Chapter 1.
Laws Component

Component: *Laws

The Laws Component (LAWS), in the law directory, provides symbolic representations of equations that are parsed in much the same way that equations are. They provide the ability to solve complex, global mathematical problems. Mathematical functions are used extensively in ACIS.

A law is represented internally by a tree of C++ classes that know their dimensions, how to evaluate themselves, and how to take their exact (symbolic) derivatives with respect to any combination of variables. In addition, law utility functions numerically integrate, differentiate, and find roots. Many questions can be answered by knowing where some combination of them is maximal or minimal.

Laws can be used to define geometry and to solve mathematical problems in solid modeling:

- The offset distance for a wire offset can use a law instead of a constant value.
- One-dimensional space can be mapped to three-dimensional space as a vector field on a curve to define the orientation of a swept surface along a nonplanar path.
- The location on a curve can be found where the radius of curvature is equal to a specified value.

Laws are functions from any finite dimensional Euclidean space to any finite dimensional Euclidean space. For example,

- An ACIS surface may be considered to be a function from two dimensional space to three dimensional space.
- An ACIS curve may be considered to be a function from one dimensional space to three dimensional space.
- An ACIS transformation may be considered to be a function from three dimensional space to three dimensional space.
- The radius of curvature of a curve may be considered to be a function from one dimensional space to one dimensional space.

Laws are parsed the same way that equations are. For example, the equation:

$f(x,y) = x^2+\cos(x)-\sin(y)$

becomes the law:

```
"X^2+COS(X)-SIN(Y)"
```

with a two-dimensional domain and a one-dimensional range.

Note *Discussions of ACIS laws often use the term* mathematical function *to differentiate this meaning of the word* function *from the use of the word function meaning a computer program routine (such as an API, a class method, a direct interface function, etc.).*

Laws can be implemented in two ways:

- Through direct interface to the classes.
- Through the APIs, which use law string parsing.

Refer to the online help for the most recent list of classes, APIs, and Scheme extensions related to laws.

Accessing Laws through Classes

Topic: *Laws

The base class law has seven subclasses that divide laws by how many sublaws they evolve and how they are parsed. Some laws use ACIS classes such as curves, wires, surfaces, or transformations as part of their definition. These laws are derived from either unary_data_law or from multiple_data_law. Other ACIS classes are wrapped by a class derived from the class law_data so that they have a more uniform interface.

The seven classes derived from base law are:

constant_law . . . . . Defines a data member that has a constant value.

identity_law . . . . . Defines the input variables to the other law classes. Variables in C++ are numbered starting at zero (0). Variables using law symbols are numbered starting at one (1). Any letter of the alphabet may be followed by a number when using law symbols. When followed by a number in law symbol format, the numbers begin at 1. The law symbols (A–Z)1, X, U, or T are equivalent to the zero identity. The variables X, Y, and Z are equivalent to the indices 0, 1, and 2 in C++. U and V are equivalent to 0 and 1 in C++. T is equivalent to 0 in C++.

unary_law . . . . . Defines mathematical functions that have one sublaw argument.

binary_law . . . . . Defines mathematical functions that have two sublaw arguments. Some binary laws are associative, meaning in the case of plus, *(A+B)+C=A+(B+C)*, and some are commutative, meaning in the case of times, *A*B=B*A*.

multiple_law . . . . . Defines mathematical functions that have multiple sublaw arguments and may be commutative, meaning *f(A,B,C)=f(B,C,A)*.

law_data . . . . . Defines a base class used to handle ACIS classes that behave like functions. The derived classes from law_data are wrapper classes for specific ACIS classes, such as curves, wires, surfaces, and transforms. The purpose of the wrapper law_data class is to permit some ACIS classes to be handled in the same way as normal law classes. The wrapper functions know how to evaluate the data from the ACIS classes, how to save and restore them, and how to parse and unparse them.

unary_data_law . . . . . Similar to unary_law but can accept as input an array of elements derived from the law_data class. Therefore, it can handle laws, curves, wires, transforms, or surfaces.

multiple_data_law . . . . . Similar to multiple_law but can accept as input an array of elements derived from the law_data class. Therefore, it can handle laws, curves, wires, transforms, or surfaces.

These seven classes specify how the input arguments are to be parsed. For the most part, these classes are not accessed directly unless you are deriving a new law that does not already exist. All other law classes, which cover simple arithmetic, advanced mathematics, trigonometry, hyperbolic functions, derivatives, etc., are derived from one of these seven law classes.

For example, plus_law is derived from binary_law. binary_law parses the two input arguments that plus_law then operates on. Likewise, cos_law is derived from unary_law. unary_law parses the single input argument on which cos_law then operates.

Defining New Law Classes

Topic: Laws

An application can create a new law class that is not already a part of ACIS. In fact, an application is more likely to derive its own law class than to derive its own entity class (from ENTITY). This section describes how an application would derive a law.

Note *When an application derives its own law or entity classes for use in a save file, it breaks the ACIS geometry bus, because only that ACIS application understands how to save and restore the new information. However, laws that are derived strictly for the purposes of analysis do not break the ACIS geometry bus.*

For a new law class to be handled properly, it must meet the following requirements.

1. The new law should be derived from one of the “big six” law classes, which establish rules for parsing: constant_law, unary_law, binary_law, multiple_law, unary_data_law, or multiple_data_law.

2. The id, isa, and type methods of the new law are required for all derived law classes, but use nearly identical code that the other law classes use.

3. The evaluate method of the new law specifies how the derived law evaluates itself. This is unique for each law.

4. The deriv method of the new law specifies how to take the derivative of the law. This is unique for each law. If this is not provided, then the approximate methods are inherited. The derivative method is inherited.

5. The string and symbol methods of the new law are simple routines required for parsing.

6. A static class instances of the new law, a static class instance of the law_list, and the make_one method of the new law are required to register the new law with the parser.

Evaluators

Topic: Laws

All derived law classes are required to have the evaluate and derivative methods. Laws derived from constant laws inherit these two methods.

When calling the evaluation methods, use the most specific evaluation method possible. The more specific evaluation methods perform checking on the laws. For example, evaluateV verifies that the law returns three values, else an error is issued.

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

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

```
void cos_law::evaluate(double *x,double *answer)
{
    answer[0]=cos(sub_law->evaluateS(x));
}
```

```
public: virtual int law::return_dim () 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.

```
int law::return_dim()
{
    return 1;
}
```

```
public: virtual int law::take_dim () const;
```

The take_dim tells how many values are read from the x argument array of the evaluate method. The default is 1. All derived law classes must have this method or inherit it.

```
int law::take_dim()
{
    return 1;
}
```

Inherited Evaluators

Topic: Laws

The remaining evaluators are usually inherited.

```
public: double law::eval (
    double x    // domain value to perform evaluation
    ) const;
```

Works on laws that take and return single values. This method evaluates the given law at the value x. All law classes inherit this method for convenience. It calls the main evaluate method to make sure that no more than one argument is taken or returned by calling take_dim and return_dim.

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

This method takes two pointers to memory that the caller is responsible for creating and freeing. The x argument tells where to evaluate the law. This can be more than one dimension. The answer argument returns the evaluation. This can be more than one dimension. x should be the size returned by the take_dim method, and answer should be the size returned by the return_dim method. All derived law classes must have this method or inherit it.

```
public: double law::evaluateC_R (
    complex_number c   // second number
    ) const;
```

Works on laws that have two-dimensional domains and one-dimensional ranges. This method evaluates the given law at the value c. All law classes inherit this method for convenience. It calls the main evaluate method and does some checking of take_dim and return_dim.

```
public: void law::evaluateDM (
    double const* x,   // pointer to values used in evaluation
    double* answer,    // multi-dimension answer range for
                       // evaluation
    int n              // number of derivatives
    ) const;
```

Works on laws that accept multiple domains and return multiple range values. This method evaluates the *n*th derivative of the given law at the value x. All law classes inherit this method for convenience. It calls the main evaluate method and does some checking of take_dim and return_dim.

```
public: double law::evaluateDM_R (
    double const* x,   // pointer to values used in evaluation
    int n              // number of derivatives
    ) const;
```

Works on laws that accept multiple domains and return a single range value. This method evaluates the *n*th derivative of the given law at the value x. All law classes inherit this method for convenience. It calls the main evaluate method and does some checking of take_dim and return_dim.

```
public: double law::evaluateDR_R (
    double x,  // input value
    int n      // number of derivatives
    ) const;
```

Works on laws that accept a real (single domain) and return a single range value. This method evaluates the *n*th derivative of the given law at the value x. All law classes inherit this method for convenience. It calls the main evaluate method and does some checking of take_dim and return_dim.

```
public: SPAvector law::evaluateDR_V (
    double x,   // domain value to perform evaluation
    int n       // number of derivatives
    );
```

Works on laws that accept a real (single domain) and return a vector (range value). This method evaluates the *n*th derivative of the given law at the value x. All law classes inherit this method for convenience. It calls the main evaluate method and does some checking of take_dim and return_dim.

```
public: SPAposition law::evaluateM_P (
    double const *x// pointer to values used in evaluation
    ) const;
```

Works on laws that accept a multiple domain and return a position (range value). This method evaluates the *n*th derivative of the given law at the value x. All law classes inherit this method for convenience. It calls the main evaluate method and does some checking of take_dim and return_dim.

```
public: SPApar_pos law::evaluateM_PP (
    double const *x// pointer to values used in evaluation
    ) const;
```

Works on laws that accept a multiple domain and return a parameter position (range value). This method evaluates the *n*th derivative of the given law at the value x. All law classes inherit this method for convenience. It calls the main evaluate method and does some checking of take_dim and return_dim.

```
public: double law::evaluateM_R (
    double const* x// pointer to values used in evaluation
    ) const;
```

Works on laws that accept a multiple domain and return a real (range value). This method evaluates the *n*th derivative of the given law at the value x. All law classes inherit this method for convenience. It calls the main evaluate method and does some checking of take_dim and return_dim.

```
public: SPAvector law::evaluateM_V (
    double const *x// domain value to perform evaluation
    ) const;
```

Works on laws that accept a multiple domain and return a vector (range value). This method evaluates the *n*th derivative of the given law at the value x. All law classes inherit this method for convenience. It calls the main evaluate method and does some checking of take_dim and return_dim.

```
public: SPAposition law::evaluateNV_P (
    SPAnvector const& nv   // pointer to n–dimension vector
    ) const;
```

Works on laws that accept an *n*-dimensional vector as a single argument (domain) and return a position (range value). This method evaluates the *n*th derivative of the given law at the value x. All law classes inherit this method for convenience. It calls the main evaluate method and does some checking of take_dim and return_dim.

```
public: double law::evaluateNV_R (
    SPAnvector const& nv   // pointer to numerical vector
    ) const;
```

Works on laws that accept an *n*-dimensional vector as a single argument (domain) and return a real (range value). This method evaluates the *n*th derivative of the given law at the value x. All law classes inherit this method for convenience. It calls the main evaluate method and does some checking of take_dim and return_dim.

```
public: SPAvector law::evaluatePP_V (
    SPApar_pos const& pp   // pointer to par_pos
    ) const;
```

Works on laws that accept a parameter position (domain) and return a vector (range value). This method evaluates the *n*th derivative of the given law at the value x. All law classes inherit this method for convenience. It calls the main evaluate method and does some checking of take_dim and return_dim.

```
public: SPAposition law::evaluateP_P (
    SPAposition p  // pointer to input position
    ) const;
```

Works on laws that accept a position (domain) and return a vector (range value). This method evaluates the *n*th derivative of the given law at the value x. All law classes inherit this method for convenience. It calls the main evaluate method and does some checking of take_dim and return_dim.

```
public: SPAvector law::evaluateR_V (
    double x    // pointer to values used in evaluation
    ) const;
```

Works on laws that accept a real (domain) and return a vector (range value). This method evaluates the *n*th derivative of the given law at the value x. All law classes inherit this method for convenience. It calls the main evaluate method and does some checking of take_dim and return_dim.

deriv and derivative

The protected deriv method of the new law specifies how to take the derivative of the law. This is unique for each law. There are three ways in which ACIS laws take derivatives: symbolic, by calling ACIS class evaluators, or numeric.

For example, “COS(X^2)” has the symbolic derivative “–SIN(X^2)*2*X”. The symbolic derivatives are exact, fast, and the preferred method for taking derivatives.

A curve or surface law takes its derivative by calling ACIS class evaluators. These have procedural accuracy but cannot be further simplified.

In the case where more derivatives are required than a curve or surface provides, or, in the case where a law is so complicated that a symbolic derivative is unavailable, numeric approximations may be used. Numeric approximations are fourth order. Left– and right-handed derivatives are also provided.

Call the derivative method to access the derivative, which checks to see if a cached version of the derivative exists. If not, it calls the deriv method. The derivative method is inherited from the base law class.

C++ Example

```
law *cos_law::deriv(int w)
{
    // The expected derivative is:
    // cos(f)’ = –sin(f)*f’

    // The real work of taking the derivative is
    // performed on the following lines.
    // Remember that the derivative of the sublaw
    // also needs to be taken with respect to the
    // derivative variable w.
    // The expected derivative is:
    // cos(f)’ = –sin(f)*f’

    law *sub1 = new sin_law(sub_law);
    law *sub2 = new uminus_law(sub1);
    sub1–>remove();
    law *fprime=sub_law–>derivative(w);
    law *answer = new times_law(sub2,fprime);
    sub2–>remove();
    fprime–>remove();
    return answer;
}

protected law **dlaw;
```

The dlaw is used to hold cache derivatives with respect to each of its variables.

```
protected int dlaw_size;
```

The dlaw_size is used to tell how many cache derivatives are held by dlaw.

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

The deriv method returns a law pointer to the derivative of the given law with respect to the which variable. Variables in C++ are numbered starting at zero (0). Variables using law symbols are numbered starting at one (1). Any letter of the alphabet may be followed by a number when using law symbols. When followed by a number in law symbol format, the numbers begin at 1. The law symbols (A–Z)1, X, U, or T are equivalent to the zero identity. The variables X, Y, and Z are equivalent to the indices 0, 1, and 2 in C++. U and V are equivalent to 0 and 1 in C++. T is equivalent to 0 in C++. All derived law classes must have this method or inherit it.

```
law *plus_law::deriv(int w)
{
    // Create the law left derivative plus
    // right derivative
    law *sub1=left_law->derivative(w);
    law *sub2=right_law->derivative(w);
    law *answer = new plus_law(sub1,sub2);

    // Clean up memory
    sub1->remove();
    sub2->remove();
    return answer;
}
```

Parsing

Topic: Laws

Laws do not work by manipulating strings. However, given a string, one can create a law from it by calling the law parser, api_str_to_law. The strings passed in are documented in the online help as law symbols.

Applications deriving their own laws are required to have the representative static instance of the law, the law_list static instance, and the make_one method to register a new law with ACIS.

For the parser to work, each law class must overload these methods:

- string
- symbol
- make_one
- precedence

The new law's arguments in the static, representative instance of itself can be NULL. The representative instance is then passed as an argument into a static instance of the law_list. When the law_list instance is created, it attaches its law argument to a linked list of other laws.

All of the law symbols that the law parser recognizes are contained within the linked list of laws (law_list). Because an instance of each law is present in the law_list, the list can be traversed. The individual law_list and law_list methods can be called to find out what the parser should check for.

When a quoted character string (e.g., the law mathematical function) is passed to the api_str_to_law function, it first parses the string. The parser attempts to match the longest substrings within the law to what the symbol methods have registered. When a substring is matched, its parent class type informs the parser how to handle parenthesis groupings of elements preceding it and following it.

Once all of the relationships between the substrings in the law mathematical function have been determined, law class instances are created starting at the innermost sublaw, which is usually an identity_law type. The instances are created using the make_one method of the respective law type.

```
public: virtual char* law::string (
    law_symbol_type type   // type of law
        = DEFAULT          // standard ACIS type
    );
```

The string and symbol methods of the new law are simple routines required for parsing. The string method returns the law's name, which is used when parsing and when saving or restoring the law to or from a file. It is provided as a user-friendly interface to find out the definition of the current law. The string methods for an application deriving its own law will look similar to the following source code.

There are two symbol types associated with laws: the ACIS DEFAULT type and the MAPLE type, which is used by the popular symbolic–math software package. The string method returns the DEFAULT symbol unless the programmer tells it otherwise.

```
arcsin_law my_law;
char* str_default = my_law.string();            // str_default = ″ARCSIN″
char* str_also_default = my_law.string(DEFAULT);// str_also_default = ″ARCSIN″
char* str_maple = my_law.string(MAPLE);         // str_maple = ″arcsin″
```

C++ Example

```
char *cos_law::string(law_symbol_type type)
{
    // Derived classes must use the appropriate functions
    // ustring (unary), bstring (binary), cstring (multiple),
    // or number_string (constant).
    chr *answer;
    If (type == DEFAULT){
        answer = ustring(”COS”);
    } else if (type == ...) {
        answer = ustring(”...”);
        // answer is some other type
    }
    return answer;
}
```

The quoted return string for both string and symbol methods must agree. In this example, the return string is “COS” if the law_symbol_type is DEFAULT. This is the string that the parser is looking for and that an application can use as input to the api_str_to_law function.

The string method calls an additional function, depending on its parent’s type. For example, if its parent is a unary_law (or unary_data_law), then the ustring function is used. If its parent is a binary_law, then the bstring function is used. If its parent is a multiple_law (or multiple_data_law), then the cstring function is used. If its parent is a constant_law, then the number_string function is used with a double value instead of a string.

These additional string functions allow the sublaws of a given law to be viewed, and take care of leading and trailing parenthesis and commas during parsing for the law type.

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

The string and symbol methods of the new law are simple routines required for parsing. The symbol method for an application deriving its own law will look similar to the following source code.

C++ Example

```
char *cos_law::symbol(law_symbol_type type)
{
    If (type == DEFAULT){
        return”COS”;
    } else if (type == ...) {
        return ”...”;  // answer is some other type
    }
}
```

The quoted return string for both string and symbol methods must agree. In this example, the return string is “COS” if the law_symbol_type is DEFAULT. This is the string that the parser is looking for and that an application can use as input to the api_str_to_law function.

```
public: virtual int law::precedence ();
```

Precedence specifies the order in which laws should be evaluated. For example, multiplication is evaluated before addition. The following is a list of values that may be returned by your function. The higher order numbers for precedence are evaluated first. In the event of a tie, operators are evaluated from left to right. For example, “1–2–3” evaluates to “–4” and is equivalent to “(1–2)–3”.

```
#define PRECEDENCE_OR          1
#define PRECEDENCE_AND         2
#define PRECEDENCE_NOT         3
#define PRECEDENCE_EQUAL       4
#define PRECEDENCE_PLUS        5
#define PRECEDENCE_TIMES       6
#define PRECEDENCE_POWER       7
#define PRECEDENCE_FUNCTION    8
#define PRECEDENCE_CONSTANT    9
```

```
char make_one;
```

The make_one method of the new law is required to get the law to parse.

The following example shows the make_one method for the cos_law. It has a pointer to a sublaw (e.g., another law) as its input argument. The sublaw, however complex, is created first. Then the cos_law is created around the sublaw with a pointer to that sublaw.

C++ Example

```
unary_law *cos_law::make_one(law *sub)
{
    return new cos_law(sub);
}
```

Applications deriving their own laws are required to have the representative static instance of the law, the law_list static instance, and the make_one method.

Simplify

Topic: Laws

The law simplifier is accessed through the method simplify. When given a law, the simplify function returns a law that is equivalent to the given law but with fewer or simpler terms.

The goal of the simplifier is to improve performance; it is *not* to simplify a law as far as possible. Laws that require extensive analysis in order to simplify would defeat the purpose of improved performance. These are not simplified, and a copy of the original is returned.

```
protected law *slaw;
```

This data member of the law class holds a cache value of the most simplified version of the law.

```
protected int slevel;
```

This data member of the law class tells what level of simplification was used to create the law held by slaw. See simplify for a discussion of the levels.

```
public: law* law::simplify (
    int level      // level of simplification
        = 0,
    int show_work  // for debugging, shows simplification
        = 0
    );
```

This method calls the law simplifier, returns a simplified law, and caches its value in the slaw data member. Its two arguments are both optional. They are primarily set for experimentation and debugging. At present, only level 0 and 1 simplification are available. Level 0 is recommended for most uses.

When show_work argument is set to a positive integer, debugging information is provided. Because laws cache their simplified version, simplifying a law twice does not result in any debugging information on the second call. To trick the simplifier into not using its cache value, the level argument should be set to “666”.

```
public: virtual law* law::sub_simplify ();
```

This is a method that may be overloaded by classes to provide assistance to the simplifier. Most laws simply inherit a function that returns null.

For example, a curve law applied to a line or ellipse can return an equation in the form of a law. The sub_simplify method can then access private members of the curve law that the simplifier does not have access to.

C++ Example

```
law *norm_law::sub_simplify()
{
    // this_law was set to v*(dot(v,v))^(-0.5)
    // by the constructor
    this_law->add();
    return this_law;
}
```

Memory Management

Topic: Laws, *Memory Management

Laws are use counted. Therefore, their destructor is protected and should not be called directly. When an application destructs a law, the remove method should be called. For more efficient memory usage behind the scenes, the law classes use the add and remove methods which increment or decrement a use count, an indication of how many copies of the law are in existence.

For example, if *f* is a law for "X^2", it is an exponent_law with the arguments identity_law for x and the constant_law for 2.

When a new *df* law is created using the derivative method of *f*, the result "2*X" is a times_law with the arguments constant_law for 2 and the identity_law for *x*.

In the construction process of the *df* law, instead of creating new instances of the constant_law for 2 and the identity_law for *x*, the constructor calls the add methods, which increment their respective use counts. In this manner, memory is not taken up with duplicates of the constant_law for 2 or the identity_law for *x*.

If the remove method of either *f* or *df* is called, the remove methods of their sublaws are also called. The sublaws decrement the use count and only actually call the law::~law destructor if the use count goes to zero.

The following data members and methods are used for law memory management.

```
int use_count;
```

This data member keeps track of how many references to the law are in existence. When it reaches zero (0) through subsequent calls to remove, then the protected destructor is called.

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

This is the protected destructor. It is not called directly. In addition to the law constructor and the destructors of the main law class, a derived law may also have a destructor. All of destructors are called by remove when the use count reaches zero.

The destructor of derived classes should be a public method.

```
public static int how_many_laws;
```

This is a static data member of the law class that keeps track of how many law are in existence at any given time. It is used to check for memory leaks.

```
public: void law::add ();
```

This is not usually called by an application directly. An application calls this method to preserve a copy of this law. It is also called by the law constructors for the law being constructed, as well as for all of its sublaws. It increments the use_count.

```
public: void law::remove ();
```

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

```
public: char* law::string_and_data (
    law_data** *ld,        // array of law data
    int* size,             // size of array
    law_symbol_type type   // type of law symbol
        = DEFAULT          // standard ACIS type
    );
```

This method is inherited by all classes. Given a law, this returns an array of law_data pointers, the size of the array, and its string in the type specified. This function is used by save to create the string and the array of law_data pointers that defines the law.

```
void save();
```

This method saves the law symbol string for the given law in the DEFAULT law_symbol_type format to the save file (SAT or SAB). This saves not only the identifier string, but also all associated law_data information.

```
law *restore();
```

This function returns a law created from the law symbols in the DEFAULT law_symbol_type format that were saved to a save file. This restores not only the identifier string, but also all associated law_data information. The result is returned as a pointer to a law.

Type Identification

Topic: Laws

There are two ways of determining the type of a law. The first way identifies the class at the top of the expression tree. The second way tells what type of algebraic structure the law has.

For example, a law may be constant without being derived from the constant law class. For this reason, it is almost always best to identify a law by calling zero, integer, constant, linear, or polynomial methods. These methods set the type_flag and not_type_flag so that subsequent calls have faster performance.

Class identification is best done with the isa method. The following code fragment checks to see if my_law is a constant_law or a law derived from constant_law, such as the pi_law. The law mathematical function “5*4” is an example of a times_law that is constant. It would not pass the if statement in the following code fragment.

C++ Example

```
Law *my_law;
...
if (my_law->isa(constant_law::id())
{
    ...
}
```

Because isa only checks for class type, it is used only in cases where laws are being symbolically manipulated as in the simplifier. It should not be used to check for an algebraic property.

The id, isa, and type methods of the new law are required for all derived law classes, but include nearly identical code for all law classes.

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

This data member of the law class caches the results of previous inquiries as to the algebraic type of a law. The following is a list of allowed values:

```
#define LAW_TYPE_UNKNOWN       0
#define LAW_TYPE_ZERO          1
#define LAW_TYPE_INTEGER       2
#define LAW_TYPE_CONSTANT      3
#define LAW_TYPE_LINEAR        4
#define LAW_TYPE_POLYNOMIAL    5
#define LAW_TYPE_G_INFINITY    6
#define LAW_TYPE_G_5           7
#define LAW_TYPE_G_4           8
#define LAW_TYPE_G_3           9
#define LAW_TYPE_G_2           10
#define LAW_TYPE_G_1           11
#define LAW_TYPE_CONTINUOUS    12
#define LAW_TYPE_BOUNDED       13
#define LAW_TYPE_RATIONAL      14
```

For example, “COS(X+7)^2 + SIN(X+7)^2” is the constant 1. A call to the method constant returns true and sets the type_flag to LAW_TYPE_CONSTANT. Subsequent calls to the constant method would only need to reference this type_flag.

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

The id method’s purpose is to return a unique integer identification for the class. It gets this by calling new_law_id the first time id is called. These identifiers are used to pass values to the isa function when it is called to perform class identification. Every law class must have this method.

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

The type method's purpose is only to return its identification. The type method is only used internally to overload the == operator for laws. Every law class must have this method.

```
int cos_law::type()
{
    return id();
}
```

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

The isa method returns true or false depending upon whether or not this law or any of its ancestors have this same identification as the one passed in. For example, a plus_law returns true if the identification (id) for a plus_law, binary_law, or law is passed to it.

The source code from the cos_law for these methods could be used as shown, with the exception of the class name and the parent class unary_law::isa method used in the isa method. The derived class's isa method should use the isa method from its parent class.

The following code fragment is nearly identical for all law classes, except that the parent needs to be changed.

```
logical cos_law::isa(int t)
{
    // Derived classes should use the proper parent class
    // isa method, such as "binary_law::isa(t)", etc.
    return (t==id() || unary_law::isa(t));
}
```

```
public: logical law::zero (
    double tol     // tolerance to use in determination
        = SPAresabs // use SPAresabs by default
    );
```

The zero method returns true if the law is equivalent to zero. The law mathematical function "vec(0, 0, 0)" is considered zero by this method.

```
public: logical law::integer ();
```

The integer method returns true if the law is equivalent to a constant_law which returns an integer.

```
public: logical law::constant ();
```

The constant method returns true if its evaluate method returns the same value for all inputs. The law mathematical function "vec(0, 0, 0)" is considered constant by this method.

```
public: logical law::linear ();
```

The linear method returns true if the derivative method returns the same value for all inputs.

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

The polynomial method returns a law polynomial class which contains the coefficients of a polynomial and a law that it is a polynomial with respect to. The argument in may be null. If the in argument is not null, then the law polynomial class that is returned contains the coefficients of a polynomial with respect to the in law.

Operators and Properties

Topic: Laws

The == and != methods are used to identify whether a law is identical to another law. These methods use the same method. All three need to be defined or inherited by all derived classes.

```
public: int law::operator== (
    law& inlaw // law to compare this to
    );
```

The == method determines whether this law class is equivalent to the given inlaw class. This is inherited by all laws and calls the same method which the major law subclasses override. This is used for simplification.

```
public: int law::operator!= (
    law& inlaw // law to compare this to
    );
```

The != method determines whether this law class is not equivalent to the given inlaw class. This is inherited by all laws and calls the same method which the major law subclasses override. This is used for simplification.

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

The same method should not be called directly by the application. This is used for simplification. 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: virtual law_domain* law::domain ();
```

The domain method returns a law_domain class that contains information that defines the domain of the law. See the law_domain class in the online documentation for a further description.

```
public: virtual int law::singularities (
    double** where,// where discontinuity exist
    int** type     // type of discontinuity
    );
```

The singularities method returns true if any singularities exist for the law. It also returns an array of doubles which specify where singularities are, and an array of integers which specify the type of singularity. This is only implemented at this time for one dimensional laws.

The type is 0 if there is a discontinuity, 1 if the discontinuity is in the 1st derivative, and any integer *n* if the discontinuity is in the *n*th derivative. –1 means that it is not defined.

Accessing Laws through String Parsing

Topic: *Laws

Although accessing the law class methods directly is possible, it is much more likely that the APIs and law string parsing will be employed. This technique is easier and more straightforward to implement.

A law mathematical function is a character string enclosed in quotation marks. The law symbols used in the mathematical function are very similar to common mathematical notation and to the adaptation of mathematical notation for use in computers. The valid syntax for the character strings are given in the law symbol templates. The law mathematical functions support nesting of law symbols.

Once the character string for the law mathematical function has been created, it is passed to the api_str_to_law API along with a pointer to an output law, an array of law data, and the size of the law data array. The API parses the character string, creates the required output law class instances, and returns the address for the top-level output law class instance.

Note *The* api_str_to_law *function does not do any type checking to make sure that the law string tags agree with their* law_data *arguments.*

The disadvantage of using character strings with law symbols is that it takes more processing time to parse the strings. The advantage of using them is that they are simpler than using the law class constructors directly. The creation of supporting law class instances is managed behind the scenes.

In the following example, my_law is an instance of the law class which holds the mathematical function "x^2y^2". The law symbol syntax for my_law is "(x^2) * (y^2)". Extra spaces are ignored, such as those setting off the "*" character for times. All derived laws obey standard precedence for mathematical evaluation. Because exponents have precedence over multiplication, my_law could have been written without the parenthesis as "x^2*y^2".

Example 1-1 is a C++ code fragment that calculates the first and second partial derivatives with respect to *x* and *y* of the mathematical function "x^2y^2", and then evaluates these for differing values of *x* and *y*. The partial derivatives are shown in Figure 1-1.

C++ Example

```
// Create special law:
// my_law is x squared times y squared ; (x^2) * (y^2)
char my_function_string[50];
strcpy(my_function_string, "(x^2)*(y^2)");
double my_input[2];

// Convert a character string into law mathematical
// function class
law *my_law;
api_str_to_law( my_function_string, &my_law, 0, NULL);

// Create partial derivatives
// 1st partial derivative wrt x = (2*(y^2)*x)
// my_law->evaluateD( my_input, 1); // acquired in another way
// 1st partial derivative wrt y = (2*(x^2)*y)
law *dy = my_law->derivative(1);
// 2nd partial derivative wrt x = (2*(y^2))
// my_law->evaluateD( my_input, 2); // acquired in another way
// 2nd partial derivative wrt y = (2*(x^2))
law *ddy = dy->derivative(1);
// partial wrt x, partial wrt y = 4*x*y
law *dxdy = dy->derivative(0);

// Use new law instance to evaluate function
for (int x=0; x<10; x++) {
    for (int y=0; y<10; y++) {
        printf("input are: %d %d /t", x, y);
        my_input[0] = x;
        my_input[1] = y;

        // Evaluate original function
        // my_law = (x^2) * (y^2)
        // This is built into the original function.
        my_answer = my_law->evaluateS(my_input);
        printf("output is: %lf /t", my_answer);

        // Evaluate 1st partial derivative wrt x = (2*(y^2)*x)
        // This is built into the original function.
        my_answer = my_law->evaluateD(my_input, 1);
        printf("1st derivative wrt x is: %lf /t", my_answer);
```

```
        // Evaluate 1st partial derivative wrt y = (2*(x^2)*y)
        my_answer = dy->evaluateS(my_input);
        printf("1st partial wrt y is: %lf /t", my_answer);

        // Evaluate 2nd partial derivative wrt x = (2*(y^2))
        // This is built into the original function.
        my_answer = my_law->evaluateD(my_input, 2);
        printf("2nd partial wrt x is: %lf /t", my_answer);

        // Evaluate 2nd partial derivative wrt y = (2*(x^2))
        my_answer = ddy->evaluateS(my_input);
        printf("2nd partial wrt y is: %lf /t", my_answer);

        // Evaluate partial wrt x, partial wrt y = 4*x*y
        // This is built into the derived function.
        my_answer = dxdy->evaluateS(my_input);
        printf("partial wrt x, partial wrt y = %lf /n",
            my_answer);
    }
}

// Clean up memory usage
my_law->remove();
dy->remove();
ddy->remove();
dxdy->remove();
```

Example 1-1. API and Law Classes for Derivatives

$$f = x^2 y^2$$

$$\frac{\delta f}{\delta x} = 2 y^2 x$$

$$\frac{\delta f}{\delta y} = 2 x^2 y$$

$$\frac{\delta f}{\delta x \, \delta x} = 2 y^2$$

$$\frac{\delta f}{\delta y \, \delta y} = 2 x^2$$

$$\frac{\delta f}{\delta x \, \delta y} = 4 x y$$

Figure 1-1. Partial Derivatives of x^2y^2

The low-level implementation of laws in Scheme uses APIs and law string parsing. Laws can be used independently of ACIS to perform mathematical calculations. They can also be used within ACIS to find geometric information not otherwise so easy to obtain, such as minimum and maximum. Laws can be used to help define geometry in wire offsetting and sweeping. In Scheme, these are the wire–body:offset and sweep:law extensions.

Passing Simple Input into Laws

Topic: *Laws

Some law classes use the identity_law to acquire input. This is why first a law class instance *x* is created of the identity_law, and why this law instance is then used in the creation of subsequent laws. The integers passed to the identity_law at creation specify the index of its argument from an input list given in a later operation. The index in C++ starts at zero.

Consider the following code fragment:

```
law *x = new identity_law(8);
law *my_cos = new cos_law(x);
```

This code fragment creates an identity_law instance with an index of 8. Then it creates a cos_law class instance. Assume that my_input_array is a list of values. When one of cos_law's methods is evoked, such as cos_law–>evaluateS(my_input_array), it requires that my_input_array have at least 9 items (because indexing starts at 0); the ninth item is used as input to calculate the cosine.

This is the technique used to create more complex laws that have multiple input tags. If the C++ law classes are used directly, then the index into any later input list needs to be known at the creation of the class.

When working with the parsed strings from the law symbols, note the following:

- "x", "a1", and identity_law(0) represent the same index into the input list
- "y", "a2", and identity_law(1) represent the same index into the input list
- "z", "a3", and identity_law(2) represent the same index into the input list
- "a#" and identity_law(#–1) represent the same index into the input list

Example 1-2 is a code fragment that illustrates passing input into C++ law classes.

Note *Trigonometry law symbols assume that the input argument is in radians. Law symbols relating to curves require edges or coedges, because these are bounded. In other words, construction geometry curves are not supported.*

C++ Example

```
law *x = new identity_law(0);
law *y = new identity_law(1);
law *z = new identity_law(2);
law *a1 = new identity_law(0); // same input as x
law *a2 = new identity_law(1); // same input as y
law *a3 = new identity_law(2); // same input as z

// Create special law:
// my_law = cos(x) + sin(y) + tan(z);
law *my_cos = new cos_law(x);
law *my_sin = new sin_law(a2); // same as sin_law(y)
law *my_tan = new tan_law(z);
law *my_law2 = new plus_law( my_cos, my_sin);
law *my_law = new plus_law( my_law2, my_tan);

// Initial clean up of memory usage
x->remove();
y->remove();
z->remove();
a1->remove();
a2->remove();
a3->remove();
my_cos->remove();
my_sin->remove();
my_tan->remove();
my_law2->remove();

int my_input[3];
double my_answer;
```

```
// Use new law instance to evaluate at 1 radian increments
for (int x=0; x<10; x++) {
    for (int y=0; y<10; y++) {
        for (int z=0; z<10; z++) {
            printf(”input is: %d %d %d /t”, x, y, z);
            my_input[0] = x;
            my_input[1] = y;
            my_input[2] = z;
            // my_answer = cos(x) + sin(y) + tan(z);
            my_answer = my_law–>evaluateS(my_input);
            printf(”output is: %lf /t”, my_answer);
            my_answer = my_law–>evaluateD(my_input, 1);
            printf(”1st derivative wrt x is: %lf /t”, my_answer);
            my_answer = my_law–>evaluateD(my_input, 2);
            printf(”2nd derivative wrt x is: %lf /t”, my_answer);
            my_answer = my_law–>evaluateD(my_input, 3);
            printf(”3rd derivative wrt x is: %lf /t”, my_answer);
        }
    }
}

// Clean up memory usage
my_law–>remove();
```

Example 1-2. Passing Input to Law Classes

Using law_data to Pass Classes to Laws

Topic: *Laws

The unary_law, binary_law, and multiple_law classes are used if the application is passing only laws into a law class, in which case it becomes a pointer to a law or an array of pointers to laws, respectively. Numbers, positions, parametric positions, vectors, and vector fields, in addition to the law symbols, are passed as input to the api_str_to_law and become laws for these purposes.

On the other hand, the unary_data_law and multiple_data_law classes are used if the application is passing more complicated structures into a law class. These could be curves, wires, surfaces, transforms, or even laws. Instead of having a pointer to a law or an array of pointers to laws, the unary_data_law and multiple_data_law classes have a pointer to a law_data class or an array of pointers to law_data classes, respectively.

The base law_data class is used to handle the more complicated data structures from these special cases. The classes derived from law_data are wrapper classes for specific ACIS classes, such as curves, wires, surfaces, and transforms. The purpose of the wrapper law_data classes is to permit all ACIS classes to be handled in the same way as normal law classes. The wrapper functions know how to evaluate the data from the ACIS classes, how to save and restore them, and how to parse and unparse them.

curve_law_data

Topic: *Laws

curve_law_data defines a wrapper class for holding data from a curve that is passed into either unary_data_law or multiple_data_law.

C++ Example

```
// How to create a curve law from an edge.
EDGE *my_edge;
...
// Find the bounds of the curve.
double start = my_edge->start_param();
double end = my_edge->end_param();
// Get a copy of the curve
curve  *my_curve=my_edge->geometry()->trans_curve();
// Create a curve_law_data wrapper.
curve_law_data *my_c_law_data = new curve_law_data(
    my_curve, start, end);
// Create a curve law.
law *my_curve_law = new curve_law(my_c_law_data);
// clean up memory.
my_c_law_data->remove();
```

law_law_data

Topic: *Laws

law_law_data defines a wrapper class for holding the data from a law that is passed into either unary_data_law or multiple_data_law.

C++ Example

```
// How to create a law data from a law.
curve_law_data *my_c_law_data  // see example for curve_law_data
...
// Create a law for the integer 1.
law *my_law = new constant_law(1);
// Wrap the law in a law_data class.
law_law_data *my_l_law_data = new law_law_data(my_law);
// Clean up memory.
my_law->remove();
// Make an array of two law_datas.
law_data *my_l_law_data[2];
my_l_law_data[0] = my_c_law_data;
my_l_law_data[1] = my_l_law_data;
// Create a law that returns first derivative of
// curve.
// The constant law in my_law_data indicates how many
// derivatives to
// take.
law *my_dcurve_law = new dcurve_law(my_l_law_data, 2);
// Clean up memory.
my_c_law_data->remove();
my_l_law_data->remove();
```

path_law_data and surface_law_data

Topic: *Laws

path_law_data defines a wrapper class for holding data which is passed into either unary_data_law or multiple_data_law. Its main purpose is to handle curves and wires in a similar manner. curve_law_data and wire_law_data are derived from this class. This law is an abstract data class that is not created directly by a programmer.

surface_law_data defines a wrapper class for holding data from a surface that is passed into either unary_data_law or multiple_data_law.

C++ Example

```
// How to create a surface law from a surface.
FACE *my_face;
...
// Get the surface.
surface  *my_surface=my_face->geometry()->equation();
// Find the bounds of the surface.
SPAinterval u_range = my_surface->param_range_u();
SPAinterval v_range = my_surface->param_range_v();
// Create a surface_law_data wrapper.
surface_law_data *my_s_law_data = new surface_law_data(
    my_surface, u_range, v_range);
// Create a surface law.
law *my_surface_law = new surface_law(my_s_law_data);
// clean up memory.
my_s_law_data->remove();
```

transform_law_data

Topic: *Laws

transform_law_data defines a wrapper class for holding data from a transform that is passed into either unary_data_law or multiple_data_law.

C++ Example

```
// How to create a transform law from a transform.
// This is similar to the cure law data, but bounding
// information is contained within the wire class.
SPAtransf *my_transform;
law *my_law;
...
// Create a transform_law_data wrapper.
transform_law_data *my_w_law_data =
    new transform_law_data(my_transform);
// Create a rotate law.
// The rotate law will rotate the vectors
// returned by my_law with the my_transform.
law *my_rotate_law = new rotate_law(my_law, my_t_law_data);
// clean up memory.
my_t_law_data->remove();
```

wire_law_data

Topic: *Laws

wire_law_data defines a wrapper class for holding data from a wire that is passed into either unary_data_law or multiple_data_law.

C++ Example

```
// How to create a wire law from a wire.
// This is similar to the cure law data,
// but bounding information
// is contained within the wire class.
WIRE *my_wire;
...
// Create a wire_law_data wrapper.
wire_law_data *my_w_law_data = new wire_law_data(my_wire);
// Create a wire law.
law *my_wire_law = new wire_law(my_w_law_data);
// clean up memory.
my_w_law_data->remove();
```

Numerical Tools for Laws

Topic: *Laws, *Geometric Analysis

One of the advantages of laws is that they offer a uniform interface so that one set of numerical tools can be made for all cases. In addition to the evaluation methods, several numerical functions are available for laws. This includes:

- Finding global maximums and minimums
- Finding multidimensional local minimums
- Performing numerical integration
- Finding roots
- Performing numerical differentiation

Scheme Example

```
(define my_law (law "(20*sin(x)+x^2-10)*.2"))
;; my_law
(law:nroot my_law -10 10)
;; (-3.14655291240276 0.508720501792063 3.13209276518624
;; 5.23133412213485)
(define my_veclaw (law "vec(x,law,0)" my_law))
;; my_veclaw
(define fedge(edge:law my_veclaw -10 10))
;; fedge
(define x_axis(edge:law "vec(x,0,0)" -10 10))
;; x_axis
(define y_axis(edge:law "vec(0,x,0)" -10 20))
;; y_axis
```

Example 1-3. Finding Roots

Scheme Example

```
(define f (law "cos(x)"))
;; f
(define df (law:derivative f))
;; df
(define ddf (law:derivative df))
;; ddf
(define dddf (law:derivative ddf))
;; dddf
f
;; #[law "COS(X)"]
df
;; #[law "-SIN(X)"]
ddf
;; #[law "-COS(X)"]
dddf
;; #[law "SIN(X)"]


(define ndf(law "d(cos(x),x,1)"))
;; ndf
(define nddf(law:derivative ndf))
;; nddf
(define ndddf(law:derivative nddf))
;; ndddf
ndf
;; #[law "D(COS(X),X,1)"]
nddf
;; #[law "D(COS(X),X,2)"]
ndddf
;; #[law "D(COS(X),X,3)"]


(law:eval df 1)
;; -0.841470984807897
(law:eval ndf 1)
;; -0.841470984805174
(law:eval ddf 1)
;; -0.54030230586814
(law:eval nddf 1)
;; -0.540302277606634
(law:eval dddf 1)
;; 0.841470984807897
(law:eval ndddf 1)
;; 0.842631563290711
```

Mathematical Calculations

Topic: Laws, *Geometric Analysis

The following example defines my_law to be a law for the mathematical function $x+x^2-\cos(x)$. This law can be evaluated for a given value of *x* using the law:eval Scheme extension.

Note *Trigonometry law symbols assume that the input argument is in radians.*

Scheme Example

```
(define my_law (law "x+x^2-cos(x)"))
;; my_law
; my_law => #[law "(X+X^2)-COS(X)"]
; Evaluate the given law at 1.5 radians
(law:eval my_law 1.5)
;; 3.6792627983323
```

In addition to creating a law from a string, a law may also be created from a string and a list of laws and edges. The following example defines my_newlaw to be a law that returns the curvature of the curve in my_edge, plus the value of my_law.

Note *Law symbols relating to curves require edges or coedges, because these are bounded. In other words, construction geometry curves are not supported.*

Scheme Example

```
(define my_edge (edge:circular (position 0 0 0) 30 0 90))
;; my_edge
; my_edge => #[entity 2 1]
(define my_law (law "x+x^2-cos(x)"))
;; my_law
; my_law => #[law "(X+X^2)-COS(X)"]
(define my_newlaw (law "curC(edge1)+law2" my_edge my_law))
;; my_newlaw
; my_newlaw => #[law "CURC(EDGE1)+((X+X^2)-COS(X))"]
```

curC is a function that returns the curvature of a curve. Its argument in this case is “edge1”; the one (1) means that a curve should be the first input element. Indexing into argument lists always begins with one (1). law2 is the notation that means a law my_newlaw includes an existing law into its definition. The two (2) means that a law should be the second input element. The input list which follows the law definition contains my_edge, which has an underlying curve, and my_law as its first and second arguments, respectively. The curC law differs from other laws, such as cos, in that curC works on a data structure while cos works on a sublaw. Laws that work on data structures are called data laws.

A new law may also be created from another law by either taking the original law’s derivative or by simplifying the original law, as in the following example:

Scheme Example

```
(define my_dlaw (law:derivative "x^2+cos(x)"))
;; my_dlaw
; my_dlaw => #[law "2*X-SIN(X)"]
(define my_simplaw (law:simplify "x^2/x-3*x"))
;; my_simplaw
; my_simplaw => #[law "-(2*x)"]
```

Evaluation of Laws

Topic: Laws, *Geometric Analysis

There are five evaluate Scheme extensions for laws: law:eval, law:eval–vector, law:eval–position, law:nderivative and law:nintegrate.

law:eval, law:eval–vector, and law:eval–position are very similar in nature. law:eval returns a real or a list of reals. law:eval–vector requires that its input law be three dimensions and returns a gvector. law:eval–position returns either a position or a par–pos.

Scheme Example

```
(define my_law (law "x^2"))
;; my_law
; my_law => #[law "X^2"]
(law:eval my_law 3)
;; 9
(law:nderivative my_law 3 "x" 1)
;; 5.9999999999838
(law:nderivative my_law 3 "x" 2)
;; 1.9999997294066
```

law:nderivative in the previous example returns the numerical approximation of the first derivative of x^2 evaluated at 3. To get the exact value, evaluate the exact derivative, as seen next, which returns 6. The "n" in the command law:nderivative stands for "numerical". All law commands that contain an "n" return approximate results. All other law commands return exact results to machine precision.

Scheme Example

```
(law:eval (law:derivative "x^2") 3)
;; 6
(law:eval (law:derivative "x^2") 3)
;; 6
```

The third law evaluation extension for laws is law:nintegrate. This takes the numerical integral of a given law symbol over a specified range.

Scheme Example

```
; Define a law to integrate.
(define my_law (law "x + 1"))
;; my_law
; my_law => #[law "X+1"]
; Evaluate the law over the range 0 and 1.
(law:nintegrate my_law 0 1)
;; 1.5
```

Creating Geometry with Laws

Topic: *Laws

Using law_spl_sur and law_int_cur classes, the APIs api_curve_law, api_edge_law, and api_face_law create curves or surfaces from laws. Associated Scheme extensions are available for these functions. Laws are incorporated into wire offsetting and sweeping for creating geometry.

Creating Curves

Topic: Laws

An edge can be created that uses a law to define its path.

Scheme Example

```
(define my_law (law "(20*sin(x)+x^2-10)*.2"))
;; my_law
(law:nroot my_law -10 10)
;; (-3.14655291240276 0.508720501792063
;; 3.13209276518624 5.23133412213485)
(define my_veclaw (law "vec(x,law,0)" my_law))
;; my_veclaw
(define fedge(edge:law my_veclaw -10 10))
;; fedge
(define x_axis(edge:law "vec(x,0,0)" -10 10))
;; x_axis
(define y_axis(edge:law "vec(0,x,0)" -10 20))
;; y_axis
```

Creating Surfaces

Topic: Laws

Laws have the law_spl_sur class (derived from spl_sur) and the law_int_cur class (derived from int_cur). These two classes allow you to create a surface or a curve directly from the appropriately dimensioned law. These law curves or surfaces can be used in subsequent operations, such as sweeping, Booleans, etc.

Scheme Example

```
(define my_s1(face:law "vec(x,y,sin(x)*cos(y))" -10 10 -10 10))
;; my_s1
(define my_sb(sheet:face my_s1))
;; my_sb
(sheet:2d my_sb)
;; #[entity 3 1]

(define my_e1(edge:linear (position 7 7 -10)(position 7 -7 -10)))
;; my_e1
(define my_e2(edge:linear (position 7 -7 -10)
    (position -7 -7 -10)))
;; my_e2
(define my_e3(edge:linear (position -7 -7 -10)
    (position -7 7 -10)))
;; my_e3
(define my_e4(edge:linear (position -7 7 -10)(position 7 7 -10)))
;; my_e4

(define my_profile(wire-body (list my_e1 my_e2 my_e3 my_e4)))
;; my_profile
(define my_path(edge:linear (position 0 0 -10)(position 0 0 10)))
;; my_path
(define my_face(car(entity:faces my_sb)))
;; my_face
(define my_surf(surface:from-face my_face))
;; my_surf
(entity:delete my_sb)
;; ()
(define my_opt(sweep:options "to_face" my_surf))
;; my_opt
(define my_sweep(sweep:law my_profile my_path my_opt))
;; my_sweep
(entity:delete my_path)
;; ()
```

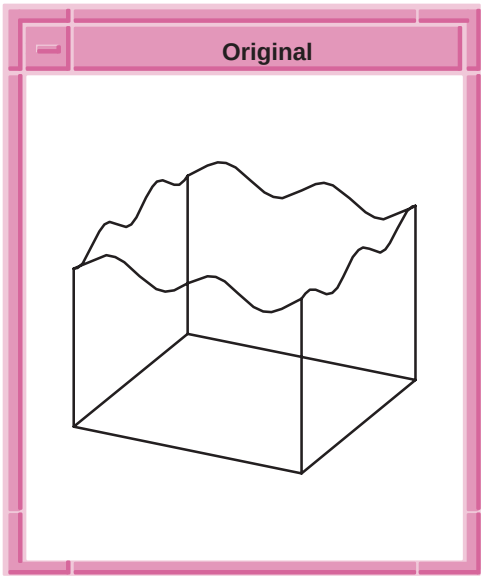

Figure 1-2. Law Surface

The following example illustrates how a face created with a law can be used in subsequent operations. First, a face is created with the law z=sin(x)*cos(y), then a profile is swept to the face, then a hemisphere is removed from the surface. The law surface is then extended with a local operation (move faces).

Scheme Example

```
; Create the face
(define s1(face:law "vec(x,y,sin(x)*cos(y))" -10 10 -10 10))
;; s1
; s1 => #[entity 2 1]
(define sb(sheet:face s1))
;; sb
; sb => #[entity 3 1]
(sheet:2d sb)
;; #[entity 3 1]
```

```
; Create a profile
(define my_edges (list
    (edge:linear (position 7 7 –10) (position 7 –7 –10))
    (edge:linear (position 7 –7 –10) (position –7 –7 –10))
    (edge:linear (position –7 –7 –10) (position –7 7 –10))
    (edge:linear (position –7 7 –10) (position 7 7 –10))))
;; my_edges
(define profile(wire–body my_edges))
;; profile

(define path(edge:linear (position 0 0 –10)(position 0 0 10)))
;; path
(define face(car(entity:faces sb)))
;; face
(define surf(surface:from–face face))
;; surf
(entity:delete sb)
;; ()

; Sweep
(define opt(sweep:options ”to_face” surf))
;; opt
(define sweep(sweep:law profile path opt))
;; sweep
; sweep => #[entity 8 1]
(entity:delete path)
;; ()
; Define a sphere and subtract it from the law surface
(define sp(solid:sphere (position 0 0 0) 5))
;; sp
(solid:subtract sweep sp)
;; #[entity 8 1]
; Pick the face on the right to extend the faces
(lop:move–faces (pick–face) (gvector 10 0 0))
;; #[entity 8 1]
```

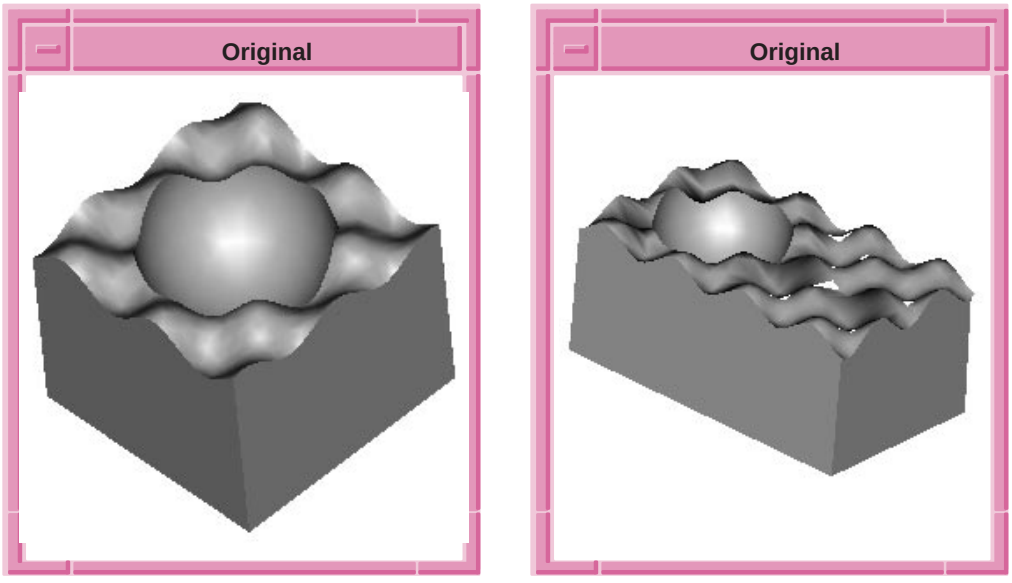

Figure 1-3. Scheme Example of Law Surface

Planar Wire Offsetting

Topic: Laws

Planar wire offsetting can accept laws as the offset distance.

Scheme Example

```
(define my_edge (edge:circular (position 0 0 0) 20))
;; my_edge
; my_edge => #[entity 2 1]
(define my_wirebody (wire-body my_edge))
;; my_wirebody
; my_wirebody => #[entity 3 1]

; Uses a law offset and no twist law.
(define my_offset (wire-body:offset my_wirebody
    "20+10*cos(x*10)"))
;; my_offset
; my_offset => #[entity 4 1]
```

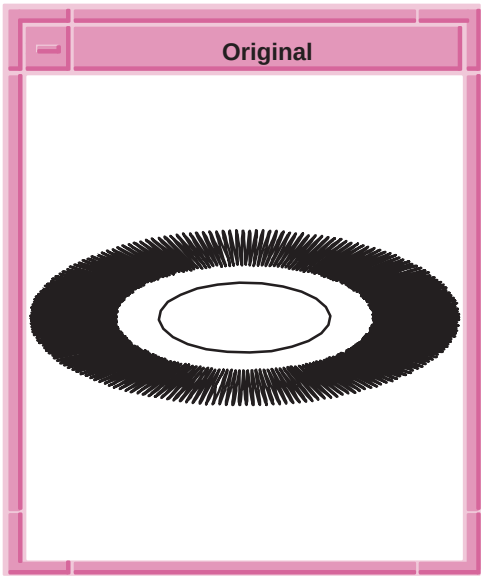

Figure 1-4. Planar Wire Offsetting

Nonplanar Wire Offsetting

Topic: Laws

Nonplanar wire offsetting has been built into wire–body:offset as an additional argument for the twist law.

Scheme Example

```
; Create edge 1.
(define my_edge1 (edge:circular (position 10 0 0) 10))
;; my_edge1 => #[entity 2 1]

; Create a wire-body from the edges.
(define my_wirebody (wire-body (list my_edge1)))
;; my_wirebody => #[entity 2 1]

; Offset a wire-body outside the original wire-body.
; Uses a constant (law) offset and a twist law.
(define my_spiral (wire-body:offset my_wirebody 5 "15*x"))
;; my_spiral => #[entity 3 1]
```

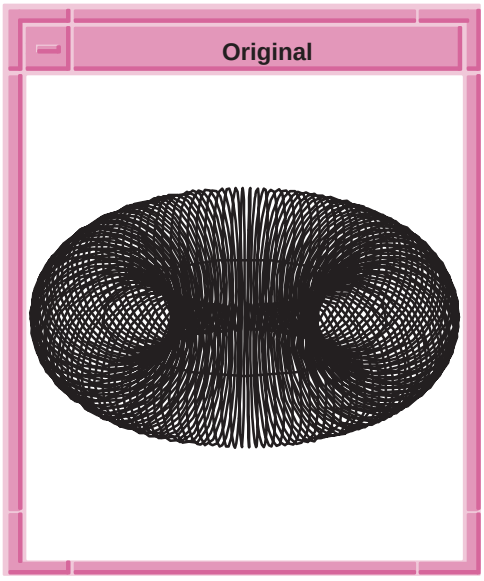

Figure 1-5. Nonplanar Wire Offsetting

Creating a Helix

Topic: Laws

Helixes can be created using api_edge_helix. This is part of the Constructors Component. It does not require that the helix be created by offsetting a line with a twist law.

Scheme Example

```
; edge:helix
; Create a helical edge.
(define axis_start1 (position 0 0 0))
;; axis_start1
(define axis_end1 (position 0 0 10))
;; axis_end1
(define start_dir1 (gvector 1 0 0))
;; start_dir1
(define radius1 5)
;; radius1
(define thread_distance1 2)
;; thread_distance1
(define helix1
    (edge:helix axis_start1
    axis_end1 start_dir1 radius1 thread_distance1))
```

```
;; helix1
; Create another helical edge.
(define helix2
    (edge:helix (position –20 0 0)
    (position –20 10 10) (gvector 0 1 0) 5 2 #f))
;; helix2
; And one more.
(define helix3
    (edge:helix (position 20 0 0)
    (position 20 0 10) (gvector 0 1 0) 5 2 #f))
;; helix3
; OUTPUT Example
```

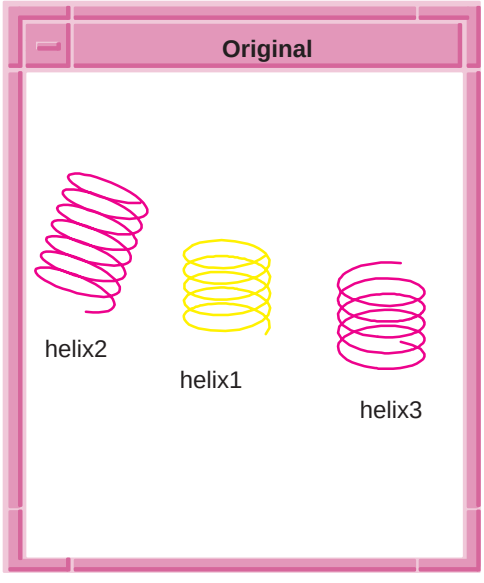

Figure 1-6. Helix Examples

Planar Sweeping

Topic: Laws

Sweeping with laws, such as the sweep:law Scheme extension, encompasses the functionality of many of the other sweep extensions. Sweeping along a planar path can accept a law as one of the arguments, such as the offset distance.

The following example sweeps a circular profile along a linear path. It uses the minimum rotation law by default for the orientation of the profile. However, as it sweeps, it uses a draft law to create a vase-like appearance.

Scheme Example

```
(option:set ”u_par” 10)
;; –1
(option:set ”v_par” 10)
;; –1
(option:set ”cone_par” #t)
;; #f
(option:set ”sil” #f)
;; #t
(define my_edge (edge:linear (position 0 0 0)
    (position 0 0 40)))
;; my_edge
; my_edge => #[entity 2 1]
(define my_path (wire–body my_edge))
;; my_path
; my_path => #[entity 3 1]
(define my_eprofile (edge:circular
    (position 0 0 0) 10))
;; my_eprofile
; my_eprofile => #[entity 4 1]
(define my_profile (wire–body my_eprofile))
;; my_profile
; my_profile => #[entity 5 1]
(define my_vase (sweep:law my_profile my_path
    (sweep:options ”draft_law” ”10*sin(x/2)”)))
;; my_vase
; my_vase => #[entity 6 1]
```

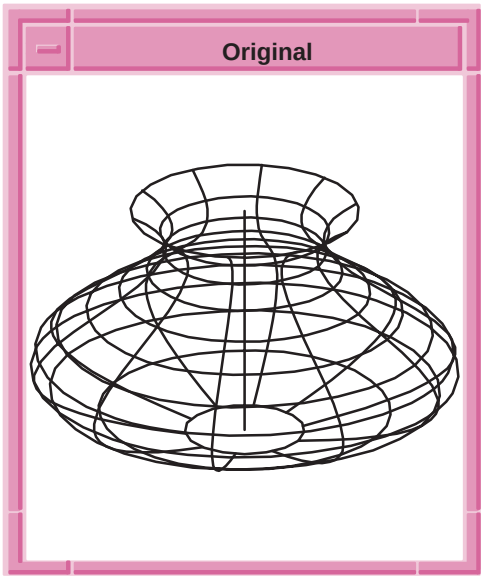

Figure 1-7. Sweeping a Vase

Nonplanar Sweeping

Topic: Laws

Sweeping with laws, such as the sweep:law Scheme extension, encompasses the functionality of many of the other sweep extensions. Sweeping is covered in more detail in another chapter of the ACIS documentation.

The important aspects of sweeping to note are:

- The profile to be swept needs to be a planar wire-body.
- The path can be nonplanar.
- The profile needs to be perpendicular to the path at the starting point of the path.
- The orientation of the profile as it is swept along a nonplanar path needs to be specified.

ACIS uses rails for orienting the profile along a nonplanar path. There are three main techniques for defining the rails.

Minimum Rotation . . . . . Requires that a direction for the first rail be specified. A vector field is then created along the path such that the rail vectors are perpendicular to the path and rotate as little as possible with respect to the neighboring rail vectors. This is the default.

Frenet ..... Creates a vector field whose rail vectors are perpendicular to the path and point in direction of curvature.

User-defined ..... Can be any valid law definition. The vector field that is created should have vector rails perpendicular to the path, but their direction with respect to the other rails is determined by the user-defined law.

When sweeping along a planar path using minimum rotation, the rail vectors all point in the same direction. If the path is nonplanar, the resulting body exhibits torque elements.

The following example sweeps a surface along a nonplanar path. It uses the minimum rotation rail law to specify how the surface orients itself on the path.

Scheme Example

```
; sweep:law
; Create a sweep path from points
(define my_plist (list (position 0 0 0)
    (position 10 0 0)(position 10 10 0)
    (position 10 10 10)))
;; my_plist
(define my_start (gvector 1 0 0))
;; my_start
; my_start => #[gvector 1 0 0]
(define my_end (gvector 0 0 1))
;; my_end
; my_end => #[gvector 0 0 1]
(define my_path (edge:spline my_plist
    my_start my_end))
;; my_path
; my_path => #[entity 2 1]

(define my_law (law "cur(edge1)" my_path))
;; my_law
; my_law => #[law "CUR(EDGE1)"]
(define my_rail (law "minrot(law1,vec(0,-1,0))" my_law))
;; my_rail
; my_rail => #[law "MINROT(CUR(EDGE1),VEC(0,-1,0),0,30)"]
(define my_edge1 (edge:linear (position 0 1 1)
    (position 0 1 -1)))
```

```
;; my_edge1
; my_edge1 => #[entity 3 1]
(define my_edge2 (edge:linear (position 0 1 –1)
    (position 0 –1 –1)))
;; my_edge2
; m_edge2 => #[entity 4 1]
(define my_edge3 (edge:linear (position 0 –1 –1)
    (position 0 –1 1)))

;; my_edge3
; my_edge3 => #[entity 5 1]
(define my_edge4 (edge:linear (position 0 –1 1)
    (position 0 1 1)))
;; my_edge4
; my–edge4 => #[entity 6 1]

(define my_profile (wire–body
    (list my_edge1 my_edge2 my_edge3 my_edge4)))
;; my_profile
; my_profile => #[entity 7 1]
(define my_sweep (sweep:law my_profile my_path
    (sweep:options ”rail_law” my_rail)))
;; my_sweep
; my_sweep => #[entity 7 1]
```

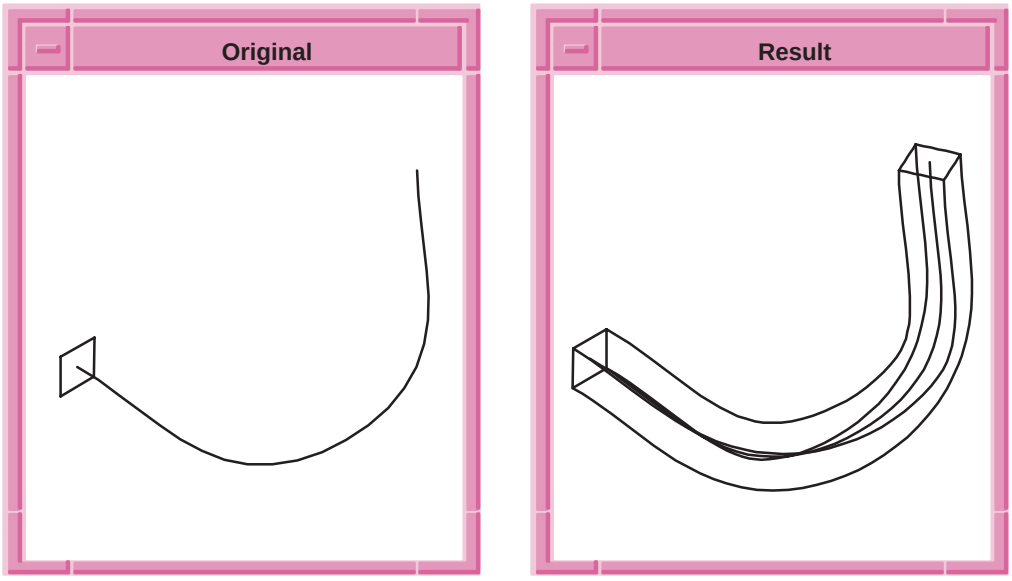

Figure 1-8. Sweep with Laws

Sweeping a Screw–Thread

Topic: Laws

The Scheme example shows how to create a screw using sweeping with laws. One of the first steps is to create a center line for the screw. The center line is used by the wire–body:offset command with a twist law option to create a helix. This helix then becomes the sweep path.

A profile for sweeping is then created. The profile is a rectangle, resulting in an auger type thread. If the profile were a diamond or a triangle shape with the hypotenuse facing away from the center line, the result would be a v–shaped thread cut.

The sweep options include a rail law to specify how the profile is to be oriented as it is swept along a nonplanar path.

The profile is then swept along the helix path using the sweep options already established. After sweeping is completed, a cylinder blank is created. The threads can then be subtracted from the cylinder, leaving the auger thread screw.

Scheme Example

```
; sweep:law
; Sweeping is much faster when the silhouette line calculation
; isn't always performed.
(option:set "sil" #f)
;; #t
; Create a center line around which to create the helix.
(define wline(wire–body (edge:linear (position 0 0 0)
    (position 0 (law:eval "8*pi") 0))))
;; wline
(define path(wire–body:offset wline 5 "x"))
;; path
; Create a profile to be swept.
(define p1(edge:linear(position 5 0 0)(position 8 0 0)))
;; p1
(define p2(edge:linear(position 8 0 0)(position 8 3 0)))
;; p2
(define p3(edge:linear(position 8 3 0)(position 5 3 0)))
;; p3
(define p4(edge:linear(position 5 3 0)(position 5 0 0)))
;; p4
(define profile (wire–body (list p1 p2 p3 p4)))
;; profile; profile => #[entity 9 1]
```

```
; Create a rail law to specify the orientation of the profile
; as it is swept.
(define my_rail (law "vec(-cos(x),0,sin(x))"))
;; my_rail
; Set up the sweep options.
(define my_sweep_ops (sweep:options "rail_law" my_rail))
;; my_weep_ops
; Perform the sweeping using the profile, the helix path, and
; the sweep options that specify the rail law.
(define my_sweep (sweep:law profile path my_sweep_ops))
;; my_sweep
; my_sweep => #[entity 10 1]
; Create a cylinder blank.
(define my_cylinder (solid:cylinder (position 0 3 0)
    (position 0 (law:eval "8*pi") 0) 7))
;; my_cylinder
; my_cylinder => #[entity 11 1]
; When the screw threads are subtracted (cut out) of the
; cylinder blank, the result is a screw.
(define my_screw (solid:subtract my_cylinder my_sweep))
;; my_screw
; my_screw => #[entity 10 1]
; Render the screw to see the image as a solid.
(render)
```

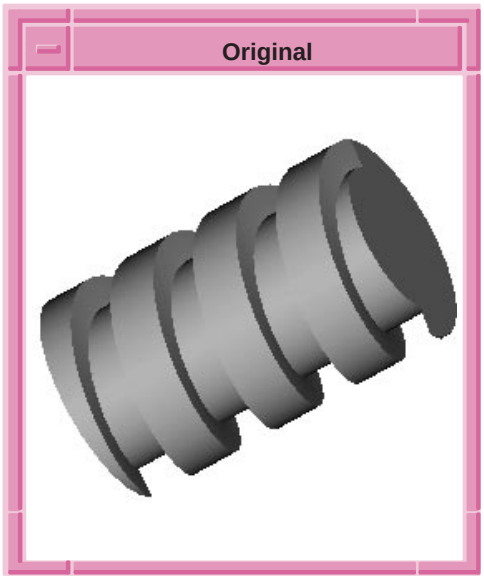

Figure 1-9. Sweeping a Screw

Vector Fields – Hedgehogs

Topic: *Laws

Laws are functions that may go from any dimension to any dimension. For example, a law used to make a surface may go from 2D-to-3D.

2D-to-3D laws may also be viewed as vector fields on a surface. (1D-to-3D laws may be viewed as vector fields on a curve and 3D-to-3D laws may be viewed as a vector field in space.) These vector fields may be illustrated as *hedgehog* markings.

The key to being able to sweep along a nonplanar path is the establishment of rails along the path. The rails serve to align the profile correctly to the path as it is swept. Rails are defined using laws, so can be arbitrarily complex to achieve unique results. The array of rails permits piecewise planar sweeping to be handled in an expected, predictable manner. The api_make_rails function and the law:make–rails Scheme extension perform automatic creation of the rail law array.

Scheme AIDE can be used to view the default rail alignment and to test user-defined rails. The law:make–rails Scheme extension can be used with law:hedgehog to see the the rails along a path. The *hedgehog* markings can also be used to display a vector field, which is similar to the rails for sweeping.

Scheme Example

```
; Create a 1D Hedgehog
(define my_edge(edge:law ”vec(cos(x),sin(x),x/10)” 0 500))
;; my_edge
(define my_base_law(law ”vec(cos(x),sin(x),x/10)”))
;; my_base_law
(define my_hair_law(law ”vec(cos(x),sin(x),0)”))
;; my_hair_law
(define h1(law:hedgehog my_hair_law my_base_law 0 50 200 ))
;; h1

; Create a 2D Hedgehog
(define my_face(face:law ”vec(x,y,cos(x)*sin(y))” –5 5 –5 5))
;; my_face
(define my_base_law(law ”vec(x,y,cos(x)*sin(y))”))
;; my_base_law
(define dx(law:derivative my_base_law))
;; dx
(define dy(law:derivative my_base_law ”y”))
;; dy
(define norm(law ”cross(law1,law2)” dx dy))
;; norm
(define h1(law:hedgehog norm my_base_law –5 5 –5 5 20 ))
;; h1

; Create a 3D Hedgehog
(define my_base_law(law ”vec(x,y,z)”))
;; my_base_law
(define my_hair_law(law ”norm(vec(x,y,z))”))
;; my_hair_law
(define h1
    (law:hedgehog my_hair_law my_base_law –10 10 –10 10 –10 10))
;; h1
```

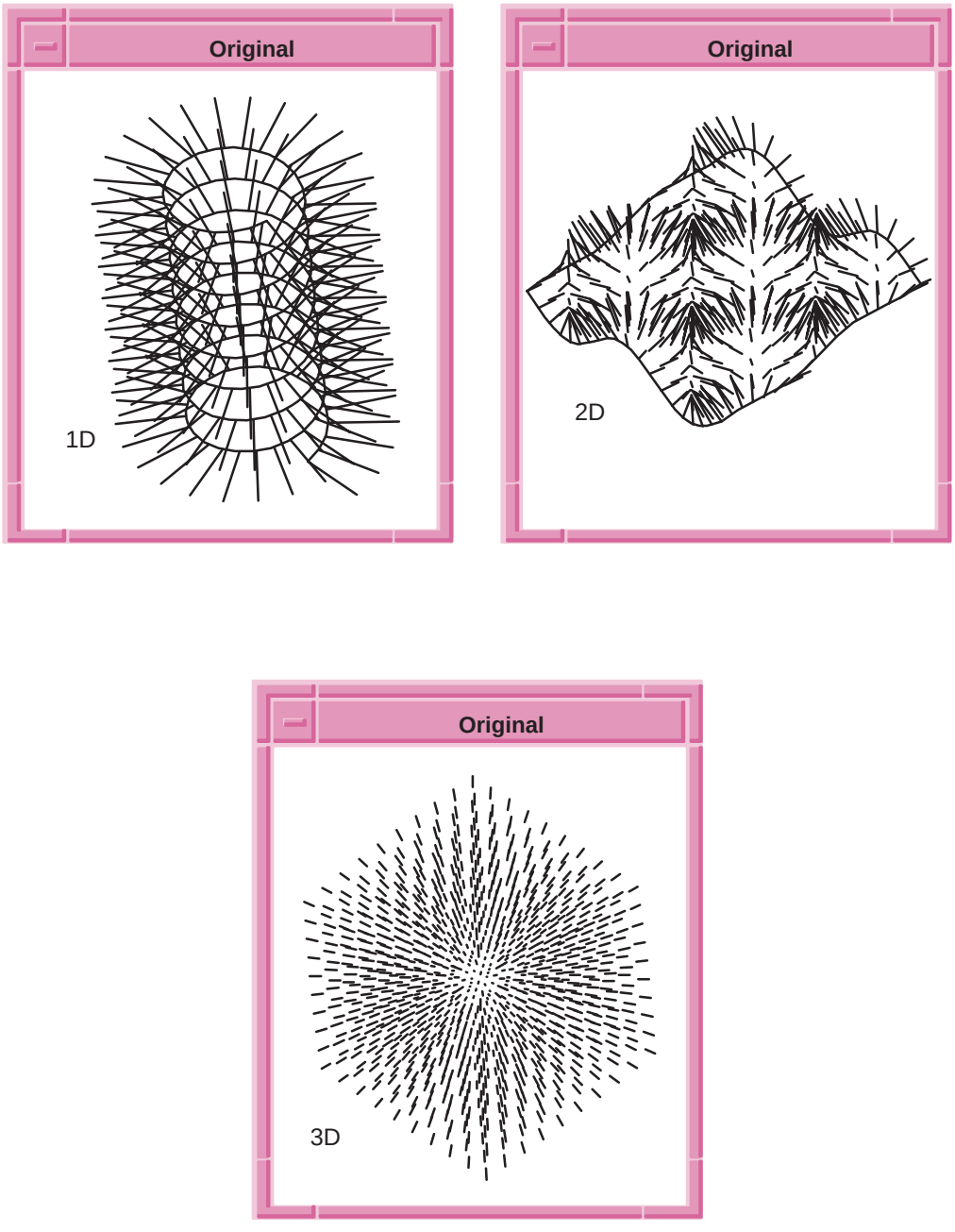

Figure 1-10. Hedgehog Examples

Using Laws for Analysis

Topic: *Laws

Laws can be used to answer questions about entities, such as locating maxima or minima of a curve, what is the closest point between two non-intersecting lines, or where all the roots are.

Finding Singularities

Topic: Laws, *Geometric Analysis

The following example shows how to use laws to find the minimum point on a curve.

Scheme Example

```
; Create the points for a test curve.
(define my_plist (list
    (position 0 0 0)(position 20 20 20)
    (position –20 20 20)(position 0 0 0)))
;; my_plist
; my_plist => (#[position 0 0 0] #[position 20 20 20]
;   #[position –20 20 20] #[position 0 0 0])
(define my_start (gvector 1 1 1))
;; my_start
; my_start => #[gvector 1 1 1]
(define my_end (gvector 1 1 1))
;; my_end
; my_end => #[gvector 1 1 1]
(define my_testcur (edge:spline my_plist my_start my_end))
;; my_testcur
; my_testcur => #[entity 2 1]

; Create a curve law.
(define my_curlaw
    (law "map(term(cur(edge1),1),edge1)" my_testcur))
;; my_curlaw
; my_curlaw => #[law "MAP(TERM(CUR(EDGE1),1),EDGE2)"]

; Find its minimum.
(define my_min (law:nmin my_curlaw 0 1))
;; my_min
; my_min => 0.713060255033984
```

```
; Find the point and mark it.
(define my_minpoint (dl-item:point
    (curve:eval-pos my_testcur my_min)))
;; my_minpoint
; my_minpoint => #[dl-item 22F733F8]
(dl-item:display my_minpoint)
;; ()
```

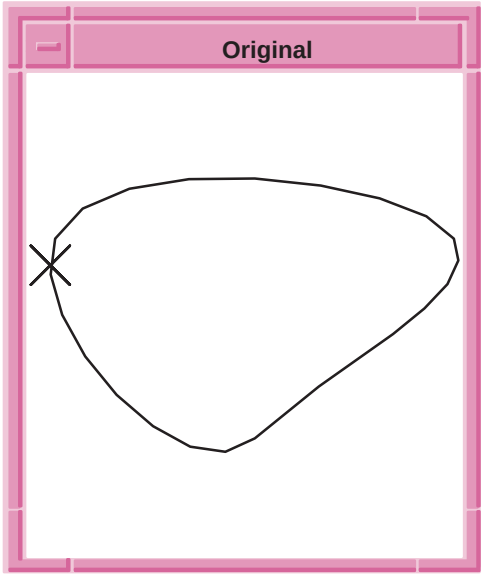

Figure 1-11. Minimum Point on a Test Curve

The following example first creates an edge, called my_edge. Then it creates a law, called my_law, that is the composition of three laws. The “map” law symbol maps that parameter domain of my_edge or “edge1” to the closed interval [0,1]. Next my_maxpoint is defined as the numerical minimum of the law my_law over the domain [0.5,1]. Then my_testcur is evaluated at my_maxpoint, and the result is plotted. The plotted point represents the point on the curve that has the lowest *x* coordinate. By using the laws “x”, “y” and “z”, the law:nmax and law:nmin extensions find a bounding box for my_testcur. The resulting bounding box is within ACIS tolerances.

Scheme Example

```
(define my_plist (list
    (position 0 0 0)(position 20 20 0)
    (position 40 0 0)(position 60 25 0)
    (position 40 50 0)(position 20 30 0)
    (position 0 50 0)))
;; my_plist
; my_plist => (#[position 0 0 0] #[position 20 20 0]
;   #[position 40 0 0] #[position 60 25 0] #[position 40 50 0]
;   #[position 20 30 0] #[position 0 50 0])
(define my_start (gvector 1 1 0))
;; my_start
; my_start => #[gvector 1 1 0]
(define my_end (gvector –1 1 0))
;; my_end
; my_end => #[gvector –1 1 0]
(define my_testcur (edge:spline my_plist my_start my_end))
;; my_testcur
; my_testcur => #[entity 2 1]
(define my_law (law ”map(Curc(edge1),edge1)” my_testcur))
;; my_law
; my_law => #[law ”MAP(CURC(EDGE1),EDGE2)”]
(define my_maxpoint (law:nmax my_law 0.5 1))
;; my_maxpoint
; my_maxpoint => 0.840637305143896
(define my_point (dl–item:point (curve:eval–pos
    my_testcur my_maxpoint)))
;; my_point
; my_point => #[dl–item 22F793F0]
(dl–item:display my_point)
;; ()
```

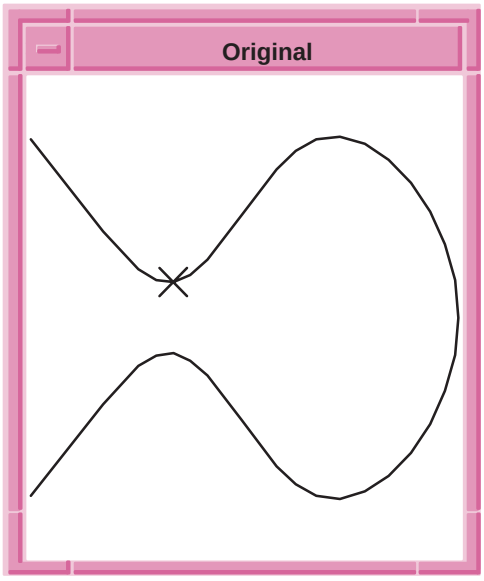

Figure 1-12. First Maximum Point on a Test Curve

The example defines my_law to be the law that returns the curvature of my_testcur with its parameter range mapped to [0,1]. The CurC law returns the curvature of its curve. Because the radius of curvature is the reciprocal of the curvature, finding where my_law is maximum finds where the radius of curvature is minimum, or where the tightest turn in the curve is located.

Skinning, Lofting, and Net Surface Singularities

Topic: Laws

Laws are used by skinning, lofting, and net surfaces to answer questions about the surface being generated.

- Laws test the integrity of what is being generated.
- The law root finder locates pathological surface cases where problems occur, such as where the uv directions aren't in the same or opposite directions, where surfaces may collapse into a degenerate type of surface, or where the skin direction is in the same direction as the surface normal.
- Laws help align the curve directions.
- ACIS has a rule that you have to place a vertex on an edge if the edge convexity changes. Laws look for changes in edge convexity along edges.
- Laws find the closest distance between two curves in the net surface operation.
- Laws help generate the desired shape and place curves through it.

Finding Intersections

Topic: Laws, *Geometric Analysis

Laws can be used to find intersections:

- Surface alignment
- Sweep intersections
- Wire offset
- Curve self-intersections
- Curve-curve intersections

In the example, two entities are created: a circle and an edge from a sine wave. These two entities do not intersect. Still, laws can be used to find the location where the two entities are closest together. Net surfaces uses this feature when the curves in the network do not intersect.

Scheme Example

```
(define my_edge1(edge:circular (position 0 0 0) 10))
;; my_edge1
; my_edge1 => #[entity 2 1]
(entity:set-color my_edge1 0)
;; ()
(define my_edge2(edge:law "vec(x,sin(x),5)" -20 20))
;; my_edge2
; my_edge2 => #[entity 3 1]
(entity:set-color my_edge2 0)
;; ()
(define my_law1 (law "cur(edge1)" my_edge1))
;; ()
(define my_law2(law "cur (edge1)" my_edge2))
;; my_law2
; my_law2 => #[law "CUR(EDGE1)"]
```

```
(law:domain my_law1)
;; ”0.000000 6.283185\n”
(law:domain my_law2)
;; ”–20.000000 20.000000\n”
(define my_law_y(law ”law1 o y” my_law2))
;; my_law_y
; my_law_y => #[law ”(CUR(EDGE1))O(Y)”]
(define dist_vec(law ”law1–law2” my_law1 my_law_y))
;; dist_vec
; dist_vec => #[law ”CUR(EDGE1)–(CUR(EDGE2))O(Y)”]
(define dist_sq(law ”dot(law1,law1)” dist_vec))
;; dist_sq
; dist_sq => #[law ”DOT(CUR(EDGE1)–(CUR(EDGE2))O(Y),
;   CUR(EDGE3)–(CUR(EDGE4))O(Y))”]
(law:nlmin2d dist_sq 3 0 1 100 ”1”)
;; x 3.088359405687513 y –9.985834540073936 z 25.000000000000007
;;  tol 0.000000000000014
;;  #t
(define my_point (dl–item:point (position –9.985834540073936
    (law:eval ”sin(–9.985834540073936)”) 5)))
;; my_point
; my_point => #[dl–item 40232000]
```