Chapter 3.
Classes Aa thru Bz

Topic: Ignore

The class interface is a set of C++ classes, including their public and protected data and methods (member functions), that an application can use directly to interact with ACIS. Developers may also derive their own classes from these classes to add application-specific functionality and data. Refer to the *3D ACIS Online Help User's Guide* for a description of the fields in the reference template.

base_curve_law_data

Class: Laws

Purpose: Abstract base class to access curve_law_data with or without the ACIS kernel.

Derivation: base_curve_law_data : path_law_data : law_data : ACIS_OBJECT : –

SAT Identifier: None

Filename: law/lawutil/law_data.hxx

Description: Refer to Purpose.

Limitations: None

References: None

Data:

None

Constructor:

```
public: base_curve_law_data::base_curve_law_data (
    double in_start          // start parameter
        = 0,                 //
    double in_end            // end parameter
        = 0                  //
    );
```

Default C++ constructor.

Destructor:

None

Methods:

```
public: virtual law*
    base_curve_law_data::law_form ()=0;
```

Returns a pointer to the law class used as part of the base_curve_law_data

```
public: virtual double base_curve_law_data::length (
    double start,           // start param
    double end              // end param
    )=0;
```

Returns the distance between the two parameters.

```
public: virtual double
base_curve_law_data::length_param (
    double base,            // starting parameter
    double length           // distance to end
    )=0;
```

Returns the parameter value at the end point.

```
public: virtual double
    base_curve_law_data::point_perp (
    SPAposition in_point    // point
    )=0;
```

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

```
public: virtual double
    base_curve_law_data::point_perp (
    SPAposition in_point,   // point
    double in_t             // parameter
    )=0;
```

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

Internal Use: full_size

Related Fncs:

law_data_array

base_pcurve_law_data

Class: Laws

Purpose: Abstract base class to access pcurve_law_data with or without the ACIS kernel.

Derivation: base_pcurve_law_data : path_law_data : law_data : ACIS_OBJECT : –

SAT Identifier: None

Filename: law/lawutil/law_data.hxx

Description: This is a wrapper to handle specific ACIS pcurve classes. These wrapper classes are used by api_str_to_law. These are returned by the law method string_and_data.

Limitations: None

References: None

Data:

None

Constructor:

```
public: base_pcurve_law_data::base_pcurve_law_data (
    double in_start          // start parameter
        = 0,                 //
    double in_end            // end parameter
        = 0                  //
    );
```

Default constructor.

Destructor:

None

Methods:

None

Related Fncs:

law_data_array

base_surface_law_data

Class: Laws

Purpose: Abstract base class to access surface_law_data with or without the ACIS kernel.

Derivation: base_surface_law_data : law_data : ACIS_OBJECT : –

SAT Identifier: None

Filename: law/lawutil/law_data.hxx

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

Limitations: None

References: None

Data:

None

Constructor:

```
public:
    base_surface_law_data::base_surface_law_data ();
```

Default constructor.

Destructor:

None

Methods:

```
public: virtual SPAposition
    base_surface_law_data::bs3_eval (
    SPApar_pos const& in_par_pos// param
    ) const=0;
```

Returns the position of the u,v parameters on the spline approximating surface.

```
public: virtual void base_surface_law_data::eval (
    SPApar_pos& uv,          // param to evaluate
    SPAposition& pos,        // output surf. position
    SPAvector* dpos,         // array of 1st derivs
    SPAvector* ddpos         // array of 2nd derivs
    )=0;
```

This takes in a uv parameter value and returns the corresponding xyz position on the surface; a vector array, which holds the derivative with respect to u and the derivative with respect to v; and an array with three vectors, which correspond to the second derivative with respect to u, the derivative with respect to u and then to v, and the second derivative with respect to v.

```
public: virtual double
    base_surface_law_data::eval_gaussian_curvature (
    SPApar_pos const& in_par_pos// param
    ) const=0;
```

Returns the Gaussian curvature at the specified parameter.

```
public: virtual double
    base_surface_law_data::eval_max_curvature (
    SPApar_pos const& in_par_pos// param
    ) const=0;
```

Returns the maximum curvature at the specified parameter.

```
public: virtual double
    base_surface_law_data::eval_mean_curvature (
    SPApar_pos const& in_par_pos// param
    ) const=0;
```

Returns the mean curvature at the specified parameter.

```
public: virtual double
    base_surface_law_data::eval_min_curvature (
    SPApar_pos const& in_par_pos// param
    ) const=0;
```

Returns the minimum curvature at the specified parameter.

```
public: virtual SPApar_pos
    base_surface_law_data::point_perp (
    SPAposition in_point    // point
    )=0;
```

Finds the parameter position on the surface perpendicular to the given position outside of the surface.

```
public: virtual SPApar_pos
    base_surface_law_data::point_perp (
    SPAposition in_point,   // point
    SPApar_pos in_par_pos   // param
    )=0;
```

Finds the parameter position on the surface perpendicular to the given position outside of the surface.

```
public: virtual logical
    base_surface_law_data::term_domain (
    int which,              // term to bound
    SPAinterval& answer     // bounds for term
    )=0;
```

Establishes the domain of a given term in the law.

Related Fncs:

law_data_array

base_transform_law_data

Class: Laws

Purpose: Abstract base class to access transform_law_data with or without the ACIS kernel.

Derivation: base_transform_law_data : law_data : ACIS_OBJECT : –

SAT Identifier: None

Filename: law/lawutil/law_data.hxx

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

Limitations: None

References: BASE SPAtransf

Data:

```
protected SPAtransf *data;
```
Holds the data.

```
protected SPAtransf *data_inverse;
```
Inverse transform.

Constructor:

```
public: base_transform_law_data::
    base_transform_law_data (
    transf const* in_data   // transform to wrap
    );
```

C++ constructor, creating a transform_law_data which is a wrapper for the ACISTRANSFORM.

Destructor:

```
public: base_transform_law_data::
    ~base_transform_law_data ();
```

Default destructor.

Methods:

```
public: virtual law_data*
    base_transform_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.

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: SPAtransf*
    base_transform_law_data::get_trans ();
```

Returns the transform.

```
public: virtual base_transform_law_data*
    base_transform_law_data::make_one (
    SPAtransf const* in_data// array of transforms
    ) const =0;
```

Returns a pointer to a law of this type. Used by parsing to create an instance of this law.

```
public: SPAvector base_transform_law_data::rotate (
    SPAvector v             // rotation vector
    );
```

Performs a rotation transformation.

```
public: SPAvector
    base_transform_law_data::rotate_inverse (
    SPAvector v             // rotation vector
    );
```

Performs an inverse rotation transformation.

```
public: SPAposition
    base_transform_law_data::transform (
    SPAposition p           // position
    );
```

Transforms the specified position by the transform in base_transform_law_data.

```
public: SPAposition
    base_transform_law_data::transform_inverse (
    SPAposition p           // position
    );
```

Performs an inverse transform on the specified position by the transform in base_transform_law_data.

Internal Use: full_size

Related Fncs: law_data_array

base_wire_law_data

Class: Laws

Purpose: Abstract base class to access wire_law_data with or without the ACIS kernel.

Derivation: base_wire_law_data : path_law_data : law_data : ACIS_OBJECT : –

SAT Identifier: None

Filename: law/lawutil/law_data.hxx

Description: Refer to Purpose.

Limitations: None

References: None

Data:

None

Constructor:

```
public: base_wire_law_data::base_wire_law_data (
    double in_start          // start parameter
        = 0,                 //
    double in_end            // end parameter
        = 0                  //
    );
```

Default constructor.

Destructor:

None

Methods:

None

Related Fncs:

law_data_array

bend_law

Class: Laws, SAT Save and Restore

Purpose: Creates a law to bend from a position around an axis in a given direction a specified amount.

Derivation: bend_law : multiple_law : law : ACIS_OBJECT : –

SAT Identifier: “BEND”

Filename: law/lawutil/main_law.hxx

Description: Creates a law to bend from a position around an axis in a given direction a specified amount.

Limitations: None

References: LAW law

Data:

None

Constructor:

```
public: bend_law::bend_law (
    law** in_sublaw,         // array of sublaws
    int in_size              // size of array
    );
```

C++ constructor, creating a law. This sets the use_count to 1 and increments the how_many_laws. It sets dlaw, slaw, and lawdomain to NULL.

```
public: bend_law::bend_law (
    SPAposition root,        // root position
    SPAunit_vector axis,     // axis
    SPAunit_vector dir,      // direction for bend
    double r                 // distance
    );
```

C++ constructor, creating a law. This sets the use_count to 1 and increments the how_many_laws. It sets dlaw, slaw, and lawdomain to NULL.

Destructor:

```
public: bend_law::~bend_law ();
```

C++ destructor, deleting a bend_law.

Methods:

```
public: char const* bend_law::class_name ();
```

This method returns a string that contains the name of this class. It is provided as a user-friendly interface to laws.

```
public: int bend_law::date () const;
```

Returns the version of ACIS that the law was added in. If a law is part of a model that is to be saved at a previous ACIS release level, this is used to indicated whether the law can be saved or not.

```
protected: law* bend_law::deriv (
    int which               // variable to take
                            // derivative with
                            // respect to
        = 0                 // default value (X
                            // or A1)
    ) const;
```

This method returns a law pointer that is the derivative of the given law with respect to the which variable. Variables in C++ are numbered starting at zero (0). The default is to take a derivative with respect to the first variable, which in a law function string is A1 or X. The variables X, Y, and Z are equivalent to the indices 0, 1, and 2, respectively.

The deriv method should *not* called directly by applications. It is called by the derivative method, which is inherited by all classes derived from law. All classes derived from law (or its children) must implement their own deriv method to perform the actual derivative calculation when called by derivative.

```
public: void bend_law::evaluate (
    double const* x,        // pointer to values
                            // used in evaluation
    double* answer          // multi-dimension answer
                            // range for evaluation
    ) const;
```

This method takes two pointers to memory that the caller is responsible for creating and freeing. The x argument tells where to evaluate the law. This can be more than one dimension. The answer argument returns the evaluation. This can be more than one dimension. x should be the size returned by the take_dim method, and answer should be the size returned by the return_dim method. All derived law classes must have this method or inherit it. This does no checking of the dimension of input and output arguments. It is preferable to call a more specific evaluator if possible.

```
public: static int bend_law::id ();
```

This method should not be called directly by the application. All derived law classes must have this method. The isa, id, and and type methods are used to identify a law's class type.

```
public: logical bend_law::isa (
    int t                   // id method return
    ) const;
```

All derived law classes must have this method. The isa, id, and and type methods are used to identify a law's class type. The methods should be the same for all law classes with the exception of the isa method that calls the isa method of its parent class. To test to see if a law is a given type, use test_law–>isa(constant_law::id()). If test_law is a constant_law or is derived from the constant_law class, this returns TRUE.

```
public: multiple_law* bend_law::make_one (
    law** in_datas,         // array of law data
    int in_size             // size of array
    ) const;
```

Returns a pointer to a law of this type. Used by parsing to create an instance of this law.

```
public: int bend_law::return_size () const;
```

The return_size tells how many values are returned in the answer argument of the evaluate method. The default is 1. All derived law classes must have this method or inherit it.

```
protected: law* bend_law::sub_inverse () const;
```

Returns a pointer to the sublaws that are used to make up the inverse law of this class.

```
public: virtual law* bend_law::sub_simplify (
    int level = 0,              //
    char const*& what = *       //
        (char const** )NULL_REF //
    ) const;
```

This is a member function that may be overloaded by classes to provide assistance to the simplifier. It helps the simplifier in dealing with this particular law.

An example is that a curve law applied to a line or ellipse can return an equation in the form of a law. The sub_simplify method can then access and private members of the curve law that the simplifier does not have access to. Most laws simply inherit a function that returns null.

```
public: char const* bend_law::symbol (
    law_symbol_type type     // type of law symbol
        = DEFAULT            // standard ACIS type
    ) const;
```

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.

This is the top-level class. The default law symbol for this class is an error message. Derived classes need to define their own symbol methods to override this error message.

```
public: int bend_law::take_size () const;
```

Returns the dimension of the law's domain (input). The default is 1. All derived law classes must have this method or inherit it.

```
public: int bend_law::type () const;
```

All derived law classes must have this method. The isa, id, and and type methods are used to identify a law's class type. The methods should be the same for all law classes.

Internal Use: full_size, hasa

Related Fncs: initialize_law, terminate_law

binary_law

Class: Laws

Purpose: Provides methods and data for laws that have two sublaws.

Derivation: binary_law : law : ACIS_OBJECT : –

SAT Identifier: None

Filename: law/lawutil/main_law.hxx

Description: A binary law is a law with two arguments. Most arithmetic functions are binary laws.

Applications should call the virtual remove method instead of the tilde (~) destructor to get rid of a law. This decrements the use_count. This is called by the law destructors for the law being destructed, as well as for all of its sublaws. remove calls the destructor if use_count falls to zero. Used for memory management.

Limitations: None

References: LAW law

Data:

```
protected law *left_law;
```
Pointer to the first argument (sublaw) of the binary law.

```
protected law *right_law;
```
Pointer to the second argument (sublaw) of the binary law.

Constructor:

```
public: binary_law::binary_law (
    law* in_left_law        // pointer to 1st sublaw
        = NULL,
    law* in_right_law       // pointer to 2nd sublaw
        = NULL
    );
```

C++ constructor, creating a binary_law. This takes two arguments (sublaws) as the operands.

Destructor:

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

Applications should call the virtual remove method instead of the tilde (~) destructor to get rid of a law.

Methods:

```
public: virtual int binary_law::associative () const;
```

Returns whether or not the given law is associative. Associative means in the case of plus, (A+B)+C=A+(B+C). The default is FALSE, meaning it is not associative. An example of a binary law that is not associative is the minus law.

```
public: virtual int binary_law::commutative () const;
```

Returns whether or not the given law is commutative. Commutative means in the case of plus, A+B=B+A. The default is FALSE, meaning it is not commutative. An example of a binary law that is not commutative is the minus law.

```
public: int binary_law::date () const;
```

Returns the version of ACIS that the law was added in. If a law is part of a model that is to be saved at a previous ACIS release level, this is used to indicated whether the law can be saved or not.

```
public: virtual law* binary_law::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: law* binary_law::fleft () const;
```

This returns a pointer to the left operand (sublaw) of the binary law. Only applications that create new laws that have parts of old laws should use this method. If the sublaw is to be used elsewhere, the add method should called.

```
public: law* binary_law::fright () const;
```

This returns a pointer to the right operand (sublaw) of the binary law. Only applications that create new laws that have parts of old laws should use this method. If the sublaw is to be used elsewhere, the add method should called.

```
public: static int binary_law::id ();
```

This method should not be called directly by the application. All derived law classes must have this method. The isa, id, and and type methods are used to identify a law's class type.

```
public: virtual logical binary_law::in_domain (
    double * where           // where to test domain
    ) const;
```

Checks to see if the given item is within the domain.

```
public: logical binary_law::isa (
    int t                   // id method return
    ) const;
```

All derived law classes must have this method. The isa, id, and and type methods are used to identify a law's class type. The methods should be the same for all law classes with the exception of the isa method that calls the isa method of its parent class. To test to see if a law is a given type, use test_law–>isa(constant_law::id()). If test_law is a constant_law or is derived from the constant_law class, this returns TRUE.

```
public: virtual binary_law* binary_law::make_one (
    law* in_left_law,       // pointer to 1st sublaw
    law* in_right_law       // pointer to 2nd sublaw
    ) const;
```

Returns a pointer to a law of this type. Used by parsing to create an instance of this law. All laws derived from binary_law have a make_one method. This is used by the parser and simplifier.

```
public: int binary_law::return_size () const;
```

Returns the dimension of the range (output) of the law. The default is 1. All derived law classes must have this method or inherit it.

```
public: int binary_law::same (
    law const*,             // 1st law to test
    law const*              // 2nd law to test
    ) const;
```

This method should not be called directly by the application. This method is called by the == method, to see if two laws of the same class type are the same.

```
public: law* binary_law::set_domain (
    SPAinterval* new_domain,// new input domain
    logical set             // what to set
        = FALSE
    );
```

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

```
public: int binary_law::singularities (
    double** where,             // where
                                // discontinuities
                                // exist
    int** type,                 // type of
                                // discontinuity
    double start                // start
        = –DBL_MAX,
    double end                  // end
        = DBL_MAX,
    double** period             // array of period
        = NULL
    ) const;
```

This specifies where in the given law there might be discontinuities. The array where notes where the discontinuity occurred. The type indicates 0 if there is a discontinuity, 1 if the discontinuity in the 1st derivative, and any integer *n* if the discontinuity is in the *n*–th derivative. –1 means that it is not defined.

```
public: char* binary_law::string (
    law_symbol_type type    // type of law
        = DEFAULT,          // standard ACIS type
    int& count              // count
        = *(int* ) NULL_REF,
    law_data_node*& ldn     // law data node
        = *(law_data_node** ) NULL_REF
    ) const;
```

This method returns a string that names (identifies) the law. This name is used when parsing and when saving or restoring the law to or from a file.

```
public: int binary_law::take_size () const;
```

Returns the dimension of the law's domain (input). The default is 1. All derived law classes must have this method or inherit it.

```
public: virtual logical binary_law::term_domain (
    int term,               // term to bound
    SPAinterval& domain     // bounds for term
    ) const;
```

Establishes the domain of a given term in the law.

```
public: int binary_law::type () const;
```

All derived law classes must have this method. The isa, id, and and type methods are used to identify a law's class type. The methods should be the same for all law classes.

Internal Use: full_size, hasa

Related Fncs: initialize_law, terminate_law