

# outputbrowser

November 24, 2009  
13:46

## Contents

1	A program to allow interactive browsing and printing of DEGAS 2's output netCDF file	1
2	INDEX	24

# 1 A program to allow interactive browsing and printing of DEGAS 2's output netCDF file

```
$Id: outputbrowser.web,v 1.5 2002/01/28 16:57:15 dstotler Exp $
```

When run without an argument, *outputbrowser* operates interactively, walking the user through the process of choosing a tally and deciding what information is to be displayed. A more explanatory “verbose” interface provides help along the way. The experienced user may find the “terse” interface more convenient.

This program can also be run via a script, with the name specified on the argument line:

```
outputbrowser ascriptfile
```

A nice feature of this code is that each interactive session (i.e., one not driven by a script) is logged into a file called *outputscript*. So, one can run the code in the interactive mode (with its relatively forgiving approach to error handling) to generate a script file. The user need only rename *outputscript* at the end of the interactive session (since the next execution of *outputbrowser* would overwrite it).

At the completion of a script-driven run, the user is given interactive control of the code. Additional tallies viewed at this point are again logged in *outputscript*. The user can rename this file or tack the resulting commands on to the end of another script file.

**Note that because some details, such as the number of zones or species, can vary from one problem to another, there is no guarantee that a given script will work with every run.** A wildcard facility has been built in to address this limitation.

We will now look in more detail at how to use *outputbrowser*.

In interactive mode, the code asks:

```
Do you want verbose (v) or terse instructions (t)?
```

The only valid responses are v, V (for verbose), t, and T (for terse). There is no corresponding command for the script files.

At the outermost level, the code is an endless loop. At the beginning of each pass through the loop, the user must specify a *tally* (from those defined when *tallysetup* was run), or choose “0” to exit the loop.

The tallies contained in this file are:

```

1 - wall & target current out
2 - wall & target energy current out
3 - wall & target current: energy spectrum
:
18 - Dalpha chord integrals
19 - Dalpha spectrum
20 - Dalpha spectrum detail
```

Choose one; use 0 to exit

The only valid responses are "0" (to quit) or an integer between "1" and "20" (or whatever the highest numbered tally is).

The corresponding syntax in the script file is:

```
Tally 5
```

to choose tally number 5.

Recall that DEGAS 2 keeps separate statistics for each of the source groups defined in the input background file. These results are combined to form the totals which appear in the text output files and elsewhere (e.g., the HDF files produced by *geomtesta*). The next step in running *outputbrowser* is to pick a group, or choose to look at the total:

```
Do you want to look at a specific source group?
Enter 0 for 'no', or 1 through 6
```

There are 6 source groups in this run. You can select one of them to view the results for it alone, or "0" to get only the total over all groups.

The syntax in the script file is:

```
Group 0
```

Having selected a particular tally, *outputbrowser* can look up its rank, that is, the number of independent variables. It will report this along with their names and the number of values associated with each:

```
'duct currents out' is of rank: 2
```

```
Its independent variables are:
```

rank	# values	name
1	6	problem_sp
2	5	diagnostic

Here, *problem\_sp* (i.e., the list of species specified in the problem input file) is the first independent variable, and a group of diagnostic sectors is the second variable. The former runs from 1 to 6; the latter 1 to 5.

The code will then prompt the user for a range for each of these:

```
Enter range indices, separated by a space for 'problem_sp'
If you want additional details on this variable, append a + sign
```

The simplest reply would be a pair of integers, e.g.,

```
1 6
```

indicating the values associated with *problem\_sp* 1 through 6. The only requirements here are that the second number be greater than or equal to the first and that both be between 1 and the number of values specified above.

Giving a single number:

```
5
```

is the same as saying `5 5`.

A wildcard `*` can be used to specify the upper end of the range. Used by itself, the code will use the full range of the independent variable. It can also be used with a lower limit:

```
2 *
```

In this case, this would be the same as specifying `2 6`.

Since the numbers associated with each independent variable are really defined only for the code's internal use, the user generally does not know how to associate them with particular physical entities. For this reason, an additional section of `outputbrowser` can be called upon to provide more details about a particular set of independent variables, say, the problem species. To get this output, the user can just append a "+" sign to the independent variable list, e.g.,

```
2 * +
```

The script syntax requires the specification of the rank of each independent variable, but is otherwise obvious:

```
Rank 1 : 1 5 +
Rank 2 : * +
```

`outputbrowser` will be printing out data in a tabular format, either to the screen or to a file. Because some of the independent variables can have thousands of values (e.g., zones), the user will want to have some control over how these data are presented. Rank 0 (scalar) data are printed directly without any user input. Likewise, there are no options for rank 1 data; these are printed down the page, one value per line. In both cases, the relative standard deviation is also printed.

Rank 2 data can be printed in one of two forms. In the tabular form, the dependence on one independent variable is laid out across the page, and the other down. No relative standard deviation is printed. If there are more than `max.cols` (presently 20) values for the first independent variable, the code abandons the tabular form and instead writes the data out in linear fashion down the page. In this case, the relative standard deviation is included. If there is only one value for the first independent variable, this form is used as well (i.e., the tabular form would also consist of a single column anyway, but without the relative standard deviation).

Rank 3 data is printed out as a sequence of 2-D slices, with the same bit of code handling the printing of the latter. Presently, the code is not set up to deal with data of rank 4 or higher.

The user can tell `outputbrowser` which independent variable to consider as the "first" (to be printed across the page in the tabular format), "second" (to go down the page), and "third" (separate pages). If the user responds to the question:

```
Would you like to re-order the independent variables for printing? (y/n)
```

with “n” or “N”, it will take the independent variable ordering to be that listed previously for this tally. If the answer is “y” or “Y”, the code asks

```
Using the integer index for each variable:
1 - problem_sp
2 - diagnostic
The first index will go across the page; second will go down;
third and more will be on separate pages
Enter the desired ordering:
```

Again, the default ordering would be equivalent to responding with 1 2. To change the ordering (say, if there were only a few values of *diagnostic*, but many of *problem\_sp*), the user would type 2 1. For a rank 3 tally, the user would be asked to select a permutation of 1, 2, and 3.

The syntax of the script file here should then be clear:

```
Order 2 1
```

If the tally is a vector quantity (such as the average neutral velocity, the user will see:

```
This tally is not a scalar quantity. Enter the dimension to be printed, 1 to 3
```

The coordinate system for vector quantities will be that specified by the output conversions for the tally, if any, during *tallysetup*. For cartesian coordinates, the dimensions 1, 2, and 3 would correspond to *x*, *y*, and *z*, respectively.

The corresponding line in the script file is:

```
Dimension 1
```

This line can be omitted for scalar or rank 1 data.

Finally, *outputbrowser* needs to know where to write the data on this tally:

```
Display this on screen or file? (s/f)
```

Typing “s” or “S” will cause the results to be printed to `stdout`. If “f” or “F” is entered, the user is prompted for a file name:

```
Enter filename to store information
```

This filename should not be longer than *FILELEN* characters long (currently 96).

To select screen output in the script file, use the line:

```
Screen
```

To instead have the data written to a file (here, `afilename`), use

```
File filename
```

§1-§1.1 [#1-#2]     **outputbrowser**A program to allow interactive browsing and printing of DEGAS 2's output netCDF file

The description of the output for each tally in the script file must be terminated by a line starting with a  
-. The script files generated by *outputbrowser* will use

-----  
At this point, *outputbrowser* will extract the requested data on this tally from the output arrays and write it to the specified destination. If additional information on one or more independent variables has been requested, that will be printed as well. The beginning of this section will be delimited by a line containing the name of the independent variable in all caps.

The code will then ask if another tally is to be investigated:

Look at another tally? (y/n)

This request is also made when the code has reached the end of an input script file. The user can bypass this question by specifying

Tally 0

as the last tally.

```
"outputbrowser.f" 1 ≡
  @m FILE 'outputbrowser.web'
  @m max_cols 20 // Number of data columns in output format
  @m get_next_token // Appears often enough to define
    assert(length ≤ len(line))
    length = parse_string(line(: length))
    p = 0
    assert(next_token(line, b, e, p))
```

Statement labels for go to's (primarily used for error control)

```
"outputbrowser.f" 1.1 ≡
  @m next_tally #:0
  @m tally_done #:0
  @m set_verbose #:0
  @m pick_tally #:0
  @m pick_group #:0
  @m pick_range #:0
  @m pick_ordering #:0
  @m pick_dim #:0
  @m pick_output #:0
  @m test_tally #:0
  @m next_line #:0
  @m range_loop #:0
  @m range_loop2 #:0
```

The main program.

```
"outputbrowser.f" 1.2 ≡
  program output_browser
    implicit_none_f77
    implicit_none_f90

    integer nargs
    character*FILELEN scriptfile

    sy_decls

    nargs = arg_count()
    if (nargs > 0) then
      call command_arg(1, scriptfile)
      if (nargs > 1)
        write (stdout, *) 'Only one script file per run'
      else
        scriptfile = ' '
      end if

      call readfilenames
      call degas_init
      call nc_read_output

      call browse(scriptfile)

      stop
    end
```

⟨ Functions and subroutines 1.3 ⟩

This routine does the browsing.

⟨Functions and subroutines 1.3⟩ ≡

**subroutine** *browse*(*scriptfile*)

*define\_dimen*(*value\_ind*, *value\_size*)

*define\_varp*(*value*, *FLOAT*, *value\_ind*)

*define\_varp*(*rsd*, *FLOAT*, *value\_ind*)

*implicit\_none\_f77*

*pr\_common*

*so\_common*

*tl\_common*

*ou\_common*

*implicit\_none\_f90*

**character\****FILELEN* *scriptfile*

**integer** *jscore*, *length*, *p*, *b*, *e*, *i*, *j*, *k*, *i1*, *i2*, *i3*, *dim*, *group*, *screen*, *nunit*, *value\_size*, *ii*, *init*, *verbose*,  
*first\_pass*

**integer** *min\_slice*<sub>*tl\_rank\_max*</sub>, *max\_slice*<sub>*tl\_rank\_max*</sub>, *index\_parameters*<sub>*tl\_index\_max*</sub>, *details*<sub>*tl\_rank\_max*</sub>

**character\***1 *choice*

**character\***2 *adddet*

**character\****LINELEN* *line*

**character\****FILELEN* *tempfile*

⟨Memory allocation interface 0⟩

*st\_decls*

**real** *extract\_output\_datum*

**external** *extract\_output\_datum*

*declare\_varp*(*value*)

*declare\_varp*(*rsd*)

*value\_size* = 1000

*var\_alloc*(*value*)

*var\_alloc*(*rsd*)

*init* = *TRUE*

**open**(*unit* = *diskout*, *file* = 'outputscript', *status* = 'unknown')

**if** (*scriptfile* ≠ '␣')

**open**(*unit* = *diskin*, *file* = *scriptfile*, *status* = 'old')

*next\_tally*: **continue**

**if** (*scriptfile* ≡ '␣') **then** ⟨User Input 1.4⟩

**else** // Read *scriptfile*

    ⟨Read Script 1.5⟩

**end if**

    /\* File output: offer the option of writing to a netCDF file? - would want to create the zone map  
    ahead of time here. \*/

    /\* Write Data \*/

**write**(*nunit*, \*)

**write**(*nunit*, \*) ''', *trim*(*tally\_name*<sub>*jscore*</sub>), ''':'

**if** (*tally\_dep\_var\_dim*<sub>*jscore*</sub> > 1) **then**

```

    write(nunit, *) 'dimension_', dim
end if
write(nunit, *)

if(max_slice_i1 > value_size) then
    var_realloc(value, value_size, max_slice_i1)
    var_realloc(rsd, value_size, max_slice_i1)
    value_size = max_slice_i1
end if

/* Rank 0 (scalar) Data */
if(tally_rank_jscore == 0) then
    i = 1
    <get datum 1.6>
    write(nunit, *) value_1, rsd_1

    /* Rank 1 Data */
else if(tally_rank_jscore == 1) then
    write(nunit, '(a10,7x,a,5x,a)') trim(tally_var_list_tally_indep_var_jscore_1), 'value',
        'rel_std_dev.'

    do i = min_slice_1, max_slice_1
        index_parameters_tally_indep_var_jscore_1 = i
        <get datum 1.6>
    end do
    do i = min_slice_1, max_slice_1
        write(nunit, '(5x,i5,5x,1pe13.5,2x,0pf7.4)') i, value_i, rsd_i
    end do

    /* Rank 2 Data */
else if(tally_rank_jscore == 2) then
    first_pass = TRUE
    <write 2D slice 1.8> /* Rank 3 Data */
else if(tally_rank_jscore == 3) then
    first_pass = TRUE
    do k = min_slice_i3, max_slice_i3
        index_parameters_tally_indep_var_jscore_i3 = k
        write(nunit, *)
        write(nunit, '(a,a,i5)') trim(tally_var_list_tally_indep_var_jscore_i3), ':', k

        <write 2D slice 1.8>
    end do /* Higher Ranks: would probably treat as generalizations of rank 3 case */
else
    write(stdout, *) 'Rank not currently supported here'
end if
do i = 1, tally_rank_jscore
    if(details_i == TRUE) then
        call dump_indep_vars(nunit, jscore, tally_indep_var_jscore_i, min_slice_i, max_slice_i)
    end if
end do

tally_done: continue

if(scriptfile == '') then
    write(stdout, *)
    write(stdout, *) 'Look at another tally?(y/n)'

```

```

    if ( $\neg$ read_string(stdin, line, length))
        goto eof
    get_next_token
    if (line(b : e)  $\equiv$  'y'  $\vee$  line(b : e)  $\equiv$  'Y')
        go to next_tally
    else // script driven
        write(nunit, *) '-----',
        go to next_tally
    end if
eof: continue
    return end

```

See also section 1.9.

This code is used in section 1.2.

Get desired output description directly from user.

⟨ User Input 1.4 ⟩ ≡

```

set_verbose: continue

write(stdout, *)
if (init ≡ TRUE) then
  write(stdout, *) 'Do you want verbose(v) or terse instructions(t)?'
  if (¬read_string(stdin, line, length)) then
    write(stdout, *) 'No, you actually have to type in "v" or "t".'
    go to set_verbose
  end if
  get_next_token
  if (line(b:e) ≡ 'v' ∨ line(b:e) ≡ 'V') then
    verbose = TRUE
  else if (line(b:e) ≡ 't' ∨ line(b:e) ≡ 'T') then
    verbose = FALSE
  else
    write(stdout, *) 'Type a "v" or a "t"'
    go to set_verbose
  end if
end if
if (verbose ≡ TRUE ∨ init ≡ TRUE) then
  write(stdout, *) 'The tallies contained in this file are:'
  write(stdout, *)
  do jscore = 1, tl_num
    write(stdout, '(i4,a,a)') jscore, ' ', trim(tally_name_jscore)
  end do
  write(stdout, *)
  write(stdout, *) 'Choose one; use 0 to exit'
else
  write(stdout, *) 'Choose next tally:'
end if

pick_tally: continue

if (¬read_string(stdin, line, length))
  goto eof
get_next_token
jscore = read_int_soft_fail(line(b:e))
if (jscore ≡ 0)
  go to eof // Quitting
if (¬tl_check(jscore)) then // Erroneous input
  write(stdout, *) 'Tally must be between 1 and ', tl_num
  write(stdout, *) 'Try again, or use 0 to quit:'
  go to pick_tally
end if

write(stdout, *)
if (so_grps > 1) then
  if (verbose ≡ TRUE ∨ init ≡ TRUE) then
    write(stdout, *) 'Do you want to look at a specific source group?'
    write(stdout, *) 'Enter 0 for "no", or 1 through ', so_grps
  else
    write(stdout, *) 'Pick source group:'
  end if
end if

```

```

    end if
pick_group: continue
    if (¬read_string(stdin, line, length))
        goto eof
    get_next_token
    group = read_int_soft_fail(line(b:e))
    if (¬so_check(group) ∧ group ≠ 0) then
        write(stdout, *) 'Group must be between 1 and ', so_grps
        write(stdout, *) 'Try again; use 0 to select all:'
        go to pick_group
    end if
else
    group = 0
end if

/* Pick Independent Variable Ranges */
write(stdout, *)
if (tally_rank_kscore > 0) then
    if (verbose ≡ TRUE) then
        write(stdout, *) '''', trim(tally_name_kscore), ''' is of rank: ', tally_rank_kscore
        write(stdout, *)
        write(stdout, *) 'Its independent variables are:'
        write(stdout, *)
        write(stdout, *) 'rank # values name'
    else
        write(stdout, *) 'Independent variables are:'
    end if
    do i = 1, tally_rank_kscore
        write(stdout,
            ' (3x,i4,5x,i5,4x,a) ' i, tally_tab_index_kscore,i, trim(tally_var_list_tally_indep_var_kscore,i)
        )
    end do

    write(stdout, *)
    do i = 1, tally_rank_kscore
        if (verbose ≡ TRUE) then
            write(stdout, *) 'Enter range indices, separated by a space for',
                trim(tally_var_list_tally_indep_var_kscore,i), '''
            write(stdout,
                *) 'If you want additional details on this variable, append a + sign'
        else
            write(stdout, *) 'Range for', trim(tally_var_list_tally_indep_var_kscore,i), '''
        end if
    end do
pick_range: continue

    if (¬read_string(stdin, line, length))
        goto eof
    get_next_token

    min_slice_i = int_undef
    max_slice_i = int_undef
    details_i = FALSE
range_loop: continue (range logic 1.7) if (next_token(line, b, e, p))
    go to range_loop

```

```

    if (max_slice_i ≡ int_undef)
        max_slice_i = min_slice_i

    if (min_slice_i < 1 ∨ min_slice_i > tally_tab_index_jscore,i ∨ max_slice_i < 1 ∨ max_slice_i >
        tally_tab_index_jscore,i ∨ min_slice_i > max_slice_i) then
        write (stdout, *) 'Range limits must be between 1 and ', tally_tab_index_jscore,i
        write (stdout, *) 'Try again:'
        go to pick_range
    end if
end do
end if

/* Reorder independent variables */
if (tally_rank_jscore ≡ 0 ∨ tally_rank_jscore ≡ 1) then
    i1 = 1
else
    write (stdout, *)
    write (stdout,
        *) 'Would you like to re-order the independent variables for printing? (y/n)'

    if (¬read_string(stdin, line, length))
        go to eof
    get_next_token
    choice = line(b : e)
    if (choice ≠ 'y' ∧ choice ≠ 'Y') then
        i1 = 1
        i2 = 2
        i3 = 3
    else
        if (verbose ≡ TRUE) then
            write (stdout, *) 'Using the integer index for each variable:'
            do i = 1, tally_rank_jscore
                write (stdout, *) i, ' ', trim(tally_var_list_tally_indep_var_jscore,i)
            end do
            write (stdout,
                *) 'The first index will go across the page; second will go down;'
            write (stdout, *) 'third and more will be on separate pages'
        end if

pick_ordering: continue

        write (stdout, *) 'Enter the desired ordering:'
        if (¬read_string(stdin, line, length))
            goto eof
        get_next_token
        i1 = read_int_soft_fail(line(b : e))
        if (tally_rank_jscore ≥ 2) then
            assert(next_token(line, b, e, p))
            i2 = read_int_soft_fail(line(b : e))
        end if
        if (tally_rank_jscore ≥ 3) then
            assert(next_token(line, b, e, p))
            i3 = read_int_soft_fail(line(b : e))
        end if
    end if

```

```

    if ( $i1 < 1 \vee i1 > tally\_rank_{j\_score} \vee (tally\_rank_{j\_score} \geq 2 \wedge (i2 < 1 \vee i2 >
        tally\_rank_{j\_score})) \vee (tally\_rank_{j\_score} \geq 3 \wedge (i3 < 1 \vee i3 > tally\_rank_{j\_score}))$ ) then
        write (stdout, *) 'These integers must all be between 1 and ', tally_rank_j_score,
            '; Try again'
        go to pick_ordering
    end if
end if
end if
pick_dim: continue

if (tally_dep_var_dim_j_score  $\equiv$  1) then
    dim = 1
else
    write (stdout, *) 'This tally is not a scalar quantity. Enter the di\
        mension to be printed, 1 to ', tally_dep_var_dim_j_score
    if ( $\neg$ read_string(stdin, line, length))
        goto eof
    get_next_token
    dim = read_int_soft_fail(line(b:e))
    if ( $dim < 1 \vee dim > tally\_dep\_var\_dim_{j\_score}$ ) then
        write (stdout, *) 'Try again; pick a number between 1 and ', tally_dep_var_dim_j_score
        go to pick_dim
    end if
end if

pick_output: continue

write (stdout, *)
write (stdout, *) 'Display this on screen or file? (s/f)'
if ( $\neg$ read_string(stdin, line, length))
    goto eof
get_next_token
choice = line(b:e)
if (choice  $\equiv$  'S'  $\vee$  choice  $\equiv$  's') then
    screen = TRUE
    nunit = stdout
else if (choice  $\equiv$  'F'  $\vee$  choice  $\equiv$  'f') then
    screen = FALSE
    write (stdout, *)
    write (stdout, *) 'Enter filename to store information'
    write (stdout, *)

    if ( $\neg$ read_string(stdin, line, length))
        goto eof
    get_next_token
    tempfile = line(b:e)
    open (unit = diskout + 1, file = tempfile, status = 'unknown')

    nunit = diskout + 1
else
    write (stdout, *) 'No, that will not work, try again.'
    go to pick_output
end if /* Write out (successful) data to script */
init = FALSE
write (diskout, *) 'Tally', j_score

```

```

write (diskout, *) '␣Group␣', group
if (tally_rankjscore > 0) then
  do i = 1, tally_rankjscore
    if (detailsi ≡ TRUE) then
      adddet = '␣+'
    else
      adddet = '␣␣'
    end if
    write (diskout, '(a,i1,a,i6,2x,i6,a)') '␣␣Rank␣', i, '␣:␣', min_slicei, max_slicei, adddet
  end do
  if (tally_rankjscore ≡ 2) then
    write (diskout, '(a,i1,2x,i1)') '␣␣Order␣', i1, i2
  else if (tally_rankjscore ≡ 3) then
    write (diskout, '(a,i1,2x,i1,2x,i1)') '␣␣Order␣', i1, i2, i3
  end if
end if
write (diskout, *) '␣Dimension␣', dim
if (screen ≡ TRUE) then
  write (diskout, *) '␣Screen'
else
  write (diskout, *) '␣File␣', trim(tempfile)
end if
write (diskout, *) '-----',

```

This code is used in section 1.3.

Read output description from script file.

⟨Read Script 1.5⟩ ≡

```

next_line: continue

if ( $\neg$ read_string(diskin, line, length)) then
  close(unit = diskin)
  scriptfile = '␣'
  go to tally_done
else
  get_next_token
  if (line(b : e) ≡ 'Tally') then
    assert(next_token(line, b, e, p))
    jscore = read_int_soft_fail(line(b : e))
  else if (line(b : e) ≡ 'Group') then
    assert(next_token(line, b, e, p))
    group = read_int_soft_fail(line(b : e))
  else if (line(b : e) ≡ 'Rank') then
    assert(next_token(line, b, e, p))
    i = read_int_soft_fail(line(b : e))
    assert(next_token(line, b, e, p))
    assert(line(b : e) ≡ ':')
    assert(next_token(line, b, e, p))

    min_slice_i = int_undef
    max_slice_i = int_undef
    details_i = FALSE
range_loop2: continue ⟨range logic 1.7⟩ if (next_token(line, b, e, p))
  go to range_loop2
  if (max_slice_i ≡ int_undef)
    max_slice_i = min_slice_i

  else if (line(b : e) ≡ 'Dimension') then
    assert(next_token(line, b, e, p))
    dim = read_int_soft_fail(line(b : e))
  else if (line(b : e) ≡ 'Order') then
    assert(next_token(line, b, e, p))
    i1 = read_int_soft_fail(line(b : e))
    assert(next_token(line, b, e, p))
    i2 = read_int_soft_fail(line(b : e))
    if (next_token(line, b, e, p)) then
      i3 = read_int_soft_fail(line(b : e))
    else
      i3 = 3    // Do we need a default for this case?
    end if
  else if (line(b : e) ≡ 'Screen') then
    screen = TRUE
    nunit = stdout
  else if (line(b : e) ≡ 'File') then
    screen = FALSE
    assert(next_token(line, b, e, p))
    tempfile = line(b : e)
    open(unit = diskout + 1, file = tempfile, status = 'unknown')
    nunit = diskout + 1
  else if (line(1 : 1) ≡ '-') then

```

```

        go to test_tally
    else
        write(stdout, *) 'Ignoring unexpected line:', line
    end if
end if

go to next_line

test_tally: continue

if (jscore  $\equiv$  0)
    go to eof // Quitting
if ( $\neg$ tl_check(jscore)) then // Erroneous input
    write(stdout, *)
    write(stdout, *) 'Tally', jscore, ' is invalid.'
    write(stdout, *) 'It must be between 1 and', tl_num
    go to next_line
end if
if ( $\neg$ so_check(group)  $\wedge$  group  $\neq$  0) then
    write(stdout, *)
    write(stdout, *) 'Invalid group:', group, ' for tally', jscore
    write(stdout, *) 'Must be between 1 and', so_grps
    go to next_line
end if
do i = 1, tally_rank_jscore
    if (min_slice_i < 1  $\vee$  min_slice_i > tally_tab_index_jscore,i  $\vee$  max_slice_i < 1  $\vee$  max_slice_i >
        tally_tab_index_jscore,i  $\vee$  min_slice_i > max_slice_i) then
        write(stdout, *)
        write(stdout, *) 'Invalid range for tally', jscore, ' rank', i
        write(stdout, *) 'Range limits must be between 1 and', tally_tab_index_jscore,i
        go to next_line
    end if
end do
if (tally_rank_jscore  $\geq$  2) then
    if (i1 < 1  $\vee$  i1 > tally_rank_jscore  $\vee$  (tally_rank_jscore  $\geq$  2  $\wedge$  (i2 < 1  $\vee$  i2 >
        tally_rank_jscore))  $\vee$  (tally_rank_jscore  $\geq$  3  $\wedge$  (i3 < 1  $\vee$  i3 > tally_rank_jscore))) then
        write(stdout, *)
        write(stdout, *) 'Invalid ordering of independent variables:', i1, i2, i3,
            ' for tally', jscore
        write(stdout, *) 'These integers must all be between 1 and', tally_rank_jscore
        go to next_line
    end if
else
    i1 = 1
end if
if (dim < 1  $\vee$  dim > tally_dep_var_dim_jscore) then
    if (tally_dep_var_dim_jscore  $\leq$  1) then
        dim = 1
    else
        write(stdout, *)
        write(stdout, *) 'Invalid dimension:', dim, ' for tally', jscore
        go to next_line
    end if
end if
end if

```

§1.5–§1.7 [#6–#8]     **outputbrowser**A program to allow interactive browsing and printing of DEGAS 2's output netCDF file

This code is used in section 1.3.

Just to avoid repeating these lines.

⟨get datum 1.6⟩ ≡

```
if (so_check(group)) then
  valuei = extract_output_datum(index_parameters, dim, out_post_grpgroup,0,o_mean, o_mean,
    tally_namejscore)
  rsdi = extract_output_datum(index_parameters, dim, out_post_grpgroup,0,o_mean, o_var,
    tally_namejscore)
else
  valuei = extract_output_datum(index_parameters, dim, out_post_all, o_mean, tally_namejscore)
  rsdi = extract_output_datum(index_parameters, dim, out_post_all, o_var, tally_namejscore)
end if
```

This code is used in sections 1.3 and 1.8.

This rather involved piece of logic is used for both user and script input.

⟨range logic 1.7⟩ ≡

```
if (line(b : e) ≡ '+' ) then
  detailsi = TRUE
else if (line(b : e) ≡ '*') then
  max_slicei = tally_tab_indexjscore,i
  if (min_slicei ≡ int_undef)
    min_slicei = 1
else
  if (line(e : e) ≡ '+') then
    e -= 1    // User left out space before + sign
    detailsi = TRUE
  end if
  if (min_slicei ≡ int_undef) then
    min_slicei = read_int_soft_fail(line(b : e))
  else
    max_slicei = read_int_soft_fail(line(b : e))
  end if
end if
```

This code is used in sections 1.4 and 1.5.

Write out 2-D slice. Used by rank 2 and 3 data; probably will be used by higher ranks when they are included.

```

< write 2D slice 1.8 > ≡
  do j = min_slice_i2, max_slice_i2
    index_parameters_tally_indep_var_jscore,i2 = j
    do i = min_slice_i1, max_slice_i1
      index_parameters_tally_indep_var_jscore,i1 = i
      < get datum 1.6 >
    end do
    if (max_slice_i1 - min_slice_i1 + 1 < max_cols ∧ max_slice_i1 > min_slice_i1) then
      if (j ≡ min_slice_i2) then
        write (nunit, '(12x,a,a3)') trim(tally_var_list_tally_indep_var_jscore,i1), '┘->'
        write (nunit, '(8x,20(5x,i5,5x))') SP(ii, ii = min_slice_i1, max_slice_i1)
        write (nunit, '(a)') trim(tally_var_list_tally_indep_var_jscore,i2)
      end if
      write (nunit, '(i5,2x,1p,20(2x,e13.5))') j, (value_ii, ii = min_slice_i1, max_slice_i1)
    else
      if (first_pass ≡ TRUE) then
        write (nunit, '(a10,2x,a10,5x,a,t39,a)') trim(tally_var_list_tally_indep_var_jscore,i1),
          trim(tally_var_list_tally_indep_var_jscore,i2), '┘value', '┘rel.┘std.┘dev.'
        first_pass = FALSE
      end if
      do i = min_slice_i1, max_slice_i1
        write (nunit, '(4x,i5,5x,i5,5x,1pe13.5,2x,0pf7.4)') i, j, value_i, rsd_i
      end do
    end if
  end do

```

This code is used in section 1.3.

A separate routine to print data related to the independent variables.

⟨Functions and subroutines 1.3⟩ +≡

```

subroutine dump_indep_vars(nunit, jscore, ivar, imin, imax)

    implicit_none_f77
    zn_common     // Common
    sc_common
    de_common
    sp_common
    rc_common
    pm_common
    pr_common
    so_common
    tl_common
    gi_common
    implicit_none_f90
    st_decls

    integer nunit, jscore, ivar, imin, imax     // Input

    integer pointer, i, sec, zone     // Local
    real energy, angle, wavelength, circum, area

    character*LINLEN type_string
    character*20 psp_label

    vc_decl(xdiff)

    vc_decls     /* Go through the list in tally.hweb */
    write(nunit, *)
    if (ivar ≡ tl_index_unknown) then
        assert('␣Unknown␣variable,␣something␣is␣wrong!' ≡ '␣')
    else if (ivar ≡ tl_index_zone) then
        write(nunit, *) '␣␣ZONES:'
        write(nunit, *)
        write(nunit, '(a,3x,a,9x,a,13x,a,11x,a)') '␣zone␣', 'center:␣x1', 'x2', 'x3', 'volume'
        do i = imin, imax
            assert(zn_check(i))
            write(nunit, '(i5,2x,1p,4(e13.5,2x))') i, zone_center_i,1, zone_center_i,2, zone_center_i,3,
                zn_volume(i)
        end do
    else if (ivar ≡ tl_index_plasma_zone) then
        write(nunit, *) '␣Nothing␣written␣for␣plasma␣zone'
    else if (ivar ≡ tl_index_test) then
        write(nunit, *) '␣TEST␣SPECIES:'
        write(nunit, *)
        write(nunit, '(a,4x,a,3x,a)') 'test', 'species_no.', 'symbol'
        do i = imin, imax
            assert(pr_test_check(i))
            write(nunit, '(i4,6x,i4,10x,a)') i, pr_test(i), sp_sy(pr_test(i))
        end do
    else if (ivar ≡ tl_index_problem_sp) then
        write(nunit, *) '␣PROBLEM␣SPECIES:'
        write(nunit, *)
        write(nunit, '(a,4x,a,5x,a,2x,a,2x,a)') 'problem_sp', 'class', 'number', 'species_no.',
            'symbol'

```

```

do i = imin, imax
  if (i ≡ 1) then
    psp_label = 'background'
  else if (i ≡ pr_background_num + 1) then
    write (nunit, *) '-----',
    psp_label = 'test'
  else
    psp_label = '□'
  end if
  if (i ≤ pr_background_num) then
    assert(pr_background_check(i))
    write (nunit, '(1x,i4,7x,a,t25,i4,6x,i4,10x,a)') i, trim( psp_label), i, pr_background(i),
    sp_sy(pr_background(i))
  else
    assert(pr_test_check(i - pr_background_num))
    write (nunit,
    '(1x,i4,7x,a,t25,i4,6x,i4,10x,a)') i, trim( psp_label), i - pr_background_num,
    pr_test(i - pr_background_num), sp_sy(pr_test(i - pr_background_num))
  end if
end do
else if (ivar ≡ tl_index_detector) then
  pointer = tally_geometry_ptrjscore
  write (nunit, *) '□DETECTOR□', trim(detector_namepointer), '':
  write (nunit, *)
  write (nunit, '(a,2x,a,4x,a,9x,a,13x,a,11x,a,10x,a,13x,a)') 'detector', 'view□no.',
  'start:□x1', 'x2', 'x3', 'end:□x1', 'x2', 'x3'
do i = imin, imax
  write (nunit, '(1x,i4,5x,i4,5x,1p,6(e13.5,2x))') i, de_view_pointer(i,
  pointer), de_view_pointsde_view_pointer(i, pointer),de_view_start,1,
  de_view_pointsde_view_pointer(i, pointer),de_view_start,2,
  de_view_pointsde_view_pointer(i, pointer),de_view_start,3,
  de_view_pointsde_view_pointer(i, pointer),de_view_end,1,
  de_view_pointsde_view_pointer(i, pointer),de_view_end,2,
  de_view_pointsde_view_pointer(i, pointer),de_view_end,3
end do
else if (ivar ≡ tl_index_test_author) then
  write (nunit, *) '□TEST□AUTHOR:'
  write (nunit, *)
  write (nunit, '(a,3x,a,3x,a,5x,a)') 'author', 'class', 'number', 'name'
do i = imin, imax
  if (i ≡ 1) then
    psp_label = 'source'
  else if (i ≡ so_type_num + 1) then
    psp_label = 'pr.□reac.'
    write (nunit, *) '-----',
  else if (i ≡ so_type_num + pr_reaction_num + 1) then
    psp_label = 'pr.□PMI'
    write (nunit, *) '-----',
  else
    psp_label = '□'
  end if
  if (i ≤ so_type_num) then

```

```

    type_string = so_name(i)
@#if 0
    if (i ≡ so_plate) then
        type_string = 'plate'
    else if (i ≡ so_puff) then
        type_string = 'puff'
    else if (i ≡ so_recomb) then
        type_string = 'recomb'
    else
        assert('Source_type_not_known' ≡ ' ')
    end if
@#endif
    write(nunit, '(i4,4x,a,t17,i4,4x,a)') i, trim(psp_label), i, trim(type_string)
    else if (i ≤ so_type_num + pr_reaction_num) then
        write(nunit, '(i4,4x,a,t17,i4,4x,a)') i, trim(psp_label), i - so_type_num,
            trim(rc_name(pr_reaction(i - so_type_num)))

    else
        write(nunit, '(i4,4x,a,t17,i4,4x,a)') i, trim(psp_label), i - so_type_num - pr_reaction_num,
            trim(pm_name(pr_pm_ref(i - so_type_num - pr_reaction_num)))
    end if
end do
else if (ivar ≡ tl_index_reaction) then
    write(nunit, *) ' REACTIONS:'
    write(nunit, *)
    if (imin ≤ pr_reaction_num) then
        write(nunit, '(a,2x,a,5x,a)') 'pr_reac.', 'reaction', 'name'
        do i = imin, min(imax, pr_reaction_num)
            write(nunit, '(1x,i4,7x,i4,4x,a)') i, pr_reaction(i), trim(rc_name(pr_reaction(i)))
        end do
        if (imax > pr_reaction_num)
            write(nunit, *) '-----'
        end if
    if (imax > pr_reaction_num) then
        write(nunit, '(10x,a,6x,a)') 'so_type', 'name'
        do i = max(imin - pr_reaction_num, 1), imax - pr_reaction_num
            write(nunit, '(1x,i4,7x,i4,4x,a)') i + pr_reaction_num, i, so_name(i)
        end do
    end if
else if (ivar ≡ tl_index_pmi) then
    write(nunit, *) ' PMI:'
    write(nunit, *)
    write(nunit, '(a,2x,a,7x,a)') 'pr_pmi', 'PMI', 'name'
    do i = imin, imax
        write(nunit, '(i4,4x,i4,3x,a)') i, pr_pm_ref(i), trim(pm_name(pr_pm_ref(i)))
    end do
else if (ivar ≡ tl_index_material) then
    write(nunit, *) 'Nothing_written_yet_for_material'
else if (ivar ≡ tl_index_source_group) then
    write(nunit, *) 'Nothing_written_yet_for_source_group'
    /* Sector, strata, and strata segment are all more likely to appear within diagnostic. */
else if (ivar ≡ tl_index_sector) then
    write(nunit, *) 'Nothing_written_yet_for_sector'

```

```

else if (ivar ≡ tl_index_strata) then
  write (nunit, *) 'Nothing written yet for strata'
else if (ivar ≡ tl_index_strata_segment) then
  write (nunit, *) 'Nothing written yet for strata segment'
else if (ivar ≡ tl_index_energy_bin) then
  pointer = tally_geometry_ptrjscore
  write (nunit, *) 'ENERGY_BINS for diagnostic', trim(diagnostic_grp_namepointer), ''':
  write (nunit, *)
  write (nunit, '(1x,a,4x,a)') 'bin', 'energy (eV)'
  do i = imin, imax
    energy = diagnostic_minpointer + (areal(i) - half) * diagnostic_deltapointer
    if (diagnostic_spacingpointer ≡ sc_diag_spacing_log)
      energy = exp(energy)
    energy /= electron_charge // J to eV
    write (nunit, '(i4,2x,1pe13.5)') i, energy
  end do
else if (ivar ≡ tl_index_angle_bin) then
  pointer = tally_geometry_ptrjscore
  write (nunit, *) 'ANGLE_BINS for diagnostic', trim(diagnostic_grp_namepointer), ''':
  write (nunit, *)
  write (nunit, '(1x,a,3x,a)') 'bin', 'angle (degrees)'
  do i = imin, imax
    angle = diagnostic_minpointer + (areal(i) - half) * diagnostic_deltapointer
    if (diagnostic_spacingpointer ≡ sc_diag_spacing_log)
      angle = exp(angle)
    angle *= const(1.8, 2) / PI // radians to degrees
    write (nunit, '(i4,3x,1pe13.5)') i, angle
  end do
else if (ivar ≡ tl_index_wavelength_bin) then
  pointer = tally_geometry_ptrjscore
  write (nunit, *) 'WAVELENGTH_BINS for diagnostic', trim(diagnostic_grp_namepointer),
  ''':
  write (nunit, *)
  write (nunit, '(1x,a,3x,a)') 'bin', 'wavelength (Angstroms)'
  do i = imin, imax
    wavelength = detector_minpointer + (areal(i) - 0.5) * detector_deltapointer
    if (detector_spacingpointer ≡ de_spacing_log)
      wavelength = exp(wavelength)
    wavelength *= const(1., 10)
    write (nunit, '(i4,5x,1pe15.7)') i, wavelength
  end do
else if (ivar ≡ tl_index_diagnostic) then
  pointer = tally_geometry_ptrjscore
  write (nunit, *) 'SECTORS for diagnostic', trim(diagnostic_grp_namepointer), ''':
  write (nunit, *) 'Note that in some instances the coordinate and \
  / or area data may not be available.'
  write (nunit, *)
  write (nunit, '(t32,a)') 'stratum'
  write (nunit, '(a,3x,a,3x,a,2x,a,2x,a,9x,a,13x,a,10x,a,7x,a)') 'diagnostic', 'sector',
  'stratum', 'segment', 'midpt:x1', 'x2', 'x3', 'iy', 'area'
  do i = imin, imax
    sec = diagnostic_sector(i, pointer)

```

```

vc_difference(sector_pointssec,sc_neg, sector_pointssec,sc_pos, xdiff)
zone = sector_zonesec
if (geometry_symmetry  $\equiv$  geometry_symmetry_plane) then /* But, by definition of this
    symmetry the problem spans the full length of the universal cell in this direction. */
    circum = universal_cell_max2 - universal_cell_min2
else if (geometry_symmetry  $\equiv$  geometry_symmetry_plane_hw) then
    circum = zone_maxzone,2 - zone_minzone,2
else if (geometry_symmetry  $\equiv$  geometry_symmetry_cylindrical) then
    /* Note that for this case, we follow the original convention of setting zone_minzone,2 =
    zone_maxzone,2 = 0, so use  $\pi$  explicitly. */
    circum = PI * (sector_pointssec,sc_neg,1 + sector_pointssec,sc_pos,1)
else if ((geometry_symmetry  $\equiv$  geometry_symmetry_cyl_hw)  $\vee$  (geometry_symmetry  $\equiv$ 
    geometry_symmetry_cyl_section)) then
    circum = half * (sector_pointssec,sc_neg,1 + sector_pointssec,sc_pos,1) * (zone_maxzone,2 -
    zone_minzone,2)
end if
area = circum * vc_abs(xdiff)

write (nunit, '(2x,i5,6x,i5,4x,i4,6x,i4,1x,3(1p13.5,2x),2x,i3,2x,e13.5)') i, sec,
    stratasec, sector_strata_segmentsec, half * (sector_pointssec,sc_neg,1 + sector_pointssec,sc_pos,1),
    half * (sector_pointssec,sc_neg,2 + sector_pointssec,sc_pos,2),
    half * (sector_pointssec,sc_neg,3 + sector_pointssec,sc_pos,3), zone_index(ziiy, sector_zone(sec)),
    area
end do
else
    write (nunit, *) '_Unknown_independent_variable'
end if

return
end

```

## 2 INDEX

*adddet*: [1.3](#), 1.4.  
*angle*: [1.9](#).  
*area*: [1.9](#).  
*areal*: 1.9.  
*arg\_count*: 1.2.  
*assert*: 1, 1.4, 1.5, 1.9.  
*b*: [1.3](#).  
*browse*: 1.2, [1.3](#).  
*choice*: [1.3](#), 1.4.  
*circum*: [1.9](#).  
*command\_arg*: 1.2.  
*const*: 1.9.  
*de\_common*: 1.9.  
*de\_spacing\_log*: 1.9.  
*de\_view\_end*: 1.9.  
*de\_view\_pointer*: 1.9.  
*de\_view\_points*: 1.9.  
*de\_view\_start*: 1.9.  
*declare\_varp*: 1.3.  
*define\_dimen*: 1.3.  
*define\_varp*: 1.3.  
*degas\_init*: 1.2.  
*details*: [1.3](#), 1.4, 1.5, 1.7.  
*detector\_delta*: 1.9.  
*detector\_min*: 1.9.  
*detector\_name*: 1.9.  
*detector\_spacing*: 1.9.  
*diagnostic*: 1.  
*diagnostic\_delta*: 1.9.  
*diagnostic\_grp\_name*: 1.9.  
*diagnostic\_min*: 1.9.  
*diagnostic\_sector*: 1.9.  
*diagnostic\_spacing*: 1.9.  
*dim*: [1.3](#), 1.4, 1.5, 1.6.  
*diskin*: 1.3, 1.5.  
*diskout*: 1.3, 1.4, 1.5.  
*dump\_indep\_vars*: 1.3, [1.9](#).  
*e*: [1.3](#).  
*electron\_charge*: 1.9.  
*energy*: [1.9](#).  
*eof*: 1.3, 1.4, 1.5.  
*exp*: 1.9.  
*extract\_output\_datum*: [1.3](#), 1.6.  
*FALSE*: 1.4, 1.5, 1.8.  
*file*: 1.3, 1.4, 1.5.  
*FILE*: [1](#).  
*FILELEN*: 1, 1.2, 1.3.

*first\_pass*: [1.3](#), 1.8.  
*FLOAT*: 1.3.  
*geometry\_symmetry*: 1.9.  
*geometry\_symmetry\_cyl\_hw*: 1.9.  
*geometry\_symmetry\_cyl\_section*: 1.9.  
*geometry\_symmetry\_cylindrical*: 1.9.  
*geometry\_symmetry\_plane*: 1.9.  
*geometry\_symmetry\_plane\_hw*: 1.9.  
*geomtesta*: 1.  
*get\_next\_token*: [1](#), 1.3, 1.4, 1.5.  
*gi\_common*: 1.9.  
*group*: [1.3](#), 1.4, 1.5, 1.6.  
*half*: 1.9.  
*i*: [1.3](#), [1.9](#).  
*ii*: [1.3](#), 1.8.  
*imax*: [1.9](#).  
*imin*: [1.9](#).  
*implicit\_none\_f77*: 1.2, 1.3, 1.9.  
*implicit\_none\_f90*: 1.2, 1.3, 1.9.  
*index\_parameters*: [1.3](#), 1.6, 1.8.  
*init*: [1.3](#), 1.4.  
*int\_undef*: 1.4, 1.5, 1.7.  
*ivar*: [1.9](#).  
*i1*: [1.3](#), 1.4, 1.5, 1.8.  
*i2*: [1.3](#), 1.4, 1.5, 1.8.  
*i3*: [1.3](#), 1.4, 1.5.  
*j*: [1.3](#).  
*jscore*: [1.3](#), 1.4, 1.5, 1.6, 1.7, 1.8, [1.9](#).  
*k*: [1.3](#).  
*len*: 1.  
*length*: 1, [1.3](#), 1.4, 1.5.  
*line*: 1, [1.3](#), 1.4, 1.5, 1.7.  
*LINELEN*: 1.3, 1.9.  
*max*: 1.9.  
*max\_cols*: [1](#), 1.8.  
*max\_slice*: [1.3](#), 1.4, 1.5, 1.7, 1.8.  
*min*: 1.9.  
*min\_slice*: [1.3](#), 1.4, 1.5, 1.7, 1.8.  
*nargs*: [1.2](#).  
*nc\_read\_output*: 1.2.  
*next\_line*: [1.1](#), 1.5.  
*next\_tally*: [1.1](#), 1.3.  
*next\_token*: 1, 1.4, 1.5.  
*nunit*: [1.3](#), 1.4, 1.5, 1.8, [1.9](#).  
*o\_mean*: 1.6.  
*o\_var*: 1.6.  
*ou\_common*: 1.3.  
*out\_post\_all*: 1.6.

*out\_post\_grp*: 1.6.  
*output\_browser*: 1.2.  
*outputbrowser*: 1.  
*outputsript*: 1.  
  
*p*: 1.3.  
*parse\_string*: 1.  
*PI*: 1.9.  
*pick\_dim*: 1.1, 1.4.  
*pick\_group*: 1.1, 1.4.  
*pick\_ordering*: 1.1, 1.4.  
*pick\_output*: 1.1, 1.4.  
*pick\_range*: 1.1, 1.4.  
*pick\_tally*: 1.1, 1.4.  
*pm\_common*: 1.9.  
*pm\_name*: 1.9.  
*pointer*: 1.9.  
*pr\_background*: 1.9.  
*pr\_background\_check*: 1.9.  
*pr\_background\_num*: 1.9.  
*pr\_common*: 1.3, 1.9.  
*pr\_pm\_ref*: 1.9.  
*pr\_reaction*: 1.9.  
*pr\_reaction\_num*: 1.9.  
*pr\_test*: 1.9.  
*pr\_test\_check*: 1.9.  
*problem\_sp*: 1.  
*psp\_label*: 1.9.  
  
*range\_loop*: 1.1, 1.4.  
*range\_loop2*: 1.1, 1.5.  
*rc\_common*: 1.9.  
*rc\_name*: 1.9.  
*read\_int\_soft\_fail*: 1.4, 1.5, 1.7.  
*read\_string*: 1.3, 1.4, 1.5.  
*readfilenames*: 1.2.  
*rsd*: 1.3, 1.6, 1.8.  
  
*sc\_common*: 1.9.  
*sc\_diag\_spacing\_log*: 1.9.  
*sc\_neg*: 1.9.  
*sc\_pos*: 1.9.  
*screen*: 1.3, 1.4, 1.5.  
*scriptfile*: 1.2, 1.3, 1.5.  
*sec*: 1.9.  
*sector\_points*: 1.9.  
*sector\_strata\_segment*: 1.9.  
*sector\_zone*: 1.9.  
*set\_verbose*: 1.1, 1.4.  
*so\_check*: 1.4, 1.5, 1.6.  
*so\_common*: 1.3, 1.9.  
*so\_grps*: 1.4, 1.5.  
*so\_name*: 1.9.  
*so\_plate*: 1.9.  
  
*so\_puff*: 1.9.  
*so\_recomb*: 1.9.  
*so\_type\_num*: 1.9.  
*SP*: 1.8.  
*sp\_common*: 1.9.  
*sp\_sy*: 1.9.  
*st\_decls*: 1.3, 1.9.  
*status*: 1.3, 1.4, 1.5.  
*stdin*: 1.3, 1.4.  
*stdout*: 1.2, 1.3, 1.4, 1.5.  
*strata*: 1.9.  
*sy\_decls*: 1.2.  
  
*tally*: 1.  
*tally\_dep\_var\_dim*: 1.3, 1.4, 1.5.  
*tally\_done*: 1.1, 1.5.  
*tally\_geometry\_ptr*: 1.9.  
*tally\_indep\_var*: 1.3, 1.4, 1.8.  
*tally\_name*: 1.3, 1.4, 1.6.  
*tally\_rank*: 1.3, 1.4, 1.5.  
*tally\_tab\_index*: 1.4, 1.5, 1.7.  
*tally\_var\_list*: 1.3, 1.4, 1.8.  
*tallysetup*: 1.  
*tempfile*: 1.3, 1.4, 1.5.  
*test\_tally*: 1.1, 1.5.  
*tl\_check*: 1.4, 1.5.  
*tl\_common*: 1.3, 1.9.  
*tl\_index\_angle\_bin*: 1.9.  
*tl\_index\_detector*: 1.9.  
*tl\_index\_diagnostic*: 1.9.  
*tl\_index\_energy\_bin*: 1.9.  
*tl\_index\_material*: 1.9.  
*tl\_index\_max*: 1.3.  
*tl\_index\_plasma\_zone*: 1.9.  
*tl\_index\_pmi*: 1.9.  
*tl\_index\_problem\_sp*: 1.9.  
*tl\_index\_reaction*: 1.9.  
*tl\_index\_sector*: 1.9.  
*tl\_index\_source\_group*: 1.9.  
*tl\_index\_strata*: 1.9.  
*tl\_index\_strata\_segment*: 1.9.  
*tl\_index\_test*: 1.9.  
*tl\_index\_test\_author*: 1.9.  
*tl\_index\_unknown*: 1.9.  
*tl\_index\_wavelength\_bin*: 1.9.  
*tl\_index\_zone*: 1.9.  
*tl\_num*: 1.4, 1.5.  
*tl\_rank\_max*: 1.3.  
*trim*: 1.3, 1.4, 1.8, 1.9.  
*TRUE*: 1.3, 1.4, 1.5, 1.7, 1.8.  
*type\_string*: 1.9.  
  
*unit*: 1.3, 1.4, 1.5.

*universal\_cell\_max*: 1.9.

*universal\_cell\_min*: 1.9.

*value*: 1.3, 1.6, 1.8.

*value\_ind*: 1.3.

*value\_size*: 1.3.

*var\_alloc*: 1.3.

*var\_realloc*: 1.3.

*vc\_abs*: 1.9.

*vc\_decl*: 1.9.

*vc\_decls*: 1.9.

*vc\_difference*: 1.9.

*verbose*: 1.3, 1.4.

*wavelength*: 1.9.

*xdiff*: 1.9.

*zi\_iy*: 1.9.

*zn\_check*: 1.9.

*zn\_common*: 1.9.

*zn\_volume*: 1.9.

*zone*: 1.9.

*zone\_center*: 1.9.

*zone\_index*: 1.9.

*zone\_max*: 1.9.

*zone\_min*: 1.9.

⟨Functions and subroutines 1.3, 1.9⟩ Used in section 1.2.  
⟨Memory allocation interface 0⟩ Used in section 1.3.  
⟨Read Script 1.5⟩ Used in section 1.3.  
⟨User Input 1.4⟩ Used in section 1.3.  
⟨get datum 1.6⟩ Used in sections 1.3 and 1.8.  
⟨range logic 1.7⟩ Used in sections 1.4 and 1.5.  
⟨write 2D slice 1.8⟩ Used in section 1.3.

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

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

**CHANGE FILE:** (none).

**GLOBAL LANGUAGE:** FORTRAN.