

# defineback

November 24, 2009  
13:48

## Contents

<b>1</b>	<b>A Program for Defining 2-D Plasmas</b>	<b>1</b>
<b>2</b>	<b>Example Input File</b>	<b>2</b>
<b>3</b>	<b>Plasma Data</b>	<b>3</b>
<b>4</b>	<b>Source Group Data</b>	<b>7</b>
<b>5</b>	<b>Iterative Problems</b>	<b>12</b>
<b>6</b>	<b>The Main Program</b>	<b>13</b>
<b>7</b>	<b>Main Subroutine</b>	<b>14</b>
<b>8</b>	<b>Read a row-oriented source file</b>	<b>23</b>
<b>9</b>	<b>Setup a source group</b>	<b>25</b>
<b>10</b>	<b>Parse tabular plasma data header</b>	<b>29</b>
<b>11</b>	<b>Read row-oriented plasma data</b>	<b>32</b>
<b>12</b>	<b>Parse a single label for row-oriented data</b>	<b>37</b>
<b>13</b>	<b>INDEX</b>	<b>39</b>

## 1 A Program for Defining 2-D Plasmas

This program is designed to be used in conjunction with `definegeometry2d`. Like `definegeometry2d`, the bulk of the data details are provided in one or more text files that can be generated by some other code. The data formats used here are sufficiently simple that they can also be produced manually. The specification of the “background” consists of two pieces:

1. The densities, velocities, and temperatures of the background (plasma) species. For simplicity, we will refer to these as the “plasma” species, even though in some problems, these species may be electrically neutral (e.g., for the purposes of studying neutral-neutral interactions).
2. A description of the neutral sources (e.g., recycling of a plasma flow onto a material surface).

This initial version of `defineback` is still “rough around the edges”. While many opportunities for improvement are apparent, the most promising prospect is to incorporate this functionality into a graphical tool that also carries out the geometry definition. What comes next will be determined by project needs and manpower availability; stay tuned.

As in `definegeometry2d`, the input file for `defineback` is not listed in `degas2.in`. Again, the reason is that this code is, in principle, just one of many different ways to provide background data to `DEGAS 2`.

The command line for the execution of `defineback` specifies the name of the main `defineback` input file. This file will contain pointers to other files that provide the plasma and source information.

`defineback 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). This is true not only for the main input file, `an_input_file`, but also for the text files to which it refers.

## 2 Example Input File

Here is an example input file:

```
plasma_file mrx_plasma3
```

```
new_source_group
  source_type puff
  source_geom surface
  source_species D2
  source_root_sp D2
  source_nflights 1000
  specify_flux
  source_file mrx_source_2.in
end_source_group
```

```
new_source_group
  source_type puff
  source_geom surface
  source_species D2
  source_root_sp D2
  source_nflights 1000
  specify_flux
  source_stratum      1
  source_segment      *
  source_strength     1.e20
end_source_group
```

### 3 Plasma Data

This program contains by default a rather general mechanism for specifying the plasma data via a text file. The data can either be formatted in labeled columns or in more freely formatted rows. The ordering of the data in either case can match that of the zones or can be specified by a list of zone numbers.

There is essentially only one keyword associated with specifying the plasma data, `plasma_file`. The simplest form of this keyword, shown above, has only one argument, the name of the file containing the plasma data to be read in. In this case, the data in the file are assumed to be in the “tabular” format described below.

To use the alternative “row” formatting option, a second argument `row` must appear after the file name. E.g., for the example above,

```
plasma_file mrx_plasma3 row
```

An additional argument controls the specification of the plasma data in the third dimension of nearly symmetric 3-D cases (in which the *geometry\_symmetry* variable of `geomint.hweb` is *geometry\_symmetry\_plane\_hw*, *geometry\_symmetry\_cyl\_hw*, or *geometry\_symmetry\_cyl\_section*). The default assumption (i.e., that obtained without any additional argument to the `plasma_file` keyword) is that the plasma data are to be made symmetric in the third dimension. In these cases, the code expects the zone numbers provided to be those of the “reference zones” for which  $zn\_index(zone, zi\_ptr) = zone$ . As the plasma parameters are read in for these zones, the code will loop over the other zones pointing to them and assign them those same plasma parameters. Unless the user has set up the geometry so that these reference zones are numbered consecutively and beginning at 1 (`definegeometry2d` and `boxgen` currently do not do this when creating nearly symmetric 3-D geometries), the zone numbers should be explicitly listed.

If the plasma data are to vary in the third dimension in nearly symmetric 3-D cases, or if the user wishes to explicitly specify all plasma data, an added argument of `3D` can be added to the `plasma_file` keyword. E.g.,

```
plasma_file mrx_plasma3 row 3D
```

or, if the tabular format is to be used,

```
plasma_file mrx_plasma3 3D
```

The default behavior can also be obtained by setting this argument to `2D`.

There is also a keyword *plasma\_coords* allowing the user to specify the coordinate system of the plasma velocities as either *cartesian* or *cylindrical*. E.g.,

```
plasma_coords cylindrical
```

In two-dimensional problems, this choice is governed by the symmetry specified in `definegeometry2d`. As such, the user can ignore this keyword. It is provided only for completeness.

**Tabular Format** The key assumption for the tabular format is that all data pertaining to a particular zone are contained on a single line of the file. If the zone number is provided as the first number of a line, it will be used in assigning the data values on that line; in principle, any ordering of the zones could be used. Note that this zone number is the “absolute” zone, not the plasma zone number. If the zone number is not specified, the line number is used as the zone number. Problems having both the geometry and plasma data specified by some other code make this approach particularly simple.

A header line at the top of the file informs the code of the ordering of the data on each line. For example,

zone	T(1)	N(1)	T(2)	N(2)
1	1.0000000475e-03	1.0000000000e+01	1.0000000475e-03	1.0000000000e+01
2	3.0000000000e+00	3.0000000000e+19	3.0000000000e+00	3.0000000000e+19
3	1.0000000475e-03	1.0000000000e+01	1.0000000475e-03	1.0000000000e+01

Here, the first column contains the zone number. The second and third columns give the temperature and density of the first background species (referring to their ordering in the input file provided for `problemsetup`). The last two columns have the temperature and density for the second background species. Note that the assumed units are eV for temperatures and  $\text{m}^{-3}$  for densities.

Velocities (in meters per second) would be indicated by a string like `V2(1)` to indicate the velocity in the  $x_2$  (e.g.,  $y$ ) direction for the first background species. The columns may be in any order. The strings can be either lower or upper case (e.g., `ZONE` or `n(1)` would also be recognized). If particular data are not provided (such as the velocities in the above example), default values of 0 will be used.

Cases in which the plasma mesh has been created with the `sonnet_mesh` or `uedge_mesh` commands in `definegeometry2d` provide the best opportunity for simplifying the ordering of the plasma data in `defineback`. At present, the zones representing such a plasma mesh are defined in nested do loops according to:

```
zone=0
do ix=1,nx
  do iz=1,nz
    zone=zone+1
  .
  .
  .
```

where  $ix$  represents the first of the two indices used by `UEDGE` or `Sonnet` and  $iz$  represents the second. So, if the data are written out by looping over the  $iz$  index first, the zone number would not need to be specified.

If some other ordering were easier to generate, a separate column of zone numbers would be needed, computed according to  $zone = iz + (ix - 1)nz$ .

**Row Format** The tabular format can be easier to read, but more difficult to generate and manipulate. For that reason, a more loosely structured, row-oriented format is provided. Here's an example of such a file:

```
#
  N(1)
#
  2.17708e+18  2.34353e+18  2.67808e+18  3.02909e+18  3.40510e+18
3.81293e+18
  4.25755e+18  4.74202e+18  5.27144e+18  5.84497e+18  1.20000e+20
1.20000e+20
  1.20000e+20  1.20000e+20  1.20000e+20  1.20000e+20  1.20000e+20
1.20000e+20
  1.20000e+20  1.20000e+20  1.20000e+20  1.20000e+20  1.20000e+20
1.20000e+20
  1.20000e+20  1.20000e+20  1.20000e+20  1.20000e+20  1.20000e+20
1.20000e+20
#
  T(2) J
#
  1.77525e-19  1.77495e-19  1.77435e-19  1.77372e-19  1.77304e-19
1.77230e-19
  1.77150e-19  1.77062e-19  1.76967e-19  1.76863e-19  4.00000e-17
4.00000e-17
  4.00000e-17  4.00000e-17  4.00000e-17  4.00000e-17  4.00000e-17
4.00000e-17
  4.00000e-17  4.00000e-17  4.00000e-17  4.00000e-17  4.00000e-17
4.00000e-17
  4.00000e-17  4.00000e-17  4.00000e-17  4.00000e-17  4.00000e-17
4.00000e-17
```

These data are read in row-by-row. As in the tabular case, the default is that the data are listed according to zone number. If some other order is required, a list of zones can be provided *as the first item in the file*. The labels are parsed in the same way as those for the tabular format. Note that the labels actually serve as the delimiters for the data. That is, the reading routine will just keep reading row after row until it reaches another data label, so the data do not have to be formatted in any particular way (you can even have one data value per line - i.e., a single column of data). The only constraint is that there be no more than 136 characters on a line.

Again, the default unit for the temperature data is eV. The temperature can also be specified in joules. A units string indicating as much should be added to the label, as shown above. To avoid confusion, the code will also recognize "eV" as a units label for the temperature. *Note that this facility is currently not available for tabular data; those temperatures must all be in eV.*

**Non-default Plasma Data** The *get\_n\_t* subroutine that reads the plasma file has a simple, but flexible interface consisting of the string that follows the *plasma\_file* keyword in the **defineback** input file. As such, the user can easily replace this subroutine with a more sophisticated piece of code that can adapt available data to the geometry of a particular case. For example, some or even all of the plasma zones may be generated by **definegeometry2d**. If the user has some notion of the spatial variation of the plasma data, a specific version of the *get\_n\_t* subroutine can be written to map that description on to the DEGAS 2 geometry using the internal geometry arrays that are available via the common blocks.

A more general implementation might pack several arguments into the string following the *plasma\_file* keyword. For example, the string might contain multiple file names, as well as integer and floating point parameters. The string would be parsed in whatever manner is desired (although use of DEGAS 2's string manipulation routines is recommended).

An example of such a subroutine, used in work on the National Compact Stellarator Experiment (NCSX), is included here as an example. The version of the *get\_n\_t* subroutine to be used is controlled by file names. The default routine is contained in the file *def2dplasma.web*. A replacement routine will be loaded automatically by the **Makefile** if it is contained in a file named *usr2dplasma.web*. The NCSX example file is called *ncsxplasma.web*. In practice, one would use such a file by executing commands analogous to:

```
cd $HOME/degas2/src
cp ncsxplasma.web usr2dplasma.web
touch usr2dplasma.web
cd ../SUN
gmake defineback
```

Likewise, to revert to the default subroutine,

```
cd $HOME/degas2/src
rm usr2dplasma.web
touch def2dplasma.web
cd ../SUN
gmake defineback
```

Note the use of the **touch** command to make sure **gmake** recognizes the change.

The NCSX example shows how the stratum label available in **definegeometry2d** can be used to locate the zones generated by the triangulation of a particular polygon specified in a **definegeometry2d** input file. Namely, the polygons in the NCSX problem were drawn so that plasma parameters would be constant over them (they corresponded to magnetic flux surfaces); each of the polygons was assigned a unique stratum number. That information is available in the polygon netCDF file generated by **definegeometry2d** (the *g2\_polygon\_stratum* array); the zone numbers associated with each polygon are also there (in *g2\_polygon\_zone*). The two arrays are combined in this *get\_n\_t* subroutine to form a mapping array, *zone\_stratum*, that provides the stratum number associated with each zone. This then allows the subroutine to determine the plasma parameters to be assigned to those zones.

## 4 Source Group Data

The rest of the input file describes the sources to be used in the run. As the above example input file shows, each source group is specified in its own section. The specification is begun with a *new\_source\_group* command and ended with *end\_source\_group*. The order of the keywords in between *should not* matter.

For very simple sources (e.g., consisting of a two or three source segments), the requisite data can all be specified explicitly in the **defineback** input file. In other cases, only the “global” parameters for the source group appear in the **defineback** input file, and the segment-dependent data (*source\_stratum*, *source\_segment*, and *source\_strength*; in nearly symmetric 3-D cases, also need *source\_segment\_iy*) are placed in a separate file pointed to by the *source\_file* keyword. This option also permits the specification of local plasma data that can be used in the case of a plasma recycling flux to compute the sheath potential or even the magnitude of the flux. As in the plasma data files, both tabular and row formats are permitted for the source files.

Each of the keywords (in *italics* occupies a line 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.

*source\_type* **typ** gives the type of the source. The only types currently treated in DEGAS 2 are **typ = plate**, **puff**, and **recomb**. The last is not needed in a **defineback** input file since a recombination source is established automatically by the inclusion of the appropriate reaction in the **problemsetup** input file.

*source\_geom* **geom** specifies the geometry of the source. The only valid value in **defineback** at this time is **geom = surface** (e.g., a line in a poloidal plane of an axisymmetric torus). However, there is no default set, so this must be included with each source group. Other options will be implemented as needed.

*source\_species* **sym** determines the species to be generated by the source. This species must be in the list of test species for the problem. The **sym** is the species symbol that appears in the *species\_infile*.

*source\_root\_sp* **sym** determines the species underlying the source. For a *plate* source, this would be the ion corresponding to the neutral *source\_species*. For a *puff* source, this would be the same as *source\_species*. The code currently requires that *source\_root\_sp* be specified. Eventually, code will be added to infer a reasonable default value from the other information provided.

*source\_nflights*  $n_{\text{flights}}$  overrides a default value of 100 flights per source group. This keyword allows the user to avoid having to edit the background netCDF file to set the number of flights.

*source\_puff\_temp*  $T$  specifies the temperature  $T$  of a puff type source in Kelvin. If not set, a default value of 300 K will be used.

*specify\_flux* informs the code that the *source\_strength* will represent a flux of particles (particles per square meter per second). This is the default.

*specify\_current* informs the code that the *source\_strength* will represent a current of particles (particles per second).

*source\_stratum*  $i_1 i_2 i_3 \dots$  is used together with the *source\_segment* and *source\_strength* keywords to explicitly specify the location and strength of a number (preferably a small number) of source surfaces. These three keywords must all have the same number of arguments. The keywords and their arguments have no preferred order. The value of *source\_stratum* corresponds to the *strata* associated with a sector, e.g., as assigned with the *stratum* keyword in **definegeometry2d**.

*source\_segment*  $i_1 i_2 i_3 \dots$  identifies which segments of the stratum specified by *source\_stratum* are to be used. If the *stratum* values have been chosen judiciously in running *definegeometry2d*, the user should be able to infer the desired segment values by counting off the segments of the corresponding polygon in the *definegeometry2d* input file (with the first side of the polygon being segment 0, the second 1, etc.). If all of the segments of a stratum are to be used, with the same source strength, a single wild-card argument of **\*** can be given with *source\_segment*; *source\_stratum* and *source\_strength* should have only one argument in this case.

*source\_segment.iy*  $i_1 i_2 i_3 \dots$  identifies the segments' third dimension label in nearly symmetric 3-D cases (in which the *geometry\_symmetry* variable of **geomint.hweb** is *geometry\_symmetry\_plane.hw*, *geometry\_symmetry\_cyl.hw*, or *geometry\_symmetry\_cyl\_section*). These labels correspond to the integer indices associated with the third dimension discretization set up in **definegeometry2d**. There, the user defines  $n_y$  surfaces in the third dimension (not counting those that define the universal cell). The resulting zones have  $i_y$  labels ranging from 0 to  $n_y - 2$ .

By default, all (i.e.,  $n_y - 1$ ) of the sectors having the specified *source\_stratum* and *source\_segment* are included. If this is the user's intent, the *source\_segment.iy* keyword should not appear; a wildcard argument (e.g., **\***) cannot be used. The specified source current will be divided equally amongst these segments.

If the  $i_y$  values are specified, the user is effectively describing a source segment by three labels: the stratum, the segment number, and the  $i_y$ . Each such triplet will be assigned the corresponding source current (via the *source\_strength* keyword).

*source\_strength*  $S_1 S_2 S_3 \dots$  gives the source strength (with units determined by the appearance of either the *specify\_flux* or *specify\_current* keywords) to be used in conjunction with *source\_stratum* and *source\_segment*.

*source\_file* **filename** is an alternative method of specifying the location and strength of the source (i.e., in lieu of using *source\_stratum*, *source\_segment*, and *source\_strength*). The file **filename** is a text file of the usual type in which blank lines and comment lines beginning with **#** are ignored. The tabular and row-oriented formats available for these files are described below.

**Tabular Format** The tabular format is somewhat restrictive. Unlike the analogous option for the plasma data, no header line is needed. Each line specifies a source segment:

$$i_{\text{stratum}} \ i_{\text{segment}} \ S_{\text{strength}}$$

That is, the stratum and segment numbers followed by its strength. This is the only permissible order for the three parameters. No wild cards are allowed. The units of the source strength are determined by which of *specify\_flux* and *specify\_current* is in effect. The third dimension  $i_y$  label has not been implemented for nearly symmetric 3-D cases. All sectors having the specified  $i_{\text{stratum}}$ ,  $i_{\text{segment}}$  value will be included, and the source current will be divided evenly amongst them.

When the source group is processed (following the *end\_source\_group* command), the code will print out the endpoint coordinates (only  $x$  and  $z$ ) of the sector associated with each source segment to provide the user with feedback on their specification.

**Row Format** The row format appears to be the more attractive option for the user because of its greater flexibility. For this reason, efforts to extend this code’s functionality have focussed on it. Roughly speaking, the format is the same as that of a row-oriented plasma data file; only the labels used will differ.

The simplest form for a source file would have sections for the stratum number (using the label `stratum`; the code will also recognize `STRATUM` and `Stratum`), the stratum segment number (`segment`; the alternative capitalizations will also work), the segment  $i_y$  label (`segment_iy`, etc.; for nearly symmetric 3-D cases; see the above description of the `source_segment_iy` keyword for additional details) and source strength (`f` or `F`; “f” stands for flux, although the meaning of the data provided is determined by which of `specify_flux` and `specify_current` is in effect).

Fluid plasma codes usually maintain plasma data in small zones directly adjacent to the target for the purpose of setting boundary conditions. The zones are too small to merit direct inclusion in the neutral transport simulation. However, the data contained in them can be used for a recycling source to compute the local sheath potential or even the plasma flux.

**Sheath Model** The basic default parameters for the sheath model used when simulating a plate source are set when the sectors associated with a source are defined, e.g., during `definegeometry2d`: `plasma_e_ion_delta` = 0 and `plasma_e_ion_mult` = 1. Since this step precedes the plasma specification, it necessarily is ignorant of the plasma parameters in the vicinity of that sector.

When `defineback` sets up the segments of a plate source it will attempt to rectify this situation. If specific plasma data for the sector are not specified (as described below), provided the electron temperature in the plasma zone adjacent to the source sector is positive, it will set `plasma_e_ion_delta` =  $3T_e$  (explicitly assuming that electrons are the first background species), leaving `plasma_e_ion_mult` set to 1. In this case, the  $3T_e$  represents a simple value for the sheath potential.

If specific plasma parameters are available for the source sector, such as those computed by a fluid plasma code for the purposes of setting boundary conditions, they can be included in a row-oriented source file. The same types of labels used for the plasma density, temperature, and velocity in a plasma data file will be recognized. The difference, of course, is that the values will correspond to the source segments, rather than to zones.

The sheath model currently in use needs only the electron temperature  $T_e$ , the temperature of the source ion  $T_i$ , and the source ion species’ flow velocity vector  $\vec{v}_i$ . The result is a value for the incident ion energy  $E_i$

$$E_i = 3T_i + \left(\frac{1}{2} + \phi\right) T_e, \quad (1)$$

$$\phi = -\ln\left(\frac{\sqrt{2\pi} |\vec{v}_i|}{\sqrt{T_e/m_e}}\right) \quad (2)$$

Since the model depends on only the magnitude of the flow velocity, `defineback` will also proceed to evaluate these expressions if just the parallel velocity (labeled by `v_par`) is provided. The result is placed into the appropriate element of `plasma_e_ion_delta` and the corresponding entry in `plasma_e_ion_mult` is set to zero.

**Plasma Flux Computation** The strength of the source can be computed directly by `defineback` if the requisite data are provided in a row-oriented source file. In this case, there is no need to explicitly specify the strength of the source in the input file(s) (more precisely, if a strength is specified, the flux will not be computed).

The code will need:

1. The source ion density, e.g., labeled `N(2)`,
2. The source ion *parallel* velocity, e.g., labeled `v_par` (also, `V_PAR`, `V_Par`, etc.),
3. The *parallel* (i.e., taking into account the field line pitch and angle of incidence on the surface) area, e.g., labeled `Area` (also `area` or `Area`).

The *specify\_current* option needs to be given to prevent the perpendicular area from being factored in when the internal source arrays are loaded. As noted above, if the electron temperature is also provided, the code will evaluate the incident ion energy using Eqs. (1) and (2).

**Additional Comments on Source Locations** The stratum is just the stratum label assigned in the `definegeometry2d` input file to the polygon that contains the solid surface that the source ions will be striking. This number should be the same for all of the elements of each source group. The segment number is determined by the location of that solid surface within the list of polygon points. One way to automate the setting of the segment number is to arrange the polygon points in the `definegeometry2d` input file so that the first points in the polygon are the ones that will be used as source segments. Then, you'll know that the segment number goes from 1 to the desired number of segments. Note that for `definegeometry2d` to keep track of the segment numbers, the polygon has to be specified in a clockwise direction. An alternative approach would be to define an auxiliary stratum for each source group consisting of just the desired segments. Again, the segment numbers would just range from 1 to the number of source segments.

**Screen Output from Source Group Processing** As *defineback* processes each source group, it will print to `stdout` information on each segment of that group. For example:

```

Processing source group  1

  source          stratum segment          sector points
  segment stratum segment  iy   sector      x1   start   x3
x1   end      x3
  1         7      50      0     5  1.59600E+00  1.45000E-01
1.59600E+00  1.40000E-01
  2         7      52      0     7  1.59600E+00  1.55000E-01
1.59600E+00  1.50000E-01
  3         7      49      0     9  1.59600E+00  1.40000E-01
1.59600E+00  1.37600E-01
:

```

Obviously, the first line identifies the source group number. The items in the table are

**source segment** This is the running count of segments in this source group. It can be used subsequently to identify this segment in the background netCDF file, when running the code in a debugger, or when using the *sourcetest* code. Note that this ordering may not (occasionally it may) correspond to the order of the source segments listed in the input file(s) provided to *defineback*. The other output columns can be used to relate the two lists.

**stratum** Is the stratum label of the source sector (i.e., *strata<sub>sector</sub>*). This corresponds to the “stratum” value specified in the input file.

**stratum segment** Is the stratum segment label of the sector (i.e., *sector\_strata\_segment<sub>sector</sub>*). This corresponds to the “segment” value specified in the input file.

**segment iy** Used only for 3-D cases, this is the  $i_y$  value of the zone associated with this sector and corresponds to the “segment\_iy” value specified in the input file.

**sector** Is the geometry’s *sector* number for this source segment.

**sector points** Provides the “start” and “end” points contained in the *sector\_points* array for this *sector*. Presently, the “x2” coordinate is not used (even in 3-D cases; the  $i_y$  value effectively provides that information).

## 5 Iterative Problems

This code is now capable of being used in conjunction with the UEDGE plasma code (via the version of subroutine *get\_n\_t* in the file `uedgeback.web`). Moreover, the ability to iterate between UEDGE and DEGAS 2, previously available only with `readbackground` and `updatebackground` has been incorporated. None of the code in `defineback` is UEDGE-specific, however. Establishing an analogous iterative capability with another code would involve replacing just the *get\_n\_t* routine.

Three different indicators inform `defineback` that it is being used in this iterative fashion.

1. The *iterative\_solution* keyword appears at the beginning of the `defineback` input file. This keyword needs no arguments; it confirms the user's intention to pursue an iterative solution.
2. An `oldsourcefile` is present; the actual filename is specified in the `degas2.in` file. This file contains the values of the source "class" data (currents and weights) from the initial run of `defineback` in an iterative run; i.e., this file gets written out only if the *iterative\_solution* keyword has been invoked and if a name has been specified for it. Likewise, it is read in on a subsequent ("update") run if the `oldsourcefile` has a name and is present; otherwise, `defineback` will *not* run in update mode.
3. The number of source groups *so\_grps* is greater than zero when the *get\_n\_t* subroutine is called. Directly informing the *get\_n\_t* subroutine that `defineback` is running in "update" mode would require modifying its argument list. Instead, we use the value of *so\_grps* as a proxy. When the `oldsourcefile` is read in, it sets *so\_grps* to its value from the previous invocation of `defineback`. This value reaches the *get\_n\_t* subroutine via the source "class" data.

The consistency of these three flags is checked. The most detailed checks are on the members of the source "class" that should remain the same from one iteration to the next. In particular, the current value of *so\_grps* is required to be the same as that obtained from the *oldsourcefile*; the same is true for the number of source segments and their locations within the geometry.

By storing the initial source data in the `oldsourcefile`, changes in the source current distribution on subsequent iterations can be accounted for by weighting factors (see: D. P. Stotler et al., *Contrib. Plasma Phys.* **40** (2000) 221). This is a form of correlated sampling. After some number of iterations, the distribution will have changed enough that this approach becomes ineffective. The parameters used to control this decision are *so\_rel\_wt\_min* and *so\_rel\_wt\_max*. The sampling arrays are then recomputed and the `oldsourcefile` over-written. Note that if an `oldsourcefile` filename is specified, `defineback` will look for that file. If it exists and can be read, `defineback` will assume that it is being invoked to update the plasma and source data as was just described. Hence, the user needs to delete an existing `oldsourcefile` before the initial run of `defineback`. On the other hand, if the `oldsourcefile` filename in `degas2.in` is not specified, the iterations will proceed, but without the benefits of this weight rescaling.

```
$Id: defineback.web,v 1.4 2007/06/19 17:33:43 dstotler Exp $
"defineback.f" 1 ≡
@m FILE 'defineback.web'
```

## 6 The Main Program

```
"defineback.f" 6 ≡
  program defineback
    implicit_none_f77
    implicit_none_f90

    integer nargs
    character*FILELEN inputfile

    sy_decls

    nargs = arg_count()
    if (nargs ≠ 1) then
      assert('Command_line_must_specify_an_input_file' ≡ ' ')
    end if
    call command_arg(1, inputfile)

    call readfilenames
    call read_geometry

    call nc_read_elements
    call nc_read_species
    call nc_read_reactions
    call nc_read_materials
    call nc_read_pmi
    call nc_read_problem

    call setup_back_arrays

    call setup_background(inputfile)

    call erase_geometry

  @#if 0
    call mem_check
  @#endif

  stop
end

⟨ Functions and subroutines 7 ⟩
```

## 7 Main Subroutine

```

"defineback.f" 7 ≡
  @m code_undefined 0
  @m code_int 1
  @m code_real 2

  @m code_int_min 1
  @m code_zone 1
  @m code_stratum 2
  @m code_segment 3
  @m code_segment_iy 4
  @m code_multiplicity 5
  @m code_all_segs 6
  @m code_int_max 6

  @m code_real_min 11 // These two lists need to be separate
  @m code_density 11
  @m code_temperature_J 12
  @m code_temperature_eV 13
  @m code_electron_temperature 14
  @m code_velocity_1 15
  @m code_velocity_2 16
  @m code_velocity_3 17
  @m code_velocity_par 18
  @m code_area 19
  @m code_strength 20
  @m code_e_delta 21
  @m code_e_mult 22
  @m code_e_sheath 23
  @m code_real_max 23

  @m open_file(aunit, aname)
    open(unit = aunit, file = aname, status = 'old', form = 'formatted', iostat = open_stat)
    assert(open_stat ≡ 0)
  @m next_line #:0
  @m input_done #:0
  @m next_stratum #:0
  @m strata_done #:0
  @m next_segment #:0
  @m segments_done #:0
  @m next_iy #:0
  @m iys_done #:0
  @m next_strength #:0
  @m strengths_done #:0
  @m next_tabline #:0
  @m next_rowline #:0
  @m next_datum #:0

  @m increment_iseg iseg++
  if (iseg > dim_segments) then
    dim_segments = iseg
    var_realloca(source_int_params)

```

```

var_realloca(source_real_params)
do i = code_int_min, code_int_max
  source_int_params_iseg,i = int_uninit
end do
source_int_params_iseg,code_all_segs = FALSE
source_int_params_iseg,code_multiplicity = 0
do i = code_real_min, code_real_max
  source_real_params_iseg,i = real_uninit
end do
end if

```

⟨Functions and subroutines 7⟩ ≡

**subroutine** *setup\_background*(*inputfile*)

```

define_dimen(int_params_ind, code_int_min, code_int_max)
define_dimen(real_params_ind, code_real_min, code_real_max)
define_dimen(segment_ind, dim_segments)
define_varp(source_int_params, INT, int_params_ind, segment_ind)
define_varp(source_real_params, FLOAT, real_params_ind, segment_ind)

define_varp(old_current, FLOAT, source_seg_ind)
define_varp(old_rel_wt, FLOAT, source_seg_ind)
define_varp(old_tot_curr, FLOAT, source_grp_ind)

implicit_none_f77
gi_common    // Common
zn_common
bk_common
pr_common
so_common
sc_common
sp_common
implicit_none_f90

character*FILELEN inputfile    // Input

real density, temperature, dummy, circum, area,    /* Local */
      rtest, real_mult, mult, puff_temp

integer type, geom, nflights, length, p, b, e, nunit_n, specify_flux, nunit_t, num_segments, iseg,
      i, open_stat, targ, disk_in2, dim_segments, itest, data_type, data_code, update, old_grps,
      old_seg_tot, grp, iterative, jr

character*1 back_label
character*LINELEN line, type_string, nt_string, file_format
character*FILELEN source_file

external phi_sheath    // External
real phi_sheath

sp_decl(species)
sp_decl(root_sp)
bk_decl(back)
zn_decl(zone)
sc_decl(sector)
vc_decl(xdiff)
vc_decl(targ.v)

```

```

declare_varp(source_int_params)
declare_varp(source_real_params)

declare_varp(old_current)
declare_varp(old_rel_wt)
declare_varp(old_tot_curr)

⟨Memory allocation interface 0⟩
st_decls

open_file(diskin, inputfile) /* Try reading old sources file. If present, assume that this run is an
    update ⇒ update = TRUE. */
call nc_read_old_sources(update)

if (update ≡ FALSE) then
    so_set_run_flags
    so_grps = 0
    so_seg_tot = 0
else
    assert(so_grps > 0)
    assert(so_seg_tot > 0)
    var_alloc(old_current)
    var_alloc(old_rel_wt)
    var_alloc(old_tot_curr)
    old_grps = so_grps
    old_seg_tot = so_seg_tot
    do grp = 1, so_grps
        old_tot_curr_grp = so_tot_curr(grp)
    end do
    do iseg = 1, so_seg_tot
        old_current_iseg = source_current_iseg
        old_rel_wt_iseg = source_segment_rel_wt_iseg
    end do
end if

dim_segments = 0
iterative = FALSE /* Use this as a default since these problems are all likely to be 2-D so that
    the velocity transformations are governed by geometry_symmetry. */
if ((geometry_symmetry ≡ geometry_symmetry_plane) ∨ (geometry_symmetry ≡
    geometry_symmetry_plane_hw) ∨ (geometry_symmetry ≡ geometry_symmetry_oned)) then
    background_coords = plasma_coords_cartesian
else if ((geometry_symmetry ≡ geometry_symmetry_cylindrical) ∨ (geometry_symmetry ≡
    geometry_symmetry_cyl_hw) ∨ (geometry_symmetry ≡ geometry_symmetry_cyl_section)) then
    background_coords = plasma_coords_cylindrical
else
    assert('Unexpected_value_of_geometry_symmetry' ≡ '␣')
end if /* Loop to read through input file. */
next_line: continue

if (¬read_string(diskin, line, length)) then
    close (unit = diskin)
    goto input_done
else
    assert(length ≤ len(line))
    length = parse_string(line(: length))
    p = 0

```

```

assert(next_token(line, b, e, p))
if (line(b : e) ≡ 'plasma_file') then
  assert(next_token(line, b, e, p))
  nt_string = line(b :)
  if (next_token(line, b, e, p)) then
    file_format = line(b : e)
  else
    file_format = char_undef
  end if
  if (file_format ≡ 'row') then
    call get_n_t_row(nt_string)
  else // Default; dedicated subroutine also
    if (update ≡ TRUE) then /* There are versions of the get_n_t routine that reset the source
      arrays in this case. This only makes sense when the user wants an iterative solution. */
      assert(iterative ≡ TRUE)
    end if
    call get_n_t(nt_string)
  end if
else if (line(b : e) ≡ 'plasma_coords') then
  assert(next_token(line, b, e, p))
  if (line(b : e) ≡ 'cartesian') then
    background_coords = plasma_coords_cartesian
  else if (line(b : e) ≡ 'cylindrical') then
    background_coords = plasma_coords_cylindrical
  else
    assert('␣Illegal␣value␣of␣plasma_coords' ≡ '␣')
  end if
else if (line(b : e) ≡ 'iterative_run') then
  iterative = TRUE
else if (line(b : e) ≡ 'new_source_group') then
  so_grps++
  type = int_undef
  geom = int_undef
  species = int_undef
  root_sp = int_undef
  puff_temp = const(3., 2) * boltzmanns_const
  nflights = 100
  specify_flux = TRUE
  num_segments = int_uninit /* We can initialize these arrays out to their current dimensions.
    If they get expanded during the process of adding segments, the additional initializations
    will be done segment-by-segment (see increment_iseq). */
  if (dim_segments > 0) then
    do iseq = 1, dim_segments
      do i = code_int_min, code_int_max
        source_int_params_iseq,i = int_uninit
      end do
      source_int_params_iseq,code_all_segs = FALSE
      source_int_params_iseq,code_multiplicity = 0
      do i = code_real_min, code_real_max
        source_real_params_iseq,i = real_uninit
      end do

```

```

    end do
  end if

else if (line(b : e) ≡ 'source_type') then
  assert(next_token(line, b, e, p))
  type_string = line(b : e)
  assert(type ≡ int_undef)
  do i = 1, so_type_num
    if (type_string ≡ so_name(i)) then
      type = i
    end if
  end do
  assert(type ≠ int_undef) // Also do a final check before using

else if (line(b : e) ≡ 'source_geom') then
  assert(next_token(line, b, e, p))
  assert(geom ≡ int_undef)
  if (line(b : e) ≡ 'point') then
    geom = so_point
  else if (line(b : e) ≡ 'line') then
    geom = so_line
  else if (line(b : e) ≡ 'surface') then
    geom = so_surface
  else if (line(b : e) ≡ 'volume') then
    geom = so_volume
  end if
  assert(geom ≠ int_undef)

else if (line(b : e) ≡ 'source_species') then
  assert(next_token(line, b, e, p))
  assert(species ≡ int_undef)
  species = sp_lookup(line(b : e))
  assert(sp_check(species))

else if (line(b : e) ≡ 'source_root_sp') then
  assert(next_token(line, b, e, p))
  assert(root_sp ≡ int_undef)
  root_sp = sp_lookup(line(b : e))
  assert(sp_check(root_sp))

else if (line(b : e) ≡ 'source_nflights') then
  assert(next_token(line, b, e, p))
  nflights = read_integer(line(b : e))

else if (line(b : e) ≡ 'source_puff_temp') then
  assert(next_token(line, b, e, p))
  puff_temp = read_real(line(b : e)) * boltzmanns_const
  assert(puff_temp ≥ zero)

else if (line(b : e) ≡ 'specify_flux') then
  specify_flux = TRUE

else if (line(b : e) ≡ 'specify_current') then
  specify_flux = FALSE

else if (line(b : e) ≡ 'source_stratum') then
  iseg = 0

```

```

next_stratum: continue
  if ( $\neg$ next_token(line, b, e, p)) then
    if (num_segments  $\equiv$  int_uninit) then
      num_segments = iseg
    else
      assert(num_segments  $\equiv$  iseg)
    end if
    goto strata_done
  else
    increment_iseg
    source_int_paramsiseg,code_stratum = read_integer(line(b : e))
  end if
  goto next_stratum
strata_done: continue

  else if (line(b : e)  $\equiv$  'source_segment') then
    iseg = 0
next_segment: continue
  if ( $\neg$ next_token(line, b, e, p)) then
    if (num_segments  $\equiv$  int_uninit) then
      num_segments = iseg
    else
      assert(num_segments  $\equiv$  iseg)
    end if
    goto segments_done
  else
    increment_iseg
    if (line(b : e)  $\equiv$  '**') then
      source_int_paramsiseg,code_all_segs = TRUE
      source_int_paramsiseg,code_segment = int_unused
    else
      source_int_paramsiseg,code_all_segs = FALSE
      source_int_paramsiseg,code_segment = read_integer(line(b : e))
    end if
  end if
  goto next_segment
segments_done: continue

  else if (line(b : e)  $\equiv$  'source_segment_iy') then
    /* Note that this does not allow for a wildcard specification (so that inputs for symmetric
    problems can be reused without modification). */
    iseg = 0
next_iy: continue
  if ( $\neg$ next_token(line, b, e, p)) then
    if (num_segments  $\equiv$  int_uninit) then
      num_segments = iseg
    else
      assert(num_segments  $\equiv$  iseg)
    end if
    goto iys_done
  else
    increment_iseg
    source_int_paramsiseg,code_segment_iy = read_integer(line(b : e))

```

```

    end if
    goto next_iy
iys_done: continue

    else if (line(b : e) ≡ 'source_strength') then
        iseg = 0
next_strength: continue
        if (¬next_token(line, b, e, p)) then
            if (num_segments ≡ int_uninit) then
                num_segments = iseg
            else
                assert(num_segments ≡ iseg)
            end if
            goto strengths_done
        else
            increment_iseg
            source_real_paramsiseg,code_strength = read_real(line(b : e))
        end if
        goto next_strength
strengths_done: continue

    else if (line(b : e) ≡ 'source_file') then
        iseg = 0
        assert(next_token(line, b, e, p))
        source_file = line(b : e)
        disk2 = disk1 + 1
        open_file(disk2, source_file)
        if (next_token(line, b, e, p)) then
            file_format = line(b : e)
        else
            file_format = 'tabular'
        end if
        if (file_format ≡ 'tabular') then
next_tabline: continue
            if (read_string(disk2, line, length)) then /* I suppose we could generalize this if-then
                to match the others above and add a format specification below so that this file could
                contain just a subset of stratum, segment, and strength. But, do not see the need. */
                assert(length ≤ len(line))
                length = parse_string(line(: length))
                p = 0
                increment_iseg
                assert(next_token(line, b, e, p))
                source_int_paramsiseg,code_stratum = read_integer(line(b : e))
                assert(next_token(line, b, e, p))
                source_int_paramsiseg,code_segment = read_integer(line(b : e))
                assert(next_token(line, b, e, p))
                source_real_paramsiseg,code_strength = read_real(line(b : e))
                goto next_tabline
            end if
            assert(num_segments ≡ int_uninit)
            num_segments = iseg
            close(unit = disk2)
        else if (file_format ≡ 'row') then

```

```

        < Read Row Source File 8 >
    end if

    else if (line(b : e) ≡ 'end_source_group') then
        < Setup Source Group 9 >
    else
        assert('Unknown_keyword' ≡ line(b : e))
    end if
end if
goto next_line

```

*input\_done*: **continue**

```

    /* Write densities and temperatures out to a file designed to mimic format of outputbrowser text
       files so these can be used with the AVS/Express graphing package. */
    nunit_n = disk_in + 5
    open(unit = nunit_n, file = 'density_1.txt', status = 'unknown')
    write(nunit_n, '(i6.6)') zn_num
    do back = 1, bk_num
        nunit_t = nunit_n + 1
        write(back_label, '(i1)') back
        open(unit = nunit_t, file = 'temperature_' || back_label || '.txt', status = 'unknown')
        write(nunit_t, '(i6.6)') zn_num
        do zone = 1, zn_num
            if (zn_type(zone) ≡ zn_plasma) then
                density = bk_n(back, zone)
                temperature = bk_temp(back, zone)
            else
                density = zero
                temperature = zero
            end if
            if (back ≡ 1) then
                write(nunit_n, '(4x,i6,6x,i4,6x,1pe13.5,2x,0pf7.4)') zone, back, density, dummy
            end if
            write(nunit_t,
                '(4x,i6,6x,i4,6x,1pe13.5,2x,0pf7.4)') zone, back, temperature/electron_charge,
                dummy
        end do
        close(unit = nunit_t)
    end do
    close(unit = nunit_n) /* Note that we do var_reallocb here so that set_background_sources can
                           do an explicit var_realloc once the final so_grps is known. The other (segment-based) arrays,
                           however, will still be growing one segment at a time. */
    if (update ≡ FALSE) then
        var_reallocb(source_base_ptr)
        var_reallocb(source_num_segments)
        var_reallocb(source_type)
        var_reallocb(source_geometry)
        var_reallocb(source_num_flights)
        var_reallocb(source_num_checkpoints)
        var_reallocb(source_species)
        var_reallocb(source_root_species)
        var_reallocb(source_parameters)
        var_reallocb(source_total_current)
    end if

```

```

    var_reallocb(source_weight_norm)
    var_reallocb(source_scale_factor)
end if
@#if 0
    var_reallocb(source_current)
    var_reallocb(source_segment_ptr)
    var_reallocb(source_segment_rel_wt)
    var_reallocb(source_segment_prob_alias)
    var_reallocb(source_segment_ptr_alias)
@#endif
    var_reallocb(source_int_params)
    var_reallocb(source_real_params)
if (so_seg_tot == 0) then
    assert('No sources were defined!' == ' ')
end if
if (update == FALSE) then
    call set_background_sources
else
    assert(update == TRUE)
    assert(iterative == TRUE)
    if (pr_bkrc_num > 0) then
        do jr = 1, pr_bkrc_num
            so_grps++
            so_seg_tot = so_base(so_grps) - 1
            call set_background_rate(jr, so_grps, update)
        end do
    end if
end if
end if
if (update == FALSE) then
    /* The iterative flag is essentially equivalent to the expected uedge argument. */
    call init_wt_alias(iterative)
else
    assert(iterative == TRUE)
    assert(so_grps == old_grps)
    assert(so_seg_tot == old_seg_tot)
    call update_wt_alias(old_current, old_rel_wt, old_tot_curr)

    var_free(old_current)
    var_free(old_tot_curr)
    var_free(old_rel_wt)
end if
    call write_background
    /* Updated plasma sector properties and possibly relabeled vacuum zones. */
    call write_geometry

    var_free(source_int_params)
    var_free(source_real_params)

return
end

```

See also sections 10, 11, and 12.

This code is used in section 6.

## 8 Read a row-oriented source file

⟨ Read Row Source File 8 ⟩ ≡

```

next_rowline: continue
  if (read_string(diskin2, line, length)) then
    assert(length ≤ len(line))
    length = parse_string(line(: length))
    p = 0
    assert(next_token(line, b, e, p))
    itest = read_int_soft_fail(line(b : e))
    rtest = read_real_soft_fail(line(b : e))    /* Note that read_real_soft_fail can read an integer, but
      read_int_soft_fail cannot read a real (should be tested on a variety of platforms or made part of
      sysdeptest). So, the following test only works in this order. */
    if (itest ≠ int_undef) then
      data_type = code_int
    else if (rtest ≠ real_undef) then
      data_type = code_real
    else
      data_type = code_undefined
    end if
    if (data_type ≡ code_undefined) then
      if (num_segments ≡ int_uninit) then
        if (iseg ≠ 0)
          num_segments = iseg
        else
          assert(num_segments ≡ iseg)
        end if
      call parse_label(line, data_code, back)
      assert(data_code ≠ code_undefined)
      iseg = 0    /* Two special cases, both dealing with temperature. Note that this is the only usage
        of the back index, i.e., to identify the electron temperature. All other background parameters
        are assumed to be those of the source root species. */
      real_mult = one
      if (data_code ≡ code_temperature_eV) then
        real_mult = electron_charge
        data_code = code_temperature_J
      end if
      if (data_code ≡ code_temperature_J) then
        /* Separate these two ifs since back may not be defined. */
        if (sp_sy(pr_background(back)) ≡ 'e')
          data_code = code_electron_temperature
        end if
      goto next_rowline
    else    /* Contains a line of data */
      assert(data_code ≠ code_undefined)
next_datum: continue
  increment_iseg
  if (data_type ≡ code_int) then
    source_int_paramsiseg, data_code = read_integer(line(b : e))
  else if (data_type ≡ code_real) then

```

```
      source_real_paramsiseg,data_code = read_real(line(b : e)) * real_mult
end if
if (next_token(line, b, e, p)) then
  goto next_datum
else
  goto next_rowline
end if
end if    // label or data
end if    // string presence
close(unit = diskin2)
```

This code is used in section 7.

## 9 Setup a source group

⟨ Setup Source Group 9 ⟩ ≡

```

    assert(num_segments > 0) /* Check to see if we are supposed to compute the source strength
        (current) from the density, velocity, and parallel area. Here, the velocity must be  $v_{\parallel}$  since we
        cannot convert between  $\vec{v}$  components and  $v_{\parallel}$  without pitch data, etc. Likewise, we have to have
        the parallel area. If we read in the pitch at the same time, we could compute the area in the
        poloidal plane like we do above. */
    if (source_real_params_1,code_strength ≡ real_uninit) then
        assert(source_real_params_1,code_density ≠ real_uninit)
        assert(source_real_params_1,code_velocity_par ≠ real_uninit)
        assert(source_real_params_1,code_area ≠ real_uninit)
        assert(specify_flux ≡ FALSE)
        do iseg = 1, num_segments
            source_real_params_iseg,code_strength = source_real_params_iseg,code_density *
                abs(source_real_params_iseg,code_velocity_par) * source_real_params_iseg,code_area
        end do
    end if /* Compute more detailed sheath parameters if the appropriate data are present. The
        current implementation of phi_sheath depends only on the magnitude of the velocity. However, a
        more detailed model is possible, hence, the use of the vector velocity as an argument. For now,
        assume that only the magnitude counts and use the information at hand. */
    if ((source_real_params_1,code_temperature_J ≠ real_uninit) ∧
        (source_real_params_1,code_electron_temperature ≠ real_uninit) ∧
        ((source_real_params_1,code_velocity_par ≠ real_uninit) ∨ (source_real_params_1,code_velocity_1 ≠
            real_uninit))) then
        do iseg = 1, num_segments
            assert(source_real_params_iseg,code_temperature_J > zero)
            assert(source_real_params_iseg,code_electron_temperature > zero)
            if ((source_real_params_iseg,code_velocity_par ≠ real_uninit) ∧ (source_real_params_iseg,code_velocity_1 ≡
                real_uninit)) then
                vc_set(targ_v, source_real_params_iseg,code_velocity_par, zero, zero)
            else
                vc_set(targ_v, source_real_params_iseg,code_velocity_1, source_real_params_iseg,code_velocity_2,
                    source_real_params_iseg,code_velocity_3)
            end if
            assert(vc_abs(targ_v) > zero)
            source_real_params_iseg,code_e_delta = const(3.0) * source_real_params_iseg,code_temperature_J +
                half * source_real_params_iseg,code_electron_temperature
            source_real_params_iseg,code_e_sheath = phi_sheath(vc_args(targ_v),
                source_real_params_iseg,code_temperature_J, source_real_params_iseg,code_electron_temperature) *
                source_real_params_iseg,code_electron_temperature
            source_real_params_iseg,code_e_mult = zero
        end do
    end if /* Now pull it all together into the final description of the source. */
    if (update ≡ FALSE) then
        var_realloca(source_base_ptr)
        var_realloca(source_num_segments)
        var_realloca(source_type)
        var_realloca(source_geometry)

```

```

var_realloca(source_num_flights)
var_realloca(source_num_checkpoints)
var_realloca(source_species)
var_realloca(source_root_species)
var_realloca(source_parameters)
var_realloca(source_total_current)
var_realloca(source_weight_norm)
var_realloca(source_scale_factor)
so_base(so_grps) = so_seg_tot + 1
assert(type ≠ int_undef)
so_type(so_grps) = type
assert(geom ≠ int_undef)
so_geom(so_grps) = geom
so_nflights(so_grps) = nflights
so_chkpt(so_grps) = 0
assert(sp_check(species))
so_species(so_grps) = species // Use neutralize_species?
assert(sp_check(root_sp))
so_root_sp(so_grps) = root_sp
so_params(so_grps, so_param_puff_temp) = puff_temp
so_tot_curr(so_grps) = zero
so_scale(so_grps) = one
else
  /* update == TRUE. The idea is to only reset source_current and so_tot_curr. Also allow so_grps
  and so_seg_tot to work as in the other case since they appear as array indices everywhere. */
  assert(iterative ≡ TRUE)
  assert(so_base(so_grps) ≡ so_seg_tot + 1)
  assert(so_type(so_grps) ≡ type)
  assert(so_geom(so_grps) ≡ geom)
  so_tot_curr(so_grps) = zero
end if
write(stdout, *)
write(stdout, *) 'Processing source group', so_grps
write(stdout, *)
write(stdout, '(2x,a,12x,a,2x,a,30x,a)') 'source', 'stratum', 'segment', 'sector_points'
write(stdout, '(2x,a,2x,a,2x,a,4x,a,4x,a,6x,a,4x,a,4x,a,13x,a,5x,a,5x,a)') 'segment',
'stratum', 'segment', 'iy', 'sector', 'x1', 'start', 'x3', 'x1', 'end', 'x3'
/* Go through all of the segments one time just to count the number of sectors associated with
each (due to the "all segments" option or multiple iy values). The "strength" is divided by this
"multiplicity" factor, so we need to know its value before setting the final source arrays. */
do sector = 1, nsectors
do iseg = 1, num_segments /* Having code_segment_iy be int_uninit is designed to handle two
cases without any user input: the fully symmetric (2-D) case and the nearly symmetric case in
which all sectors labeled with the same stratum and segment numbers are to be included. */
if ((sc_plasma_check(sector_type_pointer_sector,sc_plasma) ∨
sc_vacuum_check(sector_type_pointer_sector,sc_vacuum)) ∧ (strata_sector ≡
source_int_params_iseg,code_stratum) ∧ ((zn_index(sector_zone_sector,
zi_iy) ≡ source_int_params_iseg,code_segment_iy) ∨ (source_int_params_iseg,code_segment_iy ≡
int_uninit)) ∧ ((sector_strata_segment_sector ≡ source_int_params_iseg,code_segment) ∨
(source_int_params_iseg,code_all_segs ≡ TRUE))) then
source_int_params_iseg,code_multiplicity ++
end if

```

```

end do
end do /* Now set up the final source arrays. */
do sector = 1, nsectors
do iseg = 1, num_segments
if ((sc_plasma_check(sector_type_pointer_sector, sc_plasma)  $\vee$ 
sc_vacuum_check(sector_type_pointer_sector, sc_vacuum))  $\wedge$  (strata_sector  $\equiv$ 
source_int_params_iseg, code_stratum)  $\wedge$  ((zn_index(sector_zone_sector,
zi_iy)  $\equiv$  source_int_params_iseg, code_segment_iy)  $\vee$  (source_int_params_iseg, code_segment_iy  $\equiv$ 
int_uninit))  $\wedge$  ((sector_strata_segment_sector  $\equiv$  source_int_params_iseg, code_segment)  $\vee$ 
(source_int_params_iseg, code_all_segs  $\equiv$  TRUE))) then
zone = sector_zone_sector
so_seg_tot++
@#if 0
write(stdout, '(2x, i4, 5x, i4, 5x, i4, 5x, i4, 4x, i5, 1x, 4(1pe13.5, 2x))') iseg,
source_int_params_iseg, code_stratum, /* This is the subsequent version. Replace use of
source_int_params here with the equivalent sector-based quantity below. */
write(stdout, '(2x, i4, 5x, i4, 5x, i4, 5x, i4, 4x, i5, 1x, 4(1pe13.5, 2x))')
so_seg_tot - so_base(so_grps) + 1, source_int_params_iseg, code_stratum,
source_int_params_iseg, code_segment,
@#else
write(stdout,
'(2x, i4, 5x, i4, 5x, i4, 5x, i4, 4x, i5, 1x, 4(1pe13.5, 2x))') so_seg_tot - so_base(so_grps) + 1,
strata_sector, sector_strata_segment_sector, zn_index(sector_zone_sector, zi_iy), sector,
sector_points_sector, sc_neg, 1, sector_points_sector, sc_neg, 3, sector_points_sector, sc_pos, 1,
sector_points_sector, sc_pos, 3
@#endif
if (update  $\equiv$  FALSE) then
var_realloca(source_current)
var_realloca(source_segment_ptr)
var_realloca(source_segment_rel_wt)
var_realloca(source_segment_prob_alias)
var_realloca(source_segment_ptr_alias)
end if
assert(source_real_params_iseg, code_strength > zero  $\wedge$  source_real_params_iseg, code_strength  $\neq$ 
real_uninit)
assert(source_int_params_iseg, code_multiplicity > 0)
mult = one / float(source_int_params_iseg, code_multiplicity)
if (specify_flux  $\equiv$  TRUE) then
vc_difference(sector_points_sector, sc_neg, sector_points_sector, sc_pos, xdiff)
assert(xdiff2  $\equiv$  zero)
if (geometry_symmetry  $\equiv$  geometry_symmetry_plane) then /* But, by definition of this
symmetry the problem spans the full length of the universal cell in this direction. */
circum = universal_cell_max2 - universal_cell_min2
else if (geometry_symmetry  $\equiv$  geometry_symmetry_plane_hw) then
circum = zone_maxzone, 2 - zone_minzone, 2
else if (geometry_symmetry  $\equiv$  geometry_symmetry_cylindrical) then
/* Note that for this case, we follow the original convention of setting zone_minzone, 2 =
zone_maxzone, 2 = 0, so use  $\pi$  explicitly. */
circum = PI * (sector_points_sector, sc_neg, 1 + sector_points_sector, sc_pos, 1)
else if ((geometry_symmetry  $\equiv$  geometry_symmetry_cyl_hw)  $\vee$  (geometry_symmetry  $\equiv$ 
geometry_symmetry_cyl_section)) then

```

```

        circum = half * (sector_pointssector,sc_neg,1 + sector_pointssector,sc_pos,1) * (zone_maxzone,2 -
            zone_minzone,2)
    else
        assert('Unexpected_geometry_symmetry_in_setting_up_source' ≡ ' ')
    end if
    area = circum * vc_abs(xdiff)
    assert(area > zero)
    source_currentso_seg_tot = source_real_paramsiseg,code_strength * area * mult
else /* Specify current */
    source_currentso_seg_tot = source_real_paramsiseg,code_strength * mult
end if
so_tot_curr(so_grps) += source_currentso_seg_tot
if (update ≡ FALSE) then
    source_segment_ptrso_seg_tot = sector
    source_segment_rel_wtso_seg_tot = one
else
    assert(source_segment_ptrso_seg_tot ≡ sector)
    /* source_segment_rel_wt gets set in update_wt_alias. */
end if /* Assign sheath properties using the old DEGAS model since we lack specific boundary
parameters (and fluid velocities). An alternative (e.g., in the UEDGE case) would be to use
the phi_sheath function. Assumes that the first background species is "e". */
if (so_type(so_grps) ≡ so_plate) then
    assert(sc_plasma_check(sector_type_pointersector,sc_plasma))
    targ = sector_type_pointersector,sc_plasma
    if ((source_real_paramsiseg,code_e_delta ≠ real_uninit) ∧ (source_real_paramsiseg,code_e_sheath ≠
        real_uninit) ∧ (source_real_paramsiseg,code_e_mult ≠ real_uninit)) then
        plasma_e_ion_deltatarg = source_real_paramsiseg,code_e_delta
        plasma_e_ion_sheathtarg = source_real_paramsiseg,code_e_sheath
        plasma_e_ion_multtarg = source_real_paramsiseg,code_e_mult
    else
        assert(sp_sy(pr_background(1)) ≡ 'e')
        assert(bk_temp(1, zone) > zero)
        plasma_e_ion_deltatarg = const(3.) * bk_temp(1, zone)
        plasma_e_ion_multtarg = one
        plasma_e_ion_sheathtarg = zero
    end if
end if
end do
end do
if (update ≡ FALSE) then
    so_nseg(so_grps) = so_seg_tot - so_base(so_grps) + 1
else
    assert(so_nseg(so_grps) ≡ so_seg_tot - so_base(so_grps) + 1)
end if
end if

```

This code is used in section 7.

## 10 Parse tabular plasma data header

This routine takes the text description of the plasma data format provided by the user and translates it into an integer code that can be more readily used in reading the data. This additional step also facilitates testing the format. One additional capability simplifies the specification of data for a large number of background species.

```
"defineback.f" 10 ≡
  @m format_max LINELEN    // format_code is parsed from a line
     // Reserve first 9 values for background indices
  @m form_undefined 0
  @m form_zone 10
  @m form_density 11
  @m form_temperature 12
  @m form_velocity_1 13
  @m form_velocity_2 14
  @m form_velocity_3 15
  @m token_loop #:0

⟨Functions and subroutines 7⟩ +≡
  subroutine parse_format(format_string, bk_num, format_code, count)
    implicit_none_f77
    implicit_none_f90

    character*LINELEN format_string    // Input
    integer bk_num

    integer count    // Output
    integer format_code format_max

    integer p, b, e, length, b_repeat, b_code, bi, sub_count, i, j    // Local
    character*LINELEN line

    ⟨Memory allocation interface 0⟩
    st_decls

    line = format_string
    length = parse_string(line)
    p = 0
    count = 0
    b_repeat = FALSE
      /* If the number of background species exceeds 10, need to do this more intelligently. */
    assert(bk_num < 10)

token_loop: continue
    if (~next_token(line, b, e, p))
      goto eof
    count++
    if ((line(b : e) ≡ 'zone') ∨ (line(b : e) ≡ 'ZONE')) then
      format_code count = form_zone
    else
      if ((line(b : b) ≡ 'n') ∨ (line(b : b) ≡ 'N')) then
        format_code count = form_density
      else if ((line(b : b) ≡ 't') ∨ (line(b : b) ≡ 'T')) then
```

```

    format_code_count = form_temperature
else if ((line(b : b) ≡ 'v') ∨ (line(b : b) ≡ 'V')) then
    if (line(b + 1 : b + 1) ≡ '1') then
        format_code_count = form_velocity_1
    else if (line(b + 1 : b + 1) ≡ '2') then
        format_code_count = form_velocity_2
    else if (line(b + 1 : b + 1) ≡ '3') then
        format_code_count = form_velocity_3
    else
        write(stderr, *) '␣Illegal␣format␣token,␣', line(b : e)
        assert( $\mathcal{F}$ )
    end if
else
    write(stderr, *) '␣Illegal␣format␣token,␣', line(b : e)
    assert( $\mathcal{F}$ )
end if    /* Note: had originally wanted this test to look like index(line(b:e), '(') > 0, but
    had trouble getting ftangle to deal with it (the array index switching option is to blame). The
    workaround is a bit heavy-handed but effective. Here's John Krommes' explanation:

```

This handles the entire first argument as a string, essentially defeating the index-reversing feature. So that it doesn't get confused by the comma for figuring out the first argument to **\$TRANSLIT**, I've replaced the comma by *C*; it then translates it back.

```

*/
  count++
  b_code = $TRANSLIT(index(line(b:e)C'('), 'C', ',','') + 1
  assert(b_code > 1)
  b_code = b + b_code - 1 // Relate to start of whole line
  if (line(b_code : b_code) ≡ 'b' ∨ line(b_code : b_code) ≡ 'B') then
    b_repeat = TRUE
    format_code_count = 1
  else
    assert(b_repeat ≡ FALSE)
    bi = read_int_soft_fail(line(b_code : b_code))
    if (bi ≡ int_undef) then
      write(stderr, *) 'Bad background code in token', line(b:e)
      assert(F)
    end if
    format_code_count = bi
  end if
end if
goto token_loop
eof: continue
assert(count > 0)
if ((b_repeat ≡ TRUE) ∧ (bk_num > 1)) then
  sub_count = count
  do i = 2, bk_num
    do j = 1, sub_count
      if ((format_code_j ≡ form_density) ∨ (format_code_j ≡ form_temperature) ∨ (format_code_j ≡
        form_velocity_1) ∨ (format_code_j ≡ form_velocity_2) ∨ (format_code_j ≡ form_velocity_3))
        then
          count++
          format_code_count = format_code_j
          /* If not a zone number, then only other possibility is a background number. */
        else if (format_code_j ≠ form_zone) then
          count++
          format_code_count = i
        end if
      end do
    end do
  end if
end if
return
end

```

## 11 Read row-oriented plasma data

This is analogous to the default routine, but the data are arranged in rows rather than in columns. Again, if zone number data are provided, they are used to control the ordering of the data values. If not, the data are assumed to be arranged by ascending zone number. The data are read using DEGAS 2's string-handling routines, giving the user flexibility in the appearance of the data. This also permits the user to provide fewer data values than zones, if desired.

```
"defineback.f" 11 ≡
  @m next_line #:0
  @m zone_loop #:0
  @m density_loop #:0
  @m temperature_loop #:0
  @m velocity_loop #:0

  @m zone_iy_mapm(iy, zone) zone_iy_map(zone-1)*dim_y+iy

⟨ Functions and subroutines 7 ⟩ +≡
  subroutine get_n_t_row(nt_string)

    define_dimen(iy_zone_ind, dim_y * zn_num)
    define_varp(zone_list, INT, zone_ind)
    define_varp(n_iy, INT, zone_ind)
    define_varp(zone_iy_map, INT, iy_zone_ind)

    implicit_none_f77
    bk_common // Common
    zn_common
    gi_common
    implicit_none_f90

    character*FILELEN nt_string // Input

    integer p, b, e, length, data_code, count, v_index, /* Local */
      open_stat, diskin3, itest, dim_y, iy, symmetric_plasma
    real temperature_mult, rtest
    character*LINELLEN line
    character*FILELEN plasma_file

    zn_decl(zone)
    zn_decl(iy_zone)
    bk_decl(back)

    declare_varp(zone_list)
    declare_varp(n_iy)
    declare_varp(zone_iy_map)

    ⟨ Memory allocation interface 0 ⟩
    st_decls

    var_alloc(zone_list)
    var_alloc(n_iy)
    dim_y = 1
    do zone = 1, zn_num
      zone_list_zone = int_undef
      n_iy_zone = 0
```

```

end do /* Assuming here that we will frequently want the plasma data to be toroidally symmetric
with the values in a single plane being specified in the input file. We use here the values of
zn_index(zone, zi_ptr) to count the other zones which point to them and set up a mapping that
we can use later in the subroutine. Note that we make no assumptions about the value of this
zone index iy. The "reference" zones are identified as the ones having zn_index(zone, zi_ptr)
= zone; for them, n_iy_zone holds the number of zones referring to it. For the others,
zn_index(zone, zi_ptr) ≠ zone and n_iy_zone serves as a pointer into the mapping array. */
do zone = 1, zn_num
  if (zn_index(zone, zi_ptr) ≠ zone) then
    n_iy_zn_index(zone, zi_ptr)++
    dim_y = max(n_iy_zn_index(zone, zi_ptr), dim_y)
    n_iy_zone = n_iy_zn_index(zone, zi_ptr)
  end if
end do

assert((dim_y ≡ 1) ∨ (geometry_symmetry ≡ geometry_symmetry_plane_hw) ∨ (geometry_symmetry ≡
geometry_symmetry_cyl_hw) ∨ (geometry_symmetry ≡ geometry_symmetry_cyl_section))

var_alloc(zone_iy_map)
do zone = 1, zn_num
  do iy = 1, dim_y
    zone_iy_mapm(iy, zone) = int_undef
  end do
end do
if (dim_y > 1) then
  do zone = 1, zn_num
    if (zn_index(zone, zi_ptr) ≠ zone) then
      assert((n_iy_zone > 0) ∧ (n_iy_zone ≤ dim_y))
      zone_iy_mapm(n_iy_zone, zn_index(zone, zi_ptr)) = zone
    end if
  end do
end if

p = 0
assert(next_token(nt_string, b, e, p))
plasma_file = nt_string(b : e) /* Continue parsing this string to see if the symmetry is specified.
The next token must be row for this routine to be called. */
assert(next_token(nt_string, b, e, p))
assert(nt_string(b : e) ≡ 'row')
if (next_token(nt_string, b, e, p)) then
  if (nt_string(b : e) ≡ '2D') then
    symmetric_plasma = TRUE
  else if (nt_string(b : e) ≡ '3D') then
    symmetric_plasma = FALSE
  else
    assert('Unexpected plasma symmetry specification' ≡ nt_string(b : e))
  end if
else
  symmetric_plasma = TRUE // Default case
end if

diskin3 = diskin + 2
open_file(diskin3, plasma_file)

data_code = code_undefined

```

```

    count = 0
next_line: continue
    if (read_string(diskin3, line, length)) then
        assert(length ≤ len(line))
        length = parse_string(line(: length))
        p = 0
        assert(next_token(line, b, e, p))
        itest = read_int_soft_fail(line(b : e))
        rtest = read_real_soft_fail(line(b : e))
        if ((itest ≡ int_undef) ∧ (rtest ≡ real_undef)) then
            call parse_label(line, data_code, back)
            assert(data_code ≠ code_undefined)
            count = 0
            goto next_line
        else /* Contains a line of data */
            assert(data_code ≠ code_undefined)
            if (data_code ≡ code_zone) then
zone_loop: continue
                count++
                zone_list_count = read_integer(line(b : e))
                assert(zn_check(zone_list_count))
                assert(zn_type(zone_list_count) ≡ zn_plasma)
                assert((zn_index(zone_list_count, zi_ptr) ≡ zone_list_count) ∨ (symmetric_plasma ≡ FALSE))
                if (next_token(line, b, e, p)) then
                    goto zone_loop
                else
                    goto next_line
                end if
            else if (data_code ≡ code_density) then
density_loop: continue
                count++
                if (zn_check(zone_list_count)) then
                    zone = zone_list_count
                else
                    zone = count
                end if
                assert(zn_check(zone))
                assert((zn_index(zone, zi_ptr) ≡ zone) ∨ (symmetric_plasma ≡ FALSE))
                assert(zn_type(zone) ≡ zn_plasma)
                assert(bk_check(back))
                bk_n(back, zone) = read_real(line(b : e))
                if ((dim_y > 1) ∧ (symmetric_plasma ≡ TRUE)) then
                    do iy = 1, dim_y
                        iy_zone = zone_iy_mapm(iy, zone)
                        if ((zn_check(iy_zone)) ∧ (zn_type(iy_zone) ≡ zn_plasma)) then
                            bk_n(back, iy_zone) = bk_n(back, zone)
                        end if
                    end do
                end if
            end if
            if (next_token(line, b, e, p)) then
                goto density_loop
            end if

```

```

else
  goto next_line
end if

else if ((data_code ≡ code_temperature_J) ∨ (data_code ≡ code_temperature_eV)) then
temperature_loop: continue
  count++
  if (zn_check(zone_list_count)) then
    zone = zone_list_count
  else
    zone = count
  end if
  assert(zn_check(zone))
  assert((zn_index(zone, zi_ptr) ≡ zone) ∨ (symmetric_plasma ≡ FALSE))
  assert(zn_type(zone) ≡ zn_plasma)
  assert(bk_check(back))
  if (data_code ≡ code_temperature_eV) then
    temperature_mult = electron_charge
  else
    temperature_mult = one
  end if
  bk_temp(back, zone) = read_real(line(b : e)) * temperature_mult
  if ((dim_y > 1) ∧ (symmetric_plasma ≡ TRUE)) then
    do iy = 1, dim_y
      iy_zone = zone_iy_mapm(iy, zone)
      if ((zn_check(iy_zone)) ∧ (zn_type(iy_zone) ≡ zn_plasma)) then
        bk_temp(back, iy_zone) = bk_temp(back, zone)
      end if
    end do
  end if
  if (next_token(line, b, e, p)) then
    goto temperature_loop
  else
    goto next_line
  end if

else if ((data_code ≡ code_velocity_1) ∨ (data_code ≡ code_velocity_2) ∨ (data_code ≡
velocity_loop: continue
  count++
  if (zn_check(zone_list_count)) then
    zone = zone_list_count
  else
    zone = count
  end if
  assert(zn_check(zone))
  assert((zn_index(zone, zi_ptr) ≡ zone) ∨ (symmetric_plasma ≡ FALSE))
  assert(zn_type(zone) ≡ zn_plasma)
  assert(bk_check(back))
  if (data_code ≡ code_velocity_1) then
    v_index = 1
  else if (data_code ≡ code_velocity_2) then
    v_index = 2

```

```

else
  v_index = 3
end if
assert((v_index ≥ 1) ∧ (v_index ≤ 3))
bk_v(back, zone)v_index = read_real(line(b : e))
if ((dim_y > 1) ∧ (symmetric_plasma ≡ TRUE)) then
  do iy = 1, dim_y
    iy_zone = zone_iy_mapm(iy, zone)
    if ((zn_check(iy_zone)) ∧ (zn_type(iy_zone) ≡ zn_plasma)) then
      bk_v(back, iy_zone)v_index = bk_v(back, zone)iy_zone
    end if
  end do
end if
if (next_token(line, b, e, p)) then
  goto velocity_loop
else
  goto next_line
end if

else
  assert(' Unexpected value of data_code ' ≡ ' ')
end if // data_code
end if // string content
goto next_line

end if // string presence
close(unit = diskin3)

return
end

```

## 12 Parse a single label for row-oriented data

This routine is analogous to *parse\_format*, but intended for use with row-oriented data. The line containing the label is passed in. A code describing the data is passed back, along with the background index, if applicable.

⟨Functions and subroutines 7⟩ +≡

```

subroutine parse_label(line, data_code, back)

    implicit_none_f77
    bk_common    // Common
    implicit_none_f90

    character*LINELEN line    // Input
    integer data_code    // Output
    bk_decl(back)

    integer p, b, e, p2, b2, e2, b_code    // Local

    ⟨Memory allocation interface 0⟩
    st_decls

    data_code = code_undefined
    back = int_undef
    p = 0
    assert(next_token(line, b, e, p))
    if ((line(b : e) ≡ 'zone') ∨ (line(b : e) ≡ 'ZONE') ∨ (line(b : e) ≡ 'Zone')) then
        data_code = code_zone
        back = int_undef
    else if ((line(b : e) ≡ 'stratum') ∨ (line(b : e) ≡ 'STRATUM') ∨ (line(b : e) ≡ 'Stratum')) then
        data_code = code_stratum
        back = int_undef
    else if ((line(b : e) ≡ 'segment') ∨ (line(b : e) ≡ 'SEGMENT') ∨ (line(b : e) ≡ 'Segment')) then
        data_code = code_segment
        back = int_undef
    else if ((line(b : e) ≡ 'segment_iy') ∨ (line(b : e) ≡ 'SEGMENT_IY') ∨ (line(b : e) ≡ 'Segment_iy'))
        then
            data_code = code_segment_iy
            back = int_undef
    else if ((line(b : e) ≡ 'area') ∨ (line(b : e) ≡ 'AREA') ∨ (line(b : e) ≡ 'Area')) then
        data_code = code_area
        back = int_undef
    else if ((line(b : b) ≡ 'n') ∨ (line(b : b) ≡ 'N')) then
        data_code = code_density
    else if ((line(b : b) ≡ 'f') ∨ (line(b : b) ≡ 'F')) then    // "Flux"
        data_code = code_strength
    else if ((line(b : b) ≡ 't') ∨ (line(b : b) ≡ 'T')) then
        data_code = code_temperature_eV    /* Note the use of a second set of string indices so as to not
            mess up the ones that will be used below to read the background index. */
        p2 = p
        if (next_token(line, b2, e2, p2)) then
            if (line(b2 : e2) ≡ 'J') then
                data_code = code_temperature_J
            else if (line(b2 : e2) ≡ 'eV') then

```

```

        data_code = code_temperature_eV
    else
        write(stderr, *) 'Unexpected token, ', line(b2 : e2)
        assert( $\mathcal{F}$ )
    end if
end if
else if ((line(b : b)  $\equiv$  'v')  $\vee$  (line(b : b)  $\equiv$  'V')) then
    if (line(b + 1 : b + 1)  $\equiv$  '1') then
        data_code = code_velocity_1
    else if (line(b + 1 : b + 1)  $\equiv$  '2') then
        data_code = code_velocity_2
    else if (line(b + 1 : b + 1)  $\equiv$  '3') then
        data_code = code_velocity_3
    else if ((line(b + 1 : b + 4)  $\equiv$  '_par')  $\vee$  (line(b + 1 : b + 4)  $\equiv$  '_PAR')  $\vee$  (line(b + 1 : b + 4)  $\equiv$  '_Par'))
        then
            data_code = code_velocity_par
        else
            write(stderr, *) 'Illegal format token, ', line(b : e)
            assert( $\mathcal{F}$ )
        end if
    end if
    /* Stuff ending up here represents further analysis of the label. Look for the background
    number. This expects the background number to be a single digit. See the comment in
    parse_format regarding this convoluted FWEB code. */
    if (data_code  $\neq$  code_undefined) then
        b_code = $TRANSLIT(index(line(b : e)C'(', 'C', ', ', ') + 1
        if (b_code > 1) then
            b_code = b + b_code - 1 // Relate to start of whole line
            back = read_int_soft_fail(line(b_code : b_code))
            assert(bk_num < 10)
        end if
    end if
end if

return
end

```

## 13 INDEX

- abs: 9.
- aname: 7.
- area: 7, 9.
- arg\_count: 6.
- assert: 6, 7, 8, 9, 10, 11, 12.
- aunit: 7.
- b: 7, 10, 11, 12.
- b\_code: 10, 12.
- b\_repeat: 10.
- back: 7, 8, 11, 12.
- back\_label: 7.
- background\_coords: 7.
- bi: 10.
- bk\_check: 11.
- bk\_common: 7, 11, 12.
- bk\_decl: 7, 11, 12.
- bk\_n: 7, 11.
- bk\_num: 7, 10, 12.
- bk\_temp: 7, 9, 11.
- bk\_v: 11.
- boltzmanns\_const: 7.
- b2: 12.
- cartesian: 1.
- char\_undef: 7.
- circum: 7, 9.
- code\_all\_segs: 7, 9.
- code\_area: 7, 9, 12.
- code\_density: 7, 9, 11, 12.
- code\_e\_delta: 7, 9.
- code\_e\_mult: 7, 9.
- code\_e\_sheath: 7, 9.
- code\_electron\_temperature: 7, 8, 9.
- code\_int: 7, 8.
- code\_int\_max: 7.
- code\_int\_min: 7.
- code\_multiplicity: 7, 9.
- code\_real: 7, 8.
- code\_real\_max: 7.
- code\_real\_min: 7.
- code\_segment: 7, 9, 12.
- code\_segment\_iy: 7, 9, 12.
- code\_stratum: 7, 9, 12.
- code\_strength: 7, 9, 12.
- code\_temperature\_eV: 7, 8, 11, 12.
- code\_temperature\_J: 7, 8, 9, 11, 12.
- code\_undefined: 7, 8, 11, 12.
- code\_velocity\_par: 7, 9, 12.
- code\_velocity\_1: 7, 9, 11, 12.
- code\_velocity\_2: 7, 9, 11, 12.
- code\_velocity\_3: 7, 9, 11, 12.
- code\_zone: 7, 11, 12.
- command\_arg: 6.
- const: 7, 9.
- count: 10, 11.
- cylindrical: 1.
- data\_code: 7, 8, 11, 12.
- data\_type: 7, 8.
- declare\_varp: 7, 11.
- define\_dimen: 7, 11.
- define\_varp: 7, 11.
- defineback: 1, 6.
- defnegeometry2d: 1.
- def2dplasma: 1.
- density: 7.
- density\_loop: 11.
- dim\_segments: 7.
- dim\_y: 11.
- diskin: 7, 11.
- diskin2: 7, 8.
- diskin3: 11.
- dummy: 7.
- e: 7, 10, 11, 12.
- electron\_charge: 7, 8, 11.
- end\_source\_group: 1.
- eof: 10.
- erase\_geometry: 6.
- e2: 12.
- FALSE: 7, 9, 10, 11.
- file: 7.
- FILE: 1.
- file\_format: 7.
- FILELEN: 6, 7, 11.
- float: 9.
- FLOAT: 7.
- form: 7.
- form\_density: 10.
- form\_temperature: 10.
- form\_undefined: 10.
- form\_velocity\_1: 10.
- form\_velocity\_2: 10.
- form\_velocity\_3: 10.
- form\_zone: 10.
- format\_code: 10.
- format\_max: 10.
- format\_string: 10.
- ftangle: 10.
- geom: 7, 9.
- geometry\_symmetry: 1, 7, 9, 11.

*geometry\_symmetry\_cyl\_hw*: 1, 7, 9, 11.  
*geometry\_symmetry\_cyl\_section*: 1, 7, 9, 11.  
*geometry\_symmetry\_cylindrical*: 7, 9.  
*geometry\_symmetry\_oned*: 7.  
*geometry\_symmetry\_plane*: 7, 9.  
*geometry\_symmetry\_plane\_hw*: 1, 7, 9, 11.  
*get\_n\_t*: 1, 7.  
*get\_n\_t\_row*: 7, 11.  
*gi\_common*: 7, 11.  
*grp*: 7.  
*g2\_polygon\_stratum*: 1.  
*g2\_polygon\_zone*: 1.  
*half*: 9.  
*i*: 7, 10.  
*implicit\_none\_f77*: 6, 7, 10, 11, 12.  
*implicit\_none\_f90*: 6, 7, 10, 11, 12.  
*increment\_iseg*: 7, 8.  
*index*: 10, 12.  
*init\_wt\_alias*: 7.  
*input\_done*: 7.  
*inputfile*: 6, 7.  
*INT*: 7, 11.  
*int\_params\_ind*: 7.  
*int\_undef*: 7, 8, 9, 10, 11, 12.  
*int\_uninit*: 7, 8, 9.  
*int\_unused*: 7.  
*iostat*: 7.  
*iseg*: 7, 8, 9.  
*iterative*: 7, 9.  
*iterative\_solution*: 1.  
*itest*: 7, 8, 11.  
*ix*: 1.  
*iy*: 9, 11.  
*iy\_zone*: 11.  
*iy\_zone\_ind*: 11.  
*iys\_done*: 7.  
*iz*: 1.  
  
*j*: 10.  
*jr*: 7.  
  
*len*: 7, 8, 11.  
*length*: 7, 8, 10, 11.  
*line*: 7, 8, 10, 11, 12.  
*LINELEN*: 7, 10, 11, 12.  
  
*max*: 11.  
*mem\_check*: 6.  
*mult*: 7, 9.  
  
*n\_iy*: 11.  
*nargs*: 6.  
*nc\_read\_elements*: 6.  
  
*nc\_read\_materials*: 6.  
*nc\_read\_old\_sources*: 7.  
*nc\_read\_pmi*: 6.  
*nc\_read\_problem*: 6.  
*nc\_read\_reactions*: 6.  
*nc\_read\_species*: 6.  
*ncsxplasma*: 1.  
*neutralize\_species*: 9.  
*new\_source\_group*: 1.  
*next\_datum*: 7, 8.  
*next\_iy*: 7.  
*next\_line*: 7, 11.  
*next\_rowline*: 7, 8.  
*next\_segment*: 7.  
*next\_stratum*: 7.  
*next\_strength*: 7.  
*next\_tabline*: 7.  
*next\_token*: 7, 8, 10, 11, 12.  
*nflights*: 7, 9.  
*nsectors*: 9.  
*nt\_string*: 7, 11.  
*num\_segments*: 7, 8, 9.  
*nunit\_n*: 7.  
*nunit\_t*: 7.  
*nz*: 1.  
  
*old\_current*: 7.  
*old\_grps*: 7.  
*old\_rel\_wt*: 7.  
*old\_seg\_tot*: 7.  
*old\_tot\_curr*: 7.  
*oldsourcefile*: 1.  
*one*: 8, 9, 11.  
*open\_file*: 7, 11.  
*open\_stat*: 7, 11.  
*outputbrowser*: 7.  
  
*p*: 7, 10, 11, 12.  
*parse\_format*: 10, 12.  
*parse\_label*: 8, 11, 12.  
*parse\_string*: 7, 8, 10, 11.  
*phi\_sheath*: 7, 9.  
*PI*: 9.  
*plasma\_coords*: 1.  
*plasma\_coords\_cartesian*: 7.  
*plasma\_coords\_cylindrical*: 7.  
*plasma\_e\_ion\_delta*: 1, 9.  
*plasma\_e\_ion\_mult*: 1, 9.  
*plasma\_e\_ion\_sheath*: 9.  
*plasma\_file*: 1, 11.  
*plate*: 1.  
*pr\_background*: 8, 9.  
*pr\_bkrc\_num*: 7.

*pr\_common*: 7.  
*puff*: 1.  
*puff\_temp*: 7, 9.  
*p2*: 12.  
  
*read\_geometry*: 6.  
*read\_int\_soft\_fail*: 8, 10, 11, 12.  
*read\_integer*: 7, 8, 11.  
*read\_real*: 7, 8, 11.  
*read\_real\_soft\_fail*: 8, 11.  
*read\_string*: 7, 8, 11.  
*readfilenames*: 6.  
*real\_mult*: 7, 8.  
*real\_params\_ind*: 7.  
*real\_undef*: 8, 11.  
*real\_uninit*: 7, 9.  
*right*: 1.  
*root\_sp*: 7, 9.  
*row*: 11.  
*rtest*: 7, 8, 11.  
  
*sc\_common*: 7.  
*sc\_decl*: 7.  
*sc\_neg*: 9.  
*sc\_plasma*: 9.  
*sc\_plasma\_check*: 9.  
*sc\_pos*: 9.  
*sc\_vacuum*: 9.  
*sc\_vacuum\_check*: 9.  
*sector*: 1, 7, 9.  
*sector\_points*: 1, 9.  
*sector\_strata\_segment*: 1, 9.  
*sector\_type\_pointer*: 9.  
*sector\_zone*: 9.  
*segment\_ind*: 7.  
*segments\_done*: 7.  
*set\_background\_rate*: 7.  
*set\_background\_sources*: 7.  
*setup\_back\_arrays*: 6.  
*setup\_background*: 6, 7.  
*so\_base*: 7, 9.  
*so\_chkpt*: 9.  
*so\_common*: 7.  
*so\_geom*: 9.  
*so\_grps*: 1, 7, 9.  
*so\_line*: 7.  
*so\_name*: 7.  
*so\_nflights*: 9.  
*so\_nseq*: 9.  
*so\_param\_puff\_temp*: 9.  
*so\_params*: 9.  
*so\_plate*: 9.  
*so\_point*: 7.  
  
*so\_rel\_wt\_max*: 1.  
*so\_rel\_wt\_min*: 1.  
*so\_root\_sp*: 9.  
*so\_scale*: 9.  
*so\_seg\_tot*: 7, 9.  
*so\_set\_run\_flags*: 7.  
*so\_species*: 9.  
*so\_surface*: 7.  
*so\_tot\_curr*: 7, 9.  
*so\_type*: 9.  
*so\_type\_num*: 7.  
*so\_volume*: 7.  
*source\_base\_ptr*: 7, 9.  
*source\_current*: 7, 9.  
*source\_file*: 1, 7.  
*source\_geom*: 1.  
*source\_geometry*: 7, 9.  
*source\_grp\_ind*: 7.  
*source\_int\_params*: 7, 8, 9.  
*source\_nflights*: 1.  
*source\_num\_checkpoints*: 7, 9.  
*source\_num\_flights*: 7, 9.  
*source\_num\_segments*: 7, 9.  
*source\_parameters*: 7, 9.  
*source\_puff\_temp*: 1.  
*source\_real\_params*: 7, 8, 9.  
*source\_root\_sp*: 1.  
*source\_root\_species*: 7, 9.  
*source\_scale\_factor*: 7, 9.  
*source\_seg\_ind*: 7.  
*source\_segment*: 1.  
*source\_segment\_iy*: 1.  
*source\_segment\_prob\_alias*: 7, 9.  
*source\_segment\_ptr*: 7, 9.  
*source\_segment\_ptr\_alias*: 7, 9.  
*source\_segment\_rel\_wt*: 7, 9.  
*source\_species*: 1, 7, 9.  
*source\_stratum*: 1.  
*source\_strength*: 1.  
*source\_total\_current*: 7, 9.  
*source\_type*: 1, 7, 9.  
*source\_weight\_norm*: 7, 9.  
*sourcetest*: 1.  
*sp\_check*: 7, 9.  
*sp\_common*: 7.  
*sp\_decl*: 7.  
*sp\_lookup*: 7.  
*sp\_sy*: 8, 9.  
*species*: 7, 9.  
*species\_infile*: 1.  
*specify\_current*: 1.  
*specify\_flux*: 1, 7, 9.

*st\_decls*: 7, 10, 11, 12.  
*status*: 7.  
*stderr*: 10, 12.  
*stdout*: 9.  
*strata*: 1, 9.  
*strata\_done*: 7.  
*stratum*: 1.  
*strengths\_done*: 7.  
*sub\_count*: 10.  
*sy\_decls*: 6.  
*symmetric\_plasma*: 11.  
  
*targ*: 7, 9.  
*targ\_v*: 7, 9.  
*temperature*: 7.  
*temperature\_loop*: 11.  
*temperature\_mult*: 11.  
*token\_loop*: 10.  
*TRUE*: 7, 9, 10, 11.  
*type*: 7, 9.  
*type\_string*: 7.  
  
*uedge*: 7.  
*unit*: 7, 8, 11.  
*universal\_cell\_max*: 9.  
*universal\_cell\_min*: 9.  
*update*: 7, 9.  
*update\_wt\_alias*: 7, 9.  
*usr2dplasma*: 1.  
  
*v\_index*: 11.  
*var\_alloc*: 7, 11.  
*var\_free*: 7.  
*var\_realloc*: 7.  
*var\_realloca*: 7, 9.  
*var\_reallocb*: 7.  
*vc\_abs*: 9.  
*vc\_args*: 9.  
*vc\_decl*: 7.  
*vc\_difference*: 9.  
*vc\_set*: 9.  
*vec*: 1.  
*velocity\_loop*: 11.  
  
*web*: 1.  
*write\_background*: 7.  
*write\_geometry*: 7.  
  
*xdiff*: 7, 9.  
  
*zero*: 7, 9.  
*zi\_iz*: 9.  
*zi\_ptr*: 1, 11.  
*zn\_check*: 11.  
*zn\_common*: 7, 11.  
  
*zn\_decl*: 7, 11.  
*zn\_index*: 1, 9, 11.  
*zn\_num*: 7, 11.  
*zn\_plasma*: 7, 11.  
*zn\_type*: 7, 11.  
*zone*: 1, 7, 9, 11.  
*zone\_ind*: 11.  
*zone\_iz\_map*: 11.  
*zone\_iz\_mapm*: 11.  
*zone\_list*: 11.  
*zone\_loop*: 11.  
*zone\_max*: 9.  
*zone\_min*: 9.  
*zone\_stratum*: 1.

⟨Functions and subroutines 7, 10, 11, 12⟩ Used in section 6.  
⟨Memory allocation interface 0⟩ Used in sections 12, 11, 10, and 7.  
⟨Read Row Source File 8⟩ Used in section 7.  
⟨Setup Source Group 9⟩ Used in section 7.

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

**WEB FILE:** "/u/dstotler/degas2/src/defineback.web".

**CHANGE FILE:** (none).

**GLOBAL LANGUAGE:** FORTRAN.