MICA Workspace: A Symbolic Computational Environment for Signal Analysis

by

Adam J. Spanbauer

B.S. Physics, Massachusetts Institute of Technology (2011)
B.S. Mathematics, Massachusetts Institute of Technology (2011)

Submitted to the Department of Mechanical Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Mechanical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2013

Author .....
Department of Mechanical Engineering
May 10, 2013

Certified by .....
Ian W. Hunter
Hatsopoulos Professor of Mechanical Engineering
Thesis Supervisor

Accepted by .....
David E. Hardt
Ralph E. and Eloise F. Cross Professor of Mechanical Engineering
Chairman, Department Committee on Graduate Theses

MICA Workspace: A Symbolic Computational Environment for Signal Analysis

by

Adam J. Spanbauer

Submitted to the Department of Mechanical Engineering
on May 10, 2013, in partial fulfillment of the
requirements for the degree of
Master of Science in Mechanical Engineering

Abstract

With initiatives such as edX beginning to spread low-cost education through the world, a question has arisen. How can we provide hands-on experience at minimal cost? Virtual online experiments are not enough. To build science and engineering skill and intuition, students must set up and run experiments with their own hands.

Measurement, Instrumentation, Control, and Actuation (MICA), is a system under development at the MIT BioInstrumentation Lab. The MICA system consists of low-cost hardware and software. The hardware is a set of small, instrumentation-grade sensors and actuators which communicate wirelessly with a miniature computer. This computer hosts a web service which a user can connect to in order to control the MICA hardware and analyze the data. This web-based environment is called MICA Workspace.

MICA Workspace is a numeric and symbolic environment for signal analysis. The main component is a new symbolic mathematics engine. Some features of this engine include symbolic integration and differentiation, tensor manipulation with algorithms such as singular value decomposition, expression simplification, and optimal SI unit handling.

This thesis is intended to map the terrain surrounding the construction of MICA Workspace, the software part of MICA. I do not describe techniques in detail when clear and precise sources exist elsewhere. Instead, I describe the purposes and limitations of such techniques, and provide references to technical sources.

Thesis Supervisor: Ian W. Hunter
Title: Hatsopoulos Professor of Mechanical Engineering

Acknowledgments

First, I would like to thank the community of the MIT BioInstrumentation Lab. It is a special environment that inspires tremendous growth for all within, as scientists, as engineers, and as people.

Ian Hunter, my advisor, has created and maintained this special environment. He provides limitless ideas, funding, and freedom to explore.

Adam Wahab acted as a mentor and guide to me as I entered the realm of engineering.

Brian Hemond was the pioneer of the MICA project, putting years worth of work into it before I joined the lab, and has been an excellent source of advice.

Lynette Jones has been crucial in getting word out about MICA, especially with regards to papers and conferences.

Ramiro Piñón and Jaya Narain helped with MICA as undergraduates. As my first mentees, I may have learned as much from them as they have from me.

I would also like to thank those who have helped me to revise this thesis: Alison Cloutier, Ashin Modak, Sarah Bricault, Adam Wahab, and Cathy Hogan.

Finally, I want to recognize all life, from my family, friends, and colleagues, back through the aeons to the very first self-replicating systems. Their collective effort has formed this community of beings who explore the world, and has formed a world worth exploring.

Contents

Chapter 1

Overview

This thesis describes the design and construction of MICA's data analysis environment, MICA Workspace. This overview explains MICA Workspace's design, as well as the motivation behind it. First, I will describe the structure of the MICA system [1] [2]. Second, I will elaborate on the contents of this thesis. Third, I will describe MICA's hardware, to provide a context for MICA Workspace. Finally, I will describe existing symbolic systems, and why it is important for MICA to have its own.

1.1 Structure of the MICA System

As seen in Figure 1-1, a physical experiment, instrumented by MICA sensors and often actuated by MICA actuators, communicates via radio with a MICA radio USB device connected to a BeagleBone^{TM} miniature computer [3]. This miniature computer hosts MICA Workspace. The user can connect to MICA Workspace with any device that has a web browser.

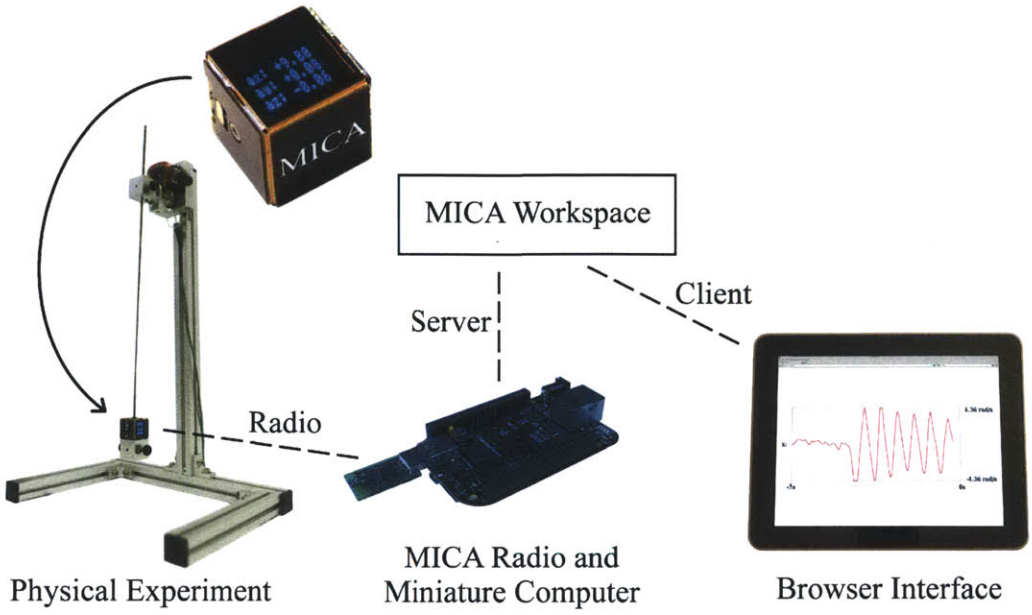

Figure 1-1: MICA system setup

MICA Workspace is a data analysis environment. It has two main parts: a symbolic engine and a computational mathematics engine. The flow of data through the symbolic engine, from user input to a simplified output, is shown in Figure 1-2. When the user enters an expression, the symbolic engine parses the expression, first into Polish notation, and then into an abstract syntax tree, which is a form easily manipulated by a computer. Then the symbolic engine runs a substitution step, where symbols, mathematical functions, and symbolic functions are substituted. Next, the expression is simplified by repeatedly applying simplification patterns. In order to perform certain operations, such as symbolic differentiation, the symbolic engine will apply a different set of patterns to get the result. Certain specialized operations cannot be handled by simple pattern application. In these situations, the symbolic engine calls upon the computational mathematics engine to solve the problem. The computational mathematics engine is capable of certain types of symbolic integration, as well as unit simplification. The computational mathematics engine represents these types of problems as problems in a formal mathematical framework, and uses algorithms from computational mathematics to do its work. Finally, the resulting

simplified abstract syntax tree is converted to LaTeX code so that it can be displayed to the user.

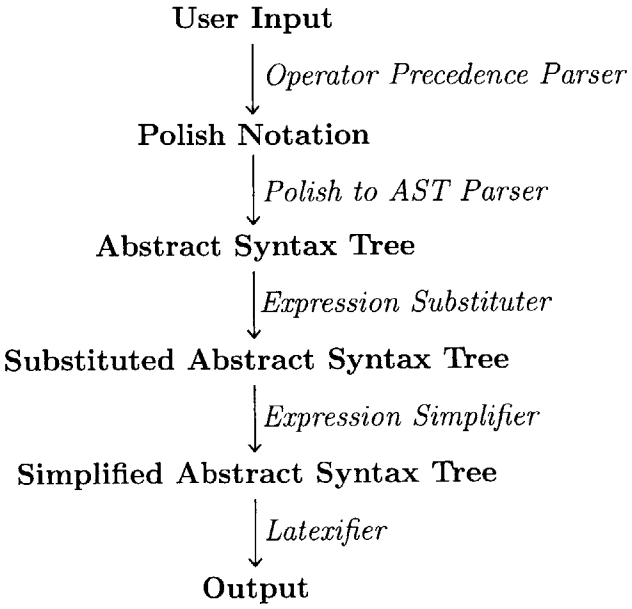

Figure 1-2: Flow of data through the symbolic engine.

1.2 Thesis Structure

This thesis introduces the topics related to MICA Workspace, building from concrete to abstract forms of data manipulation.

Chapter 2 describes strategies that I used throughout the entire system in order to maintain easily understood code.

Chapter 3 describes how to parse human-friendly infix expressions into computer-friendly abstract syntax trees. This manipulation is simply a map from one representation to another.

Chapter 4 handles the problem of unit simplification, describing a way to represent SI units in algebraic terms. It also introduces our plan to reduce SI units by a technique from computational mathematics, called reduction by a Gröbner basis.

Chapter 5 handles the software engineering problems related to describing mathematical objects in code, and describes the algorithms actually used to do the computational mathematics.

Chapter 6 covers the use of term-rewriting systems to manipulate expressions, starting with the simple problem of differentiation, and moving on to simplifying expressions, computing with tensors, and producing Latex code for an abstract syntax tree.

Chapter 7 covers some potential improvements and future plans.

1.3 MICA Hardware

MICA is a system for constructing and instrumenting experiments, and analyzing the resulting data. The goal is to make the system inexpensive enough that it can be used as an educational tool by students throughout the world. MICA sensors and actuators are cubes like the ones rendered in Figure 1-3, usually 25 mm on each side.

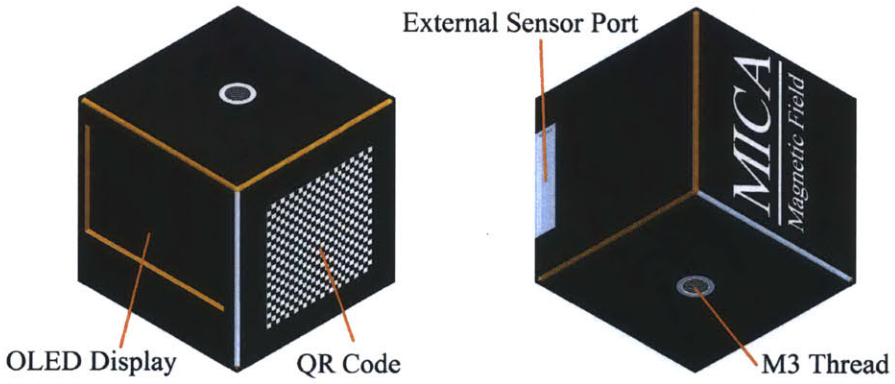

Figure 1-3: CAD rendering of the front and back of a MICA cube.

Each cube has an organic light-emitting diode (OLED) display, a piezo speaker, a 3-axis accelerometer, a 3-axis gyroscope, a lithium-polymer battery, and a radio. A MICA cube displaying a measured magnetic field on its OLED is shown in Figure 1-4. The radio can send about 5 kb/s to a MICA USB radio connected to a BeagleBone

miniature computer. This computer starts a web server over a virtual USB to ethernet interface. This allows a student to use the MICA system in their web browser without having to install any software on their computer.

Figure 1-4: MICA 3-axis magnetic field sensor, measuring the field of a magnet.

Each cube has a primary sensor or actuator. There are currently cubes for magnetic field, pressure (liquid or gas), IR temperature, and biopotential, as well as a laser actuator. The 72 MHz Cortex^{TM}-M3 microcontroller on each cube is able to package the sensor data for radio transfer, as well as perform data processing for presentation to the user, such as taking the standard deviation of a signal as shown in Figure 1-5.

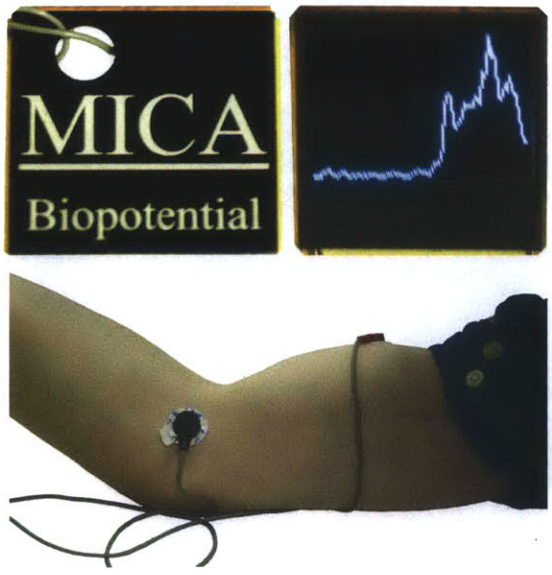

Figure 1-5: This MICA Biopotential cube is displaying the electromyogram (EMG), the standard deviation of a voltage signal, from a contracting biceps muscle.

Each cube is constructed by adhering square FR4 stiffener onto the back of a flexible printed circuit board. This is commonly called rigid-flex construction. The edges of the flexible printed circuit board are exposed copper, allowing the board to be folded into a cube and soldered together. An unfolded board can be seen in Figure 1-6.

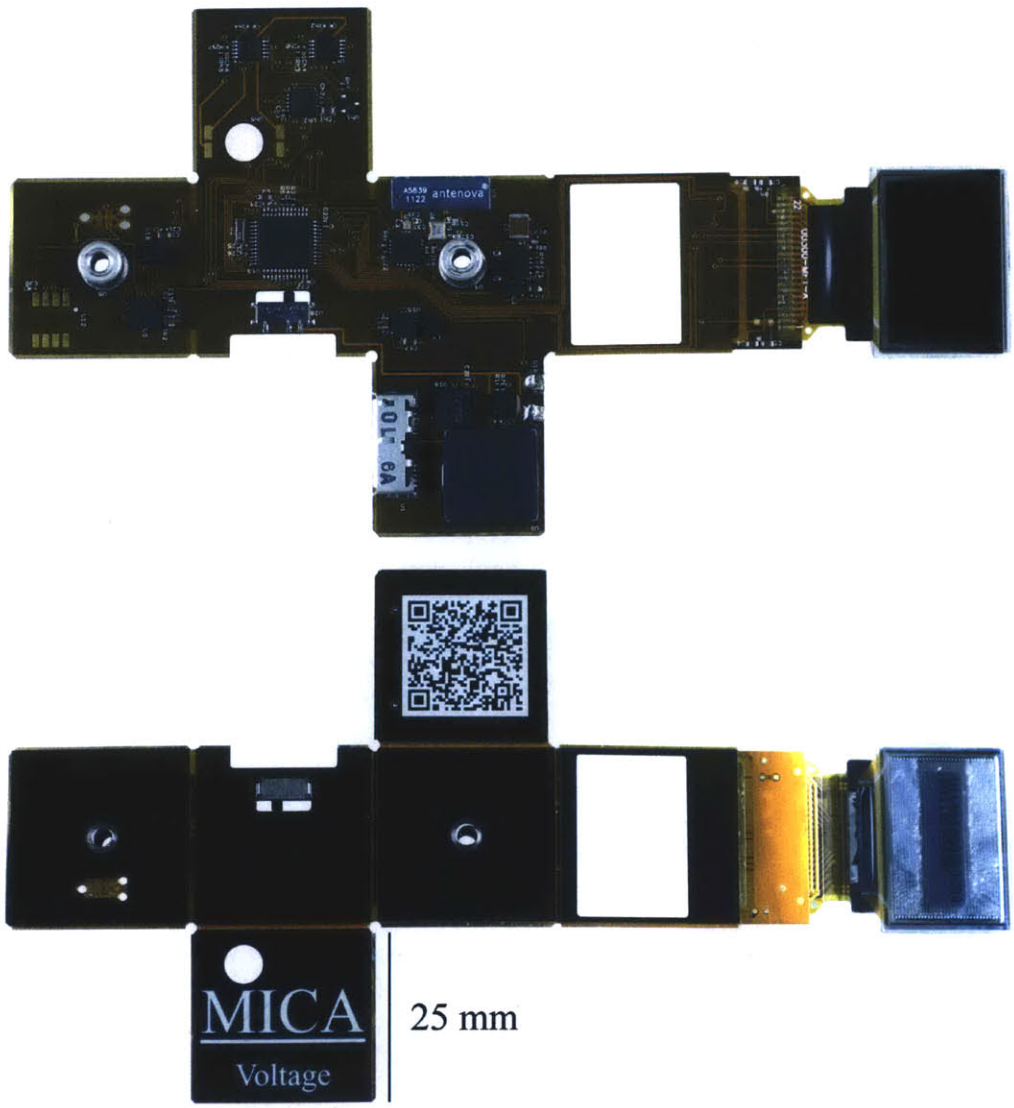

Figure 1-6: Inside and outside views of an unfolded MICA cube.

Each cube has an external sensor port. This is useful when a cube would be too large

or heavy, or when the environment is harsh (for example, measuring the temperature of a beaker of water). MICA currently has six external sensors similar to the one shown in Figure 1-7: accelerometer / gyroscope, gas flow, magnetic field, temperature probe, thermopile, and biopotential, as well as a general connector for development or for those who want to incorporate their own sensor.

Figure 1-7: MICA external thermopile for IR temperature measurement.

These sensors are meant to be quite general. Currently the selection of sensors and actuators is fairly limited, but the MICA platform allows for the rapid development of others. The goal is for MICA to eventually be able to sense and actuate every physical quantity with high accuracy. Care has been taken to make instrumentation-quality sensors. For example, all features of the electrocardiogram shown taken with the MICA Biopotential sensor are clearly visible in Figure 1-8.

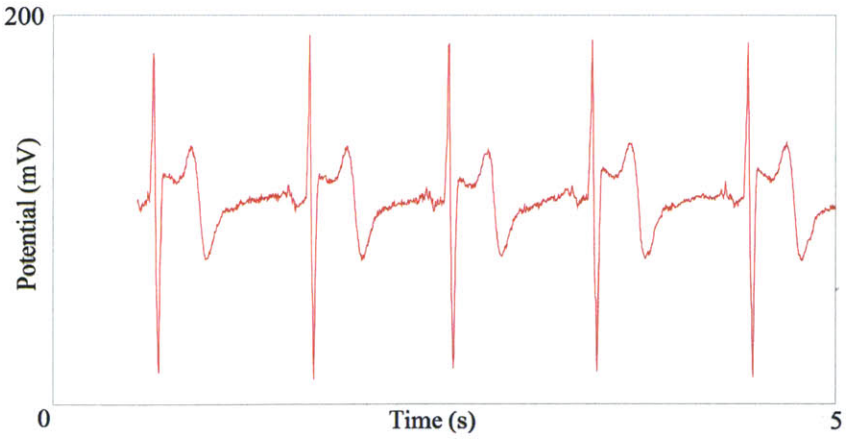

Figure 1-8: Electrocardiogram (ECG) taken with the MICA Biopotential sensor. It can also perform electromyographic (EMG) and electrooculographic (EOG) measurements.

The MICA sensors and actuators communicate with a MICA USB radio. The MICA USB radio relays these communications through a virtual serial port to the BeagleBone miniature computer. The miniature computer hosts two servers. One server is written in JavaScript, running in node.js, and directly handles and serves the data. The other is an iPython Notebook server, which acts as the web-based user interface and handles data analysis and symbolic computation [5].

This thesis focuses on the design of the data analysis environment presented to the user. The main part of this environment is a symbolic engine written in Python. This symbolic engine consists of two parts: a rewriting system that operates on abstract syntax trees representing mathematical functions, and a computational mathematics system for representing and manipulating certain algebraic objects.

1.4 Existing Symbolic Systems

There are several reasons why MICA needs its own symbolic engine. The primary reason is that as a data analysis platform rather than a mathematics platform, MICA needs a different set of features than existing symbolic engines. For example, all numbers in MICA Workspace have a unit associated with them, and as in Figure 1-9, MICA Workspace can plot incoming sensor data in real time.

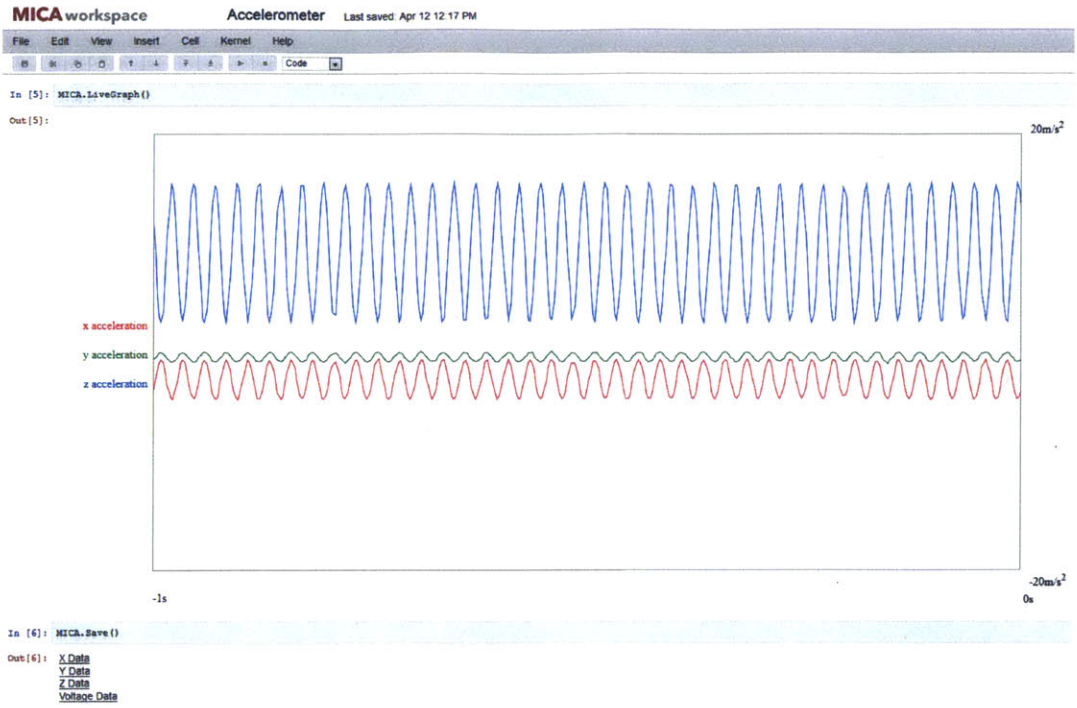

Figure 1-9: Real-time graph of sensor data as displayed by MICA Workspace

The second reason is cost. MICA's goal is to bring hands-on education to the world, so cost reduction is important. Substantially developed symbolic engines, such as Mathematica [6] and Maple [7], have student license costs comparable to the cost of a set of MICA hardware.

The final reason is ease of use. As a data analysis environment aimed at students, the ideal syntax for MICA Workspace is different than that of existing symbolic engines. Open symbolic engines such as SymPy [8] tend to have a steep learning curve. Further, MICA Workspace can potentially use features from open source software while retaining its own syntax aimed at students. For example, MICA Workspace uses NumPy [9] to compute the singular value decomposition of a matrix.

Chapter 2

Methods and Strategies

Many symbolic algorithms in MICA workspace are long and recursive. I imposed a set of restrictions on my coding style which make it easier to reason about these algorithms as a software engineer. First, maintaining a functional style restricts the flow of information into functions as arguments and out of functions as returned values. Second, using a carefully defined type system is essential to routing information effectively.

2.1 Functional Style and Referential Transparency

Effort was made to preserve referential transparency and maintain a functional style. In particular, every function that is a part of the symbolic system has the following properties:

> The output of a method depends only on the parameters, not, for example, on global state or user input.
>
> Methods do not mutate their parameters.

These properties simplify reasoning about a program by restricting the flow of information. The cost is that information has to be copied and moved around explicitly, potentially

making the program somewhat less efficient.

2.2 Types

Each piece of data processed by the symbolic engine is tagged with a type. Typically, a symbolic method takes an arbitrary piece of data and processes it in different ways depending on its type. Data is stored as a tuple, which is an immutable, ordered list whose first entry is the type, and subsequent entries depend on that type. There is a distinction between data that is part of an expression and data that is not. This is important, for example, with methods where a distinction is made between a function and a piece of data representing the need to evaluate a function with particular parameters. Pieces of data within an expression are called nodes.

Types:

- VALUES are a specific number along with a unit.
  ("VALUE", number, unit)
- VALUE NODES are nodes holding a VALUE.
  ("VALUE_NODE", number, unit)
- NAME NODES instruct the evaluator to look up a name in the current context. If the retrieved piece of data is a type that may take arguments, the arguments field is applied by extending the context.
  ("NAME_NODE", name, arguments)
- FUNCTIONS name a set of variables to be used as arguments in that function's expression.
  ("FUNCTION_NODE", param names, expression)
- SYMBOLICS are methods that may rely on the structure of the expression passed to it. Expressions passed to symbolic functions are passed directly without being evaluated first.

("SYMBOLIC", reference to python method)

- EXTERNALS take VALUES and produce another VALUE.

  ("EXTERNAL", reference to python method that processes the numbers, reference to python method that processes the units.)

- TENSORS are represented as nested tuples of expressions.

  ("TENSOR", rank of tensor, tuple containing the dimensions of the tensor, nested tuple of expressions)

- TENSOR NODES are TENSORS within an expression.

  ("TENSOR_NODE", rank of tensor, tuple containing the dimensions of the tensor, nested tuple of expressions)

A unit in the symbolic engine is stored as a tuple of integers, where each integer represents the power of a corresponding SI unit.

These representations of data tend to take a lot of space when written out completely, but I would like to give one example. The piece of data representing the expression x^2 is the following tuple:

```
("NAME_NODE", "^", (("NAME_NODE","x",()),("VALUE_NODE",2,
(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0))))
```

The list of 0s indicate that the 2 in x^2 has no unit.

Chapter 3

Parsing

Parsing is the process of converting a string in some language into a represention of the information that that string contains. MICA Workspace takes mathematical expressions that a user inputs and converts them into a form easily manipulated by a computer.

Expressions in MICA Workspace are designed to mimic the form a student might use when writing mathematics on paper. This is intended to make the language accessible to new learners. There are several infix operators, including addition, subtraction, multiplication, division, and exponentiation, which obey the usual order of operations. Function application is also represented in the usual way, $f(x, y)$. There is some overloading of symbols, for example the - symbol can be used as an infix operator or as a unary prefix operator to denote multiplication by -1. Parentheses are used for both function application and to change the usual order of operations. An example of a valid expression is:

$$-f(x, y) * (2 + 3) \hat{} 4$$

A parsing operation, as seen in Figure 3-1, is split into two steps. First, the expression is converted to Polish notation with an operator precedence parser. Polish notation is a way of writing expressions which makes the order of operations explicit. Polish notation represents expressions as a string of the form `(operator arg1 arg2 ...)`

where the arguments may themselves be expressions in Polish notation. Next, the Polish notation is parsed into an abstract syntax tree, which is a representation well-suited for computation [10]. In an abstract syntax tree, the arguments to an operator are the children of that operators's node.

$$-f(x,y) * (2+3) \hat{} 4$$

```
(* (- (f x y)) (^ (+ 2 3) 4))
```

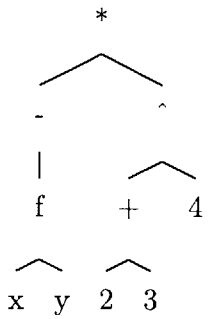

Figure 3-1: Parsing begins with an infix expression (top), which is first converted into Polish notation (middle), and then an abstract syntax tree (bottom).

3.1 Operator Precedence Parser

In the first step of parsing, the operator precedence parser takes an infix expression and produces an expression in Polish notation in linear time with the length of the expression. Roughly, the strategy is to break the infix expression into a set of tokens, and then combine the tokens into an expression in Polish notation, starting with the highet priority operators. A sketch of the algorithm follows:

to_polish(**infix_expression**):

Let **parts** be an empty list

Let **i**=0

While **i** < length of **infix_expression**:

if **infix_expression**[**i**] is the start of a number, find the end of the number. Append the number to **parts**.

if **infix_expression**[**i**] is the start of a name, find the end of the name. If the next character is not '(', append the name to **parts**. Otherwise run *to_polish* on each comma separated parameter of this function and append the string (name param1 param2 ...) to **parts**.

if **infix_expression**[**i**] is '(', run *to_polish* on the inside of the parentheses, and append the string (result) to **parts**.

if **infix_expression**[**i**] is '[', run *to_polish* on each comma separated expression within the brackets, and append the string [result1,result2,...] to **parts**.

if **infix_expression**[**i**] is an operator, append that operator to **parts**.

let **i** be the index into the first character that hasn't been looked at yet.

If **parts**[0] is '-', replace **parts**[0] and **parts**[1] collectively by (-u **parts**[1]) where -u represents unary minus.

For each operator **op** from highest to lowest priority:

For each index **i** into **parts** such that that **parts**[**i**]=**op**, replace **parts**[**i**-1],parts[**i**], and **parts**[**i**+1] collectively by a single entry (**parts**[**i**] **parts**[**i**-1] **parts**[**i**+1])

Return **parts**[0]

3.2 Polish to Abstract Syntax Tree Parser

In the second step, the parser takes a string in Polish notation and produces its abstract syntax tree. The reason this is done in a separate step is to isolate the operator precedence parser from the details of the data types.

parseExpression(**polish_expression**):

- if **polish_expression** is a string representing a floating point number, return ("VALUE_NODE", number, no unit)

  if **polish_expression**[0] is '(', let **tokens** be the list of space separated entries within the parenthesis. Then return ("NAME_NODE", **tokens**[0], tuple containing the result of applying *parseExpression* to each of the other **tokens**)

  if **polish_expression**[0] is '[', return a TENSOR_NODE with the same structure as the nested pattern of brackets, applying *parseExpression* to each of the inner expressions.

  otherwise, return ("NAME_NODE", **polish_expression**, ())

Chapter 4

Unit Simplification

Data analysis environments should have good unit handling. Units provide physical intuition and an easy way to check if a computation makes sense. All MICA Workspace VALUES carry an SI unit, and computations automatically keep track of units. For example, consider the expression

$$(5 * T) * (3 * A) * (0.1 * m),$$

where T is Tesla, A is Ampere, and m denotes meter. When this expression is evaluated, the numerical parts are multiplied together, and the exponents of each component unit are added together. This results in a single VALUE representing

$$1.5 * T * A * m.$$

However, the unit is not in its simplest possible form. We can write any unit in powers of kilogram, meter, second, and Amp:

$$1.5 * T * A * m = 1.5 * kg * m/s^2$$

To phrase this another way, all 17 standard SI units are expressible as monomials generated by kg, m, s, and A. MICA Workspace goes a step further, automatically

bringing units to their simplest form in SI.

$$1.5 * T * A * m = 1.5 * N$$

A good measure for the simplest form of a unit is the monomial representation with the lowest degree. To perform this simplification, we will find a way of expressing units as an algebraic object and use a concept from computational mathematics called a Gröbner basis [11]. Roughly, a Gröbner basis helps us take a list of identities between polynomials, such as

$$N = kg * m/s^2$$

and turn them into a fast reduction process to a specified normal form. In this chapter, we will explore the mathematics of Gröbner basis. In chapter 5, we will construct a Gröbner basis that allows us to quickly reduce units.

4.1 Mathematical Representation of the SI Unit System

The goal of this section is to give an intuitive feel for the mathematical objects involved in reduction by Gröbner bases. It is not my intent to to develop the theory completely, but rather provide enough information to enable the reader to recognize situations where computing a Gröbner basis might be useful. Artin's *Algebra* [12] develops all of the tools necessary to understand Gröbner bases, and there are many sources which develop Gröbner bases specifically [13] [14] [15].

In order to understand reduction by a Gröbner basis, we need some definitions from abstract algebra. I will keep this as brief and concrete as possible.

A **monoid** is a set along with an associative operation $\cdot$ that takes two elements of

that set and produces another one. A monoid also has an identity I, satisfying

$$I \cdot x = x$$

for any x in the set. An example of a monoid is the set of integers along with multiplication: Take any two integers, and you can produce a third by multiplication. Its identity is 1.

A **group** is a type of monoid. A group also has an inverse satisfying

$$x^{-1} \cdot x = I$$

The set of integers with addition form a group, but the set of integers with multiplication do not form a group, since 2 does not have an integer multiplicative inverse.

A **ring** is a set, along with two operations:

- An operation called addition which makes the set into a commutative group (that is, $x + y = y + x$)
- An operation called multiplication which makes the set into a monoid. Furthermore, addition and multiplication satisfy the distributive rule

$$x * (y + z) = x * y + x * z$$

The set of integers with addition and multiplication are a ring, called **Z**.

You can take any ring **R** and produce new rings called polynomial rings. The polynomial ring in variables x,y is called **R**[x,y]. **R**[x,y] is the set of polynomials in x and y, with coefficients in the original ring **R**, and with the usual rules for polynomial addition and multiplication.

Let us stop briefly to build a representation of units. As a first pass, numbers with

units can be described as the monomials in the polynomial ring

$$\mathbf{Z}[kg, m, s, A]$$

However, there are a few features that are missing: First, polynomial rings do not have inverse powers. We can add inverse powers by making our ring bigger:

$$\mathbf{Z}[kg, kg^{-1}, m, m^{-1}, s, s^{-1}, A, A^{-1}]$$

Unfortunately, in this ring, 1 and $kg * kg^{-1}$ are different elements. Also, we want to add more symbols, such as $N = kg * m * s^{-2}$. We keep extending our ring, adding units and inverse units until we have an enormous, 34 variable polynomial ring.

$$\mathbf{Z}[kg, kg^{-1}, m, m^{-1}, s, s^{-1}, A, A^{-1}, N, N^{-1}, ...]$$

Now we can express every unit. However, we still want to equate some elements of this giant polynomial ring, such as $kg * kg^{-1} = 1$ and $N = kg * m * s^{-2}$.

The way to equate certain elements of a ring is a process called taking the quotient of a ring by an ideal. To do this, we will need a few more definitions.

An **ideal I** is a subset of a ring **R**, satisfying two properties:

- The sum of two elements of **I** is also in **I**.
- The product of an element of **I** and any element of **R** is in **I**.

As an example, the even numbers are an ideal of **Z**.

This next example of an ideal is used in our algebraic representation of units. A set of polynomials $(P_1...P_N)$ within a polynomial ring $\mathbf{R}[X_1...X_M]$ generate an ideal. This ideal is the set of all linear combinations of $(P_1...P_N)$ with coefficients in $\mathbf{R}[X_1...X_M]$. Notice that this is the smallest possible ideal containing $(P_1...P_N)$. Thus all ideals of a polynomial ring are generated by some set of polynomials.

The **quotient** of a ring **R** by an ideal **I** is a new ring called **R/I**. It is defined by an

equivalence relation $\sim$ on the elements of $\mathbf{R}$. We say that $a \sim b$ if $a - b$ is in $\mathbf{I}$. This equivalence relation splits the elements of $\mathbf{R}$ into equivalence classes, which are the largest subsets of $\mathbf{R}$ where all elements are equivalent by $\sim$. These equivalence classes are the elements of $\mathbf{R}/\mathbf{I}$. The equivalence class of x is written as $< x >$. For example, $\mathbf{Z}/$(the even numbers) has two elements, $< 0 >$ which is the set of even numbers, and $< 1 >$ which is the set of odd numbers. You can check that $\mathbf{Z}/$(the even numbers) is a ring as well. This quotient ring encodes the well known facts about even and odd numbers: even+even is even, even+odd is odd, odd+odd is even, and similarly for multiplication.

For our representation of units, we first build an ideal that encodes all of our identities. We represent all of our identities as polynomials that are equal to 0, for example

$$kg * kg^{-1} - 1$$

$$N - kg * m * s^{-2}$$

...

Then we construct the ideal generated by these polynomials. We will call this our SI ideal.

Now we have a way to represent the SI unit system: A value with a unit is a monomial in the quotient ring

$$\mathbf{Z}[kg, kg^{-1}, m, m^{-1}, s, s^{-1}, A, A^{-1}, N, N^{-1}, ...] \ / \ \text{(SI ideal)}$$

Now that we have an algebraic description of the SI unit system, we will find a Gröbner basis to help us reduce them.

4.2 Gröbner Bases

In this thesis, the term 'Gröbner Basis' is used to refer to a 'reduced Gröbner basis'. The reduced Gröbner basis of an ideal is the simplest of any Gröbner basis for that ideal. It always exists and is unique. Certain computations require that the reduced Gröbner basis is used. The reduced form is harder to compute than an arbitrary Gröbner basis, but it is generally worthwhile, especially in cases such as this unit system where calculating the Gröbner basis is a one-time computation.

A Gröbner basis can be used to get from an element of a polynomial ring to the simplest element of it's equivalence class by some particular ideal. Gröbner bases can take a long time to compute. Computation of the Gröbner basis for the SI unit system took my (somewhat unoptimized) program about one day on one 2.9 GHz core. However, once the Gröbner basis has been computed, the reduction of polynomials is fast. The computation of a Gröbner basis is a very general manipulation of information. It is a generalization of several other algorithms. For example, the Gröbner basis of a set of elements of $\mathbf{Z}$ is their GCD. Also, Gaussian elimination and the computation of a Gröbner basis reduce a set of linear expressions in the same way.

The Gröbner basis $\mathbf{B}$ of an ideal $\mathbf{I}$ is a special set of polynomials that generate $\mathbf{I}$. $\mathbf{B}$ is uniquely determined by the property that $\mathbf{B}$ reduces any element of $\mathbf{I}$ to 0. In order to reduce a polynomial P with respect to $\mathbf{B}$, one divides P by the polynomials in $\mathbf{B}$, keeping only the remainder. As motivation, this is the equivalent of reducing a number in modular arithmetic by dividing and keeping the remainder. In fact, the structure of arithmetic modulo n is the ring $\mathbf{Z}$/(ideal of multiples of n). The Gröbner basis of the ideal (multiples of n) is just the number n, so the usual rule to reduce a number modulo n can be viewed as reduction by a Gröbner basis.

Polynomial division depends on the way the monomials are ordered (called a monomial order). Not all orders are consistent. Two common, consistent orders are lexicographic order and graded lexicographic order. The first step in both cases is to order the

variables, *e.g.* $x > y > z$. In lexicographic order, the power of the largest variable is compared, for example $x^3y^4 < x^4$. If the power of the largest variable is the same, the next largest is compared. In graded lexicographic order, the first step is to compare the degrees of the monomials. If the degrees are the same, then lexicographic order is used. In graded lexicographic order, $x^3y^4 > x^4$.

Both the Gröbner basis $\mathbf{B}$, and the result of reducing a polynomial by $\mathbf{B}$, depend on the monomial order that is chosen. In particular, the leading term of a polynomial reduced by $\mathbf{B}$ is the smallest of any polynomial in its equivalence class. This property guarantees that reduction by the SI Gröbner basis will reduce a unit to the simplest possible form.

We now have a clear goal: In preparation for reducing SI units, we must compute the Gröbner basis of our SI ideal, choosing the graded lexicographic monomial order. Once we have computed it, we can reduce a unit by dividing by this Gröbner basis.

The computational mathematics subsystem of MICA Workspace, which contains algorithms to compute the Gröbner basis, is described in the next section.

Chapter 5

Computing Gröbner Bases

5.1 Representing Algebraic Objects

Mathematical objects are not easily expressed in the usual object-oriented structure that common programming languages use. In the future, I plan to write a language representing mathematical information properly. However, for basic computation with simple algebraic structures, an object-oriented structure suffices. In order to compute the Gröbner basis for SI, I built a set of python classes, shown in Figure 5-1, representing the objects of abstract algebra described in section 4.1. Based on past experience in organizing mathematical objects in code, this structure is fairly efficient in terms of code reuse and readability. For example, I was able to abstract away the concept of a set with elements. I did this by dynamically generating a class representing the structure of an object's elements for each object. The organization I chose does have some drawbacks. For example, I have defined additive and multiplicative groups separately since fields use both. Further, I did not strictly adhere to a functional style due to efficiency concerns. Here is the class structure.

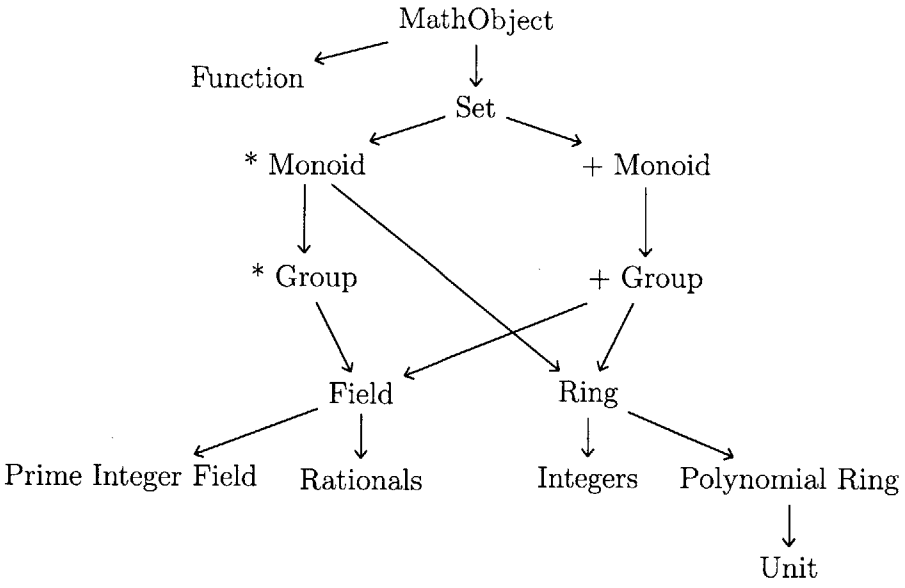

Figure 5-1: This directed graph displays the class structure for mathematical objects. A class below another class is a subclass, which represents a mathematical object containing the one above it.

Each algebraic object is made up of simpler ones, as well as additional features and constraints. Below are the primary features and constraints added to each mathematical object, in addition to the behavior they inherit as a subclass.

Functions have a domain, a range, an implementation, and automatically memoize their computations.

Sets dynamically generate a unique class representing the form of its elements.

Monoids (+ and *) define a single binary operation, + or *, on their elements. They also generate an identity element with respect to that operation.

Groups (+ and *) define a unary inverse operation that acts on their elements.

Fields and Rings do not enforce the distributive law. Specific implementations of operations are expected to satisfy it.

Prime integer fields, rationals, and integers provide specific implementations of

their operations, as well as ways to generate and display elements.

Polynomial rings take a specific field and set of variables, and provide an implementation of polynomial arithmetic. A polynomial in n variables is represented as a dictionary mapping n-tuples of integers to an element of its field. Each tuple represents the powers of the variable. The field element it maps to is the coefficient of that monomial.

To go with these representations of mathematical objects, I wrote algorithms, building up to Buchberger's algorithm for computing Gröbner bases and Czichowski's algorithm for computing integrals of rational functions. Having established a good representation of these algebraic structures, it was not difficult to follow resources for computational mathematics that phrase the algorithms in terms of the mathematical objects, as is done for Czichowski's algorithm in [16]. These standard algorithms include polynomial division by multiple polynomials in multiple variables, partial fraction decomposition, and Hermite reduction. These algorithms are standard building blocks in computational mathematics. They are typically computationally optimal at what they do, and have been studied and explained clearly and thoroughly elsewhere, *e.g.* [16].

5.2 Buchberger's Algorithm

Buchberger's algorithm is an algorithm for computing Gröbner bases. While faster algorithms exist, for example the Faugére F4 and F5 algorithms, Buchberger's algorithm is simpler and fast enough for our purposes. Some clear resources exist [13] [14].

As mentioned, computing the Gröbner basis for SI under graded lexicographic order took about one day. The basis consists of 675 polynomials. Note that using graded reverse lexicographic order is typically less computationally expensive than using graded lexicographic order, and would have worked equally well. Alternatively, an open source piece of software, Macaulay2 [17], can compute Gröbner bases.

As described in section 4.2, to reduce a unit, we divide by this basis. Figure 5-2

demonstrates the unit reduction system in MICA Workspace.

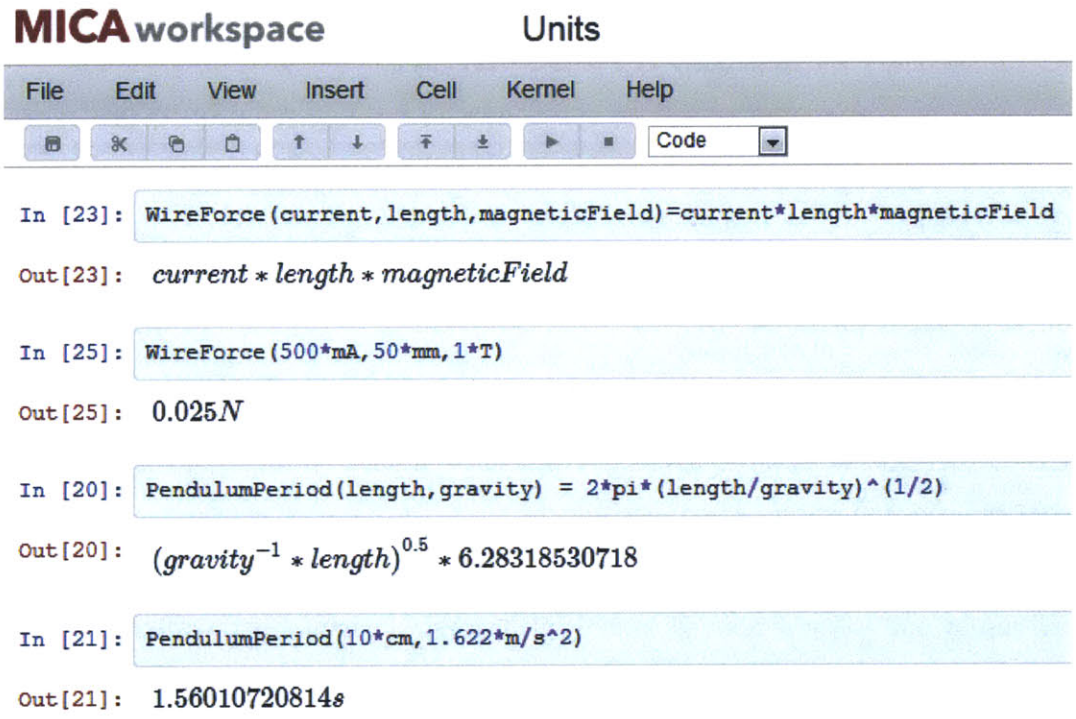

Figure 5-2: This is a demonstration of unit reduction in MICA Workspace. First, we calculate the force on a length of current-carrying wire by a magnetic field. Second, we find the period of a 10cm pendulum on the moon.

For other applications of Gröbner bases and other examples of computations on mathematical objects, see [18].

5.3 Czichowski's Algorithm

Czichowski's Algorithm relates the integral of a rational function to a Gröbner basis calculation regarding the polynomials which make up the rational function. Czichowski's algorithm gives the integral of rational function in x as a sum of rational functions in x and logs of polynomials in α and x where α is an algebraic number, such as the integral in Figure 5-3.

$$\int \frac{x^7 - 24x^4 - 4x^2 + 8x - 8}{x^8 + 6x^6 + 12x^4 + 8x^2} dx = \frac{1}{x} + \frac{6x}{x^4 + 4x^2 + 4} + \frac{-x + 3}{x^2 + 2} + log(x) + C$$

Figure 5-3: Integral computed by the implementation of Czichowski's algorithm within the MICA system.

The integrals given by Czichowski's algorithm are not always continuous, due to issues related to branch cuts in complex analysis. It would be useful to extend the implementation of Czichowski's algorthm in MICA Workspace to include a process called Rioboo conversion, which would give a continuous integral in terms of arc tangents [16]. There is a general symbolic integration algorithm, which either produces an integral or proves that an integral in terms of basic functions does not exist. While general symbolic integration is certainly useful, it is rather complex to implement. Further, being able to integrate rational functions allows us to integrate some very interesting things, such as transfer functions and rational Chebyshev approximations [19].

Chapter 6

Expression Simplification

When a user enters an expression, MICA Workspace is expected to perform some manipulation, usually evaluation or simplification. Other operations include taking derivatives, fitting data to a function, or converting an expression into LaTeX for presentation to the user. In general, simplifying and otherwise handling expressions can be done by examining the expression for patterns that represent a reduction or manipulation, and repeatedly applying them until termination.

This general strategy of applying reduction patterns to simplify expressions, including units, mathematical expressions, and more general abstract syntax trees, is studied in the field of rewriting systems. To build a system to simplify expressions, we usually need two pieces of information:

> A set of equalities (such as the relations between SI units or the distributive property).
>
> Some notion of a fully simplified expression, called a normal form.

Our goal is to take these pieces of information and turn them into a set of directional reductions which can be applied in any order to yield a fully simplified expression. This set of reductions is called a confluent term rewriting system. Confluence is the property that the reduction can occur in any order, and guarantees that equivalent

expressions will always be reduced to the same form. The Knuth-Bendix completion algorithm [20] is a general algorithm for this task, taking only a set of equalities and a well-ordering, and attempting to construct a confluent term rewriting system. However, this algorithm can be slow or non-terminating, and has trouble with more complicated rewrite schema. In constructing each of the reduction systems, I rely on heuristics or algorithms specialized to particular mathematical objects, such as reduction by a Gröbner basis for the unit system as discussed in section 4.2.

The symbolic engine within MICA workspace is primarily a set of interconnected term-rewriting systems. After parsing, expressions are passed to the main term rewriting system, which does expression simplification. A symbolic function is an instruction to run a different term-rewriting system on the expression or expressions it takes as arguments. To most people, differentiation will be the most obvious symbolic function, because it is often taught in classes as a term-rewriting system, though not by that terminology.

6.1 Differentiation

I will denote a term-rewriting system by a set of directional maps from one expression in Polish notation to another. I use Polish notation in this section rather than a human-friendly form because Polish notation displays the tree structure of the expression clearly, and demonstrates that no nonlocal information is needed for rewriting. I use the letter `n` to denote an aritrary expression (or node). For example, the rule `(sin n1) -> (cos n1)` can rewrite $sin(x^2)$ to $cos(x^2)$. To restrict a node to a particular type, I write it as `type:n1`

I write differentiation as `(D expr name)` where expr is an arbitrary expression, and name is expected to be a name node of the variable we are differentiating with respect to.

Here are some of the familiar rewrite rules for differentiation:

`(D x name:x) -> 1` (Derivative of the identity map is 1)

`(D y name:x) -> 1` (Derivative of a constant is 0)

`(D value:n x) -> 0` (Derivative of a number is 0)

The chain rule is the most important. It lets us split a derivative into simpler derivatives. Recall the multiple parameter form:

$$\frac{\partial}{\partial x} f(g_1(x)...g_N(x)) = \sum_{n=1}^{N} \frac{\partial f}{\partial g_n} \frac{\partial g_n}{\partial x}$$

So we should include the rewrite schema:

`(D (name:f n1...nN) x) ->`

`(+ (* (` $\frac{\partial f}{\partial n1}$ `n1...nN) (D n1 x)) ... (* (` $\frac{\partial f}{\partial nN}$ `n1...nN) (D nN x)))`

For each external function (*e.g.* sine, exponential, log, +, ∗, *etc.*), I found $\frac{\partial f}{\partial ni}$ by hand and stored it in a dictionary of derivatives that the chain rule uses in its rewrite schema.

Those rules encode everything about taking a derivative, as long as you know the derivatives of basic functions. Notice that the rewrite system terminates: The only rule we could be worried about is the chain rule. However, notice that the chain rule turns the derivative of an n-deep abstract syntax tree into a finite sum of products involving only derivatives of n-1-deep or less abstract syntax trees. Thus the chain rule terminates. It is also confluent, because we define exactly one way to move forward with the simplification at each step. The chain rule generates multiple derivatives that might be acted on by our patterns, but none of our patterns allow interaction between two derivatives in separate branches of the abstract syntax tree.

Finally, we need to know how to actually compute the result of applying these rules until they terminate. For the derivative, this is simple: as we apply rules, the parts of the abstract syntax tree with derivatives in them recede deeper and deeper into the tree, until they dissappear at the bottom. In situations like this, where we can

guarantee that a portion of our tree no longer has any simplifications to be made, we can simply walk down the tree, simplifying as we go. This is implemented by a recursive function which takes an expression of the form (D expr n), selects and applies the appropriate pattern above (calling itself recursively to apply the chain rule), and returns the result of applying the pattern.

Differentiating in this way yields unsimplified expressions. For example, `(D (* x y) x)` reduces to $x * 0 + 1 * y$. Simplification is harder to make confluent and terminating, partly because people have different opinions on what constitutes a full simplification. Factoring is a good example. Do you prefer x^2+2x+1 or $(x+1)^2$? Simplifying general expressions is covered in the next two sections. It is done in two parts: expression substitution, and expression simplification.

6.2 Expression Substitution

Simplifying general expressions is done in two parts. First, NAMES are looked up, FUNCTIONS are substituted, and SYMBOLIC functions are run. Once this is done, the resulting expressions are simplified by another term rewriting system.

NAMES are looked up in a dictionary called the context. The context maps NAMES to expressions. When substituting an expression, we start with a basic context, called the global context. The global context holds all of the predefined symbols, for example c is a VALUE representing $3 * 10^8 * m/s$, and sin is an EXTERNAL function handled by python's math.sin. The global context also holds user-defined functions.

As an expression is evaluated, the context can be changed by function evaluation. For example, if we define $f(x) = x^2$, and then evaluate the expression $f(3) + f(4)$, we must evaluate x^2 in two different contexts, one where $x = 3$, and one where $x = 4$. To compute the substitution of an expression, we have a recursive function that takes an abstract syntax tree and a context. Initially, we give it the global context. As we travel recursively down the abstract syntax tree, we substitute names

by looking them up in the context, and we evaluate SYMBOLIC functions by calling their term-rewriting system. We substitute a FUNCTION by extending the context. First, we evaluate the arguments to the FUNCTION in the current context. Then we extend the context, mapping the argument names to the expression they evaluated to. Finally, we evaluate the expression of the FUNCTION in this new context.

There is one last caveat. Occasionally, there may be a name conflict when extending a context. To avoid this, we use a process from lambda calculus called alpha conversion. If we would otherwise overwrite an entry in the context, we generate and use a different name instead. We can do this because parameter names do not matter, *e.g.* the function $f(x) = x^2$ is the same as the function $f(y) = y^2$.

6.3 Expression Simplification

Once the expression has been substituted, we can begin to simplify it. There are a number of simplifications that should be applied in all situations. For example, anything multiplied by 0 yields 0. For most obvious simplification rules, see [21].

I decided upon some heuristic simplification rules. For example, I decided to expand integer powers of sums, *e.g.* $(x+1)^2 \rightarrow x^2 + 2x + 1$. In deciding which rewrite rules to include, I tried to be pragmatic. Expanded integer powers tend to be easier to work with, though it does make working with large powers, *e.g.* $(x+1)^{100}$, awkward. However, large integer powers like these are uncommon in practice. There are a lot of possibilities and strategies for making up rewrite rules for simplification, and none of them seem to stand out clearly from a theoretical perspective. Each set of rules have advantages and disadvantages. Therefore, I simply decided upon some simplification rules that seemed most useful in most real situations. Here are the rewrite rules for the simplification term-rewriting system.

```
(^ x 0) -> 1

(^ x 1) -> x
```

```
(^ (^ a b) c) -> (^ a (* b c))

(* x1 ... xn 0) -> 0

(* x1 ... xn 1) -> (* x1 ... xn )

(+ x1 ... xn 0) -> (+ x1 ... xn )
```

Products containing sums distribute.

If there are multiple values in a product (or a sum), combine them.

```
(* a (^ b c) (^b d)) -> (* a (^ b (+ c d)))

(* a b (^b d)) -> (* a (^ b (+ 1 d)))

(* a b b) -> (* a (^ b 2))
```

Multinomial theorem for expanding $(x_1 + ... + x_n)^m$

Combine products in a sum that differ only by a value: $2 * a + 3 * a$->$5 * a$

```
(/ a b) -> (* a (^ b -1))
```

In order to apply this term rewriting system, we do something similar to the derivative, where we dive down into the abstract syntax tree, replacing portions of the tree with their simplified versions. However, unlike in the derivative term-rewriting system, applying a rewrite rule may allow a new simplification further up the tree. We cannot apply these rules to termination by a single pass down the tree. A simple way to apply these rules until they terminate is to travel down the tree repeatedly until the tree stops changing. It may be worth exploring faster methods, but in practice, this works well. Some demonstrations of expression simplification are shown in Figure 6-1.

These rules perform a general sort of simplification, useful in many, but not all, contexts. More fully developed computer algebra systems use different sets of simplification rules in different contexts. [10]

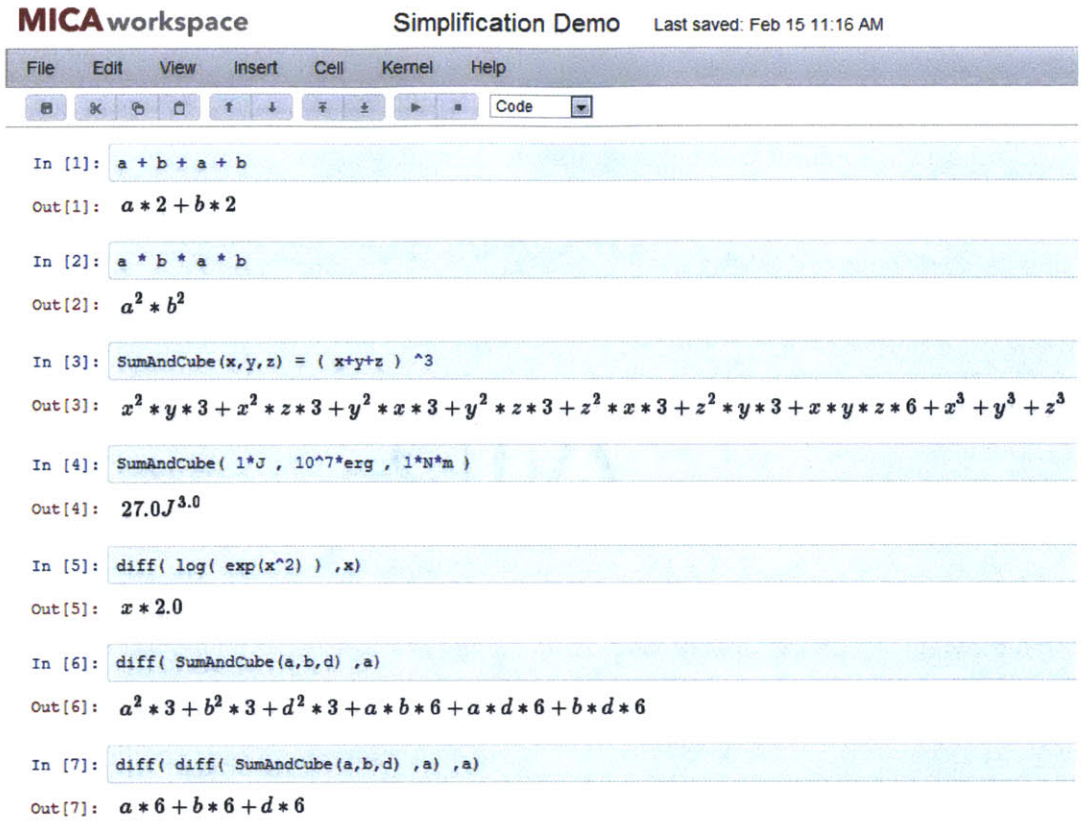

Figure 6-1: MICA Workspace session demonstrating simplification and differentiation.

6.4 Tensors

A data analysis environment certainly needs good tensor handling. Vectors are a natural representation for data, and many standard data analysis techniques are based on linear manipulations.

A TENSOR is represented as a nested tuple, which is a tree structure. If the nth index of the TENSOR has size m, then all nodes n levels deep have m children. The bottom nodes of the tree hold expressions. A few symbolic functions are defined on TENSORS, the most fundamental of which are the outer product and the contraction of two indices. The inner product is a contraction of the outer product, but it can be

computed directly more efficiently.

Like all other types in this symbolic system, TENSORS are a tuple of a few pieces of information. In the following descriptions of functions on TENSORS, I will not mention the detail of pulling the neccesary data from the TENSOR data structure or constructing the new TENSOR data structure after the manipulation.

We define a functional form called deepMap as the analogue of map for a tree-structure. It takes a TENSOR T and a symbolic method f. It returns a new TENSOR which has the same structure as T, but with f applied to each value. The implementation is simple and optimal in computational cost: recursively walk down the tree, at each level reconstructing the TENSOR's shape. When you reach a bottom node, apply f.

We also define a slight generalization of deepMap, called deepCombine, which is used to combine multiple TENSORS of the same shape, in an expression by expression way. It takes a list of TENSORS of the same shape and a function f that takes a list of expressions and returns a single expression. It returns a single TENSOR of the same shape whose value at a given position is f applied to the expressions at that position in the list of TENSORS. Its implementation is also simple. Recursively walk down the set of tensors, at each level reconstructing the general shape. When you reach the bottom, apply f to all of the expressions there.

The outer product is an application of deepMap. Given two TENSORS A and B, construct the function f which takes an expression e and deepMaps the symbolic product of e into B. Then deepMap f into A.

Contraction is more involved. Given a TENSOR T and two indices $i < j$ to contract, recursively walk down the structure of T, reconstructing the shape at each step. When you reach the node i levels deep, deepCombine by symbolic sum the list of the TENSORS obtained by choosing each particular value for i. In order to obtain these TENSORS, branch on the value of i and continue walking down and reconstructing the TENSOR. When you reach a node j levels deep, let j be the chosen value of i, and continue reconstructing until you reach the bottom.

To obtain the inner product of A and B, you can take their outer product and contract by the last index of A and the first index of B. However, the intermediate step of computing the outer product of A and B is inefficient, since it computes values not used in the subsequent contraction. For example, matrix multiplication of size n square matrices with this method is $O(n^4)$. It's possible to reduce this to $O(n^3)$ by creating a special purpose inner-product algorithm by following the contraction algorithm and transitioning to B once you reach the bottom of A.

Finally, in the case that our TENSOR contains only VALUES, we can use a Python scientific computing package called NumPy [9] to do numeric computations, such as singular value decomposition. In this case, we convert our data structures to NumPy's format, use NumPy to perform the SVD, and convert the results back to our system, the results of which can be seen in Figure 6-2.

```
< is the outer product operator.
> is the inner product operator.

In [1]: X(x,y,z) = [1,2,3] < [x,y,z]

Out[1]:
⎛   x      y      z   ⎞
⎝x * 2.0 y * 2.0 z * 2.0⎠
⎝x * 3.0 y * 3.0 z * 3.0⎠

In [2]: Contract( X(x,y,z) ,0,1 )

Out[2]: y * 2.0 + z * 3.0 + x

In [3]: A = 10 * [[1,2,3],[4,5,6]]

Out[3]:
⎛10.0 20.0 30.0⎞
⎝40.0 50.0 60.0⎠

In [4]: SVDU(A) > SVDS(A) > SVDV(A)

Out[4]:
⎛10.0 20.0 30.0⎞
⎝40.0 50.0 60.0⎠
```

Figure 6-2: MICA Workspace session demonstrating some features of tensor manipulation.

6.5 Latexification

Expressions are converted into LaTeX for presentation to the user. Latexification is a term-rewriting system that collapses the abstract syntax tree into a string. The strategy is to walk down the abstract syntax tree, converting the local structure into LaTeX. In most cases, the mapping is obvious. If L represents Latexification, noting that `\frac` is the LaTeX command for fractions, a simple example is:

```
(L (/ n1 n2)) -> \frac{(L n1)}{(L n2)}
```

As another example, to render a tensor, look at the rank and generate a matrix environment of the appropriate shape. The entries of the matrix are the Latexification of the entries in the tensor. There is one situation that is not quite so obvious. In order to render infix expressions without including extraneous parentheses, more than just local information is neccesary; the identity of the parent node is also needed. For example,

Polish	→	Latexified	=	Rendered
`(* x (+ y z))`	→	`x*(y+z)`	=	$x * (y + z)$
`(^ x (+ y z))`	→	`x^{y+z}`	=	x^{y+z}

If the parent operator has higher priority than the child, parentheses are needed. The exception is when the parent introduces a new environment, such as division's `\frac{}{}` or exponentiation's `^{}`.

Latexification, like differentiation, can be computed by a single pass of a recursive function. The only modification is to make available the parent node by handing it down in each recursive call.

Chapter 7

Future Work

MICA Workspace is an ambitious project; it seeks to be a complete environment for data collection and analysis. It is usable in its current state, but in somewhat limited contexts. As of the time of writing, May 2013, students are interacting with MICA for the first time. Inevitably, feedback and further interaction with students will shape the form and features of MICA Workspace. Feedback from real users is nearly always the best indicator for growth direction, but there are a few potential upgrades that deserve to be mentioned.

7.1 Abstraction for Term-Rewriting Systems

As it stands, each term-rewriting system is implemented separately. As the symbolic engine grows, many new term-rewriting systems will have to be built. It would be a good idea to build in the notion of an abstract term-rewriting system. A few important features for this potential term-rewriting system are:

A markup language or user interface for expressing rewrite rules.

Automatic tools that may guarantee confluence or termination, such as Knuth-Bendix Completion.

Ability to handle rewrite schema, such as the multinomial theorem.

This improvement would cleanly isolate the development of the underlying machinary of the general symbolic engine from the particulars of mathematical operations we would like to implement.

7.2 Automatic Low-Parameter Symbolic Function Fitting

When I see a task that people can complete effectively but computers cannot, I like to explore strategies for teaching it to a computer. One general problem where computers lag behind humans is in exploring mathematical systems. A specific example of such a problem that humans face commonly is the problem of symbolic function fitting. Given a set of data, what is a simple, low parameter function that fits the data well? I would like to find a way of automatically solving this problem, because it moves models from the realm of the computer (high parameter numerical models), to the realm of the human (symbolic functions), where the user can reason easily.

I have done some preliminary work towards solving this problem, and I would like to eventually integrate it with MICA Workspace. The strategy is to apply smooth parameterized mutations to an abstract syntax tree, and mutate in a direction that allows a good fit to the function without too much complexity. Akaike's Information Criterion states that the cost of introducing extra parameters or nodes to the abstract syntax tree should be exponentially higher than the cost of having a poor fit. I have not worked out the details of the cost function, or a well justified set of smooth mutations, but some preliminary guesses have yielded good results. My preliminary program was able to automatically identify the function

$$f(x) = \frac{x^3}{e^{0.05*x} - 1}$$

exactly from 20 data points uniformly sampled from a characteristic interval on the function. Figure 7-1 shows the progression of this identification from one to four parameters

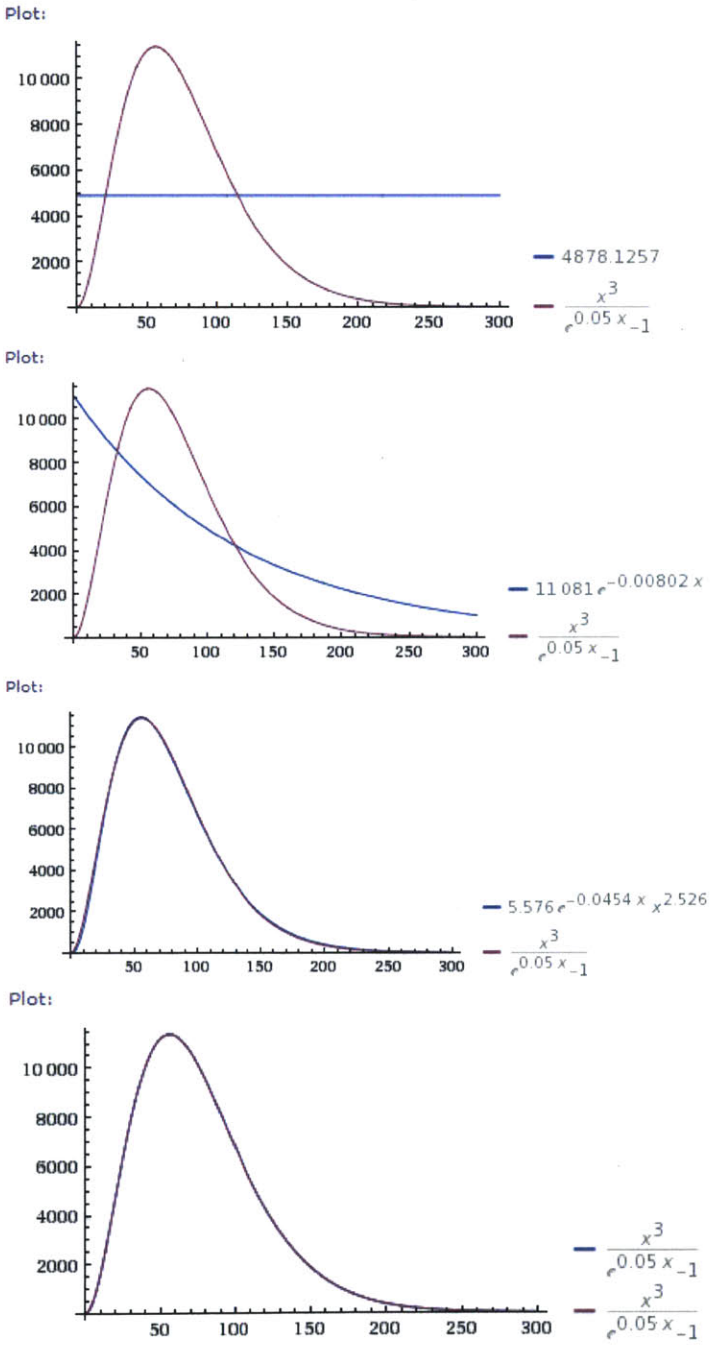

Figure 7-1: 1,2,3, and 4 parameter functions identified from numerical data.

Chapter 8

Conclusion

For education platforms such as edX [22] to supply quality science and engineering education, they will need to give students hands-on experience. MICA aims to provide that experience in the form of a low cost wireless sensor and actuator platform. To explore the resulting data and its implications, students need a data analysis environment with features such as unit handling and differentiation. MICA Workspace provides these features by using ideas from computational mathematics and symbolics.

Bibliography

[1] Wahab, A.; Spanbauer, A.; Hunter, I.; Hughey, B.; Jones, L. *MICA: an innovative approach to remote data acquisition.*

[2] Spanbauer, A.; Wahab, A.; Hemond, B.; Hunter, I.; Jones, L. *Measurement, instrumentation, control and analysis (MICA): A modular system of wireless sensors.* IEEE International Conference on Body Sensor Networks, 17-21, 2013.

[3] Beagle Board Open Project. http://beagleboard.org/

[4] ARM Cortex^{TM}-M3 Processor. http://www.arm.com

[5] Prez, F.; Granger, B. *IPython: A System for Interactive Scientific Computing, Computing in Science and Engineering.* vol. 9, no. 3, pp. 21-29, May/June 2007, doi:10.1109/MCSE.2007.53. http://ipython.org

[6] Wolfram Research, Inc. *Mathematica.* 2010, http://www.wolfram.com/mathematica/

[7] Maplesoft, a division of Waterloo Maple Inc. *Maple.* http://www.maplesoft.com/

[8] SymPy Development Team. *SymPy: Python library for symbolic mathematics.* 2009, http://www.sympy.org

[9] Jones, E; Oliphant, T; Peterson, P; and others. *SciPy: Open Source Scientific Tools for Python* 2001, http://www.scipy.org

[10] Joel S. Cohen. *Computer Algebra and Symbolci Computation: Elementary Algorithms.* A K Peters, Ltd, 2002.

[11] Bruno Buchberger, *An Algorithm for Finding the Basis Elements of the Residue Class Ring of a Zero Dimensional Polynomial Ideal.* Ph.D. dissertation, University of Innsbruck. English translation by Michael Abramson in Journal of Symbolic Computation 41: 471511. 2006.

[12] Michael Artin. *Algebra.* Pearson, 1991.

[13] Becker, T.; Weispfenning, V. *Gröbner Bases: A Computational Approach to Commutative Algebra.* Springer, 1993.

[14] Victor Adamchik. *15-355: Modern Computer Algebra, Lecture Notes.* http://www.andrew.cmu.edu/course/15-355/

[15] Geddes, K.; Czapor, S.; Labahn, G. *Algorithms for Computer Algebra.* Kluwer Academic Publishers, 1992.

[16] Manuel Bronstein. *Symbolic Integration I: Transcendental Functions.* Springer, 2005.

[17] Grayson, D.; Stillman, M. *Macaulay2, a software system for research in algebraic geometry.* http://www.math.uiuc.edu/Macaulay2/

[18] Cohen, A.; Cuypers, H.; Sterk, H.; (Editors). *Some Tapas of Computer Algebra.* Springer, 1999.

[19] Press, W.; Teukolsky, S.; Vetterling, W.; Flannery, B. *Numerical Recipes in C.* Cambridge University Press. 1992.

[20] Knuth, D.; Bendix, P. *Simple word problems in universal algebras.* Computational Problems in Abstract Algebra (Ed. J. Leech) pages 263297. 1970.

[21] Joel S. Cohen. *Computer Algebra and Symbolci Computation: Mathematical Methods.* A K Peters, Ltd, 2003.

[22] edX Project. https://www.edx.org/