Chapter 7.
Classes Sa thru Zz

Topic: Ignore

sin_law

Class: Laws, SAT Save and Restore

Purpose: Provides methods and data for the sine mathematical function.

Derivation: sin_law : unary_law : law : ACIS_OBJECT : –

SAT Identifier: “SIN”

Filename: law/lawutil/main_law.hxx

Description: Refer to Purpose.

Limitations: None

References: None

Data:

None

Constructor:

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

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

Destructor:

None

Methods:

```
public: char const* sin_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* sin_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 sin_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 sin_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 sin_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* sin_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: virtual law* sin_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* sin_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 SIN.

```
public: int sin_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

sqrt_law

Class: Laws, SAT Save and Restore

Purpose: Provides methods and data for the square root mathematical function.

Derivation: sqrt_law : unary_law : law : ACIS_OBJECT : –

SAT Identifier: "SQRT"

Filename: law/lawutil/main_law.hxx

Description: Refer to Purpose.

Limitations: None

References: None

Data:

None

Constructor:

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

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

Destructor:

None

Methods:

```
public: char const* sqrt_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* sqrt_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 sqrt_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 sqrt_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 sqrt_law::in_domain (
    double* where           // where to test domain
    ) const;
```

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

```
public: logical sqrt_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* sqrt_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: virtual law* sqrt_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* sqrt_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 SQRT.

```
public: int sqrt_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

surface_law

Class: Laws, Geometric Analysis, SAT Save and Restore

Purpose: Returns the position on a surface.

Derivation: surface_law : unary_data_law : law : ACIS_OBJECT : –

SAT Identifier: “SURF”

Filename: law/lawutil/main_law.hxx

Description: The parameterization of the surface_law is equal to the parameterization of the underlying ACIS surface.

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: None

Data:

None

Constructor:

```
public: surface_law::surface_law (
    base_surface_law_data*   // surface law
         in_law_data         // data that
                             // contains a
                             // surface
    );
```

C++ constructor, creating a surface_law. The in_law_data is a surface_law_data structure that holds an ACIS surface data structure and a starting and ending parameter.

Destructor:

None

Methods:

```
public: char const* surface_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* surface_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 surface_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 surface_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 surface_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_data_law* surface_law::make_one (
    law_data* in_data       // pointer to law data
    ) const;
```

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

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

Returns the dimension of the range (output) of the law. The default for a surface is 3. All derived law classes must have this method or inherit it.

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

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

```
public: char const* surface_law::symbol (
    law_symbol_type type    // type of law symbol
        = 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 SURF.

```
public: int surface_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 surface_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 surface_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

surfnorm_law

Class: Laws, Geometric Analysis, SAT Save and Restore

Purpose: Composes a law mathematic function that returns the normal to a surface at a given position.

Derivation: surfnorm_law : unary_law : law : ACIS_OBJECT : –

SAT Identifier: “SURFNORM”

Filename: law/lawutil/main_law.hxx

Description: surfnorm returns the normal to a surface at a given *uv* position.

ACIS defines its own parameter range for a surface which is used by this law. This law does not normalize the returned vector, because many applications only require the direction of the vector and not its normalized value.

Limitations: None

References: LAW law

Data:

None

Constructor:

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

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

Destructor:

```
public: surfnorm_law::~surfnorm_law ();
```

C++ destructor.

Methods:

```
public: char const* surfnorm_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 surfnorm_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* surfnorm_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 surfnorm_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 surfnorm_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 surfnorm_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(surfnorm_law::id()). If test_law is a surfnorm_law or is derived from the surfnorm_law class, this returns TRUE.

```
public: unary_law* surfnorm_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 surfnorm_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: virtual law* surfnorm_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* surfnorm_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 SURFNORM.

```
public: int surfnorm_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 surfnorm_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

term_law

Class: Laws, SAT Save and Restore

Purpose: Provides methods for the term mathematical function that returns a single dimensional element of a multidimensional function.

Derivation: term_law : multiple_law : law : ACIS_OBJECT : –

SAT Identifier: “TERM”

Filename: law/lawutil/main_law.hxx

Description: The term_law class supports the term_law function. It is used for picking off elements out of an array.

For example, assume my_law is a vector field defined by "vec(x, x+1, x+2, x+3)". A declaration in Scheme like (law:eval "term(my_law, 4)" 1) evaluates the fourth coordinate of my_law, x+3, at the value 1. It returns 4. A declaration like (law:eval "term(my_law, 3)" 1) evaluates the third coordinate of my_law, x+2, at the value 1. It returns 3.

Limitations: None

References: None

Data:

None

Constructor:

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

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

```
public: term_law::term_law (
    law* in_law,             // pointer to law
    int which                // which law
                             // (starts at 1)
    );
```

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

Destructor:

None

Methods:

```
public: char const* term_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* term_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 term_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 term_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 term_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* term_law::make_one (
    law** subs,             // array of sublaws
    int 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 term_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* term_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* term_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 TERM.

```
public: int term_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 term_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

times_law

Class: Laws

Purpose: Provides methods for the times, or multiplication, mathematical function.

Derivation: times_law : binary_law : law : ACIS_OBJECT : –

SAT Identifier: None

Filename: law/lawutil/main_law.hxx

Description: Refer to Purpose.

Limitations: None

References: None

Data: None

Constructor:

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

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

Destructor:

None

Methods:

```
public: int times_law::associative () const;
```

Returns whether or not the given law is associative. Associative means in the case of times, (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* times_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 times_law::commutative () const;
```

Returns whether or not the given law is commutative. Commutative means in the case of times, 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* times_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 times_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 times_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 times_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* times_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* times_law::polynomial (
    law* in                  // pointer to 1st sublaw
    ) const;
```

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

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

Returns PRECEDENCE_TIMES 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* times_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* times_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 times_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

transform_law

Class: Laws, SAT Save and Restore

Purpose: Applies an ACIS transform to a law that returns a three dimensional position.

Derivation: transform_law : multiple_data_law : law : ACIS_OBJECT : –

SAT Identifier: "TRANS"

Filename: law/lawutil/main_law.hxx

Description: Applies an ACIS transform to a law that returns a three dimensional position. The sublaw must be a law with a three dimensional range. The output of the sublaw is transformed by the ACIS transformation passed to it. The transform may rotate, scale, and translate the output of the sublaw.

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: None

Data:

None

Constructor:

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

C++ constructor, creating a transform_law. Accepts an array of size 2, where the first term of the array is a law_law_data class instance which contains the law to be transformed. The second term is a transform_law_data class instance that contains the ACIS transformation.

Destructor:

None

Methods:

```
public: char const* transform_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* transform_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 transform_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: void transform_law::evaluate_with_side (
    double const* x,         // input value
    double* answer,          // answer
    int const* side          // left (-1) or right (1)
        = NULL               // default is 0
    ) const;
```

Select which side of the input value to evaluate from, then evaluate. If the input value is an endpoint, this can be important for numerical optimization.

```
public: SPAtransf transform_law::get_trans (
    logical& f_simple_trans // true = identity
    );
```

This method returns the transform from the law transform.

```
public: static int transform_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 transform_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_data_law* transform_law::make_one (
    law_data** 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 transform_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.

```
protected: law* transform_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* transform_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* transform_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 TRANS.

```
public: int transform_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 transform_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 transform_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

unary_data_law

Class: Laws

Purpose: Provides methods and data for laws that have one law data member.

Derivation: unary_data_law : law : ACIS_OBJECT : –

SAT Identifier: None

Filename: law/lawutil/main_law.hxx

Description: This law has one law data argument, such as the curve_law, wire_law, and surface_law. These are parsed with one law tag. For example, the "CUR(EDGE1)" is followed by an edge. The edge is the law data. This permits an application to pass an ACIS entity and other class in the form of a law_data class 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 law_data *data;
```

This is a pointer to the data structure used as input to the unary data law.

Constructor:

```
public: unary_data_law::unary_data_law (
    law_data* in_sub_law    // pointer to law data
        = NULL
    );
```

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

Destructor:

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

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

Methods:

```
public: int unary_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: virtual law* unary_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: law_data* unary_data_law::fsub () const;
```

This returns the sublaw that is passed into this law. Only applications that create new laws that have parts of old laws should use this method. An example of this is the simplifier; the law “abs(x^2)” simplifies to “x^2”, where “x^2” is the part of the old law used in the new law. If the sublaw is to be used elsewhere, the add method should called.

```
public: static int unary_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 unary_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 unary_data_law*
    unary_data_law::make_one (
    law_data* in_data        // pointer to law data
    ) 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 unary_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 and parsing. For a law to be saved and restored, it must have or inherit this method. This method is called by the == method, to see if two laws are the same.

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

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

```
public: int unary_data_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* unary_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 unary_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

unary_law

Class: Laws

Purpose: Provides methods and data for laws that have one sublaw.

Derivation: unary_law : law : ACIS_OBJECT : –

SAT Identifier: None

Filename: law/lawutil/main_law.hxx

Description: A unary law is a law with one argument. Most trigonometry functions are unary 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 *sub_law;
```

This is a pointer to a sublaw that the unary law is to act upon.

Constructor:

```
public: unary_law::unary_law (
    law* in_sub_law          // pointer to sublaw
    );
```

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

Destructor:

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

Do not call this destructor directly. Instead, call the virtual remove method, which decrements the use_count.

Methods:

```
public: int unary_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* unary_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* unary_law::fsub () const;
```

This returns the sublaw that is passed into this law. Only applications that create new laws that have parts of old laws should use this method. An example of this is the simplifier; the law “abs(x^2)” simplifies to “x^2”, where “x^2” is the part of the old law used in the new law. If the sublaw is to be used elsewhere, the add method should called.

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

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

```
public: logical unary_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 unary_law* unary_law::make_one (
    law* in_sub_law         // pointer to 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 unary_law have a make_one method. This is used by the parser and simplifier.

If *f1* is a law for "X^3" and *f2* is the law for "COS(any law)", then f2–>make_one(f1) returns a law which is "COS(X^3)".

```
public: int unary_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 unary_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* unary_law::set_domain (
    SPAinterval* new_domain, // new input domain
    logical set              // change domain
        = FALSE
    );
```

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

```
public: int unary_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* unary_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 unary_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 unary_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 unary_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

vector_law

Class: Laws, Geometric Analysis, SAT Save and Restore

Purpose: Combines one dimensional laws into a multi-dimensional law.

Derivation: vector_law : multiple_law : law : ACIS_OBJECT : –

SAT Identifier: "VEC"

Filename: law/lawutil/main_law.hxx

Description: The range of all of the sublaws must be one dimensional. The dimension of the range of the resulting law is the number of sublaws. The term "law" is the opposite of the "vector law", in the sense a law takes a multi-dimensional input and returns a single dimension.

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: None

Data:

None

Constructor:

```
public: vector_law::vector_law (
    law** in_sublaw,         // array of sublaws
                             // with 1
                             // dimensional ranges
    int in_dim               // size of array
    );
```

C++ constructor, creating a vector_law. Accepts an array of one dimensional range laws and creates a vector_law with an in_dim dimensional range.

```
public: vector_law::vector_law (
    SPApar_pos p             // parameter position
    );
```

C++ constructor, creating a vector_law. Accepts a SPApar_pos and creates a vector_law with a two-dimensional range. The constant method returns true for this law.

```
public: vector_law::vector_law (
    SPAposition p            // position
    );
```

C++ constructor, creating a vector_law. Accepts a SPAposition and creates a vector_law with a three-dimensional range. The constant method returns true for this law.

```
public: vector_law::vector_law (
    SPAunit_vector u         // vector
    );
```

C++ constructor, creating a vector_law. Accepts a vector and creates a vector_law with a three-dimensional range. The constant method returns true for this law.

```
public: vector_law::vector_law (
    SPAvector v             // vector
    );
```

C++ constructor, creating a vector_law. Accepts a vector and creates a vector_law with a three-dimensional range. The constant method returns true for this law.

Destructor:

None

Methods:

```
public: char const* vector_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* vector_law::deriv (
    int which               // variable to take
                            // derivative with
                            // respect to default
        = 0                 // 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 vector_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: void vector_law::evaluate_with_guess (
    double const* x,         // input value
    double* answer,          // answer
    double const* guess      // best guess
        = NULL
    ) const;
```

Evaluate, with a best guess as to the answer to minimize processing.

```
public: void vector_law::evaluate_with_side (
    double const* x,         // input value
    double* answer,          // answer
    int const* side          // left (-1) or right (1)
        = NULL               // default is 0
    ) const;
```

Select which side of the input value to evaluate from, then evaluate. If the input value is an endpoint, this can be important for numerical optimization.

```
public: static int vector_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 vector_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* vector_law::make_one (
    law** subs,             // array of sublaws
    int 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 vector_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* vector_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* vector_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 VEC.

```
public: int vector_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