Appendix I

Geometry Files (GEM Files)

I.1 Introduction

SIMION provides a variety of methods for defining the geometry of electrode/pole points in a potential array: The Modify function, geometry files (GEM), STL 3D CAD files, or externally/programmatically defined potential arrays such as via the SL Libraries. Modify allows the user to interactively create, modify, and view electrode/pole point geometry. Geometry files are typically used for complex 3D geometry and/or an any geometry definitions that may need to be scaled (e.g. doubled without introducing the jags) or parameterized.

Geometry files can either be used in conjunction with Modify or as a stand-alone method for geometry definition within New. The Modify function has a geometry file development system within it. Modify's geometry file development system provides a quick way to write, test, and modify geometry files.

Caution
GEM files are an advanced SIMION feature. It requires that you learn a geometry definition language. However, the effort you spend learning to create geometry files will add powerful capabilities to your SIMION bag of tricks.

SIMION 7 Note
SIMION 8.0 and SIMION 7.0 GEM files are identical, though 8.0.x version may add certain extensions.[a]

[a] 8.0.4 added Lua macro and variable substitution support. See the page "GEM Geometry File > Macro Support" in the supplementary electronic documentation (Help menu).

What Is a Geometry File?

A geometry file is an ASCII file with a `.GEM` extension (e.g. `TEST.GEM`) that uses a 3D solid geometry modeling language to define the desired electrode/pole array geometry via a series of fill (and other) instructions. Geometry instructions are similar in structure to C language functions. However, unlike C language functions, geometry language instructions are nested (somewhat like in PASCAL) to enhance their power.

How Does SIMION Process a `.GEM` File?

The geometry language compiler in SIMION reads the selected geometry file and converts its `fill` (and other) instructions into a decision list in RAM. SIMION then uses this decision list to determine the fate of each point in the target potential array. ***This is done one point at a time.***

SIMION takes the coordinates of the point and ***looks from the last Fill defined*** toward the first Fill defined in the decision list until it encounters the first Fill's volume that contains the point. When and if this Fill is encountered the point is changed to the Fill's value and the search stops for that point.

The effect is the same as if each Fill were applied to the potential array as it was encountered while reading the geometry file (not an efficient approach). ***Each point will have the type and value of the last Fill in the geometry file that changed it. This is a very important concept to remember!***

Two Examples of a SIMION `.GEM` File

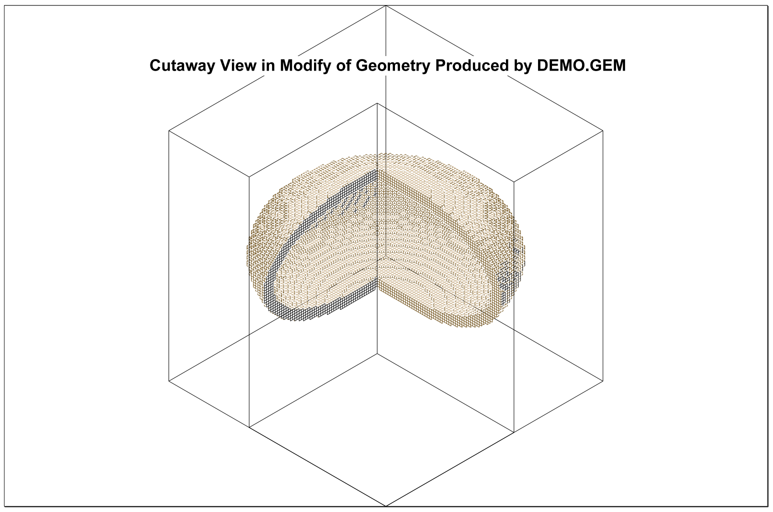


Figure I.1: Electrode geometry created by demo.gem with cutaway view in Modify.

Below are two listings of `.GEM` files that create a 101x, 101y, 101z 3D potential array and insert a hollow one volt electrode point ellipsoid in the middle. Note: These geometry files are coded in two very different (though legal) styles. It is suggested that you study these simple examples carefully. They provide an introduction to the world of SIMION geometry files:

A Questionable Geometry Definition Style

```
pa_define(101,101,101,p,n)locate(50,50,50){e(1){fill{within{sphere ↩
    (,,,45,25,45)}notin{sphere(,,,40,20,40)}}}}
```

A Suggested Geometry Definition Style

Example I.1 demo.gem

```
; This geometry file creates a 101,101,101 planar non-mirrored 3d PA and
; inserts a hollow ellipsoid in the middle
; Define PA to create

pa_define(101,101,101,planar,non-mirrored)
; Define ellipsoid

locate(50,50,50,1,0,0,0)             ; locate geometry origin in center of  ↩
   PA
  {
  electrode(1)                       ; electrode of one volt
    {
    fill                             ; solid fill
      {
      within{sphere(0,0,0,45,25,45)} ; within ellipsoid centered at  ↩
         geometry
                                     ;   origin rx = 45, ry = 25, rz = 45
      notin{sphere(0,0,0,40,20,40)}  ; not in ellipsoid centered at  ↩
         geometry
                                     ;   origin rx = 40, ry = 20, rz = 40
                                     ;   (hollow out ellipsoid)
      }
    }
  }
```

A Quick Demo of Geometry Files

Do the following if you want a demonstration of geometry files (Figure I.1). Start SIMION. Click the New button. Click the Use Geometry File button. Navigate to the `geometry` directory (in the SIMION program folder) and select the `demo.gem` file. `demo.gem` is the example above. After the PA is created and the geometry is inserted, look at it with the `View` function. Cut the ellipsoid in half to verify that it is hollow.

Now exit View and enter the Modify function. Click the GeomF button to access the geometry development facility. Click the Edit button (accesses the text editor) to view the geometry file. Exit the text editor. Now erase the potential array by clicking the Erase Entire PA button. Click the Cancel button to return to Modify and verify that the array is erased. Click the GeomF button to reenter the geometry development facility. Now click the Insert into PA button and the geometry will be reinserted. ***The above example should serve to wet your appetite for the material below.***

I.2 Geometry Language Rules

Both of the above geometry definitions will generate the same potential array and electrode geometry and will run at the same speed. However, the second example will be easy to support and modify later. Remember, you have the freedom to make your geometry definitions as cryptic or verbose as you like. There are several characteristics of the language that should be apparent:

Upper and Lower Case: SIMION ***ignores the case*** of the geometry instructions entered. You may use upper and lower case freely to improve readability.

Blank Lines and Indention's: Blank lines are ignored. Use blank lines to create good visual separation of various regions of geometry definitions. You may indent as desired. When properly used, indention can significantly improve instruction readability (particularly nested geometry language instructions).

The Semicolon ; Starts an In-line Comment: In-line comments begin with a semicolon. All information after the semicolon (including the semicolon) is ignored by the geometry compiler (to the end of the current line). ***These comments have no effect on the speed of geometry files (so use them!)***.

Line length Limits: The geometry compiler ignores characters ***beyond column 200*** in all lines.

Instruction Oriented Language Structure: Geometry files are composed of a ***nested*** collection of geometry instructions (instructions are contained within other instructions). Examples of the three formats used for geometry instructions are discussed below:

1. Type 1: `sphere(„,45,25,45)`
2. Type 2: `fill{}`
3. Type 3: `locate(50,50,50){}`

The Instruction Name - Required: All Types: All geometry instructions begin with their name (e.g. `sphere`). SIMION supports the use of synonyms for many of its geometry instructions. For example: The letter "e" is a synonym of "electrode". The specific discussions of each instruction give its synonyms (if any).

Parameter List ( ) - Required: Types 1 & 3: A parameter list (when required) immediately follows the instruction name and is delimited by a left and right parenthesis (e.g. ( )). One or more parameters (as required by the specific instruction) are entered within the parenthesis delimiters. Parameters can be numbers or words (depending on the instruction) and are separated by commas and/or spaces—e.g. `(1,2,3)` is the same as `(1 2 3)`.

Most parameters have a default value. That is, if you skip them, SIMION will assume a value for them. For example: `circle()` is equivalent to `circle(0,0,10)`. You may use commas to skip (use default values for) one or more parameters. For example: `circle(10„30)` is equivalent to `circle(10,0,30)`.

The specific discussions of each instruction give its parameter requirements (if any) along with its default parameter values.

The Instruction Scope { } - Required: Types 2 & 3: Many instructions have what is known as a scope (or range of effect). ***The scope of an instruction is limited to instructions that appear within its wavy brackets (e.g.*** `{ }`***).*** These scope brackets always appear immediately after the instruction name (type 2) or after the required parameter list (type 3).

Scope is a very important concept in geometry files because it is used for nesting instructions (placing instructions inside of instructions). It delimits the range of effect or bounds of an instruction. The following instruction segment serves as an example:

```
electrode(100)      ; use electrode of 100 volts
  {
  fill{....}        ; fill something with 100 volt electrode points
  electrode(200)    ; use electrode of 200 volts
    {
    fill{...}       ; fill something with 200 volt electrode points
    }
  fill{...}         ; fill something with 100 volt electrode points
  }
fill{...}           ; fill something with 0 V electrode points (default)
```

In the example above the first `fill` uses electrode points of 100 volts. The second `fill` uses electrode points of 200 volts because it is inside the scope of a 200 volt electrode definition. The third `fill` uses electrode points of 100 volts because it is in the scope of the 100 volt electrode points. The fourth `fill` is using the default value of 0 volt electrode (assuming that this is not an included geometry file).

As of 8.0.4, Lua macros, including variable substitutions, can be embedded in GEM files.[1]

I.2.1 Classes of Instructions

There are several classes of instructions. The following discusses each class of instruction and gives the names of the instructions in that class:

PA Definition Class

Instructions: `pa_define()`

Defines potential array to create if `.GEM` called from New function. This must always be the first instruction in `.GEM` file if used (optional instruction).

Include Class

Instructions: `include()`

The Include Class instruction allows a `.GEM` file to reference another `.GEM` file. Thus you can keep component definitions in separate `.GEM` files and have a base `.GEM` file reference and insert these components in their desired locations.

Point Definition Class

Instructions: `electrode(){}`, `non_electrode(){}`, `pole(){}` (synonym for `electrode`), `non_pole(){}` (synonym for `non_electrode`)

The Point Definition Class of instructions define the point type and potential to use in fills ***within*** their scope. ***Note: pole and electrode are synonyms.***

[1] "GEM Geometry File > Macro Support" page in supplemental electronic documentation (Help menu).

Fill Class

Instructions: `fill{},edge_fill{},rotate_fill(){},rotate_edge_fill(){}`

Fill Class instructions define the four types of fills supported in geometry files: fill is used for full defined volume fills, `edge_fill` is used to change only the boundary points of the defined volume. `rotate_fill` is used to create a fill volume by rotating a surface of revolution through an angle. `rotate_edge_fill` is used to change only the boundary points of a fill volume produced by rotating a surface of revolution through an angle.

The scope of these instructions contains the volume inclusion and exclusion instructions (e.g. `Withins` and `Notins` below) to be used to determine the volume they act upon. To be acted on by a fill instruction (any of the four types) a point ***must be within a defined inclusion volume (e.g. `within`) and not within any defined exclusion volume (e.g. `notin`)***.

Within Class

Instructions: `within{},within_inside{},within_inside_or_on{},notin{}, notin_inside{},notin_inside_or_on{}`

The six Within Class instructions always appear within the scope of a Fill Class instruction. They contain one or more Test Class instructions (e.g. `circle()`). Each Test Class instruction within the scope of a Within Class instruction is tested and the results are ***logically ANDed***. Thus in the following statement group: `within{sphere(0,0,0,25)sphere(20,0,0,25)}` only those points that fall within both sphere definitions will be considered to be within the inclusion volume.

`within`, `within_inside`, and `within_inside_or_on` are used to define inclusion volumes. `notin`, `notin_inside`, and `notin_inside_or_on` are used to define exclusion volumes. More than one of the six `within` or `notin` type instructions can appear within the scope of any Fill Class instruction. If more than one of each appears their results are ***logically ORed***. Thus a point (to be acted on by a fill) ***must be within at least one of the `Withins` and not within any of the `Notins`***.

There are ***three forms*** of each Within Class instruction. Each form uses a ***slightly different*** inclusion test to help you make your geometry definitions more precise, and less scaling dependent. The following explains the inclusion tests performed for each type:

`within` and `notin`

The inclusion test performed for this class of instructions requires that the array point be within 0.5 ***grid unit*** beyond the outer boundary to be considered included:

```
within{circle(0,0,r)}    points included < r + (0.5 gu)
notin{circle(0,0,r)}     points included < r + (0.5 gu)
```

`within_inside` and `notin_inside`

The inclusion test performed for this class of instructions requires that the array point be ***inside but not on*** the outer boundary to be considered included:

```
within_inside{circle(0,0,r)}    points included < r
notin_inside{circle(0,0,r)}     points included < r
```

`within_inside_or_on` and `notin_inside_or_on`

The inclusion test performed for this class of instructions requires that the array point be inside, on, ***but not outside*** the outer boundary to be considered included:

```
within_inside{circle(0,0,r)}    points included <= r
notin_inside{circle(0,0,r)}     points included <= r
```

Test Class

Instructions: `box(),centered_box(),corner_box()` `box3d(),centered_box3d(),` `corner_box3d(),circle(),cylinder(),sphere()` `parabola(),hyperbola()` `points(),points3d(),polyline()`

Test Class instructions can ***only*** appear within the scope of a Within Class instruction (e.g. `within` or `notin`). They test to see if the point in question is within their volume. If it is, they return a logical TRUE. If more than one Test Class instruction appears within a Within Class instruction (e.g. within) their results are logically ANDed. ***All tests must return TRUE in a Within Class instruction for the result to be considered TRUE.***

Note that test class instructions contain both 2D (e.g. `circle()`) and 3D (e.g. `sphere()`) instructions. All 2D instructions are defined in terms of an xy plane (e.g. z = 0). However all 2D instructions are ***assumed*** to extend to plus or minus 10^6 in the z axis direction so they can be used with 3D arrays. Any 3D instruction can also be used in a 2D array. It will test TRUE in the areas where it intersects the z = 0 plane (xy plane).

Of course all this becomes much more interesting (complex) when one sprinkles Location Class instructions within the scope of a geometry file (below). ***Note: Test Class instructions have minimal definition options because Locate Class instruction(s) can always be used to more precisely define them.***

Location Class

Instructions: `locate(){},project(){}` (synonym for `locate`)

Location Class instructions are used to locate (e.g. project) the geometry axis within their scope (by location, scaling and orientation) onto the geometry axis outside their scope. These are the ***workhorse instructions of geometry files*** because they can appear within ***any*** scope in any geometry file. ***Most errors result from the misuse of `Locate` instructions.***

As a contrived example: Let's assume we want a cylindrical tube (with elliptical hole) with ends cut at a 45 degree angle (forming elliptical ends). Further, we want the resulting tube pointed down and centered on the x-axis. See the instruction fragment shown below:

```
locate(0,0,0,1,-90)                ; swing cylinder -90 degrees (cw) to ↩
    center on x-axis
    {
    fill                           ; volume fill with current point and type
```

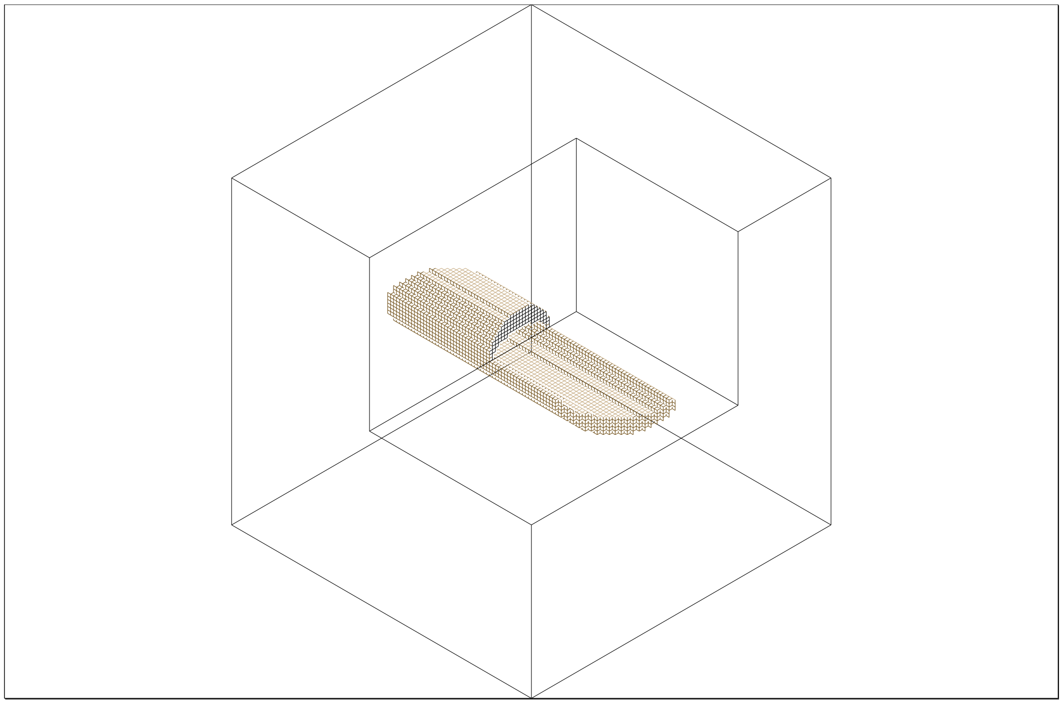


Figure I.2: Electrode geometry created by demo2.gem.

```
        {
        within                ; include volume
            {
            circle(0,0,10)        ; radius 10 tube with infinite z extent
            locate(0,0,0,1,45)    ; swing box volume ccw 45 degrees
                {
                box(-20,-100,20,100)    ; box of x width of 40 (infinite z  ↩
                   extent)
                }            ; use large values of y so circle limits  ↩
                   volume in y
            }
        notin{circle(0,0,8,4)}    ; bores 8 rx by 4 ry elliptical hole in  ↩
           circular rod
        }
    }
```

To understand what goes on here (without yet knowing instruction details) ***one must work from the inside out when dealing with `Locate`s*** (projects). Thus the box must be thought of as rotated ccw 45 degrees in azimuth in order to compute its intersection volume with the circular rod of the `circle` instruction. The result is then rotated cw 90 degrees in azimuth to center it along the x-axis beyond the outer locate (Figure I.2 - `demo2.gem`). ***Wasn't that fun!***

I.2.2 Instruction Nesting Rules

Geometry instruction classes have rules concerning when and where they can appear in a geometry file. This is called instruction nesting rules. These nesting rules make use of the notion of nesting levels. ***The geometry language compiler enforces these rules!***

A geometry file has three nesting levels: Base Nesting Level, Fill Nesting Level, and Within Nesting Level. A geometry file is at Base Nesting Level whenever instructions are outside the scope of a Fill Class instruction. The level is at Fill Nesting Level whenever instructions are inside the immediate scope of any Fill Class instruction. Finally, the level is at Within Nesting Level whenever instructions are inside the immediate scope of any Within Class instruction. The sample geometry code fragment below serves to demonstrate nesting levels:

```
e(...)
  {                         ; at Base Nesting Level
  locate(...)
    {                       ; at Base Nesting Level
    fill
      {                     ; at Fill Nesting Level
      locate(...)
        {                   ; at Fill Nesting Level
        within
          {                 ; at Within Nesting Level
          circle(...)
          locate(...){box(...)}
          }                 ; return to Fill Nesting Level
        }                   ; at Fill Nesting Level
      notin
        {                   ; at Within Nesting Level
        hyperbola(...)
        }                   ; return to Fill Nesting Level
      }                     ; return to Base Nesting Level
    }                       ; at Base Nesting Level
  }                         ; at Base Nesting Level
```

The following gives the classes of commands that can appear within each of the three nesting levels:

Classes Legal in Base Nesting Level

- Point Definition Class
- Include Class
- Fill Class
- Location Class

Classes Legal in Fill Nesting Level

- Within Class
- Location Class

Classes Legal in Within Nesting Level

- Test Class
- Location Class

Note:

1. Location Class instructions (e.g. `locate`) are ***legal at any nesting level.***
2. Also, a PA Definition Class instruction (e.g. `pa_define`) must always be the first base level instruction in a `.GEM` file ***if used***.

I.3 Geometry Instructions

The following is a detailed discussion of each legal geometry instruction. Instruction synonyms (if any) appear after the "or:".

I.3.1 `box` or: `box2d`

Format:	`box(xmin, ymin, xmax, ymax)`
Default Values:	`box(0,0,10,10)`
Class:	Test Class
When Legal:	Within Nesting Level

Defines 2D box (e.g. on current xy plane: z = 0). Parameters define the 2D min. and max. corner locations of the box. The box extends plus or minus 10^6 in z. Returns TRUE if point within its bounds.

```
box(-5,,50,60)
```

Defines 2D box with lower left corner at -5x, 0y and upper right corner at 50x, 60y. zmin is -10^6 and zmax is 10^6.

I.3.2 `box3d`

Format:	`box3d(xmin, ymin, zmin, xmax, ymax, zmax)`
Default Values:	`box3d(0,0,0,10,10,10)`
Class:	Test Class
When Legal:	Within Nesting Level

Defines 3D box. Parameters define the 3D min. and max. corner locations of the box. Returns TRUE if point within its bounds.

```
box(-5,,-10,50,60,100)
```

Defines 3D box with lower left corner at -5x, 0y, -10z and upper right corner at 50x, 60y, 100z.

I.3.3 `centered_box` or: `centered_box2d`, `cent_box`, `cent_box2d`

Format:	`centered_box(xc, yc, xw, yw)`
Default Values:	`centered_box(0,0,10,10)`
Class:	Test Class
When Legal:	Within Nesting Level

Defines 2D centered box (e.g. on current xy plane). Parameters define the center point and dimensions of the box. The centered box extends plus or minus 10^6 in z. Returns TRUE if point within its bounds.

```
centered_box(-5,,50,60)
```

Defines 2D centered box with center at -5x, 0y and dimensions of 50x wide, 60y high. zmin is -10^6 and zmax is 10^6.

I.3.4 `centered_box3d` or: `cent_box3d`

Format:	`centered_box3d(xc, yc, zc, xw, yw, zw)`
Default Values:	`centered_box3d(0,0,0,10,10,10)`
Class:	Test Class
When Legal:	Within Nesting Level

Defines 3D centered box. Parameters define the center point and dimensions of the box. Returns TRUE if point within its bounds.

```
centered_box(-5,,-10,50,60,100)
```

Defines 3D centered box with center at -5x, 0y, -10z and dimensions of 50x wide, 60y high, 100z deep.

I.3.5 `circle` or: `ellipse`

Format:	`circle(xc, yc, rx, ry)`
Default Values:	`circle(0,0,10,10)`
Class:	Test Class
When Legal:	Within Nesting Level

Defines 2D circle or ellipse (e.g. on current xy plane). Parameters define the center and radii. The 2D circle extends plus or minus 10^6 in z. Note: If ry is defaulted, a circle of radius rx will be drawn. Returns TRUE if point within its bounds.

```
circle(15,20,30,10)
```

Defines and ellipse centered 15x, 20y with radii of 30 rx, 10 ry. zmin is -10^6 and zmax is 10^6.

I.3.6 `corner_box` or: `corner_box2d`, `corn_box`, `corn_box2d`

Format:	`corner_box(xmin, ymin, xw, yw)`
Default Values:	`corner_box(0,0,10,10)`
Class:	Test Class
When Legal:	Within Nesting Level

Defines 2D corner box (e.g. on current xy plane). Parameters define the min. corner point (***lower left***) and dimensions of the box. The corner box extends plus or minus 10^6 in z. Returns TRUE if point within its bounds.

```
corner_box(-5,,50,60)
```

Defines 2D corner box with lower left corner at -5x, 0y and dimensions of 50x wide, 60y high. zmin is -10^6 and zmax is 10^6.

I.3.7 `corner_box3d` or: `corn_box3d`

Format:	`corner_box3d(xmin, ymin, zmin, xw, yw, zw)`
Default Values:	`corner_box3d(0,0,0,10,10,10)`
Class:	Test Class
When Legal:	Within Nesting Level

Defines 3D corner box. Parameters define the min. corner point (lower left) and dimensions of the box. Returns TRUE if point within its bounds.

```
corner_box3d(-5,,-10,50,60,100)
```

Defines 3D corner box with lower left corner at -5x, 0y, -10z and dimensions of 50x wide, 60y high, 100z deep.

I.3.8 `cylinder`

Format:	`cylinder(xc, yc, zc, rx, ry, length)`
Default Values:	`cylinder(0,0,0,10,10,10)`
Class:	Test Class
When Legal:	Within Nesting Level

Defines 3D circular or elliptical cylinder. Parameters define the center at one end, radii, and length. ***Length is always converted to its absolute value and extends in the minus z axis direction. Note: If ry is defaulted, a cylinder of radius rx will be drawn.*** Returns TRUE if point within its bounds.

```
cylinder(15,20,30,10,,50)
```

Defines a circular cylinder with one end centered at 15x, 20y, 30z with a radius of 10 and length of 50 (e.g. extending from z = 30 to z = -20).

I.3.9 `edge_fill` or: `edge_fill_volume`

Format:	`edge_fill { }`
Default Values:	NA
Class:	Fill Class
When Legal:	Base Nesting Level

Defines a volume edge fill using the currently active point type and potential. It raises the nesting level to Fill Nesting Level within its scope (e.g. { }). ***Its scope must contain at least one `Within` or `Notin` instruction.*** If no `Within` instruction is supplied, the point is assumed within, subject to rejection by `Notin` tests. ***There is no limit to the number of `Within`s or `Notin`s that can appear inside the scope of a Fill Class instruction.***

The `edge_fill` instruction fills only the boundary points (edge) of the equivalent fill volume. Thus if a normal fill would create a solid sphere, an `edge_fill` would create a spherical shell. This has the same function as the Edge command in Modify.

```
edge_fill
    {
    within{sphere(0,0,0,50,20,50)}
    notin{circle(0,0,20,10)}
    }
```

Creates an ellipsoid shell with an elliptical shell passing through its center in the z direction.

I.3.10 `electrode` or: `e`, `p`, `elect`, `pole`, `electrode_points`, `pole_p-oints`

Format:	`electrode(potential) { }`
Default Values:	`electrode(0) { }`
Class:	Point Definition Class
When Legal:	Base Nesting Level

Defines fill point type of electrode or pole and its associated potential to use within the instruction's scope (e.g. {}).

```
electrode(1)
    {
    fill{...}    ;  electrode/pole points of one volt used for fill
    }
```

I.3.11 `fill` or: `fill_volume`

Format:	`fill { }`
Default Values:	NA
Class:	Fill Class
When Legal:	Base Nesting Level

Defines a full volume fill using the currently active point type and potential. It raises the nesting level to Fill Nesting Level within its scope (e.g. { }). ***Its scope must contain at least one within or notin instruction.*** If no within instruction is supplied, the point is assumed within, subject to rejection by notin tests. ***There is no limit to the number of withins or notins that can appear inside the scope of a fill class instruction.***

Each within or notin instruction defines a separate volume (through the use of tests: e.g. `circle()`). Points are checked against these tests to determine if the various `Withi-ns` and/or `Notins` are TRUE. ***A point is considered to be within the fill if it is inside at least one `Within` volume and not inside any `Notin` volumes.*** The results of multiple `Withins` and/or `Notins` are ***ORed*** with their own kind to satisfy the above test.

```
fill
    {
    within{circle(0,0,20)}
    within{circle(100,20,20)}
    notin_inside{circle(0,0,10)}
    notin_inside{circle(100,20,10)}
    }
```

Fills two hollow circles (2D potential array) or two hollow tubes (3D potential array). The tubes are centered at 0x, 0y and 100x, 20y. Both have an outside radius of 20 and an inside radius of 10. This of course assumes that there are no locates outside of the fill and thus the units are in potential array grid units.

I.3.12 `hyperbola`

Format:	`hyperbola(xc, yc, rx, ry)`
Default Values:	`hyperbola(0,0,10,10)`
Class:	Test Class
When Legal:	Within Nesting Level

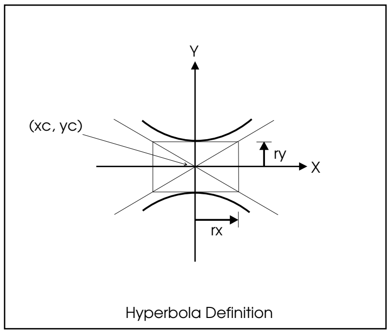


Figure I.3: Hyperbola definition.

Defines 2D hyperbola in the y-axis direction only (e.g. on current xy plane). Parameters define the center and focus radial offset of vertices in x and y. The 2D hyperbola extends plus or minus 10^6 in z. Returns TRUE if point within its bounds.

Function used: $(y - yc)^2/ry^2 - (x - xc)^2/rx^2 = 1$

```
hyperbola(50,0,10,20)
```

Defines hyperbola extending in y directions with center at 50x, 0y with radius to x vertices of 10 and y vertices of 20. Hint: Use `locate` instruction to orient hyperbolas in x direction when desired:

```
locate(50,0,0,,,-90){hyperbola(0,0,20,10)}
```

Example above defines matching x direction hyperbola to y direction hyperbola defined above it. zmin is -10^6 and zmax is 10^6.

I.3.13 `include` or: `include_file`

Format:	`includeFilename`
Default Values:	None
Class:	Include Class
When Legal:	Base Nesting Level

Includes (inserts) geometry instructions from the referenced `.GEM` file (Filename: e.g. test.gem) at the point of the include instruction's location in the referencing geometry file. Included geometry files can reference other geometry files via includes. The geometry compiler limits include nesting to 15 levels deep to protect against accidental `include` recursion (an include file directly or indirectly calling itself).

The Filename cannot be defaulted and must be a legal geometry file name (e.g. test.gem - both long and short filenames are supported). The `.GEM` file extension will automatically be added (e.g. TEST changed to `TEST.GEM`) when needed.

A suggested use for include files involves components. You can define a component in an include `.GEM` file and then reference it from a calling `.GEM` file:

```
;lens1.gem file image
locate(,,,,-90)            ;swing lens to align with x-axis
    {                    ;centered on origin
    fill                    ;simple volume fill
        {
        within{cylinder(0,0,-2,50,,4)}    ;cylinder with r = 50 centered at ↩
             origin
        notin{circle(0,0,5)}    ;define aperture in lens of r = 5
        }
    }
;end of lens1.gem file image
```

```
;fragment of referencing geometry file
locate(10){ e(1){ include(lens1) } }    ;locate lens1 at 10x with ele  ↩
   points of 1 volt
locate(35){ e(2){ include(lens1) } }    ;locate lens1 at 35x with ele  ↩
   points of 2 volts
locate(48){ e(3){ include(lens1) } }    ;locate lens1 at 48x with ele  ↩
   points of 3 volts
```

The above example works only if lens1 is defined in some useful orientation for the calling locates to use (as in the example above). Moreover, it is often useful for the point type to be specified in the referencing `.GEM` file as opposed to the included `.GEM` file (as shown above).

I.3.14 `locate` or: `project`, `project_it`, `transform`

Format:	`locate(x,y,z,scale,az,el,rt) { }`
Default Values:	`locate(0,0,0,1,0,0,0) { }`
Class:	Location Class
When Legal:	Any Nesting Level

Locates (projects) geometry coordinates within its scope (internal geometry coordinates) into the geometry coordinates active just outside its scope (external geometry coordinates) using the defined transformation parameters:

x,y,z Amount to offset the geometry origin when translating it from ***internal*** to ***external*** coordinates.

locate(10,20,30) { }: Projects the ***internal*** geometry origin 0x, 0y, 0z to 10x, 20y, 30z in ***external*** geometry coordinates.

scale Scaling factor to use when translating ***internal*** coordinates to the equivalent ***external*** coordinates.

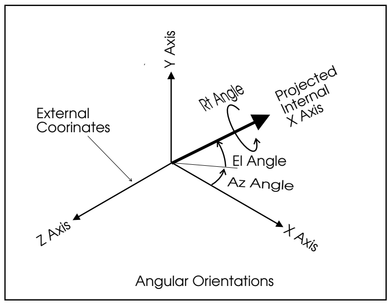


Figure I.4: Angular orientation.

locate(,,,2) { }: Scales internal coordinates by a ***factor of two*** to translate them into the external coordinates (internal objects are doubled in size as projected).

az Azimuth angle to apply when projecting the internal coordinates into external coordinates. Azimuth angle is ***degrees of ccw*** rotation about the ***y-axis*** looking down the positive ***y-axis*** toward the origin.

locate(,,,,90) { }: Azimuth angle of 90 degrees. Internal z-axis made parallel to external x-axis. Internal x-axis made parallel to external negative z-axis. Internal y-axis remains parallel to external y-axis (assuming el = rt = 0).

el Elevation angle to apply when projecting the internal coordinates into external coordinates. Elevation angle is ***degrees of ccw*** rotation about the z-axis looking down the positive z-axis toward the origin .

locate(,,,,,90) { }: Elevation angle of 90 degrees. Internal x-axis made parallel to external y-axis. Internal y-axis made parallel to external negative x-axis. Internal z-axis remains parallel to external z-axis (assuming az = rt = 0).

rt Rotation angle to apply when projecting the internal coordinates into external coordinates. Rotation angle is ***degrees of ccw*** rotation about the x-axis looking down the positive x-axis toward the origin.

locate(,,,,,,90) { }: Rotation angle of 90 degrees. Internal y-axis made parallel to external z-axis. Internal z-axis made parallel to external negative y-axis. Internal x-axis remains parallel to external x-axis (assuming az = el = 0).

Order in which Transforms are Applied (IMPORTANT): The above transformations are applied in the following order (via a 3D transfer matrix) :

1. The rotation (`rt`) transformation is applied first, creating a new interim coordinate system.
2. The elevation (`el`) transformation is applied next to the interim coordinate system creating the next interim coordinate system.
3. The azimuth (`az`) transformation is then applied to the interim coordinate system creating the next interim coordinate system.

4. The scaling (`scale`) transformation is then applied.

5. Finally the origin offset (`x`,`y`,`z`) transformation is applied.

How Locates are Actually Used by SIMION: SIMION converts each locate instruction into a 3D transfer matrix. Transfer matrices of nested locate instructions are multiplied to obtain the aggregate 3D transfer matrix to translate test instruction coordinates into potential array coordinates. The inverse of this aggregate 3D transfer matrix is used in translating potential array coordinates into the current test instruction coordinates.

Example of Nested Locate Instructions: Sometimes it is easier to visualize a transformation (not get lost) by using nested locates: ***Remember: Always apply nested locates from the innermost locate working outward toward the outermost locate!***

```
;Single locate twists rectangular solid 30  along x-axis, 45  y-axis, and ↩
   then -90   z-axis
locate(0,0,0,1,-90,45,30)    ;First: rt ccw 30 , el ccw 45 , and then az cw ↩
    90
    {
    fill{within{cent_box3d(0,0,0,70,3,30)}}
    }
;Triple locate twists rectangular solid 30  along x-axis, 45  y-axis, and ↩
   then -90   z-axis
locate(,,,,-90)           ;Third: az cw 90  along z-axis
    {
    locate(,,,,,45)        ;Second: el ccw 45  along y-axis
        {
        locate(,,,,,,30)    ;First: rt ccw 30  along x-axis
             {
            fill{within{cent_box3d(0,0,0,70,3,30)}}
             }
        }
    }
```

I.3.15 `non_electrode` or: `n`, `non_e`, `non_p`, `non_pole`, `non_electrode_points`, `non_pole_points`

Format:	`non_electrode(potential) { }`
Default Values:	`non_electrode(0) { }`
Class:	Point Definition Class
When Legal:	Base Nesting Level

Defines fill point type of non_electrode or non_pole and its associated potential to use within the instruction's scope.

```
non_electrode()
    {
    fill{...}    ;fill with non_electrode points of zero volts
    }
```

I.3.16 `notin`

Format:	`notin { }`
Default Values:	NA
Class:	Within Class
When Legal:	Fill Nesting Level

Must be called from within at the Fill Nesting Level (within a Fill Class instruction: `fi-ll`). Holds one or more Test Class instructions within its scope that define a volume (3D potential array) or area (2D potential array). ***There is no limit to the number of Test Class instructions that can appear within the scope of a Within Class Instruction.***

If a potential array point is contained within the intersection volume (or area) expanded 0.5 ***grid unit*** (as projected) of the tests contained in its scope, a TRUE is returned. Thus, all the test class instructions (e.g. `circle()`) inside the scope of a Within Class instruction (e.g. `within{}`) must return TRUE for the Within Class instruction to return TRUE to the Fill Class instruction.

All that is required for a point to be considered NOT within a fill's volume is that ***at least one*** `notin` instruction returns a TRUE.

```
fill
   {        ;within right half of ellipsoid
   within{sphere(0,0,0,50,30,50) box(0,-30,50,30)}
           ;notin inner ellipsoid circle combo
   notin{sphere(0,0,0,45,25,45) circle(0,0,35)}
   }
```

The example above creates the right half of an ellipsoid (x >= 0) centered 0x, 0y, 0z and outer shell of 50rx, 30ry, 50rz with an inner ellipsoid circle removed from it (a complex shape).

I.3.17 `notin_inside`

Format:	`notin_inside { }`
Default Values:	NA
Class:	Within Class
When Legal:	Fill Nesting Level

Must be called from within at the Fill Nesting Level (within a Fill Class instruction: `fi-ll`). Holds one or more Test Class instructions within its scope that define a volume (3D potential array) or area (2D potential array). ***There is no limit to the number of Test Class instructions that can appear within the scope of a Within Class Instruction.***

If a potential array point is contained ***within but not on or outside*** the intersection volume boundary (or area) of the tests contained in its scope, a TRUE is returned. Thus, all the test class instructions (e.g. `circle()`) inside the scope of a Within Class instruction (e.g.

`within{}`) must return TRUE for the Within Class instruction to return TRUE to the Fill Class instruction.

All that is required for a point to be considered NOT within a fill's volume is that at least one `notin_inside` instruction returns a TRUE.

```
fill
   {        ;within right half of ellipsoid
   within{sphere(0,0,0,50,30,50) box(0,-30,50,30)}
           ;notin inner ellipsoid circle combo
   notin_inside{sphere(0,0,0,45,25,45) circle(0,0,25)}
   }
```

The example above creates the right half of an ellipsoid (x >= 0) centered 0x, 0y, 0z and outer shell of 50rx, 30ry, 50rz with an inner ellipsoid circle removed from it (a complex shape). ***Note: The inner ellipsoid boundary is included but its interior is excluded by the*** **`notin_inside`** ***instruction.***

Note: The normal `notin` also includes as notin all points that are within 0.5 grid unit (as projected) of the volume boundary. This can cause problems when changing scales (e.g. hole sizes are not preserved predictably).

The `notin_inside` and the `notin_inside_or_on` were added to enhance the power of the `Notin` class of commands for use in geometry definitions where change of scale (e.g. doubling) could lead to unexpected notin size changes.

I.3.18 `notin_inside_or_on`

Format:	`notin_inside_or_on { }`
Default Values:	NA
Class:	Within Class
When Legal:	Fill Nesting Level

Must be called from within at the Fill Nesting Level (within a Fill Class instruction: `fi-ll`). Holds one or more Test Class instructions within its scope that define a volume (3D potential array) or area (2D potential array). ***There is no limit to the number of Test Class instructions that can appear within the scope of a Within Class Instruction.***

If a potential array point is contained ***within, on, but not outside*** the intersection volume boundary (or area) of the tests contained in its scope, a TRUE is returned. Thus, all the test class instructions (e.g. `circle()`) inside the scope of a Within Class instruction (e.g. `within{}`) must return TRUE for the Within Class instruction to return TRUE to the Fill Class instruction.

All that is required for a point to be considered NOT within a fill's volume is that ***at least one*** `notin_inside_or_on` instruction returns a TRUE.

```
fill
   {        ;within right half of ellipsoid
   within{sphere(0,0,0,50,30,50) box(0,-30,50,30)}
           ;notin inner ellipsoid circle combo
   notin_inside_or_on{sphere(0,0,0,45,25,45) circle(0,0,25)}
   }
```

The example above creates the right half of an ellipsoid (x >= 0) centered 0x, 0y, 0z and outer shell of 50rx, 30ry, 50rz with all points ***within or on*** an inner ellipsoid circle removed from it (a complex shape).

Note: The normal `notin` also includes as notin all points that are within 0.5 ***grid unit*** (as projected) of the volume boundary. This can cause problems when changing scales (e.g. hole sizes are not preserved predictably).

The `notin_inside` and the `notin_inside_or_on` were added to enhance the power of the Notin class of commands for use in geometry definitions where change of scale (e.g. doubling) could lead to unexpected notin size changes.

I.3.19 `pa_define`

Format:	`pa_define(nx, ny, nz, Sym, Mirror, Type, ng)`
Default Values:	`pa_define(100,20,1,Cyl,Y,Elect,100)`
Class:	PA Define Class
When Legal:	When First Instructions in File

Defines potential array to create if `NAME.GEM` called by New function. Instruction is ignored when called from within Modify. ***However, its parameters are always checked for errors.*** If it exists, it must always be the first command in the `.GEM` file (optional command).

nx The x dimension of the potential array. ***Must always be 3 or greater.***

ny The y dimension of the potential array. ***Must always be 3 or greater.***

nz The z dimension of the potential array. ***Must always be 1 or greater.*** A value of 1 defines a 2D array. Values greater than one define a 3D array.

Sym The symmetry of the array: `cylindrical` or `planar`. The compiler just checks the ***first letter***. Thus "`c`" or "`p`" are sufficient. ***Note: 3D arrays must always have planar symmetry.***

Mirror The array mirroring can be: `None`, `X`, `Y`, `Z`, `XY`, `YZ`, `XZ`, `XZY`. The compiler just scans the string for "`X`", "`Y`", and "`Z`". If none of these three characters are found a mirroring of None is assumed. Legal mirroring varies by array symmetry and if 2D or 3D:

- All 3D arrays: All mirroring options are legal
- Planar 2D arrays: All mirroring ***except z*** are legal
- Cylindrical 2D arrays: y mirroring is required, x is legal, ***z is illegal***

Type The array type: Electrostatic or Magnetic. The compiler just checks the ***first letter***. Thus "`E`" or "`M`" are sufficient.

ng The magnetic scaling parameter (discussed elsewhere in main manual). Must always be 1 or greater.

Example: `pa_define(101,51,71,p,yz,m,30)`

The array defined above has dimensions of nx = 101, ny = 51, nz = 71. It is 3D planar with mirroring in y and z. The array's type is magnetic with a ng scaling value of 30.

I.3.20 `parabola`

Format:	`parabola(xv, yv, focus_offset)`
Default Values:	`parabola(0,0,10)`
Class:	Test Class
When Legal:	Within Nesting Level

Defines 2D y direction parabola (e.g. on current xy plane: z = 0). Parameters define the vertex and focus_offset in y. Both positive and negative numbers for focus_offset are legal (0 is illegal). The 2D parabola extends plus or minus 10^6 in z. Returns TRUE if point within its bounds.

Function used: (y - yv) = $(x - xv)^2/(4 * \text{focus\_offset})$

```
parabola(15,20,30)
```

Defines parabola in positive y direction with vertex at 15x, 20y with focus_offset of 30. Hint: Use `Locate` instruction to rotate parabola to x-axis when desired. zmin is -10^6 and zmax is 10^6.

I.3.21 `points` or: `points2d`

Format:	`points(x,y, x,y, ...)`
Default Values:	None
Class:	Test Class
When Legal:	Within Nesting Level

Defines a collection of 2D (e.g. on current xy plane: z = 0) points (unconnected points). Parameters define the x,y coordinates of each point. ***Up to 100 x,y points can be defined in the parameter list.*** Each 2D point extends plus or minus 10^6 in z. Returns TRUE if point within its bounds of one of the defined points.

```
fill                           ;solid fill with current point def
    {
    locate(50,50)           ;shift center to 50x, 50y
        {
        within{points(    -15,-15    ;create three points at ends of ↩
           triangle
                        15,-15
                        0,15)}
        }
    }
```

Defines three points at the ends of triangle. This produces three lines in a 3D array. Note the use of `locate` to make the definition easier. zmin is -10^6 and zmax is 10^6.

SIMION tries to map each point into a point (2D) or line (3D) in the potential array. However, odd scaling and orientations can result in up to ***four*** PA points being changed instead of one. ***It is recommended that points be defined in potential array coordinates to avoid this issue.***

I.3.22 `points3d`

Format:	`points3d(x,y,z x,y,z ...)`
Default Values:	None
Class:	Test Class
When Legal:	Within Nesting Level

Defines a collection of 3D points (unconnected points). Parameters define the x,y,z coordinates of each point. ***Up to 65 x,y,z points can be defined in the parameter list.*** Returns TRUE if potential array point is within bounds of one of the defined points.

```
fill                            ;solid fill with current point def
    {
    locate(50,50)            ;shift center to 50x, 50y
        {
        within{points3d(    -15,-15,0    ;create three 3D points at ends of ↩
            triangle
                        15,-15,0
                        0,15,0)}
        }
    }
```

Defines three points at the ends of triangle on the z = 0 plane. Note the use of `locate` to make the definition easier.

SIMION tries to map each point into a point in the potential array. However, odd scaling and orientations can result in up to six potential array points being changed instead of one. ***It is recommended that points be defined in potential array coordinates to avoid this issue.***

I.3.23 `polyline`

Format:	`polyline(x,y, x,y, ...)`
Default Values:	None
Class:	Test Class
When Legal:	Within Nesting Level

Defines 2D (e.g. on current xy plane: z = 0) polyline (connected line segments). Parameters define the x,y endpoints of the connected line segments. ***Up to 100 x,y pairs can be defined in the parameter list.*** SIMION assumes the polyline defines a 2D closed area (automatically forcing closure with extra line segment if required). The 2D polyline closed area extends plus or minus 10^6 in z. Returns TRUE if point within its bounds.

```
fill                             ; solid fill with current point def
    {
    locate(50,50)                ; shift center to 50x, 50y
        {
        within{ellipse(0,0,40,30)}  ; ellipse 40rx, 30ry
        notin{polyline( -15,-15     ; create triangular hole
                         15,-15
                          0,15)}
        }
    }
```

Defines an ellipse with a triangular hole in it. SIMION automatically closes the triangle polyline. Note the use of `locate` to make the definition easier. This produces a tube in a 3D array. zmin is -10^6 and zmax is 10^6.

Remember this is an area (volume) fill test. You can backtrack along a polyline to attempt grids. However, odd angles and scaling may result in a pretty shabby grid definition. ***It is recommended that you not backtrack, but rather define a volume, edge fill it, and later (below) erase the unwanted edge boundary portions with non-electrode fills (see example in `rotate_edge_fill` below).***

I.3.24 `rotate_edge_fill` or: `rotate_edge_fill_volume`

Format:	`rotate_edge_fill(angle_of_revolution) { }`
Default Values:	`rotate_edge_fill(360) { }`
Class:	Fill Class
When Legal:	Base Nesting Level

Defines a ***volume of revolution*** edge fill using the currently active point type and potential. It raises the nesting level to Fill Nesting Level within its scope (e.g. { }). ***Its scope must contain at least one `Within` or `Notin` instruction.*** If no `Within` instruction is supplied, the point is assumed within, subject to rejection by `Notin` tests. ***There is no limit to the number of `Within`s or `Notin`s that can appear inside the scope of a Fill Class instruction.***

Each `Within` or `Notin` instruction defines a separate ***area of intersection*** with upper half of the xy-plane (z = 0 and y >= 0) through the use of tests (e.g. `circle()`). The coordinate system used is the coordinate system's scope that `rotate_edge_fill` instruction appears in. ***`Locate` instructions appearing within the scope of the `rotate_edge_fill` do not change this test coordinate system. They merely change where the tests may intersect its xy plane.***

This fill area is then rotated ***ccw*** `angle_of_revolution` degrees around the x-axis looking down the positive x axis toward the origin (same as rt angle in `locate`). All potential array points that fall on the ***EDGE*** of this volume of revolution will be changed to the currently active type and potential.

The .GEM file fragment below uses a `rotate_edge_fill` to create a parabolic grid. ***It is important that you take the time to understand this instruction fragment if you are to successfully use*** **`rotate_edge_fill`** ***properly.***

```
;fragment below makes a 180 degree parabolic mirror grid surface

pa_define(101,101,101,p,n)    ;101, 101, 101 3D planar non-mirror

locate(10,50,50)        ;locate point of rotation for grid center
    {
    e(1)                ;use one volt electrode points
        {
        rotate_edge_fill(180)    ;180 degree rotate fill
            {        ;parabolic mirror in x-axis direction
            within{locate(,,,,,-90){parabola(0,0,10)} box(0,0,50,100)}
            }
        }
    n(0)                ;zero volt non-electrode to erase
        {
        fill            ;erase edge electrodes cut planes
            {        ;erase edge electrodes in z = 0 plane
            within{locate(,,,,,-90){parabola(0,0,10)} box3d ↩
                (-0.5,-1000,0.5,1000,1000,0)}
                    ;erase edge electrodes in x = 50 plane
            within{box3d(49.5,-1000,-1000,50.5,1000,1000)}
            }
        }
    }
```

Creates a 180 degree parabolic grid in the x-axis direction in a 3D array. Note: The use of a `locate` instruction to rotate the parabola into an x-axis parabola. The second fill (volume fill - fill) is used to erase edge electrode points at the z = 0 and x = 50 cut planes. This results in a half a parabolic grid with no grids in the cut planes.

I.3.25 `rotate_fill` or: `rotate_fill_volume`

Format:	`rotate_fill(angle_of_revolution) { }`
Default Values:	`rotate_fill(360) { }`
Class:	Fill Class
When Legal:	Base Nesting Level

Defines a ***volume of revolution*** fill using the currently active point type and potential. It raises the nesting level to Fill Nesting Level within its scope (e.g. { }). ***Its scope must contain at least one*** **`Within`** ***or*** **`Notin`** ***instruction.*** If no `Within` instruction is supplied, the point is assumed within, subject to rejection by `Notin` tests. ***There is no limit to the number of*** **`Withins`** ***or*** **`Notins`** ***that can appear inside the scope of a Fill Class instruction.***

Each `Within` or `Notin` instruction defines a separate ***area of intersection*** with the upper half of the xy-plane (z = 0 and y > 0) through the use of tests (e.g. `circle()`). The coordinate system used is the coordinate system's scope that the `rotate_fill` instruction appears in. ***`Locate` instructions appearing within the scope of the `rotate_fill` do not change this test coordinate system. They merely change where the tests may intersect its xy plane.***

This fill area is then rotated ccw angle_of_revolution degrees around the ***x-axis*** looking down the positive x-axis toward the origin (same as rt angle in `locate`). All potential array points that fall within this volume of revolution will be changed to the currently active type and potential.

The `.GEM` file fragment below uses two `rotate_fills` to create the internals for a spherical ESA. ***It is important that you take the time to understand this instruction fragment if you are to successfully use `rotate_fill` properly.***

```
;fragment below makes inner workings of spherical ESA

pa_define(101,101,101,p,n)    ;101, 101, 101 3D planar non-mirror

locate(50,10,10)        ;locate point of rotation for ESA
    {
    e(1)                ;use one volt electrode points
        {
        rotate_fill(90)    ;90 degree rotate fill
            {        ;inner spherical surface
            within{circle(0,0,40) centered_box(0,0,70,200)}
            }
        }
    e(2)                ;use two volt electrode points
        {
        rotate_fill(90)    ;90 degree rotate fill
            {        ;outer spherical surface
            within{centered_box(0,0,70,140)}    ;90 degree solid cylinder
            notin{circle(0,0,60)}    ;with spherical inner surface
            }
        }

    }
```

The above example creates a 90 degree spherical ESA in a 3D array. The inner electrode is one volt with a spherical radius of 40 and a width of 70. The outer electrode is two volts with a spherical radius of 60 and a width of 70. This might serve as starting point for a real .PA# definition file for a 90 degree spherical ESA.

I.3.26 `sphere` or: `ellipsoid`

Format:	`sphere(xc, yc, zc, rx, ry, rz)`
Default Values:	`sphere(0,0,0,10,10,10)`
Class:	Test Class
When Legal:	Within Nesting Level

Defines 3D sphere or ellipsoid. Parameters define the center and radii. ***Note: If ry is defaulted, ry will be set to rx. Likewise, if rz is defaulted, rz will be set to ry.*** Returns TRUE if point is within its bounds.

```
sphere(15,20,30,45,20,45)
```

Defines and ellipsoid centered at 15x, 20y, 30z with radii of 45rx, 20ry, 45rz.

I.3.27 `within`

Format:	`within { }`
Default Values:	NA
Class:	Within Class
When Legal:	Fill Nesting Level

Must be called from inside the Fill Nesting Level (within a Fill Class instruction: `fil-l`). Holds one or more Test Class instructions within its scope that define a volume (3D potential array) or area (2D potential array). ***There is no limit to the number of Test Class instructions that can appear within the scope of a Within Class instruction.***

If a potential array point is contained within the intersection volume (or area) expanded 0.5 grid unit (as projected) of the tests in its scope a TRUE is returned. Thus, all the test class instructions (e.g. `circle()`) inside the scope of a Within Class instruction (e.g. `within{}`) must return TRUE for the Within Class instruction to return TRUE to the Fill Class instruction.

All that is required for a point to be considered within a fill's volume is that ***at least one*** ***`Within`*** instruction returns a TRUE. This within designation will be revoked if ***at least one*** ***`Notin`*** instruction returns a TRUE.

```
fill
   {        ;within right half of ellipsoid
   within{sphere(0,0,0,50,30,50) box(0,-30,50,30)}
   notin{sphere(0,0,0,45,25,45)}    ;notin inner ellipsoid
   }
```

The example above creates the right half of a ellipsoid shell (x >= 0) centered at 0x, 0y, 0z and an outer shell of 50rx, 30ry, 50rz with an inner shell of 45rx, 25ry, 45rz.

I.3.28 `within_inside`

Format:	`within_inside { }`
Default Values:	NA
Class:	Within Class
When Legal:	Fill Nesting Level

Must be called from within at the Fill Nesting Level (within a Fill Class instruction: `fi-ll`). Holds one or more Test Class instructions within its scope that define a volume (3D potential array) or area (2D potential array). ***There is no limit to the number of Test Class instructions that can appear within the scope of a Within Class Instruction.***

If a potential array point is contained ***within but not on or outside*** the intersection volume boundary (or area) of the tests contained in its scope, a TRUE is returned. Thus, all the test class instructions (e.g. `circle()`) inside the scope of a Within Class instruction (e.g. `within{}`) must return TRUE for the Within Class instruction to return TRUE to the Fill Class instruction.

Note: All that is required for a point to be considered NOT within a fill's volume is that ***at least one `Notin`*** class instruction returns a TRUE.

```
fill
    {       ;within right half of ellipsoid
    within_inside{sphere(0,0,0,50,30,50) box(0,-30,50,30)}
    }
```

The example above creates the right half of an ellipsoid (x >= 0) centered at 0x, 0y, 0z and an outer shell of 50rx, 30ry, 50rz. ***Note: Points actually on the ellipsoid boundary are NOT included.*** Only points that are physically ***inside*** the boundary (but not exactly on it) will be considered within.

Note: The normal `within` also includes as within all points that are within 0.5 grid unit (as projected) of the volume boundary. This can cause problems when changing scales with holes created in electrodes (e.g. surfaces of revolution - `rotate_fill()`) using non-electrode points and withins (e.g. hole sizes are not preserved predictably).

The `within_inside` and the `within_inside_or_on` were added to enhance the power of the `within` class of commands for use in geometry definitions where change of scale (e.g. doubling) could lead to unexpected within size changes.

I.3.29 `within_inside_or_on`

Format:	`within_inside_or_on { }`
Default Values:	NA
Class:	Within Class
When Legal:	Fill Nesting Level

Must be called from within at the Fill Nesting Level (within a Fill Class instruction: `fill`). Holds one or more Test Class instructions within its scope that define a volume (3D potential array) or area (2D potential array). ***There is no limit to the number of Test Class instructions that can appear within the scope of a Within Class Instruction.***

If a potential array point is contained ***within, on, but not outside*** the intersection volume boundary (or area) of the tests contained in its scope, a TRUE is returned. Thus, all the test class instructions (e.g. `circle()`) inside the scope of a Within Class instruction (e.g. `within{}`) must return TRUE for the Within Class instruction to return TRUE to the Fill Class instruction.

Note: All that is required for a point to be considered NOT within a fill's volume is that at least one `Notin` class instruction returns a TRUE.

```
fill
    {        ;within right half of ellipsoid
    within_inside_or_on{sphere(0,0,0,50,30,50) box(0,-30,50,30)}
    }
```

The example above creates the right half of an ellipsoid (x >= 0) centered at 0x, 0y, 0z. Note: No points that are even slightly outside the outer boundary are considered within.

Note: The normal `within` also includes as within all points that are within 0.5 grid unit (as projected) of the volume boundary. This can cause problems when changing scales (e.g. hole sizes are not preserved predictably).

The `within_inside` and the `within_inside_or_on` were added to enhance the power of the Within class of commands for use in geometry definitions where change of scale (e.g. doubling) could lead to unexpected within size changes.

I.4 Developing, Testing, and Using Geometry Files

SIMION provides geometry file development tools within the Modify function (Figure I.5). Geometry files can be utilized by either New or Modify to define electrode geometry.

I.4.1 Geometry Examples Provided With SIMION

There is a collection of geometry file examples in the `examples\geometry` subdirectory (in the SIMION program folder). This directory contains a collection of trivial and non-trivial examples. Examples include: ESAs, quads, traps, lenses, and ICR cells. The New function can be used directly on most of these `.GEM` files to quickly see what they create. It is intended that they should serve to get you started on your geometry file adventures. ***Be sure to scan the `README.html` file in the subdirectory for any directions and/or cautions.***

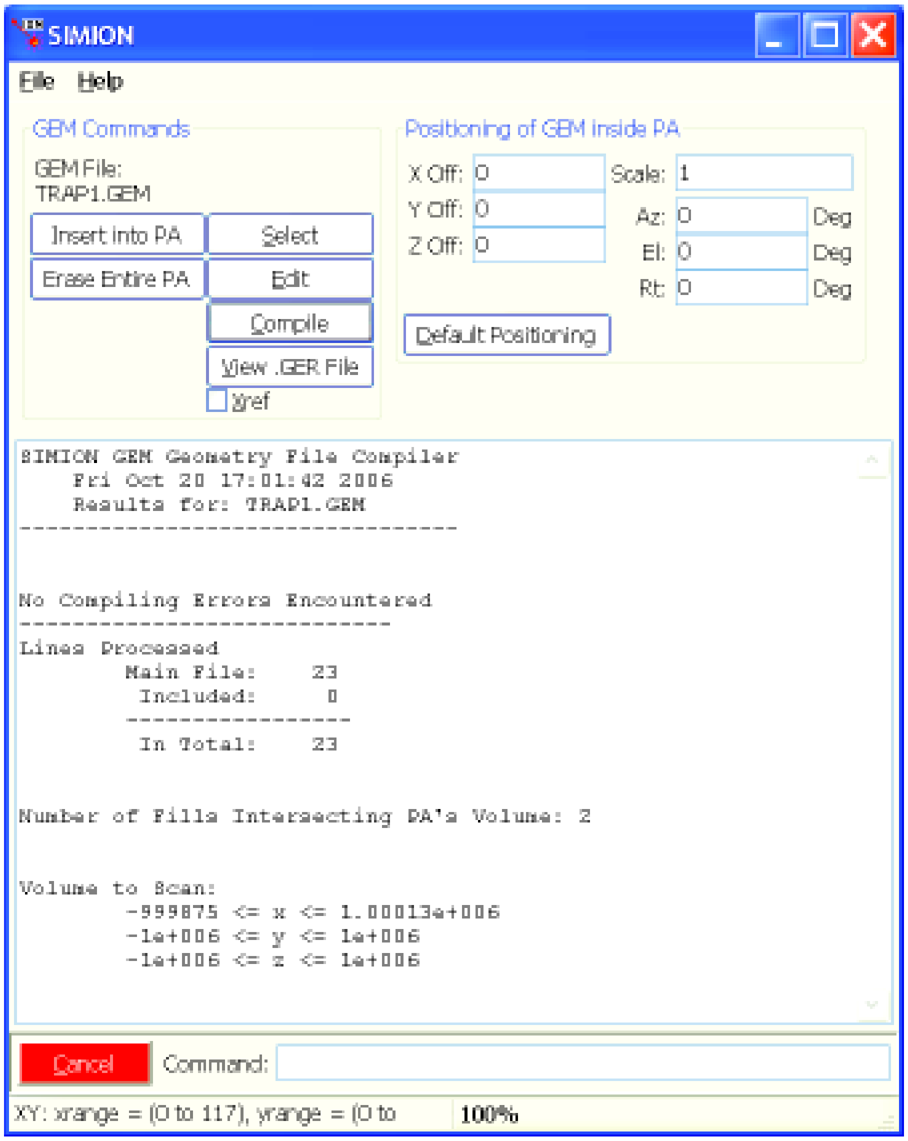
I.4.2 Accessing Geometry Files Via the New Function:

When you click on the New button (on the Main Menu Screen) or try to Modify an empty potential array (automatically invoking the New function), the potential array definition screen has a Use Geometry File button. If you click this button, the file selection dialog box will ask you to select a geometry file to be automatically inserted in the new potential array.

If you have a `pa_define` instruction as the first instruction in the selected `.GEM` file, SIMION will use it (in lieu of the New definition screen values) to create the desired potential array.

Note: When inserting geometry files from the New Function, the initial transformation will always be defaulted to no transformation (identity transformation). ***Thus your `.GEM` file must contain all the transformations required to place the geometry properly in the target potential array.***

If you have a geometry compiler error, you must use the geometry file development tools in Modify (via the GeomF button) to find and fix the problem(s).



```
SIMION GEM Geometry File Compiler
    Fri Oct 20 17:01:42 2006
    Results for: TRAP1.GEM
---------------------------------


No Compiling Errors Encountered
----------------------------
Lines Processed
        Main File:     23
         Included:      0
        -----------------
         In Total:     23


Number of Fills Intersecting PA's Volume: 2


Volume to Scan:
        -999875 <= x <= 1.00013e+006
        -1e+006 <= y <= 1e+006
        -1e+006 <= z <= 1e+006
```

Figure I.5: SIMION's Geometry Development Screen.

I.4.3 Accessing the Modify Geometry Development Tools

The geometry file development tools are accessed in Modify via the GeomF button (on lower left edge of Modify screen). If there is no currently active `.GEM` file, SIMION will ask you to select one via an automatic call to the file selection dialog box. Assuming you're trying this for the first time, switch to the `examples\geometry` directory and select `trap1.gem`. The Geometry Development screen above (Figure I.5) will appear:

An Introduction to the GEM Development Tools: Figure I.5 above shows the Geometry Development Screen. It is used to edit, test compile and insert geometry definitions into the currently active potential array. Geometry development tools include a geometry language compiler, Status Screen, `.GEM` file access buttons, positioning controls, and geometry insertion and erasure controls.

Running Another Editor From SIMION: The geometry development tools access the Notepad editor by default for geometry file inspection and modification. If you prefer another text editor, see Appendix G *(Text Editing)* on how to link SIMION to your editor

The Geometry File Development Cycle: The geometry file development cycle involves the following steps:

1. Create a new geometry file.
2. Test compile and edit until there are no compiler errors.
3. Insert the geometry into the potential array and examine it.
4. If there are geometry errors, edit geometry file, erase potential array, and repeat from step 2.

Creating a New Geometry File: You may now use the editor (Notepad by default) to type in your geometry file. Be sure to use indention to define your nesting levels (as in the examples above). Indention of nested instructions makes them much easier to read and understand later. Be sure to use comments to improve geometry instruction readability. Name and save your file (with an extension `.GEM`, e.g. `TEST.GEM`).

Now, click the New button in SIMION to create a blank potential array to use with the geometry definitions (you may want to Remove all PAs from RAM first to reduce the clutter).

If there are errors in the GEM file, you can enter the Modify screen and click GeomF and then cilck Compile. If the compiler finds an error it will beep and display a message to tell you what it didn't like. Errors are also written to the `.GER` file (e.g. `TEST.GER` is error file for `TEST.GEM`). Click the View `.GER` File button to access the `.GER` file.

There are times when a more complete listing may help you to understand what is going on with the compiler. This is done by selecting the Xref check box before clicking the Compile button. The compiler then generates a complete cross-reference listing to the Status Screen and `.GER` file (even if there are no errors). The cross-reference listing is useful in understanding how the compiler interpreted the various geometry instructions.

Editing the Current .GEM file: Normally the error message on the Status Screen is enough to point out your error. To edit the current `.GEM` file, you may click the Edit button. The editor will be called (Notepad by default), allowing you to the fix the error, re-save the file, quit the editor, and recompile.

Erasing the Potential Array: The Erase Entire PA button is provided to allow you to remove all electrode/pole definitions from the potential array before inserting geometry definitions into it. ***When an array is erased, all points are converted to non-electrode (pole) points of zero volts (Mags).*** Use this button between successive geometry insertion attempts to remove the clutter of the past.

Warning
This erases everything in the potential array - not just what was inserted by the geometry file.

Inserting Geometry Definitions Into a PA: The Insert into PA button is used to actually insert geometry definitions into the current potential array. When this button is clicked the geometry compiler compiles the selected `.GEM` file (and all its include files). If there are no errors the geometry definitions are then added to the potential array. Note: Your potential array is not erased first (that is your responsibility). ***Geometry definitions are added to any that may already be in the potential array.*** A progress bar at the bottom of the screen is provided to assure you SIMION has not died.

Using the Initial Transformation: While your `.GEM` file will probably contain one or more `Locate` instructions, you may still want final control over just where these geometry definitions actually are placed in your potential array. SIMION provides panel objects that can be used to define this initial transformation. You may think of the initial transformation as the outermost `Locate` instruction (the entire `.GEM` file is within its scope). Normally (by default or if the Default Position button is clicked) this transformation does nothing (identity transformation).

As an example, let's say that we want to double the potential array twice and then insert the geometry into this expanded array. We would Double the array twice, click the GeomF button, erase the potential array, set the scale to 4.0 (to compensate for array doubling twice), and insert the geometry into the potential array. The main advantage of doing things this way (instead of simply doubling the array) is that the resulting geometry is smoother (none of the Double caused jags).