defineback

May 8, 2013
13:44

Contents

1 A Program for Defining 2-D Plasmas **1**

2 Example Input File **2**

3 Plasma Data **3**

4 Source Group Data **7**

5 Iterative Problems **13**

6 Time Dependence **14**

7 The Main Program **15**

8 Main Subroutine **16**

9 Read a row-oriented source file **28**

10 Setup a source group **30**

11 Calculate the current for a surface source from the flux **37**

12 Assign sheath parameters for a plate source **38**

13 Parse tabular plasma data header **39**

14 Read row-oriented plasma data **42**

15 Parse a single label for row-oriented data **47**

16 INDEX **49**

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).

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 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 we 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 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.

The minimum specification of a volumetric source, *source_type* `vol_source`, requires a zone, a strength, and a temperature; a drift velocity can also be specified. For this reason, its parameters must appear in an external *source_file* in row format with labels analogous to those of a plasma data file. No species or "back" index is required since a source by definition applies to only one species. Because a zone number completely determines the location of the source in the problem, *stratum*, *segment*, or *segment_iy* values are not needed.

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`, `vol_source`, `plt_e_bins`, `snapshot` and `recomb`. The last two are 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. A snapshot source is set up automatically in a time dependent run if a *snapshotfile*, generated at the end of the previous step, is present.

The `plt_e_bins` source type allows the user to specify the energy distribution for the recycling ions via the additional parameters *source_e_bin_num*, *source_e_bin_min*, *source_e_bin_max*, and *source_e_bin_spacing*. The four parameters only describe the grid on which this distribution is specified. Since this energy distribution can vary from one segment to the next within this source group, the total number of parameters required to specify all of these distributions can be quite large. As such, they can only be input via the `row` option of *source_file* (`E_bin_prob` and variants).

source_geom `geom` specifies the geometry of the source. The most frequently used value is `geom` = `surface` (e.g., a line in a poloidal plane of an axisymmetric torus). Obviously, for a volume source `geom` = `volume`. Note that there is no default set, so *source_geom* must be included with each source group.

time_interval $t_{\rm init}$ $t_{\rm final}$ specifies a time dependent run from $t_{\rm init}$ to $t_{\rm final}$ (in seconds) provided $t_{\rm final} > t_{\rm init}$. If this keyword is not present, the run is assumed to be steady-state.

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` or `vol_source` source, this would be the same as *source_species*. The code currently requires that *source_root_sp* be specified.

source_nflights $n_{\rm flights}$ 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.

source_puff_exponent α specifies the peakedness of the angular distribution of a gas puff. Namely, the puffed particle velocities will have angles θ, the angle relative to the surface normal, distributed according to $\cos^\alpha(\theta)$. The default value is 1, although values as large as 2.5 are reasonable [e.g., see B. Farizon et al., *Nucl. Instrum. Meth. Phys. Res. B* **101** (1995) 287].

source_e_bin_num $n_{\rm bins}$ is the number of bins used to specify the energy distribution at each source group segment in the `plt_e_bins` source type. The default is 4; the maximum value is *so_e_bins_num_max*.

source_e_bin_min $E_{\rm min}$ is the minimum energy in eV of the energy distribution to be specified with the `plt_e_bins` source type. This value represents the "left edge" of the first bin. The default value is $E_{\rm min} = 1$ eV.

source_e_bin_max $E_{\rm max}$ is the maximum energy in eV of the energy distribution to be specified with the `plt_e_bins` source type. This value represents the "right edge" of the $n_{\rm bins}$-th bin. The default value is $E_{\rm min} = 10^4$ eV.

source_e_bin_spacing `spacing` specifies the spacing of the energy distribution used with the `plt_e_bins` source type. The only supported values are: `spacing` = `linear`, which results in an energy distribution evenly spaced over $n_{\rm bins}$ between $E_{\rm min}$ and $E_{\rm max}$, and `spacing` = `log`, which results in an energy distribution evenly spaced over $n_{\rm bins}$ between $\ln(E_{\rm min})$ and $\ln(E_{\rm max})$. The default value is `spacing` = `log`.

source_time_variation `varn` specifies the time dependence of the source in time dependent runs. The only supported values are: `varn` = `delta` causes all source particles to be launched at time $t_{\rm init}$ (see *time_interval* above), and `varn` = `uniform` causes the initial times of the source particles to be uniformly distributed between $t_{\rm init}$ and $t_{\rm final}$. The default value is `delta`. More complex time variation can be obtained by discretizing the time variation of the source into a sequence of intervals, each with approximately constant strength.

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. For a volume source, it's particles per cubic meter per second. In time dependent runs, this just becomes particles per square meter or per cubic meter.

specify_current informs the code that the *source_strength* will represent a current of particles (particles per second). In time dependent runs, this is just particles.

source_stratum i_1 i_2 i_3 ... 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 ... 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 ... 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 ... 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 format` 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. The optional **format** argument should be set to `tabular` or `row`, respectively; the default is `tabular`.

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_{stratum} i_{segment} S_{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_{stratum}, i_{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 "surface" geometry 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).

A volumetric source (*source_type vol_source*) is simpler, needing only a zone (labels: `zone`, `ZONE`, or `Zone`), a strength (labels `f` or `F`; the meaning again determined by which of *specify_flux* and *specify_current* is in effect), a temperature (labels `t` or `T`), and perhaps a drift velocity (e.g., `v1` or `V1`). No stratum, segment, or segment i_y are required.

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 it so happens, e.g. in testing, $T_e \leq 0$, then *plasma_e_ion_delta* is set to 3 eV.

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

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

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

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).

Binned Energy Distribution Note first that this energy distribution characterizes the particles as they strike a material surface, not at the sheath entrance. I.e., the particles will not be accelerated through the sheath, and there is no need to specfiy parameters for the sheath.

The only quantity specified in the *source_file* is the relative probability of the distribution over the $n_{\rm bin}$ (*source_e_bin_num*) bins, labeled with `E_bin_prob` (or `E_BIN_PROB`, `e_bin_prob`). From this distribution, *defineback* will compute a normalized, cumulative distribution suitable for sampling during the run of the main code.

The primary difference between this and other quantities in the *source_file* is that there are $n_{\rm bins}$ parameters for each source segment. These will be read in from the file with the first $n_{\rm bins}$ numbers assigned to the first segment, the second $n_{\rm bins}$ to the second segment, etc. As with the other quantities, the data are read in one row at a time. The user is otherwise free to arrange these data in whatever manner is convenient or most readable.

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 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., $strata_{sector}$). This corresponds to the "stratum" value specified in the input file.

stratum segment Is the stratum segment label of the sector (i.e., $sector_strata_segment_{sector}$). 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.

6 Time Dependence

The default and most frequently used mode of running DEGAS 2 simulates a steady state or time independent system. The reason for this is physically obvious: unconstrained by magnetic field lines, neutral particles can travel through a plasma much more quickly than charged species, and in many cases, the neutral profile equilibrates on a time scale short compared with global transport times. For a rapidly evolving system, however, a time dependent calculation may be needed. Only two keywords are required in the *defineback* input file to enable time dependent operation: *time_interval* specifies the initial and final times for the solution interval, and *source_time_variation* specifies the manner in which the source(s) varies in time. Note also that the strength of the source in a time dependent run represents a number of particles (*specify_current*) or particles per square meter (*specify_flux*).

A typical time dependent application would be an explicit, synchronized coupling with a plasma simulation code. At each step, the output from the plasma code would be input to *defineback*, which would then set up a time dependent run of DEGAS 2, presumably over the same time interval. The DEGAS 2 results can either be computed at the end of interval (via a "snapshot" estimator) or averaged over it (all other estimators); see *tallysetup*.

Regardless, at the end of the interval, the positions, velocities, and weights of all test particles in the vessel are written out to the snapshot netCDF file (*sn* class; the name of the "snapshotfile" must appear in the *degas2.in* file). If this file is present for a subsequent run of *defineback*, a new source group is automatically created to sample from these particles in the next run of DEGAS 2. The corresponding data are incorporated into the background netCDF file by *defineback*, potentially resulting in a much larger file. *defineback* will compare the time associated with these particles with the value of $t_{\rm init}$ and print out a warning if they differ by more than roundoff error. The number of flights for this source group defaults to 100. The appropriate value for a particular run depends on the strength of the snapshot source relative to the other sources in the run. E.g., if they are comparable, the number of flights used should be as well.

```
$Id: b16091b79707692e8aa43ba810e1ae636ccde8a5 $
```

"defineback.f" 1 ≡

```
@m FILE 'defineback.web'
```

7 The Main Program

```
"defineback.f" 7 ≡
    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 8⟩
```

8 Main Subroutine

"defineback.f" 8 ≡

```
@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_e_bin_min 24    // Needs to be the last code since so_e_bins_num_max come after it.
@m code_real_max code_e_bin_min + so_e_bins_num_max − 1

@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

@m increment_gparams(num_params) so_gparams_list_size += num_params
  if (so_gparams_list_size > so_gparams_list_dim) then
    var_realloc(source_gparameters_list, so_gparams_list_dim, so_gparams_list_size)
    var_realloc(source_gparameters_data, so_gparams_list_dim, so_gparams_list_size)
    so_gparams_list_dim = so_gparams_list_size
  end if

@m increment_giparams(num_params) so_giparams_list_size += num_params
  if (so_giparams_list_size > so_giparams_list_dim) then
    var_realloc(source_giparameters_list, so_giparams_list_dim, so_giparams_list_size)
    var_realloc(source_giparameters_data, so_giparams_list_dim, so_giparams_list_size)
    so_giparams_list_dim = so_giparams_list_size
  end if

@m increment_params(num_params) so_params_list_size += num_params
  if (so_params_list_size > so_params_list_dim) then
    var_realloc(source_parameters_list, so_params_list_dim, so_params_list_size)
    so_params_list_dim = so_params_list_size
  end if

@m sector_segment_test(sector, iseg) (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))
```

⟨Functions and subroutines 8⟩ ≡

```
  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, puff_exponent, e_bin_min, e_bin_max, e_bin_spacing, total, cumul
integer type, geom, nflights, length, p, b, e, nunit_n, specify_flux, nunit_t, num_segments, iseg,
      i, open_stat, targ, diskin2, dim_segments, itest, data_type, data_code, update, old_grps,
      old_seg_tot, grp, iterative, jr, iparam, time_varn, standalone, e_bin_num, i_bin, i_code,
      e_bin_entry

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

  so_gparams_list_size = 0
  so_gparams_list_dim = 1
  so_params_list_size = 0
  so_params_list_dim = 1
  so_params_data_size = 0
  so_params_data_dim = 1
  so_giparams_list_size = 0
  so_giparams_list_dim = 1
```

```
        so_iparams_list_size = 0
        so_iparams_list_dim = 1
        so_iparams_data_size = 0
        so_iparams_data_dim = 1
        var_alloc(source_gparameters_list)
        var_alloc(source_parameters_list)
        var_alloc(source_gparameters_data)
        var_alloc(source_parameters_data)
        var_alloc(source_giparameters_list)
        var_alloc(source_iparameters_list)
        var_alloc(source_giparameters_data)
        var_alloc(source_iparameters_data)
    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) ≡ 'time_interval') then
    assert(next_token(line, b, e, p))
    so_time_initial = read_real(line(b:e))    /* For now assuming that the interval consists of an
          initial and final time pair. May eventually allow for just the specification of a time step. */
    assert(next_token(line, b, e, p))
    so_time_final = read_real(line(b:e))
    if (so_time_final > so_time_initial) then
      so_time_dependent = TRUE
    else
      so_time_dependent = FALSE
    end if

  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
    puff_exponent = one
    e_bin_num = 4
    e_bin_min = one * electron_charge
    e_bin_max = const(1., 4) * electron_charge
    e_bin_spacing = so_e_bins_spacing_log
    nflights = 100
    specify_flux = TRUE
```

```
num_segments = int_uninit
time_varn = so_delta_fn
  /* 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_iseg). */
if (dim_segments > 0) then
  do iseg = 1, dim_segments
    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 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) ≡ 'source_puff_exponent') then
  assert(next_token(line, b, e, p))
  puff_exponent = read_real(line(b : e))
  assert(puff_exponent > const(−1.0))
else if (line(b : e) ≡ 'source_e_bin_num') then
  assert(next_token(line, b, e, p))
  e_bin_num = read_integer(line(b : e))
  assert((e_bin_num > 0) ∧ (e_bin_num ≤ so_e_bins_num_max))
else if (line(b : e) ≡ 'source_e_bin_min') then
  assert(next_token(line, b, e, p))
  e_bin_min = read_real(line(b : e)) ∗ electron_charge
  assert(e_bin_min > zero)
else if (line(b : e) ≡ 'source_e_bin_max') then
  assert(next_token(line, b, e, p))
  e_bin_max = read_real(line(b : e)) ∗ electron_charge
  assert(e_bin_max > zero)
else if (line(b : e) ≡ 'source_e_bin_spacing') then
  assert(next_token(line, b, e, p))
  if (line(b : e) ≡ 'linear') then
    e_bin_spacing = so_e_bins_spacing_linear
  else if (line(b : e) ≡ 'log') then
    e_bin_spacing = so_e_bins_spacing_log
  else
    assert('␣Unexpected␣e_bin_spacing␣option␣' ≡ '␣')
  end if
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 (¬next_token(line, b, e, p)) then
    if (num_segments ≡ int_uninit) then
      num_segments = iseg
    else
      assert(num_segments ≡ iseg)
    end if
    goto strata_done
  else
    increment_iseg
```

```
        source_int_params_{iseg,code_stratum} = read_integer(line(b : e))
      end if
      goto next_stratum
strata_done: continue
    else if (line(b : e) ≡ 'source_segment') then
      iseg = 0
next_segment: 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 segments_done
      else
        increment_iseg
        if (line(b : e) ≡ '*') then
          source_int_params_{iseg,code_all_segs} = TRUE
          source_int_params_{iseg,code_segment} = int_unused
        else
          source_int_params_{iseg,code_all_segs} = FALSE
          source_int_params_{iseg,code_segment} = read_integer(line(b : e))
        end if
      end if
      goto next_segment
segments_done: continue
    else if (line(b : e) ≡ '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 (¬next_token(line, b, e, p)) then
        if (num_segments ≡ int_uninit) then
          num_segments = iseg
        else
          assert(num_segments ≡ iseg)
        end if
        goto iys_done
      else
        increment_iseg
        source_int_params_{iseg,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_params_{iseg,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)
          diskin2 = diskin + 1
          open_file(diskin2, 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(diskin2, 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_params_{iseg,code_stratum} = read_integer(line(b : e))
              assert(next_token(line, b, e, p))
              source_int_params_{iseg,code_segment} = read_integer(line(b : e))
              assert(next_token(line, b, e, p))
              source_real_params_{iseg,code_strength} = read_real(line(b : e))
              goto next_tabline
            end if
            assert(num_segments ≡ int_uninit)
            num_segments = iseg
            close(unit = diskin2)
          else if (file_format ≡ 'row') then
            ⟨ Read Row Source File 9 ⟩
          end if

        else if (line(b : e) ≡ 'source_time_variation') then
          assert(next_token(line, b, e, p))
          if (line(b : e) ≡ 'delta') then
            time_varn = so_delta_fn
          else if (line(b : e) ≡ 'uniform') then
            time_varn = so_time_uniform
          else
```

```
        assert('Unsupported␣time␣variation' ≡ line(b : e))
      end if
    else if (line(b : e) ≡ 'end_source_group') then
      ⟨ Setup Source Group 10 ⟩
    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 = diskin + 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_time_variation)
@#if 0
```

```
    var_reallocb(source_parameters)
@#else
    var_reallocb(source_num_gparameters)
    var_reallocb(source_num_parameters)
    var_reallocb(source_gparameters_base)
    var_reallocb(source_parameters_base)
    var_reallocb(source_parameters_data_base)
    var_reallocb(source_num_giparameters)
    var_reallocb(source_num_iparameters)
    var_reallocb(source_giparameters_base)
    var_reallocb(source_iparameters_base)
    var_reallocb(source_iparameters_data_base)
@#endif
    var_reallocb(source_total_current)
    var_reallocb(source_weight_norm)
    var_reallocb(source_scale_factor)
  end if

  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
    if (so_time_dependent ≡ TRUE) then
      standalone = TRUE
      call set_snapshot_source(standalone)
    end if
  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

  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 13, 14, and 15.

This code is used in section 7.

9 Read a row-oriented source file

⟨ Read Row Source File 9 ⟩ ≡

```
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
        e_bin_entry = FALSE   /* 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 (bk_check(back)) then
            if (sp_sy(pr_background(back)) ≡ 'e')
              data_code = code_electron_temperature
          end if
        end if

        goto next_rowline
      else   /* Contains a line of data */
        assert(data_code ≠ code_undefined)
next_datum: continue

        if (e_bin_entry ≡ FALSE) then
```

```
        increment_iseg
      end if

      if (data_type ≡ code_int) then
        source_int_params_{iseg,data_code} = read_integer(line(b : e))
      else if (data_type ≡ code_real) then
        source_real_params_{iseg,data_code} = read_real(line(b : e)) * real_mult    /* There are e_bin_num
            probabilities for each segment with the binned energy distribution. The first entry is
            handled in the usual manner as indicated above. We suppress incrementing of iseg until
            all of the bins have been read in. The resetting of data_code to code_e_bin_min restarts
            the process. At that same point, the input distribution is converted into the normalized,
            cumulative distribution needed for sampling. */
        if (data_code ≥ code_e_bin_min) then
          e_bin_entry = TRUE
          if (data_code < code_e_bin_min + e_bin_num − 1) then
            data_code++
          else
            e_bin_entry = FALSE    /* Compute normalized cumulative energy distribution. */
            total = zero
            do i_bin = 1, e_bin_num
              i_code = code_e_bin_min + i_bin − 1
              total += source_real_params_{iseg,i_code}
            end do
            assert(total > zero)
            cumul = zero
            do i_bin = 1, e_bin_num
              i_code = code_e_bin_min + i_bin − 1
              cumul += source_real_params_{iseg,i_code} / total
              source_real_params_{iseg,i_code} = cumul
            end do
            assert(source_real_params_{iseg,data_code} ≡ one)
            data_code = code_e_bin_min    // Reset to trigger increment of iseg
          end if
        end if
      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 8.

10 Setup a source group

⟨ Setup Source Group 10 ⟩ ≡

```
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_|| since we
    cannot convert between v⃗ components and v_|| 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)
```

$var_realloca(source_num_flights)$
$var_realloca(source_num_checkpoints)$
$var_realloca(source_species)$
$var_realloca(source_root_species)$
$var_realloca(source_time_variation)$

$var_realloca(source_num_gparameters)$
$var_realloca(source_num_parameters)$
$var_realloca(source_gparameters_base)$
$var_realloca(source_parameters_base)$
$var_realloca(source_parameters_data_base)$
$var_realloca(source_num_giparameters)$
$var_realloca(source_num_iparameters)$
$var_realloca(source_giparameters_base)$
$var_realloca(source_iparameters_base)$
$var_realloca(source_iparameters_data_base)$

$var_realloca(source_total_current)$
$var_realloca(source_weight_norm)$
$var_realloca(source_scale_factor)$
$so_base(so_grps) = so_seg_tot + 1$
$assert(type \neq int_undef)$
$so_type(so_grps) = type$
$assert(geom \neq int_undef)$
$so_geom(so_grps) = geom$
$so_nflights(so_grps) = nflights$
$so_chkpt(so_grps) = 0$
$assert(sp_check(species))$
$so_species(so_grps) = species$ // Use *neutralize_species*?
$assert(sp_check(root_sp))$
$so_root_sp(so_grps) = root_sp$
$so_t_varn(so_grps) = time_varn$

$source_num_gparameters_{so_grps} = 0$
$source_num_parameters_{so_grps} = 0$
$source_gparameters_base_{so_grps} = so_gparams_list_size$
$source_parameters_base_{so_grps} = so_params_list_size$
$source_parameters_data_base_{so_grps} = so_params_data_size$
$source_num_giparameters_{so_grps} = 0$
$source_num_iparameters_{so_grps} = 0$
$source_giparameters_base_{so_grps} = so_giparams_list_size$
$source_iparameters_base_{so_grps} = so_iparams_list_size$
$source_iparameters_data_base_{so_grps} = so_iparams_data_size$

$so_tot_curr(so_grps) = zero$
$so_scale(so_grps) = one$

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. */

$assert(iterative \equiv TRUE)$
$assert(so_base(so_grps) \equiv so_seg_tot + 1)$
$assert(so_type(so_grps) \equiv type)$
$assert(so_geom(so_grps) \equiv geom)$
$so_tot_curr(so_grps) = zero$

end if /* Set global source parameters. Presently only have these for the puff type. */

```
if (so_type(so_grps) ≡ so_puff) then
  source_num_gparameters_{so_grps} = 2
  increment_gparams(source_num_gparameters_{so_grps})
  so_gparams_list(1, so_grps) = so_gparam_puff_temp
  so_gparams_data(1, so_grps) = puff_temp
  so_gparams_list(2, so_grps) = so_gparam_puff_exponent
  so_gparams_data(2, so_grps) = puff_exponent
      /* Segment based parameters for plate source (presently for testing only). */
else if (so_type(so_grps) ≡ so_plate) then
  source_num_parameters_{so_grps} = 3
  increment_params(source_num_parameters_{so_grps})
  so_params_list(1, so_grps) = so_param_e_ion_delta
  so_params_list(2, so_grps) = so_param_e_ion_mult
  so_params_list(3, so_grps) = so_param_e_ion_sheath
      /* Externally specified volume source (not recombination) */
else if (so_type(so_grps) ≡ so_vol_source) then
  source_num_parameters_{so_grps} = 4
  increment_params(source_num_parameters_{so_grps})
  so_params_list(1, so_grps) = so_param_v1
  so_params_list(2, so_grps) = so_param_v2
  so_params_list(3, so_grps) = so_param_v3
  so_params_list(4, so_grps) = so_param_temperature
      /* Plate source with binned energy distribution. */
else if (so_type(so_grps) ≡ so_plt_e_bins) then
  source_num_giparameters_{so_grps} = 2
  increment_giparams(source_num_giparameters_{so_grps})
  so_giparams_list(1, so_grps) = so_giparam_e_bins_num
  so_giparams_data(1, so_grps) = e_bin_num
  so_giparams_list(2, so_grps) = so_giparam_e_bins_spacing
  so_giparams_data(2, so_grps) = e_bin_spacing

  source_num_gparameters_{so_grps} = 2
  increment_gparams(source_num_gparameters_{so_grps})
  so_gparams_list(1, so_grps) = so_gparam_e_bins_min
  so_gparams_list(2, so_grps) = so_gparam_e_bins_delta
  if (e_bin_spacing ≡ so_e_bins_spacing_linear) then
    so_gparams_data(1, so_grps) = e_bin_min
    so_gparams_data(2, so_grps) = (e_bin_max − e_bin_min) / areal(e_bin_num)
  else if (e_bin_spacing ≡ so_e_bins_spacing_log) then
    so_gparams_data(1, so_grps) = log(e_bin_min)
    so_gparams_data(2, so_grps) = (log(e_bin_max / e_bin_min)) / areal(e_bin_num)
  end if

  source_num_parameters_{so_grps} = e_bin_num
  increment_params(source_num_parameters_{so_grps})
  do i_bin = 1, e_bin_num
    so_params_list(i_bin, so_grps) = so_param_e_bin_prob
  end do
end if

write(stdout, *)
write(stdout, *) ' Processing source group ', so_grps
write(stdout, *)
    /* Have separate sections for the two geometries; originally just had surface sources. */
```

```
if (so_geom(so_grps) ≡ so_surface) then
  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 (sector_segment_test(sector, iseg)) 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 (sector_segment_test(sector, iseg)) then
        zone = sector_zone_{sector}
        so_seg_tot++

        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}
        if (update ≡ 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)

          if (source_num_parameters_{so_grps} > 0) then
            so_params_data_size += source_num_parameters_{so_grps}
            if (so_params_data_size > so_params_data_dim) then
              var_realloc(source_parameters_data, so_params_data_dim, so_params_data_size)
              so_params_data_dim = so_params_data_size
            end if
          end if
        end if
        assert(source_real_params_{iseg,code_strength} > zero ∧ source_real_params_{iseg,code_strength} ≠
            real_uninit)
        assert(source_int_params_{iseg,code_multiplicity} > 0)
        mult = one / float(source_int_params_{iseg,code_multiplicity})
        if (specify_flux ≡ TRUE) then
          ⟨ Calculate Source Current 11 ⟩
        else    /* Specify current */
          source_current_{so_seg_tot} = source_real_params_{iseg,code_strength} * mult
        end if
        so_tot_curr(so_grps) += source_current_{so_seg_tot}
```

```
          if (update ≡ FALSE) then
            source_segment_ptr_{so_seg_tot} = sector
            source_segment_rel_wt_{so_seg_tot} = one
          else
            assert(source_segment_ptr_{so_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
            ⟨ Assign Sheath Parameters 12 ⟩
          else if (so_type(so_grps) ≡ so_plt_e_bins) then
            assert(source_num_parameters_{so_grps} > 0)
            i_bin = 0
            do iparam = 1, source_num_parameters_{so_grps}
              if (so_params_list(iparam, so_grps) ≡ so_param_e_bin_prob) then
                      // Should be all of them!
                so_params_data(iparam, so_seg_tot,
                      so_grps) = source_real_params_{iseg,code_e_bin_min+i_bin}
                i_bin++
              end if
            end do
            assert(i_bin ≡ e_bin_num)
          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    /* Corresponding process for volume source. */
  else if (so_geom(so_grps) ≡ so_volume) then
    write(stdout, '(2x,a,2x,a,5x,a,7x,a,13x,a,13x,a)') 'segment', 'zone', 'iy', 'x1', 'x2',
        'x3'
      /* Go through all "segments" and count number of zones associated with each. Note the check of
          code_segment_iy against int_uninit: just following what was done for sector_segment_test. */
    do zone = 1, zn_num
      do iseg = 1, num_segments
        if (((zn_type(zone) ≡ zn_plasma) ∨ (zn_type(zone) ≡ zn_vacuum)) ∧
              (source_int_params_{iseg,code_zone} ≡ zone)) then
          source_int_params_{iseg,code_multiplicity}++
        end if
      end do
    end do    /* Then make a second pass and process all of the segments. */
    do zone = 1, zn_num
      do iseg = 1, num_segments
        if (((zn_type(zone) ≡ zn_plasma) ∨ (zn_type(zone) ≡ zn_vacuum)) ∧
              (source_int_params_{iseg,code_zone} ≡ zone)) then
          so_seg_tot++    /* PRINT OUTPUT FOR THIS SEGMENT */
          write(stdout, '(2x,i4,3x,i6,3x,i4,1x,3(1pe13.5,2x))') so_seg_tot − so_base(so_grps) + 1,
```

```
    zone, zn_index(zone, zi_iy), zone_center_{zone,1}, zone_center_{zone,2}, zone_center_{zone,3}
if (update ≡ 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)    //
  if (source_num_parameters_{so_grps} > 0) then
    so_params_data_size += source_num_parameters_{so_grps}
    if (so_params_data_size > so_params_data_dim) then
      var_realloc(source_parameters_data, so_params_data_dim, so_params_data_size)
      so_params_data_dim = so_params_data_size
    end if
  end if
end if
assert(source_real_params_{iseg,code_strength} > zero ∧ source_real_params_{iseg,code_strength} ≠
    real_uninit)
assert(source_int_params_{iseg,code_multiplicity} > 0)
mult = one / float(source_int_params_{iseg,code_multiplicity})
if (specify_flux ≡ TRUE) then    /* The generalization is actually of a volumetric source.
      There is no analog of the plasma-parameter based flux calculation done for the surface
      source. So, this is relatively simple. */
  assert(zn_volume(zone) > zero)
  source_current_{so_seg_tot} = source_real_params_{iseg,code_strength} * zn_volume(zone) * mult
else
  source_current_{so_seg_tot} = source_real_params_{iseg,code_strength} * mult
end if
so_tot_curr(so_grps) += source_current_{so_seg_tot}
if (update ≡ FALSE) then
  source_segment_ptr_{so_seg_tot} = zone
  source_segment_rel_wt_{so_seg_tot} = one
else
  assert(source_segment_ptr_{so_seg_tot} ≡ zone)
      /* source_segment_rel_wt gets set in update_wt_alias. */
end if    /* Corresponding setting of parameters for a generic (i.e., externally specified, not
    recombination) volume source. Could in principle skip the check on source type, but keep
    for consistency with the preliminary parameter-related assignments. */
if (so_type(so_grps) ≡ so_vol_source) then
  do iparam = 1, source_num_parameters_{so_grps}
    if (so_params_list(iparam, so_grps) ≡ so_param_temperature) then
          /* This is the only one that needs to be non-zero. */
      assert((source_real_params_{iseg,code_temperature_J} ≠
          real_uninit) ∧ (source_real_params_{iseg,code_temperature_J} > zero))
      so_params_data(iparam, so_seg_tot,
          so_grps) = source_real_params_{iseg,code_temperature_J}

    else if (so_params_list(iparam, so_grps) ≡ so_param_v1) then
      if (source_real_params_{iseg,code_velocity_1} ≠ real_uninit) then
        so_params_data(iparam, so_seg_tot, so_grps) = source_real_params_{iseg,code_velocity_1}
      else
        so_params_data(iparam, so_seg_tot, so_grps) = zero
```

```
            end if
          else if (so_params_list(iparam, so_grps) ≡ so_param_v2) then
            if (source_real_params_{iseg,code_velocity_2} ≠ real_uninit) then
              so_params_data(iparam, so_seg_tot, so_grps) = source_real_params_{iseg,code_velocity_2}
            else
              so_params_data(iparam, so_seg_tot, so_grps) = zero
            end if
          else if (so_params_list(iparam, so_grps) ≡ so_param_v3) then
            if (source_real_params_{iseg,code_velocity_3} ≠ real_uninit) then
              so_params_data(iparam, so_seg_tot, so_grps) = source_real_params_{iseg,code_velocity_3}
            else
              so_params_data(iparam, so_seg_tot, so_grps) = zero
            end if
          end if
        end do
      end if
    end if    // Segment-zone test
  end do    // Segments
end do    // Zones
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
else
  assert('␣This␣source␣geometry␣not␣handled␣in␣defineback' ≡ '␣')
end if    // Source geometry
```

This code is used in section 8.

11 Calculate the current for a surface source from the flux

⟨ Calculate Source Current 11 ⟩ ≡

```
vc_difference(sector_points_{sector,sc_neg}, sector_points_{sector,sc_pos}, xdiff)
assert(xdiff_2 ≡ zero)
if (geometry_symmetry ≡ geometry_symmetry_plane) then    /* But, by definition of this symmetry
      the problem spans the full length of the univeral cell in this direction. */
  circum = universal_cell_max_2 − universal_cell_min_2
else if (geometry_symmetry ≡ geometry_symmetry_plane_hw) then
  circum = zone_max_{zone,2} − zone_min_{zone,2}
else if (geometry_symmetry ≡ geometry_symmetry_cylindrical) then    /* Note that for this case,
      we follow the original convention of setting zone_min_{zone,2} = zone_max_{zone,2} = 0, so use π
      explicitly. */
  circum = PI ∗ (sector_points_{sector,sc_neg,1} + sector_points_{sector,sc_pos,1})
else if ((geometry_symmetry ≡ geometry_symmetry_cyl_hw) ∨ (geometry_symmetry ≡
      geometry_symmetry_cyl_section)) then
  circum = half ∗ (sector_points_{sector,sc_neg,1} + sector_points_{sector,sc_pos,1}) ∗ (zone_max_{zone,2} −
      zone_min_{zone,2})
else
  assert('Unexpected␣geometry␣symmetry␣in␣setting␣up␣source' ≡ '␣')
end if
area = circum ∗ vc_abs(xdiff)
assert(area > zero)
source_current_{so_seg_tot} = source_real_params_{iseg,code_strength} ∗ area ∗ mult
```

This code is used in section 10.

12 Assign sheath parameters for a plate source

⟨ Assign Sheath Parameters 12 ⟩ ≡

```
assert(sc_plasma_check(sector_type_pointer_{sector,sc_plasma}))
targ = sector_type_pointer_{sector,sc_plasma}
if ((source_real_params_{iseg,code_e_delta} ≠ real_uninit) ∧ (source_real_params_{iseg,code_e_sheath} ≠
      real_uninit) ∧ (source_real_params_{iseg,code_e_mult} ≠ real_uninit)) then
  plasma_e_ion_delta_{targ} = source_real_params_{iseg,code_e_delta}
  plasma_e_ion_sheath_{targ} = source_real_params_{iseg,code_e_sheath}
  plasma_e_ion_mult_{targ} = source_real_params_{iseg,code_e_mult}
      /* For testing now; may eventually replace the above. */
  assert(source_num_parameters_{so_grps} > 0)    // Should be 3!
  do iparam = 1, source_num_parameters_{so_grps}
    if (so_params_list(iparam, so_grps) ≡ so_param_e_ion_delta) then
      so_params_data(iparam, so_seg_tot, so_grps) = source_real_params_{iseg,code_e_delta}
    else if (so_params_list(iparam, so_grps) ≡ so_param_e_ion_sheath) then
      so_params_data(iparam, so_seg_tot, so_grps) = source_real_params_{iseg,code_e_sheath}
    else if (so_params_list(iparam, so_grps) ≡ so_param_e_ion_mult) then
      so_params_data(iparam, so_seg_tot, so_grps) = source_real_params_{iseg,code_e_mult}
    end if
  end do
else
  assert(sp_sy(pr_background(1)) ≡ 'e')    /* In some problems, the electron temperature can be
      zero, e.g., in low resolution tests. Since that can lead to zero test velocity when the source is
      sampled, establish a minimum plasma_e_ion_delta of 3 eV. */
  if (bk_temp(1, zone) > zero) then
    plasma_e_ion_delta_{targ} = const(3.) * bk_temp(1, zone)
  else
    plasma_e_ion_delta_{targ} = const(3.) * electron_charge
  end if
  plasma_e_ion_mult_{targ} = one
  plasma_e_ion_sheath_{targ} = zero
  assert(source_num_parameters_{so_grps} > 0)    // Should be 3!
  do iparam = 1, source_num_parameters_{so_grps}
    if (so_params_list(iparam, so_grps) ≡ so_param_e_ion_delta) then
      so_params_data(iparam, so_seg_tot, so_grps) = const(3.) * bk_temp(1, zone)
    else if (so_params_list(iparam, so_grps) ≡ so_param_e_ion_mult) then
      so_params_data(iparam, so_seg_tot, so_grps) = one
    else if (so_params_list(iparam, so_grps) ≡ so_param_e_ion_sheath) then
      so_params_data(iparam, so_seg_tot, so_grps) = zero
    end if
  end do
end if
```

This code is used in section 10.

13 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" 13 ≡
  @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 8⟩ +≡
    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(F)
    end if
  else
    write(stderr, *) '␣Illegal␣format␣token,␣', line(b : e)
    assert(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

    return
  end
```

14 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" 14 ≡
  @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 8⟩ +≡
    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*LINELEN 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
      zn_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
              if (next_token(line, b, e, p)) then
```

```
        goto density_loop
      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 ≡
          code_velocity_3)) then
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
```

15 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 8⟩ +≡
  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 : e) ≡ 'e_bin_prob') ∨ (line(b : e) ≡ 'E_BIN_PROB') ∨ (line(b : e) ≡ 'E_bin_prob'))
          then
      data_code = code_e_bin_min
      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(F)
        end if
      end if
    else if ((line(b : b) ≡ 'v') ∨ (line(b : b) ≡ 'V')) then
      if (line(b + 1 : b + 1) ≡ '1') then
        data_code = code_velocity_1
      else if (line(b + 1 : b + 1) ≡ '2') then
        data_code = code_velocity_2
      else if (line(b + 1 : b + 1) ≡ '3') then
        data_code = code_velocity_3
      else if ((line(b+1 : b+4) ≡ '_par') ∨ (line(b+1 : b+4) ≡ '_PAR') ∨ (line(b+1 : b+4) ≡ '_Par'))
            then
        data_code = code_velocity_par
      else
        write(stderr, *) '␣Illegal␣format␣token,␣', line(b : e)
        assert(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 ≠ 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

    return
  end
```

16 INDEX

abs: 10.
aname: 8.
area: 8, 11.
areal: 10.
arg_count: 7.
assert: 7, 8, 9, 10, 11, 12, 13, 14, 15.
aunit: 8.

b: 8, 13, 14, 15.
b_code: 13, 15.
b_repeat: 13.
back: 8, 9, 14, 15.
back_label: 8.
background_coords: 8.
bi: 13.
bk_check: 9, 14.
bk_common: 8, 14, 15.
bk_decl: 8, 14, 15.
bk_n: 8, 14.
bk_num: 8, 13, 15.
bk_temp: 8, 12, 14.
bk_v: 14.
boltzmanns_const: 8.
b2: 15.

cartesian: 1.
char_undef: 8.
circum: 8, 11.
code_all_segs: 8.
code_area: 8, 10, 15.
code_density: 8, 10, 14, 15.
code_e_bin_min: 8, 9, 10, 15.
code_e_delta: 8, 10, 12.
code_e_mult: 8, 10, 12.
code_e_sheath: 8, 10, 12.
code_electron_temperature: 8, 9, 10.
code_int: 8, 9.
code_int_max: 8.
code_int_min: 8.
code_multiplicity: 8, 10.
code_real: 8, 9.
code_real_max: 8.
code_real_min: 8.
code_segment: 8, 15.
code_segment_iy: 8, 10, 15.
code_stratum: 8, 15.
code_strength: 8, 10, 11, 15.
code_temperature_eV: 8, 9, 14, 15.
code_temperature_J: 8, 9, 10, 14, 15.
code_undefined: 8, 9, 14, 15.
code_velocity_par: 8, 10, 15.
code_velocity_1: 8, 10, 14, 15.
code_velocity_2: 8, 10, 14, 15.
code_velocity_3: 8, 10, 14, 15.
code_zone: 8, 10, 14, 15.
command_arg: 7.
const: 8, 10, 12.
count: 13, 14.
cumul: 8, 9.
cylindrical: 1.

data_code: 8, 9, 14, 15.
data_type: 8, 9.
declare_varp: 8, 14.
define_dimen: 8, 14.
define_varp: 8, 14.
defineback: 1, 7.
definegeometry2d: 1.
def2dplasma: 1.
degas2: 1.
density: 8.
density_loop: 14.
dim_segments: 8.
dim_y: 14.
diskin: 8, 14.
diskin2: 8, 9.
diskin3: 14.
dummy: 8.

e: 8, 13, 14, 15.
e_bin_entry: 8, 9.
e_bin_max: 8, 10.
e_bin_min: 8, 10.
e_bin_num: 8, 9, 10.
e_bin_spacing: 8, 10.
electron_charge: 8, 9, 12, 14.
end_source_group: 1.
eof: 13.
erase_geometry: 7.
e2: 15.

FALSE: 8, 9, 10, 13, 14.
`file`: 8.
FILE: 1.
file_format: 8.
FILELEN: 7, 8, 14.
float: 10.
FLOAT: 8.
`form`: 8.
form_density: 13.
form_temperature: 13.
form_undefined: 13.
form_velocity_1: 13.
form_velocity_2: 13.

form_velocity_3: 13.
form_zone: 13.
format_code: 13.
format_max: 13.
format_string: 13.
ftangle: 13.

geom: 8, 10.
geometry_symmetry: 1, 8, 11, 14.
geometry_symmetry_cyl_hw: 1, 8, 11, 14.
geometry_symmetry_cyl_section: 1, 8, 11, 14.
geometry_symmetry_cylindrical: 8, 11.
geometry_symmetry_oned: 8.
geometry_symmetry_plane: 8, 11.
geometry_symmetry_plane_hw: 1, 8, 11, 14.
get_n_t: 1, 8.
get_n_t_row: 8, 14.
gi_common: 8, 14.
grp: 8.
g2_polygon_stratum: 1.
g2_polygon_zone: 1.

half: 10, 11.

i: 8, 13.
i_bin: 8, 9, 10.
i_code: 8, 9.
implicit_none_f77: 7, 8, 13, 14, 15.
implicit_none_f90: 7, 8, 13, 14, 15.
in: 1.
increment_giparams: 8, 10.
increment_gparams: 8, 10.
increment_iseg: 8, 9.
increment_params: 8, 10.
index: 13, 15.
init_wt_alias: 8.
input_done: 8.
inputfile: 7, 8.
INT: 8, 14.
int_params_ind: 8.
int_undef: 8, 9, 10, 13, 14, 15.
int_uninit: 8, 9, 10.
int_unused: 8.
`iostat`: 8.
iparam: 8, 10, 12.
iseg: 8, 9, 10, 11, 12.
iterative: 8, 10.
iterative_solution: 1.
itest: 8, 9, 14.
ix: 1.
iy: 10, 14.
iy_zone: 14.
iy_zone_ind: 14.
iys_done: 8.
iz: 1.

j: 13.
jr: 8.

len: 8, 9, 14.
length: 8, 9, 13, 14.
line: 8, 9, 13, 14, 15.
LINELEN: 8, 13, 14, 15.
log: 10.

max: 14.
mem_check: 7.
mult: 8, 10, 11.

n_iy: 14.
nargs: 7.
nc_read_elements: 7.
nc_read_materials: 7.
nc_read_old_sources: 8.
nc_read_pmi: 7.
nc_read_problem: 7.
nc_read_reactions: 7.
nc_read_species: 7.
ncsxplasma: 1.
neutralize_species: 10.
new_source_group: 1.
next_datum: 8, 9.
next_iy: 8.
next_line: 8, 14.
next_rowline: 8, 9.
next_segment: 8.
next_stratum: 8.
next_strength: 8.
next_tabline: 8.
next_token: 8, 9, 13, 14, 15.
nflights: 8, 10.
nsectors: 10.
nt_string: 8, 14.
num_params: 8.
num_segments: 8, 9, 10.
nunit_n: 8.
nunit_t: 8.
nz: 1.

old_current: 8.
old_grps: 8.
old_rel_wt: 8.
old_seg_tot: 8.
old_tot_curr: 8.
oldsourcefile: 1.
one: 8, 9, 10, 12, 14.
open_file: 8, 14.
open_stat: 8, 14.
outputbrowser: 8.

p: 8, 13, 14, 15.
parse_format: 13, 15.
parse_label: 9, 14, 15.
parse_string: 8, 9, 13, 14.
phi_sheath: 8, 10.
PI: 11.
plasma_coords: 1.
plasma_coords_cartesian: 8.
plasma_coords_cylindrical: 8.
plasma_e_ion_delta: 1, 12.
plasma_e_ion_mult: 1, 12.
plasma_e_ion_sheath: 12.
plasma_file: 1, 14.
plate: 1.
pr_background: 9, 12.
pr_bkrc_num: 8.
pr_common: 8.
puff_exponent: 8, 10.
puff_temp: 8, 10.
p2: 15.

read_geometry: 7.
read_int_soft_fail: 9, 13, 14, 15.
read_integer: 8, 9, 14.
read_real: 8, 9, 14.
read_real_soft_fail: 9, 14.
read_string: 8, 9, 14.
readfilenames: 7.
real_mult: 8, 9.
real_params_ind: 8.
real_undef: 9, 14.
real_uninit: 8, 10, 12.
right: 1.
root_sp: 8, 10.
row: 14.
rtest: 8, 9, 14.

sc_common: 8.
sc_decl: 8.
sc_neg: 10, 11.
sc_plasma: 8, 12.
sc_plasma_check: 8, 12.
sc_pos: 10, 11.
sc_vacuum: 8.
sc_vacuum_check: 8.
sector: 1, 8, 10, 11, 12.
sector_points: 1, 10, 11.
sector_segment_test: 8, 10.
sector_strata_segment: 1, 8, 10.
sector_type_pointer: 8, 12.
sector_zone: 8, 10.
segment: 1.
segment_ind: 8.
segment_iy: 1.
segments_done: 8.
set_background_rate: 8.
set_background_sources: 8.
set_snapshot_source: 8.
setup_back_arrays: 7.
setup_background: 7, 8.
sn: 1.
snapshotfile: 1.
so_base: 8, 10.
so_chkpt: 10.
so_common: 8.
so_delta_fn: 8.
so_e_bins_num_max: 1, 8.
so_e_bins_spacing_linear: 8, 10.
so_e_bins_spacing_log: 8, 10.
so_geom: 10.
so_giparam_e_bins_num: 10.
so_giparam_e_bins_spacing: 10.
so_giparams_data: 10.
so_giparams_list: 10.
so_giparams_list_dim: 8.
so_giparams_list_size: 8, 10.
so_gparam_e_bins_delta: 10.
so_gparam_e_bins_min: 10.
so_gparam_puff_exponent: 10.
so_gparam_puff_temp: 10.
so_gparams_data: 10.
so_gparams_list: 10.
so_gparams_list_dim: 8.
so_gparams_list_size: 8, 10.
so_grps: 1, 8, 10, 12.
so_iparams_data_dim: 8.
so_iparams_data_size: 8, 10.
so_iparams_list_dim: 8.
so_iparams_list_size: 8, 10.
so_line: 8.
so_name: 8.
so_nflights: 10.
so_nseg: 10.
so_param_e_bin_prob: 10.
so_param_e_ion_delta: 10, 12.
so_param_e_ion_mult: 10, 12.
so_param_e_ion_sheath: 10, 12.
so_param_temperature: 10.
so_param_v1: 10.
so_param_v2: 10.
so_param_v3: 10.
so_params_data: 10, 12.
so_params_data_dim: 8, 10.
so_params_data_size: 8, 10.
so_params_list: 10, 12.

so_params_list_dim: 8.
so_params_list_size: 8, 10.
so_plate: 10.
so_plt_e_bins: 10.
so_point: 8.
so_puff: 10.
so_rel_wt_max: 1.
so_rel_wt_min: 1.
so_root_sp: 10.
so_scale: 10.
so_seg_tot: 8, 10, 11, 12.
so_set_run_flags: 8.
so_species: 10.
so_surface: 8, 10.
so_t_varn: 10.
so_time_dependent: 8.
so_time_final: 8.
so_time_initial: 8.
so_time_uniform: 8.
so_tot_curr: 8, 10.
so_type: 10.
so_type_num: 8.
so_vol_source: 10.
so_volume: 8, 10.
source_base_ptr: 8, 10.
source_current: 8, 10, 11.
source_e_bin_max: 1.
source_e_bin_min: 1.
source_e_bin_num: 1.
source_e_bin_spacing: 1.
source_file: 1, 8.
source_geom: 1.
source_geometry: 8, 10.
source_giparameters_base: 8, 10.
source_giparameters_data: 8.
source_giparameters_list: 8.
source_gparameters_base: 8, 10.
source_gparameters_data: 8.
source_gparameters_list: 8.
source_grp_ind: 8.
source_int_params: 8, 9, 10.
source_iparameters_base: 8, 10.
source_iparameters_data: 8.
source_iparameters_data_base: 8, 10.
source_iparameters_list: 8.
source_nflights: 1.
source_num_checkpoints: 8, 10.
source_num_flights: 8, 10.
source_num_giparameters: 8, 10.
source_num_gparameters: 8, 10.
source_num_iparameters: 8, 10.
source_num_parameters: 8, 10, 12.
source_num_segments: 8, 10.
source_parameters: 8.
source_parameters_base: 8, 10.
source_parameters_data: 8, 10.
source_parameters_data_base: 8, 10.
source_parameters_list: 8.
source_puff_exponent: 1.
source_puff_temp: 1.
source_real_params: 8, 9, 10, 11, 12.
source_root_sp: 1.
source_root_species: 8, 10.
source_scale_factor: 8, 10.
source_seg_ind: 8.
source_segment: 1.
source_segment_iy: 1.
source_segment_prob_alias: 10.
source_segment_ptr: 10.
source_segment_ptr_alias: 10.
source_segment_rel_wt: 8, 10.
source_species: 1, 8, 10.
source_stratum: 1.
source_strength: 1.
source_time_variation: 1, 8, 10.
source_total_current: 8, 10.
source_type: 1, 8, 10.
source_weight_norm: 8, 10.
sourcetest: 1.
sp_check: 8, 10.
sp_common: 8.
sp_decl: 8.
sp_lookup: 8.
sp_sy: 9, 12.
species: 8, 10.
species_infile: 1.
specify_current: 1.
specify_flux: 1, 8, 10.
st_decls: 8, 13, 14, 15.
standalone: 8.
`status`: 8.
stderr: 13, 15.
stdout: 10.
strata: 1, 8, 10.
strata_done: 8.
stratum: 1.
strengths_done: 8.
sub_count: 13.
sy_decls: 7.
symmetric_plasma: 14.

tallysetup: 1.
targ: 8, 12.
targ_v: 8, 10.
temperature: 8.

temperature_loop: 14.
temperature_mult: 14.
time_interval: 1.
time_varn: 8, 10.
token_loop: 13.
total: 8, 9.
TRUE: 8, 9, 10, 13, 14.
type: 8, 10.
type_string: 8.

uedge: 8.
`unit`: 8, 9, 14.
universal_cell_max: 11.
universal_cell_min: 11.
update: 8, 10.
update_wt_alias: 8, 10.
usr2dplasma: 1.

v_index: 14.
var_alloc: 8, 14.
var_free: 8.
var_realloc: 8, 10.
var_realloca: 8, 10.
var_reallocb: 8.
vc_abs: 10, 11.
vc_args: 10.
vc_decl: 8.
vc_difference: 11.
vc_set: 10.
vec: 1.
velocity_loop: 14.
vol_source: 1.

web: 1.
write_background: 8.
write_geometry: 8.

xdiff: 8, 11.

zero: 8, 9, 10, 11, 12.
zi_iy: 8, 10.
zi_ptr: 1, 14.
zn_check: 14.
zn_common: 8, 14.
zn_decl: 8, 14.
zn_decls: 14.
zn_index: 1, 8, 10, 14.
zn_num: 8, 10, 14.
zn_plasma: 8, 10, 14.
zn_type: 8, 10, 14.
zn_vacuum: 10.
zn_volume: 10.
zone: 1, 8, 10, 11, 12, 14.
zone_center: 10.
zone_ind: 14.
zone_iy_map: 14.
zone_iy_mapm: 14.
zone_list: 14.
zone_loop: 14.
zone_max: 11.
zone_min: 11.
zone_stratum: 1.

⟨ Assign Sheath Parameters 12 ⟩ Used in section 10.
⟨ Calculate Source Current 11 ⟩ Used in section 10.
⟨ Functions and subroutines 8, 13, 14, 15 ⟩ Used in section 7.
⟨ Memory allocation interface 0 ⟩ Used in sections 15, 14, 13, and 8.
⟨ Read Row Source File 9 ⟩ Used in section 8.
⟨ Setup Source Group 10 ⟩ Used in section 8.

COMMAND LINE: `"fweave -f -i! -W[ -ykw800 -ytw40000 -j -n/ /Users/dstotler/degas2/src/defineback.web"`.

WEB FILE: `"/Users/dstotler/degas2/src/defineback.web"`.

CHANGE FILE: `(none)`.

GLOBAL LANGUAGE: Fortran.