MathPad^2: A System for the Creation and Exploration of Mathematical Sketches

Joseph J. LaViola Jr. Robert C. Zeleznik

Brown University *

Abstract

We present *mathematical sketching*, a novel, pen-based, modeless gestural interaction paradigm for mathematics problem solving. Mathematical sketching derives from the familiar pencil-and-paper process of drawing supporting diagrams to facilitate the formulation of mathematical expressions; however, with a mathematical sketch, users can also leverage their physical intuition by watching their hand-drawn diagrams animate in response to continuous or discrete parameter changes in their written formulas. Diagram animation is driven by implicit associations that are inferred, either automatically or with gestural guidance, from mathematical expressions, diagram labels, and drawing elements. The modeless nature of mathematical sketching enables users to switch freely between modifying diagrams or expressions and viewing animations. Mathematical sketching can also support computational tools for graphing, manipulating and solving equations; initial feedback from a small user group of our mathematical sketching prototype application, MathPad^2, suggests that it has the potential to be a powerful tool for mathematical problem solving and visualization.

CR Categories: H.5.2 [Information Interfaces and Presentation]: User Interfaces—Interaction Styles G.4 [Mathematics of Computing]: Mathematical Software—User Interfaces;

Keywords: pen-based interfaces, mathematical sketching, gestures

1 Introduction

Diagrams and illustrations are often used to help explain mathematical concepts. They are commonplace in math and physics textbooks and provide a form of physical intuition about abstract principles [Hecht 2000; Varberg and Purcell 1992; Young 1992]. Similarly, students often draw pencil-and-paper diagrams for math problems to help in visualizing relationships among variables, constants, and functions. With the drawing as a guide, they can write the appropriate math to solve the problem. However, such static diagrams generally assist only in the initial formulation of mathematical expressions, not in the "debugging" or analysis of those expressions. This can be a severe limitation, even for simple problems with natural mappings to the temporal dimension, or for problems with complex spatial relationships.

By animating sketched diagrams from changes in associated mathematical expressions, users can evaluate different formulations with their physical intuitions about motion. They can sense mismatches between animated and expected behaviors and can often see that a formulation is incorrect and also make better educated guesses as to why. Alternatively, correct formulations can be explored intuitively, perhaps to home in on an aspect of the problem to study with a more conventional numerical or graphing technique. It is beyond the scope of this paper to evaluate the educational merits of mathematical sketching; however, we are convinced that the rapid creation of mathematical sketches can unlock a range of insight, even for such simple formulations as the ballistic 2D motion of a spinning football, where correlations among position, rotation and their derivatives can be challenging to comprehend.

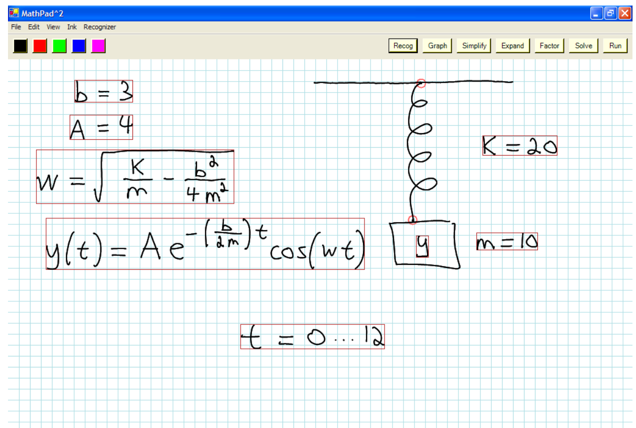

Figure 1: A mathematical sketch used to explore damped harmonic oscillation. It shows a spring and mass drawing and the necessary equations for animating the sketch. The label inside the mass associates the mathematics with the drawing.

This paper presents MathPad^2, a prototype application for creating mathematical sketches (see Figure 1). MathPad^2 incorporates a novel gestural interaction paradigm that lets users modelessly create handwritten mathematical expressions using familiar math notation and free-form diagrams, including associations between the two, with only a stylus. We posit that because users must write down both the math and the diagrams themselves, MathPad^2 will not only be general enough to apply to a full spectrum of problems, but may also support deeper mathematical understanding than alternative approaches including professionally created interactive illustrations. Thus, MathPad^2's central design principle is to be broadly applicable by enforcing the notion that as much behavior as possible be specified by user-written mathematical expressions, not by rules or behaviors implicitly embedded in the system.

To present MathPad^2 from a system's perspective, we describe various user interface and recognition options and discuss why we chose our current design. The next section considers related work and is followed by sections on MathPad^2's gestural user interface and on its visual parsing engine. We then present an example scenario of how an introductory physics student might use the system. Finally, we discuss informal feedback from a small user group and

*Email: {jjl,bcz}@cs.brown.edu

how mathematical expression recognition and parsing accuracy affects system usability.

2 Related Work

The idea of using computers to create dynamic illustrations of mathematical concepts has a long history. One of the earliest dynamic illustration environments was Borning's ThingLab, a simulation laboratory environment for constructing dynamic models of experiments in geometry and physics, that relied heavily on constraint solvers and inheritance classes [Borning 1979]. Other systems such as Interactive Physics™ and The Geometer's SketchPad™ also let the user create dynamic illustrations; these systems are all WIMP-based (Windows, Icons, Menus, Pointers) resulting in a significant amount of mode switching and loss of fluidity within the interface. In addition, they do not allow the user to write handwritten mathematics to create these illustrations. Because MathPad^2 uses handwritten mathematical expressions, users can leverage their knowledge of mathematical notation in order to create mathematical sketches. Java applets that provide both interactive and dynamic illustrations have also been developed for exploring various mathematical principles [Laleuf and Spalter 2001; Spalter and Simpson 2000]. However, these applets are not general, typically provide limited control over the illustration, and rarely show the user the mathematics behind the illustration.

One of the key contributions of the MathPad^2 system is its gestural user interface. This type of interface has been used in many different applications including cooperative objected-oriented design [Damm et al. 2000], conceptual design in 2D [Gross and Do 1996] and 3D [Igarashi et al. 1999; Zeleznik et al. 1996], musical score creation [Forsberg et al. 1998], prototyping user interfaces [Landay and Myers 1995], and whiteboard systems [Mynatt et al. 1999; Moran et al. 1997]. While these gestural interfaces have worked well for their particular applications, they are either modal or have limited drawing domains. In contrast, MathPad^2 strives for a modeless gestural interface that allows fluid transitions among drawing free-form shapes, writing mathematics, and performing gestural actions.

Alvarado [Alvarado 2000] and Kara [Kara et al. 2004] let the user make sketched diagrams that are recognized as drawing primitives with domain knowledge from specific disciplines and then animated. Although these systems provide powerful illustrations of physics and mathematical concepts, they are limited because of their domain knowledge and because they hide the underlying mathematical formulations from the user.

The primary focus of systems such as Mathematica™, Maple™, MathCad™, and Matlab™ has been entering mathematics for computation, symbolic mathematics, and illustration. These tools can create dynamic illustrations using mathematics as input. However, the mathematical notation used in these systems is one-dimensional, requiring unconventional notation for concepts that would be intuitive using 2D handwritten mathematics. In addition, these systems do not let the user create diagrams in a natural pencil-and-paper style.

Finally, a number of systems let users enter 2D handwritten mathematics in the context of math recognition and parsing, such as those found in [Zanibbi et al. 2002; Chan and Yeung 2000a; Matsakis 1999; Miller and Viola 1998]. Only a few of these systems go beyond just exploring and developing recognition technology. For example, Chan developed a simple pen-based calculator [Chan and Yeung 2001] while xThink, Inc. developed MathJournal™, a system designed to solve equations, perform symbolic manipulation, and make graphs. However, we are not aware of any system that lets users make dynamic illustrations with handwritten 2D mathematics.

3 MathPad^2 and Mathematical Sketching

Mathematical sketching is the process of making simple illustrations from a combination of handwritten 2D mathematical expressions and sketched diagrams. Combining mathematical expressions with diagram elements, called an *association*, is done either implicitly using diagram labels as input to an inferencing engine or manually using a simple gestural user interface. The formulations in the mathematical sketch of Figure 1 are written as digital ink and converted, using our mathematical expression recognition and parsing engine, for further processing into equivalent one-dimensional string representations required by our computational back end. Individual diagram elements can be associated with various mathematical expressions and will behave accordingly. The user can view the sketch results as an animation that is based on the underlying mathematical specification.

The following three sections describe the user interface, the sketch parser, and the animation engine of our MathPad^2 system.

3.1 User Interface

An important goal of the MathPad^2 prototype is to facilitate mathematical problem solving without imposing any interaction burden beyond what would arise with traditional media. Since pencil-and-paper users switch fluidly between writing equations and drawing supporting diagrams, a modeless interface is highly desirable. Although a simple free-hand drawing pen would suffice to mimic pencil and paper, we want to support computational activities including formula manipulation, diagram rectification (see section 3.2.4), and animation. This functionality requires extending the notion of a free-hand pen, either implicitly by parsing the user's 2D input on the fly or explicitly by letting the user perform gestural operations. We chose an interface that combines both in an effort to reduce the complexity and the ambiguities that arise in many hand-drawn mathematical sketches – we use parsing to recognize mathematical expressions and make associations, while we use gestures to segment expressions and perform editing operations.

The challenge for MathPad^2's gestural user interface is for its gestures not to interfere with the entry of drawings or equations and yet still be direct and natural enough to feel fluid. The following sections describe the gestural interface summarized in Figure 2.

3.1.1 Writing Mathematical Expressions

Inking Writing mathematical expressions and diagrams in MathPad^2 is straightforward: users draw with a stylus on a Tablet PC as they would using pencil and paper. The only complication in writing expressions is how errant strokes are corrected. Although the stylus can be flipped over to use its eraser, we found that a gestural action not requiring flipping was both more accurate (because of hardware characteristics of the stylus) and more convenient. We therefore first designed a *scribble erase* gesture in which the user scribbles with the pen back and forth over the ink strokes to be deleted. The drawback of this first implementation was that it created too many false positives, recognizing scribble erase gestures

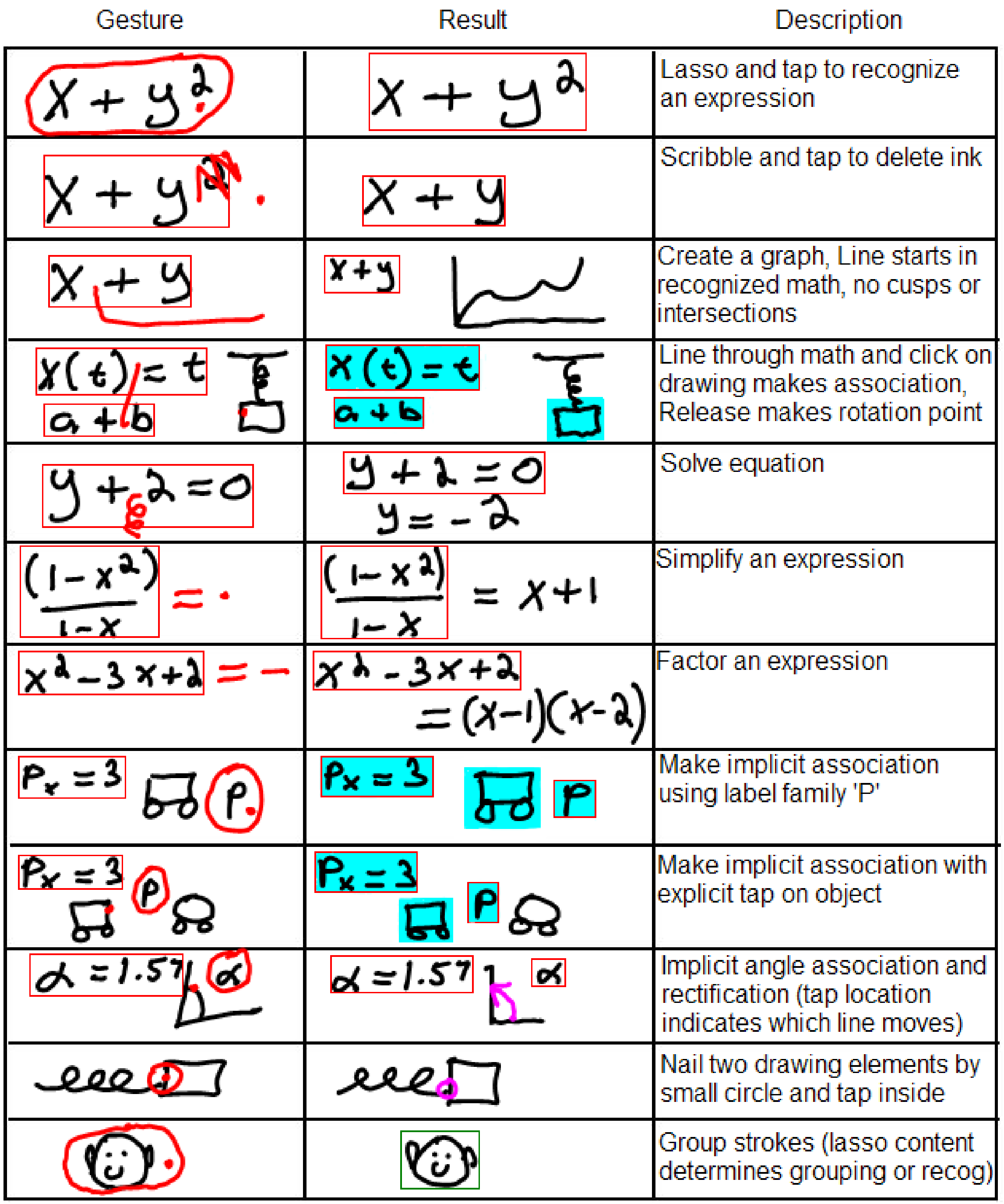

Figure 2: Gestures for interacting with MathPad^2. Gesture strokes in the first column are shown here in red for clarity. In the second column, cyan-highlighted strokes provide association feedback (the highlighting color changes each time a new association is made), and magenta strokes show nail and angle association/rectification feedback. Note that the first and last gesture are equivalent: the system examines the ink strokes inside the lasso to determine whether to perform stroke grouping or recognition.

when in fact the user had intended to draw ink and not erase anything. To alleviate this problem we decided on the use of a compound gesture because of its relative simplicity and ease of execution. Thus our current definition of scribble erase is the same scribble stroke as before followed directly by a tap. In practice, users found this compound gesture easy to learn, effective in eliminating false positives, and not significantly more difficult or slower than the simple scribble gesture.

Parsing Expressions Once mathematical expressions are drawn, they must be parsed by the system and converted to strings for use by a computational engine (Matlab in this case). Our initial attempt, clicking on a recognize button that attempted to recognize all math on the page, was problematic because it was hard to algorithmically determine "lines of math" accurately, especially when the expressions were closely spaced, at unusual scales or in unusual 2D arrangements. We therefore opted for a manual segmentation alternative in which the user explicitly selects a set of strokes comprising a single mathematical expression by drawing a lasso. Since in a modeless interface making a lasso cannot be distinguished from drawing a closed curve, we needed to disambiguate the two actions. An approach that worked well for some was to lasso lines of math while pressing the barrel button on the stylus. However, many people inadvertently triggered this button when trying to draw, while others found pressing a barrel button awkward. Once again, the solution was to use compound gestures – this time drawing a lasso around a line of math followed by a tap. This modified gesture has worked robustly within our dozen-user feedback group.

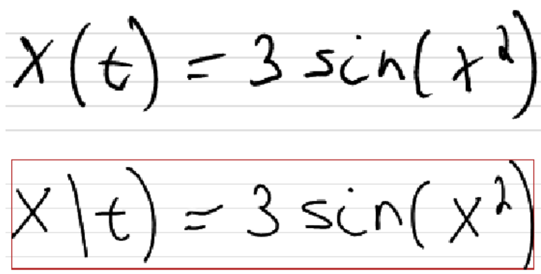

Figure 3: A written mathematical expression and a recognized one. Even though the recognized expression is presented in the user's own handwriting, recognition errors are easily discernible.

Feedback Ideally, recognition would not require any feedback to the user – the system would simply understand what the user had written. However, due to the complexity and ambiguities of mathematical notation, it is essential that users know how MathPad^2 interprets their input expressions. We chose to show the system's recognition in two ways. First, a bounding box rectangle is drawn around the user's digital ink for each recognized expression. Second, each ink symbol within a recognized expression is replaced with the corresponding canonical version of that symbol (a training example of the user's own handwriting) that occupies the same bounding box as the original stroke (see Figure 3). Our theoretical basis for this feedback is twofold: users can generally disambiguate characters of their own handwriting even if they are quite similar, and users often want to preserve the look, feel, and spatial relationships of their notation for reasons of esthetics, subtle information detail, and ease of editing [Zanibbi et al. 2001]. Although we had earlier provided an option for modifying the layout of the recognized ink to correspond to the 2D parse relationships of the math, this was often distracting since our implementation did not preserve all the layout subtleties of the user's original handwriting; nonetheless, we are currently investigating techniques for improved 2D layout constrained to the style of the user's input handwriting. Some users have expressed an interest in a high-quality typeset feedback option to be used primarily as an alternate view in a less technically interesting "clean-up" phase.

Correcting Recognition Errors When users identify a recognition error, they can correct it simply by scribble erasing the offending symbols and rewriting them, or they can invoke a pull-down menu of alternatives. Whenever the stylus hovers over a recognized expression, a small green button appears in the lower right corner of its bounding box. Pressing on this button displays a menu of alternate recognitions and the displayed ink is updated to reflect the alternate expression selected.

3.1.2 Making Diagrams

Diagrams are sketched in the same way as mathematical expressions except the diagrams need not be recognized. In balancing the value of a primitives-based drawing system against the added interaction overhead of specifying primitives, we decided that for our initial prototype our only primitive would be unrecognized ink strokes. We believe that a primitives-based approach not only would require a more elaborate user interface, but would also make it more difficult to find the source of an error since the diagram would be in part user-specified and in part specified by the primitive's hidden rules or behaviors.

Nailing Diagram Components In reviewing a broad range of mathematical illustrations, we noticed that the single low-level behavior of stretching a diagram element could be a very powerful technique. Thus, we support the concept of "nails" to pin a diagram element to the background or to pin a point on a diagram element to another diagram element. If either diagram element is moved, the other element either moves rigidly to stay attached at the nail if it has only one nail, or stretches so that all its nails maintain their points of attachment. Nails, although used primarily to create non-rigid objects, can also create binary grouping relationships. We do not currently detect or support cyclic nail relationships.

The user creates a nail by drawing a circle around the appropriate location in the drawing and making a tap gesture inside it (the tap disambiguates the nail gesture from a circle that is part of a drawing). A nail is distinguished from a math recognition gesture (see above) because the nail circle must intersect one or two drawing elements but fully contain neither, whereas the math recognition lasso must fully contain at least one stroke. The system finds and links together all drawing elements that intersect the circle of the nail gesture. The link is then symbolized by centering a small red circle on the nail location.

Grouping Diagram Components Since many drawings involve creating one logical object from a set of strokes (drawing elements), we overload the math recognition gesture to perform a grouping operation if the lasso is drawn around diagram strokes. We use the Microsoft Tablet PC SDK Divider API to classify the strokes within a lasso as being either drawings or text. If the strokes are drawings, they are grouped, otherwise they are considered to be an expression and mathematical recognition is performed. However, due to the complexity of this classification and the immaturity of Microsoft's implementation, this approach is not yet reliable and will require further research. In the meantime, we also provide an explicit gesture for grouping objects that can be distinguished from a recognition gesture based on the location of the tap. If the tap falls on the lasso, then we perform a grouping operation, otherwise we recognize the expression.

3.1.3 Associations

The essence of a mathematical sketch comes from making associations between mathematical expressions and diagrams. Associations are made between scalar mathematical expressions and angle arcs or one of the three special values of a diagram element, its *x*, *y*, or rotation coordinate. After an association is made, changes in mathematical expressions can be reflected as changes in the diagram and vice versa.

Associations between mathematical expressions and drawing elements can be made both explicitly and implicitly. Implicit associations are based on the familiar variable names and constant labels found in math and physics text illustrations. To create an implicit association, users draw a variable name or constant value near the intended drawing element and then use the math recognition gesture to recognize the label. If the recognition gesture's tap falls within the gesture's lasso, then the label is linked to the nearest drawing primitive; otherwise, the tap location is used to specify both the drawing element to be linked to the label and the drawing element's center of rotation (this point is only used for rotational labels). When labeling an angle arc, the location of the tap on the arc determines the *active line* – the line attached to the arc that will move when the angle changes. The apex of the angle is then marked with a green dot, and the active line is indicated with an arrowhead on the angle arc. Note, we do not detect or support cyclical association relationships, such as the specification of each angle in a triangle. MathPad^2 uses the recognized label and linked drawing element to infer associations with other expressions on the page (see Section 3.2.2).

For slightly more control over associations and to reduce the density of information in a diagram, associations can also be created explicitly without using variable name labels. The user can make an explicit association by drawing a line through a set of related math expressions and then tapping on a drawing element. After this line is drawn, drawing elements change color as the stylus hovers over them to indicate the potential for an association. However, this technique provides greater flexibility than the implicit association technique in specifying the precise point of rotation because, instead of just tapping on the drawing element (which sets the point of rotation to be the center of the drawing element), the user can press down on the element to select it, move the stylus, and then lift it to the desired center of rotation, even if it is not on the drawing element.

With both implicit and explicit associations, MathPad^2 provides an option for visualizing which drawing elements, labels and expressions are associated by filling the bounding boxes of all associated components with the same semi-transparent pastel color.

3.1.4 Using MathPad^2's ToolSet

In addition to diagram associations, MathPad^2 supports a range of computational functions on recognized mathematical expressions, including graphing, solving, simplifying, and factoring. This set of functions is rudimentary but gives some representative early results on adding advanced features to the system.

Users can graph recognized functions with a simple line gesture that begins on the function and ends at the graph's intended location. This line gesture is distinguished from other drawn lines by starting within the function expression's bounding box, being too long to be a mathematical symbol, and having no cusps or self-intersections. This gesture produces a movable, resizable graph widget displaying a plot of the function (see Figure 4). Additional graph gestures that end on this widget will overlay or replace the function being graphed depending on the state of the 'hold plot" check box found in the upper left corner of the widget.

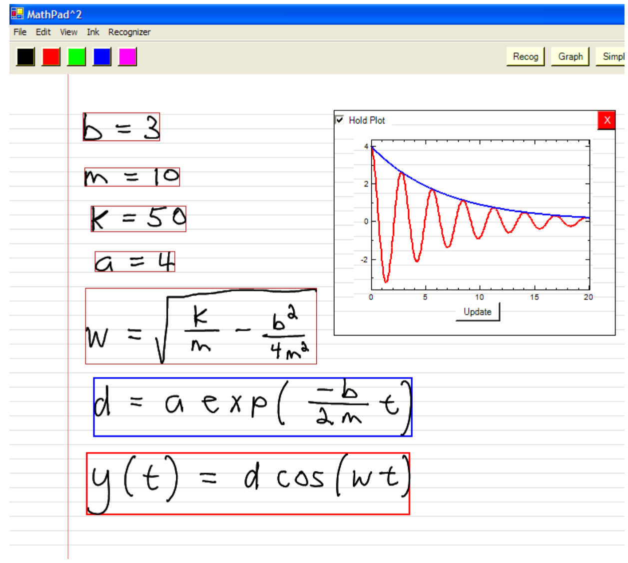

Figure 4: Two plots created using graph gestures. Expression bounding boxes are colored to correspond with plot lines.

The graph widget uses default values for the domain of plotted functions based on a very simple heuristic: the domain is 0...5 for functions of *t* and −5...5 for functions of any other variable. We are currently developing ways to choose better defaults based on function characteristics. In any case, users can change the domain or range by selecting a region of the graph to zoom in on or by writing a new value below the start and/or end of the graph and then clicking the update button. Optional visual feedback is provided to show correspondences between a plot line and a mathematical expression by coloring the line the same as the expression bounding boxes.

In addition to graphing, MathPad^2 enables the user to solve equations. Users invoke the solver by making a squiggle gesture that starts inside the bounding box of a recognized mathematical expression. The system presents the solution to the user at the end of the squiggle gesture either as typeset symbols or ink in the user's handwriting style. Closely related gestures for simplifying and factoring expressions are presented in Figure 2.

3.2 Sketch Parser

We have focused so far on the user interface for specifying the input description of a mathematical sketch. We now discuss the components that prepare a sketch for execution by recognizing expressions, inferring associations, and rectifying drawings.

3.2.1 Mathematical Expression Recognition

There have been many different approaches to the recognition and parsing of mathematical expressions [Chan and Yeung 2000b]. After evaluating these recognition and parsing techniques and because of the lack of publicly available systems, we decided to implement our own custom recognizer and parsing engine. We chose a writer-dependent system because these systems tend to be more accurate than independent systems since the recognizer can be tailored to a particular user's handwriting [Connell and Jain 2002]. In addition, we could use the user's writing samples to present recognition results to the user in her own handwriting, so as to keep a pencil-and-paper look and feel.

Our multistage recognizer uses three different recognition techniques to form a hybrid solution [Li and Yeung 1997; Connell and Jain 2000; Smithies et al. 1999]. The first stage is the preprocessing step where ink strokes are normalized and filtered to reduce noise and made size- and translation-invariant. In addition, dominant points [Li and Yeung 1997] and their directions as well as other statistical features are calculated for use in the classification steps. The second stage in the recognition algorithm, coarse classification, is used to decrease the number of possible mathematical symbol candidates by rejecting unlikely ones and is intended to be fast and not fully exhaustive. The coarse classification scheme uses two separate algorithms. The first uses the direction information as input to a dynamic programming algorithm that calculates the optimal degree of difference or similarity between two characters [Li and Yeung 1997]. The second algorithm performs a statistical feature-set classification based on [Smithies et al. 1999]. The results of both classifiers are then merged using an averaging scheme normalized by their respective standard deviations. The third stage of the recognizer is fine classification, which takes the list of mathematical symbol candidates from the coarse classifier and uses dynamic programming once again to determine the best possible match with the training data [Connell and Jain 2000]. The fine classification's dynamic program is similar to that used by the coarse classifier but also uses dominant points as input, which helps detect small differences between similar mathematical symbols.

After the mathematical symbols are recognized they must be two-dimensionally parsed in terms of exponents, subscripts, fractions, and others constructs [Blostein and Grbavec 2001]. Therefore, using a simple grammar, our parser takes into account not only the symbols but their bounding boxes as well. Because Matlab™ is our computational back end, the parsing system converts the 2D mathematical expressions into 1D Matlab specific expressions.

3.2.2 Association Inferencing System

As mentioned in section 3.1.3, implicit associations require a technique for determining the written mathematical expressions that should be associated with a particular drawing element based on the variable label linked to the element. An expression should be associated with a drawing element if that expression takes part in the computation of any variable that falls in the *label family* of the drawing element's variable label.

A label family is defined by its name, a root string. Members of the label family are variables that include that root string and a component subscript (e.g., x for its x-axis component) or a function specification. For example, if the user labels a drawing element ϕ_o, the inferencing system determines the label family to be ϕ and finds all mathematical expressions having members of the ϕ label family on the left-hand side of the equal sign: ϕ, ϕ_o, $\phi(t)$, $\phi_x(t)$, and so on. The inferencing system then finds all the variable names appearing on the right-hand side, determines their label families, and then continues the search. This process terminates when there are no more variable names to search for. Once all the related mathematical expressions have been found, they are sorted to represent a logical flow of operations that can be executed by a computational engine (e.g., Matlab), and the implicit association is made. For example, in Figure 1 the system would first store all of the terminal expressions, then the expressions that are not functions of t, and finally the time dependent functions. Note that our prototype does not currently handle interrelated equations such as $x(t) = t^2 y(t)$ and $y(t) = x(t) - t$. However, in future versions of the system, we plan to support these types of dependencies by detecting them, solving them with our computational engine, and providing the user with the ability to interactively select the appropriate solutions to use in their sketches.

3.2.3 Defining Drawing Dimensions

Although an explicit Cartesian coordinate system can be created within a mathematical sketch diagram, many diagrams contain enough information that this can be done implicitly. MathPad^2 can infer coordinate systems in two ways: by using the initial locations of diagram elements and by labeling linear dimensions within a diagram (see Figure 5).

When two different drawing elements are associated with expressions so that each drawing element has a different value for one of its coordinates (x or y), then an implicit coordinate system can be defined. The distance along the coordinate shared between the two drawing elements establishes a dimension for the coordinate system, and the location of the drawing elements implies the location of the coordinate system origin.

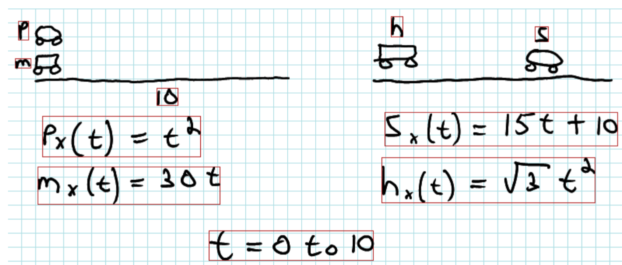

Figure 5: Two methods for inferring coordinate systems: the mathematical sketch on the left uses labeling of the ground line while the one on the right uses initial conditions.

Alternatively, if only one drawing element is associated with math, then the dimension of the coordinate system can still be inferred if another drawing element is associated with a numerical label. Whenever a numerical label is applied to a drawing element, MathPad^2 analyzes the drawing element: if it is a horizontal or vertical line, the corresponding x or y axis dimension is established; otherwise, we apply the label to the best-fit line to the drawing element and then establish the dimension of both coordinate axes.

If not enough information has been specified to define a coordinate system implicitly, then a default coordinate system is used. Like our graph defaults, we expect that we can improve these defaults from inferences based on an analysis of the function characteristics of the associated expressions.

3.2.4 Drawing Rectification

Mathematical sketches often have inherent ambiguities between what the mathematics specifies and what the user draws: that is, mathematical expressions often do not jibe precisely with their associated visual relationships in user drawings. *Rectification* is the process of fixing the correspondence between drawings and mathematics so that something meaningful is displayed. For example, a user might draw a diagram containing an angle, and then write some math that specifies that angle numerically. The drawn angle is unlikely to match the math description exactly and so either the drawing must be adjusted to match the math or vice versa.

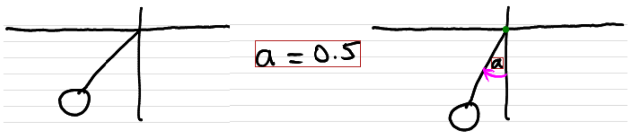

Figure 6: The effects of labeling an angle: the drawing is rectified based on the initial value of a (in radians). The green dot shows the rotation point and the magenta arrow indicates which part of the drawing will rotate.

Rectification is a difficult problem tantamount to the general constraint satisfaction problem. However, rectification in MathPad^2 is simplified since it is designed to handle only simple acyclic relationships between drawing elements and we do not check for cycles during this process. Our current implementation demonstrates rectification in the context of angle relationships. Mismatches between numerical descriptions of angles and their diagram counterparts are readily discernible. When an angle such as a in Figure 6 is associated with math, MathPad^2 rectifies the drawing in one of two ways. First, the angle between the two lines connected by the angle arc is computed. Next, the system determines if a mathematical expression corresponding to the angle label already exists. If so, it moves the active line, as determined by the angle arc, to the correct place based on the mathematical specification. If not, it uses the angle computed from the drawing as the numerical specification of the angle's value. Currently, this angle is represented internally and used during simulation.

A similar rectification process occurs when the same coordinate of more than two drawing elements is specified. If one or two coordinates are specified, no rectification is typically necessary since the coordinates of the elements can be used to establish dimensions of a coordinate system (see Section 3.2.3). However, when a third drawing element is added, it must be rectified to the coordinate system defined by the previous two drawing elements. A full treatment of the rectification issues of multi-element drawings is still under development.

3.3 Mathematical Sketch Animation and Output

The final subsystem of MathPad^2 is the animation engine. When users create a diagram and associates it with math, they are effectively drawing the initial conditions of an animation. MathPad^2 defines initial conditions as the value of expressions when the variable t is at its initial value, as defined by users when they write the mathematical expression $t = T_{initial}$ ... T_{final}.[1]

In order to compute the animation, the animation subsystem first checks all drawing elements with associations to see if they are *animatable*. Animatable drawing elements are associated with functions of time. Currently the system supports x and y translational movement, rotation about a given point, and changing the value of an arc. The system takes these animatable drawing elements and sends their associated math to Matlab, which executes the math and returns the results.

[1]In actuality, the user draws a combination of initial values (for x, y, and rotation coordinate associations) and initial constants (for arc associations). In typical diagrams, however, initial constants are equivalent to initial conditions.

By default, MathPad^2 maps all animations so that they last four seconds in wall-clock time. Once again, our default is not appropriate for many animations, and so it can be overridden by specifying the duration as part of the time specification:

$t = T_{initial}$ ... T_{final} *in Seconds.*

4 An Example 2D Projectile Motion Scenario

We now present a scenario for how a user, say a high school physics student, might use MathPad^2 and mathematical sketching as a tool for better understanding mathematical concepts.

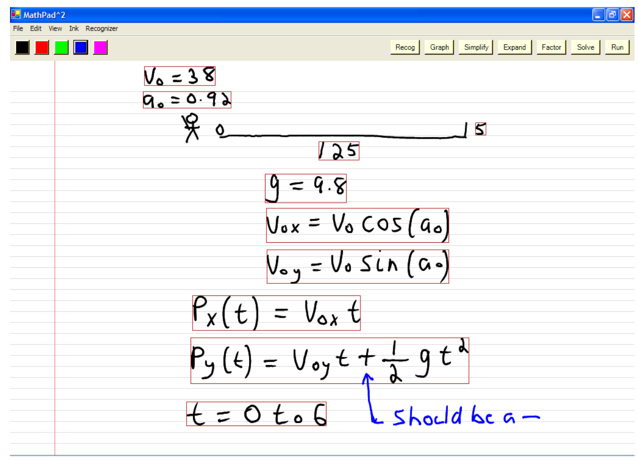

Figure 7: An ill-specified mathematical sketch for determining whether a baseball will fly over a fence for a home run. After running the sketch, the user will clearly see the error, since the ball will fly upward in a parabolic fashion. The remedy is shown in blue.

The student is interested in determining whether a baseball player can hit the ball over a fence given an initial velocity and angle. She first writes down a simple playing field as shown in Figure 7. Next she writes down the known quantities: the initial angle a_o, initial velocity v_o, and a gravitational term g. From her knowledge of projectile motion, she then writes down the mathematics as shown in Figure 7, labels the drawing making the required associations, and runs the simulation.

The simulation shows the ball moving upward against gravity, which does not look correct. The user then checks the equations and realizes that the equation $P_y(t) = v_{oy}t + \frac{1}{2}gt^2$ has a sign error. She scratches out the $+$, writes in a $-$, and re-recognizes the expression. She then runs the simulation again: the ball takes on the correct motion and barely makes it over the fence.

Next she wants to see how much farther the ball will go if v_o is increased. She scratches out the current value, writes in a larger one, and runs the simulation again. The ball does go farther but it also stops short of the ground, which leads her to the question "when will the ball hit the ground with these new parameters?" So, she takes equation $P_y(t)$, sets it equal to zero, and solves it with the squiggle gesture. Finally, she takes the second value for t, changes the time field, runs the simulation, and finds this time value is correct for the ball to hit the ground under the new parameters.

In this scenario, the student's understanding of the mathematical concept is augmented not only because she could visualize the behavior of the mathematical system, but also because she had to write down the mathematics in order to generate the animation. It is this unique feature that makes MathPad^2 and mathematical sketching a powerful problem-solving tool.

5 Discussion

About a dozen users have seen and tried the MathPad^2 prototype. Their response was quite positive. Users found the gestural interface easy to learn and use and commented on the fluidity of the modeless interaction style. One user stated that he wished he had had this software in high school. Many users asked whether they could solve more complex problems, such as a double pendulum, that often require open-form solutions. We believe that a much broader class of problems, including multi-body collisions and simple ordinary and partial differential equations, could be supported by extending mathematical sketching to handle basic programming constructs, such as loops and conditionals. We also recognize the desire for a macro facility that would let mathematical sketches be saved as functions for reuse within other sketches.

The user feedback also highlighted the effect of misrecognition and incorrect parsing on the overall usability of MathPad^2. In general, recognition and parsing errors increase in proportion to the complexity of the mathematical expressions. Users could generally resolve errors by simply scratching out the offending symbols, redrawing them and then re-recognizing the expression, although in some cases, they would need to perform this process multiple times, since our recognition algorithms do not adjust on the basis of previous attempts.

By training the symbol recognizer for each user, symbol recognition was generally reasonably robust. However, some users did encounter persistent symbol-recognition errors that can be attributed to ambiguous entry of specific pairs or groups of symbols, for instance, when distinguishing between 5 and s, x and $+$, and c, 1, and (. In some cases, we tried to exploit contextual knowledge to avoid common conflicts [Lee and Wang 1995]. For example, we would replace the 0 in $l0g$ with an o to form log, while a 5 in $5in$ would be replaced with an s to form sin. A few users chose to reduce the set of trained symbols to improve accuracy at the cost of limiting the scope of mathematical expressions. Other users changed the way they drew certain symbols such as drawing a 5 with two strokes to differentiate from a single stroke s.

We did not provide any means for the user to train the expression parser. Instead, we used a constant set of parsing rules based on the spatial relationships between symbol bounding boxes. As a result, parsing performance exhibited greater per-user variance depending on how closely the user's natural handwriting corresponded to our rules. Common parsing errors were to mischaracterize superscripts when the base symbol was an ascender letter (e.g., d, b) or subscripts when the base symbol was a descender letter (e.g., y, p). Also, some users wrote rather large subscripts and superscripts, which often resulted in parsing errors. We believe that most of these errors can be avoided by training the parser as well as the symbol recognizer. In other cases, parsing errors are more complicated, as when a misrecognized symbol triggers a parsing error. For example, in Figure 1, if the closing parenthesis of the exponential is misrecognized as a 1, our parser would aggregate the $cos(wt)$ as a subscript of the 1. However, we believe these kinds of errors can largely be avoided by exploiting context both within and between expressions, for example by avoiding subscripts for integers. Last, we need to present parsing results in a more natural way than our 1D representation, as by the technique described in [Zanibbi et al. 2001].

The current state of MathPad^2's recognition and parsing system can be a limiting factor for some users, especially those who must adjust their handwriting significantly to be neat and to correspond to our notion of the "correct" way to lay out expressions. Still, with adequate training most users find the system usable for the domain of mathematical sketches we currently support. To achieve broader acceptance and to support more complex mathematical sketches, we must continually incorporate the state of the art in mathematical recognition and parsing techniques.

6 Conclusion

We have discussed the novel concept of mathematical sketches – combinations of handwritten mathematics and free-hand diagrams – that allow rapid animated visualization of mathematical formulations. In addition, we described the system design issues encountered in creating a prototype mathematical sketching system, MathPad^2, including the design of a modeless gestural interface and techniques for implicitly associating and rectifying diagrams with mathematical expressions. Despite the restriction of the current prototype to closed-form expressions, we conjecture that mathematical sketching can be a powerful assistant in formulating and visualizing mathematical concepts.

Acknowledgements

We are grateful to Kazutoshi Yamazaki for implementing our custom recognition and parsing engine and to David Salesin for suggestions on improving an earlier version of MathPad^2. We also thank Andries van Dam, John Hughes, David Laidlaw, and the members of the Brown University Computer Graphics Group for valuable guidance and discussion, as well as the anonymous SIGGRAPH reviewers for their feedback. This work is supported in part by a gift from Microsoft and grants from the National Science Foundation and the Joint Advanced Distributed Co-Laboratory.

References

ALVARADO, C. J. 2000. *A Natural Sketching Environment: Bringing the Computer into Early Stages of Mechanical Design*. Master's thesis, Massachusetts Institute of Technology.

BLOSTEIN, D., AND GRBAVEC, A. 2001. Recognition of mathematical notation. In *Handbook of Character Recognition and Document Image Analysis*, World Scientific, H. Bunke and P. Wang, Eds., 557–582.

BORNING, A. 1979. *ThingLab: A Constraint-Oriented Simulation Laboratory*. PhD thesis, Stanford University.

CHAN, K.-F., AND YEUNG, D.-Y. 2000. An efficient syntactic approach to structural analysis of on-line handwritten mathematical expressions. *Pattern Recognition 33*, 3, 375–384.

CHAN, K.-F., AND YEUNG, D.-Y. 2000. Mathematical expression recognition: A survey. *International Journal on Document Analysis and Recognition 3*, 1, 3–15.

CHAN, K.-F., AND YEUNG, D.-Y. 2001. Pencalc: A novel application of on-line mathematical expression recognition technology. In *Proceedings of the Sixth International Conference on Document Analysis and Recognition*, 774–778.

CONNELL, S. D., AND JAIN, A. K. 2000. Template-based on-line character recognition. *Pattern Recognition 34*, 1, 1–14.

CONNELL, S. D., AND JAIN, A. K. 2002. Writer adaptation for online handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence 24*, 3, 329–346.

DAMM, C. H., HANSEN, K. M., AND THOMSEN, M. 2000. Tool support for cooperative object-oriented design: Gesture-based modelling on an electronic whiteboard. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press, 518–525.

FORSBERG, A., DIETERICH, M., AND ZELEZNIK, R. 1998. The music notepad. In *Proceedings of the 11th Annual ACM Symposium on User Interface Software and Technology*, ACM Press, 203–210.

GROSS, M. D., AND DO, E. Y.-L. 1996. Ambiguous intentions: A paper-like interface for creative design. In *Proceedings of the 9th Annual ACM Symposium on User Interface Software and Technology*, ACM Press, 183–192.

HECHT, E. 2000. *Physics: Calculus*. Brooks/Cole.

IGARASHI, T., MATSUOKA, S., AND TANAKA, H. 1999. Teddy: A sketching interface for 3d freeform design. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press/Addison-Wesley Publishing Co., 409–416.

KARA, L. B., GENNARI, L., AND STRAHOVICH, T. F. 2004. A sketch-based interface for the design and analysis of simple vibratory mechanical systems. In *Proceedings of ASME International Design Engineering Technical Conferences*.

LALEUF, J. R., AND SPALTER, A. M. 2001. A component repository for learning objects: A progress report. In *Proceedings of the First ACM/IEEE-CS Joint Conference on Digital Libraries*, ACM Press, 33–40.

LANDAY, J. A., AND MYERS, B. A. 1995. Interactive sketching for the early stages of user interface design. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press/Addison-Wesley Publishing Co., 43–50.

LEE, H.-J., AND WANG, J.-S. 1995. Design of a mathematical expression recognition system. In *Proceedings of the Third International Conference on Document Analysis and Recognition*, IEEE Press, 1084–1087.

LI, X., AND YEUNG, D.-Y. 1997. On-line handwritten alphanumeric character recognition using dominant points in strokes. *Pattern Recognition 30*, 1, 31–44.

MATSAKIS, N. E. 1999. *Recognition of Handwritten Mathematical Expressions*. Master's thesis, Massachusetts Institute of Technology.

MILLER, E. G., AND VIOLA, P. A. 1998. Ambiguity and constraint in mathematical expression recognition. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, 784–791.

MORAN, T. P., CHIU, P., AND VAN MELLE, W. 1997. Pen-based interaction techniques for organizing material on an electronic whiteboard. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology*, ACM Press, 45–54.

MYNATT, E. D., IGARASHI, T., EDWARDS, W. K., AND LAMARCA, A. 1999. Flatland: New dimensions in office whiteboards. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM Press, 346–353.

SMITHIES, S., NOVINS, K., AND ARVO, J. 1999. A handwriting-based equation editor. In *Proceedings of Graphics Interface'99*, 84–91.

SPALTER, A. M., AND SIMPSON, R. M. 2000. Integrating interactive computer-based learning experiences into established curricula: A case study. In *Proceedings of the 5th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education*, ACM Press, 116–119.

VARBERG, D., AND PURCELL, E. J. 1992. *Calculus with Analytic Geometry*. Prentice Hall.

YOUNG, H. D. 1992. *University Physics*. Addison-Wesley Publishing Company.

ZANIBBI, R., NOVINS, K., ARVO, J., AND ZANIBBI, K. 2001. Aiding manipulation of handwritten mathematical expressions through style-preserving morphs. In *Proceedings of Graphics Interface 2001*, 127–134.

ZANIBBI, R., BLOSTEIN, D., AND CORDY, J. 2002. Recognizing mathematical expressions using tree transformation. *IEEE Transactions on Pattern Analysis and Machine Intelligence 24*, 11, 1–13.

ZELEZNIK, R. C., HERNDON, K. P., AND HUGHES, J. F. 1996. Sketch: An interface for sketching 3d scenes. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press, 163–170.