

outputbrowser

November 15, 2001
15:47

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 1.3 2001/07/16 20:57:51 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:

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 afilename
```

§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 scriptfile 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 tl_rank_max, max_slice tl_rank_max, index_parameters tl_index_max, details tl_rank_max
    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_jscore), ' ':'
  if (tally_dep_var_dim_jscore > 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, '(4x,i4,6x,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,i4)') 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 } \equiv

set_verbose: **continue**

```

write(stdout, *)
if (init  $\equiv$  TRUE) then
    write(stdout, *) 'Do you want verbose (v) or terse instructions (t)?'
    if ( $\neg$ 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)  $\equiv$  'v'  $\vee$  line(b : e)  $\equiv$  'V') then
        verbose = TRUE
    else if (line(b : e)  $\equiv$  't'  $\vee$  line(b : e)  $\equiv$  'T') then
        verbose = FALSE
    else
        write(stdout, *) 'Type a "v" or a "t",'
        go to set_verbose
    end if
end if
if (verbose  $\equiv$  TRUE  $\vee$  init  $\equiv$  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_namejscore)
    end do
    write(stdout, *)
    write(stdout, *) 'Choose one; use 0 to exit'
else
    write(stdout, *) 'Choose next tally:'
end if

pick_tally: continue

if ( $\neg$ read_string(stdin, line, length))
    goto eof
get_next_token
jscore = read_int_soft_fail(line(b : e))
if (jscore  $\equiv$  0)
    go to eof // Quitting
if ( $\neg$ 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  $\equiv$  TRUE  $\vee$  init  $\equiv$  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

pick_group: continue
    if ( $\neg$ read_string(stdin, line, length))
        goto eof
    get_next_token
    group = read_int_soft_fail(line(b : e))
    if ( $\neg$ so_check(group)  $\wedge$  group  $\neq$  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_jscore > 0) then
    if (verbose  $\equiv$  TRUE) then
        write(stdout, *) ' ', trim(tally_name_jscore), ' is of rank: ', tally_rank_jscore
        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_jscore
        write(stdout,
            '(3x,i4,6x,i4,4x,a)') i, tally_tab_index_jscore,i, trim(tally_var_list_tally_indep_var_jscore,i)
    end do

    write(stdout, *)
    do i = 1, tally_rank_jscore
        if (verbose  $\equiv$  TRUE) then
            write(stdout, *) ' Enter range indices, separated by a space for ',
            trim(tally_var_list_tally_indep_var_jscore,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_jscore,i), ' '
        end if
    end if

pick_range: continue
    if ( $\neg$ 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 ( $max\_slice_i \equiv int\_undef$ )
   $max\_slice_i = min\_slice_i$ 

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, *) '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 ( $tally\_rank_{jscore} \equiv 0 \vee tally\_rank_{jscore} \equiv 1$ ) then
  i1 = 1
else
  write (stdout, *)
  write (stdout,
    *) 'Would you like to re-order the independent variables for printing? (y/n)'

if ( $\neg read\_string(stdin, line, length)$ )
  go to eof
get_next_token
choice = line(b : e)
if (choice  $\neq$  'y'  $\wedge$  choice  $\neq$  'Y') then
  i1 = 1
  i2 = 2
  i3 = 3
else
  if (verbose  $\equiv$  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 ( $\neg read\_string(stdin, line, length)$ )
    goto eof
  get_next_token
  i1 = read_int_soft_fail(line(b : e))
  if ( $tally\_rank_{jscore} \geq 2$ ) then
    assert(next_token(line, b, e, p))
    i2 = read_int_soft_fail(line(b : e))
  end if
  if ( $tally\_rank_{jscore} \geq 3$ ) then
    assert(next_token(line, b, e, p))
    i3 = read_int_soft_fail(line(b : e))
  end if

```

```

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, *) 'These integers must all be between 1 and', tally_rankjscore,
  ' ;Try again',
  go to pick_ordering
  end if
end if
end if

pick_dim: continue
if (tally_dep_var_dimjscore ≡ 1) then
  dim = 1
else
  write (stdout, *) 'This tally is not a scalar quantity. Enter the dimension to be printed, 1 to', tally_dep_var_dimjscore
  if ( $\neg \text{read\_string}(\text{stdin}, \text{line}, \text{length})$ )
    goto eof
  get_next_token
  dim = read_int_soft_fail (line(b : e))
  if (dim < 1  $\vee$  dim > tally_dep_var_dimjscore) then
    write (stdout, *) 'Try again; pick a number between 1 and', tally_dep_var_dimjscore
    go to pick_dim
  end if
end if

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

```

```

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.

$\langle \text{Read Script 1.5} \rangle \equiv$

```

next_line: continue

  if ( $\neg \text{read\_string}(\text{diskin}, \text{line}, \text{length})$ ) then
    close (unit = diskin)
    scriptfile = '''
    go to tally_done
  else
    get_next_token
    if (line(b : e)  $\equiv$  'Tally') then
      assert (next_token(line, b, e, p))
      jscore = read_int_soft_fail(line(b : e))
    else if (line(b : e)  $\equiv$  'Group') then
      assert (next_token(line, b, e, p))
      group = read_int_soft_fail(line(b : e))
    else if (line(b : e)  $\equiv$  '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)  $\equiv$  ':')
      assert (next_token(line, b, e, p))
      min_slicei = int_undef
      max_slicei = int_undef
      detailsi = FALSE
    range_loop2: continue  $\langle \text{range logic 1.7} \rangle$ 
    if (next_token(line, b, e, p))
      go to range_loop2
    if (max_slicei  $\equiv$  int_undef)
      max_slicei = min_slicei
    else if (line(b : e)  $\equiv$  'Dimension') then
      assert (next_token(line, b, e, p))
      dim = read_int_soft_fail(line(b : e))
    else if (line(b : e)  $\equiv$  '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)  $\equiv$  'Screen') then
      screen = TRUE
      nunit = stdout
    else if (line(b : e)  $\equiv$  '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)  $\equiv$  '-') 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 ≡ 0)
    go to eof      // Quitting
if (¬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 (¬so_check(group) ∧ group ≠ 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_rankjscore
    if (min_slicei < 1 ∨ min_slicei > tally_tab_indexjscore,i ∨ max_slicei < 1 ∨ max_slicei >
        tally_tab_indexjscore,i ∨ min_slicei > max_slicei) then
        write (stdout, *)
        write (stdout, *) 'Invalid range for tally', jscore, ', rank', i
        write (stdout, *) 'Range limits must be between 1 and', tally_tab_indexjscore,i
        go to next_line
    end if
end do

if (tally_rankjscore ≥ 2) then
    if (i1 < 1 ∨ i1 > tally_rankjscore ∨ (tally_rankjscore ≥ 2 ∧ (i2 < 1 ∨ i2 >
        tally_rankjscore)) ∨ (tally_rankjscore ≥ 3 ∧ (i3 < 1 ∨ i3 > tally_rankjscore))) 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_rankjscore
        go to next_line
    end if
end if

if (dim < 1 ∨ dim > tally_dep_var_dimjscore) then
    if (tally_dep_var_dimjscore ≤ 1) then
        dim = 1
    else
        write (stdout, *)
        write (stdout, *) 'Invalid dimension:', dim, 'for tally', jscore
        go to next_line
    end if
end if

```

This code is used in section 1.3.

§1.6–§1.7 [#7–#8] **outputbrowserA** program to allow interactive browsing and printing of DEGAS 2's output netCDF file

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  $j = \text{min\_slice}_{i2}, \text{ max\_slice}_{i2}$ 
     $\text{index\_parameters}_{\text{tally\_indep\_var}_{jscore,i2}} = j$ 
    do  $i = \text{min\_slice}_{i1}, \text{ max\_slice}_{i1}$ 
       $\text{index\_parameters}_{\text{tally\_indep\_var}_{jscore,i1}} = i$ 
      ⟨ get datum 1.6 ⟩
    end do
    if ( $\text{max\_slice}_{i1} - \text{min\_slice}_{i1} + 1 < \text{max\_cols} \wedge \text{max\_slice}_{i1} > \text{min\_slice}_{i1}$ ) then
      if ( $j \equiv \text{min\_slice}_{i2}$ ) then
        write (nunit, '(12x,a,a3)') trim(tally_var_listtally_indep_varjscore,i1), ',->
        write (nunit, '(6x,20(6x,i4,5x))') SP(ii, ii = min_slicei1, max_slicei1)
        write (nunit, '(a)') trim(tally_var_listtally_indep_varjscore,i2)
      end if
      write (nunit, '(i4,2x,1p,20(2x,e13.5))') j, (valueii, ii = min_slicei1, max_slicei1)
    else
      if (first_pass ≡ TRUE) then
        write (nunit, '(a10,2x,a10,5x,a,t39,a)') trim(tally_var_listtally_indep_varjscore,i1),
          trim(tally_var_listtally_indep_varjscore,i2), 'value', 'rel.std.dev.'
        first_pass = FALSE
      end if
      do  $i = \text{min\_slice}_{i1}, \text{ max\_slice}_{i1}$ 
        write (nunit, '(4x,i4,6x,i4,6x,1pe13.5,2x,0pf7.4)') i, j, valuei, rsdi
      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
  implicit none_f90
  st_decls

  integer nunit, jscore, ivar, imin, imax // Input
  integer pointer, i, sec // Local
  real energy, angle, wavelength
  character*LINELEN type_string
  character*20 psp_label

  /* 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, *) 'ZZONES:' 
    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, '(i4,2x,1p,4(e13.5,2x))') i, zone_centeri,1, zone_centeri,2, zone_centeri,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_ptr_jscore
    write(nunit, *) 'DETECTOR', trim(detector_name_pointer), ':'
    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_points de_view_pointer(i, pointer), de_view_start,1,
            de_view_points de_view_pointer(i, pointer), de_view_start,2,
            de_view_points de_view_pointer(i, pointer), de_view_start,3,
            de_view_points de_view_pointer(i, pointer), de_view_end,1,
            de_view_points de_view_pointer(i, pointer), de_view_end,2,
            de_view_points de_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
#endifif
  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, *)
    write(nunit, '(t30,a)') 'stratum'
    write(nunit, '(a,2x,a,2x,a,2x,a,2x,a,9x,a,13x,a)') 'diagnostic', 'sector', 'stratum',
    'segment', 'midpt:x1', 'x2', 'x3'
    do i = imin, imax
        sec = diagnostic_sector(i, pointer)
        write(nunit, '(2x,i4,6x,i4,4x,i4,6x,i4,1x,3(1pe13.5,2x))') 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),

```

§1.9 [#10] **outputbrowser**A program to allow interactive browsing and printing of DEGAS 2's output netCDF file

```
    half * (sector-pointssec,sc-neg,3 + sector-pointssec,sc-pos,3)  
end do  
else  
    write (nunit, *) 'Unknown independent variable'  
end if  
return  
end
```

2 INDEX

adddet: [1.3](#), [1.4](#).
angle: [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](#).
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](#).

geomtesta: [1](#).
get_next_token: [1](#), [1.3](#), [1.4](#), [1.5](#).
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.
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.

value: 1.3, 1.6, 1.8.
value_ind: 1.3.
value_size: [1.3](#).
var_alloc: 1.3.
var_realloc: 1.3.
verbose: [1.3](#), 1.4.

wavelength: [1.9](#).

zn_check: 1.9.
zn_common: 1.9.

zn-volume: 1.9.
zone-center: 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.