Data transformation: A semantically-based approach to function discovery

THONG H. PHAN *(phant@cpsc.ucalgary.ca)*
Computer Science Department, University of Calgary *(403) 220-7140/241-3288*
2500 University Drive N.W., Calgary, Alberta, Canada, T2N 1N4

IAN H. WITTEN *(ihw@waikato.ac.nz)*
Computer Science Department, University of Waikato
Hamilton, New Zealand

ABSTRACT

This paper presents the method of *data transformation* for discovering numeric functions from their examples. Based on the idea of transformations between functions, this method can be viewed as a semantic counterpart to the more common approach of formula construction used in most previous discovery systems. Advantages of the new method include a flexible implementation through the design of transformation rules, and a sound basis for rigorous mathematical analysis to characterize what can be discovered. The method has been implemented in a discovery system called "Linus," which can identify a wide range of functions: rational functions, quadratic relations, and many transcendental functions, as well as those that can be transformed to rational functions by combinations of differentiation, logarithm and function inverse operations.

Keywords: Function discovery, data transformation

Running head: A semanticalled-based approach to function discovery

1 Introduction

Function discovery is the problem of finding a symbolic formula for a function f from a set of its examples $\{(x, y) \mid y = f(x)\}$. Its difficulty hinges upon the expressiveness of the language used to describe functions. Because of this, earlier discovery systems that deal with numeric functions often focus their attention on a specific type of functions, such as those that can be built from combinations of arithmetic operators and symbolic variables [2, 5, 6, 7], or from other simple functions [1, 8, 11, 14]. Although the systems have different emphases and scopes, they share the approach of syntactic construction of formulas based on features that can be detected numerically in the examples.

This paper presents the method of data transformation, pioneered by the discovery system FFD [13] and based on the idea of *transformations between functions*. While the method can be viewed as a semantic counterpart of the formula construction approach used in previous discovery systems, its semantic nature permits rigorous analysis of its discovery capability through the use of a transformation-based language for describing functions. Our contributions in this direction are: (1) A description language that—even in its basic form—is far more expressive than those used in other discovery systems; (2) A robust implementation that is easily extended to encompass different languages; and (3) A general model of data transformation that can be applied to domains other than real-valued functions. In this paper, we focus on the description language and the implementation of the method in the domain of real-valued functions; details of the generalized model can be found elsewhere [9].

The paper is organized as follows. First, Section 2 defines a transformation-based description language and illustrates the basic idea underlying the data transformation method. Section 3 discusses two potential problems in implementing this method and how the discovery system LINUS resolves them. Section 4 characterizes the discovery capability of LINUS in terms of its description language's expressiveness, the sensitivity of its basic transformations to error, and how it copes with the inexact implementation of each transformation. Finally, Section 5 summarizes two extensions that enable LINUS to deal with situations where requests for new examples are impractical and the examples may be inexact.

Name	Transformation
Function inverse	$Inv: \quad (x,y) \mapsto (y,x)$
Reciprocal	$R: \quad (x,y) \mapsto (x,1/y)$
Natural logarithm	$L_c: \quad (x,y) \mapsto (x,\ln(c \times y))$
Differentiation	$D_{(c,y_c)}: \quad (x,y) \mapsto (x,dy/dx)$
Factoring	$F_{(c,y_c)}: \quad (x,y) \mapsto (x,(y-y_c)/(x-c))$

Table 1: Basic transformation rules

2 Description language and discovery method

Both description language and discovery method are based on the idea of transformation from one function to another. A function $y = f(x)$ is represented by a composition of invertible transformations $T_k \circ \cdots \circ T_2 \circ T_1$ that maps its examples (x,y) to examples (u,v) of another function $v = g(u)$ in some predefined class. The discovery operation is to find a composition $T_k \circ \cdots \circ T_2 \circ T_1$ that transforms examples of f to those of g, a function whose formula can be determined directly from its examples. The formula for f is then recovered by applying the inverse of $T_k \circ \cdots \circ T_2 \circ T_1$ to the formula found for g.

Clearly, the representational power of the language and the practicality of the discovery method both depend on the transformations used and the basic functions that can be recognized directly. In this section, we present a description language that is sufficiently expressive to include a wide range of functions, but is still fairly simple to implement.

2.1 The description language

We define a description language $\mathcal{L}$ whose members are pairs of equations of the form:

$$\begin{aligned} T_k \circ \cdots \circ T_2 \circ T_1(x,y) &= (u,v) \\ P(u,v) &= 0. \end{aligned}$$

Each T_i is one of the five transformations defined in Table 1; x, y, u and v are symbolic variables; and $P(u,v)$ is a formula that can be determined from the values of u and v. By convention, y denotes the function's value, x denotes its argument, and $T_k \circ \cdots \circ T_2 \circ T_1$ denotes the composition of transformations $T_k(\cdots(T_2(T_1(x,y))))$. The sequence $T_k \circ \cdots \circ T_2 \circ T_1$ is called a *transformation sequence* and the formula $P(u,v)$ is called a *matching pattern*. Since the expressions on the right-hand side of the defining equations are always (u,v) and 0, the full description can be abbreviated to a pair $(T_k \circ \cdots \circ T_2 \circ T_1, P(u,v))$.

In Table 1, the subscripts c of L_c and $F_{(c,y_c)}$ denote parameters that must be chosen when applying these transformations. In the case of L_c, while in principle c may have any non-zero value, the implementation only uses two: $+1$ and -1. In effect, L_c may be interpreted as two alternative transformations L_{+1} and L_{-1}, at most one of which is applicable depending on the sign of the y-values in the example set. In the case of $F_{(c,y_c)}$, y_c represents the function's value at $x = c$, so that, despite appearances, only one parameter is involved. In the case of $D_{(c,y_c)}$, although the operation itself requires no parameter, one pair of values (c, y_c) is retained so that the transformation can be inverted to yield a unique result. For brevity, the subscripts are omitted unless the actual values of the parameters are germane to the discussion.

In general, different sets of transformations and matching patterns will lead to different description languages. However, the discovery mechanism to be discussed does not depend on the set of transformations nor on the matching pattern, although they certainly influence its successful implementation. Later in the paper, we discuss the consequences of choosing the transformations in Table 1. Furthermore, while the matching pattern may have any form, it is expedient in practice to select formulas whose symbolic forms are easily determined from their examples. In the current implementation, three forms of matching pattern are provided: Constant patterns $v + c$, quadratic patterns $v^2 + c_1uv + c_2v + c_3u^2 + c_4u + c_5$, and rational patterns $h_1(u)v + h_2(u)$ where h_1, h_2 are non-constant polynomials. (The label "rational pattern" derives from the explicit form of v when the equation $h_1(u)v + h_2(u) = 0$ is solved.)

A formal interpretation

A description in the above form represents a function $y = f(x)$ that satisfies the system of simultaneous equations:

$$\begin{cases} g_1(x,y) &= u \\ g_2(x,y) &= v \\ P(u,v) &= 0. \end{cases}$$

Here, g_1, g_2 are the formulas obtained by combining the sequence $T_k \circ \cdots \circ T_2 \circ T_1(x,y)$, each one as defined in Table 1, into the pair $(g_1(x,y), g_2(x,y))$. Normally, one would simplify g_1 and g_2 using the standard rules of function composition, symbolic differentiation, and algebraic manipulation. The transformations in Table 1 thus can be interpreted as *syntactic substitution* rules and so the description is simply an abbreviation of the corresponding system of equations.

From a semantic perspective, they also serve to define *transformations* from one function $y = f(x)$ to another $v = g(u)$. In other words, a transformation maps all examples (x,y) of the function f into examples (u,v) of a different function g. This is the basis of the data transformation method. However, for the set of transformations in Table 1, this interpretation introduces some semantic problems that must be resolved first. For instance, when $y = f(x)$ is non-monotonic, its inverse $x = f^{-1}(y)$ will not be a single-valued function, and so the transformation Inv must be handled carefully if it is to be used effectively. In a similar way, the transformation L_c requires that the y values must be either all positive or all negative. These semantic problems and the methods to circumvent them are discussed in more detail in the next section.

Example of a function description

As an example, consider the function $y = \log(x^2 + 2) + x + 1$. One expression of it in $\mathcal{L}$ is:

$$\begin{aligned} R \circ F_{(1,\frac{5}{3})} \circ D_{(1,2+\log 3)}(x,y) &= (u,v) \\ uv - 2v + \tfrac{3}{2}u^2 + 3 &= 0. \end{aligned}$$

In the abbreviated form, this becomes $(R \circ F_{(1,\frac{5}{3})} \circ D_{(1,2+\log 3)},\ uv - 2v + \frac{3}{2}u^2 + 3)$. The parameters c_1 and c_2 in $F_{(c_1,y_{c_1})}$ and $D_{(c_2,y_{c_2})}$ are both arbitrarily chosen to be 1, and so

$y_{c_1} = \frac{5}{3}$ and $y_{c_2} = 2 + \log 3$ in this example.[1] To show that this description represents the original function, one simply performs the necessary transformations and solves the resulting system of equations. Applying the sequence of transformations to (x, y) yields

$$\begin{aligned} R \circ F_{(1,\frac{5}{3})} \circ D_{(1,2+\log 3)}(x,y) &= R \circ F_{(1,\frac{5}{3})}(x, dy/dx) \\ &= R\left(x, \frac{dy/dx - \frac{5}{3}}{x-1}\right) = \left(x, \frac{x-1}{dy/dx - \frac{5}{3}}\right). \end{aligned}$$

Equating this to (u, v) and eliminating u and v using $uv - 2v + \frac{3}{2}u^2 + 3 = 0$ gives

$$\frac{x-1}{dy/dx - \frac{5}{3}} = \frac{-\frac{3}{2}x^2 - 3}{x-2}.$$

Solving for dy/dx produces the differential equation $\dfrac{dy}{dx} = \dfrac{2x}{x^2+2} + 1$, which has solutions of the form $y = \log(x^2 + 2) + x + C$, where C is the constant of integration. The subscript of $D_{(1,2+\log 3)}$ provides an initial condition for this differential equation, because by definition the subscript (c, y_c) denotes the value y_c of the function at $x = c$ prior to the application of $D_{(c,y_c)}$. In this case, $x = 1$ and $y = 2 + \log 3$ give $C = 1$ and $y = \log(x^2 + 2) + x + 1$.

2.2 The method of data transformation

The discovery problem is to find a transformation sequence $T_k \circ \cdots \circ T_2 \circ T_1$ and a matching pattern $P(u, v)$ such that the solution of the system of equations

$$\left\{ \begin{aligned} T_k \circ \cdots \circ T_2 \circ T_1(x,y) &= (u,v) \\ P(u,v) &= 0 \end{aligned} \right.$$

satisfies examples of the unknown function. (Here we assume that the examples provided are sufficient to identify the unknown function and that computation errors are negligible; these issues will be discussed later.)

The method of data transformation proceeds by first finding a sequence $T_k \circ \cdots \circ T_2$ and a pattern $P(u, v)$, which together define the system of equations

$$\left\{ \begin{aligned} T_k \circ \cdots \circ T_2(\hat{x},\hat{y}) &= (u,v) \\ P(u,v) &= 0 \end{aligned} \right.$$

[1]For illustration, the values of y_{c_1} and y_{c_2} are given here as $\frac{5}{3}$ and $2 + \log 3$ respectively. In practice, they are estimated from the current data and expressed as floating point numbers.

where the new examples $(\hat{x}, \hat{y})$ are obtained by applying T_1 to the examples (x, y) of f. If this system has a solution $\hat{f}$, the method then applies the inverse T_1^{-1} of T_1 to the symbolic form of $(\hat{x}, \hat{f})$ to obtain the formula for f. Otherwise, it continues with other transformation sequences. The procedure is applied recursively so that the problem of finding a formula for f is reduced to a search for a formula that satisfies the relation $P(u, v) = 0$.

From an operational perspective, the discovery system searches the space of transformation sequences for one that converts examples of the unknown function into those of a predefined pattern. After each transformation, it checks whether the transformed images match that pattern. Because of the use of finite precision arithmetic, it is necessary to permit some small mismatch between the examples and the best-fit pattern. If the difference is acceptable, the solution is the description formed by the sequence of transformations used so far and the pattern matched. In general, a description for the unknown function $y = f(x)$ corresponds to an equation in x, y and the derivatives of f, rather than to an expression for y solely in terms of x. If explicit formulas are desired, the system may be augmented with a symbolic algebra system to solve the implicit equations whenever possible.

An illustration of the discovery operation

To illustrate, consider the set of examples (x, y) shown in Figure 1a. To find a formula satisfying these, we search for a sequence that transforms them into examples of some recognizable function. Figures 1b, c, and d depict the results of one particular series of transformations. In each of these graphs, the horizontal scale represents the u-axis and the vertical scale the v-axis.

The data points in each graph can be viewed as examples of some particular function that is related to the original by the corresponding sequence of transformations, shown in the top left corner of the graph. In Figure 1b, they have been transformed by the single operation $D_{(1,2+\log 3)}$. In Figures 1c and 1d, they have been transformed by the sequences $F_{(1,\frac{5}{3})} \circ D_{(1,2+\log 3)}$ and $R \circ F_{(1,\frac{5}{3})} \circ D_{(1,2+\log 3)}$ respectively. As it turns out, the transformed examples of Figure 1d satisfy the relation $-\frac{1}{2}uv + v - \frac{3}{4}u^2 - \frac{3}{2} = 0$. This means that the

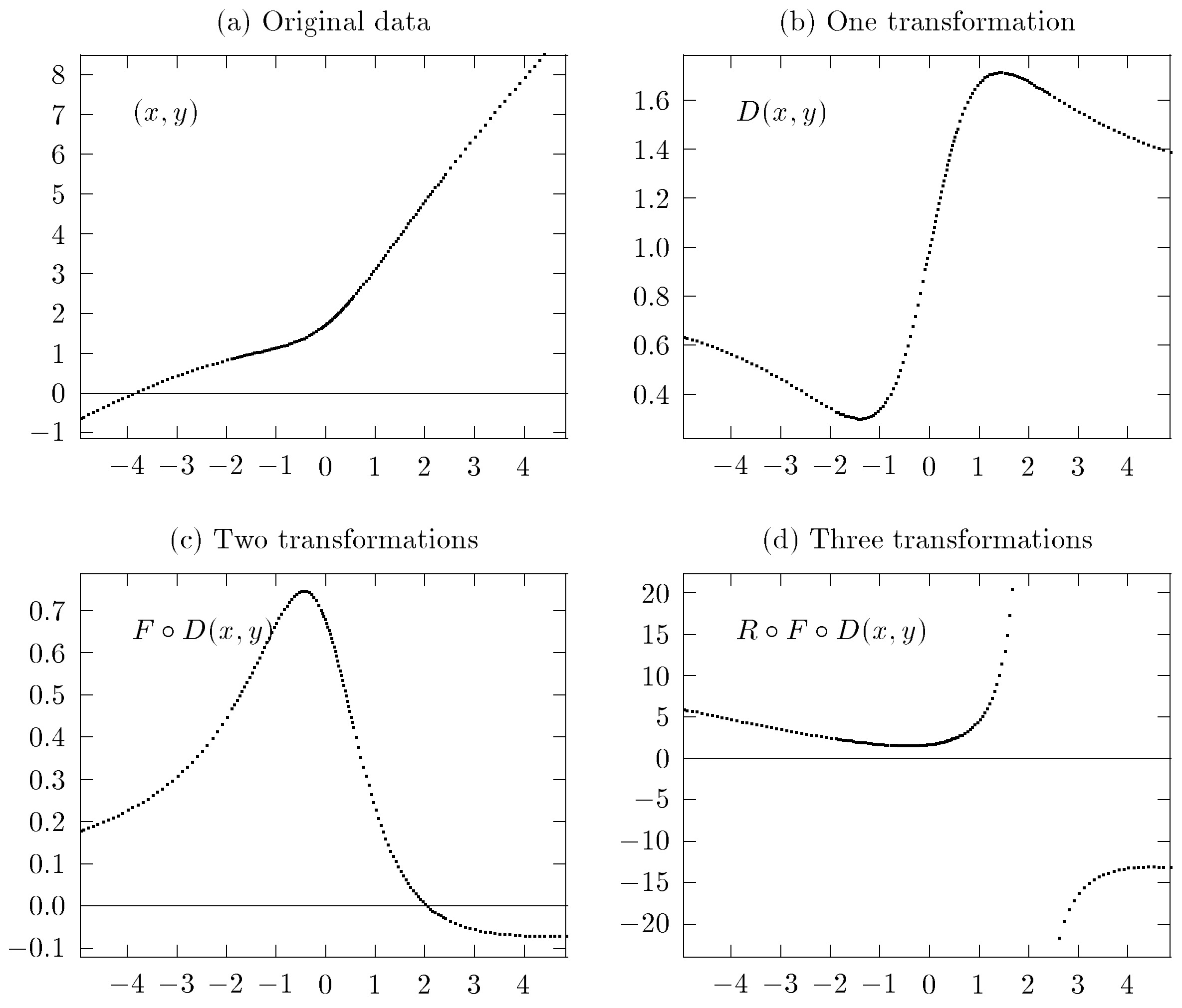

Figure 1: Graphs of transformed examples

unknown function has a description:

$$\begin{aligned} R \circ F_{(1,\frac{5}{3})} \circ D_{(1,2+\log 3)}(x, y) &= (u, v) \\ -\tfrac{1}{2}uv + v - \tfrac{3}{4}u^2 - \tfrac{3}{2} &= 0. \end{aligned}$$

Solving for y gives the formula $y = \log(x^2 + 2) + x + C$, where C is an integration constant. The subscript of $D_{(1,2+\log 3)}$ serves to recover the value of the constant, in this case $C = 1$.

Relation to other discovery schemes

Discovery schemes that work with empirical data exhibit two complementary approaches: Data analysis and formula construction. In the former approach, the examples are analyzed using numerical or statistical tools to determine what type of formulas may be applicable.

Examples include Gerwin's discovery system [3], Coper [5], Schaffer's E* algorithm [12], KEDS [11], Kepler [14]. In the latter approach, possible solutions are constructed that satisfy certain qualitative features detected in the data. Examples of this approach include the Bacon discovery systems [7], Abacus [2], and Fahrenheit [6, 15]. Since the features are detected by analyzing the data, both approaches can be viewed as extremes of a single spectrum. The degree of sophistication in the tools used for data analysis indicates whether a system should be categorized as data analysis or formula construction, or a combination of both.

Data transformation can be viewed as a discovery method that subsumes both the above approaches. First, possible solutions are formed implicitly by building transformation sequences incrementally. Second, numerical features of the examples can be used to find a matching pattern that best fits them, or to select the transformation to be applied next. However, the new method differs in one important respect: In order for successive transformations to be applicable, the transformed images of the initial data must represent examples of some function. In other words, they must not imply the existence of examples that have different values for the same argument.

This requirement is what really distinguishes the method of data transformation from the common approach of formula construction. The former is a semantically-based transformation between functions, while the latter is a syntactically-based manipulation of partially constructed formulas.

From a practical perspective, successful implementation of the method hinges upon whether the above requirement can be satisfied. Of course, this is trivial if each transformation, when applied to examples of a function, is guaranteed to produce only examples of another function. However, as explained in the next section, that is not the case for the language $\mathcal{L}$. As a result, the practicality of the method depends on the transformations used to define the description language. In the next section, we elaborate the main features of the discovery system LINUS [9], whose implementation has been generalized to include other description languages like the one in Section 2.1.

3 The implementation: Linus

Linus is a function discovery system that implements the method of data transformation. Since the method works on the premise that successive application of transformations is always possible, its implementation depends critically on the description language being used. In this section, we discuss the two major problems in implementing the method using the language $\mathcal{L}$, and sketch the solutions adopted by Linus.

3.1 Implementation problems

Data transformation operates by applying the basic transformations one at a time to the examples being considered. There are two conditions that must be satisfied: (1) All transformations are always applicable; and (2) They can be computed sufficiently accurately. In the case of the language $\mathcal{L}$, this causes problems for the transformations Inv, L, D and F.

First, Inv and L cannot be applied to an arbitrary set of examples. Since inverting a non-monotonic function yields a relation rather than a function, Inv cannot be applied to any set of examples that implies a non-monotonic function because it violates the requirement that the transformed data must represent examples of some (single-valued) function. The problem with L is slightly different: It simply cannot be applied to functions that have both positive and negative values because the results are undefined in the domain of real-valued functions.

More generally, if the application of a transformation is constrained in any way, certain sets of examples may fail to satisfy that constraint. As a result, some functions may not be discovered using the data transformation method even though they can be described in the language being considered. Thus the implementation may be incomplete in the sense that some descriptions are not discoverable because their corresponding transformation sequences cannot be generated from the example set. We call this the problem of incomplete generation of function descriptions.

Second, the computation of D and F refers to the function implied by the current examples, rather than the examples themselves. D computes the function's derivative and F computes its ratio with respect to some linear factor. Because the formula for that function is not available, one can only provide numerical approximations to these transformations. It is clear that successive application of D and F will increase the error inherent in numerical approximation. We call this the problem of inexact computation of transformations. Note that it is not easily overcome by the common practice of using examples that are closely spaced in x to improve computation accuracy. An upper bound on spacing is impossible to guarantee after a transformation sequence that includes Inv has been applied to the original examples. This limits the utility of formulas for error estimation based on power series analysis, which relies on knowledge of spacing between data points.

3.2 Effect on the implementation's robustness

The combined effect of the above problems is that a discovery system using the data transformation approach must rely on "good" examples being available, ones that lead to an accurate description of the function they imply. Assuming that a description for the unknown function exists, it follows that there must exist examples that allow the application of the transformation sequence in that description. The problem is to find them.

While it is trivial to find good examples if the function's description is known, it is impossible to select them in the absence of any information about the function. The data transformation method is inherently susceptible to the quality of the examples supplied. If they are not carefully chosen, certain transformation sequences may not be constructible because some transformations' constraints are violated. To compound the problem, the inexact nature of numerical computation can easily lead to meaningless results, since the cumulative error can grow large enough to prevent successful pattern matching.

This dependence on the quality of the examples supplied suggests that an adaptive scheme is needed. The solution used in Linus is to permit on-demand example selection and to monitor the computation error of all transformations with respect to the examples under consideration. This allows it to circumvent the constraints and improve the numerical accuracy with which transformation sequences are computed.

3.3 The solution in Linus

Linus's design requires that (1) specific examples must be selectable on demand; (2) all transformations must be numerically applicable to the examples selected; and (3) all transformations must be numerically reversible. The first two combine to overcome the problem of incomplete generation of function descriptions caused by constraints on the transformations Inv and L. The third provides a mechanism for estimating the errors incurred by inexact arithmetic and determining how they can be reduced.

To satisfy these requirements, Linus employs a data selection scheme that adapts to the discovery task at hand. Furthermore, it treats individual transformations as reversible numerical tools, rather than as semantic mappings between functions—notwithstanding the fact that the latter interpretation forms the basis of the data transformation method. Together, the requirements ensure that, with a simple manipulation of the current example set, Linus can always apply any transformation at any time. The mathematical validity of the transformations will be determined when the description found is verified. The overall result is that Linus is no longer affected by a badly chosen set of examples, and that numerical computation can be as accurate as is practically possible. The implementation of these ideas is elaborated below.

On-demand data selection and data splitting

Linus is empowered to select examples as they are needed to overcome constraints on the transformations. By examining the examples currently available, it can determine which ones would be best to use in the next operation. If these are not available, a request is issued for them before proceeding further. Most importantly, if the current examples prevent the application of a transformation, Linus determines which new examples will enable the blocked transformation to proceed, and specifically requests them.

For both transformations Inv and L, this is done in combination with a data splitting scheme. In the case of Inv, the examples are split into monotonic segments, each of which is processed as a separate discovery task. Monotonicity can be determined by sorting the examples by their x component and then examining their y components in order. In the case of L, the ordered list is divided into segments of adjacent examples with the same sign in the y component, so that either L_{+1} or L_{-1} will be applicable to each segment. (Examples that are exactly zero are discarded.) In both cases, to qualify as a solution to the original discovery problem, any description found for one segment must also satisfy all the other segments. A verification procedure checks that the description satisfies all known examples—even those that were omitted during the initial search for the unknown function's description.

This data splitting scheme guarantees that all transformations are always applicable, and so avoids the problem of incomplete generation of function descriptions caused by constraints on the transformations Inv and L. There is no danger of successive splitting resulting in subsets containing too few examples, because LINUS can request additional examples where needed.

It is important to note that the examples being requested are not, in most cases, examples of the unknown function, but examples of their transformed images. For this reason, LINUS requires the ability to select specific examples *on demand*, rather than working from a pre-supplied set of examples.[2] The requirement of numerically reversible transformations helps in translating specific requests for the transformed images into requests for examples of the unknown function.

Local approximation and automatic error analysis

While the problem of inexact computation is unavoidable, LINUS attempts to reduce its effect by the use of piecewise approximations and continuous error feedback. Using least-square curve-fitting routines, it computes piecewise approximations to the function implied by the current examples. The symbolic forms of these approximations, called *local approximations*, are then substituted for the unknown function whenever its symbolic form is required. For instance, to apply D or F, the local approximations are differentiated or factored, and the results evaluated numerically to provide the transformed images of the examples. The fitting error can be used to estimate the computation error of D and F.

[2]In practice, if LINUS must work with pre-supplied examples, it simulates new examples using an interpolation routine.

The local approximations also serve to identify places where more examples are needed to ensure a desired level of accuracy. If the fitting errors exceed some predefined error tolerance, more examples are requested before any transformation is attempted. Because the fitting procedure utilizes a fixed number of examples, this leads to a finer piecewise approximation of the function implied by the current examples. As a result, the new local approximations are more accurate in representing those examples. (The local approximations only reflect the examples being considered, which could still contain errors from inexact computation or noise in the original data.)

To provide immediate feedback on the computation error, Linus employs the following procedure. Let (u, v) denote an example before transformation T is applied, and $T(u, v) = (\hat{u}, \hat{v})$ the result afterwards. Applying T^{-1} to $(\hat{u}, \hat{v})$ gives $(\tilde{u}, \tilde{v})$. If the new result $(\tilde{u}, \tilde{v})$ is also approximated by the current local approximation, then the example set is said to "support" the transformation T. Otherwise, more examples are requested in the regions where the errors exceed the desired accuracy. This leads to more accurate local approximations, which helps to ensure that the error caused by applying T remains insignificant.

This procedure assumes that the implementation of both T and T^{-1} is reasonably accurate and that the error of T^{-1} does not cancel out the error of T. Combined with the formal analysis of error propagation described below, this gives an estimate of the error caused by applying T to the current examples. Linus uses this information to decide whether the cumulative error remains low enough to allow an acceptable solution.

Remarks

The problem of incomplete generation of function descriptions is caused by the inherent constraints of a transformation preventing its application to a set of examples. By treating transformations as numerical tools rather than as semantic mappings between functions, Linus can apply numerically any sequence of transformations, even ones that appear to violate the semantic constraints.

In practice, Linus has two options when faced with an example set that disallows the application of a transformation T. It could manipulate the set so that T can be applied numerically without insisting that the transformation being performed is semantically meaningful. This is the data splitting scheme sketched above. Or it could combine T with some other transformation sequence $\mathcal{T}$ such that the new sequence $\mathcal{T} \circ T$ represents a semantic mapping between functions that *is* applicable to the current example set. This option is used by Linus to implement certain sequences of transformations that contain Inv and L. The idea is to reduce the number of data splittings, which, for obvious reasons, should not be performed indiscriminately.

From a theoretical perspective, Linus effectively defers the semantic interpretation of the transformation T to the point when the final system of equations must be solved to find the overall solution. If there is a solution to that system, then the application of T (or the sequence containing it) is semantically meaningful. The fact that Linus needs to perform certain manipulations on the current examples to permit the application of T is inconsequential, except to the extent that such manipulations sidestep the constraints on T.

4 Analysis of Linus's discovery capability

In this section, we characterize Linus in terms of the representational power of the description language it employs, describe how it monitors the propagation of error through each transformation in the language, and explain how it copes with the inexact implementation of the transformations.

4.1 Expressiveness of the language $\mathcal{L}$

The basic language $\mathcal{L}$ implemented in Linus contains the five transformations in Table 1 and a constant matching pattern $v + c$. The propositions in this section show that $\mathcal{L}$ is sufficiently expressive to include a wide range of functions: rational functions, quadratic relations, and many transcendental functions, as well as those that can be transformed to rational functions by combinations of logarithm, differentiation and function inverse operations. The propositions provide supporting arguments for the choice of $\mathcal{L}$ as the base language for describing functions.

The claims of expressiveness follow from the recursive nature of the data transformation method: If a function $y = f(x)$ can be transformed to another expressible function $v = g(u)$, then it is also expressible. This is formally stated in the following proposition, whose proof follows immediately from the definition of $\mathcal{L}$.

Proposition 1 *Let $y = f(x)$ be a function that can be described in $\mathcal{L}$ as $(T_k \circ \cdots \circ T_1, P(u, v))$. If a function $\hat{y} = \hat{f}(\hat{x})$ has the property that $T_0(\hat{x}, \hat{y}) = (x, y)$ where T_0 is a transformation, then $\hat{f}$ can be described as $(T_k \circ \cdots \circ T_1 \circ T_0, P(u, v))$. In this case, $\hat{f}$ is said to be transformable to f by the transformation T_0.*

This proposition is the main tool for proving that certain classes of functions are expressible in $\mathcal{L}$. The following propositions present the basic classes of expressible functions that can be used to derived other expressible function classes. Proofs of these propositions can be found in the Appendix.

Basic claims of expressiveness

For convenience, we use the symbol T^* to denote a sequence $T_k \circ \cdots \circ T_2 \circ T_1$ of transformations of the same type T. Similarly, $\{T, T'\}^*$ denotes a sequence composed only of transformations of the types occurring between the brackets $\{\}$, in this case T and T'.

Proposition 2 *Descriptions of the form $(\{R, F\}^*, v + c)$ describe all and only rational functions.*

This proposition states that, using only a constant pattern, the two transformations R and $F_{(c,y_c)}$ are sufficient to express all rational functions. The others may be viewed as auxiliary transformations that serve to expand the set of functions expressible in $\mathcal{L}$.

The next proposition states that any function satisfying a quadratic relation can be transformed to a rational function. Combined with Propositions 1 and 2, this means that any function that can be transformed to a function that satisfies a quadratic relation using the transformations defined in Table 1 is also expressible in $\mathcal{L}$.

Proposition 3 *A function $y = f(x)$ that satisfies a quadratic relation $a_{0,2}y^2+a_{1,1}xy+a_{2,0}x^2+a_{0,1}y + a_{1,0}x + a_{0,0} = 0$ can be transformed to a rational function $v = g(u)$ that satisfies the relation $h_1(u)v + h_2(u) = 0$, where h_1 and h_2 are polynomials of degree at most 2.*

Together with Proposition 2, this proves an important feature of the language $\mathcal{L}$: The set of expressible functions remains unchanged when the matching pattern $P(u, v)$ is changed from a constant pattern $v + c$ to a rational pattern $h_1(u)v + h_2(u)$ or a quadratic pattern $v^2 + c_1uv + c_2v + c_3u^2 + c_4u + c_5$. However, from a practical point of view, the transformation sequences in many descriptions are much shorter when the patterns are expressed using quadratic or rational patterns. Linus takes advantage of this property to accelerate the discovery of functions expressible in $\mathcal{L}$.

Examples of function classes expressible in $\mathcal{L}$

Using the basic results above, the following propositions illustrate some classes of transcendental functions that are expressible in $\mathcal{L}$. They demonstrate two simple techniques to determine expressible functions. In Proposition 4, a specific form of description is selected and its associated system of equations is solved to provide the general formula of the function class it describes. In Proposition 5, a function is shown to be expressible in $\mathcal{L}$ by exhibiting a transformation sequence that reduces it to another expressible function.

Proposition 4 *Descriptions of the form $(\{R, F\}^* \circ D^*, v+c)$ encompass all and only functions in this class:*

$$\begin{aligned} y \;=\; & p_k + \sum_i c_{1,i}x^{n_{1,i}} \log(a_{1,i}x + b_{1,i}) + \sum_i c_{2,i}x^{n_{2,i}} \tan^{-1}(a_{2,i}x + b_{2,i}) \\ & + \sum_i \frac{c_{3,i}}{(a_{3,i}x + b_{3,i})^i} + \sum_i \sum_j \frac{a_{4,i,j}x + b_{4,i,j}}{(h_i)^j}; \end{aligned}$$

where $n_{1,i}$, $n_{2,i}$ are non-negative integers; p_k is a polynomial with degree at most k; each h_i is an irreducible polynomial of degree 2; each of $a_{i,j}$, $a_{4,i,j}$, $b_{i,j}$, $b_{4,i,j}$, and $c_{k,i}$ is a real constant; and each $\sum_i$ represents a finite summation of similar terms.

Proposition 5 *If a function g is expressible in $\mathcal{L}$, then any function $y = f(x)$ that satisfies one of the following relations is also expressible in $\mathcal{L}$:*

1. *$g(y) = x$, $g(x) = \log y$, $g(y) = \log x$, $g(\log y) = x$, $g(\log x) = y$, $g(\log x) = \log y$, $g(\log y) = \log x$; and*

2. *Solutions of differential equations of form either $y' = yg(x)$ or $y' = g(y)$, where y' is the derivative of y with respect to x.*

Remarks

In addition to showing that certain function classes are expressible in $\mathcal{L}$, the propositions also identify the matching patterns that are redundant. Any proof that shows a function class to be expressible with respect to some pattern P also shows that the language's expressiveness will not be increased by the use of functions in that class as matching patterns. This is because functions that match those patterns can be further transformed to functions that match P. As a result, such patterns are redundant in that they do not extend the expressiveness of the language.

However, from a practical perspective, redundant patterns can be used to improve performance. Instead of waiting for more transformations to reduce such patterns to simpler ones, the system could include a matching routine that recognizes them directly. In particular, from a known class of expressible functions, one can implement a special configuration, which we call "accelerated mode," in which patterns of the form $f(u) - v$ are used directly, where f is any function in that class. The trade-off is between a certain but small increase in matching time and the possibility of greatly reduced discovery time because fewer transformations are performed.

From a theoretical perspective, it can be argued that non-constant matching patterns are simply different implementations of the data transformation concept. Instead of matching examples to a non-constant pattern, one could design a sequence of transformations that accomplishes the same effect but results in a constant. In a sense, transformations and matching patterns are just two aspects of one operation: The conversion of functions from one form to another. A non-constant pattern P can be viewed as an aggregated transformation that maps a function $v = g(u)$ that satisfies the relation $P(u, v) = 0$ into a constant. Under this interpretation, a non-constant pattern is an implementation shortcut that combines a constant pattern with an aggregated transformation.

4.2 Propagation of computation error

A major concern with the data transformation method is that it may be sensitive to errors caused by inexact computation, or by noise in the original examples. This sensitivity depends both on the transformations implemented and on the nature of the noise.

We now show how errors, either in the examples or in the computation, are propagated through each transformation defined in Table 1, and how they can be estimated in practice. Throughout the analysis, the error relative to the true value is denoted by δ, and may have a different value for each example. In practice, either the maximum error is chosen as an overall upper bound, or the individual propagated error is estimated for each example separately. Either way, the solution is considered acceptable if the accumulated error falls below a user-specified accuracy tolerance (for example, 3 digits of accuracy).

In the following analysis, suppose that for each transformation T we have $T(u,v) = (\hat{u},\hat{v})$ and $T(u, v+\delta v) = (\hat{u},\tilde{v})$, where $\tilde{v}$ may be a function in $\hat{u}$, $\hat{v}$ and δ. The propagation error $E_T(\delta)$ for a transformation T with initial error δ is defined as $E_T(\delta) = \tilde{v}/\hat{v} - 1$ when $\hat{v} \neq 0$ and $\tilde{v}$ when $\hat{v} = 0$. Thus δ is the relative error with respect to the true value v, and $E_T(\delta)/\delta$ is the amplification factor caused by the transformation T.

The definition of E_T requires both $T(u,v) = (\hat{u},\hat{v})$ and $T(u,v+\delta v) = (\hat{u},\tilde{v})$ to have the same first component $\hat{u}$. Otherwise, the comparison between $\hat{v}$ (the case of $\delta = 0$) and $\tilde{v}$ (the case of $\delta \neq 0$) would be meaningless for E_{Inv}. Furthermore, E_T does not contain any error that is incurred *during* the computation of T; it concerns only the propagation of previous errors. If desired, the accumulated error of the sequence $T_k \circ T_{k-1} \circ \cdots \circ T_1$ could be estimated as

$$\xi_k + E_{T_k}(\xi_{k-1} + E_{T_{k-1}}(\cdots E_{T_2}(\xi_1 + T_1(\xi_0))\cdots)),$$

where ξ_0 is the initial error in the data and $\xi_1, \ldots, \xi_k$ are the errors of computing $T_1, \ldots, T_k$. In LINUS, the error ξ for both D and F includes the fitting error of the local approximations as well as the error caused by floating-point computation. Since the value of ξ is implementation dependent, the analysis is restricted to the propagated error E_T of each transformation T. The

Propagation formula	Estimated error
$E_R(\delta) = -\dfrac{\delta}{1+\delta}$	$-\dfrac{\delta}{1+\delta}$
$E_{Inv}(\delta) = -\dfrac{g'(v+\delta v)}{g(v+\delta v)}\delta v$	$-\dfrac{\delta}{1+\delta}\cdot \hat{g}$
$E_L(\delta) = \begin{cases} \log_{\lvert v\rvert} \lvert 1+\delta\rvert & \text{if } \lvert v\rvert \neq 1 \\ \ln\lvert 1+\delta\rvert & \text{if } \lvert v\rvert = 1 \end{cases}$	$c\delta$
$E_D(\delta) = \delta + \dfrac{v}{v'}\delta'$	$(k+1)\delta$
$E_F(\delta) = \dfrac{\delta}{1-v_c/v}$	$\dfrac{n\delta}{n-1-\delta}$

Table 2: Propagation of error

symbol δ thus denotes the sum of all errors that are accumulated before T is applied. Table 2 summarizes the estimated error propagation for the transformations in $\mathcal{L}$. These estimates are derived below; the formulas for their inverses can be obtained in a similar manner.

Estimating propagation error

Of the five transformations Inv, R, L, D and F, only the error propagated by R is independent of the value of the examples being transformed. The transformation $R(u, v+\delta v) = (x, 1/(v+\delta v))$ gives the formula for E_R as

$$E_R(\delta) = \frac{1/(v+\delta v)}{1/v} - 1 = \frac{v}{v+\delta v} - 1 = -\frac{\delta}{1+\delta}.$$

For other transformations, the value of $E_T(\delta)$ depends on both δ and v. Consequently, the exact form of $E_T(\delta)$ is derived first, and then a simpler estimate that does not depend on v is given. The estimated formula can be used in place of the exact formula provided that the assumption stated in each case is true.

In the case of Inv, its definition gives $Inv(u, v+\delta v) = (v+\delta v, u)$. Let g be the function implied by the transformed examples when $\delta = 0$. Assuming that g is analytic within the neighborhood of $v+\delta v$, $g(v)$ can be expressed in terms of $g(v+\delta v)$ using the power series expansion

$$g(v) = g(v+\delta v) - g'(v+\delta v)\delta v + \frac{1}{2}g''(v+\delta v)\delta^2 v^2 - \cdots$$

where g' and g'' are the derivatives of g with respect to v. Using only the first two terms gives $g(v) \approx g(v+\delta v) - g'(v+\delta v)\delta v$. Therefore,

$$E_{Inv}(\delta) = \frac{g(v)}{g(v+\delta v)} - 1 = \frac{g(v+\delta v) - g'(v+\delta v)\delta v}{g(v+\delta v)} - 1 \approx -\frac{g'(v+\delta v)}{g(v+\delta v)}\delta v.$$

Let $\tilde{g}$ be the corresponding local approximation computed from the transformed examples $(\hat{v}, u) = (v + \delta v, u)$, and $\tilde{g}\,'$ its first derivative. Since $\tilde{g}$ approximates g, we estimate g' using $\tilde{g}\,'$. That gives $E_{Inv}(\delta) \approx -\dfrac{\tilde{g}\,'(\hat{v})}{u}\delta v$. Since $v = \dfrac{v+\delta v}{1+\delta} = \dfrac{\hat{v}}{1+\delta}$, we have $E_{Inv}(\delta) \approx -\dfrac{\delta}{1+\delta}\hat{g}(\hat{v})$, where $\hat{g}(\hat{v})$ is computed from the transformed images $(\hat{v}, u)$ as $\hat{v}\tilde{g}\,'(\hat{v})/u$.

Next, assuming that $|v|$, $|v+\delta v| \neq 0$, we have $L(u, v+\delta v) = (u, \ln|v+\delta v|)$ and

$$E_L(\delta) = \begin{cases} \log_{|v|}|1+\delta| & \text{if } |v| \neq 1 \\ \ln|1+\delta| & \text{if } |v| = 1 \end{cases}$$

Since $\log_a(b) = -\log_{\frac{1}{a}}(b)$, we can assume $|v| > 1$ without loss of generality. In general, for any constant c, if $|v|^{c\delta} \geq 1+\delta$ the propagated error can be estimated as $E_L(\delta) = \log_{|v|}|1+\delta| \leq \log_{|v|}(|v|^{c\delta}) = c\delta$. Alternatively, we can compute $\log_{|v|}|1+\delta|$ directly from the transformed examples $(u, \tilde{v}) = (u, \ln(v+v\delta))$ by the formula $\log_{|v|}|1+\delta| = \ln(1+\delta)/(\ln\tilde{v} - \ln(1+\delta))$.

In the case of D, its definition gives $D(u, v+\delta v) = (u, v' + \delta v' + v\delta')$, where δ' is the derivative of δ with respect to u. A simplified formula for $E_D(\delta)$ is obtained if we assume that, in the neighborhood of each example (u, v), δ can be approximated by av^k, where a and k are constants. In that case, $\delta' = akv^{k-1}v'$ and so

$$E_D(\delta) = \frac{v' + \delta v' + v\delta'}{v'} - 1 = \delta + \frac{v}{v'}\delta' \approx \delta + \frac{v}{v'}akv^{k-1}v' = \delta + akv^k = (1+k)\delta.$$

Thus, unless δ cannot be piecewise approximated by polynomials of the form cv^k where k is a small constant, it can be assumed that $E_D(\delta) \approx (k+1)\delta$. (This condition can be checked by examining the values of δ in consecutive examples.)

Finally, the definition for F gives $F_{c,v_c}(u, v+\delta v) = (u, (v+\delta v - v_c)/(u-c))$. Thus,

$$E_F(\delta) = \frac{(v+\delta v - v_c)/(u-c)}{(v-v_c)/(u-c)} - 1 = \frac{\delta}{1 - v_c/v} = \frac{\delta}{1 - (1+\delta)\dfrac{v_c}{v+\delta v}}.$$

Based on this formula, we avoid choosing c so that (c, v_c) coincides with any example $(u, v+\delta v)$, since that would make $E_F(\delta) = -1$ for that example. This is not surprising because in that case we would be computing the indeterminate $0/0$. While the result of the transformation could still be estimated from nearby examples, the error analysis breaks down since $E_F(\delta) = -1$ effectively means that the value estimated is unreliable.

In general, the best choice for c is where $v_c = 0$, assuming that $(c, 0)$ is not one of the examples being transformed and that we can reliably estimate the zero-crossings of the function implied by those examples. This makes $E_F(\delta) = \delta$, which means that the initial error is not amplified by F. If that choice is impractical, we choose c so that $|v_c|$ is less than any v component of the examples. With this choice, $E_F(\delta)$ is bounded above by $\dfrac{n\delta}{n-1-\delta}$, where $1/n$ is the ratio between v_c and the smallest v component of the examples.

4.3 Computation error and the correctness of the solution

The analysis so far concerns only the propagation of error through each transformation. In practice, we need to take into account the additional error introduced by the inexact implementation of each transformation. While in principle we could derive the total error for any sequence of transformations, this is neither practical nor productive as an error tracking method.

In this section, we present an alternative method that LINUS uses to monitor the accumulated error and verify the solution's correctness without any need to know the total computation error. If formal verification is desired, one could then derive the necessary formula for the total error from the transformation sequence found. To begin, we define what it means for a set of examples to "support" a transformation, and give an outline of LINUS's discovery algorithm. The solution's correctness follows from LINUS's ability to request a set of examples that supports the transformation sequence needed to express the unknown function.

A definition of numerical accuracy

Definition 1 Let $S = \{\,(x, y)\,\}$ be a set of examples, $\epsilon > 0$ a specified error tolerance, and H a predefined class of approximation functions. For any transformation T, let $\bar{T}$ denote its (inexact) implementation.

1. S is said to be *approximated to within* ϵ by a function $h \in H$ if, for all $(x, y) \in S$,

$|h(x)| \leq \epsilon$ when $y = 0$ and $|h(x)/y - 1| \leq \epsilon$ when $y \neq 0$.

2. S is said to be *piecewise approximated to within* ϵ by a set of functions $h_i \in H$ if S can be split into several non-overlapping subsets S_i such that each S_i is approximated to within ϵ by h_i. The functions h_i are called the *local approximations* of S.

3. For any invertible transformation T, a set S is considered *numerically accurate for the application of* T *with respect to* ϵ if both S and $\bar{T}^{-1}(\bar{T}(S)) = \{ (\tilde{x}, \tilde{y}) \mid (\tilde{x}, \tilde{y}) = \bar{T}^{-1}(\bar{T}(x, y)) \}$ are piecewise approximated to within ϵ by the same set of local approximations in H.

In general, S can have many different sets of local approximations, any one of which will be acceptable. In practice, the local approximations are determined by partitioning S into non-overlapping subsets and, for each one, finding the function in H that best approximates it in terms of relative error magnitude. If no acceptable local approximations are found, we either repeat with a different partitioning of S or report that S can only be approximated to within a larger error tolerance.

Note that unless T and T^{-1} can be computed exactly, S is not guaranteed to be numerically accurate for the application of T. Since the implementation $\bar{T}$ and $\bar{T}^{-1}$ is likely to be inexact, this leads to some discrepancy between $\bar{T}^{-1}(\bar{T}(S))$ and S. Definition 1 proposes one particular test for acceptable discrepancy: Both the computed results and the original examples must satisfy the same set of local approximations within the error tolerance ϵ. If this condition is satisfied, we say that S "supports" the transformation T within the error allowed by ϵ.

Normally, ϵ should be chosen according to the accuracy desired for the solution, which is denoted by the parameter TOLERANCE in the algorithm below, and the relative magnitude of the error that may be present in the data. A typical choice for ϵ would be somewhere between these two. This ensures that the effect of inexact computation remains insignificant, and that numerical accuracy can be achieved even with relatively inaccurate data.

Linus's discovery algorithm

Let $Task(\mathcal{T}, \mathcal{T}(S_i))$ denote the problem of finding a description that approximates the examples in $\mathcal{T}(S_i) = \{ (u, v) \mid (u, v) = \mathcal{T}(x, y), (x, y) \in S_i \}$ to within a specified TOLERANCE. Note that the notation $\mathcal{T}(S_i)$ is used to denote both the application of the transformation sequence $\mathcal{T}$ to each example in the set S_i, and the set of transformed examples resulted from that application. The algorithm seeks a sequence $\mathcal{T}$ such that $\mathcal{T}(S_i)$ satisfies some predefined relation $P = 0$. The description $(\mathcal{T}, P)$ represents a potential solution to the discovery problem, and the algorithm is repeated to find other solutions as necessary.

1. Let $i = 1$ and $S_i = S_0$, where S_0 is the initial set of examples of the unknown function. Put $Task(\phi, S_i)$, where ϕ is the identity transformation $\phi(x, y) = (x, y)$, into the queue of unfinished tasks.

2. Choose one $Task(\mathcal{T}, \mathcal{T}(S))$ from the queue, and for each transformation T:

   2.1 If T cannot be applied to $\mathcal{T}(S)$, query for the appropriate examples that will allow its application, let $\mathcal{T}(S)$ be that set, and update the original untransformed set S_i.

   2.2 If $\mathcal{T}(S)$ is not numerically accurate for the application of T, query for additional examples, add them to $\mathcal{T}(S)$, update the set S_i, recompute the local approximations, and repeat this step until it is numerically accurate.

   2.3 Apply T to $\mathcal{T}(S)$, let the result be $T \circ \mathcal{T}(S)$, and use it as input to the pattern matching procedure.

   2.4 If any pattern P matches and if S_i is numerically accurate for the application of $T \circ \mathcal{T}$ with respect to TOLERANCE, put $(T \circ \mathcal{T}, P)$ into the queue of potential solutions, and go to Step 3 if no more solutions are needed, or repeat Step 2 for other solutions.

   2.5 Put $Task(T \circ \mathcal{T}, T \circ \mathcal{T}(S))$ into the queue of unfinished tasks and repeat Step 2.

3. Verify each potential solution $(\mathcal{T}, P)$ by querying for more examples and checking that they are approximated by $(\mathcal{T}, P)$ to within TOLERANCE:

3.1 If any description $(\mathcal{T}, P)$ fails, add the new examples to S_i, apply $\mathcal{T}$ to S_i, let the result be $\mathcal{T}(S_i)$, and place $Task(\mathcal{T}, \mathcal{T}(S_i))$ in the queue.

3.2 If all descriptions fail, repeat from Step 2. Otherwise, report each $(\mathcal{T}, P)$ that passes as a potential solution of the discovery problem, with respect to the example set S_i obtained so far.

4. If additional verification is desired, increase i by 1, let $S_i = S_{i-1}$, and repeat from Step 3.

Correctness of the solution

Given that inexact computation is inevitable, a description is considered correct if it approximates the examples to within the specified TOLERANCE. By the test in Step 3, the algorithm only produces solutions that satisfy this condition. Assuming that the implementation of each transformation is sufficiently accurate that the cumulative error is less than TOLERANCE, Steps 2.2 and 2.4 of the algorithm will always succeed. As a result, any solution found will approximate the examples available. The algorithm only restarts when Step 3 encounters *new* examples that are not approximated by the description found.

In practice, because of the use of fixed-precision arithmetic, it is possible that the algorithm could fail to select a set of examples to support a particular transformation or sequence of transformations. This would happen if, at some time during the discovery process, the accumulated error due to inexact computation exceeded the value of TOLERANCE. In this case the algorithm cannot continue without revising its expectation of solution accuracy (for example, by using a larger TOLERANCE) or employing higher-precision arithmetic.

Since there is a limit on how accurately we can compute, it is more practical to continue with a lower accuracy and eventually report an "approximate solution" that matches the currently available examples within a larger error tolerance. In a sense, TOLERANCE is the desired accuracy level, whereas the actual difference between examples of the description found and those of the unknown function is the level of accuracy obtained.

5 Extensions for practical application

In its simplest form, LINUS assumes that examples are available on demand and that the examples are error-free. Clearly, this ideal is rarely achieved in practice: Inevitably there is a limit to the number and accuracy of examples that are available. Consequently, LINUS has been extended to permit its application in passive environments, where requests for specific examples are not possible, and in noisy environments, where examples are contaminated with noise from some unknown process.

Discovery with a fixed set of examples

When the example set is fixed, LINUS computes the local approximations of the examples available and uses them to provide estimated values for other examples subsequently requested. In the current implementation, this is done by an interpolation routine embedded in the data selection procedure. The error of each estimate is derived from the fitting error of the local approximations and treated as a type of computation error. The overall discovery operation remains unchanged.

The usefulness of a solution depends on whether the available examples are sufficient to produce local approximations that closely resemble segments of the unknown function. For applications where sufficiently finely sampled examples are available, LINUS will find an acceptable solution. Otherwise, it simply provides a description for the set of examples given. In this case, LINUS is no worse than any application-independent discovery scheme that must work with a fixed set of examples.

Discovery with noisy examples

The current implementation of LINUS also includes a separate data-filtering routine, which is implemented as a special transformation that only applies when certain conditions are satisfied. Presently, data filtering will occur if (1) the user requests it by setting a system parameter, (2) LINUS detects abrupt changes among adjacent examples that cannot be explained by vertical asymptotes, or (3) the derivative it computes oscillates quickly over a short interval. At the present, the available filters include ones built from polynomials and rational functions, as well as a parameterized version of the low-pass Savitzky-Golay digital filter [4]. LINUS tries each filter and selects the one that produces the smoothest data.

Under this approach, LINUS discovers a description for the filtered examples, not a description for the noise-free examples of the unknown function. Since the latter are unavailable, statistical methods such as the χ^2 test [10] can be employed to assess the "goodness" of the description in approximating the unfiltered examples. If this assessment fails, LINUS can be instructed to use a different filter and repeat the discovery process.

It can be expected that if the noise is small, or if the filtering succeeds in reducing it to a small level, a solution will be found—although with a reduced accuracy. Otherwise, at a certain point in the process, LINUS will report that the accumulated error exceeds the desired accuracy, in which case the noise present is too great to allow successful discovery. In this case, data transformation is inappropriate as a discovery method for such data.

6 Summary

This paper presents a method for discovering functions that is based on the idea of transforming from one function to another. While the method is simple, its successful implementation depends critically on the transformations selected. In particular, it depends on whether the transformations can always be applied and on how accurately they can be computed. These are the problems of incomplete generation of function descriptions and inexact computation of transformations.

The paper shows how both problems are addressed by the discovery system LINUS. First, advantage can be taken of on-demand example selection based on the ongoing analysis of computation errors and the requirement of each transformation. Second, transformations can be specified that are robust in that they can all be applied in any situation, if necessary by splitting the problem into subproblems. Third, the accuracy of computations can be monitored and improved, where possible, after each transformation.

These ideas have been implemented in the function discovery system LINUS. The paper provides an analysis of Linus's ability in terms of its theoretical discovery power and its treatment of computation error both in theory and in practice. A full description of Linus and its capability can be found elsewhere [9]. Suffice to say that it has successfully discovered a large number of algebraic, trigonometric, exponential and logarithmic functions of varying degree of complexity, along with solutions to first- and second-order ordinary and autonomous differential equations.

Appendix

Let $y = f(x)$ be a function that can be described in $\mathcal{L}$ as $(T_k \circ \cdots \circ T_1, P(u,v))$, where T_i, $1 \le i \le k$, are the transformations in Table 1 and $P(u,v)$ is a formula in u and v. Furthermore, each transformation T has an *inverse* T^{-1} defined so that $T^{-1} \circ T(x,y) = T \circ T^{-1}(x,y) = (x,y)$. The inverse T^{-1} is not to be confused with the function inverse transformation Inv, which interchanges its two arguments.

Proposition 2 *Descriptions of the form $(\{R,F\}^*, v+c)$ describe all and only rational functions.*

PROOF: The proposition has two parts: (1) Any rational function f can be described using only R, F and a constant pattern $v+c$; and (2) No other function is described by this form.

For the second part, the description for f is first rewritten in terms of $R^{-1}(x,y) = (x, 1/y)$ and $F^{-1}_{(c,y_c)}(x,y) = (x, y*(x-c)+y_c)$. Solving the new system of equations

$$\begin{cases} (x,y) &= T_1^{-1} \circ \cdots \circ T_k^{-1}(u,v) \\ v+c &= 0, \end{cases}$$

where each T_i^{-1} is either R^{-1} or F^{-1}, gives a formula for y that can be simplified to a ratio of two polynomials.

For the first part, let p_i denotes a polynomial with degree i or less, and $f(x) = \dfrac{p_n(x)}{p_m(x)}$ be the rational function under consideration. Without loss of generality, assume that $n \ge m$. If $n < m$, the following will show that the reciprocal of f can be expressed in $\mathcal{L}$, and f can then be described with an additional reciprocal transformation.

Suppose that $f(a) = y_a$, where a is any value such that y_a finite. Then $f(a) - y_a = \dfrac{p_n(a)}{p_m(a)} - y_a = 0$, and so $p_n(a) - y_a p_m(a) = 0$. Since $p_n(x) - y_a p_m(x)$ is a polynomial with degree at most n and equal to 0 at $x = a$, it may be written as $(x-a)p_{n-1}(x)$, where $p_{n-1}(x)$ is a polynomial with degree at most $n-1$. Applying $F_{(a,y_a)}$ to $(x, f(x))$ yields

$$F_{(a,y_a)}\left(x,\ \frac{p_n}{p_m}\right) = \left(x,\ (\frac{p_n}{p_m} - y_a)/(x-a)\right) = \left(x,\ \frac{p_n - y_a p_m}{(x-a)p_m}\right) = \left(x,\ \frac{p_{n-1}}{p_m}\right).$$

If $n-1 \geq m$, the polynomial degree of the numerator p_{n-1} can be further reduced by other factoring transformations. If $n-1 < m$, a reciprocal transformation is applied to obtain

$$R\left(x,\ \frac{p_{n-1}}{p_m}\right) = \left(x,\ \frac{p_m}{p_{n-1}}\right).$$

In the rational function $\frac{p_m}{p_{n-1}}$, the numerator now has a higher degree than the denominator. This process of factoring and (when appropriate) taking the reciprocal continues until both numerator and denominator have degree 0, at which point the right-hand side matches a constant pattern. Let $T_k \circ \cdots \circ T_1$ be the accumulated sequence of transformations and c the negation of the final constant. The original function $f(x) = \frac{p_n}{p_m}$ can then be expressed as $(T_k \circ \cdots \circ T_1, v + c)$, where each T_i is either R or F. $\Box$

Note that R and F can be applied alternately without any need to know the degrees of the numerator and denominator. In the worst case, applying F when the degree of the denominator exceeds that of the numerator merely results in the new numerator having the same degree as the denominator.

Proposition 3 *A function $y = f(x)$ that satisfies a quadratic relation $a_{0,2}y^2 + a_{1,1}xy + a_{2,0}x^2 + a_{0,1}y + a_{1,0}x + a_{0,0} = 0$ can be transformed to a rational function $v = g(u)$ that satisfies the relation $h_1(u)v + h_2(u) = 0$, where h_1 and h_2 are polynomials of degree at most 2.*

PROOF: Consider the transformation sequence $Inv \circ F_{(x_0,y_0)}$, where x_0 is chosen so that y_0 is finite. With this choice, $Inv \circ F_{(x_0,y_0)}(x,y) = (\frac{y-y_0}{x-x_0}, x)$, and $a_{0,2}y_0^2 + a_{1,1}x_0y_0 + a_{2,0}x_0^2 + a_{0,1}y_0 + a_{1,0}x_0 + a_{0,0} = 0$. To prove the proposition, we show that $u = \frac{y-y_0}{x-x_0}$ and $v = x$ satisfy the relation $h_1(u)v + h_2(u) = 0$. In other words, the constants b_0, b_1, b_2, c_0, c_1 and c_2 in $h_1(u)v + h_2(u) = (b_0 + b_1u + b_2u^2)v + c_0 + c_1u + c_2u^2$ can be selected so that

$$\begin{aligned}(b_0 + b_1u + b_2u^2)v + c_0 + c_1u + c_2u^2 &= \sum_{i=0}^{2}(b_iv + c_i)u^i = \sum_{i=0}^{2}(b_ix + c_i)\left(\frac{y-y_0}{x-x_0}\right)^i \\ &= (x-x_0)^{-1}\sum_{i=0}^{2}(b_ix + c_i)(y-y_0)^i(x-x_0)^{1-i}\end{aligned}$$

equals zero for all examples of the function $y = f(x)$.

This condition is satisfied by the values

$$\begin{aligned} b_0 &= a_{2,0}, & c_0 &= a_{1,0} + a_{2,0}x_0 + a_{1,1}y_0, \\ b_1 &= a_{1,1}, & c_1 &= a_{0,1} + 2a_{0,2}y_0, \\ b_2 &= a_{0,2}, & c_2 &= -a_{0,2}x_0, \end{aligned}$$

where $a_{i,j}$ are the coefficients in the relation $a_{0,2}y^2 + a_{1,1}xy + a_{2,0}x^2 + a_{0,1}y + a_{1,0}x + a_{0,0} = 0$ that f satisfies, and (x_0, y_0) is the parameter in the F transformation selected. $\square$

Proposition 4 *Descriptions of the form $(\{R, F\}^* \circ D^*, v+c)$ encompass all and only functions in this class:*

$$\begin{aligned} y \;=\; & p_k + \sum_i c_{1,i} x^{n_{1,i}} \log(a_{1,i}x + b_{1,i}) + \sum_i c_{2,i} x^{n_{2,i}} \tan^{-1}(a_{2,i}x + b_{2,i}) \\ & + \sum_i \frac{c_{3,i}}{(a_{3,i}x + b_{3,i})^i} + \sum_i \sum_j \frac{a_{4,i,j}x + b_{4,i,j}}{(h_i)^j}; \end{aligned}$$

where $n_{1,i}$, $n_{2,i}$ are non-negative integers; p_k is a polynomial with degree at most k; each h_i is an irreducible polynomial of degree 2; each of $a_{i,j}$, $a_{4,i,j}$, $b_{i,j}$, $b_{4,i,j}$, and $c_{k,i}$ is a real constant; and each $\sum_i$ represents a finite summation of similar terms.

PROOF: This proposition has two parts: (1) Any function of the above form can be described as $(\{R, F\}^* \circ D^*, v + c)$; and (2) No other function is described by this form. The first follows from Proposition 2 and the fact that any function of the above form can be transformed to a rational function by a sufficient number of differentiations. For the second part, let $f(x)$ be the function described as $(\{R, F\}^* \circ D^*, v + c)$. That means,

$$(x, f(x)) \;=\; \{\, D^{-1}\,\}^* \circ \{\, R^{-1}, F^{-1}\,\}^*(x, -c) \;=\; \{\, D^{-1}\,\}^*(x, g(x)),$$

where $g(x)$ is a rational function (by Proposition 2). Using the technique of partial fraction expansion [10], g can be written as a finite summation of the form

$$g(x) \;=\; p_n + \sum_i \frac{c_{3,i}}{(a_{3,i}x + b_{3,i})^i} + \sum_i \sum_j \frac{a_{4,i,j}x + b_{4,i,j}}{(h_i)^j}.$$

Since each term in the formula for g can be integrated separately, it is straightforward to apply the rules of indefinite integration to show that repeated integration of any term in g will only lead to terms shown in the proposition. $\square$

Proposition 5 *If a function g is expressible in $\mathcal{L}$, then any function $y = f(x)$ that satisfies one of the following relations is also expressible in $\mathcal{L}$:*

1. *$g(y) = x$, $g(x) = \log y$, $g(y) = \log x$, $g(\log y) = x$, $g(\log x) = y$, $g(\log x) = \log y$, $g(\log y) = \log x$; and*

Implicit form	Explicit form $y = f(x)$	Transformation sequence
$g(y) = \log x$	$y = g^{-1}(\log x)$	$L \circ Inv$
$g(\log y) = x$	$y = e^{g^{-1}(x)}$	$Inv \circ L$
$g(\log x) = y$	$y = g(\log(x))$	$Inv \circ L \circ Inv$
$g(\log x) = \log y$	$y = e^{g(\log(x))}$	$L \circ Inv \circ L \circ Inv$
$g(\log y) = \log x$	$y = e^{g^{-1}(\log x)}$	$L \circ Inv \circ L$

Table 3: Transformation sequences for some transcendental functions

2. *Solutions of differential equations of form either $y' = yg(x)$ or $y' = g(y)$, where y' is the derivative of y with respect to x.*

PROOF: For the first part, let $(\mathcal{T}, P)$ be the description for the function $g(u) = v$. In the first two cases, $g(y) = x$ and $g(x) = \log y$, the transformation sequences needed to describe $f(x)$ are $\mathcal{T} \circ Inv$ and $\mathcal{T} \circ L$ respectively. In conventional terms, the new functions are the inverse and exponential of g; they can be written as $f(x) = g^{-1}(x)$ and $f(x) = e^{g(x)}$ respectively. The remaining five cases are summarized in Table 3. The abbreviated description of each of these functions is $(\mathcal{T} \circ \mathcal{T}', P)$, where $\mathcal{T}'$ is the transformation sequence shown.

For the second part, consider the transformation sequences $D \circ L$ and $R \circ D \circ Inv$. By definition,

$$D \circ L(x, y) \quad = \quad D(x, \log y) \quad = \quad \left(x, \frac{d \log y}{dx}\right) \quad = \quad (x, y'/y),$$

and

$$\begin{aligned} R \circ D \circ Inv(x, y) \quad &= \quad R \circ D(y, x) \quad = \quad R(y, dx/dy) \\ &= \quad \left(y, \frac{1}{dx/dy}\right) \quad = \quad (y, dy/dx) \quad = \quad (y, y'). \end{aligned}$$

Consequently, solutions $y = f(x)$ of ordinary differential equations of the form $y' = yg(x)$ can be written as $(\mathcal{T} \circ D \circ L, P)$, where $(\mathcal{T}, P)$ is the description of g. Similarly, solutions $y = f(x)$ of autonomous differential equations of the form $y' = g(y)$ can be expressed as $(\mathcal{T} \circ R \circ D \circ Inv, P)$, where $(\mathcal{T}, P)$ is the description of g. $\square$

References

[1] Dzeroski, S. and Todorovski, L. (1993). "Discovering Dynamics." *Proc. International Conference on Machine Learning.* California: Morgan Kaufmann.

[2] Falkenhainer, B.C. and Michalski, R.S. (1990). "Integrating quantitative and qualitative discovery in the ABACUS system." In *Machine Learning: An Artificial Intelligence Approach, Volume III.* California: Morgan Kaufmann.

[3] Gerwin, D.G. (1974). "Information processing, data inferences, and scientific generalization." *Behavioral Science*, **19**(5): 314–325, University of Louisville, Kentucky.

[4] Hamming, R.W. (1983). *Digital Filters*, Second edition. Englewood, New Jersey: Prentice-Hall.

[5] Kokar, M.M. (1986). "Determining arguments of invariant functional descriptions." *Machine Learning*, **1**: 403–22.

[6] Langley, P. and Zytkow, J. (1989). "Data-driven approaches to empirical discovery." *Artificial Intelligence*, **40**, 283–312.

[7] Langley, P., Zytkow, J.M., Simon, H.A. and Bradshaw, G.L. (1986). "The search for regularity: Four aspects of scientific discovery." In *Machine Learning: An Artificial Intelligence Approach, Volume II.* California: Morgan Kaufmann.

[8] Nordhausen, B. and Langley, P. (1990). "A robust approach to numeric discovery." *Proc. International Conference on Machine Learning.* California: Morgan Kaufmann.

[9] Phan, T.H. (1994). *Function Discovery using Data Transformation.* PhD dissertation, University of Calgary, Alberta.

[10] Råde, L. and Westergren, B. (1990). *BETA: Mathematics Handbook*, Second Edition, Boca Raton: CRC Press.

[11] Rao, R.B. and Lu, S.C.Y. (1992). "A knowledge-based equation discovery system for engineering domains." *Proceedings of the Eighth Conference on Artificial Intelligence for Applications.* California: IEEE Press.

[12] Schaffer, C. (1990). "A proven domain-independent scientific function finding algorithm." *Proc. AAAI Conference*. California: AAAI Press.

[13] Wong, P. (1991). *Machine Discovery of Function Forms*. PhD dissertation, University of Waterloo, Ontario.

[14] Wu, Y.H. and Wang, S.L. (1989). "Discovering knowledge from observational data." *Proc. IJCAI Workshop on Knowledge Discovery in Databases*. California: Morgan Kaufmann.

[15] Zembowicz, R. and Zytkow, J.M. (1991). "Discovery of Equations: Experimental Evaluation of Convergence." *Proc. AAAI Conference*. Massachusetts: MIT Press.