1);
        for tt in myrels1 do
             algsubs(tt, %);
        end do;
        for tt in myrels2 do
             algsubs(tt, expand(%));
        end do;
        PP[k] := sort(%, vars, descending);
        k := k + 1;
    end do;
end do;
```

Now we call *Maple*'s `CodeGeneration:-C` routine which converts the 231 entries of the array `PP` into C statements and appends the result to to file `outfile`:

⟨*mass-matrix.maple* 87c⟩+≡ ◁89a 90a▷

```
CodeGeneration:-C(convert(PP,list), resultname=pxp, output=outfile);
```

Finally we append a closing brace to the output file to match the opening brace and make the entire contents of the file into a single C block.

⟨*mass-matrix.maple* 87c⟩+≡ ◁89b

```
fp := fopen(outfile, APPEND):
fprintf(fp, "} /* encloses everything in a block */\n"):
fclose(fp):
```

6.4.1 Sample mass matrix generated in *Maple*

The file *mass-matrix-check.maple* is a *Maple* script that computes and prints the mass matrix for an element specified by the coordinates of its vertices. We use this script to verify that its output is identical to that of the unit test described in Section 6.3. This serves as a good verification device because the results are arrived at via different algorithmic paths.

We begin by reading the catalog of the shape functions:

⟨*mass-matrix-check.maple* 90b⟩≡ 90c ▷

```
read "catalog-all.maple";
```

We define a triangle by the coordinates of its vertices, then compute the edge vectors and the Jacobian J:

⟨*mass-matrix-check.maple* 90b⟩+≡ ◁90b 90d ▷

```
nodes := [ <0.,0.>, <4.,-1.2>, <1.1,3.7> ]:

V[1] := nodes[1]:
V[2] := nodes[2]:
V[3] := nodes[3]:

E[1] := V[3] - V[2]:
E[2] := V[1] - V[3]:
E[3] := V[2] - V[1]:

J := E[1][1]*E[2][2] - E[1][2]*E[2][1]:
```

We compute the 21 basis functions Q_i defined in (4.22):

⟨*mass-matrix-check.maple* 90b⟩+≡ ◁90c 91 ▷

```
for i from 1 to 3 do
    j := i+1:
    k := i+2:
    if j > 3 then j := j - 3 end if:
    if k > 3 then k := k - 3 end if:

    Q[6*(i-1)+1] := P||(6*(i-1)+1);

    Q[6*(i-1)+2] :=
              E[k][1] * P||(6*(i-1)+2)
```

```
            - E[j][1] * P||(6*(i-1)+3);

    Q[6*(i-1)+3] :=
              E[k][2] * P||(6*(i-1)+2)
            - E[j][2] * P||(6*(i-1)+3);

    Q[6*(i-1)+4] :=
              E[k][1]^2 * P||(6*(i-1)+4)
            + E[i][1]^2 * P||(6*(i-1)+5)
            + E[j][1]^2 * P||(6*(i-1)+6);

    Q[6*(i-1)+5] :=
              E[k][2]^2 * P||(6*(i-1)+4)
            + E[i][2]^2 * P||(6*(i-1)+5)
            + E[j][2]^2 * P||(6*(i-1)+6);

    Q[6*(i-1)+6] := 2 * (
              E[k][1] * E[k][2] * P||(6*(i-1)+4)
            + E[i][1] * E[i][2] * P||(6*(i-1)+5)
            + E[j][1] * E[j][2] * P||(6*(i-1)+6) );

end do:

for i from 1 to 3 do
    j := i+1:
    if j > 3 then j := j - 3 end if:
    ei2 := E[i]^%T . E[i];
    ej2 := E[j]^%T . E[j];
    eij := E[i]^%T . E[j];
    h := sqrt((ei2*ej2 - eij^2)/ei2);
    Q[18+i] := h * P||(18+i);
end do:
```

Then we apply the defintion (6.1) of mass matrix and the integration formula (4.17) to cacluate and print the lower triangle of the 21×21 mass matrix:

⟨*mass-matrix-check.maple* 90b⟩+≡ ◁90d

```
for i from 1 to 21 do
    printf("%2d  ", i-1);
    for j from 1 to i do
        Qi := Q[i](1-t,t-s,s,E[1], E[2], E[3]);
        Qj := Q[j](1-t,t-s,s,E[1], E[2], E[3]);
        M[i,j] := J * int(int( Qi * Qj, s=0..t), t=0..1):
        printf("%8.4f ", M[i,j]);
    end do;
    printf("\n");
end do:
```

The easiest way to run this script is through the command line:

```
maple mass-matrix-check.maple > /tmp/z.M
```

This runs the script through *Maple* and writes the result to file */tmp/z.M*.

Alternatively, it is possible to run the script from within *Maple*. By default, the output is printed to the terminal. To redirect the output to the file */tmp/z.M*, execute the following *Maple* commands:

```
restart;
writeto("/tmp/mass_matrix");
read mass-matrix-check.maple;
writeto(terminal);
```

6.5 Symbolic computation of the stiffness matrix

The *Maple* script for the symbolic computation of the stiffness matrix is mostly similar to the script for computing the mass matrix which we described in Section 6.4. The main novelty is the application of the chain rule for computing the gradient $\nabla_{\mathbf{x}} P_i$. Therefore we present the script with little commentary except for the places where we need to highlight the differences.

The preamble of the script is identical to that of the previous one, the only difference being the name of the output file which now is called *dpxdp.incl*:

⟨*stiffness-matrix.maple* 92⟩≡ 93a ▷

```
read "catalog-all.maple";
a := Vector(2, symbol=convert('a', 'local'));
b := Vector(2, symbol=convert('b', 'local'));
c := Vector(2, symbol=convert('c', 'local'));
myrels1 := [
    a^%T . a = aa,
    b^%T . b = bb,
    c^%T . c = cc,
    a^%T . b = ab,
    b^%T . c = bc,
    c^%T . a = ca ];
myrels2 := [
    ab/aa=x1, bc/aa=x2, ca/aa=x3,
    ab/bb=x4, bc/bb=x5, ca/bb=x6,
    ab/cc=x7, bc/cc=x8, ca/cc=x9 ];
outfile := "/tmp/dpxdp.incl":
fp := fopen(outfile, WRITE):
fprintf(fp, "/* Do not edit!\n"):
fprintf(fp, " *\n"):
```

```
fprintf(fp, " * This C file was machine-generated by running\n"):
fprintf(fp, " * the script %s through Maple.\n", "stiffness-matrix.maple"):
fprintf(fp, " * Creation date: %s\n", StringTools[FormatTime]()):
fprintf(fp, " * Maple version: %s\n", kernelopts(version)):
fprintf(fp, "*/\n"):
fprintf(fp, "{ /* encloses everything in a block */\n"):
for tt in myrels2 do
    fprintf(fp, "double %a = %a;\n", rhs(tt), lhs(tt))
end do:
fclose(fp);
```

To apply the chain rule of differentiation, we define the matrix A and its inverse, matrix B, accroding to (**??**) and (**??**).

⟨*stiffness-matrix.maple* 92⟩+≡ ◁92 93b▷

```
A := < < u[1] | v[1] | w[1] >,
       < u[2] | v[2] | w[2] >,
       <    1 |    1 |    1 > >;

B := < < -a[2] | a[1] | v[1]*w[2]-w[1]*v[2]>,
       < -b[2] | b[1] | w[1]*u[2]-u[1]*w[2]>,
       < -c[2] | c[1] | u[1]*v[2]-v[1]*u[2]> > / (a[1]*b[2]-b[1]*a[2]);
```

The 1×2 vector $\nabla_{\mathbf{x}} P_i$ which is the gradient of the composite function $P\big(\boldsymbol{\lambda}(\mathbf{x})\big)$ with respect to $\mathbf{x}$, equals the product of the 1×3 vector DP_i, which is the gradient of P_i with respect to the barycentric coordinates, times the first two columns of the matrix B.

In the symbolic representation of the matrix B, the Jacobian J appears as the denominator. To simplify the computed expressions, we introduce a matrix BJ which is the matrix B without its denominator:

⟨*stiffness-matrix.maple* 92⟩+≡ ◁93a 93c▷

```
BJ := simplify(B * denom(B[1,1]));
```

We use the matrix BJ instead of the matrix B in the application of the chain rule. This introduces an extra factor of J^2 in the resulting expressions. *The user should is responsible for dividing the results by* J^2.

⟨*stiffness-matrix.maple* 92⟩+≡ ◁93b 94a▷

```
vars := [seq(x||k, k=1..9)]:
k := 1:

for i from 1 to 21 do
    # compute grad P_i and simplify:
    Pix := P || i || x (1-t, t-s, s, a, b, c);
    Piy := P || i || y (1-t, t-s, s, a, b, c);
    Piz := P || i || z (1-t, t-s, s, a, b, c);
    < Pix | Piy | Piz >;
    for tt in myrels1 do
        algsubs(tt, %);
```

```
    end do;
    gradP[i] := % . LinearAlgebra[SubMatrix](BJ,1..3,1..2);
end do;
```

Remark. Recall that the `x`, `y`, `z` suffixes in the file *catalog-all.maple* are shorthhand notation for the barycentric coordinates λ_1, λ_2, λ_3 and should not be confused with Cartesian coordinates.

⟨*stiffness-matrix.maple* 92⟩+≡ ◁93c 94b▷

```
for i from 1 to 21 do
    for j from 1 to i do
        int(int(gradP[i] . gradP[j]^%T , s=0..t), t=0..1);
        for tt in myrels1 do
            algsubs(tt, expand(%));
        end do;
        for tt in myrels2 do
            algsubs(tt, expand(%));
        end do;
        collect(expand(%), vars, distributed);
        DPDP[k] := sort(%, vars, descending);
        k := k + 1;
    end do;
end do;
```

Finally, we convert the result to C and write the result to the output file:

⟨*stiffness-matrix.maple* 92⟩+≡ ◁94a

```
CodeGeneration:-C(convert(DPDP,list), resultname=pxp, output=outfile);
fp := fopen(outfile, APPEND):
fprintf(fp, "} /* encloses everything in a block */\n"):
fclose(fp):
```

6.5.1 Sample stiffness matrix generated in *Maple*

The file *stiffness-matrix-check.maple* is a *Maple* script that computes and prints the mass matrix for an element specified by the coordinates of its vertices. We use this script to verify that its output is identical to that of the unit test described in Section 6.3. This serves as a good verification device because the results are arrived at via different algorithmic paths.

The script begins exactly like the one in Section 6.4.1.

⟨*stiffness-matrix-check.maple* 94c⟩≡ 95a▷

```
read "catalog-all.maple":

nodes := [ <0.,0.>, <4.,-1.2>, <1.1,3.7> ]:

V[1] := nodes[1]:
V[2] := nodes[2]:
```

```
V[3] := nodes[3]:

E[1] := V[3] - V[2]:
E[2] := V[1] - V[3]:
E[3] := V[2] - V[1]:

J := E[1][1]*E[2][2] - E[1][2]*E[2][1]:
```

We define the A and B matrices which which are needed for computing the derivatives the by the chain rule:

⟨*stiffness-matrix-check.maple* 94c⟩+≡ ◁94c 95b▷

```
A := < <V[1]|V[2]|V[3]>,<1|1|1> >:
B := A^(-1):
```

The matrices A and B are mappings between the barycentric coordinates to Cartesian coordinates. We express these through the variables `lambda` and `X`:

⟨*stiffness-matrix-check.maple* 94c⟩+≡ ◁95a 95c▷

```
lambda := B . <x[1], x[2], 1>;
X := A . <lambda1, lambda2, lambda3>;
```

The calculation of the basis function Q_i is identical to the corresponding part of the previous script:

⟨*stiffness-matrix-check.maple* 94c⟩+≡ ◁95b 96▷

```
for i from 1 to 3 do
    j := i+1:
    k := i+2:
    if j > 3 then j := j - 3 end if:
    if k > 3 then k := k - 3 end if:

    Q[(6*(i-1)+1)] := P||(6*(i-1)+1);

    Q[(6*(i-1)+2)] :=
              E[k][1] * P||(6*(i-1)+2)
            - E[j][1] * P||(6*(i-1)+3);

    Q[(6*(i-1)+3)] :=
              E[k][2] * P||(6*(i-1)+2)
            - E[j][2] * P||(6*(i-1)+3);

    Q[(6*(i-1)+4)] :=
              E[k][1]^2 * P||(6*(i-1)+4)
            + E[i][1]^2 * P||(6*(i-1)+5)
            + E[j][1]^2 * P||(6*(i-1)+6);

    Q[(6*(i-1)+5)] :=
              E[k][2]^2 * P||(6*(i-1)+4)
            + E[i][2]^2 * P||(6*(i-1)+5)
```

```
            + E[j][2]^2 * P||(6*(i-1)+6);

    Q[(6*(i-1)+6)] := 2 * (
              E[k][1] * E[k][2] * P||(6*(i-1)+4)
            + E[i][1] * E[i][2] * P||(6*(i-1)+5)
            + E[j][1] * E[j][2] * P||(6*(i-1)+6) );

end do:

for i from 1 to 3 do
    j := i+1:
    if j > 3 then j := j - 3 end if:
    ei2 := E[i]^%T . E[i];
    ej2 := E[j]^%T . E[j];
    eij := E[i]^%T . E[j];
    h := sqrt((ei2*ej2 - eij^2)/ei2);
    Q[18+i] := h * P||(18+i);
end do:
```

Now we apply the defintion (**??**) of stiffness matrix and the integration formula (4.17) to cacluate and print the lower triangle of the 21×21 stiffness matrix:

⟨*stiffness-matrix-check.maple* 94c⟩+≡ ◁95c

```
for i from 1 to 21 do
    printf("%2d  ", i-1);
    Q[i](lambda[1], lambda[2], lambda[3], E[1], E[2], E[3]);
    qi := unapply(%, [ x[1], x[2] ] );
    qix := unapply(D[1](qi)(X[1],X[2]), [lambda1, lambda2, lambda3]);
    qiy := unapply(D[2](qi)(X[1],X[2]), [lambda1, lambda2, lambda3]);
    for j from 1 to i do
        Q[j](lambda[1], lambda[2], lambda[3], E[1], E[2], E[3]);
        qj := unapply(%, [ x[1], x[2] ] );
        qjx := unapply(D[1](qj)(X[1],X[2]), [lambda1, lambda2, lambda3]);
        qjy := unapply(D[2](qj)(X[1],X[2]), [lambda1, lambda2, lambda3]);
        integrand :=
            qix(1-t,t-s,s) * qjx(1-t,t-s,s)
            +
            qiy(1-t,t-s,s) * qjy(1-t,t-s,s);
        M[i,j] := J * int(int( integrand, s=0..t), t=0..1):
        printf("%8.4f ", M[i,j]);
    end do;
    printf("\n");
end do:
```

See the instruction near the end of Section 6.4.1 for running this script.

6.6 Symbolic computation of the bending matrix

The *Maple* script for the symbolic computation of the bending matrix is mostly similar to the scripts for computing the mass and stiffness matrices which we described in the previous section. The main novelty is the application of the formula (4.15) to compute the Laplacian $\Delta_{\mathbf{x}} P_i$. We present the script with little commentary except for the places where we need to highlight the differences with the previoous ones.

The preamble of the script is as before, except the output file name which now is called *dpxdp.incl*:

ADD EXPLANATION

⟨*bending-matrix.maple* 97⟩≡

```
# read the note in 00how-to-compute-the-laplaican

read "catalog-all.maple";
a := Vector(2, symbol=convert('a', 'local'));
b := Vector(2, symbol=convert('b', 'local'));
c := Vector(2, symbol=convert('c', 'local'));
myrels1 := [
    a^%T . a = aa,
    b^%T . b = bb,
    c^%T . c = cc,
    a^%T . b = ab,
    b^%T . c = bc,
    c^%T . a = ca ];

myrels3 := [ bc = -(bb + cc -aa)/2, ca = -(cc + aa  -bb)/2, ab = -(aa + bb - cc)/2 ];

fracs := [seq(seq(aa^i*bb^j*cc^(2-i-j), j=-4..4), i=-4..4)];
fracs_short := select(x -> degree(numer(x)) <= 4, fracs);
myrels_x := [ seq( fracs_short[k] = x||k, k=1..nops(fracs_short)) ];

### myrels_x := [ cc^4/aa^2 = x1, bb*cc^3/aa^2 = x2, bb^2*cc^2/aa^2 = x3, bb^3*cc/aa^2 = x4, bb^4/aa^2 = x

A := < < u[1] | v[1] | w[1] >,
       < u[2] | v[2] | w[2] >,
       <    1 |    1 |    1 > >;

B := < < -a[2] | a[1] | v[1]*w[2]-w[1]*v[2]>,
       < -b[2] | b[1] | w[1]*u[2]-u[1]*w[2]>,
       < -c[2] | c[1] | u[1]*v[2]-v[1]*u[2]> > / (a[1]*b[2]-b[1]*a[2]);
BJ := simplify(B * denom(B[1,1]));
```

```
Z := Matrix(3,3, shape=symmetric);

Z[1,1] := aa; Z[2,2] := bb; Z[3,3] := cc; Z[1,2] := ab; Z[2,3] := bc; Z[3,1] := ca;

D2P := Matrix(3,3,shape=symmetric);

k := 1:

# compute the Laplacian of P_i and simplify:
for i from 1 to 21 do
    DD :=
          [ P || i || xx (1-t, t-s, s, a, b, c),
        P || i || yy (1-t, t-s, s, a, b, c),
        P || i || zz (1-t, t-s, s, a, b, c),
        P || i || xy (1-t, t-s, s, a, b, c),
        P || i || yz (1-t, t-s, s, a, b, c),
        P || i || zx (1-t, t-s, s, a, b, c) ];

    for tt in myrels1 do
        algsubs(tt, %);
    end do;

    DD := %;

    D2P[1,1] := DD[1];
    D2P[2,2] := DD[2];
    D2P[3,3] := DD[3];
    D2P[1,2] := DD[4];
    D2P[2,3] := DD[5];
    D2P[3,1] := DD[6];

    LP[i] := LinearAlgebra:-Trace(Z.D2P);
end do:

PP := Vector(21*22/2):

k := 1: for i from 1 to 21 do
    for j from 1 to i do

        int(int(LP[i] * LP[j],  s=0..t), t=0..1);

        subs(myrels3, %);

        PP[k] := subs(myrels_x, expand(%));
        k := k + 1;
    end do; # end of the j loop
end do;     # end of the i loop

outfile := "/tmp/d2pxd2p.incl":
```

```
fp := fopen(outfile, WRITE):
fprintf(fp, "/* Do not edit!\n"):
fprintf(fp, " *\n"):
fprintf(fp, " * This C file was machine-generated by running\n"):
fprintf(fp, " * the script %s through Maple.\n", "bending-matrix.maple"):
fprintf(fp, " * Creation date: %s\n", StringTools[FormatTime]()):
fprintf(fp, " * Maple version: %s\n", kernelopts(version)):
fprintf(fp, "*/\n"):
fprintf(fp, "{ /* encloses everything in a block */\n"):

myrels_r := [ seq(rhs(i) = lhs(i), i in myrels_x) ];
mystr := CodeGeneration:-C(myrels_r, optimize, output=string):
mystrlist := StringTools:-Split(mystr, "\n"):
for i in mystrlist do
    if i <> "" then
        fprintf(fp, "double %s\n", i);
    end if;
end do:
fclose(fp);

CodeGeneration:-C(convert(PP,list), resultname=pxp, output=outfile);

fp := fopen(outfile, APPEND):
fprintf(fp, "} /* encloses everything in a block */\n"):
fclose(fp):
```

6.6.1 Sample bending matrix generated in *Maple*

ADD EXPLNATION HERE

⟨*bending-matrix-check.maple* 99⟩≡

```
read "catalog-all.maple":

nodes := [ <0.,0.>, <4.,-1.2>, <1.1,3.7> ]:

V[1] := nodes[1]:
V[2] := nodes[2]:
V[3] := nodes[3]:

E[1] := V[3] - V[2]:
E[2] := V[1] - V[3]:
E[3] := V[2] - V[1]:

J := E[1][1]*E[2][2] - E[1][2]*E[2][1]:
A := < <V[1]|V[2]|V[3]>,<1|1|1> >:
B := A^(-1):
lambda := B . <x[1], x[2], 1>;
X := A . <lambda1, lambda2, lambda3>;
for i from 1 to 3 do
```

```
    j := i+1:
    k := i+2:
    if j > 3 then j := j - 3 end if:
    if k > 3 then k := k - 3 end if:

    Q[(6*(i-1)+1)] := P||(6*(i-1)+1);

    Q[(6*(i-1)+2)] :=
              E[k][1] * P||(6*(i-1)+2)
            - E[j][1] * P||(6*(i-1)+3);

    Q[(6*(i-1)+3)] :=
              E[k][2] * P||(6*(i-1)+2)
            - E[j][2] * P||(6*(i-1)+3);

    Q[(6*(i-1)+4)] :=
              E[k][1]^2 * P||(6*(i-1)+4)
            + E[i][1]^2 * P||(6*(i-1)+5)
            + E[j][1]^2 * P||(6*(i-1)+6);

    Q[(6*(i-1)+5)] :=
              E[k][2]^2 * P||(6*(i-1)+4)
            + E[i][2]^2 * P||(6*(i-1)+5)
            + E[j][2]^2 * P||(6*(i-1)+6);

    Q[(6*(i-1)+6)] := 2 * (
              E[k][1] * E[k][2] * P||(6*(i-1)+4)
            + E[i][1] * E[i][2] * P||(6*(i-1)+5)
            + E[j][1] * E[j][2] * P||(6*(i-1)+6) );

end do:

for i from 1 to 3 do
    j := i+1:
    if j > 3 then j := j - 3 end if:
    ei2 := E[i]^%T . E[i];
    ej2 := E[j]^%T . E[j];
    eij := E[i]^%T . E[j];
    h := sqrt((ei2*ej2 - eij^2)/ei2);
    Q[18+i] := h * P||(18+i);
end do:

for i from 1 to 21 do
    Q[i](lambda[1], lambda[2], lambda[3], E[1], E[2], E[3]);
    qi := unapply(%, [ x[1], x[2] ] );
    qixx := D[1,1](qi)(X[1],X[2]);
    qiyy := D[2,2](qi)(X[1],X[2]);
    LQ[i] := unapply(qixx + qiyy, [lambda1, lambda2, lambda3]);
end do:
```

```
for i from 1 to 21 do
    printf("%2d  ", i-1);
    for j from 1 to i do
        integrand := LQ[i](1-t,t-s,s) * LQ[j](1-t,t-s,s);
            M[i,j] := J * int(int( integrand, s=0..t), t=0..1):
        printf("%8.4f ", M[i,j]);
    end do;
    printf("\n");
end do:
```

Chapter 7

The boundary series

A boundary patch is a connected set of one or more boundary edges which have a common type of boundary data. The precise definition was given in Section 2.2.4. In that section we developed code that identifies boundary patches from the user specified data and enumerates them, beginning with zero. We arranged things so that upon the triangulation of the domain, edges on the mesh boundary inherit patch numbers from their parent edge segments.

In this chapter we make a linked list from the edges of each boundary patch. We sort each list so that the edges are back to back within each patch, that is, the second node of an edge connects to the first node of the next edge. Additionally, we enforce a positive orientation of the boundary in the sence that traversing the linked list correponds to traversing the domain's boundary in the poisitive direction. We call the resulting linked list a *boundary series* corresponding to that patch. See Section 1.1.3 for the meaning of positive orientation.

The member `bseries` of the `Mesh` structure is an array of pointers to a boundary series, one per boundary patch. The number of boundary patches which has been computed in ⟨*function identify_boundary_patches()* (never defined)⟩ is stored in the `npatches` member of the `Mesh` structure. Here we allocate memory for `bseries` array, create the of boundary series and set the array elements to point the them.

⟨*function create_boundary_series()* 103⟩≡ (108c)

```
void create_boundary_series(Mesh *mesh)
{
    int i;

    MAKE_VECTOR(mesh->bseries, mesh->nbseries);
    for (i = 0; i < mesh->nbseries; i++)
        mesh->bseries[i] = NULL;

    for (i = 0; i < mesh->nedges; i++) {
        Edge *edge = &mesh->edges[i];
```

```
        if (edge->bc != NULL) {
            int series = edge->bc->patch;
            mesh->bseries[series]
                    = List_push(mesh->bseries[series], edge);
        }
    }

    /* TEMPORARY: print bseries */
    for (i = 0; i < mesh->nbseries; i++) {
        List *p;
        printf("edges series %d:\n", i);
        for (p = mesh->bseries[i]; p != NULL; p = p->rest) {
            Edge *edge = p->first;
            printf("\tbseries %d, edge %d\n", edge->bc->patch, edge->edgeno);
        }
    }
}
```

At this point, edges in a boundary series are not in any particular order. Here we sort each boundary series to put the edges in their natural order as described in the opening paragraph of this chapter. The sorting algorithm we use here is rather native and has a complexity of $O(n^2)$ where n is the number the edges to sort. It is possible to reduce the complexity to $O(n \ln n)$ through the use of hash tables but we don't do it here because its impact on the overall performance of the solver is doubtful.

The function `sort_bseries` that will be described below, receives a pointer `bseries` to a boundary series. It removes the first link and start a new linked list, designated as `list`, which has the removed link as its only member. Then `list` is extended by repeatedly removing links from `bseries` and merging into `list`. This is done in two distinct phases.

In phase 1, we we build up the linked list `list` 'from the left' by repeatedly selecting and removing links from `bseries` and *pushing* them into `list`. The criterion for selectting links is that they 'fit' in the sense that node number 2 of the edge being pushed equals node number 1 of the edge that it is being pushed into. Figure 7.1 illustrates this.

After exhausting all edges that can be pushed from the left, we shift to phase 2 where we continue building the list by selecting and removing links from `bseries` and *appending* them to `list`. The criterion for selectting links is that they 'fit' in the sense that node number 1 of the edge being appended equals node number 2 of the edge that it is being appended to. Figure 7.2 illustrates this.

We introduce functions `select_n1()` and `select_n2()` to help with selecting the desired edge. Each of these receive a pointer to and edge and a node number. The first one returns true if the node number of the edge's first node matches the given node number. The second one does the same with the edge's second node number. The selector functions will be passed as arguments to our sorting

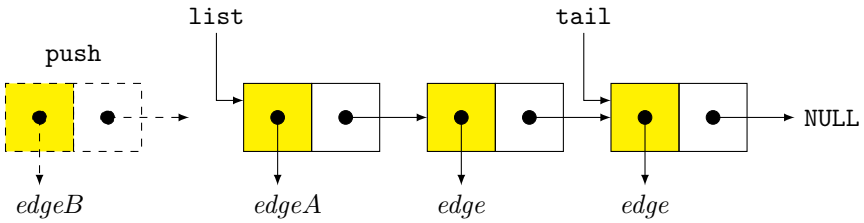


Figure 7.1: Phase 1 of builing a linked list of sorted edges. Links are 'pushed' from the left. Node 2 of the edge B matches node 1 of edge A. The `list` pointer is set to point to the leftmost link after each insertion. The `tail` pointer continues pointing to the last link.

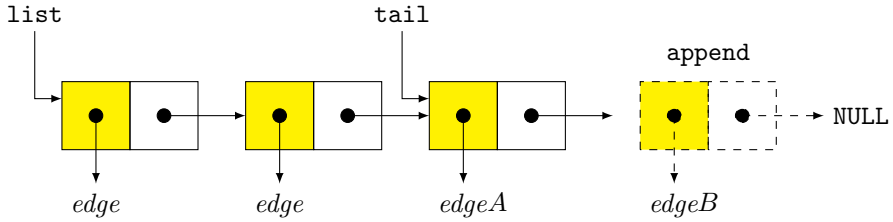


Figure 7.2: Phase 2 of building a linked list of sorted edges. Links are appended to the right. Node 1 of edge B matches node 2 of edge A. The `list` pointer remains stationary. The `tail` pointer is advanced to the newly appeded link.

function therefore we `typedef` their prototype:

⟨*edge selector functions* 106a⟩≡ (108c)
```
typedef int (*EdgeSelector)(Edge *e, int n);

static int select_n1(Edge *e, int n)
{
    return e->n1->nodeno == n;
}

static int select_n2(Edge *e, int n)
{
    return e->n2->nodeno == n;
}
```

Another helper funciton is `extract_edge()` which searches a linked list of edges for an edge with a specified first or second node number. If the desired edge is found, the corresponding link is removed from the list and the shortened list is returned. The `edge` argument is set to point to the located edge. If no suitable link is found, it returns the unmodified list and sets the `edge` argument to `NULL`.

⟨*function extract_edge()* 106b⟩≡ (108c)
```
static List *extract_edge(List *list, int n,
        EdgeSelector select, Edge **edge)
{
    List **pp, *p;
    Edge *q;

    for (pp = &list; (p = *pp) != NULL; pp = &p->rest) {
        q = p->first;
        if (select(q, n))
            break;
    }

    if (p == NULL)
        *edge = NULL;
    else {
        *edge = q;
        *pp = p->rest;
    }

    return list;
}
```

With the help of this auxiliary function, sorting a boundary series is quite simple. The function `sort_bseries` receives a pointer to a bounday series and returns a pointer to the sorted series. If the argument is `NULL`, it returns `NULL`. Otherwise it begins by extracting the first element from the given series and initializes a new linked list, called `list`, with it. The pointer `tail` is set to point to the last

link of `list`. It will be used in phase 2 of the sorting process.

⟨*function sort_bseries()* 107a⟩≡ (108c) 107b▷

```
List *sort_bseries(List *bseries)
{
    List *list, *tail;

    if (bseries == NULL)
        return NULL;

    list = bseries;
    bseries = list->rest;
    list->rest = NULL;
    tail = list;
```

Now we are ready to perform phase 1:

⟨*function sort_bseries()* 107a⟩+≡ (108c) ◁107a 107c▷

```
    while (bseries != NULL) {
        Edge *edge = list->first;
        int n1 = edge->n1->nodeno;

        bseries = extract_edge(bseries, n1, select_n2, &edge);
        if (edge == NULL)
            break;
        list = List_push(list, edge);
    }
```

At this point if `bseries` is exhausted, then the list is completely sorted and there is nothing else to do. We return:

⟨*function sort_bseries()* 107a⟩+≡ (108c) ◁107b 107d▷

```
    if (bseries == NULL)
        return list;
```

But in the more likely event when `bseries` is not exhausted, we shift to phase 2:

⟨*function sort_bseries()* 107a⟩+≡ (108c) ◁107c 108a▷

```
    do {
        Edge *edge = tail->first;
        int n2 = edge->n2->nodeno;
        bseries = extract_edge(bseries, n2, select_n1, &edge);
        if (edge == NULL)
            break;
        List_append(tail, List_list(edge, NULL));
        tail = tail->rest;
    } while (bseries != NULL);
```

At this point `bseries` should be exhausted otherwise we have a bug in the program. Check this and abort the program if need be. Otherwise return `list` to the caller:

⟨*function sort_bseries()* 107a⟩+≡ (108c) ◁107d

```
    if (bseries != NULL)
        ABORT("shouldn't be here!");
    else
        return list;
}
```

7.1 The files

⟨*boundary.h* 108b⟩≡

```
#ifndef H_BOUNDARY_H
#define H_BOUNDARY_H

#include "linked-list.h"
#include "fem.h"

void create_boundary_series(Mesh *mesh);
List *sort_bseries(List *bseries);

#endif /* H_BOUNDARY_H */
```

⟨*boundary.c* 108c⟩≡

```
#include "abort.h"
#include "array.h"
#include "boundary.h"

⟨function create_boundary_series() 103⟩
⟨edge selector functions 106a⟩
⟨function extract_edge() 106b⟩
⟨function sort_bseries() 107a⟩
```

Chapter 8

The front end

In this chapter we describe the contents of the file *main.c* which serves as a front end to the FEM software. It is expected that the user will vary some parts of it to suit specific needs.

8.1 The structure of *main.c*

Here is the overall structure of the file *main.c*:

⟨*main.c* 109a⟩≡
```
⟨included headers for main.c 109b⟩
int main(int argc, char **argv)
{
    ⟨declarations for main.c 111c⟩
    ⟨parse command line options 111d⟩
    ⟨load problem module 112f⟩
    ⟨perform user requested action 113f⟩
    ⟨clean up 114f⟩
}
```

We include the standard headers

⟨*included headers for main.c* 109b⟩≡ (109a) 109c▷
```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

for declarations of functions such as `sprintf()`, `strchr()` from the C standard library and preprocessor symbols, such as `EXIT_SUCCESS`. Additionally we include the header

⟨*included headers for main.c* 109b⟩+≡ (109a) ◁109b 111b▷
```
#include <dlfcn.h>
```

which is *not* a part of the C standard library but a POSIX extension which should be available of all Unix or Unix-like operating systems that conform to the POSIX standard. Among other things, this header file declares the functions `dlopen()` and `dlsym()` which are interfaces to the dynamic linking loader.

8.2 Parsing command line options

FEM has to be invoked with at least one command line option that specifies a module (see Chapter 1) to load. To load the module *my_module.so*, for instance, we may invoke the program through either of the two equivalent forms:

```
fem -m my_module.so
fem --module my_module.so
```

Most option flags have a *long form* and a *short form* variants. A flag in the long form consists of two hyphens followed by a descriptive name, such as `--module`. A flag in the short form consists of a hyphen and a single letter, such as `-m`. The long form flag need not be spelled out fully. For instance, `--mod` or even `--m` may be sufficient if that uniquely identifies the flag among all possible flags.

The flags `-h` or `--help` cause FEM to print the list of all available options and exit. Here is what we see:

```
fem 0.1, July 2007

Usage: fem [OPTIONS]...

  -h, --help              Print help and exit
  -V, --version           Print version and exit
  -m, --module=module.so  Load problem specification module
  -a, --area=max_area     Max area of an element  (default='1.0')
  -s, --show-mesh         Write mesh to an OFF (geomview) file
  -d, --dump-mesh         Dump mesh details
```

8.2.1 Gengetopt

The parsing of the command line options is a straightforward but tedious task which is best handled by automated tools rather than in ad hoc ways. In FEM we use the open source *GNU gengetopt* command line parser generator which may be obtained from

```
<http://www.gnu.org/software/gengetopt/gengetopt.html>.
```

Since the documentation is available online, we won't reproduce it here. In brief, gengetopt reads a list of command line option definitions from a user-supplied

text file which we call *cmdline.ggo*. Here is the file in its entirety:

⟨*cmdline.ggo* 111a⟩≡
```
option "module" m "Load problem specification module"
    string typestr="module.so" required

option "area" a "Max area of an element"
    float default="1.0" typestr="max_area" optional

option "show-mesh" s "Write mesh to an OFF (geomview) file" optional

option "dump-mesh" d "Dump mesh details " optional
```

The parser is generated by invoking gengetopt as:

```
gengetopt --set-package=fem --set-version="0.1, July 2007" < cmdline.ggo
```

This generates the parser and writes it to the files *cmdline.c* and *cmdline.h*. The main interface to the parse is the `cmdline_parser()` function which we will described in the next section.

8.2.2 Calling the parser

The command line data captured in `main()`'s `argc` and `argv` arguments is passed to the function `cmdline_parser()` for analysis. The header file *cmdline.h* declares the function's prototype:

⟨*included headers for main.c* 109b⟩+≡ (109a) ◁109c 112a▷
```
#include "cmdline.h"
```

Additionally, *cmdline.h* declares a '`struct gengetopt_args_info`' structure which will hold the results of the parser's analysis. For each long form command line flag '*xxx*', the structure has a member named '*xxx*_given' which is a boolean variable which indicates if the long flag '*xxx*' (or its short equivalent) was or was not specified on the command line. If '*xxx*' takes an argument, then the structure has a member named '*xxx*_arg' which holds the value of that argument.

⟨*declarations for main.c* 111c⟩≡ (109a) 112b▷
```
struct gengetopt_args_info args_info;
```

We call the parser as:

⟨*parse command line options* 111d⟩≡ (109a) 112c▷
```
if (cmdline_parser(argc, argv, &args_info) != 0)
    ABORT("Command line parser failed.  Out of memory?\n");
```

The command may fail, possibly due to lack of sufficient memory, in which case the preprocessor macro, `ABORT()`, defined in the header file *abort.h*, will print the specified message and terminate the program.

⟨*included headers for main.c* 109b⟩+≡ (109a) ◁111b 112d▷
```
#include "abort.h"
```

8.3 Loading the dynamic module

The '`--module`' option is the only required flag of our program. (We have marked it "required" in *cmdline.ggo*, the oxymoron oddity of 'option' and 'required' notwithstanding.) The argument to '`--module`' is the file name of the problem specification module. The `dlopen()` dynamic linking loader expects to receive a full (relative or absolute) path to the file name, otherwise it looks for the module in a set of system-specific places (see the `dlopen()` documentation for details.) In our program we allow to user to the enter the name of the module file with or without a path specification. If no path is specified, we prepend a `./` to the file name to make `dlopen()` read the module from the current directory. The string `module` holds the modified name:

⟨*declarations for main.c* 111c⟩+≡ (109a) ◁111c 112e▷
```
char *module;
```

⟨*parse command line options* 111d⟩+≡ (109a) ◁111d
```
if (strchr(args_info.module_arg, '/') == NULL) {
    MAKE_VECTOR(module, strlen(args_info.module_arg) + 3);
    sprintf(module, "./%s", args_info.module_arg);
} else {
    MAKE_VECTOR(module, strlen(args_info.module_arg) + 1);
    sprintf(module, "%s", args_info.module_arg);
}
```

The preprocessor macro `MAKE_VECTOR()` is defined in filenamearray.h:

⟨*included headers for main.c* 109b⟩+≡ (109a) ◁112a 113b▷
```
#include "array.h"
```

Now we call `dlopen()` to load the specified module:

⟨*declarations for main.c* 111c⟩+≡ (109a) ◁112b 113c▷
```
void *handle;
```

⟨*load problem module* 112f⟩≡ (109a) 113a▷
```
handle = dlopen(module, RTLD_LAZY);
if (handle == NULL)
    ABORT("%s\n", dlerror());
```

The `module` string is no longer needed therefore we free the memory associated with it:

⟨*load problem module* 112f⟩+≡ (109a) ◁112f 113d▷
```
FREE_VECTOR(module);
```

The only externally visible object in the module is the function `get_fem()` which returns a structure that contains the complete problem specification. We call `dlsym()` to locate that object within the module and assign it to a local pointer which we also name `get_fem`:

⟨*included headers for main.c* 109b⟩+≡ (109a) ◁112d 114d▷
```
#include "fem.h"
```

⟨*declarations for main.c* 111c⟩+≡ (109a) ◁112e 113e▷
```
Mesh *(*get_fem)(double);
char *error;
```

⟨*load problem module* 112f⟩+≡ (109a) ◁113a
```
*(void **)(&get_fem) = dlsym(handle, "get_fem");
if ((error = dlerror()) != NULL)
    ABORT("%s\n", error);
```

The ugly cast in the chunk above is not really necessary but it makes the program conform to standard C. The problem is, `dlsym()` returns a 'pointer to a function' cast to 'poiter to void'. Standard C forbids assigning a 'pointer to void' to a 'pointer to a function'. Because of this, our compiler, *gcc*, issues a warning when compiling the program in the strict mode with the `-pedantic` flag. The tricky cast comes from an example in `dlsym()`'s manual page distributed with Linux.

8.4 Running the program

The program's '`--area`' flag, which takes a required a numerical argument, *max_area*, specifies an upper bound to the area that a mesh triangle may have. The smaller the value, the finer the mesh.

In the file *cmdline.ggo* we have given `default="1.0"` for the value of the '`--area`' flag which means that if the user does not give the '`--area`' flag, then *max_area* is taken to be 1.0.

Here we extract the *max_area* value received by the parser and store it in a variable named `max_area`:

⟨*declarations for main.c* 111c⟩+≡ (109a) ◁113c 114a▷
```
double max_area;
```

⟨*perform user requested action* 113f⟩≡ (109a) 114b▷
```
max_area = args_info.area_arg;
if (max_area <= 0.0)
    ABORT("argument to ‘--area’ should be positive");
```

Now call `get_fem()` to read the problem specification, mesh the domain, and produce the `fem` structure:

⟨*declarations for main.c* 111c⟩+≡ (109a) ◁113e
```
Mesh *mesh;
```

⟨*perform user requested action* 113f⟩+≡ (109a) ◁113f 114c▷
```
mesh = get_fem(max_area);
```

There is no more need for the problem specification module therefore we unlink it:

⟨*perform user requested action* 113f⟩+≡ (109a) ◁114b 114e▷
```
dlclose(handle);
```

Now we look at the remaining command line options and perform the actions requested by the user:

⟨*included headers for main.c* 109b⟩+≡ (109a) ◁113b
```
#include "show_mesh_in_geomview.h"
#include "dump_mesh.h"
```

⟨*perform user requested action* 113f⟩+≡ (109a) ◁114c
```
if (args_info.show_mesh_given) {
    show_mesh_in_geomview(mesh);
    return EXIT_SUCCESS;
} else if (args_info.dump_mesh_given)
    dump_mesh(mesh);
else
    fprintf(stderr, "Nothing to do?\n");
```

Finally we clean up and exit:

⟨*clean up* 114f⟩≡ (109a)
```
cmdline_parser_free(&args_info);
return EXIT_SUCCESS;
```

Appendix A

Argyris shape function via *Maple*

Maple's notation is sufficiently close to that of standard mathematics so as to make many of its commands described in this section understandable to those unfamiliar with it. We will explain the meanings of some of the more intricate constructions.

A *Maple* script is a sequence of statements which are executed in the given order. *Maple* statements are terminated by a colon or semicolon. In a typical interactive *Maple* session, the result of evaluation of each statement is printed to screen if the terminator is a semicolon. The printing is suppressed if the terminator is a colon. The latter is useful to avoid cluttering the screen with the printout of lengthy intermediate results. The special variable "%" holds the result of the previous statement. The assignment operator is "`:=`". Thus "`x := 5;`" assigns the value 5 to the variable x.

Maple distinguishes between a function f and its value, just as one does in ordinary mathematics. To evaluate a function $f : \mathbf{R}^3 \to \mathbf{R}$ a at a generic point (x, y, z), we write "`apply(f, [x,y,z])`" or simply "`f(x,y,z)`", the latter variant being identical to the usual mathematical notation. The result of the application is an *expression* involving the three variables x, y and z. Conversely, given an expression `expr` in variables x, y and z, we may produce a corresponding function $f : \mathbf{R}^3 \to \mathbf{R}$ by "`f := unapply(expr, [x,y,z])`". There is no standard mathematical notation for this operation.

We begin our script by defining a generic fifth degree polynomial function p in three variables:

$$p(\lambda_1, \lambda_2, \lambda_3) = \sum_{k=0}^{5} \sum_{i=0}^{k} \sum_{j=0}^{k-i} C_{i,j,k-i-j} \lambda_1^i \lambda_2^j \lambda_3^{k-i-j}.$$

⟨*argyris-elem.maple* 115⟩≡ 116a▷

```
add(add(add(C[i,j,k-i-j] * lambda1^i * lambda2^j * lambda3^(k-i-j),
        j=0..k-i), i=0..k), k=0..5):
p := unapply(%, [lambda1, lambda2, lambda3]);
```

The `nops` command returns the number of "terms" in a *Maple* object. We may apply `nops` to verify that the polynomial $p(\lambda_1, \lambda_2, \lambda_3)$ is a sum of 56 monomials:

⟨*argyris-elem.maple* 115⟩+≡ ◁115 116b▷

```
nops(p(lambda1,lambda2,lambda3));
```

We isolate those terms of $p(\lambda_1, \lambda_2, \lambda_3)$ that are free of λ_3. There are 21 of them:

⟨*argyris-elem.maple* 115⟩+≡ ◁116a 116c▷

```
sub_terms := remove(has,
        convert(p(lambda1,lambda2,lambda3), list), lambda3);
nops(sub_terms);
```

Then make a list of the C_{ijk} coefficients of those terms:

⟨*argyris-elem.maple* 115⟩+≡ ◁116b 116d▷

```
vars := map(coeffs, sub_terms, [lambda1,lambda2]);
```

The 21 coefficients stored in `var` are:

$$C_{0,0,0}, C_{0,1,0}, C_{0,2,0}, C_{0,3,0}, C_{0,4,0}, C_{0,5,0}, C_{1,0,0}, C_{1,1,0}, C_{1,2,0}, C_{1,3,0}, C_{1,4,0},$$
$$C_{2,0,0}, C_{2,1,0}, C_{2,2,0}, C_{2,3,0}, C_{3,0,0}, C_{3,1,0}, C_{3,2,0}, C_{4,0,0}, C_{4,1,0}, C_{5,0,0},$$

Next, we introduce the three edge vectors $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ (see Figure 4.1):

⟨*argyris-elem.maple* 115⟩+≡ ◁116c 116e▷

```
a := <a1,a2>;   b := <b1,b2>;   c := <c1,c2>;
```

The vertices $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ enter through their barycentric coordinates $(1, 0, 0)$, $(0, 1, 0)$, $(0, 0, 1)$, therefore they receive no explicit mention in the computation.

We are ready to construct the system of 21 equations that determine the Argyris shape function P_1. The value of p is 1 at the vertex $\mathbf{u}$ and 0 at the other two vertices:

⟨*argyris-elem.maple* 115⟩+≡ ◁116d 116f▷

```
sys := {
    p(1,0,0) = 1,
    p(0,1,0) = 0,
    p(0,0,1) = 0,
```

The two first derivatives of p at each vertex in the direction of edges that meet at that vertex are zero. According to (4.11a), (4.11b), (4.11c):

⟨*argyris-elem.maple* 115⟩+≡ ◁116e 117a▷

```
    (D[2] - D[1])(p)(1,0,0) = 0,
    (D[1] - D[3])(p)(1,0,0) = 0,

    (D[3] - D[2])(p)(0,1,0) = 0,
```

```
      (D[2] - D[1])(p)(0,1,0) = 0,

      (D[1] - D[3])(p)(0,0,1) = 0,
      (D[3] - D[2])(p)(0,0,1) = 0,
```

The three second derivatives of p at each vertex in the directions of edges that meet at that vertex are zero. According to (4.11d), (4.11e), (4.11f):

⟨*argyris-elem.maple* 115⟩+≡ ◁116f 117b▷

```
      (D[1,1] - 2*D[1,2] + D[2,2])(p)(1,0,0) = 0,
      (D[2,2] - 2*D[2,3] + D[3,3])(p)(1,0,0) = 0,
      (D[3,3] - 2*D[3,1] + D[1,1])(p)(1,0,0) = 0,

      (D[1,1] - 2*D[1,2] + D[2,2])(p)(0,1,0) = 0,
      (D[2,2] - 2*D[2,3] + D[3,3])(p)(0,1,0) = 0,
      (D[3,3] - 2*D[3,1] + D[1,1])(p)(0,1,0) = 0,

      (D[1,1] - 2*D[1,2] + D[2,2])(p)(0,0,1) = 0,
      (D[2,2] - 2*D[2,3] + D[3,3])(p)(0,0,1) = 0,
      (D[3,3] - 2*D[3,1] + D[1,1])(p)(0,0,1) = 0,
```

Derivatives at midpoints of the three edges, in the direction of outward normal to the respective edge are zero. According to (4.14):

⟨*argyris-elem.maple* 115⟩+≡ ◁117a 117c▷

```
      - a^%T . (
            a*D[1](p)(0,1/2,1/2)
          + b*D[2](p)(0,1/2,1/2)
          + c*D[3](p)(0,1/2,1/2) ) = (a^%T . a) * 0,

      - b^%T . (
            a*D[1](p)(1/2,0,1/2)
          + b*D[2](p)(1/2,0,1/2)
          + c*D[3](p)(1/2,0,1/2) ) = (b^%T . b) * 0

      - c^%T . (
            a*D[1](p)(1/2,1/2,0)
          + b*D[2](p)(1/2,1/2,0)
          + c*D[3](p)(1/2,1/2,0) ) = (c^%T . c) * 0,
  }:
```

where the notation `c^%T` means the transpose of the vector `c`.

This completes the construction of the system of 21 equations for the shape function P_1. To construct a corresponding system for the shape function P_k for $k = 2, \ldots, 21$, move the "1" from the right hand side of the first equation to the right hand side of kth equation.

Now we call *Maple*'s `solve` to solve the system of 21 equations in `sys` for the 21 unknowns in `vars`:

⟨*argyris-elem.maple* 115⟩+≡ ◁117b 118a▷

```
sol := solve(sys, vars):
```

The solution in `sol` consists of a set of 21 equations of the form:

$$C_{0,0,0} = \cdots, \qquad C_{0,1,0} = \cdots, \qquad \cdots, \qquad C_{5,0,0} = \cdots$$

for each of the 21 C_{ijk} coefficients that were shown earlier. The "$\cdots$" are huge expressions involving the remaining 35 C_{ijk} coefficients as well as a_1, a_2, b_1, b_2, c_1, c_2 and λ_1, λ_2, λ_3. We change these equalities into assignments by applying the `assign()` function:

⟨*argyris-elem.maple* 115⟩+≡ ◁117c 118b▷

```
assign(sol);
```

This removes 21 of the 56 C_{ijk} coefficients in $p(\lambda_1, \lambda_2, \lambda_3)$. We can verify that some C_{ijk} coefficients remain by asking *Maple*:

⟨*argyris-elem.maple* 115⟩+≡ ◁118a 118c▷

```
has(p(lambda1,lambda2,lambda3), C);
```

to which it responds "*true*".

The huge expressions noted above simplify considerably by applying the equations $\lambda_1 + \lambda_2 + \lambda_3 = 1$ and $\mathbf{a} + \mathbf{b} + \mathbf{c} = \mathbf{0}$ to eliminate λ_3 and $\mathbf{c}$:

⟨*argyris-elem.maple* 115⟩+≡ ◁118b 118d▷

```
P := simplify(subs(c1=-a1-b1,c2=-a2-b2,
    p(lambda1,lambda2,1-lambda1-lambda2))):
```

As we could have expected, there are no more C_{ikj} coefficients in the resulting expression. We ask *Maple*:

⟨*argyris-elem.maple* 115⟩+≡ ◁118c 118e▷

```
has(P, C);
```

to which it responds "*false*".

We rearrange `P` into a sum of monomials in λ_1 and λ_2:

⟨*argyris-elem.maple* 115⟩+≡ ◁118d 118f▷

```
P := collect(P, [lambda1,lambda2], distributed, factor);
```

The result, although not immense, will fill up about half a printed page therefore we won't show it here. A close inspection of it, however, exhibits many recognizable combinations such as $a_1^2 + a_2^2$ and $a_1 b_1 + a_2 b_2$. We perform extensive manual simplification and rearrangement of terms of `P`, and call the result `P1`:

⟨*argyris-elem.maple* 115⟩+≡ ◁118e 119▷

```
r :=  (a^%T . b) / (b^%T . b);
s :=  (c^%T . a) / (c^%T . c);
P1 := lambda1^2 * (6*lambda1^3 - 15*lambda1^2
        + 10*lambda1 - 30*lambda2*lambda3 * (s*lambda2+r*lambda3));
```

The typeset form of `P1` is:

$$P_1 = \lambda_1^2\big(6\lambda_1^3 - 15\lambda_1^2 + 10\lambda_1 - 30\lambda_2\lambda_3(s\lambda_2 + r\lambda_3)\big),$$

where $r = (\mathbf{a}\cdot\mathbf{b})/\|\mathbf{b}\|^2$ and $s = (\mathbf{c}\cdot\mathbf{a})/\|\mathbf{c}\|^2$.

Note that we have restored the parameters c_1, c_2 and λ_3 which were eliminated in the computation of `P`. Finally, to verify that no errors were introduced in the process of manual manipulation, we compute the difference between `P1` and `P`:

⟨*argyris-elem.maple* 115⟩+≡ ◁118f

```
simplify(subs(c1=-a1-b1, c2=-a2-b2, lambda3 =1-lambda1-lambda2,
        P1 - P));
```

If all is well, *Maple* will respond with "0".

The sequence of *Maple* commands described in this section are stored in the script file *argyris-elem.maple*. At the *Maple* command type:

```
read "argyris-elem.maple";
```

to read and execute the script. This will compute `P1`. You may edit the file according to the instruction in the paragraph following the construction of the script's `sys` variable to compute the Argyris shape functions `P2` through `P21`. Actually it is sufficient to compute `P1` through `P7`; the rest may be obtained from these by cyclic permutations of variables. Here are the *simplified forms* of the first seven shape functions:

$$\begin{aligned}
P_1 &= \lambda_1^2\big(6\lambda_1^3 - 15\lambda_1^2 + 10\lambda_1 - 30\lambda_2\lambda_3(s\lambda_2 + r\lambda_3)\big) \\
&\qquad\qquad \text{where } r = (\mathbf{a}\cdot\mathbf{b})/\|\mathbf{b}\|^2 \text{ and } s = (\mathbf{c}\cdot\mathbf{a})/\|\mathbf{c}\|^2, \\
P_2 &= -\lambda_1^2\lambda_2(3\lambda_1^2 - 4\lambda_1 + 5\lambda_3^2 - r\lambda_2\lambda_3) \qquad \text{where } r = 12 + 7(\mathbf{b}\cdot\mathbf{c})/\|\mathbf{c}\|^2, \\
P_3 &= -\lambda_1^2\lambda_3(3\lambda_1^2 - 4\lambda_1 + 5\lambda_2^2 - r\lambda_2\lambda_3) \qquad \text{where } r = 12 + 7(\mathbf{b}\cdot\mathbf{c})/\|\mathbf{b}\|^2, \\
P_4 &= -\lambda_1^2\lambda_2(\lambda_2^2 + \lambda_1 - 1 + 2\lambda_3^2 - r\lambda_2\lambda_3)/2 \qquad \text{where } r = (\mathbf{b}\cdot\mathbf{c})/\|\mathbf{c}\|^2, \\
P_5 &= -\lambda_1^2\lambda_3(\lambda_3^2 + \lambda_1 - 1 + 2\lambda_2^2 - r\lambda_2\lambda_3)/2 \qquad \text{where } r = (\mathbf{b}\cdot\mathbf{c})/\|\mathbf{b}\|^2, \\
P_6 &= \lambda_1^2(1 - 2\lambda_1)\lambda_2\lambda_3/2, \\
P_7 &= -16\lambda_1\lambda_2^2\lambda_3^2.
\end{aligned}$$

A.1 The C interface to Argyris shape functions

In our finite element solver we need to evaluate the Argyris shape functions P_1 through P_{21} at prescribed quadrature points on each element. (The quadrature points are given in terms of the barycentric coordinates λ_1, λ_2, λ_3). In addition to the value of the function, the solver needs the values of its three first derivatives and six second derivatives—a total of 10 values per shape function per point. We find it convenient to calculate all the required values of the shape

functions only once, during the initialization stage of the program, and store them in an array of the type "`double Pvals[N][21][10]`" where N is the number of points at which the functions will be evaluated. The first index in `Pvals` refers to the evaluation point. The second index refers to the shape function P_i. Note that since array indices in C are zero-based, the shape function P_1 through P_{21} are actually indexed 0 through 20.

For the third index, which refers to the type of derivative, we use a mnemonic device to access entries in a convenient way. We let:

⟨*derivative codes* 120a⟩≡ (128)
```
enum which_deriv {
        D000, D100, D010, D001, D200, D020, D002, D110, D011, D101 };
```

where `Dijk` indicates the $\partial^{i+j+k}/\partial\lambda_1^i\partial\lambda_2^j\partial\lambda_3^k$ derivative. Thus `Pvals[42][13][D110]` holds the value of $\partial^2 P_{14}/\partial\lambda_1\partial\lambda_2$ at the point indexed 42.

In an N-point quadrature, the total storage needed for the `Pvals` array is: `N*21*10*sizeof(double)`. For instance, if `N` is 78 (corresponding to a quadrature strength of 20; see Appendix D) and `sizeof(double)` is 8, then the storage is $78 \times 21 \times 10 \times 8 = 131040$ bytes or approximately 130 kilobytes. We need to compute the `Pvals` array for every element in the triangulated domain, however we won't need its values for more than one element at a time, therefore the storage for `Pvals` is fixed, independent of the number of elements.

The `Pvals` array is computed by calling the function function `get_shape_data()` which receives a pointer to an element, a pointer to an array of barycentric coordinates, and a pointer to a pre-allocated array in which it calculates and stores the values of P_i and its derivatives. The prototype of `get_shape_data()` is:

⟨*prototype of get_shape_data* 120b⟩≡
```
void get_shape_data(Elem *elem, Point3d *points, int npoints, double ***Pvals);
```

The body of the function `get_shape_data()` consists of hand-crafted formulas obtained by manually simplifying *Maple*'s output, as described above. The function is quite large but has a simple and repetitive structure. We have chosen not to split it into multiple smaller function for efficiency; in this combined form the calculations of `aa`, ..., `ca` and `x`, ...`z3` (see the material near the top of the function's listing) are shared among the 21 subsequent blocks that compute the data for P_1 to P_{21}.

We have relegated the lengthy listing of the body of `get_shape_data()` function to the end of this chapter in order not to disturb the flow of exposition.

A.2 Computing the basis functions

Equation (4.21) gives the

A.3 The function get_shape_data()

This section contains the complete listing of the function `get_shape_data()` which was described in Section A.1. Its purpose to evaluate the Argyris shape functions $P_i(\lambda_1, \lambda_2, \lambda_3)$, $i = 1, \ldots, 21$ and their first and second derivatives (210 values) at a presecribed set of points. The expressions for P_i and their derivatives are hand-crafted formulas obtained by simplifying *Maple*'s output manually as described in Section A.

Alert! Within the body of `get_shape_data()`, we use the single-letter identifiers `x`, `y` and `z` for barycentric coordinates `lambda1`, `lambda2` and `lambda3` to maintain some readability in the code; *they should not be confused for Cartesian coordinates.*

⟨*function get_shape_data()* 121⟩≡ (127)

```
void get_shape_data(Elem *elem, Point3d *points, int npoints, double ***Pvals)
{
    double u1 = elem->n[0]->x;
    double u2 = elem->n[0]->y;
    double v1 = elem->n[1]->x;
    double v2 = elem->n[1]->y;
    double w1 = elem->n[2]->x;
    double w2 = elem->n[2]->y;
    double a1 = w1-v1;
    double a2 = w2-v2;
    double b1 = u1-w1;
    double b2 = u2-w2;
    double c1 = v1-u1;
    double c2 = v2-u2;
    double aa = a1*a1+a2*a2;
    double bb = b1*b1+b2*b2;
    double cc = c1*c1+c2*c2;
    double ab = a1*b1+a2*b2;
    double bc = b1*c1+b2*c2;
    double ca = c1*a1+c2*a2;
    int m;

    for (m = 0; m < npoints; m++) {

        /*!!  NOTE: x, y, z stand for lambda1, lambda2, lambda3 !!*/

        double x = points[m].x;
        double y = points[m].y;
        double z = points[m].z;
        double x2 = x*x;
        double y2 = y*y;
        double z2 = z*z;
        double x3 = x2*x;
        double y3 = y2*y;
```

```
double z3 = z2*z;


{ /* P1 */
    double r = ab/bb;
    double s = ca/cc;
    Pvals[m][0][D000] = x2*(6*x3-15*x2+10*x-30*y*z*(s*y+r*z));
    Pvals[m][0][D100] = -30*x*(-x3+2*x2-x+2*y*z*(s*y+r*z));
    Pvals[m][0][D010] = -30*x2*z*(r*z+2*s*y);
    Pvals[m][0][D001] = -30*x2*y*(s*y+2*r*z);
    Pvals[m][0][D200] = 60*(2*x3-3*x2+x-y*z*(s*y+r*z));
    Pvals[m][0][D020] = -60*s*x2*z;
    Pvals[m][0][D002] = -60*r*x2*y;
    Pvals[m][0][D110] = -60*x*z*(2*s*y+r*z);
    Pvals[m][0][D011] = -60*x2*(s*y+r*z);
    Pvals[m][0][D101] = -60*x*y*(s*y+2*r*z);
}

{ /* P2 */
    double r = 12 + 7*bc/cc;
    Pvals[m][1][D000] = -x2*y*(3*x2-4*x+5*z2-r*y*z);
    Pvals[m][1][D100] = -2*x*y*(6*x2-6*x+5*z2-r*y*z);
    Pvals[m][1][D010] = -x2*(3*x2-4*x+5*z2-2*r*y*z);
    Pvals[m][1][D001] =  -x2*y*(10*z-r*y);
    Pvals[m][1][D200] = -2*y*(18*x2-12*x+5*z2-r*y*z);
    Pvals[m][1][D020] = 2*r*x2*z;
    Pvals[m][1][D002] = -10*x2*y;
    Pvals[m][1][D110] = -2*x*(6*x2-6*x+5*z2-2*r*y*z);
    Pvals[m][1][D011] = -2*x2*(5*z-r*y);
    Pvals[m][1][D101] = -2*x*y*(10*z-r*y);
}

{ /* P3 */
    double r = 12 + 7*bc/bb;
    Pvals[m][2][D000] = -x2*z*(3*x2-4*x+5*y2-r*y*z);
    Pvals[m][2][D100] = -2*x*z*(6*x2-6*x+5*y2-r*y*z);
    Pvals[m][2][D010] = -x2*z*(10*y-r*z);
    Pvals[m][2][D001] = -x2*(3*x2-4*x+5*y2-2*r*y*z);
    Pvals[m][2][D200] = -2*z*(18*x2-12*x+5*y2-r*y*z);
    Pvals[m][2][D020] = -10*x2*z;
    Pvals[m][2][D002] = 2*r*x2*y;
    Pvals[m][2][D110] = -2*x*z*(10*y-r*z);
    Pvals[m][2][D011] = -2*x2*(5*y-r*z);
    Pvals[m][2][D101] = -2*x*(6*x2-6*x+5*y2-2*r*y*z);
}

{ /* P4 */
    double r = bc/cc;
    Pvals[m][3][D000] = -x2*y*(y2+x-1+2*z2-r*y*z)/2;
    Pvals[m][3][D100] = -x*y*(2*y2+3*x-2+4*z2-2*r*y*z)/2;
```

```
    Pvals[m][3][D010] = -x2*(3*y2+x-1+2*z2-2*r*y*z)/2;
    Pvals[m][3][D001] = -x2*y*(4*z-r*y)/2;
    Pvals[m][3][D200] = -y*(y2+3*x-1+2*z2-r*y*z);
    Pvals[m][3][D020] = -x2*(3*y-r*z);
    Pvals[m][3][D002] = -2*x2*y;
    Pvals[m][3][D110] = -x*(6*y2+3*x-2+4*z2-4*r*y*z)/2;
    Pvals[m][3][D011] =  -x2*(2*z-r*y);
    Pvals[m][3][D101] = -x*y*(4*z-y*r);
}

{ /* P5 */
    double r = bc/bb;
    Pvals[m][4][D000] = -x2*z*(z2+x-1+2*y2-r*y*z)/2;
    Pvals[m][4][D100] = -x*z*(4*y2+3*x+2*z2-2-2*r*y*z)/2;
    Pvals[m][4][D010] =  -x2*z*(4*y-r*z)/2;
    Pvals[m][4][D001] = -x2*(2*y2+x+3*z2-1-2*r*y*z)/2;
    Pvals[m][4][D200] = -z*(2*y2+z2+3*x-1-r*y*z);
    Pvals[m][4][D020] = -2*x2*z;
    Pvals[m][4][D002] = -x2*(3*z-r*y);
    Pvals[m][4][D110] = -x*z*(4*y-r*z);
    Pvals[m][4][D011] = -x2*(2*y-r*z);
    Pvals[m][4][D101] = -x*(4*y2+3*x+6*z2-2-4*r*y*z)/2;
}

{ /* P6 */
    Pvals[m][5][D000] = x2*(1-2*x)*y*z/2;
    Pvals[m][5][D100] = x*(1-3*x)*y*z;
    Pvals[m][5][D010] = x2*(1-2*x)*z/2;
    Pvals[m][5][D001] = x2*(1-2*x)*y/2;
    Pvals[m][5][D200] = (1-6*x)*y*z;
    Pvals[m][5][D020] = 0;
    Pvals[m][5][D002] = 0;
    Pvals[m][5][D110] = x*(1-3*x)*z;
    Pvals[m][5][D011] = x2*(1-2*x)/2;
    Pvals[m][5][D101] = x*(1-3*x)*y;
}

{ /* P7 */
    double r = bc/cc;
    double s = ab/aa;
    Pvals[m][6][D000] = y2*(6*y3-15*y2+10*y-30*z*x*(s*z+r*x));
    Pvals[m][6][D100] = -30*y2*z*(s*z+2*r*x);
    Pvals[m][6][D010] = -30*y*(-y3+2*y2-y+2*z*x*(s*z+r*x));
    Pvals[m][6][D001] = -30*y2*x*(r*x+2*s*z);
    Pvals[m][6][D200] = -60*r*y2*z;
    Pvals[m][6][D020] = 60*(2*y3-3*y2+y-z*x*(s*z+r*x));
    Pvals[m][6][D002] = -60*s*y2*x;
    Pvals[m][6][D110] = -60*y*z*(s*z+2*r*x);
    Pvals[m][6][D011] = -60*y*x*(2*s*z+r*x);
    Pvals[m][6][D101] = -60*y2*(s*z+r*x);
```

```
}

{ /* P8 */
    double r = 12 + 7*ca/aa;
    Pvals[m][7][D000] = -y2*z*(3*y2-4*y+5*x2-r*z*x);
    Pvals[m][7][D100] =  -y2*z*(10*x-r*z);
    Pvals[m][7][D010] = -2*y*z*(6*y2-6*y+5*x2-r*z*x);
    Pvals[m][7][D001] = -y2*(3*y2-4*y+5*x2-2*r*z*x);
    Pvals[m][7][D200] = -10*y2*z;
    Pvals[m][7][D020] = -2*z*(18*y2-12*y+5*x2-r*z*x);
    Pvals[m][7][D002] = 2*r*y2*x;
    Pvals[m][7][D110] = -2*y*z*(10*x-r*z);
    Pvals[m][7][D011] = -2*y*(6*y2-6*y+5*x2-2*r*z*x);
    Pvals[m][7][D101] = -2*y2*(5*x-r*z);
}

{ /* P9 */
    double r = 12 + 7*ca/cc;
    Pvals[m][8][D000] = -y2*x*(3*y2-4*y+5*z2-r*z*x);
    Pvals[m][8][D100] = -y2*(3*y2-4*y+5*z2-2*r*z*x);
    Pvals[m][8][D010] = -2*y*x*(6*y2-6*y+5*z2-r*z*x);
    Pvals[m][8][D001] = -y2*x*(10*z-r*x);
    Pvals[m][8][D200] = 2*r*y2*z;
    Pvals[m][8][D020] = -2*x*(18*y2-12*y+5*z2-r*z*x);
    Pvals[m][8][D002] = -10*y2*x;
    Pvals[m][8][D110] = -2*y*(6*y2-6*y+5*z2-2*r*z*x);
    Pvals[m][8][D011] = -2*y*x*(10*z-r*x);
    Pvals[m][8][D101] = -2*y2*(5*z-r*x);
}

{ /* P10 */
    double r = ca/aa;
    Pvals[m][9][D000] = -y2*z*(z2+y-1+2*x2-r*z*x)/2;
    Pvals[m][9][D100] = -y2*z*(4*x-r*z)/2;
    Pvals[m][9][D010] = -y*z*(2*z2+3*y-2+4*x2-2*r*z*x)/2;
    Pvals[m][9][D001] = -y2*(3*z2+y-1+2*x2-2*r*z*x)/2;
    Pvals[m][9][D200] = -2*y2*z;
    Pvals[m][9][D020] = -z*(z2+3*y-1+2*x2-r*z*x);
    Pvals[m][9][D002] = -y2*(3*z-r*x);
    Pvals[m][9][D110] = -y*z*(4*x-z*r);
    Pvals[m][9][D011] = -y*(6*z2+3*y-2+4*x2-4*r*z*x)/2;
    Pvals[m][9][D101] =  -y2*(2*x-r*z);
}

{ /* P11 */
    double r = ca/cc;
    Pvals[m][10][D000] = -y2*x*(x2+y-1+2*z2-r*z*x)/2;
    Pvals[m][10][D100] = -y2*(2*z2+y+3*x2-1-2*r*z*x)/2;
    Pvals[m][10][D010] = -y*x*(4*z2+3*y+2*x2-2-2*r*z*x)/2;
    Pvals[m][10][D001] =  -y2*x*(4*z-r*x)/2;
```

```
    Pvals[m][10][D200] = -y2*(3*x-r*z);
    Pvals[m][10][D020] = -x*(2*z2+x2+3*y-1-r*z*x);
    Pvals[m][10][D002] = -2*y2*x;
    Pvals[m][10][D110] = -y*(4*z2+3*y+6*x2-2-4*r*z*x)/2;
    Pvals[m][10][D011] = -y*x*(4*z-r*x);
    Pvals[m][10][D101] = -y2*(2*z-r*x);
}

{ /* P12 */
    Pvals[m][11][D000] = y2*(1-2*y)*z*x/2;
    Pvals[m][11][D100] = y2*(1-2*y)*z/2;
    Pvals[m][11][D010] = y*(1-3*y)*z*x;
    Pvals[m][11][D001] = y2*(1-2*y)*x/2;
    Pvals[m][11][D200] = 0;
    Pvals[m][11][D020] = (1-6*y)*z*x;
    Pvals[m][11][D002] = 0;
    Pvals[m][11][D110] = y*(1-3*y)*z;
    Pvals[m][11][D011] = y*(1-3*y)*x;
    Pvals[m][11][D101] = y2*(1-2*y)/2;
}

{ /* P13 */
    double r = ca/aa;
    double s = bc/bb;
    Pvals[m][12][D000] = z2*(6*z3-15*z2+10*z-30*x*y*(s*x+r*y));
    Pvals[m][12][D100] = -30*z2*y*(r*y+2*s*x);
    Pvals[m][12][D010] = -30*z2*x*(s*x+2*r*y);
    Pvals[m][12][D001] = -30*z*(-z3+2*z2-z+2*x*y*(s*x+r*y));
    Pvals[m][12][D200] = -60*s*z2*y;
    Pvals[m][12][D020] = -60*r*z2*x;
    Pvals[m][12][D002] = 60*(2*z3-3*z2+z-x*y*(s*x+r*y));
    Pvals[m][12][D110] = -60*z2*(s*x+r*y);
    Pvals[m][12][D011] = -60*z*x*(s*x+2*r*y);
    Pvals[m][12][D101] = -60*z*y*(2*s*x+r*y);
}

{ /* P14 */
    double r = 12 + 7*ab/bb;
    Pvals[m][13][D000] = -z2*x*(3*z2-4*z+5*y2-r*x*y);
    Pvals[m][13][D100] = -z2*(3*z2-4*z+5*y2-2*r*x*y);
    Pvals[m][13][D010] =  -z2*x*(10*y-r*x);
    Pvals[m][13][D001] = -2*z*x*(6*z2-6*z+5*y2-r*x*y);
    Pvals[m][13][D200] = 2*r*z2*y;
    Pvals[m][13][D020] = -10*z2*x;
    Pvals[m][13][D002] = -2*x*(18*z2-12*z+5*y2-r*x*y);
    Pvals[m][13][D110] = -2*z2*(5*y-r*x);
    Pvals[m][13][D011] = -2*z*x*(10*y-r*x);
    Pvals[m][13][D101] = -2*z*(6*z2-6*z+5*y2-2*r*x*y);
}
```

```
{ /* P15 */
    double r = 12 + 7*ab/aa;
    Pvals[m][14][D000] = -z2*y*(3*z2-4*z+5*x2-r*x*y);
    Pvals[m][14][D100] = -z2*y*(10*x-r*y);
    Pvals[m][14][D010] = -z2*(3*z2-4*z+5*x2-2*r*x*y);
    Pvals[m][14][D001] = -2*z*y*(6*z2-6*z+5*x2-r*x*y);
    Pvals[m][14][D200] = -10*z2*y;
    Pvals[m][14][D020] = 2*r*z2*x;
    Pvals[m][14][D002] = -2*y*(18*z2-12*z+5*x2-r*x*y);
    Pvals[m][14][D110] = -2*z2*(5*x-r*y);
    Pvals[m][14][D011] = -2*z*(6*z2-6*z+5*x2-2*r*x*y);
    Pvals[m][14][D101] = -2*z*y*(10*x-r*y);
}

{ /* P16 */
    double r = ab/bb;
    Pvals[m][15][D000] = -z2*x*(x2+z-1+2*y2-r*x*y)/2;
    Pvals[m][15][D100] = -z2*(3*x2+z-1+2*y2-2*r*x*y)/2;
    Pvals[m][15][D010] = -z2*x*(4*y-r*x)/2;
    Pvals[m][15][D001] = -z*x*(2*x2+3*z-2+4*y2-2*r*x*y)/2;
    Pvals[m][15][D200] = -z2*(3*x-r*y);
    Pvals[m][15][D020] = -2*z2*x;
    Pvals[m][15][D002] = -x*(x2+3*z-1+2*y2-r*x*y);
    Pvals[m][15][D110] =  -z2*(2*y-r*x);
    Pvals[m][15][D011] = -z*x*(4*y-x*r);
    Pvals[m][15][D101] = -z*(6*x2+3*z-2+4*y2-4*r*x*y)/2;
}

{ /* P17 */
    double r = ab/aa;
    Pvals[m][16][D000] = -z2*y*(y2+z-1+2*x2-r*x*y)/2;
    Pvals[m][16][D100] =  -z2*y*(4*x-r*y)/2;
    Pvals[m][16][D010] = -z2*(2*x2+z+3*y2-1-2*r*x*y)/2;
    Pvals[m][16][D001] = -z*y*(4*x2+3*z+2*y2-2-2*r*x*y)/2;
    Pvals[m][16][D200] = -2*z2*y;
    Pvals[m][16][D020] = -z2*(3*y-r*x);
    Pvals[m][16][D002] = -y*(2*x2+y2+3*z-1-r*x*y);
    Pvals[m][16][D110] = -z2*(2*x-r*y);
    Pvals[m][16][D011] = -z*(4*x2+3*z+6*y2-2-4*r*x*y)/2;
    Pvals[m][16][D101] = -z*y*(4*x-r*y);
}

{ /* P18 */
    Pvals[m][17][D000] = z2*(1-2*z)*x*y/2;
    Pvals[m][17][D100] = z2*(1-2*z)*y/2;
    Pvals[m][17][D010] = z2*(1-2*z)*x/2;
    Pvals[m][17][D001] = z*(1-3*z)*x*y;
    Pvals[m][17][D200] = 0;
    Pvals[m][17][D020] = 0;
    Pvals[m][17][D002] = (1-6*z)*x*y;
```

```
            Pvals[m][17][D110] = z2*(1-2*z)/2;
            Pvals[m][17][D011] = z*(1-3*z)*x;
            Pvals[m][17][D101] = z*(1-3*z)*y;
        }

        { /* P19 */
            Pvals[m][18][D000] = -16*x*y2*z2;
            Pvals[m][18][D100] = -16*y2*z2;
            Pvals[m][18][D010] = -32*x*y*z2;
            Pvals[m][18][D001] = -32*x*y2*z;
            Pvals[m][18][D200] = 0;
            Pvals[m][18][D020] = -32*x*z2;
            Pvals[m][18][D002] = -32*x*y2;
            Pvals[m][18][D110] = -32*y*z2;
            Pvals[m][18][D011] = -64*x*y*z;
            Pvals[m][18][D101] = -32*y2*z;
        }

        { /* P20 */
            Pvals[m][19][D000] = -16*y*z2*x2;
            Pvals[m][19][D100] = -32*y*z2*x;
            Pvals[m][19][D010] = -16*z2*x2;
            Pvals[m][19][D001] = -32*y*z*x2;
            Pvals[m][19][D200] = -32*y*z2;
            Pvals[m][19][D020] = 0;
            Pvals[m][19][D002] = -32*y*x2;
            Pvals[m][19][D110] = -32*z2*x;
            Pvals[m][19][D011] = -32*z*x2;
            Pvals[m][19][D101] = -64*y*z*x;
        }

        { /* P21 */
            Pvals[m][20][D000] = -16*z*x2*y2;
            Pvals[m][20][D100] = -32*z*x*y2;
            Pvals[m][20][D010] = -32*z*x2*y;
            Pvals[m][20][D001] = -16*x2*y2;
            Pvals[m][20][D200] = -32*z*y2;
            Pvals[m][20][D020] = -32*z*x2;
            Pvals[m][20][D002] = 0;
            Pvals[m][20][D110] = -64*z*x*y;
            Pvals[m][20][D011] = -32*x2*y;
            Pvals[m][20][D101] = -32*x*y2;
        }

    }
}
```

⟨*shape_functions.c* 127⟩≡

```
#include "shape_functions.h"
⟨function get_shape_data() 121⟩
```

⟨*shape_functions.h* 128⟩≡

```
#ifndef H_SHAPE_FUNCTIONS_H
#define H_SHAPE_FUNCTIONS_H

#include "twb-quad.h"
#include "fem.h"

⟨derivative codes 120a⟩

void get_shape_data(Elem *elem, Point3d *points, int npoints, double ***Pvals);

#endif /* H_SHAPE_FUNCTIONS_H */
```

Appendix B

Utility functions

The functions described in the chapter are not essential to FEM's operation. They provide support for producing graphics output and debugging information.

B.1 Front end to *Geomview* graphics

Geomview is a open source software for producing graphics on *X11 Window System* which is a common display protocol for Unix-like workstations. *Geomview* may be downloaded from free from ⟨`http://www.geomview.org/`⟩.

The function `show_mesh_in_geomview()` receives a pointer to a `Mesh` structure which points to a triangulated domain and writes a file named *zz.off* in *Geomview*'s **O**bject **F**ile **F**ormat, which then may be read and displayed by *Geomview*, by typing:

```
geomview zz.off
```

on the command line. By default, *Geomview* displays its graphics on a dark background. To produce graphics on a white background, replace the above with:

```
geomview -b 1 1 1 zz.off
```

Figure B.1 shows a sample result.

The general structure of the contents of a `*.off` file is as follows.

```
{ appearance { +edge }
    OFF
    ... data goes here ...
}
```

The part marked as`... data goes here ...` consists of three sections.

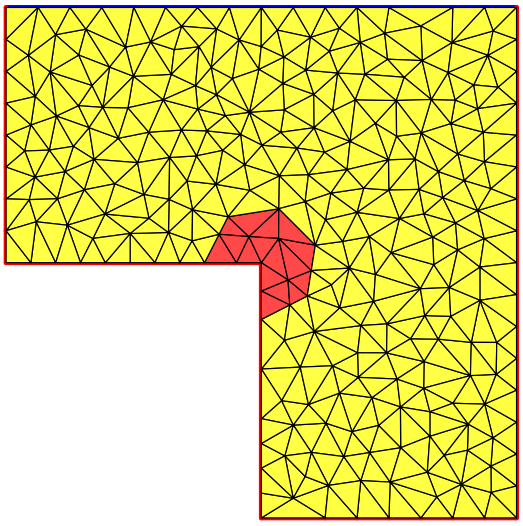


Figure B.1: This is the output of *Geomview* on the L-shaped domain described in the file *_ell-shaped.c* in Section 1.2. The mesh was created by `demo 0.01 > z1.off` and displayed with `geomview -b 1 1 1 z1.off`. There is one reentrant vertex. The vertex's local neighborhood is shown in red.

The first section consists of a single line containing three numbers, e.g.,

```
271 476 -1
```

The first number is the number of nodes (in the rest of this section we will call it n), the second is the number of triangles (in the rest of this section we will call it t), and third one is a dummy placeholder; it is read but not used. By the way, the specific numbers shown above correspond to the mesh in Figure B.1.

The second section consists of a sequence of n pairs of floating point numbers where each pair represents the x and y coordinates of a node. The numbers must be separated by whitespace (e.g., space, tab, newline) otherwise no special formatting requirement is imposed.

The third section consists of a sequence of t septuplets of numbers, one septuplet per triangle. The first element of each septuplet is the number 3, indicating that our objects are triangles. (An *OFF* file may define polygonal objects other than triangles.) The next 3 elements of a septuplet are the node numbers of the vertices of the triangle, specified in the counter-clockwise order. The final 3 elements in the septuplet are floating point numbers in the range $[0, 1]$ that specify the RGB components of the triangle's color.

Several variations in the contents of a `*.off` file are possible. Refer to the website

`http://www.geomview.org/docs/html/OFF.html`

for details.

Here is `show_mesh_in_geomview()`:

⟨*show_mesh_in_geomview.c-old* 131a⟩≡

```
#include <stdio.h>
#include "show_mesh_in_geomview.h"

void show_mesh_in_geomview(Mesh *mesh)
{
    int i;

    puts("{ appearance { +edge }");
    puts("OFF");
    printf("%d %d -1\n", mesh->nnodes, mesh->nelems);

    for (i = 0; i < mesh->nnodes; i++)
        printf("%g %g 0.0\n", mesh->nodes[i].x, mesh->nodes[i].y);

    for (i = 0; i < mesh->nelems; i++) {
        printf("3 %d %d %d ",
                mesh->elems[i].n[0]->nodeno,
                mesh->elems[i].n[1]->nodeno,
                mesh->elems[i].n[2]->nodeno);

        if (mesh->elems[i].marker == -1)
            puts("1.0 1.0 0.0");
        else
            puts("1.0 0.0 0.0");
    }

    puts("}");
}
```

An associated header file declares `show_mesh_in_geomview()`'s prototype:

⟨*show_mesh_in_geomview.h* 131b⟩≡

```
#ifndef H_SHOW_MESH_IN_GEOMVIEW_H
#define H_SHOW_MESH_IN_GEOMVIEW_H

#include "fem.h"

void show_mesh_in_geomview(Mesh *mesh);

#endif /* H_SHOW_MESH_IN_GEOMVIEW_H */
```

B.2 Cancel the previous section

I rewrote `show_mesh_in_geomview()` but haven't gotten around to change the doucmentation. Here is the replacement. But even this is temporary and will have to be rewritten after we analyze the boundary data more fully.

(REWRITE THIS SECTION)

⟨*show_mesh_in_geomview.c* 132⟩≡

```
#include <stdio.h>
#include "fem.h"
#include "abort.h"
#include "show_mesh_in_geomview.h"

static void draw_triangles(Mesh *mesh)
{
    int i;

    puts("OFF");
    printf("%d %d -1\n", mesh->nnodes, mesh->nelems);

    for (i = 0; i < mesh->nnodes; i++)
        printf("%g %g 0.0\n", mesh->nodes[i].x, mesh->nodes[i].y);

    for (i = 0; i < mesh->nelems; i++) {
        printf("3 %d %d %d ",
                mesh->elems[i].n[0]->nodeno,
                mesh->elems[i].n[1]->nodeno,
                mesh->elems[i].n[2]->nodeno);

        if (mesh->elems[i].marker == -1)
            puts("1.0 1.0 0.0");
        else
            puts("1.0 0.0 0.0");
    }
}


static void draw_edge(Edge *edge)
{
    double R, G, B;

    /*
    switch (edge->bc->bc_type) {
        case BC_DIRICHLET:
            R = 1.0; G = 0.0; B = 0.0;
            break;
        case BC_NEUMANN:
            R = 0.0; G = 0.0; B = 1.0;
            break;
```

```
            default:
                ABORT("shouldn't be here");
        }
        */

        switch(edge->bc->patch) {
            case 0:
                R = 1.0; G = 0.0; B = 0.0;  /* red */
                break;
            case 1:
                R = 0.0; G = 1.0; B = 0.0;  /* green */
                break;
            case 2:
                R = 0.0; G = 0.0; B = 1.0;  /* blue */
                break;
            case 3:
                R = 1.0; G = 0.0; B = 1.0;  /* magenta */
                break;
            case 4:
                R = 0.0; G = 1.0; B = 1.0;  /* cyan */
                break;
            default:    /* temporary, need to extend */
                R = 0.0; G = 0.0; B = 0.0;  /* black */
            }

        puts("{ SKEL");
        puts("2 1");
        printf("%g %g 0\n", edge->n1->x, edge->n1->y);
        printf("%g %g 0\n", edge->n2->x, edge->n2->y);
        printf("2 0 1 %g %g %g\n", R, G, B);
        puts("}");
    }

    static void draw_boundary_edges(Mesh *mesh)
    {
        int i;

        for (i = 0; i < mesh->nedges; i++) {
            Edge *edge = &mesh->edges[i];

            if (edge->bc != NULL)
                draw_edge(edge);
        }
    }

    void show_mesh_in_geomview(Mesh *mesh)
    {
        puts("LIST");

        puts("{ appearance { linewidth 4 }");
```

```
    puts("LIST");
    draw_boundary_edges(mesh);
    puts("}");

    puts("{ appearance { +edge }");
    draw_triangles(mesh);
    puts("}");
}
```

B.3 A dump of the `mesh` structure

The function `dump_mesh()` produces a formatted output of most of the data in a `Mesh` structure for debugging.

⟨*dump_mesh.c* 134a⟩≡

```
#include <stdio.h>
#include "fem.h"

⟨function dump_nodes 134b⟩
⟨function dump_edges 134c⟩
⟨function dump_elems 135a⟩
⟨function dump_mesh 135b⟩
```

⟨*function dump_nodes* 134b⟩≡ (134a)

```
static void dump_nodes(Mesh *mesh)
{
    int i;

    printf("--- Dump of %d nodes ---\n", mesh->nnodes);

    for (i = 0; i < mesh->nnodes; i++) {
        Node *np = &mesh->nodes[i];
        printf("node %3d: ( %8.2g, %8.2g )\n",
                np->nodeno, np->x, np->y);
    }
}
```

⟨*function dump_edges* 134c⟩≡ (134a)

```
static void dump_edges(Mesh *mesh)
{
    int i;

    printf("--- Dump of %d edges ---\n", mesh->nedges);

    for (i = 0; i < mesh->nedges; i++) {
        Edge *ep = &mesh->edges[i];
        printf("edge %3d:  (%3d -> %3d)  ",
```

```
                ep->edgeno, ep->n1->nodeno, ep->n2->nodeno);
        if (ep->bc != NULL)
            printf("patch=%d, bc=%d", ep->bc->patch, ep->bc->bc_type);
        putchar('\n');
    }
}
```

⟨*function dump_elems* 135a⟩≡ (134a)

```
static void dump_elems(Mesh *mesh)
{
    int i;

    printf("--- Dump of %d elems ---\n", mesh->nelems);

    for (i = 0; i < mesh->nelems; i++) {
        Elem *ep = &mesh->elems[i];
        printf("elem %3d  node [%3d, %3d, %3d]   edge [%3d, %3d, %3d]",
                ep->elemno,
                ep->n[0]->nodeno, ep->n[1]->nodeno, ep->n[2]->nodeno,
                ep->e[0]->edgeno, ep->e[1]->edgeno, ep->e[2]->edgeno);

        if (ep->marker != -1)
            printf("  (region %d)", ep->marker);

        putchar('\n');

    }
}
```

⟨*function dump_mesh* 135b⟩≡ (134a)

```
void dump_mesh(Mesh *mesh)
{
    dump_nodes(mesh);
    putchar('\n');
    dump_edges(mesh);
    putchar('\n');
    dump_elems(mesh);
}
```

⟨*dump_mesh.h* 135c⟩≡

```
#ifndef H_DUMP_MESH_H
#define H_DUMP_MESH_H

void dump_mesh(Mesh *mesh);

#endif /* H_DUMP_MESH_H */
```

Appendix C

The *Triangle* library

Triangle is an open-souce software tool for unstructured triangulation of two dimensional polygonal regions. It is written by JONATHAN RICHARD SHEWCHUK ⟨`jrs@cs.berkeley.edu`⟩ and may be downloaded from:

⟨`http://www.cs.cmu.edu/ quake/triangle.html`⟩.

Triangle generates high-quality unstructured triangular meshes suitable for finite element computations.

The *Triangle* distribution contains stand-alone programs that read domain data from a file and triangulate and display the mesh. It also contains a C library which may be linked with other programs to make *Triangle* functions available to it. In our work we use the library. In this section we describe the relevant parts of that library.

C.1 Describing a doamin

By a *domain* we mean a region in the two dimensional plane bounded by a polygon. The region may have hole, which are also polygonal. Figure C.1(a) shows a domain with two holes. A domain may be partitioned into *regions* (i.e., subdomains) bounded by straight line segments. Figure C.1(b) shows the previous domain now partitioned into two regions.

In *Triangle* the points that define the domain are called *nodes*. The line segments that connect the nodes are called *segments*. Thus the domain shown in Figure C.1(a) has 13 nodes and 13 segments. The domain shown in Figure C.1(b) has 16 nodes and 17 segments. Note that the introduction of the node at E splits the segment AB into two segments AE and EB.

Triangle respects the segments in the sense that it creates no triangle that straddles a segment. Figures C.1(c) and C.1(d) shows the result of trangulation the domains shown in Figures C.1(a) and C.1(b), resepectively. Observe how

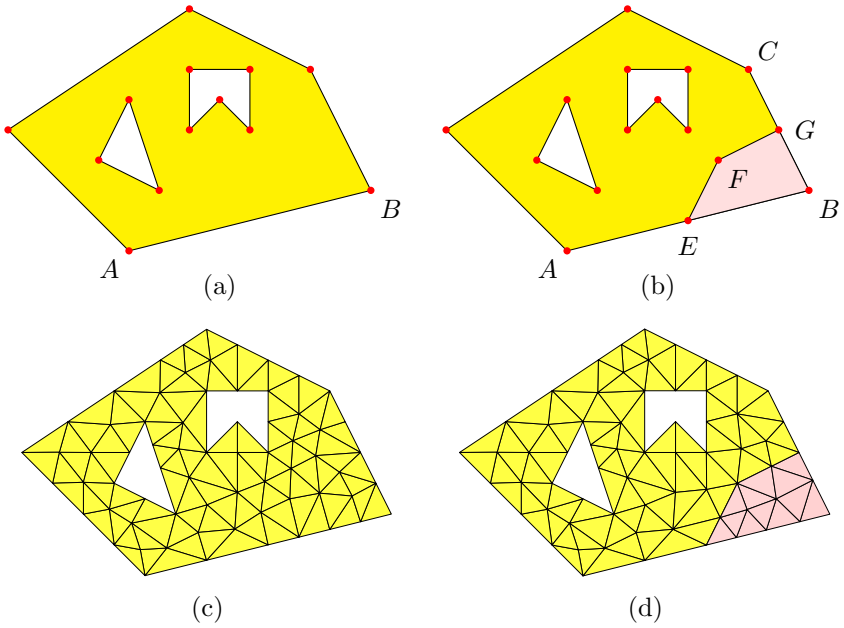


Figure C.1: The domain in (a) has 13 nodes, 13 segments and two holes. It consists of a single region. The domain in (b) is like domain (a) but has been divided into two regions by adding 3 new nodes E, F and G, two new segments EF and FG, and splitting the segments AB and BC into four segments AE, EB, BG, GC. It has 16 nodes and 17 segments. Triangulations of (a) and (b) result in domains in (c) and (d). Note that triangulation preserves the region boundary EFG.

the internal boundary between the two regions of Figures C.1(b) is preserved in Figures C.1(d).

Triangle vertices in the triangulated domain are *output nodes* to distinguish them from the originally specified nodes which are called *input nodes*. Input nodes form a subset of output nodes.

The introduction of new nodes subdivides the originally specified segments, called *input segments* into subsegments called *output segments*. For instance, comparing Figures C.1(b) and C.1(d) we see that each of the input segments AE and EF has been subdivided into two output segments. Input segments form a subset of output segments.

Line segments that interconnect the output nodes are called *edges*. Therefore output segments form a subset of edges. In effect, segments are boundary edges.

C.2 *Triangle*'s interface

Triangle's interface is defined in its header file *triangle.h* which is sufficiently commented to serve as the sole reference manual for the *Triangle* library. The file declares a C structure called `struct triangulateio` and the prototype for the function `triangulate()` which is the only interface to the *Triangle* library. The prototype of `triangulate()` is:

```
void triangulate(char *opts,
    struct triangulateio *in,
    struct triangulateio *out,
    struct triangulateio *vorout);
```

The `opts` arguments is a string of one-character switches which encode instructions to *Triangle* of what to do. A great deal of options are possible here. We will describe what we need for our work. Our encoded string is `QzpAjeq30a`x which is interpreted as:

Q Quiet operation; don't print out statistics.

z Node numbering begins with zero.

p The input data specifies a segment list.

A The input data specifies regional attributes.

j Discard duplicate vertices, if any.

e Produce edge list data.

q30 Accept no triangles with an angle less than 30 degrees.

ax Produce triangulation where no triangle has area greater than x.

The `vorout` argument is for producing Voronoi triangulation which we don't need therefore we set it to `NULL` in our calls `triangulate()`.

The `in` argument is a pointer to a `struct triangulateio` object, provided by the user, which contains the domain specification before triangulation.

The `out` argument is a pointer to a `struct triangulateio` object which is filled in by *Triangle* and contains the domain specification after triangulation.

The user is reponsible for allocating memory for the `in` and `out` structures and initializing some of those structures' members, as we describe in the following sections.

C.2.1 Initializing the `in` structure

The following members of the `in` structure need to be initialized. Storage must be allocated for arrays.

`int numberofpoints;`

The number of nodes (n) on input.

`double *pointlist;`

Pointer to an array of of length $2n$. The coordinates (x_i, y_i) of node number i are stored in the array's $2i$ and $2i + 1$ elements.

`int *pointmarkerlist;`

Pointer to an array of of length n. This is used to associate a user-defined number with each node. (Probably we won't need this in which case this item will be deleted.)

`int numberofsegments;`

Number of segments (s) on input.

`int *segmentlist;`

Pointer to an array of of length $2s$. Node numbers of the end points of the segment i are stored in in the array's $2i$ and $2i + 1$ elements.

`int *segmentmarkerlist;`

Pointer to an array of of length s. Associates an integer marker with each input segment. Typically these carry information on boundary conditions applied to that segment. When a segment is partitione into subsegments upon triangulation, subsegments inherit their parent's marker value.

`int numberofholes;`

Number of holes (h) in the domain.

`double *holelist;`

Pointer to an array of of length $2h$. The array's $2i$ and $2i+1$ elements hold the x and y coordinates of an arbitrary point inside the hole. If domain has not holes, set `numberofholes` to zero and `holelist` to `NULL`.

`int numberofregions;`

Number of regions (r) that the domain is divided into.

`double *regionlist;`

Pointer to an array of of length $4r$. The coordinates x and y of an arbitrary point inside region i are stored in the array's $4i$ and $4i + 1$ elements. The $4i+2$ element holds a region *attribute*, which is a user-specified number associated with the region. We use this slot to store a unique number that identifies the region. Upon triangulation, triangles that fall within the region inherit the region's attribute value. See the description of `triangleattributelist` in the next section.

The value in the $4i + 3$ element is not used by our program therefore we leave it uninitialized.

If domain is not divided into regions, set `numberofregions` to zero and `regionlist` to `NULL`.

C.2.2 Interpreting the `out` structure

In this section we describe the meanning of the members of the `struct triangulateio` object pointed to by `out` after returning from a call to `triangle()`.

Important: Since the lengths of the output arrays of are not known to the user before calling `triangle()`, it is best to leave it to *Triangle* to allocate memory for those arrays as needed. We request *Triangle* to do so by initializing the array pointer to `NULL`. These initializations are noted in the following description. Other members need not be initialized.

`int numberofpoints;`

The number of nodes (n) on output.

`double *pointlist;`

Pointer to an array of of length $2n$. The coordinates (x_i, y_i) of node number i are stored in the array's $2i$ and $2i + 1$ elements. Initialize to `NULL`.

`int *pointmarkerlist;`

Pointer to an array of of length n. Initialize to `NULL`. (Probably we won't need this in which case this item will be deleted.)

`int numberofsegments;`

Number of segments (s) on output.

`int *segmentlist;`

Pointer to an array of of length $2s$. Node numbers of the end points of segment i are stored in in the array's $2i$ and $2i + 1$ elements. Initialize to `NULL`.

`int *segmentmarkerlist;`

Pointer to an array of of length s. Segments inherit their parents' marker values that were specified on input. Initialize to `NULL`.

`int numberofholes;`

Number of holes (h) in the domain. This equals `numberofholes` specified on input since number of holes does not change upon triangulation.

`double *holelist;`

Copy of the `holelist` pointer from input. Note that the *pointer* is copied, not the object that it points to. Therefore don't `free` the input `holelist` if there is need for future reference to it.

`int numberofregions;`

Number of regions (r) that the domain is divided into. This equals `numberofregions` specified on input since number of regions does not change upon triangulation.

`double *regionlist;`

Copy of the `holelist` pointer from input. Comments pertaining `holelist` apply.

`int numberofedges;`

Number of edges (e) on output.

`int *edgelist;`

Pointer to an array of of length $2e$. Node numbers of the end points of edge i are stored in in the array's $2i$ and $2i + 1$ elements. Initialize to `NULL`.

`int *edgemarkerlist;`

Pointer to an array of of length e of edge marker values.

Edges that are also segments, receive the segment's marker value. All others receive a deault marker value of 0 if interior to the domain and a default markder value of 1 if on the boundary. In our use of *Triangle* all boundary edges are also segments therefore the default value of 1 does not apply. Initialize to `NULL`.

`int numberoftriangles;`

Number of triangles (t) on output.

`int *trianglelist;`

Pointer to an array of of length $3t$. Node numbers of the vertices of triangle i, in a counter-clockwise order, are stored in in the array's $3i$, $3i + 1$ and $3i + 2$ elements. Initialize to `NULL`.

`int numberoftriangleattributes;`

Number of attributes (k) stored per triangle.

`double *triangleattributelist;`

Pointer to an array of of length kt. Triangle attributes are stored in groups of k numbers in the array.

In our application we associate a unique region number to each of the domain's regions. Triangles in a region inherit the region's attribute. By examininig the triangle's attribute value we can tell which region is belongs to. Initialize to `NULL`.

Appendix D

Integration on a triangle

The classical n point Gaussian quadrature approximates the integral of a function f on the interval $[-1, 1]$ by a weighted sum of n values of f:

$$\int_{-1}^{1} f(\xi)\, d\xi \approx \sum_{j=1}^{n} w_j f(p^{(j)}).$$

The Gaussian quadrature points $p^{(j)}$ and weights w_j depend on n but are independent of f. It can be shown, see e.g., Atkinson [Atk], that the approximation error is zero for all polynomials of degree $2n - 1$. Integration over an arbitrary interval $[a, b]$ is handled by an affine change of variables $\chi : \xi \mapsto \big((b-a)/2\big)\xi + \big((b+a)/2\big)$ that maps the the *standard interval* $[-1, 1]$ onto $[a, b]$:

$$\int_{a}^{b} f(x)\, dx = \frac{b-a}{2} \int_{-1}^{1} f\Big(\frac{b-a}{2}\xi + \frac{b-a}{2}\Big)\, d\xi. \tag{D.1}$$

The (constant) derivative of χ which appears as the factor $(b-a)/2$ of the integral on the right hand side, equals the the ratio of the lengths of the interval $[a, b]$ and the standard interval $[-1, 1]$.

There are several ways of extending the idea of Gaussian quadrature to integration on triangles but none is as elegant as Gaussian quadrature on intervals. One highly efficient method is provided by Taylor, Wingate and Bos [TWB] where for a large selection of the values n they give quadrature points $p^{(j)}$ and weights w_j for quadrature on the *standard triangle* defined as:

$$\mathcal{T}_{\text{std}} = \{\langle \xi_1, \xi_2 \rangle \in \mathbf{R}^2 \,:\, \xi_1 \geqslant -1,\, \xi_2 \geqslant -1,\, \xi_1 + \xi_2 \leqslant 0\} \tag{D.2}$$

which has an area of 2 (just as the one-dimensional standard interval has length 2). An n point quadrature over $\mathcal{T}_{\text{std}}$ takes the form:

$$\int_{\mathcal{T}_{\text{std}}} f(\xi)\, d\xi \approx \sum_{j=1}^{n} w_j f(p^{(j)}). \tag{D.3}$$

To integrate over an arbitary triangle $\mathcal{T}$, we view $\mathcal{T}$ as the image under an affine map χ of the standard triangle $\mathcal{T}_{\rm std}$ then change variables to reduce the problem to integration over $\mathcal{T}_{\rm std}$:

$$\int_{\mathcal{T}} f(x)\,dx = \frac{|\mathcal{T}|}{2} \int_{\mathcal{T}_{\rm std}} f\big(\chi(\xi)\big)\,d\xi \tag{D.4}$$

where $|\mathcal{T}|$ is the area of $\mathcal{T}$. The factor $|\mathcal{T}|/2$ of the integral on the right hand side is the (constant) Jacobian determinant of χ which also equals the ratio of the areas of the triangle $\mathcal{T}$ and the standard triangle $\mathcal{T}_{\rm std}$. Compare with (D.1).

TWB provide n-point quadrature tables for 14 choices of n. To each n there corresponds a number d, called the quadrature's *strength*, with the property that the n-point approximation error is zero (within the floating point accuracy) for all polynomials of degree less or equal d. The table below shows the 14 matching n and d pairs. We see, for instance, that with 10 quadrature points we get exact values for all polynomials of degree five in two variables.

$\boldsymbol{n}$	3	6	10	15	21	28	36	45	55	66	78	91	105	120
$\boldsymbol{d}$	2	4	5	7	9	11	13	14	16	18	20	21	23	25

Quadrature points in [TWB] are expressed in terms of barycentric coordinates[1] associated with the vertices $\langle 1, -1\rangle$ and $\langle -1, 1\rangle$ of $\mathcal{T}_{\rm std}$ (see the definition in (D.2). Here is an extract from their tables for $n = 10$ (which corresponds to $d = 5$):

```
0.0000000000000     1.0000000000000     0.0262712099504
1.0000000000000     0.0000000000000     0.0262716612068
0.0000000000000     0.0000000000000     0.0274163947600
0.2673273531185     0.6728199218710     0.2348383865823
0.6728175529461     0.2673288599482     0.2348412238268
0.0649236350054     0.6716530111494     0.2480251793114
0.6716498539042     0.0649251690029     0.2480304922521
0.0654032456800     0.2693789366453     0.2518604605529
0.2693767069140     0.0654054874919     0.2518660533658
0.3386738503896     0.3386799893027     0.4505789381914
```

There are ten rows corresponding to the 10 quadrature points. The first two numbers in each row are the barycentric coordinates and the third number is the weight.

Barycentric coordinates are invariant under affine mappings therefore to integrate a function f defined on an arbitrary triangle $\mathcal{T}$, ...

[1] Let the points A, B and C be the vertices of an arbitrary (non-degenerate) triangle. Any point P inside the triangle may be written as a unique convex combination of the vertices:

$$P = \lambda_1 A + \lambda_2 B + \lambda_3 C,$$

where $0 \leqslant \lambda_i \leqslant 1$ for $i = 1, 2, 3$ and $\lambda_1 + \lambda_2 + \lambda_3 = 1$. The coefficients λ_i are called P's *barycentric coordinates*. Given any two of the barycentric coordinates, the third is determined from $\lambda_1 + \lambda_2 + \lambda_3 = 1$.

D.1 Interface to the TWB quadrature tables

The barycentric coordinates and weight for a quadrature point are stored in a `TWB_qdat` structure:

⟨*the twb data structure* 145a⟩≡
```
typedef struct {
    double lambda1;
    double lambda2;
    double weight;
    double lambda2;
} TWB_qdat;
```

Each of the 14 quadrature tables is stored as an array of `TWB_qdat` structures. The arrays are named `quaddata`n where n is the number of quadrature points. Here is the table `quaddata6` corresponding to $n = 6$, $d = 4$:

⟨*quadrature data for n=6, d=4* 145b⟩≡
```
static TWB_qdat quaddata6[] = {
    { 0.0915762135098, 0.0915762135098, 0.2199034873106 },
    { 0.8168475729805, 0.0915762135098, 0.2199034873106 },
    { 0.0915762135098, 0.8168475729805, 0.2199034873106 },
    { 0.1081030181681, 0.4459484909160, 0.4467631793560 },
    { 0.4459484909160, 0.1081030181681, 0.4467631793560 },
    { 0.4459484909160, 0.4459484909160, 0.4467631793560 },
    { 0.0, 0.0, -1.0 }  /* terminator */
};
```

The extraneous line, makred "terminator", is added as an end-of-the-table marker. In this way we don't have to store the length of the table as an additional variable when pasing a pointer to it to a function. The end of the table will be detected by the presence of the negative weight -1.0; all weights are positive in a TWB quadrature.

The 14 tables are wrapped in an array of an anonymous structure type used for internal management of the tables. There is no user interface to structure or the array:

⟨*the 14 tables* 145c⟩≡
```
static struct {
    int npoints;
    int degree;
    TWB_qdat *quaddata;
} QuadTables[] = {
    {   3,   2,    quaddata3 },
    {   6,   4,    quaddata6 },
    {  10,   5,   quaddata10 },
    {  15,   7,   quaddata15 },
    {  21,   9,   quaddata21 },
    {  28,  11,   quaddata28 },
    {  36,  13,   quaddata36 },
```

```
    {  45,  14,   quaddata45 },
    {  55,  16,   quaddata55 },
    {  66,  18,   quaddata66 },
    {  78,  20,   quaddata78 },
    {  91,  21,   quaddata91 },
    { 105,  23,  quaddata105 },
    { 120,  25,  quaddata120 },
};
```

The sole user interface to the quadrature tables is through the function `twb_qdat()` declared as:

⟨*prototype of twb_qdat()* 146a⟩≡
```
TWB_qdat *twb_qdat(int *degree, int *npoints);
```

It receives a request for a quadrature table of strength `*d`. It compares the requested strength to available strengths and increases it to the next greater one if there is no exact match. If the requested strength exceeds the largest possible, it sets it to the largest possible. Then it sets `*n` to the number of quadrature points (that is, to the length of the table) and returns a pointer to the table. Thus the argument `d` is an *in/out* variable while the arguement `n` is an *out* variable.

Here is the implementation of `twb_qdat`:

⟨*function twb_qdat()* 146b⟩≡
```
TWB_qdat *twb_qdat(int *d, int *n)
{
    TWB_qdat *qdat;
    int i;

    for (i = 0; i < ntables; i++)
        if (*d <= QuadTables[i].degree)
            break;

    if (i == ntables)
        i = ntables - 1;

    *d = QuadTables[i].degree;
    *n = QuadTables[i].npoints;
    qdat = QuadTables[i].quaddata;

    for (i = 0; i < *n; i++)
        qdat[i].lambda3 = 1.0 - qdat[i].lambda1 - qdat[i].lambda2;

    return qdat;
}
```

D.2 The files *twb-quad.c* and *twb-quad.h*

The header file *twb-quad.h* declares the `TWB_qdat` structure and the `twb_qdat()` function:

⟨*twb-quad.h* 147⟩≡
```
#ifndef H_TWB_QUAD_H
#define H_TWB_QUAD_H

typedef struct {
    double lambda1;
    double lambda2;
    double weight;
    double lambda3;
} TWB_qdat;

TWB_qdat *twb_qdat(int *degree, int *npoints);

#endif /* H_TWB_QUAD_H */
```

The file *twb-quad.c* consists of 14 tables, one of which is shown in ⟨*quadrature data for n=6, d=4* 145b⟩ and the definition of the function `twb_qdat()` which is shown in ⟨*function twb_qdat()* 146b⟩. The complete *twb-quad.c* is over 800 lines long therefore we won't include it in this document.

Appendix E

Interface to UMFPACK

The UMFPACK is an open-source library of function, made by Tim Davis, for solving sparse linear systems. It may be obtained from:

⟨`http://www.cise.ufl.edu/research/sparse/umfpack/`⟩.

Here we will summarise the very limited interface to UMFPACK that we need for our work. For complete details refer to UMFPACK's manual.

E.1 Sparse matrix storage

A sparse matrix of `m` rows, `n` columns containing `nz` entires is encoded into three arrays:

```
int Ap[n+1];
int Ai[nz];
double Ax [nz];
```

The particular method of storage, known as 'compressed column form', is described in UMFPACK's manual. We don't need to know the details of the encoding other than to know the types and sizes of the arrays in order to use the library.

Additionally, UMFPACK provides an auxiliary 'triplet form' method of storage which is particularly suited to finite element computations.

A triplet form consists of three vectors:

```
int Ti[nt];
int Tj[nt];
double Tx [nt];
```

The `k`-th triplet (`Ti[k]`, `Tj[k]`, `Tx[k]`) holds the (i, j, a_{ij}) values of the entry a_{ij} in row i and column j of the matrix.

The length `nt` of the the vectors `Ti`, `Tj`, `Tx` is at least equal to the number of

entries `nz` but it may longer. If the triplet form contains duplicate records, that is, if multiple `k` values produce the same i an j, then the corresponding values of aij are summed.

The UMFPACK function `umfpack_di_triplet_to_col()` translates a 'triplet form' to the 'compressed column form'. Its prototype is:

```
int umfpack_di_triplet_to_col(
    int n_row,
    int n_col,
    int nz,
    const int Ti[ ],
    const int Tj[ ],
    const double Tx[ ],
    int Ap[ ],
    int Ai[ ],
    double Ax[ ],
    int Map[ ]
);
```

The arguments `n_row`, `n_col` and `nz` are the numbers of rows, columns and entries of the sparse matrix. Upon entry, the input arrays `Ti`, `Tj`, `Tx` hold the matrix in the triplet form. Upon return, the output arrays `Ap`, `Ai`, `Ax` hold the matrix in the compressed column form. Storage for all arrays should be provided by the caller. The `Map` argument is not used in our application and will be set to `NULL`.

The function `umfpack_di_triplet_to_col()` returns one of the following status flags:

`UMFPACK_OK` if successful.

`UMFPACK_ERROR_argument_missing` if `Ap`, `Ai`, `Ti`, and/or `Tj` are missing.

`UMFPACK_ERROR_n_nonpositive` if `n_row <= 0` or `n_col <= 0`.

`UMFPACK_ERROR_invalid_matrix` if `nz < 0`, or if for any `k`, `Ti[k]` and/or `Tj[k]` are not in the range `0` to `n_row-1` or `0` to `n_col-1`, respectively.

`UMFPACK_ERROR_out_of_memory` if unable to allocate sufficient workspace.

E.2 Symbolic analysis

UMFPACK's `umfpack_di_symbolic()` reorders the columns of the matrix to reduce fill-in upon LU factorization. It returns its result in an object pointed to by the `*Symbolic` arguemnt in the prototype:

```
int umfpack_di_symbolic(
    int n_row,
```

```
    int n_col,
    const int Ap[ ],
    const int Ai[ ],
    const double Ax[ ],
    void **Symbolic,
    const double Control[UMFPACK_CONTROL],
    double Info[UMFPACK_INFO]
);
```

Storage for the object is allocated as needed. It is the caller's responsibility to free the memory afterward by calling the function:

```
void umfpack_di_free_symbolic(void **Symbolic);
```

The arguments `n_row` and `n_col` are the numbers of rows and columns of the matrix. The input arguments `Ap`, `Ai`, `Ax` hold the matrix in the compressed column form. The `Control] and [[Info` arguements are not used in our application and will be set to `NULL`.

The function `umfpack_di_symbolic()` returns a large number of status values. In our code if the return value is anything other than `UMFPACK_OK` we consider it a fatal error, we print the numerical value of status and exit the program.

E.3 The LU factorization

The UMFPACK function `umfpack_di_numeric()` receives a sparse matrix and the symbolic analysis object computed in `umfpack_di_symbolic()`, performs an LU factorization for the matrix and returns the result in an object pointed at by the `*Numeric` argument in the protoype:

```
int umfpack_di_numeric(
    const int Ap[ ],
    const int Ai[ ],
    const double Ax[ ],
    void *Symbolic,
    void **Numeric,
    const double Control[UMFPACK_CONTROL],
    double Info[UMFPACK_INFO]
);
```

Storage for the object is allocated as needed. It is the caller's responsibility to free the memory afterward by calling the function:

```
void umfpack_di_free_numeric(void **Numeric);
```

The input argumentsAp, `Ai`, `Ax` hold the matrix in the compressed column form. The `Symbolic` arguemnt referts to the result of the symbolic analysis of a prior call to `umfpack_di_symbolic()`. The `Control] and [[Info` arguements are not used in our application and will be set to `NULL`.

The function `umfpack_di_numeric()` returns a large number of status values. In our code if the return value is anything other than `UMFPACK_OK` we consider it a fatal error, we print the numerical value of status and exit the program.

E.4 The solver

The UMFPACK function `umfpack_di_solve()` solves the linear system $AX = B$ where A is a sparse matrix and B is a given vector. The solution is returned in the vector `X` in the prototype:

```
int umfpack_di_solve(
    int sys,
    const int Ap[ ],
    const int Ai[ ],
    const double Ax[ ],
    double X[ ],
    const double B[ ],
    void *Numeric,
    const double Control[UMFPACK_CONTROL],
    double Info[UMFPACK_INFO]
);
```

Memory for `X` should be allocated by the caller.

The argument `sys` passes one of a large number of option to the solver. The only option relevant to our application is `UMFPACK_A`. The input argumentsAp, `Ai`, `Ax` hold the matrix in the compressed column form. The `Numeric` arguemnt referts to the result of a prior call to `umfpack_di_numeric()`. The `Control]` `and [[Info` arguements are not used in our application and will be set to `NULL`.

The function `umfpack_di_solve()` returns a large number of status values. In our code if the return value is anything other than `UMFPACK_OK` we consider it a fatal error, we print the numerical value of status and exit the program.

Appendix F

Index of chunks

⟨*mass-stiffness-bending.h* 87b⟩ 87b
⟨*stiffness-matrix-check.maple* 94c⟩ 94c, 95a, 95b, 95c, 96
⟨*stiffness-matrix.maple* 92⟩ 92, 93a, 93b, 93c, 94a, 94b
⟨*the file pxp.incl* 80⟩ 80
⟨*unit test for get_matrix()* 86⟩ 86, 87a
⟨*boundary.c* 108c⟩ 108c
⟨*boundary.h* 108b⟩ 108b
⟨*edge selector functions* 106a⟩ 106a, 108c
⟨*function create_boundary_series()* 103⟩ 103, 108c
⟨*function extract_edge()* 106b⟩ 106b, 108c
⟨*function sort_bseries()* 107a⟩ 107a, 107b, 107c, 107d, 108a, 108c
⟨*clean up* 114f⟩ 109a, 114f
⟨*cmdline.ggo* 111a⟩ 111a
⟨*declarations for main.c* 111c⟩ 109a, 111c, 112b, 112e, 113c, 113e, 114a
⟨*included headers for main.c* 109b⟩ 109a, 109b, 109c, 111b, 112a, 112d, 113b, 114d
⟨*load problem module* 112f⟩ 109a, 112f, 113a, 113d
⟨*main.c* 109a⟩ 109a
⟨*parse command line options* 111d⟩ 109a, 111d, 112c
⟨*perform user requested action* 113f⟩ 109a, 113f, 114b, 114c, 114e
⟨*argyris-elem.maple* 115⟩ 115, 116a, 116b, 116c, 116d, 116e, 116f, 117a, 117b, 117c, 118a, 118b, 118c, 118d, 118e, 118f, 119
⟨*derivative codes* 120a⟩ 120a, 128
⟨*function get_shape_data()* 121⟩ 121, 127
⟨*prototype of get_shape_data* 120b⟩ 120b
⟨*shape_functions.c* 127⟩ 127
⟨*shape_functions.h* 128⟩ 128
⟨*dump_mesh.c* 134a⟩ 134a
⟨*dump_mesh.h* 135c⟩ 135c
⟨*function dump_edges* 134c⟩ 134a, 134c
⟨*function dump_elems* 135a⟩ 134a, 135a
⟨*function dump_mesh* 135b⟩ 134a, 135b
⟨*function dump_nodes* 134b⟩ 134a, 134b
⟨*show_mesh_in_geomview.c* 132⟩ 132
⟨*show_mesh_in_geomview.c-old* 131a⟩ 131a
⟨*show_mesh_in_geomview.h* 131b⟩ 131b
⟨*function twb_qdat()* 146b⟩ 146b
⟨*prototype of twb_qdat()* 146a⟩ 146a
⟨*quadrature data for n=6, d=4* 145b⟩ 145b
⟨*the 14 tables* 145c⟩ 145c
⟨*the twb data structure* 145a⟩ 145a
⟨*twb-quad.h* 147⟩ 147

Appendix G

Index of identifiers