

definegeometry2d

November 15, 2001
15:47

Contents

1	A program for Defining 2-D Geometries	1
2	Introduction	2
3	Example Input File	3
4	How To Use This Code	8
5	Keyword Reference	9
6	Other Considerations	15
7	The Main Program	17
8	Main Subroutine	19
9	Read and process DG file	26
10	Read wall file	30
11	Convert mesh data into polygons	32
12	Specify and break up a new polygon	34
13	Read mesh file from UEDGE	37
14	Compute minimum and maximum coordinates	38
15	Print wall coordinates to a file for user examination	39
16	Read input lines specifying a polygon	41
17	Fill in data for this zone	47
18	Find the centroid of a triangle	48
19	Compute the center of a quadrilateral	50
20	Set up default sectors	51
21	Print data from Triangle	53

22 C interface routine To triangle	56
23 INDEX	61

1 A program for Defining 2-D Geometries

This program is intended to be an extension of the logic used in *readgeometry*. Potential couplings to other geometry packages, such as the DG code developed in Garching and AVS/Express, have been taken into consideration. The initial implementation will be with a text-based input file, but eventually this will be expanded to provide an interactive (graphical?) interface. The intermediate specification of the geometry purely in terms of polygons is written out to a netCDF file; the option of reading this file and completing the geometry specification from there should be added.

Note that this program makes extensive use of Jonathan Shewchuck's **Triangle** package to break polygons up into triangles. The user needs to download its source code from <http://www.cs.cmu.edu/~quake/triangle.html> and arrange for the **Makefile** to find it (the default location is `$HOME/Triangle`).

The DG code as currently distributed is already very useful in creating input for **definegeometry2d**. DG may eventually be modified so that it will permit the graphical construction of the polygons that now must be assembled manually, usually with some trial and error. DG is distributed as part of the **SOLPS** package. For more information, see <http://www.rzg.mpg.de/~dpc/solps.html>.

2 Introduction

This code takes a human-readable input file that, together with the other file(s) it references, describes a geometry you want to use in DEGAS 2. The purpose of this code is to transform that description into DEGAS 2's internal description. The hierarchy of geometry-related objects in DEGAS 2 is, from the lowest level to the highest level is:

1. surface,
2. cell,
3. polygon,
4. zone.

Individual cells are composed of surfaces, as discussed in the documentation for the internal geometry (e.g., see *geometry.web*). The next level up is a polygon. This program will call various routines to break up (non-convex) polygons into cells. Finally, a zone may consist of one or more polygons; properties are constant across a zone. For example, a single zone might be used to represent the vacuum region around the plasma which is comprised of several (possibly disconnected) polygons. Or, a plasma flux surface on which density and temperature are constant might be a single zone. Note that polygons are used only within external interfaces such as this code. Cells and surfaces are essentially used only by the code itself. Zones are used primarily for the specification of input and output data.

The interfaces between the plasma (or vacuum) and a solid surface (or exit) are represented by sectors, which are established for diagnostic purposes. In general, a sector is defined by a surface and a zone. This code makes an attempt to identify and label these sectors automatically. Somewhat arbitrarily, a sector between solid and plasma zones is a “target” sector and one between a solid and vacuum zone is a “wall” sector. For further details, see the sector class described in *sector.hweb*. Currently, this code provides no facility for defining “diagnostics” from these sectors or for defining “detectors” from the input file. Instead, these have to be defined explicitly in the code.

Note that the input file for `definegeometry2d` is not listed in `degas2.in`. Although this might be viewed as an omission, it is consistent in that `definegeometry2d` is, in principle, one of many different means of generating DEGAS 2 geometries and, thus, is not as fundamental as the other entries in `degas2.in`.

Instead, the command line for the execution of `definegeometry2d` specifies the name of the main `definegeometry2d` input file. This file will contain pointers to other files that provide externally generated, detailed information about the geometry.

```
definegeometry2d an_input_file
```

Like other text input files used in DEGAS 2, blank lines, spaces, and lines beginning with a comment character # are ignored (comments can also appear at the end of a line).

3 Example Input File

Let us introduce the components of the input file with a nontrivial example:

```
# There are two parts to the input file. The first provides global
# information.
#
# It's a good idea to set the symmetry of the problem first.

symmetry cylindrical

# The bulk of the data describing the "hardware" in this problem are
# in a file generated by the DG code.

dg_file Run35/bypass_12.dgo

# The bounds keyword specifies the size of the universal cell.

bounds 0.3 1.1 -0.8 0.7

# Another big chunk of data required in setting up the problem is the
# mesh which will store the plasma-related data. In this case, the
# mesh is provided in the Sonnet format.

sonnet_mesh Run10/cmod.carre.008 61 26

# The end_prep command terminates the first part of the input file. The
# primary task undertaken with the execution of this command is the
# generation of plasma zones from the quadrilaterals described by the
# plasma mesh.

end_prep

# In the second part of the input file, the user must define zones that
# will completely fill (without overlap) the universal cell. These
# zones are specified as the union of one or more polygons. The
# new_zone command begins the definition of the next zone; in this
# case, a solid zone. The zone definition is terminated by the next
# "new_zone" command (or an "end" or "quit"). The code will assume that
# the polygon will be closed by connecting the last point specified
# to the first point. All polygons should be convex.

new_zone solid

# The following five polygons all belong to that single zone.
# The polygons are enumerated and labeled to facilitate external reference.

# The definition of a polygon is initiated by a "new_polygon" command
# and finished "breakup_polygon", "triangulate_polygon", or
# "triangulate_to_zones". Each of these breaks the polygon down into
# DEGAS 2's internal description consisting of cells and surfaces.
```

```
# In the last case, each of the resulting triangles is assigned a new
# zone number.
#
# Keywords like "outer", "wall", and "edge" specify single points or a
# range of points used to construct the polygon. Other keywords like
# "stratum" or "material" specify properties associated with the polygon.
# Note, however, that these properties will apply to all subsequent
# polygons until overridden by new property settings. Finally, the
# "print_polygon" command (commented out since this input file was well
# past the testing stage) writes out the points in the polygon to a
# text file so they can be analyzed or plotted externally. This is
# useful for debugging problem polygons.

# Polygon 1: inner limiter
new_polygon
  stratum 1
  material mo      # For all solid zones (default temp. and recycling)
  outer 0 1
  wall 6 0 1
  wall 3 3 3
  edge 0 0 * reverse
#   print_polygon poly_1
breakup_polygon

# Polygon 2: upper divertor
new_polygon
  stratum 2
  outer 1 2
  wall 5 11 0
#   print_polygon poly_2
breakup_polygon

# Polygon 3: outer limiter
new_polygon
  stratum 3
  outer 2 3
  wall 5 49 11
#   print_polygon poly_3
triangulate_polygon

# Polygon 4: gas box, right side
new_polygon
  stratum 4
  outer 3 0
  wall 5 68 49
#   print_polygon poly_4
breakup_polygon

# Polygon 5: gas box, left side
new_polygon
  stratum 5
  outer 0
```

```
edge 0 0 0 0
wall 2 0 2
wall 4 1 15
wall 5 71 68
#      print_polygon poly_5
breakup_polygon

new_zone solid

# Polygon 6: outer target
new_polygon
stratum 6
wall 7 0 15
edge 61 61 *
wall 7 24 26
#      print_polygon poly_6
breakup_polygon

new_zone vacuum

# Polygon 7: private flux region
new_polygon
stratum 7
edge 0 10 0 0
edge 52 61 0 0
wall 4 6 0
wall 2 1 0
#      print_polygon poly_7
triangulate_to_zones

new_zone vacuum

# Polygon 8: gas box entrance
new_polygon
stratum 8
edge 61 61 0 0
wall 7 15 15
wall 4 7 6
#      print_polygon poly_8
breakup_polygon

new_zone vacuum

# Polygon 9: gas box
new_polygon
stratum 9
wall 7 15 3
wall 5 51 71
wall 4 15 7
#      print_polygon poly_9
triangulate_to_zones
```

```
new_zone vacuum

# Polygon 10: gas box exit
new_polygon
stratum 10
wall 7 3 2
wall 5 50 51
#      print_polygon poly_10
breakup_polygon

new_zone vacuum

# Polygon 11: lower main chamber
new_polygon
stratum 11
edge 61 42 26 26
wall 5 38 50
wall 7 2 0
wall 7 26 24
#      print_polygon poly_11
triangulate_to_zones

new_zone vacuum

# Polygon 12: upper main chamber
new_polygon
stratum 12
edge 42 0 26 26
wall 3 3 3
wall 6 1 1
wall 5 0 38
#      print_polygon poly_12
triangulate_to_zones

new_zone exit

# Polygon 13: core plasma
new_polygon
stratum 13
edge 11 51 0 0
#      print_polygon poly_13
breakup_polygon

# The description of this geometry in terms of polygons is written out
# to a netCDF file with this name so that it may be used again later.

polygon_nc_file Run35/bypass_12_poly_a.nc

# The "end" command instructs the code to finish setting up the
# internal geometry arrays, to check the geometry for consistency,
# and to write its netCDF file. Since the process of generating
# a geometry as complex as this one is understandably iterative, the
```

§1 [#1] definegeometry2d

A program for Defining 2-D Geometries 7

```
# code also has a "quit" command (that can be used anywhere except
# within a polygon definition) that quietly terminates the code without
# performing any of these finalization steps.
```

```
end
```

4 How To Use This Code

Again, this input file consists of two sections:

1. A preparatory stage in which global parameters are set and file names are specified,
2. A “construction” section containing the specification of polygons that will fill the universal cell.

Virtually all of the information in the input file is “high level”, e.g., point coordinates rarely appear. Instead, most of the detailed information regarding points and their connectivity is contained in the other files that can, and should, be generated by other packages. The names of these files are specified in the preparatory stage. The formats that are currently in use are (all of these are text files):

dg_file An output (with extension .dgo) file from the DG code.

wallfile A simple text file containing a specified number of simply connected “walls”, with each wall consisting of a sequence of (X, Z) values.

sonnet_mesh A 2-D mesh specified in the Sonnet format; e.g., as generated by the **Sonnet** or **Carre** codes.

uedge_mesh A specific format used to transfer data between the **UEDGE** and **DEGAS 2** codes.

Both *dg_file* and *wallfile* result in the definition of “walls” that will be used to construct the polygons in the second section using the *wall* keyword. A portion of the boundary of a plasma mesh imported via *sonnet_mesh* or *uedge_mesh* can be incorporated into a polygon via the *edge* command. The corners of the universal cell can be added to a polygon with the *outer* keyword.

There are then two possible approaches to generating a new geometry:

1. If the problem possesses a basic symmetry, e.g., “plasma parameters are constant on a flux surface”, you can specify only those bounding surfaces and let *definegeometry2d* subdivide them (semi-arbitrarily) into smaller zones. First, construct a wall file containing the coordinates of each of the bounding surfaces, with each as a separate wall (see below under the *wallfile* keyword). Then, in the *definegeometry2d* input file, create a polygon out of each pair of adjacent walls, i.e., surfaces. Using the *triangulate_to_zones* keyword will then break these polygons up into smaller zones. The sizes of the triangles will be almost entirely controlled by the number of points used in the walls (see, however, the *triangle_area* keyword).
2. If a mesh composed of triangles or quadrilaterals already exists, the easier approach may be to write that information in the Sonnet format. Each mesh zone in the Sonnet file format is defined by five points. These coordinates are given on three consecutive lines. The “corners” appear on the first and third lines; the second line contains the “center” point. Inside *definegeometry2d*, the four corners must trace out the shape in a clockwise fashion. While there is some freedom in the way these points are labeled, for definiteness, we will say that the first line contains the second and third points (in that order) of a quadrilateral; the third line has the first and fourth points. For a triangle, just make the two points on the third line the same (the code will also agree to call it a triangle if the third and fourth points are the same). If the center point is set exactly to zero, built in routines will be used to compute an appropriate center. As in the above example, additional data about the hardware can be specified and translated into “walls”. The mesh edges can be used together with the walls to specify the rest of the polygons required to fill the universal cell.

The need for and assignment of properties to a polygon or zone varies from one problem to another; their usage should be largely obvious. The *stratum* label is in a sense completely arbitrary. However, it is worth assigning its value with some thought since it can be used in specifying source locations. The code keeps track of the segment number of each polygon (with the first segment being denoted by “0”, the second by “1”, etc.; the same labels can be thought of as applying to the first point of the segment) as the polygon is broken up and translated into internal geometry components. Together, the segment and stratum numbers allow the user to later refer back to individual segments (e.g., as the location of a surface source) of each polygon.

5 Keyword Reference

Each line in the input file is of the form:

keyword arguments

with some keywords having multiple arguments, others having none at all. Lower case single letters (perhaps with subscripts) are used to represent integer arguments. Upper case single letters correspond to real (i.e., floating point) arguments. All other arguments are strings. This first group of keywords is intended only for use in the preparatory stage of the code; invoking them after the *end_prep* command will likely result in the code crashing or behaving strangely.

Preparatory Section

symmetry `sym` describes the symmetry of the geometry with `sym = plane, cylindrical` (“toroidal symmetry”), or `oned` (for a system with two symmetry directions). This must appear prior to the `end_prep` keyword.

dg-file `filename` specifies the name of an output file from the DG code (with the extension `.dgo`). If `filename` does in fact contain `.dgo`, the dimensions therein are assumed to be in millimeters; filenames without this extension are assumed to have data in meters (data perhaps generated by some other means). The coordinates of the “nodes” (or points) specified in the file are read in. Their ordering in the file is that of the “elements” (or segments) to which the nodes belong. A list of elements in terms of the node numbers is compiled as the file is read in. Each of the nodes is then characterized as being of a particular type:

1. `node_no_elements` if the node has no corresponding elements,
2. `node_one_element` if the node is associated with only one element,
3. `node_regular` if the node is associated with two elements with normals of the same sense,
4. `node_mixed_normals` if the node is associated with two elements having normals in opposite sense (i.e., the node is either the “start” or the “end” of both elements),
5. `node_many_elements` if the node represents the intersection of more than two elements.

Once each of the nodes has been characterized, the resulting information permits the translation of the elements into “walls” that can be used in the subsequent specification of polygons. The code loops over all of the irregular nodes (that are connected to at least one element). Each of these serves as the start of a wall. The rest of the wall is mapped out from the adjacent elements using the well-defined connectivity associated with the regular nodes comprising those elements. The process is stopped, of course, when the trail ends at another irregular node. A subsequent search of the elements that have not been assigned to walls is performed to identify closed surfaces consisting of only regular nodes. The upshot of this procedure is to be cognizant of the location of irregular nodes when using DG. Handling them carefully there can result in a simpler set of walls and, hence, ease the task of designing the polygons.

sonnet_mesh `filename` n_x n_z specifies a file, `filename`, containing a plasma mesh in the Sonnet format. Files not generated by Sonnet (or, equivalently, by Carre) can be used, but the user should familiarize themselves with the details of the routine that reads the file, `read_sonnet_mesh`. The Sonnet format specifies the plasma mesh as a n_x by n_z rectangular grid of quadrilaterals. The designation of the x and z directions is somewhat arbitrary and does not necessarily have any correspondence with the (Cartesian) X and Z coordinates referred to elsewhere in this program. The usual tokamak configuration has the x coordinate following along flux surfaces and the z coordinate going across flux surfaces. Each quadrilateral is given as a set of four corners and a center so that the mesh can be treated as completely unstructured and no connectivity information is needed. On the other hand, corners of adjacent mesh cells that are intended to match should match *exactly*. In defining the surrounding polygons, the user will undoubtedly need to be familiar with the mesh’s Connectivity in defining the surrounding polygons. The ordering of the corners is important. See below and in `read_sonnet_mesh` for more details.

uedge-mesh filename specifies the name of a file, *filename*, generated by the UEDGE code for the purpose of transferring data to DEGAS 2. The file contains both geometry and plasma data, although only the former is read in by the *read_uedge_mesh* routine. The first few lines of the file contain additional information describing the mesh. The first line is assumed to be a comment. The first number on the second line is assumed to be n_x . Likewise, the third line provides n_z . The next three lines are ignored. As with the Sonnet format, a center and four corners are given for each quadrilateral and no connectivity information is needed. See the routine *read_uedge_mesh* for more details on the order of the corners.

wallfile filename specifies the name of a file, *filename*, that contains a list of walls, each comprised by two or more points. Like other DEGAS 2 text files, *wallfiles* can contain blank lines and comments (beginning with #). Spacing should not matter. The first line provides the number of walls in the file, *num_new_walls*. The second line lists the number of points comprising each wall. I.e., this line must contain *num_new_walls* integers. The rest of the file contains the coordinates in ordered pairs, with one (X, Z) pair per line, starting with the first point of the first wall. The other points in that wall follow on consecutive lines. The points in the second wall (if there is one) come immediately after that with no delimiter (the code is using only the number of points provided in line two to determine which point goes in which wall), and so on. For simple one wall systems, the file can be easily created by hand. For more complicated systems involving several walls, the user may benefit from writing a code to generate the file.

print_min_max loops over all of the nodes (read in via the *dg_file* or *wallfile* keywords) and prints to standard out the maximum and minimum R and Z values (in meters). These can be useful in choosing suitable bounds for the universal cell.

bounds X_{min} X_{max} Z_{min} Z_{max} specifies the corners of the universal cell in X ($X_{\min} \rightarrow X_{\max}$) and Z ($Z_{\min} \rightarrow Z_{\max}$). The *outer* keyword can be used to incorporate these points into polygons.

print_walls format filename instructs the code to write out the current set of walls. If *filename* is not provided, the data are written to standard out. There are two possible *formats*. The first, *tabular*, has the walls laid out in successive columns, suitable for plotting with an external program. The second, *linear*) is consistent with the format used by the *wallfile* keyword.

end_prep terminates the preparatory stage of the code. Dynamically allocated arrays associated with nodes, elements, and walls are trimmed to their final sizes, and the universal cell is established. If a plasma mesh has been specified via the *sonnet_mesh* or *uedge_mesh* keywords, the constituent quadrilaterals are translated into polygons and then decomposed into the internal elements of the DEGAS 2 geometry with each quadrilateral corresponding to a plasma zone. One advantage of making these the first zones defined in the problem is that transfer of zone-based output information to other codes (such as a fluid plasma code) is simplified.

Construction Section The rest of the keywords are associated with the “construction” section of the input file. While they may (even legally) be used prior to the *end_prep* command, such usage is not encouraged in the interest of maintaining the simplicity of the input file. One exception is the *quit* keyword that can be used anywhere in the input file (except inside a polygon definition). There is a further subdivision in that all keywords regarding a specific polygon must appear between its *new_polygon* command and the command that denotes its completion (*clear_polygon*, *breakup_polygon*, *triangulate_polygon*, or *triangulate_to_zones*).

new_zone type starts a new zone of type *type*. All polygons specified after this command and prior to the next *new_zone* command will be added to this zone. The possible values of *type* are **vacuum**, **plasma**, **solid**, and **exit**.

polygon_nc_file filename will cause the netCDF file containing the polygon description of the geometry (based on the g2 class, as described in the file **geometry2d.hweb**) to be named *filename*. The default file name is **polygon.nc**.

quit causes the code to terminate immediately. This is useful for preliminary runs in which the geometry is known to be inconsistent or incomplete, but the code needs to be run so that walls or polygons can be printed out for external examination.

end denotes completion of the geometry specification. A variety of derived relationships are set up by subroutine calls. The geometry is checked (thoroughly) and written to its netCDF file.

new_polygon begins the definition of a new polygon. The following keywords can be used in describing this polygon.

outer i j k adds to the current polygon the *i*th, *j*th, and *k*th points of the bounding rectangle of the problem (defined with the *bounds* keyword, i.e., the minimum and maximum *X* and *Z* used to define DEGAS 2’s “universal cell”). The numbering of the corners is clockwise:

- | | |
|----------|-----------------------------|
| 0 (or 4) | (X_{\min} , Z_{\min}) |
| 1 | (X_{\min} , Z_{\max}) |
| 2 | (X_{\max} , Z_{\max}) |
| 3 | (X_{\max} , Z_{\min}) |

There can be any number of arguments (e.g., see the above example input file).

wall i_{wall} j_{start} j_{stop} reverse takes points from wall number *i_{wall}* that was specified with either the *wallfile* or *dg_file* command. The parameters *j_{start}* and *j_{stop}* prescribe which points from that wall to use. Note that the first point of the wall is numbered 0 so that the only valid values of *j_{start}* and *j_{stop}* are between 0 and the number of elements in the wall. If the wall needs to be traversed in the reverse direction, *j_{stop}* can be set less than *j_{start}*. If *j_{start} = j_{stop}*, a single point is added to the polygon. If both *j_{start}* and *j_{stop}* are replaced by *, all of the points in the wall are used. The wildcard character can also be used as: *j_{start} ** to indicate that all of the points from *j_{start}* to the end of the wall should be used. The additional argument **reverse** can be added at the end of the line to reverse the ordering of the points indicated by *j_{start}*, *j_{stop}*, and / or *. The primary usage of **reverse**, however, will be in conjunction with *.

edge $i_{x,\text{start}} \ i_{x,\text{stop}} \ i_{z,\text{start}} \ i_{z,\text{stop}}$ *reverse* takes an “edge” from a plasma mesh defined with either the *sonnet_mesh* or *wedge_mesh* keywords and adds it to the current polygon. To properly refer to an edge, the arguments must be such that either $i_{x,\text{start}} = i_{x,\text{stop}}$ or $i_{z,\text{start}} = i_{z,\text{stop}}$. Valid values of $i_{x,\text{start}}$ and $i_{x,\text{stop}}$ will be between n_x , inclusive. Likewise, $i_{z,\text{start}}$ and $i_{z,\text{stop}}$ will be between 0 and n_z . Either pair can be replaced with * to indicate that the entire edge should be added to the polygon. The code used to process the arguments “knows” about the four corners comprising each mesh zone so that *edge* * 0 0 actually refers $n_x + 1$ points (see also the additional detail below on corners). As with the *wall* keyword, the *reverse* argument can be used to indicate that the points should be added to the polygon in the order opposite to that indicated by the other arguments.

stratum i_{stratum} labels the polygon as belonging to stratum number i_{stratum} . This is a semi-arbitrary label that can be used in specifying sources, setting up diagnostics, etc. It will be applied to subsequent polygons until used again.

material *mat* sets the material for the polygon to the material having the symbol *mat*. A material with this symbol must appear in the *materials.infile* (from the *degas2.in* file) for this run. It will be applied to all subsequent polygons until used again. Note that this setting only affects the calculation when one edge of this polygon ends up being a target or wall sector.

temperature T associates a temperature of T Kelvin (a real number) to the polygon. This value can affect, e.g., the energy of desorbed or puffed molecules. It will be applied to all subsequent polygons until used again. Note that this setting only affects the calculation when one edge of this polygon ends up being a target or wall sector.

recyc_coef R associates a recycling coefficient of R (R a real number, $0 < R \leq 1$) to the polygon. It will be applied to all subsequent polygons until used again. Note that this setting only affects the calculation when one edge of this polygon ends up being a target or wall sector.

triangle_area A specifies a minimum area in square meters for the triangles to be created by *Triangle* (with A a real number) when the *triangulate_to_zones* command is executed. Practically, it seems that this ends up as little more than a suggestion. The reason is that we cannot allow *Triangle* to add points along the boundary of the polygon. Although this would not compromise the definition of DEGAS 2 surfaces from the triangulation of the current polygon, the surface(s) of the adjacent polygon(s) would not be subdivided in the same way. Hence, the exact correspondence of adjacent surfaces that the code relies upon for establishing connectivity would be destroyed. *Triangle* can, however, add points to the interior of a polygon; this parameter A can be used to control that process.

triangle_hole $n \ X \ Z$ specifies an integer number of holes, n . This is currently constrained to be a single hole, $n = 1$. The coordinates (real numbers) of the hole are given as X and Z . The hole location is used by *Triangle* when either the *triangulate_polygon* or *triangulate_to_zones* command is executed on a polygon that encircles a region that is *not* to be broken up into triangles. The hole location can be anywhere in this interior region.

print_polygon *filename* prints the points of the current polygon to a file, *filename*. If *filename* is not present, the coordinates are written to standard out.

clear_polygon will cause the code to completely forget about this polygon in subsequent operations. This is useful for checking the integrity of several polygons with a single run of the code.

breakup_polygon instructs the code to use DEGAS 2's original *decompose_polygon* routine to break the polygon up into the internal geometry elements. This routine expects the polygon to be traced out in a clockwise direction. Note also that *decompose_polygon* is far from foolproof and will occasionally fail with a "unable to decompose polygon" error. This problem can usually be remedied by choosing a different point of the polygon as the first point. On the other hand, the functionality of the vastly more robust *triangulate_polygon* routine is equivalent. The two approaches may result in different run efficiencies, but no such variability has ever been established.

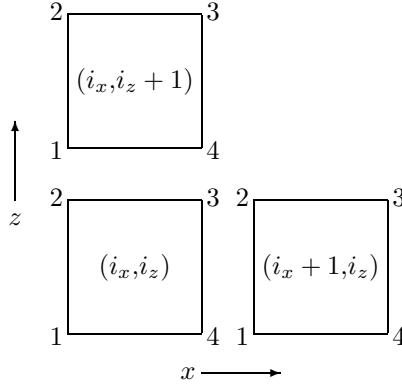
triangulate_polygon instructs the code to use the **Triangle** package to break the polygon up into DEGAS 2's internal geometry elements. The **Triangle** routine is extremely tolerant and robust; the input polygon can be traced out in either direction. However, a clockwise orientation is recommended since this will preserve (as *strata_segment* labels) the relationship between the resulting bounding segments and the original polygon points. This command does only a basic triangulation and is only recommended for use with polygons in solid zones.

triangulate_to_zones instructs the code to use the **Triangle** package to break the polygon up into DEGAS 2's internal geometry elements. The **Triangle** routine is extremely tolerant and robust; the input polygon can be traced out in either direction. Each resulting triangle is added to the geometry as a separate zone (of the current type). A second refining call to **Triangle** beyond the one analogous to that done with *triangulate_to_zones* is performed that permits the application of additional constraints such as a minimum area. A clockwise orientation is recommended since this will preserve (as *strata_segment* labels) the relationship between the resulting bounding segments and the original polygon points.

6 Other Considerations

Regarding corners with the edge keyword: Because the **UEDGE** and **Sonnet** mesh formats both provide four corners for each mesh zone, the number of points along each “edge” of the mesh is one greater than the number of zones along that edge. This fact must be taken into account in defining polygons along these edges.

For clarity, here is the convention used to number the corners in *definegeometry2d*:



In the **Sonnet** mesh format, the first line contains corners 2 and 3. The second line specifies the center. Corners 1 and 4 are on the third line. The **UEDGE** mesh format specifies the corners for each point in the order 0 (center), 1, 4, 2, and 3.

For a particular (i_x, i_z) specified via arguments to the *edge* keyword, the logic that determines the corner to be used is:

1. If $i_x = 0$ and $i_z = 0$, corner 1 of the $(1, 1)$ zone (recall that there is no $(0, 0)$ zone).
2. If $i_x = 0$, but $i_z \neq 0$, corner 2 of the $(1, i_z)$ zone.
3. If $i_x \neq 0$, but $i_z = 0$, corner 4 of the $(i_x, 1)$ zone.
4. Otherwise, use corner 3 of the (i_x, i_z) zone.

Regarding detectors and diagnostics: Ideally the user would also be able to specify detectors and diagnostic sectors via this input file. Some such facility may be possible without a great deal of effort. However, a graphical approach to their specification would likely prove more flexible and easier to use. Once an overall graphical approach to running `definegeometry2d` has been contemplated, a decision regarding the handling of detectors and diagnostic sectors will be made. In the interim, the best approach is to insert hardwired code into this file; specific examples can be made available on request.

Special note on comments and debuggers: Part of this file is written in C. The current Sun Workshop debugger (there may be others) utilizes the original FWEB file in stepping through the source code of the C routines. Getting the object file and the source code lines to match up requires exercising some care in writing comments. First, do not use the // comment style on a single line. Second, do not put blank lines before or after multiline (delimited by /* */) comments. There may be other requirements as well.

```
$Id: definegeometry2d.web,v 1.3 2001/07/16 20:45:11 dstotler Exp $
"definegeometry2d.f" 1≡
@m FILE 'definegeometry2d.web'

@m dg_loop #:0
@m wall_loop #:0
@m wall_loop2 #:0
@m wall_break #:0
@m next_surface #:0
```

The unnamed module.

```
"definegeometry2d.f" 6.1≡
@Lc:   (Functions and subroutines 7)
#define REAL double // Need to be sure this does not overlap with local macros
#include <stdio.h>
#include "triangle.h" // Dependency explicitly included in Makefile
(C Functions 21)C
```

7 The Main Program

```
"definegeometry2d.f" 7 ≡
@m element_start 0
@m element_end 1
@m node_undefined 0
@m node_regular 1
@m node_no_elements 2
@m node_one_element 3
@m node_many_elements 4
@m node_mixed_normals 5
@m sec_undef 0
@m sec_p1 1
@m sec_p2 2
@m sec_miss 3
@m sec_done 40 // Try to match to maximum?
@m poly_stratum 1 // Indices for poly_int_props
@m poly_material 2
@m poly_num_holes 3
@m poly_int_max 3
@m poly_temperature 1 // Indices for poly_real_props
@m poly_recyc_coef 2
@m poly_min_area 3
@m poly_hole_x 4
@m poly_hole_z 5
@m poly_real_max 5
@m clear_polygon 0
@m breakup_polygon 1
@m triangulate_polygon 2
@m triangulate_to_zones 3
@m max_triangles 1700
@m increment_num_elements num_elements++
dim_elements = max(dim_elements, num_elements)
var_realloc(element_nodes)
var_realloc(element_missing)
var_realloc(element_assigned)

@m increment_num_nodes num_nodes++
dim_nodes = max(dim_nodes, num_nodes)
var_realloc(nodes)
var_realloc(node_type)
var_realloc(node_element_count)

@m increment_num_walls num_walls++
dim_walls = max(dim_walls, num_walls)
var_realloc(wall_nodes)
var_realloc(wall_elements)
var_realloc(wall_element_count)

@m increment_g2_num_polygons g2_num_polygons++
```

```

var_realloc(g2_polygon_xz)
var_realloc(g2_polygon_segment)
var_realloc(g2_polygon_points)
var_realloc(g2_polygon_zone)
var_realloc(g2_polygon_stratum)
var_realloc(poly_int_props)
var_realloc(poly_real_props)
g2_polygon_stratumg2_num_polygons = int_unused
do i_inc_g2 = 0, g2_num_points - 1
    g2_polygon_xzg2_num_polygons,i_inc_g2,g2_x = zero
    g2_polygon_xzg2_num_polygons,i_inc_g2,g2_z = zero
    g2_polygon_segmentg2_num_polygons,i_inc_g2 = int_unused
end do

@m open_file(aunit, fname)
open(unit = aunit, file = fname, status = 'old', form = 'formatted', iostat = open_stat)
if (open_stat ≠ 0) then
    write(stderr, *) 'Cannot open file', line(b : e), ', error number', open_stat
    return
end if

@m mesh_xzm(jxz, i, ix, iz) mesh_xz(iz-1)*nxd+ix,i,jxz // "m" for macro.

```

{Functions and subroutines 7} ≡

```

program definegeometry2d
implicit none_f77
implicit none_f90
character*FILELEN filename

call readfilenames

call command_arg(1, filename)
call nc_read_materials
call def_geom_2d_main(filename)

stop
end

```

See also sections 8, 13, 14, 15, 16, 17, 18, 19, and 20.

This code is used in section 6.1.

8 Main Subroutine

```

⟨ Functions and subroutines 7 ⟩ +≡
subroutine def_geom_2d_main(filename)

define_dimen(node_ind, dim_nodes)
define_dimen(element_ind, dim_elements)
define_dimen(element_ends_ind, element_start, element_end)
    /* Add “2d” to distinguish from wall_ind in sector.hweb. */
define_dimen(wall2d_ind, dim_walls)
define_dimen(mesh_corner_ind, 0, 4)
define_dimen(mesh_xz_ind, nxd * nzd)
define_dimen(poly_int_ind, poly_int_max)
define_dimen(poly_real_ind, poly_real_max)

define_varp(nodes, FLOAT, g2_xz_ind, node_ind)
define_varp(node_type, INT, node_ind)
define_varp(node_element_count, INT, node_ind)

define_varp(element_nodes, INT, element_ends_ind, element_ind)
define_varp(element_missing, INT, element_ind)
define_varp(element_assigned, INT, element_ind)

define_varp(wall_nodes, INT, g2_points_ind0, wall2d_ind)
define_varp(wall_elements, INT, g2_points_ind, wall2d_ind)
define_varp(wall_element_count, INT, wall2d_ind)
/* Since we can have only one adjustable dimension with pointers, combine the two into a single one
here. We can use the desired 3-D representation in subroutines accessing these arrays. */
define_varp(mesh_xz, FLOAT, g2_xz_ind, mesh_corner_ind, mesh_xz_ind)

define_varp(poly_int_props, INT, poly_int_ind, g2_poly_ind)
define_varp(poly_real_props, FLOAT, poly_real_ind, g2_poly_ind)

implicit none_f77
g2_common
implicit none_f90

character*FILELEN filename    // Input /* Local variables, by section:
integer diskin2, length, p, b, e, open_stat, zone, fileid,      /* Main section */
       prep_done, i_inc_g2, i_prop
integer current_int_props1:poly_int_max, zonearray1, facearray0:1
real current_real_props1:poly_real_max
character*LINELEN line, keyword
character*FILELEN tmpfilename

integer ielement, section, i, inode,      /* Process DG File */
       miss_elem, start, end, iwall, iseg, ielement2
       real temp_x, temp_z, mult

integer mesh_sense, ix, iz, n    // Mesh Polygons

vc_decl(yhat)
vc_decl(test_vec_1)
vc_decl(test_vec_2)
vc_decl(test_vec_3)
vc_decl(center)

```

```

real x_min, x_max, z_min, z_max, xb_min, xb_max, /* Min., Max.*/
      yb_min, yb_max, zb_min, zb_max, vol
vc_decl(min_corner)
vc_decl(max_corner)

integer nunit // Print Walls
character*LINELEN file_format
character*FILELEN walloutfile

integer num_new_walls, this_node // Wall File
real x_wall, z_wall
character*FILELEN wallinfile

integer symmetry // Symmetry

integer process_polygon, ntriangles, refine, j // Zone, Polygon
integer temp_int_props1:poly_int_max, temp_segment0:max_triangles-1,0:3
real temp_polygon0:g2_num_points-1,g2_x:g2_z, temp_triangles0:max_triangles-1,0:3,g2_x:g2_z,
      temp_real_props1:poly_real_max, temp_holes0:0,g2_x:g2_z
character*LINELEN new_zone_type

integer write_poly_nc // polygon.nc file
character*FILELEN polygon_nc_file

integer num_nodes, num_elements, num_walls, /* Local pointers*/
      dim_nodes, dim_elements, dim_walls, nxd, nzd

declare_varp(nodes)
declare_varp(node_type)
declare_varp(node_element_count)

declare_varp(element_nodes)
declare_varp(element_missing)
declare_varp(element_assigned)

declare_varp(wall_nodes)
declare_varp(wall_elements)
declare_varp(wall_element_count)

declare_varp(mesh_xz)

declare_varp(poly_int_props)
declare_varp(poly_real_props)

⟨Memory allocation interface 0⟩
st_decls
vc_decls
nc_decls
g2_ncdecl
gi_ext /* PREPARATION STAGE */
open(unit = diskin, file = filename, status = 'old', form = 'formatted')
diskin2 = diskin + 1

zone = 0
g2_num_polygons = 0
symmetry = geometry_symmetry_none
xb_min = zero
xb_max = zero

```

```

yb_min = zero
yb_max = zero
zb_min = zero
zb_max = zero
nxd = 0
nzd = 0
num_walls = 0
num_nodes = 0
num_elements = 0
facearray0 = 0
facearray1 = 0 /* To permit these to be reallocated by any of the three possible macros, helps to
make the initial allocation of size mem_inc. */
dim_walls = mem_inc
dim_nodes = mem_inc
dim_elements = mem_inc

var_alloc(nodes)
var_alloc(node_type)
var_alloc(node_element_count)

var_alloc(element_nodes)
var_alloc(element_missing)
var_alloc(element_assigned)

var_alloc(wall_nodes)
var_alloc(wall_elements)
var_alloc(wall_element_count)

prep_done = FALSE
current_int_props poly_stratum = 0
current_int_props poly_material = 0
current_int_props poly_num_holes = 0
current_real_props poly_temperature = const(3., 2)
current_real_props poly_recyc_coef = one
current_real_props poly_min_area = const(1., -3)
current_real_props poly_hole_x = real_undef
current_real_props poly_hole_z = real_undef
write_poly_nc = TRUE
polygon_nc_file = 'polygon.nc'

loop1: continue

assert(read_string(diskin, line, length))
assert(length ≤ len(line))
length = parse_string(line(:length))
p = 0
assert(next_token(line, b, e, p))
keyword = line(b:e)

if (keyword ≡ 'dg_file') then
  ⟨ Process DG File 9 ⟩

else if (keyword ≡ 'sonnet_mesh' ∨ keyword ≡ 'uedge_mesh') then
  assert(next_token(line, b, e, p))
  tmpfilename = line(b:e) // Sun F90 chokes if don't do this
  open_file(diskin2, tmpfilename) /* E.g., sonnet_mesh filename 120 24 */

```

```

if (keyword ≡ 'sonnet_mesh') then
    assert(next_token(line, b, e, p))
    nxd = read_integer(line(b : e))
    assert(next_token(line, b, e, p))
    nzd = read_integer(line(b : e))
else if (keyword ≡ 'uedge_mesh') then
    assert(next_token(line, b, e, p))
    read(diskin2, *) // First line is a comment
    read(diskin2, *) nxd
    read(diskin2, *) nzd
end if
var_alloc(mesh_xz)
if (keyword ≡ 'sonnet_mesh') then
    call read_sonnet_mesh(diskin2, nxd, nzd, mesh_xz)
else if (keyword ≡ 'uedge_mesh') then
    call read_uedge_mesh(diskin2, nxd, nzd, mesh_xz)
end if

else if (keyword ≡ 'wallfile') then
    ⟨ Read Wall File 10 ⟩

else if (keyword ≡ 'print_min_max') then
    call find_min_max(num_nodes, nodes, node_type, x_min, x_max, z_min, z_max)
    write(stdout, *) 'Xrange', x_min, '→', x_max
    write(stdout, *) 'Zrange', z_min, '→', z_max

else if (keyword ≡ 'bounds') then
    ⟨ Set Bounds 8.1 ⟩

else if (keyword ≡ 'symmetry') then
    ⟨ Set Symmetry 8.2 ⟩

else if (keyword ≡ 'print_walls') then
    assert(next_token(line, b, e, p))
    file_format = line(b : e)
    if (next_token(line, b, e, p) then
        walloutfile = line(b : e)
        nunit = diskout
    else
        walloutfile = char_undef
        nunit = stdout
    end if
    call print_walls(nunit, file_format, walloutfile, num_walls, wall_element_count, wall_nodes, nodes)
else if (keyword ≡ 'end_prep') then
    ⟨ End Prep 8.3 ⟩

else if (keyword ≡ 'new_zone') then
    zone++
    assert(next_token(line, b, e, p))
    new_zone_type = line(b : e)
else if (keyword ≡ 'new_polygon') then
    ⟨ New Polygon 12 ⟩
else if (keyword ≡ 'polygon_nc_file') then
    assert(next_token(line, b, e, p))

```

```

if (line(b : e) ≡ 'none' ∨ line(b : e) ≡ 'NONE') then
    write_poly_nc = FALSE
else
    polygon_nc_file = line(b : e)
end if

else if (keyword ≡ 'quit' ∨ keyword ≡ 'end') then
    go to eof
end if
go to loop1

eof: continue
close (unit = diskin)

var_reallocb(g2_polygon_xz)
var_reallocb(g2_polygon_segment)
var_reallocb(g2_polygon_points)
var_reallocb(g2_polygon_zone)
var_reallocb(g2_polygon_stratum)
var_reallocb(poly_int_props)
var_reallocb(poly_real_props)

if (write_poly_nc ≡ TRUE) then
    fileid = nccreate(trim(polygon_nc_file), NC_CLOBBER, nc_stat)
    g2_ncdef(fileid)
    call ncedef(fileid, nc_stat)
    g2_ncwrite(fileid)
    call ncclose(fileid, nc_stat)
end if

if (keyword ≡ 'end') then
    call boundaries_neighbors

    call default_sectors(g2_num_polygons, g2_polygon_points, g2_polygon_xz, g2_polygon_segment,
                           g2_polygon_zone, poly_int_props, poly_real_props)

    call diag_sector_setup
    call end_sectors
    call null_detector_setup
    call end_detectors
    call check_geometry
    call write_geometry
    call erase_geometry
end if

var_free(nodes)
var_free(node_type)
var_free(node_element_count)
var_free(element_nodes)
var_free(element_missing)
var_free(element_assigned)
var_free(wall_nodes)
var_free(wall_elements)
var_free(wall_element_count)

```

```

var_free(mesh_xz)
var_free(poly_int_props)
var_free(poly_real_props)

return end

```

Set dimensions of universal cell.

$\langle \text{Set Bounds 8.1} \rangle \equiv$

```

assert(next_token(line, b, e, p))
xb_min = read_real(line(b : e))
assert(next_token(line, b, e, p))
xb_max = read_real(line(b : e))
assert(next_token(line, b, e, p))
zb_min = read_real(line(b : e))
assert(next_token(line, b, e, p))
zb_max = read_real(line(b : e))
if (next_token(line, b, e, p)) then
    yb_min = read_real(line(b : e))
    assert(next_token(line, b, e, p))
    yb_max = read_real(line(b : e))
else
    yb_min = zero
    yb_max = one
end if

```

This code is used in section 8.

Set symmetry parameter.

$\langle \text{Set Symmetry 8.2} \rangle \equiv$

```

assert(next_token(line, b, e, p))
if (line(b : e)  $\equiv$  'cylindrical') then
    symmetry = geometry_symmetry_cylindrical
else if (line(b : e)  $\equiv$  'plane') then
    symmetry = geometry_symmetry_plane
else if (line(b : e)  $\equiv$  'oned') then
    symmetry = geometry_symmetry_oned
else
    assert('Unexpected_symmetry_specification'  $\equiv$  ' ')
end if

```

This code is used in section 8.

End preparation stage.

$\langle \text{End Prep 8.3} \rangle \equiv$

```

dim_elements = num_elements
var_reallocb(element_nodes)
var_reallocb(element_missing)
var_reallocb(element_assigned)

dim_nodes = num_nodes
var_reallocb(nodes)
var_reallocb(node_type)
var_reallocb(node_element_count)

dim_walls = num_walls
var_reallocb(wall_nodes)
var_reallocb(wall_elements)
var_reallocb(wall_element_count)

call init_geometry

if (symmetry  $\equiv$  geometry_symmetry_cylindrical) then
    yb_max = two * PI
else if (symmetry  $\equiv$  geometry_symmetry_plane  $\vee$  symmetry  $\equiv$  geometry_symmetry_oned) then
    yb_max = one
end if
vc_set(min_corner, xb_min, yb_min, zb_min)
vc_set(max_corner, xb_max, yb_max, zb_max)
call universal_cell(symmetry, min_corner, max_corner, vol)

if (nxd * nzd > 0) then
    ⟨ Set Mesh Polygons 11 ⟩
else /* Allocate something since it appears in argument lists later. */
    nxd = 1
    nzd = 1
    var_alloc(mesh_xz)
end if
prep_done = TRUE

```

This code is used in section 8.

9 Read and process DG file

```

⟨ Process DG File 9 ⟩ ≡
  assert(next_token(line, b, e, p))
  tmpfilename = line(b : e)
  open_file(diskin2, tmpfilename) /* Read DG's .dgo file */
  ielement = 0
  section = sec_undef /* Assume that if the file has a ".dgo" that it came from DG and needs to be
                       converted from mm to meters. Assume that it is otherwise in meters. */
  if(index(line(b : e), '.dgo') > 0) then
    mult = const(1., -3)
  else
    mult = one
  end if

dg_loop: continue
  if(read_string(diskin2, line, length)) then
    assert(length ≤ len(line))
    /* We do not want to have to manually change the format of the files DG writes. The node
       coordinates are currently comma-delimited (with possible additional spaces). Replace the
       commas in the current line with spaces so we can use our usual string utilities. */
    do i = 1, length
      if(line(i : i) ≡ ',')
        line(i : i) = ' '
    end do
    length = parse_string(line(: length))
    p = 0
    assert(next_token(line, b, e, p))
    if(line(b : e) ≡ 'p1') then
      section = sec_p1
    else if(line(b : e) ≡ 'p2') then
      section = sec_p2
    else if(line(b : e) ≡ 'misselem') then
      section = sec_miss
      /* ADD SECTION TO READ jedgi1, jedgo2, jedgo1, jedgi2 SECTIONS AND PROCESS */
    else if(line(b : e) ≡ 'ffinish') then
      section = sec_done
    else
      if(section ≡ sec_p1 ∨ section ≡ sec_p2) then
        temp_x = read_real_soft_fail(line(b : e))
        if(temp_x ≠ real_undef) then // Failure
          assert(next_token(line, b, e, p))
          temp_z = read_real_soft_fail(line(b : e))
        end if
        if(temp_x ≡ real_undef ∨ temp_z ≡ real_undef) then
          section = sec_undef
          go to dg_loop // Can trash this line since it did not match anything we've planned for
        end if
        temp_x *= mult
        temp_z *= mult
      inode = 0
    end if
  end if
end if

```

```

if (num_nodes > 0) then
  do i = 1, num_nodes
    if (nodesi,g2_x ≡ temp_x) then
      if (nodesi,g2_z ≡ temp_z) then
        assert(inode ≡ 0)
        inode = i // is a duplicate
      end if
    end if
  end do
end if
if (inode ≡ 0) then
  increment_num_nodes // is a new node
  inode = num_nodes
  nodesnum_nodes,g2_x = temp_x
  nodesnum_nodes,g2_z = temp_z
end if

if (section ≡ sec_p1) then
  increment_num_elements
  element_nodesnum_elements,element_start = inode
  element_missingnum_elements = FALSE // will be set later
  element_assignednum_elements = FALSE
else if (section ≡ sec_p2) then
  ielement ++
  element_nodesielement,element_end = inode
end if

else if (section ≡ sec_miss) then
  assert(ielement ≡ num_elements) // Compare p1 and p2 reads
  miss_elem = read_int_soft_fail(line(b : e))
  if (miss_elem > 0 ∧ miss_elem ≤ num_elements) then
    element_missingmiss_elem = TRUE
  else
    assert(miss_elem ≡ int_undef) // Assume we've stumbled into another section
    section = sec_undef
    go to dg_loop
  end if
end if
end if
if (section ≠ sec_done)
  go to dg_loop
end if
close (unit = diskin2) /* Now have elements and nodes. Characterize all of the nodes. */
do inode = 1, num_nodes
  node_element_countinode = 0
  start = FALSE end = FALSE
  do ielement = 1, num_elements
    if (element_missingielement ≡ FALSE) then
      if (element_nodesielement,element_start ≡ inode) then
        node_element_countinode ++
        start = TRUE
      else if (element_nodesielement,element_end ≡ inode) then
        node_element_countinode ++
      end if
    end if
  end do

```

```

        end if
    end do
    if (node_element_countiinode ≡ 0) then
        node_typeiinode = node_no_elements
    else if (node_element_countiinode ≡ 1) then assert ( start ≡ TRUE ∨ end ≡ TRUE )
        node_typeiinode = node_one_element
    else if (node_element_countiinode ≡ 2) then if ( start ≡ TRUE ∧ end ≡ TRUE ) then
        node_typeiinode = node_regular
    else
        node_typeiinode = node_mixed_normals
    end if
    else if (node_element_countiinode > 2) then
        node_typeiinode = node_many_elements
    end if
end do /* Set up the walls. First, start walls at all irregular nodes that start an element. */
do inode = 1, num_nodes
if (node_typeiinode ≠ node_regular ∧ node_typeiinode ≠ node_no_elements) then
    do ielement = 1, num_elements
        if (element_nodesielement,element_start ≡ inode) then
            increment_num_walls
            wall_elementsnum_walls,1 = ielement
            wall_nodesnum_walls,0 = inode
            element_assignedielement = TRUE
        end if
    end do
end if
end do /* Next, follow each of those walls until they hit another irregular node. */
do iwall = 1, num_walls
    iseg = 1
wall_loop: continue
    inode = element_nodeswall_elementsiwall,iseg,element_end
    assert(inode > 0 ∧ inode ≤ num_nodes)
    wall_nodesiwall,iseg = inode
    if (node_typeiinode ≡ node_regular) then
        iseg ++
        assert(iseg ≤ g2_num_points)
        wall_elementsiwall,iseg = 0
        do ielement = 1, num_elements
            if (element_nodesielement,element_start ≡ inode) then
                assert(wall_elementsiwall,iseg ≡ 0)
                wall_elementsiwall,iseg = ielement
                element_assignedielement = TRUE
                go to wall_loop
            end if
        end do
    else // End the wall at the next irregular node.
        wall_element_countiwall = iseg
    end if
end do /* Finally, follow all other elements connected only by regular nodes (these should be closed
surfaces). */
do ielement = 1, num_elements
    if (element_missingielement ≡ FALSE ∧ element_assignedielement ≡ FALSE) then

```

```

increment_num_walls    // An unassigned element...
wall_elements_num_walls,1 = ielement
wall_nodes_num_walls,0 = element_nodesielement,element_start
element_assignedielement = TRUE
iseg = 1
wall_loop2: continue
inode = element_nodeswall_elements num_walls,iseg,element_end
assert(inode > 0 ∧ inode ≤ num_nodes)
wall_nodesnum_walls,iseg = inode
assert(node_typeinode ≡ node_regular)
/* Search for adjacent element. Can start at ielement+1 because all previous elements have been
   assigned. If it has already been assigned, we're done with this wall. */
do ielement2 = ielement + 1, num_elements
  if ((element_nodesielement2,element_start ≡ inode) ∧ (element_assignedielement2 ≡ FALSE)) then
    iseg++
    assert(iseg ≤ g2_num_points)
    wall_elementsnum_walls,iseg = ielement2
    element_assignedielement2 = TRUE
    go to wall_loop2
  end if
end do
wall_element_countnum_walls = iseg    // End of this wall
end if
end do

@if 0
  // Move to after end of prep. stage. OK?
  var_reallocb(wall_nodes)
  var_reallocb(wall_elements)
  var_reallocb(wall_element_count)
#endif

```

This code is used in section 8.

10 Read wall file

This is intended to duplicate the wallfile functionality provided in *readgeometry*. At the same time, this format should allow a file generated with the “linear” format of this code’s *print_walls* keyword to be read back in. Hence, the numbers on the second line of the file represent the **number of points, not the number of segments**, as was the case in *readgeometry*. The points listed are checked against the existing list of nodes. If new, they are added. Either way, the new walls are stored as a list of nodes, just as are those generated from a DG file. Note that we presently make no attempt to update the elements. Only the node numbers and coordinates are used later in the code.

$\langle \text{Read Wall File 10} \rangle \equiv$

```

assert(next_token(line, b, e, p))
wallinfile = line(b : e)
open(unit = diskin2, file = wallinfile, status = 'old', form = 'formatted')

assert(read_string(diskin2, line, length))
assert(length ≤ len(line))
length = parse_string(line(: length))
p = 0

assert(next_token(line, b, e, p))
num_new_walls = read_int_soft_fail(line(b : e))
var_realloc(wall_nodes, dim_walls, num_walls + num_new_walls)
var_realloc(wall_elements, dim_walls, num_walls + num_new_walls) // not used
var_realloc(wall_element_count, dim_walls, num_walls + num_new_walls)

assert(read_string(diskin2, line, length))
assert(length ≤ len(line))
length = parse_string(line(: length))
p = 0
do iwall = 1, num_new_walls
    assert(next_token(line, b, e, p))
    wall_element_countnum_walls+iwall = read_int_soft_fail(line(b : e))
end do

do iwall = num_walls + 1, num_walls + num_new_walls
    wall_element_countiwall-- // iseg starts at 0
    do iseg = 0, wall_element_countiwall
        assert(read_string(diskin2, line, length))
        assert(length ≤ len(line))
        length = parse_string(line(: length))
        p = 0
        assert(next_token(line, b, e, p))
        x_wall = read_real_soft_fail(line(b : e))
        assert(next_token(line, b, e, p))
        z_wall = read_real_soft_fail(line(b : e))

        if (num_nodes > 0) then
            do inode = 1, num_nodes
                if ((abs(x_wall - nodesinode,g2-x) < geom_epsilon) ∧ (abs(z_wall - nodesinode,g2-z) <
geom_epsilon)) then
                    this_node = inode
                    go to wall_break
                end if
            end do
        end if
    end do
end do

```

```
    end do
  end if
  increment_num_nodes
  nodes_{num_nodes,g2_x} = x_wall
  nodes_{num_nodes,g2_z} = z_wall
  this_node = num_nodes

wall_break: continue
  wall_nodes_{iwall,iseg} = this_node
end do
end do
num_walls += num_new_walls
dim_walls = max(dim_walls, num_walls)
close(unit = diskin2)
```

This code is used in section 8.

11 Convert mesh data into polygons

$\langle \text{Set Mesh Polygons 11} \rangle \equiv$

```

/* A separate question is the direction around the physical rectangle. The polygon sent to
   decompose_polygon must be traversed in a clockwise direction in physical space. */
vc_set(yhat, zero, one, zero)
vc_set(test_vec_1, mesh_xzm(g2_x, 2, 1, 1) - mesh_xzm(g2_x, 1, 1, 1), zero, mesh_xzm(g2_z, 2, 1,
   1) - mesh_xzm(g2_z, 1, 1, 1))
vc_set(test_vec_2, mesh_xzm(g2_x, 3, 1, 1) - mesh_xzm(g2_x, 2, 1, 1), zero, mesh_xzm(g2_z, 3, 1,
   1) - mesh_xzm(g2_z, 2, 1, 1))
vc_cross(test_vec_1, test_vec_2, test_vec_3)
if (vc_product(test_vec_3, yhat) > zero) then
  mesh_sense = 1
else if (vc_product(test_vec_3, yhat) < zero) then
  mesh_sense = 2
else
  assert('First_cell_of_mesh_degenerate' ≡ ' ')
end if
do ix = 1, nxd
  do iz = 1, nzd
    zone++
    n = 0
    increment_g2_num_polygons
    g2_polygon_xz_g2_num_polygons,n,g2_x = mesh_xzm(g2_x, 1, ix, iz)
    g2_polygon_xz_g2_num_polygons,n,g2_z = mesh_xzm(g2_z, 1, ix, iz);
    n++
    if (mesh_sense ≡ 1) then
      g2_polygon_xz_g2_num_polygons,n,g2_x = mesh_xzm(g2_x, 2, ix, iz)
      g2_polygon_xz_g2_num_polygons,n,g2_z = mesh_xzm(g2_z, 2, ix, iz);
      n++
      g2_polygon_xz_g2_num_polygons,n,g2_x = mesh_xzm(g2_x, 3, ix, iz)
      g2_polygon_xz_g2_num_polygons,n,g2_z = mesh_xzm(g2_z, 3, ix, iz);
      n++
      g2_polygon_xz_g2_num_polygons,n,g2_x = mesh_xzm(g2_x, 4, ix, iz)
      g2_polygon_xz_g2_num_polygons,n,g2_z = mesh_xzm(g2_z, 4, ix, iz);
      n++
    else if (mesh_sense ≡ 2) then
      g2_polygon_xz_g2_num_polygons,n,g2_x = mesh_xzm(g2_x, 4, ix, iz)
      g2_polygon_xz_g2_num_polygons,n,g2_z = mesh_xzm(g2_z, 4, ix, iz);
      n++
      g2_polygon_xz_g2_num_polygons,n,g2_x = mesh_xzm(g2_x, 3, ix, iz)
      g2_polygon_xz_g2_num_polygons,n,g2_z = mesh_xzm(g2_z, 3, ix, iz);
      n++
      g2_polygon_xz_g2_num_polygons,n,g2_x = mesh_xzm(g2_x, 2, ix, iz)
      g2_polygon_xz_g2_num_polygons,n,g2_z = mesh_xzm(g2_z, 2, ix, iz);
      n++
    else
      assert('mesh_sense_improperly_set' ≡ ' ')
    end if
    g2_polygon_xz_g2_num_polygons,n,g2_x = g2_polygon_xz_g2_num_polygons,0,g2_x
  end do
end do

```

```

g2_polygon_xzg2_num_polygons,n,g2_z = g2_polygon_xzg2_num_polygons,0,g2_z
do i = 0, n - 1
  g2_polygon_segmentg2_num_polygons,i = i
end do
g2_polygon_segmentg2_num_polygons,n = 0 /* Now also have g2_polygon_stratum to set, but not
   clear if there is any intelligent use for it since we already have the zone number. */
g2_polygon_pointsg2_num_polygons = n
g2_polygon_zoneg2_num_polygons = zone
  /* These are really not used for solid zones and will probably contain the default values. */
do i_prop = 1, poly_int_max
  poly_int_propsg2_num_polygons,i_prop = current_int_propsi_prop
end do
do i_prop = 1, poly_real_max
  poly_real_propsg2_num_polygons,i_prop = current_real_propsi_prop
end do

zonearray1 = zone
call decompose_polygon(n, g2_polygon_xzg2_num_polygons,0,g2_x, zonearray, 1, facearray)
  /* Check to see if the mesh center is zero. If so, set using existing routines (note: triangles can
   now be specified in a Sonnet mesh format, but only if corner 4 matches corner 1 or 3). */
if ((mesh_xzm(g2_x, 0, ix, iz) ≡ zero) ∧ (mesh_xzm(g2_z, 0, ix, iz) ≡ zero)) then
  if (((mesh_xzm(g2_x, 4, ix, iz) ≡ mesh_xzm(g2_x, 1, ix, iz)) ∧ (mesh_xzm(g2_z, 4, ix,
    iz) ≡ mesh_xzm(g2_z, 1, ix, iz))) ∨ ((mesh_xzm(g2_x, 4, ix, iz) ≡ mesh_xzm(g2_x, 3,
    ix, iz)) ∧ (mesh_xzm(g2_z, 4, ix, iz) ≡ mesh_xzm(g2_z, 3, ix, iz)))) then
    call triangle_centroid(mesh_xzm(g2_x, 1, ix, iz), center)
  else
    call quad_center(mesh_xzm(g2_x, 1, ix, iz), center)
  end if
else
  vc_set(center, mesh_xzm(g2_x, 0, ix, iz), zero, mesh_xzm(g2_z, 0, ix, iz))
end if
call update_zone_info(zone, ix, iz, "plasma", n, g2_polygon_xzg2_num_polygons,0,g2_x, center)
enddo
enddo

```

This code is used in section 8.3.

12 Specify and break up a new polygon

```

⟨ New Polygon 12 ⟩ ≡
  assert(prep_done ≡ TRUE)
  process_polygon = clear_polygon
  call specify_polygon(diskin, num_walls, wall_element_count, wall_nodes, nodes, xb_min, xb_max,
    zb_min, zb_max, nxd, nzd, mesh_xz, temp_polygon0,g2-x, n, temp_int_props, temp_real_props,
    process_polygon)
  if (process_polygon ≠ clear_polygon) then
    ix = 0
    iz = 0
    do i_prop = 1, poly_int_max
      if (temp_int_propsi-prop ≠ int_uninit)
        current_int_propsi-prop = temp_int_propsi-prop
    end do
    do i_prop = 1, poly_real_max
      if (temp_real_propsi-prop ≠ real_uninit)
        current_real_propsi-prop = temp_real_propsi-prop
    end do
  if (process_polygon ≡ breakup_polygon) then
    increment_g2_num_polygons
    do i = 0, n
      g2_polygon_xzg2_num_polygons,i,g2-x = temp_polygoni,g2-x
      g2_polygon_xzg2_num_polygons,i,g2-z = temp_polygoni,g2-z
      if (i < n) then
        g2_polygon_segmentg2_num_polygons,i = i
      else
        g2_polygon_segmentg2_num_polygons,i = 0
      end if
    end do
    g2_polygon_pointsg2_num_polygons = n
    g2_polygon_zoneg2_num_polygons = zone
    g2_polygon_stratumg2_num_polygons = current_int_propspoly-stratum
    do i_prop = 1, poly_int_max
      poly_int_propsg2_num_polygons,i-prop = current_int_propsi-prop
    end do
    do i_prop = 1, poly_real_max
      poly_real_propsg2_num_polygons,i-prop = current_real_propsi-prop
    end do
    zonearray1 = zone
    call decompose_polygon(n, g2_polygon_xzg2_num_polygons,0,g2-x, zonearray, 1, facearray)
    /* Try to handle carefully those cases where we can. For n > 4, just use the first point. If this
     zone really needs a center, can break up into triangles first. */
    if (n ≡ 3) then
      call triangle_centroid(temp_polygon0,g2-x, center)
    else if (n ≡ 4) then
      call quad_center(temp_polygon0,g2-x, center)
    else
      vc_set(center, temp_polygon0,g2-x, zero, temp_polygon0,g2-z)
    end if
  end if
end if

```

```

call update_zone_info(zone, ix, iz, new_zone_type, n, g2_polygon_xzg2_num_polygons,0,g2_x, center)
else if (process_polygon ≡ triangulate_polygon ∨ process_polygon ≡ triangulate_to_zones) then
  if (process_polygon ≡ triangulate_to_zones) then
    refine = TRUE // we care
  else
    refine = FALSE // we're just breaking 'em up.
  end if
  if (current_int_props poly_num_holes > 0) then /* Not sure of the need for multiple holes.
    Easiest to implement just a single one. Note that the segment labels for the initial polygon
    are assigned in poly2triangles and transferred to the resulting triangles in temp_segment. */
    assert(current_int_props poly_num_holes ≡ 1)
    temp_holes0,g2_x = current_real_props poly_hole_x
    temp_holes0,g2_z = current_real_props poly_hole_z
  end if
  call poly2triangles(n, temp_polygon0,g2_x, current_real_props poly_min_area,
    current_int_props poly_num_holes, temp_holes0,g2_x, refine, ntriangles,
    temp_triangles0,0,g2_x, temp_segment0,0)
  assert(ntriangles < max_triangles)
  n = 3 // Number of points in a triangle !
  do j = 0, ntriangles - 1
    increment_g2_num_polygons
    do i = 0, n
      g2_polygon_xzg2_num_polygons,i,g2_x = temp_trianglesj,i,g2_x
      g2_polygon_xzg2_num_polygons,i,g2_z = temp_trianglesj,i,g2_z
      g2_polygon_segmentg2_num_polygons,i = temp_segmentj,i
    end do
    g2_polygon_pointsg2_num_polygons = n
    if (process_polygon ≡ triangulate_to_zones ∧ j > 0)
      zone = zone + 1 /* Write out stratum numbers with each of the newly created zones.
        Should probably incorporate this information into geometry2d. */
    @#if 0
      if (process_polygon ≡ triangulate_to_zones ∨ j ≡ 0) then
        write(35, *) zone, current_int_props poly_stratum
      end if
    @#endif
    g2_polygon_zoneg2_num_polygons = zone
    g2_polygon_stratumg2_num_polygons = current_int_props poly_stratum
    do i_prop = 1, poly_int_max
      poly_int_propsg2_num_polygons,i_prop = current_int_propsi_prop
    end do
    do i_prop = 1, poly_real_max
      poly_real_propsg2_num_polygons,i_prop = current_real_propsi_prop
    end do
    zonearray1 = zone
    call decompose_polygon(n, g2_polygon_xzg2_num_polygons,0,g2_x, zonearray, 1, facearray)
    if (j ≡ 0 ∨ process_polygon ≡ triangulate_to_zones)
      call triangle_centroid(temp_trianglesj,0,g2_x, center)
      call update_zone_info(zone, ix, iz, new_zone_type, n, g2_polygon_xzg2_num_polygons,0,g2_x, center)
    end do
  end if

```

end if

This code is used in section 8.

13 Read mesh file from UEDGE

```

⟨Functions and subroutines 7⟩ +≡
subroutine read_uedge_mesh(nunit, nxd, nzd, mesh_xz)

implicit none f77
implicit none f90

integer nunit, nxd, nzd // Input
real mesh_xz_nzd,nxd,0:4,g2_x:g2_z // Output
integer ix, iz, i // Local
integer corner_ptr0:4
real dummy /* Have decided to not deal with this information explicitly here. The main reason
for doing this is that we need to know the null type to decide which parameters will appear
here and that information is assumed to be known. Consistency checks of the geometry will
be performed at the end. */

read(nunit, *) // xcut1
read(nunit, *) // xcut2
read(nunit, *) /* The specific ordering to be used in the rest of this code is that
a polygon will be traced out in the clockwise direction by following the corners in order
1 → 4. The center is denoted as corner 0. The corner_ptr array is used to translate the known
orientation of UEDGE mesh corners into the desired one.

corner_ptr0 = 0
corner_ptr1 = 1
corner_ptr2 = 4
corner_ptr3 = 2
corner_ptr4 = 3

read(nunit, *) SP(((mesh_xziz,ix,corner_ptri,g2_x, ix = 1, nxd), iz = 1, nzd), i = 0, 4)
read(nunit, *) SP(((mesh_xziz,ix,corner_ptri,g2_z, ix = 1, nxd), iz = 1, nzd), i = 0, 4)

dummy = zero
open(unit = nunit + 1, file = 'uedge_sonnet', status = 'unknown', form = 'formatted')
write(nunit + 1, *)
write(nunit + 1, *) 'Element_output:'
write(nunit + 1, *)
write(nunit + 1, '(a,f16.13)') 'R*Btor=0', dummy
write(nunit + 1, ')ncut=0'
write(nunit + 1, *)
do iz = 1, nzd
    do ix = 1, nxd /* The specific ordering to be used in the rest of this code is that a polygon will
be traced out in the clockwise direction by following the corners in order 1 → 4. The corners
used here with mesh_x and mesh_z are chosen so as to translate the known orientation of
the Sonnet mesh corners into the desired one. The center is denoted as corner 0. */
    write(nunit + 1, '(a,i4,a,i3,a,i3,a,e17.10,a,e17.10,a,6x,a,e17.10,a,e17.10,a)')
        'Element', (iz - 1) * nxd + ix - 1, '=(, ix - 1, ', ',', iz - 1, ')':=(, mesh_xziz,ix,2,g2_x,
        ',', mesh_xziz,ix,2,g2_z, ')', '(', mesh_xziz,ix,3,g2_x, ',', mesh_xziz,ix,3,g2_z, ')
    write(nunit + 1, '(a,e17.10,13x,a,e17.10,a,e17.10,a)') 'Field_ratio=0', dummy,
        '(', mesh_xziz,ix,0,g2_x, ',', mesh_xziz,ix,0,g2_z, ')',
    write(nunit + 1, '(29x,a,e17.10,a,e17.10,a,6x,a,e17.10,a,e17.10,a)') '(', mesh_xziz,ix,1,g2_x, ',',
        mesh_xziz,ix,1,g2_z, ')', '(', mesh_xziz,ix,4,g2_x, ',',
        mesh_xziz,ix,4,g2_z, ')'

```

```

write(nunit + 1, *)'-----\
-----',
end do
end do
close(unit = nunit + 1)
return
end

```

14 Compute minimum and maximum coordinates

Search *nodes* array for maximum and minimum coordinate values that have been entered so far. These can then be used in setting the arguments of the *bounds* keyword.

```

⟨ Functions and subroutines 7 ⟩ +≡
subroutine find_min_max(num_nodes, nodes, node_type, x_min, x_max, z_min, z_max)
  implicit none_f77
  implicit none_f90

  integer num_nodes // Input
  integer node_type*
  real nodes*,g2_x:g2_z

  real x_min, x_max, z_min, z_max // Output
  integer inode // Local
  x_min = const(1., 5)
  x_max = zero
  z_min = const(1., 5)
  z_max = zero

  do inode = 1, num_nodes
    if (node_typeinode ≠ node_no_elements) then
      x_min = min(x_min, nodesinode,g2_x)
      x_max = max(x_max, nodesinode,g2_x)
      z_min = min(z_min, nodesinode,g2_z)
      z_max = max(z_max, nodesinode,g2_z)
    end if
  end do

  assert(x_max > x_min)
  assert(z_max > z_min)

  return
end

```

15 Print wall coordinates to a file for user examination

There are two possible formats. The first (“tabular”) has the walls laid out in successive columns, suitable for plotting with an external program. The second (“linear”) is consistent with the format used by the “wallfile” keyword. The user can provide the name of the file. If absent, the data are written to stdout.

```
"definegeometry2d.f" 15 ==
@m table_size 4 // Walls per table section

⟨Functions and subroutines 7⟩ +≡
subroutine print_walls(nunit, file_format, walloutfile, num_walls, wall_element_count, wall_nodes,
nodes)
implicit none_f77
implicit none_f90

integer nunit, num_walls // Input
integer wall_element_count_*, wall_nodes_*,0:g2_num_points-1
real nodes_*,g2_x:g2_z
character*LINELEN file_format
character*FILELEN walloutfile

integer num_sections, isec, istart, iend, max_segs, /* Local */
iwall, iseg, inode, b, e
character*csec
character*130 wall_line // Holds table_size (i.e., 4) pairs
character*31 wall_chunk // Space to write x,z pair

st_decls

if (file_format == 'linear') then
  if (walloutfile != char_undef) then
    open(unit = nunit, file = trim(walloutfile), status = 'unknown')
  end if
  write(nunit, *) num_walls
  write(nunit, *) SP(wall_element_count_iwall + 1, iwall = 1, num_walls)
  do iwall = 1, num_walls
    do iseg = 0, wall_element_count_iwall
      inode = wall_nodes_iwall,iseg
      write(nunit, '(1x,1pe18.10,2x,e18.10)') nodes_inode,g2_x, nodes_inode,g2_z
    end do
  end do
else if (file_format == 'tabular') then
  num_sections = (num_walls / table_size)
  if (num_sections * table_size < num_walls) then
    num_sections++
  else if (num_sections * table_size > num_walls) then
    assert('Problem calculating num_sections' == ' ')
  end if
  do isec = 1, num_sections
    if (walloutfile != char_undef) then
      write(csec, '(i1)') isec
      open(unit = nunit, file = trim(walloutfile) || csec, status = 'unknown')
    end if
  end do
end if
```

```

istart = table_size * (isec - 1) + 1
iend = min(table_size * isec, num_walls)
max_segs = 0
do iwall = istart, iend
  max_segs = max(max_segs, wall_element_count_iwall)
end do
do iseg = -1, max_segs
  wall_line = '||'
  b = 1
  e = 2
  do iwall = istart, iend
    if (iseg ≡ -1) then
      write(wall_chunk, '(5x,a,i2.2,11x,a,i2.2,6x)') 'X_', iwall, 'Z_', iwall
    else if (iseg ≤ wall_element_count_iwall) then
      inode = wall_nodes_iwall,iseg
      write(wall_chunk, '(1pe14.6,1x,e14.6,2x)') nodes_inode,g2_x, nodes_inode,g2_z
    else /* KaleidaGraph likes using periods to denote empty data cells. It appears that
           Excel can live with them as well. */
      write(wall_chunk, '(7x,a,14x,a,7x)') '.', '.'
    end if
    wall_line = wall_line(b : e) || wall_chunk
    e += 30
  end do
  write(nunit, *) wall_line(b : e)
end do
if (walloutfile ≡ char_undef) then
  write(nunit, *)
  write(nunit, *) '-----',
  write(nunit, *)
else
  close(unit = diskout)
end if
end do
else
  write(stderr, *) 'The format for the wall file must be either "linear" or "tabular"',
end if

return
end

```

16 Read input lines specifying a polygon

```
"definegeometry2d.f" 16 ≡
@m poly-loop #:0
@m outer-loop #:0

⟨ Functions and subroutines 7 ⟩ +≡
subroutine specify_polygon(nunit, num_walls, wall_element_count, wall_nodes, nodes, xb_min,
    xb_max, zb_min, zb_max, nxd, nzd, mesh_xz, polygon, n, int_props, real_props, process_polygon)
implicit none f77
ma_common // Common
implicit none f90

integer nunit, num_walls, nxd, nzd // Input
integer wall_element_count_*, wall_nodes_*,0:g2_num_points-1
real xb_min, xb_max, zb_min, zb_max
real nodes_*,g2_x:g2_z, mesh_xz_nzd,nxd,0:4,g2_x:g2_z

integer n, process_polygon // Output
integer int_props_1:poly_int_max
real polygon_0:g2_num_points-1,g2_x:g2_z, real_props_1:poly_real_max
integer length, p, b, e, i, wall, start, stop , temp, /* Local*/
    ix_start, ix_stop, iz_start, iz_stop, ix_step, iz_step, corner, ixi, izi, num, ix, iz, nout, step, i_prop
character*FILELEN polyoutfile
character*LINELEN line, keyword
st_decls

n = 0 /* Set these to peculiar values. If not explicitly set in this routine, they will be overridden
        by the "current" values in the calling routine. */
do i_prop = 1, poly_int_max
    int_props_i_prop = int_uninit
end do
do i_prop = 1, poly_real_max
    real_props_i_prop = real_uninit
end do

poly-loop: continue
assert(read_string(diskin, line, length))
assert(length ≤ len(line))
length = parse_string(line(:length))
p = 0
assert(next_token(line, b, e, p))
keyword = line(b : e)

if (keyword ≡ 'outer') then
    ⟨ Outer Keyword 16.1 ⟩
else if (keyword ≡ 'wall') then
    ⟨ Wall Keyword 16.2 ⟩
else if (keyword ≡ 'edge') then
    ⟨ Edge Keyword 16.3 ⟩
else if (keyword ≡ 'stratum') then
```

```

assert(next_token(line, b, e, p))
int_props_poly_stratum = read_integer(line(b : e))

else if (keyword ≡ 'material') then
  assert(next_token(line, b, e, p))
  int_props_poly_material = ma_lookup(line(b : e))
  assert(ma_check(int_props_poly_material))

else if (keyword ≡ 'temperature') then
  assert(next_token(line, b, e, p))
  real_props_poly_temperature = read_real(line(b : e))
  assert(real_props_poly_temperature > zero)

else if (keyword ≡ 'recyc_coef') then
  assert(next_token(line, b, e, p))
  real_props_poly_recyc_coef = read_real(line(b : e))
  assert(real_props_poly_recyc_coef > zero ∧ real_props_poly_recyc_coef ≤ one)

else if (keyword ≡ 'triangle_area') then
  assert(next_token(line, b, e, p))
  real_props_poly_min_area = read_real(line(b : e))
  assert(real_props_poly_min_area > zero)

else if (keyword ≡ 'triangle_hole') then
  assert(next_token(line, b, e, p))
  int_props_poly_num_holes = read_integer(line(b : e))
  if (int_props_poly_num_holes > 0) then
    assert(int_props_poly_num_holes ≡ 1)
    assert(next_token(line, b, e, p))
    real_props_poly_hole_x = read_real(line(b : e))
    assert(next_token(line, b, e, p))
    real_props_poly_hole_z = read_real(line(b : e))
  end if

else if (keyword ≡ 'print_polygon') then
  if (next_token(line, b, e, p)) then
    polyoutfile = line(b : e)
    nout = diskout
  else
    polyoutfile = char_undef
    nout = stdout
  end if
  if (polyoutfile ≠ char_undef) then
    open(unit = nout, file = trim(polyoutfile), status = 'unknown')
  end if
  write(nout, '(8x,a,15x,a)') 'X', 'Z'
  do i = 0, n - 1
    write(nout, '(1x,1pe14.6,2x,e14.6)') polygoni,g2-x, polygoni,g2-z
  end do

else if (keyword ≡ 'breakup_polygon') then
  polygonn,g2-x = polygon0,g2-x
  polygonn,g2-z = polygon0,g2-z
  process_polygon = breakup_polygon

```

```

return

else if (keyword  $\equiv$  'triangulate_polygon') then
    polygonn,g2-x = polygon0,g2-x
    polygonn,g2-z = polygon0,g2-z
    process_polygon = triangulate_polygon
    return

else if (keyword  $\equiv$  'triangulate_to_zones') then
    polygonn,g2-x = polygon0,g2-x
    polygonn,g2-z = polygon0,g2-z
    process_polygon = triangulate_to_zones
    return

else if (keyword  $\equiv$  'clear_polygon') then
    process_polygon = clear_polygon
    return

else
    write (stderr, *) 'Unexpected_polygon_keyword'
    return

end if

go to poly_loop

end

```

Add points along universal cell to current polygon.

```

⟨ Outer Keyword 16.1 ⟩ ≡
outer_loop: continue
    if (next_token(line, b, e, p)) then
        i = read_integer(line(b : e))
        if (i  $\equiv$  0  $\vee$  i  $\equiv$  4) then
            polygonn,g2-x = xb_min
            polygonn,g2-z = zb_min
        else if (i  $\equiv$  1) then
            polygonn,g2-x = xb_min
            polygonn,g2-z = zb_max
        else if (i  $\equiv$  2) then
            polygonn,g2-x = xb_max
            polygonn,g2-z = zb_max
        else if (i  $\equiv$  3) then
            polygonn,g2-x = xb_max
            polygonn,g2-z = zb_min
        end if
        n++
        assert(n  $\leq$  g2_num_points - 1)
        go to outer_loop
    end if

```

This code is used in section 16.

Take points from a wall and add to current polygon.

```

⟨ Wall Keyword 16.2 ⟩ ≡
  assert(next_token(line, b, e, p))
  wall = read_integer(line(b : e))
  assert(next_token(line, b, e, p))
  if (line(b : e) ≡ '*') then
    start = 0 stop = wall_element_countwall
  else
    start = read_integer(line(b : e))
    assert(next_token(line, b, e, p))
    if (line(b : e) ≡ '*') then stop = wall_element_countwall
    else stop = read_integer(line(b : e))
    end if
  end if
  if (next_token(line, b, e, p)) then
    assert(line(b : e) ≡ 'reverse')
    temp = start start = stop stop = temp
  end if if ( start ≤ stop ) then
    step = 1 else if ( start > stop ) then
    step = -1
  end if
  do i = start , stop , step
    assert(i ≥ 0 ∧ i ≤ wall_element_countwall)
    polygonn,g2-x = nodeswall.nodeswall,i,g2-x
    polygonn,g2-z = nodeswall.nodeswall,i,g2-z
    n++
    assert(n ≤ g2_num_points - 1)
  end do

```

This code is used in section 16.

Take points along a mesh edge and add to current polygon.

```

⟨Edge Keyword 16.3⟩ ≡
  assert(next_token(line, b, e, p))
  if (line(b : e) ≡ '*') then
    ix_start = 0
    ix_stop = nxd
  else
    ix_start = read_integer(line(b : e))
    assert(next_token(line, b, e, p))
    ix_stop = read_integer(line(b : e))
  end if
  assert(next_token(line, b, e, p))
  if (line(b : e) ≡ '*') then
    iz_start = 0
    iz_stop = nzd
  else
    iz_start = read_integer(line(b : e))
    assert(next_token(line, b, e, p))
    iz_stop = read_integer(line(b : e))
  end if
  if (next_token(line, b, e, p)) then
    assert(line(b : e) ≡ 'reverse')
    temp = ix_start
    ix_start = ix_stop
    ix_stop = temp
    temp = iz_start
    iz_start = iz_stop
    iz_stop = temp
  end if
  if (iz_stop ≡ iz_start) then
    iz_step = 0
    if (ix_stop ≥ ix_start) then
      ix_step = 1
      num = ix_stop - ix_start + 1
    else
      ix_step = -1
      num = ix_start - ix_stop + 1
    end if
  else if (ix_stop ≡ ix_start) then
    ix_step = 0
    if (iz_stop ≥ iz_start) then
      iz_step = 1
      num = iz_stop - iz_start + 1
    else
      iz_step = -1
      num = iz_start - iz_stop + 1
    end if
  end if
  assert('Not following a mesh edge' ≡ ' ')
end if
ix = ix_start

```

```
iz = iz_start
do i = 1, num
  if (ix ≡ 0 ∧ iz ≡ 0) then
    corner = 1
    ixi = 1
    izi = 1
  else if (ix ≡ 0) then
    corner = 2
    ixi = 1
    izi = iz
  else if (iz ≡ 0) then
    corner = 4
    ixi = ix
    izi = 1
  else
    corner = 3
    ixi = ix
    izi = iz
  end if
  polygonn,g2_x = mesh_xzizi,ixi,corner,g2_x
  polygonn,g2_z = mesh_xzizi,ixi,corner,g2_z
  ix += ix_step
  iz += iz_step
  n++
end do
```

This code is used in section 16.

17 Fill in data for this zone

NOTE: the center point is set using the argument *center* from the first call to this routine for the current zone (i.e., the routine should be called for each polygon comprising a complex zone).

<Functions and subroutines 7> +≡

```

subroutine update_zone_info(zone, ind_x, ind_z, new_zone_type, n, polygon, center)

implicit none_f77
zn_common // Common
implicit none_f90

integer zone, ind_x, ind_z, n // Input
real polygon0:g2_num_points-1,g2_x:g2_z
character(*) new_zone_type

vc_decl(center)

vc_decls
gi_ext

if (zn_volume(zone) ≡ real_undef) then
    zn_volume(zone) = zero
    zn_index(zone, 1) = ind_x
    zn_index(zone, 2) = ind_z
    if (new_zone_type ≡ "solid") then
        zn_type_set(zone, zn_solid)
    else if (new_zone_type ≡ "exit") then
        zn_type_set(zone, zn_exit)
    else if (new_zone_type ≡ "vacuum") then
        zn_type_set(zone, zn_vacuum)
    else if (new_zone_type ≡ "plasma") then
        zn_type_set(zone, zn_plasma)
    else if (new_zone_type ≡ "exit") then
        zn_type_set(zone, zn_exit)
    else
        write (stderr, *) 'Unknown_zone_type:', new_zone_type
        assert( $\mathcal{F}$ )
    end if
    call set_zn_min_max(n, polygon0,g2_x, zone, T)
    vc_copy(center, zone_center_zone)
else
    call set_zn_min_max(n, polygon0,g2_x, zone,  $\mathcal{F}$ )
end if
    zn_volume(zone) = zn_volume(zone) + polygon_volume(n, polygon0,g2_x)
return
end

```

18 Find the centroid of a triangle

This is computed using the fact that the centroid (equivalently, the center of gravity and the intersection of the medians) is at trilinear coordinates $\frac{1}{a} : \frac{1}{b} : \frac{1}{c}$, where a , b , and c are the lengths of the sides of the triangles (for additional information see <http://mathworld.wolfram.com/TrilinearCoordinates.html>).

```
<Functions and subroutines 7> +≡
subroutine triangle_centroid(triangle, center)
  implicit none_f77
  implicit none_f90

  real triangle0:3,g2_x:g2_z // Input
  vc_decl(center) // Output

  real area, k, a, b, c, alpha, gamma, den, lc // Local

  vc_decl(a_vertex)
  vc_decl(b_vertex)
  vc_decl(c_vertex)
  vc_decl(a_vec)
  vc_decl(b_vec)
  vc_decl(c_vec)
  vc_decl(a_hat)
  vc_decl(c_hat)
  vc_decl(bc_cross)
  vc_decl(c_perp)
  vc_decl(neg_y)
  vc_decl(m_test1)
  vc_decl(m_test2)

  vc_decls
    /* Associate point 0 with vertex A, 1 with B, and 2 with C. Here are the lengths of the sides. */
    vc_set(a_vertex, triangle0,g2_x, zero, triangle0,g2_z)
    vc_set(b_vertex, triangle1,g2_x, zero, triangle1,g2_z)
    vc_set(c_vertex, triangle2,g2_x, zero, triangle2,g2_z)
    /* Vectors for two of the sides and the corresponding unit vectors. */
    vc_difference(b_vertex, a_vertex, c_vec)
    vc_unit(c_vec, c_hat)
    vc_difference(c_vertex, b_vertex, a_vec)
    vc_unit(a_vec, a_hat)
    /* Perpendicular to c, define using cross product to be sure it's pointing the right direction. */
    vc_set(neg_y, zero, -one, zero)
    vc_cross(c_hat, neg_y, c_perp) /* Lengths of all 3 sides. */
    vc_difference(a_vertex, c_vertex, b_vec)
    a = vc_abs(a_vec)
    b = vc_abs(b_vec)
    c = vc_abs(c_vec) /* Area by using the cross product: */
    vc_cross(b_vec, c_vec, bc_cross)
    area = half * vc_abs(bc_cross) /* Use  $\alpha$ ,  $\beta$ ,  $\gamma$  to denote the trilinear coordinate values. A constant
       of proportionality  $k$  relates these to the distances of the points to the triangle sides.  $k$  can be
       determined for any particular set of coordinates via the area of the triangle.
```

$$k \equiv \frac{2\Delta}{a\alpha + b\beta + c\gamma},$$

In the case of the centroid with $\alpha = 1/a$, etc., the denominator is just 3. */
 $\alpha = \text{one} / a$
 $\gamma = \text{one} / c$
 $k = \text{two} * \text{area} / \text{const}(3)$ /* The vectors used in the Web page referred to in the introduction to this routine do not appear to be a consistent set. The vectors we use here along the sides trace out the triangle in a clockwise direction. The Cartesian coordinates of the point of interest are written in terms of some distance l_c along side c (starting at vertex A) plus a distance $k\gamma$ along \hat{c}_\perp (by definition of γ). A similar expression is written going from vertex C along $-\hat{a}$, and the two are solved for the two distances along the sides. We only need one here:

$$l_c = \frac{-k\alpha + \gamma k(\hat{a}_1\hat{c}_1 + \hat{a}_3\hat{c}_3) + \hat{a}_3(A_1 - C_1) + \hat{a}_1(C_3 - A_3)}{\hat{a}_1\hat{c}_3 - \hat{a}_3\hat{c}_1}.$$

Note: l_c can be negative for a really stretched out triangle. */
 $\text{den} = \text{a_hat}_1 * \text{c_hat}_3 - \text{a_hat}_3 * \text{c_hat}_1$
 $\text{assert}(\text{den} > \text{zero})$
 $\text{lc} = (-k * \alpha + \gamma * k * (\text{a_hat}_1 * \text{c_hat}_1 + \text{a_hat}_3 * \text{c_hat}_3) + \text{a_hat}_3 * (\text{a_vertex}_1 - \text{c_vertex}_1) + \text{a_hat}_1 * (\text{c_vertex}_3 - \text{a_vertex}_3)) / \text{den}$
/* Finally, the Cartesian coordinates are given in general by

$$\vec{x} = \vec{A} + l_c \hat{c} + k\gamma \hat{c}_\perp.$$

*/
 $\text{vc_xvt}(\text{a_vertex}, \text{c_hat}, \text{lc}, \text{center})$
 $\text{vc_xvt}(\text{center}, \text{c_perp}, k * \gamma, \text{center})$
/* Verify that this point is indeed inside the triangle. */
 $\text{vc_difference}(\text{center}, \text{a_vertex}, \text{m_test1})$
 $\text{vc_cross}(\text{m_test1}, \text{c_vec}, \text{m_test2})$
 $\text{assert}(\text{vc_product}(\text{m_test2}, \text{neg_y}) > \text{zero})$
 $\text{vc_difference}(\text{center}, \text{b_vertex}, \text{m_test1})$
 $\text{vc_cross}(\text{m_test1}, \text{a_vec}, \text{m_test2})$
 $\text{assert}(\text{vc_product}(\text{m_test2}, \text{neg_y}) > \text{zero})$
 $\text{vc_difference}(\text{center}, \text{c_vertex}, \text{m_test1})$
 $\text{vc_cross}(\text{m_test1}, \text{b_vec}, \text{m_test2})$
 $\text{assert}(\text{vc_product}(\text{m_test2}, \text{neg_y}) > \text{zero})$
return
end

19 Compute the center of a quadrilateral

```
⟨ Functions and subroutines 7 ⟩ +≡
subroutine quad_center(quad, center)

implicit none_f77
implicit none_f90

real quad0:4,g2_x:g2_z      // Input
vc_decl(center)
// Output /* Use this as a temporary approximation. To be useful, zone must be convex.
vc_set(center, const(0.25) * (quad0,g2_x + quad1,g2_x + quad2,g2_x + quad3,g2_x), zero,
      const(0.25) * (quad0,g2_z + quad1,g2_z + quad2,g2_z + quad3,g2_z))

return
end
```

20 Set up default sectors

This code has been lifted directly out of *readgeometry.web*. In fact, were it not for the problem-specific macros there, could have replaced that code with this subroutine call as well.

```
"definegeometry2d.f" 20 ≡
@m next_surface #:0

⟨Functions and subroutines 7⟩ +≡
subroutine default_sectors(num_polygons, polygon_points, polygon_xz, polygon_segment,
polygon_zone, poly_int_props, poly_real_props)
implicit none_f77
zn_common // Common
sc_common
implicit none_f90

integer num_polygons // Input
integer polygon_segment*,0:g2-num_points-1, polygon_points*, polygon_zone*,
poly_int_props*,1:poly_int_max
real polygon_xz*,0:g2-num_points-1,g2-x:g2-z, poly_real_props*,1:poly_real_max
integer i_poly, j, sector1, face1, zone1, zone2 // Local
⟨Memory allocation interface 0⟩

gi_ext /* Set up default sectors. These are taken to be at the interface between plasma / vacuum
and solid / exit zones. The procedure consists of first cycling through all of the polygons of zone
type solid or exit and checking each surface to see if there is a plasma or vacuum zone on the
other side. The filter used here is that find_poly_zone arranges for the type of zone1 to match
that of polygon i_poly. This is done so that, for example, a target sector is identified by having
zone1 be a solid and zone2 be plasma. The target sector is then associated with zone2 (and its
corresponding face, -face1); zone1 is passed to define_sector as a check. */
do i_poly = 1, num_polygons
  do j = 0, polygon_pointsi_poly-1
    if (polygon_xzi_poly,j,g2-x ≠ polygon_xzi_poly,j+1,g2-x ∨ polygon_xzi_poly,j,g2-z ≠
      polygon_xzi_poly,j+1,g2-z) then
      call find_poly_zone(polygon_xzi_poly,j,g2-x, polygon_xzi_poly,j+1,g2-x,
      zn_type(polygon_zonei_poly), zone1, face1, zone2)
      if (zn_check(zone1) ∧ zn_check(zone2)) then
        if ((zn_type(zone1) ≡ zn_solid) ∧ (zn_type(zone2) ≡ zn_plasma)) then
          sector1 = define_sector(poly_int_propsi_poly,poly_stratum, polygon_segmenti_poly,j,-face1,
          zone2, zone1)
          define_sector_target(sector1, poly_int_propsi_poly,poly_material,
          poly_real_propsi_poly,poly_temperature * boltzmanns_const,
          poly_real_propsi_poly,poly_recyc_coef)
        else if ((zn_type(zone1) ≡ zn_solid) ∧ (zn_type(zone2) ≡ zn_vacuum)) then
          sector1 = define_sector(poly_int_propsi_poly,poly_stratum, polygon_segmenti_poly,j,-face1,
          zone2, zone1)
          define_sector_wall(sector1, poly_int_propsi_poly,poly_material,
          poly_real_propsi_poly,poly_temperature * boltzmanns_const,
          poly_real_propsi_poly,poly_recyc_coef)
        else if ((zn_type(zone1) ≡ zn_exit) ∧ ((zn_type(zone2) ≡ zn_plasma))) then
```

```
sector1 = define_sector(poly_int_propsi-poly,poly,stratum, polygon_segmenti-poly,j, -face1,  
zone2, zone1)  
define_sector_exit(sector1)  
else if ((zn_type(zone1) ≡ zn_exit) ∧ ((zn_type(zone2) ≡ zn_vacuum))) then  
sector1 = define_sector(poly_int_propsi-poly,poly,stratum, polygon_segmenti-poly,j, -face1,  
zone2, zone1)  
define_sector_exit(sector1)  
else  
    goto next_surface  
end if  
end if  
end if next_surface continue  
end do  
end do  
return end
```

21 Print data from Triangle

This is taken directly from the *tricall* example program that is provided with Triangle.

```
< C Functions 21 >C ≡
void report(io, markers, reporttriangles, reportneighbors, reportsegments, reportedges,
reportnorms, reportholes)
struct triangulateio *io;
int markers;
int reporttriangles;
int reportneighbors;
int reportsegments;
int reportedges;
int reportnorms;
int reportholes;
{
int i, j;
for (i = 0; i < io→numberofpoints; i++)
{
    printf("Point %4d:", i);
    for (j = 0; j < 2; j++)
    {
        printf("% .6g", io→pointlisti*2+j);
    }
    if (io→numberofpointattributes > 0)
    {
        printf("\n\nattributes");
    }
    for (j = 0; j < io→numberofpointattributes; j++)
    {
        printf("% .6g", io→pointattributelisti*io→numberofpointattributes+j);
    }
    if (markers)
    {
        printf("\nmarker %d\n", io→pointmarkerlisti);
    }
    else
    {
        printf("\n");
    }
}
printf("\n");

if (reporttriangles ∨ reportneighbors)
{
    for (i = 0; i < io→numberoftriangles; i++)
    {
        if (reporttriangles)
        {
            printf("Triangle %4d points:", i);
            for (j = 0; j < io→numberofcorners; j++)
            {

```

```

    printf(" %4d", io->trianglelist[i*io->numberofcorners+j]);
}
if (io->numberoftriangleattributes > 0)
{
    printf(" attributes");
}
for (j = 0; j < io->numberoftriangleattributes; j++)
{
    printf(" %.6g", io->triangleattributelist[i*io->numberoftriangleattributes+j]);
}
printf("\n");
if (reportneighbors)
{
    printf("Triangle %d neighbors:", i);
    for (j = 0; j < 3; j++)
    {
        printf(" %d", io->neighborlist[i*3+j]);
    }
    printf("\n");
}
printf("\n");
if (reportsegments)
{
    for (i = 0; i < io->numberofsegments; i++)
    {
        printf("Segment %d points:", i);
        for (j = 0; j < 2; j++)
        {
            printf(" %d", io->segmentlist[i*2+j]);
        }
        if (markers)
        {
            printf(" marker %d\n", io->segmentmarkerlist[i]);
        }
        else
        {
            printf("\n");
        }
    }
    printf("\n");
}
if (reportededges)
{
    for (i = 0; i < io->numberofedges; i++)
    {
        printf("Edge %d points:", i);
        for (j = 0; j < 2; j++)
        {
    
```

```

printf(" %4d", io→edgelisti*2+j);
}
if (reportnorms ∧ (io→edgelisti*2+1 ≡ -1))
{
  for (j = 0; j < 2; j++)
  {
    printf(" %.6g", io→normlisti*2+j);
  }
}
if (markers)
{
  printf("marker %d\n", io→edgemarkerlisti);
}
else
{
  printf("\n");
}
printf("\n");
}
if (reportholes)
{
  for (i = 0; i < io→numberofholes; i++)
  {
    printf("Hole %d points:", i);
    for (j = 0; j < 2; j++)
    {
      printf(" %.6g", io→holelisti*2+j);
    }
  }
  printf("\n");
}
}

```

See also section 22.

This code is used in section 6.1.

22 C interface routine To triangle

This is based loosely on the *tricall* example that is provided with the Triangle package.

```
(C Functions 21)C +≡
void poly2triangles_(npoints, polygon, area, nholes, hole, refine, ntriangles, triangles, markers)
{
    int *npoints; // Input
    int *nholes;
    int *refine;
    double *area;
    double (*polygon)2;
    double (*hole)2;

    int *ntriangles; // Output
    double (*triangles)4,2;
    int (*markers)4;
}

struct triangulateio in, mid, out; // Local
struct triangulateio *final;
int i, j, n, nh, pt, jt, k, idup, nunq, ip, munq;
int pointmapg2-num_points, pointinvmapg2-num_points;
int segmapg2-num_points,2;

n = *npoints; /* Have rewritten this code to eliminate duplicate points and segments from the
   input polygon. */
nunq = 0; // Number of unique points
for (i = 0; i < n; ++i)
{
    idup = 0; // = 0 for a new point
    if (i > 0)
    {
        for (j = 0; j < i; ++j)
        {
            if ((polygoni,0 ≡ polygonj,0) ∧ (polygoni,1 ≡ polygonj,1))
            {
                pointmapi = pointmapj; // A duplicate point;
                idup = 1; // keep track of which it matched
            }
        }
    }
    if (idup ≡ 0)
    {
        pointmapi = nunq; // Unique point number as fn. of original
        pointinvmapnunq = i; // The inverse of that
        nunq++; // Increment after =i arrays start at 0
    }
} /* Analogous code for segments. Note that the orientation of the segments is unimportant to
   Triangle, so we must test both directions in identifying duplicates. */
munq = 0; // Number of unique segments
for (i = 0; i < n; ++i)
{
    /* We need to explicitly close our polygon by setting up a segment from the last point to the
       first point. Can do that most transparently by defining ip as: */
    if (i ≡ n - 1)
```

```

{
  ip = 0;
}
else
{
  ip = i + 1;
}
if (pointmapi ≡ pointmapip)
{
  idup = 1; // Throw out trivial segments
}
else
{
  idup = 0;
  if (munq > 0)
  {
    for (j = 0; j < munq; ++j)
    {
      if (((pointmapi ≡ segmapj,0) ∧ (pointmapip ≡ segmapj,1)) ∨ (((pointmapi ≡
          segmapj,1) ∧ (pointmapip ≡ segmapj,0))))
      {
        idup = 1; // A genuine duplicate segment
      }
    }
  }
  if (idup ≡ 0)
  {
    // Define map for unique segments
    segmapmunq,0 = pointmapi; // 0 and 1 just denote the ends
    segmapmunq,1 = pointmapip;
    munq++; // Again, so arrays start at 0
  }
} /* Can now allocate and set arrays that will be passed to Triangle. */
in.numberofpoints = nunq;
in.numberofpointattributes = 0;
in.pointlist = (double *) malloc(in.numberofpoints * 2 * sizeof(double));
in.pointmarkerlist = (int *) malloc(in.numberofpoints * sizeof(int));
j = 0;
for (i = 0; i < nunq; ++i)
{
  in.pointlistj = polygonpointinvmapi,0;
  j++;
  in.pointlistj = polygonpointinvmapi,1;
  j++; /* Point markers are intended to be assigned in a manner analogous to that used above
         for breakup-polygon. However, Triangle only respects nonzero markers. So, add 1 here and
         subtract when transferring back to main code. */
  in.pointmarkerlisti = pointinvmapi + 1;
}
in.numberofsegments = munq;
in.segmentlist = (int *) malloc(in.numberofsegments * 2 * sizeof(int));
j = 0;
for (i = 0; i < munq; ++i)

```

```

{
  in.segmentlistj = segmapi,0;
  j++;
  in.segmentlistj = segmapi,1;
  j++;
}

nh = *nholes;
in.numberofholes = nh;
if (nh > 0)
{
  in.holelist = (double *) malloc(in.numberofholes * 2 * sizeof(double));
  j = -1;
  for (i = 0; i < nh; ++i)
  {
    j++;
    in.holelistj = holei,0;
    j++;
    in.holelistj = holei,1;
  }
}
else
{
  in.holelist = (double *) NULL;
}
in.numberoftregions = 0;
in.numberoftriangles = 0;
in.regionlist = (double *) NULL;
in.segmentmarkerlist = (int *) NULL;
@if 0
printf("Input_point_set:\n\n");
report(&in, 0, 0, 0, 0, 0, 0, 1);
#endif
mid.pointlist = (double *) NULL;
mid.pointmarkerlist = (int *) NULL;
mid.trianglelist = (int *) NULL;
mid.segmentlist = (int *) NULL;
mid.segmentmarkerlist = (int *) NULL;
mid.edgelist = (int *) NULL;
mid.edgemarkerlist = (int *) NULL;
mid.normlist = (double *) NULL;
triangulate("pzQ", &in, &mid, (struct triangulateio *) NULL);
#if 0
printf("Initial_triangulation:\n\n");
report(&mid, 0, 1, 0, 1, 0, 0, 1);
#endif
out.pointlist = (double *) NULL;
out.pointmarkerlist = (int *) NULL;
out.trianglelist = (int *) NULL;
out.segmentlist = (int *) NULL;
out.segmentmarkerlist = (int *) NULL;
out.edgelist = (int *) NULL;

```

```

out.edgemarkerset = (int *) NULL;
out.normlist = (double *) NULL;

if (*refine == TRUE)
{
    char args1_16 = "rpqa";
    char args2 = "zYQ";
    char areastr9;

    sprintf(areastr9, "%8.6f", *area);
    strcat(args1, areastr9);
    strcat(args1, args2);

@#if 0
    The minimum area here should probably be user specified.
    triangulate("rpqzYQ", &mid, &out, (struct
        triangulateio *) NULL);
@#endif
    triangulate(args1, &mid, &out, (struct triangulateio *) NULL);

@#if 0
    printf("Final triangulation:\n\n");
    report(&out, 0, 1, 0, 1, 0, 0, 1);
@#endif

    final = &out;
}
else
{
    final = &mid;
}

if (final->numberofcorners != 3)
{
    printf("STOP!!! Triangle thinks triangles have %i corners",
        final->numberofcorners);
}
if (final->numberoftangles > max_triangles - 1)
{
    printf("STOP!!! Number of triangles %i exceeds dimensions of triangles array!",
        final->numberoftangles);
}

for (i = 0; i < final->numberoftangles; ++i)
{
    for (j = 0; j < final->numberofcorners; ++j)
    {
        pt = final->trianglelist[i*final->numberofcorners+j];
        jt = final->numberofcorners - j; // Reverse order for decompose_polygon
        triangles[i,jt,0] = final->pointlist[2*pt];
        triangles[i,jt,1] = final->pointlist[2*pt+1];
        /* Subtract 1 here to undo the shift made prior to the first call. */
        markers[i,jt] = final->pointmarkerlist[pt] - 1;
    }
    triangles[i,0,0] = triangles[i,3,0];
    triangles[i,0,1] = triangles[i,3,1];
    markers[i,0] = markers[i,3];
}
*ntriangles = final->numberoftangles;

```

```
free(in.pointlist);
free(in.segmentlist);
free(in.pointmarkerlist);
free(in.regionlist);
free(in.segmentmarkerlist);
free(mid.pointlist);
free(mid.segmentlist);
free(mid.pointmarkerlist);
free(mid.trianglelist);
free(mid.segmentmarkerlist);
free(mid.edgelist);
free(mid.edgemarkerlist);
free(mid.normlist);
free(out.pointlist);
free(out.segmentlist);
free(out.pointmarkerlist);
free(out.trianglelist);
free(out.segmentmarkerlist);
free(out.edgelist);
free(out.edgemarkerlist);
free(out.normlist);
if (nh > 0)
{
    free(in.holelist);
}
}
```

23 INDEX

a: 18.
a_hat: 18.
a_vec: 18.
a_vertex: 18.
abs: 10.
alpha: 18.
aname: 7.
area: 18, 22^C.
areastrings: 22^C.
args1: 22^C.
args2: 22^C.
assert: 8, 8.1, 8.2, 9, 10, 11, 12, 14, 15, 16, 16.1, 16.2, 16.3, 17, 18.
aunit: 7.
b: 8, 15, 16, 18.
b_vec: 18.
b_vertex: 18.
bc_cross: 18.
be: 22^C.
boltzmanns_const: 20.
boundaries_neighbours: 8.
bounds: 1, 14.
breakup_polygon: 1, 7, 12, 16, 22^C.
c: 18.
c_hat: 18.
c_perp: 18.
c_vec: 18.
c_vertex: 18.
center: 8, 11, 12, 17, 18, 19.
char_undef: 8, 15, 16.
check_geometry: 8.
clear_polygon: 1, 7, 12, 16.
command_arg: 7.
const: 8, 9, 14, 18, 19.
corner: 16, 16.3.
corner_ptr: 13.
csec: 15.
current_int_props: 8, 11, 12.
current_real_props: 8, 11, 12.
declare_varp: 8.
decompose_polygon: 1, 11, 12, 22^C.
def_geom_2d_main: 7, 8.
default_sectors: 8, 20.
define_dimen: 8.
define_sector: 20.
define_sector_exit: 20.
define_sector_target: 20.
define_sector_wall: 20.
define_varp: 8.
definegeometry2d: 1, 7.
degas2: 1.
den: 18.
dg_file: 1.
dg_loop: 1, 9.
diag_sector_setup: 8.
dim_elements: 7, 8, 8.3.
dim_nodes: 7, 8, 8.3.
dim_walls: 7, 8, 8.3, 10.
diskin: 8, 12, 16.
diskin2: 8, 9, 10.
diskout: 8, 15, 16.
dummy: 13.
e: 8, 15, 16.
edge: 1.
edgelist: 21^C, 22^C.
edgemarkerlist: 21^C, 22^C.
element_assigned: 7, 8, 8.3, 9.
element_end: 7, 8, 9.
element_ends_ind: 8.
element_ind: 8.
element_missing: 7, 8, 8.3, 9.
element_nodes: 7, 8, 8.3, 9.
element_start: 7, 8, 9.
end_detectors: 8.
end_prep: 1.
end_sectors: 8.
eof: 8.
erase_geometry: 8.
facearray: 8, 11, 12.
face1: 20.
FALSE: 8, 9, 12.
file: 7, 8, 10, 13, 15, 16.
FILE: 1.
file_format: 8, 15.
fileid: 8.
FILELEN: 7, 8, 15, 16.
filename: 7, 8.
final: 22^C.
find_min_max: 8, 14.
find_poly_zone: 20.
FLOAT: 8.
form: 7, 8, 10, 13.
free: 22^C.
gamma: 18.
geom_epsilon: 10.
geometry: 1.
geometry_symmetry_cylindrical: 8.2, 8.3.
geometry_symmetry_none: 8.
geometry_symmetry_oned: 8.2, 8.3.

geometry_symmetry_plane: 8.2, 8.3.
geometry2d: 12.
gi_ext: 8, 17, 20.
g2_common: 8.
g2_ncdecl: 8.
g2_ncdef: 8.
g2_ncwrite: 8.
g2_num_points: 7, 8, 9, 15, 16, 16.1, 16.2, 17, 20, 22^C.
g2_num_polygons: 7, 8, 11, 12.
g2_points_ind: 8.
g2_points_ind0: 8.
g2_poly_ind: 8.
g2_polygon_points: 7, 8, 11, 12.
g2_polygon_segment: 7, 8, 11, 12.
g2_polygon_stratum: 7, 8, 11, 12.
g2_polygon_xz: 7, 8, 11, 12.
g2_polygon_zone: 7, 8, 11, 12.
g2_x: 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 16.1, 16.2, 16.3, 17, 18, 19, 20.
g2_xz_ind: 8.
g2_z: 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 16.1, 16.2, 16.3, 17, 18, 19, 20.
half: 18.
here: 22^C.
hole: 22^C.
holelist: 21^C, 22^C.
hweb: 1, 8.
i: 8, 13, 16, 21^C, 22^C.
i_inc_g2: 7, 8.
i_poly: 20.
i_prop: 8, 11, 12, 16.
idup: 22^C.
ielement: 8, 9.
ielement2: 8, 9.
iend: 15.
implicit_none_f77: 7, 8, 13, 14, 15, 16, 17, 18, 19, 20.
implicit_none_f90: 7, 8, 13, 14, 15, 16, 17, 18, 19, 20.
in: 1, 22^C.
increment_g2_num_polygons: 7, 11, 12.
increment_num_elements: 7, 9.
increment_num_nodes: 7, 9, 10.
increment_num_walls: 7, 9.
ind_x: 17.
ind_z: 17.
index: 9.
init_geometry: 8.3.
inode: 8, 9, 10, 14, 15.
INT: 8.
int_props: 16.
int_undef: 9.
int_uninit: 12, 16.
int_unused: 7.
io: 21^C.
iostat: 7.
ip: 22^C.
isec: 15.
iseg: 8, 9, 10, 15.
istart: 15.
iwall: 8, 9, 10, 15.
ix: 7, 8, 11, 12, 13, 16, 16.3.
ix_start: 16, 16.3.
ix_step: 16, 16.3.
ix_stop: 16, 16.3.
ixi: 16, 16.3.
iz: 7, 8, 11, 12, 13, 16, 16.3.
iz_start: 16, 16.3.
iz_step: 16, 16.3.
iz_stop: 16, 16.3.
izi: 16, 16.3.
j: 8, 20, 21^C, 22^C.
jt: 22^C.
jxz: 7.
k: 18, 22^C.
keyword: 8, 16.
lc: 18.
len: 8, 9, 10, 16.
length: 8, 9, 10, 16.
line: 7, 8, 8.1, 8.2, 9, 10, 16, 16.1, 16.2, 16.3.
LINELEN: 8, 15, 16.
loop1: 8.
m_test1: 18.
m_test2: 18.
ma_check: 16.
ma_common: 16.
ma_lookup: 16.
malloc: 22^C.
markers: 21^C, 22^C.
material: 1.
materials_infile: 1.
max: 7, 10, 14, 15.
max_corner: 8, 8.3.
max_segs: 15.
max_triangles: 7, 8, 12, 22^C.
mem_inc: 8.
mesh_corner_ind: 8.
mesh_sense: 8, 11.
mesh_x: 13.
mesh_xz: 7, 8, 8.3, 12, 13, 16, 16.3.

mesh_xz_ind: 8.
mesh_xzm: 7, 11.
mesh_z: 13.
mid: 22^C.
min: 14, 15.
min_corner: 8, 8.3.
minimum: 22^C.
miss_elem: 8, 9.
mult: 8, 9.
munq: 22^C.
n: 8, 16, 17, 22^C.
nc: 8.
NC_CLOBBER: 8.
nc_decls: 8.
nc_read_materials: 7.
nc_stat: 8.
ncclose: 8.
nccreate: 8.
ncendef: 8.
neg_y: 18.
neighborlist: 21^C.
new_polygon: 1.
new_zone: 1.
new_zone_type: 8, 12, 17.
next_surface: 1, 20.
next_token: 8, 8.1, 8.2, 9, 10, 16, 16.1, 16.2, 16.3.
nh: 22^C.
nholes: 22^C.
node_element_count: 7, 8, 8.3, 9.
node_ind: 8.
node_many_elements: 7, 9.
node_mixed_normals: 7, 9.
node_no_elements: 7, 9, 14.
node_one_element: 7, 9.
node_regular: 7, 9.
node_type: 7, 8, 8.3, 9, 14.
node_undefined: 7.
nodes: 7, 8, 8.3, 9, 10, 12, 14, 15, 16, 16.2.
normlist: 21^C, 22^C.
nout: 16.
npoints: 22^C.
ntriangles: 8, 12, 22^C.
NULL: 22^C.
null_detector_setup: 8.
num: 16, 16.3.
num_elements: 7, 8, 8.3, 9.
num_new_walls: 1, 8, 10.
num_nodes: 7, 8, 8.3, 9, 10, 14.
num_polygons: 20.
num_sections: 15.
num_walls: 7, 8, 8.3, 9, 10, 12, 15, 16.
numberofcorners: 21^C, 22^C.
numberofedges: 21^C.
numberofholes: 21^C, 22^C.
numberofpointattributes: 21^C, 22^C.
numberofpoints: 21^C, 22^C.
numberoffregions: 22^C.
numberofsegments: 21^C, 22^C.
numberoftriangleattributes: 21^C.
numberoftriangles: 21^C, 22^C.
nunit: 8, 13, 15, 16.
nunq: 22^C.
nxd: 7, 8, 8.3, 11, 12, 13, 16, 16.3.
nzd: 8, 8.3, 11, 12, 13, 16, 16.3.
one: 8, 8.1, 8.3, 9, 11, 16, 18.
open_file: 7, 8, 9.
open_stat: 7, 8.
out: 22^C.
outer: 1.
outer_loop: 16, 16.1.
p: 8, 16.
parse_string: 8, 9, 10, 16.
PI: 8.3.
pointattributelist: 21^C.
pointinvmap: 22^C.
pointlist: 21^C, 22^C.
pointmap: 22^C.
pointmarkerlist: 21^C, 22^C.
poly_hole_x: 7, 8, 12, 16.
poly_hole_z: 7, 8, 12, 16.
poly_int_ind: 8.
poly_int_max: 7, 8, 11, 12, 16, 20.
poly_int_props: 7, 8, 11, 12, 20.
poly_loop: 16.
poly_material: 7, 8, 16, 20.
poly_min_area: 7, 8, 12, 16.
poly_num_holes: 7, 8, 12, 16.
poly_real_ind: 8.
poly_real_max: 7, 8, 11, 12, 16, 20.
poly_real_props: 7, 8, 11, 12, 20.
poly_recyc_coef: 7, 8, 16, 20.
poly_stratum: 7, 8, 12, 16, 20.
poly_temperature: 7, 8, 16, 20.
polygon: 8, 16, 16.1, 16.2, 16.3, 17, 22^C.
polygon_nc_file: 1, 8.
polygon_points: 20.
polygon_segment: 20.
polygon_volume: 17.
polygon_xz: 20.
polygon_zone: 20.
polyoutfile: 16.
poly2triangles: 12.
poly2triangles_: 22^C.

prep_done: 8, 8.3, 12.
print_min_max: 1.
print_polygon: 1.
print_walls: 1, 8, 10, 15.
printf: 21^C, 22^C.
probably: 22^C.
process_polygon: 8, 12, 16.
pt: 22^C.

quad: 19.
quad_center: 11, 12, 19.
quit: 1.

read_int_soft_fail: 9, 10.
read_integer: 8, 16, 16.1, 16.2, 16.3.
read_real: 8.1, 16.
read_real_soft_fail: 9, 10.
read_sonnet_mesh: 1, 8.
read_string: 8, 9, 10, 16.
read_uedge_mesh: 1, 8, 13.
readfilenames: 7.
readgeometry: 1, 10, 20.
REAL: 6.1^C.
real_props: 16.
real_undef: 8, 9, 17.
real_uninit: 12, 16.
recyc_coef: 1.
refine: 8, 12, 22^C.
regionlist: 22^C.
report: 21^C, 22^C.
reportedges: 21^C.
reportholes: 21^C.
reportneighbors: 21^C.
reportnorms: 21^C.
reportsegments: 21^C.
reporttriangles: 21^C.
reverse: 1.

sc_common: 20.
sec_done: 7, 9.
sec_miss: 7, 9.
sec_p1: 7, 9.
sec_p2: 7, 9.
sec_undef: 7, 9.
section: 8, 9.
sector: 1, 8.
sector1: 20.
segmap: 22^C.
segmentlist: 21^C, 22^C.
segmentmarkerlist: 21^C, 22^C.
set_zn_min_max: 17.
should: 22^C.
sonnet_mesh: 1.
SP: 13, 15.

specified: 22^C.
specify_polygon: 12, 16.
sprintf: 22^C.
st_decls: 8, 15, 16.
start: 8, 9, 16, 16.2.
status: 7, 8, 10, 13, 15, 16.
stderr: 7, 15, 16, 17.
stdout: 8, 16.
step: 16, 16.2.
strata_segment: 1.
stratum: 1.
strcat: 22^C.
symmetry: 1, 8, 8.2, 8.3.

table_size: 15.
temp: 16, 16.2, 16.3.
temp_holes: 8, 12.
temp_int_props: 8, 12.
temp_polygon: 8, 12.
temp_real_props: 8, 12.
temp_segment: 8, 12.
temp_triangles: 8, 12.
temp_x: 8, 9.
temp_z: 8, 9.
temperature: 1.
test_vec_1: 8, 11.
test_vec_2: 8, 11.
test_vec_3: 8, 11.
The: 22^C.
this_node: 8, 10.
tmpfilename: 8, 9.
triangle: 18.
triangle_area: 1.
triangle_centroid: 11, 12, 18.
triangle_hole: 1.
triangleattributelist: 21^C.
trianglelist: 21^C, 22^C.
triangles: 22^C.
triangulate: 22^C.
triangulate_polygon: 1, 7, 12, 16.
triangulate_to_zones: 1, 7, 12, 16.
triangulateio: 21^C, 22^C.
tricall: 21, 22.
trim: 8, 15, 16.
TRUE: 8, 8.3, 9, 12, 22^C.
two: 8.3, 18.

uedge_mesh: 1.
unit: 7, 8, 9, 10, 13, 15, 16.
universal_cell: 8.3.
update_zone_info: 11, 12, 17.
user: 22^C.

var_alloc: 8, 8.3.

var_free: 8.
var_realloca: 7.
var_reallocb: 8, 8.3, 9.
var_reallocc: 10.
vc_abs: 18.
vc_copy: 17.
vc_cross: 11, 18.
vc_decl: 8, 17, 18, 19.
vc_decls: 8, 17, 18.
vc_difference: 18.
vc_product: 11, 18.
vc_set: 8.3, 11, 12, 18, 19.
vc_unit: 18.
vc_xvt: 18.
vol: 8, 8.3.

wall: 1, 16, 16.2.
wall_break: 1, 10.
wall_chunk: 15.
wall_element_count: 7, 8, 8.3, 9, 10, 12, 15, 16, 16.2.
wall_elements: 7, 8, 8.3, 9, 10.
wall_ind: 8.
wall_line: 15.
wall_loop: 1, 9.
wall_loop2: 1, 9.
wall_nodes: 7, 8, 8.3, 9, 10, 12, 15, 16, 16.2.
wallfile: 1.
wallfiles: 1.
wallinfile: 8, 10.
walloutfile: 8, 15.
wall2d_ind: 8.
web: 1, 20.
write_geometry: 8.
write_poly_nc: 8.

x_max: 8, 14.
x_min: 8, 14.
x_wall: 8, 10.
xb_max: 8, 8.1, 8.3, 12, 16, 16.1.
xb_min: 8, 8.1, 8.3, 12, 16, 16.1.
xcut1: 13.
xcut2: 13.

yb_max: 8, 8.1, 8.3.
yb_min: 8, 8.1, 8.3.
yhat: 8, 11.

z_max: 8, 14.
z_min: 8, 14.
z_wall: 8, 10.
zb_max: 8, 8.1, 8.3, 12, 16, 16.1.
zb_min: 8, 8.1, 8.3, 12, 16, 16.1.
zcut: 13.
zero: 7, 8, 8.1, 11, 12, 13, 14, 16, 17, 18, 19.

⟨ C Functions 21, 22 ⟩ Used in section 6.1.
⟨ Edge Keyword 16.3 ⟩ Used in section 16.
⟨ End Prep 8.3 ⟩ Used in section 8.
⟨ Functions and subroutines 7, 8, 13, 14, 15, 16, 17, 18, 19, 20 ⟩ Used in section 6.1.
⟨ Memory allocation interface 0 ⟩ Used in sections 20 and 8.
⟨ New Polygon 12 ⟩ Used in section 8.
⟨ Outer Keyword 16.1 ⟩ Used in section 16.
⟨ Process DG File 9 ⟩ Used in section 8.
⟨ Read Wall File 10 ⟩ Used in section 8.
⟨ Set Bounds 8.1 ⟩ Used in section 8.
⟨ Set Mesh Polygons 11 ⟩ Used in section 8.3.
⟨ Set Symmetry 8.2 ⟩ Used in section 8.
⟨ Wall Keyword 16.2 ⟩ Used in section 16.

COMMAND LINE: "fweave -f -i! -W[-ykw700 -ytw40000 -j -n/
/u/dstotler/degas2/src/definegeometry2d.web".

WEB FILE: "/u/dstotler/degas2/src/definegeometry2d.web".

CHANGE FILE: (none).

GLOBAL LANGUAGE: FORTRAN.