Chapter 29.
Classes Ca thru Cz

Topic: Ignore

check_status_list

Class: Debugging

Purpose: Implements the list of return codes for status checking.

Derivation: check_status_list : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kernint/d3_chk/chk_stat.hxx

Description: Both curve and surface checkers return similar lists of return codes when a curve or surface is illegal, or NULL if it is OK. Note that in both cases the subdivision and self–intersection tests are only performed if everything else worked.

Return codes are as follows:

check_irregular	very scrunched up/twisted. (Subdivision fails.)
check_self_intersects	curve/surface self intersects.
check_bad_closure	either the curve/surface says it is closed when it isn't, or that it is not closed when it is.
check_bs3_null	no bs3 curve or surface. It does not check vertex blends. No further tests are made in this case.
check_bs3_coi_verts	either adjacent control vertices are coincident or, for degenerate surfaces, vertices which should be coincident are not.
check_bad_degeneracies	the degenerate edges on a surface make it untreatable (for example, two adjacent edges degenerate). This code is never returned for curves.
check_untreatable_singularities	a singularity on the surface is beyond the scope of singularity current processing. This code is never returned for curves.
check_non_G0	position continuous.
check_non_G1	curve/surface not G1. For surfaces this may mean that parameter lines are not tangent continuous. But it is G0.
check_non_G2	not G2, though it is G0 and G1 with smooth parameter lines, if a surface.
check_non_C1	not C1, though it is G1. It may or may not have been G2.
check_unknown	status is unknown. This does not get returned.
check_inconsistent	Data mismatch in the given edge.

Limitations: None

References: None

Data:

None

Constructor:

None

Destructor:

```
public: check_status_list::~check_status_list ();
```

Destructor, destroying the list from here on.

Methods:

```
public: check_status_list*
    check_status_list::add_error (
    check_status status     // error to add to list
    );
```

Adds an error to the front of the list. Returns a new check_status_list with the new error incorporated.

```
public: check_status_list*
    check_status_list::add_list (
    check_status_list* list // error list
    );
```

Adds a list of errors to the front of the list. Returns the new start.

```
public: logical check_status_list::contains (
    check_status wanted     // status to check for
    ) const;
```

Checks for a particular status.

```
public: check_status_list*
    check_status_list::next ();
```

Returns the next element of the error check_status_list, or NULL if there is none.

```
public: check_status check_status_list::status ();
```

Returns the status code of this element of the list.

Related Fncs:

None

COEDGE

Class: Model Topology, SAT Save and Restore

Purpose: Relates EDGEs with adjacent EDGEs and owning ENTITYs.

Derivation: COEDGE : ENTITY : ACIS_OBJECT : –

SAT Identifier: “coedge”

Filename: kern/kernel/kerndata/top/coedge.hxx

Description: The coedge is closely related to an edge. A coedge stores its relationships with adjacent edges and with superior owning entities. (In some contexts, the coedge may be viewed as the use of an edge by a face or wire.) The data structures formed by these relationships (stored as pointers) and their interpretation depends upon the nature of the owning entity.

The typical case is when the coedge’s associated edge is part of a well–formed, manifold, solid body shell and when that edge is adjacent to exactly two faces. This results in two coedges, each associated with a loop in one of the faces. (In principle the two faces could be the same, and even the loops could be the same.) All the coedges in each loop are linked into a doubly-linked circular list using the next and previous pointers. The two coedges for each edge are linked through their partner pointers.

Several deviations are possible from the typical case.

– A loop may not necessarily be closed for either a partially defined or infinite face boundary. In this case, the next and previous lists are not circular, but terminate with NULL pointers.
– A shell may not be closed and have “free” edges at its boundary. For such edges, there is only one coedge with a NULL partner pointer.
– Nonmanifold shells, where more than two faces meet in an edge, links the partner pointers for the coedges (still one for each face) in a circular list.
– Coedges on faces whose underlying geometry is a parametric surface must maintain a pointer to a pcurve, which represents the curve underlying the edge in the parametric space of the surface. Coedges on analytic surfaces are not required to have pcurves.
– Wires as owning entities are handled differently. An object may be a directed or undirected graph made up of one or more disjoint wires, each of which is a collection of connected edges. In this case, each edge has exactly one coedge. The coedges are linked in circular lists around each vertex using next and previous pointers according to which end of the coedge lies at the vertex. The next or previous pointer of a coedge on an open edge may be set to itself, indicating that there is no next or previous coedge on this branch of the wire.
– A shell may be of mixed dimensionality, containing both faces and unembedded edges. The unembedded edges are connected together in wires and belong to the shell. Where they meet faces of the shell, the vertices have multiple edge pointers, one for each face group, and one for each wire attached.

The sg_get_coedges_of_wire and sg_q_coedges_around_vertex may be useful for generating lists of coedges on other topological entities. The kernel/kerndata/geometry/geometry.hxx file contains several other geometric inquiry functions. When adding a pcurve to a coedge, sg_add_pcurve_to_coedge may be helpful.

Limitations: None

References:

KERN	EDGE, ENTITY, PCURVE
by KERN	EDGE, LOOP, WIRE, pattern_holder

Data:

```
protected COEDGE *next_ptr;
```
Pointer to provide a doubly–linked list of coedges in a loop,or circular lists at each end in a general unembedded graph.

```
protected COEDGE *partner_ptr;
```
Pointer to partner coedge, or NULL if this coedge is unembedded or attached to a free edge.

```
protected COEDGE *previous_ptr;
```
Pointer to provide a doubly–linked list of coedges in a loop,or circular lists at each end in a general unembedded graph.

```
protected EDGE *edge_ptr;
```
Pointer to the single edge on which this coedge and all its partners lie.

```
protected ENTITY *owner_ptr;
```
Pointer to the owning loop or wire. There is always a loop if the coedge is embedded in a face, or a wire if it is part of an unembedded graph. If the coedge is an unembedded one in a mixed–dimensionality shell, then this pointer is NULL.

```
protected PCURVE *geometry_ptr;
```
Pointer to the description of the edge geometry referred to the parametric space of the face in which it is embedded. This will be NULL if the edge is not embedded, or if the face is not parametrically described. It may be NULL even if the face is parametric.

```
protected REVBIT sense_data;
```
Relationship between the direction of the coedge and that of the underlying edge. When embedded in a face, the coedges must run clockwise about the (outward) face normal, that is at any point on the coedge, if the face normal is “upwards” and the coedge tangent is “forwards”, then the face lies to the “left”.

Constructor:

```
public: COEDGE::COEDGE ();
```

C++ allocation constructor requests memory for this object but does not populate it. The allocation constructor is used primarily by restore. Applications should call this constructor only with the overloaded new operator, because this reserves the memory on the heap, a requirement to support roll back and history management.

```
public: COEDGE::COEDGE (
    EDGE*,                  // EDGE
    REVBIT,                 // sense
    COEDGE*,                // previous COEDGE
    COEDGE*                 // next COEDGE
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator, because this reserves the memory on the heap, a requirement to support roll back and history management.

The arguments initialize the EDGE (and indirectly the partner EDGE), the sense_data, the previous COEDGE, and the next COEDGE. COEDGE back-pointers are also set in the two argument COEDGEs, but are only valid if all the COEDGEs are part of a conventional simple LOOP.

Destructor:

```
public: virtual void COEDGE::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual COEDGE::~COEDGE ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new COEDGE(...) then later x–>lose.)

Methods:

```
protected: virtual logical
     COEDGE::bulletin_no_change_vf (
    ENTITY const* other,        // other entity
    logical identical_comparator// comparator
    ) const;
```

Virtual compare function for api_get_modified_faces.

```
public: logical COEDGE::copy_pattern_down (
    ENTITY* target              // target
    ) const;
```

Returns whether or not patterns are copied down.

```
public: virtual void COEDGE::debug_ent (
    FILE*                   // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: EDGE* COEDGE::edge () const;
```

Returns the pointer to the single EDGE on which this COEDGE and all its partners lie.

```
public: VERTEX* COEDGE::end () const;
```

Returns the end VERTEX pointer from the associated EDGE, if any, taking into account the sense of the COEDGE.

```
public: VERTEX* COEDGE::end (
    REVBIT                  // sense
    );
```

Returns the end VERTEX pointer from the associated EDGE, if any, taking into account the sense of the COEDGE.

```
public: logical COEDGE::ends_at_singularity () const;
```

Determines if the coedge ends at a surface singularity.

```
public: virtual SPAparameter COEDGE::end_param ()
const;
```

Finds the end parameter of the COEDGE.

```
public: virtual SPAposition COEDGE::end_pos () const;
```

Finds the end position of the COEDGE.

```
public: PCURVE* COEDGE::geometry () const;
```

Returns the pointer to the description of the EDGE geometry referred to the parametric space of the FACE in which it is embedded. The pointer is NULL if the EDGE is not embedded, or if the FACE is not parametric.

```
public: void COEDGE::get_all_patterns (
    VOID_LIST& list             // list
    );
```

Returns all patterns in the list.

```
public: virtual int COEDGE::identity (
    int                     // level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier COEDGE_TYPE. If level is specified, returns COEDGE_TYPE for that level of derivation from ENTITY. The level of this class is defined as COEDGE_LEVEL.

```
public: virtual logical
    COEDGE::is_deepcopyable () const;
```

Returns TRUE if this can be deep copied.

```
public: logical COEDGE::is_pattern_child () const;
```

Returns TRUE if this is a pattern child. An entity is a “pattern child” when it is not responsible for creating new entities when the pattern is applied. Instead, some owning entity takes care of this.

```
public: LOOP* COEDGE::loop () const;
```

Returns the owner of the COEDGE if it is a LOOP; otherwise, it returns NULL.

```
public: TCOEDGE* COEDGE::make_tolerant ();
```

Make a tolerant TCOEDGE out of this COEDGE.

```
public: COEDGE* COEDGE::next () const;
```

Returns the next COEDGE in a doubly-linked list of COEDGEs.

```
public: COEDGE* COEDGE::next (
    REVBIT rev              // sense
    ) const;
```

Returns the next pointer if the sense_data is FORWARD; otherwise returns the previous pointer.

```
public: ENTITY* COEDGE::owner () const;
```

Returns the pointer to the LOOP or WIRE that owns the COEDGE. There is always a LOOP if the COEDGE is embedded in a FACE, or a WIRE if it is part of an unembedded graph. If the COEDGE is an unembedded one in a mixed-dimensionality SHELL, the function may return the SHELL.

```
public: virtual SPAinterval
    COEDGE::param_range () const;
```

Finds the parameter range of the COEDGE as an interval.

```
public: COEDGE* COEDGE::partner () const;
```

Returns the pointer to the partner COEDGE. The return will be NULL if the COEDGE is unembedded or attached to a free EDGE.

```
public: COEDGE* COEDGE::previous () const;
```

Returns the previous COEDGE in a doubly-linked list of COEDGEs.

```
public: COEDGE* COEDGE::previous (
    REVBIT rev              // sense
    ) const;
```

Returns the previous pointer if the sense_data is FORWARD; otherwise returns the next pointer.

```
public: logical COEDGE::remove_from_pattern_list ();
```

Returns TRUE if this is removed from the pattern list.

```
public: void COEDGE::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

```
if (restore_version_number < PATTERN_VERSION
    read_ptr                          APATTERN index
        if (apat_idx != (APATTERN*)(–1)))
        restore_cache();
read_ptr                              Pointer to record in save file for
                                      next COEDGE in loop or wire
read_ptr                              Pointer to record in save file for
                                      previous COEDGE in loop or wire
read_ptr                              Pointer to record in save file for
                                      partner COEDGE on edge
read_ptr                              Pointer to record in save file for
                                      EDGE on which coedge lies
if (restore_version_number < COEDGE_SENSE_VERSION)
    read_int                          Direction of coedge with respect to
                                      the edge enumeration
else
    read_logical                      either “forward” or “reversed”
read_ptr                              Pointer to record in save file for
                                      LOOP or wire to which coedge
                                      belongs
read_ptr                              Pointer to record in save file for
                                      parameter space curve PCURVE
                                      or geometry
```

```
public: REVBIT COEDGE::sense () const;
```

Returns the relationship between the direction of the COEDGE and that of the underlying EDGE. At any point on the COEDGE, if the FACE normal is upwards and the COEDGE tangent is forward, then the FACE lies to the left. This implies that the outer LOOPs are counterclockwise and the inner LOOPs are clockwise with respect to the FACE normal.

```
public: REVBIT COEDGE::sense (
    REVBIT rev              // sense
    ) const;
```

Return the sense of the COEDGE compounded with the sense argument. Useful when traversing COEDGEs in reverse direction.

```
public: void COEDGE::set_edge (
    EDGE*,                  // underlying EDGE
    logical reset_pattern   // reset or not
        = TRUE
    );
```

Sets the COEDGE to use the given underlying EDGE. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: virtual void COEDGE::set_geometry (
    PCURVE*,                // PCURVE
    logical reset_pattern   // reset or not
        = TRUE
    );
```

Sets the COEDGE's parameter-space geometry to be the given PCURVE. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
protected: virtual void COEDGE::set_geometry_ptr (
    PCURVE*                 // PCURVE
    );
```

Sets the COEDGE's parameter-space geometry to be the given PCURVE. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void COEDGE::set_loop (
    LOOP*,                  // LOOP
    logical reset_pattern   // reset or not
        = TRUE
    );
```

Sets the owning ENTITY to be a LOOP. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void COEDGE::set_next (
    COEDGE*,                // next COEDGE
    REVBIT                  // sense
        = FALSE,
    logical reset_pattern   // reset or not
        = TRUE
    );
```

Sets the COEDGE's next COEDGE pointer. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void COEDGE::set_next_no_rev (
    COEDGE*,                    // coedge
    logical reset_pattern       // reset or not
        = TRUE
);
```

Sets the COEDGE's next_no_rev COEDGE pointer. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void COEDGE::set_owner (
    ENTITY*,                // entity
    logical reset_pattern   // reset or not
        = TRUE
    );
```

Sets the COEDGE's owner. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void COEDGE::set_partner (
    COEDGE*,                // partner COEDGE
    logical reset_pattern   // reset or not
        = TRUE
    );
```

Sets the COEDGE's partner to be the given COEDGE. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void COEDGE::set_pattern (
    pattern* in_pat             // pattern
    );
```

Set the current pattern.

```
public: void COEDGE::set_previous (
    COEDGE*,                    // previous COEDGE
    REVBIT                      // sense
        = FALSE,                //
    logical reset_pattern       //
    = TRUE                      //
    );
```

Sets COEDGE's previous COEDGE pointer, taking the sense into account. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void COEDGE::set_previous_no_rev (
    COEDGE*,                    // coedge
    logical reset_pattern       // reset or not
        = TRUE
);
```

Sets COEDGE's previous_no_rev COEDGE pointer. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void COEDGE::set_sense (
    REVBIT,                     // sense
    logical reset_pattern       // reset or not
        = TRUE
    );
```

Sets the sense of the COEDGE with respect to the underlying EDGE. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void COEDGE::set_shell (
    SHELL*,                  // owning SHELL
    logical reset_pattern    // reset or not
        = TRUE
    );
```

Sets the owning entity to be a SHELL. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void COEDGE::set_wire (
    WIRE*,                   // owning WIRE
    logical reset_pattern    // reset or not
        = TRUE
    );
```

Sets the owning entity to be a WIRE. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: SHELL* COEDGE::shell () const;
```

Returns the owner of the COEDGE if it is a SHELL; otherwise, it returns NULL.

```
public: VERTEX* COEDGE::start () const;
```

Returns the start VERTEX pointer from the associated EDGE, if any, taking into account the sense of the COEDGE.

```
public: VERTEX* COEDGE::start (
    REVBIT                   // sense
    );
```

Returns the start VERTEX pointer from the associated EDGE, if any, taking into account the sense of the COEDGE.

```
public: logical
    COEDGE::starts_at_singularity () const;
```

Determines if the coedge starts at a surface singularity.

```
public: virtual SPAparameter
    COEDGE::start_param () const;
```

Finds the start parameter of the COEDGE.

```
public: virtual SPAposition COEDGE::start_pos ()
const;
```

Finds the start position of the COEDGE.

```
public: virtual const char*
    COEDGE::type_name () const;
```

Returns the string “coedge”.

```
public: WIRE* COEDGE::wire () const;
```

Returns the owner of the COEDGE if it is a WIRE; otherwise, it returns NULL.

Related Fncs:

is_COEDGE

CONE

Class: Model Geometry, SAT Save and Restore

Purpose: Defines a cone as an object in the model.

Derivation: CONE : SURFACE : ENTITY : ACIS_OBJECT : –

SAT Identifier: “cone”

Filename: kern/kernel/kerndata/geom/cone.hxx

Description: CONE is a model geometry class that contains a pointer to a (lowercase) cone, the corresponding construction geometry class. In general, a model geometry class is derived from ENTITY and is used to define a permanent model object. It provides model management functionality, in addition to the geometry definition.

CONE is one of several classes derived from SURFACE to define a specific type of surface. The cone class defines a cone by the base ellipse and the sine and cosine of the major half-angle.

Along with the usual SURFACE and ENTITY class methods, CONE has member methods to provide access to specific implementations of the geometry. For example, methods are available to set and retrieve the axes, center, and other information about a cone.

A use count allows multiple references to a CONE. The construction of a new CONE initializes the use count to 0. Methods are provided to increment and decrement the use count, and after the use count returns to 0, the entity is deleted.

Limitations: None

References: KERN cone

Data:

None

Constructor:

```
public: CONE::CONE ();
```

C++ allocation constructor requests memory for this object but does not populate it. The allocation constructor is used primarily by restore. Applications should call this constructor only with the overloaded new operator, because this reserves the memory on the heap, a requirement to support roll back and history management.

```
public: CONE::CONE (
    cone const&             // cone object
    );
```

C++ copy constructor requests memory for this object and populates it with the data from the object supplied as an argument. Applications should call this constructor only with the overloaded new operator, because this reserves the memory on the heap, a requirement to support roll back and history management.

```
public: CONE::CONE (
    SPAposition const&,      // center point
    SPAunit_vector const&,   // cone axis
    SPAvector const&,        // major axis
    double                   // major:minor ratio
        = 1,
    double                   // half angle sine
        = 0,
    double                   // half angle cosine
        = 1
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator, because this reserves the memory on the heap, a requirement to support roll back and history management.

Makes a CONE from the center point, a SPAunit_vector defining the cone-axis, a SPAvector defining the major-axis (including length), the ratio of the minor to major-axis, the sine of the cone half angle, and the cosine of the half angle:

Destructor:

```
public: virtual void CONE::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual CONE::~CONE ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new CONE(...) then later x–>lose.)

Methods:

```
protected: virtual logical
     CONE::bulletin_no_change_vf (
    ENTITY const* other,        // other entity
    logical identical_comparator// comparator
    ) const;
```

A virtual compare function for api_get_modified_faces.

```
public: double CONE::cosine_angle () const;
```

Returns the cosine of the half-angle defining the CONE.

```
public: virtual void CONE::debug_ent (
    FILE*                   // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: SPAunit_vector const& CONE::direction ()
const;
```

Returns the normal to the plane of the ellipse defining the CONE; i.e., the cone-axis.

```
public: surface const& CONE::equation () const;
```

Returns the surface equation of the CONE.

```
public: surface& CONE::equation_for_update ();
```

Returns a pointer to surface equation for update operations. Before performing a change it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: virtual int CONE::identity (
    int                     // level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier CONE_TYPE. If level is specified, returns CONE_TYPE for that level of derivation from ENTITY. The level of this class is defined as CONE_LEVEL.

```
public: virtual logical CONE::is_deepcopyable (
    ) const;
```

Returns TRUE if this can be deep copied.

```
public: SPAvector const& CONE::major_axis () const;
```

Returns the major-axis of the ellipse defining the CONE.

```
public: void CONE::operator*= (
    SPAtransf const&        // transform
    );
```

Transforms a CONE. Before performing a change it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: double CONE::radius_ratio () const;
```

Returns the ratio of the minor-axis length to the major-axis length of the ellipse defining the CONE.

```
public: void CONE::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

cone::restore_data — Cone data definition.

```
public: SPAposition const& CONE::root_point () const;
```

Returns the center of the ellipse defining the CONE.

```
public: void CONE::set_cosine_angle (
    double                  // cosine angle
    );
```

Sets the CONE's cosine angle to the given angle. Before performing a change it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void CONE::set_direction (
    SPAunit_vector const&   // direction
    );
```

Sets the CONE's direction to the given SPAunit_vector. Before performing a change it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void CONE::set_major_axis (
    SPAvector const&        // major axis
    );
```

Sets the CONE's major axis to the given SPAvector. Before performing a change it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void CONE::set_radius_ratio (
    double                  // major:minor rad. ratio
    );
```

Sets the CONE's major to minor radius ratio to the given value. Before performing a change it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void CONE::set_root_point (
    SPAposition const&      // root point
    );
```

Sets the CONE's root point to the given SPAposition. Before performing a change it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void CONE::set_sine_angle (
    double                  // sine angle
    );
```

Sets the CONE's sine angle to the given angle. Before performing a change it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: double CONE::sine_angle () const;
```

Returns the sine of the half-angle defining the CONE.

```
public: surface* CONE::trans_surface (
    SPAtransf const&            // transform
        = * (SPAtransf* ) NULL_REF,
    logical                     // reversed
        = FALSE
    ) const;
```

Returns the transformed surface equation of the CONE. If the logical is TRUE, the surface is reversed.

```
public: virtual const char* CONE::type_name () const;
```

Returns the string “cone”.

Internal Use: full_size

Related Fncs: is_CONE

cone

Class: Construction Geometry, SAT Save and Restore

Purpose: Defines the elliptical single cone.

Derivation: cone : surface : ACIS_OBJECT : –

SAT Identifier: “cone”

Filename: kern/kernel/kerngeom/surface/condef.hxx

Description: The cone class defines an elliptical single cone. It is defined by a base ellipse and the sine and cosine of the major half-angle of the cone. The normal of the base ellipse represents the axis of the cone.

As special cases, the cross-section may be circular, or the *cone* may be a cylinder.

The ellipse has the same data structure as an ellipse curve; i.e., center, normal, major axis, radius ratio.

The polarity (sign) of the trigonometric functions define the slant of the surface of the cone and the sense of the surface.

- If sine_angle has different polarity than cosine_angle, the cross-section decreases in the direction of the axis of the cone (ellipse surface normal) as shown in the Figure.
- If sine_angle has the polarity as cosine_angle, the cross-section increases in the direction of the axis of the cone (ellipse surface normal).
- If cosine_angle is positive (+), the sense of the surface is away from the axis of the cone (surface is convex).
- If cosine_angle is negative (–), the sense of the surface is toward the axis of the cone (surface is concave).
- If sine_angle is identically zero (sine_angle == 0), the cone is a cylinder.
- If cosine_angle is identically zero (cosine_angle ==0), the cone is planar.

The surface stops at the apex, if any; i.e., this surface type does not represent a double cone.

There is a set of parameter-based functions. ACIS only requires them to have defined results if the surface is parametric (i.e., method parametric returns TRUE), but components and applications may expect results for all surface types.

The *u*-parameter direction is along the generators of the cone, with zero representing the intersection of the generator with the base ellipse, and parameter increasing in the direction of the cone axis; i.e., the normal of the base ellipse, if reverse_u is FALSE, and in the opposite direction if reverse_u is TRUE. The *v*-parameter direction is along a cross-sectional ellipse clockwise around the cone axis, parameterized as for the base ellipse.

The *u*-parameter scaling factor stores the factor that when multiplied by the *u*-parameter of a point gives the 3D distance of that point along the cone surface from the base ellipse. The *u*-parameter is always 0.0 on the cone base ellipse. This enables the parameterization to be preserved if the cone is offset.

To evaluate the position corresponding to a given *uv* pair, first evaluate the base ellipse at parameter *v*, and subtract the center point to give vector *V*. Let *s* and *c* be sine_angle and cosine_angle if cosine_angle is positive, or –sine_angle and –cosine_angle if not. Let *R* be the length of the major axis of the base ellipse, negated if reverse_u is TRUE. Then:

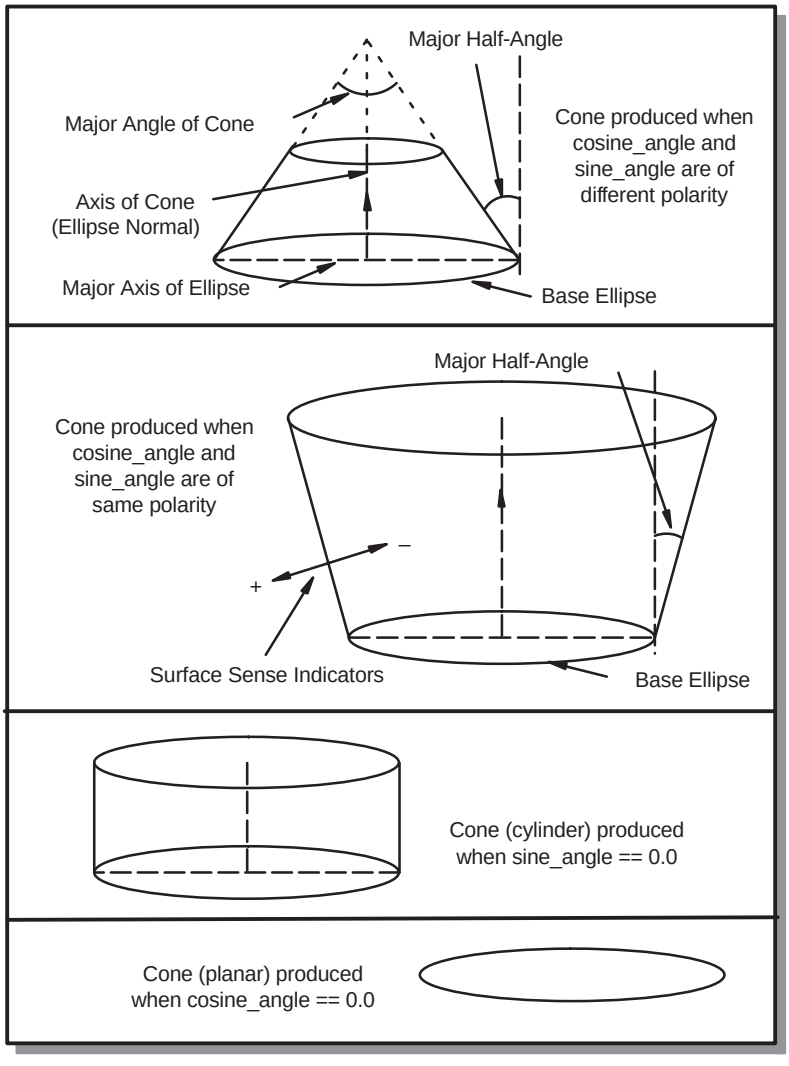
```
pos = base.center + (1 + s*u)* V + c*u*R*base.normal
```

This parameterization is left-handed for a convex cone (cosine_angle > 0) with reverse_u FALSE or for a concave cone with reverse_u TRUE, and right-handed otherwise.

When the cone is transformed, the sense of reverse_u is inverted if the transform includes a reflection. A negation requires no special action.

In summary, cones are:

– Not TRUE parametric surfaces.
– Are closed in *v* but not in *u*.
– Periodic in *v* (–pi to pi with period 2 pi) but not in *u*.
– Singular in *u* at the apex; nonsingular for all other *u* and *v* values.



Limitations: None

References: KERN ellipse
by KERN CONE

Data:

`public double cosine_angle;`
Cosine of the angle between major generator and axis.

```
public double sine_angle;
```
Sine of the angle between major generator and axis. By convention, both sine and cosine are made exactly zero to indicate that the curve is undefined.

```
public double u_param_scale;
```
Scaling of the *u* parameter lines.

```
public ellipse base;
```
Cross-section at right angles to axis.

```
public logical reverse_u;
```
Required to support transformation independent parameterization. The *u* parameter direction (along generators) is normally in the same general direction as the cone axis (normal to the base ellipse), but is reversed if the value is TRUE.

Constructor:

```
public: cone::cone ();
```

C++ allocation constructor requests memory for this object but does not populate it.

```
public: cone::cone (
    cone const&             // cone
    );
```

C++ copy constructor requests memory for this object and populates it with the data from the object supplied as an argument.

```
public: cone::cone (
    ellipse const&,         // base ellipse
    double,                 // sine of half-angle
    double,                 // cosine of half-angle
    double                  // u parameter scale
        = 0.0
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Constructs a cone with given ellipse as base, and given sine and cosine of its half-angle.

```
public: cone::cone (
    SPAposition const&,      // center
    SPAunit_vector const&,   // axis direction
    SPAvector const&,        // major radius
    double,                  // radius ratio
    double,                  // sine of half-angle
    double,                  // cosine of half-angle
    double                   // u parameter scale
        = 0.0
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Construct a cone with axis through given point and in given direction:

Destructor:

```
public: cone::~cone ();
```

C++ destructor, deleting a cone. The destructor is explicitly defined to avoid multiple copies.

Methods:

```
public: virtual int cone::accurate_derivs (
    SPApar_box const&           // parameter box
        = * (SPApar_box* ) NULL_REF
    ) const;
```

Return the number of derivatives that evaluate can find accurately and directly, rather than by finite differencing, over the given portion of the surface. If there is no limit to the number of accurate derivatives, returns the value ALL_SURFACE_DERIVATIVES.

```
public: virtual SPAbox cone::bound (
    SPAbox const&,              // box in object
                                // space
    SPAtransf const&            // transformation
        = * (SPAtransf* ) NULL_REF
    ) const;
```

Returns a box around the portion of a surface bounded by a box in object space.

```
public: virtual SPAbox cone::bound (
    SPApar_box const&               // given box in
        = * (SPApar_box* ) NULL_REF,// param space
    SPAtransf const&                // transformation
        = * (SPAtransf* ) NULL_REF
    ) const;
```

Returns a box around the portion of a surface bounded in parameter space.

```
public: logical cone::circular () const;
```

Classification routine that sets the base ratio to 1; therefore, the base is circular.

```
public: virtual logical cone::closed_u () const;
```

Reports whether the surface is closed, smoothly or not, in the *u*-parameter direction.

```
public: virtual logical cone::closed_v () const;
```

Reports whether the surface is closed, smoothly or not, in the *v*-parameter direction.

```
public: logical cone::contracting () const;
```

Classification routine that returns TRUE if the sine angle and cosine angle are of opposite signs and FALSE, otherwise.

```
public: logical cone::cylinder () const;
```

Classification routine that returns TRUE if the sine angle is essentially zero (within SPAresnor).

```
public: virtual void cone::debug (
    char const*,            // leader string
    FILE*                   // file name
        = debug_file_ptr
    ) const;
```

Prints out the details of cone to a file.

```
public: virtual surface* cone::deep_copy (
    pointer_map*  pm        // list of items within
        = NULL              // the entity that are
                            // already deep copied
    ) const;
```

Creates a copy of an item that does not share any data with the original. Allocates new storage for all member data and any pointers. Returns a pointer to the copied item.

```
public: virtual void cone::eval (
    SPApar_pos const&,      // parameter
    SPAposition&,           // position
    SPAvector*              // first derivative –
        = NULL,             // array of length 2,
                            // in order xu, xv
                            // vector * = NULL
    SPAvector*              // second derivatives –
        = NULL              // array of length 3, in
                            // order xuu, xuv, xvv
    ) const;
```

Finds the point on a parametric surface with given parameter values, and optionally the first and second derivatives as well or instead.

```
public: virtual int cone::evaluate (
    SPApar_pos const&,              // parameter
    SPAposition&,                   // pt on surface
                                    // at parameter
    SPAvector**                     // ptr array to
        = NULL,                     // vector array
    int                             // # derivatives
        = 0,
    evaluate_surface_quadrant       // the evaluation
        = evaluate_surface_unknown  // location
    ) const;
```

Calculates derivatives, of any order up to the number requested, and store them in vectors provided by the user. This function returns the number it was able to calculate; this is equal to the number requested in all but the most exceptional circumstances. A certain number are evaluated directly and (more or less) accurately; higher derivatives are automatically calculated by finite differencing. The accuracy of the finite difference derivatives decreases with the order of the derivative as the cost increases. Any of the pointers may be NULL, in which case the corresponding derivatives will not be returned. Otherwise they must point to arrays long enough for all the derivatives of that order; i.e., 2 for the first derivatives, 3 for the second, etc.

```
public: virtual SPAunit_vector cone::eval_outdir (
    SPApar_pos const&       // parameter
    ) const;
```

Finds the outward direction from the surface at a point with given parameter values.

```
public: surf_princurv cone::eval_prin_curv (
    SPApar_pos const& param // parameter
    ) const;
```

Finds the principle axes of curvature of the surface at a point with given parameter values.

```
public: virtual void cone::eval_prin_curv (
    SPApar_pos const&,      // parameter
    SPAunit_vector&,        // first axis direction
    double&,                // curvature in first
                            // direction
    SPAunit_vector&,        // second axis direction
    double&                 // curvature in second
                            // direction
    ) const;
```

Find the principal axes of curvature of the surface at a point with given parameter values, and the curvatures in those direction.

```
public: logical cone::expanding () const;
```

Returns TRUE if the cosine angle and the sine angle are of the square sign.

```
public: logical cone::flat () const;
```

Classification routine that checks whether the cosine angle is essentially zero (within SPAresnor). This is a criterion for treating a cone as completely planar, although this should never occur.

```
public: virtual SPAposition cone::get_apex () const;
```

Determines the apex of the cone if it is not a cylinder.

```
public: virtual curve* cone::get_path () const;
```

Returns the sweep path for a cone.

```
public: virtual sweep_path_type cone::get_path_type (
) const;
```

Returns the sweep path type for a cone.

```
public: virtual curve* cone::get_profile (
    double param            // parameter
    ) const;
```

Returns the v parameter line sweep information for the cone.

```
public: virtual law* cone::get_rail () const;
```

Returns the rail law for the sweep path for a cone.

```
public: logical cone::hollow () const;
```

Returns TRUE if the cosine angle is negative.

```
public: virtual logical
    cone::left_handed_uv () const;
```

Indicates whether the parameter coordinate system of the surface is right or left-handed. A convex cone has a left-handed coordinate system if *reverse-u* is FALSE, right-handed if it is TRUE. The converse is TRUE for a hollow curve.

```
public: virtual surface* cone::make_copy () const;
```

Makes a copy of this cone on the heap, and return a pointer to it.

```
public: virtual surface& cone::negate ();
```

Negates the cone.

```
public: virtual surf_normcone cone::normal_cone (
    SPApar_box const&,          // parameter bounds
    logical                     // approximation ok?
        = FALSE,
    SPAtransf const&            // transformation
        = * (SPAtransf* ) NULL_REF
    ) const;
```

Returns a cone bounding the normal direction of a curve. The cone is deemed to have its apex at the origin, and has a given axis direction and (positive) half-angle. If the logical argument is TRUE, then a quick approximation may be found. The approximate result may lie completely inside or outside the guaranteed bound (obtained with a FALSE argument), but may not cross from inside to outside. Flags in the returned object indicate whether the cone is in fact the best available, and if not whether this result is inside or outside the best cone.

```
public: virtual surface& cone::operator*= (
    SPAtransf const&        // transformation
    );
```

Transforms this cone by the given transformation.

```
public: cone cone::operator– () const;
```

Returns the inverse of the cone; i.e., with opposite normal.

```
public: virtual logical cone::operator== (
    surface const&          // surface name
    ) const;
```

Tests two surfaces for equality. It is not guaranteed to say *equal* for effectively-equal surfaces, but it is guaranteed to say *not equal* if they are not equal.

```
public: virtual SPApar_pos cone::param (
    SPAposition const&,          // position name
    SPApar_pos const&            // parameter position
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Finds the parameter values of a point on a surface, given an optional first guess.

```
public: virtual logical cone::parametric () const;
```

Determines if a cone is parametric and returns FALSE. A cone is not considered to be parameterized, as surface properties are easy to find in object space. Wherever a point evaluator has a SPApar_pos argument, this is ignored, so would normally be NULL or defaulted.

```
public: virtual double cone::param_period_u () const;
```

Returns the period of a periodic parametric surface, 0 if the surface is not periodic in the *u*-parameter or not parametric. For a cone the *u*-parameter is nonperiodic.

```
public: virtual double cone::param_period_v () const;
```

Returns the period of a periodic parametric surface, 0 if the surface is not periodic in the *v*-parameter or not parametric. For a cone the *v*-parameter always has period 2 * p2

```
public: virtual SPApar_box cone::param_range (
    SPAbox const&             // bounding box
        = * (SPAbox* ) NULL_REF
    ) const;
```

Return the principal parameter range of a surface If a box is provided the parameter range may be restricted to a portion of the surface that is guaranteed to contain all portions of the surface that lie within the region of interest. If none is provided and the parameter range in some direction is unbounded then an empty interval is returned.

```
public: virtual SPAinterval cone::param_range_u (
    SPAbox const&             // bounding box
        = * (SPAbox* ) NULL_REF
    ) const;
```

Return the principal parameter range of a surface in the *u*-parameter direction.

```
public: virtual SPAinterval cone::param_range_v (
    SPAbox const&           // bounding box
        = * (SPAbox* ) NULL_REF
    ) const;
```

Return the principal parameter range of a surface in the *v*-parameter direction.

```
public: virtual SPApar_vec cone::param_unitvec (
    SPAunit_vector const&,  // direction
    SPApar_pos const&       // parameter position
    ) const;
```

Finds the rate of change in surface parameter corresponding to a unit velocity in a given object-space direction at a given position in parameter space.

```
public: virtual logical cone::periodic_u () const;
```

Reports whether the surface is periodic in the *u*-parameter direction; i.e., it is smoothly closed, so faces can run over the seam.

```
public: virtual logical cone::periodic_v () const;
```

Reports whether the surface is periodic in the *u*-parameter direction; i.e., it is smoothly closed, so faces can run over the seam. A cone is periodic in the *v*-direction, not in the *u*.

```
public: virtual SPAunit_vector cone::point_normal (
    SPAposition const&,         // position
    SPApar_pos const&           // parameter
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Returns normal at point on cone.

```
public: virtual SPAunit_vector cone::point_outdir (
    SPAposition const&,          // position
    SPApar_pos const&            // parameter position
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Finds an outward direction from the surface at a point on the surface. This will usually be the normal, but if the point is the apex of the cone, this routine still returns an outward direction, being the (positive or negative) axis direction.

```
public: virtual void cone::point_perp (
    SPAposition const&,              // point
    SPAposition&,                    // foot
    SPAunit_vector&,                 // direction
    surf_princurv&,                  // curvature
    SPApar_pos const&                // param guess
        = * (SPApar_pos* ) NULL_REF,
    SPApar_pos&                      // actual param
        = * (SPApar_pos* ) NULL_REF,
    logical f_weak                   // weak flag
        = FALSE
    ) const;
```

Finds the point on the surface nearest to the given point. Optionally, finds the normal to and principal curvatures of the surface at that point. If the surface is parametric, returns the parameter values at the found point.

```
public: void cone::point_perp (
    SPAposition const& pos,          // point
    SPAposition& foot,               // foot
    SPApar_pos const&                // position
        param_guess                  // param guess
        = * (SPApar_pos* ) NULL_REF,
    SPApar_pos& param_actual         // actual param
        = * (SPApar_pos* ) NULL_REF,
    logical f_weak                   // weak flag
        = FALSE
    ) const;
```

Find the point on the surface nearest to the given point and optionally the normal to and principal curvatures of the surface at that point:

```
public: void cone::point_perp (
    SPAposition const& pos,          // point
    SPAposition& foot,               // foot
    SPAunit_vector& norm,            // direction
    SPApar_pos const&                // position
        param_guess                  // param guess
        = * (SPApar_pos* ) NULL_REF,
    SPApar_pos& param_actual         // actual param
        = * (SPApar_pos* ) NULL_REF,
    logical f_weak                   // weak flag
        = FALSE
    ) const;
```

Find the point on the surface nearest to the given point. Optionally, finds the normal to and principal curvatures of the surface at that point. If the surface is parametric, returns the parameter values at the found point.

```
public: surf_princurv cone::point_prin_curv (
    SPAposition const& pos,      // point
    SPApar_pos const&            // parameter position
        param_guess              // parameter guess
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Finds the principle axes of the curvature of the surface at a given point.

```
public: virtual void cone::point_prin_curv (
    SPAposition const&,          // point
    SPAunit_vector&,             // first axis dir.
    double&,                     // curvature in first
                                 // direction
    SPAunit_vector&,             // second axis dir
    double&,                     // curvature in
                                 // second direction
    SPApar_pos const&            // parameter guess
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Find the principal axes of curvature of the surface at a given point, and the curvatures in those directions

```
public: logical cone::positive () const;
```

Returns TRUE if the sine angle is negative. This function is often used in determining senses of intersections. Returns TRUE if the surface normal is in the same general direction as the cone axis; i.e., their dot product is positive, and FALSE if the normal and axis are in opposite directions. Only really meaningful if cylinder returns FALSE, but consistent with the other functions even in the cylinder case.

```
public: void cone::restore_data ();
```

Restores the data from a save file. The restore operation switches on a table defined by static instances of the restore_su_def class. This invokes a simple friend function which constructs an object of the right derived type. Then it calls the appropriate base class member function to do the actual work.

cone::restore_data	Restore the information for the base cone
read_real	Sine of cone angle
read_real	Cosine of cone angle
if (restore_version_number < CONE_SCALING_VERSION)	
// the u parameter scale is obtained from the ellipse major axis	
else	
read_real	u parameter scale
if (restore_version_number < SURFACE_VERSION)	
// the reverse u flag is set to FALSE	
else	
read_logical	*u* parameter reversed, either “forward” or “reversed”
surface::restore_data	Generic surface data

```
public: virtual void cone::save () const;
```

Saves the cone’s type and ellipse type, or cone_id, then calls cone::save_data.

```
public: void cone::save_data () const;
```

Save occurs as derived class switching goes through the normal virtual function mechanism. The save_data and restore_data function for each class can be called in circumstances when we know what type of surface we are expecting and have one in our hand to be filled in.

```
public: logical cone::shallow () const;
```

Returns TRUE if the cosine angle is small (less than 0.1 in absolute value). This can be used as a warning of possible algorithmic problems because of a large half-angle.

```
public: virtual logical cone::singular_u (
    double                  // constant u-parameter
    ) const;
```

Report whether the surface parameterization is singular at the *u*-parameter value. The only singularity recognized is where every value of the nonconstant parameter generates the same object-space point, and these can only occur at the ends of the parameter range as returned by the functions above. A cone is singular for the *u*-parameter corresponding to the apex, nonsingular for every other *u* value.

```
public: virtual logical cone::singular_v (
    double                  // constant v-parameter
    ) const;
```

Reports whether the surface parameterization is singular at the *v*-parameter value. The only singularity recognized is where every value of the nonconstant parameter generates the same object-space point, and these can only occur at the ends of the parameter range as returned by the functions above. A cone is singular for the *u*-parameter corresponding to the apex, nonsingular for every other v value.

```
public: virtual int cone::split_at_kinks_v (
    spline**& pieces,       // output pieces
    double curvature = 0.0  // minimum curvature
    ) const;
```

Divides a surface into separate pieces which are smooth (and therefore suitable for offsetting or blending). The surface is split if the curvature exceeds the minimum curvature argument. If it is closed after this, it is then split into two. The functions return the number of pieces. The split pieces are stored in pieces argument.

```
public: virtual logical cone::test_point_tol (
    SPAposition const&,             // point
    double                          // tolerance
        = 0,
    SPApar_pos const&               // param guess
        = * (SPApar_pos* ) NULL_REF,
    SPApar_pos&                     // actual param
        = * (SPApar_pos* ) NULL_REF
    ) const;
```

Test if a point lies on the surface to user-supplied precision. Defaults to system precision (SPAresabs).

```
public: virtual int cone::type () const;
```

Returns type code for surface; i.e., cone_type.

```
public: virtual char const* cone::type_name () const;
```

Returns the string “cone”.

```
public: virtual logical cone::undef () const;
```

Tests for uninitialized cone.

```
public: virtual curve* cone::u_param_line (
    double                  // constant u-parameter
    ) const;
```

Construct a parameter line on the surface. A *u*-parameter line runs in the direction of increasing *u*-parameter, at constant *v.* The parameterization in the nonconstant direction matches that of the surface, and has the range obtained by use of param_range_u or param_range_v appropriately. The new curve is constructed in free store, so it is the responsibility of the caller to ensure that it is correctly deleted.

```
public: virtual curve* cone::v_param_line (
    double                  // constant v-parameter
    ) const;
```

Construct a parameter line on the surface. A *v*-parameter line runs in the direction of increasing *v*, at constant *u*. The parameterization in the nonconstant direction matches that of the surface, and has the range obtained by use of param_range_u or param_range_v appropriately. The new curve is constructed in free store, so it is the responsibility of the caller to ensure that it is correctly deleted.

Internal Use: full_size

Related Fncs:

restore_cone

```
friend: cone operator* (
    cone const&,            // item to copy
    SPAtransf const&        // transform
    );
```

Return a cone being (a copy of) this cone transformed by the given SPAtransf.

CURVE

Class: Model Geometry, SAT Save and Restore

Purpose: Defines a generic curve as an object in the model.

Derivation: CURVE : ENTITY : ACIS_OBJECT : –

SAT Identifier: “curve”

Filename: kern/kernel/kerndata/geom/curve.hxx

Description: CURVE is a model geometry class that contains a pointer to a (lowercase) curve, the corresponding construction geometry class. In general, a model geometry class is derived from ENTITY and is used to define a permanent model object. It provides model management functionality, in addition to the geometry definition.

The CURVE class provides the basic framework for the range of curve geometries implemented at any time in the modeler. Additional classes are derived from CURVE to define specific types of curves, such as COMPCURV, ELLIPSE, INTCURVE, and STRAIGHT.

Along with the usual ENTITY class methods, CURVE has member methods to provide access to specific implementations of the geometry. For example, a curve can be transformed by a given transform operator.

A use count allows multiple references to a CURVE. The construction of a new CURVE initializes the use count to 0. Methods are provided to increment and decrement the use count, and after the use count returns to 0, the entity is deleted.

Limitations: None

References:

KERN	ENTITY
by KERN	EDGE, PCURVE, TCOEDGE, pattern_holder

Data:

None

Constructor:

```
public: CURVE::CURVE ();
```

C++ allocation constructor requests memory for this object but does not populate it. The allocation constructor is used primarily by restore. Applications should call this constructor only with the overloaded new operator, because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void CURVE::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual CURVE::~CURVE ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new CURVE(...) then later x–>lose.)

Methods:

```
public: virtual void CURVE::add ();
```

Increments the CURVE's use count. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void CURVE::add_owner (
    ENTITY*,                // owner
    logical                 // increment use
        = TRUE
    );
```

Add this owner to the list.

```
protected: virtual logical
     CURVE::bulletin_no_change_vf (
    ENTITY const* other,        // other entity
    logical identical_comparator// comparator
    ) const;
```

A virtual compare function for api_get_modified_faces.

```
public: virtual void CURVE::debug_ent (
    FILE*                   // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: virtual logical CURVE::deletable () const;
```

Indicates whether this entity is normally destroyed by lose (TRUE), or whether it is shared between multiple owners using a use count, and so gets destroyed implicitly when every owner has been lost (FALSE). The default for CURVE is FALSE.

```
public: virtual curve const&
    CURVE::equation () const;
```

Returns the curve equation for reading only, or NULL for a generic CURVE.

```
public: virtual curve& CURVE::equation_for_update ();
```

Returns a pointer to curve's equation for update operations. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: int CURVE::get_owners (
    ENTITY_LIST& list       // list
    ) const;
```

Returns the number of owners in the list.

```
public: virtual int CURVE::identity (
    int                     // level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier CURVE_TYPE. If level is specified, returns CURVE_TYPE for that level of derivation from ENTITY. The level of this class is defined as CURVE_LEVEL.

```
public: virtual logical CURVE::is_deepcopyable (
    ) const;
```

Returns TRUE if this can be deep copied.

```
public: virtual logical CURVE::
    is_use_counted () const;
```

Returns TRUE if the entity is use counted.

```
public: virtual SPAbox CURVE::make_box (
    APOINT*,                // first point on curve
    APOINT*,                // second point on curve
    SPAtransf const*,       // transform
    double                  // tolerance
        = 0.0
    ) const;
```

Determines a bounding SPAbox for the portion of the curve through two points. The curve definition must be such as to be able to determine uniquely the portion lying between any two points lying on it.

```
public: virtual void CURVE::operator*= (
    SPAtransf const&        // transform
    );
```

Transforms a CURVE. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: virtual void CURVE::remove (
    logical lose_if_zero    // flag to start lose
        = TRUE
    );
```

Decrements the CURVE's use count. If the use count becomes 0, the CURVE is deleted. Before performing a change, it checks whether the data structure is posted on the bulletin board. If not, the routine calls backup to put an entry on the bulletin board.

```
public: void CURVE::remove_owner (
    ENTITY*,                // owner
    logical                 // decrement use
        = TRUE,
    logical                 // lose if zero
        = TRUE
    );
```

Remove this owner from the list.

```
public: void CURVE::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

```
if (restore_version_number < PATTERN_VERSION
    read_ptr                        APATTERN index
        if (apat_idx != (APATTERN*)(–1)))
        restore_cache();
if ( !get_standard_save_flag() )
    read_int                        use count data
```

```
public: virtual void CURVE::set_use_count (
    int val                 // new value
    );
```

Sets the count for the number of instances of this CURVE class. This is used by the lose method. Refinements are not destructed until use_count goes to zero.

```
public: virtual curve* CURVE::trans_curve (
    SPAtransf const&            // transform
        = * (SPAtransf* ) NULL_REF,
    logical                     // reversed flag
        = FALSE
    ) const;
```

Transforms a curve equation by the given transform. If the logical is TRUE if the curve is reversed.

```
public: virtual const char*
    CURVE::type_name () const;
```

Returns the string “curve”.

```
public: virtual int CURVE::use_count () const;
```

Returns the use count of the CURVE.

Internal Use: full_size

Related Fncs: is_CURVE

curve

Class: Construction Geometry, SAT Save and Restore

Purpose: Provides methods and data common to all curve subclasses.

Derivation: curve : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kerngeom/curve/curdef.hxx

Description: The curve class is a base class from which all specific curve geometry classes (straight, ellipse, and intcurve) are derived. It defines a large variety of virtual functions for generic interaction with curves.

Consider each curve in ACIS as a parametric curve that maps an interval of the real line into a 3D vector space (object space). The mapping is continuous and one-to-one, except for closed curves.

Considered as a function of its parameter, a curve is assumed to have a continuous first derivative whose length is bounded above and below by nonzero constants. There is no hard and fast rule about the values of these bounds or about the rate of change of the length of the derivative.

The curve class defines the following virtual functions:

- Determine the parameter range of the curve.
- Determine if the curve is periodic.
- Determine the parameter value on the curve corresponding to a point on the curve.
- Determine position, direction, and curvature at a point or parameter value on the curve. Split a curve at a parameter value.
- Determine the type of the curve. Print the curve data to a file.

Limitations: None

References:

by KERN	BDY_GEOM_PCURVE, BDY_GEOM_PLANE, ELEM1D, blend_spl_sur, blend_support, bounded_curve, curve_law_data, off_surf_int_cur, offset_int_cur, offset_int_interp, pcur_int_cur, pipe_spl_sur, proj_int_cur, proj_int_interp, rot_spl_sur, stripc, subset_int_cur, sum_spl_sur, surf_surf_int, sweep_spl_sur, taper_spl_sur, tube_spl_sur, var_blend_spl_sur, wire_law_data
BASE	SPAinterval

Data:

```
protected SPAinterval subset_range;
```

Any curve may be subsetted to a given parameter range.

Constructor:

```
public: curve::curve ();
```

C++ allocation constructor requests memory for this object but does not populate it.

Destructor:

```
public: virtual curve::~curve ();
```

C++ destructor, deleting a curve. Ensures a derived class destructor is consulted when destroying a curve.

Methods:

```
public: virtual int curve::accurate_derivs (
    SPAinterval const&          // portion of curve
        = * (SPAinterval*) NULL_REF
    ) const;
```

Returns the number of derivatives found by evaluate accurately and relatively directly, rather than by finite differencing over the given portion of the curve. If there is no limit to the number of accurate derivatives, returns the value ALL_CURVE_DERIVATIVES.

```
public: virtual const double*
    curve::all_discontinuities (
    int& n_discont,          // number of
                             // discontinuities
    int order                // order
    );
```

Returns the number and parameter values of discontinuities of the curve, up to the given order (maximum three). The array is read-only.

```
public: virtual double curve::approx_error () const;
```

Returns the maximum error between the approximate evaluation of a curve and the true evaluation of the curve.

```
public: SPAbox curve::bound (
    double start,               // start parameter
    double end,                 // end parameter
    SPAtransf const& t          // transformation
        = * (SPAtransf* ) NULL_REF
    ) const;
```

Retained temporarily for historical reasons.

```
public: virtual SPAbox curve::bound (
    SPAbox const&,              // region of interest
    SPAtransf const&            // transformation
        = * (SPAtransf* ) NULL_REF
    ) const = 0;
```

Returns an object space bounding box surrounding the portion of the curve within the given box. There is no guarantee that the box will be minimal.

```
public: virtual SPAbox curve::bound (
    SPAinterval const&,          // given range
    SPAtransf const&             // transformation
        = * (SPAtransf* ) NULL_REF
    ) const = 0;
```

Returns a box surrounding the portion of the curve between two parameter values. The resulting box is not necessarily the minimal one.

```
public: virtual SPAbox curve::bound (
    SPAposition const&,          // first
    SPAposition const&,          // second position
    SPAtransf const&             // transformation
        = * (SPAtransf* ) NULL_REF
    ) const = 0;
```

Finds a box around the curve, or portion thereof, bounded by points on the curve increasing in parameter value. The points lie on the curve as supplied, not as transmitted. The resulting box is not necessarily the minimal one.

```
public: virtual void curve::change_event ();
```

Notifies the derived type that the curve has been changed, such as when the subset_range has changed, so that it can update itself. The default version of the function does nothing.

```
public: virtual check_status_list* curve::check (
    const check_fix& input           // set of flags
        = * (const check_fix*)       // for fixes
        NULL_REF,                    // allowed
    check_fix& result = * (check_fix*)
        NULL_REF,                    // fixes applied
    const check_status_list*         // checks to be
        = (const check_status_list*)// made, default
        NULL_REF                     // is none
);
```

Checks for any data errors in the curve, and corrects the errors if possible. The various arguments provide control over which checks are made, which fixes can be applied and which fixes were actually applied. The function returns a list of errors that remain in the curve on exit.

The default for allowed fixes is none (nothing fixed). If the list of checks to be made is null, then every possible check will be made. Otherwise, the function will only check for things in the list. The return value for the function will then be a subset of this list.

```
public: virtual logical curve::closed () const = 0;
```

Indicates whether a curve is closed; i.e.,joins itself (smoothly or not) at the ends of its principal parameter range. This function should always return TRUE if periodic returns TRUE.

```
public: virtual void curve::closest_point (
    SPAposition const& pos,         // position
    SPAposition& foot,              // foot position
    SPAparameter const& param_guess // input guess
        = * (SPAparameter* )NULL_REF,// value of
                                    // param
    SPAparameter& param_actual      // actual value
        = * (SPAparameter* )NULL_REF// for param
    ) const;
```

Finds the closest point on the curve (the foot) to the given point, and optionally its parameter value. If an input parameter value is supplied (as the first parameter argument), the foot found is only a local solution nearest to the supplied parameter position. Any of the return value arguments may be a NULL reference, in which case it is simply ignored.

```
public: curve* curve::copy_curve () const;
```

Makes a copy of the given curve.

```
public: virtual void curve::debug (
    char const*,            // leader
    FILE*                   // file pointer
        = debug_file_ptr
    ) const = 0;
```

Displays a description of the curve.

```
public: virtual curve* curve::deep_copy (
    pointer_map* pm          // list of items within
        = NULL               // the entity that are
                             // already deep copied
    ) const = 0;
```

Creates a copy of an item that does not share any data with the original. Allocates new storage for all member data and any pointers. Returns a pointer to the copied item.

In a *deep* copy, all the information about the copied item is self-contained in a new memory block. By comparison, a *shallow* copy stores only the first instance of the item in memory, and increments the reference count for each copy.

The pointer_map keeps a list of all pointers in the original object that have already been deep copied. For example, a deep_copy of a complex model results in self contained data, but identical sub-parts within the model are allowed to share a single set of data.

```
public: virtual const double*
    curve::discontinuities (
    int& n_discont,          // number of
                             // discontinuities
    int order                // order of curve
    ) const;
```

Returns the number and parameter values of discontinuities of the curve, of the given order (maximum three), in a read–only array.

```
public: virtual int curve::discontinuous_at (
    double t                 // parameter value
    ) const;
```

Determines whether a particular parameter value is a discontinuity.

```
public: virtual curve_boundcyl
    curve::enclosing_cylinder (
    SPAinterval const&           // parameter interval
        = * (SPAinterval*) NULL_REF
    ) const = 0;
```

Returns a cylinder that encloses the portion of the curve bounded by the given parameter interval.

```
public: virtual void curve::eval (
    double,                     // parameter
    SPAposition&,               // position
    SPAvector&                  // first derivative
        = * (SPAvector* ) NULL_REF,
    SPAvector&                  // second derivative
        = * (SPAvector* ) NULL_REF,
    logical                     // repeat
        = FALSE,
    logical                     // approx. return ok
        = FALSE
    ) const;
```

Evaluate a curve at a given parameter value, giving position and first and second derivatives (all optionally). The first logical argument, if TRUE, is a guarantee from the calling code that the most recent call to any curve or surface member function was in fact to the routine for the same curve as the current call. It allows an implementation to cache useful intermediate results to speed up repeated calculations, but must be used with extreme care.

The second logical argument may be set to TRUE if an approximate return value is acceptable. Here *approximate* is not well-defined, but may be assumed to be sufficient for visual inspection of the curve.

```
public: virtual int curve::evaluate (
    double,                             // parameter
                                        // value
    SPAposition&,                       // point on curve
                                        // at given
                                        // parameter
    SPAvector**                         // pointers to
        = NULL,                         // derivative
                                        // vectors, size
                                        // nd. Any ptrs
                                        // may be NULL,
                                        // ==> the
                                        // corresponding
                                        // derivative
                                        // won't be
                                        // returned.
    int                                 // # derivatives
        = 0,                            // required (nd)
    evaluate_curve_side                 // the evaluation
        = evaluate_curve_unknown        // location –
                                        // above or below
    ) const;
```

Calculates derivatives, of any order up to the number requested, and store them in vectors provided by the user. This function returns the number it was able to calculate; this is equal to the number requested in all but the most exceptional circumstances. A certain number are evaluated directly and (more or less) accurately; higher derivatives are automatically calculated by finite differencing. The accuracy of these decreases with the order of the derivative, as the cost increases.

```
public: virtual int curve::evaluate_iter (
    double,                          // parameter
    curve_evaldata*,                 // supplying
                                     // initial
                                     // values, and
                                     // set to reflect
                                     // the results of
                                     // the evaluation
    SPAposition&,                    // point on curve
                                     // at given
                                     // parameter
    SPAvector**                      // pointers to
        = NULL,                      // derivative
                                     // vectors, size
                                     // nd. Any ptrs
                                     // may be NULL,
                                     // ==> the
                                     // corresponding
                                     // derivative
                                     // won't be
                                     // returned.
    int                              // # derivatives
        = 0,                         // required (nd)
    evaluate_curve_side              // the evaluation
        = evaluate_curve_unknown     // location –
                                     // above or below
    ) const;
```

The evaluate_iter function is just like evaluate, but is supplied with a data object which contains results from a previous close evaluation, for use as initial values for any iteration involved. The default implementation simply ignores this value and calls evaluate.

```
public: virtual SPAvector curve::eval_curvature (
    double,                  // parameter
    logical                  // repeat
        = FALSE,
    logical                  // approximate return ok
        = FALSE
    ) const;
```

Finds the curvature at the given parameter value on the curve. Refer to eval for description of logical arguments.

```
public: virtual SPAvector curve::eval_deriv (
    double,                 // parameter
    logical                 // repeat
        = FALSE,
    logical                 // approximate return ok
        = FALSE
    ) const;
```

Finds the derivative (direction and magnitude) at the given parameter value on the curve. Refer to eval for description of logical arguments.

```
public: virtual double curve::eval_deriv_len (
    double,                 // parameter
    logical                 // repeat
        = FALSE,
    logical                 // approximate return ok
        = FALSE
    ) const;
```

Finds the magnitude of the derivative at the given parameter value on the curve. Refer to eval for description of logical arguments.

```
public: virtual SPAunit_vector curve::eval_direction
(
    double,                 // parameter
    logical                 // repeat
        = FALSE,
    logical                 // approximate return ok
        = FALSE
    ) const;
```

Finds the tangent direction at the given parameter value on the curve. This function is not virtual; it always just takes the direction of the derivative. Refer to eval for description of logical arguments.

```
public: virtual SPAposition curve::eval_position (
    double,                 // parameter value
    logical                 // repeat
        = FALSE,
    logical                 // approximate return ok
        = FALSE
    ) const;
```

Finds the point on a curve corresponding to a given parameter value. Refer to eval for description of logical arguments.

```
public: virtual curve_extremum* curve::find_extrema (
    SPAunit_vector const&   // direction
    ) const = 0;
```

Finds the extrema of a curve in a given direction.

```
protected: virtual int
    curve::finite_difference_derivatives (
    double,                 // parameter
    SPAposition&,           // pt of curve at given
                            // parameter
    SPAvector**,            // ptrs to vectors, size
                            // nd. Any ptrs may be
                            // NULL ==> corresponding
                            // derivative won't be
                            // returned.
    int,                    // number of derivatives
                            // required
    int,                    // number of derivatives
                            // already evaluated and
                            // directly evaluable in
                            // neighborhood of param
                            // (nfound)
    double,                 // finite differencing
                            // step to use
    evaluate_curve_side     // evaluation location –
                            // above, below or don't
                            // care
    ) const;
```

Evaluate higher derivatives than are available accurately in evaluate by finite differencing.

```
public: virtual const discontinuity_info&
    curve::get_disc_info (
    ) const;
```

Returns read–only access to a discontinuity_info object, if there is one. The default version of the function returns NULL.

```
public: virtual int curve::high_curvature (
    double k,               // maximum curvature
    SPAinterval*& spans     // interval list
    );
```

Finds regions of high curvature of the curve. This method stores an array of intervals in spans argument over which the curvature exceeds k. It returns the number of intervals stored.

```
public: virtual law* curve::law_form ();
```

Returns a law pointer that is the same the curve or else a NULL pointer.

```
public: virtual double curve::length (
    double,                 // first parameter
    double                  // second parameter
    ) const = 0;
```

Arc length. Returns the algebraic distance along the curve between the given parameters. The sign is positive if the parameter values are given in increasing order and negative if they are in decreasing order.

```
public: virtual double curve::length_param (
    double,                 // datum parameter
    double                  // arc length
    ) const = 0;
```

The inverse of the length function. Returns the parameter value of the point on the curve at the given algebraic arc length from that defined by the datum parameter. The result is not defined for a bounded nonperiodic curve if the datum parameter is outside the parameter range, or if the length is outside the range bounded by the values for the ends of the parameter range.

```
public: void curve::limit (
    SPAinterval const&      // given range
    );
```

Subset this curve in place, ensuring canonical results if the underlying curve is bounded or periodic.

```
public: virtual curve* curve::make_copy () const = 0;
```

Make a copy of the given curve. This is a pure virtual function to ensure that each derived class defines its own.

```
public: virtual curve_evaldata*
    curve::make_evaldata () const;
```

Construct a data object to retain evaluation information across calls to evaluate_iter(). This is to allow subsidiary calls within an iterative evaluator to start iteration much closer to the required result than is possible just using the curve information itself. The default returns NULL, indicating that no special information is required or usable.

```
public: virtual curve& curve::negate () = 0;
```

Reverse the sense of the curve.

```
public: logical curve::operator!= (
    curve const& rhs        // curve name
    ) const;
```

Tests two curve for equality.

```
public: virtual curve& curve::operator*= (
    SPAtransf const&        // transformation
    ) = 0;
```

Transforms a curve.

```
public: virtual logical curve::operator== (
    curve const&            // curve
    ) const;
```

Tests two curves for equality. This is not guaranteed to state equal for effectively-equal curves, but is guaranteed to state not equal if the curves are not equal. The result can be used for optimization. The default is not equal.

```
public: virtual double curve::param (
    SPAposition const&,             // point
    SPAparameter const&             // param guess
        = * (SPAparameter* ) NULL_REF
    ) const = 0;
```

Finds the parameter value of a given point on the curve.

```
public: virtual double
    curve::param_period () const = 0;
```

Returns the period of a periodic curve, 0 if the curve is not periodic.

```
public: virtual SPAinterval curve::param_range (
    SPAbox const&           // region of interest
        = * (SPAbox* ) NULL_REF
    ) const = 0;
```

Returns the principal parameter range of a curve. The definition of a periodic curve is for all parameter values, by reducing the given parameter modulo the period into this principal range. For an open unbounded curve, the principal range is conventionally the empty interval. For bounded open or nonperiodic curves the curve evaluation functions are defined only for parameter values in this range. If a region of interest is provided, a valid range is always returned, even for an unbounded curve, representing a portion of the curve that is guaranteed to include all segments that lie within the region of interest.

```
public: virtual pcurve* curve::pcur (
    int                     // integer denoting the
                            // parameter curve
    ) const;
```

Returns *n*th parametric curve. If this curve is defined with respect to n or more surfaces and the *n*th is parametric, NULL if not. If the argument is negative, then returns the pcurve corresponding to the absolute value of the argument, but reversed in sense.

```
public: virtual logical curve::pcur_present (
    int                     // integer denoting the
                            // parameter curve
    ) const;
```

Determines if the *n*th parameter-space curve is defined.

```
public: virtual logical curve::periodic () const = 0;
```

Indicates if a curve is periodic; i.e., joins itself smoothly at the ends of its principal parameter range, so that edges span the seam.

```
public: virtual SPAvector curve::point_curvature (
    SPAposition const&,             // point
    SPAparameter const&             // param guess
        = * (SPAparameter* ) NULL_REF
    ) const = 0;
```

Finds the curvature of the curve at given point.

```
public: virtual SPAunit_vector curve::point_direction
(
    SPAposition const&,             // point
    SPAparameter const&             // param guess
        = * (SPAparameter* ) NULL_REF
    ) const = 0;
```

Finds tangent direction of curve at given point.

```
public: virtual void curve::point_perp (
    SPAposition const&,             // point
    SPAposition&,                   // foot
    SPAunit_vector&,                // vector
    SPAvector&,                     // curvature
    SPAparameter const&             // param guess
        = * (SPAparameter* ) NULL_REF,
    SPAparameter&                   // actual param
        = * (SPAparameter* ) NULL_REF,
    logical f_weak                  // weak flag
        = FALSE
    ) const = 0;
```

Finds the foot of the perpendicular from the given point to the curve and tangent to the curve at that point, and its parameter value. If an input parameter value is supplied (as the first parameter argument), the perpendicular found is the one nearest to the supplied parameter position; otherwise, it is the one that the curve is nearest to the given point. Any of the return value arguments may be a NULL reference, in which case it is simply ignored.

```
public: void curve::point_perp (
    SPAposition const& pos,         // point
    SPAposition& foot,              // foot
    SPAparameter const&             // parameter
        param_guess                 // param guess
        = * (SPAparameter* ) NULL_REF,
    SPAparameter& param_actual      // actual param
        = * (SPAparameter* ) NULL_REF,
    logical f_weak                  // weak flag
        = FALSE
    ) const;
```

Finds the foot of the perpendicular from the given point to the curve and tangent to the curve at that point, and its parameter value. If an input parameter value is supplied (as the first parameter argument), the perpendicular found is the one nearest to the supplied parameter position; otherwise, it is the one that the curve is nearest to the given point. Any of the return value arguments may be a NULL reference, in which case it is simply ignored.

```
public: void curve::point_perp (
    SPAposition const& pos,         // point
    SPAposition& foot,              // foot
    SPAunit_vector& norm,           // dir. vector
    SPAparameter const&             // parameter
        param_guess                 // param guess
        = * (SPAparameter* ) NULL_REF,
    SPAparameter& param_actual      // actual param
        = * (SPAparameter* ) NULL_REF,
    logical f_weak                  // weak flag
        = FALSE
    ) const;
```

Finds the foot of the perpendicular from the given point to the curve and tangent to the curve at that point, and its parameter value. If an input parameter value is supplied (as the first parameter argument), the perpendicular found is the one nearest to the supplied parameter position; otherwise, it is the one that the curve is nearest to the given point. Any of the return value arguments may be a NULL reference, in which case it is simply ignored.

```
public: void curve::restore_data ();
```

Restore the data for a curve of known type. The base class version only restores the subset_range member. For convenience it can be called by derived class versions.

The restore operation switches on a table defined by static instances. This invokes a simple friend function which constructs an object of the right derived type. Then it calls the appropriate base class member function to do the actual work. The restore_data function for each class can be called in circumstances when it is known what type of surface is to be expected and a surface of that type is on hand to be filled in.

if (restore_version_number >= BNDCUR_VERSION)
    read_interval Interval for the subset range.

```
public: virtual void curve::save () const = 0;
```

Function to save a curve of unknown type, together with its type code for later retrieval.

```
public: void curve::save_curve () const;
```

Function to be called to save a curve of unknown type, or NULL. Just checks for NULL, then calls save.

```
public: void curve::save_data () const;
```

Function to save a curve of a known type, where the context determines the curve type, so no type code is necessary. The base class version just saves the subset_range member – for convenience it can be called by derived class versions.

```
public: virtual curve* curve::split (
    double,                          // parameter
    SPAposition const&               // position
        = * (SPAposition* ) NULL_REF
    );
```

Splits curve at given parameter value if possible (if the curve is defined or approximated by one or more splines). Constructs a new curve coincident with *nd* with the same parameterization as the given parameter value and modify the given curve to represent only the remainder of the curve. If the curve cannot be split, returns NULL. The default is to make the curve nonsplittable.

```
public: curve* curve::subset (
    SPAinterval const&      // range
    ) const;
```

Constructs a subsetted copy.

```
public: logical curve::subsetted () const;
```

Indicates whether the curve has a significant subset range.

```
public: virtual curve_tancone curve::tangent_cone (
    SPAinterval const&,          // given range
    logical                      // approximation ok
        = FALSE,
    SPAtransf const&             // transformation
        = * (SPAtransf* ) NULL_REF
    ) const = 0;
```

Returns a cone bounding the tangent direction of a curve. The curve is deemed to have its apex at the origin and have a given axis direction and (positive) half angle. If the logical argument is TRUE, then a quick approximation may be found. The approximate result may lie wholly within or wholly outside the guaranteed bound (obtained with a FALSE argument), but may not cross from inside to outside. Flags in the returned object indicate whether the cone is in fact the best available and if not whether this result is inside or outside the best cone.

```
public: logical curve::test_point (
    SPAposition const& pos,          // point
    SPAparameter const&              // parameter
        param_guess                  // param guess
        = * (SPAparameter* ) NULL_REF,
    SPAparameter& param_actual       // actual param
        = * (SPAparameter* ) NULL_REF
    ) const;
```

Tests point-on-curve, optionally returning the exact parameter value if the point is on the curve.

```
public: virtual logical curve::test_point_tol (
    SPAposition const&,              // position
    double                           // parameter
        = 0,
    SPAparameter const&              // first param
        = * (SPAparameter* ) NULL_REF,
    SPAparameter&                    // second param
        = * (SPAparameter* ) NULL_REF
    ) const = 0;
```

Test point-on-curve, optionally returning the exact parameter value if the point is on the curve

```
public: virtual int curve::type () const = 0;
```

Returns an identifier that specifies the curve type.

```
public: virtual char const*
    curve::type_name () const = 0;
```

Returns the string “curve”.

```
public: virtual logical curve::undef () const;
```

Indicates whether the curve is properly defined. A NULL or generic curve is always undefined—other curves depend on their contents.

```
public: logical curve::undefined () const;
```

Indicates whether the curve is properly defined. A NULL or generic curve is always undefined—other curves depend on their contents.

```
public: void curve::unlimit ();
```

Removes the parameter limits from this curve.

```
public: curve* curve::unsubset () const;
```

Constructs a copy of the unbounded curve underlying this one.

Internal Use: full_size

Related Fncs:

---

restore_curve

curve_bounds

Class: Construction Geometry, Geometric Analysis

Purpose: Specifies the curve bounds of interest.

Derivation: curve_bounds : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kernint/intcusf/cusfint.hxx

Description: This class describes the start and end of a curve, with particular reference to a curve-surface intersection.

Limitations: None

References: BASE SPAparameter, SPAposition

Data:

`public SPAparameter end_param;`
The end parameter on the curve.

`public SPAparameter start_param;`
The start parameter on the curve.

`public point_surf_rel end_rel;`
The end relation. TRUE means that the end point is on the surface; FALSE means that the end point is off the surface.

```
public point_surf_rel start_rel;
```
The start relation. TRUE means that the start point is on the surface; FALSE means that the start point is off the surface.

```
public SPAposition end_point;
```
The end position, which can be NULL.

```
public SPAposition start_point;
```
The start position, which can be NULL.

Constructor:

```
public: curve_bounds::curve_bounds (
    curve const&,                   // curve
    SPAposition const&              // start position
        = * (SPAposition*) NULL_REF,// on curve
    SPAposition const&              // end position
        = * (SPAposition*) NULL_REF
// on curve
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates curve bounds given the curve and the start and end positions on the curve.

```
public: curve_bounds::curve_bounds (
    logical,                // start relation
    SPAposition const&,     // start position
    double,                 // start param. on curve
    logical,                // end relation
    SPAposition const&,     // end position
    double                  // end parameter on curve
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates curve bounds given the start and end relations, positions, and parameters. If the start or end relation is TRUE, it means that the start or end point, respectively, is on the surface. If the start position or end position is NULL, that side of the cure is unbounded for the intersection.

```
public: curve_bounds::curve_bounds (
    SPAposition const&,      // start position
    double,                  // start parameter
    SPAposition const&,      // end position
    double                   // end parameter
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates curve bounds given the start and end positions and parameters. If the start position or end position is NULL, that side of the cure is unbounded for the intersection.

Destructor:

None

Methods:

```
public: void curve_bounds::debug (
    FILE*                    // file name
        = debug_file_ptr
    ) const;
```

Writes debug information about curve_bounds to the printer or to the specified file.

Related Fncs:

delete_curve_surf_ints

curve_curve_int

Class: Intersectors

Purpose: Represents the intersection of a curve with another curve and returns the intersections as a list.

Derivation: curve_curve_int : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kernint/intcucu/intcucu.hxx

Description: This class represents the intersection of a curve (curve1) with another curve (curve2). The intersections are returned as a list. The list can be walked using the “next” member of the class. This class saves the intersection point as int_point, and saves its corresponding parameter on curve1 as param1, and on curve2 as param2. The relation between curve1 and curve2 at this intersection point are saved as low_rel, which is the relation on the lower parameter side of the intersection and high_rel, which is higher parameter side of the intersection with respect to curve1.

Limitations: None

References: BASE SPApar_pos, SPAposition

Data:

`public curve_curve_int *next;`
Pointer to allow linked lists of curve_curve_ints.

`public curve_curve_rel high_rel;`
Relation of curves on the higher-parameter side of curve1.

`public curve_curve_rel low_rel;`
Relation of curves on the lower-parameter side of curve1.

`public curve_curve_userdata *userdata;`
Pointer to an arbitrary object to store user data. If non–NULL, it is deleted when this object is deleted. It is the responsibility of the user’s class derived from this to ensure that the destructor does what is necessary.

`public double param1;`
Intersection parameter on curve1.

`public double param2;`
Intersection parameter on curve2.

`public logical uv_set;`
TRUE if the surface parameters have been set. FALSE by default.

`public SPApar_pos uv;`
Surface parameters if the curves are known to lie on a surface.

`public SPAposition int_point;`
Intersection point.

Constructor:

```
public: curve_curve_int::curve_curve_int (
    curve_curve_int*,        // ”next” list pointer
    SPAposition const&,      // intersection point
    double,                  // first curve parameter
    double,                  // second curve parameter
    SPApar_pos&              // actual param
    = * (SPApar_pos* ) NULL_REF
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Destructor:

```
public: curve_curve_int::~curve_curve_int ();
```

C++ destructor, deleting a curve_curve_int.

Methods:

```
public: void curve_curve_int::debug (
    FILE*                    // file name
        = debug_file_ptr
    );
```

Writes debug information about curve_curve_int to standard output or to the specified file.

Related Fncs:

None

curve_interp

Class: Construction Geometry

Purpose: Contains arrays to be interpolated and the information necessary for the interpolation.

Derivation: curve_interp : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/spline/bs3_crv/fit.hxx

Description: The main way of constructing a new intcurve (as opposed to a copy of an existing one), and the only way to make one with an int_cur of a derived type, is using an object of the curve_interp class, or a class derived from it. This class contains arrays of points to be interpolated, and the information necessary for the interpolation. Each derived class must supply two virtual functions, true_point specifies how the interpolation information is used, and make_int_cur constructs an int_cur of the appropriate derived class, which is usually defined at the same time as the derived curve_interp class. This class can also take parameter values at the points that are interpolated. These values can be given as an array of doubles in the data member param.

The first task in constructing an intcurve is to generate a sequence of points along the true curve in some context-dependent way, subject to certain conditions discussed later. For each point there must be a position in object space, a curve direction, and possibly one or more positions in the parameter space of any surfaces involved. All of this information goes into the base class-any further information needed by the virtual functions supplied by the application can be added to the derived class. The (derived) curve_interp object is then passed to the intcurve constructor, together with an optional region of interest, resulting in the interpolating curve of the correct type.

Restrictions on Input Point Lists

The current curve fitting algorithm takes points pair wise from the input point lists, constructs a Hermite interpolant, cubic in object space, and quadratic in parameter space, and tests its mid-point for a valid fit. On failure, it subdivides the interpolant into two at the true mid point and tries again for each half. At each stage it tests the box containing the span end points and the points of nearest approach of its two end tangents against a supplied region of interest. If there is no overlap, it assumes that the true curve between those end points lies entirely outside the region of interest, and does not attempt to fit the spline any further. The application must ensure that the initial points supplied in the curve_interp object are near enough so that all these operations and assumptions are valid. Although there are efficiency/reliability trade-offs to be considered, it is unlikely that a precise analysis of boxing would be justified.

Hermite interpolation

This involves determining the points of nearest approach of the end tangents in object space, and determining the distance (measured algebraically along the tangent direction) from the start point to point of nearest approach on the start tangent, and from the point of nearest approach on the end tangent to the end point. For the interpolation to be successful, both distances must be positive, and their ratio should not be too large or small—for example, a factor of 10. Similarly, in each parameter space, the intersection point of the end tangents should lie on the correct side of the end points to ensure that the Bezier quadratic defined by these three control points has the correct initial and final tangent directions.

True point finding

It is essential that the mid-point of the initial Hermite fit is close enough to the true curve for the application-supplied true_point function to find a valid point. It will normally need to be closer to the required curve than to any other curve satisfying the interpolation conditions, and there should not be any high-frequency undulations in the surfaces between the initial approximation and the true curve. This condition is impossible to specify or test precisely, and so is subject to a series of heuristics. The main one currently used is to reduce the allowable angle between the end tangent directions of a span to roughly 30 degrees. This, together with the hard point order requirements for the initial Hermite interpolation, has proved effective for surface-surface intersections and silhouette lines in ACIS so far.

Boxing

This requires that the whole of the span of the true curve between the given end points lies within the box containing those two end points and the points of nearest approach between the end tangents. As a special case, it is permissible for the end points of a nonperiodic curve to have zero-length direction vectors. This is for the case that the curve direction is ill-defined by the normal procedure, and it is inconvenient to go to higher order. With an intersection curve, for example, the surfaces may be tangent, so that the curve direction can only be determined by a second-order method. In this case, the interpolation process continually adjusts the end direction of the curve to give zero curvature there, then calling true_point with that direction to adjust the direction to be valid if it only has one degree of freedom. This appears to be effective and cheap.

Limitations: None

References: by KERN point_data, point_obj_data, point_surf_data
BASE SPAinterval

Data:

```
public double const *param;
```
The parameter values at the given points. This value is NULL if the fitting process chooses its own parameter values.

```
public double fitol;
```
The tolerance allowed on fitted splines.

```
public int nobj;
```
The number of object-space curves being interpolated.

```
public int npts;
```
The number of points to be interpolated. If the curve is periodic, this number is negative; i.e., the last point and the direction are the same as the first point and direction. All arrays, both object-space and parameter-space, are of this length.

```
public int nsurf;
```
The number of surface-related records.

```
public int nvalid;
```
The number of intervals in valid.

```
public interp_obj_data *objdata;
```
The pointer to an array of object-space curve data records.

```
public interp_surf_data *sfdata;
```
The pointer to an array of objects describing surface-related information.

```
public SPAinterval *valid;
```
The array of parameter intervals within which the fit is in tolerance. Portions outside these intervals are entirely outside the region of interest, and so they may be well outside the tolerance. This is always kept in numerical order and disjoint.

Constructor:

```
public: curve_interp::curve_interp (
    int,                    // # array entries
    double,                 // fit tolerance
    int                     // # obj.–sp. curves
        = 0,
    int                     // number of surfaces
        = 0
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

```
public: curve_interp::curve_interp (
    int,                        // # array entries
    SPAposition const*,         // array of points
    SPAvector const*,           // array of tangents
    double,                     // fit tolerance
    int                         // # sur.–rel. objs
        = 0
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

Creates a curve interpolation by accepting a list of positions and tangent directions for the curve that is to be interpolated or fit, depending upon whether the tolerance is 0.

Destructor:

```
public: virtual curve_interp::~curve_interp ();
```

C++ destructor, deleting a curve_interp.

Methods:

```
public: void curve_interp::fit (
    SPAbox const&               // given precision region
    );
```

Fits the object-space splines and possible parameter-space splines to the specified initial lists of points.

```
public: virtual int_cur*
    curve_interp::make_int_cur () = 0;
```

Constructs an int_cur to represent the fitted curve. This is used by the intcurve constructor for fitting curves to the point lists. This method must be provided for every class derived from this one, and it constructs the appropriate object to represent that type of curve, normally derived from the base class, int_cur.

```
public: bs3_curve curve_interp::obj_bs (
    int                         // object–space curve
        = 0
    );
```

Extracts the *n*th object-space curve after fitting. The result becomes the property of the caller, and subsequent calls return NULL.

```
public: double curve_interp::obj_fitol (
    int                     // tolerance
        = 0
    );
```

Extract the actual fit tolerance achieved for the given object–space curve.

```
public: bs2_curve curve_interp::par_bs (
    int                     // parameter-space curve
    );
```

Extracts the *n*th parameter-space curve after fitting. The result becomes the property of the caller, and subsequent calls return NULL.

```
public: virtual void curve_interp::true_point (
    double,                 // tolerance
    point_data&             // point data
    ) const = 0;
```

Finds the true-point in 3D for a given parameter value. The input position, direction, and parameter values are approximate; the exact values are provided as output.

```
public: SPAinterval const* curve_interp::valid_range
(
    int                     // valid interval
    );
```

Extracts the *n*th valid interval from the object, where *n* ranges from 0 to nvalid – 1. This method returns NULL if *n* is outside the range.

Related Fncs:

None

curve_irregularities

Class: Construction Geometry, Geometric Analysis

Purpose: Implements a linked list of parameter values at which a curve has a C1 (tangent direction) or G1 (tangent magnitude) discontinuity.

Derivation: curve_irregularities : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/spline/sg_bs3c/bs3ccont.hxx

Description: This class implements a linked list of parameter values at which a curve has a C1 (tangent direction) or G1 (tangent magnitude) discontinuity.

Limitations: None

References: None

Data:

`public curve_irregularities *next;`
The next in curve irregularity in the list and is NULL terminated.

`public double par_val;`
The parameter value at which the discontinuity exists.

`public irr_type ir;`
Type of irregularity: either C1 (tangent direction) or G1 (tangent magnitude) discontinuity.

Constructor:

None

Destructor:

None

Methods:

None

Related Fncs:

None

curve_law_data

Class: Laws, Geometric Analysis, SAT Save and Restore

Purpose: Creates a wrapper to an ACIS curve class.

Derivation: curve_law_data : base_curve_law_data : path_law_data : law_data : ACIS_OBJECT : –

SAT Identifier: “EDGE#”

Filename: kern/kernel/kernutil/law/law.hxx

Description: This is a law data class that holds a pointer to a curve.

Limitations: None

References:

KERN	curve
by KERN	law_int_cur
BASE	SPAposition, SPAvector

Data:

```
protected curve *acis_curve;
```
This holds a pointer to the underlying ACIS curve.

```
protected double *tvalue;
```
This holds the parameter values.

```
protected int *which_cached;
```
This holds the time tags.

```
protected int derivative_level;
```
This holds how many derivatives are cached.

```
protected int point_level;
```
This holds the size of the tvalue array.

```
protected SPAposition *cached_f;
```
This holds the positions of the curve used in evaluation.

```
protected SPAvector *cached_ddf;
```
This holds vectors representing the second derivative of the curve at the positions given in cached_f.

```
protected SPAvector *cached_df;
```
This holds vectors representing the first derivative of the curve at the positions given in cached_f.

Constructor:

```
public: curve_law_data::curve_law_data (
    curve const& in_acis_curve, // underlying ACIS
                                // curve
    double in_start             // start parameter
        = 0,
    double in_end               // end parameter
        = 0
    );
```

C++ constructor, creating a curve_law_data which is a wrapper for the ACIS curve. Because the ACIS curve does not store the starting and ending parameter positions, these must be provided.

Destructor:

```
public: curve_law_data::~curve_law_data ();
```

Applications are required to call this destructor for their law data types.

Methods:

```
public: double curve_law_data::curvature (
    double para             // parameter to evaluate
    );
```

Returns the curvature of the curve at the given parameter position.

```
public: curve* curve_law_data::curve_data ();
```

Returns a pointer to the reference curved stored as part of the curve_law_data.

```
public: virtual law_data* curve_law_data::deep_copy (
    base_pointer_map* pm    // list of items within
        = NULL              // the entity that are
                            // already deep copied
    ) const;
```

Creates a copy of an item that does not share any data with the original. Allocates new storage for all member data and any pointers. Returns a pointer to the copied item.

```
public: SPAvector curve_law_data::eval (
    double para,            // parameter to evaluate
    int deriv,              // which derivative
                            // to take
    int side                // left or right –
        = 0                 // sided evaluation
    );
```

Returns the position or one of its derivatives of the underlying curve at the give parameter position. The position is returned as a vector.

```
public: law*curve_law_data::law_form ();
```

Returns a pointer to the law class used as part of the curve_law_data.

```
public: double curve_law_data::length (
    double start,           // start parameter
    double end              // end parameter
    );
```

Arc length. Returns the algebraic distance along the curve between the given parameters. The sign is positive if the parameter values are given in increasing order and negative if they are in decreasing order.

```
public: double curve_law_data::length_param (
    double base,            // datum parameter
    double length           // arc length
    );
```

Returns the parameter value of the point on the curve at the given algebraic arc length from that defined by the datum parameter. This method is the inverse of the length method. The result is not defined for a bounded nonperiodic curve if the datum parameter is outside the parameter range, or if the length is outside the range bounded by the values for the ends of the parameter range.

```
public: double curve_law_data::point_perp (
    SPAposition in_point    // point
    );
```

Finds the point on the curve nearest to the given point.

```
public: double curve_law_data::point_perp (
    SPAposition in_point,   // point
    double in_t             // parameter
    );
```

Finds the point on the curve nearest to the given point.

```
public: virtual void curve_law_data::save ();
```

Saves the curve law data and the curve.

```
public: law_data* curve_law_data::set_domain (
    SPAinterval* new_domain // new input domain
    );
```

Establishes the domain of the law. Permits the law to be altered for the its input array size.

```
public: void curve_law_data::set_levels (
    int in_point_level      // number of positions
        = 4,
    int in_derivative_level // number of derivatives
        = 2
    );
```

This establishes the number of parameter values to store in tvalue, which in turn establishes the positions for cached_f.

```
public: int curve_law_data::singularities (
    double** where,         // total discontinuities
    int** type,             // type of discontinuity
    double start,           // start
    double end              // end
    );
```

Returns the number, type (first, second, or third order) and parameter values of the discontinuities on the underlying curve, if any.

```
public: char const* curve_law_data::symbol (
    law_symbol_type type    // type of law symbol
    );
```

Returns the string that represents this law class's symbol. The symbol is used for parsing the law and for saving and restoring law-based geometry. For a law to be saved and restored, it must have or inherit this method.

The default law symbol for this class is EDGE.

Related Fncs:

restore_law, restore_law_data, save_law

curve_surf_int

Class: Intersectors, Geometric Analysis

Purpose: Represents the intersection of a curve with a surface and returns the intersections as a list.

Derivation: curve_surf_int : ACIS_OBJECT : –

SAT Identifier: None

Filename: kern/kernel/kernint/intcusf/cusfint.hxx

Description: This class represents the intersection of a curve with a surface. This class returns all intersections inside the bounding box in a list. This list can be walked using the “next” data member of the class.

If an end point lies on the surface, the intersection is first coerced to that point and then checked against the bounding box. In this way, a vertex is treated the same for every edge running through it.

Limitations: None

References: BASE SPApar_pos, SPAparameter, SPAposition

Data:

`public const void *data1;`
General purpose pointer used to store the element on a meshsurf from which this intersection originated.

`public const void *data2;`
General purpose pointer used to store the element on a compcurv from which this intersection originated.

`public curve_surf_int *next;`
The pointer to the list.

`public curve_surf_rel high_rel;`
The relationship between the curve and the surface in the neighborhood of the intersection, in the positive parameter direction.

`public curve_surf_rel low_rel;`
The relationship between the curve and the surface in the neighborhood of the intersection, in the negative parameter direction.

`public curve_surf_userdata *userdata;`
Pointer to an arbitrary object to store user data. If non–NULL, it is deleted when this object is deleted. It is the responsibility of the user’s class derived from this to ensure that the destructor does what is necessary.

`public logical fuzzy;`
This is TRUE if the intersection is not tightly defined (a tangency or small-angle crossing).

`public SPApar_pos surf_param;`
The parameters of the intersection point on the surface.

```
public SPAparameter high_param;
```
The high end of the parameter range if it is fuzzy; the same as param if it is not fuzzy.

```
public SPAparameter low_param;
```
The low end of the parameter range if it is fuzzy; the same as param if it is not fuzzy.

```
public SPAposition int_point;
```
The point of intersection.

```
public SPAparameter param;
```
The parameters of the intersection point on the curve.

```
public double tolerance;
```
Supports tolerant modeling. The value is used to record the tolerance value of the intersection. It is defaulted to SPAresabs.

Constructor:

```
public: curve_surf_int::curve_surf_int (
    curve_surf_int*,         // next intersection
    SPAposition const&,      // intersection point
    double,                  // param at intersection
    curve_surf_rel           // relationship before
        = curve_unknown,     // param
    curve_surf_rel           // relationship after
        = curve_unknown      // param
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments.

```
public: curve_surf_int::curve_surf_int (
    curve_surf_int const&    // curve-surf
                             // intersection
    );
```

C++ copy constructor requests memory for this object and populates it with the data from the object supplied as an argument.

Destructor:

```
public: curve_surf_int::~curve_surf_int ();
```

C++ destructor, deleting a curve_surf_int.

Methods:

```
public: void curve_surf_int::debug (
    FILE*                   // file name
        = debug_file_ptr
    ) const;
```

Writes debug information about curve_surf_int to the printer or to the specified file.

Related Fncs:

delete_curve_surf_ints