Chapter 6.
Classes Ma thru Rz

Topic: Ignore

minus_law

Class: Laws, SAT Save and Restore

Purpose: Provides methods for the minus, or subtraction, mathematical function.

Derivation: minus_law : binary_law : law : ACIS_OBJECT : –

SAT Identifier: "–"

Filename: law/lawutil/main_law.hxx

Description: Refer to Purpose.

Limitations: None

References: None

Data:

None

Constructor:

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

C++ constructor, creating a minus_law. This has two pointers to sublaws passed in as arguments.

Destructor:

None

Methods:

```
public: char const* minus_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.

```
protected: law* minus_law::deriv (
    int which               // variable to take
        = 0                 // derivative with
                            // respect to 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 implements the code to perform the actual derivative calculation and caches its value in memory. All classes derived from law (or its children) must implement their own deriv method.

The deriv method should *not* be called directly by applications. Applications should call the derivative method instead, which is inherited by all classes derived from law. The derivative method accesses the cached derivative value in memory, if one exists; otherwise it calls the deriv method.

```
public: void minus_law::evaluate (
    double const* x,        // 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 freezing. 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.

```
public: static int minus_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 minus_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: binary_law* minus_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.

```
public: law_polynomial* minus_law::polynomial (
    law* in                 // input law
    ) const;
```

The minus_law has its own rules for governing how the polynomial degree for a given top-level law is determined.

```
public: int minus_law::precedence () const;
```

Returns PRECEDENCE_PLUS that tell what the precedence of the law is during evaluation.

The valid precedence values are PRECEDENCE_PLUS (1), PRECEDENCE_TIMES (2), PRECEDENCE_POWER (3), PRECEDENCE_FUNCTION (4), and PRECEDENCE_CONSTANT (5),

This is used for simplification and parsing. For a law to be saved and restored, it must have or inherit this method.

```
public: virtual law* minus_law::sub_simplify (
    int level                   // level of
        = 0,                    // simplification
    char const*& what           // text string
        = * (char const** )     // describing the
        NULL_REF                // simplified law
    ) const;
```

This is a member function that may be overloaded by derived classes to provide assistance to the simplifier. It helps the simplifier in dealing with this particular law. This method is called by the simplifier but generally not called directly by the application.

For example, a law class such as plus_law might use an equation “x + x”. The sub_simplify method could return this equation as “2*x”. The sub_simplify method can access the private members of the law that the simplifier does not have access to. Most laws simply inherit a function that returns null.

```
public: char const* minus_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.

The default law symbol for this class is –.

```
public: int minus_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.

Related Fncs:

initialize_law, terminate_law

multiple_data_law

Class: Laws

Purpose: Provides methods and data for laws that have multiple law data members.

Derivation: multiple_data_law : law : ACIS_OBJECT : –

SAT Identifier: None

Filename: law/lawutil/main_law.hxx

Description: This law has multiple law data arguments, such as the curve_law, wire_law, and surface_law. These are parsed with law tags. For example, the “MAP(LAW1, EDGE2)” is followed by a law and an edge. The law and the edge are law datas. These are passed to the map_law. This permits an application to pass a ACIS entities and other classes in the form of law_data classes into 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

Data:

```
protected int size;
```
This is the number of elements in the data set to the multiple data law.

```
protected law_data **datas;
```
This is an array of law data pointer used by the the multiple data law.

Constructor:

```
public: multiple_data_law::multiple_data_law ();
```

Constructor for the multiple law data.

```
public: multiple_data_law::multiple_data_law (
    law_data** in_sub_law,   // array of law data
    int in_size              // size of array
    );
```

C++ constructor, creating a multiple_data_law. This takes two arguments. The first is an array of pointers to the sublaw data members. The second is the number of sublaw data elements in the law list.

Destructor:

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

Use the remove method instead. Destructor for the law.

Methods:

```
public: int multiple_data_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: law* multiple_data_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: int multiple_data_law::fsize () const;
```

Returns the size of the sublaw array.

```
public: law_data** multiple_data_law::fsubs () const;
```

Returns an array of law data elements associated with sublaws.

```
public: static int multiple_data_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 multiple_data_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 multiple_data_law*
    multiple_data_law::make_one (
    law_data** in_data,      // 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.

All laws derived from unary_law have a make_one method. This is used by the parser and simplifier and should not be called by the application directly.

```
public: int multiple_data_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 is used for simplification. It must have or inherit this method. This method is called by the == method, to see if two laws are the same.

```
public: law* multiple_data_law::set_domain (
    SPAinterval* new_domain, // new input domain
    logical set              // flag for setting
        = FALSE
    );
```

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

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

Returns a string that represents the current law function. The law function is composed of its symbol, associated parentheses, and the strings associated with its sublaws. It is provided as a user-friendly interface to laws. A derived class must override this function to be able to save a law.

```
public: int multiple_data_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

multiple_law

Class: Laws

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

Derivation: multiple_law : law : ACIS_OBJECT : –

SAT Identifier: None

Filename: law/lawutil/main_law.hxx

Description: A multiple law is a law with one or more arguments. Some special operations, such as maximum, minimum, or vector, are multiple laws.

Multiple laws with two arguments differ from binary laws in how they parse. A multiple law starts with the operator and is followed by the operands in parenthesis and separated by commas. A binary law expects the left operand, the operator, and then the right operand. The dot_law is a multiple law.

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 int sub_num;
```

Integer specifying how many sublaws are used by this law.

```
protected law **sub;
```

Pointer to the first sublaw that is used as input to this law.

Constructor:

```
public: multiple_law::multiple_law (
    law** in_sub            // array of sublaws
        = NULL,
    int in_sub_num          // size of array
        = 0
    );
```

C++ constructor, creating a multiple_law. This takes two arguments. The first is a pointer to the first law in a law list. The second is the number of sublaws in the law list.

Destructor:

```
protected: multiple_law::~multiple_law ();
```

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

Methods:

```
public: virtual logical
    multiple_law::commutative () const;
```

Returns whether or not the given law is commutative. Commutative means in the case of the dot product, “DOT(A,B)” is equals to ”DOT(B,A)”. The default is FALSE, meaning it is not commutative. An example of a multiple law that is not commutative is the cross law; “CROSS(A,B)” is not equal to “CROSS(B,A)”.

```
public: int multiple_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 indicate whether the law can be saved or not.

```
public: virtual law* multiple_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** multiple_law::fsub () const;
```

This returns a pointer to the first operand (sublaw) of the multiple 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: int multiple_law::fsub_num () const;
```

Returns number of operands (sublaws) of the multiple law.

```
public: static int multiple_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 multiple_law::in_domain (
    double * where          // where to test domain
    ) const;
```

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

```
public: logical multiple_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 multiple_law*
    multiple_law::make_one (
    law** in_sub,           // array of sublaws
    int in_sub_num          // size of array
    ) const;
```

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

```
public: int multiple_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 multiple_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* multiple_law::set_domain (
    SPAinterval* new_domain,// new input domain
    logical set             // flag for setting
        = FALSE
    );
```

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

```
public: int multiple_law::singularities (
    double** where,         // where discontinuity
                            // exist
    int** type,             // type of discontinuity
    double start                // start
        = -DBL_MAX,
    double end                  // end
        = DBL_MAX,
    double** period         // 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* multiple_law::string (
    law_symbol_type type     // type of law symbol
        = DEFAULT,           // standard ACIS type
    int& count               // count
        = *(int* ) NULL_REF,
    law_data_node*& ldn      // law data node
        = *(law_data_node** ) NULL_REF
    ) const;
```

Returns a string that represents the current law function. The law function is composed of its symbol, associated parentheses, and the strings associated with its sublaws. It is provided as a user-friendly interface to laws. A derived class must override this function to be able to save a law.

```
public: int multiple_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: logical multiple_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 multiple_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

negate_law

Class: Laws, SAT Save and Restore

Purpose: Provides methods for the unary minus, or negation, mathematical function.

Derivation: negate_law : unary_law : law : ACIS_OBJECT : –

SAT Identifier: "–"

Filename: law/lawutil/main_law.hxx

Description: Refer to Purpose.

Limitations: None

References: None

Data:

None

Constructor:

```
public: negate_law::negate_law (
    law* in_sublaw          // pointer to sublaw
    );
```

C++ constructor, creating a negate_law. This has a pointer to a sublaw passed in as one of its arguments.

Destructor:

None

Methods:

```
public: char const* negate_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.

```
protected: law* negate_law::deriv (
    int which               // variable to take
        = 0                 // derivative with
                            // respect to 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 implements the code to perform the actual derivative calculation and caches its value in memory. All classes derived from law (or its children) must implement their own deriv method.

The deriv method should *not* be called directly by applications. Applications should call the derivative method instead, which is inherited by all classes derived from law. The derivative method accesses the cached derivative value in memory, if one exists; otherwise it calls the deriv method.

```
public: void negate_law::evaluate (
    double const* x,         // 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 freezing. 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.

```
public: static int negate_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 negate_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: unary_law* negate_law::make_one (
    law* in_sublaw          // pointer to sublaw
    ) const;
```

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

```
public: law_polynomial* negate_law::polynomial (
    law* in                 // input law
    ) const;
```

The negate_law has its own rules for governing how the polynomial degree for a given top-level law is determined.

```
public: virtual law* negate_law::sub_simplify (
    int level                   // level of
        = 0,                    // simplification
    char const*& what           // text string
        = * (char const** )     // describing the
        NULL_REF                // simplified law
    ) const;
```

This is a member function that may be overloaded by derived classes to provide assistance to the simplifier. It helps the simplifier in dealing with this particular law. This method is called by the simplifier but generally not called directly by the application.

For example, a law class such as plus_law might use an equation “x + x”. The sub_simplify method could return this equation as “2*x”. The sub_simplify method can access the private members of the law that the simplifier does not have access to. Most laws simply inherit a function that returns null.

```
public: char const* negate_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.

The default law symbol for this class is –.

```
public: int negate_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.

Related Fncs:

initialize_law, terminate_law

norm_law

Class: Laws, SAT Save and Restore

Purpose: Provides methods for the normalize mathematical function.

Derivation: norm_law : unary_law : law : ACIS_OBJECT : –

SAT Identifier: "NORM"

Filename: law/lawutil/main_law.hxx

Description: This law normalizes the length of my_law to be of unit length. This is accomplished by dividing each dimension element by the square root of the sum of the squares of all of the return elements. This is applicable to a law that returns any dimension.

Limitations: None

References: LAW law

Data:

None

Constructor:

```
public: norm_law::norm_law (
    law* in_sublaw          // pointer to sublaw
    );
```

C++ constructor, creating a norm_law. This has two or more pointers to sublaws passed in as arguments.

Destructor:

```
public: norm_law::~norm_law ();
```

Destructor for this law class.

Methods:

```
public: char const* norm_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.

```
protected: law* norm_law::deriv (
    int which               // variable to take
        = 0                 // derivative with
                            // respect to 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 implements the code to perform the actual derivative calculation and caches its value in memory. All classes derived from law (or its children) must implement their own deriv method.

The deriv method should *not* be called directly by applications. Applications should call the derivative method instead, which is inherited by all classes derived from law. The derivative method accesses the cached derivative value in memory, if one exists; otherwise it calls the deriv method.

```
public: void norm_law::evaluate (
    double const* x,        // 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 freezing. The x argument tells where to evaluate the laws. This is passed into all laws making up the maximum function. This can be more than one dimension. The answer argument returns the result of the evaluation from all sublaws that is maximum. 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.

```
public: static int norm_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 norm_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: unary_law* norm_law::make_one (
    law* in_sublaw          // pointer to sublaw
    ) const;
```

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

```
public: int norm_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: virtual law* norm_law::sub_simplify (
    int level                   // level of
        = 0,                    // simplification
    char const*& what           // text string
        = * (char const** )     // describing the
        NULL_REF                // simplified law
    ) const;
```

This is a member function that may be overloaded by derived classes to provide assistance to the simplifier. It helps the simplifier in dealing with this particular law. This method is called by the simplifier but generally not called directly by the application.

For example, a law class such as plus_law might use an equation “x + x”. The sub_simplify method could return this equation as “2*x”. The sub_simplify method can access the private members of the law that the simplifier does not have access to. Most laws simply inherit a function that returns null.

```
public: char const* norm_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.

The default law symbol for this class is NORM.

```
public: int norm_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

path_law_data

Class: Laws

Purpose: Creates a wrapper for either a curve or wire class for input into a law.

Derivation: path_law_data : law_data : ACIS_OBJECT : –

SAT Identifier: None

Filename: law/lawutil/law_data.hxx

Description: This allows curves and wires behave in the same way. It places a global parameterization to all of the curves in a wire.

Limitations: None

References: None

Data:

```
protected double end;
```
The ending parameter value of the path.

```
protected double start;
```
The starting parameter value of the path.

Constructor:

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

C++ constructor, creating a path_law_data. This is an abstract data class. An instance of this is never created directly. One of the derived classes from this class calls this constructor method.

Destructor:

None

Methods:

```
public: virtual double path_law_data::curvature (
    double para              // parameter for test
    )=0;
```

Returns the curvature of the path at the given parameter.

```
public: double path_law_data::data_end ();
```

Returns the ending parameter of the path.

```
public: double path_law_data::data_start ();
```

Returns the ending parameter of the path.

```
public: virtual SPAvector path_law_data::eval (
    double para,            // parameter position
    int deriv,              // number of derivatives
    int side                // left (–1) or right (1)
        = 0                 // default = 0
    )=0;
```

This method evaluates the *n*th derivative of the given law at the parameter position value. All law classes inherit this method for convenience. It calls the main evaluate member function and does some checking of take_dim and return_dim.

```
public: virtual int path_law_data::singularities (
    double** where,         // where discontinuities
                            // may exist
    int** type,             // type of discontinuity
    double start,           // start
    double end              // end
    );
```

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.

Internal Use: full_size

Related Fncs: law_data_array

pcurve_law

Class: Laws, Spline Interface, Construction Geometry, Object and Parameter Spaces, SAT Save and Restore

Purpose: Creates a law to support parameter curve calculations.

Derivation: pcurve_law : unary_data_law : law : ACIS_OBJECT : –

SAT Identifier: “PCUR”

Filename: law/lawutil/main_law.hxx

Description: pcurve_law's returns a SPApar_pos uv value given a parameter value. It returns two values and takes 1.

Limitations: None

References: None

Data:

None

Constructor:

```
public: pcurve_law::pcurve_law (
    base_pcurve_law_data*    // array of
     in_law_data             // base_pcurve_law_data
    );
```

C++ initialize constructor requests memory for an instance of pcurve_law and populates it with the data supplied as arguments. This sets the use_count to 1 and increments the how_many_laws. It sets dlaw, slaw, and lawdomain to NULL.

Destructor:

None

Methods:

```
public: char const* pcurve_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 pcurve_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* pcurve_law::deriv (
    int which               // which derivative
        = 0                 // to take
    ) 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: law_domain* pcurve_law::domain ();
```

Returns a law_domain class which contains information about the domain of the law.

```
public: void pcurve_law::evaluate (
    double const* x,         // array of input values
    double* answer           // array of output values
    ) 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 pcurve_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 pcurve_law::isa (
    int t                    // number from id
                             // method
    ) 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: unary_data_law* pcurve_law::make_one (
    law_data* in_data       // array of data
    ) const;
```

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

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

The return_dim 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.

```
public: int pcurve_law::singularities (
    double** where,         // location of
                            // singularity
    int** type,             // type of singularity
    double start                // start
        = –DBL_MAX,
    double end                  // end
        = DBL_MAX,
    double** period         // 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 const* pcurve_law::symbol (
    law_symbol_type type    // type of law
        = DEFAULT
    ) 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.

The default law symbol for this class is PCUR.

```
public: logical pcurve_law::term_domain (
    int term,               // term to change
    SPAinterval& domain     // domain to establish
    ) const;
```

Establishes the domain of a given term in the law.

```
public: int pcurve_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.

Related Fncs:

initialize_law, terminate_law

plus_law

Class: Laws, SAT Save and Restore

Purpose: Provides methods for the plus, or addition, mathematical function.

Derivation: plus_law : binary_law : law : ACIS_OBJECT : –

SAT Identifier: "+"

Filename: law/lawutil/main_law.hxx

Description: Refer to Purpose.

Limitations: None

References: None

Data:

None

Constructor:

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

C++ constructor, creating a plus_law. This has two pointers to sublaws passed in as arguments.

Destructor:

None

Methods:

```
public: int plus_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: char const* plus_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 plus_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.

```
protected: law* plus_law::deriv (
    int which               // variable to take
        = 0                 // derivative with
                            // respect to 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 implements the code to perform the actual derivative calculation and caches its value in memory. All classes derived from law (or its children) must implement their own deriv method.

The deriv method should *not* be called directly by applications. Applications should call the derivative method instead, which is inherited by all classes derived from law. The derivative method accesses the cached derivative value in memory, if one exists; otherwise it calls the deriv method.

```
public: void plus_law::evaluate (
    double const* x,         // 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 freezing. 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.

```
public: static int plus_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 plus_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: binary_law* plus_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.

```
public: law_polynomial* plus_law::polynomial (
    law* in                  // input law
    ) const;
```

The plus_law has its own rules for governing how the polynomial degree for a given top-level law is determined.

```
public: int plus_law::precedence () const;
```

Returns PRECEDENCE_PLUS that tell what the precedence of the law is during evaluation.

The valid precedence values are PRECEDENCE_PLUS (1), PRECEDENCE_TIMES (2), PRECEDENCE_POWER (3), PRECEDENCE_FUNCTION (4), and PRECEDENCE_CONSTANT (5),

This is used for simplification and parsing. For a law to be saved and restored, it must have or inherit this method.

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

Check to see if the plus_law is linear in one dimension.

```
public: virtual law* plus_law::sub_simplify (
    int level                    // level of
        = 0,                     // simplification
    char const*& what            // text string
        = * (char const** )      // describing the
        NULL_REF                 // simplified law
    ) const;
```

This is a member function that may be overloaded by derived classes to provide assistance to the simplifier. It helps the simplifier in dealing with this particular law. This method is called by the simplifier but generally not called directly by the application.

For example, a law class such as plus_law might use an equation “x + x”. The sub_simplify method could return this equation as “2*x”. The sub_simplify method can access the private members of the law that the simplifier does not have access to. Most laws simply inherit a function that returns null.

```
public: char const* plus_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.

The default law symbol for this class is +.

```
public: int plus_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.

Related Fncs:

initialize_law, terminate_law