AME

A-LEVEL MATHEMATICS ENGINE

Contents

1 Analysis **1**

1.1 Current Systems 2
1.2 End Users 3
1.3 User Needs 3
1.3.1 Required Calculations 5
1.4 Objectives 8

2 Documented Design **11**

2.1 Overall Overview 13
2.2 Compute Module 14
2.2.1 Data Structures 15
2.2.2 Matrix 15
2.2.3 MatrixExt 15
2.2.4 Expr 16
2.2.5 Compute 19
2.2.6 Table 19
2.2.7 Shape 20
2.2.8 Algorithms 21
2.2.9 The Simplification Algorithm 21
2.2.10 The Polynomial Solving Algorithm 25
2.2.11 The Integration Algorithm 29
2.2.12 Matrix Operations 32
2.2.13 Statistical Operations 33
2.2.14 Geometric Operations 33
2.3 Interpret 34

2.3.1 Data Structures . . . 34
2.3.2 Interpret . . . 35
2.3.3 Expression . . . 35
2.3.4 Value . . . 36
2.3.5 Algorithms . . . 37
2.3.6 Parser Combinators . . . 37
2.3.7 Lexer . . . 40
2.3.8 Parser . . . 41
2.3.9 Interpreter . . . 42
2.4 Server . . . 43
2.5 Client . . . 44
2.5.1 GUI Design . . . 45
2.5.2 Functionality . . . 46
2.5.3 Main Process . . . 46
2.5.4 Renderer Process . . . 46
2.6 Walkthrough of Execution . . . 48

3 Technical Solution **49**
3.1 Syntax Reference . . . 49
3.2 Guide to Compiling The Code . . . 52
3.2.1 Installing Prerequisites . . . 52
3.2.2 Building The Server . . . 53
3.2.3 Building The Client . . . 53
3.2.4 Running the Automated Tests . . . 53
3.3 Guide to Reading The Code . . . 53

4 Testing **55**
4.1 Test Plan . . . 56
4.2 Automated Testing . . . 58
4.2.1 Test Output . . . 58
4.3 Manual Testing . . . 63
4.3.1 Test 1 . . . 63
4.3.2 Test 2 . . . 65
4.3.3 Test 3 . . . 67
4.3.4 Test 4 . . . 68
4.3.5 Test 5 . . . 71
4.3.6 Test 6 . . . 74
4.3.7 Test 7 . . . 76
4.3.8 Test 8 . . . 78
4.3.9 Test 9 . . . 80
4.3.10 Test 10 . . . 82

4.3.11 Test 11 . . . 84
4.3.12 Test 12 . . . 85
4.3.13 Test 13 . . . 86
4.3.14 Test 14 . . . 88
4.3.15 Test 15 . . . 90
4.3.16 Test 16 . . . 92
4.3.17 Test 17 . . . 93
4.3.18 Test 18 . . . 94

5 Evaluation **95**
5.1 Current Limitations . . . 96
5.2 User Feedback . . . 97
5.3 Future Extensions . . . 98
5.4 Overall Evaluation . . . 99

A Source Code **101**
A.1 Compute Component . . . 101
A.1.1 AME/Compute/Expr.hs . . . 101
A.1.2 AME/Compute/Error.hs . . . 104
A.1.3 AME/Compute/Simplify.hs . . . 105
A.1.4 AME/Compute/Solver.hs . . . 118
A.1.5 AME/Compute/Solver/Polynomial.hs . . . 123
A.1.6 AME/Compute/Solver/Newton.hs . . . 128
A.1.7 AME/Compute/Matrix/Base.hs . . . 130
A.1.8 AME/Compute/Matrix.hs . . . 134
A.1.9 AME/Compute/Calculus/Derivatives.hs . . . 143
A.1.10 AME/Compute/Calculus/Integrals.hs . . . 145
A.1.11 AME/Compute/Geometry.hs . . . 166
A.1.12 AME/Compute/Statistics.hs . . . 169
A.2 Interpret Component . . . 172
A.2.1 AME/Interpret/AST.hs . . . 172
A.2.2 AME/Interpret.hs . . . 175
A.2.3 AME/Parser/Combinators.hs . . . 190
A.2.4 AME/Parser/Lexer.hs . . . 195
A.2.5 AME/Parser.hs . . . 198
A.3 Server Component . . . 215
A.3.1 Main.hs . . . 215
A.4 Client Component . . . 225
A.4.1 Renderer Process . . . 225
A.4.2 Main Process . . . 237
A.5 Testing . . . 239

1 ANALYSIS

Computer Algebra Systems (CAS) are systems which allow users to perform a variety of mathematical calculations, usually designed to emulate operations that are typically performed by hand, such as simplification of mathematical equations and equation solving. CAS programs also provide an environment in which the user can run these calculations, allowing the user to define variables and functions, thus calculations may depend on the results of other calculations, which allows many kinds of problems to be either fully or partially solved using CAS. An example session using the CAS system SymPy is shown below:

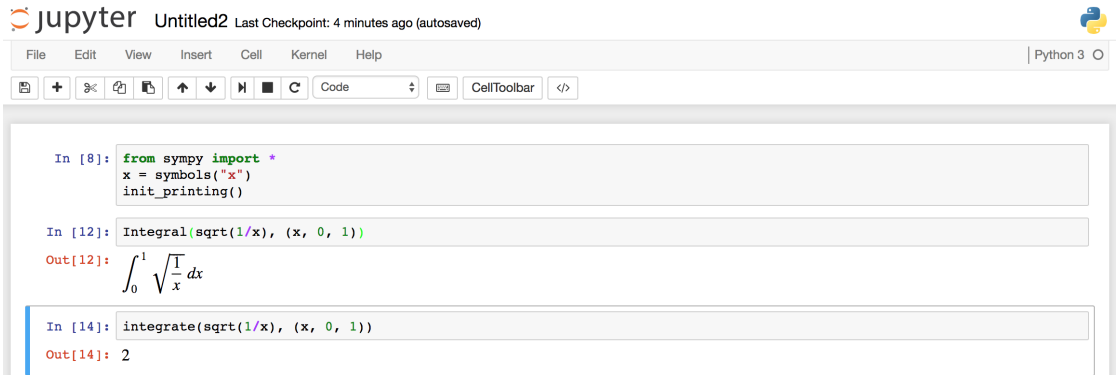


Figure 1.1: An example of solving an integral using SymPy

However, current systems can be difficult to use without specialised knowledge: many CAS programs use a programming language as their input method, and so in order to use these systems, the user must be able to program, which makes them much less accessible to most users. Additionally, for many users popular CAS programs contain far more features than are ever needed, and this can result in such programs being very large and expensive. My program aims to be a solution to these problems.

1.1 CURRENT SYSTEMS

Some of the most popular CAS systems include Mathematica and Matlab (Commercial), PARI/GP and SageMath (Free). The commercial systems can be very expensive, for example with Mathematica, price ranges from £210 for non-commercial users[1] , to £2,475 for industry users[2] . While free alternatives exists, these are not as well maintained, or as feature-complete as the commercial solutions.

All of these programs also take their input in the form of a programming language, e.g, Python for SageMath and the Wolfram language for Mathematica. The exception is Wolfram Alpha which uses an English natural language interface:

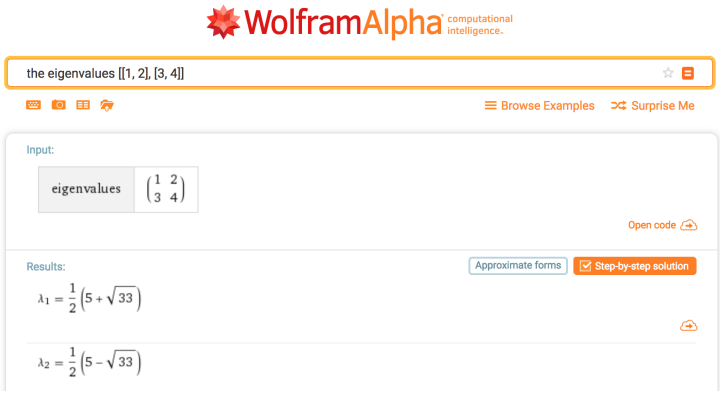


Figure 1.2: An example of a calculation with Wolfram Alpha.

However, Wolfram Alpha is the most limited out of all of these, as it does not offer any environment for the computations, instead only allowing single computations at a time. This can be seen from the image above, as there is only space to input a single question.

Alternatively, graphical calculators, such as the TI-Nspire or Casio FX series, can perform many of the same operations, but these are expensive (£99 for the Casio FX-9860GII [3]) and are often severely limited by the power of the hardware they contain, especially RAM capacity, whereas solutions based on PC software have access to much more powerful hardware.

My program seeks to rectify these problems by allowing a natural language input, providing an environment for performing these calculations, and being freely available.

[1] https://www.wolfram.com/mathematica/pricing/home-hobby-individuals.php
[2] https://www.wolfram.com/mathematica/pricing/industry-individuals.php
[3] https://www.casio.co.uk/products/calculators/graphical-calculators/

1.2 END USERS

The primary end user for my program is myself. I currently use a Casio FX-9860GII, but I often find it is not powerful enough for the problems I wish to solve, for example, symbolic integrals. I considered buying Mathematica, however the high cost and steep learning curve motivated me to pursue this project instead.

In addition to myself, the program is designed for use by A-level mathematics students and I will be showing it to my class at school when it is finished. Many A-level students do not know about or use CAS programs, however, for students going on to study STEM subjects, this would be useful, as CAS is extensively used in many courses. My program could act as a simpler, introductory system so that when students come to use more advanced systems, such as Matlab (widely used in universities) the learning curve would not be as steep.

I talked to my maths class at school about this idea and they said that they thought that it would work if well implemented, and they also suggested that I review the exam syllabus to determine what calculations I should include.

1.3 USER NEEDS

- A program which can perform many of the kinds of calculations needed for the A-level Mathematics specification (The specific calculations which can be performed is discussed in the Required Calculations section).
- The ability to run the program on each of the major platforms (Windows, macOS and Linux).
- The program should be freely available and easy to download and run.
- The ability to load and save previous calculation sessions.
- The ability to define variables and functions with the results of previous calculations.
- An intuitive natural language interface, with an easy to use GUI.

ACCEPTABLE LIMITATIONS

- Only some cases of some calculations can be correctly and/or accurately performed (discussed in the Required Calculations section).
- The natural language interface may not be very sophisticated. In particular, it is not expected to detect or alert the user to ambiguous inputs, handle alternative

phrasings or spellings, or to accept a more than one word for a given term (i.e handling synonyms). It is also not expected to provide detailed error messages when the user inputs a syntax error, although some level of basic error reporting is expected.

1.3.1 REQUIRED CALCULATIONS

At A-level, many kinds of calculations are required, such as equation solving, integration, differentiation, geometry problems, summations and matrix operations. However, many of these calculations are very difficult to perform for some or all kinds of input. Therefore, it is not feasible to implement all of them in the proposed solution.

For this program, I am referring to the Edexcel GCE Mathematics (2008) specification [4], which is the course taught by my school.

Based on the importance and abundance of different types of calculations in the A-level specification, I have divided the calculations into three categories: Category 1, where implementation is required and is required to work for all of the input categories specified, Category 2, where implementation is required, but may not work for every input within the given category, and Category 3, which is calculations which will *not* be implemented.

In this case the meaning of a calculation "working" is that it provides either: an error message saying that the calculation cannot be performed for a given reason (e.g a matrix cannot be inverted, because the matrix is not invertible), or a correct solution to the calculation. However, it must *not* produce an incorrect or incomplete solution to the calculation. To express all of the input categories I will first make some definitions:

- An elementary expression is defined as any arbitrary combination of sums, multiplications, divisions and exponentiation, also possibly involving trigonometric functions, their inverses, the natural logarithm and absolute value function. e.g $\frac{\sin(2)}{(1+5)} \cdot \| \log(4) \|$
- An elementary function is a function that outputs an elementary expression in terms of its inputs. e.g $f(x) = \frac{e^x}{atan(\sqrt{x})}$
- A polynomial is a function consisting of a sum of multiples of non-negative integer powers of a variable. e.g $f(x) = 3x^2 + x^3 + 1$
- A rational functional is any function consisting of any polynomial divided by any other polynomial. e.g $f(x) = \frac{x^3+2x+1}{x^2+1}$. Note that any polynomial is also a rational function.
- A univariate function or expression is a function or expression with only one variable in it. e.g $f(x) = 1 + x$ is univariate, but $f(x, y) = \frac{x+y}{xy}$ is not.
- A linear equation is an equation which contains only constants and multiples of variables. e.g $2x + 3y = 2$ is linear, but $x^2 + 1 = 0$ is not.
- A dataset is one or more ordered collections of numbers.

[4] https://qualifications.pearson.com/en/qualifications/edexcel-a-levels/mathematics-2008.html

CATEGORY 1

Computation	*Acceptable Inputs*	*Specification Modules*
Evaluation of Arithmetic Expressions	All elementary expressions.	All
Solving Equations Numerically	All univariate polynomials.	All
Solving Simultaneous Equations	All systems of linear equations.	All
Differentiation	All elementary functions.	C1, C2 (basic functions), C3 (trigonometric functions)
Integration	All univariate rational functions, and trigonometric functions on their own.	C1, C2 (polynomials), C3 (trig functions), C4, FP2 (rational functions)
Matrix Inversion	All matrices	FP1
Matrix Determinants	All matrices	FP1
Matrix Arithmetic	All matrices	FP1, FP2
Intersections of Shapes	All circles and lines.	C1, C2
Equations of Shapes	All circles and lines.	C1, C2, FP2
Averages (Mean, Median, Mode)	All datasets.	S1-4
Spread (Variance, Standard Deviation)	All datasets.	S1-4
Correlation (Chi Squared, Pearson/Spearman Correlation)	All datasets.	S3, S4

CATEGORY 2

Computation	*Acceptable Inputs*	*Specification Modules*
Solving Equations Numerically	All elementary functions.	C3, C4 (trigonometric functions), C2 (log functions)
Integration	Functions of logs and of square roots, simple instances of integration by parts.	C4, FP2 (integration by parts)
Matrix Eigenvalues	All matrices	FP3
Matrix Eigenvectors	All matrices	FP3

CATEGORY 3

Computation	*Acceptable Inputs*	*Specification Modules*
Evaluation of Arithmetic Expressions	Any other expression, including expressions of hyperbolic trigonometric function.	FP3
Exact Solutions of Equations	Any equation.	All
Solving Differential Equations	All differential equations.	C4, FP2
Integration	Any other elementary function.	C4, FP2 (integration by substitution)
Summation	All elementary functions.	C1 (arithmetic series), C2 (geometric series), C3 (sums of powers), C4 (method of differences)

1.4 OBJECTIVES

1. The program should have a GUI that allows the user to edit a calculation session.
    - a) It should allow the user to save the current session, save a copy of the current session (commonly called "save as"), load a previously saved session or create a new session.
    - b) It should accept a line of input from the user, and display the result of the calculation.
    - c) It should also display all previous user input with its respective output.
2. The program should be able to perform calculations as specified in the Required Calculations section.
    - a) It should be able to perform all Category 1 and Category 2 computations.
    - b) The output of all Category 1 computations should be either an error message or a correct and complete solution to the computation.
    - c) The output of all Category 2 computations should be either an error message or a solution to the computation.
    - d) Any additional unspecified calculations should be performed to Category 2 standard.
3. The program should process calculations via an English natural language interface.
    - a) It should be able to parse natural language expressions containing English words as well as mathematical notation where appropriate.
    - b) Every specified computation should be able to be inputted via the natural language interface.
    - c) All types of input to the computation, i.e numbers, matrices, vectors specifications of shapes, and datasets should have a representation in the natural language interface.
    - d) It may accept a limited number of alternative phrasings for a given computation, although this is strictly optional.
    - e) It should alert the user when they have entered incorrect syntax, and optionally attempt to provide an explanation of what is wrong.

4. The program should run on all major platforms.
    - a) It should be able to run on Windows, macOS and Linux
    - b) It should provide a consistent user experience and interface across all platforms

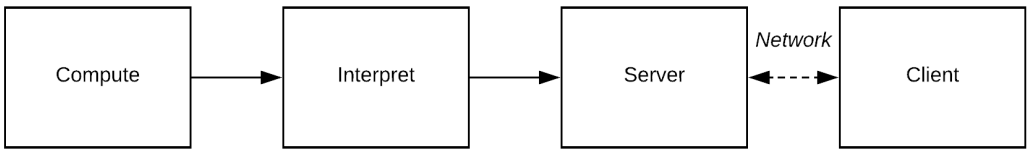
2

DOCUMENTED DESIGN

In order to make my project easier to design, I have broken it up into four separate components:

- Compute: This performs all the computations required by my objectives.
- Interpret: This contains the natural language interface, as well as an interpreter for the language it defines.
- Server: This uses the compute and interpret modules to build a server that takes input phrases over then network, parses and interprets them and then sends the results of the computation back over the network.
- Client: This contains a GUI which interacts with and manages an instance of the server module.

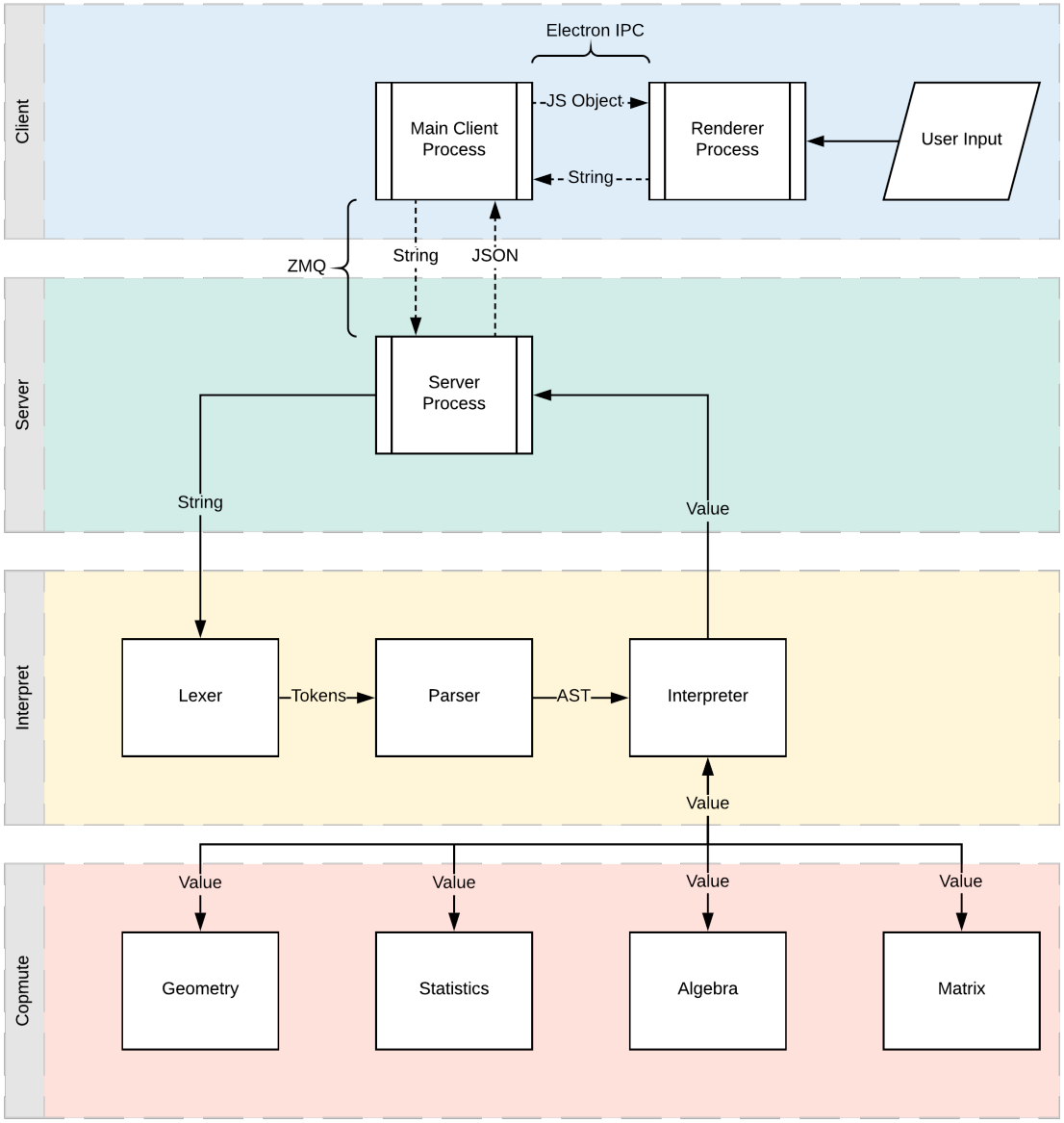
In order to make each of these modules easy to integrate with the other, the compute and interpret components are written as separate libraries, rather than simply including the code into one large server executable. This is also advantageous as it speeds up compile times massively, as when a change to one of these libraries occurs, the whole server executable need not be rebuilt, and instead the library can be rebuilt and the server module simply re-linked to it. This can all be represented in the following data flow diagram:



Where the solid lines indicate that one component library is used in another component. The separation of the client and server instead of one integrated GUI component was partially due to technical reasons. I decided early on that I wanted to write the compute and interpret components in Haskell and the GUI in JavaScript as there are no easy to use Haskell GUI libraries currently exist. Unfortunately there is no simple way to get these two languages to interact directly so it had to be done over the network.

2.1 OVERALL OVERVIEW

The following diagram shows a diagram of the system as a whole, in order to make it easier to follow where each module fits into the system. Direct function calls are represented as solid lines, and calls over some transport medium (network, IPC etc.) are represented as dashed lines. Text on lines represents the data structure being transferred.



2.2 COMPUTE MODULE

This component is simply a collection of functions to perform all the computations. It does not have any overall data or execution flow at it is a library and not an executable. It consists of nine different modules which implement different sections of these computations:

- Expr
  - Defines expressions, equations and the possible errors that may arise.
- Simplify
  - Simplification algorithm
  - Substitution of variables in expressions
- Solver
  - Numerical solvers for polynomials and other equations
- Integrals
  - An algorithm for integration, and a simple wrapper around this to provide definite integrals.
- Derivatives
  - Implements differentiation
- Statistics
  - Implements averages, spread measures, correlation measures and the construction of datasets.
- Matrix
  - Implements eigenvalues and vectors, matrix arithmetic, determinants and inverses.
- Geometry
  - Implements intersections and Cartesian equations of shapes.

However, in order for these to be implemented so that they inter-operate easily and can be easily used, appropriate data structures must be designed.

2.2.1 DATA STRUCTURES

There are four main user defined data structures that the compute module uses: Matrix, Expr, MatrixExt and Compute. All of these are generic data structures, that is, they are parameterised over some given types. Matrix and MatrixExt over the type of their values, Expr over the type of its constants and the type of its variables, and Compute over the type that it is representing. Two other data structures, Shape and Table are used by the Geometry and Statistics modules respectively to represent shapes and datasets.

2.2.2 MATRIX

This represents a matrix of values, that is a rectangle of values with a defined width and height. The height and width are recorded in separate fields in order to make it easier to check them in functions which require certain conditions (e.g inverses require a square matrix). The values are represented as a two dimensional array of values. The matrix structure is generic over the type of its values. So this need not even be some kind of number type, but most operations involving matrices are defined with conditions on the types that its values can take. So a matrix can be represented (in pseudocode):

```
1 STRUCT Matrix<T>
2     FIELD Width: Integer
3     FIELD Height: Integer
4     FIELD Values: Array<Array<T>>
5 END STRUCT
```

2.2.3 MATRIXEXT

This is simply a data structure that can be either a scalar value or a matrix value. This is used to implement a number type that can take either a matrix or a scalar. This is useful for operations such as multiplication which can take either and have very different behaviour for each types. This kind of overloading is not supported in Haskell, and so this type must be used instead. In Haskell, this is easily defined using Sum types (also called tagged unions). These are data types which can be one or more different structures, and facilities are provided to allow pattern matching on these types to determine which variant a given value is. In Haskell the definition of MatrixExt would look like this:

```
1 data MatrixExt t = ScalarValue t
2                  | MatrixValue (Matrix t)
```

In more traditional procedural languages, this would be implemented as a union containing the actual value of the type, and a tag defining which union member is being used currently. Hence the name tagged union. Using this, MatrixExt can be represented in pseudocode:

```
STRUCT MatrixExt<T>
    FIELD IsAMatrix: Boolean
    UNION
        FIELD MatrixValue: Matrix<T>
        FIELD ScalarValue: T
    END UNION
END STRUCT
```

2.2.4 EXPR

This represents any elementary expression. It is parameterised over two types: the variable type and the value type. These represent the variables that the expression can vary over, and the type of the constants that can be present in the expression. It is represented as a large sum type:

```
data Expr var val = Const var
                  | Variable var
                  | Sum [Expr var val]
                  | Mul [Expr var val]
                  | Div (Expr var val) (Expr var val)
                  | Pow (Expr var val) (Expr var val)
                  | Sin (Expr var val)
                  | Cos (Expr var val)
                  | Tan (Expr var val)
                  | ASin (Expr var val)
                  | ACos (Expr var val)
                  | ATan (Expr var val)
                  | Exp (Expr var val)
                  | Log (Expr var val)
                  | Abs (Expr var val)
```

This can be represented in pseudocode similar to the representation of MatrixExt, however it would be very large so I will not give it here. Each of the cases represent the corresponding mathematical operations. I.e, `Sum xs` where `xs` is a list represents the sum of all the elements of the list, and the same is true for `Mul` and products. Similarly, `Sin x` represents the sine of some other expression `x` and so on.

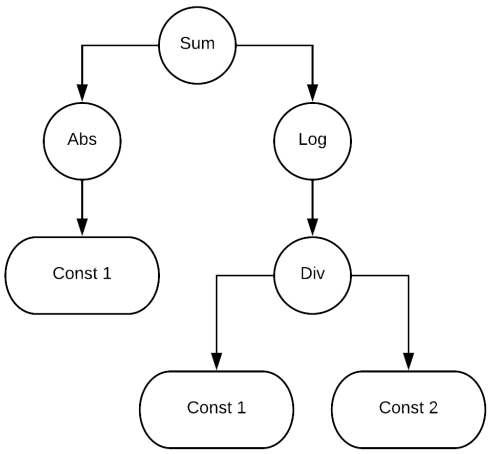
As can be seen from the type, all of the operations are defined as containing Exprs themselves. This makes this type a *recursive type*, and it forms a *tree structure*. Where any of the nodes are these operations, and the leaf nodes are either Const or Variable. Because of this recursive tree structure, complicated expressions can be encoded, for example:

```
1 Sum [Abs (Const 1), Log (Div (Const 1) (Const 2))]
```

Represents the mathematical expression:

$$1 + log(\frac{1}{2})$$

We can see the tree structure of this particular example by drawing it out as a diagram:



The purpose of parameterising Expr over the type of values is obvious: That way it is not tied down to any specific type so if it needed to be switched later for some reason that should be very easy. However, the reason for parameterising over the type of variables is more subtle. A naive implementation of Expr may simply put the type of variables simply as a String. However, by requiring a type for the variables, it solves the problem of naming conflicts.

In the naive definition, any two values of `Variable "x"` are identical. However in this definition, if I were to define two separate types:

```
data Var1 = X

data Var2 = X
```

And then construct values, we can see that the following values are very different:

```
a :: Expr Var1 val
a = Variable X

b :: Expr Var2 val
b = Variable X

a /= b
```

(Here `x :: t` means `x` has type `t`)

This solves the name conflict problem easily: To construct an expression we simply define a type for all of its variables, calling them whatever is convenient, and then use that type as our variable type. For example:

```
data CircleVars = X | Y

circleExpr :: Expr CircleVars Double
circleExpr = Sum [Pow (Variable X) (Const 2), Pow (Variable Y)
    (Const 2)]
```

This also solves the problem of undefined variables. We can see this in the above example: it is simply impossible to construct a value of type `Expr CircleVars Double` that uses a variable other than `X` or `Y` because there is no way to construct a value of type `CircleVars` other than `X` or `Y`.

2.2.5 COMPUTE

Lastly, Compute represents the result of some computation that may or may not produce an error. For example:

```
1 someComputation :: Integer -> Compute Integer
2 someComputation = ...
```

`someComputation` is a function which takes an integer and attempts to return an integer, but may or may not fail in the process. Compute can be represented as follows:

```
1 data Compute a = Success a
2                | Failure String
```

Where the string component of `Failure` is an error message. This is convenient to use for any operation which may fail. For example, some instances of matrix multiplication will fail (i.e if the matrices are not the right sizes), so in my program I don't give an implementation of `(*) :: Matrix t -> Matrix t -> Matrix t` but instead `(*) :: Compute (Matrix t) -> Compute (Matrix t) -> Compute (Matrix t)`.

2.2.6 TABLE

Table represents a dataset. It consists of a field of variable names and then a field with a collection of sets of numbers. Each variable name has an associated set of numbers which represent the values of that variable in the data set. This can be represented as:

```
1 STRUCT Table<T>
2     FIELD Variables: List<String>
3     FIELD Values: List<List<T>>
4 END STRUCT
```

There were several alternatives for representing this data structure: for example, I could have used a hash table, mapping Strings to Lists of values. However, since it unlikely for there to be more than two or three variables in any given table for the purposes of this program, I deemed that was unnecessary and that a linear search through the table in order to find the correct list of numbers would not be too slow.

2.2.7 SHAPE

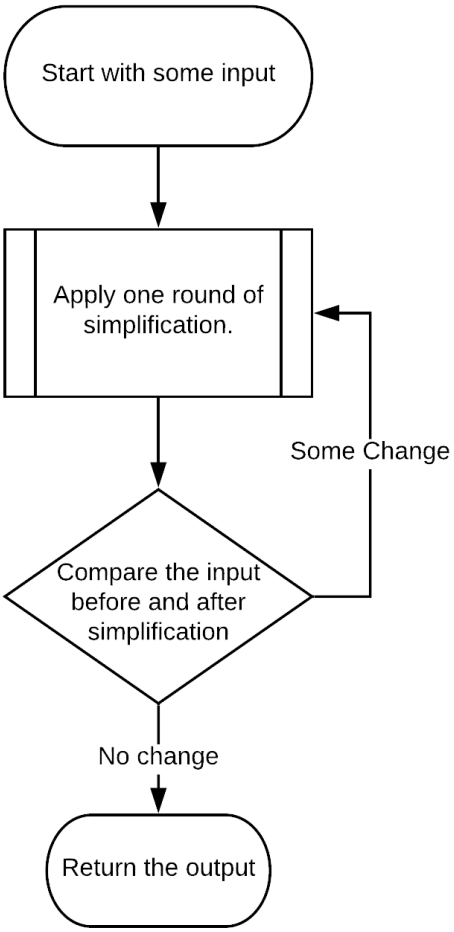
This is used to represent either circles or lines that can be used in geometrical operations. For both shape types, I had to pick a representation to use for them. For circles, I picked a representation of two fields: the centre point, and the radius. This was an easy choice as using this representation leads to significantly simpler algorithms than an alternative representation such as three points that the circle passes through. For lines, I chose to use a point through which the line passes, and the gradient of the line. This was because I could then use algorithms for finding intersections with vectors instead of with lines. These are much more widespread and have been heavily optimised as they are needed for many applications in compute graphics. Putting this all together, we represent Shape as follows:

```
1 data Shape v = Line (v, v) v
2              | Circle (v, v) v
```

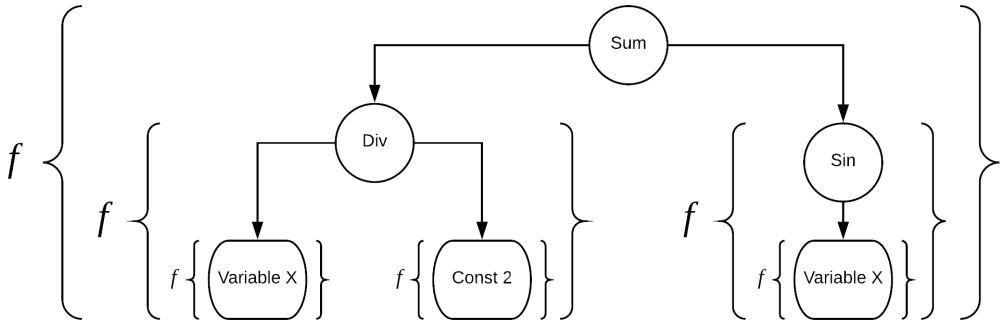
2.2.8 ALGORITHMS

2.2.9 THE SIMPLIFICATION ALGORITHM

This algorithm was one of the most time consuming algorithms to design and debug in the entire project. The core algorithm goes as follows:



In each round of simplification, 71 passes are performed on the input, with 19 different rewrite rules. These rules are mapped over the tree of the expression recursively in the following way: For some rewriting function, it is applied to every part of the tree, starting at the bottom and working its way up, in a post-order traversal. This is more easily explained in an example diagram. For some rewriting function f on the input `Sum [(Div (Variable X) (Const 2)), Sin (Variable X)]`, the algorithm evaluates the following:



These rewriting functions are very small functions that use pattern matching to match a handful of cases and simplify them, while simply ignoring the remaining cases. For example:

```
1 idmul (Mul [1, x]) = return x
2 idmul x = return x
```

This matches all multiplications by 1 and simplifies them, ignoring all other cases. A more complicated one might be like this:

```
1 inmul (Pow s@(Sum _) (Const c)) = return $ Mul $ replicate c x
2 inmul x = return x
```

This looks for constant powers of sums and rewrites them as a multiplication of the sum a constant number of times. i.e $(1+x)^3 = (1+x)(1+x)(1+x)$ or in Haskell:

```
1 inmul (Pow (Sum [1, x]) (Const 3)) == Mul [Sum [1, x], Sum [1, x],
    Sum [1, x]]
```

The rewriting functions I use in my program are as follows:

- `idsum`: Simplifies cases of empty sums, additions of zero and sums of one element.
- `idmul`: Simplifies cases of empty products, products of one element and multiplications by one.
- `iddiv`: Simplifies divisions by one.
- `idpow`: Simplifies exponentiation by one.
- `invle`: Simplifies $\log(\exp(x))$ and $\exp(\log(x))$ to x.
- `intrg`: Simplifies $\tan(\tan^{-1}(x))$, $\tan^{-1}(\tan(x))$ and the equivalents for $\cos$ and $\sin$.
- `kvmul`: Simplifies multiplication by zero.
- `kvpow`: Simplifies exponentiation by zero, one to any power and zero to any power.
- `cstsimpl`: Evaluates arbitrary expressions involving no variables to a single `Const` value.
- `cstrnd`: Rounds all constants to five decimal places.
- `sumsum`: Simplifies sums of sums.
- `mulmul`: Simplifies products of products.
- `mulsum`: Multiplies out products of sums.
- `muldiv`: Multiplies out products of fractions.
- `lksum`: Collects like terms in sums.
- `lkmul`: Collects like terms in products.
- `inmul`: Multiplies out integer powers of sums.
- `smdiv`: Puts sums of fractions over a common denominator.
- `cvdiv`: Divides through by powers of variables in a fraction.

Looking at this from a purely numerical standpoint, this should take a long time: one expression with 10 nodes was measured to need 246 rounds of simplification, for a total of 17,466 passes over the data, resulting in 174,660 function applications to the data. Despite this huge number, this operation took only 400ms, which is more than quick enough for a responsive application. This shows one advantage of using a high-level compiled language such as Haskell: the compiler is incredibly good at optimising.

2.2.10 THE POLYNOMIAL SOLVING ALGORITHM

This algorithm was designed to robustly solve polynomials, as an alternative to using the general numerical solver. This is because polynomial solving is a Category 1 computation so we need to be correct and complete - the other solver is based on Newton's method and is incredibly temperamental, often missing roots or returning false positives. This method is based on the following mathematics:

Theorem 1 For any continuous function, $f(x)$, if $f(a)$ and $f(b)$ have different signs then there must be at least one root in the interval (a, b). **Proof** For any x, y with different signs, $x < 0 < y$. Thus if $f(a)$ and $f(b)$ have different signs then $f(a) < 0 < f(b)$ but by the intermediate value theorem, for any x with $f(a) < x < f(b)$ there must exist y such that $f(y) = x$ and $a < y < b$ hence there must exist c such that $f(c) = 0$ with $a < c < b$ therefore there is a root in the interval (a, b). *QED*.

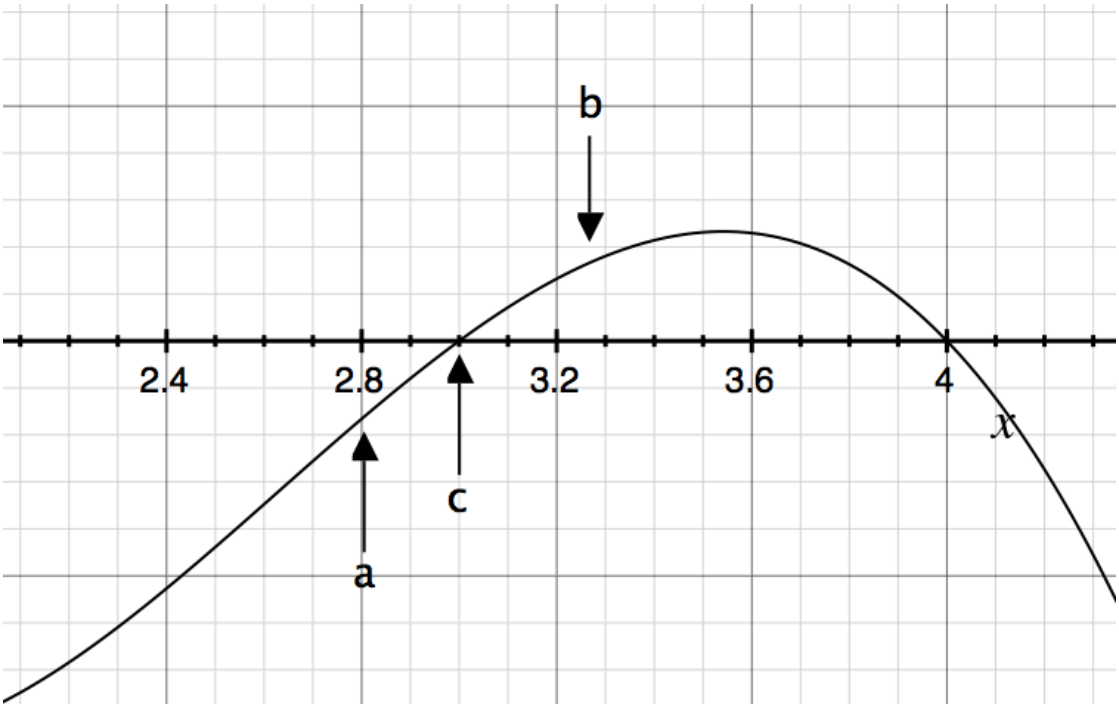


Figure 2.1: An Example of Theorem 1

Theorem 2 If a differentiable non-constant function $f(x)$ has at least one root in the interval (a, b) and no turning points in the same interval, then there is exactly one root in the interval. **Proof** If there was more than one root, we can, without loss of generality, assume that there are two roots α and β, with $\alpha < \beta$, in the interval (a, b). Because there are no turning points in the interval, the gradient at α must be non-zero, and without loss of generality, let us assume that the gradient is positive. Thus there must exist some ϵ with $0 < \epsilon < \beta - \alpha$ such that $f(\alpha + \epsilon) > 0$ and $f'(\alpha + \epsilon) > 0$.

However, since $f(\beta) = 0 < f(\alpha + \epsilon)$, it must be that $f'(x) < 0$ for some $\alpha + \epsilon < x < \beta$, as the function decreases. However, since $f'(\alpha + \epsilon) > 0$ and $f'(x) < 0$ there is a change of sign and hence a root of $f'(x)$ in the interval $(\alpha + \epsilon, \beta)$. This root would indicate a turning point. However, we know there are no turning points in the interval (a, b), so this is a contradiction. Therefore there must be exactly one root in the interval. *QED*.

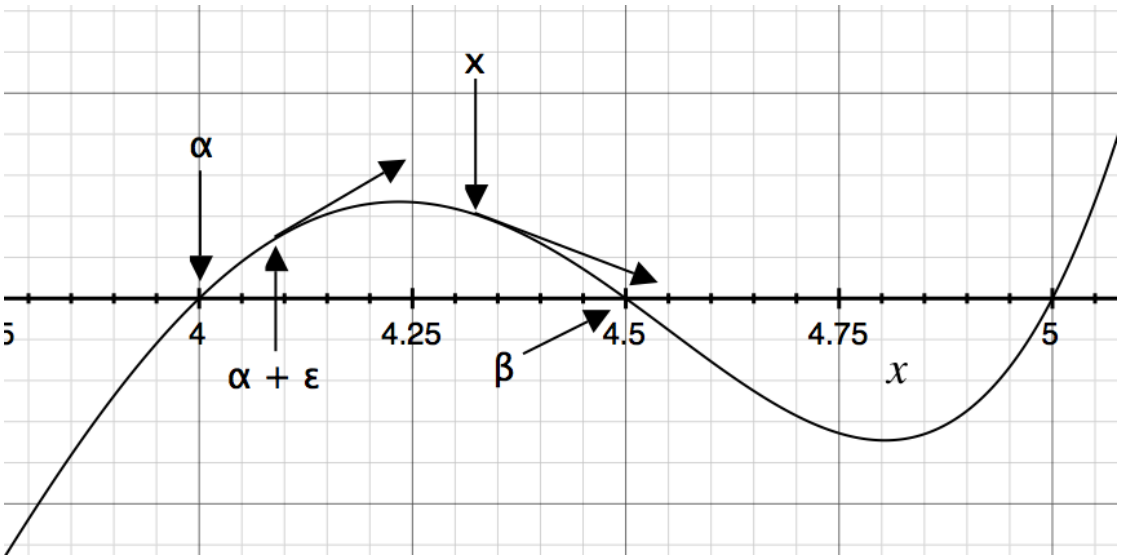


Figure 2.2: An Example of Theorem 2

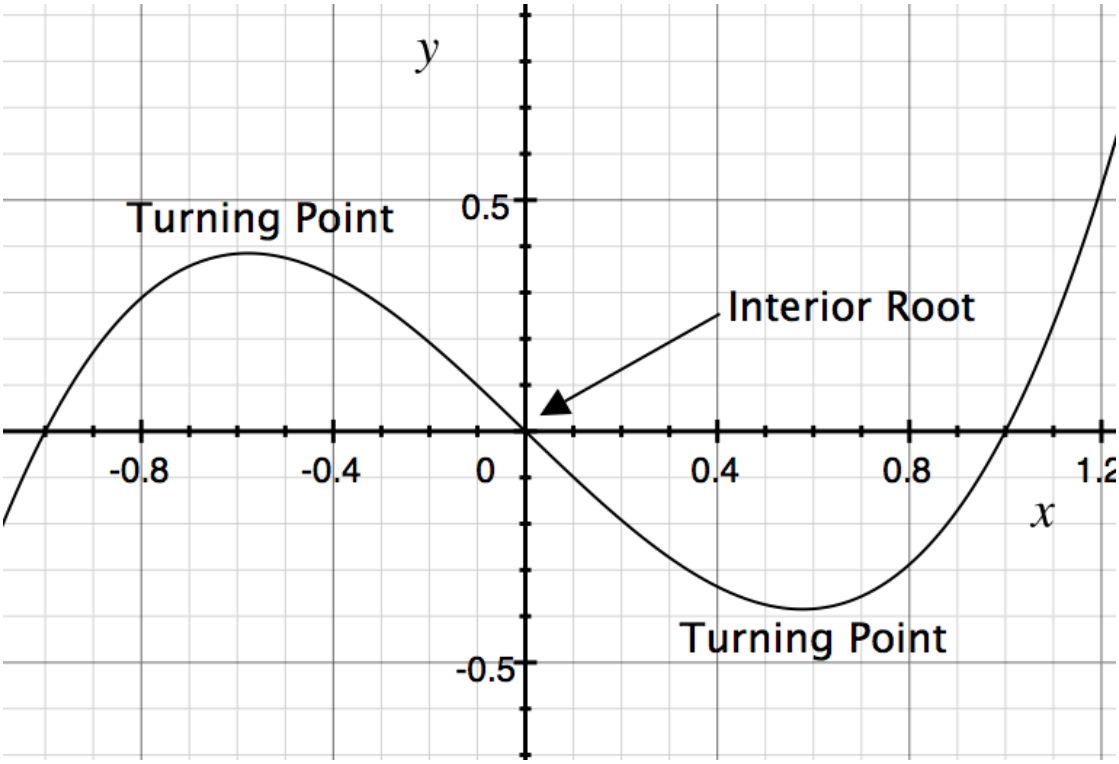
Theorem 3 [1] For a polynomial $p(x) = a_0 + a_1x + ... + a_nx^n$, if α is a root of $p(x)$ then

$$\|\alpha\| \leq \max\left(1, \sum_{i=0}^{n-1} \left\| \frac{a_i}{a_n} \right\|\right)$$

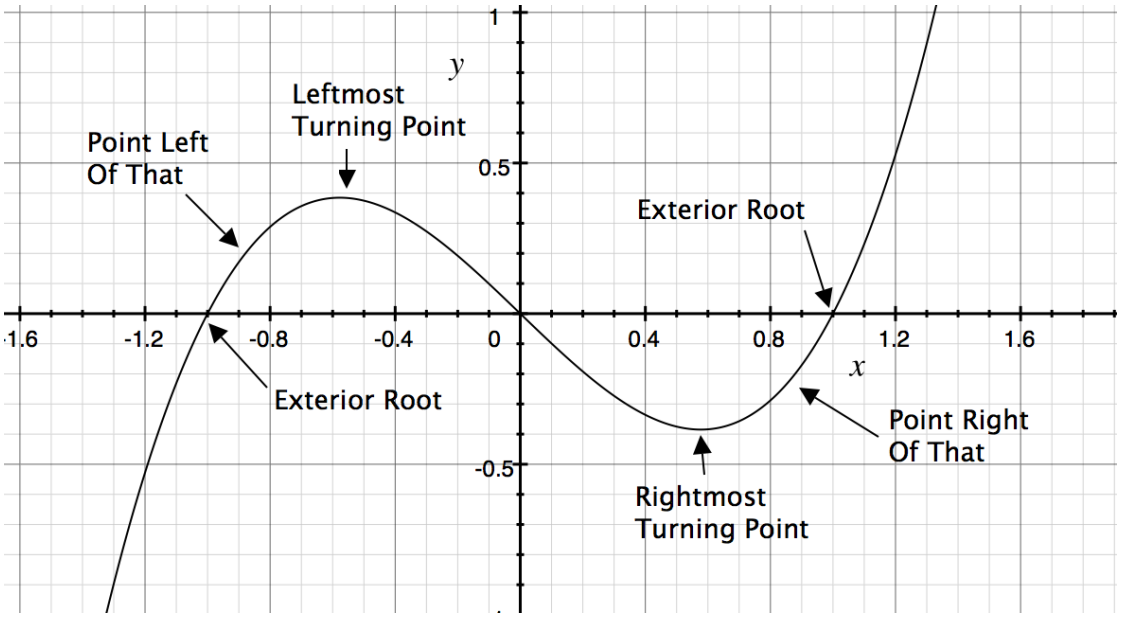
[1] Hirst, Holly P.; Macey, Wade T. (1997). "Bounding the Roots of Polynomials". The College Mathematics Journal. 28 (4): 292–295

Algorithm

To turn these theorems into an algorithm we first separate the roots of the polynomial into two categories: interior and exterior. We call a root interior if it is between two turning points, exterior otherwise. To find all the interior roots we must find all the turning points and iterate through them pairwise, checking at each step if there is a root between the pair using Theorem 1, and if there is, we can find the root using a bisection search. This method will definitely find all of the interior roots because for each pair of turning points which have at least one root between them, we find one root, but we know by Theorem 2 that there is only one root. Therefore there can't be any interior roots that we don't find.



To find the exterior roots, we can use Theorem 3. This will give us a bound on the magnitude of the exterior roots. First we look at the leftmost turning point, and we test a random point left of this to see whether it is above or below the turning point. If the the turning point is positive and the point is below the turning point, then we know the function is decreasing and so there must be an exterior root to the left. Similarly, if the turning point is negative, and the point is above it then the function is increasing so there is an exterior root to the left. We then apply the same procedure to the right side, looking at a point to the right of the rightmost turning point.



If we know these exterior roots exist we can then find them by computing a bound using Theorem 3, (and negating this if we are looking on the left) and then performing a bisection search between the outside turning point and the bound.

We can see that this algorithm is recursive, as it requires the computation of the turning points of a polynomial of degree n which requires solving a polynomial of degree $n-1$. So to ensure this terminates, the cases for $n = 0, 1, 2$ should be manually implemented using the quadratic formula etc.

2.2.11 THE INTEGRATION ALGORITHM

The integration algorithm operates much like the simplification algorithm, except only one round is applied. This round consists of eight different passes. Each of these passes handles a different class of functions. Most of these are simply direct translations of given formulas. However, I will focus on the algorithm for integrating rational functions which is significantly more complicated.

The method I employ to integrate rational functions is the *partial fraction decomposition* method. This is a naive algorithm for solving this problem - other more efficient algorithms such as the Rothstein-Trager algorithm do exist, but are more complicated to understand and implement.

The core of the partial fraction method is to express some rational function $R(x) = \frac{p(x)}{q(x)}$ as the sum of partial fractions. A partial fraction is any expression of any of the following forms:

$$\frac{a}{bx+c} \text{ or } \frac{a}{bx^2+cx+d} \text{ or } \frac{ax+b}{cx^2+dx+e}$$

For example,

$$\frac{3x^2+2x+1}{(x-1)(x^2+1)} = \frac{2}{x^2+1} + \frac{3}{x-1}$$

This is then easily integrable as formulas for the integration of all the partial fraction forms are readily available.

In order to transform a rational function into this form we must first make sure that the degree of the numerator is less than that of the denominator. If it is not, we can perform polynomial long division and obtain a polynomial, which can be integrated separately, and a new rational function which satisfies this property.

If we have $R(x) = \frac{p(x)}{q(x)}$, we can factorise the denominator to give:

$$R(x) = \frac{p(x)}{(x-r_0)(x-r_1)...(x-r_n)f(x)}$$

Where r_i are all constant and unique. Substituting in our partial fraction expression with unknown coefficients:

$$\frac{A_0}{x-r_0} + \frac{A_1}{x-r_1} + ... + \frac{A_n}{x-r_n} + \frac{g(x)}{f(x)} = \frac{p(x)}{(x-r_0)(x-r_1)...(x-r_n)f(x)}$$

If we now multiply through by $(x - r_0)$, we are left with:

$$A_0 + \frac{(x-r_0)A_1}{x-r_1} + ... + \frac{(x-r_0)A_n}{x-r_n} + \frac{(x-r_0)g(x)}{f(x)} = \frac{p(x)}{(x-r_1)...(x-r_n)f(x)}$$

Thus if we set $x = r_0$ in this expression, all the terms apart from the first term will go to zero:

$$A_0 = \frac{p(r_0)}{(r_0-r_1)...(r_0-r_n)f(r_0)}$$

Thus in order to compute the coefficient of the ith partial fraction we make a new function:

$$R_i(x) = \frac{p(x)}{(x-r_0)...(x-r_{i-1})(x-r_{i+1})...(x-r_n)f(x)}$$

And then:

$$A_i = R_i(r_i)$$

Thus it remains to compute $g(x)$. In order to do this, we multiply through by $f(x)$:

$$\frac{f(x)A_0}{x-r_0} + \frac{f(x)A_1}{x-r_1} + ... + \frac{f(x)A_n}{x-r_n} + g(x) = \frac{p(x)}{(x-r_0)(x-r_1)...(x-r_n)}$$

And so:

$$g(x) = \frac{p(x)}{(x-r_0)(x-r_1)...(x-r_n)} - f(x)\left(\frac{A_0}{x-r_0} + \frac{A_1}{x-r_1} + ... + \frac{A_n}{x-r_n}\right)$$

When this is simplified, it will then be a polynomial. (In my program this does not use the main simplifier but instead calculates the function and then manually performs polynomial long division).

This does not, however, handle the case where all of r_i are not unique, i.e there are repeated roots. In the case that we have a repeated root:

$$R(x) = \frac{p(x)}{(x-r)^k f(x)}$$

Our partial fractions expansion will contain terms for:

$$R(x) = \frac{B_1}{(x-r)} + \frac{B_2}{(x-r)^2} + ... + \frac{B_k}{(x-r)^k} + \frac{g(x)}{f(x)}$$

In order to find B_1 we can follow our earlier method. However, to find B_2 and above, we can use the following identity:

$$R_k(x) = \frac{1}{(k-1)!}\frac{\mathrm{d}^k}{\mathrm{dx}^k}\left[\frac{p(x)}{f(x)}\right]$$

$$B_k = R_k(r)$$

Thus we can use this identity to process all of the repeated roots, and then use the first method for all the normal roots. Then to get the final result we take all of our partial fractions, one for each root plus extras for repeated roots and one for $f(x)$, and we integrate them separately and add up the results.

This method, however, cannot integrate all rational functions, as my polynomial solver lacks the capability to find complex roots. The reason this is a problem is that if $f(x)$ has a degree greater than two, then we will not have a formula to integrate $\frac{g(x)}{f(x)}$. We do know that it must be the product of irreducible quadratic factors, but we have no way to factorise it as irreducible quadratics have complex roots.

2.2.12 MATRIX OPERATIONS

The main algorithm here is the Laplace algorithm for calculating determinants. This algorithm is also adapted in my program to calculate inverse matrices, and hence eigenvectors eigenvalues, and simultaneous equation solving.

To compute the determinant via Laplace's expansion of some matrix we first must define the (i,j)*th minor*. The (i,j)th minor of a matrix is the determinant of the matrix formed when the ith column and jth row are removed from the original matrix. The (i,j)th minor of a matrix A is represented a $M_{i,j}(A)$. For example,

$$M_{1,2}\begin{bmatrix}1 & 2 & 3\\ 4 & 5 & 6\\ 7 & 8 & 9\end{bmatrix} = \det\begin{bmatrix}\square & 2 & 3\\ \square & \square & \square\\ \square & 8 & 9\end{bmatrix} = \det\begin{bmatrix}2 & 3\\ 8 & 9\end{bmatrix} = -6$$

With the minor defined, we can calculate the determinant of any matrix using Laplace's expansion:

$$\det A = a_{1,1}M_{1,1}(A) - a_{1,2}M_{1,2}(A) + a_{1,3}M_{1,3}(A) - ... + a_{1,n}(-1)^{1+n}M_{1,n}(A)$$

Where $a_{1,j}$ represents the jth element along the top row of the matrix. We can immediately see this algorithm is recursive, but it always terminates as each time we compute n determinants of $(n-1)$ by $(n-1)$ matrices, so this eventually decreases to the base case of $n = 1$, since the determinant of a 1-element matrix is just the only element of the matrix. This algorithm is very inefficient with a time complexity of $O(n!)$, but for our purposes ($n < 5$ usually), this will not affect us.

With our definition of the minor we can also compute inverse matrices easily, using the cofactor formula:

$$A^{-1} = \frac{1}{\det A}\begin{bmatrix}(-1)^{1+1}M_{1,1}(A) & (-1)^{1+2}M_{1,2}(A) & ... & (-1)^{1+n}M_{1,n}(A)\\ (-1)^{2+1}M_{2,1}(A) & (-1)^{2+2}M_{2,2}(A) & ... & (-1)^{2+n}M_{2,n}(A)\\ \vdots & \vdots & \ddots & \vdots\\ (-1)^{n+1}M_{n,1}(A) & (-1)^{n+2}M_{n,2}(A) & ... & (-1)^{n+n}M_{n,n}(A)\end{bmatrix}$$

Using this we then solve simultaneous linear equations by compute the inverse coefficient matrix, find eigenvalues by taking the determinant of $M - \lambda I$ for any matrix M and then solving for λ and find eigenvectors by iterating the multiplication of a random vector by the inverse of a given matrix.

2.2.13 STATISTICAL OPERATIONS

All statistical operations are simply represented as operations on lists and values as defined by built in Haskell modules such as the Prelude, and the module Data.List. These are used to provide direct translations of all the formulas required, with the additional simple procedure that all functions read their data from a Table rather than just accept it as input.

2.2.14 GEOMETRIC OPERATIONS

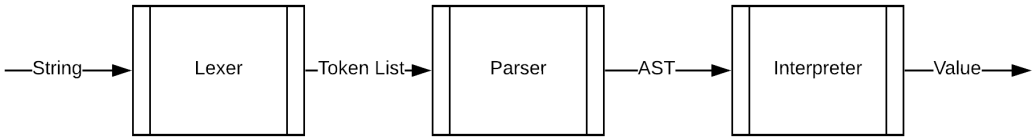
In order to compute the intersection of two shapes, we find formulas ahead of time for each of the possible intersection methods (line-line, line-circle and circle-circle) and then simply substitute in the appropriate numbers to calculate the result. Similarly, equations of circles and lines are represented by templates which are just filled in with the appropriate data and then fed to the simplifier to make them look more natural.

2.3 INTERPRET

This module contains four main components:

- The AST and Value datatypes
- The Lexer
- The Parser
- The Interpreter

This entire module is designed to turn strings of user input into output values. And each of the four modules process a bit of this conversion:



2.3.1 DATA STRUCTURES

This component contains three main data structures:

- The expression structure which is used to represent a single computation to be performed.
- The value structure represents the type that these computations output and require as input.
- The Interpret type. This is used to represent the result of a calculation which may or may not fail, and has some associated state with it.

2.3.2 INTERPRET

`Interpret a` represents a computation that produces a value of type `a` but may or may not fail but also has an associated state. In this case the associated state is the environment of variables - that is, a list connecting variable names and their corresponding values. `Interpret` is not constructed directly as a data type but is instead made from monad transformers. These are types that are predefined to add some sort of capabilities onto a given type. For example, the ExceptT monad transformer adds error handling capabilities to a given type (similar to `Compute` from earlier), and the StateT transformer adds an associated mutable state to a given type. We can combine these to form `Interpret` and that allows us to perform certain actions:

```
1 someComputation :: Int -> Interpret Int
2 someComputation = do -- We can use do-notation because
3                      -- it is defined for the monad transformers
4           state <- get -- Read the associated state
5           put $ state + 1 -- Set the associated state
6           lift $ throwError "AAAAHHH" -- Use the error handling we
                added
7           return $ 1 + 2 -- We can still perform normal computations
                too.
```

In this example, we could write `Interpret` as:

```
1 type Interpret a = ExceptT String (State Int) a
```

Since here we have a state that was an integer and errors of type String. In the real definition, the state is instead `[(String, Value)]` and the errors are a custom error type (`InterpretError`).

2.3.3 EXPRESSION

The `Expression` type represents any kind of computation that possibly be performed in my program, and each variant has fields that contain the input it needs to be processed. For example, the `Determinant` variant has one field which is the matrix to calculate the determinant of. Many of these variants are combined together into one huge type with 52 different variants, ranging from `MulE` which means multiply two expressions together, to `DefiniteIntegral` which integrates one expression between the limits specified but another two expressions, to `Value` which is a computation that always returns the specified value.

2.3.4 VALUE

This represents all possible values that a computation could take as input or produce as output. At the moment this consists of numbers (including scalars and matrices), shapes, expressions, equations, lists of other values, tables and points. This just a wrapper around all the types from different modules in the Compute component:

```
data Value = Number Number
           | Shape (Shape Number)
           | Expr (Expr String Number)
           | Eqn (Equation String Number)
           | List [Value]
           | Tbl (Table String Number)
           | Point Number Number
```

2.3.5 ALGORITHMS

2.3.6 PARSER COMBINATORS

Both the lexer and parser are based on *parser combinators*. These are functions which allow you to build a up a parser, starting from small base parsers and using other functions to combine them. To explain how these are designed, we can build up a simple system from scratch. First we must define Parser. One common definition goes like this:

```
1 data Parser c a = Parser {
2   parse :: [c] -> Either String (a, [c])
3 }
```

That is to say, a parser is some function `parse` that takes some list of input and returns either an error message, or a return value and the remaining input. Using this definition we can easily build some basic parsers:

```
1 eat :: Parser c c
2 eat = Parser $ \s -> case s of
3   [] -> Left "Unexpected EOF!"
4   (x:xs) -> Right (x, xs)
```

This is the most basic possible parser - it simply matches one item of input and returns it:

```
1 parse eat [1, 2, 3, 4] == Right (1, [2, 3, 4])
2 parse eat [] = Left "Unexpected EOF!"
```

We could also define a slightly more complicated parser:

```
1 match :: (c -> Bool) -> Parser c c
2 match pred = Parser $ \s -> case s of
3   [] -> Left "Unexpected EOF!"
4   (x:xs) -> if pred x
5               then Right (x, xs)
6               else Left "Predicate failed!"
```

This will match one item of input that satisfies a given predicate. From here we can easily make many other parsers. For example:

```
eat :: Parser c c
eat = match (const True)

exact :: Eq c => c -> Parser c c
exact = match . (==)
```

Here, `exact` is a parser that matches exactly the given element. However, we still have not come across the main power of parse combinators: that the can be combined. First let us define the function:

```
fmap :: (a -> b) -> (Parser c a -> Parser c b)
fmap f a = Parser $ \s -> case parse a s of
                            Right (v, ss) -> Right (f v, ss)
                            Left err -> Left err

(<$>) = fmap -- Just to make things look a little nicer
```

This can then be used to modify the results of parsers:

```
oneAsInt :: Parser Char Int
oneAsInt = (read . return) <$> exact '1'
```

Here, `oneAsInt` matches exactly the character '1'. This is then fed to `(read . return)` which converts it into an integer. Thus, in a sense, `<$>` behaves exactly the same way as regular `$` except it works on parsers. Let us define another two functions:

```
(>>=) :: Parser c a -> (a -> Parser c b) -> Parser c b
a >>= f = Parser $ \s -> case parse a s of
                            Right (v, ss) -> parse (f v) ss
                            Left err -> Left err

return :: a -> Parser c a
return v = Parser $ \s -> Right (v, s)
```

This first function allows to combine two parsers in a sequence:

```
1 exact12 = exact '1' >>= \value -> exact '2'
```

Where `value` is the result of the first parser. This is very useful on its own, but becomes even more useful when we pair it with do-notation. Do-notation in Haskell is a special syntax that is desugared into calls to the >>= function:

```
1 exact12 = do  -- This has the exact same underlying representation
    as above.
2   value <- exact '1'
3   exact '2'
```

This is probably the most powerful feature of parser combinators, as it allows you to combine arbitrary parsers even choose which parsers to use based on results of previous parsers. This is also useful, because Haskell provides many built in functions that are simple combinations of `(>>=)` and `return`, that simplify some expressions. For example, >> allows us to combine two parser in sequence by ignoring the results of the first parser and simply returning those of the second:

```
1 exact12 = exact '1' >> exact '2'
```

However, there is one thing missing before we can truly represent any grammar:

```
1 (<|>) :: Parser c a -> Parser c a -> Parser c b
2 a <|> b = Parser $ \s -> case parse a s of
3                            Right (v, ss) -> Right (v, ss)
4                            Left _ -> parse b s
```

This function allows us to parse, either some value, or another value. How it works is to parse one parser on the given input and if it fails, try again with the other parser. This now lets us write grammars such as:

```
1 ab_or_ba :: Parser Char Char
2 ab_or_ba = (exact 'a' >> exact 'b') <|> (exact 'b' >> exact 'a')
```

There are also many built-in functions defined in terms of <|>. The most useful one is some, which behaves exactly like you'd expect, parsing one or more copies of a parser:

```
number :: Parser Char Int
number = read <$> some digit
    where digit = exact '0' <|> exact '1' <|> exact '2' <|>
                  exact '3' <|> exact '4' <|> exact '5' <|>
                  exact '6' <|> exact '7' <|> exact '8' <|> exact '9'
```

This will parse all positive integers. From this we can see that by defining a few simple functions and a few basic parsers, complicated parsers can be easily built up. This is very similar to how the actual system is implemented in my program.

There are several alternatives to parser combinators that I could have used. Firstly, I could have used a parser generator like ANTLR or happy. These programs take a grammar specification file using BNF, and produce code to parse such inputs. The problem with this is that an extra pass would be required to take the parse tree that they provide, and rework it into my expression type. Alternatively, I could have used a pre-made parser combinator library such as Parsec. This has the advantage that the error reporting is significantly more useful and advanced. However, I could not use it as there is a major difference in Parsec in that that <|> function is not backtracking (when the first parser fails it does not try the second parser on the original input, but instead where the first parser left off), which would require a complete redesign of my grammar.

2.3.7 LEXER

The first application of these parser combinators is the lexer, where a string is turned into a list of tokens. A token is simply a small string which has been categorised to make it easier to parse later on. In my lexer, the input string is divided up into numbers, punctuation, newlines and any remaining input is left as a string. This makes it much easier to use in the main parser, because we don't have to worry about whitespace as that has already been removed, and we can easily match a certain type of input just by matching on a certain variant of the token type.

2.3.8 PARSER

The main parser is responsible for parsing all of the grammar of the natural language interface, using the tokens from the lexer. This is done in the following way:

- For each of the variants in the expression type. i.e for every computation that can be performed:
- A parser is defined to parse the format of that particular computation.
- Then a higher level parser is defined which combines all of the parsers in a given category.
- Finally all the category parsers are combined together in one giant expression parser.

There is one exception to this rule, however, which is infix expressions. Infix expressions could be written as regular recursive expression parsing rules. However, this would require the use of left recursion, which is where a recursive parser makes a recursive call *before* it parses any other input, or in code:

```
1 leftRecursive = do
2   a <- leftRecursive
3   b <- someOtherStuff
4   return $ f a b
```

The fundamental problem with parser combinators as a parsing method is that they cannot parse grammars containing left recursion. Fortunately it is easy to remove left recursion from the grammar by treating it as a special case. The algorithm used for parsing infix expressions proceeds as follows:

- Parse many expressions, separated by any kind of infix operation. For example, "1 + 2 * 3".
- Put these all into a list.
- For each operator, ordered from highest to lower precedence, do the following:
  - While there are still occurrences of the operator in the list,
  - go through and replaces one occurrence of the pattern "a, op, b" (where op is the operator) with another expression corresponding to operator. For example: replace an occurrence of "a, *, b" with `MulE a b`.
- At this point there should only one expression in the list and this is the output of the algorithm.

2.3.9 INTERPRETER

Finally, the interpreter algorithm executes a parsed expression from the parser and produces an output value. This algorithm operates using the `Interpret` data type and so has an associated state (the variable environment) and can throw errors. The way the interpreter works is simply to use pattern matching on the input expression and handle all of the variants of expression separately. For each variant execution comes in three steps:

- First, recursively execute all of the arguments to this computation, as the arguments are expressions, but our computations operate on values.
- Next, check that the values are of the correct type for this computation (e.g you can't find the eigenvalues of a circle or solve a dataset)
- Then, perform the actual computation and return the output value.

2.4 SERVER

The server is really just a simple wrapper around the interpret module, that connects it to the network. There were two major design choices to be made here:

1. How should the server connect to the client over the network?
2. How should the data being transferred to the client be represented?

To answer the first question. My initial three choices were a simple web service (HTTP REST), simple TCP sockets, or ZMQ. I decided not to use web services because while there are plenty of good web service libraries for Haskell, these all have a lot of additional overhead, both in terms of the code required and also the extra processing, for such a simple use case.

With TCP sockets, my main problem was that TCP is a stream based protocol and not a packet based protocol, so I would have to implement additional code to know when the whole of the response had been delivered. So, instead I chose ZMQ sockets.

ZMQ[2] is a messaging library that implements an interface very similar to traditional TCP sockets except that it is a packet based protocol, solving my earlier problem. Additionally, one mode of operation for ZMQ (called REQ-REP mode) requires every request made by the client to the server is followed by a response from the server - the client cannot send another request before the server has responded. This has massively simplified my implementation of the client side, as it prevents any kind of race conditions.

For the second point, I chose to use JSON, as on the Haskell side this is easy for me to serialise without the use of external libraries, and on the JavaScript side, JSON parsing is built in to the language.

[2] http://zeromq.org

2.5 CLIENT

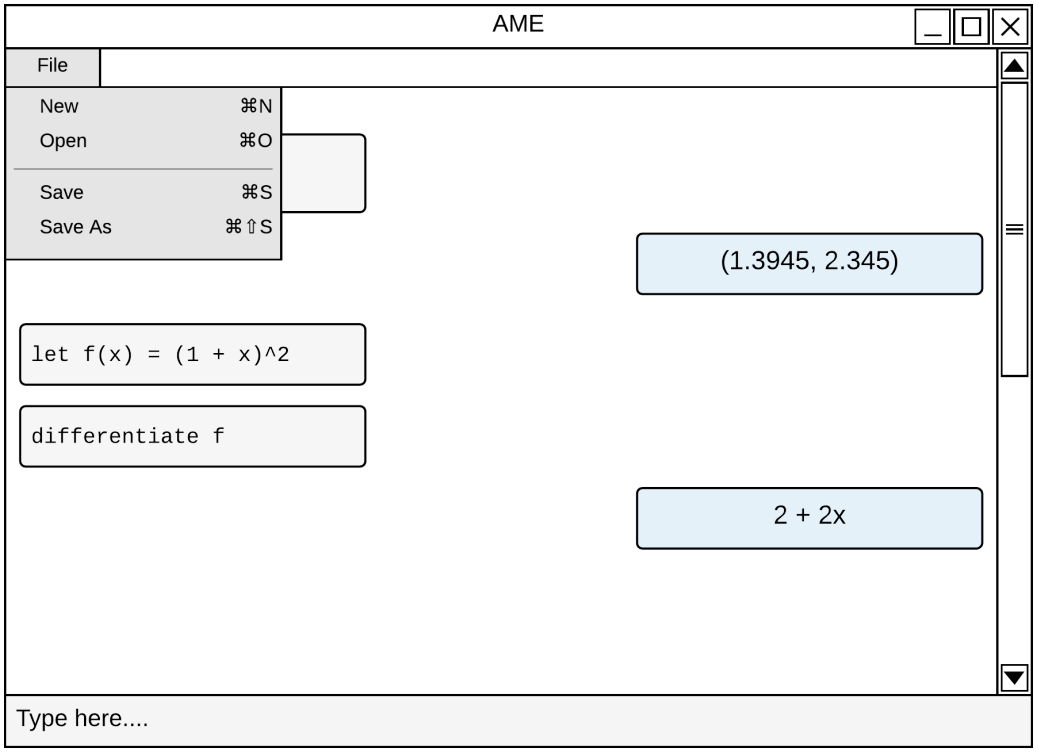
Early on in the project, I decided I wanted to use HTML and web technologies to create my GUI. This was because my language of choice, Haskell, has no good GUI libraries. Additionally, using HTML would provide a consistent user interface wherever I viewed it. I chose to use the Electron platform to host this GUI.

Electron is platform that contains a lightweight version of Google Chrome with all the tabs, search bar and other menus removed. This mean that it can be configured to show a given web page, and because the user can't navigate the browser anywhere else, this will act like a native desktop app. Electron also contains a version of NodeJS which allows JavaScript to be run outside of a web browser, and can be used to access things outside the reach of regular JavaScript such as the filesystem, and additionally there are bindings to ZMQ for NodeJS.

The advantage of using Electron here is that it can easily be used across different platforms as versions of Electron exists for all three platforms, and because the version of Chrome used is always exactly the same, the user interface will be identical on all platforms.

2.5.1 GUI DESIGN

For my GUI design, I decided to try to emulate the style of a messaging app. This would allow the layout to be familiar my target audience (A-Level students, ages 17-18). However I also added a standard menu bar with a "File" menu to allow opening and saving files. In order to make it easier to see what I had to implemented I created a mockup of the UI:



Here, the users input is displayed on the left, and the responses from the server to a given input are displayed below and to the right of the corresponding input. There is an input box at the bottom for the user to type into, and the file menu is shown at the top. Since calculation sessions may easily be larger than the screen size, I decided to add a scrollbar to allow the user to easily see any part of the session.

2.5.2 FUNCTIONALITY

Electron-based apps are built in two parts: the renderer process, which is the JavaScript running inside Google Chrome, and the main process, which runs in NodeJS. Communication between the two is achieved with a built-in inter-process communication (IPC) system.

2.5.3 MAIN PROCESS

In the main process all we do is communicate with the server process. To do this we:

- Start up an instance of the server process.
- Create a new ZMQ socket and connect it to the server
- Listen for requests from the renderer process over IPC
- When these are received:
  - Forward them on to the server
  - Wait for the response, and send it back to the renderer via IPC.

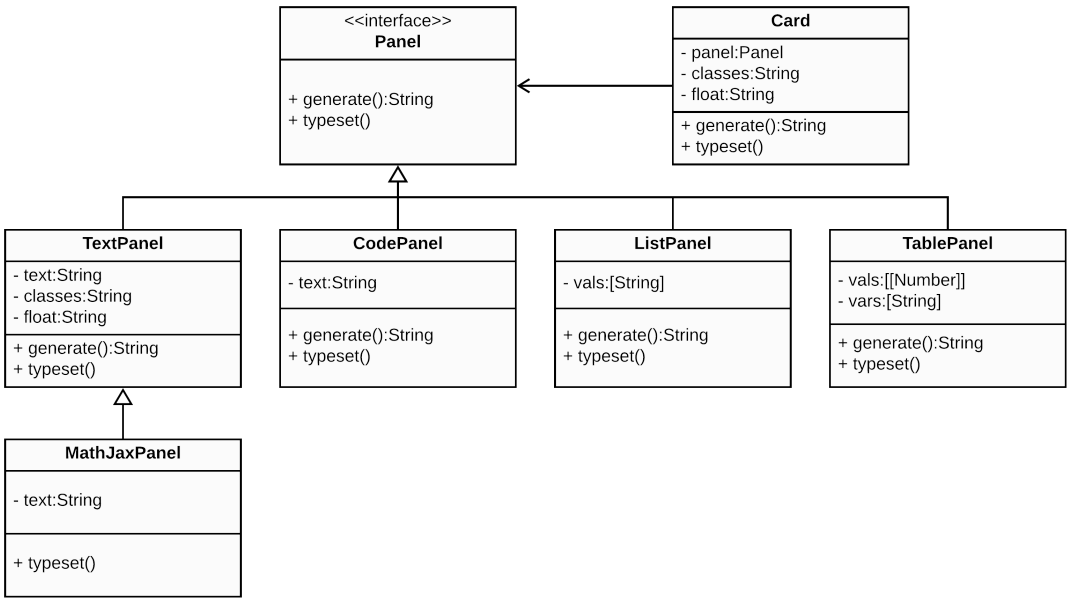
2.5.4 RENDERER PROCESS

This process manages all the messages on the screen and also handles displaying the servers responses. A log of all user input should also be kept to allow saving and loading of sessions. This process executes the following steps in a loop:

- Wait for the user to input something.
- Send it to the main process through IPC.
- Add a message to the screen with the users input.
- Wait for a response from the server.
- When it comes, parse the response and add the appropriate type of message to the screen.

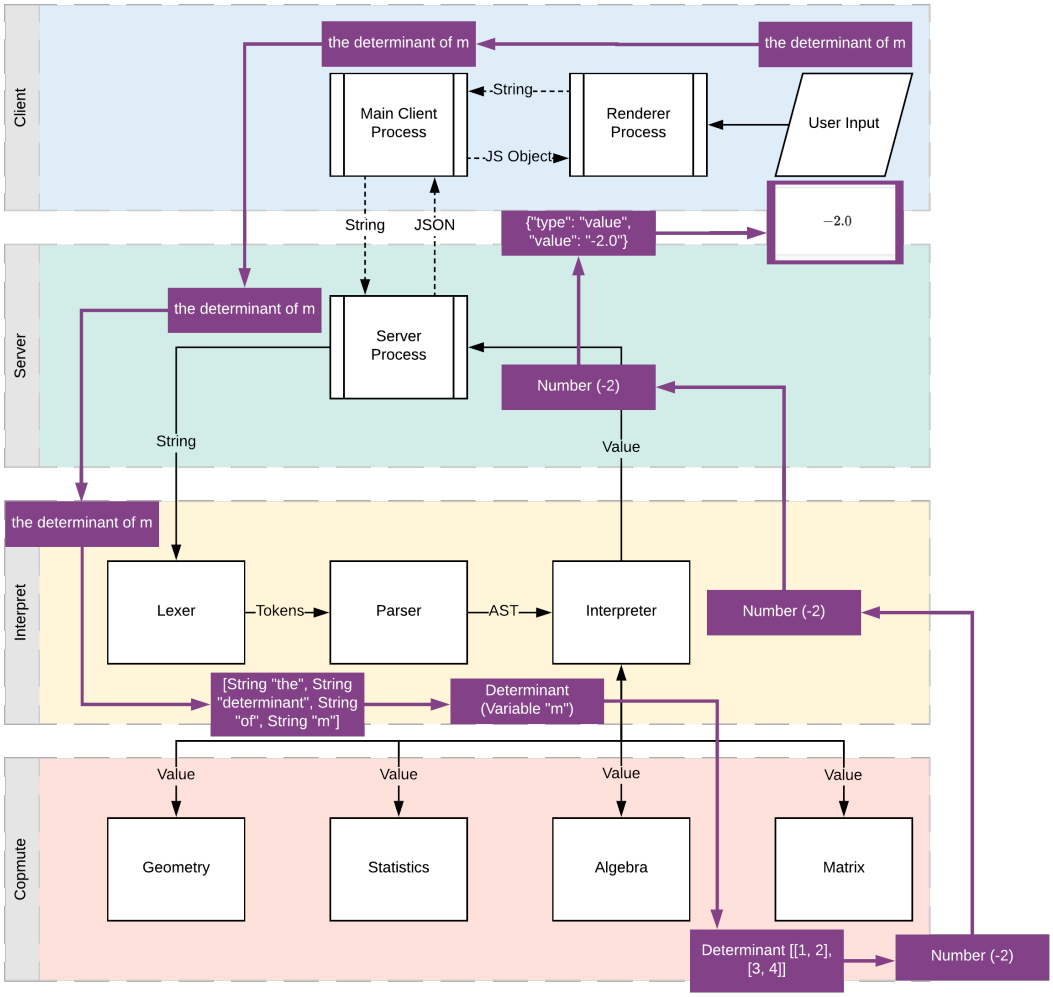
In order to represent all the different kinds of messages and to display them in an appealing way, I have a different kind of message for user input, as well as errors, and every different type of value that the server could respond with (shapes, datasets, equations etc.).

To represent this I defined a class called a Card which has two methods, one for generating the HTML to be displayed by that message, and one to do any post-processing once the message has been added to the screen. Inside the Card is a Panel. There are different subclasses of Panel each displaying different kinds of data. By varying the arguments to Card as well as the Panel type used, different kinds of messages can be created. This is summarised in the UML diagram below:



2.6 WALKTHROUGH OF EXECUTION

In order to further explain how all these pieces fit together, I have created a diagram showing all the intermediate stages of the execution of the input `the determinant of m`, in an environment where the variable `m` is set to the matrix $\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$. This is overlayed onto of the overview diagram from the start of this section:



3 TECHNICAL SOLUTION

3.1 SYNTAX REFERENCE

A reference to the syntax used in the natural language interface:

- Variables: Any string of letters, numbers or underscores (as long as numbers are not the first character). E.g: `x`, `X123`, `hello_world` are all valid variable names. A variable is represented simply by its name.
- Any number is represented in ordinary decimal format. E.g `1.0`, `-2`, `3.345` are all valid numbers.
- A vector is represented as a list of numbers, separated by commas. E.g `[1, 2, 3, 4]`.
- A matrix is represented as a list of row vectors. E.g `[[1, 2], [3, 4]]`
- `let <name> be a new variable <name>` will create an unbound variable called `<name>`.
- `the equation of <shape>` will find the equation of the given shape.
- `the intersection of <shape> and <shape>` will find the intersection of two shapes.
- `the line connecting <point> and <point>` will find the line between two points.
- `the line passing though <point> with gradient <number>` will find the line with the given properties.

- `the line with intercept <number> and gradient <number>` will find the line with the given intercept and gradient.
- `the circle with centre <point> and radius <number>` will create a circle with the given centre and radius
- `the circle with centre <point> passing through <point>` will create the circle with the given properties.
- `<expr> = <expr>` creates an equation equating the two expressions.
- `<expr> * <expr>`, `<expr> / <expr>`, `<expr> + <expr>`, `<expr> - <expr>` represent their respective operations as you would expect.
- `<expr> ^ <expr>` represents exponentiation.
- `sin(<expr>)`, `cos(<expr>)`, `tan(<expr>)`, `asin(<expr>)`, `acos(<expr>)`, `atan(<expr>)`, `log(<expr>)`, `exp(<expr>)`, `sqrt(<expr>)`, represent their respective operations as you would expect.
- `|<expr>|` represents the absolute value of an expression.
- `substitute <name> for <expr> in <expr>` substitutes an unbound variable for the given value in the given expression.
- `evaluate <expr>` evaluates an expression with no variables.
- `solve <equation>` solves the given equation.
- `solve { <equation list> }` where `<equation list>` is a list of equations, separated by commas. This will solve the given system of equations.
- `integrate <expr>` integrates the given expression
- `the integral of <expr> between <number> and <number` calculates the definite integral of the expression between the given limits.
- `differentiate <expr>` calculates the derivative of the expression.
- `the table where <table list>` where `<table list>` is a list of `<name> = <vector>` separated by semi-colons. E.g `the table where x = [1, 2, 3, 4]; y = [1, 2, 3, 4]`
- `the mean of <name> in <table>` represents the mean of the given variable in the given table.

- Syntax for all the other single variable statistical functions is the same, except that the starting text is: `the sample mean of`, `the median of`, `the mode of`, `the variance of`, `the standard deviation of`, `the sample variance of`, `the sample standard deviation of`.
- `the chi squared value of <name> and <name> in <table>` represents the chi-squared value of the two variables in the given table.
- `the Pearson correlation of <name> and <name> in <table>` represents the product-moment correlation coefficient of the two variables in the given table.
- `the Spearman correlation of <name> and <name> in <table>` represents the Spearman correlation coefficient of the two variables in the given table.
- `the dot product of <vector> and <vector>` computes the dot product of two vectors.
- `the magnitude of <vector>` calculates the magnitude of the given vector.
- `determinant of <matrix>` calculates the determinant of the given matrix.
- `the inverse of <matrix>` gives the inverse of the matrix.
- `the angle between <vector> and <vector>` this gives the angle between the two vectors.
- `the characteristic equation of <matrix>` gives the characteristic equation of the given matrix.
- `eigenvalues of <matrix>` calculates the eigenvalue of the matrix.
- `eigenvectors of <matrix>` calculates the eigenvectors of the matrix.
- `let <name> = <expr>` assigns `<expr>` to the name `<name>`
- `let (<name>, <name>, ...) = <expr>`. If ‘ is a list this will assign a different element of the list to each variable name.
- `let <name>(<args>) = <expr>`, where `<args>` is a list of variable names separated by commas. This will create new variables for each of the arguments of the function, and then assigns the expression to the function name variable.

3.2 GUIDE TO COMPILING THE CODE

Compiling this project is not easy, and so if you are using macOS (or can get access to a macOS platform) it is vastly easier to simply use the pre-built app I have provided.

Instructions are given here for compiling on Ubuntu and macOS. For Windows, the easiest method is via Windows Subsystem for Linux. To do this, you must have the latest Windows 10 version, and then simply download the Ubuntu app from the Windows store and follow the instructions given in the Ubuntu section.

All of the following is given as commands to run in a terminal. This can be found on macOS as Terminal in the the Utilities folder in your applications. On Ubuntu a terminal can be opened by searching for terminal in the sidebar. On Windows, this can be found as the Ubuntu app in the start menu.

3.2.1 INSTALLING PREREQUISITES

MACOS

The brew package manager is required. This can be obtained from *http://brew.sh*. With brew installed, run:

```
brew install node zmq ghc python3 git

pip install pyzmq

npm install -g electron

cabal install transformers mtl monad-loops
```

UBUNTU

```
sudo add-apt-repository ppa:chris-lea/zeromq

sudo apt-get update

sudo apt-get install nodejs zeromq3 python3 ghc git

pip install pyzmq

npm install -g electron

cabal install transformers mtl monad-loops
```

3.2.2 BUILDING THE SERVER

From here, you can put all of the Haskell code into a folder, with the directory structure as specified in the filenames, and run the following command in the terminal, from the folder with the code:

```
ghc Main.hs
```

This should produce an executable named Main which is the server.

3.2.3 BUILDING THE CLIENT

Create a new folder, start a terminal in it and run:

```
git clone https://github.com/electron/electron-quick-start

cd electron-quick-start
```

Then, find the folder you created, and within it there should be folder named electron-quick-start. Copy the index.html, main.css, main.js and renderer.js files into this folder, replacing the files that are already there. Then, also copy the server executable from the last step into this folder. Then from the terminal window, run

```
npm install zeromq

npm rebuild zeromq --runtime=electron --target=1.7.1

npm start
```

And the client should launch. Packaging this into a Windows exe or Mac app is significantly more complicated and is not covered here.

3.2.4 RUNNING THE AUTOMATED TESTS

Copy the test code (given in the testing section) into a file named test.py. After building the server, copy the executable into the folder where the test program is. Then, from a terminal, run:

```
python3 test.py
```

3.3 GUIDE TO READING THE CODE

The structure of the code is very similar to the structure outlined in the design section, except that it has been slightly more split up. The different files correspond to the following sections, as shown in the design overview diagram:

- Compute/Matrix: AME/Compute/Matrix.hs, AME/Compute/Matrix/Base.hs
- Compute/Algebra: AME/Compute/Expr.hs, AME/Compute/Error.hs, AME/-Compute/Solver.hs, AME/Compute/Simplify.hs, AME/Compute/Solver/Polynomial.hs, AME/Compute/Solver/Newton.hs
- Compute/Calculus: AME/Compute/Calculus/Derivatives.hs, AME/Compute/-Calculus/Integrals.hs
- Compute/Statistics: AME/Compute/Statistics.hs
- Compute/Geometry: AME/Compute/Geometry.hs
- Interpret/Lexer: AME/Parser/Lexer.hs AME/Parser/Combinators.hs
- Interpret/Parser: AME/Parser.hs AME/Parser/Combinators.hs
- Interpret/Interpreter: AME/Interpret.hs
- Server: Main.hs
- Client/Main Process: main.js
- Client/Renderer Process: renderer.js, index.html, main.css

For each component, I would recommend a different order of code files to read:

- For Compute, the main algorithms are in the AME/Compute/Simplify.hs and AME/Compute/Calculus/Integrals.hs files, although I would recommend reading AME/Compute/Expr.hs and AME/Compute/Error.hs first as they provide the background for the types used in the more complicated modules.
- For Interpret and Server, I would recommend starting with Main.hs to gain an overview of the system, followed by AME/Parser/Combinators.hs to see the building blocks used in AME/Parser.hs, and then AME/Interpret.hs
- For the Client, I recommend having main.js and renderer.js open at the same time, as it shows more clearly the information flow between the two processes.

4 TESTING

In order to properly test this program, manual testing alone cannot be used as there are simply too many possible edge cases and different values to test for all of the required calculations. Therefore, a large proportion of the tests for the required computations are instead performed automatically: I have written a program to generate a large number of random test cases for each computation (currently configured at 1000 tests each). The program then interfaces directly with the server and performs each of the random tests, comparing the result to the expected result which is calculated using Pythons numpy library, a widely used library for numerical computations.

In order for automated testing to be effective, limits on the acceptable success rate for a given category of tests need to be defined. For my program, I am defining that for Category 1 computations, the success rate should be at least 99% for the test to be considered to have passed, and for Category 2 computations, it should be at least 80%. Since the computation input happens via the regular server interface, the automated tests not only verify the correctness of the calculation but that the parser correctly parses the expression.

Additionally, in order to make this program easier to test, each of the four components will be tested separately, and then even if a failure occurs in one component, the other module tests will not be unfairly biased by that result (e.g, if an error message shows up incorrectly because of a bug in the Compute component, this will not cause a GUI test to fail).

4.1 TEST PLAN

Test Name	*Objectives*	*Reference*	*Standard*	*Status*
Arithmetic Expression Evaluation	2.a, 2.b	Automated	Category 1	Passed (100%)
Polynomial Solving	2.a, 2.b	Automated	Category 1	Passed (99.3%)
Simultaneous Equations	2.a, 2.b	Automated	Category 1	Passed (99.8%)
Polynomial Differentiation	2.a, 2.b	Automated	Category 1	Passed (99.5%)
General Differentiation	2.a, 2.b, 3.b	Test 1	Category 1	Passed
Polynomial Integration	2.a, 2.b	Automated	Category 1	Passed (99.5%)
Rational Function Integrals	2.a, 2.b, 3.b	Test 2	Category 1	Passed
Matrix Inversion	2.a, 2.b, 3.b	Automated	Category 1	Passed (99.4%)
Matrix Determinants	2.a, 2.b, 3.b	Automated	Category 1	Passed (100%)
Matrix Multiplication	2.a, 2.b, 3.b	Automated	Category 1	Passed (100%)
Matrix Addition	2.a, 2.b, 3.c	Test 3	Category 1	Passed
Circle-Circle Intersection	2.a, 2.b, 1.b	Test 4	Category 1	Passed
Circle-Line Intersection	2.a, 2.b, 1.b	Test 5	Category 1	Passed
Line-Line Intersection	2.a, 2.b, 1.b	Test 6	Category 1	Passed
Circle Equations	2.a, 2.b, 3.c	Test 7	Category 1	Passed
Line Equations	2.a, 2.b, 3.c	Test 8	Category 1	Passed
Averages (Mean, Median, Mode)	2.a, 2.b, 3.b	Automated	Category 1	Passed (100%)
Spread (Variance, etc.)	2.a, 2.b, 3.b	Automated	Category 1	Passed (100%)
Correlation (Chi Squared, Pearson-/Spearman)	2.a, 2.b, 3.b	Automated	Category 1	Passed (100%)

Equation Solving	2.a, 2.c, 3.a	Test 9	Category 2	Passed
General Integration	2.a, 2.c	Test 10	Category 2	Passed
Matrix Eigenvalues	2.a, 2.c	Test 11	Category 2	Passed
Matrix Eigenvectors	2.a, 2.c	Test 12	Category 2	Passed
Expression Substitution	2.a, 2.d	Automated	Category 2	Passed (100%)
Save As Operation	1.a 1.c	Test 13	Core Objective	Passed
Save Operation	1.a	Test 14	Core Objective	Passed
Open Operation	1.a	Test 15	Core Objective	Passed
New Operation	1.a	Test 16	Core Objective	Passed
Syntax Errors	3.e	Test 17	Core Objective	Passed
Calculation Errors	2.a, 2.b, 1.b	Test 18	Core Objective	Passed
Run on macOS	4.a	All Tests	Core Objective	Passed
Run on Windows	4.a	N/A	Core Objective	Untestable
Run on Linux	4.a	N/A	Core Objective	Untestable
Consistent UI	4.b	N/A	Core Objective	Untestable

4.2 AUTOMATED TESTING

The code for the automated testing is available in Appendix A.

4.2.1 TEST OUTPUT

```
Testing:  Matrix Determinants
Generated 200 tests.
Generated 400 tests.
Generated 600 tests.
Generated 800 tests.
Generated 1000 tests.
Passed 1000 tests.
Overall success rate:  100.0%
Testing:  Matrix Multiplication
Generated 200 tests.
Generated 400 tests.
Generated 600 tests.
Generated 800 tests.
Generated 1000 tests.
Passed 1000 tests.
Overall success rate:  100.0%
Testing:  Matrix Inverses
Generated 200 tests.
Failed test:  [[6, 4, 5, 8], [4, 2, 7, 5], [4, 3, 1,
6], [8, 6, 7, 7]]
Generated 400 tests.
Failed test:  [[2, 4], [3, 6]]
Failed test:  [[5, 5], [6, 6]]
Generated 600 tests.
Failed test:  [[8, 5, 8, 5], [4, 8, 3, 8], [3, 4, 8,
4], [7, 4, 5, 4]]
Failed test:  [[7, 7], [5, 5]]
Generated 800 tests.
Failed test:  [[8, 8, 5], [9, 9, 3], [8, 8, 4]]
Generated 1000 tests.
Passed 994 tests.
Failed 6 tests.
Overall success rate:  99.4%
```

```
Testing:  Linear Simultaneous Equations
Generated 200 tests.
Generated 400 tests.
Failed test:  [[[4, 6], [6, 9]], [5, 4]]
Failed test:  [[[9, 6], [3, 2]], [4, 3]]
Generated 600 tests.
Generated 800 tests.
Generated 1000 tests.
Passed 998 tests.
Failed 2 tests.
Overall success rate:  99.8%
Testing:  Solver
Failed test:  [23]
Failed test:  [44, 53, 81]
Generated 200 tests.
Failed test:  [23, 42, 51]
Generated 400 tests.
Failed test:  [51, 82, 86]
Failed test:  [77, 87]
Generated 600 tests.
Failed test:  [28, 34, 95]
Generated 800 tests.
Failed test:  [42, 74]
Generated 1000 tests.
Passed 993 tests.
Failed 7 tests.
Overall success rate:  99.3%
Testing:  Evaluation & Substitution
Generated 200 tests.
Generated 400 tests.
Generated 600 tests.
Generated 800 tests.
Generated 1000 tests.
Passed 1000 tests.
Overall success rate:  100.0%
Testing:  Integration & Differentiation
Failed test:  [0, 86, 90, 90]
Generated 200 tests.
Failed test:  [0, 81, 84, 90]
```

```
Generated 400 tests.
Failed test:  [0, 84, 88, 96]
Generated 600 tests.
Failed test:  [0, 34, 34, 38]
Generated 800 tests.
Failed test:  [0, 80, 85, 90]
Generated 1000 tests.
Passed 995 tests.
Failed 5 tests.
Overall success rate:  99.5%
Testing:  Mean, Median & Mode
Generated 200 tests.
Generated 400 tests.
Generated 600 tests.
Generated 800 tests.
Generated 1000 tests.
Passed 1000 tests.
Overall success rate:  100.0%
Testing:  (Sample) Variance & Standard Deviation
Generated 200 tests.
Generated 400 tests.
Generated 600 tests.
Generated 800 tests.
Generated 1000 tests.
Passed 1000 tests.
Overall success rate:  100.0%
Testing:  Chi Squared, Pearson/Spearman Correlation
Generated 200 tests.
Generated 400 tests.
Generated 600 tests.
Generated 800 tests.
Generated 1000 tests.
Passed 1000 tests.
Overall success rate:  100.0%
Testing:  Sums & Products
Generated 200 tests.
Generated 400 tests.
Generated 600 tests.
Generated 800 tests.
```

```
Generated 1000 tests.
Passed 1000 tests.
Overall success rate:  100.0%
Testing:  Subtraction, Exponentiation & Division
Generated 200 tests.
Generated 400 tests.
Generated 600 tests.
Generated 800 tests.
Generated 1000 tests.
Passed 1000 tests.
Overall success rate:  100.0%
Testing:  Sin, Cos & Tan
Generated 200 tests.
Generated 400 tests.
Generated 600 tests.
Generated 800 tests.
Generated 1000 tests.
Passed 1000 tests.
Overall success rate:  100.0%
Testing:  Inverse Sin, Cos & Tan
Generated 200 tests.
Generated 400 tests.
Generated 600 tests.
Generated 800 tests.
Generated 1000 tests.
Passed 1000 tests.
Overall success rate:  100.0%
Testing:  Log & Sqrt
Generated 200 tests.
Generated 400 tests.
Generated 600 tests.
Generated 800 tests.
Generated 1000 tests.
Passed 1000 tests.
Overall success rate:  100.0%
Testing:  Abs & Exp
Generated 200 tests.
Generated 400 tests.
Generated 600 tests.
```

```
Generated 800 tests.
Generated 1000 tests.
Passed 1000 tests.
Overall success rate:  100.0%
Passed 15980 tests total.
Failed 20 tests total.
Overall success rate:  99.875%
```

4.3 MANUAL TESTING

4.3.1 TEST 1

We will test differentiation on three different elementary functions, one involving exponentials and logs, one powers and fractions and one trigonometry. The results will be compared to the equivalent computation on Wolfram Alpha. This test also tests that the natural language input for differentiation expressions is parsed correctly, and that arithmetic functions like $\sin$ and $\cos$ are parsed correctly.

CASE 1

AME:

```
differentiate sin(2*x^2) + 2*cos(x)
```

$$4.0 \cdot x \cdot \cos\left(2.0 \cdot x^{2.0}\right) + -2.0 \cdot \sin\left(x\right)$$

Wolfram Alpha:

Derivative:

$$\frac{d}{dx}(\sin(2\,x^2) + 2\,\cos(x)) = 4\,x\,\cos(2\,x^2) - 2\,\sin(x)$$

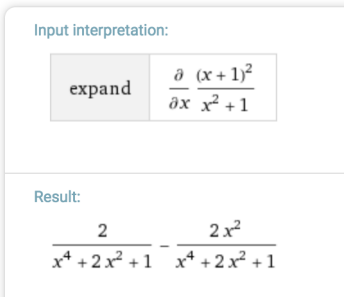
CASE 2

AME:

```
differentiate (x + 1)^2/(x^2 + 1)
```

$$\frac{2.0 + -2.0 \cdot x^{2.0}}{1.0 + 2.0 \cdot x^{2.0} + x^{4.0}}$$

Wolfram Alpha:



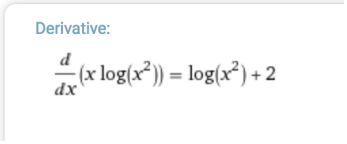
CASE 3

AME:

```
differentiate x*log(x^2)
```

$$2.0 + \log\left(x^{2.0}\right)$$

Wolfram Alpha:



4.3.2 TEST 2

Three different rational functions, one with only linear factors, one with a quadratic factor and one with a repeated root, will be integrated between 0 and 1, and the result will be compared to the same calculation on Wolfram Alpha. This test also tests that the natural language input for integration expressions is parsed correctly.

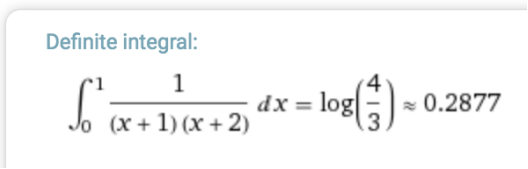
CASE 1

AME:

```
integrate 1/((x + 1) * (x + 2)) between 0 and 1
```

$$0.28768$$

Wolfram Alpha:



CASE 2

AME:

```
integrate 1/((x^2 + 2)) between 0 and 1
```

$$0.43521$$

Wolfram Alpha:

Definite integral:

$$\int_0^1 \frac{1}{x^2+2}\,dx = \frac{\tan^{-1}\left(\frac{1}{\sqrt{2}}\right)}{\sqrt{2}} \approx 0.43521$$

CASE 3

AME:

```
integrate 1/((x + 0.5)^2) between 0 and 1
```

1.33333

Wolfram Alpha:

Definite integral:

$$\int_0^1 \frac{1}{(x+0.5)^2}\,dx = 1.33333$$

4.3.3 TEST 3

One matrix addition calculation and one matrix subtraction calculation will be performed. The expected outcomes will be calculated by hand and compared with the observed result. This test also checks that infix expressions are parsed correctly.

CASE 1

Expected:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} + \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 1+1 & 2+2 \\ 3+3 & 4+4 \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 6 & 8 \end{bmatrix}$$

Observed:

```
[[1, 2], [3, 4]] + [[1, 2], [3, 4]]
```

$$\begin{bmatrix} 2.0 & 4.0 \\ 6.0 & 8.0 \end{bmatrix}$$

CASE 2

Expected:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} - \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} 0-0 & 1-(-1) \\ 1-(-1) & 0-0 \end{bmatrix} = \begin{bmatrix} 0 & 2 \\ 2 & 0 \end{bmatrix}$$

Observed:

```
[[0, 1], [1, 0]] - [[0, -1], [-1, 0]]
```

$$\begin{bmatrix} 0.0 & 2.0 \\ 2.0 & 0.0 \end{bmatrix}$$

4.3.4 TEST 4

Three different circle-circle intersection calculations will be tested, for each of the three different cases: one where there are two intersection points, one where there is one intersection point and one where there are no intersection points. These will be compared against the same calculation on Wolfram Alpha.

CASE 1

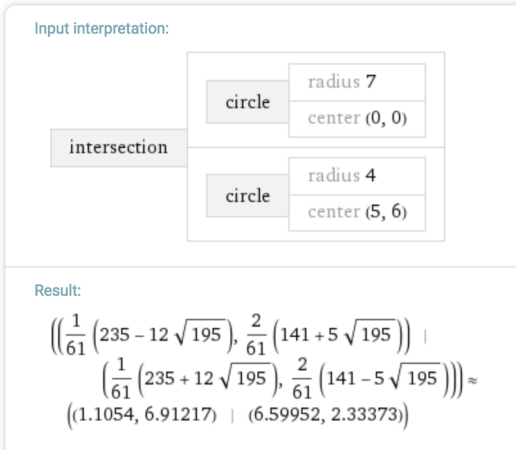
AME:

```
the intersection of the circle with radius 4 and
center (5, 6) and the circle with radius 7 and
center (0, 0)
```

$$(6.599533590711134, 2.3337220077407217)$$

$$(1.1053954713978804, 6.912170440501766)$$

Wolfram Alpha:



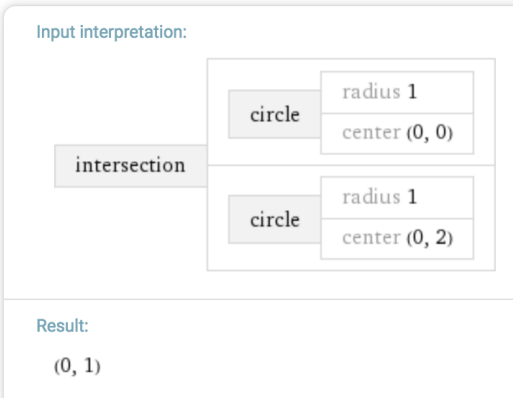
CASE 2

AME:

```
the intersection of the circle with radius 1 and
center (0, 0) and the circle with radius 1 and
center (0, 2)
```

$(-0.0, 1.0)$

Wolfram Alpha:

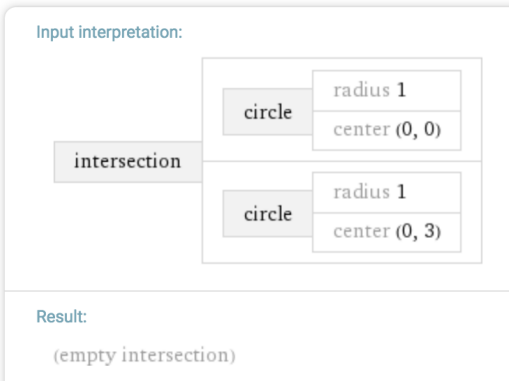


CASE 3

AME:

```
the intersection of the circle with radius 1 and
center (0, 0) and the circle with radius 1 and
center (0, 3)
```

Wolfram Alpha:

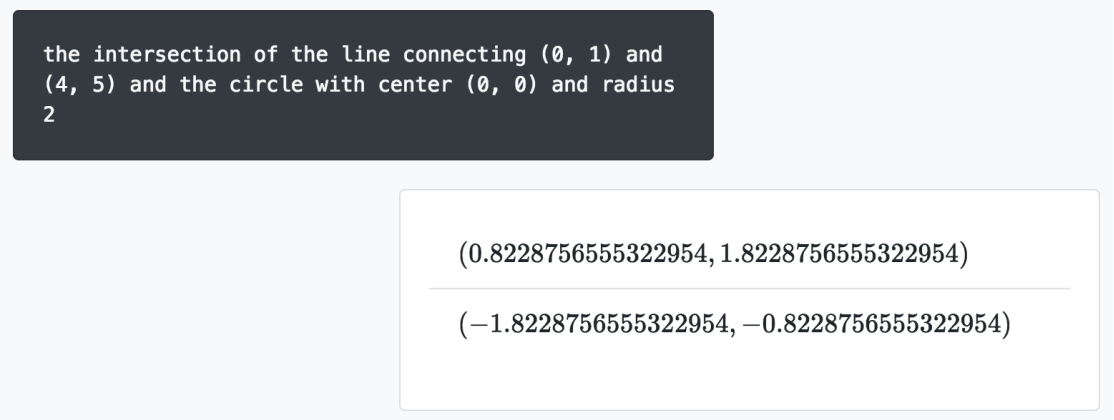

Input interpretation:
intersection
circle
radius 1
center (0, 0)
circle
radius 1
center (0, 3)
Result:
(empty intersection)

4.3.5 TEST 5

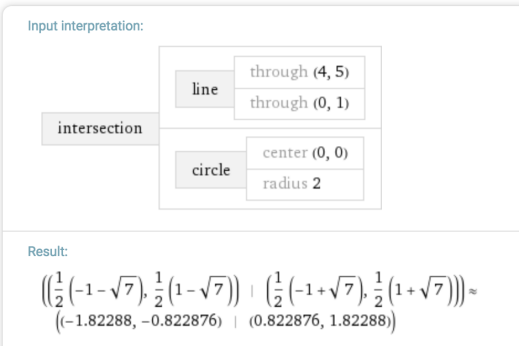
Three different line-circle intersection calculations will be tested, for each of the three different cases: one where there are two intersection points, one where there is one intersection point and one where there are no intersection points. These will be compared against the same calculation on Wolfram Alpha.

CASE 1

AME:



Wolfram Alpha:



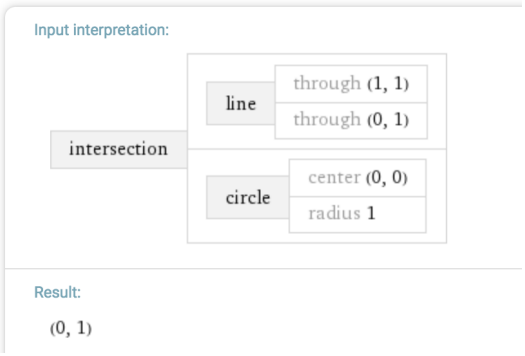
CASE 2

AME:

```
the intersection of the line connecting (1, 1) and
(0, 1) and the circle with center (0, 0) and radius
1
```

$(0.0, 1.0)$

Wolfram Alpha:

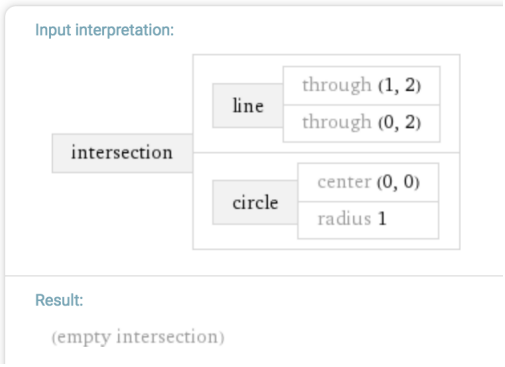


CASE 3

AME:

```
the intersection of the line connecting (1, 2) and
(0, 2) and the circle with center (0, 0) and radius
1
```

Wolfram Alpha:


Input interpretation:
intersection
line
through (1, 2)
through (0, 2)
circle
center (0, 0)
radius 1
Result:
(empty intersection)

4.3.6 TEST 6

Two different line-line intersection calculations will be tested, for each of the two different cases: one where there is one intersection point and one where there are no intersection points. These will be compared against the same calculation on Wolfram Alpha. This test also verifies that the natural language parser for line expressions works correctly.

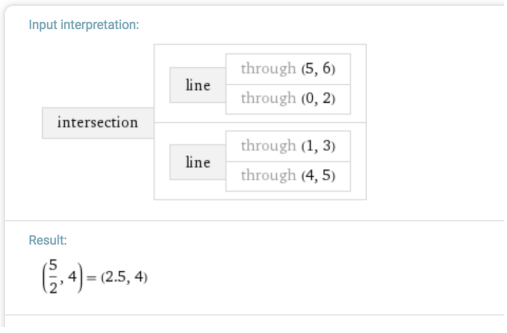
CASE 1

AME:

```
the intersection of the line connecting (5, 6) and
(0, 2) and the line connecting (1, 3) and (4, 5)
```

$$(2.4999999999999996, 3.9999999999999996)$$

Wolfram Alpha:

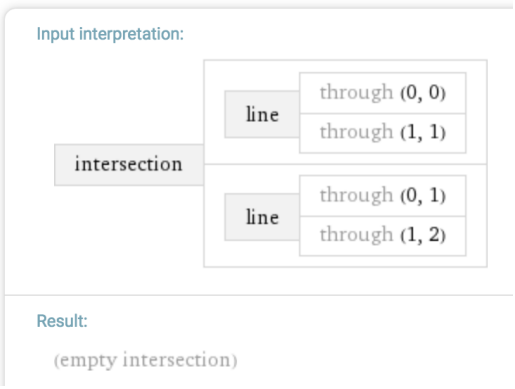


CASE 2

AME:

```
intersection of the line connecting (0, 0) and (1,
1) and the line connecting (0, 1) and (1, 2)
```

Wolfram Alpha:



4.3.7 TEST 7

The equations of two different circles will be calculated and compared to the same calculation on Wolfram Alpha. This test also tests whether the natural language parser for circle definitions is correct.

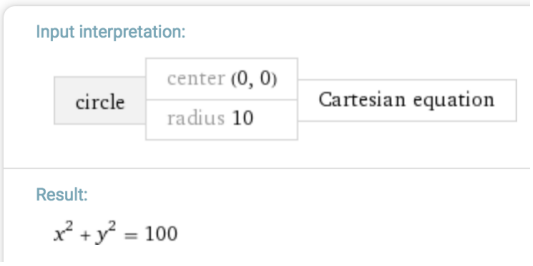
CASE 1

AME:

```
the equation of the circle with radius 10 and
center (0, 0)
```

$$x^{2.0} + y^{2.0} = 100.0$$

Wolfram Alpha:



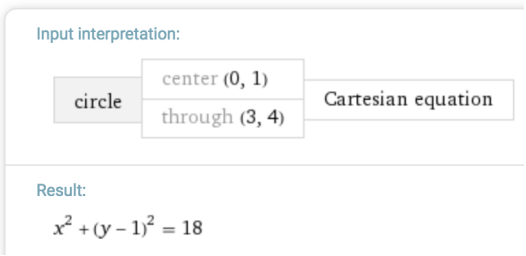
CASE 2

AME:

```
the equation of the circle with center (0, 1) and
point (3, 4)
```

$$1.0 + x^{2.0} + y^{2.0} + -2.0 \cdot y = 18.0$$

Wolfram Alpha:



4.3.8 TEST 8

The equations of three different lines will be calculated and compared to the same calculation on Wolfram Alpha. This test also tests whether the natural language parser for line definitions is correct.

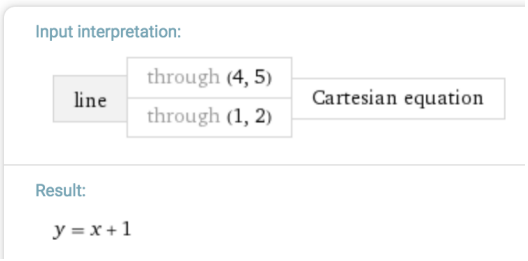
CASE 1

AME:

```
equation of the line connecting (4, 5) and (1, 2)
```

$$y = 1.0 + x$$

Wolfram Alpha:



CASE 2

AME:

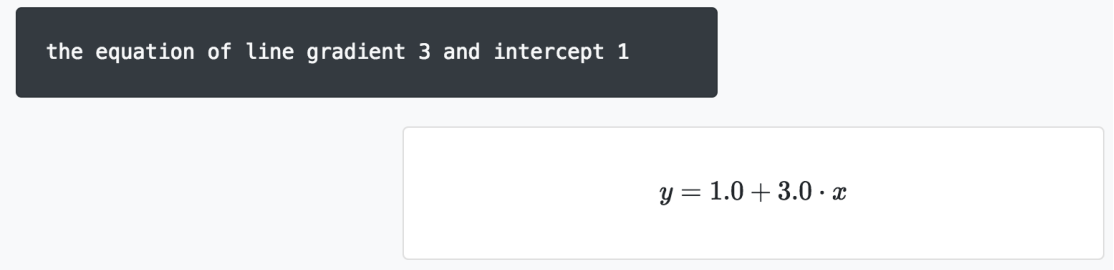
```
equation of the line passing through (1, 2) with
gradient 3
```

$$y = -1.0 + 3.0 \cdot x$$

Wolfram Alpha:

CASE 3

AME:



Wolfram Alpha:

4.3.9 TEST 9

Two different non-polynomial equations will be solved and compared to the same calculation on Wolfram Alpha. This test also tests whether "solve" expressions are parsed correctly.

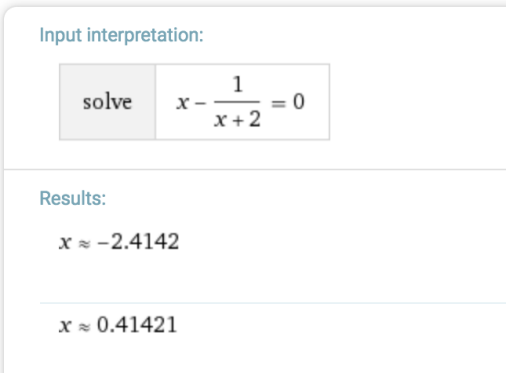
CASE 1

AME:

```
solve x - 1/(x + 2) = 0
```

0.41421

Wolfram Alpha:



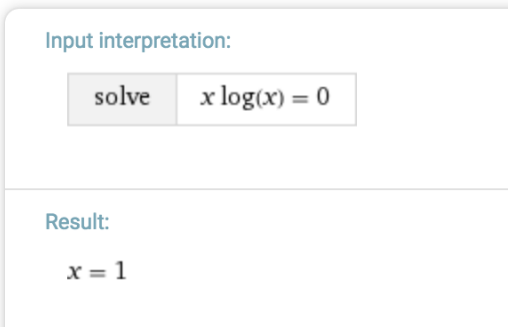
CASE 2

AME:

```
solve x*log(x) = 0
```

1.0

Wolfram Alpha:



4.3.10 TEST 10

Three different expressions will be integrated between 0 and 1: one function involving square roots, one involving logs and one that requires integration by parts. These will be compared to the same calculation on Wolfram Alpha. This test also tests whether "integrate" expressions are parsed correctly.

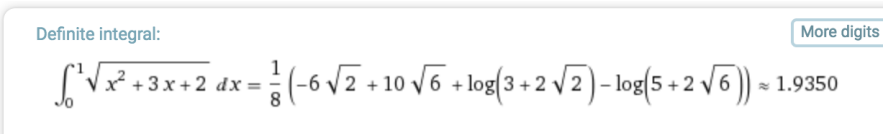
CASE 1

AME:

```
integrate sqrt(x^2 + 3*x + 2) between 0 and 1
```

$$1.93499$$

Wolfram Alpha:



CASE 2

AME:

```
integrate log(3 + x) + log(x^2 + 1) between 0 and 1
```

$$1.51328$$

Wolfram Alpha:

Definite integral:

$$\int_0^1 (\log(3+x) + \log(x^2+1))\,dx = -3 + \frac{\pi}{2} + \log\left(\frac{512}{27}\right) \approx 1.5133$$

CASE 3

AME:

```
integrate exp(x)*sin(x) between 0 and 1
```

$$0.90933$$

Wolfram Alpha:

Definite integral:

$$\int_0^1 e^x \sin(x)\,dx = \frac{1}{2}(1 + e\sin(1) - e\cos(1)) \approx 0.90933$$

4.3.11 TEST 11

The eigenvalues of a matrix will be calculated and compared to the same calculation on Wolfram Alpha. This test also tests the natural language parser for eigenvalue expressions.

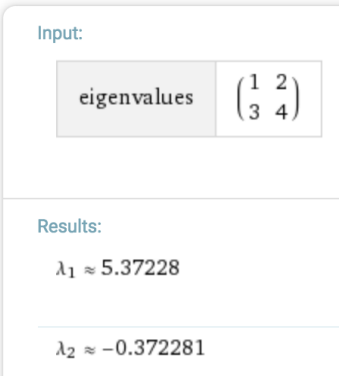
AME:

```
eigenvalues of [[1, 2], [3, 4]]
```

$$5.37228$$

$$-0.37228$$

Wolfram Alpha:



4.3.12 TEST 12

The eigenvectors of a matrix will be calculated and compared to the same calculation on Wolfram Alpha. This test also tests the natural language parser for eigenvector expressions.

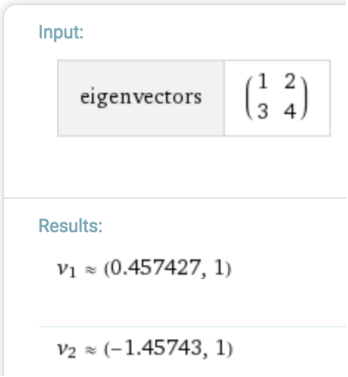
AME:

```
the eigenvectors of [[1, 2], [3, 4]]
```

$$\begin{bmatrix} 0.50589 \\ 1.10594 \end{bmatrix}$$

$$\begin{bmatrix} 0.35551 \\ -0.24393 \end{bmatrix}$$

Wolfram Alpha:



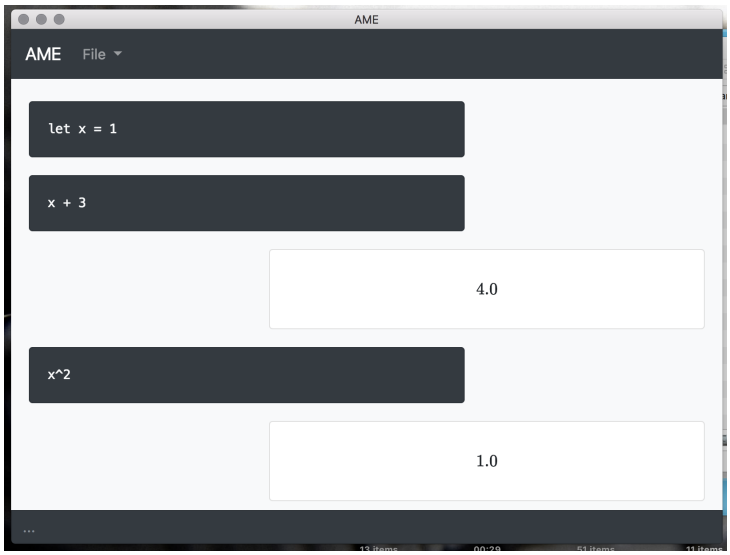
These do not immediately seem to be the same, however, these are actually correct since eigenvalues are only unique up to a scale factor:

$$\begin{bmatrix} 0.50589 \\ 1.10594 \end{bmatrix} = 1.10594 \cdot \begin{bmatrix} 0.45743 \\ 1 \end{bmatrix}$$

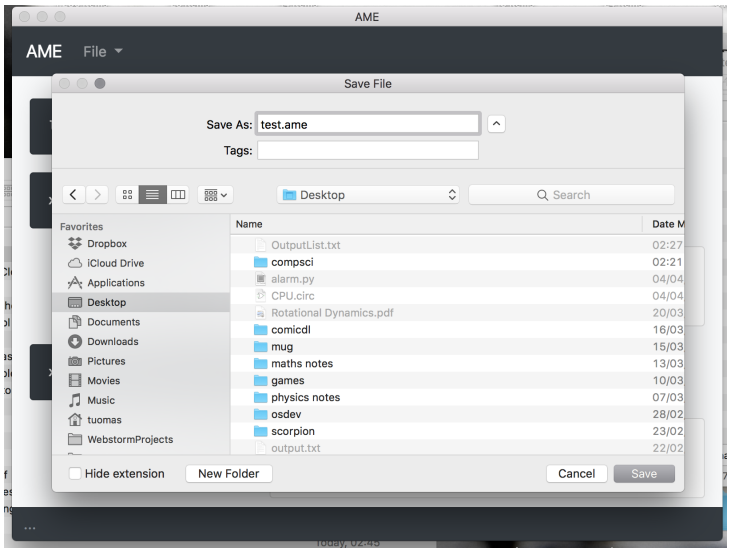
$$\begin{bmatrix} 0.35551 \\ -0.24393 \end{bmatrix} = -0.24393 \cdot \begin{bmatrix} -1.4574 \\ 1 \end{bmatrix}$$

4.3.13 TEST 13

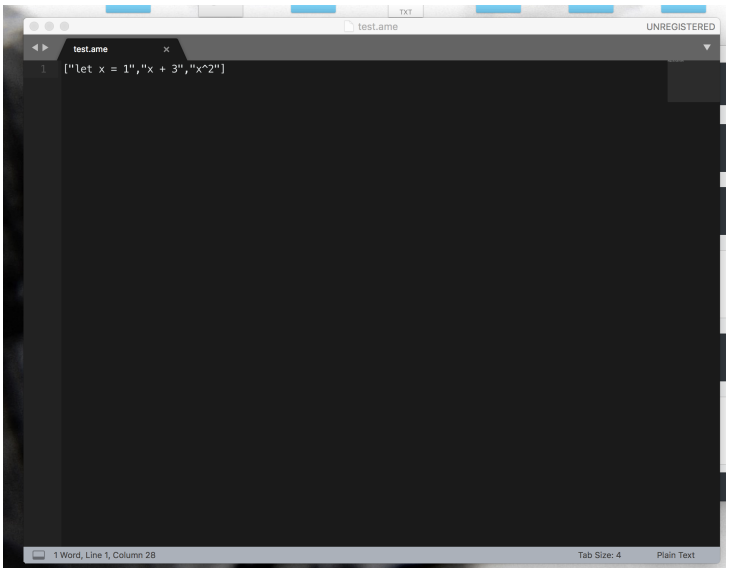
If some input is entered:



And then the save as button is pressed, a dialog box should pop up asking where to save the file to.



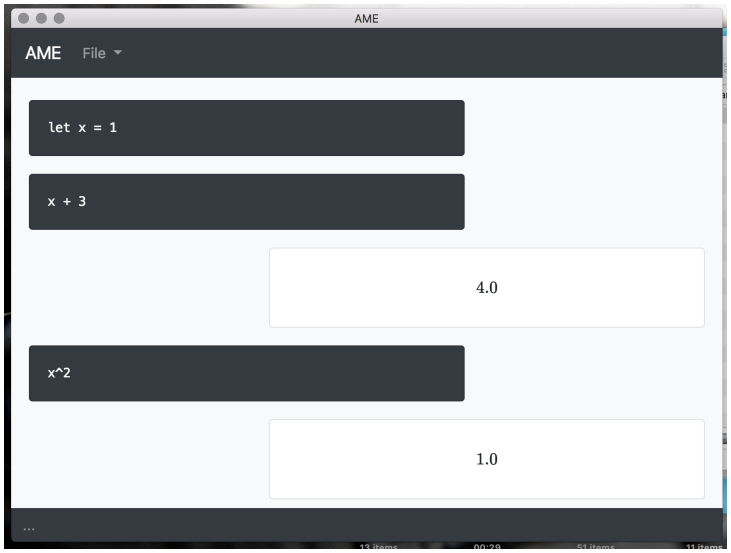
Then if when save is clicked, a filed should be created containing the input that was in the current session.



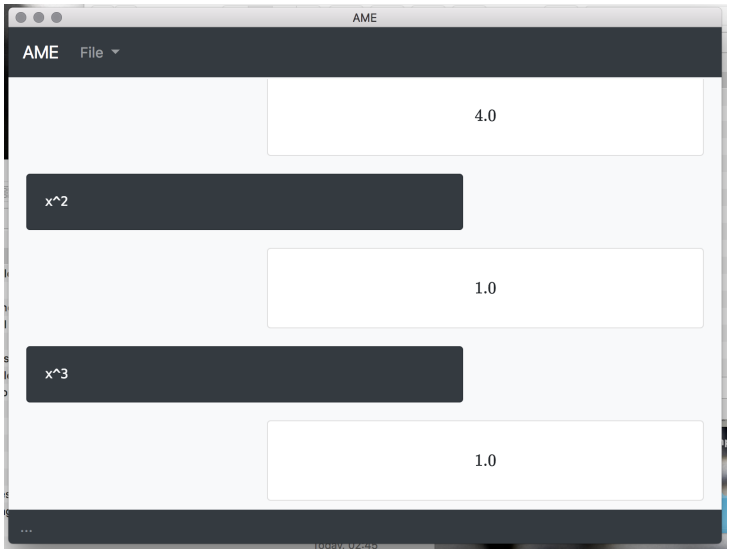
Since this session contains more than one statement of user input, this also verifies that previous user input is displayed.

4.3.14 TEST 14

If a sessions that has been previously saved is open,

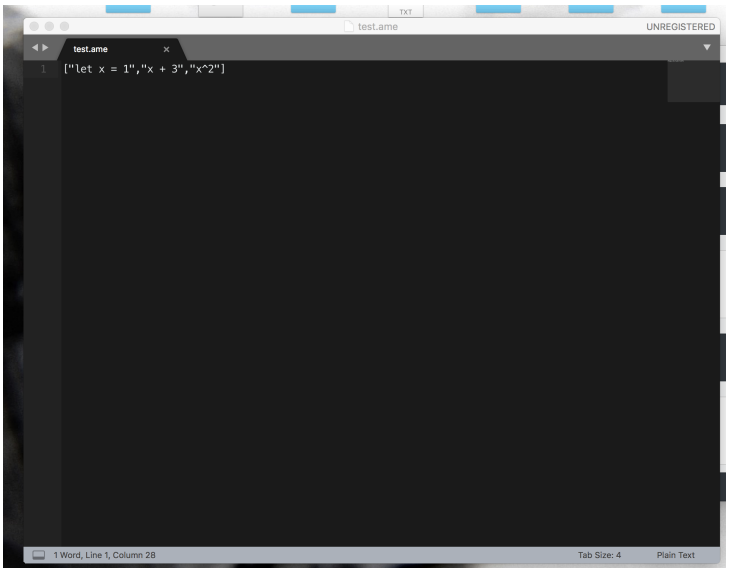


and some more input is added to it,

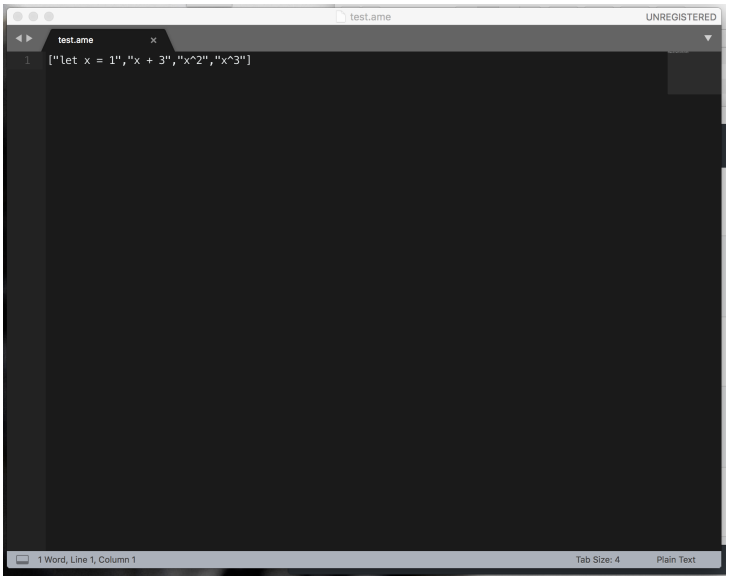


and then the save button is pressed, the save file should change to reflect the new input.

Before:

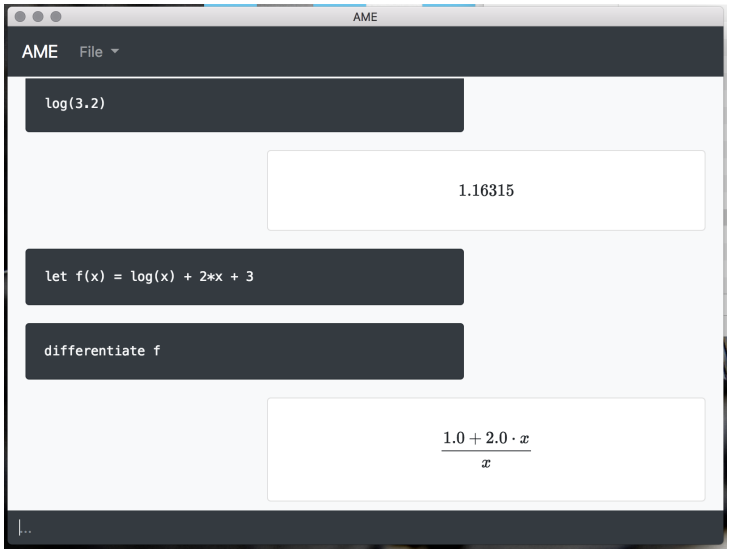

TXT
test.ame
UNREGISTERED
test.ame
1
["let x = 1","x + 3","x^2"]
1 Word, Line 1, Column 28
Tab Size: 4
Plain Text

After:

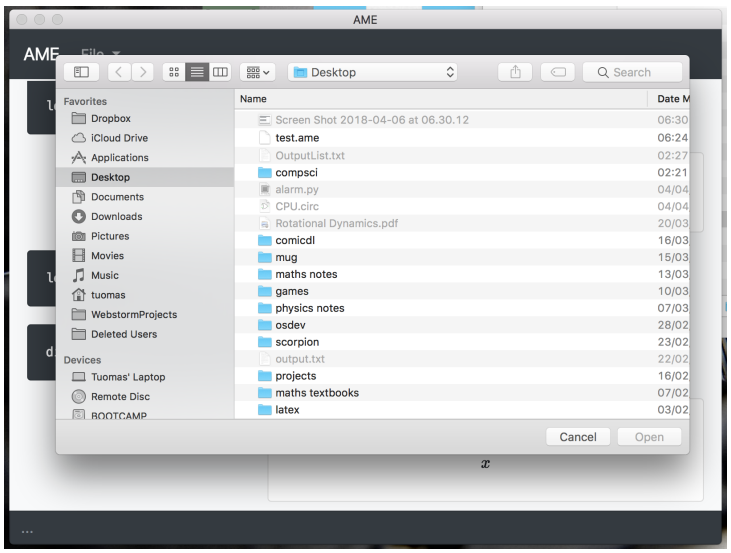

test.ame
UNREGISTERED
test.ame
1
["let x = 1","x + 3","x^2","x^3"]
1 Word, Line 1, Column 1
Tab Size: 4
Plain Text

4.3.15 TEST 15

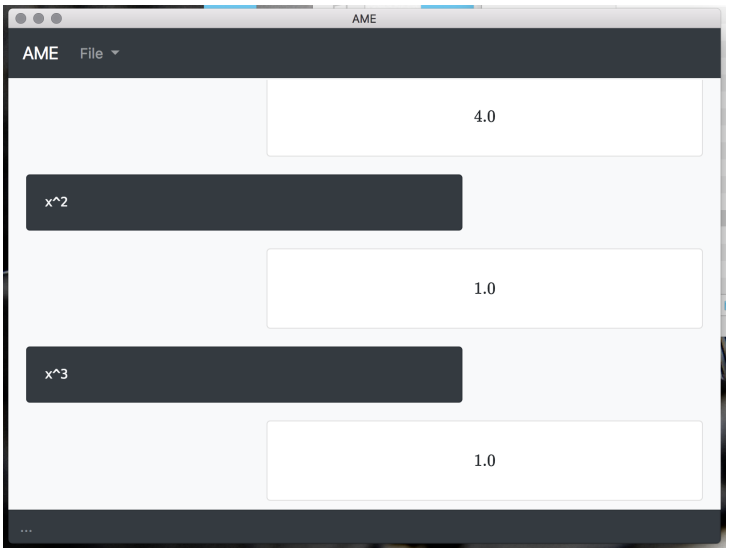
If a session is running,



And open is pressed, a dialog box should open asking which file you wish to open,

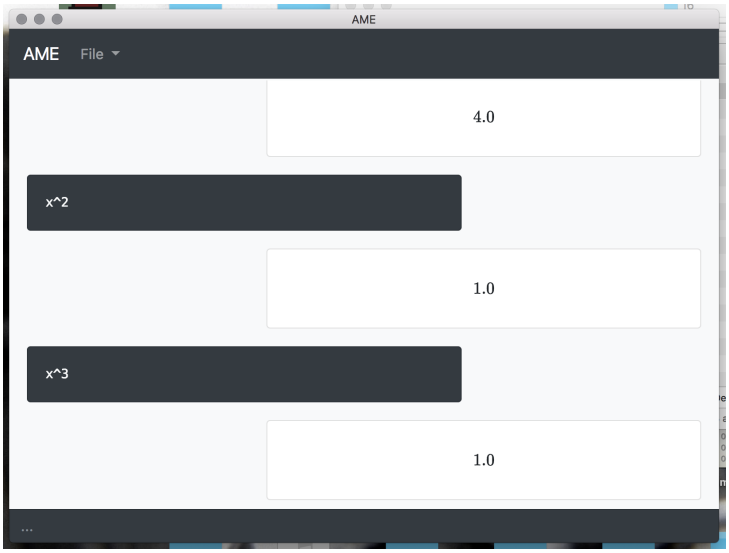


And then if a valid file is selected, the input in the current session should be replaced with the input from the saved file.

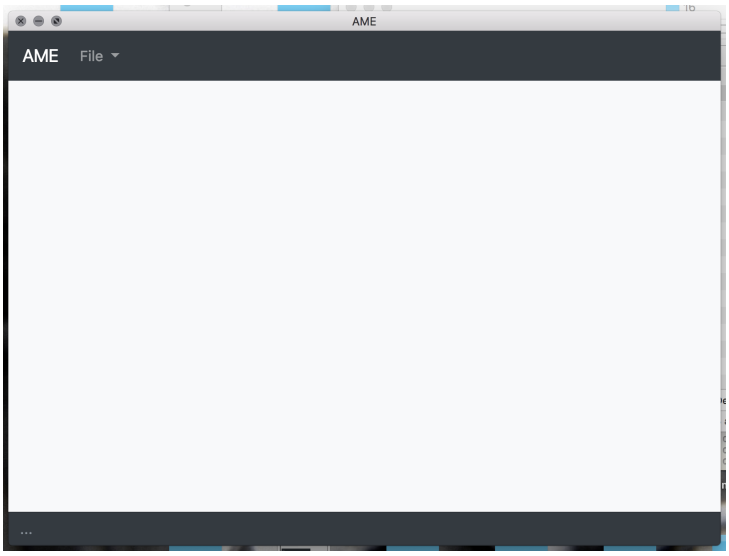

AME
AME
File
4.0
x^2
1.0
x^3
1.0

4.3.16 TEST 16

If a session is running,

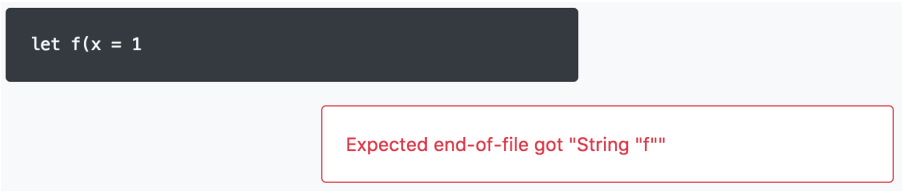


And new is pressed, all of the current input should be removed.



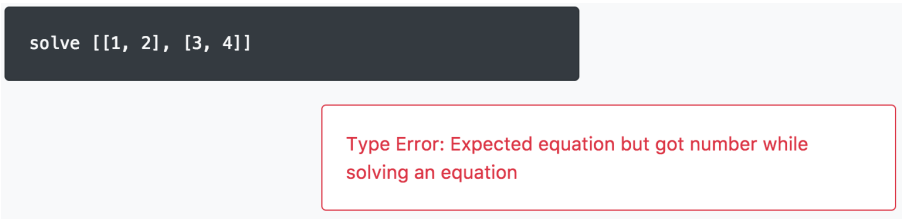
4.3.17 TEST 17

If the input is not valid, the parser should alert the user and attempt to provide an explanation as to what went wrong. Here the explanation is not very clear, but it is provided and the user is alerted to this by the error message.

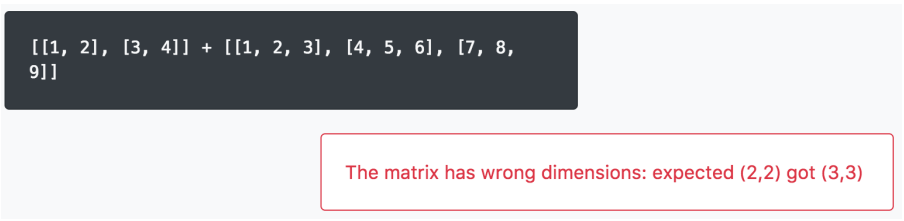


4.3.18 TEST 18

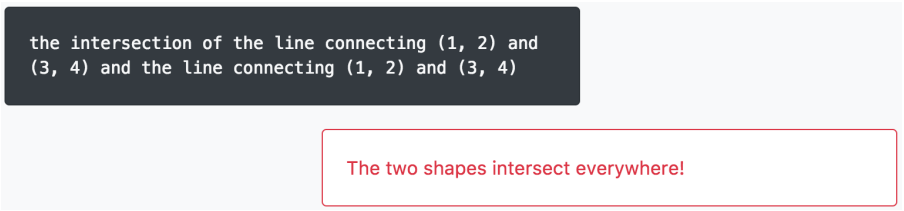
In some cases, valid input can specify invalid calculations, such as adding two matrices of the wrong size or attempting to subtract a circle from a dataset. In such cases error messages should be produced for the user. Specifically, if a value with the wrong type is given to a function a type error should be produced.



If, in a matrix calculation, the wrong size matrix is used, an error should be produced.



And if the user attempts to calculate the intersection of two identical shapes, an error should be produced.



This also verifies that the GUI can display error messages correctly.

5 EVALUATION

Objective	*Met*	*Comment*
1.a) File Operations	Yes	This functionality is quite basic at the moment and could be further extended to incorporate different file formats (such as Mathematica or Jupyter notebooks) in the future.
1.b) User Input	Yes	This is implemented using the messaging-style GUI.
1.c) Input History	Yes	
2.a) Calculations	Yes	This works fairly well, although there are some issues with this that could be improved (see the Future Extensions section)
2.b) Category 1	Yes	All are implemented and complete except for one issue, which is mentioned below.
2.c) Category 2	Yes	
2.d) Additional Calculations	Yes	One additional calculation was added because it was used anyway internally and it was not difficult to implement.
3.a) Natural Language Interface	Yes	This has worked very well and is the most reliable part of the project!
3.b) Calculations	Yes	
3.c) Values	Yes	
3.d) Alternative Phrasings	Yes	Not many alternative syntaxes are currently accepted and I would work on this if I had more time on the project.

3.e) Error Messages	Yes	While this feature is implemented, it is currently very basic and needs serious work before it could be considered useful.
4.a) Multi-platform Support		See below
4.b) Consistent UI	Yes	As the same version of Chrome is used to display the UI across all platforms, this can be guaranteed.

5.1 CURRENT LIMITATIONS

RATIONAL FUNCTION INTEGRATION

The only Category 1 calculation that is currently not fully complete is rational function integration. The reasoning behind this is explained in the code, but unfortunately, this could not be fully fixed without either: implementing a more sophisticated polynomial factoring algorithm (such as Cantor-Zassenhaus), implementing a reliable numerical solver for complex-valued functions, or using a completely different algorithm for this calculation. While the first option of these is probably the most viable, it still has significant technical challenges, as the algorithms are very complex and thus difficult to implement.

SYNTAX ERRORS

Providing explanations for syntax errors was an optional requirement. However, the current system is significantly floored due to the way that my parser combinators are implemented. Currently the errors reported are due to the order of operations in my source code, and not necessarily related to the error that actually occurred. Fundamentally, this is because of a bad design choice I made in the lexer. Currently, in the lexer, there is no distinction between identifiers (variable names) and keywords (words that are used as part of syntax). This then causes problems in my main parser, where it can't distinguish between incorrectly written syntax, and just a list of random variable names, so it sometimes reports the latter instead of the former.

To fix this I would have to rewrite the lexer, and parts of the parser. Unfortunately, this will not really fix the problem, as while it would tell the user more accurately where the error occurred, it would still not provide any kind of useful explanation as to why. For this the easiest method (which is used by many different programming languages) is to write parsers not only for correct syntax, but also for incorrect syntax, so that you can provide a descriptive error message for each case. This would unfortunately require a significant time investment as the number of cases required for a project like this would be very large.

MULTI-PLATFORM SUPPORT

Theoretically, my program should run on any program for which Google Chrome (or Chromium), NodeJS, ZMQ and GHC are supported. This includes Windows, macOS, Linux, BSD, Unix, Android and iOS, however actually getting the project to build on any of these platforms is a significant challenge. Currently, the project has been tested extensively on macOS and has been successfully compiled (although not tested) on Windows (albeit under Windows Subsystem for Linux). The reason I have not marked this objective as met is that I haven't had a chance to test it on either Windows or Linux.

5.2 USER FEEDBACK

I showed my final program to ten members of my maths class. I asked them each to rate my program in three categories: ease of use, interface design and range of functionality. The results are as follows:

Student	*Ease of Use / 10*	*Interface Design / 10*	*Functionality / 10*
A	7	8	5
B	5	6	7
C	8	9	9
D	9	10	8
E	6	8	6
F	5	7	4
G	8	7	6
H	9	9	8
I	10	8	9
J	6	7	4
Average	*7.3*	*7.9*	*6.7*

Additionally, they had some comments about these categories:

EASE OF USE

- The error messages for syntax errors were not very helpful, although the error messages for the calculations were suprisingly good.
- There is no way to copy and paste into or from the input box.
- The application is a very large file which makes it difficult to share.

INTERFACE DESIGN

- Tables and lists are represented very clearly, and the messaging style interface is instantly familiar.
- The color scheme means some text isn't always clear.
- The file menu isn't part of the applications menu bar, but is part of the main window.

FUNCTIONALITY

- Solving differential equations, vector operations and approximating integrals should be supported.
- The statistical calculations worked well and were very intuitive.
- There is no way to edit tables, so if you get a number wrong you have to enter the whole table again.

5.3 FUTURE EXTENSIONS

FILE FORMATS

Currently, the system uses one file format that simply records the input that the user entered so that the session can be effectively "replayed" at a later date. This could be significantly improved by also recording the output of each of these calculations, and the variable state, so that any computationally expensive calculations would not have to be re-run every time a session is loaded.

Additionally, the system could add support for saving to (or even opening from) Jupyter notebooks (used by SciPy and SageMath) or Mathematica files. This would make it easier to transition from my program to another program, if, for example, you needed to perform a computation that my program does not support.

CALCULATIONS

Currently the selection of calculations is quite limited. This could be improved by adding support for new calculations or by upgrading some Category 3 calculations to Category 2.

Additionally, if I were to revisit this project, I would want to make all the calculations much more accurate. The main problem with this in my project has been my choice to represent numbers as double-precision floating point numbers. While these are very accurate, they have caused many, many problems in my code due to the inconsistent

behaviour in division, rounding, multiplication, and the lack of precision has also caused problems when dealing with large integers. The most important revision to this program would be to swap this out for some kind of arbitrary precision type.

ALTERNATIVE SYNTAX

While the current natural language interface works surprisingly well, it does not accept many alternative syntaxes. This means that if even a single word is misspelled or the word ordering is different, the parser will not be able to parse the users input. This could be extended in two ways. Either,

- The system could have many more alternative phrasings, optional words, and a more permissive parser (e.g instead of matching words exactly, match words close to the given word), coded in manually.
- Or, a statistical system could be used which detects the users intent based on statistically identifying key words in the input. This is the approach taken in Wolfram Alpha.

The problem with the second approach that led to me initially rejecting it is that then the guessed intent of the user needs to be communicated clearly to the user, as this could often not be guessed correctly, which would lead to very confusing results for the user. However, I think if I were to revisit the project, I would build a prototype for this, to see if it is viable.

5.4 OVERALL EVALUATION

While there some areas that could be improved and lots of possibilities for future additions, I feel that the project has gone very well overall, meeting all (but one) of the objectives. I have personally been using the program and have found it very helpful (I even used it to compute the answers to some of the examples given in the Design section!) and I have found it increasingly replacing my use of Wolfram Alpha.

I think that the user feedback was generally positive and I feel that if I were to try to distribute the program amongst the class they would be receptive towards it. This is exactly how I intended it, as only a few of them have prior programming experience. I think the project has been an overall success, with lots more ideas I could pursue in the future to improve it.

A SOURCE CODE

A.1 COMPUTE COMPONENT

A.1.1 AME/COMPUTE/EXPR.HS

```haskell
{-# LANGUAGE FlexibleInstances #-}

module AME.Compute.Expr (
    Expr(..),
    Equation(..),
    (=:=),
    X,
    x
) where

import AME.Compute.Error

--  This module contains a generalised expression class,
--  that represents elementary expressions of two parameters:
--      * A type of variables
--      * A field of constants
--  This is a portable representation used for all operations in
    ame-compute.

data Expr var const = Var var
                    | Const const
                    | Sum [Expr var const]
```

```
22                      | Mul [Expr var const]
23                      | Div (Expr var const) (Expr var const)
24                      | Pow (Expr var const) (Expr var const)
25                      | Exp (Expr var const)
26                      | Log (Expr var const)
27                      | Sin (Expr var const)
28                      | Cos (Expr var const)
29                      | Tan (Expr var const)
30                      | ASin (Expr var const)
31                      | ACos (Expr var const)
32                      | ATan (Expr var const)
33                      | Abs (Expr var const)
34                     deriving (Show, Read, Eq, Ord)
35
36 data Equation var const = Equation (Expr var const) (Expr var const)
       deriving (Eq, Show, Read, Ord)
37 --Equation is an equation with       a lhs       and     a rhs
38
39 -- (to make constructing equations a little prettier)
40 (=:=) :: Expr var const -> Expr var const -> Equation var const
41 (=:=) = Equation
42 infixr 2 =:=
43
44 -- The below classes are from the Haskell prelude and are
       implemented for
45 -- the sake of convenience for the rest of this library, as they
       will probably not be used by
46 -- the next stages of the application.
47
48 -- If the constant field supports addition, subtraction,
       multiplication, then Expr does too.
49 instance Num const => Num (Expr var const) where
50     fromInteger = Const . fromInteger
51     --Integers become constant integers.
52     a + b = Sum [a, b]
53     negate a = Mul [(Const $ negate 1), a]
54     a * b = Mul [a, b]
55     abs = Abs
56     signum = error "Expr does not support signum!"
57     -- If we use signum somethings gone horribly wrong, so crash the
```

```
        program with an error.
58
59 -- If the constant field supports division, then Expr does too
60 instance Fractional const => Fractional (Expr var const) where
61     fromRational = Const . fromRational
62     --Rationals become constant rationals
63     (/) = Div
64
65 -- If the constant field supports floating point operations, Expr
       does too
66 instance Floating const => Floating (Expr var const) where
67     pi = Const pi
68     exp = Exp
69     log = Log
70     (**) = Pow
71     sin = Sin
72     cos = Cos
73     tan = Tan
74     asin = ASin
75     acos = ACos
76     atan = ATan
77     sinh = error "Hyperbolic functions and their inverses are not
           supported!"
78     cosh = error "Hyperbolic functions and their inverses are not
           supported!"
79     tanh = error "Hyperbolic functions and their inverses are not
           supported!"
80     asinh = error "Hyperbolic functions and their inverses are not
           supported!"
81     acosh = error "Hyperbolic functions and their inverses are not
           supported!"
82     atanh = error "Hyperbolic functions and their inverses are not
           supported!"
83     -- If we use this, something has gone horribly wrong.
84
85 data X = X deriving (Show, Eq, Ord)
86
87 x :: Expr X const
88 x = Var X
```

A.1.2 AME/COMPUTE/ERROR.HS

```haskell
{-# LANGUAGE TypeSynonymInstances, FlexibleInstances #-}

module AME.Compute.Error (
    NumericalError(..),
    Except.throwError,
    Except.catchError,
    runCompute,
    Compute,
    IsIntegral,
    isIntegral,
    asInt
) where

import qualified Control.Monad.Except as Except
import Control.Monad

-- A module providing a type for general numerical errors,
-- used for all functions where they could occur.

-- A type that can tell if it is an integral value, used for the
   const type in Expr var const
class IsIntegral a where
    isIntegral :: a -> Bool
    asInt :: a -> Maybe Int

instance IsIntegral Double where
    isIntegral a = abs (a - (fromIntegral $ truncate a)) < 0.000001
    asInt a = Just $ round a

instance IsIntegral Float where
    isIntegral a = abs (a - (fromIntegral $ truncate a)) < 0.000001
    asInt a = Just $ round a

--obvious implementations of IsIntegral for our two basic floating
   types

data NumericalError = DivisionByZero -- obvious
                    | UnexpectedFreeVariable -- when trying to
```

```
                        evaluate, we had variables
37                    | MatrixHasWrongDimensions Int Int Int Int --
                        when trying to multiply two matrices
38                    | MatrixShouldBeSquare Int Int -- when trying to
                        compute determinants, eigenvalues etc.
39                    | MatrixNotInvertible -- obvious
40                    | MatrixOperationNotSupported String -- when a
                        trig or log function is called on a matrix
41                    | IntersectsEverywhere -- when trying to find
                        the intersection of two shapes which are the
                        same
42                    | ShouldBeLinear -- when trying to solve
                        simultaneous equations
43                    | WrongNumberOfEquations Int Int -- when trying
                        to solve simultaneous equations
44                    | NoPartialIntegrals -- when trying to integrate
                        and there is more than one free variable.
45                   deriving (Eq, Show, Ord)
46
47 -- A transformer that adds exception handling to a type.
48 type Compute t = Except.Except NumericalError t
49
50 -- Evaluate a Compute expression.
51 runCompute :: Compute t -> Either NumericalError t
52 runCompute = Except.runExcept
```

A.1.3 AME/COMPUTE/SIMPLIFY.HS

```
1  {-# LANGUAGE LambdaCase, RankNTypes, ScopedTypeVariables,
       ConstraintKinds, FlexibleInstances #-}
2
3  module AME.Compute.Simplify (
4      substitute,
5      evaluate,
6      emap,
7      simplify,
8      simplifyEqn,
9      freeVariables,
10     emapt,
11     Simplifiable
12 ) where
```

```
13
14 import AME.Compute.Expr
15 import AME.Compute.Error
16 import Control.Monad
17 import Data.Foldable
18 import Data.List
19 import Data.Function
20 import Data.Bifunctor
21 import Data.Maybe
22
23 --  This module contains functions for:
24 --      * Simplifying algebraic expressions
25 --      * Evaluating expressions
26 --      * Substituting values into expressions
27 --  Technically evaluate and simplify give the same result on
   expressions
28 --  with no free variables but evaluate is _much_ faster.
29
30 -- substitute a variable for a value in an expression.
31 -- i.e) substitute X (Const 1) (Sum [Exp X, Div X (Const 2)]) = Sum
   [Expr (Const 1), Div (Const 1) (Const 2)]
32 substitute :: Eq var => var -> Expr var const -> Expr var const ->
   Compute (Expr var const)
33 substitute var value expr = return $ emapt (\case
34     Var v | v == var -> value
35     x -> x) expr
36
37 emap :: (Expr var const -> Expr var const) -> (Expr var const ->
   Expr var const)
38 emap f (Sum xs) = Sum $ map (f . emap f) xs
39 emap f (Mul xs) = Mul $ map (f . emap f) xs
40 emap f (Div a b) = Div (f $ emap f a) (f $ emap f b)
41 emap f (Pow a b) = Pow (f $ emap f a) (f $ emap f b)
42 emap f (Exp e) = f $ Exp $ emap f e
43 emap f (Log e) = f $ Log $ emap f e
44 emap f (Abs e) = f $ Abs $ emap f e
45 emap f (Sin e) = f $ Sin $ emap f e
46 emap f (Cos e) = f $ Cos $ emap f e
47 emap f (Tan e) = f $ Tan $ emap f e
48 emap f (ASin e) = f $ ASin $ emap f e
```

```
49 emap f (ACos e) = f $ ACos $ emap f e
50 emap f (ATan e) = f $ ATan $ emap f e
51 emap f x = f x
52
53 -- emapt will execute a function over all the terminal
       subexpressions (Const and Var)
54 emapt :: (Expr var1 const1 -> Expr var2 const2) -> Expr var1 const1
       -> Expr var2 const2
55 emapt f (Sum es) = Sum $ map (emapt f) es
56 emapt f (Mul es) = Mul $ map (emapt f) es
57 emapt f (Div ea eb) = Div (emapt f ea) (emapt f eb)
58 emapt f (Pow ea eb) = Pow (emapt f ea) (emapt f eb)
59 emapt f (Exp e) = Exp (emapt f e)
60 emapt f (Log e) = Log (emapt f e)
61 emapt f (Sin e) = Sin (emapt f e)
62 emapt f (Cos e) = Cos (emapt f e)
63 emapt f (Tan e) = Tan (emapt f e)
64 emapt f (ASin e) = ASin (emapt f e)
65 emapt f (ACos e) = ACos (emapt f e)
66 emapt f (ATan e) = ATan (emapt f e)
67 emapt f (Abs e) = Abs (emapt f e)
68 emapt f (Var x) = f (Var x)
69 emapt f (Const c) = f (Const c)
70
71 -- evaluate a expression that does not contain any free variables.
72 evaluate :: (Eq const, Floating const) => Expr var const -> Compute
       const
73 evaluate (Var _) = throwError UnexpectedFreeVariable
74 evaluate (Const c) = return c
75 evaluate (Sum es) = sum <$> mapM evaluate es
76 evaluate (Mul es) = product <$> mapM evaluate es
77 evaluate (Div ea eb) = do
78     a <- evaluate ea
79     b <- evaluate eb
80     if b /= 0 -- We need to check for division by zero:
81               -- Haskell may not throw an error, if we leave it on
                     its own here.
82         then return $ a / b
83         else throwError DivisionByZero
84 evaluate (Pow ea eb) = liftM2 (**) (evaluate ea) (evaluate eb)
```

```
85 evaluate (Exp e) = liftM exp $ evaluate e
86 evaluate (Log e) = liftM log $ evaluate e
87 evaluate (Sin e) = liftM sin $ evaluate e
88 evaluate (Cos e) = liftM cos $ evaluate e
89 evaluate (Tan e) = liftM tan $ evaluate e
90 evaluate (ASin e) = liftM asin $ evaluate e
91 evaluate (ACos e) = liftM acos $ evaluate e
92 evaluate (ATan e) = liftM atan $ evaluate e
93 evaluate (Abs e) = liftM abs $ evaluate e
94
95 -- The strategy for simplification:
96 -- Simplify obvious cases of sums and products, to give as flat as
       possible structure.
97 -- Simplify obvious cases of functions and their inverses:
       log(exp(x)) = x etc.
98 -- Multiply out all of the products of sums.
99 -- Simplify all the constants.
100 -- Get all fractions over a common denominator in sums and combine
       them.
101 -- Cancel variables and constants only - anything else is beyond the
       scope of this (already complicated) function.
102 -- This does mean that many divisions with an obvious answer will
       not work, simply because it can only cancel variables.
103 -- To anyone asking why i would possibly want to make all the
       functions monadic,
104 -- a) to keep the API consistent and b) because (only) cstsimpl may
       throw a DivisionByZero error.
105
106 -- The constraints on an Expr (in terms of var and const) that must
       be present in order to simplify it.
107 type Simplifiable var const = (Ord const, Ord var, Eq var,
       IsIntegral const, Floating const)
108
109 -- Simplify an algebraic expression.
110 simplify :: Simplifiable var const => Expr var const -> Compute
       (Expr var const)
111 simplify e = do
112     e' <- simplifyPass e -- Apply one pass of simplification.
113     if e' == e -- If it does not change, we're done here.
114         then return e'
```

```
115         else simplify e' -- Otherwise do it again.
116
117 simplifyEqn :: Simplifiable var const => Equation var const ->
        Compute (Equation var const)
118 simplifyEqn (Equation a b) = Equation <$> (simplify a) <*> (simplify
        b)
119
120 -- Apply one round of simplification.
121 simplifyPass :: Simplifiable var const => Expr var const -> Compute
        (Expr var const)
122 simplifyPass e = foldlM runPass e passes
123     where passes = [Pass mulmul True, Pass sumsum True, Pass mulsum
            True,
124                     Pass idsum True, Pass idmul True, Pass idpow
                        True, Pass iddiv True,
125                     Pass invle True, Pass intrg True,
126                     Pass mulmul True, Pass muldiv True, Pass sumsum
                        True,
127                     Pass cstsimpl True,
128                     Pass idsum True, Pass idmul True, Pass idpow
                        True, Pass iddiv True,
129                     Pass mulmul True, Pass muldiv True, Pass sumsum
                        True,
130                     Pass kvmul True, Pass kvpow True, Pass kvdiv
                        True,
131                     Pass idsum True, Pass idmul True, Pass idpow
                        True, Pass iddiv True,
132                     Pass mulmul True, Pass muldiv True, Pass sumsum
                        True,
133                     Pass lksum True,
134                     Pass idsum True, Pass idmul True, Pass idpow
                        True, Pass iddiv True,
135                     Pass mulmul True, Pass muldiv True, Pass sumsum
                        True,
136                     Pass lkmul True, Pass inmul True,
137                     Pass idsum True, Pass idmul True, Pass idpow
                        True, Pass iddiv True,
138                     Pass mulmul True, Pass muldiv True, Pass sumsum
                        True,
139                     Pass idsum True, Pass idmul True, Pass idpow
```

```
                        True, Pass iddiv True,
140                     Pass smdiv True,
141                     Pass idsum True, Pass idmul True, Pass idpow
                            True, Pass iddiv True,
142                     Pass mulmul True, Pass muldiv True, Pass sumsum
                            True,
143                     Pass kvmul True, Pass kvpow True, Pass kvdiv
                            True,
144                     Pass cvdiv True,
145                     Pass idsum True, Pass idmul True, Pass idpow
                            True, Pass iddiv True,
146                     Pass mulmul True, Pass muldiv True, Pass sumsum
                            True,
147                     Pass cstrnd True]
148             -- The order of passes was found by trial and error -
                    whatever order you use will give a _technically_
                    correct answer
149             -- but using this order makes it look nice too.
150
151 -- A single simplification pass.
152 -- contains a function to apply to the expression, and a bool
        stating whether this should be applied recursively.
153 data Pass = Pass PassFunc Bool
154
155 -- A function to apply to an expression during a simplification pass
156 type PassFunc = forall var const. Simplifiable var const => Expr var
        const -> Compute (Expr var const)
157
158 --Apply a pass to an expression, recursively if needed.
159 runPass :: Simplifiable var const => Expr var const -> Pass ->
        Compute (Expr var const)
160 runPass e (Pass f False) = f e
161 runPass e p@(Pass _ True) = runPass' e p
162
163 -- Recursively apply a pass to an expression, managing the monadic
        effects of the pass
164 runPass' :: Simplifiable var const => Expr var const -> Pass ->
        Compute (Expr var const)
165 runPass' (Sum as) p@(Pass f _) = (Sum <$> mapM (flip runPass' p) as)
        >>= f
```

```
166 runPass' (Mul as) p@(Pass f _) = (Mul <$> mapM (flip runPass' p) as)
      >>= f
167 runPass' (Div a b) p@(Pass f _) = (liftM2 Div (runPass' a p)
      (runPass' b p)) >>= f
168 runPass' (Pow a b) p@(Pass f _) = (liftM2 Pow (runPass' a p)
      (runPass' b p)) >>= f
169 runPass' (Exp x) p@(Pass f _) = liftM Exp (runPass' x p) >>= f
170 runPass' (Log x) p@(Pass f _) = liftM Log (runPass' x p) >>= f
171 runPass' (Sin x) p@(Pass f _) = liftM Sin (runPass' x p) >>= f
172 runPass' (Cos x) p@(Pass f _) = liftM Cos (runPass' x p) >>= f
173 runPass' (Tan x) p@(Pass f _) = liftM Tan (runPass' x p) >>= f
174 runPass' (ASin x) p@(Pass f _) = liftM ASin (runPass' x p) >>= f
175 runPass' (ACos x) p@(Pass f _) = liftM ACos (runPass' x p) >>= f
176 runPass' (ATan x) p@(Pass f _) = liftM ATan (runPass' x p) >>= f
177 runPass' (Abs x) p@(Pass f _) = liftM Abs (runPass' x p) >>= f
178 runPass' x (Pass f _) = f x
179
180 -- Identity simplification passes: removing additions of zero, and
      multiplications, divisions and exponentiations by one.
181 idsum :: PassFunc
182 idsum (Sum [x]) = return x
183 idsum (Sum []) = return 0 -- Empty sum
184 idsum (Sum x) = return $ Sum $ filter (\x -> case x of
185     Const v -> not $ closeEnough 0.0000001 0 v
186     _ -> True) x
187 idsum x = return x
188
189 idmul :: PassFunc
190 idmul (Mul [x]) = return x
191 idmul (Mul []) = return 1 -- Empty product
192 idmul (Mul x) = return $ Mul $ filter (\x -> case x of
193     Const v -> not $ closeEnough 0.0000001 1 v
194     _ -> True) x
195 idmul x = return x
196
197 closeEnough :: (Ord e, Num e) => e -> e -> e -> Bool
198 closeEnough e a b = abs (a - b) <= e
199
200 iddiv :: PassFunc
201 iddiv (Div a 1) = return a
```

```
202 iddiv x = return x
203
204 idpow :: PassFunc
205 idpow (Pow x 1) = return x
206 idpow x = return x
207
208 -- Inverse simplification passes: simplify things like log(exp(x)) =
      x, sin(asin(x)) = x etc.
209
210 invle :: PassFunc
211 invle (Log (Exp x)) = return x
212 invle (Exp (Log x)) = return x
213 invle x = return x
214
215 intrg :: PassFunc
216 intrg (Sin (ASin x)) = return x
217 intrg (ASin (Sin x)) = return x
218 intrg (Cos (ACos x)) = return x
219 intrg (ACos (Cos x)) = return x
220 intrg (Tan (ATan x)) = return x
221 intrg (ATan (Tan x)) = return x
222 intrg x = return x
223
224 -- Known value passes: remove multiplications by zero, powers of 1,
      powers of zero and division of something by itself.
225 kvmul :: PassFunc
226 kvmul (Mul x) | 0 `elem` x = return 0
227 kvmul x = return x
228
229 kvpow :: PassFunc
230 kvpow (Pow 1 x) = return 1
231 kvpow (Pow x 0) = return 1
232 kvpow (Pow 0 x) = return 0 -- Putting this line _after_ the previous
      guarantees 0^0 == 1
233 kvpow x = return x
234
235 kvdiv :: PassFunc
236 kvdiv (Div a b) | a == b = return 1
237 kvdiv x = return x
238
```

```
239 -- Constant simplification pass: evaluates constant arithmetic
240 cstsimpl :: PassFunc
241 cstsimpl (Sum x) | any isConst x = let (consts, rest) = partition
        isConst x in
242     if closeEnough 0.0000001 0 $ sum $ map constPart consts
243         then return $ Sum rest
244         else return $ Sum $ (Const $ sum $ map constPart consts) :
                rest
245 cstsimpl (Mul x) | any isConst x = let (consts, rest) = partition
        isConst x in
246     if closeEnough 0.0000001 0 $ product $ map constPart consts
247         then return 0
248         else if closeEnough 0.0000001 1 $ product $ map constPart
                consts
249             then return $ Mul rest
250             else return $ Mul $ (Const $ product $ map constPart
                    consts) : rest
251     where closeEnough e a b = abs (a - b) <= e
252 cstsimpl (Pow (Const a) (Const b)) = return $ Const $ a ** b
253 cstsimpl (Div a 0) = throwError DivisionByZero
254 cstsimpl (Div (Const a) (Const b)) = return $ Const $ a / b
255 cstsimpl (Div a (Const b)) = return $ Mul [Const (recip b), a]
256 cstsimpl (Exp (Const a)) = return $ Const $ exp a
257 cstsimpl (Log (Const a)) = return $ Const $ log a
258 cstsimpl (Sin (Const a)) = return $ Const $ sin a
259 cstsimpl (Cos (Const a)) = return $ Const $ cos a
260 cstsimpl (Tan (Const a)) = return $ Const $ tan a
261 cstsimpl (ASin (Const a)) = return $ Const $ asin a
262 cstsimpl (ACos (Const a)) = return $ Const $ acos a
263 cstsimpl (ATan (Const a)) = return $ Const $ atan a
264 cstsimpl (Abs (Const a)) = return $ Const $ abs a
265 cstsimpl x = return x
266
267 isConst :: Expr var const -> Bool
268 isConst (Const _) = True
269 isConst _ = False
270
271 constPart :: Expr var const -> const
272 constPart (Const a) = a
273 constPart _ = undefined
```

```
274
275 -- Round all constants to 5 d.p we really don't need any more than
        that.
276
277 cstrnd :: PassFunc
278 cstrnd (Const c) = return $ Const $ (fromIntegral $ fromJust $ asInt
        $ c * (10^5)) / 100000
279 cstrnd x = return x
280
281 -- Simplify combinations of sums and products
282
283 --sums of sums are just sums
284 sumsum :: PassFunc
285 sumsum (Sum xs) = return $ Sum $ concat $ map parts xs
286     where parts (Sum xs) = xs
287           parts x = [x]
288 sumsum x = return x
289
290 --products of products are just products
291 mulmul :: PassFunc
292 mulmul (Mul xs) = return $ Mul $ concat $ map parts xs
293     where parts (Mul xs) = xs
294           parts x = [x]
295 mulmul x = return x
296
297 --multiply out products of sums
298 mulsum :: PassFunc
299 mulsum (Mul xs) = case sums of -- Do we have any sums in our product?
300     [] -> return $ Mul xs -- If no, just give it back unchanged
301     _ -> let s@(Sum ys) = head $ sums in -- else get the first one
302             return $ Sum $ map (\x -> Mul $ x: removingOne s xs) ys
303             -- remove one copy of the thing we just multiplied by
304             -- and distribute the remaining expressions over the
                   elements of the sum
305     where sums = filter isSum xs
306 mulsum x = return x
307
308 muldiv :: PassFunc
309 mulduv (Mul [Const c, b]) = Mul [Const c, b]
310 muldiv (Mul xs) = case denoms of
```

```
311     [] -> return $ Mul numers
312     _ -> return $ Div (Mul numers) (Mul denoms)
313     where numers = map numer xs
314           denoms = concatMap denom xs
315           numer (Div a b) = a
316           numer x = x
317           denom (Div a b) = [b]
318           denom _ = []
319 muldiv x = return x
320
321 isSum :: Expr var const -> Bool
322 isSum (Sum _) = True
323 isSum x = False
324
325 --Collect like terms
326
327 lksum :: PassFunc
328 lksum (Sum xs) = return $ Sum [Mul [Const c, x] | (c, x) <- zip
        coeffs bases]
329     where bases = nub $ map noCoeff xs
330           coeffs = [sum $ map onlyCoeff $ filter ((== x) . noCoeff)
                  xs | x <- bases]
331 lksum x = return x
332
333 noCoeff :: Expr var const -> Expr var const
334 noCoeff (Mul xs) | (any isConst xs) && (not $ all isConst xs) = case
        filter (not . isConst) xs of
335     [x] -> x
336     xs' -> Mul xs'
337 noCoeff x = x
338
339 onlyCoeff :: Num const => Expr var const -> const
340 onlyCoeff (Mul xs) | (any isConst xs) && (not $ all isConst xs) =
        let Const c = head $ filter isConst xs in c
341 onlyCoeff x = 1
342
343 lkmul :: PassFunc
344 lkmul (Mul xs) = return $ Mul $ [Pow x c | (c, x) <- zip pows bases]
        ++ sums
345     where bases = nub $ map noPower rest
```

```
346           pows = [sum $ map onlyPower $ filter ((== x) . noPower)
                  rest | x <- bases]
347           (sums, rest) = partition isSum xs
348 lkmul x = return x
349
350 noPower :: Expr var const -> Expr var const
351 noPower (Pow a b) = a
352 noPower x = x
353
354 onlyPower :: Num const => Expr var const -> Expr var const
355 onlyPower (Pow a b) = b
356 onlyPower x = 1
357
358 -- lkmul will leave sums alone entirely, so sums will only be
       multiplied out if they are
359 -- represented as a Mul of a repeated value so convert sums to
       integer powers into repeated muls.
360 inmul :: PassFunc
361 inmul (Pow s@(Sum _) (Const c)) | isIntegral c = return $ Mul $
       replicate (fromJust $ asInt c) s
362 inmul x = return x
363
364 -- Simplify sums of quotients by finding a common denominator
365
366 smdiv :: PassFunc
367 smdiv (Sum xs) | any isDiv xs = return $ Div (Sum $ divs' ++ rest')
       (Mul cd)
368     where (divs, rest) = partition isDiv xs
369           divparts = map unDiv divs
370           cd = map snd divparts
371           divs' = [Mul [Mul $ removingOne d cd, n] | (n, d) <-
                  divparts]
372           rest' = [Mul [Mul cd, r] | r <- rest]
373           unDiv (Div a b) = (a, b)
374 smdiv x = return x
375
376 isDiv :: Expr var const -> Bool
377 isDiv (Div _ _) = True
378 isDiv _ = False
379
```

```
removingOne :: Eq a => a -> [a] -> [a]
removingOne a = uncurry (++) . bimap tail id . partition (== a)
--                    |               |         split the input into two lists:
--                    |               |         one of all the a's, and the other the rest
--                    |         remove one element of the list of a's
--               recombine the lists


-- Cancel variables on quotients

cvdiv :: PassFunc
cvdiv c@(Div a b) = return $ (Div `on`) (flip (foldr ($)) $ map (uncurry removePower) cancelVars) a b
    where cancelVars = freeVariables $ Sum [a, b]
cvdiv x = return x

removePower :: (Ord const, Num const, Eq var) => var -> const -> Expr var const -> Expr var const
removePower x 1 (Var y) | x == y = 1
removePower x p (Pow (Var y) (Const c)) | x == y = Pow (Var y) (Const $ max (c - p) 0)
removePower x p (Div a b) = Div (removePower x p a) b
removePower x p (Mul xs) = Mul $ map (removePower x p) xs
removePower x p (Sum xs) = Sum $ map (removePower x p) xs
removePower _ _ x = x

-- Find the highest common integer powers of free variables in an expression
freeVariables :: (Ord const, Num const, Eq var) => Expr var const -> [(var, const)]
freeVariables (Var x) = [(x, 1)]
freeVariables (Sum xs) = map (minimumBy (compare `on` snd)) $ -- find the minimum power of each variable. ^
                         filter ((== length xs) . length) $ -- remove the variables which are not present in all elements. ^
                         groupBy ((==) `on` fst) $  -- group them by which variable they correspond to ^
```

```
409                           concatMap freeVariables xs -- get the free
                                  variables of all the components of the
                                  sum ^
410 --When summing, we select the lowest power, as that must be the
    highest common power.
411 freeVariables (Mul xs) = filter ((/= 0) . snd) $ -- remove the terms
    of power 0
412                           map (bimap head sum . unzip) $
413                           -- the same thing only we add the powers.
                                  and dont care if it doesnt appear in all
                                  elements
414                           groupBy ((==) `on` fst) $
415                           concatMap freeVariables xs
416 --Multiplying adds indices
417 freeVariables (Div a b) = freeVariables a
418 freeVariables (Pow (Var x) (Const c)) = [(x, c)]
419 freeVariables (Pow a _) = []
420 freeVariables _ = []
```

A.1.4 AME/COMPUTE/SOLVER.HS

```
1  {-# LANGUAGE ConstraintKinds #-}
2
3  module AME.Compute.Solver (
4    solve,
5    solveSimultaneous,
6    solveExactPoly,
7    isPolynomial,
8    findRoots,
9    polyCoeffs,
10   Soluble
11 ) where
12
13 import AME.Compute.Expr
14 import AME.Compute.Error
15 import AME.Compute.Simplify
16 import AME.Compute.Matrix.Base
17 import AME.Compute.Solver.Polynomial
18 import AME.Compute.Solver.Newton
19 import Data.List
20 import Control.Monad
```

```
21 import Data.Function
22
23 type Soluble var const = (Eq var, Ord var, Ord const, Floating
       const, IsIntegral const)
24
25 --Generalized solver, this will figure out if it is a polynomial or
       not and if not,
26 --solve via newton's method.
27 solve :: Soluble var const => var -> Equation var const -> Compute
       [const]
28 solve var (Equation lhs rhs) = do
29     expr <- simplify $ lhs - rhs
30     findRoots var expr
31
32 --Solve _linear_ simulatneous equations via matrix inverses.
33 solveSimultaneous :: Soluble var const => [Equation var const] ->
       Compute [(var, const)]
34 solveSimultaneous eqs = do
35     exprs' <- exprs
36     let vars = maximumBy (compare `on` length) $ map varSet exprs'
37     --the total variables are the maximum.....set of free
          variables in the expressions
38     when (length vars /= length eqs) $ throwError $
          WrongNumberOfEquations (length vars) (length eqs)
39     --if we have either too many or too few equations, throw an error
40     when (not $ all isLinear exprs') $ throwError $ ShouldBeLinear
41     --similarly if not all of the equations are linear, throw an
          error
42     let mat = matrix $ [[coeffOf var expr | var <- vars] | expr <-
          exprs' ]
43     --make the matrix of coefficients
44     let res = vector $ [negate $ constTerm expr | expr <- exprs']
45     --the result vector is the vector of the constant terms
46     imat <- inverse mat
47     --invert the coefficients
48     let vals = concat $ values $ imat `multiply` res
49     --and multiply by the constants
50     return $ zip vars vals
51     --attach these back to the variables they apply to
52     where exprs = mapM simplify $ map (\(Equation lhs rhs) -> lhs -
```

```
        rhs) eqs
53          --turn all the equations into simple expressions of lhs -
                rhs
54          multiply a' b' = matrix [[(l `row` a') `dot` (k `column`
                b') | k <- [1..columns b']] | l <- [1..rows a']]
55          --multiply two matrices.. this skips all the error
                checking as
56          --we already know they will have the right dimensions
57
58 varSet :: Eq var => Expr var const -> [var]
59 varSet (Var v) = [v] --the set of variables of x is [x]
60 varSet (Const _) = [] --constants contain no variables
61 varSet (Sum xs) = foldl1 union $ map varSet xs
62 --the variables of a sum of terms is the union of the variables of
      the terms
63 varSet (Mul xs) = foldl1 union $ map varSet xs
64 --same with products
65 varSet (Exp x) = varSet x
66 varSet (Log x) = varSet x
67 varSet (Abs x) = varSet x
68 varSet (Sin x) = varSet x
69 varSet (Cos x) = varSet x
70 varSet (Tan x) = varSet x
71 varSet (ASin x) = varSet x
72 varSet (ACos x) = varSet x
73 varSet (ATan x) = varSet x
74
75 --exactly the following are linear:
76 --x, c*x, c with c constant
77 --and any sum given that all the terms are linear.
78 isLinear :: Expr var const -> Bool
79 isLinear (Var _) = True
80 isLinear (Mul [Const _, Var _]) = True
81 isLinear (Const _) = True
82 isLinear (Sum xs) = all isLinear xs
83 isLinear _ = False
84
85 --extract the coefficient of a variables
86 coeffOf :: (Num const, Eq var) => var -> Expr var const -> const
87 coeffOf var (Var v) = if v == var then 1 else 0
```

```
88 --coefficient of x = 1
89 coeffOf var (Const _) = 0
90 --coefficient of c = 0
91 coeffOf var (Mul [Const c, Var v]) = if v == var then c else 0
92 --coefficient of cx = c
93 coeffOf var (Sum xs) = sum $ map (coeffOf var) xs
94 --coefficient of a sum is the sum of the coefficients of that
      variable in the terms
95 coeffOf _ _ = 0
96
97 --get the constant term of an expression
98 constTerm :: Num const => Expr var const -> const
99 constTerm (Var _) = 0
100 --constant part of x = 0
101 constTerm (Const c) = c
102 --constant part of c = c
103 constTerm (Mul [Const _, Var _]) = 0
104 --constant part of ax = 0
105 constTerm (Sum xs) = sum $ map constTerm xs
106 --constant part of a sum is the sum of the constant parts of the
      terms
107 constTerm _ = 0
108
109 findRoots :: (Eq var, Ord const, Floating const, IsIntegral const)
      => var -> Expr var const -> Compute [const]
110 findRoots var expr = if isPolynomial var expr
111     then solveExactPoly $ polyCoeffs var expr
112     --if this is a polynomial handle it specially,
113     else newtonSolve var expr (isTrig expr)
114     --otherwise use the general solver.
115
116 --Detect if a function is a trig function
117 --to trigger special behaviour in solvePoly that
118 --prevents it from freezing
119 isTrig :: Expr var const -> Bool
120 isTrig (Sum xs) = any isTrig xs
121 isTrig (Mul xs) = any isTrig xs
122 isTrig (Sin _) = True
123 isTrig (Cos _) = True
124 isTrig (Tan _) = True
```

```
125 isTrig (ATan _) = True
126 isTrig (ACos _) = True
127 isTrig (ASin _) = True
128 isTrig (Div a b) = isTrig a || isTrig b
129 isTrig (Pow a b) = isTrig a || isTrig b
130 isTrig _ = False
131
132 solveExactPoly :: (Ord const, Floating const) => [(const, const)] ->
      Compute [const]
133 solveExactPoly = return . solvePoly
134
135 --Something is a polynomial if all its terms are monomial
136 isPolynomial :: (Num const, Ord const, Eq var, IsIntegral const) =>
      var -> Expr var const -> Bool
137 isPolynomial var (Sum xs) = all (isMonomial var) xs
138 isPolynomial var expr = isMonomial var expr
139
140 --Exactly the following are monomial:
141 --  x, c*x, c*x^n, x^n, c where c si
142 isMonomial :: (Ord const, Num const, Eq var, IsIntegral const) =>
      var -> Expr var const -> Bool
143 isMonomial var (Mul [Const _, Var v]) = v == var
144 isMonomial var (Mul [Const _, Pow (Var v) (Const c)]) = isIntegral c
      && (v == var) && (c >= 0)
145 isMonomial var (Var v) = v == var
146 isMonomial var (Pow (Var v) (Const c)) = isIntegral c && (v == var)
      && (c >= 0)
147 isMonomial var (Const _) = True
148 isMonomial _ _ = False
149
150 fromJust :: Maybe a -> a
151 fromJust (Just x) = x
152
153 --Find the polynomial coefficients of a given expression
154 polyCoeffs :: (Eq var, Num const, IsIntegral const) => var -> Expr
      var const -> [(const, const)]
155 polyCoeffs var (Sum xs) = concatMap (polyCoeffs var) xs
156 polyCoeffs var (Var v) | v == var = [(1, 1)]
157 polyCoeffs var (Pow (Var v) (Const c)) | (v == var) && isIntegral c
      = [(fromIntegral $ fromJust $ asInt c, 1)]
```

```
158 polyCoeffs var (Mul [Const a, Var v]) | v == var = [(1, a)]
159 polyCoeffs var (Mul [Const a, Pow (Var v) (Const c)]) | (v == var)
       && isIntegral c = [(fromIntegral $ fromJust $ asInt c, a)]
160 polyCoeffs var (Const c) = [(0, c)]
161 polyCoeffs _ _ = []
162
163 --find the maximum degree of the polynomial
164 degree :: (Num const, Ord const) => [(const, const)] -> const
165 degree coeffs = maximum $ map fst $ filter ((/= 0) . snd) coeffs
166 --                  ^           ^                  ^
167 --by finding the maximum...of the powers...of the terms that aren't
       zero
```

A.1.5 AME/COMPUTE/SOLVER/POLYNOMIAL.HS

```
1  {-# LANGUAGE FlexibleInstances, ScopedTypeVariables #-}
2
3  module AME.Compute.Solver.Polynomial (
4    solveLinear,
5    solveQuadratic,
6    solveHigherOrder,
7    solvePoly
8  ) where
9
10 import Data.List
11 import Data.Function
12 import Data.Ord
13
14 --Solve a polynomial represented as a list of coefficients
15 solvePoly :: (Floating t, Ord t) => [(t, t)] -> [t]
16 solvePoly coeffs' = case n of
17     2 -> solveQuadratic (get 2) (get 1) (get 0)
18     1 -> solveLinear (get 1) (get 0)
19     0 -> solveLinear 0 (get 0)
20     --Specialised solvers for quadratics and linear equations.
21     _ -> solveHigherOrder coeffs
22     where n = degree coeffs
23           get k = case lookup k coeffs of
24             Just x -> x
25             Nothing -> 0
26           coeffs = sortOn (Down . fst) coeffs'
```

```
27                    --Sort by power, in reverse
28                    --This is necessary for solveHigherOrder to work
29
30 degree :: (Num const, Ord const) => [(const, const)] -> const
31 degree coeffs = maximum $ map fst $ filter ((/= 0) . snd) coeffs
32
33 solveLinear :: (Eq t, Fractional t) => t -> t -> [t]
34 solveLinear 0 0 = [0]
35 --0 = 0x + 0 has one solution 0
36 solveLinear 0 _ = []
37 --0 = 0x + a has no solutions for a /= 0
38 solveLinear a b = [-b/a]
39 --0 = ax + b has one solution -b/a
40
41 --solve a quadratic with the quadratic formula.
42 solveQuadratic :: (Ord t, Floating t) => t -> t -> t -> [t]
43 solveQuadratic a b c = case disc `compare` 0 of
44     LT -> []
45     EQ -> [cp]
46     --If the discriminant is zero, we have one repeated root
47     GT -> [cp + sqrt disc, cp - sqrt disc]
48     --Otherwise we can have two roots.
49     where disc = (b^2 - 4*a*c)/(4*a^2)
50           cp = -b/(2*a)
51
52 -- approximately solving higher order polynomials is also _robustly_
   possible
53 -- (i.e not using the erratic joke we call Newton's method)
54 -- using the following algorithm:
55 -- let f(x) be a polynomial of degree > 1, then the roots of f(x)
   must fall into two categories
56 -- 1) it must fall between two stationary points that are not also
   points of inflection,
57 --    (we call these the interior roots of f(x))
58 -- 2) or it must fall somewhere outside (or on) one of the outermost
   stationary or inflection points
59 --    (we call these the exterior roots) (this also covers the case
   of curves with no stationary points)
60 -- to find the interior roots, we find all the stationary points and
   iterate over them pairwise
```

```
61 -- for each pair, we check whether they are on different sides of
   the x-axis, and if so, there must be a
62 -- root between them by the intermediate value theorem and hence we
   find it using the bisection method.
63 -- to find the exterior roots, we first check which of two
   categories the curve is in:
64 -- 1) if the curve has stationary points, we find the outermost
   stationary points. we check if they
65 --    whether they are on the wrong side of the x-axis and it they
   are not, we then perform a bisection
66 --    search between the outermost point and the Lagrange bound for
   the root interval [1].
67 -- 2) if the curve has no stationary points it must have odd degree
   and hence at least one root,
68 --    thus we find the inflection point (there must be one) and
   check whether its above or below zero,
69 --    and perform the same search as before.
70 -- [1]: Lagrange derived the bound of  max {1, sum from i = 0 to n -
   1 of |ai/an|}
71 --      on the magnitude of the roots of a polynomial with real
   coefficients.
72 --      See: Lagrange J-L (1798) Traite de la resolution des
   equations numeriques. Paris.
73
74 solveHigherOrder :: (Ord t, Floating t) => [(t, t)] -> [t]
75 solveHigherOrder coeffs = unionBy (closeEnough 0.0000001) interior
   exterior
76                          --the final root set is the union of
                               interior and exterior roots
77     where closeEnough e a b = abs (a - b) <= e
78           coeffs' = polyDiff coeffs
79           --The first derivative of the function
80           coeffs'' = polyDiff coeffs'
81           --The second derivative
82           sps = map ((,) <*> (polyEval coeffs)) $ solvePoly coeffs'
83           --The stationary points are the roots of the first
                derivative
84           ips = map ((,) <*> (polyEval coeffs)) $ solvePoly coeffs''
85           --The inflection points are the roots of the second
                derivative
```

```
86          cisps = sortBy (compare `on` fst) $ unionBy (closeEnough
                0.0000001 `on` fst) sps ips
87          --Combine these and sort them, left to right
88          interior = case map findInterior (zip cisps $ tail cisps)
                of
89          --we find the interior roots my looking between each pair
                of roots
90          --(zip x $ tail x, generates the list [(a, b), (b, c), (c,
                d)] if x = [a, b, c, d])
91              [] -> []
92              --If there are none, that's okay
93              xs -> foldl1 (unionBy $ closeEnough 0.0000001) xs
94              --Otherwise remove duplicates.
95          findInterior ((x1, fx1), (x2, fx2)) = if signum fx1 /=
                signum fx2
96            --To find interior roots we first check that they are on
                  different sides of the x axis
97            then [bisection x1 x2 0]
98            --and if they are use a bisection search between them
99            else []
100           --and if no, give up.
101         exterior = (findExterior True $ minimumBy (compare `on`
                fst) cisps)
102                 ++  (findExterior False $ maximumBy (compare
                        `on` fst) cisps)
103         --the exterior roots are left of the minimum defined point
104         --and right of the maximum defined point
105         findExterior leftmost (x1, fx1) = case (dir, fx1 <= 0) of
106           (True, False) -> []
107           --If we are looking upwards, but x1 is > 0, there are no
                solutions
108           (False, True) -> []
109           --If we are looking downwards, but x1 is <= 0 there are
                no solutions
110           (True, True) -> [bisection x1 lgbound 0]
111           --If we are looking upwards and x1 is <= 0 bisect
                between here
112           --and the lagrange bound.
113           (False, False) -> [bisection x1 lgbound 0]
114           --If we are looking downwards and x1 > 0, do the same.
```

```
115             where dir = if leftmost
116                         then polyEval coeffs (x1 - 0.1) > fx1
117                         else polyEval coeffs (x1 + 0.1) > fx1
118                   --Check which direction we are looking by
                        evaluating a point on the exterior side
119                   --of the last point.
120                   lgbound = if leftmost then negate lgbound' else
                        lgbound'
121                   --The lagrange bound for the maximum magnitude of
                        the roots.
122                   lgbound' = max 1 (sum $ map (\i -> abs $ (a i)/an)
                        $ range (n - 1))
123                   n = degree coeffs
124                   an = getCoeff coeffs n
125                   a i = getCoeff coeffs i
126                   range 0 = return 0
127                   range n = n : range (n - 1)
128           bisection x1 x2 n = if (closeEnough 0.0000001 fx1 0)
129             --To perform a bisection search, we first check if the
                  endpoints are zero.
130             then x1
131             else if (closeEnough 0.0000001 fx2 0)
132                 then x2
133                 --if so, we are done
134                 else case (fx1 > 0, fm > 0) of
135                 --otherwise, check whether the midpoint and one end
                      point is > 0
136                       (True, True) -> bisection m x2 (n + 1)
137                       --If both are > 0 we need to bisect between
                            the midpoint and the other endpoint
138                       (False, True) -> bisection x1 m (n + 1)
139                       --If the midpoint is >0 but the first end
                            point is not, we bisect between the
                            midpoint
140                       --and that end point
141                       (True, False) -> bisection x1 m (n + 1)
142                       --Similarly, but with the positions swapped
143                       (False, False) -> bisection m x2 (n + 1)
144                       --If both are > 0 we need to bisect between
                            the midpoint and the other endpoint
```

```
145             where fx1 = polyEval coeffs x1
146                   fx2 = polyEval coeffs x2
147                   m = (x1 + x2)/2
148                   fm = polyEval coeffs m
149
150 getCoeff :: (Eq const, Num const) => [(const, const)] -> const ->
      const
151 getCoeff coeffs k = case lookup k coeffs of
152     Just x -> x
153     Nothing -> 0
154
155 --Evaluate a polynomial at a given input.
156 polyEval :: Floating const => [(const, const)] -> const -> const
157 polyEval coeffs val = sum $ map (\(deg, coeff) -> coeff * val **
      deg) coeffs
158
159 polyDiff :: (Eq const, Num const) => [(const, const)] -> [(const,
      const)]
160 polyDiff = map (\(deg, coeff) -> if deg == 0 then (0, 0) else (deg -
      1, coeff * deg))
```

A.1.6 AME/COMPUTE/SOLVER/NEWTON.HS

```
1  module AME.Compute.Solver.Newton (
2    newtonSolve
3  ) where
4
5  import AME.Compute.Expr
6  import AME.Compute.Error
7  import AME.Compute.Simplify
8  import AME.Compute.Calculus.Derivatives
9  import Data.List
10
11 --Solve equations approximately with newton's method
12 newtonSolve :: (Eq var, Ord const, Floating const) => var -> Expr
       var const -> Bool -> Compute [const]
13 newtonSolve v e isTrig = do
14       --to solve something with newton's method
15       as <- newtonSolve' 1 e []
16       --start looking from both 1
17       bs <- newtonSolve' (-1) e []
```

```
18      -- ...and -1, in order to skip out biases in the middle bit
19      return $ as `union` bs
20    where newtonSolve' x00 e' acc = do
21            let x0 = case acc of
22                       [] -> x00 --If we've only just started,
                             start looking from x00 (any number will
                             do here)
23                       (r:_) -> r + 0.1 --If we've already got a
                             root, start looking a small distance from
                             that root,
24                                        --because if that root is
                                           1, then setting x0 = x00
                                           will get a
                                           DivisionByZero error.
25            root <- newtonRoot v x0 isTrig e' -- Find a root of e'
26            if isTrig && (length acc == 4) -- If we have a trig
                 function, we dont want to find too many roots (there
                 should be infinite)
27                then return acc
28                else case root of
29                     Just r -> newtonSolve' x00 (e' / (0.01 * (Var v
                          - Const r))) (r : acc)
30                     -- if we find a root, add it to the list of
                          roots, then find more roots,
31                     -- avoiding the one we just found by dividing
                          the function by (x - r) (we multiply by 0.01
32                     -- to make it avoid r even more) so that when it
                          is close to r it is a large value.
33                     Nothing -> return acc
34
35 newtonRoot :: (Eq var, Ord const, Floating const) => var -> const ->
       Bool -> Expr var const -> Compute (Maybe const)
36 newtonRoot v x0 isTrig e = do
37    fx0 <- f x0
38    --taking one root in newton's method, we iterate the newton
         steps on it
39    iterateNewton fx0 x0 0
40    where e' = differentiate v e -- the derivative of e with respect
         to v
41          f c = do -- evaluate the function at c
```

```
42              se <- substitute v (Const c) e
43              -- substitute v for c and evaluate
44              evaluate se
45            f' c =  do -- the same thing but with the derivative
46              e'' <- e' -- except differentiate returns Compute (Expr
                    _ _) so we need to evaluate it
47              se' <- substitute v (Const c) e''
48              evaluate se'
49            newtonStep x0 = do -- perform one step of newton's method
50              fx <- f x0
51              fx' <- f' x0
52              return $ x0 - fx/fx' -- x1 = x0 - f(x0)/f'(x0)
53            iterateNewton fx0 x0 n = do
54              x1 <- newtonStep x0
55              fx1 <- f x1
56              if n > 100
57                  -- if we have done more than one hundred steps,
                       assume we can't find a root
58                  then return Nothing
59                  else if (abs fx1 < 0.00001)
60                      then return $ Just x1 -- if we find a root,
                           return it
61                      else iterateNewton fx1 x1 (n + 1) -- otherwise
                           keep looking
```

A.1.7 AME/COMPUTE/MATRIX/BASE.HS

```
1  module AME.Compute.Matrix.Base where
2
3  import AME.Compute.Expr
4  import AME.Compute.Error
5  import AME.Compute.Simplify
6  import Control.Monad
7  import Data.Maybe
8
9  --A matrix is represented as a list of rows of data, and is
       parameterised over
10 --a type so any data can be stored
11 data Matrix t = Matrix {
12     rows :: Int,
13     columns :: Int,
```

```
14     values :: [[t]]
15 } deriving (Eq, Show, Ord)
16
17 --Construct a matrix from a list of rows
18 matrix :: [[t]] -> Matrix t
19 matrix ts = Matrix (length ts) (length $ head ts) ts
20
21 --Construct the identity matrix for a given size by
22 --append rows and columns to a smaller identity matrix
23 identity :: Num t => Int -> Matrix t
24 identity 1 = matrix [[1]]
25 identity n = matrix $ (1 : replicate (n - 1) 0) : [0 : row | row <-
       values $ identity $ n - 1]
26
27 --The instance of functor for matrix
28 --maps a function over all the data values in the matrix
29 instance Functor Matrix where
30     fmap f = matrix . map (map f) . values
31
32 --Scale a matrix by a scalar
33 scale :: Num t => t -> Matrix t -> Matrix t
34 scale = fmap . (*)
35
36 --A vector is simply a matrix with 1 column of data
37 type Vector t = Matrix t
38
39 --Construct a vector from a list by wrapping each element in its own
       row.
40 vector :: [t] -> Vector t
41 vector = matrix . map return
42
43 --Check if a matrix is a column vector by check that every row has
       only one element
44 isVector :: Matrix t -> Bool
45 isVector m = (== 1) $ length $ head $ values m
46
47 --The dot product of two vectors is the sum of the point-wise
       product of their components
48 dot :: Num t => Vector t -> Vector t -> t
49 dot a b = sum $ zipWith (*) (head $ values $ a)  (head $ values $ b)
```

```
50
51 --The Generalized Pythagorean theorem is used to compute magnitudes
52 magnitude :: Floating t => Matrix t -> t
53 magnitude a = sqrt $ sum $ map (**2) $ concat $ values a
54
55 normalise :: Floating t => Matrix t -> Matrix t
56 normalize v = (recip $ magnitude v) `scale` v
57
58 row :: Int -> Matrix t -> Vector t
59 row i m = matrix $ return $ values m !! (i - 1)
60
61 column :: Int -> Matrix t -> Vector t
62 column i m = matrix $ return $ map (!! (i - 1)) $ values m
63
64 --The determinant is computed using a laplace expansion on the first
     row of the matrix
65 --This is computed as follows:
66 --  * For the ith element of the first row,
67 --  * Remove the ith row and ith column of the matrix,
68 --  * Compute the determinant of that matrix ("(i, i) cofactor")
69 --  * Multiply it by the element of the first row
70 --  * Repeat for all elements and sum
71 --This is particularly inefficient (O(n!)) but will be sufficient
     for our purposes
72 determinant :: Num t => Matrix t -> Compute t
73 determinant m | (rows m == 1) && (columns m == 1) = return $ head $
     head $ values m
74 determinant m | (rows m == columns m) = sum <$> (mapM (\(i, x) -> (x
     *) <$> (cofactor 1 i m)) $ enumerate $ head $ values m)
75 determinant m = throwError $ MatrixShouldBeSquare (rows m) (columns
     m)
76
77 --The (i, j) minor is the determinant of the matrix with the ith row
     and jth columns removed.
78 minor :: Num t => Int -> Int -> Matrix t -> Compute t
79 minor i j m = determinant $ matrix [[val | (l, val) <- enumerate
     row, l /= j] | (k, row) <- enumerate $ values m, k /= i]
80
81 --The (i, j) cofactor matrix is simply the (i, i) minor,
82 --only negated if i + j is odd
```

```
83 cofactor :: Num t => Int -> Int -> Matrix t -> Compute t
84 cofactor i j m = (if even $ i + j then id else negate) <$> minor i j
       m
85
86 --enumerate [a, b, c, d] = [(1, a), (2, b), (3, c), (4, d)]
87 enumerate :: [t] -> [(Int, t)]
88 enumerate = zip [1..]
89
90 --We compute the inverse matrix using the cofactor formula.
91 inverse :: (Eq t, Fractional t) => Matrix t -> Compute (Matrix t)
92 inverse m = do
93     when (rows m /= columns m) $ throwError $ MatrixShouldBeSquare
           (rows m) (columns m)
94     --We can only invert square matrices
95     det <- determinant m
96     --Compute the determinant of the matrix
97     if det == 0
98         then throwError MatrixNotInvertible
99         --If its zero, we give up
100        else do
101            let row j = mapM id [cofactor i j m | i <- [1..columns
                   m]]
102            --Compute a row of the inverse matrix by taking the (i,
                   j) cofactors
103            --Where i = 1 -> n, and j is the row number, for an nxn
                   matrix.
104            values <- mapM row [1..rows m]
105            --Assemble the inverse matrix from the rows
106            let mat = matrix values
107            return $ (recip det) `scale` mat
108            --Scale it down by the determinant
109
110 --A dummy variable used for finding eigenvalues.
111 data Lambda = Lambda deriving (Show, Eq, Ord)
112
113 --The characteristic equation is the polynomial whose roots are the
       eigenvalues of
114 --a given matrix. We can construct it easily:
115 --The eigenvalue equation is: Mv = lv (l is the eigenvalue, v the
       eigenvector)
```

```
116 --So if we replace l with lI, it should make no difference to v, Mv
      = Ilv
117 --Now we can move the rhs over to the lhs, Mv - Ilv = 0
118 --And factorize out the v: (M - Il)v = 0
119 --since the matrix (M - Il) takes v to zero, we know that (M - Il)
120 --must have determinant zero. So by forming a matrix of expressions,
      in terms of l
121 --we can use our determinant function to form an equation in l.
122 characteristic :: (Ord t, Floating t) => Matrix t -> Compute (Expr
      Lambda t)
123 characteristic m = determinant $ fmap Const m `subtract` ((Var
      Lambda) `scale` identity (rows m))
124     where subtract a' b' = matrix $ zipWith (zipWith (-)) (values
          a') (values b')
```

A.1.8 AME/COMPUTE/MATRIX.HS

```
1  {-# LANGUAGE FlexibleInstances, TypeSynonymInstances, LambdaCase #-}
2
3  module AME.Compute.Matrix (
4    Matrix(..),
5    MatrixExt(..),
6    matrix,
7    scale,
8    Vector,
9    vector,
10   dot,
11   magnitude,
12   normalize,
13   row,
14   column,
15   determinant,
16   inverse,
17   isMatrix,
18   isScalar,
19   asMatrix,
20   asScalar,
21   determinantE,
22   dotE,
23   angleE,
24   magnitudeE,
```

```
25   characteristic,
26   Lambda(..),
27   eigenvalues,
28   eigenvaluesE,
29   eigenvectors,
30   eigenvectorsE
31 ) where
32
33 import AME.Compute.Expr
34 import AME.Compute.Error
35 import AME.Compute.Simplify
36 import AME.Compute.Solver
37 import AME.Compute.Matrix.Base
38 import Control.Monad
39 import Data.Maybe
40
41 -- A type which extends a given domain to include arbitrary
      dimension matrices.
42 -- i.e if R is the reals, then MatrixExt R is the reals plus real
      matrices.
43 -- As not all operations are defined numeric class instances are
      defined for Compute (MatrixExt t).
44 data MatrixExt t = MatrixV (Matrix t) | ScalarV t
45                  deriving (Show, Eq, Ord)
46
47 --fmap'ing over a matrixext will map over all data in a matrix,
48 --or just apply the function to a scalar
49 instance Functor MatrixExt where
50     fmap f (MatrixV m) = MatrixV $ fmap f m
51     fmap f (ScalarV v) = ScalarV $ f v
52
53 isMatrix :: MatrixExt t -> Bool
54 isMatrix (MatrixV _) = True
55 isMatrix _ = False
56
57 asMatrix :: MatrixExt t -> Matrix t
58 asMatrix (MatrixV m) = m
59
60 isScalar :: MatrixExt t -> Bool
61 isScalar = not . isMatrix
```

```
62
63 asScalar :: MatrixExt t -> t
64 asScalar (ScalarV t) = t
65
66 --an error sensitive version of determinant that
67 --does type checking to ensure you pass it a matrix.
68 determinantE :: Num t => Compute (MatrixExt t) -> Compute (MatrixExt
       t)
69 determinantE a = do
70     a' <- a
71     case a' of
72         (MatrixV m) -> ScalarV <$> determinant m
73         (ScalarV _) -> throwError $ MatrixOperationNotSupported
              "Can't find determinant of scalar!"
74
75
76 dotE :: Num t => Compute (MatrixExt t) -> Compute (MatrixExt t) ->
       Compute (MatrixExt t)
77 dotE a b = do
78     a' <- a
79     b' <- b
80     --evaluate the errors in a and b
81     case (a', b') of
82         --Check that we are passed two vectors
83         (MatrixV ma, MatrixV mb) -> if isVector ma && isVector mb
84                                     then return $ ScalarV $ sum
                                            $ zipWith (*) (concat $
                                            values ma) (concat $
                                            values mb)
85                                     --then take the sum of the
                                            pointwise product of
                                            their values
86                                     else throwError $
                                            MatrixOperationNotSupported
                                            "Cannot dot two matrices
                                            that are not column
                                            vectors!"
87         (ScalarV _, ScalarV _) -> throwError $
              MatrixOperationNotSupported "Cannot dot two scalars!"
88         _ -> throwError $ MatrixOperationNotSupported "Cannot dot a
```

```
            vector and scalar!"
89
90 magnitudeE :: Floating t => Compute (MatrixExt t) -> Compute
       (MatrixExt t)
91 magnitudeE a = do
92     a' <- a
93     case a' of
94         (ScalarV _) -> throwError $ MatrixOperationNotSupported
               "Cannot find magnitude of a scalar!"
95         (MatrixV ma) -> if isVector ma
96                             then return $ ScalarV $ magnitude ma
97                             else throwError $
                                    MatrixOperationNotSupported "Cannot
                                    find magnitude of matrix that is not
                                    a vector!"
98
99 --Find the angle between two vectors using the following relation:
100 --a `dot` b = (magnitude a) * (magnitude b) * (cos angle)
101 angleE :: Floating t => Compute (MatrixExt t) -> Compute (MatrixExt
       t) -> Compute (MatrixExt t)
102 angleE a b = do
103     a' <- a
104     b' <- b
105     case (a', b') of
106         --Check we are given two vectors
107         (MatrixV ma, MatrixV mb) -> if isVector ma && isVector mb
108                                         then return $ ScalarV $ acos
                                                $ sum (zipWith (*)
                                                (concat $ values ma)
                                                (concat $ values mb)) /
                                                (magnitude ma * magnitude
                                                mb)
109                                         else throwError $
                                                MatrixOperationNotSupported
                                                "Cannot find angle
                                                between two matrices that
                                                are not column vectors!"
110         (ScalarV _, ScalarV _) -> throwError $
               MatrixOperationNotSupported "Cannot find angle between
               two scalars!"
```

```
111         _ -> throwError $ MatrixOperationNotSupported "Cannot find
                angle between a scalar and a vector!"
112
113 --Compute the eigenvalues by finding the roots of the characteristic
    equation
114 eigenvalues :: (Ord t, Floating t, IsIntegral t) => Matrix t ->
    Compute [t]
115 eigenvalues m = do
116     cr <- characteristic m
117     solve Lambda $ Equation cr 0
118
119 eigenvaluesE :: (Ord t, IsIntegral t, Floating t) => Compute
    (MatrixExt t) -> Compute [MatrixExt t]
120 eigenvaluesE a = do
121     a' <- a
122     case a' of
123         MatrixV m -> eigenvalues m >>= (return . map ScalarV)
124         ScalarV _ -> throwError $ MatrixOperationNotSupported
                "Cannot find eigenvalues of a scalar!"
125
126 --Find the eigenvector of a matrix given an eigenvalue using the
    inverse power iteration algorithm.
127 --This works by first computing the inverse of (M - Il') where l' is
    a value very close to the given eigenvalue
128 --and then taking a random vector and repeatedly multiplying by this
    by the matrix we computed earlier.
129 --It is important that we a) normalize (M' - Il') to keep rounding
    errors low and
130 --b) do not use exactly the eigenvalue we are given, otherwise it
    will multiply to zero.
131 eigenvector :: (Ord t, IsIntegral t, Floating t) => t -> Matrix t ->
    Compute (Vector t)
132 eigenvector v m = do
133     im <- inverse m'
134     --Find the inverse of M - Il'
135     MatrixV res <- (return $ MatrixV $ normalize im) ** 10 * (return
            $ MatrixV $ vector $ replicate (rows m) 1)
136                     -- normalize this.....take it to the power 10
                            .... and multiply it by an arbitrary vector
137     return res
```

```
138     where m' = m `subtract` ((v + 0.01) `scale` identity (rows m))
139           -- M    -     l' = l + 0.01     *        I
140           subtract a' b' = matrix $ zipWith (zipWith (-)) (values
                  a') (values b')
141
142 --The eigenvectors of a matrix are all the eigenvectors
        corresponding to all the eigenvalues
143 eigenvectors :: (Ord t, IsIntegral t, Floating t) => Matrix t ->
        Compute [Vector t]
144 eigenvectors m = do
145     vs <- eigenvalues m
146     mapM (flip eigenvector m) vs
147
148 eigenvectorsE :: (Ord t, IsIntegral t, Floating t) => Compute
        (MatrixExt t) -> Compute [MatrixExt t]
149 eigenvectorsE a = do
150     a' <- a
151     case a' of
152         MatrixV m -> (map MatrixV) <$> eigenvectors m
153         ScalarV _ -> throwError $ MatrixOperationNotSupported
                "Cannot find eigenvectors of a scalar!"
154
155 --Define numeric instances of Compute (MatrixExt t) which has Compute
156 --as some operations fail on matrices.
157 instance Num t => Num (Compute (MatrixExt t)) where
158     a + b = do
159         ra <- a
160         rb <- b
161         when (isScalar ra && isMatrix rb || isScalar rb && isMatrix
                ra) $
162             throwError $ MatrixOperationNotSupported "Cannot add a
                    matrix and a scalar!"
163         --Can only add values of the same type.
164         case (isScalar ra, isScalar rb) of
165             --Scalar addition is defined as usual
166             (True, True) -> return $ ScalarV $ asScalar ra +
                    asScalar rb
167             (False, False) -> do
168                 let a' = asMatrix ra
169                 let b' = asMatrix rb
```

```
170                 if (rows a' == rows b') && (columns a' == columns b')
171                     --Matrix addition only works on matrices of the
                            same shape
172                     then return $ MatrixV $ matrix $ zipWith
                            (zipWith (+)) (values a') (values b')
173                     --If they are, simply add the values pairwise.
174                     else throwError $ MatrixHasWrongDimensions (rows
                            a') (columns a') (rows b') (columns b')
175
176     a * b = do
177         ra <- a
178         rb <- b
179         case (isScalar ra, isScalar rb) of
180             --Every case of matrix and scalar multiplication is
                    defined differently ;p
181             (False, False) -> do
182                 let a' = asMatrix ra
183                 let b' = asMatrix rb
184                 if (rows b' == columns a')
185                     --To multiply two matrices we first check that
                            they are of the form
186                     --axb and bxc respectively.
187                     then return $ MatrixV $ matrix [[(l `row` a')
                            `dot` (k `column` b') | k <- [1..columns b']]
                            | l <- [1..rows a']]
188                     --Then we find the (i, j) element of the
                            resultant matrix as the dot product of the
                            ith row of the first matrix
189                     --with the jth column of the second matrix.
190                     else throwError$ MatrixHasWrongDimensions (rows
                            b') (columns b') (columns a') (columns b')
191             (True, False) -> return $ MatrixV $ (asScalar ra)
                    `scale` (asMatrix rb)
192             (False, True) -> return $ MatrixV $ (asScalar rb)
                    `scale` (asMatrix ra)
193             --Matrix * scalar and scalar * Matrix just scale the
                    matrix
194             (True, True) -> return $ ScalarV $ (asScalar ra) *
                    (asScalar rb)
195             --scalar * scalar is defined as usual
```

```
196
197     negate a = case runCompute a of
198         Right (MatrixV a') -> return $ MatrixV $ fmap negate a'
199         Right (ScalarV a') -> return $ ScalarV $ negate a'
200         Left a' -> throwError a'
201     fromInteger = return . ScalarV . fromInteger
202     --Abs is defined as expected for scalars
203     --and for matrices, it is the matrix with all the elements
        absolute valued.
204     --This does _not_ compute the magnitude of the matrix
205     --that is done by magnitude.
206     abs a = case runCompute a of
207         Right (MatrixV a') -> return $ MatrixV $ fmap abs a'
208         Right (ScalarV a') -> return $ ScalarV $ abs a'
209         Left a' -> throwError a'
210
211     --I can't think of where we would ever use this.
212     signum a = case runCompute a of
213         Right (ScalarV a') -> return $ ScalarV $ signum a'
214         Right (MatrixV _) -> error "Signum not supported for
            matrices!"
215         Left a' -> throwError a'
216
217 instance (Eq t, Fractional t) => Fractional (Compute (MatrixExt t))
    where
218     recip a = do
219         ra <- a
220         if isScalar ra
221             then return $ ScalarV $ recip $ asScalar ra
222             else MatrixV <$> (inverse $ asMatrix ra)
223     --Taking the reciprocal of a matrix just yields the inverse
        matrix,
224     --and scalar reciprocals are defind as usual.
225
226     fromRational = return . ScalarV . fromRational
227
228 -- But isnt this just a (constrained, endomorphic) fmap on
    MatrixExt?? (with an extra param) Noooo: there is more than one
    possible
229 -- implementation of fmap for MatrixExt and here we want a slightly
```

```
    different one than we defined earlier.
liftME :: Floating t => (t -> t) -> String -> (Compute (MatrixExt t)
    -> Compute (MatrixExt t))
liftME f n a = do
    a' <- a
    case a' of
        (MatrixV _) -> throwError $ MatrixOperationNotSupported $ n
            ++ " cannot be applied to matrices!"
        (ScalarV v) -> return $ ScalarV $ f v

instance (Eq t, IsIntegral t, Floating t) => Floating (Compute
    (MatrixExt t)) where
    --All floating point operations are left undefined for matrices,
    --and defined as usual for scalars.
    pi = return $ ScalarV pi
    exp = liftME exp "exp"
    log = liftME log "log"
    sin = liftME sin "sin"
    cos = liftME cos "cos"
    tan = liftME tan "tan"
    asin = liftME asin "asin"
    acos = liftME acos "acos"
    atan = liftME atan "atan"
    sinh = liftME sinh "sinh"
    cosh = liftME cosh "cosh"
    tanh = liftME tanh "tanh"
    asinh = liftME asinh "asinh"
    acosh = liftME acosh "acosh"
    atanh = liftME atanh "atanh"
    --To raise a MatrixExt to a power we need to check:
    a ** b = do
        ra <- a
        rb <- b
        case (isScalar ra, isScalar rb) of
            --what type of value they are.
            (False, False) -> throwError $
                MatrixOperationNotSupported "Cannot raise matrix to
                matrix power!"
            --If both matrices, give up
            (True, False) -> throwError $
```

```
                MatrixOperationNotSupported "Cannot raise scalar to
                    matrix power!"
264             --If one we try to raise a scalar to a matrix power,
                    give up.
265             (False, True) -> if isIntegral (asScalar rb)
266             --A matrix to a scalar power, is only defined for
                    integral powers
267                 then product $ replicate (fromJust $ asInt $
                        asScalar rb) a
268                 --In which case we replicate the equation enough
                        times and then multiply them all together
269                 else throwError $ MatrixOperationNotSupported
                        "Cannot raise matrix to non-integral power!"
270                 --Otherwise give up
271             (True, True) -> return $ ScalarV $ (asScalar ra) **
                    (asScalar rb)
272             --A scalar to a scalar power is defined as usual.
273
274 --Defining IsIntegral for MatrixExt is slightly tricky:
275 --  * IsIntegral is well defined for ScalarV, but for MatrixV,
276 --  * we definately know that a matrix is _not_ an integral,
277 --  * but can we really  say it is not _not_ an integral?
278 --  * its not really meaningful.
279 --  * But im going to assign it false for all matrices
280 --  * This will prevent it from confusing the simplifier as well.
281 instance IsIntegral t => IsIntegral (Compute (MatrixExt t)) where
282     isIntegral a = case runCompute a of
283         Right (MatrixV _) -> False
284         Right (ScalarV t) -> isIntegral t
285         Left _ -> False
286
287     asInt a = case runCompute a of
288         Right (MatrixV _) -> Nothing
289         Right (ScalarV t) -> asInt t
290         Left _ -> Nothing
```

A.1.9 AME/COMPUTE/CALCULUS/DERIVATIVES.HS

```
1 module AME.Compute.Calculus.Derivatives (differentiate) where
2
3 import AME.Compute.Expr
```

```haskell
import AME.Compute.Error
import AME.Compute.Simplify


-- Find the partial derivative of an expression with respect to a variable.

differentiate :: (Eq var, Eq const, Floating const) => var -> Expr var const -> Compute (Expr var const)
differentiate var (Const _) = return 0
differentiate var (Var v) | v == var = return 1
differentiate var (Mul []) = return 0
-- the empty product is constant.. this _should_ only appear as an edge case for the generalized product rule
differentiate var (Mul (x:xs)) = do
    x' <- differentiate var x
    xs' <- differentiate var (Mul xs)
    return $ x' * Mul xs + xs' * x
-- d/dx (uv) = vu' + uv' (only recursively cos we have more than two thing in our product)
differentiate var (Sum xs) = Sum <$> mapM (differentiate var) xs
-- d/dx (u + v) = u' + v'
differentiate var (Div u v) = do
    u' <- differentiate var u
    v' <- differentiate var v
    return $ (u' * v - v' * u) / (v^2)
-- d/dx (u/v) = (vu' - uv')/(v^2)
differentiate var (Pow (Const c) v) = do
    v' <- differentiate var v
    return $ v' / (Const $ log c) * Pow (Const c) v
differentiate var (Pow v (Const c)) | c /= 0 = do
    v' <- differentiate var v
    return $ v' * (Const c) * (Pow v (Const $ c - 1))
differentiate var (Pow v (Const c)) | c == 0 = return 0
differentiate var (Pow u v) = do
    u' <- differentiate var u
    v' <- differentiate var v
    return $ (v' * log u + u' * v / u) * (Pow u v)
-- d/dx (u^v) = (v'log(u) + u'v/u)u^v
differentiate var (Log u) = differentiate var u >>= \u' -> return $
```

```
    u' * recip u
-- d/dx (log u) = u' * 1/u
differentiate var (Exp u) = differentiate var u >>= \u' -> return $
    u' * exp u
-- d/dx (e^u) = u'e^u
differentiate var (Sin u) = differentiate var u >>= \u' -> return $
    u' * cos u
-- d/dx (sin(u)) = u'cos(u)
differentiate var (Cos u) = differentiate var u >>= \u' -> return $
    negate $ u' * sin u
-- d/dx (cos(u)) = -u'sin(u)
differentiate var (Tan u) = differentiate var u >>= \u' -> return $
    u' * (1/cos u)^2
-- d/dx (tan(u)) = (sec(u))^2
differentiate var (ASin u) = differentiate var u >>= \u' -> return $
    u' / sqrt (1 - u^2)
-- d/dx (asin(u)) = u'/sqrt(1 - u^2)
differentiate var (ACos u) = differentiate var u >>= \u' -> return $
    negate $ u' / sqrt (1 - u^2)
-- d/dx (acos(u)) = -u'/sqrt(1 - u^2)
differentiate var (ATan u) = differentiate var u >>= \u' -> return $
    u' / (1 + u^2)
-- d/dx (atan(u)) = u'/(1 + u^2)
differentiate var (Abs u) = differentiate var u >>= \u' -> return $
    u * u' / abs u
-- d/dx |u| = uu'/|u|
```

A.1.10 AME/COMPUTE/CALCULUS/INTEGRALS.HS

```
{-# LANGUAGE RankNTypes, LambdaCase #-}

module AME.Compute.Calculus.Integrals (
    integrate,
    definiteIntegral
) where

import AME.Compute.Error
import AME.Compute.Expr
import AME.Compute.Simplify
import AME.Compute.Solver.Polynomial
import AME.Compute.Calculus.Derivatives
```

```
13 import Control.Monad
14 import Data.Maybe
15 import Data.List
16 import Data.Function
17 import Debug.Trace
18
19
20 --Integrate some functions.
21 --This is capable of integrating:
22 --  * Most rational functions (including all with denominator of
   degree < 4)
23 --  * Square roots of polynomials of degree < 3,
24 --  * Logs of most polynomials (all with degree < 4 + most others)
25 --  * Simple trig expressions (i.e sin, cos, tan, sec etc. of a*x)
26 --  * Simple applications of integration by parts (i.e e^x*sin(x) or
   x*e^x)
27 --The reason why this is limited in its polynomial parts is that I
   have no
28 --method of factorizing products of irreducible polynomials, i.e any
   polynomial
29 --that contains at most one irreducible quadratic factor should be
   doable.
30 integrate :: Simplifiable var const => var -> Expr var const ->
   Compute (Maybe (Expr var const))
31 integrate var expr = do
32     --To simplify things, we only integrate expressions with
33     --zero or one free variables. It would be easy to adapt the
          existing
34     --algorithm to handle others, but it would take a long time.
35     case freeVariables expr of
36         [(v, _)] | v == var -> dointegral
37         [] -> dointegral
38         fv -> throwError NoPartialIntegrals
39     where dointegral = do
40             --Simplify it before _and_ afterwards.
41             expr' <- simplify expr
42             let expr'' = subSingleVar var expr'
43             val <- (integrate' `thenTry` byparts) var expr''
44             --We try integration by parts seperately
45             --so that the integrate' function doesn't end up looping
```

```
                on
46              --integration by parts indefinately..
47              case val of
48                  Just val' -> Just <$> simplify val'
49                  Nothing -> return Nothing
50
51  integrate' :: IFunc var const
52  integrate' var expr = do
53      expr' <- simplify expr
54      let expr'' = subSingleVar var expr'
55      --Try all of these things in this specific order
56      --Because we need to integrate basic forms first,
57      --or else polydiv and partialfraction will loop forever.
58      val <- (basicforms `thenTry`
59              rationalforms `thenTry`
60              polydiv `thenTry`
61              partialfraction `thenTry`
62              rootforms `thenTry`
63              logforms `thenTry`
64              expforms `thenTry`
65              trigforms) var expr''
66      case val of
67          Just val' -> Just <$> simplify val'
68          Nothing -> return Nothing
69
70  --We could do a numerical definite integrator, but for now
71  --we just use the fundamental theorem of calculus to evaluate the
72  --indefinite integral at the two end points.
73  definiteIntegral :: Simplifiable var const => var -> Expr var const
        -> const -> const -> Compute (Maybe const)
74  definiteIntegral var expr a b = do
75      i <- integrate var expr
76      case i of
77          Nothing -> return Nothing
78          Just i' -> do
79              s1 <- substitute var (Const a) i'
80              s2 <- substitute var (Const b) i'
81              x1 <- evaluate s1
82              x2 <- evaluate s2
83              return $ Just $ x2 - x1
```

```
84
85 --Substitute a single variable x with a*x for some constant a
86 --to make the integrate' function more general.
87 subSingleVar :: (Num const, Eq var) => var -> Expr var const -> Expr
       var const
88 subSingleVar var (Var v) | v == var = Mul [Const 1, Var v]
89 --x = 1*x
90 subSingleVar var m@(Mul [Const _, Var v]) | v == var = m
91 --a*x = a*x
92 subSingleVar _ c@(Const _) = c
93 subSingleVar var (Sum xs) = Sum $ map (subSingleVar var) xs
94 subSingleVar var (Mul xs) = Mul $ map (subSingleVar var) xs
95 subSingleVar var (Div a b) = Div (subSingleVar var a) (subSingleVar
       var b)
96 subSingleVar var (Pow a b) = Pow (subSingleVar var a) (subSingleVar
       var b)
97 subSingleVar var (Exp a) = Exp (subSingleVar var a)
98 subSingleVar var (Log a) = Log (subSingleVar var a)
99 subSingleVar var (Abs a) = Abs (subSingleVar var a)
100 subSingleVar var (Sin a) = Sin (subSingleVar var a)
101 subSingleVar var (Cos a) = Cos (subSingleVar var a)
102 subSingleVar var (Tan a) = Tan (subSingleVar var a)
103 subSingleVar var (ASin a) = ASin (subSingleVar var a)
104 subSingleVar var (ACos a) = ACos (subSingleVar var a)
105 subSingleVar var (ATan a) = ATan (subSingleVar var a)
106
107 --An integration function, it takes a variable to integrate with
       respect to
108 --an expression to integrate, and then might return another
       expression that can
109 --include error messages.
110 type IFunc var const = Simplifiable var const => var -> Expr var
       const -> Compute (Maybe (Expr var const))
111
112 --Try one function and if it fails try another.
113 thenTry :: IFunc var const -> IFunc var const -> IFunc var const
114 thenTry a b var expr = do
115     a' <- a var expr
116     case a' of
117         Just r -> return $ Just r
```

```
118         Nothing -> b var expr
119
120 basicforms :: IFunc var const
121 basicforms var (Div _ (Const 0)) = throwError DivisionByZero
122 --int(ax) = 1/2ax^2
123 basicforms var (Mul [Const c, Var v]) | v == var = return $ Just $
        Mul [Const (c / 2), Pow (Var v) 2]
124 basicforms var (Mul (Const c:xs)) = (fmap ((Const c) *)) <$>
        (integrate' var $ Mul xs)
125 --int(k*f(x)) = k*int(f(x))
126 basicforms var (Mul [x]) = integrate' var x
127 basicforms var (Sum xs) = (fmap Sum) <$> mapM id <$> mapM
        (integrate' var) xs
128 --int(a + b) = int(a) + int(b)
129 basicforms var (Const c) = return $ Just $ (Const c) * (Var var)
130 --int(k) = kx
131 basicforms var (Var v) | v == var = return $ Just $ (1/2) * (Pow
        (Var v) 2)
132 --int(x) = 0.5x^2
133 basicforms var (Pow (Mul [Const a, Var v]) (Const (-1))) | v == var
        = return $ Just $ 1/(Const a) * log (Var v)
134 --int((ax)^-1) = 1/a * ln(x)
135 basicforms var (Pow (Mul [Const a, Var v]) (Const b)) | (var == v) =
136     return $ Just $ (Const $ a ** b / (b + 1)) * Pow (Var v) (Const
            $ b + 1)
137 --int((ax)^b) = (a^b/(b+1))*x^(b+1) for integral b /= -1
138 basicforms var (Div (Const a) (Mul [Const b, Var v])) | v == var =
        return $ Just $ (Const $ a/b) * log (Var v)
139 --int(a/bx) = (a/b) * ln(x)
140 basicforms var (Div (Const a) (Pow (Mul [Const b, Var v]) (Const
        c))) | v == var =
141     fmap ((Const a) *) <$> (basicforms var $ Pow (Mul [Const b, Var
            v]) (Const $ negate c))
142 --int(a/(bx)^c) = a * int((bx)^(-c))
143 basicforms _ _ = return Nothing
144
145 --integrate basic rational functions to make partialfraction work
146 rationalforms :: IFunc var const
147 rationalforms var (Div (Const a) b) | b `isPolynomialIn` var = case
        degree coeffs of
```

```
148         0 -> integrate var $ Const (a / c0) -- this _shouldn't_
                happen, but just in case it does..
149         1 -> return $ Just $ Const (a * recip c1) * Log (Const c1 *
                Var var + Const c0)
150         -- int(1/(ax + b)) = 1/a * ln (ax + b)
151         2 -> let ndisc = 4*c2*c0 - c1*c1
152                  srnd = sqrt ndisc in if ndisc > 0
153                     then return $ Just $ Const (2*a/srnd) * ATan
                            (Const (2*c2 / srnd) * (Var var) + Const (c1
                            / srnd))
154                     else return Nothing -- if ndisc is < 0 this
                            should have
155                     --been handled by partialfraction
156         -- int(1/(ax^2 + bx + c)) = 2/sqrt(4ac - b^2) * atan((2ax +
                b)/(sqrt(4ac - b^2)))
157         _ -> return Nothing
158     where coeffs = polyCoeffs var b
159           get = getCoeff coeffs
160           c0 = get 0
161           c1 = get 1
162           c2 = get 2
163 rationalforms var (Div (Mul [Const a, Var v]) b) | (v == var) && (b
    `isPolynomialIn` v) = case degree coeffs of
164         2 -> let ndisc = 4*c2*c0 - c1*c1
165                  srnd = sqrt ndisc in if ndisc > 0
166                     then return $ Just $ Const (a / (2*c2)) * Log b
167                         - Const (a * c1 /(c2 * srnd)) * ATan (Const
                              (2*c2 / srnd) * (Var var) + Const (c1 /
                              srnd))
168                     else return Nothing
169         _ -> return Nothing
170         -- this will be handled by the polynomial divider in polydiv
                if degree < 2
171         -- if degree > 2 then it will be split into partial
                fractions.
172     where coeffs = polyCoeffs var b
173           get = getCoeff coeffs
174           c0 = get 0
175           c1 = get 1
176           c2 = get 2
```

```
177 rationalforms _ _ = return Nothing
178
179 --Perform polynomial long division.
180 polydiv :: IFunc var const
181 polydiv var (Div (Sum xs) b) | ((Sum xs) `isPolynomialIn` var) &&
182                                (b `isPolynomialIn` var) =
183     (fmap Sum) <$> mapM id <$> mapM (integrate var) (map (/b) xs)
184 polydiv var (Div a b) | (a `isPolynomialIn` var) && (b
    `isPolynomialIn` var) =
185     case (q, r) of
186         ([], []) -> return $ Just $ Const 0
187         ([], _) -> return Nothing
188         (_, []) -> integrate var $ repoly var q
189         (_, _) -> integrate var $ repoly var q + (repoly var r)/b
190     where (q, r) = coeffDiv [] acoeffs bcoeffs
191           acoeffs = polyCoeffs var a
192           bcoeffs = polyCoeffs var b
193 polydiv _ _ = return Nothing
194
195 --Evaluate a polynomial at a given value
196 polyEval :: Floating const => [(const, const)] -> const -> const
197 polyEval coeffs val = sum $ map (\(deg, coeff) -> coeff * val **
    deg) coeffs
198
199 --Differentiate a polynomial by multiplying the coefficient
200 --by power and subtracting one from the power.
201 polyDiff :: (Eq const, Num const) => [(const, const)] -> [(const,
    const)]
202 polyDiff = map (\(deg, coeff) -> if deg == 0 then (0, 0) else (deg -
    1, coeff * deg))
203
204 --Divide one polynomial by another, using the following algorithm:
205 --   * Divide the leading term by the other leading term
206 --   * Add that to the result.
207 --   * Multiply it by the divisor.
208 --   * Subtract that from the dividend and then repeat these steps.
209 coeffDiv :: (Floating t, Ord t) => [(t, t)] -> [(t, t)] -> [(t, t)]
    -> ([(t, t)], [(t, t)])
210 coeffDiv q r d = if nonzero r && (degree r >= degree d)
211     then let t = [lead r `tdiv` lead d] in coeffDiv (q `polyAdd` t)
```

```
        (r `polySub` (t `polyMul` d)) d
212     else (q, r)
213     where nonzero [] = False
214           nonzero xs = any (/= 0) $ map snd xs
215           lead xs = maximumBy (compare `on` fst) $ filter ((/= 0) .
                 snd) xs
216           tdiv (d1, c1) (d2, c2) = (d1 - d2, c1 / c2)
217
218 --Multiply out two polynomials by multiplying one polynomial
219 --by each term of the other, and then multiplying through by that
      factor
220 polyMul :: (Num t, Ord t) => [(t, t)] -> [(t, t)] -> [(t, t)]
221 polyMul [] _ = []
222 polyMul _ [] = []
223 --                sum up.........multiply through one term....for
      each term
224 polyMul xs ys = foldl1 polyAdd $ map (\x -> map (`tmul` x) ys) xs
225     where tmul (d1, c1) (d2, c2) = (d1 + d2, c1 * c2)
226
227 --Subtract two polynmials, by subtracting the coefficients of same
      powers
228 polySub :: (Num t, Ord t) => [(t, t)] -> [(t, t)] -> [(t, t)]
229 polySub xs ys = map (\case
230           [(d, c)] -> if (d, c) `elem` ys
231               then (d, negate c)
232               else (d, c)
233           [(d1, c1), (d2, c2)] -> (d1, c1 - c2)) $ groupBy ((==)
                `on` fst) $ sortBy (compare `on` fst) $ xs ++ ys
234           -- subtract coefficients for each power ...group...similar
                powers....sorting by powers...on the union of both
                polynomials
235 --The same but for adding
236 polyAdd :: (Num t, Ord t) => [(t, t)] -> [(t, t)] -> [(t, t)]
237 polyAdd xs ys = map (\case
238           [x] -> x
239           [(d1, c1), (d2, c2)] -> (d1, c1 + c2)) $ groupBy ((==)
                `on` fst) $ sortBy (compare `on` fst) $ xs ++ ys
240
241 data T = T deriving (Eq, Ord)
242
```

```
243 coeffGCD :: (Floating t, Num t, Ord t, IsIntegral t) => [(t, t)] ->
        [(t, t)] -> [(t, t)]
244 coeffGCD a [(0, 0)] = a
245 coeffGCD a b = let (_, r) = coeffDiv [] a b in coeffGCD b
        (simplifyPoly T r)
246
247 --Perform the partial fraction algorithm on rational functions
248 --whose denominators are factorizable into linear factors and
249 --at most one quadratic.
250 --(FYI: The multiplicity of an nth root is n.. i.e a double root has
        multiplicity 2 etc.)
251 partialfraction :: IFunc var const
252 partialfraction var (Div a b) | (a `isPolynomialIn` var) &&
253                                 (b `isPolynomialIn` var) &&
254                                 (degree $ polyCoeffs var a) <
                                        (degree $ polyCoeffs var b) = do
255                                 --degree of denomiator should be <
                                        numerator otherwise its handled
                                        by polydiv
256     let rootcoeffs = map (foldl1 polyMul) $ map (\(x, m) ->
            replicate m [(1, 1), (0, negate x)]) roots
257             ---       multiply out.....................(x - a)^n
                    for an nth root a.............for each root
258     let rcoeffs = foldl1 polyMul rootcoeffs
259     --            multiply all these factors together... this is the
            denominator divided by all non-linear factors.
260     let denoms = map (\(x, m) -> (Var var + Const (negate x), m))
            roots
261     --           For each root, record its linear factor (x - a),
            and its multiplicity
262     let finaldenom = simplifyPoly var $ fst $ coeffDiv [] bcoeffs
            rcoeffs
263     --  The denominator of the non-linear component is the
            denominator divided by all the roots.
264     let finalpoly = repoly var finaldenom
265     let fracPower r m d p = do
266     -- processing each root, with a given mulitplicity, denominator,
            and the current power we are looking at.
267     -- p is not always m, because for m > 1 we consider fracPower
            for all values of p < m.
```

```
268     -- Here we use the "cover up method" for partial fractions:
269     --  * For a given factor, remove that factor from the
            denominator of the function
270     --  * Then find the root of that factor, and evaluate the new
            expression with at that root
271     -- i.e for 1/(x - 1)(x - 2), we remove (x - 1) and thus evaluate
            1/(x - 2) at the root of (x - 1),
272     -- giving the coefficient of 1/(x - 1) in the partial fraction
            expansion as 1/(1 - 2) = -1.
273             let basedenom = Mul $ finalpoly : (map (\(d, m) -> Pow d
                    (Const $ fromIntegral m)) $ filter ((/= d) . fst)
                    denoms)
274             -- First we compute the denominator of the new
                    expression by multiplying all the factors together
275             -- including the non linear factor, except for the one
                    we are looking at. (Here, d is the factor we computed
                    earlier,
276             -- and m is the multiplicity)
277             let basef' = Div a basedenom
278             -- The to compute the new expression we just divide the
                    numerator by the new denominator
279             basef <- simplify basef'
280             -- Then simplify that
281             expr <- nthderivative (m - p) basef
282             -- To compute the coefficient of a factor with power > 1,
283             -- we must slightly modify the algorithm above, as
                    follows:
284             -- If a root r has multiplicty m, then the mth
                    derivative of
285             -- the function will have a root r with multiplicty 1,
286             -- hence by taking the mth derivative we can just
                    evaluate it as normal
287             -- (I have a proof of this if you want to see it.)
288             sub <- substitute var (Const r) expr
289             val <- evaluate sub
290             let denom = Pow d (Const $ fromIntegral p)
291             let coeff = val/(fromIntegral $ factorial $ m - p)
292             -- And we have to divide by (m - p)! to correct for the
                    error incurred when taking the derivatives
293             basedenom' <- simplify $ Const coeff * basedenom
```

```
294             -- Finally we compute the modified denominator as we
                   first computed, mulitplied by our term coefficient
295             -- this is used later to compute the non-linear component
296             return $ (basedenom', denom, coeff)
297     let frac (r, m) = mapM (fracPower r m (Var var + Const (negate
            r))) [1..m]
298     --All the partial fraction terms for a given root are given by
299     --evaluating fracPower on powers between 1 and m
300     fracnumers <- concat <$> mapM frac roots
301     --Concatenate all the terms for each root
302     let fntb = simplifyPoly var $ acoeffs `polySub` (foldl1 polyAdd
            $ map (\(x, _, _) -> polyCoeffs var x) fracnumers)
303     -- Subtract all the multiplied denominators we got earlier from
            the numerator.
304     let finalnumer = simplifyPoly var $ fst $ coeffDiv [] fntb
            rcoeffs
305     -- And divide this by all the linear components to get the
            numerator of the non-linear component
306     -- Maths says this _should_ always divide evenly, but it still
            freaks me out a bit.
307     let final = Div (repoly var finalnumer') (repoly var finaldenom')
308         --So the non-linear component is just the _simplified_
               numerator divided by the denominator
309             where finalnumer' = fst $ coeffDiv [] finalnumer pgcd
310                   --To simplify these we take the gcd of them, and
                         divide both by it
311                   --effectively cancelling down the fraction.
312                   finaldenom' = fst $ coeffDiv [] finaldenom pgcd
313                   pgcd = coeffGCD finalnumer finaldenom
314     --Then to integrate each term, we follow the usual rules.
315     let intterm (Div (Const c) (Pow (Sum [Var var, Const nr]) (Const
            p))) = case p of
316             --int(c/(x + nr)) = c*log(x + nr)
317             1 -> Const c * Log (Sum [Var var, Const nr])
318             --int(c/(x + nr)^p) = c/(1 - p) * (x - nr)^(p - 1)
319             _ -> Div (Const (c / (1 - p))) (Pow (Sum [Var var, Const
                    nr]) (Const $ p - 1))
320     is <- do
321             final' <- integrate var final
322             --Integrate the non-linear term
```

```
323             let fracnumers' = map (Just . intterm) $ map (\(_, d, c)
                    -> Div (Const c) d) fracnumers
324             --Integrate all the linear terms
325             return $ final' : fracnumers'
326     let mis = sequence is
327     case mis of
328         Nothing -> return Nothing
329         Just mis' -> return $ Just $ Sum mis'
330         --If none of that failed, add them all together
331     where closeEnough e a b = abs (a - b) <= e
332           --sometimes my solving algorithm is a little unstable
333           --so we check if two numbers are _veeery_ close instead of
334           --_exactly_ equal
335           factorial 0 = 1
336           factorial p = p * factorial (p - 1)
337           --simple recursive factorial definition
338           nthderivative 0 expr = return expr
339           --the zeroth derivative is just the function
340           nthderivative n expr = do
341             expr' <- nthderivative (n - 1) expr
342             differentiate var expr'
343           --the nth derivative is the derivative of the (n - 1)th
                derivative
344           acoeffs = polyCoeffs var a
345           --the coefficients of the numerator
346           bcoeffs = polyCoeffs var b
347           -- ... and of the denominator
348           broots = solvePoly bcoeffs
349           -- solve the denominator using the specialized polynomial
                solver.
350           multiplicity coeffs n r = if closeEnough (0.000001)
                (polyEval coeffs r) 0
351             then multiplicity (polyDiff coeffs) (n + 1) r
352             else n
353           --we can use the above theorem about multiplicity to find
                the multiplicity of a root
354           --by counting how many derivatives we can take before it
                is no longer a root.
355           roots = map ((,) <*> multiplicity bcoeffs 0) broots
356 partialfraction _ _ = return Nothing
```

```
357
358 --something is polynomial in a variable if all its terms are
       monomial in that variable
359 isPolynomialIn :: (Num const, Ord const, Eq var, IsIntegral const)
       => Expr var const -> var -> Bool
360 isPolynomialIn (Sum xs) var = all (isMonomial var) xs
361 isPolynomialIn expr var = isMonomial var expr
362
363 -- The following expressions are monomials:
364 -- x, c*x, x^n, c*x^n, c*(k*x)^n, c for constant c, k and integer n
365 isMonomial :: (Num const, Ord const, Eq var, IsIntegral const) =>
       var -> Expr var const -> Bool
366 isMonomial var (Mul [Const _, Var v]) = v == var
367 isMonomial var (Mul [Const _, Pow (Var v) (Const c)]) = isIntegral c
       && (v == var) && (c >= 0)
368 isMonomial var (Var v) = v == var
369 isMonomial var (Pow (Var v) (Const c)) = isIntegral c && (v == var)
       && (c >= 0)
370 isMonomial var (Pow (Mul [Const a, Var v]) (Const c)) = (v == var)
       && isIntegral c && (c >= 0)
371 isMonomial var (Mul [Const _, Pow (Mul [Const a, Var v]) (Const c)])
       = (v == var) && isIntegral c && (c >= 0)
372 isMonomial var (Const _) = True
373 isMonomial _ _ = False
374
375 --Find the coefficients of the terms in a polynomial.
376 --Basically the same as above but recording c and k appropriately
377 polyCoeffs :: (Eq var, Floating const, IsIntegral const) => var ->
       Expr var const -> [(const, const)]
378 polyCoeffs var (Sum xs) = concatMap (polyCoeffs var) xs
379 polyCoeffs var (Var v) | v == var = [(1, 1)]
380 polyCoeffs var (Pow (Var v) (Const c)) | (v == var) && isIntegral c
       = [(c, 1)]
381 polyCoeffs var (Mul [Const a, Var v]) | v == var = [(1, a)]
382 polyCoeffs var (Mul [Const a, Pow (Var v) (Const c)]) | (v == var)
       && isIntegral c = [(c, a)]
383 polyCoeffs var (Pow (Mul [Const a, Var v]) (Const c)) | (v == var)
       && isIntegral c = [(c, a ** c)]
384 polyCoeffs var (Mul [Const b, (Pow (Mul [Const a, Var v]) (Const
       c))]) | (v == var) && isIntegral c = [(c, b * a ** c)]
```

```
385 polyCoeffs var (Const c) = [(0, c)]
386 polyCoeffs _ _ = []
387
388 --Turn a list of coefficients into a polynomial.
389 repoly :: (Fractional const, Ord const) => var -> [(const, const)]
       -> Expr var const
390 repoly var xs = let xs' = concatMap f $ filter ((not . closeEnough
       0.000001 0) . snd) xs
391         --turn terms into expressions... for each term that is not
              _almost_ 0, which we just remove.
392                 f x = case x of
393                         (0, c) -> [Const c]
394                         --Zeroth powers are constants
395                         (1, c) -> [Const c * Var var]
396                         --First powers are just variables
397                         (d, c) -> [Const c * Pow (Var var)
                              (Const d)] in
398                         --everything else is a regular power
399             case xs' of
400                 [] -> Sum [Const 0]
401                 _ -> Sum xs'
402                 --Add up all the expressions
403             where closeEnough e a b = abs (a - b) <= e
404
405 --simplify a polynomial by converting it into an expression
406 --and then back to a list of coefficients
407 simplifyPoly :: Simplifiable var const => var -> [(const, const)] ->
       [(const, const)]
408 simplifyPoly var p = polyCoeffs var $ repoly var p
409
410 --get the coefficient of a given power, returning 0 if the power is
       not present
411 getCoeff :: (Eq const, Num const) => [(const, const)] -> const ->
       const
412 getCoeff coeffs k = case lookup k coeffs of
413     Just x -> x
414     Nothing -> 0
415
416 --get the highest present power
417 degree :: (Num const, Ord const) => [(const, const)] -> const
```

```
418 --by.................filtering those that are not coefficient 0
419 degree coeffs = case filter ((/= 0) . snd) coeffs of
420     [] -> 0
421     xs -> maximum $ map fst $ xs
422     --and selecting the highest
423
424 --integrate square roots of polynomials
425 rootforms :: IFunc var const
426 rootforms var (Pow a (Const 0.5)) | a `isPolynomialIn` var = case
       degree coeffs of
427     -- These are just formulas copied from the formula book.
428     0 -> return $ Just $ Const (c0 ** 0.5) * (Var var)
429     --except this one, which really _should_ be handled by the
          simplification algorithm
430     --but just in case it doesnt, we integrate sqrt(c) as sqrt(c)x
          for a constant c
431     1 -> return $ Just $ (Const (2*c0/(3*c1)) + 2*(Var var)/3) * Pow
          (Const c0 + (Var var)*Const c1) (0.5)
432     2 -> return $ Just $ (Const c1 + (Const $ 2*c2)*(Var
          var))/(Const $ 4*c2)
433               * Pow ((Const c2)*(Var var)**2 + (Const c1)*(Var var) +
                     (Const c0)) 0.5 +
434               (Const $ (4*c2*c0 - c1*c1)/(8*c2**1.5)) * Log ((Const $
                     2*c2)*(Var var) + (Const c1)
435               + 2*Pow (Const (c2*c2) * Pow (Var var) 2 + Const
                     (c1*c2) * (Var var) + Const (c2*c0)) 0.5)
436     _ -> return Nothing
437     where coeffs = polyCoeffs var a
438           get = getCoeff coeffs
439           c0 = get 0
440           c1 = get 1
441           c2 = get 2
442 rootforms _ _ = return Nothing
443
444 --integrate logs of some polymomials
445 logforms :: IFunc var const
446 logforms var (Log a) | a `isPolynomialIn` var = case degree coeffs of
447     0 -> if c0 > 0
448             then return $ Just $ Const (log c0) * (Var var)
449             else return Nothing
```

```
450     --for constants, we first check the argument is > 0 as log is
            not defined otherwise
451     --and then we integrate log(c) as xlog(c) where c is constant
452     1 -> return $ Just $ (Var var + Const (c0/c1)) * Log(Const c1 *
            Var var + Const c0) - Var var
453     --int(log(ax + b)) = (x + b/a)log(ax + b) - x
454     2 -> let ndisc = 4*c2*c0 - c1*c1
455              srnd = sqrt ndisc in if ndisc > 0
456         --for quadratics, if it has no solutions
457         then return $ Just $ Const (srnd/c2) * ATan (((Const $ 2*c2)
                * (Var var) + (Const c1)) / (Const srnd)) -
458               2*(Var var) + (Const (c1 / (2 * c2)) + (Var var)) * Log
                    (Const c2 * (Var var) ** 2 + Const c1 * (Var var) +
                    Const c0)
459         --use atan to integrate it (a long and truly boring formula)
460         else case solvePoly coeffs of
461             --otherwise solve it and
462             [] -> error "we just ruled this out above, duh"
463             --if its a repeated root, bring down the power as a
                    multiple and integrate
464             [r] -> return $ Just $ Const (2 * c2) * (Var var - Const
                    r) * Log (Var var - Const r) - 2 * Var var
465             --or if it has two roots, split it up into logs of
                    linear factors and integrate
466             [r1, r2] -> return $ Just $  Const c2 * ((Var var -
                    Const r1) * Log (Var var - Const r1) + (Var var -
                    Const r2) * Log (Var var - Const r2) - 2 * Var var)
467     --for general polynomials, solve it
468     _ -> case solvePoly coeffs of
469         [] -> return Nothing
470         --if this is a product of only quadratic factors, give up
471         xs -> do
472             let rcoeffs = foldl1 polyMul $ map (\x -> [(1, 1), (0,
                    negate x)]) xs
473             --otherwise, multiply out all the linear factors
474             let fcoeffs = fst $ coeffDiv [] coeffs rcoeffs
475             --and divide the polynomial by it, leaving the remaining
                    non-linear factor
476             final' <- logforms var (Log $ repoly var $ fcoeffs)
477             --try and integrate this non-linear bit
```

```
478             case final' of
479                 Just final -> return $ Just $ final + Sum (map
                        intlin xs)
480                 --and if we can, integrate the linear bits and add
                        it together
481                 Nothing -> return Nothing
482     where coeffs = polyCoeffs var a
483           get = getCoeff coeffs
484           c0 = get 0
485           c1 = get 1
486           c2 = get 2
487           intlin x = (Var var - Const x) * Log (Var var - Const x) -
                  Var var
488           --int(x - c) = (x - c)log(x - c) - x
489 logforms _ _ = return Nothing
490
491 expforms :: IFunc var const
492 expforms var (Exp (Mul [Const a, Var v])) | v == var = return $ Just
       $ Mul [Const (recip a), Exp (Mul [Const a, Var v])]
493 --int(exp(ax)) = 1/a * exp(ax)
494 expforms _ _ = return Nothing
495
496 --Integrate a _few_ trig functions
497 --Anything more complicated really requires substitution
498 --which is waaaaaaay to sophisticated for the current system to
       handle,
499 --especially the simplification engine.
500 trigforms :: IFunc var const
501 trigforms var (Cos (Mul [Const a, Var v])) | v == var = return $
       Just $ Mul [Const (recip a), Sin (Mul [Const a, Var v])]
502 --int(cos(ax)) = 1/a sin(ax)
503 trigforms var (Sin (Mul [Const a, Var v])) | v == var = return $
       Just $ Mul [Const (negate $ recip a), Cos (Mul [Const a, Var v])]
504 --int(sin(ax)) = -1/a cos(ax)
505 trigforms var (Tan (Mul [Const a, Var v])) | v == var = return $
       Just $ Mul [Const (negate $ recip a), Log (Cos (Mul [Const a, Var
       v]))]
506 --int(tan(ax)) = -1/a log(cos(ax))
507 trigforms var (Div 1 (Cos (Mul [Const a, Var v]))) | v == var =
       return $ Just $
```

```
508     Mul [Const (recip a), Log (Div 1 (Cos $ Mul [Const a, Var v]) +
            Tan (Mul [Const a, Var v]))]
509 --int(sec(ax)) = 1/a log(sec(ax) + tan(ax))
510 trigforms var (Div 1 (Sin (Mul [Const a, Var v]))) | v == var =
    return $ Just $
511     Mul [Const (recip a), Log (Div 1 (Sin $ Mul [Const a, Var v]) +
            Div 1 (Tan (Mul [Const a, Var v])))]
512 --int(cosec(ax)) = 1/a log(cosec(ax) + cot(ax))
513 trigforms var (Div 1 (Tan (Mul [Const a, Var v]))) | v == var =
    return $ Just $
514     Mul [Const (recip a), Log (Sin (Mul [Const a, Var v]))]
515 --int(cot(ax)) = 1/a log(sin(ax))
516 trigforms var (ATan (Mul [Const a, Var v])) | v == var = return $
    Just $ Sum [
517     Mul [Var var, ATan (Mul [Const a, Var var])], Mul [-0.5, Log
            (Sum [Pow (Var var) 2, 1])]]
518 --int(atan(ax)) = x atan(ax) - 1/2 log(x^2 + 1)
519 trigforms _ _ = return Nothing
520
521 --Simple integration by parts
522 --This will do recursive integration by parts
523 --operating either until 5 steps are performed
524 --at which point it gives up
525 --or until either we get a multiple of our original function
526 --or a function we know how to integrate
527 byparts :: IFunc var const
528 --don't try and integrate cf(x) for c constant, or it will fail.
529 byparts var (Mul [Const _, _]) = return Nothing
530 byparts var a@(Mul xs) = do
531     sa <- simplify a
532     let u:vs' = sortOn ilate xs
533     --use the ilate rules to sort out our choice for u
534     v' <- simplify $ Mul vs'
535     --multiply together all the vs and simplify to get our v'
536     let step (u, v', coeff, _) = do
537     --one step of integration by parts consists of
538             du <- differentiate var u
539             --differentiating u
540             u' <- simplify du
541             maybeV <- integrate' var v'
```

```
542             --and trying to integrate v'
543             case maybeV of
544                 Nothing -> return Nothing
545                 --if this doesn't work, we give up
546                 Just v -> do
547                     s <- simplify $ Mul [Const (negate coeff), u', v]
548                     --otherwise we multiply these two together, and
                        multiply by
549                     -- -1 or 1 depending on which step we are on.
550                     return $ Just (u', v, negate coeff, s)
551     let testSame (_, _, _, s) = return $ noCoeff s `eqStruc` noCoeff
            sa
552     --test if we got our original function again
553     let testIntegrable (_, _, _, s) = isJust <$> (integrate' var s)
554     --test if we have an easily integrable function
555     let testCombined v = do
556     --test both tests and if one works, return a string indicating
            which.
557             same <- testSame v
558             if same
559                 then return $ Just "same"
560                 else do
561                     integrable <- testIntegrable v
562                     if integrable
563                         then return $ Just "integrable"
564                         else return Nothing
565     let mulTerm (u, _, coeff, _) (_, v, _, _) =  Mul [Const coeff,
            u, v]
566     let sumUp steps = Sum $ zipWith mulTerm steps (tail steps)
567     let startVal = (u, v', 1, a)
568     --start on our inital u and v, and coefficient 1
569     maybeSteps <- iterateWhileMaybeWithMaxM 5 step testCombined
            startVal
570     --try iterating step while our test is not satisfied but also
            does not fail, with a maximum number
571     --of iterations of 5
572     case maybeSteps of
573         Nothing -> return Nothing
574         --if this fails, give up
575         Just (method, steps) -> do
```

```
576             --else check which method we got
577             let result = sumUp $ startVal : steps
578             --and sum up all the steps
579             let (_, _, _, finalexpr) = last steps
580             case method of
581                 "integrable" -> do
582                     maybeFinal <- integrate var finalexpr
583                     --if its an integrable function, we need to
                        integrate the last term
584                     case maybeFinal of
585                         Nothing -> return Nothing
586                         Just final -> do
587                             s <- simplify $ result + final
588                             return $ Just s
589                 "same" -> do
590                     --if its a multiple of our original, we move it
                        to the other side of the equation
591                     --and solve for our original integral
592                     s <- simplify $ Const (recip $ (1 - (onlyCoeff
                        finalexpr / onlyCoeff a))) * result
593                     return $ Just s
594     where ilate (ACos _) = 0
595           ilate (ASin _) = 0
596           ilate (ATan _) = 0
597           --ilate rules say inverse trig goes first
598           ilate (Log _) = 1
599           --then logs
600           ilate a | a `isPolynomialIn` var = 2
601           ilate (Div a b) | (a `isPolynomialIn` var) && (b
                `isPolynomialIn` var) = 2
602           --then polynomials and rational functions
603           ilate (Sin _) = 3
604           ilate (Cos _) = 3
605           ilate (Tan _) = 3
606           --then trig
607           ilate (Exp _) = 4
608           --then exponential
609           ilate _ = 5
610           --to check if two things are structurally equal
611           --we perform our usual check, but sorting the elements
```

```
                first
612             --to avoid the order of our terms messing up our test.
613             eqStruc (Sum a) (Sum b) = (length a == length b) && (all
                    id $ zipWith eqStruc (sort a) (sort b))
614             eqStruc (Mul a) (Mul b) = (length a == length b) && (all
                    id $ zipWith eqStruc (sort a) (sort b))
615             eqStruc (Pow a b) (Pow c d) = (a `eqStruc` c) && (b
                    `eqStruc` d)
616             eqStruc (Div a b) (Div c d) = (a `eqStruc` c) && (b
                    `eqStruc` d)
617             eqStruc (Sin a) (Sin b) = a `eqStruc` b
618             eqStruc (Cos a) (Cos b) = a `eqStruc` b
619             eqStruc (Tan a) (Tan b) = a `eqStruc` b
620             eqStruc (ASin a) (ASin b) = a `eqStruc` b
621             eqStruc (ACos a) (ACos b) = a `eqStruc` b
622             eqStruc (ATan a) (ATan b) = a `eqStruc` b
623             eqStruc (Exp a) (Exp b) = a `eqStruc` b
624             eqStruc (Log a) (Log b) = a `eqStruc` b
625             eqStruc (Abs a) (Abs b) = a `eqStruc` b
626             eqStruc (Const a) (Const b) = a == b
627             eqStruc (Var a) (Var b) = a == b
628             eqStruc _ _ = False
629 byparts _ _ = return Nothing
630
631 --Remove the coefficients from a monomial
632 noCoeff :: Num const => Expr var const -> Expr var const
633 noCoeff (Const _) = 1
634 noCoeff (Mul [Const _, a]) = a
635 noCoeff (Mul ((Const _):xs)) = Mul xs
636 noCoeff x = x
637
638 --Take only the coefficient from a monomial
639 onlyCoeff :: Num const => Expr var const -> const
640 onlyCoeff (Const c) = c
641 onlyCoeff (Mul ((Const c):_)) = c
642 onlyCoeff _ = 1
643
644 --The following function performs the following equivalent
        imperative code:
645 --      x = val
```

```
646 --      i = 0
647 --      while(!test(x) && i < max) {
648 --          x = func(x)
649 --          i++
650 --      }
651 --      if (i < max) {
652 --          return x
653 --      } else {
654 --          raise exception
655 --      }
656 --where both test and func can raise exceptions
657 iterateWhileMaybeWithMaxM :: Monad m => Int -> (a -> m (Maybe a)) ->
        (a -> m (Maybe b)) -> a -> m (Maybe (b, [a]))
658 iterateWhileMaybeWithMaxM 0 _ _ _ = return Nothing
659 iterateWhileMaybeWithMaxM max func test val = do
660     maybeRet <- func val
661     case maybeRet of
662         Nothing -> return Nothing
663         Just ret -> do
664             done <- test ret
665             case done of
666                 Just token -> return $ Just (token, [ret])
667                 Nothing -> do
668                     rest <- iterateWhileMaybeWithMaxM (max - 1) func
                            test ret
669                     case rest of
670                         Just (token, rest) -> return $ Just (token,
                                ret : rest)
671                         Nothing -> return Nothing
```

A.1.11 AME/COMPUTE/GEOMETRY.HS

```
1 module AME.Compute.Geometry (
2   Shape(..),
3   Cartesian2(..),
4   linePointGradient,
5   linePointPoint,
6   lineInterceptGradient,
7   circleCenterRadius,
8   circleCenterPoint,
9   equationOf,
```

```
10   intersectionOf
11 ) where
12
13 import AME.Compute.Expr
14 import AME.Compute.Error
15 import AME.Compute.Solver.Polynomial
16 import AME.Compute.Solver
17 import AME.Compute.Simplify
18
19 -- Variables parameterising a cartesian equation in two dimensions
20 data Cartesian2 = X | Y deriving (Eq, Show, Ord)
21
22 --Represents lines as a point and a gradient
23 --and circles a center and radius
24 data Shape t = Circle (t, t) t
25              | Line (t, t) t
26              deriving (Show, Eq)
27
28 --an intercept of c is just the point (0, c)
29 --so put it straight into our representation
30 lineInterceptGradient :: Num t => t -> t -> Shape t
31 lineInterceptGradient c m = Line (0, c) m
32
33 linePointGradient :: (t, t) -> t -> Shape t
34 linePointGradient = Line
35
36 --the line between two points simplify starts at either point
37 --and has the gradient delta y/delta x
38 linePointPoint :: Fractional t => (t, t) -> (t, t) -> Shape t
39 linePointPoint (x1, y1) (x2, y2) = linePointGradient (x1, y1) $ (y2
       - y1) / (x2 - x1)
40
41 circleCenterRadius :: (t, t) -> t -> Shape t
42 circleCenterRadius = Circle
43
44 --Work out the radius by computing the distance from the center to
       the point
45 --with pythagoras
46 circleCenterPoint :: Floating t => (t, t) -> (t, t) -> Shape t
47 circleCenterPoint (x1, y1) (x2, y2) = Circle (x1, y1) (sqrt $ (x1 -
```

```
    x2) ** 2 + (y1 - y2) ** 2)
48
49 --The equation of a circle is given by x^2 + y^2 = r^2
50 --and for a line is given by (y - y1) = m(x - x1) and hence y = mx -
    mx1 + y1
51 equationOf :: Num t => Shape t -> Equation Cartesian2 t
52 equationOf (Circle (cx, cy) r) = Equation ((Var X - Const cx)^2 +
    (Var Y - Const cy)^2)  ((Const r)^2)
53 equationOf (Line (ox, oy) m) = Equation (Var Y) ((Const m) * (Var X
    - Const ox) + (Const oy))
54
55 intersectionOf :: (Ord t, Floating t, IsIntegral t) => Shape t ->
    Shape t -> Compute [(t, t)]
56 intersectionOf (Line (ax, ay) am) (Line (bx, by) bm) = if am == bm
57     --if they are parallel
58     then if (ax == bx) && (ay == by)
59         then throwError IntersectsEverywhere
60         --they're either the same
61         else return []
62         --or don't intersect
63     else return [(x, y)]
64     --otherwise find the intersection point
65     where x = (bm*bx - am*ax + ay - by)/(bm - am)
66           y = am * (x - ax) + ay
67 intersectionOf (Line (px, py) m) (Circle (cx, cy) r) = return $ map
    (\t -> (px + t, py + m*t)) ts
68 --trying to find the intersection of a line and a circle results in
    a quadratic in the relative
69 --distances between the points
70     where ts = solveQuadratic (1 + m*m) (2*(dx + m*dy)) (dx*dx +
        dy*dy - r*r)
71           dx = px - cx
72           dy = py - cy
73 intersectionOf a@(Circle _ _) b@(Line _ _) = intersectionOf b a
74 intersectionOf c@(Circle (ax, ay) ar) (Circle (bx, by) br) | ay /=
    by = do
75 --Two intersect two circles that are not in the same horizontal line
76     lhs' <- substitute Y sub lhs
77     xs <- solve X $ Equation lhs' rhs
78     --make a substituteion and solve
```

```
79      ys <- mapM (>>= evaluate) $ map (\x -> substitute X (Const x)
            sub) xs
80      --then evaluate those x values to determine the whole coordinates
81      return $ zip xs ys
82      where sub = Const (((br*br - ar*ar) - (bx*bx - ax*ax) - (by*by -
            ay*ay)) / (2*(ay - by))) + Mul [Const ((bx - ax)/(ay - by)),
            Var X]
83            --the substitution (br^2 - ar^2) - (bx^2 - ax^2) - (ay^2 -
                ay^2) / 2*(ay - by) + (bx - ax)/(ay - by)x
84            --transforms the equation of a circle into an equation in
                one variable.
85          Equation lhs rhs = equationOf c
86 --same thing but subbing for X instead of Y
87 intersectionOf c@(Circle (ax, ay) ar) (Circle (bx, by) br) | ax /=
       bx = do
88      lhs' <- substitute Y sub lhs
89      ys <- solve X $ Equation lhs' rhs
90      xs <- mapM (>>= evaluate) $ map (\y -> substitute Y (Const y)
            sub) ys
91      return $ zip xs ys
92      where sub = Const (((br*br - ar*ar) - (bx*bx - ax*ax) - (by*by -
            ay*ay)) / (2*(ax - bx))) + Mul [Const ((by - ay)/(ax - bx)),
            Var Y]
93            Equation lhs rhs = equationOf c
94 --otherwise they intersect everywhere
95 intersectionOf c@(Circle (ax, ay) ar) (Circle (bx, by) br) | (ay ==
       by) && (ax == bx) && (ar == br) = throwError IntersectsEverywhere
```

A.1.12 AME/COMPUTE/STATISTICS.HS

```
1  module AME.Compute.Statistics where
2
3  import AME.Compute.Error
4  import Data.Function
5  import Data.List
6  import Data.Maybe
7  import Data.Ord
8
9  --A data table has some columns with variable names and then rows of
       data.
10 data Table var val = Table [var] [[val]]
```

```
11                    deriving (Eq, Show)
12
13 --get the values of a particular variable in a table
14 dataValues :: Eq var => var -> Table var val -> [val]
15 dataValues var (Table labels dat) = snd $ head $ filter ((== var) .
       fst) $ zip labels dat
16
17 --compute the mean as sigma x / n
18 mean :: (Eq var, Fractional val) => var -> Table var val -> val
19 mean var tbl = (sum xs) / (fromIntegral $ length xs)
20     where xs = dataValues var tbl
21
22 --the sample mean is the same
23 sampleMean :: (Eq var, Fractional val) => var -> Table var val -> val
24 sampleMean var tbl = (sum xs) / (fromIntegral (length xs))
25     where xs = dataValues var tbl
26
27 --compute the median
28 median :: (Eq var, Ord val, Fractional val) => var -> Table var val
       -> val
29 median var tbl = median' xs
30     where xs = sort $ dataValues var tbl
31           median' [a] = a
32           median' [a, b] = (a + b) / 2
33           median' (_:xs) = median' (init xs)
34
35 --compute the mode by finding the element thats mentioned the most
       times
36 mode :: (Eq var, Ord val, Eq val) => var -> Table var val -> val
37 mode var tbl = last $ sortOn count $ sortOn Down $ nub xs
38  --by taking one of...finding the longest sublist.. of data grouped
       by its values
39     where xs = dataValues var tbl
40           count v = length $ filter (== v) xs
41
42 --compute variance as sigma x^2/n - xbar^2
43 variance :: (Eq var, Fractional val) => var -> Table var val -> val
44 variance var tbl = (sum $ map (^2) xs) / (fromIntegral $ length xs)
       - (sum xs)^2 / ((fromIntegral $ length xs)^2)
45     where xs = dataValues var tbl
```

```
46
47 --and sample variance is the same except times n/(n - 1)
48 sampleVariance :: (Eq var, Fractional val) => var -> Table var val
       -> val
49 sampleVariance var tbl = ((sum $ map (^2) xs) - (sum xs)^2 /
       (fromIntegral $ length xs)) / (fromIntegral (length xs) - 1)
50         where xs = dataValues var tbl
51
52 --standard deviation = sqrt(variance)
53 standardDeviation :: (Eq var, Floating val) => var -> Table var val
       -> val
54 standardDeviation var tbl = sqrt $ variance var tbl
55
56 --sample standard deviation = sqrt(sample variance)
57 sampleStandardDeviation :: (Eq var, Floating val) => var -> Table
       var val -> val
58 sampleStandardDeviation var tbl = sqrt $ sampleVariance var tbl
59
60 --compute the chi-squared value by doing sigma (o - e)^2/e for all
       observed, o,
61 --and expected, e, values.
62 chiSquared :: (Eq var, Fractional val) => var -> var -> Table var
       val -> val
63 chiSquared vara varb tbl = sum $ zipWith (/) (map (^2) $ zipWith (-)
       xs ys) ys
64 --                          sum ..divide by e..square.....subtract o
       from e
65     where xs = dataValues vara tbl
66           ys = dataValues varb tbl
67
68 --pearson correlation is the pmcc of the raw data
69 pearsonCorrelation :: (Eq var, Floating val) => var -> var -> Table
       var val -> val
70 pearsonCorrelation vara varb tbl = pearsonCorrelation' xs ys
71     where xs = dataValues vara tbl
72           ys = dataValues varb tbl
73
74 -- the product moment correlation coefficient (pmcc)
75 pearsonCorrelation' :: Floating t => [t] -> [t] -> t
76 pearsonCorrelation' xs ys = (s xs ys) / (sqrt $ (s xs xs) * (s ys
```

```
    ys))
77          --r = sxy / sqrt(sxx * syy)
78      where s xs ys = (sum $ zipWith (*) xs ys) - (sum xs) * (sum ys)
            / (fromIntegral $ length xs)
79          --sab = sigma a*b - sigma a * sigma b / n
80
81  --spearman correlation is almost the same, except its the pmcc of
    the ranks
82  spearmanCorrelation :: (Eq var, Eq val, Ord val, Floating val) =>
    var -> var -> Table var val -> val
83  spearmanCorrelation vara varb tbl = pearsonCorrelation' xs' ys'
84      where xsorted = zip (sort xs) ([0..] :: [Int])
85            ysorted = zip (sort ys) ([0..] :: [Int])
86            --sort them first so they're in the right order
87            xs' = map (\x -> fromIntegral $ fromJust $ lookup x
                  xsorted) xs
88            ys' = map (\y -> fromIntegral $ fromJust $ lookup y
                  ysorted) ys
89            --find the ranks of the values
90            xs = dataValues vara tbl
91            ys = dataValues varb tbl
```

A.2 INTERPRET COMPONENT

A.2.1 AME/INTERPRET/AST.HS

```
1  module AME.Interpret.AST (
2      Number,
3      Value(..),
4      Expression(..),
5      Statement(..)
6  ) where
7
8  -- This module contains a representation of all the possible values
   that a calculation could take
9  -- as well as all possible operations that can be performed.
10
11 import AME.Compute.Expr
12 import AME.Compute.Error
13 import AME.Compute.Matrix
14 import AME.Compute.Geometry
```

```
15 import AME.Compute.Statistics
16
17 -- The main number type. This consists of Double extended
18 -- with matrices of Doubles (MatrixExt Double) and then wrapped
19 -- in Compute to allow possible failure of arithmetic operations.
20 -- This lets us use the Num and Fractional implementations for
21 -- Num a => Num (Compute (MatrixExt a)) etc. that were defined in
       ame-compute
22 type Number = Compute (MatrixExt Double)
23
24 -- All the type of values that a computation could take.
25 data Value = Number Number
26            | Shape (Shape Number)
27            | Expr (Expr String Number)
28            | Eqn (Equation String Number)
29            | List [Value]
30            | Tbl (Table String Number)
31            | Point Number Number
32           deriving (Show, Eq)
33
34 -- All the possible types of computation that can be represented
35 -- Most of these are direct wrappers around computations from
       ame-compute
36 -- And a few others such as Variable which represents a variable,
37 -- Value represents a constant value, and MakeVariable a computation
       to
38 -- produce a fresh unbound variable.
39 data Expression = Variable String
40                 | MakeVariable String
41                 | Value Value
42                 | EquationOf Expression
                      -- Begin AME.Compute.Geometry
43                 | IntersectionOf Expression Expression
44                 | LinePointGradient Expression Expression
45                 | LineInterceptGradient Expression Expression
46                 | LinePointPoint Expression Expression
47                 | MakePoint Expression Expression
48                 | CircleCenterRadius Expression Expression
49                 | CircleCenterPoint Expression Expression
                      -- End AME.Compute.Geometry
```

```
50                  | MakeEquation Expression Expression
                      -- Begin AME.Compute.Expr
51                  | MulE Expression Expression
52                  | SumE Expression Expression
53                  | DivE Expression Expression
54                  | PowE Expression Expression
55                  | SubE Expression Expression
56                  | SinE Expression
57                  | CosE Expression
58                  | TanE Expression
59                  | AbsE Expression
60                  | ExpE Expression
61                  | LogE Expression
62                  | ASinE Expression
63                  | ACosE Expression
64                  | ATanE Expression
65                  | SqrtE Expression
                      -- End AME.Compute.Expr
66                  | Substitute String Expression Expression
                      -- Begin AME.Compute.Simplify
67                  | Evaluate Expression
                      -- End AME.Compute.Simplify
68                  | Solve Expression (Maybe String)
                      -- Begin AME.Compute.Solver
69                  | SolveSimultaneous [Expression]
                      -- End AME.Compute.Solver
70                  | Integral Expression
                      -- Begin AME.Compute.Calculus.Integrals
71                  | DefiniteIntegral Expression Expression Expression
                      -- End AME.Compute.Calculus.Integrals
72                  | Derivative Expression (Maybe String)
                      -- AME.Compute.Calculus.Derivatives
73                  | Mean String Expression
                      -- Begin AME.Compute.Statistics
74                  | SampleMean String Expression
75                  | Mode String Expression
76                  | Median String Expression
77                  | Variance String Expression
78                  | SampleVariance String Expression
79                  | StdDev String Expression
```

```
80                  | SampleStdDev String Expression
81                  | ChiSquared String String Expression
82                  | PearsonCorrelation String String Expression
83                  | SpearmanCorrelation String String Expression
                      -- End AME.Compute.Statistics
84                  | DotProduct Expression Expression
                      -- Begin AME.Compute.Matrix
85                  | Magnitude Expression
86                  | Determinant Expression
87                  | InverseMatrix Expression
88                  | AngleBetween Expression Expression
89                  | CharacteristicEquation Expression
90                  | Eigenvalues Expression
91                  | Eigenvectors Expression
                      -- End AME.Compute.Matrix
92                deriving (Show, Eq)
93
94 -- The main datatype which the interpreter works with:
95 -- Assign takes an expression and tries to assign it
96 -- to either a single variable or several variables. If it is the
    expression
97 -- is a list it unpacks the list so each variable has a different
    list element.
98 -- If it is not a list it expects there to be only one variable.
99 -- AssignArgs defines a new function with the given string as a name
100 -- and defines a new variable for each argument name string.
101 data Statement = Assign [String] Expression
102                | AssignArgs String [String] Expression
103                | Expression Expression
104               deriving (Show, Eq)
```

A.2.2 AME/INTERPRET.HS

```
1 {-# LANGUAGE LambdaCase #-}
2
3 module AME.Interpret (
4   Env(..),
5   InterpretError(..),
6   Interpret,
7   Type(..),
8   runInterpret,
```

```
 9   liftCompute,
10   eval,
11   exec
12 ) where
13
14 import Control.Monad.Trans.Except
15 import Control.Monad.Trans.State
16 import Control.Monad.Trans.Class
17 import Control.Monad
18 import Data.Maybe
19 import Data.List
20 import AME.Compute.Error
21 import AME.Compute.Expr
22 import AME.Compute.Simplify
23 import AME.Compute.Geometry
24 import AME.Compute.Solver
25 import AME.Compute.Calculus.Integrals
26 import AME.Compute.Calculus.Derivatives
27 import AME.Compute.Statistics
28 import AME.Compute.Matrix
29 import AME.Interpret.AST
30
31 --An error that can occur in the interpreter.
32 --Includes all the matrix and numerical errors from ame-compute
33 --As well as some semantic errors.
34 data InterpretError = NumericalError NumericalError
35                     | TypeError [Type] Type String
36                     | UnpackError String
37                     | NoVariable String
38                     | SpecifyVariable
39                     | CantIntegrate
40                     | OnlyForMatrices
41                     deriving (Eq, Show)
42
43 --A scope, that contains a list of names and values for variable.
44 newtype Env = Env [(String, Value)]
45             deriving (Show, Eq)
46
47 --The main interpreter monad, has ExceptT InterpretError (can raise
      exceptions of type InterpretError)
```

```
48 --and State Env (keeps a mutable copy of an Env)
49 type Interpret a = ExceptT InterpretError (State Env) a
50
51 --The type of all the types of data.
52 --Mainly used for generating nice error messages.
53 data Type = NumberT
54           | ShapeT
55           | ExprT
56           | EqnT
57           | ListT
58           | TblT
59           | PointT
60           deriving (Show, Eq)
61
62 --Take an initial value for the mutable Env, an Interpret value
63 --and return the final value of the Env, and either an answer or an
       error
64 runInterpret :: Env -> Interpret a -> (Either InterpretError a, Env)
65 runInterpret env f = runState (runExceptT f) env
66
67 --Take a Compute expression and run it in the interpreter
68 --this is basically a Except NumericalError, so all we need to do
69 --is wrap it in an identity State Env (return) and then map the
70 --NumericalError to the appropriate InterpretError
71 liftCompute :: Compute a -> Interpret a
72 liftCompute a = case runCompute a of
73     Left err -> throwError $ NumericalError err
74     Right a -> return a
75
76 --Get a variable value from the interpreter or throw an error if it
       doesnt exist.
77 getVar :: String -> Interpret Value
78 getVar s = do
79     Env vars <- lift get
80     case lookup s vars of
81         Just val -> return val
82         Nothing -> throwError $ NoVariable s
83
84 typeOf :: Value -> Type
85 typeOf (Number _) = NumberT
```

```
86 typeOf (Shape _) = ShapeT
87 typeOf (Expr _) = ExprT
88 typeOf (Eqn _) = EqnT
89 typeOf (List _) = ListT
90 typeOf (Tbl _) = TblT
91 typeOf (Point _ _) = PointT
92
93 --Ensure that val has one of the given types, and throw a TypeError
       if it does not.
94 ensure :: [Type] -> Value -> String -> Interpret ()
95 ensure ty val name = if realty `elem` ty
96     then return ()
97     else throwError $ TypeError ty realty name
98     where realty = typeOf val
99
100 --Convert a mathematical expression into an Expr (the underlying
       representation for mathematical expressions)
101 --with two functions, one for if the value in the expression is not
       constant
102 --and one to provide a shortcut for when it is constant.
103 --I.e liftExpr a Sin sin "sin" will use Sin to form a symbolic
       expression involving Sin when the argument
104 --is not constant, but will just evaluate sin a when a is constant.
105 liftExpr :: Expression -> (Expr String Number -> Expr String Number)
       -> (Number -> Number) -> String -> Interpret Value
106 liftExpr a func numfunc name = do
107     a' <- eval a
108     ensure [NumberT, ExprT] a' $ "finding " ++ name
109     --ensure the argument is either a number or an expression
110     case a' of
111         Number a'' -> return $ Number $ numfunc a''
112         --if its constant, use the shortcut.
113         Expr a'' -> do
114             let fa = func a''
115             --otherwise make the symbolic expression
116             fa' <- liftCompute $ simplify fa
117             --simplify it
118             return $ Expr fa'
119             --and return that instead.
120
```

```
121 --Lift an expression into a function that takes a variable, and a
       table.
122 liftTable :: String -> Expression -> (String -> Table String Number
       -> Number) -> String -> Interpret Value
123 liftTable s t f n = do
124     t' <- eval t
125     ensure [TblT] t' $ "finding a " ++ n
126     --We only want the argument to be a table
127     let Tbl t'' = t'
128     return $ Number $ f s t''
129     --Then apply our function
130
131 --The same as above but with two variables and a table.
132 liftTable2 :: String -> String -> Expression -> (String -> String ->
       Table String Number -> Number) -> String -> Interpret Value
133 liftTable2 a b t f n = do
134     t' <- eval t
135     ensure [TblT] t' $ "finding a " ++ n
136     let Tbl t'' = t'
137     return $ Number $ f a b t''
138
139 --Almost the same thing, except this is for functions of matrices
140 --The difference here is that we only need to ensure that the
       argument
141 --is a number... if the operation can't be applied to matrices, the
       underlying function
142 --_should_ catch that itself.
143 liftMatrix :: Expression -> (Number -> Number) -> String ->
       Interpret Value
144 liftMatrix a f s = do
145     a' <- eval a
146     ensure [NumberT] a' $ "finding a " ++ s
147     let Number a'' = a'
148     return $ Number $ f a''
149
150 -- This almost certainly not the best way to design this system.
151 -- Initially I had thought about some kind of system which combined
       the
152 -- parser, interpreter and typechecker for each case, so we could
       write one combined
```

```
153 -- expression for each case, but this would be more difficult to
    implement,
154 -- so instead we just look throught each case of the huge Expression
    type
155 -- and evaluate them seperately.
156 eval :: Expression -> Interpret Value
157 eval (Variable s) = getVar s
158 eval (MakeVariable s) = return $ Expr $ Var s
159 --Making a new variable just returns a var expression
160 eval (Value v) = return v
161 --Values evaluate to themselves.
162 eval (SumE a b) = do
163     a' <- eval a
164     b' <- eval b
165     ensure [NumberT, ExprT] a' "adding"
166     ensure [NumberT, ExprT] b' "adding"
167     --Adding two values, we want them to be both either expressions
        or numbers
168     case (a', b') of
169         (Number a'', Number b'') -> return $ Number $ a'' + b''
170         --If they are both numbers, evaluate the sum now
171         (Number a'', Expr b'') -> Expr <$> liftCompute (simplify $
            Sum [Const a'', b''])
172         (Expr a'', Number b'') -> Expr <$> liftCompute (simplify $
            Sum [Const b'', a''])
173         (Expr a'', Expr b'') -> Expr <$> liftCompute (simplify $ Sum
            [a'', b''])
174         --Otherwise make a symbolic expression, lifting numbers into
            expressions where needed.
175 --Do the exact same thing for multiplying, dividing and
    exponentiating.
176 --I couldn't abstract this into a seperate function because each
    case is different in subtle ways.
177 --i.e) they all have different functions, but also division and
    exponentiation are non-commutative, etc.
178 eval (MulE a b) = do
179     a' <- eval a
180     b' <- eval b
181     ensure [NumberT, ExprT] a' "multiplying"
182     ensure [NumberT, ExprT] b' "multiplying"
```

```
183     case (a', b') of
184         (Number a'', Number b'') -> return $ Number $ a'' * b''
185         (Number a'', Expr b'') -> Expr <$> liftCompute (simplify $
              Mul [Const a'', b''])
186         (Expr a'', Number b'') -> Expr <$> liftCompute (simplify $
              Mul [Const b'', a''])
187         (Expr a'', Expr b'') -> Expr <$> liftCompute (simplify $ Mul
              [a'', b''])
188 eval (DivE a b) = do
189     a' <- eval a
190     b' <- eval b
191     ensure [NumberT, ExprT] a' "dividing"
192     ensure [NumberT, ExprT] b' "dividing"
193     case (a', b') of
194         (Number a'', Number b'') -> return $ Number $ a'' / b''
195         (Number a'', Expr b'') -> Expr <$> liftCompute (simplify $
              Div (Const a'') b'')
196         (Expr a'', Number b'') -> Expr <$> liftCompute (simplify $
              Div a'' (Const b''))
197         (Expr a'', Expr b'') -> Expr <$> liftCompute (simplify $ Div
              a'' b'')
198 eval (PowE a b) = do
199     a' <- eval a
200     b' <- eval b
201     ensure [NumberT, ExprT] a' "exponentiating"
202     ensure [NumberT, ExprT] b' "exponentiating"
203     case (a', b') of
204         (Number a'', Number b'') -> return $ Number $ a'' ** b''
205         (Number a'', Expr b'') -> Expr <$> liftCompute (simplify $
              Pow (Const a'') b'')
206         (Expr a'', Number b'') -> Expr <$> liftCompute (simplify $
              Pow a'' (Const b''))
207         (Expr a'', Expr b'') -> Expr <$> liftCompute (simplify $ Pow
              a'' b'')
208 eval (SubE a b) = do
209     a' <- eval a
210     b' <- eval b
211     ensure [NumberT, ExprT] a' "subtracting"
212     ensure [NumberT, ExprT] b' "subtracting"
213     case (a', b') of
```

```
214         (Number a'', Number b'') -> return $ Number $ a'' - b''
215         (Number a'', Expr b'') -> Expr <$> liftCompute (simplify $
               Sum [Const a'', Mul [-1, b'']])
216         (Expr a'', Number b'') -> Expr <$> liftCompute (simplify $
               Sum [a'', Const (negate b'')])
217         (Expr a'', Expr b'') -> Expr <$> liftCompute (simplify $ Sum
               [a'', Mul [-1, b'']])
218 --All the trig and algebraic functions use the liftExpr we defined
       earlier.
219 eval (SinE a) = liftExpr a Sin sin "sine"
220 eval (CosE a) = liftExpr a Cos cos "cosine"
221 eval (TanE a) = liftExpr a Tan tan "tangent"
222 eval (ASinE a) = liftExpr a ASin asin "inverse sine"
223 eval (ACosE a) = liftExpr a ACos acos "inverse cosine"
224 eval (ATanE a) = liftExpr a ATan atan "inverse tangent"
225 eval (AbsE a) = liftExpr a Abs abs "absolute value"
226 eval (ExpE a) = liftExpr a Exp exp "exponential"
227 eval (LogE a) = liftExpr a Log log "logarithm"
228 --Square root is similar but we dont have a dedicated function so we
       translate
229 --it as a power of 0.5.
230 eval (SqrtE a) = do
231     a' <- eval a
232     ensure [NumberT, ExprT] a' "finding square root"
233     case a' of
234         Number a'' -> return $ Number $ sqrt a''
235         Expr a'' -> return $ Expr $ Pow a'' (Const 0.5)
236 --Make an equation by wrapping two expressions up as the lhs and rhs
237 --As usual with expressions, lift Numbers to Expressions
238 --and type check to make sure the arguments are either numbers or
       expressions.
239 eval (MakeEquation a b) = do
240     a' <- eval a
241     b' <- eval b
242     ensure [ExprT, NumberT] a' "forming equation"
243     ensure [ExprT, NumberT] b' "forming equation"
244     let a'' = case a' of
245             Number n -> Const n
246             Expr e -> e
247     let b'' = case b' of
```

```
248             Number n -> Const n
249             Expr e -> e
250     sa <- liftCompute $ simplify a''
251     sb <- liftCompute $ simplify b''
252     return $ Eqn $ Equation sa sb
253 --Make a point from two numbers.
254 eval (MakePoint a b) = do
255     a' <- eval a
256     b' <- eval b
257     ensure [NumberT] a' "making a point"
258     ensure [NumberT] b' "making a point"
259     let Number a'' = a'
260         Number b'' = b'
261     return $ Point a'' b''
262 --All the following are simple wrappers around constructing shapes
263 --checking for points and numbers appropriately
264 eval (LinePointGradient a b) = do
265     a' <- eval a
266     b' <- eval b
267     ensure [PointT] a' "making a line"
268     ensure [NumberT] b' "making a line"
269     let Point cx cy = a'
270         Number m = b'
271     return $ Shape $ linePointGradient (cx, cy) m
272 eval (LineInterceptGradient a b) = do
273     a' <- eval a
274     b' <- eval b
275     ensure [NumberT] a' "making a line"
276     ensure [NumberT] b' "making a line"
277     let Number a'' = a'
278         Number b'' = b'
279     return $ Shape $ lineInterceptGradient a'' b''
280 eval (LinePointPoint a b) = do
281     a' <- eval a
282     b' <- eval b
283     ensure [PointT] a' "making a line"
284     ensure [PointT] b' "making a line"
285     let Point ax ay = a'
286         Point bx by = b'
287     return $ Shape $ linePointPoint (ax, ay) (bx, by)
```

```
288 eval (CircleCenterRadius a b) = do
289     a' <- eval a
290     b' <- eval b
291     ensure [PointT] a' "making a circle"
292     ensure [NumberT] b' "making a circle"
293     let Point cx cy = a'
294         Number r = b'
295     return $ Shape $ circleCenterRadius (cx, cy) r
296 eval (CircleCenterPoint a b) = do
297     a' <- eval a
298     b' <- eval b
299     ensure [PointT] a' "making a circle"
300     ensure [PointT] b' "making a circle"
301     let Point cx cy = a'
302         Point rx ry = b'
303     return $ Shape $ circleCenterPoint (cx, cy) (rx, ry)
304 --Find the equation of a shape
305 eval (EquationOf a) = do
306     a' <- eval a
307     ensure [ShapeT] a' "finding the equation of a shape"
308     let Shape s = a'
309     --Map over all the constants and expressions ("terminals")
310     --And change the variables to the equivalent strings.
311     --keeping constants the same.
312     let revarExpr = emapt $ \case
313             Var X -> Var "x"
314             Var Y -> Var "y"
315             Const c -> Const (c :: Number)
316     let Equation lhs rhs = equationOf s
317     lhs' <- liftCompute $ simplify $ revarExpr lhs
318     rhs' <- liftCompute $ simplify $ revarExpr rhs
319     return $ Eqn $ Equation lhs' rhs'
320 -- Find the intersection of two shapes.
321 eval (IntersectionOf a b) = do
322     a' <- eval a
323     b' <- eval b
324     ensure [ShapeT] a' "finding the intersection of two shapes"
325     ensure [ShapeT] b' "finding the interseciton of two shapes"
326     let Shape sa = a'
327         Shape sb = b'
```

```
328     is <- liftCompute $ intersectionOf sa sb
329     return $ List $ map (uncurry Point) is
330 -- Substitute a variable in one expression for another expression
331 eval (Substitute v e a) = do
332     e' <- eval e
333     a' <- eval a
334     ensure [ExprT, NumberT] e' "substituting an expression"
335     ensure [ExprT, EqnT] a' "substituting an expression"
336     let dosub a'' = liftCompute $ case e' of
337             Number e'' -> substitute v (Const e'') a'' -- To
                substitute a number wrap it in constant first
338             Expr e'' -> substitute v e'' a''
339     case a' of
340         Eqn (Equation lhs rhs) -> Eqn <$> (Equation <$> dosub lhs
            <*> dosub rhs)
341         -- To substitute into an equation, substitute on both sides.
342         Expr a'' -> Expr <$> dosub a''
343 eval (Evaluate e) = do
344     e' <- eval e
345     ensure [ExprT] e' "evaluating an expression"
346     let Expr e'' = e'
347     res <- liftCompute $ evaluate e''
348     return $ Number res
349 eval (Solve e v) = do
350     e' <- eval e
351     ensure [EqnT] e' "solving an equation"
352     let Eqn e''@(Equation lhs rhs) = e'
353     let fv = case freeVariables lhs of -- If we dont provide a
            variable to solve for look through these:
354             [] -> case freeVariables rhs of
355                 [] -> return "x" -- If we have no variables, just
                    use x
356                 [(var, _)] -> return var -- If we have one variable,
                    great solve for that
357                 _ -> throwE SpecifyVariable -- If more than one,
                    throw an error
358             [(var, _)] -> return var -- Again, if we have only one
                use that
359             _ -> throwE SpecifyVariable -- Otherwise throw an error.
360     var <- maybe fv return v
```

```
361     vals <- liftCompute $ solve var e''
362     case vals of
363         [x] -> return $ Number x -- If we only have one solution,
                use that
364         _ -> return $ List $ map Number vals -- Otherwise wrap the
                list of solutions in a List
365 eval (SolveSimultaneous es) = do
366     es' <- mapM eval es
367     mapM_ (flip (ensure [EqnT]) "solving simultaneous equations") es'
368     -- mapM ensure over all the expressions to make sure they're all
           equations.
369     let getInner (Eqn e) = e
370     vals <- liftCompute $ solveSimultaneous $ map getInner es'
371     return $ List $ map (Number . snd) $ sortOn fst vals
372     -- Take the solutions, and sort them in alphabetical order,
373     -- then discard the variable names and return the solutions as a
           list.
374 eval (Integral e) = do
375     e' <- eval e
376     ensure [ExprT] e' "integrating"
377     let Expr e'' = e'
378     let fv = case freeVariables e'' of
379             ((var, _):_) -> var  -- If we have only a free variable,
                integrate with respect to that
380             [] -> "x" -- Otherwise integrate with respect to x
381     res <- liftCompute $ integrate fv e''
382     case res of
383         Nothing -> throwE CantIntegrate -- If integrate returns
                Nothing, throw an error.
384         Just res' -> return $ Expr res'
385 eval (DefiniteIntegral e a b) = do
386     e' <- eval e
387     a' <- eval a
388     b' <- eval b
389     ensure [ExprT] e' "computing a definite integral"
390     ensure [NumberT] a' "computing a definite integral"
391     ensure [NumberT] b' "computing a definite integral"
392     let Number a'' = a'
393         Number b'' = b'
394         Expr e'' = e'
```

```
395     let fv = case freeVariables e'' of -- The same variable finding
            procedure as for Integral
396             ((var, _):_) -> var
397             [] -> "x"
398     res <- liftCompute $ definiteIntegral fv e'' a'' b''
399     case res of
400         Nothing -> throwE CantIntegrate
401         Just res' -> return $ Number res'
402 eval (Derivative e s) = do
403     e' <- eval e
404     ensure [ExprT] e' "finding a derivative"
405     let Expr e'' = e'
406     fv <- case s of
407             Just v -> return v -- If we have are given a variable to
                differentiate by, good
408             Nothing -> case freeVariables e'' of
409                 [(var, _)] -> return var -- Otherwise if we have
                    only one use that
410                 [] -> return "x" -- If we have none, use x
411                 _ -> throwE SpecifyVariable -- And if we have more
                    than one throw an error.
412     res <- liftCompute $ differentiate fv e''
413     sres <- liftCompute $ simplify res
414     return $ Expr sres -- differentiate, simplify and return the
            result.
415 -- For all of the stats functions, use liftTable we defined earlier.
416 eval (Mean s e) = liftTable s e mean "mean"
417 eval (SampleMean s e) = liftTable s e sampleMean "sample mean"
418 eval (Variance s e) = liftTable s e variance "variance"
419 eval (SampleVariance s e) = liftTable s e sampleVariance "sample
        variance"
420 eval (StdDev s e) = liftTable s e standardDeviation "standard
        deviation"
421 eval (SampleStdDev s e) = liftTable s e sampleStandardDeviation
        "sample standard deviation"
422 eval (Median s e) = liftTable s e median "median"
423 eval (Mode s e) = liftTable s e mode "mode"
424 eval (ChiSquared a b e) = liftTable2 a b e chiSquared "chi squared
        value"
425 eval (PearsonCorrelation a b e) = liftTable2 a b e
```

```
    pearsonCorrelation "Pearson correlation"
eval (SpearmanCorrelation a b e) = liftTable2 a b e
    spearmanCorrelation "Spearman correlation"
eval (DotProduct a b) = do
    a' <- eval a
    b' <- eval b
    ensure [NumberT] a' "computing a dot product"
    ensure [NumberT] b' "computing a dot product"
    let Number a'' = a'
        Number b'' = b'
    return $ Number $ a'' `dotE` b''
eval (AngleBetween a b) = do
    a' <- eval a
    b' <- eval b
    ensure [NumberT] a' "computing an angle between vectors"
    ensure [NumberT] b' "computing an angle between vectors"
    let Number a'' = a'
        Number b'' = b'
    return $ Number $ a'' `angleE` b''
-- The matrix functions use liftMatrix as defined earlier
eval (Magnitude e) = liftMatrix e magnitudeE "magnitude"
eval (Determinant e) = liftMatrix e determinantE "determinant"
eval (InverseMatrix e) = liftMatrix e recip "inverse matrix"
eval (CharacteristicEquation e) = do
    e' <- eval e
    ensure [NumberT] e' "finding a characteristic equation"
    let Number e'' = e'
    n <- liftCompute e''
    case n of -- Since numbers can be both matrices and scalars
        ScalarV _ -> throwE OnlyForMatrices -- Throw an error if we
            have a scalar
        MatrixV m -> do -- Otherwise
            expr <- liftCompute $ characteristic m -- Compute the
                characteristic
            let revarExpr = emapt $ \case -- And remap all the
                terminals:
                    Var Lambda -> Var "x" -- Variables
                        (characteristic uses Lambda) go to string
                        variables
                    Const c -> Const (return $ ScalarV c) -- And the
```

```
                        constants go from Double to Number
459             expr' <- liftCompute $ simplify expr
460             return $ Expr $ revarExpr expr'
461 eval (Eigenvalues e) = do
462     e' <- eval e
463     ensure [NumberT] e' "finding eigenvalues"
464     let Number e'' = e'
465     vals <- liftCompute $ eigenvaluesE e''
466     -- Although we can only find the eigenvalues of a matrix,
467     -- eigenvaluesE will handle throwing the error for us if we get
        a scalar.
468     return $ List $ map (Number . return) vals
469 eval (Eigenvectors e) = do
470     e' <- eval e
471     ensure [NumberT] e' "finding eigenvectors"
472     let Number e'' = e'
473     vals <- liftCompute $ eigenvectorsE e''
474     -- This is the same as above with the eigenvalue errors.
475     return $ List $ map (Number . return) vals
476
477 -- To Assign a value to a variable
478 assign :: String -> Value -> Interpret ()
479 assign s v = do
480     Env vars <- lift get -- Get the current variables
481     lift $ put $ Env $ (s, v) : vars -- And put this new one on the
        front
482     -- Because of the way that reading variables is implemented
        (with lookup)
483     -- It will always get the first matching definition of a variable
484     -- so putting one on the front effectively "updates" the
        variable if it is already defined.
485
486 exec :: Statement -> Interpret (Maybe Value)
487 exec (Expression expr) = Just <$> eval expr
488 exec (Assign ss expr) = do
489     val <- eval expr
490     case ss of -- Look at the variables we are trying to unpack to
491         -- If there are none, we have made mistake.
492         [] -> error "Someone tried to unpack into no values."
493         -- If there's one, great put the whole value into that.
```

```
494         [s] -> assign s val
495         _ -> do -- If there's more than one:
496             -- First we have to make sure that its a list we're
                unpacking
497             ensure [ListT] val "trying to unpack a value"
498             let List vs = val
499             case length ss `compare` length vs of -- Then look at
                how many values there are
500                 GT -> throwE $ UnpackError "Too few values!" --
                    Throw an error if thats not right
501                 LT -> throwE $ UnpackError "Too many values!"
502                 EQ -> zipWithM_ assign ss vs -- And if it is, assign
                    each variable to each corresponding value.
503     return Nothing
504 exec (AssignArgs s as e) = do
505     mapM_ (\a -> assign a $ Expr (Var a)) as -- For a function, take
            every argument and assign it a fresh variable
506     val <- eval e
507     assign s val -- Then just assign the function body to the
            function name.
508     return Nothing
```

A.2.3 AME/PARSER/COMBINATORS.HS

```
1  {-# LANGUAGE LambdaCase #-}
2
3  -- A Library for Parser Combinators
4  -- This is type: Parser s a, and a set of functions that can
       generate primitive parsers
5  -- And combine them to form more complex parsers. The main interface
       is via do-notation
6  -- do
7  --     a <- someParser
8  --     b <- anotherParser
9  --     someOperation a b
10 -- This interface allows you to construct parsers that accept the
       results of previous parsers
11 -- and use them in other operations, combining in a way that
       respects any errors generated by the parsers.
12 -- A simple example would be:
13 -- test :: Parser Char Int
```

```
14 -- test = do
15 --     exact '('
16 --     a <- exact '0' <|> test
17 --     exact ')'
18 --     return a
19 -- Which matches any valid number of nested brackets around a zero.
       (i.e 0, (0), ((0)), (((0))) etc.)
20 module AME.Parser.Combinators where
21
22 import Control.Monad
23 import Control.Applicative
24
25 -- The possible values that a parser could expect:
26 --  A specific value, the end of the file, any character, or some
       described group.
27 data Vals s = S s | EOF | ANY | Describe String
28             deriving (Show, Eq)
29
30 -- All the errors that the parser could throw.
31 -- Expected a b: a was expected but b was observed
32 -- WrongVal s: s was observed but another value was expected.
33 -- UnknownError: something else.
34 -- UnderlyingError: an error in the lexer, not the parser.
35 data ParseError s = Expected (Vals s) (Vals s)
36                   | WrongVal s
37                   | UnknownError
38                   | UnderlyingError (ParseError Char)
39                   deriving (Show, Eq)
40
41 -- Parser s a: takes a list of s and produces a value of type a.
42 newtype Parser s a = Parser {
43     -- Parse the given input list, returning either the remaining
          input and a result, or an error.
44     parse :: [s] -> Either (ParseError s) ([s], a)
45 }
46
47 instance Functor (Parser s) where
48     --Parse a value, and apply a given function to the result, if
          the parse was successful.
49     fmap f p = Parser $ fmap (fmap f) .  parse p
```

```
50              --                              parse the original
51              --          if it is successful, apply f
52              -- wrap it in a new parser
53
54 -- A superclass required by Haskell's typesystem to implement Monad.
55 instance Applicative (Parser s) where
56     pure = return
57     f <*> a = do
58         f' <- f
59         a' <- a
60         return $ f' a'
61
62 -- The real powerhouse of parser combinators.
63 instance Monad (Parser s) where
64     -- return :: a -> Parser s a
65     -- Make a parser that returns a given value without consuming
       any input.
66     return = Parser . (return .) . flip (,)
67     --                              take the value and put it in a
       tuple
68     --                              take the input and put it in the
       other slot
69     --                Indicate the parser was successful
70     --       Wrap it in a new parser
71     -- (>>=) :: Parser s a -> (a -> Parser s b) -> Parser s b
72     -- The most powerful operation available for parsers:
73     -- a >>= f means, construct a parser which, given some input
74     -- parses the input using parser a, then, if it is successful,
75     -- takes the result and applies a function f, which generates
       another parser
76     -- which is the used to parse the remaining input. This is
       powerful because it is
77     -- the underlying mechanism for do-notation. E.g)
78     -- do
79     --    a <- p
80     --    f a
81     -- is equivalent to p >>= \a -> f a. Haskell's compiler
82     -- compiles do-notation by deconstructing into calls to this
       function.
83     a >>= f = Parser $ \s -> case parse a s of
```

```
84                Left err -> Left err
85                Right (s', a') -> parse (f a') s'
86
87 instance Alternative (Parser s) where
88     -- (<|>) :: Parser s a -> Parser s a -> Parser s a
89     -- Try using parser a, if it fails, use parser b instead.
90     -- Crucially, this bactracks - if a fails after partially
          parsing the input
91     -- b is started from the start of the text, NOT where a left off,
92     -- unlike in some other parser combinator libraries such as
          Parsec.
93     a <|> b = Parser $ \s -> case parse a s of
94         Left err -> parse b s
95         Right x -> Right x
96     -- empty :: Parser s a
97     -- A parser that always fails.
98     empty = throw UnknownError
99
100 -- Construct a parser which always fails with the given error.
101 throw :: ParseError s -> Parser s a
102 throw = Parser . const . Left
103
104 -- Try using a parser. If it doesn't work,
105 -- give the error to the provided function, and use the parser
106 -- that it provides as output, instead.
107 catch :: Parser s a -> (ParseError s -> Parser s a) -> Parser s a
108 catch p f = Parser $ \s -> case parse p s of
109             Left err -> parse (f err) s
110             Right x -> Right x
111
112 -- A parser that consumes one character from the input stream and
       returns it.
113 eat :: Parser s s
114 eat = Parser $ \case
115         [] -> Left $ Expected ANY EOF -- In the case of an end of
                 file, report that we expected something.
116         s:ss -> Right (ss, s)
117
118 -- A parser that expectes it to be the end of input.
119 done :: Parser s ()
```

```
120 done = Parser $ \case
121         [] -> Right ([], ()) -- If the input is empty, good
122         s:_ -> Left $ Expected EOF (S s) -- Otherwise we got too much
123
124 -- A parser that consumes one character and checks it matches the
    given predicate
125 -- If not, it fails.
126 match :: (s -> Bool) -> Parser s s
127 match f = do
128     c <- eat
129     if f c
130         then return c
131         else throw $ WrongVal c
132
133 -- A simple wrapper around match that matches exactly the input
    provided
134 exact :: Eq s => s -> Parser s ()
135 exact = void . match . (==)
136
137 -- This is useful because String ~ [Char] so in order to exactly
138 -- match a String, on a Parser Char (i.e the lexer) we must use this.
139 word :: Eq s => [s] -> Parser s ()
140 word = mapM_ exact
141
142 -- Take a parser, and if it fails with a WrongVal
143 -- (i.e it is a `match` parser that failed)
144 -- Then replace that with a provided, more descriptive
145 -- error message.
146 expecting :: Vals s -> Parser s a -> Parser s a
147 expecting v p = p `catch` \case
148         WrongVal c -> throw $ Expected v (S c)
149         err -> throw err
150
151 -- Try and parse something. If it fails, ignore it and return
    Nothing.
152 opt :: Parser s a -> Parser s (Maybe a)
153 opt p = (Just <$> p) `catch` const (return Nothing)
154
155 -- A convenience function for parsing lists of things seperated by
    another expression
```

```haskell
-- i.e lists of numbers seperated by commas.
sepBy :: Parser s a -> Parser s b -> Parser s [a]
sepBy a b = do
    first <- a
    rest <- many $ b >> a
    return $ first : rest
```

A.2.4 AME/PARSER/LEXER.HS

```haskell
module AME.Parser.Lexer (
    lex,
    Token(..),
    string,
    number,
    exactSt,
    exactSp,
    exactWords
) where

-- Use our parser combinator library to create a lexer.
-- A lexer takes the raw input string and turns it into a sequence of tokens.
-- This makes it easier to do things like ignoring whitespace, and differentiating
-- punctuation and identifiers

import Prelude hiding (lex)
import AME.Parser.Combinators
import Control.Applicative
import Control.Monad

-- A token can either be a string, punctuation or a number of a newlines
-- whitespace is not present because it is automatically filtered out.
data Token = String String
           | Special Char
           | Num Double
           | Newline
           deriving (Show, Eq)

```

```
29 isString :: Token -> Bool
30 isString (String _) = True
31 isString _ = False
32
33 isNum :: Token -> Bool
34 isNum (Num _) = True
35 isNum _ = False
36
37 -- Convenience function to match any string
38 -- Used for the main parser
39 string :: Parser Token String
40 string = (\(String s) -> s) <$> match isString
41
42 -- Match an exact given string
43 exactSt :: String -> Parser Token ()
44 exactSt = exact . String
45
46 -- Match an exact punctuation character
47 exactSp :: Char -> Parser Token ()
48 exactSp = exact . Special
49
50 -- Match exactly the given words. This is different from exactSt
51 -- because due to the lexer, there are no spaces in String's.
52 -- This allows more freedom in parsing more than one space between
     words.
53 exactWords :: String -> Parser Token ()
54 exactWords = mapM_ (exact . String) . words
55
56 number :: Parser Token Double
57 number = (\(Num n) -> n) <$> match isNum
58
59 -- The actual lexer parser functions:
60
61 -- Match one newline. All the lexer parsers have nice error messages.
62 parseNewline :: Parser Char Token
63 parseNewline = expecting (Describe "a newline") $ do
64     exact '\n'
65     return Newline
66
67 parseNumber :: Parser Char Token
```

```
68 parseNumber = expecting (Describe "a number") $ do
69     minus <- opt $ exact '-' -- Maybe a minus sign
70     lhs <- some digit -- Followed by some digits
71     rhs <- opt $ do
72         exact '.' -- and maybe a decimal point
73         some digit -- and some more digits.
74     let num = case rhs of
75             Just rhs' -> lhs ++ "." ++ rhs' -- Put all these bits
                together as one string
76             Nothing -> lhs
77     return $ Num $ case minus of
78         Nothing -> read num
79         Just () -> negate $ read num -- And read it into a float.
80     where digit = match (`elem` "0123456789")
81
82 parseSpecial :: Parser Char Token
83 parseSpecial = expecting (Describe "a punctuation character") $
       Special <$> match (`elem` specials)
84     where specials = "[]()-_+,:;^*/|={}"
85
86 parseString :: Parser Char Token
87 parseString = expecting (Describe "a string") $ do
88     first <- match (`elem` alpha) -- The first character of a string
          is only alphabetical
89                                   -- Not alphanumeric
90     rest <- many $ match (`elem` alphanumeric) -- The rest are
          alphanumeric
91     return $ String $ first : rest -- Combine the characters to make
          a string.
92     where alpha =
          "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
93           numeric = "0123456789"
94           alphanumeric = alpha ++ numeric
95
96 -- A token can be any one of the cases decribed above.
97 parseToken :: Parser Char Token
98 parseToken = expecting (Describe "a newline, number, punctuation
       character or string") $
99     parseNewline <|> parseNumber <|> parseSpecial <|> parseString
100
```

```
101 parseSpaces :: Parser Char ()
102 parseSpaces = void $ many $ exact ' '
103
104 -- The main lexer parser allows spaces basically anywhere in the
    input
105 -- parsing them and throwing them away when encountered.
106 lexer :: Parser Char [Token]
107 lexer = do
108     parseSpaces
109     toks <- many (parseSpaces >> parseToken)
110     parseSpaces
111     done
112     return toks
113
114 lex :: String -> Either (ParseError Char) [Token]
115 lex = fmap snd . parse lexer
```

A.2.5 AME/PARSER.HS

```
1 module AME.Parser (
2   parse
3 ) where
4
5 -- The main parser. This consists of many parsers each of which
    parses a small
6 -- amount of the grammar and then a couple of parsers which combine
    these all together.
7 -- There were several alternatives to writing a parser like this
    using parser combinators:
8 -- 1) I could have used a parser generator, and then just write a
    grammar specification which
9 --    could be used to generate a parser. Tools such as happy exist
    to do this for Haskell.
10 --    This would produce a much more efficient parser. However, it
    adds the additional step of
11 --    generating the parser on each build, as well as that it
    requires a lot of additional
12 --    code to generate the proper AST from the parse tree that it
    outputs.
13 -- 2) Use another parsing library such as attoparsec or Parsec. Both
    of these are alternative
```

```
14 --    parser combinator libraries, however they are both not ideal:
   attoparsec because of its
15 --    extremely fast performance, has no support for lexers, and
   limits me to using ByteString
16 --    instead of String, which brings other problems such as
   encoding and how to get it to interact
17 --    with the rest of the system. Parsec on the other hand, forces
   you to use its own lexer
18 --    which does not quite fit my use-case and does not use
   backtracking (<|>), which
19 --    would require a redesign of my grammar.
20 -- The entry point for the parser is parseStatement
21
22 import Prelude hiding (lex)
23 import Control.Applicative
24 import Control.Monad
25 import Data.Maybe
26 import Data.List
27 import AME.Parser.Lexer
28 import AME.Parser.Combinators hiding (parse)
29 import qualified AME.Parser.Combinators as C
30 import AME.Interpret.AST
31 import AME.Compute.Error
32 import AME.Compute.Matrix
33 import AME.Compute.Statistics
34 import Debug.Trace
35
36 -- Optionally allow the user to write "a" or "the"
37 -- This makes the language have much better flow as
38 -- the expressions are more like natural english.
39 determiner :: Parser Token ()
40 determiner = void $ opt $ exactSt "a" <|> exactSt "the"
41
42 -- A combinator that takes three parsers a, b and c
43 -- and will parse either a then b then c or c then b then a
44 -- this is useful for when you have a list of properties
45 -- but dont care about the order. I.e "with radius 4 and center (0,
   1)"
46 -- or "with center (0, 1) and radius 4".
47 -- It returns a tuple with the result of parsers a and c
```

```
48 eitherWay :: Parser s a -> Parser s b -> Parser s c -> Parser s (a,
       c)
49 eitherWay a b c = (do
50         a' <- a
51         b
52         c' <- c
53         return (a', c')) <|> (do
54         c' <- c
55         b
56         a' <- a
57         return (a', c'))
58
59 -- The actual parsers are below. This should be mostly
       self-explanatory,
60 -- at least, that is the goal of parser combinators.
61
62 -- Importantly, <$> is a synonym for fmap.
63 -- This mean that it is basically the same as $
64 -- So it just applies the given function to the result of the parser
65 -- i.e Variable <$> string, parses a string. This will give a String
66 -- so Variable <$> takes this string and applies Variable to it
67 -- giving an Expression.
68
69 -- In general, f <$> p can be rewritten as:
70 -- do
71 --    a <- p        -- parse p
72 --    return $ f a  -- apply f to the result, and return it
73
74 parseVariable :: Parser Token Expression
75 parseVariable = Variable <$> string
76
77 -- The reason for (return . ScalarV) here is that Number = Compute
       (MatrixExt Double)
78 -- ScalarV takes the Double and converts it to a MatrixExt Double
79 -- Then return takes this and puts it into Compute.
80 parseScalar :: Parser Token Number
81 parseScalar = (return . ScalarV) <$> number
82
83 parseRow :: Parser Token [Double]
84 parseRow = do
```

```
85      exactSp '['
86      vals <- number `sepBy` exactSp ','
87      exactSp ']'
88      return vals
89
90  parseVector :: Parser Token Number
91  parseVector = (return . MatrixV . vector) <$> parseRow
92
93  parseMatrix :: Parser Token Number
94  parseMatrix = parseVector <|> do
95      exactSp '['
96      rows <- parseRow `sepBy` exactSp ','
97      exactSp ']'
98      return $ return $ MatrixV $ matrix rows
99
100 parseNumber :: Parser Token Value
101 parseNumber = Number <$> (parseScalar <|> parseMatrix)
102
103 parseTable :: Parser Token Value
104 parseTable = do
105     exactWords "a table where"
106             -- Here we just flip the arguments of sepBy
107             -- so this is just the whole contents of the do-block,
108             -- seperated by (exactSp ';')
109     columns <- (`sepBy` exactSp ';') $ do
110         var <- string -- A variable name
111         exactSp '=' -- Then =
112         ds <- map (return . ScalarV) <$> parseRow -- Then a row of
                numbers
113         return (var, ds)
114     return $ Tbl $ uncurry Table $ unzip columns
115     -- The internal representation of Table is as two lists
116     -- one of variable names, and then the other of the respective
        lists of values.
117     -- but since columns is a list of tuples (variable name, values)
        we
118     -- apply unzip to give a tuple of two lists, as required by
        Table.
119     -- uncurry just makes Table accept a tuple instead of two
        seperate arguments
```

```
120     -- (unzip :: [(a, b)] -> ([a], [b]))
121     -- (uncurry :: (a -> b -> c) -> ((a, b) -> c))
122
123 parseValueExpr :: Parser Token Expression
124 parseValueExpr = Value <$> (parseNumber <|> parseTable)
125
126 parseCircle :: Parser Token Expression
127 parseCircle = do
128     determiner
129     exactWords "circle with"
130     -- In order two parse both the case where we provide
131     -- the center and radius, and the center and a point,
132     -- The second parser, which is either parseCirclRadius or
        parseCirclePoint
133     -- returns a function, cont, which we can apply the center to to
        get
134     -- out final AST.
135     (center, cont) <- eitherWay
136         (exactSt "center" >> parseExpr)
137         (exactSt "and")
138         (parseCircleRadius <|> parseCirclePoint)
139     return $ cont center
140     where parseCircleRadius = do
141             exactSt "radius" -- In the case of center and radius
142             radius <- parseExpr
143             return $ flip CircleCenterRadius radius -- The function
                is CircleCenterRadius
144           parseCirclePoint = do
145             determiner
146             exactSt "point" -- For center and point
147             point <- parseExpr
148             return $ flip CircleCenterPoint point -- The function is
                CircleCenterPoint
149
150 parseLine :: Parser Token Expression
151 parseLine = do
152     determiner
153     exactSt "line"
154     parseLinePointGradient <|> parseLineInterceptGradient <|>
        parseLinePointPoint
```

```
155     where parseLinePointGradient = do
156             (point, gradient) <- eitherWay
157                 (exactWords "passing through" >> parseExpr)
158                 (return ()) -- Nothing between these two
159                 (exactWords "with gradient" >> parseExpr)
160             return $ LinePointGradient point gradient
161           parseLineInterceptGradient = do
162             exactSt "with"
163             (intercept, gradient) <- eitherWay
164                 (exactSt "intercept" >> parseExpr)
165                 (exactSt "and")
166                 (exactSt "gradient" >> parseExpr)
167             return $ LineInterceptGradient intercept gradient
168           parseLinePointPoint = do
169             exactWords "passing through" <|> exactSt "connecting"
170             a <- parseExpr
171             exactSt "and"
172             b <- parseExpr
173             return $ LinePointPoint a b
174
175 parseEquationOf :: Parser Token Expression
176 parseEquationOf = do
177     opt $ exactSt "the"
178     -- We can't use determiner here because "a equation of" would
        not make sense but "the equation of" does.
179     exactWords "equation of"
180     shape <- parseExpr
181     return $ EquationOf shape
182
183 parseIntersectionOf :: Parser Token Expression
184 parseIntersectionOf = do
185     opt $ exactSt "the"
186     exactWords "intersection of"
187     a <- parseExpr
188     exactSt "and"
189     b <- parseExpr
190     return $ IntersectionOf a b
191
192 parsePoint :: Parser Token Expression
193 parsePoint = do
```

```
194     opt $ determiner >> exactSt "point" -- "the point", "a point",
            "point" or nothing
195     exactSp '('
196     a <- parseExpr
197     exactSp ','
198     b <- parseExpr
199     exactSp ')'
200     return $ MakePoint a b
201
202 -- Parse a simple numeric function of one argument.
203 -- Given a function and a name, this parses name(expression)
204 -- and passes the expression value to the function.
205 -- i.e) parseListExpr SinE "sin" would parse "sin(1)" and pass 1 to
        SinE
206 parseLiftExpr :: (Expression -> Expression) -> String -> Parser
        Token Expression
207 parseLiftExpr f n = do
208     exactSt n
209     exactSp '('
210     res <- f <$> parseExpr
211     exactSp ')'
212     return res
213
214 parseAbs :: Parser Token Expression
215 parseAbs = do
216     exactSp '|'
217     res <- AbsE <$> parseExpr
218     exactSp '|'
219     return res
220
221 -- We take the whole list of possiblities, and apply parseLiftExpr
        to each
222 -- and then foldl1 (<|>) combines these into one parser that can
        parser any of the list.
223 -- So this will parse "sin(x)", "cos(x)", .... "sqrt(x)"
224 parseExprFunc :: Parser Token Expression
225 parseExprFunc = foldl1 (<|>) $ map (uncurry parseLiftExpr)
226                     [(SinE, "sin"),
227                      (CosE, "cos"),
228                      (TanE, "tan"),
```

```
229                       (ExpE, "exp"),
230                       (LogE, "log"),
231                       (ASinE, "asin"),
232                       (ACosE, "acos"),
233                       (ATanE, "atan"),
234                       (SqrtE, "sqrt")]
235
236 -- Simply parse a bracketed expression.
237 parseBracket :: Parser Token Expression
238 parseBracket = do
239     exactSp '('
240     e <- parseExpr
241     exactSp ')'
242     return e
243
244 parseSubstitute :: Parser Token Expression
245 parseSubstitute = do
246     (var, e) <- (do                    -- Either parse
247         exactSt "substitute"           -- "substitute"
248         var <- string                  -- a variable name
249         exactSt "for" <|> exactSp '='  -- "for" or "="
250         e <- parseExpr                 -- an expression
251         return (var, e)) <|> (do       -- Or parse
252         exactSt "set"                  -- "set"
253         var <- string                  -- a variable name
254         exactSp '='                    -- "="
255         e <- parseExpr                 -- an expression
256         return (var, e))
257         -- This allows the parser to parse "substitute x for 2 in y"
258         -- or "substitute x = 2 in y" or "set x = 2 in y" but not
               "set x for 2 in y"
259     exactSt "in"
260     body <- parseExpr
261     return $ Substitute var e body
262
263 parseEvaluate :: Parser Token Expression
264 parseEvaluate = do
265     exactSt "evaluate" <|> exactWords "the value of"
266     e <- parseExpr
267     return $ Evaluate e
```

```
268
269 parseSolve :: Parser Token Expression
270 parseSolve = do
271     exactSt "solve" <|> exactWords "the solution of"
272                     <|> exactWords "the solutions of"
273                     <|> exactWords "the root of"
274                     <|> exactWords "the roots of"
275     e <- parseExpr
276     var <- opt $ do    -- Optionally provide the variable to solve
        for.
277         exactSt "for"
278         string
279     return $ Solve e var
280
281 parseSolveSimultaneous :: Parser Token Expression
282 parseSolveSimultaneous = do
283     exactSt "solve" <|> exactWords "the solution of"
284                     <|> exactWords "the solutions of"
285                     <|> exactWords "the root of"
286                     <|> exactWords "the roots of"
287     exactSp '{'
288     es <- parseExpr `sepBy` (exact Newline <|> exactSp ';')
289     exactSp '}'
290     return $ SolveSimultaneous es
291
292 -- e.g "the variable x", "a variable x", "variable x"
293 parseMakeVariable :: Parser Token Expression
294 parseMakeVariable = do
295     determiner
296     exactSt "variable"
297     v <- string
298     return $ MakeVariable v
299
300 parseIntegral :: Parser Token Expression
301 parseIntegral = do
302     exactSt "integrate" <|> exactWords "the integral of"
303                         <|> exactWords "the indefinite integral of"
304     e <- parseExpr
305     return $ Integral e
306
```

```
parseDefiniteIntegral :: Parser Token Expression
parseDefiniteIntegral = do
    exactSt "integrate" <|> exactWords "the integral of"
                        <|> exactWords "the definite integral of"
    e <- parseExpr
    (a, b) <- (do               -- Either parse:
        exactSt "between"       -- "between"
        a <- parseExpr          -- a
        exactSt "and"           -- "and"
        b <- parseExpr          -- b
        return (a, b)) <|> (do -- Or parse:
        exactSt "from"          -- "from"
        a <- parseExpr          -- a
        exactSt "to"            -- "to"
        b <- parseExpr          -- b
        return (a, b))
    return $ DefiniteIntegral e a b

parseDerivative :: Parser Token Expression
parseDerivative = do
    exactSt "differentiate" <|> exactWords "the derivative of"
    e <- parseExpr
    var <- opt $ do
        exactWords "with respect to"
        string
    return $ Derivative e var

-- This is a very general parser. Many of the statistics commands
   take the form of
-- "the _ of _ in _" for one form of the command (i.e "the standard
   deviation of x in table1")
-- or "the sample _ of _ in _" for the sample form of the command
   (i.e "the sample variance of y in table2")
-- This takes two functions, one to apply for the regular version
   and one for the sample version,
-- as well as the command name, and produces a parser for it.
parseStatSampleFunc :: (String -> Expression -> Expression) ->
    (String -> Expression -> Expression) -> String -> Parser Token
   Expression
parseStatSampleFunc m sm n = do
```

```
341     opt $ exactSt "the"
342     sample <- isJust <$> opt (exactSt "sample")
343     -- sample is True if "sample" is present and False otherwise
344     exactWords n -- parse the command name
345     exactSt "of"
346     var <- string
347     exactSt "in"
348     tbl <- parseExpr
349     if sample
350         then return $ sm var tbl -- use the alternative function if
                sample is True
351         else return $ m var tbl
352
353 -- This is similar, except its for commands which do not have a
    "sample" variant, like median and mode.
354 parseStatFunc :: (String -> Expression -> Expression) -> String ->
    Parser Token Expression
355 parseStatFunc m n = do
356     opt $ exactSt "the"
357     exactWords n
358     exactSt "of"
359     var <- string
360     exactSt "in"
361     tbl <- parseExpr
362     return $ m var tbl
363
364 parseStatFuncs :: Parser Token Expression
365 parseStatFuncs = parseStatSampleFunc Mean SampleMean "mean" <|>
366                  parseStatFunc Mode "mode" <|>
367                  parseStatFunc Median "median" <|>
368                  parseStatSampleFunc Variance SampleVariance
                        "variance" <|>
369                  parseStatSampleFunc StdDev SampleStdDev "standard
                        deviation"
370
371 -- This is also similar to parseStatFunc, except that its for
372 -- commands of the form "the _ of _ and _ in _" i.e stats functions
    of two variables
373 -- e.g "the spearman correlation of x and y in table1"
374 parseStatDiFunc :: (String -> String -> Expression -> Expression) ->
```

```
    String -> Parser Token Expression
parseStatDiFunc f n = do
    opt $ exactSt "the"
    exactWords n
    exactSt "of"
    a <- string
    exactSt "and"
    b <- string
    exactSt "in"
    tbl <- parseExpr
    return $ f a b tbl

parseStatFuncs2 :: Parser Token Expression
parseStatFuncs2 = parseStatDiFunc ChiSquared "chi squared value" <|>
                  parseStatDiFunc PearsonCorrelation "pearson
                      correlation" <|>
                  parseStatDiFunc SpearmanCorrelation "spearman
                      correlation"

-- This parses the operations on matrices of the form:
-- "the _ of _", (e.g "the determinant of m")
-- by taking a function to wrap the value in, and a command name
parseMatFunc :: (Expression -> Expression) -> String -> Parser Token
    Expression
parseMatFunc f n = do
    opt $ exactSt "the"
    exactWords n
    exactSt "of"
    mat <- parseExpr
    return $ f mat

parseMatFuncs :: Parser Token Expression
parseMatFuncs = parseMatFunc Magnitude "magnitude" <|>
                parseMatFunc Determinant "determinant" <|>
                parseMatFunc InverseMatrix "inverse" <|>
                parseMatFunc Eigenvalues "eigenvalues" <|>
                parseMatFunc Eigenvectors "eigenvectors" <|>
                parseMatFunc CharacteristicEquation "characteristic
                    equation"

```

```
410 -- This is the matrix equivalent of parseStatDiFunc. It parses
    commands of the form:
411 -- "the _ of _ and _", (e.g "the dot product of a and b")
412 parseMatFunc2 :: (Expression -> Expression -> Expression) -> String
    -> Parser Token Expression
413 parseMatFunc2 f n = do
414     opt $ exactSt "the"
415     exactWords n
416     a <- parseExpr
417     exactSt "and"
418     b <- parseExpr
419     return $ f a b
420
421 parseMatFuncs2 :: Parser Token Expression
422 parseMatFuncs2 = parseMatFunc2 DotProduct "dot product of" <|>
423                  parseMatFunc2 AngleBetween "angle between"
424
425 -- This parses all of the "terminal" expressions.
426 -- This means all of the expressions without any kind of binary
    operations
427 -- i.e "the determinant of m" is a terminal expression, "1 + 2" is
    not.
428 parseExprNoOp :: Parser Token Expression
429 parseExprNoOp = parseBracket <|>
430                 parseValueExpr <|>
431                 parseEquationOf <|>
432                 parseIntersectionOf <|>
433                 parseCircle <|>
434                 parseLine <|>
435                 parsePoint <|>
436                 parseAbs <|>
437                 parseExprFunc <|>
438                 parseSubstitute <|>
439                 parseEvaluate <|>
440                 parseSolve <|>
441                 parseSolveSimultaneous <|>
442                 parseMakeVariable <|>
443                 parseDefiniteIntegral <|>
444                 parseIntegral <|>
445                 parseDerivative <|>
```

```
446                 parseStatFuncs <|>
447                 parseStatFuncs2 <|>
448                 parseMatFuncs <|>
449                 parseMatFuncs2 <|>
450                 parseVariable
451
452 -- This takes a function and a character,
453 -- and when it encounters that character, it returns both the
       function and the character.
454 -- It is used to parse the operations that can occur between
       expressions.
455 parseBinop :: (Expression -> Expression -> Expression) -> Char ->
       Parser Token (Expression -> Expression -> Expression, Char)
456 parseBinop f s = exactSp s >> return (f, s)
457
458 parseBinops :: Parser Token (Expression -> Expression -> Expression,
       Char)
459 parseBinops = parseBinop SumE '+' <|>
460               parseBinop MulE '*' <|>
461               parseBinop DivE '/' <|>
462               parseBinop PowE '^' <|>
463               parseBinop SubE '-' <|>
464               parseBinop MakeEquation '='
465
466 -- This is the main expression parser. It contains a simple
       algorithm for determining operator precedence.
467 -- I could have used an operator precedence parser, but I didnt know
       at the start which operators i was going to have
468 -- so i used this, which is very easily extendable. I could also
       have used
469 -- Dijkstra's shunting yard algorithm, but this is not very easy to
       implement on a functional language
470 -- as it needs state (a stack).
471 parseExpr :: Parser Token Expression
472 parseExpr = do
473     first <- parseExprNoOp -- parse the first terminal
474     rest <- many $ do      -- followed by as many as you want of:
475         b <- parseBinops   -- an operation
476         e <- parseExprNoOp -- and another terminal.
477         return (b, e)
```

```
478     -- First we take all the terminals and operations and put them
            into one list of
479     -- Either Operation Expression so all the operations are wrapped
            in Left
480     -- and all the terminals in Right.
481     let ls = Right first : concatMap (\(a, b) -> [Left a, Right b])
            rest
482          --   The first is an expression
483          --                Concatenate the lists of
484          --                                        Left for each
                operator and right for each expression
485     -- SplitOn is a helper function which takes a list [a] and a
            predicate
486     -- and splits it into ([a], a, [a]). The first list corresponds
            to the part
487     -- of the list _before_ the first element for which the
            predicate is FALSE.
488     -- The second element of the tuple is the first element of the
            list for which the predicate is FALSE.
489     -- The final list is the rest of the list after that element.
490     -- I.e) splitOn odd [1, 2, 3, 4, 5, 6] would be ([1], 2, [3, 4,
            5, 6])
491     --  or  splitOn (== 3) [3, 3, 4, 3, 3] would be ([3, 3], 4, [3,
            3])
492     let splitOn f ls = (prefix, head suffix, tail suffix)
493           where (prefix, suffix) = span f ls
494     -- Our predicate for splitOn looks for the first Left value in
            our list where
495     -- the operator character is the one we are looking for.
496     let isCorrect op (Left (_, o)) = o == op
497         isCorrect _ _ = False
498     -- To remove one instance of a given operator from the list,
499     -- We first splitOn the negative of our predicate, this will look
500     -- for the first place where we have our operator. Then, the
            element
501     -- just before the operator must be our lhs, and just after the
            operator
502     -- must be the rhs. Then we take the lhs and rhs and combine
            them into one new
503     -- expression according to the operator function. Then, we
```

```
        remove the lhs
504     -- rhs and operation from the list and replace them with the new
        expression.
505     let removeOne (op, f) ls = init prefix ++ [Right (f rhs lhs)] ++
        tail suffix
506                              -- remove lhs
507                              --                replace operation with
                                    new expression
508                              --
                                    remove rhs
509             where (prefix, _, suffix) = splitOn (not . isCorrect op)
                    ls
510                   Right lhs = last prefix -- The lhs is the last
                        element before the operation
511                   Right rhs = head suffix -- The rhs is the first
                        element after the operation
512     -- To remove all of the occurrences of a given operator, we
        simply iterate
513     -- removeOne until there are no more occurences of the operator
        left.
514     let removeAll (op, f) ls = head $ dropWhile (any (isCorrect op))
        vals
515                              --
                                    the iterations of removeOne
516                              --        drop them while the operator
                                    is still present
517                              -- the first value of the remaining
                                    iterations
518             where vals = iterate (removeOne (op, f)) ls
519     -- The core of the algorithm is then very simple:
520     -- We remove all occurences of all operators. The key to the
        algorithm
521     -- is doing this in order from highest to lower precedence.
522     -- The problem with this algorithm is that it assumes all
        operations are right-associative,
523     -- which is not the case for some operators, e.g exponentiation,
        but it is good enough for now.
524     let precs = [('^', PowE),
525                  ('/', DivE),
526                  ('*', MulE),
```

```
527                   ('+', SumE),
528                   ('-', SubE),
529                   ('=', MakeEquation)]
530     let Right final = head $ foldr removeAll (reverse ls) (reverse
            precs)
531                             -- Fold the list over removeAll with the
                                   all the operators
532                             -- because we are using foldr and not
                                   foldl, we reverse the lists.
533                       -- At the end there should be only one element
                             left so get that
534     -- And since it is an expression it should be wrapped in Right,
            so unwrap it.
535     return final
536
537 parseExpression :: Parser Token Statement
538 parseExpression = Expression <$> parseExpr
539
540 -- Parse the variable names for an assign statement
541 -- this is either simple variable or a list of variables in brackets,
542 -- i.e "a" or "(a, b, c)"
543 parseNames :: Parser Token [String]
544 parseNames = parseSingle <|> parseMultiple
545     where parseSingle = return <$> string
546           parseMultiple = do
547                 exactSp '('
548                 names <- string `sepBy` exactSp ','
549                 exactSp ')'
550                 return names
551
552 -- Parse the signature of a function in an AssignArgs statement:
553 -- e.g "sin(x)" or "atan2(x, y)"
554 parseFuncNames :: Parser Token (String, [String])
555 parseFuncNames = do
556     n <- string
557     exactSp '('
558     as <- string `sepBy` exactSp ','
559     exactSp ')'
560     return (n, as) -- Return the function name, and the arguments
561
```

```
562 parseAssign :: Parser Token Statement
563 parseAssign = do
564     opt $ exactSt "let"
565     names <- parseNames
566     exactSp '='
567     e <- parseExpr
568     return $ Assign names e
569
570 parseFuncAssign :: Parser Token Statement
571 parseFuncAssign = do
572     opt $ exactSt "let"
573     (n, as) <- parseFuncNames
574     exactSp '='
575     e <- parseExpr
576     return $ AssignArgs n as e
577
578 parseStatement :: Parser Token Statement
579 parseStatement = do
580     st <- parseAssign <|> parseFuncAssign <|> parseExpression
581     done -- And then we expect the input to be empty, i.e each input
        to parse
582          -- must contain precisely one statement.
583     return st
584
585 -- A simple wrapper around parseStatement and lex which takes a
    string and parses it fully
586 parse :: String -> Either (ParseError Token) Statement
587 parse s = case lex s of
588     Left err -> Left $ UnderlyingError err
589     Right toks -> snd <$> C.parse parseStatement toks
```

A.3 SERVER COMPONENT

A.3.1 MAIN.HS

```
1 {-# LANGUAGE FlexibleInstances, OverlappingInstances, LambdaCase #-}
2
3 module Main where
4
5 -- This is the main server side interface to AME.
6 -- It is a simple ZMQ based socket server that takes an input string
```

```
7  -- and returns a result in JSON.
8
9  import Data.IORef
10 import Control.Monad
11 import Data.List
12 import AME.Compute.Geometry
13 import AME.Compute.Expr
14 import AME.Compute.Error
15 import AME.Compute.Matrix
16 import AME.Compute.Simplify
17 import AME.Compute.Statistics
18 import AME.Interpret
19 import AME.Interpret.AST
20 import AME.Parser
21 import AME.Parser.Combinators (ParseError(..), Vals(..))
22 import AME.Parser.Lexer (Token)
23 import System.ZMQ4.Monadic
24 import System.Environment
25 import Data.ByteString.Char8 (pack, unpack)
26
27 -- a "safe" number.. this is what we want to end up with
28 -- i.e no Compute means no possible errors.
29 type SNumber = MatrixExt Double
30
31 roundSNum :: SNumber -> SNumber
32 roundSNum (ScalarV s) = ScalarV $ roundToDp 5 s
33 roundSNum (MatrixV m) = MatrixV $ matrix $ map (map $ roundToDp 5) $
       values m
34
35 roundToDp :: Int -> Double -> Double
36 roundToDp n c = fromIntegral (truncate $ c * (10^n)) / 10^n
37
38 -- To show these we output a string of Latex.
39 -- This is then displayed by the client side nicely using MathJax
40 instance Show (MatrixExt Double) where
41     -- Matrices in latex are represented like so:
42     -- [[1, 2, 3], [4, 5, 6]] would be
43     -- \begin{bmatrix}
44     --  1 & 2 & 3 \\
45     --  4 & 5 & 6
```

```haskell
    -- \end{bmatrix}
    show (MatrixV m) = "\\\\begin{bmatrix}" ++ intercalate
        "\\\\\\\\" (map (intercalate "&" . map show) $ values m) ++
        "\\\\end{bmatrix}"
                        -- initial prefix     join the rows with
                           "\\"  take each column and join it with
                           "&"s           final suffix
    show (ScalarV s) = show s
                       -- Scalars are just numbers

-- A "safe" value type.. i.e the same but nothing wrapped in Compute.
data SVal = SNumber SNumber
          | SShape (Shape SNumber)
          | SExpr (Expr String SNumber)
          | SList [SVal]
          | SEqn (Equation String SNumber)
          | STbl (Table String SNumber)
          | SPoint SNumber SNumber
          deriving (Show, Eq)

-- Combine all the errors into one big error type.
data CombinedError = InterpretError InterpretError
                   | ParseError (ParseError Token)
                   deriving (Show, Eq)

-- Take the Computes from the inside of Value and
-- move it to the outside, leaving the rest of the Value
-- as an SVal.
-- This is easy because Compute is a Monad
toSVal :: Value -> Compute SVal
toSVal (Number n) = (SNumber . roundSNum) <$> n -- Evaluate n, get
    the inside and apply SNumber
toSVal (Shape c) = SShape <$> case c of
    Circle (x, y) r -> do
        x' <- roundSNum <$> x -- Evaluate all the parameters of the
            circle
        y' <- roundSNum <$> y
        r' <- roundSNum <$> r
        return $ Circle (x', y') r' -- And wrap it back up in a
            Circle
```

```
79      Line (x, y) m -> do
80          x' <- roundSNum <$> x
81          y' <- roundSNum <$> y
82          m' <- roundSNum <$> m
83          return $ Line (x', y') m' -- Do the same for Line
84 toSVal (Expr e) = SExpr <$> toSExpr e -- Expressions use toSExpr as
       defined later
85 toSVal (Eqn (Equation lhs rhs)) = do
86      lhs' <- toSExpr lhs
87      rhs' <- toSExpr rhs -- Equations are the same but operator on
           both sides
88      return $ SEqn (Equation lhs' rhs')
89 toSVal (List vs) = SList <$> mapM toSVal vs -- And lists map toSVal
       over all their values first.
90 toSVal (Tbl (Table vars vals)) = do
91      vals' <- mapM sequence vals -- Evaluate all the values in the
           table.
92                                  -- This works as sequence (sequence
                                       :: Monad m => [m a] -> m [a])
93                                  -- takes a row and turns it into a
                                       normal array wrapped in Compute
94                                  -- And then mapM applies this to
                                       every row and combines the
                                       results into
95                                  -- one big Compute.
96      return $ STbl (Table vars vals')
97 toSVal (Point a b) = do
98      a' <- a
99      b' <- b
100     return $ SPoint a' b'
101
102 -- This just takes all the Numbers in an Expr and turns them into
       SNumbers
103 toSExpr :: Expr String Number -> Compute (Expr String SNumber)
104 toSExpr (Const c) = (Const . roundSNum) <$> c -- Evaluate c, then
       wrap it back in Const
105 toSExpr (Var v) = return (Var v) -- Variables stay the same
106 toSExpr (Mul xs) = Mul <$> mapM toSExpr xs -- Mul and Sum evaluate
       all of their arguments with mapM
107 toSExpr (Sum xs) = Sum <$> mapM toSExpr xs
```

```
toSExpr (Div a b) = liftM2 Div (toSExpr a) (toSExpr b) -- liftM2 is like <$> but for functions of two arguments
toSExpr (Pow a b) = liftM2 Pow (toSExpr a) (toSExpr b)
toSExpr (Sin a) = Sin <$> toSExpr a -- All the rest just recursively apply toSExpr to their contents in order
toSExpr (Cos a) = Cos <$> toSExpr a -- to propagate it down the tree.
toSExpr (Tan a) = Tan <$> toSExpr a
toSExpr (ASin a) = ASin <$> toSExpr a
toSExpr (ACos a) = ACos <$> toSExpr a
toSExpr (ATan a) = ATan <$> toSExpr a
toSExpr (Exp a) = Exp <$> toSExpr a
toSExpr (Log a) = Log <$> toSExpr a
toSExpr (Abs a) = Abs <$> toSExpr a

showTy :: Type -> String
showTy NumberT = "number"
showTy ShapeT = "shape"
showTy ExprT = "expression"
showTy EqnT = "equation"
showTy ListT = "list"
showTy TblT = "data table"
showTy PointT = "point"

-- Describe any of the possible errors in english to display to the user.
describeError :: CombinedError -> String
describeError (InterpretError e) = case e of
    TypeError tys ty ex -> "Type Error: Expected " ++ intercalate " or " (map showTy tys) ++ " but got " ++ showTy ty ++ " while " ++ ex
                                                    -- For each possible type, show it and seperate them by " or "
    NumericalError t -> describeNumericalError t
                        -- Numerical errors handled seperately below.
    UnpackError e -> "Error while unpacking: " ++ e
    NoVariable v -> "The variable " ++ v ++ " does not exist."
    SpecifyVariable -> "Please specify which variable the operation
```

```
        is with respect to."
139     CantIntegrate -> "This function cannot be integrated."
140     OnlyForMatrices -> "Can't find a characteristic equation of a
            scalar."
141 describeError (ParseError e) = case e of
142     Expected a b -> "Expected " ++ showVals a ++ " got " ++ showVals
            b
143     WrongVal a -> "Got the wrong value: " ++ show a
144     UnknownError -> "Unknown parser error!"
145     UnderlyingError e -> case e of
146     -- Handle UnderlyingErrors exactly the same because theyre just
            ParseErrors that occured in the lexer.
147         Expected a b -> "Expected " ++ showVals a ++ " got " ++
                showVals b
148         WrongVal a -> "Got the wrong value: " ++ show a
149         UnknownError -> "Unknown parser error!"
150
151 showVals :: Show s => Vals s -> String
152 showVals (S s) = "\"" ++ show s ++ "\""
153 showVals (Describe s) = s
154 showVals EOF = "end-of-file"
155 showVals ANY = "any"
156
157 describeNumericalError :: NumericalError -> String
158 describeNumericalError DivisionByZero = "Tried to divide by zero."
159 describeNumericalError UnexpectedFreeVariable = "The expression
        still has free variables."
160 describeNumericalError (MatrixHasWrongDimensions a b c d) = "The
        matrix has wrong dimensions: expected " ++ show (a, b) ++ " got "
        ++ show (c, d)
161 describeNumericalError (MatrixShouldBeSquare a b) = "The matrix
        should be square, but instead has dimensions " ++ show (a, b)
162 describeNumericalError (MatrixOperationNotSupported reason) = "The
        operation is not supported: " ++ reason
163 describeNumericalError MatrixNotInvertible = "The matrix is not
        invertible!"
164 describeNumericalError IntersectsEverywhere = "The two shapes
        intersect everywhere!"
165 describeNumericalError ShouldBeLinear = "Cannot solve non-linear
        simultaneous equations."
```

```
166 describeNumericalError (WrongNumberOfEquations a b) = "Wrong number
    of equations: expected " ++ show a ++ " got " ++ show b
167 describeNumericalError NoPartialIntegrals = "Cannot find partial
    integrals!"
168
169 -- To show a given SVal, we need to produce a string of Latex for
    the front-end to render.
170 showValue :: SVal -> String
171 showValue (SNumber s) = show s -- For numbers we use the Show
    SNumber instance defined above
172 showValue (SShape s) = case s of
173     -- Circles are given in the form of center and radius
174     Circle (x, y) r -> "\\\\text{the circle with center }" ++ show
        (x, y) ++ "\\\\text{ and radius }" ++ show r
175     -- And lines are given as a point and a gradient
176     Line (x, y) m -> "\\\\text{the line passing through }" ++ show
        (x, y) ++ "\\\\text{ with gradient }" ++ show m
177     -- The LaTex "\text{...}" includes some text in the output
        verbatim instead of treating the words as variables.
178 showValue (SExpr expr) = showSExpr expr
179 -- Expressions handled seperately below
180 showValue (SEqn (Equation lhs rhs)) = showSExpr lhs ++ " = " ++
    showSExpr rhs
181 showValue (SPoint a b) = show (a, b)
182
183 showSExpr :: Expr String SNumber -> String
184                     -- Wrap in "{}"s to avoid precedence issues.
185                     -- Map showSExpr over all of the constituents
186                     -- And then seperate them with either "+" or "*"
                           for Sum or Mul
187 showSExpr (Sum xs) = "{" ++ intercalate "+" (map showSExpr xs) ++ "}"
188 showSExpr (Mul xs) = "{" ++ intercalate "\\\\cdot " (map showSExpr
    xs) ++ "}"
189 showSExpr (Pow a b) = "{" ++ showSExpr a ++ "}^{" ++ showSExpr b ++
    "}"
190                     -- Latex for a/b is "\frac{a}{b}"
191 showSExpr (Div a b) = "\\\\frac{" ++ showSExpr a ++ "}{" ++
    showSExpr b ++ "}"
192 -- Variables are just a string
193 showSExpr (Var v) = v
```

```
194 -- Constants use the Show SNumber definition provided earlier.
195 showSExpr (Const c) = show c
196 -- \sin, \cos etc work as expected. \left( is a starting ( and
    \right) an end bracket.
197 showSExpr (Sin x) = "\\\\sin{\\\\left(" ++ showSExpr x ++
    "\\\\right)}"
198 showSExpr (Cos x) = "\\\\cos{\\\\left(" ++ showSExpr x ++
    "\\\\right)}"
199 showSExpr (Tan x) = "\\\\tan{\\\\left(" ++ showSExpr x ++
    "\\\\right)}"
200 showSExpr (ASin x) = "\\\\asin{\\\\left(" ++ showSExpr x ++
    "\\\\right)}"
201 showSExpr (ACos x) = "\\\\acos{\\\\left(" ++ showSExpr x ++
    "\\\\right)}"
202 showSExpr (ATan x) = "\\\\atan{\\\\left(" ++ showSExpr x ++
    "\\\\right)}"
203 showSExpr (Exp x) = "\\\\exp{\\\\left(" ++ showSExpr x ++
    "\\\\right)}"
204 showSExpr (Log x) = "\\\\log{\\\\left(" ++ showSExpr x ++
    "\\\\right)}"
205 showSExpr (Abs x) = "{\\\\left|" ++ showSExpr x ++ "\\\\right|}"
206
207 -- Take an Interpret value as well as the current variable state and
    lift out all the errors and state changes.
208 liftInterpret :: Env -> Interpret a -> Either CombinedError (a, Env)
209 liftInterpret e a = case runInterpret e a of -- Run the interpret.
210         (Right val, e') -> Right (val, e') -- If it is succesfull,
              return it
211         (Left err, _) -> Left (InterpretError err) -- Otherwise wrap
              the given error in CombinedError.
212
213 -- Similarly, Lift a Parse value by simply wrapping the errors in
    CombinedError
214 liftParse :: Either (ParseError Token) a -> Either CombinedError a
215 liftParse a = case a of
216     Left err -> Left (ParseError err)
217     Right val -> Right val
218
219 -- In order to show valid string in the JSON output
220 -- We need to escape the quote and escape characters
```

```
221 escape :: String -> String
222 escape = concatMap $ \case
223     '"' -> "\\\""
224     x -> return x
225
226 -- One evaluation step. Takes a reference to the current variable
    state
227 -- as well as an input string, and returns a string of JSON to be
    sent to the client.
228 replStep :: IORef Env -> String -> ZMQ z String
229 replStep ref line = do
230     -- Read the value of the current state
231     env <- liftIO $ readIORef ref
232     let res = do
233             -- Try parse the input
234             ast <- liftParse $ parse line
235             -- Try executing the parsed expression, maybe returning
                an unsafe Value.
236             (unsafeVal, e) <- liftInterpret env $ exec ast
237             val <- case unsafeVal of
238                 -- If a value is returned, convert it to a safe SVal.
239                 Just unsafeVal' -> (Just . fst) <$> liftInterpret
                    env (liftCompute $ toSVal unsafeVal')
240                 -- Otherwise leave it with nothing.
241                 Nothing -> return Nothing
242             return (val, e)
243     case res of
244         -- If all of that succeeded
245         Right (val, e) -> do
246             -- Write the updated state back to the reference
247             liftIO $ writeIORef ref e
248             -- The serialize to JSON
249             case val of
250                 -- Nothing returns an empty object.
251                 Nothing -> return "{\"type\": \"empty\"}"
252                 Just val' -> case val' of
253                     -- Lists are Tables are handeled differently
                        from other values so the can be displayed
                        differently on
254                     -- the front-end.
```

```
255                     -- Lists have a values field which is a list of
                           strings of all of the list values displayed.
256                     -- Tables have two fields, one for the variables
                           which is simple a list of strings
257                     -- And one for all the values, similar to the
                           representation for Lists.
258                     SList ls -> return $ "{ \"type\": \"list\",
                           \"values\": [" ++ intercalate "," (map (show
                           . showValue) ls) ++ "]}"
259                     STbl (Table vars vals) -> return $ "{ \"type\":
                           \"table\", \"vars\": [" ++ intercalate ","
                           (map show vars) ++ "], \"vals\": " ++ show
                           vals ++ "}"
260                     other -> return $ "{ \"type\": \"value\",
                           \"value\": \"" ++ escape (showValue other) ++
                           "\"}"
261         -- If its an error, indicate this and then also return its
               description.
262         Left err -> return $ "{\"type\": \"error\", \"error\": \""
               ++ escape (describeError err) ++ "\"}"
263
264 -- Main entry point for the entire server side.
265 main :: IO ()
266 --     Run a ZMQ server.
267 main = runZMQ $ do
268     -- Create a new empty variable state.
269     ref <- liftIO $ newIORef (Env [])
270     -- Create a fresh socket
271     sock <- socket Rep
272     -- Bind it to the place specified by the program arguments
273     -- Because ZMQ operates exactly the same regardless of the actual
274     -- Socket type, this could use ipc, tcp etc. So the server
275     -- Could be on the same machine or remote.
276     [loc] <- liftIO getArgs
277     bind sock loc
278     forever $ do
279         -- recieve one line of input from the socket
280         line <- receive sock
281         -- Apply replStep to this
282         ret <- replStep ref (unpack line)
```

```
283         -- Print the return JSON
284         liftIO $ print ret
285         -- And then send it back through the socket.
286         send sock [] (pack ret)
```

A.4 CLIENT COMPONENT

A.4.1 RENDERER PROCESS

INDEX.HTML

```
1  <!DOCTYPE html>
2  <html>
3
4  <head>
5      <meta charset="utf-8">
6      <meta name="viewport" content="width=device-width,
           initial-scale=1, shrink-to-fit=no">
7      <title>AME</title>
8      <link rel="stylesheet"
          href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta
9      /css/bootstrap.min.css"
          integrity="sha384-/Y6pD6FV/Vv2HJnA6t+vslU6fwYXjCFtcEp
10     HbNJ0lyAFsXTsjBbfaDjzALeQsN6M" crossorigin="anonymous">
11     <link rel="stylesheet" href="main.css">
12     <script type="text/x-mathjax-config">
13         MathJax.Hub.Config({tex2jax: {inlineMath: [['$','$']]}});
14     </script>
15 </head>
16
17 <body class="bg-light">
18     <nav class="navbar navbar-expand navbar-dark bg-dark sticky-top">
19         <a class="navbar-brand" href="#">AME</a>
20         <div class="navbar-collapse collapse">
21             <ul class="nav navbar-nav">
22                 <li class="nav-item btn-group">
23                     <a class="nav-link dropdown-toggle"
                           id="dropdownMenuLink" data-toggle="dropdown"
                           aria-haspopup="true" aria-expanded="false">
24                         File
25                     </a>
```

```
26
27                  <div class="dropdown-menu no-select"
                        aria-labelledby="dropdownMenuLink">
28                      <a class="dropdown-item" href="#"
                            id="new">New</a>
29                      <a class="dropdown-item" href="#"
                            id="open">Open</a>
30                      <a class="dropdown-item" href="#"
                            id="save">Save</a>
31                      <a class="dropdown-item" href="#"
                            id="save-as">Save As</a>
32                  </div>
33              </li>
34          </ul>
35      </div>
36  </nav>
37
38  <div class="container-fluid" id="main-container">
39      <div id="card-container">
40          <br/>
41      </div>
42      <form class="fixed-bottom bg-dark" id="input-form">
43          <input class="form-control bg-dark text-light"
                type="text" placeholder="..." id="main-input">
44      </form>
45  </div>
46
47
48  <script src="https://code.jquery.com/jquery-3.2.1.slim.min.js"
        integrity="sha384-KJ3o2DKtIkvYIK3UENzmM7KCkRr/rE9/Qpg6aAZGJwFD
49  MVNA/GpGFF93hXpG5KkN" crossorigin="anonymous" onload="window.$ =
        window.jQuery = module.exports;"></script>
50  <script
        src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.11.0/u
51  md/popper.min.js"
        integrity="sha384-b/U6ypiBEHpOf/4+1nzFpr53nxSS+GLCkfwBdF
52  NTxtclqqenISfwAzpKaMNFNmj4" crossorigin="anonymous"></script>
53  <script
        src="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta/js/
54  bootstrap.min.js" integrity="sha384-h0AbiXch4ZDo7tp9hKZ4Ts
```

```
55      Hbi047NrKGLO3SEJAg45jXxnGIfYzk4Si90RDIqNm1"
            crossorigin="anonymous"></script>
56      <script type="text/javascript" async
            src="https://cdnjs.cloudflare.com/ajax/libs/mathjax/2.7
57      .2/MathJax.js?config=TeX-MML-AM_CHTML"></script>
58      <script src="renderer.js"></script>
59  </body>
60
61  </html>
```

MAIN.CSS

```
1  nav {
2      -webkit-app-region: drag;
3  }
4
5  #main-input {
6      border-radius: 0 !important;
7  }
8
9  #main-input:focus {
10     border-color: transparent !important;
11 }
12
13 #card-container {
14     position: absolute;
15     left: 10px;
16     right: 10px;
17     top: 58px;
18     bottom: 38px;
19     overflow-y: scroll;
20     overflow-x: hidden;
21 }
22
23 .card {
24     margin: 10px;
25 }
26
27 .no-select {
28     -webkit-user-select: none;
29     user-select: none;
```

```
}

table {
    text-align: center;
}

th {
    text-align: center;
}
```

RENDERER.JS

```
const { ipcRenderer, remote } = require('electron');
const dialog = remote.dialog;
const fs = remote.require('fs');

// Here there is a set of Card classes which can be used to build
   cards to display input and output
// Card is the base class and it accepts a Content class, which is
   some kind of Panel (TextPanel etc)
// The card has two methods, generate and typeset, for generating
   the html, and a hook to be executed when it is
// typeset.

// When writing a card
function write_card(card, index) {
    console.log(card);
    if (index !== undefined) {
        //If we have an index, look for the corresponding input card
           and then go up
        //two levels and append after that.
        $($("#card-container .input-card").get(index -
           1)).parent().parent().after(card.generate())
    } else {
        //Otherwise append the generated html to the end of the card
           container
        $("#card-container").append(card.generate());
    }
    // Call the typeset hook
    card.typeset();
}
```

```
24
25 function Card(content, float, classes) {
26     this.content = content
27     this.classes = classes || "";
28     this.float = float || "right";
29     // A regular card just applies the classes its given
30     // And adds float right if there is no float supplied, otherwise
       the user option is used.
31     this.generate = function() {
32         return `
33         <div class="row justify-content-${this.float == "right" ?
           "end" : "start"}">
34             <div class="col-8">
35                 <div class="card ${this.classes}">
36                     ${this.content.generate()}
37                 </div>
38             </div>
39         <\div><br/>
40         `
41     }
42
43     this.typeset = function() {
44         this.content.typeset();
45     }
46
47     return this;
48 }
49
50 // A TextPanel is a simple panel that displays raw text in the card
   body
51 // and optionally has extra classes added.
52 function TextPanel(text, classes) {
53     this.text = text;
54     this.classes = classes || "";
55
56     this.generate = function() {
57         return `
58         <div class="card-body ${this.classes}">
59             ${this.text}
60         </div>
```

```
61          `
62      }
63
64      this.typeset = function() {}
65      return this;
66  }
67
68  // MathJaxPanel wraps the input in $$ so that it is typeset
69  // by MathJax as Latex
70  function MathJaxPanel(text) {
71      // Make this a TextPanel where the text is wrapped with $$
72      TextPanel.bind(this)("$$" + text + "$$");
73      // When typesetting, tell MathJax that it should redraw the page.
74      this.typeset = function() {
75          MathJax.Hub.Typeset();
76      }
77      return this;
78  }
79
80  // The CodePanel is similar to a TextPanel but uses the <code>
        element
81  // instead of <div> so that the output uses a monospace font.
82  function CodePanel(text) {
83      this.text = text;
84
85      this.generate = function() {
86          return `<code class="card-body bg-dark
                text-light">${this.text}</code>`
87      }
88
89      this.typeset = function() {}
90
91      return this;
92  }
93
94  // An InputCard is a card with bg-dark to give a dark background,
95  // that floats to the left, and has a single code panel with the
        given text.
96  function InputCard(text) {
97      return Card.bind(this)(new CodePanel(text), "left", "bg-dark
```

```
        input-card")
98  }
99
100 // A ValueCard displays a latex value using MathJax
101 // And it floats right.
102 function ValueCard(text) {
103     return Card.bind(this)(new MathJaxPanel(text))
104 }
105
106 // ErrorCards have a red border and red text
107 // with a text panel with the error message in.
108 function ErrorCard(text) {
109     return Card.bind(this)(new TextPanel(text, "text-danger"),
        "right", "border-danger")
110 }
111
112 //For { "type": "list" } results
113 function ListPanel(vals) {
114     this.vals = vals;
115
116     this.generate = function() {
117         res = []
118         for (val of this.vals) {
119             // Make a list group item for each value, wrapping the
                value in $ $ for inline MathJax
120             res.push(`<li class="list-group-item">$${val}$</li>`)
121         }
122         // The put the whole thing in a list group
123         return `<ul class="list-group-flush
            card-body">${res.join("\n")}</ul>`
124     }
125
126     this.typeset = function() {
127         MathJax.Hub.Typeset();
128     }
129
130     return this
131 }
132
133 function TablePanel(vars, vals) {
```

```
134     this.vars = vars;
135     this.vals = vals;
136 
137     this.generate = function() {
138         //Start of with a table, and start a row in the table header
139         res = ["<table class=\"table card-body\">", "<thead>",
                "<tr>"]
140         for (v of this.vars) {
141             // Add a heading for each variable
142             res.push(`<th scope="col">${v}$</th>`)
143         }
144         //End the row and the header, and start the body
145         res.push("</tr>")
146         res.push("</thead>")
147         res.push("<tbody>")
148         //Transpose the list of values so instead of having a list
                of columns we have a list of rows
149         var tvals = this.vals[0].map((col, i) => this.vals.map(row
                => row[i]));
150         for (row of tvals) {
151             //Start a row
152             res.push("<tr>")
153             for (e of row) {
154                 //Make an element for every value in the row
155                 res.push(`<td>${e}$</td>`)
156             }
157             //End the row
158             res.push("</tr>")
159         }
160         //End the body and the table
161         res.push("</tbody>")
162         res.push("</table>")
163         return res.join("\n")
164     }
165 
166     this.typeset = function() {
167         MathJax.Hub.Typeset();
168     }
169 
170     return this
```

```
171 }
172
173 function needs_to_scroll() {
174     var $container = $("#card-container");
175     //If the scrolled distance, minus the initial scroll distance
176     // is equal to the height, it means we have scrolled to the
        bottom of the window,
177     // so we need to automatically scroll it when more cards are
        added
178     return $container.prop("scrollHeight") - $container.scrollTop()
        == $container.outerHeight();
179 }
180
181 function do_scroll(needed) {
182     if (needed) {
183         var $container = $("#card-container");
184         // If we need to, set the intial scroll value to the value
            we just calculated.
185         $container.scrollTop($container.prop("scrollHeight") -
            $container.outerHeight())
186     }
187 }
188
189 // Whenever we get a reply back from the server.
190 ipcRenderer.on('reply', (event, obj, index, cb) => {
191     // needs_to_scroll and do_scroll must be seperate
192     // because we need to check if we're at the bottom
193     // BEFORE we add new stuff.
194     var needed = needs_to_scroll();
195
196     switch (obj.type) {
197         case "value":
198             write_card(new ValueCard(obj.value), index);
199             break;
200         case "error":
201             write_card(new ErrorCard(obj.error), index);
202             break;
203         case "list":
204             write_card(new Card(new ListPanel(obj.values)), index);
205             break;
```

```
206         case "table":
207             write_card(new Card(new TablePanel(obj.vars, obj.vals)),
                    index);
208             break;
209     }
210
211     do_scroll(needed);
212
213     //If we have requested a callback on this card, execute it.
214     if (cb) {
215         console.log("callback");
216         reply_callback()
217     }
218 });
219
220 let curPath = null;
221 let input = [];
222 let reply_callback;
223
224 // Whenever we press enter while typing in the main input
225 $("#input-form").submit(function(evt) {
226     // Prevent this event from bubbling up and causing a page refresh
227     evt.preventDefault();
228     var needed = needs_to_scroll();
229     var x = $("#main-input").val();
230     // Make a new input card with the value of the main input
231     var card = new InputCard(x);
232     write_card(card);
233     //Save the command to the input log
234     input.push(x);
235     let index = input.length;
236     // Write the card to the screen and then send the input off to
            the server.
237     ipcRenderer.send('command', x, index);
238     do_scroll(needed);
239     $("#main-input").val("");
240 });
241
242 $("#new").click(function(evt) {
243     evt.preventDefault();
```

```
244     //Restarting the server clears the environment
245     ipcRenderer.send("restart");
246     //Remove all the cards
247     $("#card-container").children().remove()
248     //We don't have a file open so reset curPath
249     curPath = null;
250     //Reset the input log
251     input = []
252 })
253 
254 $("#open").click(function(evt) {
255     evt.preventDefault();
256     //Show a file dialog
257     dialog.showOpenDialog({
258         title: 'Open File',
259         filters: [{
260             //Only allow opening .ame files
261             name: 'AME Files', extensions: ['ame']
262         }],
263         properties: ['openFile']
264     }, ([path]) => {
265         //Restarting the server, remove the existing cards
266         ipcRenderer.send("restart");
267         $("#card-container").children().remove()
268         //Read the file and parse it as JSON.
269         //It should be a list of strings to feed to the server
270         let vals = JSON.parse(fs.readFileSync(path))
271         //Set the current path we're editing
272         curPath = path
273         input = []
274         //Setup a fake MathJax Hub so that we dont call Typeset
275         //Too many times - calling it too fast will cause some
              glitches
276         //Where a value is duplicated.
277         var realhub = MathJax.Hub;
278         MathJax.Hub = {
279             Typeset: function() {}
280         }
281         for (let [i, val] of vals.entries()) {
282             //Write the input card and add the input to the log
```

```
283             write_card(new InputCard(val))
284             input.push(val)
285             if (i < vals.length - 1) {
286                 //If this is not the last one, send a normal command
287                 ipcRenderer.send("command", val, input.length,
                        false);
288             } else {
289                 //If this is the last one, add a callback for when
                        the card is done
290                 //Having a global callback variable for this is a
                        bit hacky but neccessary
291                 //because i cant pass functions from the renderer to
                        the main process and then back
292                 reply_callback = function() {
293                     //When the last one is added restore the hub to
                            the real hub
294                     MathJax.Hub = realhub;
295                     //Typeset the whole thing
296                     realhub.Typeset();
297                     //And scroll to the bottom
298                     do_scroll(true);
299                 }
300                 ipcRenderer.send("command", val, input.length, true);
301             }
302         }
303     })
304 })
305
306 $("#save").click(function(evt) {
307     evt.preventDefault();
308     //If we are not editing a specific path,
309     //This behaves like a save-as
310     if (curPath == null) {
311         $("#save-as").click();
312     } else {
313         //Otherwise just serialise the input log to JSON
314         //and write it to the file
315         fs.writeFileSync(curPath, JSON.stringify(input));
316     }
317 })
```

```
318
319 $("#save-as").click(function(evt) {
320     evt.preventDefault();
321     dialog.showSaveDialog({
322         title: 'Save File',
323         filters: [{
324             name: 'AME Files', extensions: ['ame']
325         }]
326     }, (path) => {
327         //Pick a path and save the input log to it.
328         fs.writeFileSync(path, JSON.stringify(input));
329         //Save this as the current path, so further save operations
              will not open a dialog.
330         curPath = path;
331     })
332 })
```

A.4.2 MAIN PROCESS

MAIN.JS

```
1  const electron = require('electron')
2  const zmq = require('zeromq');
3
4  const { app, ipcMain, BrowserWindow } = electron
5
6  const child_process = require('child_process')
7  const fs = require('fs')
8
9  const path = require('path')
10 const url = require('url')
11
12 const process_m = require("process")
13
14 let mainWindow
15 let socket
16 let renderer
17 let process
18 let args
19
20 function createWindow() {
```

```
21      //Create a new window
22      mainWindow = new BrowserWindow({
23          width: 800,
24          height: 600
25      })
26      //Navigate it to index.html
27      mainWindow.loadURL(url.format({
28          pathname: path.join(__dirname, 'index.html'),
29          protocol: 'file:',
30          slashes: true
31      }))
32      //When its closed, reset this variable to null so we dont
            receive any more events
33      mainWindow.on('closed', function() {
34          mainWindow = null
35      })
36  }
37
38  function startServer() {
39      args = []
40      process =
            child_process.spawn(/*path.join(path.dirname(process_m.argv[0]),
            */"ame-repl"/*)*/, ["tcp://127.0.0.1:8081"], {stdio:
            'inherit'})
41      // Make a ZMQ socket
42      socket = zmq.socket('req');
43      // Connect it to the server.
44      socket.connect("tcp://127.0.0.1:8081")
45      console.log("server started!");
46
47      //Wait for messages from the server.
48      socket.on('message', function(msg) {
49          console.log(msg.toString());
50          //Since commands are processed in-order, the index and
                callback for this command
51          //will be at the top of the list
52          let [index, cb] = args.pop();
53          // Parse the reply and then send it on to the browser window
                JS script.
54          var obj = JSON.parse(msg.toString());
```

```
        renderer.send('reply', obj, index, cb);
    })
}

// When the app is loaded
app.on('ready', function() {
    createWindow();
    startServer();
})

// Quit when all windows are closed.
app.on('window-all-closed', function() {
    app.quit()
})

// When we get a command from the browser window
ipcMain.on('command', (event, command, index, cb) => {
    if (renderer == undefined) {
        // Update the place for us to send the reply to if needed
        renderer = event.sender;
    }
    //add the index and callback to the queue
    args.unshift([index, cb])
    // Send this command on the ZMQ socket to the server.
    socket.send(command);
});

ipcMain.on('restart', (event) => {
    process.kill();
    console.log("killed!");
    startServer();
});
```

A.5 TESTING

```
import subprocess
import zmq
import json
import random
import blessed
```

```
6  import numpy as np
7  import numpy.linalg as l
8  from numpy.linalg import LinAlgError
9  from scipy import stats
10 import math
11
12 term = blessed.Terminal(force_styling=True)
13
14 # A class that can interact with the server process of AME
15 # to perform tests
16 class AMERepl:
17     def __init__(self, ctx):
18         # make a zmq socket
19         self.sock = ctx.socket(zmq.REQ)
20         self.proc = None
21
22     #to start up,
23     def start(self):
24         # start up a server, making sure to silence the output so we
25         # dont get debugging messages
26         self.proc = subprocess.Popen(["ame-repl", "tcp://*:8080"],
               stdout=subprocess.DEVNULL)
27         # connect to the server
28         self.sock.connect("tcp://127.0.0.1:8080")
29
30     #to stop,
31     def stop(self):
32         #disconnect and then kill the server.
33         self.sock.close()
34         self.proc.kill()
35
36     def submit(self, s):
37         #send a string to the server
38         self.sock.send(s.encode())
39         #wait for a response, parse the JSON and return it.
40         return json.loads(self.sock.recv().decode())
41
42 #construct a random matrix of size n
43 def random_matrix(n=None):
44     if n is None:
```

```
45          #pick randomly if an n is not provided
46          n = random.randrange(2, 5)
47      ls = [[random.randrange(1, 10) for _ in range(n)] for _ in
            range(n)]
48      p = '[' + ", ".join([str(l) for l in ls]) + ']'
49      return ls, p
50
51  #construct a random vector of size n
52  def random_vector(n=None):
53      if n is None:
54          n = random.randrange(2, 5)
55      ls = [random.randrange(1, 10) for _ in range(n)]
56      p = '[' + ", ".join(map(str, ls)) + ']'
57      return ls, p
58
59  #construct a random polynomial with degree at most n
60  def random_poly(n=5):
61      rl = [random.randrange(1, 100) for _ in
            range(random.randrange(1, n))]
62      p = " * ".join("(x - %s)" % i for i in rl)
63      return rl, p
64
65  #construct a random dataset in one variable with a random length
66  def random_table1():
67      rl = [random.randrange(10) for _ in range(random.randrange(10,
            20))]
68      p = str(rl)
69      return rl, p
70
71  #construct a random dataset of two variables with a random length
72  def random_table2():
73      rl1 = [random.randrange(1, 10) for _ in
            range(random.randrange(10, 20))]
74      p1 = str(rl1)
75      rl2 = [random.randrange(1, 10) for _ in range(len(rl1))]
76      p2 = str(rl2)
77      return rl1, rl2, p1, p2
78
79  #Test classes consist of two functions, generate,
80  #which produces one random testcase, and check
```

```
#which takes the random test case and the response
#from the server and checks it.

class MatrixDeterminant:
    name = "Matrix Determinants"

    def generate(self, i):
        ls, p = random_matrix()
        l1 = "determinant of " + p
        return (ls, [l1])

    def check(self, ls, v):
        #determinant should be a value
        assert(v['type'] == 'value')
        #and should be close to the value numpy calculates
        assert(abs(l.det(ls) - float(v['value'])) < 0.00001)

class MatrixMultiply:
    name = "Matrix Multiplication"

    def generate(self, i):
        ls, p = random_matrix()
        ls2, p2 = random_matrix(n=len(ls))
        l1 = p + " * " + p2
        return ([ls, ls2], [l1])

    def check(self, lss, v):
        assert(v['type'] == 'value')
        out = np.matrix(lss[0]) * np.matrix(lss[1])
        v['value'] = [[int(float(f)) for f in s.split("&")] for s in
            v['value'][15:-13].split("\\\\")]
        assert(all(all(a == b for (a, b) in zip(ra, rb)) for (ra,
            rb) in zip(v['value'], out.tolist())))

class MatrixInverse:
    name = "Matrix Inverses"

    def generate(self, i):
        ls, p = random_matrix()
        l1 = "the inverse of " + p
```

```
119         return (ls, [l1])
120 
121     def check(self, ls, v):
122         try:
123             #try and compute the inverse matrix
124             out = l.inv(ls)
125             #if we can, then we should have got a value
126             assert(v['type'] == 'value')
127             #parse the latex source code.
128             v['value'] = [[float(f) for f in s.split("&")] for s in
                    v['value'][15:-13].split("\\\\")]
129             #check all the values are close to the predicted ones
130             assert(all(all(abs(a - b) < 0.0001 for (a, b) in zip(ra,
                    rb)) for (ra, rb) in zip(v['value'], out.tolist())))
131         except LinAlgError:
132             #if the matrix is not invertible, we should have got an
                    error.
133             assert(v['type'] == 'error')
134 
135 class SimulSolve:
136     name = "Linear Simultaneous Equations"
137 
138     def generate(self, i):
139         #generate a coefficient matrix
140         ls, _ = random_matrix()
141         #and an answer vector
142         rs, _ = random_vector(len(ls))
143         v = "abcdefghijklmnopqrstuv"
144         p = []
145         for l, r in zip(ls, rs):
146             #using the alphabet as variables names
147             #generate one equation from a row of the coefficient
                    matrix
148             p.append(" + ".join("%s * %s" % (a, b) for (a, b) in
                    zip(l, v)) + " = " + str(r))
149         l2 = ["let %s = a variable %s" % (x, x) for x in v[:len(rs)]]
150         l1 = "solve { " + "; ".join(p) + " }"
151         return ([ls, rs], l2 + [l1])
152 
153     def check(self, lss, v):
```

```
154         ls, rs = lss
155         try:
156             #try and solve the equations
157             ss = np.linalg.solve(np.array(ls), np.array(rs)).tolist()
158             #if we can, then we should hae a list of solutions
159             assert(v['type'] == 'list')
160             for r, l in zip(ss, v['values']):
161                 #check that each solution is close to the predicted
                        solution
162                 assert(abs(float(l) - r) < 0.001)
163         except LinAlgError:
164             #if the system could not be solved, we expect an error.
165             assert(v['type'] == 'error')
166 
167 class PolySolve:
168     name = "Solver"
169 
170     def generate(self, i):
171         rl, p = random_poly()
172         l1 = "let f(x) = " + p
173         l2 = "solve f = 0"
174         return (list(set(rl)), [l1, l2])
175 
176     def check(self, rl, r):
177         #polynomials may have one solution or more solutions
178         #so we need both lists and values.
179         assert(r['type'] in ('list', 'value'))
180         rl.sort()
181         #if its a value
182         if r['type'] == 'value':
183             #wrap it in a list so we dont have to worry about it
184             r['values'] = [r['value']]
185         #convert all the solutions to floats
186         r['values'] = [float(v) for v in r['values']]
187         r['values'].sort()
188         #there should be as many solutions as there are were roots
                initially
189         assert(len(rl) == len(r['values']))
190         for v in r['values']:
191             #check that each solution is close to one of the roots.
```

```
192             assert(any(abs(v - r) < 10**(-3) for r in rl))
193
194 class PolySub:
195     name = "Evaluation & Substitution"
196
197     def generate(self, i):
198         rl, p = random_poly()
199         k = random.randrange(10)
200         l1 = "let f(x) = " + p
201         l2 = "evaluate (substitute x = " + str(k) + " in f)"
202         return ([rl, k], [l1, l2])
203
204     def check(self, rls, v):
205         rl, k = rls
206         #we should get a value
207         assert(v['type'] == 'value')
208         #parse it
209         r = float(v['value'])
210         nr = 1
211         #and then compute the substitution ourselves
212         for l in rl:
213             nr *= (k - l)
214         #and then check that they are close.
215         assert(abs(r - nr) < 0.01)
216
217
218 class PolyIntDiff:
219     name = "Integration & Differentiation"
220
221     def generate(self, i):
222         rl, p = random_poly(4)
223         rl.append(0)
224         l1 = "let f(x) = x * " + p
225         #integrating and differentiating a function with no constant
              part
226         #should get us back the same function
227         l2 = "let g = integrate (differentiate f)"
228         #so solving it should yield the same solutions
229         l3 = "solve g = 0"
230         return (rl, [l1, l2, l3])
```

```
231
232     def check(self, rl, r):
233         #this is just the same as the PolySolver checker
234         assert(r['type'] in ('list', 'value'))
235         rl.sort()
236         if r['type'] == 'value':
237             r['values'] = [r['value']]
238         r['values'] = [round(float(v)) for v in r['values']]
239         r['values'].sort()
240         for v in r['values']:
241             assert(any(v == r for r in rl))
242
243 class StatVar:
244     name = "(Sample) Variance & Standard Deviation"
245
246     def generate(self, i):
247         rl, p  = random_table1()
248         l1 = "let t = a table where x = " + p
249         l2 = "the variance of x in t"
250         l3 = "the standard deviation of x in t"
251         l4 = "the sample variance of x in t"
252         l5 = "the sample standard deviation of x in t"
253         return (rl, [l1, [l2, l3, l4, l5]])
254
255     def check(self, rl, vs):
256         vv, sv, svv, ssv = vs
257         assert(vv['type'] == 'value')
258         r = float(vv['value'])
259         #compute the actual variance
260         nr = np.var(rl)
261         #check it against the measured value
262         assert(abs(r - nr) < 0.01)
263         #repeat this for the other operations.
264         assert(sv['type'] == 'value')
265         r = float(sv['value'])
266         nr = np.std(rl)
267         assert(abs(r - nr) < 0.01)
268         assert(svv['type'] == 'value')
269         r = float(svv['value'])
270         nr = np.var(rl, ddof=1)
```

```
271         assert(abs(r - nr) < 0.01)
272         assert(ssv['type'] == 'value')
273         r = float(ssv['value'])
274         nr = np.std(rl, ddof=1)
275         assert(abs(r - nr) < 0.01)
276 
277 #This is very similar to StatVar
278 class StatMean:
279     name = "Mean, Median & Mode"
280 
281     def generate(self, i):
282         rl, p = random_table1()
283         l1 = "let t = a table where x = " + p
284         l2 = "the mean of x in t"
285         l3 = "the median of x in t"
286         l4 = "the mode of x in t"
287         return (rl, [l1, [l2, l3, l4]])
288 
289     def check(self, rl, vs):
290         mean, median, mode = vs
291         assert(mean['type'] == 'value')
292         r = float(mean['value'])
293         nr = np.mean(rl)
294         assert(abs(r - nr) < 0.01)
295         assert(median['type'] == 'value')
296         r = float(median['value'])
297         nr = np.median(rl)
298         assert(abs(r - nr) < 0.01)
299         assert(mode['type'] == 'value')
300         r = float(mode['value'])
301         rl.sort(reverse=True)
302         nr = sorted(rl, key=rl.count)[-1]
303         assert(abs(r - nr) < 0.01)
304 
305 #And this is similar to StatMean
306 class StatCorrelation:
307     name = "Chi Squared, Pearson/Spearman Correlation"
308 
309     def generate(self, i):
310         rla, rlb, p1, p2 = random_table2()
```

```
311         l1 = "let t = a table where x = " + p1 + "; y = " + p2
312         l2 = "the chi squared value of x and y in t"
313         l3 = "the pearson correlation of x and y in t"
314         l4 = "the spearman correlation of x and y in t"
315         return ([rla, rlb], [l1, [l2, l3, l4]])
316
317     def check(self, rls, vs):
318         rla, rlb = rls
319         chi, pearson, spearman = vs
320         assert(pearson['type'] == 'value')
321         r = float(pearson['value'])
322         nr = stats.pearsonr(rla, rlb)[0]
323         assert(abs(r - nr) < 0.01)
324         assert(spearman['type'] == 'value')
325         #except we have to calculate spearmans r ourself
326         #since the numpy algorithm uses fancy tie-breaking logic
327         #that we shouldnt use.
328         r = float(spearman['value'])
329         rrla = sorted(rla)
330         rrlb = sorted(rlb)
331         nr = stats.pearsonr([rrla.index(r) for r in rla],
                [rrlb.index(r) for r in rlb])[0]
332         assert(abs(r - nr) < 0.01)
333         assert(chi['type'] == 'value')
334         r = float(chi['value'])
335         nr = stats.chisquare(rla, rlb).statistic
336         assert(abs(r - nr) < 0.01)
337
338 class SumMul:
339     name = "Sums & Products"
340
341     def generate(self, i):
342         rl, _ = random_vector()
343         l1 = " + ".join(map(str, rl))
344         l2 = " * ".join(map(str, rl))
345         return (rl, [[l1, l2]])
346
347     def check(self, rl, vs):
348         p, m = vs
349         assert(p['type'] == 'value')
```

```
350         r = float(p['value'])
351         nr = sum(rl)
352         assert(abs(r - nr) < 0.01)
353         assert(m['type'] == 'value')
354         r = float(m['value'])
355         nr = 1
356         for l in rl:
357             nr *= l
358         assert(abs(r - nr) < 0.01)
359
360 class SubDivPow:
361     name = "Subtraction, Exponentiation & Division"
362
363     def generate(self, i):
364         a, b = random.sample(range(1, 20), 2)
365         l1 = "%s - %s" % (a, b)
366         l2 = "%s / %s" % (a, b)
367         l3 = "%s ^ %s" % (a, b)
368         return ([a, b], [[l1, l2, l3]])
369
370     def check(self, ab, vs):
371         s, d, p = vs
372         a, b = ab
373         #standard by now.. compute the value ourself
374         #and assert that its close to the measure value...
375         assert(s['type'] == 'value')
376         r = float(s['value'])
377         nr = a - b
378         assert(abs(r - nr) < 0.01)
379         assert(d['type'] == 'value')
380         r = float(d['value'])
381         nr = a / b
382         assert(abs(r - nr) < 0.01)
383         assert(p['type'] == 'value')
384         r = float(p['value'])
385         nr = a ** b
386         assert(100 * abs(r - nr)/r < 1)
387
388 class Trig:
389     name = "Sin, Cos & Tan"
```

```
390
391     def generate(self, i):
392         a = (math.pi/2)*(random.random()*2 - 1)
393         l1 = "sin(%s)" % a
394         l2 = "cos(%s)" % a
395         l3 = "tan(%s)" % a
396         return (a, [[l1, l2, l3]])
397
398     def check(self, a, vs):
399         s, c, t = vs
400         assert(s['type'] == 'value')
401         r = float(s['value'])
402         nr = math.sin(a)
403         assert(abs(r - nr) < 0.01)
404         assert(c['type'] == 'value')
405         r = float(c['value'])
406         nr = math.cos(a)
407         assert(abs(r - nr) < 0.01)
408         assert(t['type'] == 'value')
409         r = float(t['value'])
410         nr = math.tan(a)
411         assert(abs(r - nr) < 0.01)
412
413 class InvTrig:
414     name = "Inverse Sin, Cos & Tan"
415
416     def generate(self, i):
417         a = random.random() * 2 - 1
418         l1 = "asin(%s)" % a
419         l2 = "acos(%s)" % a
420         l3 = "atan(%s)" % a
421         return (a, [[l1, l2, l3]])
422
423     def check(self, a, vs):
424         s, c, t = vs
425         assert(s['type'] == 'value')
426         r = float(s['value'])
427         nr = math.asin(a)
428         assert(abs(r - nr) < 0.01)
429         assert(c['type'] == 'value')
```

```
430         r = float(c['value'])
431         nr = math.acos(a)
432         assert(abs(r - nr) < 0.01)
433         assert(t['type'] == 'value')
434         r = float(t['value'])
435         nr = math.atan(a)
436         assert(abs(r - nr) < 0.01)
437
438 class LogSqrt:
439     name = "Log & Sqrt"
440
441     def generate(self, i):
442         a = 10*random.random()
443         l1 = "log(%s)" % a
444         l2 = "sqrt(%s)" % a
445         return (a, [[l1, l2]])
446
447     def check(self, a, vs):
448         l, s = vs
449         assert(l['type'] == 'value')
450         r = float(l['value'])
451         nr = math.log(a)
452         assert(abs(r - nr) < 0.01)
453         assert(s['type'] == 'value')
454         r = float(s['value'])
455         nr = math.sqrt(a)
456         assert(abs(r - nr) < 0.01)
457
458 class AbsExp:
459     name = "Abs & Exp"
460
461     def generate(self, i):
462         a = random.random() * 2 - 1
463         l1 = "exp(%s)" % a
464         l2 = "|%s|" % a
465         return (a, [[l1, l2]])
466
467     def check(self, a, vs):
468         e, b = vs
469         assert(e['type'] == 'value')
```

```
470         r = float(e['value'])
471         nr = math.exp(a)
472         assert(abs(r - nr) < 0.01)
473         assert(b['type'] == 'value')
474         r = float(b['value'])
475         nr = abs(a)
476         assert(abs(r - nr) < 0.01)
477
478 #This is the framework that holds all the tests together and produces
479 #a nice colourful report.
480 class Tester:
481     def __init__(self, *t):
482         self.ts = t
483         #start up a server
484         self.r = AMERepl(zmq.Context())
485
486     def test(self, n=100):
487         self.r.start()
488         tf = 0
489         ts = 0
490         #for each test that we are performing:
491         for t in self.ts:
492             print(term.bright_white("\nTesting: " + t.name))
493             f = 0
494             #repeat n times (usually 100)
495             for i in range(n):
496                 #generate a random test case
497                 s, ls = t.generate(i)
498                 #occasionally print how many we've generated
499                 if (i + 1) % 200 == 0:
500                     print(term.blue("Generated %s tests." % (i + 1)))
501                 out = None
502                 #for every command that we provide
503                 for l in ls:
504                     if isinstance(l, list):
505                         #if it is a list, submit every command
506                         #in the list
507                         out = [self.r.submit(ll) for ll in l]
508                     else:
509                         #otherwise just submit the whole thing
```

```
510                         out = self.r.submit(l)
511                 try:
512                     #try and verify the output against out input
513                     t.check(s, out)
514                 except Exception as e:
515                     f += 1
516                     #but if it doesnt work increase the failure
                            count and report it.
517                     print(term.bright_red("Failed test: %s" % s))
518                     print(term.bright_red(str(e)))
519             else:
520                 #when we're done with the repetitions,
521                 #print how many we failed and passed and our ratios.
522                 print(term.bright_green("Passed %s tests." % (n -
                        f)))
523                 if f > 0:
524                     print(term.bright_red("Failed %s tests." % f))
525                 print(term.bright_white("Overall success rate: %s%%"
                        % round((n - f) / n * 100, 3)))
526                 #then update the total failure and success counts.
527                 tf += f
528                 ts += n - f
529         #when we're done with all the tests, stop the server and
               print the final statistics
530         self.r.stop()
531         print(term.bright_green("\nPassed %s tests total." % ts))
532         print(term.bright_red("Failed %s tests total." % tf))
533         print(term.bright_white("Overall success rate: %s%%" %
                round(ts / (ts + tf) * 100, 3)))
534
535 #Our default selection of tests is 1000 cases of
536 #every test.
537 def test(n=1000):
538     Tester(MatrixDeterminant(),
539            MatrixMultiply(),
540            MatrixInverse(),
541            SimulSolve(),
542            PolySolve(),
543            PolySub(),
544            PolyIntDiff(),
```

```
545             StatMean(),
546             StatVar(),
547             StatCorrelation(),
548             SumMul(),
549             SubDivPow(),
550             Trig(),
551             InvTrig(),
552             LogSqrt(),
553             AbsExp()).test(n)
554 
555 #if we are executing this as python3 test.py instead of
556 #importing it as a library, then execute the
557 #default tests.
558 if __name__ == "__main__":
559     test()
```