Solving mathematical problems on touch-based devices

by

R. Mark Prosser

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2014

Abstract

Research on the use of mobile touch-based devices for mathematics has not kept up with the increasing ubiquity of such machines. Building upon MathBrush, a pen-based system for the recognition of handwritten mathematics, we investigate the iPad as a platform for doing mathematics, with a particular focus on differential equations. This thesis chronicles the work done to bring computational capabilities to MathBrush for the iPad, including the implementation of graphical plotting features, which are relevant for general mathematics as well as for differential equations. Furthermore, we explore new ideas for creating a specialized differential equation solver application, utilizing the MathBrush recognition engine. A vision for a new pen-based interface for differential equation input is presented, and its prototypical implementation is described.

Acknowledgements

Thank you to my supervisor, George Labahn, for his unwavering support and invaluable guidance throughout my academic career, and for opening the door to valuable opportunities. I am grateful to my colleagues Scott and Mirette, whose work has provided the starting point for my research, and from whom I have learned a great deal. Thank you to my additional readers, Arne Storjohann and Edward Lank for their time and perspective. Finally, I would like to thank my wife, Shima, for her limitless patience and support.

Dedication

To my son, Ethan, whose timely arrival on Pi Day provided the final motivation for completing this work.

Table of Contents

Chapter 1

Introduction

In an episode of the popular TV show, The Big Bang Theory [1], one of the main characters, Leonard, complains that there is no way to solve differential equations on his smartphone. In his fictional research, Leonard finds himself frequently having to stop and solve differential equations along the way to solving something bigger. He has an idea to develop a smartphone app to solve differential equations for him, and enlists the help of his friends Howard, Raj, and Sheldon. The fictional differential equation solving smartphone app they created was called The Lenwoloppali Differential Equation Scanner.

The scientific content of the show is loosely grounded in reality, making it particularly appealing to academics. In this episode, the show presents a very relevant idea of creating a smartphone app to solve differential equations. The idea of using a smartphone app to solve differential equations touches on a broader question: can mobile devices be useful for doing mathematics?

Leonard's motivation is certainly realistic: having to stop and solve equations can be disruptive to the flow of research. Although powerful computer algebra systems exist to solve a wide range of mathematical problems, using them can be a chore. The interface for inputing a mathematical expression using a keyboard and mouse can be cumbersome and will never be as natural as the pen-and-paper approach. The ability to simply write down an equation (or take a picture of one as the show suggests) and get the solution, would be profoundly useful. With current knowledge and technology this is not an unreasonable endeavour.

The show goes into some detail about how such an app might work. The user takes a picture of the differential equation, the app reads the equation using image processing and handwriting recognition techniques, and then runs it through a computer algebra system to obtain a solution. This approach may not work as easily in practice as it does in fiction. Optical character recognition is difficult enough for natural language text, let alone for mathematical expressions. Coupling this difficulty with the further challenge of determining the semantics and solving the equation makes it unlikely that this approach would have much success.

We are interested in the broader question: can mobile devices be useful for doing mathematics? With the increasing ubiquity of portable touch-based devices, surprisingly little work has been done to explore the possibilities of doing mathematics through touch-based interfaces. We investigate doing mathematics via pen-based input, with our experiences having been part of a project built upon the MathBrush pen-based system [5]. MathBrush has been developed over the last decade for Microsoft tablet computers and more recently for the iPad and the Surface Pro. It has become a practical tool for recognition of handwritten mathematics, and the Microsoft-based versions integrate with computer algebra systems, such as Maple, to allow the user to compute with their recognized expressions. We wish to bring the power of a computer algebra system to MathBrush for the iPad. By linking the iPad client with an open-source computer algebra server we create a fundamental tool for mobile pen-based mathematics, useful to students and academics alike. Further, we consider the specific problem of solving differential equations, and investigate how best to achieve this goal on touch-based devices using free-form handwritten input, thus loosely approximating the Lenwoloppali differential equation scanner.

1.1 Challenges

Our goals present some interesting challenges. The iPad was not designed to perform mathematics, and is certainly not equipped for computationally demanding tasks. Due to their size constraints such devices naturally have limited computing power. In contrast, communication with mobile devices has become incredibly fast and versatile, allowing for high speed data transfer rates through not only wireless internet connections but mobile telecommunication networks. The use of mobile devices has become so ubiquitous, with supporting infrastructure expanding to keep up, that it is becoming increasingly uncommon to find oneself *without* an available data connection. For these reasons, it makes sense to pursue an external, server-based solution.

There are several design questions to consider. Of the candidate computer algebra systems, which one is the most suitable for this application? The leading systems are commercial. However, since our application will be available to the public, we wish to avoid potential licensing issues that may accompany a commercial system. We therefore consider noncommercial systems, and find that Sage, a Python-based system combining many open-source math libraries, stands out among the rest.

How will the client and server communicate? Moreover, how will the mathematical semantics be conveyed between the MathBrush client and the computer algebra system? A client-server protocol must be established to communicate expression representations across the network, but the greater challenge is to extract the appropriate semantic information from the output of the computer algebra system. Initially, MathML appears to provide an ideal intermediate representation, but upon further investigation this is not so clear. Consequently, we implement a custom adapter for computational output that is easily mapped to the internal expression structure used by MathBrush.

Any mathematical system would be incomplete without graphing support. Basic plotting functionality is essential for general mathematics, and especially for visualizing numerical solutions to differential equations. Plotting essentially involves computing a set of points and presenting them on the screen, along with with the axes and other informative decorations. Additionally, the plots should be open to interaction, and this is especially expected on touch-based tablet devices. Therefore the user interface aspect of the graphing component demands a special focus. The computational element belongs with the computer algebra server; however, plotting introduces a new form of data to be conveyed back to the client which requires an augmentation of the established client-server protocol for mathematical expressions. We develop a simple and generic structure for encapsulating the relevant plot data so that it can be easily communicated to the client. Populating the structure requires an understanding of the computer algebra system's plotting tools and data representation, and displaying the plot on the client side requires rich front-end development. Thus, a nontrivial development effort is required to fulfill the graphing objectives.

Differential equations are a specific form of mathematical expression requiring a higher-level understanding on the part of the system. For example, derivative terms can be expressed in multiple ways, depending on the choice of notation and variable names. Also, we intend to support initial value problems, which contain initial conditions in addition to the governing equation. How can we ensure these kinds of multi-expressions are supported by the recognition engine? How can we make MathBrush aware of these expressions when they are written down? How can we translate these expressions into the appropriate form needed to obtain a solution from the computer algebra system? Fortunately, the recognizer has the prerequisite capabilities to support the elements that comprise differential equations and initial conditions. What is needed is a module which parses the primitive expression trees to detect differential equations and extract the information required to solve them. Although these tasks are intertwined, we find the need to separate them between the client and server to maintain the distinction between their roles and reduce dependency on the particular server implementation. We bring a thoughtful approach to these questions and considerations, and seek an implementation which applies best software development practices.

In addition to augmenting MathBrush as described thus far, we investigate specifically working with differential equations on touch-based devices, from scratch. What might a quality application look like if it was dedicated to this task alone, as per the Big Bang Theory idea? Can we create an interface which makes use of modern touch-based gestures, pen-based input, and handwriting recognition, to deliver an optimal user experience for inputting differential equations? Making use of these instruments in an integrated fashion without compromising on coherence, simplicity and intuitiveness is a tall order. We propose a dynamic and interactive editor which accepts pen-based input of coefficients, recognizing the input in-place, while providing touch-based manipulation to allow the user to build their expression.

1.2 Existing technology

Does the Lenwoloppali differential equation scanner exist? How difficult would it be to create it? We consider the practicalities of the fictional project, and consider the current state of technology with regard to doing mathematics, and in particular, solving differential equations, on touch-based devices.

1.2.1 Mathematics and optical character recognition

The idea of using a smartphone's camera to photograph a differential equation suggests that optical character recognition (OCR) techniques would need to be employed to infer the mathematical content from the image. A simpler version of this problem is OCR for typeset mathematics, a problem which has been tackled with relative success. A notable solution is the Infty project [7].

Of course, recognizing mathematics in typeset documents is significantly easier than in arbitrary handwritten documents. Typeset mathematics have a clear and precise representation, whereas handwritten documents would generally be subject to noise, inaccuracies, and a high degree variability even among "correct" representations. In addition, a smartphone camera is not a reliable scanning device, and can add additional complications due to glare, angle, or brightness issues, for instance. We have not found any OCR tools developed specifically for photographed mathematics. Technology such as the Infty project could be applied, but with low expectations of success. It appears most of the research effort in this area is being directed either towards OCR for scanned typeset documents, or towards recognizing handwritten mathematics through ink-based interfaces, which are challenging problems in their own right.

1.2.2 Mobile applications

Few mobile applications exist which deal with differential equations, and none of them resemble a "differential equation scanner". The applications we have found are either instructional-based, or keyboard interfaces for solving differential equations. Examples of mobile applications which claim to solve ordinary differential equations include:

- IVP ODE Solver for Android by Luis Fernando Jr
- XPP for iOS by Ashutosh Mohan
- WolframAlpha for Android/iOS by Wolfram Group

Although these applications may work very well within their scope, they are not equipped to accept sketch-based input. That is, they all require the user to somehow convey the input using a keyboard. This public review of XPP by a user named wmotil sums it up: "[I] find this to be the easiest to use, *once you learn how to give it instructions*" (emphasis added). It is this learning curve that we want to remove, so that the user can use free-form input via the touch of a finger or stylus.

Examples of touch-based applications for recognition of handwritten mathematics include:

- Infty Project's InftyEditor handwriting interface [7]
- S Note on Samsung's Galaxy Note android-based series
- MyScript products by Vision Objects [12]: MathPad, Calculator, and Web Equation
- MathPad^2 from Brown University [8] (not related to MyScript MathPad).

InftyEditor allows the user to input mathematics through handwriting, but it is for the purpose of typesetting and it has no computational abilities. S Note provides impressive recognition capabilities, useful for mathematical note-taking but not typesetting or computing. MathPad excels at recognizing handwritten mathematics, and can export to Latex or MathML as a paid feature. However, it does not provide recognition correction or any computational support. Calculator recognizes basic handwritten mathematics reasonably well, and provides basic computations, it does not support higher level mathematics such as integrals and derivatives. Both apps use in-place recognition, where the recognized expression replaces the original ink, whereas Infty and S Note have separate views for the ink and the recognition which are both present at the same time.

Vision Objects is one of the leading pen-based math recognition engines [9], and has recently collaborated with WolframAlpha to integrate handwriting recognition with computation via a web application. MyScript Web Equation is a web-based interface for touch-based input of mathematics, and provides a link to the WolframAlpha mathematics engine (which powers the Mathematica system). The math recognition is equipped for high level mathematics and it is possible to hand-write differential equations (through a browser on an iPad, for example) and obtain solutions through WolframAlpha. However, again, the interface does not provide recognition correction capabilities. Also, since it does not accept input of multiple expressions, it cannot recognize objects like matrices and cannot be used to solve differential equations with initial conditions.

MathPad^2 is an experimental sketch-based system for exploring mathematics. It allows the user to create dynamic illustrations by combining handwritten mathematics and free-form diagrams. It is designed for sketch-based animations, rather than general computing. It is still in its infancy and not available for public use.

MathBrush provides a comprehensive set of features for modifying the ink and selecting alternate recognition results in the form of native iOS and Microsoft tablet apps. As we will explain in the chapters that follow, it can now be used for inputting differential equations, including initial conditions, as well as solving them.

1.3 Outline

The remainder of this thesis is organized as follows.

In Chapter 2, we discuss how a link was established between the math recognizer and math engine for general computing. We explain how we arrive at our implementation of a wrapper for the Sage math engine, and describe how the overall system fits together as a fundamental tool for computing with handwritten mathematics. We also describe how we introduce the graphing element into the MathBrush iPad application. We cover some of the interesting aspects of the implementation, as well as interface considerations.

Chapter 3 is allocated to the investigation of differential equations specifically, in the context of mobile touch-based devices. We will discuss how we adapt the recognizer to support differential equations with optional initial conditions, and how we extend the computational framework described in Chapter 2 to provide solutions to such inputs. Finally, we describe a vision for an entirely new application for working with differential equations, and in Chapter 4 we discuss the progress made thus far in the form of a prototype.

The final chapter concludes by recapitulating the contributions of this thesis, comparing and contrasting the MathBrush interface approach with the new interface approach to differential equations, and outlining the future work recommended in order to realize the full potential of the objectives initiated by this thesis.

The appendix outlines some of the more peripheral work that supports our overall objectives. We pay particular attention to a novel user interface for 3D rotations known as arcball. This interface exhibits an elegant implementation using quaternions, and so we briefly shift our focus onto quaternion mathematics to provide the necessary background. We also discuss our experiences with rendering mathematics using MathJax, and using it to experiment with in-place recognition in MathBrush.

Chapter 2

Linking a symbolic recognizer to a computer algebra system

Computer algebra systems are useful, but require certain protocols for input. It is up to the user to translate the mathematics into a form understandable by the system, whether by entering text or using a graphical interface, and this can be disruptive to the flow of research. It would be useful to be able to connect a computer algebra system directly to handwritten mathematics. Ideally, one would be able to write down a mathematical problem and instantly get answers. Our goal is to bridge the gap between recognition and computing as seamlessly as possible.

This task presents some interesting challenges. Currently, it is not practical to do "serious" computing on common client devices such as the iPad. Tablet devices are generally designed to be lightweight, and they use restrictive architectures with limited computing power. A way to circumvent these limitations is to delegate the computing tasks to a server with which the client communicates. Fortunately, mobile devices are becoming so ubiquitous and well-connected that server-based applications have become standard.

A further challenge is integrating MathBrush with the computer algebra system. It is necessary for the communication between the two systems to maintain semantic information. The computed results must be understood by the client as mathematical expressions, so that further operations can be performed in the same manner, if desired. If the result of an operation can only be displayed and nothing more, then we have a dead end.

In this chapter we describe our approach to addressing these challenges, and discuss our work towards an implementation.

2.1 Computer algebra systems

The most prominent computer algebra systems are commercial: Maple and Mathematica. Maple has been investigated as a math engine for MathBrush on Microsoft-based platforms, leading to the development of an interface for interacting with computer algebra systems. We prefer to avoid using commercial systems as they come with potential licensing restrictions and complications. Open-source systems do not have these limitations, and a number of comparable systems are under active development. In particular, Sage [2] is an open-source mathematical software system that has combined several powerful systems and tools into a single interface written in Python, a high-level programming language widely used by researchers. As one of the most complete and flexible open-source systems available for doing mathematics, it is a natural choice for providing the back-end computing capabilities. Its support and usage has become very widespread and we consider it to be the best noncommercial candidate to serve as the computer algebra system for this application.

2.2 Expression representations

The primary design challenge is setting up a communication protocol between the two systems. The output of the recognizer must translate to input for the computer algebra system, and vice versa. Ideally, the protocol will be as generic as possible, enabling compatibility with other systems as future needs may arise. A universal intermediate representation capturing the semantics of the math expression would be ideal. MathML is an XML-based markup language for describing the structure and content of mathematical expressions, and is evolving as a standard for communication of mathematics. Since MathBrush already produces and interprets MathML, this appears to be an ideal choice. Unfortunately however, Sage has almost no MathML support. There has been some discussion about why this is the case, and the apparent reason is simply that the task is too complex, and/or that there is not enough interest in making it happen. William Stein, the originator of Sage, said in an online Google Groups discussion [3]:

"At ISSAC I saw a demo of how content MathML interfacing between different programs actually works when fully implemented in a particular case, and I think I'm glad Sage doesn't use it since it is way more complicated than what Sage currently does, and would take too much work to implement. The Sage pexpect[1] interfaces are as KISS (keep it simple stupid) as possible, and MathML isn't KISS."

[1]Pexpect is a Python module for controlling other applications.

Sage is not the only major math system which lacks MathML support. Even commercial systems like Maple, which claim to support MathML, reportedly are not exempt from reliability issues. We came to the conclusion that MathML has been successfully adopted as a standard for math presentation, but not (yet) for semantic content representation. There has even been some debate recently amongst Mozilla developers whether or not to drop MathML support [4].

Because of these concerns we seek an alternative approach to MathML. The overall solution involves three processes and three mathematical representations, as illustrated in Figure 2.1.

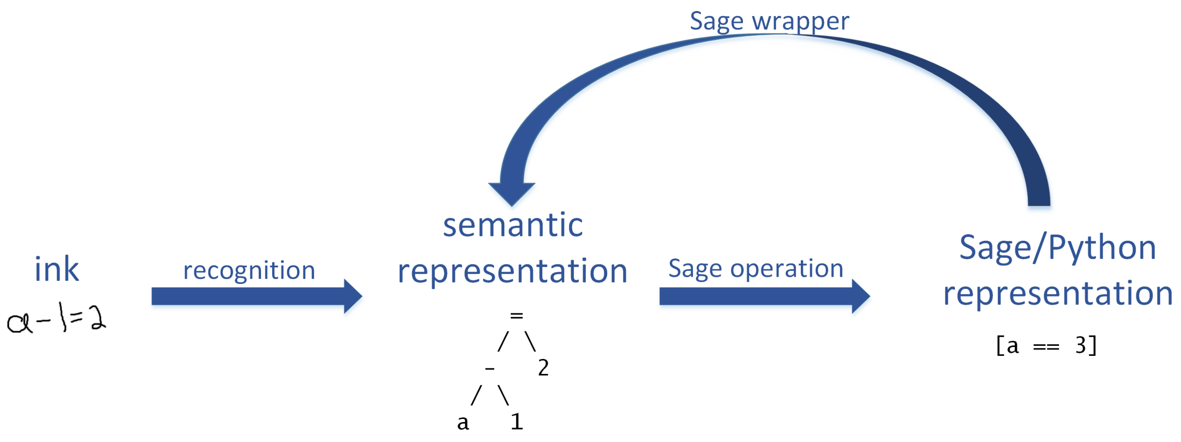


Figure 2.1: Expression representations

2.3 Sage wrapper

Sage is designed to be used as an interactive console application. An initial prototype developed by Scott MacLean used text-based interaction with a Sage process. Sage commands were generated based on the request, and results were obtained from Sage in the form of Latex expressions. The server converted the Latex into MathML and sent it back to the client which is equipped with a MathML renderer. The problem with this approach is that Latex represents presentation information, not semantics. It may be easy for a human to infer the semantic information from rendered Latex, but it is not clear how to algorithmically convert Latex into an expression tree. The translation from Latex to MathML is not guaranteed to preserve the correct semantic information and therefore introduces the possibility of error.

A second approach is to make use of Sage's internal symbolic expression structures. Output from Sage is converted from its internal representation to text when presented to the screen in the console application. Interacting directly with the expression tree (semantic representation) would be a more reliable way of obtaining the semantic information than using the output string (presentation representation). We refer to the expression tree structure developed for MathBrush as an SCG[2] expression tree. Sage's expression object can be traversed to construct the required SCG expression tree representation. Sage expression format uses various types of objects depending on the nature of the operation performed, but there are only a handful of common types that need to be understood in order to extract the semantic information.

We define an expression tree class in Python which closely parallels the SCG format defined in C++, and develop a Python wrapper to facilitate conversion to this type. Sage is Python-based and it is possible to do Python programming in Sage's Python (i.e. the version of Python provided by Sage which has references to the Sage libraries). Sage can then be used as a Python module. In this way we can programmatically invoke Sage and use its functions and object representations, and construct the appropriate expression tree.

The Python C API can be used to initiate this process from C++ code, obtain the resulting Python expression tree, and construct the SCG expression tree. The API requires considerable boilerplate code which can make it tedious to use. However, once the code is in place it works reliably.

2.4 Implementation

A simple client-server API was developed to facilitate communication between the MathBrush client and the computer algebra server. The client-communication is established via a socket connection. An open-source daemon, `xinetd` (extended Internet daemon), runs on the server machine listening for incoming client requests and delegating such requests to the server program (a server process is spawned for each request). Figure 2.2 gives an overall view of this architecture and shows how the Sage wrapper fits into the picture.

[2]Symbolic Computation Group

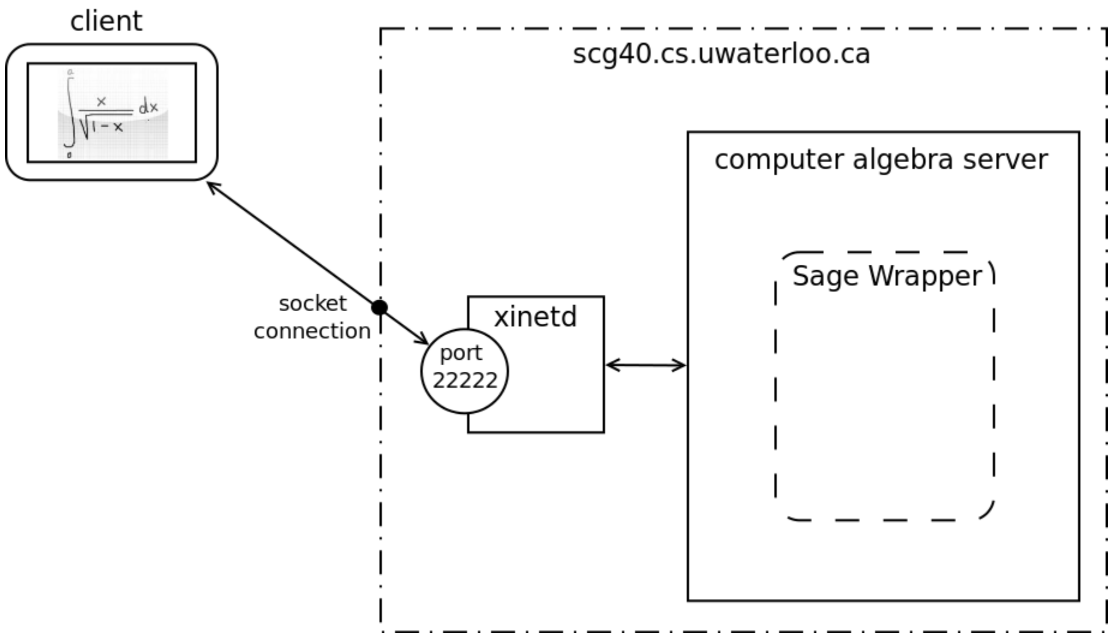


Figure 2.2: Architecture

The main work of the implementation, beyond setting up the required infrastructure, is the Python class for wrapping Sage, called `SageWrapper`. The class is designed to wrap Sage objects using a recursive constructor. The construction is determined by the type and the content of the input. An example of the usage corresponding to integer factorization is:

```
expr = ExpressionTree(factor(n)).
```

This implementation is an application of the wrapper (or adapter) design pattern, where one interface is adapted for another via an intermediary.

An incidental advantage of this approach is that there is no explicit dependency on Sage. The implementation is not tied to Sage; rather, it is tied to Python. Sage is used as a Python module, and any other Python module could also be used. This provides flexibility because it opens up access to anything that can be done in Python. Since Python is becoming a standard tool in research and computing, our server can be useful as a model for accessing open-source Python tools from within a C++ application.

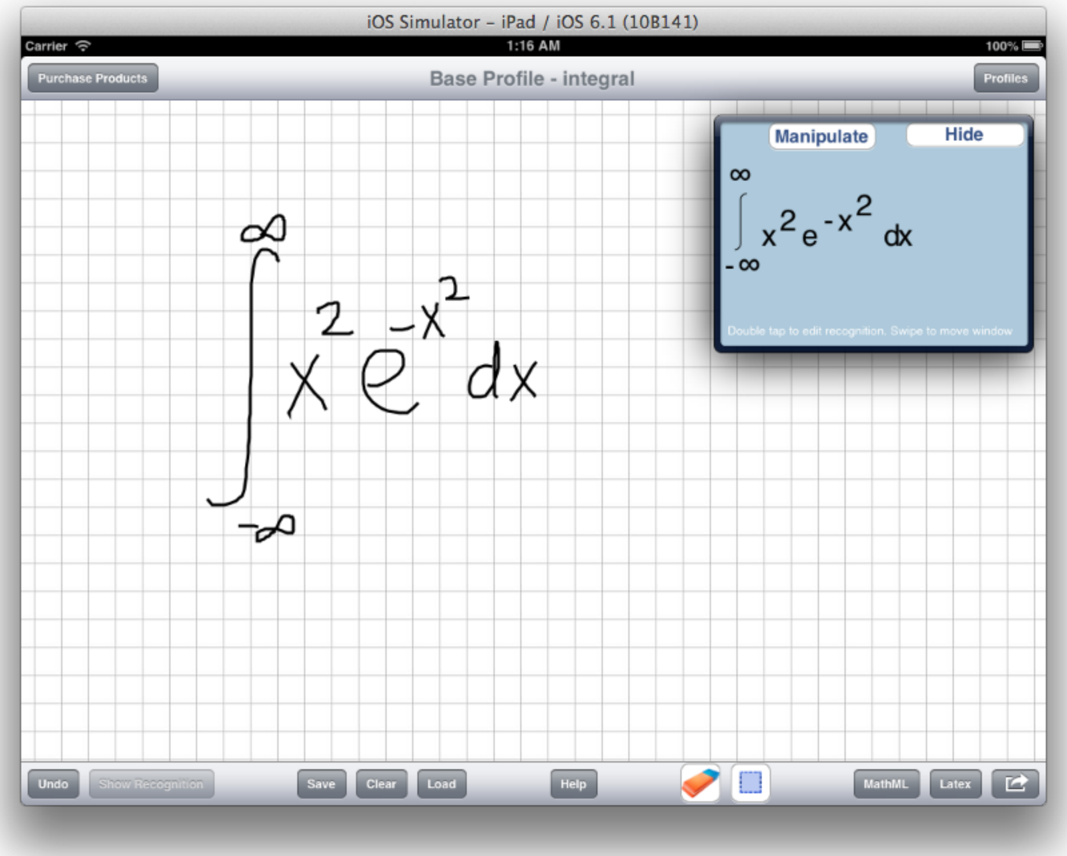
Thus far, the implementation has been based on recognizing and displaying mathematical expressions. However, the visualization of these expressions is also important, and this requires a new kind of collaboration between the client and server. As we will see in Chapter 3, plotting becomes an important feature when dealing with numerical solutions to differential equations. The next section, following the example, discusses the introduction of plotting features in MathBrush for iPad.

2.4.1 Example

We present a minimal use case of the MathBrush system for general mathematics recognition and computing. Suppose a user wishes to compute the integral

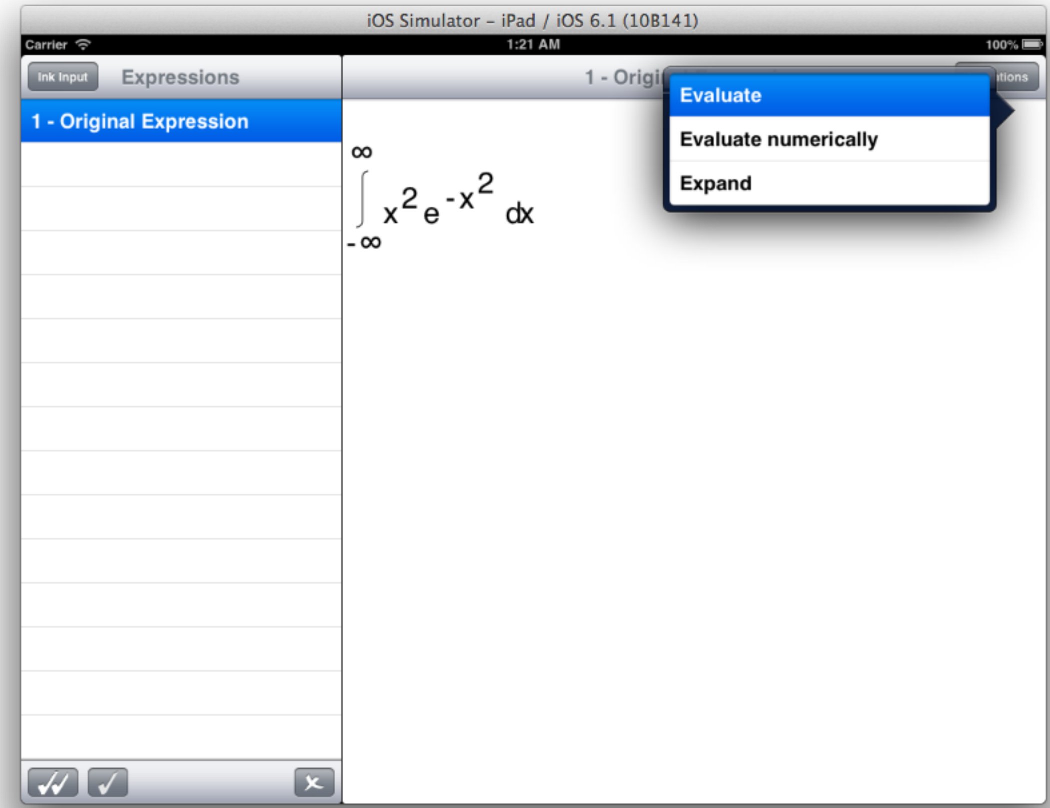
$$\int_{-\infty}^{\infty} x^2 e^{-x^2}\, dx.$$

The user enters the integral via pen-based input:



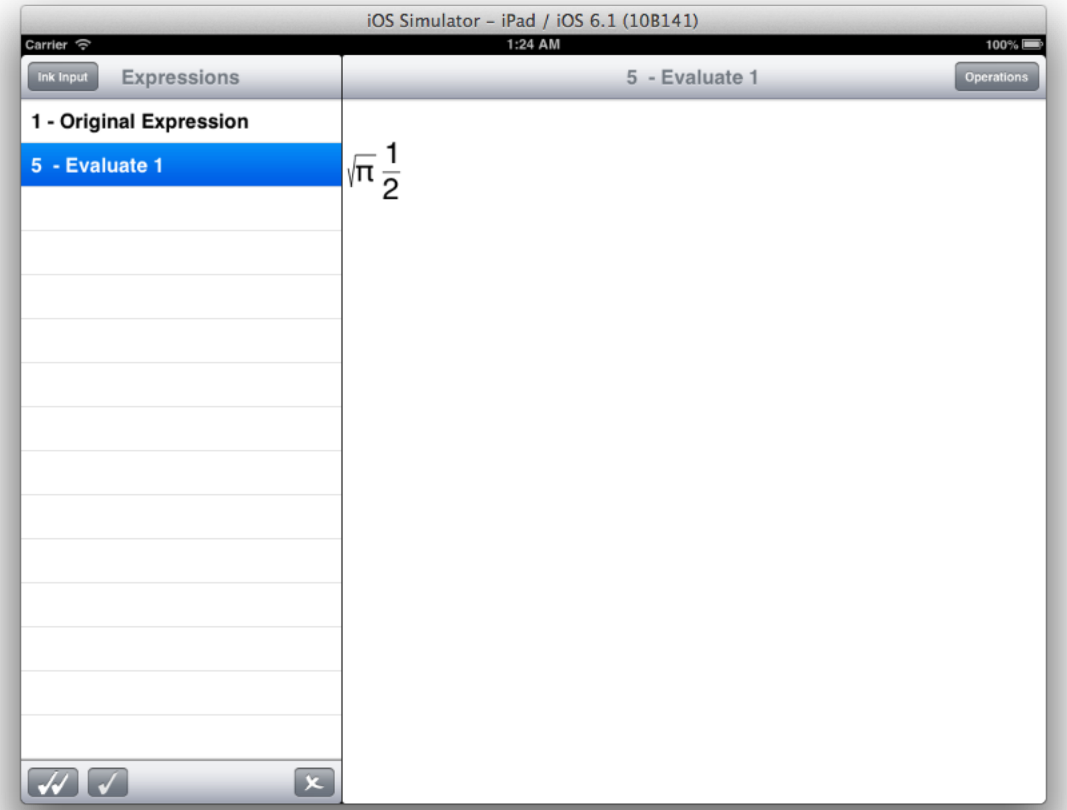
$$\int_{-\infty}^{\infty} x^2 e^{-x^2} dx$$

The user taps the Manipulate button to manipulate the expression with computer algebra commands. This takes the user to the computing interface which links to our Sage server. Tapping the operations button reveals the available operations to be performed on the expression. The user selects Evaluate:

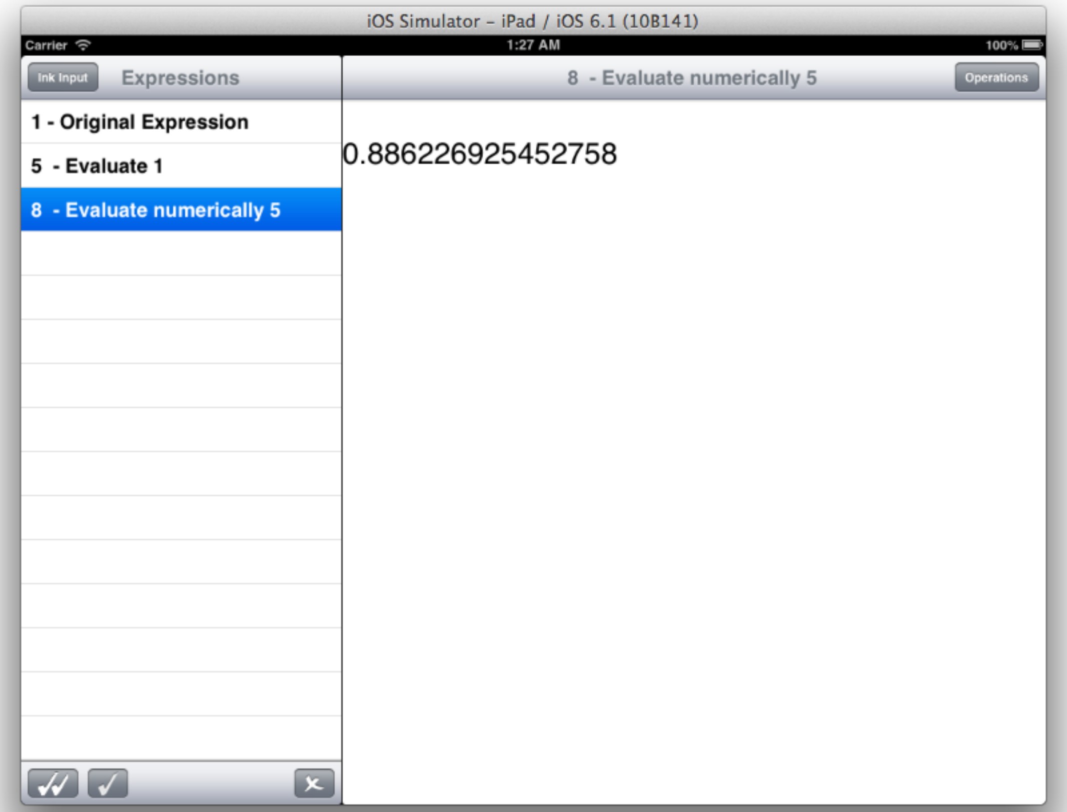


$$\int_{-\infty}^{\infty} x^2 e^{-x^2}\, dx$$

This action invokes the computer algebra server according to the implementation described previously. The resulting expression is added to the list of expressions in the left sidebar.



If a numerical value is desired, the user may select the “Evaluate numerically” operation on the second expression, yielding a third expression:


iOS Simulator – iPad / iOS 6.1 (10B141)
Carrier
1:27 AM
100%
Ink Input
Expressions
8 - Evaluate numerically 5
Operations
1 - Original Expression
5 - Evaluate 1
8 - Evaluate numerically 5

0.886226925452758

2.5 Graphics for mathematics

Our system has been designed to work with mathematical expressions and their representations. However, an important feature for any math system is the ability to visualize solutions. This is especially important when solving differential equations numerically, as tables of values are not very interesting visually. Furthermore, modern touch-based devices such as the iPad are an ideal instrument for presenting visual content, and can provide a convenient and natural way to interact with the graphs through touch gestures.

Thus far the approach has been to delegate computational tasks to the computer algebra server and display the result. However, the task of plotting, and the work it entails, cannot entirely be delegated to the server. On a basic level, plotting involves computing a set of points and displaying them to the user. However, several considerations are involved in displaying plot data, such as which colours to use, how to set up the axes with labels and tick marks, whether or not to adjust the scale or range of any of the axes to make the plot more readable, and so on. Moreover, once the data is displayed, it is desirable to enable the user to interact with it, for example by zooming or translating. Therefore, once the computer algebra server has computed the plot data, the rest is up to the client. It is therefore necessary to develop (or leverage existing) client tools to enable the rich user experience that would be expected when working with plots on an iPad.

We introduce basic plotting functionality into the MathBrush iPad app using Apple's CoreGraphics framework for two-dimensional plotting, and OpenGL ES 2.0 for three-dimensional plotting. Currently, the implementations are minimal with the exception of a special user interface for three-dimensional rotations called *arcball*, which is interesting in its own right. We include a discussion of the arcball interface and our implementation in the appendix (A.1). Examples of 2D and 3D plotting in action can be found throughout sections 3.1.3, 3.1.4 and A.1.2.

2.5.1 Plot data

In order to introduce plotting into the application, we must determine how to work with plot data. We are interested in supporting plotting in both two and three dimensions.
In two dimensions, plot data consists essentially of a set of points (x_i, y_i), for $i = 0, \ldots, n$, for some number of points, n. Through experimentation, we found that 2D plot data could be extracted from the result of Sage's `plot` command with a few lines of Python code, essentially boiling it down to a list of x-values and a list of y-values.

In the three-dimensional case, the plot data is a little more complex than merely a set of points (x_i, y_i, z_i), for $i = 0, \ldots, n$. How do those points relate to one another? Do they represent a curve or a surface? How do the points define the geometry of the surface? Typically, surface geometry is represented in terms of triangular or quadrilateral faces. Again, through some experimentation, we found that the 3D plot data could be extracted from Sage's plot representation by invoking the method `json_repr`[3]. This method returns the plot data in the form of a string containing the set of vertices (points) as well as the

[3]JSON stands for JavaScript Object Notation, and is an open standard format that uses human-readable text to transmit data objects consisting of attribute-value pairs.

faces (identified by indices into the list of vertices), and it also includes colour information (which is customizable through the options passed to Sage).

The JSON representation of a 3D plot can be viewed as a context-free grammar, described by the following rules.

$$\begin{aligned}
<\mathit{repr}> &\rightarrow [\text{``} <\mathit{str}> \text{''}] | [<\mathit{repr}>, <\mathit{repr}>, ..., <\mathit{repr}>] \\
<\mathit{str}> &\rightarrow \{\text{vertices} :<\mathit{vertices}>, \text{faces} :<\mathit{faces}>, \text{color} :<\mathit{color}>\} \\
<\mathit{vertices}> &\rightarrow [<\mathit{vertex}>, <\mathit{vertex}>, ..., <\mathit{vertex}>] \\
<\mathit{vertex}> &\rightarrow \{\text{x} : \#, \text{y} : \#, \text{z} : \#\} \\
<\mathit{faces}> &\rightarrow [<\mathit{face}>, <\mathit{face}>, ..., <\mathit{face}>] \\
<\mathit{face}> &\rightarrow [\#, \#, \#, \#] | [\#, \#, \#] \\
<\mathit{color}> &\rightarrow \text{`rrggbb'}
\end{aligned}$$

Here, nonterminals are enclosed in angle brackets, # represents a number (vertex coordinate are floats and face elements are integers), and r, g, b represent hexadecimal digits corresponding to the colours red, green, and blue, respectively. Note that a face can be represented as either a quadrilateral (often simply called quad) or a triangle. This well-defined structure of the JSON representation makes the parsing process reliable and straightforward to implement (albeit tedious).

Working with plots goes beyond the protocol we have established at this stage, which exclusively communicates in terms of math expression trees. Here, we are dealing with an entirely different form of data. In order to work with plots, we create a generic plot structure in C++ which can be easily used by both the client (in Objective-C code) and the server (in C++ code). We populate the structure by obtaining Sage's output as a string through the Python C-API and parsing out the data using the approach described above.

Working with floats

The plot structure we created needs to be transported between client and server. Since plot data primarily consists of floating point numbers, this task warrants caution: floats can be "fragile" when transported across networks due to the lack of a standard network representation[4]. Integers however are not subject to the same potential issues. We chose to safeguard against this issue by devising a method to convert between floats and ints[5].

[4]http://en.wikipedia.org/wiki/Endianness#Floating-point_and_endianness

[5]Scott MacLean contributed this approach.

Consider a floating point number

$$x = s * 2^p$$

where p is an integral exponent and $s \in [0.5, 1.0)$ is the binary significand. Multiplying s by 2^{31} and truncating yields an integer (representable using the `int` data type) which preserves the sequence of binary digits in s (note that 31 is the largest exponent that can be used without causing overflow). Therefore we have an encoding scheme for the floating point number x: we store the pair of integers,

$$(p, s * 2^{31}).$$

To decode, we simply reverse the process: given $(p, s' = s * 2^{31})$, we have

$$x = s' * 2^{-31} * 2^p.$$

The C functions `ldexp` and `frexp` can be used to implement this scheme.

Chapter 3

Differential equation solver

In this chapter we discuss our investigation of solving differential equations on touch-based devices. In particular, two approaches are described. First, we describe how we introduce the capability of solving differential equations into the MathBrush system, utilizing the link to Sage as discussed in the previous chapter. Then, we describe a vision for an application exclusively devoted to solving differential equations, and discuss the progress achieved through building a prototype.

3.1 Solving ODEs in MathBrush

Our first approach to a differential equation solver is to build upon the established framework for doing computations in MathBrush by introducing ODE-solving capabilities. The user would simply write down their differential equation in MathBrush, which would recognize it as such, and connect the user to Sage to obtain a solution. What is attractive about this interface is that it's very "hands-off" (although certainly not in the literal sense). The user can simply write down the equation using free-form pen-based input, just as they would on a blank sheet of paper, but with the benefit of having a computer algebra server at their fingertips. The MathBrush recognizer is designed to recognize general mathematical expressions, so naturally differential equations are already supported. However, some minor modifications were required in order to handle them appropriately.

There are essentially two forms of differential equation input which we wish to accept:

- a single ordinary differential equation (ODE), and

- an ODE along with initial condition equations, or an initial value problem (IVP).

A new type of expression is required for the original recognizer to support IVPs. Since an IVP consists of multiple expressions, we need to recognize multiple expressions as part of the same overall expression, which we term a multi-expression. Initially, multiple expressions would only be recognized in MathBrush if they were in a matrix. By leveraging this existing capability without requiring the presence of brackets, we are able to recognize multiple expressions, and therefore, IVPs. The recognition of prime symbols for derivative notation is also introduced, enabling primes to be used, with or without parentheses, to express derivatives.

In order to solve differential equations in MathBrush, the expressions must be translated into the appropriate form of input for Sage's ODE solvers. For basic expressions, translating to Sage is straightforward because it can be accomplished via a single traversal of the expression tree. However, ODEs are a more sophisticated form of expression, and the entire expression must be comprehensively examined in order to identify the derivatives and deduce the independent and dependent variables. A derivative has a high-level semantic meaning, but the recognizer is built on more primitive semantic meanings such as variables, fractions, arithmetic operations, equations, and so on. For example, the expression $\frac{dx}{dt}$ is recognized as a fraction where the numerator and denominator are products of two variables, but it takes a holistic understanding of the expression to recognize its meaning as a first-order derivative of variable x with respect to variable t. Also, whenever a variable is encountered in the expression, it is not necessarily clear what it is (i.e. independent variable, dependent variable, or parameter) until the whole expression has been examined. Furthermore, derivatives can be represented in various notations, such as Leibniz (using differentials) or Lagrange (using primes). It is necessary to develop a comprehensive approach to uncover the high-level semantic meanings from the expression.

3.1.1 ODE Parser

We develop a C++ library to parse out the ODE and IVP semantics from recognized expressions, in order to be able to convey that information to the computer algebra server. The general approach is to assume the given expression is either an ODE or an IVP, and parse it as such. A breakdown in the parsing process means the expression is not an ODE or an IVP.

The first task is to be able to identify a single derivative of any order in either Leibniz or Lagrange notation. With that capability in place, the workflow essentially consists of finding answers to the following three questions about a given expression:

1. Is it an ODE?
2. Is it an initial condition?
3. Is it an IVP?

If the given expression is an equation, and parsing the left-hand and right-hand sides uncovers at least one valid derivative term, then the answer to the first question is "yes". If the given expression is an equation, and the left-hand side represents a derivative evaluated at some basic[1] expression, and the right-hand side is a basic[1] expression, then the answer to the second question is "yes". The third question can be phrased in terms of the first two questions. An IVP is represented by a multi-expression (described previously) where one of the expressions is an ODE and the rest are initial conditions. If a complete set of initial conditions is specified (consistent with the order of the ODE) then the answer is "yes".

The parser captures all of the information deduced through this process of answering questions, so that at the end the high level meaning of the expression is known and the corresponding information is recorded. With this information, the required Sage code can be generated and subsequently executed to obtain a solution.

3.1.2 Sage's ODE solvers

Sage's differential equation solvers are simply wrappers for solvers provided by existing open-source packages including Maxima and SciPy. We use Maxima's symbolic and numerical ODE solvers in our implementation. A solution will take the form of either a symbolic expression or a numerical solution as a set of data points. Symbolic expressions are handled in the same way as those resulting from the more primitive Sage operations. However, numerical solutions, which take the form of tables of values, are treated as plots. In both cases, the process works seamlessly within the established framework for integrating with Sage as described in the previous chapter.

Once we have generated the Sage code representation of the expression to be solved, it can be inserted as an argument to the appropriate Sage function.

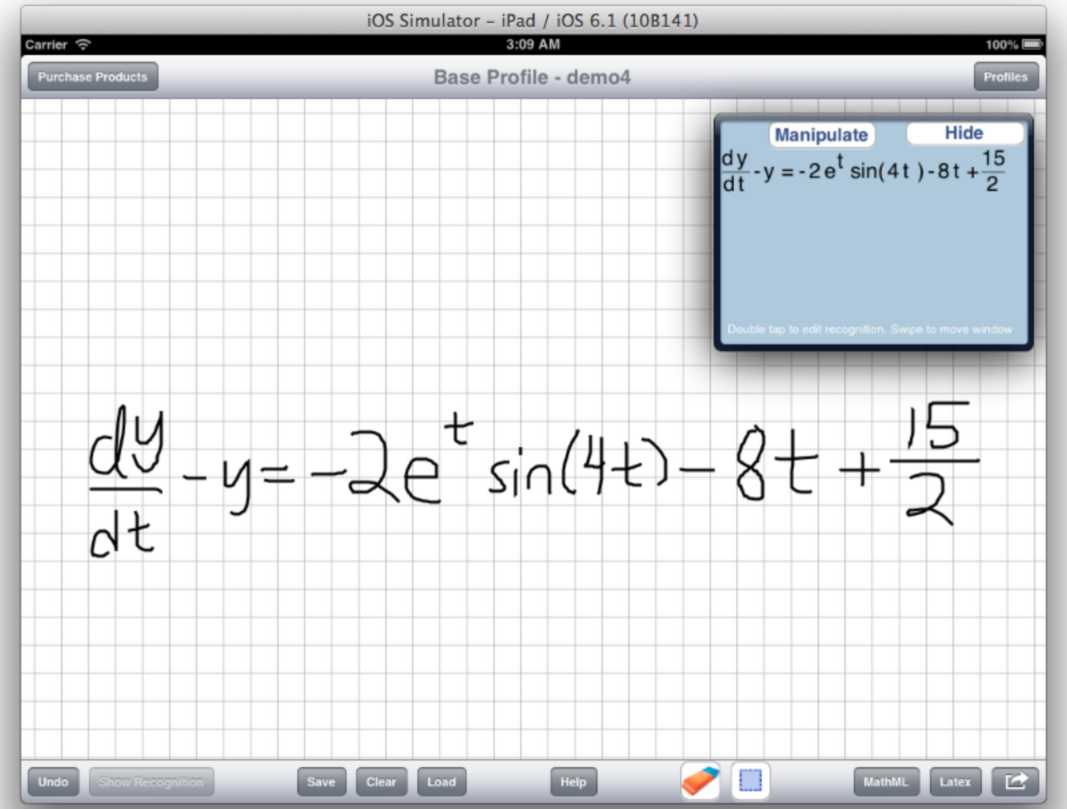
[1]By basic we mean an arithmetic expression that is constant with respect to the dependent and independent variables.

3.1.3 Examples

Consider the following linear first-order ODE.

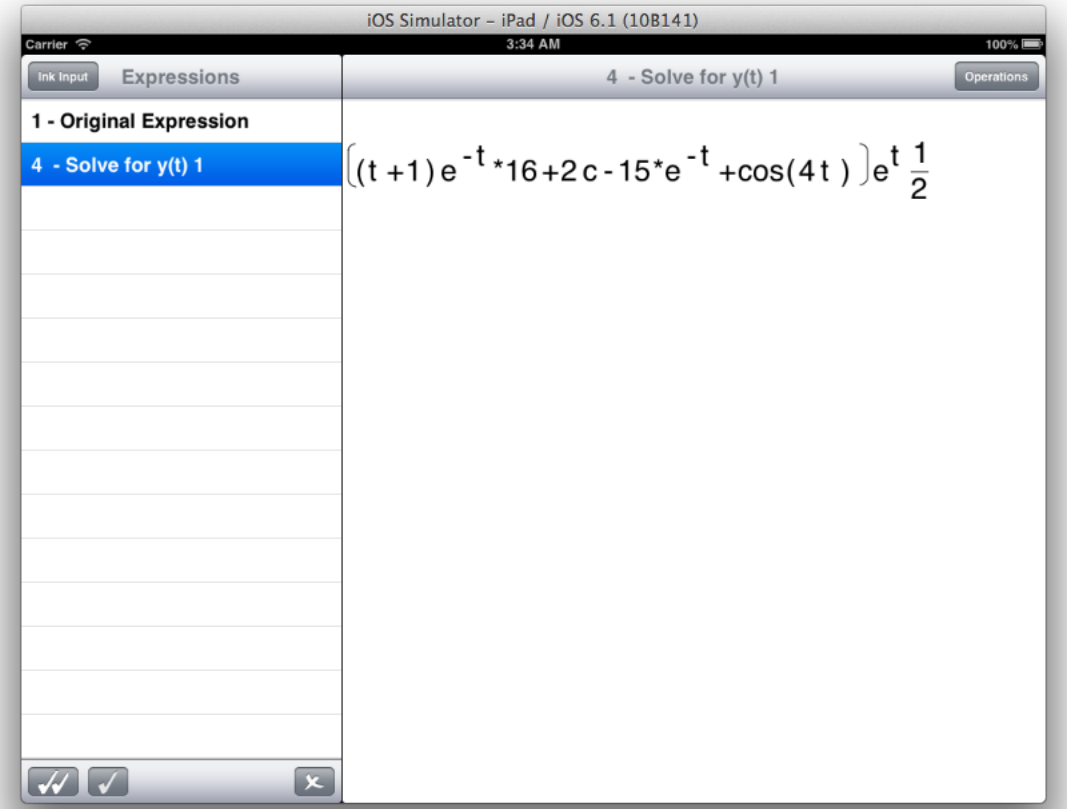
$$\frac{dy}{dt} - y = -2e^{-t}\sin(4t) - 8t + \frac{15}{2}$$

The equation may be entered in MathBrush on the iPad via pen-based input as follows.



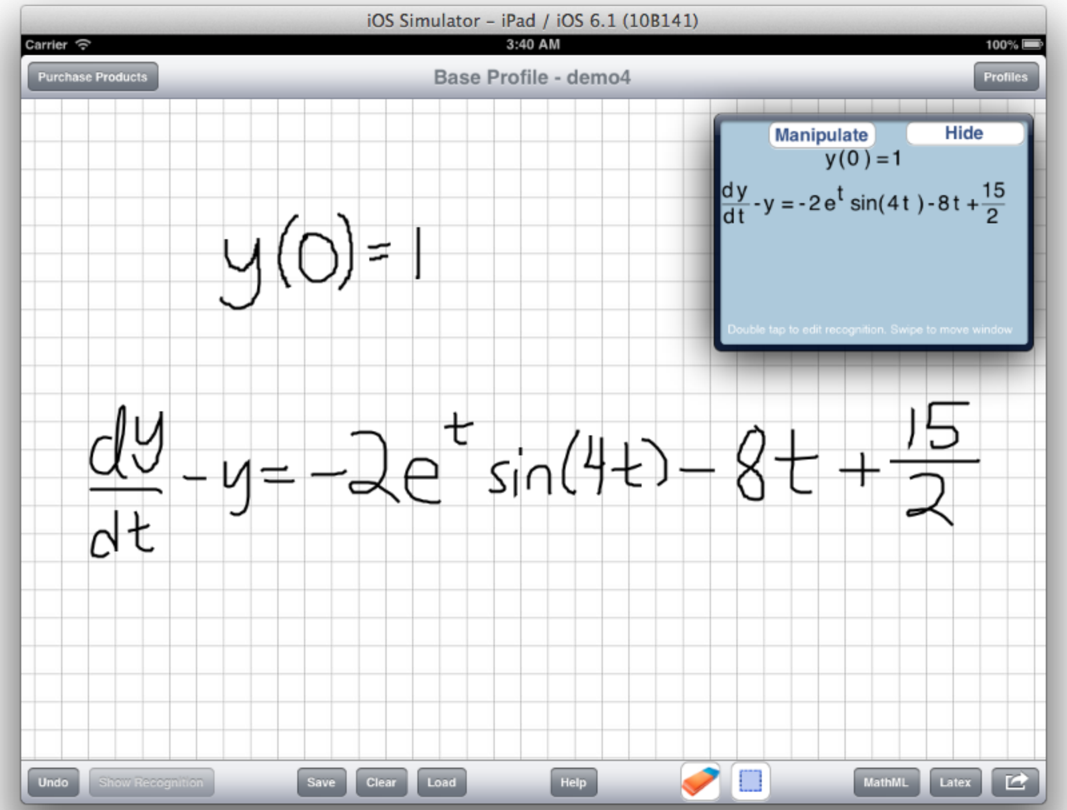
$\frac{dy}{dt}$ -y = -2 e^{t} sin(4t) -8t + $\frac{15}{2}$

Using the link to Sage we may solve this ODE symbolically:



$\left((t+1)e^{-t}*16+2c-15*e^{-t}+\cos(4t)\right)e^{t}\frac{1}{2}$

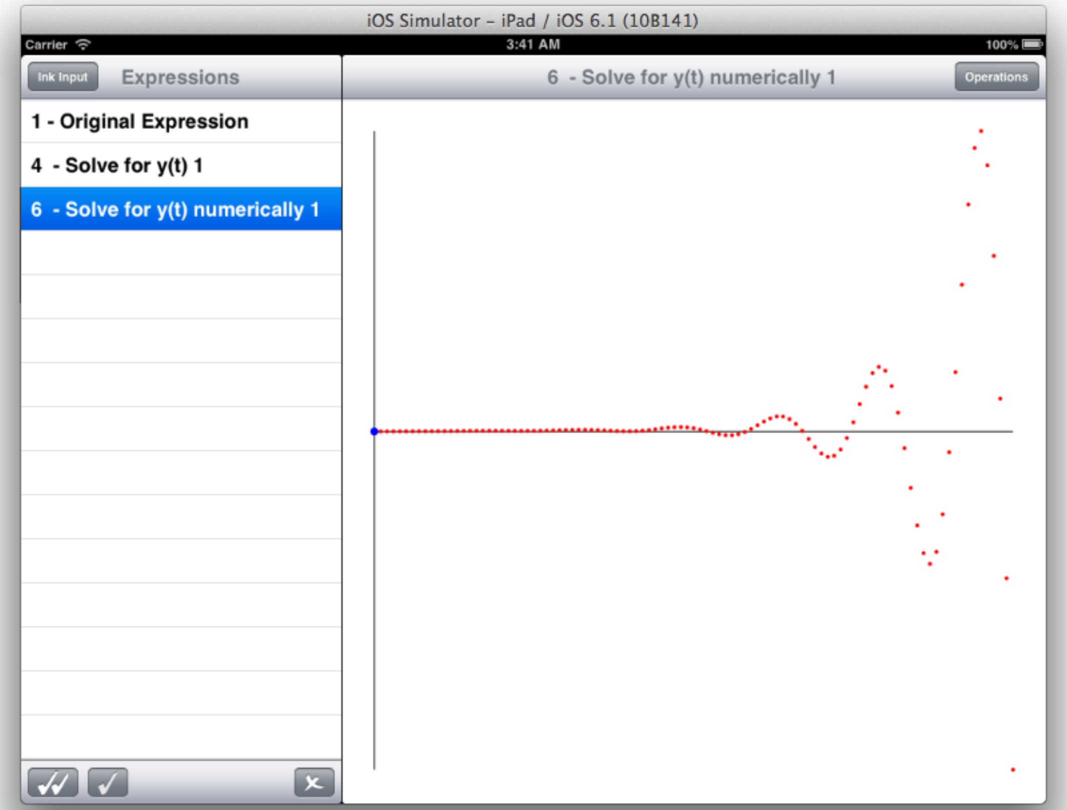
Symbolic solutions to ODEs often do not exist and hence one must revert to numerical solutions. In order to obtain a particular solution we must provide initial conditions. We may add initial conditions to the previous example as follows.



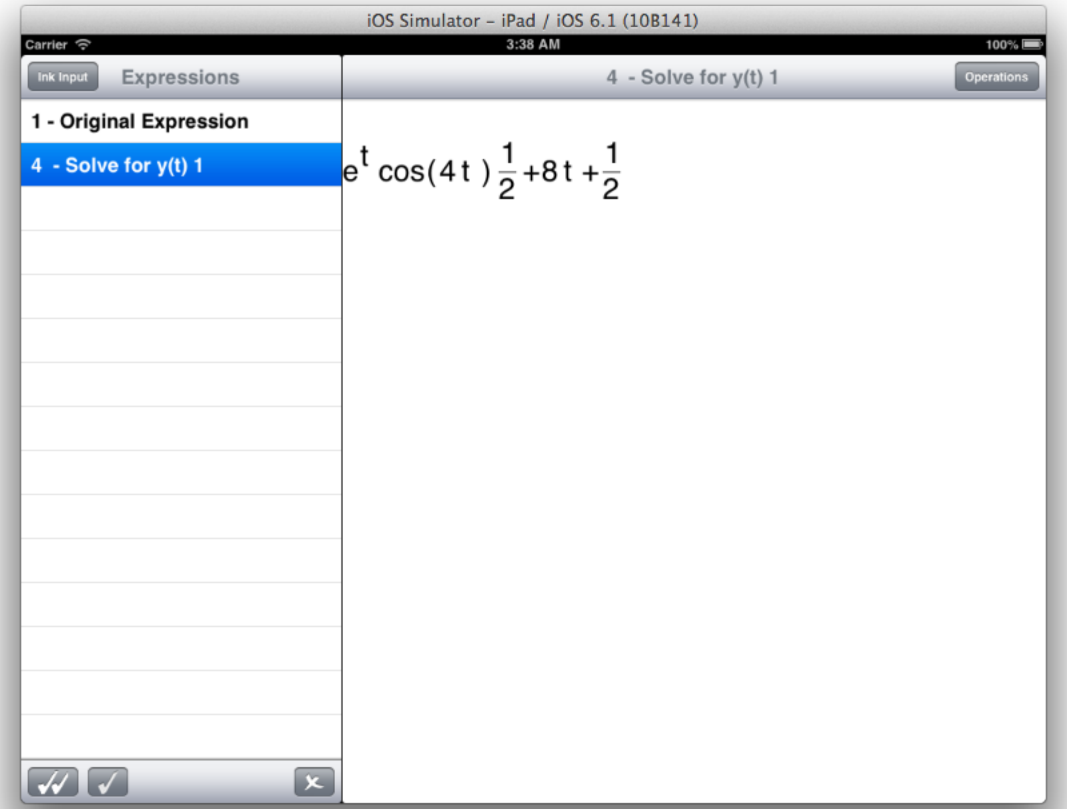
$y(0)=1$

$$\frac{dy}{dt}-y=-2e^{t}\sin(4t)-8t+\frac{15}{2}$$

This initial value problem may be solved numerically. Since the numerical solution consists of a table of values, the result is displayed as a plot:

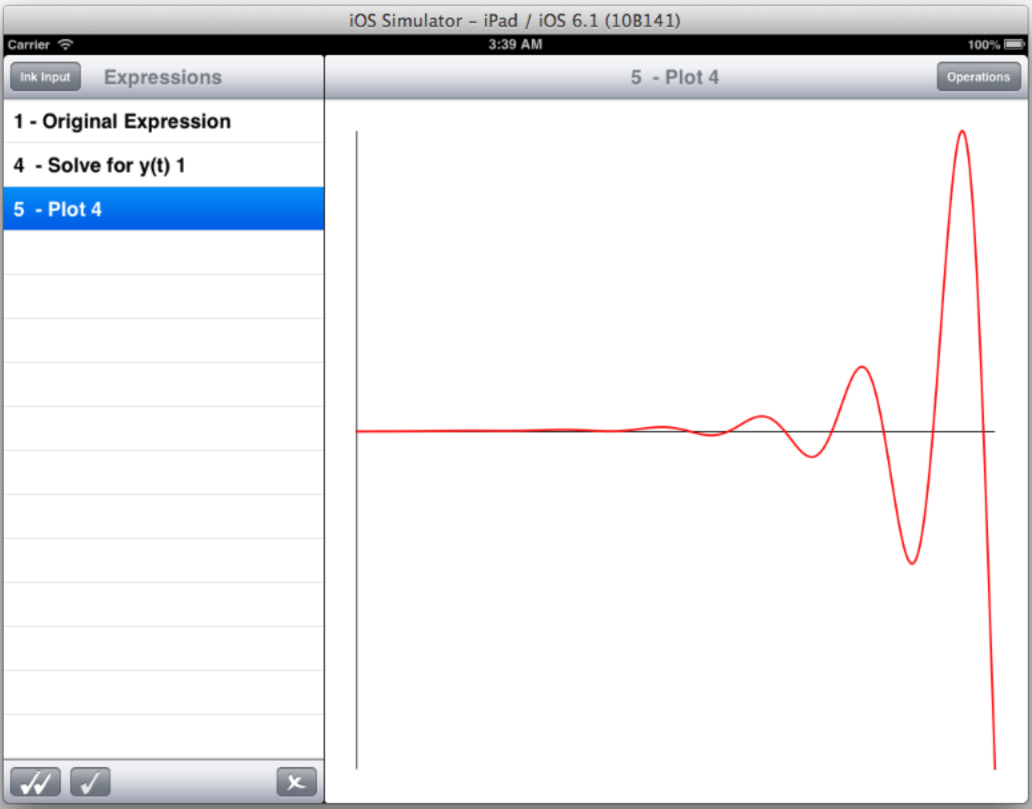

iOS Simulator – iPad / iOS 6.1 (10B141)
Carrier
3:41 AM
100%
Ink Input
Expressions
6 - Solve for y(t) numerically 1
Operations
1 - Original Expression
4 - Solve for y(t) 1
6 - Solve for y(t) numerically 1

In this case, we may solve the IVP symbolically as well:



$$e^{t}\cos(4t)\frac{1}{2}+8t+\frac{1}{2}$$

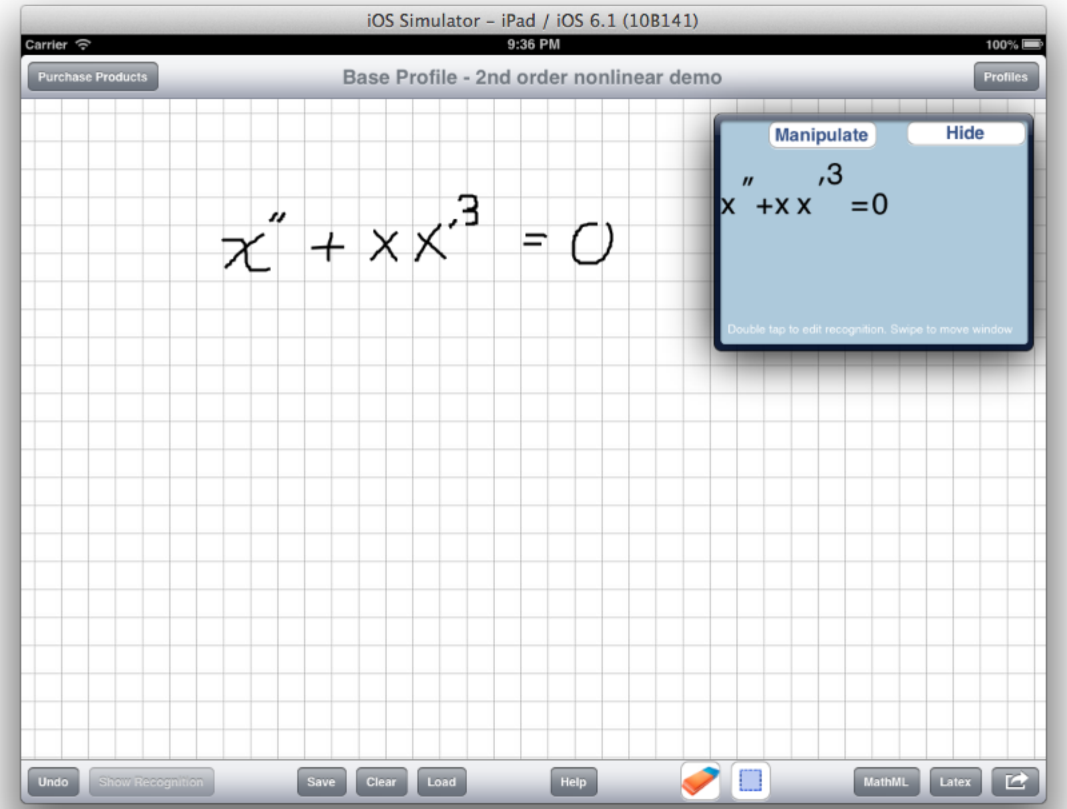
We may plot the symbolic solution as follows.



In addition, we may work with higher-order equations. For example, consider the second-order nonlinear ODE:

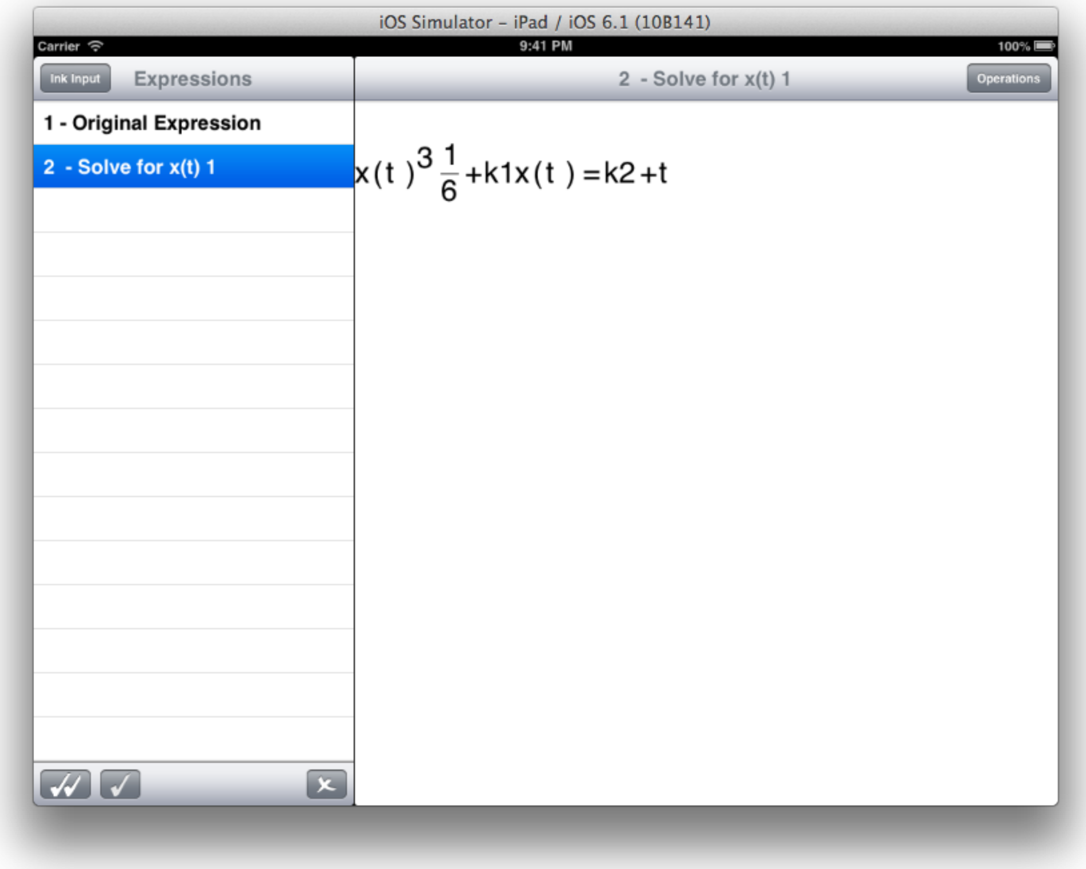
$$x'' + xx'^3 = 0.$$

The equation may be entered in ink form:


iOS Simulator – iPad / iOS 6.1 (10B141)
Carrier
9:36 PM
100%
Purchase Products
Base Profile - 2nd order nonlinear demo
Profiles
Manipulate
Hide
Double tap to edit recognition. Swipe to move window
Undo
Show Recognition
Save
Clear
Load
Help
MathML
Latex

$x'' + x x'^3 = 0$

Navigating to the computer algebra interface and selecting to solve the ODE leads to the solution:

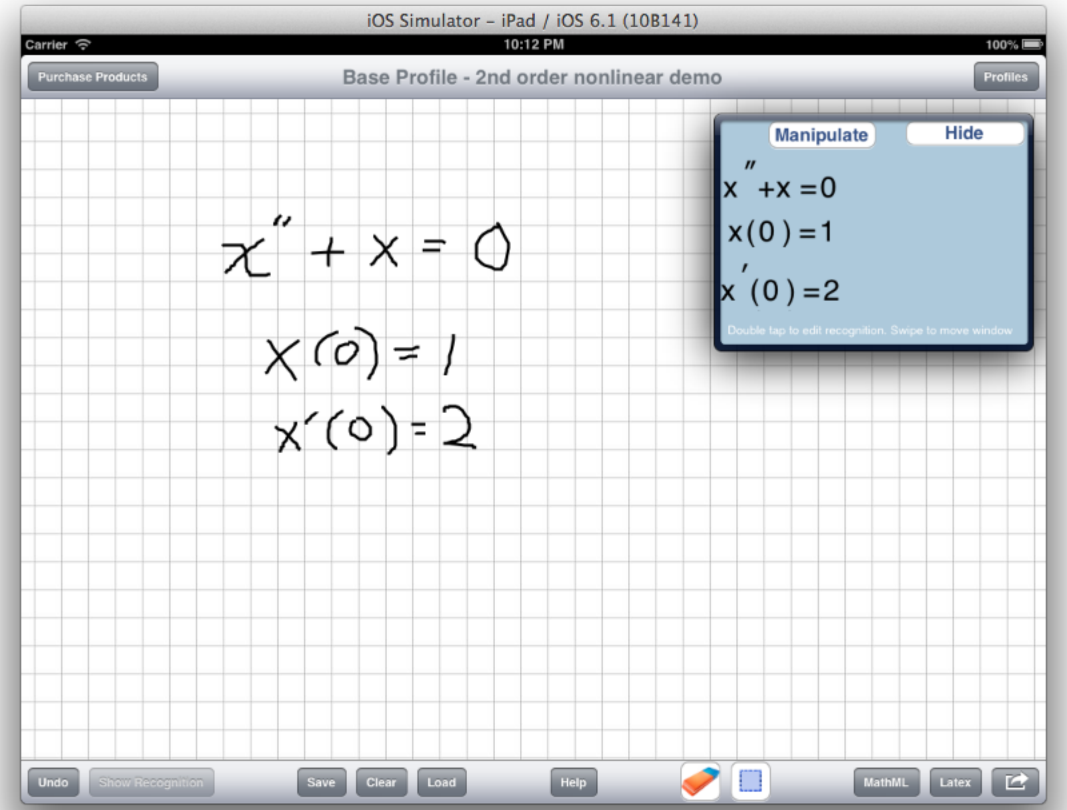


1 - Original Expression

2 - Solve for x(t) 1

$$x(t)^3 \frac{1}{6} + k1x(t) = k2 + t$$

This particular example cannot be solved by Sage using initial conditions, although it can be solved with boundary conditions. Since the current version of the ODE parser does not look for boundary conditions, we would not be able to proceed in that direction. However, suppose we modify the expression to make it linear, and provide initial conditions:

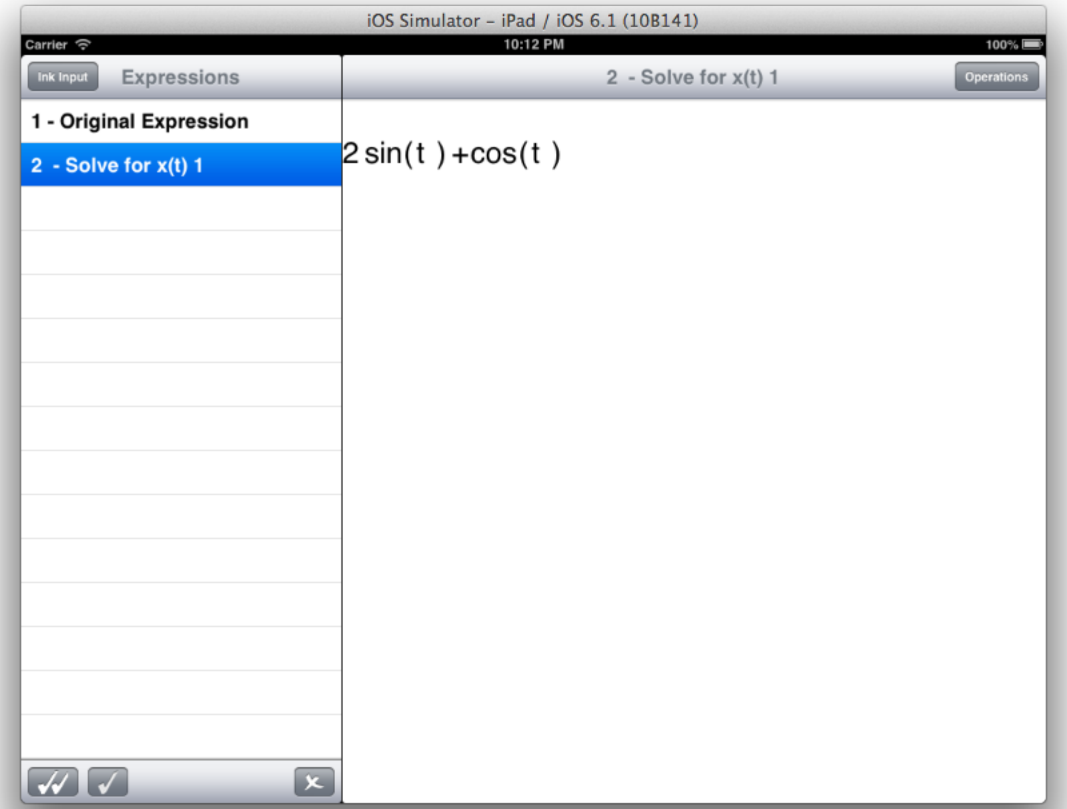

iOS Simulator – iPad / iOS 6.1 (10B141)
Carrier
10:12 PM
100%
Purchase Products
Base Profile - 2nd order nonlinear demo
Profiles
Manipulate
Hide
Double tap to edit recognition. Swipe to move window
Undo
Show Recognition
Save
Clear
Load
Help
MathML
Latex

$x'' + x = 0$

$x(0) = 1$

$x'(0) = 2$

We may solve this initial value problem:



2 sin(t) +cos(t)

Note: if only one initial condition is provided, there would be no available operations, since the ODE parser would not find a complete set of initial conditions and therefore would not recognize the multi-expression as a valid initial value problem.

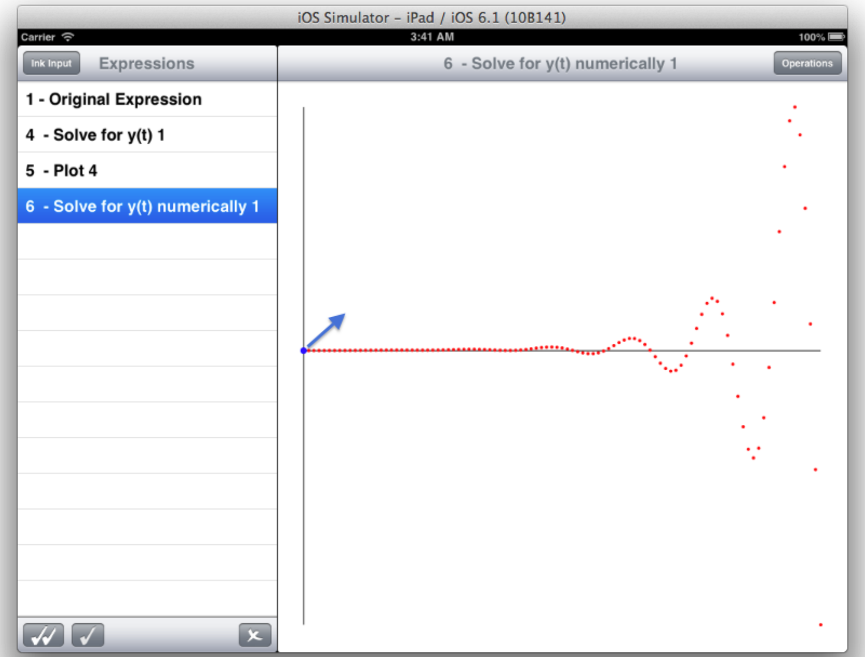
3.1.4 Moving the initial condition

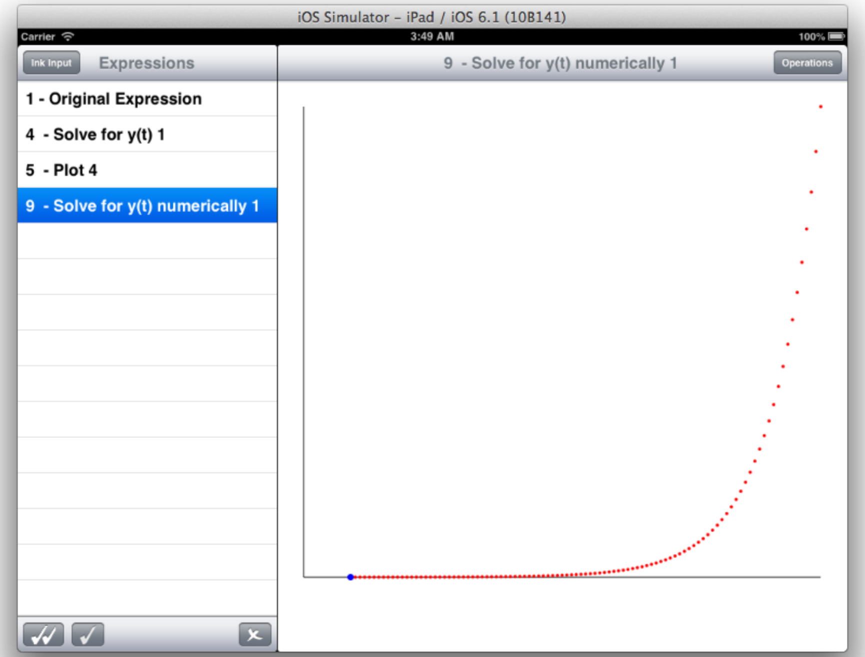
Numerical solutions to differential equations are displayed as plots rather than the raw tables of numerical values produced by the solver. In this context, a useful feature is the ability to modify the initial condition and "immediately" see the affect. Rather than going back to modify the ink representing the initial conditions, it is desirable to simply touch and drag the initial point directly on the graph, and see the change reflected in the rest of the graph automatically. We describe our implementation of this feature for IVP numerical solution plots in MathBrush.

First, we must be able to relate the user's touch point and the initial value point in the same coordinate system. Touch points are received in view coordinates, whereas the plotted points are represented in what is referred to as the user coordinate system, which is the coordinate system of the original data. When the plot is initially set up for viewing, the coordinates are translated and scaled so as to draw the graph within the view appropriately. This transformation can be captured and later used to map between the respective coordinate systems. When the user touches the initial point and drags it, the pan gesture handler first relates the touch point to the initial point to determine if it was touched. For consistency, we calculate the distance between the points in view coordinates and use an absolute threshold, rather than one that would change depending on the initial scaling. Once it has been determined that the user is dragging the initial condition, the point moves according to the user's motion until the touch is released. At that point, the final point is mapped into the user coordinate space, and the request for the numerical solution to the IVP is reissued with the new initial value. The new plot is then generated in the same manner as the original.

Of course, the new plot represents a solution to a different IVP than the original ink represents, and so the plot becomes out of sync with the original input. We do not see a reasonable remedy for this, but neither is there a real need for one. Presumably, the user will be aware that their changes are isolated to the plot alone.

Continuing with the previous first-order example, we may move the initial condition in the IVP solution as follows:


iOS Simulator - iPad / iOS 6.1 (10B141)
Carrier
3:41 AM
100%
Ink Input
Expressions
6 - Solve for y(t) numerically 1
Operations
1 - Original Expression
4 - Solve for y(t) 1
5 - Plot 4
6 - Solve for y(t) numerically 1


iOS Simulator - iPad / iOS 6.1 (10B141)
Carrier
3:49 AM
100%
Ink Input
Expressions
9 - Solve for y(t) numerically 1
Operations
1 - Original Expression
4 - Solve for y(t) 1
5 - Plot 4
9 - Solve for y(t) numerically 1

Note that when the initial condition is changed, a new request is submitted to Sage. The new solution is independent of the previous one, and therefore may have a different range of values, resulting in an adjustment of scale. Due to the extreme scale of the y-axis in this particular example, the initial point appears to lie on the x-axis, although in theory it does not. The plot is correct relative to its scale.

3.1.5 CAS operations

It is desirable to make the ODE-solving capabilities available to the user at the appropriate time, i.e. when a MathBrush user has written down a valid ODE or IVP. We therefore need to be able to detect when an ODE or IVP expression has been recognized. Of course, this is exactly what is provided by the ODE parser library, with the exception that we are not (yet) interested in the solution.

Thus, the ODE parser library serves two functions:

1. to determine if a given expression tree represents a valid ODE or IVP, and
2. to generate the Sage code representing the ODE or IVP, in order to solve it.

These two functions are inherently connected. Fulfilling the second function implicitly fulfills the first. Furthermore, while parsing an expression to determine whether or not it is a valid ODE, the information required to solve it is being gathered. However, these two functions need to be used separately. The MathBrush client needs the ODE parser library for its first function – to identify differential equations – in order to detect differential equation input. The computer algebra server needs the library for its second function.

The initial version of the client uses a simple mapping from the input root expression type to a set of operations to filter the list of operations. This mapping is specified in an XML document packaged with the application, thus confining the client's mathematical knowledge to a configuration file. However, this simple approach is insufficient when dealing with not only differential equations, but other scenarios as well[2]. Deeper analysis of the expression is needed at the client level.

To respond to the increasing complexity surrounding the interaction with the computer algebra server, the presentation of relevant mathematical operations, and the types of data

[2]For example, the "evaluate numerically" operation is relevant for numbers like π and e, but not for numbers like 1, 2, or 3.

returned by operations, we develop an organized approach to manage the activity pertaining to the computer algebra server (CAS) operations. We seek an approach which provides appropriate encapsulation and minimizes the level of understanding of the expression and operations required by the client. This was accomplished by creating a structure to contain the information about a CAS operation, aptly named `CASOperation`.

The CAS operation structure includes:

- an identifier,
- a name,
- the type of result produced by the operation (expression or plot), and
- the unknown, in the case of `solve` operations.

Using this structure, we develop a module to analyse an expression tree (utilizing the ODE parser) and return a set of relevant operations applicable to the expression. For example, if the input expression is an IVP in unknown $x(t)$, this would be confirmed by the ODE parser and the relevant operations would be identified as "solve for x(t)" and "solve for x(t) numerically". This module cleanly contains the work involved in determining the list of relevant operations for an expression. When an operation is selected by the user, the client then delegates to the computer algebra server, transmitting the given expression. Finally, the client knows what type of data to expect as a response because this is identified by the CAS operation structure. This modular approach gives the MathBrush client the ability to handle the multiple forms of expressions, operations, and results appropriately, while protecting its right to be agnostic towards the mathematics.

3.2 A dynamic and interactive ODE editor

The MathBrush approach to solving differential equations provides considerable freedom, allowing the user to simply write down by hand the problem they wish to solve. However, this advantage can also be seen as a drawback: the user has to write everything down by hand. The interface does not make it any easier for the user to provide the input. Complex expressions can take a considerable amount of time to write down, and further to become recognized in MathBrush. Having to correct errors in the recognition can be disruptive. MathBrush's recognizer is designed for arbitrary mathematical expressions. However, if the system is designed for differential equation input exclusively then perhaps it can make the input process a little easier for the user. We ask: is there anything that the interface can do to make working with differential equations easier, more efficient (perhaps even fun)?

In this section, we propose a new interface: an editor for differential equations that combines the convenience of free-form handwriting with helpful interface elements to support the input process, making it even easier than simply writing down the input.

3.2.1 An ODE builder

We develop the concept of a "builder" interface where the functions (the unknown and its derivatives) are treated as "building blocks" that are provided to the user, who supplies the coefficients via pen-based input. The user can use touch-based gestures to develop the expression by moving parts around.

Since we know the expression must involve a variable and it's derivatives up to a certain order, we can provide these items in a "toolbox" from which the user can drag items into the expression. The toolbox contains the dependent variable and its derivatives, reflecting the current user-defined settings, which are subsequently explained.

3.2.2 Modelling a differential equation

A differential equation is defined by its coefficients and the configuration of the functions. The basic parameters that determine what the differential equation will look like are:

1. The **order**, n, of the ODE.
2. The preferred **notation**: Lagrange or Leibniz. In the case of Lagrange, the parenthesized independent variable may or may not be present.
3. The **variable names**. The user can be provided with common choices[3] for variable names, or be allowed to select from a list of letters and symbols.

The default parameter settings would use $n = 2$, Lagrange notation, and variable y as a function of variable t. For simplicity, input will be restricted to equations involving terms of the form

$$c(t)Y$$

[3]We suggest $\{x, t\}$ for the independent variable, and $\{u, v, x, y, z\}$ for the dependent variable (avoiding conflicts with the variable x).

where $c(t)$ is an arbitrary expression not involving the unknown or its derivatives, and Y may be a single function or a product of functions from $\{y, y', ..., y^{(n)}\}$.

3.2.3 In-place recognition

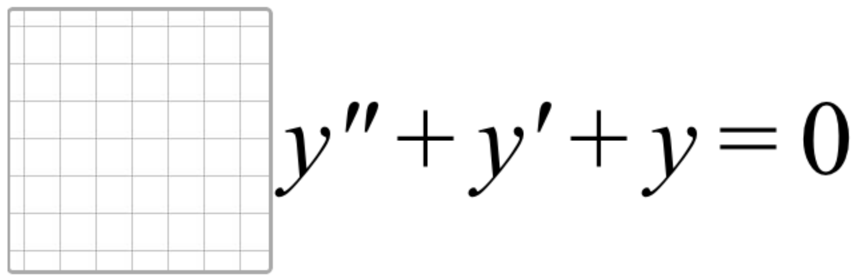
MathBrush provides two separate views for the ink and recognition. An alternative approach is to allow the recognition to occur *in-place*, allowing the ink and recognition to occupy the same space in the GUI at alternate times. This option allows for a more concise use of GUI real estate, and provides a clever way to manage the input and recognition together. The idea is that the user writes down an expression, and after a few seconds of inactivity the ink transitions into the recognized expression. This approach is used by several mathematics handwriting applications (mentioned in 1.2) and appears to work quite effectively.

Of course, there are tradeoffs to consider. Without separate, dedicated views for the ink and recognition, the transitions between the views must be coordinated thoughtfully. If the user always writes the final version of the expression at once, and if the recognizer always gets it right the first time, then there are no concerns. However, typically they will need to make corrections or add more ink to the expression. The interface must therefore provide a way for the user to switch back to the ink view. Otherwise, it must enable the user to combine ink with the rendered recognition, which is highly nontrivial. Toggling between the views can be handled using a button, or a gesture such as long-press. Recognition corrections can be handled similarly to how they are already handled in MathBrush, with a simple double-tap gesture which takes the user to the recognition correction screen. Alternatively, a simple tap gesture can be used to toggle through the recognition possibilities.

3.2.4 Manipulation

We envision an interface which allows the user to interact with the expression using touch, not only by adding handwritten ink, but by moving things around in a natural way. This interface enhances the ink canvas provided by MathBrush with the ability to easily manipulate parts of the expression, as one might wish to do with ink on paper. The user builds their differential equation by either starting with a default equation and modifying it to make it their own, or by starting with a blank slate and building their expression one piece at a time by dragging down terms from the toolbox and adding ink. The user creates space for writing ink by manipulating parts of the equation, with the guiding principle that such space should only exist where it makes sense to write an expression, such as adjacent to the left of a function in an ODE.

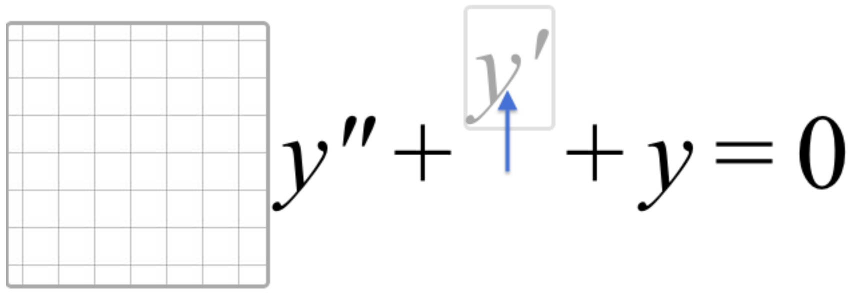
Here we explain the various elements of the interaction that we propose, using examples for clarity and illustration. In the examples, a grid background is used for the ink canvas, as in MathBrush, to make it clear where ink input is accepted. Assume we are starting with a default second-order ODE as a sum of the function and derivatives equal to zero.



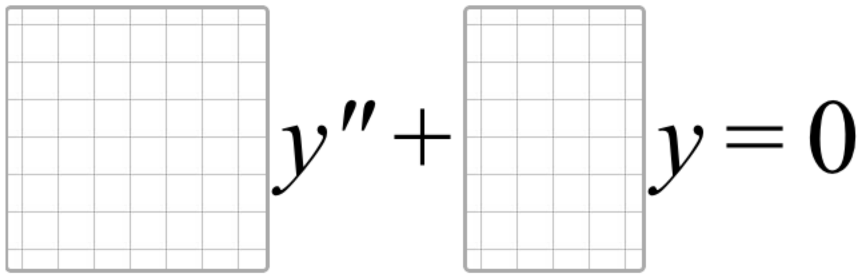
The zero in this equation would be a special placeholder, to be distinguished from an expression produced from the user's ink. This allows the ODE to be initialized to a complete ODE without forcing the user to manually provide a right-hand-side expression. The user may remove the zero to enter their own expression via touch input as described below. The reason the ink canvas already appears on the left is simply because that space may be used for the coefficient of the second derivative. There is no canvas for the right-hand side expression because the zero would need to be removed first, and space may be created for the coefficients of the other functions as described in the second subsection below.

Removing an item

A term or coefficient may be removed by simply "dragging it away". When touching the item and dragging it either upward or downward, out of the expression, the item fades out and disappears. The transparency of the item varies with the displacement from its original position, and it is not actually deleted until and unless it fades out completely and the user releases their touch. A coefficient expression may only be deleted when the recognition is displayed; otherwise, the touch would be interpreted as ink. To delete the first derivative from the sample ODE, the user touches it and drags it either up or down and out of the expression.

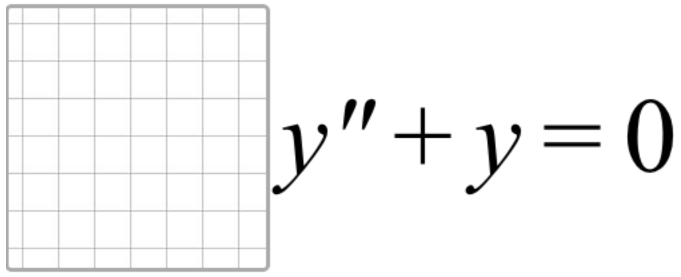


The extraneous plus sign automatically disappears with the derivative, and the absence of both leaves behind blank space for ink input.

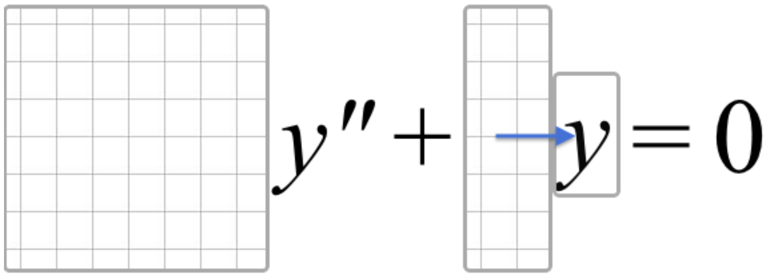


Entering a coefficient

In order to enter a coefficient for a term in the expression, the user needs to find blank space in front of the particular function. In the previous image, space has been made available through the removal of the first derivative. However, if there is no blank space, the user drags horizontally, moving the term to the right (or moving the addition operator to the left) to create space in front of it. When a term is dragged to the right, the contents to the right of it get pushed along with it, while the contents to the left stay put. For example, if we are starting with

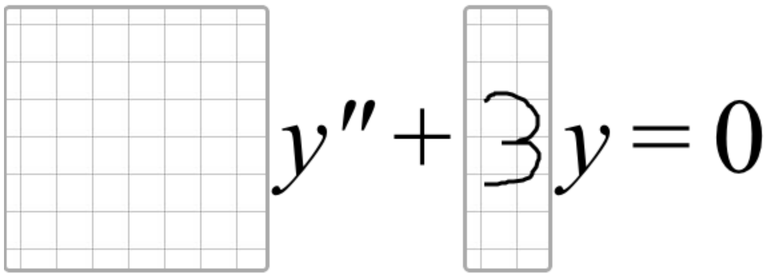


then, when y is dragged to the right, we have

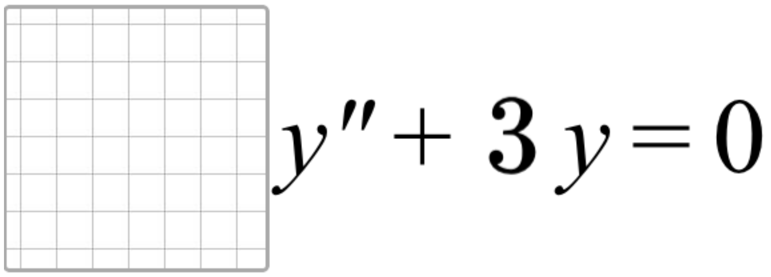


.

The user can treat the space as an ink canvas on which to write the expression. After writing the number 3, for example,



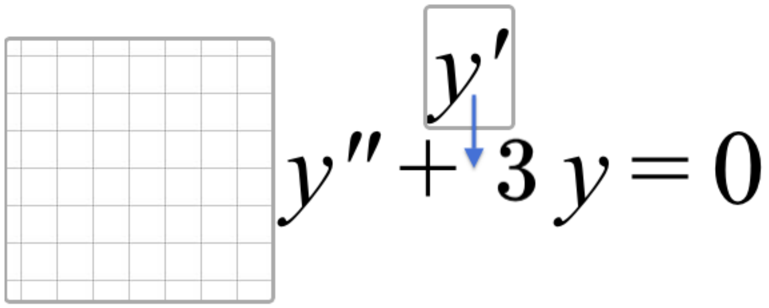
the ink would become recognized, yielding the expression



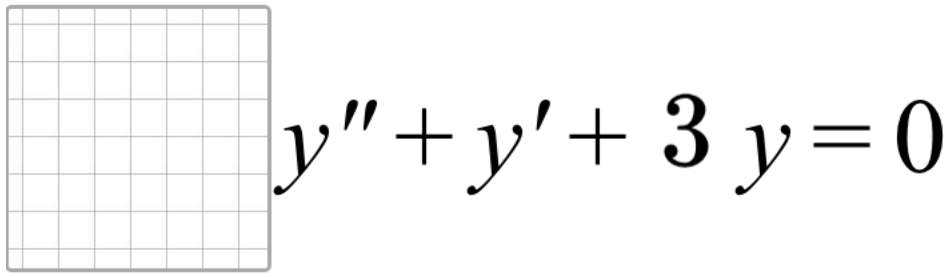
.

Inserting an item

A term may be inserted into the equation by simply dragging the function from the toolbox into the expression. Items already in the expression move to make room for the new item as it is dragged in, and once the user's touch is released, the action is complete. The user may cancel the action simply by not releasing the item within the area of the expression, in which case the equation remains unchanged. A plus sign will automatically be inserted as appropriate to separate the inserted item from neighbouring items (rather than assuming the user wishes to create nonlinear terms, which would be presumably atypical, and which can be accomplished as described further below). For example, to insert the first derivative back into the expression, the user drags y' down from the toolbox and into the expression between the plus sign and the y.

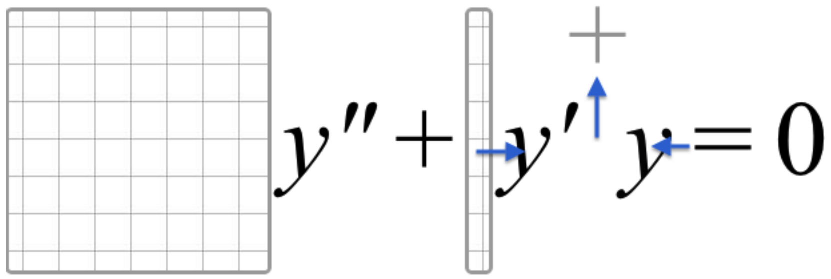


When the touch is released, a gap is created and the derivative is placed there along with a plus sign to separate it from the term $3y$.

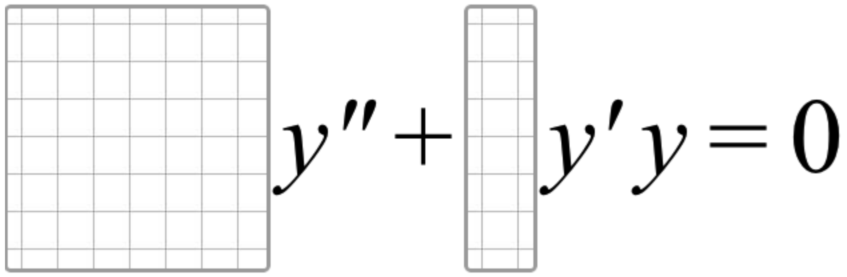


Nonlinear terms

The editor is biased toward linear ODEs, but in some cases the user may wish to create nonlinear terms. This can be accomplished via dual touch. By dragging two terms together simultaneously, the operator between them moves upward as it fades out and disappears. This action creates the affect of the operator being "squished" between the neighbouring terms, and thus ejected. For example, to combine the y' and y terms into one term, they would be touched simultaneously and dragged into one another

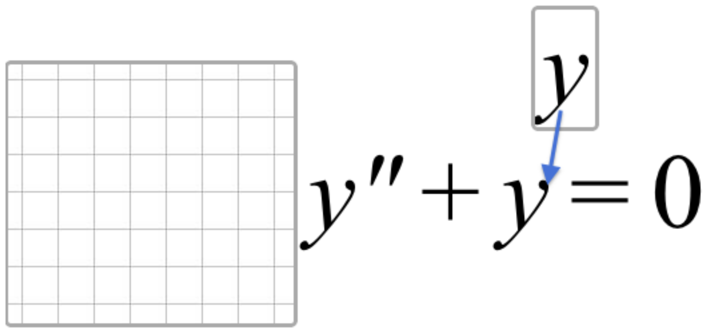


until the plus sign disappears, yielding

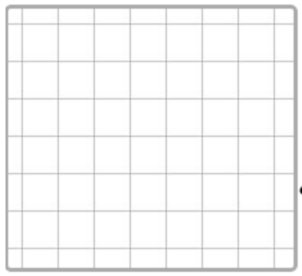


.

A function can be raised to an exponent by dragging another instance of it from the toolbox directly into the function. For example, to form the term y^2, the user would insert y (unless it is already present in the expression) and then insert it once again. This action is illustrated as follows:



.



$$y'' + y^2 = 0$$

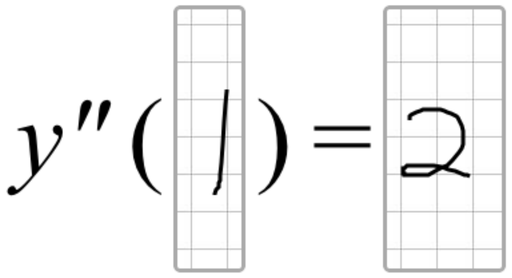
In this case, the user's intention to form a nonlinear term is clear. The system does not interpret the inserted item as a separate addend, as it would if a distinct item were inserted, since that would produce two like terms which should be collected using a single coefficient. Collecting like terms automatically is problematic because it may involve ink-based coefficient expressions. Inserting the item anywhere other than beside or onto the existing occurrence of the same item would therefore not be allowed.

Initial Conditions

Initial conditions may be edited in a similar fashion, with appropriate restrictions in place to ensure that they remain valid representations of initial conditions. Initial conditions will be generated based on the order parameter, and will be initialized to default expressions (all "zero at zero"). The user may remove values in the same way as previously described. For example, in the second-order case, the following initial conditions would be generated.

$$y'(0) = 0$$
$$y''(0) = 0$$

The user can change the second condition to $y'(1) = 2$, for example, by dragging out the zeros and writing the numbers in the available space.



The corresponding recognition would then display the initial condition as follows.

$$y''(\ 1) = \mathbf{2}$$

If required, the user could create space for larger input by panning one of the parentheses.

Remark

Note that the drag-and-drop actions we have described reflect common modern touch-based interface design. For example, app icons on an iOS device may be manipulated in a very similar to the behaviour we have prescribed for inserting and removing items in the expression. Another example is the manner in which a photo may be deleted when scrolling through a gallery on an Android device. We believe this interface demonstrates natural, modern design and therefore would be intuitive for many users.

Chapter 4

Differential equation editor prototype

The interface described in Section 3.2 reflects a very ambitious design with many moving parts and many possibilities to consider. A high level of detail has been described without mention of how it might be implemented. Indeed, implementing all of the features correctly is nontrivial and requires substantial time and effort. However, progress has been made on some of the key features, and this will be the subject of discussion for this chapter.

4.1 Model-View-Controller

First, we review the Model-View-Controller (MVC) software architectural pattern, which provides the basis for building iOS applications [11]. MVC divides the system into three components as follows.

- The **model** represents the abstraction of the entities involved the application, i.e. the "what". The application data and logic belongs in this part.
- The **controller** coordinates the activities of the application, i.e. the "how". The controller facilitates the interaction between the model and user.
- The **view** is a visual representation of information that the user sees. Views can be thought of as the controller's "minions", used to display information about the model to the user.

The interaction between these three components is defined as follows:

- The controller has direct access to both the model and the views. It typically sends commands to the model to change its state, and sends commands to the associated views to reflect those changes.
- The model is typically self-contained and agnostic toward the controller or the associated views. It provides public methods for changing its state and accessing its data.
- The views are considered “dumb” objects which are used for presentation purposes but do not own their data or implement logic, other than what pertains to the view itself.

MVC is foundational to the development of graphical user interfaces, and serves as a guideline for designing software constructs in terms of their responsibilities.

4.2 Object-oriented design

The individual pieces that comprise a differential equation expression need to be defined in terms of the GUI components. We define the following types of pieces:

- An **expression** is an area that accepts ink input for handwritten expressions. It is initially blank, at which point it may be thought of as drawable space, but blank space is collapsable to enable the dynamic interaction with other parts of the equation as described previously.
- A **function** is a representation of any of the individual derivative functions that may be involved in the expression. It’s role is simply to display the particular derivative, according to the notation parameter.
- An **operator** is a representation of the operator between terms, i.e. plus (+) or minus (–). These operators will need to move around according to the dynamics of the user interaction.

From an object-oriented design perspective, a differential equation may be represented by a doubly linked list of expressions, functions, and operator objects, as defined above. In order to define the behaviour and interaction between these objects, to realize the prescribed ODE editor, we begin with the idea that these pieces are like boxes that can be pushed around. When a piece is pushed to the right, for example, the pieces to the right of it get pushed along with it, while the pieces to the left of it stay put, and drawable space is created to fill the gap. The implementation of the prototype is based on this basic physical model. The movement of one piece, initiated by touch, creates a chain reaction affecting the other pieces.

We model these pieces as classes where each class implements its own logic governing how it responds to being moved and how it interacts with its neighbours based on both type and current state. This scenario lends itself to polymorphism: these types can be modelled under a single abstraction, while each defines its own behaviour individually. In particular, an expression object would contain logic to contract its blank (drawable) space when pushed by a neighbouring piece. This means the programmer can use these objects generically without needing to know the specific type of an object at compile-time, thus simplifying the implementation.

The following UML[1] diagram shows the essential object-oriented structure that models the pieces of the differential equation at the GUI level.

[1]Unified Modelling Language

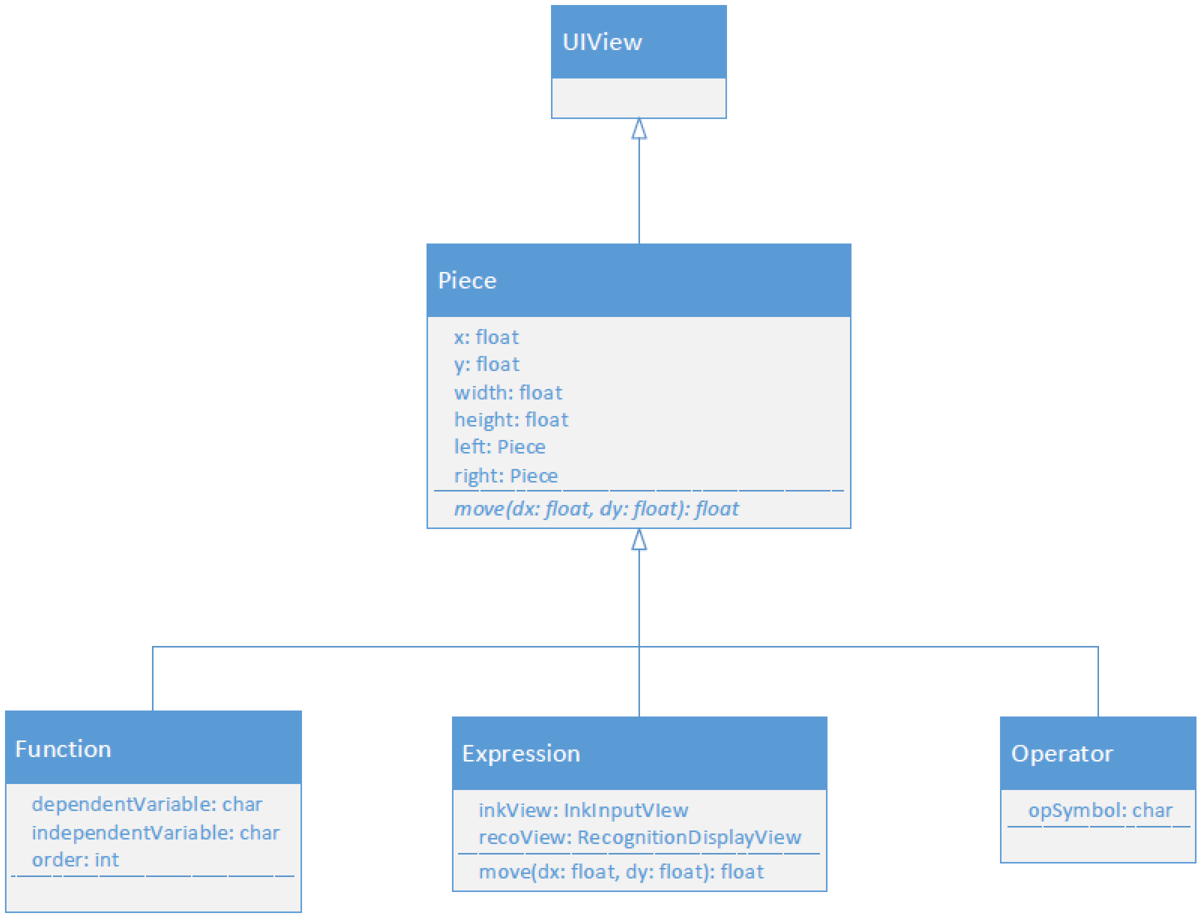


Figure 4.1: Object-oriented model for the interactive pieces of an ODE.

In light of the previous discussion on Model-View-Controller, one might argue that the logic governing the movement and interaction between pieces belongs in the model and controller, and not in the view hierarchy. Initial implementations used a separate model and generated the view configuration based on the model. This approach proved to be complicated and impractible because every interaction had to go through the model rather than letting views interact with one another directly. This is a particularly view-intensive application in which the model and view are essentially one. An alternative perspective is simply that this application does not lend itself to the MVC pattern, as the pattern cannot encapsulate all possible view-based interfaces.

4.3 In-place recognition

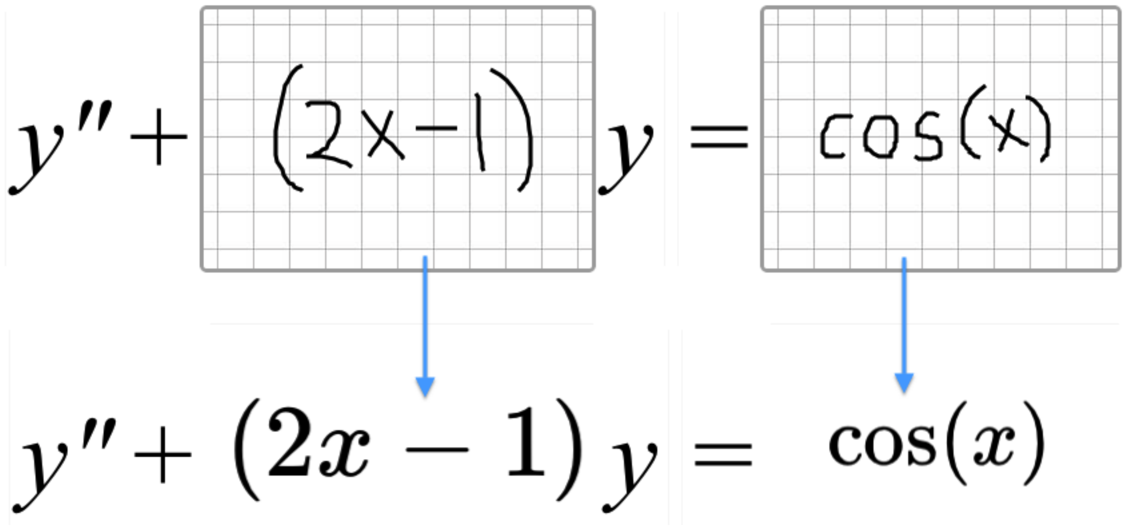
In our implementation, we use the “vertical flip” animated transition provided by iOS, triggered by the long-press gesture, to toggle between the ink view and the recognition view. This helps to make it clear to the user that there are two views, i.e. two “sides”, also hinting that they can be flipped back to the other side. Other transitions, such as “morphing”, might suggest that the transition was one-way. In fact, in the VisionObjects application mentioned previously, a morph transition is used, and there is no way to transition back to the original ink in their app.

Another important consideration is the timing of the automatic transitions between the two views. How much idle time do we allow before displaying the recognition? Should transitions be automatic at all? We want to avoid annoying the user by toggling the view before they are done writing, but it is impossible to know whether the user is finished, is thinking about what to write next, or is simply distracted. We suggest simply waiting for the recognition to complete before transitioning, which typically takes several seconds anyway. The user can then easily toggle back to the ink input if necessary.

Despite the complications introduced by in-place recognition, we believe that it fits well within the ODE “builder” interface. It not only saves space but it avoids redundancy, since the ink expressions are only parts of the overall expression. It provides a more unified interface, where the user has a sense of working on one thing, one piece at a time, without having to have to continually look back and forth between two places.

4.4 Example

The prototype has provided the basis for the images used to describe the interface in 3.2.4. We now provide an additional example of a second-order linear ODE which can be entered using the prototype. By utilizing MathBrush’s recognizer [6] we can support symbolically rich coefficients. Both the ink view and the recognition view are shown below.



Note that since Lagrange notation is ambiguous regarding the independent variable, the system infers the independent variable from the entered coefficients when possible. If the independent variable remains ambiguous, a default[2] variable is chosen.

[2]The default independent variable is x, unless x is already the dependent variable in which case t is selected.

Chapter 5

Conclusions and future work

The rise of the mobile machines presents rich new possibilities, and we ought to view the emerging technology as an important venue for interactive mathematics. This area of research has much room for exploration. Our link from MathBrush to Sage opens the door to pen-based computing on touch-based devices. The ability for a user to write down mathematical expressions and find answers at their literal fingertips represents a modern, fundamental tool for research and education. Our system is the only one known to us which seamlessly integrates correctible recognition of pen-based mathematics with computing, including support for differential equations.

However, this extended version of MathBrush has not yet received a proper round of testing and quality assurance. Because of the broad range of possibilities in terms of combinations of expression types and mathematical operations, QA can be especially important in order to ensure thorough coverage of the many possible scenarios. In particular, we need to ensure that all of the Sage expression object structures that may arise are supported by the Sage wrapper, and that potentially relevant operations are not erroneously hidden from the user. Also, more graceful, robust, and informative handling of server-side errors (as they affect the client) and connection issues would improve the general usability of the system.

Two significant aspects are missing from the current version of the ODE parser:

1. boundary conditions (as opposed to initial conditions), and
2. systems of first-order differential equations.

Extending the parser to correctly identify these types of expressions does not require a significant or unique effort, and is a worthwhile task as it would allow MathBrush to support many more forms of differential equations, such as predator prey systems.

As mentioned in Chapter 2.5, the plotting implementation is minimal, and more work is needed to bring the plotting feature up to par. The missing aspects of the implementation include appropriate labels and tick marks on the axes, and implementing gesture-based zooming and panning. In addition, the application would benefit from basic 3D lighting affects to make the depth information more apparent in 3D graphs.

Significant progress has been made towards a unique standalone pen-based differential equation solver, in both concept and (to a lesser extent) implementation. The dynamic ODE editor introduces a novel interface for working with differential equations which combines raw pen-based input with touch-based manipulation. This hybrid approach presents novel ideas for employing touch-based technology in mathematics. Our effort has been focused on prototyping the dynamic interaction between the parts of an equation, and the project as a whole is still in its infancy compared to the MathBrush iPad app. In order to complete the implementation envisioned in 3.2, the following steps would be essential.

1. The prototype is lacking the restrictions required to maintain the integrity of the expression as an ODE, and to ensure that blank space for ink input only appears where appropriate, regardless of the user's actions.

2. The interface would need to be expanded to receive the initial/boundary conditions as input, as described in 3.2.4.

3. An interface needs to be set up to allow the user to specify the parameters described in 3.2.2. These parameter choices would need to be dynamically reflected in the ODE expression.

4. The toolbox described in 3.2.1 needs to be implemented. The current prototype does not provide a way to drag items into the expression.

5. In order to utilize the computer algebra server, conversion to the expression tree format used by MathBrush is necessary. A module should be developed to take the various pieces of the user's expression including the ink input, as described in Chapter 4, and construct the appropriate SCG expression tree representation. The client could then talk to the Sage server in the same way that the MathBrush client does, to obtain solutions to the ODEs.

6. Further to the above point, a solution screen needs to be developed to present solutions to the user; MathBrush's manipulation interface may provide a basis for this.

The special interface for ODEs has some distinct advantages over the more general MathBrush interface:

1. If the user decides to change the variable names in the ODE, it is much easier to do so by changing the appropriate parameter settings than having to erase and rewrite the variables, restarting the recognition process.
2. The interface provides the benefits of in-place recognition described in 3.2.3.

Working with the coefficients as individual expressions, rather than the ODE as a single monolithic expression, has a few advantages:

1. There is less recognition work to be done since the task of recognizing the function and derivatives has been eliminated. Thus, recognition time will be reduced.
2. The recognition will be less error-prone due to the reduced complexity of each expression. This implies less interruptions for corrections.
3. Coefficient expressions are typically simple relative to the ODE as a whole. This makes in-place recognition more viable because when working with one simple expression at a time, it is not likely that much editing would be needed.

Of course, there are some potential disadvantages as well. A complex interface with so many moving parts can run the risk of overwhelming or agitating the user. Further, it can be difficult to implement correctly and robustly, which increases the risk of a negative user experience. After extended use, it may be unavoidable that the user will experience some tension between their intuition and what the interface is doing. In addition, in-place recognition could be frustrating to some users who do not appreciate having to switch back and forth between recognition and ink views. Some users may prefer to be able to see both their ink and the recognition simultaneously. Further, the automatic transition to the recognition view could be perceived as intrusive when unexpected.

Once the implementations of both interfaces are complete and have reached a high level of usability, it would be helpful to conduct a user study to evaluate their merits. This user study would help to illuminate the practical relevance of the anticipated advantages and disadvantages. Such a study might be structured to ask the user to carry out the same set of tasks using both interfaces. The user could then provide feedback and identify which interface was preferred, and why. This valuable information would provide a basis on which to proceed.

We have experimented with in-place recognition as an alternative to separate dedicated views. We have applied the approach in the dynamic ODE editor as well as in MathBrush, as described in A.2.2. Our experiences raise some questions. Is toggling between views a sufficient approach? If not, how can the ink input be related to the rendered expression to facilitate the insertion of ink strokes into the expression when rendered? These are challenges to be contemplated if we are to further pursue in-place recognition as a viable replacement for a separate recognition view.

Appendix A

Related work

A.1 Rotations in 3D

A standard feature of 3D plotting applications is the ability to rotate the plot. This feature is especially desirable on touch-based devices as it is almost instinctual for users to try to manipulate objects on the screen with the touch of their finger. But how exactly does touch input translate to rotations? A simple approach is to take the x- and y-components of the touch displacement and produce rotations around the perpendicular axes. While this approach works, it can sometimes be difficult to achieve the desired rotation as the resulting rotations can seem unnatural to the user.

A more sophisticated approach is called *arcball* rotation (or trackball rotation) which has an elegant implementation using quaternion mathematics. First, a brief review the mathematics of quaternions.

A.1.1 Quaternions

Quaternions are an extension of the complex numbers formulated by Irish mathematician William Rowan Hamilton in 1843. He understood that complex numbers could be viewed as points in a plane and could be added and multiplied together, and he was looking for a way to do the same for points in 3-space. He was stuck on the definition of multiplication of triples of numbers, until he discovered a way to multiply *quadruples.* By using three of the numbers in the quadruple as the point or vector, he had found what he was looking for. He introducing imaginary numbers j and k, related to i by the following formulas.

$$i^2 = j^2 = k^2 = ijk = -1$$

$$ij = -ji = k, \quad jk = -kj = i, \quad ki = -ik = j.$$

He then gave the form of a quaternion as $a + bi + cj + dk$ where a, b, c, d are real numbers. We can view these objects as $(a, \mathbf{v})$, where $\mathbf{v}$ is the 3-vector (b, c, d). Thus, Hamilton had developed a new number system with well-defined addition and multiplication—except that multiplication does not commute.

What quaternions have in common with rotations is the noncommutativity of multiplication; in a sequence of incremental rotations, order matters. It turns out that unit[1] quaternions can provide an elegant means of representing rotations. A rotation through an angle of θ around an axis defined by unit vector $\mathbf{u}$ is represented by a unit quaternion using an extension of Euler's formula:

$$q = e^{\frac{\theta}{2}\mathbf{u}} = \cos\left(\frac{\theta}{2}\right) + \mathbf{u}\sin\left(\frac{\theta}{2}\right), \quad \text{or} \quad q = \left(\cos\left(\frac{\theta}{2}\right),\ \sin\left(\frac{\theta}{2}\right)\mathbf{u}\right).$$

All unit quaternions can be written this way, and the set of unit quaternions is closed under multiplication Thus, quaternions representing individual rotations can be "piled up" into a single quaternion which effects the cumulative rotation. There is also a one-to-one correspondence between unit quaternions and rotation matrices[2] that can easily be computed and used to apply the rotations to a 3D model.

The appeal of this approach comes from the compactness and simplicity of the quaternion representation. It is clear what the rotation is and there is no redundancy in the representation. Also, the availability of libraries which do the required math makes it simple to implement, even if one doesn't understand how the math works. The result is simple and effective code implementing rotations.

A.1.2 Arcball rotations

Imagine a virtual sphere enclosing the object we are viewing (e.g. a three-dimensional surface plot). Imagine that when we drag that point, we are dragging the sphere, rotating

[1]The length of a quaternion $(a, \mathbf{v})$ is $\sqrt{a^2 + |\mathbf{v}|^2}$

[2]Given a unit quaternion $q = (w, x, y, z)$, the equivalent 3-by-3 rotation matrix is

$$\begin{pmatrix} 1 - 2y^2 - 2z^2 & 2xy - 2zw & 2xz + 2yw \\ 2xy + 2zw & 1 - 2x^2 - 2z^2 & 2yz - 2xw \\ 2xz - 2yw & 2yz + 2xw & -2x^2 - 2y^2 \end{pmatrix}$$

it around its centre, and the object rotates along with it. This is the idea behind arcball rotations (or trackball, as it has also been called). This approach to rotation was popularized by a 1992 paper by Ken Shoemake [10]. Here, we describe the general approach and how it relates to quaternions.

Touching the screen and dragging from one point to another defines two vectors on the surface of the virtual sphere. The cross product of the two vectors gives the perpendicular axis, $\mathbf{v}$, and the dot product can be used to obtain the angle between them, θ. We can assemble this information into a unit quaternion: $\left(\cos\left(\frac{\theta}{2}\right), \sin\left(\frac{\theta}{2}\right)\mathbf{v}\right)$. This gives a nice representation of the rotation defined by the user's action. Composing multiple rotations reduces to quaternion multiplication, and we can easily obtain the correct rotation matrix for the 3D engine to use to transform the model we are rotating.

In order to apply arcball rotations, we must map the point of touch onto the virtual sphere. First, consider the view from the perspective of the user looking at the screen and seeing an orthographic projection[3] of the sphere. Figure A.1 shows the silhouette circle of the sphere, the centre point of the circle ($\mathbf{C}$), and the user's touch point ($\mathbf{P}$). We can subtract the points to get a vector which we'll call $\mathbf{q} = (x, y)$.

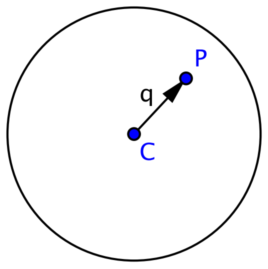


Figure A.1: User's perspective

Now, consider an aerial view of the situation (again, as an orthographic projection) where the virtual sphere is touching the planar screen of the device. The z-axis points down the page, the x-axis points to the right, and the y-axis is "coming out of the page." The user's eye would be looking up the page, along the z-axis (in the negative direction), from below the screen in the diagram. In this way we are placing the 2D vector $\mathbf{q}$, which lies in the planar screen, into the 3D scene. This view is depicted in Figure A.2.

[3]An orthographic project is a projection of a three-dimensional object onto two dimensions, such that all of the projection lines are orthogonal to the projection plane. Depth information is not discernible with this type of projection.

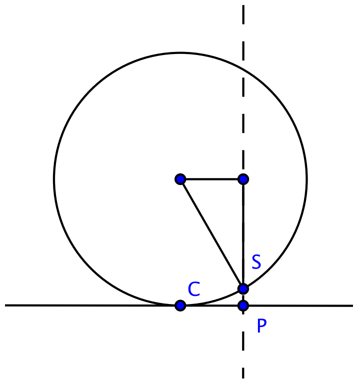
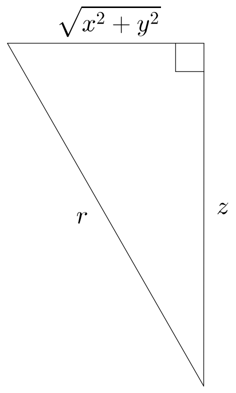


Figure A.2: Aerial view

Note that, in Figure A.2, $\mathbf{q}$ would be "coming out" of the page to the right on an angle. We want to find the point, $\mathbf{S}$, on the surface of the sphere, which represents the intersection of the projection line through $\mathbf{P}$ with the sphere. (We can regard $\mathbf{S}$ as the point on the sphere the user would have touched if it weren't for the limitations of the two-dimensional screen.) We already have the x- and y-coordinates of this point; we need to find its z-coordinate. To do so, consider the triangle in the previous figure; that is, the triangle defined by the centre of the sphere, the mapped point $\mathbf{S}$, and the point in the projection line through $\mathbf{P}$ sharing the z-coordinate of sphere's centre. This is the right triangle



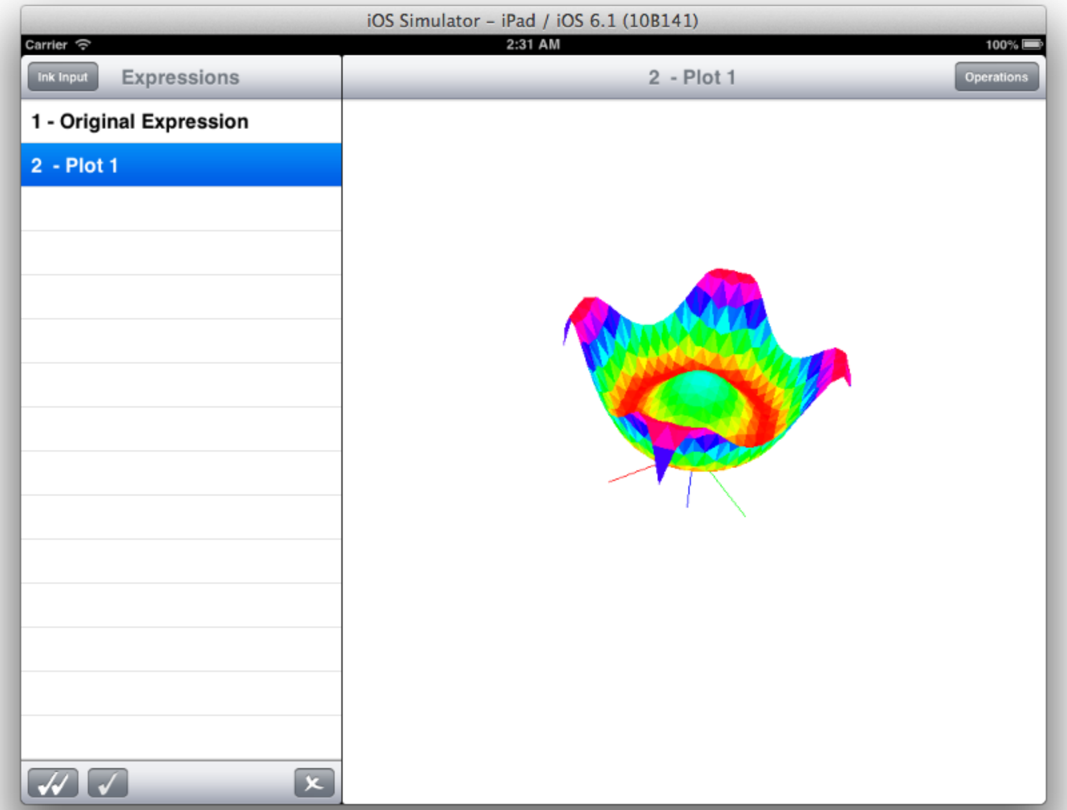
where r is the radius of the sphere and z is the value we seek. We can solve for z using Pythagoras' theorem, which together with x and y provides the mapped touch point (x, y, z) on the surface of the sphere.

We have assumed that the user has touched within the sphere (inside the silhouette circle). We can handle touches outside of the sphere by taking $z = 0$, which is the limiting case when considering the previous situation as the touch point is moved towards the edge of the circle.

In order to determine the correct axis of rotation, we apply the cross product to the two normalized vectors mapped from the initial and final points of the touch input. The cross product produces the vector perpendicular to both vectors, and this is the vector around which we wish to rotate. To find the angle of rotation, we use the dot product. For unit vectors $\mathbf{a}$ and $\mathbf{b}$, the angle between them is given by $\arccos(\mathbf{a} \cdot \mathbf{b})$. Once we have computed the axis and angle, we can store them in the form of a quaternion. This quaternion encapsulates the current rotation from the initial vector to the final vector.

The final step is to update the overall rotation with the current rotation. The overall rotation is what is applied to the model to achieve the appropriate current orientation. This is where we "pile up" the sequence of rotations, as quaternions, by multiplying them together. The result is a single quaternion which effects the cumulative rotation. The final quaternion can be converted to a rotation matrix using the appropriate formula, as mentioned previously.

Although it is difficult illustrate the behaviour of this user interface using still images (and beyond the scope of this thesis to indulge in further explanation) we include an example of a 3D plot which has been rotated, below.



This graph shows the result of the plot operation applied to the expression $1-\frac{1}{2}\sin((x^2+y^2)\pi)$ (interpreted as the surface $z = 1-\frac{1}{2}\sin((x^2+y^2)\pi)$), followed by an arcball rotation by the user.

A.2 Rendering with MathJax

While the existing MathBrush renderer for recognized expressions works well, and makes it possible to support dynamic recognition corrections, it doesn't compete with Latex, which has been developed over the past three decades and has become the de facto standard for typsetting mathematical expressions. Having recognized expressions in MathBrush rendered by LaTeXdirectly in the application brings that professional look to the application, and it allows the user to see exactly what they are getting when they export their expression to Latex or MathML from the app (one of the premium features offered by MathBrush). We decided to experiment with the possibility of rendering Latex directly within the an iOS application.

Latex does not run on iOS. However, there is an open-source JavaScript display engine for rendering both Latex and MathML in web browsers which has been adopted by many science and mathematical journal websites to render equations online. Since it is a JavaScript library, math expressions would need to be rendered as web content in order to use it in the application. We had been contacted by a company named Valletta Ventures who own a Latex typesetter application for iOS and the offered to provide an API to allow our expressions to be rendered with their typesetter. However, this would require that users buy their application, which creates a dependency on their app. Using MathJax made much more sense because it is open-source, can be used directly in our application, and it renders MathML in additional to Latex. We used the `UIWebView` class (provided by Apple's UIKit framework) to embed expressions rendered with MathJax as web content in the application. We now discuss the main technical challenge in doing so, which exemplifies how abstruse issues, seemingly removed the central problem being solved, can arise.

Our use of MathJax is not the conventional use for which it was designed. Rather than using MathJax to render some mathematical expressions included in some web content, we are exclusively interested in rendering mathematical expressions, and we are only using web content because that's how MathJax works. This presents a challenge to our intended usage. We desire the flexibility of being able to control the precise size and placement of the rendered expressions, especially when rendering expressions in-place, as discussed in Section 3.2.3. We want to display the expression in a rectangular window of minimal size which expands or contracts to the smallest size that is large enough to contain the expression. In order to accomplish this, the application needs size information from the rendered html elements, which can be retrieved via JavaScript's document object model (DOM). To further complicate matters however, MathJax is rendered asynchronously, which means the application would have to wait for the expression to be rendered before it can retrieve the correct size information. It turns out that MathJax rendering is relatively fast (even on an iPad) so the waiting period is very small. We considered two ways to make this happen.

One approach would be to fully load a new URL request each time the expression changes. The `UIWebViewDelegate` method `webViewDidFinishLoad` would provide the signal to the application that it can go ahead and retrieve the size. However, this approach is inefficient (even small incremental changes require the whole expression to be rendered) and it is visibly jarring (the expression does a little “dance” each time it changes, because the Latex source flashes on the screen momentarily before being rendered). Further, it is not clear that `webViewDidFinishLoad` would provide a reliable signal to the application, because it has been known to behave unexpectedly with respect to JavaScript.

A better approach is to use a single URL request and update the expression via JavaScript as it evolves. The challenge here is synchronizing the JavaScript and Objective-C code correctly so that the application can update the web view frame around the expression. One cannot simply ask the browser to block and wait until the rendering completes before returning the required information. And polling a JavaScript variable (e.g. a flag to report when finished) from outside the web page simply does not work. What is needed is a reliable signalling mechanism between the web page (JavaScript) and the application (Objective-C). The JavaScript code must notify the application when it completes. While there is no native method for doing this, there is a reliable workaround.

A.2.1 Signalling between JavaScript and Objective-C

The workaround[13] involves reading and parsing incoming URL requests, essentially mimicking a serialized messaging protocol. The following is a summary of the method.

In Javascript, create an iframe element with a source attribute such as

“`myapp:myaction:param1:param2`”

(i.e. a bogus URL). When this is added to the document element, it creates a request to load the (nonexistent) source and the `UIWebView` method

`webView:shouldStartLoadWithRequest:navigationType`

is invoked. As long as this method returns false the loading of the nonexistent page will be cancelled. If the iframe is removed from the document then it will be as if nothing happened—except that Objective-C code has been signalled from JavaScript code. The reason an iframe is used (as opposed to a JavaScript statement like

`window.location =` “`myapp:myaction:param1:param2`”

is to protect the JavaScript/HTML state of the original page which could otherwise become compromised in anticipation of losing the entire current page. Of course, the first time the page is loaded `webView:shouldStartLoadWithRequest:navigationType` needs to return true, but these cases can be distinguished by examining the request string that gets passed in. (Of course, a lot of information could be encoded via the request string. In this case, only one bit of information is needed).

This method works very well, and seemed to be the only successful approach to the problem of finding a bounding box for the rendered expression. This allows us to render mathematical expressions using MathJax in a flexible manner while being able to adapt to the changing dimensions of the rendered expression.

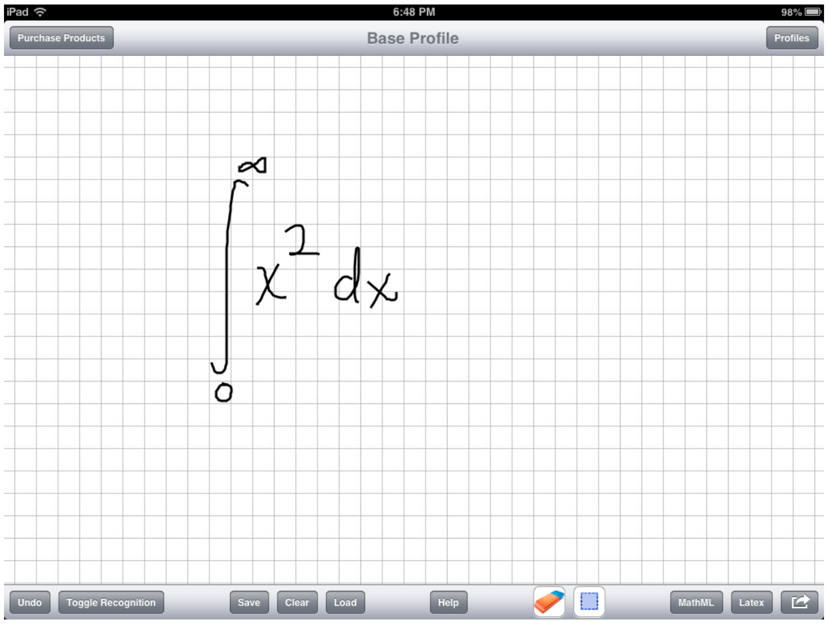
A.2.2 In-place recognition with MathBrush

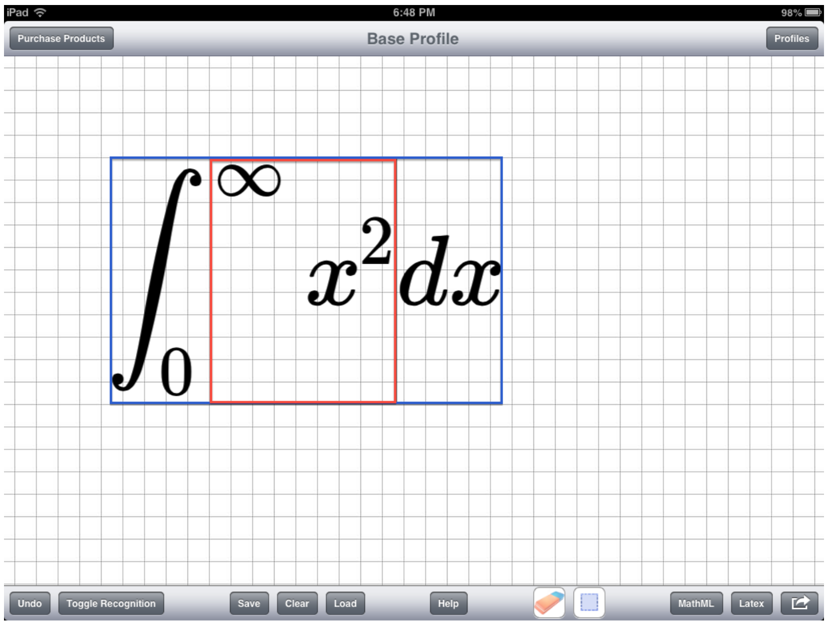
An experimental version of MathBrush was produced which replaces the hovering recognition window with in-place recognition using MathJax. In this version, the recognized expression is rendered to a web view using MathJax, and the view is superimposed above the ink, hiding it, as described in 3.2.3. Using the signalling technique described above, the main view controller can coordinate with the MathJax view to scale and translate it appropriately, so that the recognized expression is rendered approximately where the ink was written. Technically, this is accomplished by:

1. fitting the the bounding box of the rendered expression to the width or height of the bounding box of the ink expression, whichever makes the rendered expression larger, and

2. positioning the rendered expression such that its centre coincides with the centre of the ink expression.

A toggle button is used to allow the user to toggle between the ink input and the rendered expression, and ink can only be written when the ink is visible. For example, the following images show the views before and after the rendering takes place. For illustration purposes, the second image shows the the ink bounding box in red, and the rendered bounding box in blue.





The above example highlights the fact that the shape (more precisely, the aspect ratio of the bounding box) of the rendered expression may differ significantly from that of the ink (in this case, it is more than twice as wide). This is due to the natural contrast that arises between handwritten and typeset mathematics.

References

[1] "The Bus Pants Utilization." *The Big Bang Theory*, Season 4 Episode 12. CTV. 6 Jan 2011. Television.

[2] *SAGE: A Computer System for Algebra and Geometry Experimentation.* www.sagemath.org.

[3] W. Stein. *MathML*. Google Groups discussion (sage-edu). 11 Jan 2009. Accessed 1 Aug 2014. <https://groups.google.com/d/msg/sage-edu/iz8yEaQzRaU/TxfAuEoR9xAJ>

[4] B. Jacob. *We should drop MathML*. Google Groups discussion (mozilla.dev.platform). 5 May 2013. Accessed 1 Aug 2014. <https://groups.google.com/d/msg/mozilla.dev.platform/96dZw1jXTvM/hkNn65-Spf4J>

[5] G. Labahn, E. Lank, S. MacLean, M. Marzouk and D. Tausky, *MathBrush: A System for Doing Math on Pen-Based Devices.* Proc. of The Eighth IAPR Workshop on Document Analysis Systems DAS 2008.

[6] S. MacLean, G. Labahn, *A new approach for recognizing handwritten mathematics using relational grammars and fuzzy sets.* International Journal on Document Analysis and Recognition (IJDAR) 2012

[7] *Infty Project: Research Project on Mathematical Information Processing.* www.inftyproject.org.

[8] J.J. LaViola Jr and R.C. Zeleznik, *MathPad^2: A system for the creation and exploration of Mathematical sketches.* ACM Transactions on Graphics. Special Issue: Proceedings of 2004 SIGGRAPH 432-440 (2004)

[9] H. Mouchre, C. Viard-Gaudin, D.H. Kim, J.H. Kim, U. Garain, *Competition on Recognition of On-line Mathematical Expressions (CROHME 2012).* 2012 International Conference on Frontiers in Handwriting Recognition (ICFHR 2012)

[10] Shoemake, Ken. *ARCBALL: A User Interface for Specifying Three-Dimensional Orientation Using a Mouse.* 1992.

[11] G.E. Krasner, S.T. Pope, *A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System.* The Journal of Object Technology (1988).

[12] *The MyScript Equation recognizer from Vision Objects.* www.visionobjects.com.

[13] Poirot, Alexandre. *UIWebView Secrets - How to Properly Call Objective-C From JavaScript.* blog.techno-barje.fr. Techno Barje, 6 Oct 2010. Accessed 20 Aug 2013. <http://blog.techno-barje.fr/post/2010/10/06/UIWebView-secrets-part3-How-to-properly-call-ObjectiveC-from-Javascript>