Supporting Numerical Computations in Interactive Contexts [*]

Michael Gleicher
Andrew Witkin
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
USA

Abstract

As computational performance becomes more readily available, there will be an increasing variety of interactive graphical applications with iterative numerical techniques at their core. In this paper, we consider how to support the unique demands of such applications. In particular, we focus on how to set up the numerical problems which must be solved. In the context of interactive systems, this requires the ability to dynamically compose systems of equations and rapidly evaluate them and their derivatives. We present an approach called *Snap-Together Mathematics* for doing this.

Keywords: interactive systems, automatic differentiation, numerical methods, constraints

1 INTRODUCTION

Continual advancements in computer hardware are making floating point and graphics performance more accessible. These capabilities are making possible new classes of interactive graphical applications involving animation, continuous motion, and direct manipulation. Such systems will be able to employ techniques such as constraints, physical simulation, optimization, and differential control.

These new classes of applications have a lot in common. All do significant numerical calculations at their core. Many techniques used in these interactive graphical applications, such as non-linear system solving, constrained optimization, differential manipulation, and mechanical simulation are similar: they involve repetitively setting up and solving systems of linear equations. Often however, the user is insulated from the mathematics. Interfaces present the user with graphical objects and geometric relations, it is the job of the system to translate these into optimization objectives, constraint functions, and equations of motion.

There are many challenges in employing numerical techniques in interactive settings. The most obvious is performance: a system must keep up with interactive rates. Because we are interested in interaction and animation instead of quantitative prediction, we are often willing to sacrifice some accuracy for performance. Other common concerns of numerical analysis, such as stability and reliability are still crucial — users would probably not be happy if the application failed to converge.

Possibly the most unique set of demands that interactive systems place on numerical methods stems from their dynamic nature. Not only do applications repeatedly solve numerical problems, but the structure of these problems is continually changing. These changes often occur in response to user actions such as creating objects and constraints, or internal system events such as collisions.

In this paper, we consider the problem of supporting such interactive applications with numerical computations at their core. The rich literature in numerical analysis provides a variety of techniques for solving the systems of equations which arise. Therefore, we focus on the unique issues of employing these algorithms in interactive contexts. In particular, we concentrate on setting up and managing the equations which the numerical algorithms will solve. Systems must be able to rapidly and dynamically define functions, efficiently evaluate them and their derivatives, and manage sets of them and variables. This paper describes an approach to providing these functionalities in an object-oriented, general purpose manner.

1.1 An Example Application

To introduce the issues which the paper will address, consider a constraint-based drawing program as an example application. Constraint-based drawing programs use equation solving to maintain geometric relationships among objects. The use of such an approach dates back to the earliest interactive graphical applications[29]. Our *Briar* drawing program[9] attempted to address many of the issues in applying the approach[10].

Like a traditional drawing program, Briar's interface is purely graphical. A user draws with direct manipulation, just as in more conventional systems. Briar provides

[*] To Appear at Graphics Interface '93.

Snap-Dragging[3], a non-constraint-based technique to help users produce precise drawings easily. Constraints are created for the relationships which are created by Snap-Dragging operations. As the user drags objects in the drawing, these relationships are maintained. Briar employs a visual representation for constraints so that the user is never confronted with any equations. In fact, the user never even refers directly to constraints — constraints are only created and destroyed in response to direct manipulation dragging operations on the graphical objects in the drawings.

Briar exemplifies the type of application *Snap-Together Mathematics* is meant to support. At its core is a constrained optimization solver which is repeatedly called as the user manipulates objects. In order to maintain the illusion of continuous motion required for direct manipulation, these equations must be solved many times per second.

Constraints in Briar are continually being added and deleted as the user manipulates objects. The set of equations which Briar uses to represent them, and must solve to maintain them, is, therefore, also in constraint flux. The user, however, is insulated from this mathematical machinery. The interface shows graphical objects and geometric relationships. The underlying mathematical support must face the challenges of creating the corresponding equations dynamically as the model evolves and solving these equations rapidly. This paper discusses how such support can be provided in a general manner.

2 RELATED WORK

The class of graphical applications which use iterative application of numerical techniques at their core is expanding with the availability of computational performance to realize them. For example, the availability of this performance not only facilitates techniques for physically-based animation, such as [1, 22], but also causes them to evolve toward techniques for interaction simulation, such as[25, 33]. It also makes such physical simulation viable as an interactive modeling tool, as [2, 21].

Related to the methods of physical simulation are those of constrained optimization. Performing these computations at interactive rates permits using these techniques for interaction and animation problems, such as modeling free-form surfaces[8, 32], experimenting with molecular structures[28], solving physical motion control problems[34], positioning virtual cameras[13], and exploring toleranced behavior[24].

Since the earliest interactive graphical systems[29], constraints have been used to aid in the manipulation of geometry. From the early systems, numerical techniques to solve these constraints have been employed. Modern constraint-based systems, such as [4, 9, 19, 26], employ iterative numerical techniques. These non-linear system solving techniques, like the techniques for physical simulation and constrained optimization, all rely on repeatedly setting up and solving systems of linear equations based on the derivatives of model functions.

All of these numerical techniques rely on the availability of the derivatives of the mathematical functions which define the constraints, optimization objects, and objects to be simulated. The most effective means for computing derivatives is a process called *automatic differentiation* [14]. The vast majority of automatic differentiation work is concerned with computing the derivatives of functions defined at compile time[15]. This paper describes a variant of automatic differentiation which is designed to meet the dynamic needs of interactive systems.

While there is an extensive literature on the numerical techniques for constraint and simulation problems[1], there is much less discussion on how to set such problems up. Interactive systems must be able to dynamically compose functions and evaluate them and their derivatives. For this task, some authors describe implementing function composition schemes similar to what is described in this paper[18, 24, 34], however such implementations rarely provide general purpose tools. In this paper, we describe encapsulating mathematical support into a toolkit so that the variety of applications which demand these services can be supported. An early implementation of these ideas is described in [12].

3 REPRESENTING AND EVALUATING FUNCTIONS

Algebraic expressions can be represented as directed acyclic graphs. The leaves of the graph are the variables and constants. The nodes, which we call *function blocks*, represent the primitive mathematical functions. The edges of the graph represent function composition. An expression graph is not necessarily a tree because many nodes might refer to a given node. Such sharing is the result of a common subexpression and is common in the kinds of applications we are considering. For example, the function which computes the position of the end of a linkage rod in a simulation would be a shared sub-expression of anything that was connected to that point. Exploiting this sharing is important for both performance and simplicity.

We call the approach of providing tools for creating and evaluating expression graphs *Snap-Together Mathematics* . Some previous systems, such as [7] and [17], explicitly present the expression graph to the user. In such systems, users graphically manipulate graphs to edit functions. While such applications may be built with the tools described here, we have concentrated on constructing applications, such as the drawing program of section 1.1, in which the mathematics is only kept internally.

[1]See [23] for a practical introduction.

3.1 Evaluating Functions

Evaluation is probably the most important computation to be performed on expression graphs. In the interactive applications which we are considering, expression graphs will be evaluated repeatedly, so performance is critical. The most efficient way to repeatedly evaluate an expression is to compile it into machine code. Unfortunately, compiling and linking code for each dynamically created expression is prohibitively expensive in the programming environments presently available.

Other approaches to evaluating the expression graph are interpretive: traverse the graph for each evaluation. Each node of the graph computes its output value, given the values of its inputs. A set of primitive function elements are predefined at compile time to do this. Evaluation of a node involves asking its predecessors for their output values then computing the "local" function of the node.

Performance can be enhanced by using caching to exploit two types of redundancy: within an evaluation, common subexpressions need only be evaluated once (these subexpressions may be shared within one expression or between different expressions); between evaluations, certain old values might still be correct if some of the inputs did not change. Recomputation can be avoided by storing the results of a calculation and, for a later request, deciding whether this stored value is still correct. There are many possible ways to implement this cache validation; elaborate schemes might avoid some recomputation, but will require additional computation and storage to make the determination. The more expensive the evaluations become, the more effort it is worth avoiding excess evaluations.

3.2 Evaluating Derivatives

For many of the applications we are considering, we will need to be able to evaluate the derivatives of expressions with respect to some subset of their inputs. Although the techniques extend to higher derivatives, for this discussion, we will consider computing first derivatives since this is what the majority of the methods require. We will call the vector of variables which we are taking the derivatives with respect to the *working set*, and denote it as $\mathbf{w}$. For a vector expression $\mathbf{f}$, the Jacobian, or first derivative, $\mathbf{J}$ is the matrix $\partial\mathbf{f}/\partial\mathbf{w}$. In this matrix, each row corresponds to an element of $\mathbf{f}$, while each column corresponds to a variable in $\mathbf{w}$.

There are three basic approaches to computing derivatives: approximate them numerically, derive a symbolic expression for the derivatives, or compose them using a process called *automatic differentiation*. The latter approach has been shown to be superior both in performance and precision of the results[14].

To understand the process of automatic differentiation, consider how derivatives are computed manually. The chain rule allows us to decompose complicated functions into smaller pieces. For example, if our expression is $f = f(a, b, \ldots)$, then the chain rule yields

$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial a}\frac{\partial a}{\partial w} + \frac{\partial f}{\partial b}\frac{\partial b}{\partial w} + \cdots. \qquad (1)$$

Differentiation involves recursive applications of the chain rule. If we are able to evaluate the derivative of each of the primitive functions with respect to their inputs then we can apply Equation 1 recursively to build the compound expressions. The recursion bottoms out at the constants, whose derivatives are 0, and at the variables, whose derivatives are 1 with respect to themselves and 0 with respect to others.

Symbolic differentiation applies the chain rule to an expression graph to transform it into a new expression which evaluates the derivative. The resulting expression must then be simplified to take advantage of the sparsity of the derivatives. Even then, the symbolic differentiation of a vector with respect to a vector yields a matrix of *expressions* which is unwieldy to manage.

Automatic differentiation also applies the chain rule to expressions; however, rather than symbolically composing more complicated expressions, the intermediate results are combined numerically. For any node in the graph, if the inputs to equation 1 are concatenated into a vector, the equation multiplies two matrices: the "local" Jacobian of the outputs with respect to the inputs, and the derivatives of the inputs with respect to the working set.

We implement automatic differentiation by augmenting the expression graph with the ability to pass derivatives as well as values along edges. In addition to to computing its output values, each node of the expression graph must also be able to compute the value of its local derivative, also a function of its inputs. The composition process builds the "global" Jacobian by multiplying this matrix with intermediate result matrices. By passing the entire intermediate result matrices along the edges of the graph, the derivative matrix can be built in one traversal of the expression graph. The same mechanisms for sharing intermediate results by caching as discussed in the previous section apply.

A recursive descent of the expression graph computes the derivative matrix. Each node in the graph is able to respond to requests for the derivative of its output with respect to the current working set. Constants and variables not in the working set return zero in response to this query. A state variable in the current working set returns a vector with one in the position corresponding to the variable, and zeros elsewhere. After determining that its cached value is not valid, a non-terminal node recursively asks its children for their derivatives, computes its local Jacobian, and multiplies these together to produce its derivative with respect to the current working set. Figure 1 demonstrates a simple example. Edges of the expression graph pass not

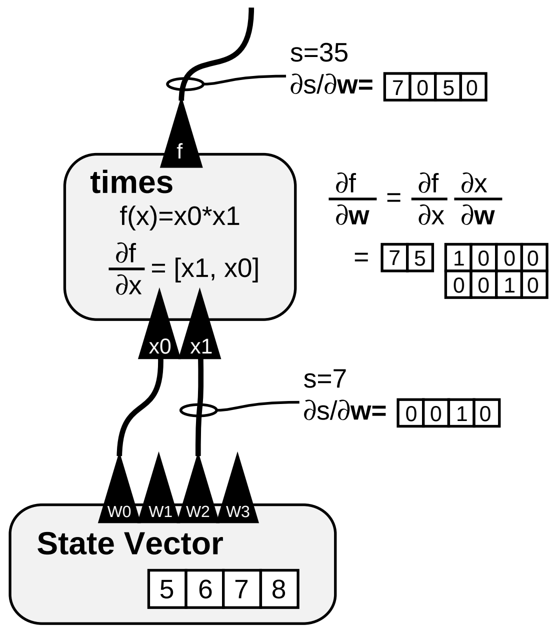

Figure 1: A simple example of derivative composition. Signals carry both values and their derivatives. The function block computes its internal Jacobian and composes the global Jacobian by multiplying the matrices.

only values, but also their derivatives.

This method of automatic differentiation assembles the Jacobian bottom-up, and is called the forward-mode. The alternative reverse-mode, or top-down, approach is presented by [14] and implemented for an interactive system by [24]. This algorithm reverses the order of the matrix multiplies, building the Jacobian matrix from the top down. It has the advantage that the intermediate result matrices are of small, fixed size. In the bottom-up approach, the size of intermediate matrices depends on the number of variables which contribute to that derivative. Because the intermediate results are fixed-sized, the top-down approach can achieve linear asymptotic complexity in places where the bottom-up approach has $O(n^2)$ complexity. However, this increased worst-case performance on dense problems comes at the expense of considerable bookkeeping, inability to fully exploit sparsity, inability to share intermediate results, much higher time constants, and difficulty in changing working sets.

It is important to recognize the generality of either of these derivative composition processes. Each node of the expression graph need only to be able to compute its *local Jacobian,* the derivative of its outputs with respect to its inputs. This matrix is a function only of the input values, not their derivatives. Given the local Jacobians, the composition process merely multiplies the matrices together to build the global derivatives.

3.3 Sparse Representations

The critical performance issue in building the Jacobian matrix, as well as most calculations that use this result, is exploiting sparsity. Bottom-up matrix passing schemes exploit sparsity by using sparse representations for the intermediate matrices. There are many possible ways to represent a sparse matrix, with many tradeoffs to consider in selecting a representation [6]. This decision is central to the design of an implementation. One particular representation with which we have had success is the half sparse matrix: a full vector of sparse vectors. We call a system based on these data structures a *sparse vector* scheme.

In the sparse vector scheme we consider every output in the expression as an independent scalar, even if higher levels will interpret them as pieces of larger structures. A function block can have multiple scalar outputs. The gradient of each scalar is a sparse vector ($\partial x/\partial \mathbf{w}$).

Sparse vectors can be represented as a list of pairs $(index, value)$, taking space linear with the number of non-zero elements. If this list is sorted by index, we can perform the essential vector operations in time linear to the number of non-zero elements. For single vector operations, such as multiplying by a scalar or finding the magnitude, the algorithms simply run through the list. For multi-vector operations, such as addition, linear combination, or dot product, we exploit the sortedness of the lists and step through both in parallel, advancing which ever has the least index. These algorithms maintain the representation invariant so sorting is not needed.

Each derivative in the expression graph is represented as a sparse vector, the derivative of a scalar with respect to a set of variables. For each graph, one set of variables is denoted as current: all derivatives are with respect to this set. Each set of variables also must contain a mapping to the corresponding column of the Jacobian.

Sparse vectors are collected into matrices which are half-sparse. While this is an unusual representation, it does permit the operations required by the numerical methods we employ. In particular, it can be rapidly multiplied by a vector or by its transpose, which are the essential computations in iterative linear system solvers like conjugate gradient[23].

4 OBJECT ORIENTED FUNCTION COMPOSITION

The graph oriented view of function leads to an obvious encapsulation. The primary computations at any internal node of the graph are inherently local: given the inputs to the node, compute the output or local Jacobian. Composition can be provided by a general purpose service, either external to the graph objects themselves, or by the generic classes of graph objects.

The most obvious realization of the function composition encapsulation is to create special objects for the major abstractions such as function blocks, variables, and constants. Although such a library is very easy to build and extend, it creates a tendency to have separate mathematical and conceptual models. For example, a mechanical simulator might keep a separate description of the mechanical object in addition to a expression graph which computed the equations required for the simulation. The user would operate on the mechanical model which in turn updates the mathematical model. Our experience was that maintaining these multiple representations complicated the development of systems.

To alleviate some of the complications of managing multiple representations, we avoid defining separate objects for graph entities. Instead, we add the ability to behave as mathematical expressions to application objects. For example, objects in a geometric modeler which represented points in space might add the ability for these points to serve as "mathematical outputs." This is possible because for an object to "speak mathematics" it only needs to respond to a few additional types of queries.

We call an object which can be used as a node in an expression graph, e.g. can have other nodes look at its output, a *port*. The only required protocol to realize the gradient passing scheme is for port objects to be able to respond with its output values and the gradients of these values with respect to the current working set. It is very easy to mix this behavior into other applications, or to extend this protocol to include other evaluations which can be computed by composition such as interval arithmetic.

4.1 Connectors

The simple port protocol provides the minimum of information for realizing function composition, evaluation and derivative evaluation. Some additional information about ports is also often useful. An obvious example is knowing the number of scalar outputs of a port. Providing names and nominal value ranges for each output provides the possibility of automated graphical display like meters, or even slider-based input[2].

One of the most useful additions to ports is typing information. For example, providing a port with the type "point in 3 space" means that it represents the position of a point in space and that it will have 3 outputs corresponding to its cartesian coordinates. Types provide a method of grouping signals together and checking types can help avoid signal mismatching. Types also add structure which can make it easier to hide the underlying mathematics.

We call a typed port a *connector.* Using connectors permits generic interfaces between layers of systems. For example, graphical constraints typically involve points on the parts they relate. By formulating the constraints as functions of point connectors, they can be defined independently of the kinds of objects they relate. Similarly, parts need only be defined to have point connectors on them, they need not know about the kinds of constraints which will be placed on them. Figure 2 shows how points on objects and constraints are coupled merely by the common protocol of the connector. Such a design permits building systems with extensible sets of parts and relationships among them. A more restrictive notion of connectors, with similar benefits, is presented in [16].

The choice of what types of connectors to provide determines what kinds of generic interfaces will be possible in systems. Positions of points are a generally useful class. Adding orientation information, such as normals, to point connectors is useful in a variety of drawing geometric modeling situations. Sometimes, more domain specific connector types are needed. For example, in our work on camera control[13], we defined a connector type which was the location a point appeared on the image plane with a given camera, and objects in the Briar drawing program described in section1.1 had connectors which provided a standardized form of the line passing through their major axis.

4.2 Collecting Variables

In defining our simple protocol for *Snap-Together Mathematics* one issue which we have not yet specified is the set of variables which we are taking derivatives with respect to. This is indicative of the larger issue of managing collections of variables. On one hand, building systems in an object-oriented manner requires the state of the system distributed with the objects themselves. But, mathematical algorithms typically require this state in the form of vectors, which are gathered, ordered collections. This ordering also gives meaning to the columns of the Jacobian matrices.

We have experimented with many representations of variables in our system, ranging from having objects allocate space in a global state vector to modifying our numerical algorithms so that they operate on distributed vectors. What has worked best for us is a combination of centralized and distributed representation. Objects each have their own state, however these variables are "gathered" into a centralized state vector for numerical computations. When an object's variable has been gathered, it knows where in the global vector to find it so it can still retrieve its value as well as index it for creating derivatives. In the context of *Snap-Together Mathematics* , derivatives can only be taken when variables are gathered as this is the only time when variables correspond to matrix columns.

The ability to scatter and gather variables has an important advantage over always keeping the variables centralized. It allows for the set of variables to be changed rapidly. This not only simplifies adding and deleting ob-

[2]Using the techniques of section 5.1, outputs may be used as controls.

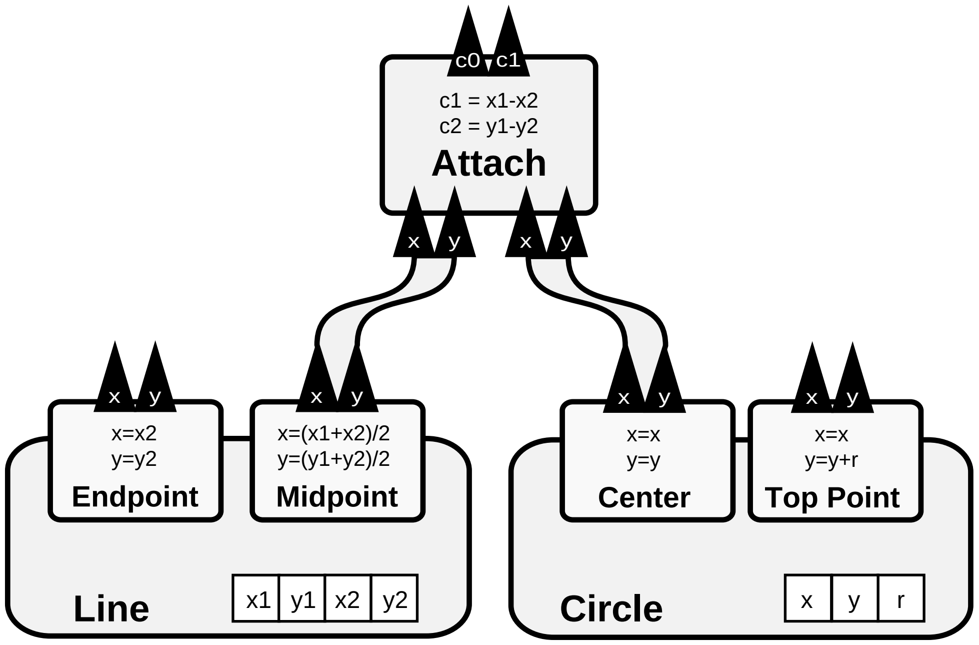

Figure 2: Connectors permit defining constraints independently of the objects they relate. In the example, the attachment constraint is defined in terms of generic 2D points.

jects, but also things like selecting sets of objects to be active and switching off certain variables which we do not want to change. Sharing a space in the collected vector provides a fast and simple way to constrain two variables to be equal.

The ability to selectively gather variables helps achieve performance in interactive constraint systems. Because there is little that can be done about the computational complexity of solving algorithms, it is important to try to minimize the size of the problems which are solved, without placing undue restrictions on the user. For example, a system might only gather variables which are likely to change on the current iteration and only solve the subset of constraints which act on these variables. A related technique is to partition the constraint problem into several smaller problems, which is examined extensively by [26].

5 IMPLEMENTATION

Our efforts to build general purpose *Snap-Together Mathematics* tools began with our original efforts to construct interactive systems[33]. Our early implementations are discussed in [12]. Experience using these tools has caused them to evolve into what has been described here. Our present toolkit, written in C++, implements sparse vector passing, scatter and gather variable collections, and a simple timestamping scheme for cache validation. We have tools which automatically generate function block code from mathematical expressions.

Snap-Together Mathematics is built on top of a more general set of mathematical tools. Our library provides interval arithmetic, vector and matrix computations, sparse matrix operations, and linear system, non-linear system, and ordinary differential equation (ODE) solving. The entire toolbox is written with object-oriented interfaces, permitting such things as switching ODE solvers on the fly and writing matrix routines independently of representations so the provided set of sparse matrix types can easily be extended.

With *Snap-Together Mathematics* and our low level mathematical toolkit, we have built a tool for experimenting with constrained optimization. The tool allows constraints and optimization criteria to be dynamically defined. Expressions can quickly and easily be added or removed from the set of constraints and objectives. This dynamic performance is what enables several of the interaction techniques we have experimented with, such as bounding variables and collisions.

This constrained optimization tool and *Snap-Together Mathematics* serve as the core for a higher level interactive graphics toolkit called *Bramble.* Bramble provides the typical abstractions provided by toolkits, such as graphical objects, display lists, views, and cameras. It is designed with the intent that applications programmers should be able to deal with this layer of abstraction, rather than that of optimization objectives and Lagrange multipliers.

What separates Bramble from similar systems such as Unidraw[30], Inventor[27], or UGA[5] is that it is built with mathematical techniques for manipulation at its heart. Providing for the needs of the differential approach permeates the more traditional things which interaction toolkits do. For example, transformation hierarchies build functional representations and viewing parameters are stored as state variables. While these typically happen behind the back of the application programmer, their existence makes interesting interaction techniques available. The previous examples enable differential inverse kinematics and through-the-lens camera controls respectively. Many of the quantities computed within the graphics library, such as the locations of shadows and the results of lighting computations, are available as *Snap-Together Mathematics* outputs and can, therefore, be constrained and controlled.

In support of Bramble, we have developed an embedded interpreter that serves as an extension language, a user interface description language, a save and load format, a rapid prototyping facility, and a debugging aid. Although similar to Tcl[20] in intent and implementation, *Whisper* is more akin to Scheme and Lisp in syntax and semantics. Whisper's extensible type system includes *Snap-Together Mathematics* classes, permitting easy, dynamic definition of mathematical functions.

5.1 Differential Techniques

Bramble uses constrained optimization to provide mappings from user controls to the parameters of objects. In short, we aim to provide the user with direct control of some aspect of a geometric model, for example the position of a point on a curve. Since all aspects of a model must be determined by the model's state vector, the desired control must be a function of these parameters. However, we cannot simply specify the outputs of these functions because they are often non-linear and underdetermined. We instead specify the motion (time derivative) of the control as it is a linear function of the time derivative of the parameters. The non-linear controls on the parameters form linear constraints on the time derivatives of the parameters. Non-linear constraints on parameters also form linear constraints on the time derivatives. An optimization objective specifies what happens to unconstrained degrees of freedom. We call this technique *differential manipulation*, and discuss it in detail in [11, 13].

In terms of implementation and numerical needs, differential techniques are similar to other techniques for non-linear constraints and control such as as Newton-Raphson solvers[23]. The central calculation is solving a system of linear equations formed with the Jacobian of the constraint equation. In solving these equations, it is important to exploit sparsity to achieve interactive performance and scalability. Because of the dynamic nature of the problems, systems cannot extensively pre-analyze the sparsity, as done in [28]. Instead, like [22] and [26], we use a conjugate gradient solver which is an iterative method and therefore allows some control of the tradeoff between performance and accuracy. We also use damping techniques, similar to those discussed in [31] or used in the Levenberg-Marquardt Method[23], allowing us to trade accuracy for stability and performance in cases of conflicting or redundant constraints.

A straightforward application of differential manipulation is interaction with parametric curves. Suppose all we knew about a curve is its parametric function, e.g. $(x, y) = \mathbf{f}(\mathbf{q}, u)$. This is sufficient information to draw the curve. With differential manipulation, this is also sufficient information to interactively manipulate the curve by controlling positions of points on it. Even if the function which computes the position of a point from the parameters is non-linear, we can control the motion of the point and compute how the parameters must change to achieve this motion by solving a linearly constrained optimization problem. The same mechanism is used to place constraints on the parametric curves.

Differential techniques make it possible to have an extensible system for manipulating parametric curves. To introduce a new type of curve, only its parametric function need be provided. Drawing and manipulation can be handled by generic routines. In [11] we describe a system which permitted addition of new curve types by an automatic compilation process. With Bramble and Whisper, we have constructed a similar system which allows the user to define a new variety of curve on the fly.

6 CONCLUSION

Our implementation of *Snap-Together Mathematics* has served as a platform on which we have built an array of interactive graphical applications including drawing, scene composition, physical and mechanical simulation, motion control, and animation. Experience in building these applications has led to a highly evolved toolkit.

By encapsulating the often needed functionality of dynamically composing mathematical expressions and rapidly evaluating them and their derivatives we have created a substrate on which to build a variety of interactive applications. As we continue to experiment with applying iterative numerical techniques to more interaction problems, we will continue to gain from having such a tool.

References

[1] D. Baraff. Analytical methods for dynamic simulation of non-penetrating rigid bodies. In *Computer Graphics* (*Proc. SIGGRAPH*), volume 23, pages 223–232. ACM, July 1989.

[2] Ronen Barzel and Alan H. Barr. A modeling system based on dynamic constaints. *Computer Graphics*, 22:179–188, 1988. Proceedings SIGGRAPH '88.

[3] Eric Bier and Maureen Stone. Snap-dragging. *Computer Graphics*, 20(4):233–240, 1986. Proceedings SIGGRAPH '86.

[4] Computervision Corporation. DesignView. Computer Program, 1992.

[5] D. Brookshire Conner, Scott S. Snibbe, Kenneth P. Herndon, Daniel C. Robbins, and Andries van Dam. Three-dimensional widgets. In *Proceedings of the 1992 Workshop on Interactive 3d Graphics*, pages 183–188, March 1992.

[6] J. S. Duff, A. M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, Oxford, UK, 1986.

[7] Kurt Fleischer and Andrew Witkin. A modeling testbed. In *Proc .Graphics Interface*, pages 127–137, 1988.

[8] Barry Fowler. Geometric manipulation of tensor-product surfaces. In *Proceedings, Interactive 3D Workshop*, 1992. (to appear).

[9] Michael Gleicher. Briar - a constraint-based drawing program. In *SIGGRAPH Video Review*, volume 77, 1992. CHI '92 Formal Video Program.

[10] Michael Gleicher. Integrating constraints and direct manipulation. In *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, pages 171–174, March 1992.

[11] Michael Gleicher and Andrew Witkin. Differential manipulation. *Graphics Interface*, pages 61–67, June 1991.

[12] Michael Gleicher and Andrew Witkin. Snap together mathematics. In Edwin Blake and Peter Weisskirchen, editors, *Advances in Object Oriented Graphics 1: Proceedings of the 1990 Eurographics Workshop on Object Oriented Graphics*. Springer Verlag, 1991. Also appears as CMU School of Computer Science Technical Report CMU-CS-90-164.

[13] Michael Gleicher and Andrew Witkin. Through-the-lens camera control. *Computer Graphics*, 26(2):331–340, July 1992. Proceedings Siggraph '92.

[14] Andreas Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic, 1989.

[15] David W. Juedes. A taxonomy of automatic differentiation tools. In Andreas Griewank and George Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation and Application*, pages 315–329. SIAM, January 1991.

[16] Devandra Kalra and Alan H. Barr. A constraint-based figure maker. In C.E. Vandoni and D. A. Duce, editors, *Proceedings Eurographics*, pages 413–424, 1991.

[17] Michael Kass. CONDOR: constraint-based data flow. *Computer Graphics*, 26:321–330, July 1992. Proceedings SIGGRAPH '92.

[18] Henry Kaufman. Constraint techniques for interactive physically-based modeling. Master's thesis, Brown Unversity, July 1991.

[19] Greg Nelson. Juno, a constraint based graphics system. *Computer Graphics*, 19(3):235–243, 1985. Proceedings SIGGRAPH '85.

[20] John K. Osterhout. Tcl: An embeddable command language. In *1990 Winter Usenix Conference Proceedings*, 1990.

[21] A. Pentland, I Essa, M. Friedmann, B. Horowitz, S. Sclaroff, and T. Starner. The thingworld modeling system. In E. F. Deprette, editor, *Algorithms and Parallel VLSI Architectures*. Elsevier Press, October 1990.

[22] John Platt. A generalization of dynamic constraints. *CGVIP: Graphical Models and Image Processing*, 54(6):516–525, November 1992.

[23] William Press, Brian Flannery, Saul Teukolsky, and William Vetterling. *Numerical Recipes in C*. Cambridge University Press, Cambridge, England, 1986.

[24] Mark Sapossnek. *Virtual Prototyping: An Interactive Approach to Geometric Tolerance Design and Analysis*. PhD thesis, Carnegie Mellon University, 1993. in preperation.

[25] Peter Schroeder and David Zeltzer. The virtual erector set: Dynamic simulation with linear recursive constraint propagation. *Computer Graphics*, 24(2):23–31, March 1990. Proceedings 1990 Symposium on Interactive 3D Graphics.

[26] Steven Sistare. Interaction techniques in constraint-based geometric modeling. In *Proceedings Graphics Interface '91*, pages 85–92, June 1991.

[27] Paul S. Strauss and Rikk Carey. An object-oriented 3d graphics toolkit. *Computer Graphics*, 26(2):341–349, July 1992. Proceedings SIGGRAPH '92.

[28] Mark C. Surles. An algorithm for linear complexity for interactive, physically-based modelling of large proteins. *Computer Graphics*, 26(2):221–230, 1992. Proceedings SIGGRAPH '92.

[29] Ivan Sutherland. *Sketchpad: A Man Machine Graphical Communication System*. PhD thesis, Massachusetts Institute of Technology, January 1963.

[30] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain specific graphical editors. In *Proceedings of the 1989 ACM SIGGRAPH Symposium on User Interface Software and Technology*, November 1989.

[31] Charles W. Wampler. Manipulator inverse kinematic solutions based on vector formulations and damped least-squares method. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(1):93–101, January 1986.

[32] William C. Welch and Andrew Witkin. Variational surface modelling. *Computer Graphics*, 26(2):157–166, July 1992. Proceedings SIGGRAPH '92.

[33] Andrew Witkin, Michael Gleicher, and William Welch. Interactive dynamics. *Computer Graphics*, 24(2):11–21, March 1990. Proceedings 1990 Symposium on Interactive 3D Graphics.

[34] Andrew Witkin and Michael Kass. Spacetime constraints. *Computer Graphics*, 22:159–168, 1988. Proceedings SIGGRAPH '88.