An Overview of: Polynomial Manipulation Language (PML)

Authors: Melinda Agyekum (mya2001@columbia.edu)
Shezan Baig (sb2284@columbia.edu)
Hari Kurup (hgk2101@columbia.edu) –Group Leader
Subadhra Sridharan (ss2355@columbia.edu)

INTRODUCTION

Polynomial equations are mathematical representations of both theoretical and practical problems and are used in a variety of professional fields. These mathematical expressions are written as the sum of the products of numbers and variables. A few practical applications which rely heavily on polynomial expressions are: missile trajectory, weather forecasting, spacecraft re-entry, building construction, and financial market calculations.

One example of a real world problem where polynomial functions are applied is testing the effectiveness of a new drug. The quantity of medication given to a person under testing can be varied and the improvement or degradation can be noted. There may be other variables, which can be applied to the same problem, like the patient's age, weight, and other existing medical conditions, if any.

Regardless of the situation, the use of polynomials can provide an individual better insight into a problem. Although polynomials are extremely important components of algebra, solving these problems manually can be a time-consuming and tedious process. It is because of this, there is a need for a system, which will perform computations in a methodical way irrespective of the problem at hand.

The polynomial manipulation language (PML) tool is a programming language built for flexible manipulation of polynomial expressions. With its extensive set of built in operations and functions, PML can be used to specify an algorithm involving polynomials. In addition, this language is easy to understand, allowing this to be a user-friendly language for programmers to enjoy.

PML is designed for manipulating symbolic mathematical computations. In contrast to numerical computation, PML emphasizes computing with symbols representing mathematical concepts. The input to algorithms will be expressions or polynomial equations, while the output of the translations will be returned in algebraic form. From such an expression, one can deduce how the change in parameters will affect the result of computation. Although PML will be able to handle numbers and symbols with equal capacity, the primary role of this application is to facilitate symbolic computational programs.

Background

Given below is a brief description about types of polynomials and equations supported by PML.

Polynomials

A polynomial is a mathematical expression involving a sum of powers multiplied by coefficients. Broadly classified, there are two types of polynomials depending on the number of unique variables within the equations. These types are called univariate and multivariate polynomials.

Univariate Polynomial

A polynomial expressed in one variable is known as a univariate polynomial. An example of a univariate polynomial can be found below in Eq. 1.

$$c_i x^i + c_{i-1} x^{(i-1)} + \dots c_0 \qquad \textbf{\textit{(Eq. 1)}}$$

In the aforementioned expression c_i, c_{i-1},… terms each represent coefficients, while the superscripts represent the degree of each term. This polynomial has only one variable, which is represented by ‘x’.

Multivariate Polynomial

A polynomial expressed in more than one variable is known as a multivariate polynomial. An example of a multivariate polynomial can be found below in Equation 2.

$$c_i x^i y^i + c_{i-1} x^{(i-1)} y^{(i-1)} + \dots c_i \qquad \textbf{\textit{(Eq. 2)}}$$

This equation is expressed using two variables ‘x’ and ‘y’.

Linear Equations

Polynomials equations of the form

$$ax + b = c \qquad \textbf{\textit{(Eq. 3)}}$$

are called linear equations, having only one variable whose degree is one. All other equations not in the form mentioned by Eq. 3 are non-liner equations.

Quadratic Equations

All polynomial equations of the form

$$ax^2 + bx + c = 0 \qquad \textbf{\textit{(Eq. 4)}}$$

are called quadratic equations. Quadratic equations are second order degree equations and the roots of the equation can be determined using the quadratic formula.

Roots and Factoring

A root of a polynomial $P(z)$ is a number z_i such that $P(z_i) = 0$. A polynomial of n degrees has n roots.

A factor is any number that divides a given number evenly (without a remainder). If r is a root of a polynomial equation $f(x) = 0$, then $(x-r)$ is a factor of the polynomial $f(x)$.

PML supports all of the above-mentioned types of polynomials and equations, and more.

LANGUAGE FEATURES

PML contains several features, enabling it to be viewed as one of the most powerful symbolic equation language tools. Brief descriptions of these features are listed below:

Symbolic Interpretation
Unlike several applications on the market, PML provides a symbolic representation of polynomial equations. This symbolism allows the user to gain a more theoretical perspective of the function being performed, maintaining the likeness of which most polynomials are represented in algebra.

Most of the existing languages do not have the capability to accept a polynomial in its natural form and then manipulate it. This deficiency implies that users must create self-devised methods to enter polynomials to the program. Below is an example of feeding programs into a language other than PML:

(*User Enters)*> 2 4 -3 3 5 0 (*Program Interprets)*> $2x^4 - 3x^3 + 5x$

This input does not represent the actual polynomial representation. There can be many more of such creative input sequence. With PML the user will be able to directly enter the equation listed below.

$$2x^4 - 3x^3 + 5x \qquad \textbf{\textit{(Eq. 5)}}$$

Language Commands
Functions and keywords provided by PML are similar to the standard mathematics terminology; this makes PML easy and intuitive to use. Individuals with a working knowledge of symbolic mathematics and some programming background can easily start coding useful programs in PML.

Function Performance
Through the use of a very intuitive language, users will be able to perform operations such as addition, subtraction, multiplication, division, factorization, simplification, and differentiation of polynomials.

Equation Evaluation
PML will be equipped to handle a few numerical evaluations. The language will have the capability to solve for the numerical roots of an expression. Also, expressions will be able to be evaluated, provided that the user enters a number, which will be substituted for a variable.

User Customization
In addition to built-in functions, users will also be able to enter a polynomial and provide the program with steps on how to manipulate the equation.

Elimination of Error
PML does not have pointers. As a result it is not possible to write programs that can corrupt memories and cause systems to crash, making PML a stable language.

LANGUAGE IMPLEMENTATION

Functional Language
As opposed to an object-oriented language, this language will be implemented as a traditional functional language. One primary reason for this decision is that a functional language is much easier for the user to understand. Also, object-oriented concepts are not necessary in our domain.

The user can invoke operations through function calls as well as create their own functions. Recursive functions are also supported in the language. Many standard functions addition, subtraction, multiplication, division, and differentiation will be stored in standard libraries. However, the users will not be restricted to use the standard functions, and they can choose to write their own functions to perform the above-mentioned operations. Users will also be permitted to create their own libraries that can be linked together with many other programs.

Interpreted
The lexical scanner and parser will be created using ANTLR. The ANTLR system will produce the necessary information for the PML interpreter to execute the user's program. One convenience of an interpreted language is that it does not have to be compiled into machine code. This enables the language to be ported to many different platforms and architectures. The users can run a PML interpreter that was designed for a specific platform without the need to recompile their existing source code. The interpreter will then interpret the source code on the fly. A user can also share source code among a group of peers and be rest-assured that there will not be any problems running it. This is because the PML interpreter will work the same on all platforms and environments.

EXAMPLES OF PML

PML provides features that are easy to learn and use. Below are a few samples of the PML language and how it can be used to implement easy and efficient code.

term() - Function which takes a polynomial and returns an array that contains the individual terms of the polynomial. This function can be used in a program that adds two polynomials on n-degree.

Consider the PML code below in Figure 1.

```
>  p1 = 2x^3 + 3x^2 + 4x + 5x^5
>  polyTemp = term(p1) ;
```

Figure 1. PML Input Statements using the *term()* function

After the above statement is executed, the variable polyTemp will contain four elements:

```
> polyTemp[1] = 2x^3
> polyTemp[2] = 3x^2
> polyTemp[3] = 4x
> polyTemp[4] = 5x^5
```

Figure 2. PML Output from *term()* function

The term() function enables the user to access each polynomial term and operate and access them when necessary.

order - Operator which can be applied to the array polyTemp, located in Figure 1, to order the terms according to its degree. Terms will be arranged, within the array, in either ascending or descending order, depending on the parameter.

```
> order desc polyTemp;
> 5x^5 2x^3 3x^2 4x
> order asc polyTemp;
> 4x 3x^2 2x^3 5x^5
```

Figure 3. PML Statement using the *order* operator

The first line in Figure 3 will cause PML to rearrange the elements in polyTemp according to the degree of each term, from highest to lowest degree. The call in the third line will arrange the elements of polyTemp, from lowest to highest order of the degrees of each term.

PML also includes the use of more traditional operators. The '**+**' operator is used to add two terms of a polynomial. The code in Figure 4 is an example using the '+' operator.

> a = $2x^2$
> b= $4x^2$
> a + b = c
>c = $6x^2$

Figure 4. PML Code using the ‘+’ operator

invert() – Function used to change the magnitude of a term use the *invert()*. This function can be used to simulate the effect of moving a term to the other side of an equation. It can also be a very useful tool to implement subtraction of polynomials.

> invert($2x^2$)
> **$-2x^2$**
> invert(-4x)
> **4x**

Figure 5. PML Code using the *invert*() function

NOTE: We have indicated indices as superscripts in the above examples. This has been done for clarity and ease of understanding. However, while typing inputs to PML, the user will have to indicate the indices with a ‘^’ symbol, e.g; 2x^2. This is because PML takes its inputs from the standard console. Similarly, PML outputs will also indicate powers with the ‘^’ symbol.

PML SCOPE AND LIMITATION

PML is a preliminary venture in providing a language specifically built for symbolic mathematics. As such, the introductory version of PML is not aimed at handling the entire gamut of polynomial operations. PML will handle addition, subtraction, multiplication, division, factorization and differentiation of polynomials. No features or language support will be provided for higher operations like partial differentiation or integration.