`problemsetup`

November 24, 2009
13:45

Contents

1 Set up file containing data specific to the current problem **1**

2 References **38**

3 INDEX **39**

1 Set up file containing data specific to the current problem

$Id: problemsetup.web,v 1.36 2007/06/19 17:41:28 dstotler Exp $

There are two levels of physics input to DEGAS 2. The input files set up by *datasetup* comprise the reference level: in practice, the sum total of the data available to the code (although in principle it could be smaller). The second level is the problem level of data. This prescribes the species, reactions, materials, and PMI which will serve as the physical model to be used in carrying out the simulation at hand. In some parts of the (internal) code, these are also referred to as the subset data since they represent a subset of the reference data.

At the problem level the concepts of test species and background species are introduced. Most simply, test species are the ones DEGAS 2 will track as they collide off of background species. The use of the word species here is important: both of the test and background lists are subsets of the "species" list. One more precise distinction between the two species types is that we assume that we know the distribution function (in space and velocity) of the background species; in fact, such information is required input to DEGAS 2. On the other hand, we are attempting to *compute* the test species distribution function, moments of which serve as the primary output of DEGAS 2.

The problem input file associated with the symbolic name in `problem_infile` in `degas2.in` consists of five sections, one for each of the problem-level physics subsets, separated by the corresponding heading (in capital letters). It is the object symbols (as opposed to the longer "names") which are used in these lists. For example,

```
TEST

0 D D2 D2+

BACKGROUND

e D+

REACTION

hchex
hionize
h2dis
h2ion
h2disión
h2pdision
h2pdis
h2pdisrec

MATERIALS

mo
mirror

PMI

dreflmo
```

```
hdesorbmo
h2desorbmo
hmirror
h2mirror
```

This file is read by *problemsetup*. The reference lists are searched for these symbols. Associations between the subset lists which are needed by the code for tracking the test species are built up. The required atomic and surface data are read in and repackaged for efficient run-time use. All of the resulting information is written to the netCDF problem file, which corresponds to the symbolic name *problemfile* in `degas2.in`.

"problemsetup.f" 1 ≡
@m FILE 'problemsetup.web'

The unnamed module.

"problemsetup.f" 1.1 ≡

```
    program problemsetup
      implicit_none_f77
      implicit_none_f90
@#if SUN ∧ FORTRAN77
      call float_abort
@#endif
      call readfilenames
      call read_problem

      call init_var0_list

      call init_reaction
      call init_pmi

      call finish_var0_list

      call nc_write_problem

      stop
    end

    ⟨ Functions and Subroutines 1.2 ⟩
```

Read in data from `problem_infile`.

```
⟨ Functions and Subroutines 1.2 ⟩ ≡
    subroutine read_problem
      implicit_none_f77
      pr_common
      rf_common
      implicit_none_f90

      integer length      // Local
      character*LINELEN line
      character*FILELEN tempfile

      ⟨ Memory allocation interface 0 ⟩      // Common
      st_decls

      problem_version = 'unknown'
      tempfile = filenames_array_{problem_infile}
      assert(tempfile ≠ char_undef)
      open(unit = diskin, file = tempfile, form = 'formatted', status = 'old')
      assert(read_string(diskin, line, length))
      assert(length ≤ len(line))
      problem_version = line(: length)

      call init_problem
      if (¬read_string(diskin, line, length)) then
        line = 'END'
      end if

loop1: continue

      if (line ≡ 'END')
        goto eof
      if (line ≡ 'TEST') then
        call read_test(diskin, line)
      else if (line ≡ 'BACKGROUND') then
        call read_background(diskin, line)
      else if (line ≡ 'REACTION') then
        call read_reaction(diskin, line)
      else if (line ≡ 'MATERIALS') then
        call read_materials(diskin, line)
      else if (line ≡ 'PMI') then
        call read_pmi(diskin, line)
      end if

      goto loop1

  eof: continue

      call check_problem

      return
    end
```

See also sections 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.21, 1.22, 1.23, 1.24, 1.25, and 1.26.

This code is used in section 1.1.

Initialize problem.

```
⟨ Functions and Subroutines 1.2 ⟩ +≡
    subroutine init_problem
      implicit_none_f77
      pr_common      // Common
      sp_common
      ps_common
      implicit_none_f90
      integer i      // Local

      ⟨ Memory allocation interface 0 ⟩

      call nc_read_elements
      call nc_read_species
      call nc_read_reactions
      call nc_read_materials
      call nc_read_pmi

      pr_test_num = 0
      pr_background_num = 0
      pr_reaction_num = 0
      pr_materials_num = 0
      pr_pmi_num = 0

      num_generics = 0
      var_alloc(generics)
      do i = 1, sp_num
        generics_i = 0
      end do

      return
    end
```

Read in test particle labels; make required entries in generic species arrays.

⟨ Functions and Subroutines 1.2 ⟩ +≡

```
subroutine read_test(unit, line)
   implicit_none_f77
   pr_common      // Common
   sp_common
   ps_common
   implicit_none_f90
   integer unit      // Input
   character*(*) line      // Output
   integer length, p, b, e, i      // Local

   ⟨ Memory allocation interface 0 ⟩
   st_decls

   var_alloc(problem_species_test)
   do i = 1, sp_num
      problem_species_test_i = 0
   end do

loop1: continue
   if (¬read_string(unit, line, length))
      line = 'END'
   if (line ≡ 'END' ∨ line ≡ 'TEST' ∨ line ≡ 'BACKGROUND' ∨ line ≡ 'REACTION' ∨ line ≡
         'MATERIALS' ∨ line ≡ 'PMI' ∨ line ≡ 'SCORES')
      goto eof
   assert(length ≤ len(line))
   length = parse_string(line(: length))

   p = 0
loop2: continue
   if (¬next_token(line(: length), b, e, p))
      goto loop1
   pr_test_num++
   var_realloca(problem_test_sp)
   pr_test(pr_test_num) = sp_lookup(line(b : e))
   assert(pr_test(pr_test_num) > 0)
   problem_species_test(pr_test(pr_test_num)) = pr_test_num

   if (generics_{sp_generic(pr_test(pr_test_num))} ≡ 0) then      /* First appearance of this generic species;
         update num_generics and generics. In case this entry in the test list is not the archetype
         generic of its isotopic family (i.e., the value of sp_generic), need to also enter num_generics in
         its slot in the generics array. E.g., if D2 were in a problem, but the archetype generic H2 were
         not. */
      num_generics++
      generics_{sp_generic(pr_test(pr_test_num))} = num_generics

      if (pr_test(pr_test_num) ≠ sp_generic(pr_test(pr_test_num)))
         generics_{pr_test(pr_test_num)} = num_generics

   else      /* This entry is equivalent to a generic species which has already been added to the generics
         array. */
      generics_{pr_test(pr_test_num)} = generics_{sp_generic(pr_test(pr_test_num))}
   end if

   goto loop2
```

eof : **return**
end

Read in background particle labels; make required entries in generic species arrays.

```
⟨Functions and Subroutines 1.2⟩ +≡
    subroutine read_background(unit, line)
      implicit_none_f77
      pr_common    // Common
      sp_common
      ps_common
      implicit_none_f90
      integer unit    // Input
      character∗(∗) line    // Output
      integer length, p, b, e, i    // Local

      ⟨Memory allocation interface 0⟩
      st_decls

      var_alloc(problem_species_background)
      do i = 1, sp_num
        problem_species_background_i = 0
      end do

loop1: continue
      if (¬read_string(unit, line, length))
        line = 'END'
      if (line ≡ 'END' ∨ line ≡ 'TEST' ∨ line ≡ 'BACKGROUND' ∨ line ≡ 'REACTION' ∨ line ≡
            'MATERIALS' ∨ line ≡ 'PMI' ∨ line ≡ 'SCORES')
        goto eof

      assert(length ≤ len(line))
      length = parse_string(line)

      p = 0
loop2: continue
      if (¬next_token(line(: length), b, e, p))
        goto loop1
      pr_background_num++
      var_realloca(problem_background_sp)
      pr_background(pr_background_num) = sp_lookup(line(b : e))
      assert(pr_background(pr_background_num) > 0)
      problem_species_background(pr_background(pr_background_num)) = pr_background_num

        /∗ Handle generics just as we did for test species: ∗/
      if (generics_{sp_generic(pr_background(pr_background_num))} ≡ 0) then
        num_generics++
        generics_{sp_generic(pr_background(pr_background_num))} = num_generics
        if (pr_background(pr_background_num) ≠ sp_generic(pr_background(pr_background_num)))
          generics_{pr_background(pr_background_num)} = num_generics
      else
        generics_{pr_background(pr_background_num)} = generics_{sp_generic(pr_background(pr_background_num))}
      end if

      goto loop2

  eof: return
    end
```

Read in reaction particle labels.

```
⟨Functions and Subroutines 1.2⟩ +≡
    subroutine read_reaction(unit, line)
      implicit_none_f77
      pr_common    // Common
      rc_common
      implicit_none_f90
      integer unit    // Input
      character*(*) line    // Output
      integer length, p, b, e    // Local

      ⟨Memory allocation interface 0⟩
      st_decls

      pr_reaction_dim = 0
      assert(pr_reaction_num ≡ 0)
loop1: continue
      if (¬read_string(unit, line, length))
        line = 'END'
      if (line ≡ 'END' ∨ line ≡ 'TEST' ∨ line ≡ 'BACKGROUND' ∨ line ≡ 'REACTION' ∨ line ≡
            'MATERIALS' ∨ line ≡ 'PMI' ∨ line ≡ 'SCORES')
        goto eof

      assert(length ≤ len(line))
      length = parse_string(line)

      p = 0
loop2: continue
      if (¬next_token(line(: length), b, e, p))
        goto loop1
      pr_reaction_dim++
      var_realloca(problem_rc)
      pr_reaction(pr_reaction_dim) = rc_lookup(line(b : e))
      assert(pr_reaction(pr_reaction_dim) > 0)
      goto loop2

  eof: continue

      pr_reaction_num = pr_reaction_dim
      if (pr_reaction_dim ≡ 0) then
        pr_reaction_dim = 1
        var_realloca(problem_rc)
      end if

      return
    end
```

Read in materials labels.

```
⟨ Functions and Subroutines 1.2 ⟩ +≡
   subroutine read_materials(unit, line)
     implicit_none_f77
     pr_common    // Common
     ma_common
     implicit_none_f90
     integer unit    // Input
     character∗(∗) line    // Output
     integer length, p, b, e, i    // Local

     ⟨ Memory allocation interface 0 ⟩
     st_decls

     var_alloc(problem_materials_sub)
     do i = 1, ma_num
       problem_materials_sub_i = 0
     end do

loop1: continue
     if (¬read_string(unit, line, length))
       line = 'END'
     if (line ≡ 'END' ∨ line ≡ 'TEST' ∨ line ≡ 'BACKGROUND' ∨ line ≡ 'REACTION' ∨ line ≡
           'MATERIALS' ∨ line ≡ 'PMI' ∨ line ≡ 'SCORES')
       goto eof

     assert(length ≤ len(line))
     length = parse_string(line)

     p = 0
loop2: continue
     if (¬next_token(line(: length), b, e, p))
       goto loop1
     pr_materials_num++
     var_realloca(problem_materials_ref)
     pr_mat_ref(pr_materials_num) = ma_lookup(line(b : e))
     assert(pr_mat_ref(pr_materials_num) > 0)
     problem_materials_sub(pr_mat_ref(pr_materials_num)) = pr_materials_num

     goto loop2

  eof: return
   end
```

Read in plasma-materials interaction labels.

```
⟨Functions and Subroutines 1.2⟩ +≡
    subroutine read_pmi(unit, line)
      implicit_none_f77
      pr_common    // Common
      pm_common
      implicit_none_f90
      integer unit    // Input
      character*(*) line    // Output
      integer length, p, b, e, i    // Local

      ⟨Memory allocation interface 0⟩
      st_decls

      var_alloc(problem_pmi_sub)
      do i = 1, pm_num
        problem_pmi_sub_i = 0
      end do

loop1: continue
      if (¬read_string(unit, line, length))
        line = 'END'
      if (line ≡ 'END' ∨ line ≡ 'TEST' ∨ line ≡ 'BACKGROUND' ∨ line ≡ 'REACTION' ∨ line ≡
            'MATERIALS' ∨ line ≡ 'PMI' ∨ line ≡ 'SCORES')
        goto eof

      assert(length ≤ len(line))
      length = parse_string(line)

      p = 0
loop2: continue
      if (¬next_token(line(: length), b, e, p))
        goto loop1
      pr_pmi_num++
      var_realloca(problem_pmi_ref)
      pr_pm_ref(pr_pmi_num) = pm_lookup(line(b : e))
      assert(pr_pm_ref(pr_pmi_num) > 0)
      problem_pmi_sub(pr_pm_ref(pr_pmi_num)) = pr_pmi_num

      goto loop2

  eof: return
    end
```

Setup initial entries in list of dependent variables. This routine does two of the three "sections" of this array: hardwired (entries corresponding to macros) and background terms (repeated for each background species). The reaction-specific entries will be set in *xs_copy* as needed.

⟨ Functions and Subroutines 1.2 ⟩ +≡

```
subroutine init_var0_list
  implicit_none_f77
  pr_common    // Common
  sp_common
  implicit_none_f90

  integer i    // Local

  ⟨ Memory allocation interface 0 ⟩

  pr_var0_num = 1    // Will force mem_alloc to be called.
  var_alloc(pr_var0_list)
  pr_var0_num = pr_var_mass_change − 1 + pr_num_change_vars ∗ (pr_background_num + pr_test_num + 1)
    // The last "1" here accounts for the generic entries.
@#if 0
  var_realloca(pr_var0_list)    // Need here since will be using later also
@#endif
  var_realloc(pr_var0_list, 1, mem_size(pr_var0_num))
  do i = 1, pr_var0_num
    pr_var0_list_i = char_uninit
  end do

  pr_var0_list_pr_var_unknown = 'unknown'
  pr_var0_list_pr_var_mass = 'mass'
  pr_var0_list_pr_var_momentum_vector = 'momentum_vector'
  pr_var0_list_pr_var_momentum_2 = 'momentum_2'
  pr_var0_list_pr_var_momentum_3 = 'momentum_3'
  pr_var0_list_pr_var_energy = 'energy'
  pr_var0_list_pr_var_angle = 'angle'
  pr_var0_list_pr_var_emitter_v_vector = 'emitter_v_vector'
  pr_var0_list_pr_var_emitter_v_2 = 'emitter_v_2'
  pr_var0_list_pr_var_emitter_v_3 = 'emitter_v_3'
  pr_var0_list_pr_var_emitter_vf_Maxwell_vector = 'emitter_vf_Maxwell_vector'
  pr_var0_list_pr_var_emitter_vf_Maxwell_2 = 'emitter_vf_Maxwell_2'
  pr_var0_list_pr_var_emitter_vf_Maxwell_3 = 'emitter_vf_Maxwell_3'
  pr_var0_list_pr_var_emitter_vth_Maxwell = 'emitter_vth_Maxwell'
  pr_var0_list_pr_var_xz_stress = 'xz_stress'    // For Couette example

  pr_var0_list_pr_var_mass_change = 'mass_change'    // Generic entries
  pr_var0_list_pr_var_momentum_change_vector = 'momentum_change_vector'
  pr_var0_list_pr_var_momentum_change_2 = 'momentum_change_2'
  pr_var0_list_pr_var_momentum_change_3 = 'momentum_change_3'
  pr_var0_list_pr_var_energy_change = 'energy_change'
  pr_var0_list_pr_var_mass_in = 'mass_in'
  pr_var0_list_pr_var_momentum_in_vector = 'momentum_in_vector'
  pr_var0_list_pr_var_momentum_in_2 = 'momentum_in_2'
  pr_var0_list_pr_var_momentum_in_3 = 'momentum_in_3'
  pr_var0_list_pr_var_energy_in = 'energy_in'
  pr_var0_list_pr_var_mass_out = 'mass_out'
  pr_var0_list_pr_var_momentum_out_vector = 'momentum_out_vector'
  pr_var0_list_pr_var_momentum_out_2 = 'momentum_out_2'
```

$pr_var0_list_{pr_var_momentum_out_3}$ = 'momentum_out_3'
$pr_var0_list_{pr_var_energy_out}$ = 'energy_out'

$assert(pr_background_num \geq 1)$
do $i = 1,\ pr_background_num$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_mass_change,\ pr_problem_sp_back(i))}$ = 'back_mass_change_' ‖ $sp_sy(pr_background(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_momentum_change_vector,\ pr_problem_sp_back(i))}$ = 'back_momentum_change_vector_' ‖ $sp_sy(pr_background(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_momentum_change_2,\ pr_problem_sp_back(i))}$ = 'back_momentum_change_2_' ‖ $sp_sy(pr_background(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_momentum_change_3,\ pr_problem_sp_back(i))}$ = 'back_momentum_change_3_' ‖ $sp_sy(pr_background(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_energy_change,\ pr_problem_sp_back(i))}$ = 'back_energy_change_' ‖ $sp_sy(pr_background(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_mass_in,\ pr_problem_sp_back(i))}$ = 'back_mass_in_' ‖ $sp_sy(pr_background(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_momentum_in_vector,\ pr_problem_sp_back(i))}$ = 'back_momentum_in_vector_' ‖ $sp_sy(pr_background(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_momentum_in_2,\ pr_problem_sp_back(i))}$ = 'back_momentum_in_2_' ‖ $sp_sy(pr_background(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_momentum_in_3,\ pr_problem_sp_back(i))}$ = 'back_momentum_in_3_' ‖ $sp_sy(pr_background(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_energy_in,\ pr_problem_sp_back(i))}$ = 'back_energy_in_' ‖ $sp_sy(pr_background(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_mass_out,\ pr_problem_sp_back(i))}$ = 'back_mass_out_' ‖ $sp_sy(pr_background(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_momentum_out_vector,\ pr_problem_sp_back(i))}$ = 'back_momentum_out_vector_' ‖ $sp_sy(pr_background(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_momentum_out_2,\ pr_problem_sp_back(i))}$ = 'back_momentum_out_2_' ‖ $sp_sy(pr_background(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_momentum_out_3,\ pr_problem_sp_back(i))}$ = 'back_momentum_out_3_' ‖ $sp_sy(pr_background(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_energy_out,\ pr_problem_sp_back(i))}$ = 'back_energy_out_' ‖ $sp_sy(pr_background(i))$
end do
$assert(pr_test_num \geq 1)$
do $i = 1,\ pr_test_num$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_mass_change,\ pr_problem_sp_test(i))}$ = 'test_mass_change_' ‖ $sp_sy(pr_test(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_momentum_change_vector,\ pr_problem_sp_test(i))}$ = 'test_momentum_change_vector_' ‖ $sp_sy(pr_test(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_momentum_change_2,\ pr_problem_sp_test(i))}$ = 'test_momentum_change_2_' ‖ $sp_sy(pr_test(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_momentum_change_3,\ pr_problem_sp_test(i))}$ = 'test_momentum_change_3_' ‖ $sp_sy(pr_test(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_energy_change,\ pr_problem_sp_test(i))}$ = 'test_energy_change_' ‖ $sp_sy(pr_test(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_mass_in,\ pr_problem_sp_test(i))}$ = 'test_mass_in_' ‖ $sp_sy(pr_test(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_momentum_in_vector,\ pr_problem_sp_test(i))}$ = 'test_momentum_in_vector_' ‖ $sp_sy(pr_test(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_momentum_in_2,\ pr_problem_sp_test(i))}$ =

'test_momentum_in_2_' $\|$ $sp_sy(pr_test(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_momentum_in_3,\ pr_problem_sp_test(i))} =$
'test_momentum_in_3_' $\|$ $sp_sy(pr_test(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_energy_in,\ pr_problem_sp_test(i))} =$ 'test_energy_in_' $\|$
$sp_sy(pr_test(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_mass_out,\ pr_problem_sp_test(i))} =$ 'test_mass_out_' $\|$
$sp_sy(pr_test(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_momentum_out_vector,\ pr_problem_sp_test(i))} =$
'test_momentum_out_vector_' $\|$ $sp_sy(pr_test(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_momentum_out_2,\ pr_problem_sp_test(i))} =$
'test_momentum_out_2_' $\|$ $sp_sy(pr_test(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_momentum_out_3,\ pr_problem_sp_test(i))} =$
'test_momentum_out_3_' $\|$ $sp_sy(pr_test(i))$
$pr_var0_list_{pr_var_problem_sp_index(pr_var_energy_out,\ pr_problem_sp_test(i))} =$ 'test_energy_out_' $\|$
$sp_sy(pr_test(i))$

end do

return

end

Finish assignments to the list of dependent variable labels. This routine will serve to make an additional assignments to *pr_var0_list* beyond those made by *xs_copy* and to clean up its memory allocation.

The only task carried out here at present is to set up additional emission rate variables corresponding to each wavelength. Here's the long-winded explanation of why this is needed: "In general" the emission rate and wavelength variables should both have the name of the line appended to the variable names. For the case of greatest interest, this would entail carrying around 3 sets of (usually) identical data - one for each hydrogen isotope. Alternatively, the problem is that we're saying "all of the atomic physics for the hydrogen isotopes is the same except the wavelength"; the fact that there is an exception strictly speaking invalidates the notion of generic species. However, the other payoffs of generic species heavily outweigh this one exception. Here, the data files carry a generic name and multiple wavelengths. Each of these should now be in the *pr_var0_list*. However, the scoring dependent variables will refer to emission rates for explicit lines. We define here those additional emission rates, one for each wavelength. The individual reaction-processing subroutines will need to make the connection between the generic emission rate variable and the specific one based on the computed value of the wavelength. If needed, this procedure could be easily extended to handle other wavelength-dependent data.

```
"problemsetup.f" 1.10 ≡
  @m max_lines 3

⟨ Functions and Subroutines 1.2 ⟩ +≡
    subroutine finish_var0_list

      implicit_none_f77
      pr_common

      implicit_none_f90

      integer i, i_e_rate, num_lines, old_num      // Local
      character*pr_tag_string_length line, lines_{max_lines}

      st_decls

      ⟨ Memory allocation interface 0 ⟩

      num_lines = 0
      old_num = pr_var0_num
      do i = 1, pr_var0_num
        if (pr_var0_list_i SP(1 : 10) ≡ 'wavelength') then
          line = pr_var0_list_i SP(11 :)
          i_e_rate = string_lookup('emission_rate' ‖ line, pr_var0_list, pr_var0_num)
          if (i_e_rate ≡ 0) then      // Add to list if not there already
            num_lines++
            pr_var0_num++
            var_realloca(pr_var0_list)
            lines_{num_lines} = line
          end if
        end if
      end do

      if (num_lines > 0) then
    @#if 0
        var_realloc(pr_var0_list, pr_var0_num, pr_var0_num + num_lines)
    @#endif
    @#if 0
        pr_var0_num += num_lines
        var_realloca(pr_var0_list)
```

```
@#endif
    do i = 1, num_lines
      pr_var0_list(old_num + i) = 'emission_rate' // lines(i)
    end do
  end if

  var_reallocb(pr_var0_list)

  return
end
```

Check problem specification. This routine sets up the arrays which will actually be used to implement reactions and PMI in the run. This involves not only matching up reactions and PMI with the test species, but also allowing for the isotopically equivalent combinations.

⟨ Functions and Subroutines 1.2 ⟩ +≡

```
subroutine check_problem
  implicit_none_f77
  pr_common    // Common
  sp_common
  rc_common
  ma_common
  pm_common
  ps_common
  implicit_none_f90
  integer i, j, k, l, m, ts_reagent, bk_reagent    // Local

  pr_test_decl(test)
  pr_background_decl(background)
  pr_generic_decl(generic)
  pr_generic_decl(generic_reagents_{pr_bkrc_reagent_max})
  pr_background_decl(back_{pr_bkrc_reagent_max})

  sp_decl(reagents_{rc_reagent_max})

  ⟨ Memory allocation interface 0 ⟩
  st_decls

  var_reallocb(problem_test_sp)
  var_reallocb(problem_background_sp)
  var_reallocb(problem_rc)
  var_reallocb(problem_materials_ref)
  var_reallocb(problem_pmi_ref)

  write (stdout, *) 'Number of:'
  write (stdout, *) '  test species = ', pr_test_num
  write (stdout, *) '  background species = ', pr_background_num
  write (stdout, *) '  materials = ', pr_materials_num
  write (stdout, *) '  reactions = ', pr_reaction_num
  write (stdout, *) '  plasma-material interactions = ', pr_pmi_num
  write (stdout, *) 'Test species are:'
  do i = 1, pr_test_num
    write (stdout, *) pr_test(i), ', ', trim(sp_name(pr_test(i)))
  end do
  write (stdout, *) 'Background species are:'
  do i = 1, pr_background_num
    write (stdout, *) pr_background(i), ', ', trim(sp_name(pr_background(i)))
  end do
  write (stdout, *) 'Reference material indices are:'
  do i = 1, pr_materials_num
    write (stdout, *) pr_mat_ref(i), ', ', trim(ma_name(pr_mat_ref(i)))
  end do
  write (stdout, *) 'Reference reaction indices are:'
  if (pr_reaction_num > 0) then
    do i = 1, pr_reaction_num
      write (stdout, *) pr_reaction(i), ', ', trim(rc_name(pr_reaction(i)))
    end do
```

```
end if
write (stdout, *) 'Reference␣PMI␣indices␣are:'
do i = 1, pr_pmi_num
  write (stdout, *) pr_pm_ref(i), ',␣', trim(pm_name(pr_pm_ref(i)))
end do

var_alloc(problem_reaction_num)
var_alloc(problem_test_reaction)
var_alloc(problem_test_background)
var_alloc(problem_pmi_case_num)
var_alloc(problem_pmi_cases)
var_alloc(problem_pmi_num_arrange)
var_alloc(problem_pmi_prod_mult)
var_alloc(problem_pmi_products)
var_alloc(problem_num_arrangements)
var_alloc(problem_prod_mult)
var_alloc(problem_test_products)

assert(pr_max_equiv ≥ max(pr_test_num, pr_background_num))
var_alloc(num_equiv)
var_alloc(equivalents)

do i = 1, pr_test_num
  do j = 1, pr_reaction_max
    pr_ts_rc(i, j) = int_unused
    pr_ts_bk(i, j) = int_unused
    pr_num_arrangements(i, j) = int_unused
    do k = 1, pr_arrangement_max
      pr_prod_mult(i, j, k) = real_unused
      do l = 1, rc_product_max
        pr_ts_prod(i, j, k, l) = int_unused
      end do
    end do
  end do
end do

do i = 1, pr_max_equiv
  do j = 1, num_generics
    num_equiv_j = 0
    equivalents_{i, j} = 0
  end do
end do

  /* For each test species, first obtain the index of the corresponding generic archetype, generic,
     increment the number of equivalents to this archetype, and store the species index of this test in
     the equivalents array. */
do i = 1, pr_test_num
  generic = generics_{pr_test(i)}
  num_equiv_{generic}++
  equivalents_{num_equiv_{generic}, generic} = pr_test(i)
end do

  /* Do same for generics of background species. Since the same species may be a test, as well as a
     background, check to see if it's already on the list (call to int_lookup). */
do i = 1, pr_background_num
```

```
    generic = generics_{pr_background(i)}
    if ((num_equiv_{generic} ≡ 0) ∨ (int_lookup(pr_background(i), equivalents_{1, generic}, pr_max_equiv) ≡ 0))
          then
      num_equiv_{generic}++
      equivalents_{num_equiv_{generic}, generic} = pr_background(i)
    endif
  end do

    /* We can only deal with binary collisions currently */
  if (pr_reaction_num > 0) then
    do j = 1, pr_reaction_num
      assert(rc_reagent_num(pr_reaction(j)) ≡ 2)
    end do
  end if

  do i = 1, pr_test_num
    pr_rc_num(i) = 0
  end do
  pr_bkrc_dim = 0

  if (pr_reaction_num > 0) then
    do j = 1, pr_reaction_num

        /* The original version of this code permitted an arbitrary ordering of test and background
          in reagent specification (restricted to binary collisions). With the introduction of
          neutral-neutral elastic scattering reactions (reagents having the same species), a specific
          ordering had to be chosen. The convention used in many outside data sources (e.g.,
          Aladdin) is consistent with "background + test". So, this ordering is chosen here as well.
          Note: assuming that we do not need code here to deal with species-specific reactions
          (asserts to follow should catch exceptions). */
      ts_reagent = 0
      do i = 1, pr_bkrc_reagent_max
        generic_reagents_{i} = 0
      end do
      do i = 1, rc_reagent_num(pr_reaction(j))
        generic_reagents_{i} = generics_{rc_reagent(pr_reaction(j), i)}
        if (pr_test_lookup(equivalents_{1, generic_reagents_{i}}) ≠ 0) then
          ts_reagent = 2     // else: is a background reaction (below)
          bk_reagent = 1
        end if
      end do

        /* Done setting generic_reagents; use to assign remaining pr arrays */
      if (ts_reagent > 0) then
        if (rc_gen(pr_reaction(j)) ≡ rc_generic_no) then
          test = pr_test_lookup(rc_reagent(pr_reaction(j), ts_reagent))
          background = pr_background_lookup(rc_reagent(pr_reaction(j), bk_reagent))
          assert(pr_test_check(test) ∧ pr_background_check(background))
          pr_rc_num(test)++
          assert(pr_rc_num(test) ≤ pr_reaction_max)
          pr_ts_rc(test, pr_rc_num(test)) = j
          pr_ts_bk(test, pr_rc_num(test)) = background
          pr_num_arrangements(test, pr_rc_num(test)) = 1
          pr_prod_mult(test, pr_rc_num(test), 1) = one
          do k = 1, rc_product_num(pr_reaction(j))
```

```
        pr_ts_prod(test, pr_rc_num(test), 1, k) = rc_product(pr_reaction(j), k)
      end do
    else if (rc_gen(pr_reaction(j)) ≡ rc_generic_yes) then
      do k = 1, num_equiv_{generic_reagents_{ts_reagent}}
        test = pr_test_lookup(equivalents_{k, generic_reagents_{ts_reagent}})
        do l = 1, num_equiv_{generic_reagents_{bk_reagent}}
          background = pr_background_lookup(equivalents_{l, generic_reagents_{bk_reagent}})
          assert(pr_test_check(test) ∧ pr_background_check(background))
          pr_rc_num(test)++
          assert(pr_rc_num(test) ≤ pr_reaction_max)
          pr_ts_rc(test, pr_rc_num(test)) = j
          pr_ts_bk(test, pr_rc_num(test)) = background
          reagents_{bk_reagent} = pr_background(background)
          reagents_{ts_reagent} = pr_test(test)
          call set_products(pr_test_args(test), sp_args(reagents), rc_args(pr_reaction(j)))
        end do
      end do
    else
      assert('Illegal␣value␣of␣rc_gen(j)' ≡ '␣')
    end if
  else   /* Reactions among background species... We have not established a protocol yet for
         dealing with reactions involving two background species. The typical reactions involve a
         test species (given velocity and position) and a background species (described only by a
         temperature, density, and flow velocity). If we're going to treat reactions between two
         background species in the same way, we'll need to specify which behaves more like a test
         particle. Rather than rely on a convention for the ordering of the reagents on input, make
         this decision here based on mass. The obvious motivation is that one of the reagents is
         inevitably an electron in which case i) its temperature determines the reaction rate, and
         ii) we will not likely be interested in it as a test particle. Whereas, the other reagent
         (e.g., an ion about to be recombined) will be sampled in preparation for determining the
         velocity of the sourced test species. */
    if (rc_gen(pr_reaction(j)) ≡ rc_generic_no) then
      do k = 1, pr_bkrc_reagent_max
        back_k = pr_background_lookup(rc_reagent(pr_reaction(j), k))
        assert(pr_background_check(back_k))
      end do
      pr_bkrc_dim++
      var_realloca(problem_background_reaction)
      var_realloca(problem_bkrc_reagents)
      var_realloca(problem_bkrc_products)
      pr_bk_rc(pr_bkrc_dim) = j
      if (sp_m(pr_background(back_2)) < sp_m(pr_background(back_1))) then
        pr_bkrc_rg(pr_bkrc_dim, 1) = back_2
        pr_bkrc_rg(pr_bkrc_dim, 2) = back_1
      else
        pr_bkrc_rg(pr_bkrc_dim, 1) = back_1
        pr_bkrc_rg(pr_bkrc_dim, 2) = back_2
      end if
      assert(rc_product_num(pr_reaction(j)) ≡ 1)
          // Otherwise, we need to add machinery to deal with arrangements, etc.
      pr_bkrc_prod(pr_bkrc_dim, 1) = rc_product(pr_reaction(j), 1)
```

```
      else if (rc_gen(pr_reaction(j)) ≡ rc_generic_yes) then
        do k = 1, num_equiv(generic_reagents(1))
          back(1) = pr_background_lookup(equivalents(k, generic_reagents(1)))
          assert(pr_background_check(back(1)))
          do l = 1, num_equiv(generic_reagents(pr_bkrc_reagent_max))
            back(pr_bkrc_reagent_max) =
                pr_background_lookup(equivalents(l, generic_reagents(pr_bkrc_reagent_max)))
            assert(pr_background_check(back(pr_bkrc_reagent_max)))
            pr_bkrc_dim++
            var_realloca(problem_background_reaction)
            var_realloca(problem_bkrc_reagents)
            var_realloca(problem_bkrc_products)
            pr_bk_rc(pr_bkrc_dim) = j
            do m = 1, pr_bkrc_reagent_max
              if (sp_m(pr_background(back(pr_bkrc_reagent_max))) < sp_m(pr_background(back(1))))
                    then
                pr_bkrc_rg(pr_bkrc_dim, m) = back(pr_bkrc_reagent_max - m + 1)
              else
                pr_bkrc_rg(pr_bkrc_dim, m) = back(m)
              end if
              reagents(m) = pr_background(back(m))
            end do
            call set_bkrc_products(pr_bkrc_dim, sp_args(reagents))
          end do
        end do
      else
        assert('Illegal␣value␣of␣rc_gen(j)' ≡ '␣')
      end if
    end if
  end do
end if    /* Have to introduce pr_bkrc_dim because netCDF allocates space for these arrays even if
    pr_bkrc_num=0. The problem arises when the netCDF file containing these variables is read back
    in. The size is reported to be non-zero, but the dimensioning variables says it should be zero.
    The awkward solution is to use pr_bkrc_dim as the dimension of these arrays and pr_bkrc_num as
    the actual number of reactions. The former is set to 1 if there are no background reactions. */
pr_bkrc_num = pr_bkrc_dim
if (pr_bkrc_dim ≡ 0) then
  pr_bkrc_dim = 1
  var_alloc(problem_background_reaction)
  var_alloc(problem_bkrc_reagents)
  var_alloc(problem_bkrc_products)
  pr_bk_rc(1) = int_unused
  do i = 1, pr_bkrc_reagent_max
    pr_bkrc_rg(1, i) = int_unused
  end do
  do i = 1, rc_product_max
    pr_bkrc_prod(1, i) = int_unused
  end do
else
  var_reallocb(problem_background_reaction)
  var_reallocb(problem_bkrc_reagents)
```

```
      var_reallocb(problem_bkrc_products)
    end if

    do i = 1, pr_test_num
      pr_pm_case_num(i) = 0
      do j = 1, pr_pmi_max
        pr_pm_cases(i, j) = int_unused
        pr_pm_num_arrange(i, j) = int_unused
        do k = 1, pr_arrangement_max
          pr_pm_prod_mult(i, j, k) = real_unused
          do l = 1, rc_product_max
            pr_pm_prod(i, j, k, l) = int_unused
          end do
        end do
      end do
    end do    /* Here is the analogous code for PMI which sets up the list of PMI in which each test
        species participates. Species-specific PMI are dealt with first: */
    do j = 1, pr_pmi_num
      if (pm_gen(pr_pm_ref(j)) ≡ pmi_generic_no) then
        test = pr_test_lookup(pm_reagent(pr_pm_ref(j)))
        pr_pm_case_num(test)++
        pr_pm_cases(test, pr_pm_case_num(test)) = j

          /* For species-specific PMI, the products can be set directly: */
        pr_pm_num_arrange(test, pr_pm_case_num(test)) = 1
        pr_pm_prod_mult(test, pr_pm_case_num(test), 1) = one
        do k = 1, pm_product_num(pr_pm_ref(j))
          pr_pm_prod(test, pr_pm_case_num(test), 1, k) = pm_product(pr_pm_ref(j), k)
        end do

          /* For PMI which refer to generic reagents and product species, proceed as above for
            reactions. The process is simplified because (1) there are no generic materials, (2) the
            ordering of test and materials is hardwired. */
      else if (pm_gen(pr_pm_ref(j)) ≡ pmi_generic_yes) then
        generic_reagents_1 = generics_{pm_reagent(pr_pm_ref(j))}
        do k = 1, num_equiv_{generic_reagents_1}
          test = pr_test_lookup(equivalents_{k, generic_reagents_1})
          assert(pr_test_check(test))
          pr_pm_case_num(test)++
          pr_pm_cases(test, pr_pm_case_num(test)) = j

          call set_pmi_products(pr_test_args(test), pm_args(pr_pm_ref(j)))
        end do

      else
        assert('Illegal␣value␣of␣pm_gen(j)' ≡ '␣')
      end if
    end do

    return
  end
```

This is a clearing-house routine used to select the product-setting routine specific for each reaction.

```
⟨Functions and Subroutines 1.2⟩ +≡
  subroutine set_products(pr_test_dummy(ts), sp_dummy(reagents), rc_dummy(r))
    implicit_none_f77
    pr_common
    rc_common
    implicit_none_f90

    pr_test_decl(ts)    // Input
    rc_decl(r)
    sp_decl(reagents_*)

    integer i    // Local
    real sum

    if (rc_reaction_type(r) ≡ 'ionize') then
      call set_prod_ionize(pr_test_args(ts), sp_args(reagents), rc_args(r))
    else if (rc_reaction_type(r) ≡ 'ionize_suppress') then
      call set_prod_ionize(pr_test_args(ts), sp_args(reagents), rc_args(r))
    else if (rc_reaction_type(r) ≡ 'chargex') then
      call set_prod_chargex(pr_test_args(ts), sp_args(reagents), rc_args(r))
    else if (rc_reaction_type(r) ≡ 'dissoc') then
      call set_prod_dissoc(pr_test_args(ts), sp_args(reagents), rc_args(r))
    else if (rc_reaction_type(r) ≡ 'dissoc_rec') then
      call set_prod_dissoc_rec(pr_test_args(ts), sp_args(reagents), rc_args(r))
    else if (rc_reaction_type(r) ≡ 'dissoc_cramd') then
      call set_prod_dissoc(pr_test_args(ts), sp_args(reagents), rc_args(r))
    else if (rc_reaction_type(r) ≡ 'elastic') then
      call set_prod_elastic(pr_test_args(ts), sp_args(reagents), rc_args(r))
    else if (rc_reaction_type(r) ≡ 'bgk_elastic') then
      call set_prod_elastic(pr_test_args(ts), sp_args(reagents), rc_args(r))
    else if (rc_reaction_type(r) ≡ 'excitation') then
      call set_prod_excite(pr_test_args(ts), sp_args(reagents), rc_args(r))
    else if (rc_reaction_type(r) ≡ 'deexcitation') then
      call set_prod_excite(pr_test_args(ts), sp_args(reagents), rc_args(r))
    else
      pr_num_arrangements(ts, pr_rc_num(ts)) = 0
      return
    end if

      /* Rescale multiplicity so that it becomes a probability */
    sum = zero
    do i = 1, pr_num_arrangements(ts, pr_rc_num(ts))
      sum = sum + pr_prod_mult(ts, pr_rc_num(ts), i)
    end do
    do i = 1, pr_num_arrangements(ts, pr_rc_num(ts))
      pr_prod_mult(ts, pr_rc_num(ts), i) = pr_prod_mult(ts, pr_rc_num(ts), i) / sum
    end do

    return
  end
```

Set up isotopic variants of ionizations. By convention, an ionization reaction specification takes the form:

$$A + B \rightarrow A + B^{+} + e; \quad (1)$$

note that B could be charged to start with.

⟨ Functions and Subroutines 1.2 ⟩ +≡

```
subroutine set_prod_ionize(pr_test_dummy(ts), sp_dummy(reagents), rc_dummy(r))
  implicit_none_f77
  ps_common
  rc_common
  pr_common
  sp_common
  implicit_none_f90

  pr_test_decl(ts)
  sp_decl(reagents_*)
  rc_decl(r)

  external species_add_check    // External
  logical species_add_check

  integer i    // Local

  sp_decl(product_2)

  st_decls

  assert(rc_product_num(r) ≡ 3)
  pr_num_arrangements(ts, pr_rc_num(ts)) = 1
  pr_prod_mult(ts, pr_rc_num(ts), 1) = one

    /* First product corresponds to the reagent which does the ionizing: its species is unaltered. */
  assert(sp_generic(rc_product(r, 1)) ≡ sp_generic(reagents_1))
  pr_ts_prod(ts, pr_rc_num(ts), 1, 1) = reagents_1

    /* By convention, the third (last) product is the electron knocked off during ionization. */
  assert(rc_product(r, 3) ≡ sp_lookup('e'))
  pr_ts_prod(ts, pr_rc_num(ts), 1, 3) = rc_product(r, 3)

    /* The second product is the result of the ionization. It corresponds to the second reagent, minus
       an electron. By looping over the generic equivalents to the second rc_product, we ensure that any
       details (e.g., quantum state) not checked by species_add_check are correctly carried through. */
  product_2 = rc_product(r, 3)
  do i = 1, num_equiv_{generics_{rc_product(r, 2)}}
    product_1 = equivalents_{i, generics_{rc_product(r, 2)}}
    if (species_add_check(2, sp_args(product), 1, sp_args(reagents_2))) then
      pr_ts_prod(ts, pr_rc_num(ts), 1, 2) = product_1
      return
    end if
  end do
  assert('Unmatched␣ionization␣product' ≡ '␣')

  return
end
```

Set up isotopic variants of charge exchange. By convention, a charge exchange reaction specification takes the form:

$$A^{+} + B \rightarrow A + B^{+}. \quad (2)$$

the superscript "+"s are used only to indicate a change in charge state; the actual charge states of A and B are intended to be arbitrary. The procedure below can handle both single and double electron charge exchange.

```
⟨ Functions and Subroutines 1.2 ⟩ +≡
    subroutine set_prod_chargex(pr_test_dummy(ts), sp_dummy(reagents), rc_dummy(r))
      implicit_none_f77
      rc_common
      pr_common
      sp_common
      ps_common
      implicit_none_f90

      pr_test_decl(ts)
      rc_decl(r)
      sp_decl(reagents_*)

      external species_add_check    // External
      logical species_add_check

      integer i, num_electrons    // Local

      sp_decl(product_3)

      st_decls

      num_electrons = 0
      assert(rc_product_num(r) ≡ 2)
      pr_num_arrangements(ts, pr_rc_num(ts)) = 1
      pr_prod_mult(ts, pr_rc_num(ts), 1) = one
          /* The first product should match up with the first reagent plus one or two electrons. */
      product_1 = sp_lookup('e')
      product_2 = reagents_1
      do i = 1, num_equiv_{generics_{rc_product(r, 1)}}
        if (species_add_check(2, sp_args(product), 1, sp_args(equivalents_{i, generics_{rc_product(r, 1)}}))) then
          pr_ts_prod(ts, pr_rc_num(ts), 1, 1) = equivalents_{i, generics_{rc_product(r, 1)}}
          num_electrons = 1
          goto loop1
        else    // Might be double charge exchange
          product_3 = product_1
          if (species_add_check(3, sp_args(product), 1, sp_args(equivalents_{i, generics_{rc_product(r, 1)}}))) then
            pr_ts_prod(ts, pr_rc_num(ts), 1, 1) = equivalents_{i, generics_{rc_product(r, 1)}}
            num_electrons = 2
            goto loop1
          end if
        end if
      end do
      assert('Unmatched␣first␣charge␣exchange␣product' ≡ '␣')
loop1: continue
          /* Likewise, the second reagent should match the second product plus one or two electrons. */
      do i = 1, num_equiv_{generics_{rc_product(r, 2)}}
```

```
      product_2 = equivalents_{i, generics_{rc_product(r, 2)}}
      if ((num_electrons ≡ 1) ∧ (species_add_check(2, sp_args(product), 1, sp_args(reagents_2)))) then
        pr_ts_prod(ts, pr_rc_num(ts), 1, 2) = product_2
        return
      else if (num_electrons ≡ 2) then
        if (species_add_check(3, sp_args(product), 1, sp_args(reagents_2))) then
          pr_ts_prod(ts, pr_rc_num(ts), 1, 2) = product_2
          return
        end if
      end if
    end do
    assert('Unmatched␣second␣charge␣exchange␣product' ≡ '␣')

    return
  end
```

Set up isotopic variants of elastic collisions. By convention, an elastic reaction specification takes the form:t

$$A^+ + B \rightarrow A^+ + B. \tag{3}$$

the superscript "+"s are used only to indicate a charge state; the actual charge states of A and B are intended to be arbitrary. Currently the only reaction allowed is that given below.

$$H^+ + H2 \rightarrow H^+ + H2. \tag{4}$$

⟨ Functions and Subroutines 1.2 ⟩ +≡

```
subroutine set_prod_elastic(pr_test_dummy(ts), sp_dummy(reagents), rc_dummy(r))
  implicit_none_f77
  rc_common
  pr_common
  sp_common
  implicit_none_f90

  pr_test_decl(ts)
  rc_decl(r)
  sp_decl(reagents_*)

  external species_add_check     // External
  logical species_add_check

  st_decls

  assert(rc_product_num(r) ≡ 2)
  pr_num_arrangements(ts, pr_rc_num(ts)) = 1
  pr_prod_mult(ts, pr_rc_num(ts), 1) = one

  assert(sp_generic(rc_product(r, 1)) ≡ sp_generic(reagents_1))
@#if 0
    /* Having tested reagents and stated assumptions in the introduction, why did we write this at
       all? */
  pr_ts_prod(ts, pr_rc_num(ts), 1, 1) = rc_product(r, 1)
@#endif
  pr_ts_prod(ts, pr_rc_num(ts), 1, 1) = reagents_1
  assert(sp_generic(rc_product(r, 2)) ≡ sp_generic(reagents_2))
@#if 0
  pr_ts_prod(ts, pr_rc_num(ts), 1, 2) = rc_product(r, 2)
@#endif
  pr_ts_prod(ts, pr_rc_num(ts), 1, 2) = reagents_2

  return
end
```

Set up isotopic variants of dissociation. This reactions is assumed to have the form:

$$\mathrm{A} + \mathrm{B} \rightarrow \mathrm{A} + \mathrm{B}_1 + \mathrm{B}_2 + \ldots \quad (5)$$

where the products B_j are the result of the dissociation of B.

⟨Functions and Subroutines 1.2⟩ +≡

```
subroutine set_prod_dissoc(pr_test_dummy(ts), sp_dummy(reagents), rc_dummy(r))
  implicit_none_f77
  rc_common
  pr_common
  sp_common
  implicit_none_f90

  pr_test_decl(ts)
  rc_decl(r)
  sp_decl(reagents_*)

  integer i    // Local

  assert(sp_generic(rc_product(r, 1)) ≡ sp_generic(reagents_1))

  if (rc_product_num(r) ≡ 3) then
    call product_perm_2(pr_test_args(ts), 1, sp_args(reagents_2), rc_args(r))
  else if (rc_product_num(r) ≡ 4) then
    call product_perm_3(pr_test_args(ts), 1, sp_args(reagents_2), rc_args(r))
  else
    assert('Illegal␣value␣of␣rc_product_num' ≡ '␣')
  end if

  do i = 1, pr_num_arrangements(ts, pr_rc_num(ts))
    pr_ts_prod(ts, pr_rc_num(ts), i, 1) = reagents_1
  end do

  return
end
```

Set up isotopic variants of dissociative recombination. This reactions is assumed to have the form:

$$A + B \rightarrow C_1 + C_2 + \ldots \tag{6}$$

where the products C_j are the result of the dissociation of C which is formed by the recombination of A and B.

⟨ Functions and Subroutines 1.2 ⟩ +≡

```
subroutine set_prod_dissoc_rec(pr_test_dummy(ts), sp_dummy(reagents), rc_dummy(r))
   implicit_none_f77
   rc_common
   pr_common
   sp_common
   implicit_none_f90

   pr_test_decl(ts)
   rc_decl(r)
   sp_decl(reagents_*)

   if (rc_product_num(r) ≡ 2) then
      call product_perm_2(pr_test_args(ts), 2, sp_args(reagents), rc_args(r))
   else if (rc_product_num(r) ≡ 3) then
      call product_perm_3(pr_test_args(ts), 2, sp_args(reagents), rc_args(r))
   else
      assert('Illegal␣value␣of␣rc_product_num' ≡ '␣')
   end if

   return
end
```

Set up isotopic variants of excitation. This routine is also suitable for deexcitation. The general form of these reactions is:

$$\mathrm{A} + \mathrm{B} \rightarrow \mathrm{A} + \mathrm{B}^*, \quad (7)$$

where A is the background species and B is the test species. The $*$ indicates a change in the electron configuration of the species. In terms of DEGAS 2's species characteristics, this is just a change in the species label; the elements comprising the reagent and product test species must be the same.

⟨ Functions and Subroutines 1.2 ⟩ +≡

```
subroutine set_prod_excite(pr_test_dummy(ts), sp_dummy(reagents), rc_dummy(r))
  implicit_none_f77
  ps_common
  rc_common
  pr_common
  sp_common
  implicit_none_f90

  pr_test_decl(ts)
  sp_decl(reagents_*)
  rc_decl(r)

  external species_add_check     // External
  logical species_add_check

  integer i     // Local

  sp_decl(product)

  st_decls

  assert(rc_product_num(r) ≡ 2)
  pr_num_arrangements(ts, pr_rc_num(ts)) = 1
  pr_prod_mult(ts, pr_rc_num(ts), 1) = one
      /* First product corresponds to the reagent which does the exciting: its species is unaltered. */
  assert(sp_generic(rc_product(r, 1)) ≡ sp_generic(reagents_1))
  pr_ts_prod(ts, pr_rc_num(ts), 1, 1) = reagents_1     /* The second product is the result of the
      excitation or deexcitation. It corresponds only to a different electron configuration from the
      reagent. So, species_add_check will suffice for picking out the appropriate product. */
  do i = 1, num_equiv_{generics_{rc_product(r, 2)}}
    product = equivalents_{i, generics_{rc_product(r, 2)}}
    if (species_add_check(1, sp_args(product), 1, sp_args(reagents_2))) then
      pr_ts_prod(ts, pr_rc_num(ts), 1, 2) = product
      return
    end if
  end do
  assert('Unmatched␣excitation␣product' ≡ '␣')

  return
end
```

This is a clearing-house routine used to select the product-setting routine specific for each reaction among background species. Presently, the only application for this is recombination.

⟨ Functions and Subroutines 1.2 ⟩ +≡

```
subroutine set_bkrc_products(bkrc_ind, sp_dummy(reagents))
  implicit_none_f77
  pr_common
  rc_common
  implicit_none_f90

  integer bkrc_ind    // Input

  sp_decl(reagents_*)
  integer i    // Local

  do i = 1, rc_product_max
    pr_bkrc_prod(bkrc_ind, i) = int_unused
  end do
  if (rc_reaction_type(pr_reaction(pr_bk_rc(bkrc_ind))) ≡ 'recombination') then
    call set_prod_recombine(bkrc_ind, sp_args(reagents))
  else
    assert('Unsupported␣reaction␣type' ≡ '␣')
    return
  end if

  return
end
```

Set up isotopic variants of recombinations. By convention, a recombination reaction specification takes the form:

$$A^- + B^+ \rightarrow AB; \qquad (8)$$

note that B could be charged to start with.

⟨Functions and Subroutines 1.2⟩ +≡

```
subroutine set_prod_recombine(bkrc_ind, sp_dummy(reagents))
  implicit_none_f77
  rc_common
  pr_common
  sp_common
  ps_common
  implicit_none_f90

  integer bkrc_ind

  sp_decl(reagents_*)

  external species_add_check    // External
  logical species_add_check

  integer i    // Local

  rc_decl(r)

  sp_decl(product_2)

  st_decls

  r = pr_reaction(pr_bk_rc(bkrc_ind))
  assert(rc_product_num(r) ≡ 1)

    /* Assuming that only one product is listed for recombination. If there are more products, not
       only does this routine have to be augmented, but will probably need to add common arrays
       to deal with product permutations. By looping over the generic equivalents to rc_product, we
       ensure that any details (e.g., quantum state) not checked by species_add_check are correctly
       carried through. */
  do i = 1, num_equiv_{generics_{rc_product(r, 1)}}
    product_1 = equivalents_{i, generics_{rc_product(r, 1)}}
    if (species_add_check(1, sp_args(product), 2, sp_args(reagents))) then
      pr_bkrc_prod(bkrc_ind, 1) = product_1
      return
    end if
  end do
  assert('Unmatched␣recombination␣product' ≡ '␣')

  return
end
```

This is a clearing-house routine used to select the product-setting routine specific for each PMI.

⟨Functions and Subroutines 1.2⟩ +≡

```
subroutine set_pmi_products(pr_test_dummy(ts), pm_dummy(p))
  implicit_none_f77
  pr_common
  pm_common
  implicit_none_f90

  pr_test_decl(ts)     // Input
  pm_decl(p)

  integer i     // Local
  real sum

  if (pm_pmi_type(p) ≡ 'reflection') then
    call set_prod_reflection(pr_test_args(ts), pm_args(p))
  else if (pm_pmi_type(p) ≡ 'adsorption') then
    call set_prod_adsorption(pr_test_args(ts), pm_args(p))
  else if (pm_pmi_type(p) ≡ 'desorption') then
    call set_prod_desorption(pr_test_args(ts), pm_args(p))
  else if (pm_pmi_type(p) ≡ 'sputter') then
    assert('set_product␣for␣sputter␣not␣available' ≡ '␣')
  end if

    /* Rescale multiplicity so that it becomes a probability */
  sum = zero
  do i = 1, pr_pm_num_arrange(ts, pr_pm_case_num(ts))
    sum = sum + pr_pm_prod_mult(ts, pr_pm_case_num(ts), i)
  end do
  if (sum > zero) then     // In some cases, pr_pm_prod_mult is ≡ 0.
    do i = 1, pr_pm_num_arrange(ts, pr_pm_case_num(ts))
      pr_pm_prod_mult(ts, pr_pm_case_num(ts), i) = pr_pm_prod_mult(ts, pr_pm_case_num(ts), i)/sum
    end do
  end if

  return
end
```

Set up isotopic variations of reflections. This is a simple case of "what comes in must go out."

```
⟨Functions and Subroutines 1.2⟩ +≡
  subroutine set_prod_reflection(pr_test_dummy(ts), pm_dummy(p))
    implicit_none_f77
    pr_common    // Common
    pm_common
    sp_common
    implicit_none_f90

    pr_test_decl(ts)    // Input
    pm_decl(p)

    assert(pm_product_num(p) ≡ 1)
    pr_pm_num_arrange(ts, pr_pm_case_num(ts)) = 1
    pr_pm_prod_mult(ts, pr_pm_case_num(ts), 1) = one

      /* The only product had better be the same as the incoming test particle */
    assert(sp_generic(pm_product(p, 1)) ≡ sp_generic(pr_test(ts)))
    pr_pm_prod(ts, pr_pm_case_num(ts), 1, 1) = pr_test(ts)

    return
  end
```

Set up isotopic variations for adsorptions. It doesn't get any easier than this.

```
⟨Functions and Subroutines 1.2⟩ +≡
  subroutine set_prod_adsorption(pr_test_dummy(ts), pm_dummy(p))
    implicit_none_f77
    pr_common    // Common
    pm_common
    implicit_none_f90

    pr_test_decl(ts)    // Input
    pm_decl(p)

    assert(pm_product_num(p) ≡ 0)
    pr_pm_num_arrange(ts, pr_pm_case_num(ts)) = 1
    pr_pm_prod_mult(ts, pr_pm_case_num(ts), 1) = one

    pr_pm_prod(ts, pr_pm_case_num(ts), 1, 1) = 0

    return
  end
```

Set up isotopic variations for desorptions. The assumptions made here reflect the algorithm used in the original DEGAS code. Namely, for most incident neutral species, the desorbed species is the same as the incident one. For an incident hydrogen atom, however, the possibility existed for desorption as a molecule. It is intended that this option be represented by two products in the reference description of the PMI, only one of which is used at a time here. In the event that a molecule is returned, the isotope of one of the atoms is required to match that of the incident H. The other atom will be assigned in a general way here based on the other isotopes present in the problem. It is assumed that the weights of these various options are established in the particular data file for this PMI and will be implemented only at run time. An analogous description holds for carbon as well as the various methane fragments.

```
⟨Functions and Subroutines 1.2⟩ +≡
  subroutine set_prod_desorption(pr_test_dummy(ts), pm_dummy(p))
    implicit_none_f77
    pr_common    // Common
    pm_common
    el_common
    ps_common
    implicit_none_f90

    pr_test_decl(ts)    // Input
    pm_decl(p)

    external species_el_count    // External
    integer species_el_count

    integer i, ic, j, count_ts, count_prod    // Local
    logical match

    pr_generic_decl(generic_prod)

    pr_pm_num_arrange(ts, pr_pm_case_num(ts)) = 0
    do i = 1, pm_product_num(p)
      generic_prod = generics_{pm_product(p, i)}
      assert(generic_prod > 0 ∧ generic_prod ≤ num_generics)
      do j = 1, num_equiv_{generic_prod}
        match = T
        do ic = 1, el_num
          count_ts = species_el_count(pr_test(ts), el_args(ic))
          count_prod = species_el_count(equivalents_{j, generic_prod}, el_args(ic))
          if (count_ts > count_prod)
            match = F
        end do
        if (match) then
          pr_pm_num_arrange(ts, pr_pm_case_num(ts))++
              /* Set this to zero; it will be determined at run time. */
          pr_pm_prod_mult(ts, pr_pm_case_num(ts), pr_pm_num_arrange(ts,
              pr_pm_case_num(ts))) = zero
          pr_pm_prod(ts, pr_pm_case_num(ts), pr_pm_num_arrange(ts, pr_pm_case_num(ts)),
              1) = equivalents_{j, generic_prod}
        end if
      end do
    end do
    assert(pr_pm_num_arrange(ts, pr_pm_case_num(ts)) > 0)

    return
  end
```

Assemble and check permutations of two products arising from the dissociation of *num_sp* species *sp* in reaction *r*. The test species *ts* is carried into the routine since to be used as an array index. Note that for simplicity duplicates (i.e., permuted lists of equivalent species) are not eliminated. This makes for more arrangements than necessary, but the overall probability for a given unique arrangement is effected nonetheless. If desired, this duplication could be eliminated through additional logic.

```
⟨ Functions and Subroutines 1.2 ⟩ +≡
  subroutine product_perm_2(pr_test_dummy(ts), num_sp, sp_dummy(sp), rc_dummy(r))
    implicit_none_f77
    rc_common
    sp_common
    pr_common
    ps_common
    implicit_none_f90

    integer num_sp    // Input

    pr_test_decl(ts)
    sp_decl(sp_num_sp)
    rc_decl(r)

    external species_add_check    // External
    logical species_add_check

    integer i, j, first_product    // Local

    sp_decl(product_2)
    pr_generic_decl(generic_1)
    pr_generic_decl(generic_2)

      /* A num_sp of 1 indicates pure dissociation ⇒ the actual first product will be the dissociating
        agent; the first dissociation product is then the second product. On the other hand, num_sp =
        2 suggests dissociative recombination where all products come from dissociation. */
    if (num_sp ≡ 1) then
      first_product = 2
    else if (num_sp ≡ 2) then
      first_product = 1
    else
      assert('Invalid␣num_sp' ≡ '␣')
    end if

      /* Obtain generic indices of the two products */
    generic_1 = generics_rc_product(r, first_product)
    generic_2 = generics_rc_product(r, first_product+1)

      /* Loop over the species equivalent to each generic and see which combinations are consistent
        with the given reagents. */
    pr_num_arrangements(ts, pr_rc_num(ts)) = 0
    do i = 1, num_equiv_generic_1
      product_1 = equivalents_i, generic_1
      do j = 1, num_equiv_generic_2
        product_2 = equivalents_j, generic_2
        if (species_add_check(2, sp_args(product), num_sp, sp_args(sp))) then
          pr_num_arrangements(ts, pr_rc_num(ts))++    // Valid arrangement
          assert(pr_num_arrangements(ts, pr_rc_num(ts)) ≤ pr_arrangement_max)
          pr_ts_prod(ts, pr_rc_num(ts), pr_num_arrangements(ts, pr_rc_num(ts)),
              first_product) = product_1
```

$pr_ts_prod(ts,\ pr_rc_num(ts),\ pr_num_arrangements(ts,\ pr_rc_num(ts)),$
$first_product + 1) = product_2$
$pr_prod_mult(ts,\ pr_rc_num(ts),\ pr_num_arrangements(ts,$
$pr_rc_num(ts))) = sp_multiplicity(product_1) * sp_multiplicity(product_2)$
end if
end do
end do
$assert(pr_num_arrangements(ts,\ pr_rc_num(ts)) > 0)$
return
end

Assemble and check permutations of three products arising from the dissociation of species *sp* in reaction *r*. The test species *ts* is carried into the routine since to be used as an array index. Note that for simplicity duplicates (i.e., permuted lists of equivalent species) are not eliminated. This makes for more arrangements than necessary, but the overall probability for a given unique arrangement is effected nonetheless. If desired, this duplication could be eliminated through additional logic.

⟨Functions and Subroutines 1.2⟩ +≡

```
subroutine product_perm_3(pr_test_dummy(ts), num_sp, sp_dummy(sp), rc_dummy(r))
  implicit_none_f77
  rc_common
  sp_common
  pr_common
  ps_common
  implicit_none_f90

  integer num_sp    // Input

  pr_test_decl(ts)
  sp_decl(sp_{num_sp})
  rc_decl(r)

  external species_add_check    // External
  logical species_add_check

  integer i, j, k, first_product    // Local

  sp_decl(product_3)
  pr_generic_decl(generic_1)
  pr_generic_decl(generic_2)
  pr_generic_decl(generic_3)

    /* A num_sp of 1 indicates pure dissociation ⇒ the actual first product will be the dissociating
       agent; the first dissociation product is then the second product. On the other hand, num_sp =
       2 suggests dissociative recombination where all products come from dissociation. */
  if (num_sp ≡ 1) then
    first_product = 2
  else if (num_sp ≡ 2) then
    first_product = 1
  else
    assert('Invalid␣num_sp' ≡ '␣')
  end if

    /* Obtain generic indices for each product */
  generic_1 = generics_{rc_product(r, first_product)}
  generic_2 = generics_{rc_product(r, first_product+1)}
  generic_3 = generics_{rc_product(r, first_product+2)}

    /* Loop over the species equivalent to each generic and see which combinations are consistent
       with the given reagents. */
  pr_num_arrangements(ts, pr_rc_num(ts)) = 0
  do i = 1, num_equiv_{generic_1}
    product_1 = equivalents_{i, generic_1}
    do j = 1, num_equiv_{generic_2}
      product_2 = equivalents_{j, generic_2}
      do k = 1, num_equiv_{generic_3}
        product_3 = equivalents_{k, generic_3}
        if (species_add_check(3, sp_args(product), num_sp, sp_args(sp))) then
```

```
          pr_num_arrangements(ts, pr_rc_num(ts))++     // Valid arrangement
          assert(pr_num_arrangements(ts, pr_rc_num(ts)) ≤ pr_arrangement_max)
          pr_ts_prod(ts, pr_rc_num(ts), pr_num_arrangements(ts, pr_rc_num(ts)),
                first_product) = product_1
          pr_ts_prod(ts, pr_rc_num(ts), pr_num_arrangements(ts, pr_rc_num(ts)),
                first_product + 1) = product_2
          pr_ts_prod(ts, pr_rc_num(ts), pr_num_arrangements(ts, pr_rc_num(ts)),
                first_product + 2) = product_3
          pr_prod_mult(ts, pr_rc_num(ts), pr_num_arrangements(ts, pr_rc_num(ts))) =
                sp_multiplicity(product_1) * sp_multiplicity(product_2) * sp_multiplicity(product_3)
        end if
      end do
    end do
  end do
  assert(pr_num_arrangements(ts, pr_rc_num(ts)) > 0)
  return
end
```

2 References

3 INDEX

assert: 1.2, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.11, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.21, 1.22, 1.23, 1.24, 1.25, 1.26.

b: 1.4, 1.5, 1.6, 1.7, 1.8.
back: 1.11.
background: 1.11.
bk_reagent: 1.11.
bkrc_ind: 1.19, 1.20.

char_undef: 1.2.
char_uninit: 1.9.
check_problem: 1.2, 1.11.
count_prod: 1.24.
count_ts: 1.24.

datasetup: 1.
diskin: 1.2.

e: 1.4, 1.5, 1.6, 1.7, 1.8.
el_args: 1.24.
el_common: 1.24.
el_num: 1.24.
eof: 1.2, 1.4, 1.5, 1.6, 1.7, 1.8.
equivalents: 1.11, 1.13, 1.14, 1.18, 1.20, 1.24, 1.25, 1.26.

file: 1.2.
FILE: 1.
FILELEN: 1.2.
filenames_array: 1.2.
finish_var0_list: 1.1, 1.10.
first_product: 1.25, 1.26.
float_abort: 1.1.
form: 1.2.
FORTRAN77: 1.1.

generic: 1.11.
generic_prod: 1.24.
generic_reagents: 1.11.
generic_1: 1.25, 1.26.
generic_2: 1.25, 1.26.
generic_3: 1.26.
generics: 1.3, 1.4, 1.5, 1.11, 1.13, 1.14, 1.18, 1.20, 1.24, 1.25, 1.26.

i: 1.3, 1.4, 1.5, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 1.14, 1.16, 1.18, 1.19, 1.20, 1.21, 1.24, 1.25, 1.26.
i_e_rate: 1.10.
ic: 1.24.
implicit_none_f77: 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.21, 1.22, 1.23, 1.24, 1.25, 1.26.
implicit_none_f90: 1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.21, 1.22, 1.23, 1.24, 1.25, 1.26.
init_pmi: 1.1.
init_problem: 1.2, 1.3.
init_reaction: 1.1.
init_var0_list: 1.1, 1.9.
int_lookup: 1.11.
int_unused: 1.11, 1.19.

j: 1.11, 1.24, 1.25, 1.26.

k: 1.11, 1.26.

l: 1.11.
len: 1.2, 1.4, 1.5, 1.6, 1.7, 1.8.
length: 1.2, 1.4, 1.5, 1.6, 1.7, 1.8.
line: 1.2, 1.4, 1.5, 1.6, 1.7, 1.8, 1.10.
LINELEN: 1.2.
lines: 1.10.
loop1: 1.2, 1.4, 1.5, 1.6, 1.7, 1.8, 1.14.
loop2: 1.4, 1.5, 1.6, 1.7, 1.8.

m: 1.11.
ma_common: 1.7, 1.11.
ma_lookup: 1.7.
ma_name: 1.11.
ma_num: 1.7.
match: 1.24.
max: 1.11.
max_lines: 1.10.
mem_alloc: 1.9.
mem_size: 1.9.

nc_read_elements: 1.3.
nc_read_materials: 1.3.
nc_read_pmi: 1.3.
nc_read_reactions: 1.3.
nc_read_species: 1.3.
nc_write_problem: 1.1.
next_token: 1.4, 1.5, 1.6, 1.7, 1.8.
num_electrons: 1.14.
num_equiv: 1.11, 1.13, 1.14, 1.18, 1.20, 1.24, 1.25, 1.26.
num_generics: 1.3, 1.4, 1.5, 1.11, 1.24.
num_lines: 1.10.
num_sp: 1.25, 1.26.

old_num: 1.10.
one: 1.11, 1.13, 1.14, 1.15, 1.18, 1.22, 1.23.

p: 1.4, 1.5, 1.6, 1.7, 1.8.
parse_string: 1.4, 1.5, 1.6, 1.7, 1.8.

pm_args: 1.11, 1.21.
pm_common: 1.8, 1.11, 1.21, 1.22, 1.23, 1.24.
pm_decl: 1.21, 1.22, 1.23, 1.24.
pm_dummy: 1.21, 1.22, 1.23, 1.24.
pm_gen: 1.11.
pm_lookup: 1.8.
pm_name: 1.11.
pm_num: 1.8.
pm_pmi_type: 1.21.
pm_product: 1.11, 1.22, 1.24.
pm_product_num: 1.11, 1.22, 1.23, 1.24.
pm_reagent: 1.11.
pmi_generic_no: 1.11.
pmi_generic_yes: 1.11.
pr: 1.11.
pr_arrangement_max: 1.11, 1.25, 1.26.
pr_background: 1.5, 1.9, 1.11.
pr_background_check: 1.11.
pr_background_decl: 1.11.
pr_background_lookup: 1.11.
pr_background_num: 1.3, 1.5, 1.9, 1.11.
pr_bk_rc: 1.11, 1.19, 1.20.
pr_bkrc_dim: 1.11.
pr_bkrc_num: 1.11.
pr_bkrc_prod: 1.11, 1.19, 1.20.
pr_bkrc_reagent_max: 1.11.
pr_bkrc_rg: 1.11.
pr_common: 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.21, 1.22, 1.23, 1.24, 1.25, 1.26.
pr_generic_decl: 1.11, 1.24, 1.25, 1.26.
pr_mat_ref: 1.7, 1.11.
pr_materials_num: 1.3, 1.7, 1.11.
pr_max_equiv: 1.11.
pr_num_arrangements: 1.11, 1.12, 1.13, 1.14, 1.15, 1.16, 1.18, 1.25, 1.26.
pr_num_change_vars: 1.9.
pr_pm_case_num: 1.11, 1.21, 1.22, 1.23, 1.24.
pr_pm_cases: 1.11.
pr_pm_num_arrange: 1.11, 1.21, 1.22, 1.23, 1.24.
pr_pm_prod: 1.11, 1.22, 1.23, 1.24.
pr_pm_prod_mult: 1.11, 1.21, 1.22, 1.23, 1.24.
pr_pm_ref: 1.8, 1.11.
pr_pmi_max: 1.11.
pr_pmi_num: 1.3, 1.8, 1.11.
pr_problem_sp_back: 1.9.
pr_problem_sp_test: 1.9.
pr_prod_mult: 1.11, 1.12, 1.13, 1.14, 1.15, 1.18, 1.25, 1.26.
pr_rc_num: 1.11, 1.12, 1.13, 1.14, 1.15, 1.16, 1.18, 1.25, 1.26.
pr_reaction: 1.6, 1.11, 1.19, 1.20.
pr_reaction_dim: 1.6.
pr_reaction_max: 1.11.
pr_reaction_num: 1.3, 1.6, 1.11.
pr_tag_string_length: 1.10.
pr_test: 1.4, 1.9, 1.11, 1.22, 1.24.
pr_test_args: 1.11, 1.12, 1.16, 1.17, 1.21.
pr_test_check: 1.11.
pr_test_decl: 1.11, 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.21, 1.22, 1.23, 1.24, 1.25, 1.26.
pr_test_dummy: 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.21, 1.22, 1.23, 1.24, 1.25, 1.26.
pr_test_lookup: 1.11.
pr_test_num: 1.3, 1.4, 1.9, 1.11.
pr_ts_bk: 1.11.
pr_ts_prod: 1.11, 1.13, 1.14, 1.15, 1.16, 1.18, 1.25, 1.26.
pr_ts_rc: 1.11.
pr_var_angle: 1.9.
pr_var_emitter_v_vector: 1.9.
pr_var_emitter_v_2: 1.9.
pr_var_emitter_v_3: 1.9.
pr_var_emitter_vf_Maxwell_vector: 1.9.
pr_var_emitter_vf_Maxwell_2: 1.9.
pr_var_emitter_vf_Maxwell_3: 1.9.
pr_var_emitter_vth_Maxwell: 1.9.
pr_var_energy: 1.9.
pr_var_energy_change: 1.9.
pr_var_energy_in: 1.9.
pr_var_energy_out: 1.9.
pr_var_mass: 1.9.
pr_var_mass_change: 1.9.
pr_var_mass_in: 1.9.
pr_var_mass_out: 1.9.
pr_var_momentum_change_vector: 1.9.
pr_var_momentum_change_2: 1.9.
pr_var_momentum_change_3: 1.9.
pr_var_momentum_in_vector: 1.9.
pr_var_momentum_in_2: 1.9.
pr_var_momentum_in_3: 1.9.
pr_var_momentum_out_vector: 1.9.
pr_var_momentum_out_2: 1.9.
pr_var_momentum_out_3: 1.9.
pr_var_momentum_vector: 1.9.
pr_var_momentum_2: 1.9.
pr_var_momentum_3: 1.9.
pr_var_problem_sp_index: 1.9.
pr_var_unknown: 1.9.
pr_var_xz_stress: 1.9.
pr_var0_list: 1.9, 1.10.
pr_var0_num: 1.9, 1.10.
problem_background_reaction: 1.11.
problem_background_sp: 1.5, 1.11.

problem_bkrc_products: 1.11.
problem_bkrc_reagents: 1.11.
problem_infile: 1.2.
problem_materials_ref: 1.7, 1.11.
problem_materials_sub: 1.7.
problem_num_arrangements: 1.11.
problem_pmi_case_num: 1.11.
problem_pmi_cases: 1.11.
problem_pmi_num_arrange: 1.11.
problem_pmi_prod_mult: 1.11.
problem_pmi_products: 1.11.
problem_pmi_ref: 1.8, 1.11.
problem_pmi_sub: 1.8.
problem_prod_mult: 1.11.
problem_rc: 1.6, 1.11.
problem_reaction_num: 1.11.
problem_species_background: 1.5.
problem_species_test: 1.4.
problem_test_background: 1.11.
problem_test_products: 1.11.
problem_test_reaction: 1.11.
problem_test_sp: 1.4, 1.11.
problem_version: 1.2.
problemfile: 1.
problemsetup: 1, 1.1.
product: 1.13, 1.14, 1.18, 1.20, 1.25, 1.26.
product_perm_2: 1.16, 1.17, 1.25.
product_perm_3: 1.16, 1.17, 1.26.
ps_common: 1.3, 1.4, 1.5, 1.11, 1.13, 1.14, 1.18, 1.20, 1.24, 1.25, 1.26.

rc_args: 1.11, 1.12, 1.16, 1.17.
rc_common: 1.6, 1.11, 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.25, 1.26.
rc_decl: 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.20, 1.25, 1.26.
rc_dummy: 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.25, 1.26.
rc_gen: 1.11.
rc_generic_no: 1.11.
rc_generic_yes: 1.11.
rc_lookup: 1.6.
rc_name: 1.11.
rc_product: 1.11, 1.13, 1.14, 1.15, 1.16, 1.18, 1.20, 1.25, 1.26.
rc_product_max: 1.11, 1.19.
rc_product_num: 1.11, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.20.
rc_reaction_type: 1.12, 1.19.
rc_reagent: 1.11.
rc_reagent_max: 1.11.
rc_reagent_num: 1.11.
read_background: 1.2, 1.5.
read_materials: 1.2, 1.7.
read_pmi: 1.2, 1.8.
read_problem: 1.1, 1.2.
read_reaction: 1.2, 1.6.
read_string: 1.2, 1.4, 1.5, 1.6, 1.7, 1.8.
read_test: 1.2, 1.4.
readfilenames: 1.1.
reagents: 1.11, 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20.
real_unused: 1.11.
rf_common: 1.2.

set_bkrc_products: 1.11, 1.19.
set_pmi_products: 1.11, 1.21.
set_prod_adsorption: 1.21, 1.23.
set_prod_chargex: 1.12, 1.14.
set_prod_desorption: 1.21, 1.24.
set_prod_dissoc: 1.12, 1.16.
set_prod_dissoc_rec: 1.12, 1.17.
set_prod_elastic: 1.12, 1.15.
set_prod_excite: 1.12, 1.18.
set_prod_ionize: 1.12, 1.13.
set_prod_recombine: 1.19, 1.20.
set_prod_reflection: 1.21, 1.22.
set_products: 1.11, 1.12.
sp: 1.25, 1.26.
SP: 1.10.
sp_args: 1.11, 1.12, 1.13, 1.14, 1.16, 1.17, 1.18, 1.19, 1.20, 1.25, 1.26.
sp_common: 1.3, 1.4, 1.5, 1.9, 1.11, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.20, 1.22, 1.25, 1.26.
sp_decl: 1.11, 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.25, 1.26.
sp_dummy: 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.25, 1.26.
sp_generic: 1.4, 1.5, 1.13, 1.15, 1.16, 1.18, 1.22.
sp_lookup: 1.4, 1.5, 1.13, 1.14.
sp_m: 1.11.
sp_multiplicity: 1.25, 1.26.
sp_name: 1.11.
sp_num: 1.3, 1.4, 1.5.
sp_sy: 1.9.
species_add_check: 1.13, 1.14, 1.15, 1.18, 1.20, 1.25, 1.26.
species_el_count: 1.24.
st_decls: 1.2, 1.4, 1.5, 1.6, 1.7, 1.8, 1.10, 1.11, 1.13, 1.14, 1.15, 1.18, 1.20.
status: 1.2.
stdout: 1.11.
string_lookup: 1.10.
sum: 1.12, 1.21.
SUN: 1.1.

tempfile: 1.2.
test: 1.11.
trim: 1.11.
ts: 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.21, 1.22, 1.23, 1.24, 1.25, 1.26.
ts_reagent: 1.11.

`unit`: 1.2, 1.4, 1.5, 1.6, 1.7, 1.8.

var_alloc: 1.3, 1.4, 1.5, 1.7, 1.8, 1.9, 1.11.
var_realloc: 1.9, 1.10.
var_realloca: 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11.
var_reallocb: 1.10, 1.11.

xs_copy: 1.9, 1.10.

zero: 1.12, 1.21, 1.24.

⟨ Functions and Subroutines 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13, 1.14, 1.15, 1.16, 1.17, 1.18, 1.19, 1.20, 1.21, 1.22, 1.23, 1.24, 1.25, 1.26 ⟩ Used in section 1.1.
⟨ Memory allocation interface 0 ⟩ Used in sections 1.11, 1.10, 1.9, 1.8, 1.7, 1.6, 1.5, 1.4, 1.3, and 1.2.

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

WEB FILE: "/u/dstotler/degas2/src/problemsetup.web".

CHANGE FILE: (none).

GLOBAL LANGUAGE: FORTRAN.