

geometry

November 24, 2009
13:38

Contents

1	Geometry routines for degas	1
2	External interface to geometry routines	3
3	Simple utility routines	4
4	Basic geometrical routines	7
5	Face intersections	13
6	Cell intersection	19
7	Locate point in geometry	26
8	A simple consistency checker for locatations	29
9	Read in the geometry specification from a netcdf file	30
10	The basic transformation routines	31
11	INDEX	32

1 Geometry routines for degas

\$Id: geometry.web,v 1.35 2006/01/19 19:22:21 dstotler Exp \$

Each cell is described by an equation of the form

$$C_j = \bigcap_{k \in K_j \cup K'_j} \{\mathbf{x} : \sigma_k f_k(\mathbf{x}) \geq 0\}.$$

where each f_k is a quadratic surface and σ_k is an orientation (± 1). K_j is the set of *faces* of cell C_j , and K'_j is the set of *cut surfaces*. Cut surfaces are used to eliminate extraneous volumes that may appear when using quadratic surfaces. We require that each cell is *connected*. The cut surfaces serve to enforce this constraint.

There are $N_c + 1$ cells numbered 0 thru N_c . Cell 0 is the universe. All the other cells are disjoint and their union is the universe. Thus

$$C_0 = \bigcup_{j=1}^{N_c} C_j,$$

and

$$V(C_i \cap C_j) = 0,$$

for all $i \neq j$, $i > 0$, $j > 0$, where $V(C)$ is the volume of C .

Cells are grouped together into *zones*. All cells other than cell 0 belong to exactly one zone. By combining cells together arbitrarily complicated zones can be defined. Zones are the basic geometrical abstraction seen by the outside world.

The information about the cells is held in an array $cells_{4, ncells}$. The “4” represents the following items pertinent to each cell; the actual integer value assigned to the first index in each case is specified in *geomint.hweb*. $cells_{c_start, j}$ is a pointer to a list of surfaces for cell j , $cells_{c_faces, j}$ is the number of faces, and $cells_{c_surfaces, j}$ is the number of faces plus cut surfaces. $cells_{c_zone, j}$ holds a zone index.

$cells_{c_start, j}$ points into an array *boundaries*. Each element of *boundaries* is a signed integer whose absolute value gives the surface index k and whose sign gives σ_k .

The faces of the j th cell are given by the $cells_{c_faces, j}$ surfaces starting at $boundaries_{cells_{c_start, j}}$ and the cut surfaces are given by the $cells_{c_surfaces, j} - cells_{c_faces, j}$ surfaces starting at $boundaries_{cells_{c_start, j} + cells_{c_faces, j}}$.

surfaces holds four pieces of information about each surface: a count, $surfaces_{s_neg, s_count, k}$, of the number of cells for which this surface is a face on the negative side; a pointer, $surfaces_{s_neg, s_start, k}$, to *neighbors*, giving the indices of the cells; and then the corresponding information for the positive side, $surfaces_{s_pos, s_count, k}$ and $surfaces_{s_pos, s_start, k}$.

surface_sectors holds four pieces of information about the sectors associated with each surface: a count, $surface_sectors_{s_neg, s_count, k}$, of the number of sectors for this surface on the negative side; a pointer, $surface_sectors_{s_neg, s_start, k}$, to *sectors*, giving the sectors for that surface; and then the corresponding information for the positive side, $surface_sectors_{s_pos, s_count, k}$ and $surface_sectors_{s_pos, s_start, k}$.

sector_points gives bounding points on sectors. *strata* gives the stratum number for each sector.

sector_surface gives the surface for each sector.

sector_zone gives the zone for each sector.

surface_coeffs holds the ten coefficients of the equations for the faces.

neighbors is a list of cells for each face.

Each surface may have a transformation associated with it. If $surface_tx_ind_{s_sign(k), t_surf, abs(k)}$ is nonzero, then a point on surface k is transformed to a point on surface $surface_tx_ind_{s_sign(k), t_surf, abs(k)}$ using the transformation $surface_tx_mx_{1..3, 1..4, surface_tx_ind_{s_sign(k), t_mx, abs(k)}}$.

After the geometry is set up, information about the geometry is held in *geom_args*. This can be declared (in non-geometry routines) by *decl_geom*.

```
"geometry.f" 1 ≡
  @m FILE 'geometry.web'
```

2 External interface to geometry routines

The geometry is held in *geom_args*. This should not be altered by any external routine. *decl_geom* declares *geom_args*.

The externally callable routines are:

locate_point(x, zone) returns the cell number for a point or -1 if the point is outside the universal cell. The zone number is returned in *zone*.

check_point(x) checks the point. This checks the point against all the cells. In the event that the point lies in multiple cells, it checks that the cells are neighbors. Returns \mathcal{T} if the check succeeded.

track(type, tmax, x0, v, cell, t, x, cell1, exit_face, cell2) tracks a particle $\mathbf{x} = \mathbf{x}_0 + \mathbf{v}t$ through a zone stopping either at the boundary of the zone (and returning \mathcal{F}) or after a time t_{\max} (and returning \mathcal{T}), whichever comes first. The updated position is given in \mathbf{x} . *cell* is the starting cell. If the result is \mathcal{T} , then the particle did not leave the zone, $t = t_{\max}$, $cell1 = cell2$ is the ending cell, and *exit_face* = 0. Else, if the result is \mathcal{F} , the particle stopped at the boundary of the zone, $t < t_{\max}$, *cell1* is the ending cell belonging to the original zone, the face through which the particle is exiting is given in *exit_face* and the next cell is given in *cell2*. If $cell2 \equiv 0$, then the particle is leaving the universal cell.

intersection_direction(face, x, v) returns the cosine of angle of \mathbf{v} with the face. A positive result implies the particle is leaving the cell.

call *surface_specular(face, x, v, v1)* specularly reflects a particle at \mathbf{x} with velocity \mathbf{v} off a surface. Returns a new velocity $\mathbf{v}1 = \mathbf{v} - 2\nabla f(\nabla f \cdot \mathbf{v})/|\nabla f|^2$, provided v was heading out of the face, otherwise $\mathbf{v}1 = \mathbf{v}$.

call *surface_reflect(face, x, v, w, v1)* makes an arbitrary reflection of a particle at \mathbf{x} with velocity \mathbf{v} off a face. The new velocity is $\mathbf{v}1$. In the coordinate system aligned with the face, this new velocity is input as \mathbf{w} , where $w_i = \mathbf{e}_i \cdot \mathbf{w}$. This coordinate system is given by the unit vectors: \mathbf{e}_3 is inward unit normal to the face, \mathbf{e}_1 is parallel to $(\mathbf{I} - \mathbf{e}_3\mathbf{e}_3) \cdot \mathbf{v}$ and $\mathbf{e}_2 = \mathbf{e}_3 \times \mathbf{e}_1$.

cell_enter(cell, x0, v, enter_face) returns the time of intersection of a trajectory $\mathbf{x}_0 + \mathbf{v}t$ with a cell where \mathbf{x}_0 is not necessarily in the cell. This is most likely only called for the universal cells when doing things like plotting, so we are not so much concerned with speed here. The returned value will be negative if \mathbf{x}_0 is already inside the cell and *geom_infinity* is returned if the trajectory misses the cell entirely. The face through which the trajectory enters is given by *enter_face*.

new_cell(exit_face, x) returns the new cell for particle exiting through face *exit_face*. Returns 0, if there are no cells on the other side of *exit_face*. This indicates that the particle is exiting the universal cell.

The unnamed module.

```
"geometry.f" 2.2 ≡
  ⟨ Functions and Subroutines 3.1 ⟩
```

3 Simple utility routines

This are expanded inline.

Evaluate surface function using Horner's rule.

⟨Functions and Subroutines 3.1⟩ ≡

```

@#if ¬SURFACE_EVAL_MACRO
  function surface_eval(surface_args, x)
    implicit_none_f77
    implicit_none_f90
    real surface_eval    // Function

    decl_surface_args    // Input
    real x3              // Input

    /* write(25,'(1p13e12.5)') x,coeff */
@#if PLANE_SURFACES
    surface_eval = sigma * (x3 * coeff_cz + x2 * coeff_cy + x1 * coeff_cx + coeff_c0)
@#else
    surface_eval = sigma * (x3 * (coeff_czz * x3 + coeff_cyz * x2 + coeff_cxz * x1 + coeff_cz) + x2 * (coeff_cyy *
      x2 + coeff_cxy * x1 + coeff_cy) + x1 * (coeff_cxx * x1 + coeff_cx) + coeff_c0)
@#endif

    return
  end
@#endif

```

See also sections 3.2, 3.3, 3.4, 3.5, 4, 4.1, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 5, 5.1, 5.2, 5.3, 5.4, 6, 6.1, 6.2, 6.3, 7, 7.1, 8, 9, 10, and 10.1.

This code is used in section 2.2.

Evaluate gradient of a surface $\sigma \nabla f(\mathbf{x})$.

⟨Functions and Subroutines 3.1⟩ +≡

```

subroutine surface_gradient(surface_args, x, grad)
  implicit_none_f77
  implicit_none_f90
  decl_surface_args // Input
  real x3
  real grad3 // Output

  @#if PLANE_SURFACES

    grad1 = sigma * coeffcx
    grad2 = sigma * coeffcy
    grad3 = sigma * coeffcz

  @#else

    grad1 = sigma * (coeffcxz * x3 + coeffcxy * x2 + 2 * coeffctx * x1 + coeffcx)
    grad2 = sigma * (coeffcyz * x3 + 2 * coeffcyy * x2 + coeffcxy * x1 + coeffcy)
    grad3 = sigma * (2 * coeffczz * x3 + coeffcyz * x2 + coeffctx * x1 + coeffcz)

  @#endif

  return
end

```

Check whether a point lies within *geom_epsilon* of a surface. If *surface_val* is \mathcal{T} , then the value of the surface function is given in *f*. It is necessary to relax the check a bit if you want to check for a point lying on the face of a cell or inside the cell because that rarely happens. A typical distance would be $f/\nabla f$. If this is smaller than a small number say *geom_epsilon*, we could say that the point lies in the cell. In order to avoid evaluating *f* twice, *surface_val* should be passed as \mathcal{T} if *f* is known and *f* should be passed to this function.

⟨Functions and Subroutines 3.1⟩ +≡

```

function in_surface(surface_args, x, surface_val, f)
  implicit_none_f77
  implicit_none_f90
  logical in_surface // Function

  decl_surface_args // Input
  real x3, f // Input
  logical surface_val
  real grad3, f0 // Local
  external surface_eval, surface_gradient // External
  real surface_eval

  call surface_gradient(surface_args, x, grad)

  if (surface_val) then
    f0 = f
  else
    f0 = surface_eval_a(surface_args, x)
  end if

  in_surface = abs(f0) ≤ geom_epsilon * vc_abs(grad)

  return
end

```

Debuggin version of *in_surface*.

```

⟨Functions and Subroutines 3.1⟩ +=
function in_surface_dump(surface_args, x, surface_val, f)
  implicit_none_f77
  implicit_none_f90
  logical in_surface_dump    // Function

  decl_surface_args    // Input
  real x3, f    // Input
  logical surface_val
  real grad3, f0    // Local
  external surface_eval, surface_gradient    // External
  real surface_eval

  call surface_gradient(surface_args, x, grad)

  if (surface_val) then
    f0 = f
  else
    f0 = surface_eval_a(surface_args, x)
  end if

  write(stderr, *) '  In in_surface_dump'
  write(stderr, *) '  f0= ', f
  write(stderr, *) '  grad= ', grad
  write(stderr, *) '  vc_abs(grad)= ', vc_abs(grad)
  write(stderr, *) '  surface_args= ', surface_args
  in_surface_dump = abs(f0) ≤ geom_epsilon * vc_abs(grad)

  return
end

```

Check whether a point lies on the positive side of a surface (within *geom_epsilon*). Here, if $f \geq 0$ the point lies on the positive side of the surface. Else, if it lies on the negative side within *geom_epsilon* it is accepted.

⟨Functions and Subroutines 3.1⟩ +≡

```

function inside_surface(surface_args, x)
  implicit_none_f77
  implicit_none_f90
  logical inside_surface    // Function

  decl_surface_args    // Input
  real x3    // Input
  real f    // Local
  external in_surface, surface_eval    // External
  real surface_eval
  logical in_surface

  inside_surface = T

  f = surface_eval_a(surface_args, x)

  if (f ≥ zero)
    return

  inside_surface = in_surface(surface_args, x, T, f)

  return
end

```

4 Basic geometrical routines

First check whether a point lies in a cell. *cell_args* is a dummy argument. *cell_surface_info*(*n*) contains information on the surface.

⟨Functions and Subroutines 3.1⟩ +≡

```

function inside_cell(cell_args, x)
  implicit_none_f77
  implicit_none_f90
  logical inside_cell    // Function

  decl_cell_args    // Input
  real x3
  integer n    // Local
  external surface_eval    // External
  real surface_eval

  inside_cell = F

  dosurfaces(n)
  if (surface_eval_a(cell_surface_info_a(n), x) < zero)
    return
  end do

  inside_cell = T

  return
end

```

Sloppy check for whether a point lies inside a particular cell. Sloppy check implies that if x lies within $geom_epsilon$ of the face, it is accepted. A strict check is done on the cut surfaces though.

```

⟨Functions and Subroutines 3.1⟩ +≡
function sloppy_inside(cell_args, x)
  implicit_none_f77
  implicit_none_f90
  logical sloppy_inside // Function

  decl_cell_args // Input
  real x3
  real f0 // Local
  integer n
  external surface_eval, in_surface // External
  real surface_eval
  logical in_surface

  sloppy_inside =  $\mathcal{F}$ 

  dofaces(n) // Do the sloppy check on faces.
  f0 = surface_eval_a(cell_surface_info_a(n), x)
  if (f0 < zero) then
    if ( $\neg$ in_surface(cell_surface_info(n), x,  $\mathcal{T}$ , f0))
      return
    end if
  end do

  docuts(n) // Do strict check on cut surfaces.
  if (surface_eval_a(cell_surface_info_a(n), x) < zero)
    return
  end do

  sloppy_inside =  $\mathcal{T}$ 

  return
end

```

why is the sloppy check done on the faces and cut surfaces in *sloppy_inside* but on surfaces in *inside_cell*.

Check that a face belongs to a cell.

```

⟨Functions and Subroutines 3.1⟩ +=
function face_in_cell(cell_args, face)
  implicit_none_f77
  implicit_none_f90
  logical face_in_cell // Function

  decl_cell_args // Input
  integer face
  integer n // Local

  face_in_cell =  $\mathcal{T}$ 

  dofaces(n)
  if (surf_list_n  $\equiv$  face)
    return
  end do

  face_in_cell =  $\mathcal{F}$ 

  return
end

```

Now similar routines to check whether the point is in a cell, but skipping the check on the face indexed by *face*. this is used when a flight exits a cell through a face to another cell, this face is excluded from the check. A strict check for the condition

$$\sigma_k f_k(\mathbf{x}) \geq 0$$

is done on all the other surfaces.

```

⟨Functions and Subroutines 3.1⟩ +=
function inside_cell_a(cell_args, face, x)
  implicit_none_f77
  implicit_none_f90
  logical inside_cell_a // Function

  decl_cell_args // Input
  integer face
  real x3
  integer n // Local
  external surface_eval // External
  real surface_eval

  inside_cell_a =  $\mathcal{F}$ 

  dosurfaces(n)
  if (surf_list_n  $\neq$  face) then
    if (surface_eval_a(cell_surface_info_a(n), x) < zero)
      return
    end if
  end do

  inside_cell_a =  $\mathcal{T}$ 

  return
end

```

Sloppy equivalent of *inside_cell_a*. Here, the check on the all the surfaces other than the face is not done strictly (the point can lie within *geom_epsilon* of the surface). A strict check is done on the cut surfaces.

⟨Functions and Subroutines 3.1⟩ +≡

```

function sloppy_inside_a(cell_args, face, x, dump)
  implicit_none_f77
  implicit_none_f90
  logical sloppy_inside_a    // Function

  decl_cell_args    // Input
  integer face
  real x3
  logical dump
  integer n    // Local
  real f0
  external surface_eval, in_surface, in_surface_dump    // External
  real surface_eval
  logical in_surface, in_surface_dump

  sloppy_inside_a =  $\mathcal{F}$ 

  dofaced(n)
  if (dump) then
    write(stderr, *) 'In_sloppy_inside_a'
    write(stderr, *) 'n=', n
    write(stderr, *) 'surf_list[n]=', surf_list_n
  end if
  if (surf_list_n  $\neq$  face) then
    f0 = surface_eval_a(cell_surface_info_a(n), x)
    if (dump) then
      write(stderr, *) 'f0=', f0
    end if
    if (f0 < zero) then
      if ( $\neg$ dump) then
        if ( $\neg$ in_surface(cell_surface_info(n), x, T, f0))
          return
        else
          if ( $\neg$ in_surface_dump(cell_surface_info(n), x, T, f0)) then
            write(stderr, *) 'in_surface_came_back_false_for_n=', n
          return
          end if
        end if
      end if
    end if
  end do

  docuts(n)    // Do strict check on cut surfaces.
  if (surface_eval_a(cell_surface_info_a(n), x) < zero)
    return
  end do

  sloppy_inside_a =  $\mathcal{T}$ 

  return
end

```

Check that a point lies on a particular face of a cell. For a point to lie on a face, it has to satisfy 2 conditions : It has to lie within *geom_epsilon* of the face and also strictly inside the cell.

⟨Functions and Subroutines 3.1⟩ +≡

```

function on_face(cell_args, face, x)
  implicit_none_f77
  implicit_none_f90
  logical on_face    // Function

  decl_cell_args    // Input
  integer face
  real x3
  external in_surface, inside_cell_a, face_in_cell    // External
  logical in_surface, inside_cell_a, face_in_cell

  check(face_in_cell(cell_args, face))
  on_face = in_surface(surface_info(face), x,  $\mathcal{F}$ , zero)  $\wedge$  inside_cell_a(cell_args, face, x)

  return
end

```

A sloppy version of *on_face*. This is the same as *on_face* except that the point doesn't have to lie strictly inside the cell; ie., it may lie within *geom_epsilon* of the faces.

⟨Functions and Subroutines 3.1⟩ +≡

```

function sloppy_on(cell_args, face, x)
  implicit_none_f77
  implicit_none_f90
  logical sloppy_on    // Function

  decl_cell_args    // Input
  integer face
  real x3
  external in_surface, sloppy_inside_a, face_in_cell    // External
  logical in_surface, sloppy_inside_a, face_in_cell

  check(face_in_cell(cell_args, face))
  sloppy_on = in_surface(surface_info(face), x,  $\mathcal{F}$ , zero)  $\wedge$  sloppy_inside_a(cell_args, face, x,  $\mathcal{F}$ )

  return
end

```

Check that a point lies outside (-1), inside (1), or on (0) a cell.

```

⟨Functions and Subroutines 3.1⟩ +≡
function cell_compare(cell_args, x)
  implicit_none_f77
  implicit_none_f90
  integer cell_compare // Function

  decl_cell_args // Input
  real x3
  integer n // Local
  real f0
  logical on
  external surface_eval, in_surface // External
  real surface_eval
  logical in_surface

  cell_compare = -1

  docuts(n)
  if (surface_eval_a(cell_surface_info_a(n), x) < zero)
    return
  end do

  on =  $\mathcal{F}$ 

  dofaces(n)
  f0 = surface_eval_a(cell_surface_info_a(n), x)
  if (in_surface(cell_surface_info(n), x,  $\mathcal{T}$ , f0))
    f0 = zero
  if (f0 < zero)
    return
  if (f0 ≡ zero)
    on =  $\mathcal{T}$ 
  end do

  if (on) then
    cell_compare = 0
  else
    cell_compare = 1 // comes here if f0 > 0
  end if

  return
end

```

5 Face intersections

surface_intersect returns the first outgoing intersection of $\mathbf{x}_0 + \mathbf{v}t$ with the surface. If no such intersection is found, return *geom_infinity*. This may return a negative result, if \mathbf{x}_0 is initially on the wrong side of the face. (In this case the particle should pass to the neighboring cell.)

To find the intersection of the surface $f(x)$ with the path of the particle given by $\mathbf{x}_0 + \mathbf{v}t$, we put this in the equation

$$f(\mathbf{x}_0 + \mathbf{v}t) = 0.$$

We get a quadratic equation,

$$at^2 + 2ht + c = 0,$$

where a , h and c are given as

$$a = \sigma [v_3(\text{coeff}_{czz}v_3 + \text{coeff}_{cyz}v_2 + \text{coeff}_{cxz}v_1) + v_2(\text{coeff}_{cyy}v_2 + \text{coeff}_{cxy} * v_1) + \text{coeff}_{cxx} * v_1^2]$$

$h = \frac{1}{2} \nabla f|_{\mathbf{x}_0} \cdot \mathbf{v}$, $c = f(\mathbf{x}_0)$ solving this equation gives the value of t (time) taken for the particle to intersect the first face in its direction.

⟨Functions and Subroutines 3.1⟩ +=

```

function surface_intersect(surface_args, x0, v)
  implicit_none_f77
  implicit_none_f90
  real surface_intersect // Function

  decl_surface_args // Input
  real x0_3, v_3
  real a, h, c, temp_3, disc // Local
  external surface_eval // External
  real surface_eval

  @if DEBUG
    external surface_intersection_direction, in_surface // External
    real surface_intersection_direction
    logical in_surface
  @endif

  @if PLANE_SURFACES
    a = zero
  @else
    a = sigma*(v_3*(coeff_czz*v_3+coeff_cyz*v_2+coeff_cxz*v_1)+v_2*(coeff_cyy*v_2+coeff_cxy*v_1)+coeff_cxx*v_1^2)
  @endif
  call surface_gradient(surface_args, x0, temp)
  h = half * vc_product(temp, v)
  c = surface_eval_a(surface_args, x0)

  @if DEBUG
    if (c < zero) then
      @if 0
        assert(in_surface(surface_args, x0, T, c))
      @else
        if (¬(in_surface(surface_args, x0, T, c))) then
          write (stderr, *) '␣␣␣zero␣in␣surface␣intersect'

```

```

    write(stderr, *) 'c=', c
    write(stderr, *) 'h=', h
    write(stderr, *) 'x0=', x0
    write(stderr, *) 'v=', v
    write(stderr, *) 'surface_args=', surface_args
    assert(in_surface(surface_args, x0, T, c))
  end if
@#endif
end if
@#endif

surface_intersect = geom_infinity

if (a ≠ zero) then // Quadratic equation
  disc = h2 - a * c
  if (disc ≤ zero) then // No roots
    if (a < zero) then // but if on wrong side, return maximum
      surface_intersect = -h / a
    end if
  else // disc > zero two distinct roots
    disc = sqrt(disc)
    if (h < zero) then
      surface_intersect = -c / (h - disc) // Avoid cancellation in -(h + disc) / a
    else // h ≥ zero
      if (a < zero)
        surface_intersect = -(h + disc) / a
      end if
    end if
  else if (h ≠ zero) then // Linear equation
    if (h < zero)
      surface_intersect = -c / (two * h)
    end if // else the equation is a constant
  end if

@#if DEBUG
  if (surface_intersect < zero) then
    assert(c < zero) // Should only have a negative intersection if point is on wrong side
  end if /* Check result, except cases with no intersection or zero velocity */
  if (surface_intersect * vc_abs(v) < geom_large) then
    vc_xvt(x0, v, surface_intersect, temp)
    assert(in_surface(surface_args, temp, F, zero))
    assert(surface_intersection_direction(surface_args, temp, v) ≥ -epsilon_angle)
  end if
@#endif

return
end

```

Compute direction of intersection. Returns cosine of angle of \mathbf{v} with $-\sigma\nabla f$. A positive result implies the particle is leaving the cell.

⟨Functions and Subroutines 3.1⟩ +≡

```

function surface_intersection_direction(surface_args, x, v)
  implicit_none_f77
  implicit_none_f90
  real surface_intersection_direction    // Function

  decl_surface_args    // Input
  real x3, v3
  real grad3    // Local
  external surface_gradient    // External

  call surface_gradient(surface_args, x, grad)
  surface_intersection_direction = -vc_product(grad, v) / (vc_abs(v) * vc_abs(grad))
  if (surface_intersection_direction < -one)
    surface_intersection_direction = -one
  if (surface_intersection_direction > one)
    surface_intersection_direction = one
  assert(abs(surface_intersection_direction) ≤ one)

  return
end

```

An externally callable version of *surface_intersection_direction*.

⟨Functions and Subroutines 3.1⟩ +≡

```

function intersection_direction(face, x, v)
  implicit_none_f77
  gi_common
  implicit_none_f90
  real intersection_direction    // Function
  integer face
  real x3, v3    // Input
  external surface_intersection_direction    // External
  real surface_intersection_direction

  intersection_direction = surface_intersection_direction(surface_info(face), x, v)

  return
end

```

Specularly reflect a particle at \mathbf{x} with velocity \mathbf{v} off a surface. Returns a new velocity $\mathbf{v}' = \mathbf{v} - 2\nabla f(\nabla f \cdot \mathbf{v})/|\nabla f|^2$, provided v was heading out of the face, otherwise $\mathbf{v}' = \mathbf{v}$.

⟨Functions and Subroutines 3.1⟩ +≡

```

subroutine surface_specular(face, x, v, v1)
  implicit none_f77
  gi_common
  implicit none_f90
  integer face // Input
  real x3, v3
  real v1_3 // Output
  real grad3, vperp // Local

  vc_decls
  external surface_gradient // External

  call surface_gradient(surface_info(face), x, grad)
  vc_unit(grad, grad)
  vperp = vc_product(grad, v) // The perpendicular component of  $\mathbf{v}$ .
  vc_xvt(v, grad, (-one + sign(one, vperp)) * vperp, v1)

  return
end

```

Arbitrary reflection of a particle at \mathbf{x} with velocity \mathbf{v} of a face. The new velocity is \mathbf{w} in the coordinate system where $w_i = \mathbf{e}_i \cdot \mathbf{w}$, and \mathbf{e}_3 is unit normal to the face, \mathbf{e}_1 is parallel to $(\mathbf{I} - \mathbf{e}_3\mathbf{e}_3) \cdot \mathbf{v}$ and $\mathbf{e}_2 = \mathbf{e}_3 \times \mathbf{e}_1$.

(Functions and Subroutines 3.1) +≡

```

subroutine surface_reflect(face, x, v, w, v1)
  implicit_none_f77
  gi_common
  implicit_none_f90
  integer face // Input
  real x3, v3, w3
  real v1_3 // Output
  real e3_3, e1_3, e2_3, e3_3, vunit_3 // Local
  real vperp
  equivalence (e1, e1_1), (e2, e1_2), (e3, e1_3)

  vc_decls
  integer i
  external surface_gradient // External

  call surface_gradient(surface_info(face), x, e3)
  vc_unit(e3, e3)

  if (vc_abs(v) > zero) then
    vc_unit(v, vunit)
    vperp = abs(one + vc_product(e3, vunit))
  else
    vperp = one
  end if

  vc_xvt(v, e3, -vc_product(e3, v), e1)
  if ((vc_abs(e1) ≤ vc_abs(v) * const(4.) * epsilon) ∨ (vperp ≤ const(4.) * epsilon)) then
    /*  $\mathbf{v}_{\parallel} = 0$ , so choose an arbitrary direction in the surface. This calculation appears particularly
    prone to roundoff errors in the (frequent) case of normal incidence (at the start of a flight).
    The original fix of comparing  $e1$  with  $\epsilon$  has failed multiple times in simulations by
    Takenaga. However, these failures were just slightly larger than  $\epsilon$  (less than 10%).
    Argue that the roundoff error applies separately to each vector component so that  $\sqrt{3}\epsilon$  is a
    more appropriate upper bound for the total error. Have now found one case that exceeds this
    by about 1%. Use  $2\epsilon$  to allow for a little more slop (this coefficient should not be allowed to
    grow continuously!). The subsequent assert using  $\sqrt{\epsilon}$  will still catch those cases which are not
    caught by this test, but should be treated with suspicion. Another case arose with a nearly
    horizontal target in which  $e1$  was orders of magnitude larger than  $\epsilon$  due to roundoff error.
    Developed the expression for  $vperp$  as an alternative test for parallel vectors; it seems to work
    better for this situation. */
    vc_set(e2, one, zero, zero)
    vc_cross(e3, e2, e1)
    if (vc_abs(e1) ≤ vc_abs(v) * const(3.) * epsilon) then
      /* Found a case (box geometry with plate source) that slipped through this with  $\vec{e}_2$  and  $\vec{e}_3$ 
      differing only by roundoff error. This result argued for loosening this test as well. */
      e2_2 = one
      vc_cross(e3, e2, e1)
    end if /* Subsequently had a case of normal incidence on a nearly vertical target. The
    normal incidence caused the first "if" above to be entered. However, the surface had enough
    inclination to get past the second one (and this was numerically a reasonable thing to do).
    The case then failed the subsequent assert because of the magnitude of  $v$ . Decided that the
    test in this assert must be dimensionless (as it would be in the off-normal incidence case), so

```

```
    add the factor of vc_abs(v) here. */
  if (vc_abs(v) > zero) then
    vc_scale(vc_abs(v), e1, e1)
  end if
end if
if (vc_abs(v) > zero) then
  assert(vc_abs(e1) / vc_abs(v) > sqrt(epsilon)) // Test, to see if anything squeaks by
end if

vc_unit(e1, e1)
vc_cross(e3, e1, e2) //  $\mathbf{e}_{1,2,3}$  form a right-handed basis set.

do i = 1, 3
   $v1_i = w_1 * e_{i, 1} + w_2 * e_{i, 2} + w_3 * e_{i, 3}$ 
end do

return
end
```

6 Cell intersection

Calculate intersection of a trajectory $\mathbf{x}_0 + \mathbf{v}t$ with a cell. *cell_intersect* returns time of the intersection as function value. The face through which the trajectory exits is given by *exit_face*. The time taken to intersect each of the faces is calculated by the function *surface_intersect*. The smallest of these times is taken as *cell_intersect*. The face of intersection is saved in *ns*. If the value of *cell_intersect* is < 0 , it is taken as 0.

(Functions and Subroutines 3.1) +≡

```

function cell_intersect(cell_args, x0, v, exit_face)
  implicit_none_f77
  implicit_none_f90
  real cell_intersect // Function

  decl_cell_args // Input
  real x0_3, v_3
  integer exit_face // Output
  real t // Local
  integer n, ns
  external surface_intersect // External
  real surface_intersect

  cell_intersect = geom_infinity

  dofaced(n)
  t = surface_intersect(cell_surface_info(n), x0, v)
  if (t < cell_intersect) then
    cell_intersect = t
    ns = n
  end if
end do

  cell_intersect = max(zero, cell_intersect)
  assert(cell_intersect < geom_infinity)

  exit_face = surf_list_ns

return
end

```

Determine the new cell for particle exiting through face *exit_face*. Returns 0, if there are no cells on the other side of *exit_face*. This indicates that the particle is exiting the universal cell. Using the face index of *exit_face*, the starting index which points to the list of cells for that face and the number of cells for the face is found. we check whether *x* lies strictly inside any of the cells and return if it does. Otherwise, a relaxed check is done for the same.

⟨Functions and Subroutines 3.1⟩ +=

```

function new_cell(exit_face, x)
  implicit_none_f77
  gi_common
  implicit_none_f90
  integer new_cell // Function
  integer exit_face // Input
  real x3
  integer n, s, i, j, face // Local
  external inside_cell_a, sloppy_inside_a // External
  logical inside_cell_a, sloppy_inside_a

  face = -exit_face // Face index for new cell
  new_cell = 0

  n = surfaces_s_sign(face), s_count, abs(face) // Number of cells
  s = surfaces_s_sign(face), s_start, abs(face) // Starting index

  if (n == 0)
    return // Outermost surface

  do i = s, s + n - 1 // Try with the strict conditions
    j = neighbors_i // Get cell number
    if (j > 0) then // Skip if it's the universe
      if (inside_cell_a(cell_info(j), face, x)) then
        new_cell = j
        return
      end if
    end if
  end do

  do i = s, s + n - 1 // Now relax the criteria
    j = neighbors_i
    if (j > 0) then
      if (sloppy_inside_a(cell_info(j), face, x, F)) then
        new_cell = j
        return
      end if
    end if
  end do

  @#if 0
  assert(new_cell > 0) // Can't find a satisfactory new cell
  @#else
  if (-(new_cell > 0)) then
    write (stderr, *) 'new_cell=<=0'
    write (stderr, *) 'x=', x
    write (stderr, *) 'exit_face=', exit_face
    write (stderr, *) 'new_cell=', new_cell
    write (stderr, *) 'n=', n
  
```

```
write(stderr, *) 's=', s
do i = s, s + n - 1
  j = neighbors_i
  write(stderr, *) 'For i=', i, ' j=', j
  if (j > 0) then
    if (sloppy_inside_a(cell_info(j), face, x, T)) then
      write(stderr, *) 'WHOA! This did not work above!'
    end if
  end if
end do
assert(new_cell > 0) // Can't find a satisfactory new cell
end if
@#endif

return
end
```

Track a particle $\mathbf{x} = \mathbf{x}_0 + \mathbf{v}t$ through a zone stopping either at the boundary of the zone (and returning \mathcal{F}) or after a time t_{\max} (and returning \mathcal{T}), whichever comes first. Give the updated position \mathbf{x} and the new cell $cell1$. If an intersection is found, the $cell1$ belongs to the original zone and, in addition, the face through which the particle is exiting is given in $exit_face$ and the new cell is given in $cell2$.

⟨Functions and Subroutines 3.1⟩ +≡

```
function track(type, tmax, x0, v, cell, t, x, cell1, exit_face, cell2, sector1, sector2)
```

```
  implicit none f77
```

```
  gi_common
```

```
  sc_common
```

```
  implicit none f90
```

```
  logical track // Function
```

```
  real tmax, x03, v3 // Input
```

```
  integer type, cell
```

```
  real t, x3 // Output
```

```
  integer cell1, exit_face, cell2, sector1, sector2
```

```
  integer j, k, k1, tx, i, start, count // Local
```

```
  real dt, tleft, t1, v13, temp3
```

```
  external cell_intersect, new_cell, in_surface // External
```

```
  real cell_intersect
```

```
  integer new_cell
```

```
  logical in_surface
```

```
  assert(type ≡ pt_geometry ∨ type ≡ pt_neutral ∨ type ≡ pt_ion)
```

```
  assert(cell > 0)
```

```
  j = cell
```

```
  vc_copy(x0, x)
```

```
  if (type ≡ pt_ion) then
```

```
    vc_set(v1, zero, zero, zero)
```

```
  else
```

```
    vc_copy(v, v1)
```

```
  end if
```

```
  t1 = zero
```

```
  sector1 = 0
```

```
  sector2 = 0
```

```
loop: continue
```

```
  cell1 = j // Update cell
```

```
  tleft = tmax - t1 /* Some things in cell_intersect and so forth choke on v1 ≡ 0 */
```

```
  if (vc_abs(v1) > zero) then
```

```
    dt = min(tleft, cell_intersect(cell_info(cell1), x, v1, k))
```

```
  else
```

```
    dt = tleft
```

```
  end if
```

```
  t1 = t1 + dt
```

```
  vc_xvt(x, v1, dt, x) // Update position
```

```
  if (dt ≡ tleft) then // We're done
```

```
    t = tmax // Set t exactly to tmax
```

```
    exit_face = 0
```

```
    cell2 = j
```

```
    track =  $\mathcal{T}$ 
```

```
    return
```

```
  end if
```

```
  /* Check for outgoing sector. This used to be inside the check on type below. */
```

```

count = surface_sectorss_sign(k), s_count, abs(k)
if (count > 0) then
  start = surface_sectorss_sign(k), s_start, abs(k)
  do i = 0, count - 1
    if (cellsc_zone, cell1  $\equiv$  sector_zonesectorsstart+i) then
      sector1 = sectorsstart+i
      assert(sector_surfacesector1  $\equiv$  k)
      goto break1
    end if
  end do
break1: continue
end if
if (type  $\neq$  pt_geometry) then
  /* Process the transformation */
  k1 = surfaces_tx_inds_sign(k), t_surf, abs(k)
  if (k1  $\neq$  0) then
    tx = surfaces_tx_inds_sign(k), t_mx, abs(k)
  @#if 0
    write(stderr, *) 'Surfacetx:□', k, k1
    write(stderr, *) 'Oldx,v:□', (xi, i = 1, 3), (v1i, i = 1, 3)
  @#endif
  k = k1
  vc_copy(x, temp)
  call transform(surfaces_tx_mx1, 1, tx, temp, x)
  vc_copy(v1, temp)
  call transform_velocity(surfaces_tx_mx1, 1, tx, temp, v1)
  if (type  $\neq$  pt_ion) then
    vc_copy(v1, v)
  end if
  @#if 0
    write(stderr, *) 'Newx,v:□', (xi, i = 1, 3), (v1i, i = 1, 3)
  @#endif
  assert(in_surface(surface_info(k), x,  $\mathcal{F}$ , zero))
  end if
end if
j = new_cell(k, x) // Find next cell
/* Check for incoming sector */
count = surface_sectorss_sign(-k), s_count, abs(k)
if (count > 0) then
  start = surface_sectorss_sign(-k), s_start, abs(k)
  do i = 0, count - 1
    if (cellsc_zone, j  $\equiv$  sector_zonesectorsstart+i) then
      sector2 = sectorsstart+i
      assert(sector_surfacesector2  $\equiv$  -k)
      goto break2
    end if
  end do
break2: continue
end if

/* Keep tracking if still in same zone and if not on a sector */

```

```
    if ( $cells_{c\_zone}, cell1 \equiv cells_{c\_zone}, j \wedge sector1 \equiv 0 \wedge sector2 \equiv 0$ )  
        goto loop  
  
     $t = t1$   
     $exit\_face = k$   
     $cell2 = j$   
     $track = \mathcal{F}$   
  
    return  
end
```

Find entering intersection of a trajectory $\mathbf{x}_0 + \mathbf{v}t$ with a cell where \mathbf{x}_0 is not necessarily in the cell. This is most likely only called for the universal cells when doing things like plotting, so we are not so much concerned with speed here. *cell_enter* returns time of the entry as function value. This will be negative if \mathbf{x}_0 is already inside the cell and *geom_infinity* is returned if the trajectory misses the cell entirely. The face through which the trajectory enters is given by *enter_face*.

If the point \mathbf{x}_0 doesn't lie inside the given *cell*, get the surface index of the surface of *cell* of which x_0 lies on the +ve side from *boundaries* and find the time for intersection with this surface. Advance \mathbf{x}_0 by $\mathbf{v}t$ and check if it lies inside *cell*. If it does, get the index of the surface and return as *enter_face*.

⟨Functions and Subroutines 3.1⟩ +≡

```

function cell_enter(cell, x0, v, enter_face)
  implicit_none_f77
  gi_common
  implicit_none_f90
  real cell_enter // Function
  integer cell // Input
  real x0_3, v_3
  integer enter_face // Output
  real x_3, v1_3, t // Local
  integer n, ns
  external surface_intersect, cell_intersect, inside_cell, inside_cell_a, inside_surface // External
  real surface_intersect, cell_intersect
  logical inside_cell, inside_cell_a, inside_surface

  cell_enter = geom_infinity
  ns = 0
  if (inside_cell(cell_info(cell), x0)) then // Already inside the cell
    vc_scale(-one, v, v1) // Reverse v
    cell_enter = -cell_intersect(cell_info(cell), x0, v1, enter_face) // and call cell_intersect
  else // On boundary or outside cell
    do n = cells_c_start, cell, cells_c_start, cell + cells_c_faces, cell - 1
      if (inside_surface(surface_info(-boundaries_n), x0)) then
        t = surface_intersect(surface_info(-boundaries_n), x0, v)
        vc_xvt(x0, v, t, x)
        if (inside_cell_a(cell_info(cell), boundaries_n, x)) then
          if (t < cell_enter) then
            cell_enter = t
            ns = n
          end if
        end if
      end if
    end do
    if (ns > 0)
      enter_face = ns
    end if
  return
end

```

7 Locate point in geometry

The first version *locate_point* assumes that the geometry is consistent. It returns the cell number of the point, or -1 if the point is outside the universal cell. The zone number is returned as an argument.

(Functions and Subroutines 3.1) +=

```

function locate_point(x, zone)
  implicit_none_f77
  gi_common
  implicit_none_f90
  integer locate_point // Function
  real x3 // Input
  integer zone // Output

  integer k, c, cell1, cell2, sector1, sector2 // Local
  real x0_3, x1_3, v3, t, tmax
  logical done
  external track, new_cell, cell_enter // External
  real cell_enter
  integer new_cell
  logical track

  external sloppy_inside // External
  logical sloppy_inside

  vc_set(v, one, zero, zero)
  t = cell_enter(0, x, v, k)
  if (t > geom_epsilon) then
    locate_point = -1
    goto break
  end if

  vc_xvt(x, v, t, x0)
  c = new_cell(-k, x0)
  assert(c > 0)

  if (t ≥ zero) then
    locate_point = c
    goto break
  end if

  tmax = -t

loop: continue
  done = track(TRACK_BASIC, tmax, x0, v, c, t, x1, cell1, k, cell2, sector1, sector2)
  if (done) then
    locate_point = cell1
    goto break
  end if
  vc_copy(x1, x0)
  tmax = tmax - t
  c = cell2
  goto loop

break: continue

```

```
if (locate_point > 0) then  
  assert(sloppy_inside(cell_info(locate_point), x)  
  zone = cellsc_zone, locate_point  
else  
  zone = -1  
end if  
  
return  
end
```

Now a more thorough check on the location of a point. This checks the point against all the cells. In the event that the point lies in multiple cells, it checks that the cells are neighbors.

⟨ Functions and Subroutines 3.1 ⟩ +≡

```

function check_point(x)
  implicit_none_f77
  gi_common
  implicit_none_f90
  logical check_point // Function
  real x3 // Input
  integer j, l // Local
  logical interior, onface
  external sloppy_inside, inside_cell, cell_compare // External
  logical sloppy_inside, inside_cell
  integer cell_compare

  check_point =  $\mathcal{F}$ 
  interior =  $\mathcal{F}$ 
  onface =  $\mathcal{F}$ 

  if ( $\neg$ sloppy_inside(cell_info(0), x)) then
    do j = 1, ncells
      if (inside_cell(cell_info(j), x))
        return
      end do
    check_point =  $\mathcal{T}$ 
  else
    do j = 1, ncells
      l = cell_compare(cell_info(j), x)
      if (l ≥ 0) then
        if (interior)
          return // Already with another cell
        if (l > 0) then
          if (onface)
            return // Already on face of another cell
          interior =  $\mathcal{T}$ 
        else // l ≡ 0
          onface =  $\mathcal{T}$ 
        end if
      end if
    end do
    check_point = onface ∨ interior
  end if

  return
end

```

8 A simple consistency checker for locatations

```

⟨ Functions and Subroutines 3.1 ⟩ +=
function check_location(lc_dummy(x))
  implicit_none_f77
  gi_common
  implicit_none_f90
  logical check_location // Function

  lc_decl(x) // Input
  external sloppy_inside, face_in_cell, in_surface // External
  logical sloppy_inside, face_in_cell, in_surface

  check_location =  $\mathcal{F}$ 

  if (lc_cell(x) ≤ 0 ∨ lc_cell(x) > ncells)
    return
  if (lc_zone(x) ≤ 0)
    return
  if (lc_zone(x) ≠ cellsc_zone, lc_cell(x))
    return
  if (¬sloppy_inside(cell_info(lc_cell(x)), lc_x(x)))
    return
  if (lc_face(x) ≠ 0) then
    if (abs(lc_face(x)) > nsurfaces)
      return
    if (lc_cell_next(x) ≤ 0 ∨ lc_cell(x) > ncells)
      return
    if (lc_zone_next(x) ≤ 0)
      return
    if (lc_zone_next(x) ≠ cellsc_zone, lc_cell_next(x))
      return
    if (¬sloppy_inside(cell_info(lc_cell_next(x)), lc_x(x)))
      return
    if (¬in_surface(surface_info(lc_face(x)), lc_x(x),  $\mathcal{F}$ , zero))
      return
    if (¬face_in_cell(cell_info(lc_cell(x)), lc_face(x)))
      return
    if (¬face_in_cell(cell_info(lc_cell_next(x)), -lc_face(x)))
      return
    end if
  check_location =  $\mathcal{T}$ 

  return
end

```

9 Read in the geometry specification from a netcdf file

⟨ Functions and Subroutines 3.1 ⟩ +=

```

subroutine read_geometry
  implicit_none_f77
  gi_common    // Common
  zn_common
  rf_common
  sc_common
  de_common
  mp_common
  implicit_none_f90
  nc_decls    // Local
  gi_ncdecl
  zn_ncdecl
  cm_ncdecl
  mp_decls
  ⟨ Memory allocation interface 0 ⟩
  sc_ncdecl
  de_ncdecl
  integer fileid
  character*FILELEN tempfile    // local
  if (mpi_master) then
    tempfile = filenames_array_geometryfile
    assert(tempfile ≠ char_undef)
    fileid = ncopen(tempfile, NC_NOWRITE, nc_stat)
    cm_ncread(fileid)
    gi_ncread(fileid)
    zn_ncread(fileid)
    sc_ncread(fileid)
    de_ncread(fileid)
    call ncclose(fileid, nc_stat)
  endif
  cm_mpibcast
  gi_mpibcast
  zn_mpibcast
  sc_mpibcast
  de_mpibcast
  return
end

```

10 The basic transformation routines

Transform a position vector, $\mathbf{y} = \mathbf{M} \cdot \mathbf{x}$. \mathbf{x} and \mathbf{y} can overlap. THIS SHOULD NOT BE ASSUMED.

⟨Functions and Subroutines 3.1⟩ +≡

```

subroutine transform(m, x, y)
  implicit_none_f77
  implicit_none_f90
  real m3, 4, x3 // Input
  real y3 // Output
  integer i // Local
  real temp3

  do i = 1, 3
    tempi = mi, 1 * x1 + mi, 2 * x2 + mi, 3 * x3 + mi, 4
  end do

  do i = 1, 3
    yi = tempi
  end do
  return
end

```

Transform a velocity vector, $\mathbf{w} = \mathbf{M} \cdot \mathbf{v}$. \mathbf{x} and \mathbf{y} can overlap. THIS SHOULD NOT BE ASSUMED.

⟨Functions and Subroutines 3.1⟩ +≡

```

subroutine transform_velocity(m, v, w)
  implicit_none_f77
  implicit_none_f90
  real m3, 4, v3 // Input
  real w3 // Output
  integer i // Local
  real temp3

  do i = 1, 3
    tempi = mi, 1 * v1 + mi, 2 * v2 + mi, 3 * v3
  end do

  do i = 1, 3
    wi = tempi
  end do
  return
end

```

11 INDEX

- a*: [5](#).
abs: [1](#), [3.3](#), [3.4](#), [5.1](#), [5.4](#), [6.1](#), [6.2](#), [8](#).
assert: [5](#), [5.1](#), [5.4](#), [6](#), [6.1](#), [6.2](#), [7](#), [9](#).
boundaries: [1](#), [6.3](#).
break: [7](#).
break1: [6.2](#).
break2: [6.2](#).
c: [5](#), [7](#).
c_faces: [1](#), [6.3](#).
c_start: [1](#), [6.3](#).
c_surfaces: [1](#).
c_zone: [1](#), [6.2](#), [7](#), [8](#).
cell: [2](#), [6.2](#), [6.3](#).
cell_args: [4](#), [4.1](#), [4.3](#), [4.4](#), [4.5](#), [4.6](#), [4.7](#), [4.8](#), [6](#).
cell_compare: [4.8](#), [7.1](#).
cell_enter: [2](#), [6.3](#), [7](#).
cell_info: [6.1](#), [6.2](#), [6.3](#), [7](#), [7.1](#), [8](#).
cell_intersect: [6](#), [6.2](#), [6.3](#).
cell_surface_info: [4](#), [4.1](#), [4.5](#), [4.8](#), [6](#).
cell_surface_info_a: [4](#), [4.1](#), [4.4](#), [4.5](#), [4.8](#).
cells: [1](#), [6.2](#), [6.3](#), [7](#), [8](#).
cell1: [2](#), [6.2](#), [7](#).
cell2: [2](#), [6.2](#), [7](#).
char_undef: [9](#).
check: [4.6](#), [4.7](#).
check_location: [8](#).
check_point: [2](#), [7.1](#).
cm_mpibcast: [9](#).
cm_ncdecl: [9](#).
cm_ncread: [9](#).
coeff: [3.1](#), [3.2](#), [5](#).
const: [5.4](#).
count: [6.2](#).
cx: [3.1](#), [3.2](#).
cxz: [3.1](#), [3.2](#), [5](#).
cxy: [3.1](#), [3.2](#), [5](#).
cxz: [3.1](#), [3.2](#), [5](#).
cy: [3.1](#), [3.2](#).
cyy: [3.1](#), [3.2](#), [5](#).
cyz: [3.1](#), [3.2](#), [5](#).
cz: [3.1](#), [3.2](#).
czz: [3.1](#), [3.2](#), [5](#).
c0: [3.1](#).
de_common: [9](#).
de_mpibcast: [9](#).
de_ncdecl: [9](#).
de_ncread: [9](#).
DEBUG: [5](#).
decl_cell_args: [4](#), [4.1](#), [4.3](#), [4.4](#), [4.5](#), [4.6](#), [4.7](#), [4.8](#), [6](#).
decl_geom: [1](#), [2](#).
decl_surface_args: [3.1](#), [3.2](#), [3.3](#), [3.4](#), [3.5](#), [5](#), [5.1](#).
disc: [5](#).
docuts: [4.1](#), [4.5](#), [4.8](#).
dofaces: [4.1](#), [4.3](#), [4.5](#), [4.8](#), [6](#).
done: [7](#).
dosurfaces: [4](#), [4.4](#).
dt: [6.2](#).
dump: [4.5](#).
e: [5.4](#).
enter_face: [2](#), [6.3](#).
epsilon: [5.4](#).
epsilon_angle: [5](#).
exit_face: [2](#), [2.1](#), [6](#), [6.1](#), [6.2](#).
e1: [5.4](#).
e2: [5.4](#).
e3: [5.4](#).
f: [3.3](#), [3.4](#), [3.5](#).
face: [2](#), [4.3](#), [4.4](#), [4.5](#), [4.6](#), [4.7](#), [5.2](#), [5.3](#), [5.4](#), [6.1](#).
face_in_cell: [4.3](#), [4.6](#), [4.7](#), [8](#).
FILE: [1](#).
fileid: [9](#).
FILELEN: [9](#).
filenames_array: [9](#).
f0: [3.3](#), [3.4](#), [4.1](#), [4.5](#), [4.8](#).
geom_args: [1](#), [2](#).
geom_epsilon: [3.3](#), [3.4](#), [3.5](#), [4.1](#), [4.5](#), [4.6](#), [4.7](#), [7](#).
geom_epsilon: [3.5](#).
geom_infinity: [2](#), [5](#), [6](#), [6.3](#).
geom_large: [5](#).
geometryfile: [9](#).
geomint: [1](#).
gi_common: [5.2](#), [5.3](#), [5.4](#), [6.1](#), [6.2](#), [6.3](#), [7](#), [7.1](#), [8](#), [9](#).
gi_mpibcast: [9](#).
gi_ncdecl: [9](#).
gi_ncread: [9](#).
grad: [3.2](#), [3.3](#), [3.4](#), [5.1](#), [5.3](#).
h: [5](#).
half: [5](#).
hweb: [1](#).
i: [5.4](#), [6.1](#), [6.2](#), [10](#), [10.1](#).
implicit_none_f77: [3.1](#), [3.2](#), [3.3](#), [3.4](#), [3.5](#), [4](#), [4.1](#), [4.3](#), [4.4](#), [4.5](#), [4.6](#), [4.7](#), [4.8](#), [5](#), [5.1](#), [5.2](#), [5.3](#), [5.4](#), [6](#), [6.1](#), [6.2](#), [6.3](#), [7](#), [7.1](#), [8](#), [9](#), [10](#), [10.1](#).
implicit_none_f90: [3.1](#), [3.2](#), [3.3](#), [3.4](#), [3.5](#), [4](#), [4.1](#), [4.3](#), [4.4](#), [4.5](#), [4.6](#), [4.7](#), [4.8](#), [5](#), [5.1](#), [5.2](#), [5.3](#), [5.4](#), [6](#), [6.1](#), [6.2](#), [6.3](#), [7](#), [7.1](#), [8](#), [9](#), [10](#), [10.1](#).
in_surface: [3.3](#), [3.4](#), [3.5](#), [4.1](#), [4.5](#), [4.6](#), [4.7](#), [4.8](#), [5](#), [6.2](#), [8](#).

- in_surface_dump*: 3.4, 4.5.
inside_cell: 4, 4.2, 6.3, 7.1.
inside_cell_a: 4.4, 4.5, 4.6, 6.1, 6.3.
inside_surface: 3.5, 6.3.
interior: 7.1.
intersection_direction: 2, 5.2.
j: 6.1, 6.2, 7.1.
k: 6.2, 7.
k1: 6.2.
l: 7.1.
lc_cell: 8.
lc_cell_next: 8.
lc_decl: 8.
lc_dummy: 8.
lc_face: 8.
lc_x: 8.
lc_zone: 8.
lc_zone_next: 8.
locate_point: 2, 7.
loop: 6.2, 7.
m: 10, 10.1.
max: 6.
min: 6.2.
mp_common: 9.
mp_decls: 9.
mpi_master: 9.
n: 4, 4.1, 4.3, 4.4, 4.5, 4.8, 6, 6.1, 6.3.
nc_decls: 9.
NC_NOWRITE: 9.
nc_stat: 9.
ncclose: 9.
ncells: 1, 7.1, 8.
ncopen: 9.
neighbors: 1, 6.1.
new_cell: 2.1, 6.1, 6.2, 7.
ns: 6, 6.3.
nsurfaces: 8.
on: 4.8.
on_face: 4.6, 4.7.
one: 5.1, 5.3, 5.4, 6.3, 7.
onface: 7.1.
PLANE_SURFACES: 3.1, 3.2, 5.
pt_geometry: 6.2.
pt_ion: 6.2.
pt_neutral: 6.2.
read_geometry: 9.
rf_common: 9.
s: 6.1.
s_count: 1, 6.1, 6.2.
s_neg: 1.
s_pos: 1.
s_sign: 1, 6.1, 6.2.
s_start: 1, 6.1, 6.2.
sc_common: 6.2, 9.
sc_mpibroadcast: 9.
sc_ncdecl: 9.
sc_ncread: 9.
sector_points: 1.
sector_surface: 1, 6.2.
sector_zone: 1, 6.2.
sectors: 1, 6.2.
sector1: 6.2, 7.
sector2: 6.2, 7.
sigma: 3.1, 3.2, 5.
sign: 5.3.
sloppy_inside: 4.1, 4.2, 7, 7.1, 8.
sloppy_inside_a: 4.5, 4.7, 6.1.
sloppy_on: 4.7.
sqrt: 5, 5.4.
start: 6.2.
stderr: 3.4, 4.5, 5, 6.1, 6.2.
strata: 1.
surf_list: 4.3, 4.4, 4.5, 6.
surface_args: 3.1, 3.2, 3.3, 3.4, 3.5, 5, 5.1.
surface_coeffs: 1.
surface_eval: 3.1, 3.3, 3.4, 3.5, 4, 4.1, 4.4, 4.5, 4.8, 5.
surface_eval_a: 3.3, 3.4, 3.5, 4, 4.1, 4.4, 4.5, 4.8, 5.
SURFACE_EVAL_MACRO: 3.1.
surface_gradient: 3.2, 3.3, 3.4, 5, 5.1, 5.3, 5.4.
surface_info: 4.6, 4.7, 5.2, 5.3, 5.4, 6.2, 6.3, 8.
surface_intersect: 5, 6, 6.3.
surface_intersection_direction: 5, 5.1, 5.2.
surface_reflect: 2, 5.4.
surface_sectors: 1, 6.2.
surface_specular: 2, 5.3.
surface_tx_ind: 1.
surface_tx_mx: 1.
surface_val: 3.3, 3.4.
surfaces: 1, 6.1.
surfaces_tx_ind: 6.2.
surfaces_tx_mx: 6.2.
t: 6, 6.2, 6.3, 7.
t_mx: 1, 6.2.
t_surf: 1, 6.2.
temp: 5, 6.2, 10, 10.1.
tempfile: 9.
tleft: 6.2.
tmax: 2, 6.2, 7.
track: 2, 6.2, 7.

TRACK_BASIC: 7.
transform: 6.2, 10.
transform_velocity: 6.2, 10.1.
two: 5.
tx: 6.2.
type: 2, 6.2.
t1: 6.2.

v: 5, 5.1, 5.2, 5.3, 5.4, 6, 6.2, 6.3, 7, 10.1.
vc_abs: 3.3, 3.4, 5, 5.1, 5.4, 6.2.
vc_copy: 6.2, 7.
vc_cross: 5.4.
vc_decls: 5.3, 5.4.
vc_product: 5, 5.1, 5.3, 5.4.
vc_scale: 5.4, 6.3.
vc_set: 5.4, 6.2, 7.
vc_unit: 5.3, 5.4.
vc_xvt: 5, 5.3, 5.4, 6.2, 6.3, 7.
vperp: 5.3, 5.4.
vunit: 5.4.
v1: 2, 5.3, 5.4, 6.2, 6.3.

w: 5.4, 10.1.

x: 3.1, 3.2, 3.3, 3.4, 3.5, 4, 4.1, 4.4, 4.5, 4.6, 4.7,
4.8, 5.1, 5.2, 5.3, 5.4, 6.1, 6.2, 6.3, 7, 7.1, 10.
x0: 2, 5, 6, 6.2, 6.3, 7.
x1: 7.

y: 10.

zero: 3.5, 4, 4.1, 4.4, 4.5, 4.6, 4.7, 4.8, 5, 5.4, 6, 6.2,
7, 8.
zn_common: 9.
zn_mpibcast: 9.
zn_ncdecl: 9.
zn_ncread: 9.
zone: 2, 7.

⟨Functions and Subroutines 3.1, 3.2, 3.3, 3.4, 3.5, 4, 4.1, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 5, 5.1, 5.2, 5.3, 5.4, 6, 6.1, 6.2, 6.3, 7, 7.1, 8, 9, 10, 10.1⟩ Used in section 2.2.
⟨Memory allocation interface 0⟩ Used in section 9.

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

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

CHANGE FILE: (none).

GLOBAL LANGUAGE: FORTRAN.