Designing a Multivariate Polynomial Class

A Starting Point

Brent M. Dingle

Texas A&M University
2004

Abstract:

The need for and usage of multivariate polynomials, and more generally Computer Algebra Systems, is readily seen in the popularity of programs such as Mathematica, Maple and Matlab. And while the need for the continued development of such systems is obviously advantageous to research efforts across most scientific and mathematical fields there has been a severe lack of educational effort made in teaching the methods necessary for designing such systems. This paper is written as a place to begin development of a (C++) class for supporting work with multivariate polynomials. There will not be much discussion on specific algorithms as the emphasis is to be on designing data structures to support a diverse number of algorithms. The hope is that given a starting point the development of new polynomial algorithms will be encouraged in younger computer scientists and programmers.

1 Introduction

This paper is created as an aid to those who may find it necessary to create their own data types or classes to represent a multivariate polynomial. While this task has obviously been accomplished for many years, it is observed that there have been very few publications on the abstraction details necessary to accomplish the task (or at least such publications are not widely distributed). With this in mind this paper was written. Herein should be found a very basic and general outline of how to begin to code the classes necessary to model a multivariate polynomial. While there are many ways to do this, the method chosen here was based on simplicity and is presented at a novice level. It is hoped that from this simple starting point the beginning student might be able to advance his or her study in the art of programming, particularly towards the study of computer algebra or geometry related fields.

The general layout of this paper begins with some basic definitions discussed in section 2. In sections 3 and 4, it moves into univariate and multivariate representations and abstractions. Section 5 discusses the desired functionality and sections 6 and 7 present some possible ways to implement multiplication and division. Section 8 suggests some potential directions to continue and section 9 is a conclusion summary. Source code for some of the classes and functions, written in C++, can be found in the Appendices.

2 What is a multivariate polynomial?

For this paper it is first necessary to present some definitions. We will begin by defining a *polynomial* as an algebraic expression that is a sum of terms, where each term contains only variables with (nonnegative) integer exponents and real coefficients. Next we will define a *univariate polynomial* as a polynomial with one variable in each term and the variable in every term has the same name. Lastly, we will define a *multivariate polynomial* as a polynomial where each term may have one or more variables of different names and the variable names in any given term need not be identical to the variable names of any other term in the polynomial.

For example the following are univariate polynomials:

- $x^3 + 3x + 14$
- $53.6y^2 + 12y$
- $936h^{23} + 14h^6 - 34h^2 + h - 3$

Whereas the below are multivariate polynomials:

- $x + y + 12$
- $7xyz + abc + x - 16c$
- $x^2 + 3xy - y^2 + 4.5$
- $-x^3 - 3x - 14$

Notice in the context of this paper univariate polynomials are a subset of multivariate polynomials. Note also that constants are considered to be multiplied by a variable raised to the zero power. Further we will be discussing things in terms of real numbers, though most everything should easily and naturally extend to complex numbers.

3 Univariate Polynomial Representation

In Computer Science it is a relatively easy task to write code to accommodate the usage of univariate polynomials. The most often used representation is that of an array or list of a type defined as:

```
TYPE PolyTerm
     integer    exponent;
     real       coefficient;
END TYPE
```

This makes things very nice to implement as all the variables are assumed to be the same name (e.g. *x*) in all terms of all the polynomials. So adding or subtracting polynomials is just a matter of searching for terms with the same exponent and adding or subtracting their coefficients. Multiplication and division are just a matter of properly modifying exponents. Notice dividing a single PolyTerm by another single PolyTerm might cause negative exponents to appear which stretches the mathematical definition of polynomial. Further division of two polynomials, each of multiple terms, only succeeds if the division results in no remainder.

While these types of polynomials are useful for a large number of problems there often are times when more than one variable would be nice.

4 Abstraction of Multivariate Polynomials

4.1 The Atom

Before beginning with code (or functionality) it is best to find a good abstraction of what is needed. Let us consider how to describe a multivariate polynomial. Obviously we need exponents and coefficients. We also will need a way to keep track of variable names. This leads to a basic type defined as:

```
TYPE Atom
     Integer  exponent
     real     coefficent
     char     var_name
END TYPE
```

Notice the type name of Atom has been used. This is mostly to emphasize a relationship of construction that will become more obvious as we continue.

4.2 Constants – Special Atoms

An important design note is the realization that all constants should be associated with the same variable name. Even though any variable raised to the zero power is one, and thus $5a^0 + 10b^0 = 15$, that realization is not obvious for a computer when it is trying to find terms with the same variable names to add together. Thus for algorithmic reasons it would be best to use the same name for all constants. To do this it would be a good idea to reserve a variable name, say perhaps the *at symbol*, @, for usage only with constants. Thus the number zero as a type Atom would have an exponent of 0, a coefficient of 0 and a var_name of @. Likewise the number seven as a type Atom would have an exponent of 0, a coefficient of 7 and a var_name of @.

4.3 Particles

Notice that our Atom type really only allows for a single variable, yet each term of a multivariate polynomial may have many variables of different names multiplied together. So now we need to be a little more creative. Let us consider some examples of atoms:

- $3x^3$
- $10y^2$
- z

Any of the above could be terms of a polynomial. But if we multiply them together, for example $3x^3 \cdot 10y^2$ we get $30x^3y^2$ which could only be a term of a multivariate polynomial. This would seem obvious but not very meaningful. However, it gives a way we might represent terms of a polynomial. Specifically, assume we arbitrarily decided that 26 is the largest number of variable names capable of being used in a single term of a multivariate polynomial. Then we could define a new type as follows:

```
TYPE Particle
     Atom     atom_list[26]
     integer  num_atoms
END TYPE
```

Our choice of Particle as the type name is because a particle would clearly be composed of atoms and thus our type names imply the relationship between the types. To be mathematically correct our particles are terms of a polynomial. As for the data structure itself , since we have assumed a maximum of 26 allowable variable names an array of 26 atoms is sufficient to cover all possible particles and num_atoms would state specifically how many unique atoms the particle actually contains. This is a straightforward easy to implement method that wastes memory. So note that it could also be implemented as a list of atoms, rather than an array.

4.3.1 Particle Coefficients

Now the observant reader will immediately see that in this representation we might end up representing $30x^3y^2$ as $3x^310y^2$. This could be corrected by moving the coefficient out of the Atom type and placing it in the Particle type, but as long as we a careful in our

handling of the coefficients everything will work. Specifically if we set all the coefficients in atom_list[] to be one, except for atom_list[0]'s we will have no problem. It is noted that this does waste some memory, but in practice allows a little bit more flexibility in coding.

4.3.1 Maintaining Order in a Particle

Another important issue when creating particles will be maintaining the order of the variables in the list of atoms. For example it is necessary that the particle x^2yz be recognized as identical to yx^2z. The easiest way to achieve this is to always keep the variables names (the atoms) in a consistent order. The most obvious way to do this is alphabetically. Thus no matter what order the user enters the particle x^2yz, whether it be as x^2yz or as yx^2z or as x^2zy or any other permutation it will always be stored as x^2yz.

4.4 Polynomials

Continuing, we see that our type particle defines a term of a multivariate polynomial rather well. Thus we conclude our abstract types with the following:

```
TYPE Polynom
     Particle   particle_list[]
     integer    num_parts
END TYPE
```

Notice that we do not really want to place a maximum number of possible terms on our polynomial so we will leave the array size undefined to indicate that while we could define particle_list as an array we would prefer it to be a dynamic list. This is done so that in discussion we might refer to it as an array structure, though in actual implementation it would be better done as a list. Thus we have our abstract description of a multivariate polynomial, but what functionality should be applied to it?

5 Functionality of Multivariate Polynomials

There are many things we may want to do with our multivariate polynomials, but for now we will restrict ourselves to just the basics of input, output, addition, subtraction, multiplication, division and obtaining the results when some or all of the variables are assigned numeric values.

It should be understood that the above types would make excellent classes in an object oriented language and it would be easy to associate functions with each type. Thus we will list and describe some, but certainly not all, of the basic functions or methods associated with each type.

5.1 Functions for the Atom

For the Atom the following functions would be adequate:

- A constructor.
  - For the constructor it would be nice to be able to construct an atom from just a constant (e.g. 12) or from a coefficient and a name (e.g. 8*y*) or from a coefficient, name and exponent (e.g. $23x^3$).
- A method (or methods) to set the coefficient, exponent and name of the atom
  - These would be the trivially obvious methods.
- A method to test for equality between two atoms
  - This would compare the exponent, coefficient and name of the two atoms.
- A method to find the multiplicative inverse of the atom
  - This would take the numeric inverse of the coefficient and set the exponent to be the negative of its current value.
- A method to find the additive inverse of the atom
  - This would simply make the coefficient of the atom be the negative of its current value.
- A method to output the coefficient, exponent and name (the value) of the atom.
  - This would be an implementation dependent function, and would need to consider proper formatting.
- A method to evaluate the numeric value of an atom if its variable was assigned a particular numeric value.
  - This method would require the most work to make it efficient, considering the exponents on the variable could be quite large (and might be positive or negative). The type of number returned most likely would be the same type as that assigned to the coefficient (e.g. real).

5.2 Functions for the Particle

For the Particle we need a little more functionality:

- A constructor.
  - It would be nice to automatically initialize the particle to be zero, or to require an atom to be sent to the constructor.
- A method to set the particle to an initial atom.
  - Most likely this method will be sent one or more atoms.
- A way to put more variable names into the particle or remove variable names from the particle.
  - These tasks would most easily be done by the below multiply and divide functions, there would be little reason to duplicate them. Recall that the ORDER the atoms are placed in the particle must be consistent. This will likely require the comparison operators less than, <, and greater than, >, to be implemented in the Atom class.
- A method to multiply a particle by an atom.
- A method to multiply a particle by another particle.
- A method to divide a particle by an atom.
- A method to divide a particle by another particle.

- A method to calculate the additive inverse of the particle.
- A method to return the value of the particle if some or all of the variables of the particle are assigned values.
    - This method would at best be able to return a particle as an answer, since the user may wish to only assign some variables values. For example what is the value of $5x^2yz$ if $x = 2$? The answer is $20yz$ which obviously is non-numeric.
- A method to determine if two particles are equal.
    - This method might not mean exact equality. In practice it is more useful to have an equality test to see if all the variables (and their exponents) match. This allows for a simple test to see if the two particles can be added or subtracted from one another. Having that test completed, to determine if they were exactly equal would be just a matter of testing their coefficients for equality.

5.3 Functions for the Polynom

Of course the functions for Polynom are even larger in number:

- A constructor.
    - Which would be nice if it automatically initialized the polynom to zero, and might be made better if it could initialize it to an atom or particle sent.
- A method to set the polynom.
    - This method would initialize a polynom based on an atom or particle sent.
- A method to add (subtract) a particle.
    - This function would allow more 'terms' to be added to the polynom. While it is not obvious, it is important that the ORDER in which the particles are placed be consistent. A lexicographical ordering is probably best, and this will require the comparison operators < and > to be implemented in the Particle class.
    - This would require another function to determine if a given particle 'matched' any of the particles in the polynom (if the particle does not match then a new term of the polynom is created, otherwise the matching particles are added together). This 'matching' would be described in the equality test function of a particles.
- A method to add (subtract) a polynom.
    - This would add the particles of the sent polynom to the given polynom.
- A method to multiply by a particle.
    - By the distributive law each particle of the polynom would be multiplied by the particle sent.
- A method to multiply by a polynom.
    - Here it would be necessary to multiply each particle of the given polynom by each particle of the sent polynom.
    - It would also be necessary to correctly add any 'like' terms after the multiplication is performed – to keep the polynom as 'simple' as possible.
- A method to divide by a particle.

    - This would be a simple function which finds the multiplicative inverse of the sent particle and multiplies each particle of the given polynom by it.
    - Care would need to be taken for dividing by zero cases.
- A method to divide by a polynom.
    - This would likely be the most complicated of all the methods and may not, in the end, be possible to perform (e.g. $x^3 / (y^2 + 8z)$).
    - To get around this, it is possible to create a fraction class, where both the numerator and denominator are of type polynom. This would be the recommended easy solution (or this function may be skipped). Or perhaps the division would return two polynoms: a quotient and a remainder.
- A method to return the value of a polynom if some or all of the variables are assigned specific numeric values.
    - Notice this could at best return a polynom since not all variables are guaranteed to be assigned a numeric value.

6 Implementation of Multiplication

Before beginning actual implementation it would be wise to study the various ways to perform polynomial multiplication – brute force, Fourier transforms, Karatsuba, etc. Likewise the various division, factoring and reduction methods should be studied – Groebner, Resultant, GCD, etc.

For this paper we will discuss two possible multiplication methods: the brute force method and a Kroenecker Method. There will only be one division method mentioned and that is in section 7. It cannot be stressed enough that these operations will be the most often used, if their speed can be increased it would be a good idea to do so. The methods presented here are adequate for simple things, but will eventually fail to be "fast enough."

6.1 Brute Force Multiplication

This is the most obvious and straightforward way to multiply two polynomials together. In current mathematically teaching it is similar to the FOIL (First Outer Inner Last) method. It goes like this:
Given two univariate polynomials p and q such that

$$p = \sum_{i=0}^{n} a_i \cdot x^i \quad \text{and} \quad q = \sum_{i=0}^{m} b_i \cdot x^i$$

Then

$$p \cdot q = \sum_{k=0}^{m+n} \left(\sum_{k=i+j} a_i b_j \right) \cdot x^k$$

To implement the above the following code would suffice, assuming *p*, *q* and *c* are all Polynoms and we want $c = p \cdot q$.

```
c = 0
FOR k = 0 to m + n
     cur_coeff = 0
     FOR i = 0 to k
          a = p.GetCoeff('x', i)
          b = q.GetCoeff('x', k – i)
          cur_coeff = cur_coeff + (a * b)
     END FOR i
     an_atom.Set(cur_coeff, 'x', k)
     c.AddAtom(an_atom)
END FOR k
```

Where p.GetCoeff('x', i) returns the coefficient of x^i in the polynomial *p*, an_atom.Set(coeff, 'x', k) sets an_atom to have a coefficient of *coeff*, a variable name of *x* and a degree of *k* and p.AddAtom(an_atom) adds *an_atom* to the polynomial *p*.

The above only works for univariate polynomials. For multivariate polynomials it is usually easiest just to perform the term by term (particle by particle) multiplication and add the result into the polynomial, with the expectation that the add function will combine like terms. Specifically the code might look like:

```
c = 0
FOR i = 1 to p.num_particles
     p_part = p.GetParticle(i)
     FOR j = 1 to q.num_particles
          q_part = q.GetParticle(j)
          c = c + (p_part * q_part)
     END FOR j
END FOR i
```

While this looks very similar to the univariate code it is likely much slower. Notice that the multiplication is a multiplication of particle types rather than numeric coefficients, which is slow. Further speed reduction is because the addition is an addition of polynom types which is much slower than adding two numeric types. In the end though, the above code will successfully multiply two polynomials.

6.2 Krönecker Multiplication

Because the brute force method is slow, particularly for multivariate multiplication, there has been extensive research into speeding it up. Back in the 1970s a particularly clever trick was advocated [Moe 76]. This trick was to map a multivariate multiplication into a univariate multiplication and then apply a fast method of univariate multiplication (preferably faster than the brute force one, but it will still work). An example (from [Moe 76] of this is as follows:

6.2.1 Example Using the Krönecker Trick, Bivariate Case

Assume you have two polynomials

$$p = 2xy + x - y + 2$$
$$q = xy + 3x + 4y - 3$$

Rewriting things so x is the variable with symbolic coefficients we see that:

$$p = (2y + 1)x + (-y + 2)$$
$$q = (y + 3)x + (4y - 3)$$

Notice the highest degree of x in p is 1 and likewise the highest degree of x in q is also 1. Thus the highest degree of y that could occur in $p \cdot q$ is 1 + 1 = 2. We want to set $x = y^d$ such that d is greater than the highest degree of y that could occur in $p \cdot q$. So we shall say $d = 3$ and $x = y^3$. Substituting this in for x we obtain:

$$p = (2y + 1)y^3 + (-y + 2) \quad = 2y^4 + y^3 - y + 2$$
$$q = (y + 3)y^3 + (4y - 3) \quad = y^4 + 3y^3 + 4y - 3$$

From here we would apply a "fast" univariate multiplication method to arrive at:

$$p \cdot q = 2y^8 + 7y^7 + 3y^6 + 7y^5 - 3y^4 + 3y^3 - 4y^2 + 11y - 6$$

We then invert the substitution by dividing by y^3 and examining the remainder and repeating the division on the quotient, until the quotient becomes zero:

$$\frac{2y^8+7y^7+3y^6+7y^5-3y^4+3y^3-4y^2+11y-6}{y^3} = 2y^5 + 7y^4 + 3y^3 + 7y^2 - 3y + 3 + \frac{-4y^2+11y-6}{y^3}$$

Which means our coefficient for the x^0 is $-4y^2 + 11y - 6$ and we continue:

$$\frac{2y^5+7y^4+3y^3+7y^2-3y+3}{y^3} = 2y^2 + 7y + 3 + \frac{7y^2-3y+3}{y^3}$$

So our coefficient for the x^1 is $7y^2 - 3y + 3$ and we continue:

$$\frac{2y^2+7y+3}{y^3} = 0 + \frac{2y^2+7y+3}{y^3}$$

And our coefficient for x^2 is $2y^2 + 7y + 3$. Thus we conclude that

$$(2xy+x-y+2) \cdot (xy+3x+4y-3) = (2y^2 + 7y + 3)\,x^2 + (7y^2 - 3y + 3)x + (-4y^2 + 11y - 6)$$
$$= (2y^2 + 7y + 3)\,x^2 + (7y^2 - 3y + 3)x + (-4y^2 + 11y - 6)$$

As the above inversion process involves only division and multiplication by atoms it should behave faster than explicit multiplication of multivariate polynomials. The proof this technique always works can be found in [Moe 76].

6.2.2 Algorithm for the Krönecker Trick, Bivariate Case

Here is the algorithm to use the Krönecker trick for the bivariate case. The general algorithm will be presented in the next section. This simple case is presented to assist in the understanding of the more general case.

Let $f(x, y)$ and $g(x, y)$ be two polynomials.

1. Let $d_y(f)$ = the maximum degree of y in $f(x, y)$.
   Let $d_y(g)$ = the maximum degree of y in $g(x, y)$.
   Let $d = d_y(f) + d_y(g) + 1$.

2. Apply the trick: substitute y^d in for x into $f(x, y)$ and $g(x, y)$ to obtain:
   $\hat{f}(y) = f(y^d, y)$
   $\hat{g}(y) = g(y^d, y)$

3. Use a 'fast' univariate multiplication routine to find $\hat{f}(y) \cdot \hat{g}(y)$.

4. Invert the substitution to obtain the multivariate answer.

6.2.3 Algorithm for the Krönecker Trick, All Cases

Let f and g be two polynomials in n variables. Name the variables x_i for $i = 1$ to n.

1. Let $d_i(f)$ = the maximum degree of variable x_i found in f.
   Let $d_i(g)$ = the maximum degree of variable x_i found in g.
   Let $m[i] = d_i(f) + d_i(g) + 1$.
   So m[i] is a bound on the degree of x_i occurring in the result of the multiplication.

2. Apply the trick.
   Substitute $(x_{i-1})^{m[i-1]}$ in for x_i into f and g for $i = n$ down to 2.
   This results in two univariate polynomials $\hat{f}(x_1)$ and $\hat{g}(x_1)$.

3. Use a 'fast' univariate multiplication routine to find $\hat{f}(x_1) \cdot \hat{g}(x_1)$.

4. Invert the substitution to obtain the multivariate answer.

6.2.4 Example Using the Krönecker Trick, Trivariate Case

Assume you have two polynomials p and q. For illustration, we rewrite things so x is the variable with symbolic coefficients:

$$p = 5x^2y + 2xz^2 - x + 1 \quad = (5y)x^2 + (2z^2 - 1)x + 1$$
$$q = 3x + y - z + 7 \quad = (3)x + (y - z + 7)$$

Note the highest degree for y in p is 1 and in q is 1. So we substitute $y^{1+1+1} = y^3$ in for x into both polynomials to arrive at:

$$p' = (5y)(y^3)^2 + (2z^2 - 1)(y^3) + 1$$
$$q' = (3)(y^3) + (y - z + 7)$$

Simplifying and writing things so y is the variable with symbolic coefficients we see:

$$p' = (5)(y^7) + (2z^2 - 1)(y^3) + 1$$
$$q' = (3)(y^3) + (1)(y) + (-z + 7)$$

Looking back at p and q we see the highest power of z in p is 2 and the highest power of z in q is 1. So we substitute $z^{2+1+1} = z^4$ in for y into p' and q' to get:

$$\hat{p} = 5z^{28} + 2z^{14} - z^{12} + 1$$
$$\hat{q} = 3z^{12} + z^4 - z + 7$$

Using a "fast" univariate multiplication method we find:

$$\hat{p} \cdot \hat{q} = 15z^{40} + 5z^{32} - 5z^{29} + 35z^{28} + 6z^{26} - 3z^{24} + 2z^{18} - z^{16} - 2z^{15} + 14z^{14} + z^{13} - 4z^{12} + z^4 - z + 7$$

Notice that with $x = y^3$ and $y = z^4$, effectively $x = z^{12}$. So to recover the symbolic coefficients of x we must invert the substitution trick by dividing by z^{12} until the quotient becomes 0, where the remainder of the first division gives us the coefficient for x^0, the remainder of the second division gives us the coefficient for x^1, and so on. Thus:

$$\hat{p} \cdot \hat{q} = (15z^4)x^3 + (5z^8 - 5z^5 + 35z^4 + 6z^2 - 3)x^2 + (2z^6 - z^4 - 2z^3 + 14z^2 + z - 4)x + (z^4 - z + 7)$$

Finishing the inversion we must now divide by z^4 until the quotient becomes 0.

$$\hat{p} \cdot \hat{q} = (15)yx^3 + ((5)y^2 + (-5z + 35)y + (6z^2 - 3))x^2 + ((2z^2 - 1)y + (-2z^3 + 14z^2 + z - 4))x + ((1)y + (-z + 7))$$

Resulting in the correct answer of:

$$\hat{p} \cdot \hat{q} = 15x^3y + 5x^2y^2 - 5x^2yz + 35x^2y + 6x^2z^2 - 3x^2 + 2xyz^2 - xy - 2xz^3 + 14xz^2 + xz - 4x + y - z + 7$$

7 Implementation of Division

Division can be a tricky thing when dealing with univariate polynomials, and can become even more difficult when dealing with multivariate polynomials. One approach to solving this problem is to always factor every polynomial, thus division becomes a simple matter of canceling like terms. While this can be a good solution, if well implemented, it adds a great deal of complexity to the code. For now we will just consider the case that we are given two polynomials, *f* and *g*, and need to know the result of *f* divided by *g*. Specifically we want to find the polynomials *q* and *r* such that $f / g = q + r / g$. In simpler words, we want to find the quotient *q* and the remainder *r* when *f* is divided by *g*.

7.1 Lexicographical Ordering

The algorithm we are about to present assumes that both polynomials are lexicographically ordered. As a refresher we will briefly review what that means. Most of this review is based on material found in [Ajwa 2003].

From a coding perspective, maintaining the lexicographical ordering of a Polynom type would best be done by making it a feature of whatever function adds Particles (terms) to a Polynom type. Thus the Polynom would always be lexicographically sorted. This will make assignments slightly slower but will speed up the division process, which is likely to occur more frequently. On this same point it is assumed the variable names have some form of alphabetical ordering imposed on them within each Particle type. This was mentioned above and illustrated by the need that *xyz* be considered the same and stored the same as *zxy*.

Going back to the theoretical, we begin with a definition. Let N denote the (nonnegative) integers. Let $\vec{v}$ and $\vec{w}$ be vectors in N^n space. The **lexicographic ordering** is defined as $\vec{v} >_{lex} \vec{w}$ if and only if the leftmost nonzero entry in $\vec{v} - \vec{w}$ positive.

For example if $\vec{v}$ = (4, 3, 2) and $\vec{w}$ = (1, 3, 6) then $\vec{v} - \vec{w}$ = (3, 0, –4) and we conclude that $\vec{v} >_{lex} \vec{w}$ because the leftmost nonnegative entry is positive. Thus if we associate these numbers with the terms $x^4y^3z^2$ and xy^3z^6 we would say $x^4y^3z^2 >_{lex} xy^3z^6$. Notice this should also imply that the order of the variables names is consistent and comparable. In this particular case we assumed *x* comes before *y* which comes before *z* as far as the order of appearance.

Now applying this to a polynomial is best explained with an example. Assume

$$p(x, y, z) = y + zx + yx^2 + 2x - y^2x^3z - x^3z^2y^4$$

Then applying a lexicographic sorting we arrive at:

$$p(x, y, z) = -x^3y^4z^2 - x^3y^2z + x^2y + xz + 2x + y$$

7.2 Division Algorithm for Two Polynomials

The below algorithm is a simplification of the Generalized Division Algorithm as presented in [Cox 1991] and described in [Ajw 1995].

Assume f and g are both non-zero polynomials (univariate or multivariate).
Let p, q and r be polynomial data types.

1. Set $q = 0$, $r = 0$, $p = f$
2. Repeat
    a. If Lead Term of g divides p Then
        i. $u = p / g$
        ii. $q = q + u$
        iii. $p = p - (u \cdot g)$

    b. Else
        i. $r = r$ + Lead Term of p.
        ii. $p = p$ – Lead Term of p.

    Until $p = 0$

It should be noted that this algorithm assumes consistent ordering of terms of the polynomials. It is recommended they be in a lexicographical ordering before attempting this division.

8 Possible Advancements

Some obvious improvements in the abstraction of the multivariate polynomial should now be obvious. The first would be the addition of a new fractional type to deal with the cases where division of two polynomials fails to come out "even" (i.e. with no remainder). This type might be declared as follows:

```
TYPE Polyfrac
     Polynom   numerator
     Polynom   denominator
END TYPE
```

Another obvious improvement would be the ability to find the zeros of a given polynom (if any). This of course would be followed quickly by the ability to solve systems of polynoms, which might lead to a polynomial matrix class. For the sake of brevity, simplicity and to encourage exploration, these improvements will not be discussed here.

Also of importance would be input functions. These would greatly depend on what type of interface was desired. Most likely there would need to exist a function for the Polynom type that would be able to parse a string into a Polynom class structure. This might be done more easily if the Polynom class parsed the string into Particles and the Particle class parsed a string into Atoms and Atoms parsed the string.

9 Conclusion

So there it is, a very simple multivariate polynomial class. Hopefully, it should now be an easy task to begin implementing your own version. In the appendix of this document you will find a basic version of the above Atom class implemented in C++. It is not necessarily the most efficient, but it will do.

References

[Ajw 1995] Iyad A. Ajwa, Zhuojun Liu, and Paul S. Wang. *Gröbner Bases Algorithm*. ICM Technical Reports Series (ICM-199502-00), 1995. Available at.

[Ala 1987] Vangalur S. Alagar and David K. Probst. *A Fast, Low-Space Algorithm for Multiplying Dense Multivariate Polynomials*. ACM Transactions on Mathematical Software, Vol. 13, No. 1, March 1987, pp. 35-57.

[Buc 2001] Bruno Buchberger. *Groebner Bases: A Short Introduction for Systems Theorists*. Proceedings of EUROCAST 2001 (8^{th} International Conference on Computer Aided Systems Theory - Formal Methods and Tools for Computer Science), Feb. 19–23, 2001, Las Palmas de Gran Canaria, (R. Moreno-Diaz, B. Buchberger, J.L. Freire eds.), Lecture Notes in Computer Science 2178, Springer, Berlin–Heidelberg–New York, 2001, pp. 1–19.

[Cox 1991] D. Cox, J. Little, and D. O'Shea. Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra. Springer-Verlag, 1991.

[Fat 2003] Richard Fateman, *Comparing the speed of programs for sparse polynomial multiplication*. ACM SIGSAM Bulletin, Vol 37, No. 1, March 2003, pp. 4 – 14.

[Moe 1976] Robert T. Moenck, *Practical Fast Polynomial Multiplication*, Proceedings of the third ACM symposium on Symbolic and algebraic computation, ACM Press, Yorktown Heights, New York, USA, 1976, pp 136 – 148.

Appendix A – Atom.h

```
// ------------------------------------------------------------------------
// Atom.h
// ©opyright 2004 Brent M. Dingle
//
// Here is declared the ATOM - the most basic part of a polynomial
// ------------------------------------------------------------------------
#ifndef _CATOM_BMD

#include <cstdlib>
#include <iostream>
#include <cctype>
#include <string>
#include "frac.h"

using namespace std;

// ------------------------------------------------------------------------
// TYPEDEFS
// to use A_COEFF_TYPE = double you must do 2 things:
// typedef A_COEFF_TYPE to double
// define A_COEFF_IS_DOUBLE
// ------------------------------------------------------------------------
typedef double   A_COEFF_TYPE;   // coefficients (numeric) type
typedef long     A_EXP_TYPE;     // exponent (numeric) type


// ------------------------------------------------------------------------
// Some odd function declarations, so that we might convert
// strings into the proper data types.
//
// It is assumed exponents will be long and coeffs will be double
// but that may change.
// ------------------------------------------------------------------------
#define AsciiToExponent   atol
#define AsciiToCoeff      atof


// ------------------------------------------------------------------------
// CONSTANTS
// ------------------------------------------------------------------------
// It turns out bad things happen if we use a valid default var name
// when we just want to play with constants
// So instead we use @
#define DEF_VAR_NAME   '@'


// ------------------------------------------------------------------------
// Some global functions used by the CAtom class, but found
// to be useful enough NOT to embed in the class itself.
// ------------------------------------------------------------------------
long SkipSpacesAndStar(const char *str, long *index, long max_len);
bool GetExp(A_EXP_TYPE *exp, const char *str, long *index, long max_len);
bool GetCoeff(A_COEFF_TYPE *coeff, const char *str, long *index, long max_len);

void PostError(const char *type, const char *func_name, const char *message);
```

```
// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
class CAtom
{
   friend class CParticle;   // multiplies defined in class CParticle

public:
   // ------------------------------------------------- public functions
   CAtom();
   CAtom(const CAtom& rhs);          // copy constructor
   CAtom(double coefficient);
   CAtom(A_COEFF_TYPE coefficient, char name);
   CAtom(A_COEFF_TYPE coefficient, char name, A_EXP_TYPE exponent);
   ~CAtom();

   A_COEFF_TYPE Eval(A_COEFF_TYPE value);
   bool Set(A_COEFF_TYPE coefficient);
   bool Set(A_COEFF_TYPE coefficient, char name);
   bool Set(A_COEFF_TYPE coefficient, char name, A_EXP_TYPE exponent);

   long ParseAndSet(const char *str, long start_index = 0);
   bool Rename(char new_name);

   bool InvertMe();

   // ------------------------------------------------- some operators
   // Output should be a global function, make the operator a friend
   friend ostream& operator<< ( ostream &, const CAtom& an_atom);
   CAtom& operator= (const CAtom &rhs);
   CAtom& operator= (const char *rhs_str);

   // see Particle.cpp for definition of * operator
   // do not define here as it MUST return a CParticle,
   // but compiler required that CAtom declare the * operator as
   // being able to function on atom * atom
   friend CParticle operator * (const CAtom& lhs, const CAtom& rhs);


   // ------------------------------------------------- public variables
   A_COEFF_TYPE   m_coeff;
   A_EXP_TYPE     m_exp;
   char           m_name;

private:
   A_COEFF_TYPE EvalPower(A_COEFF_TYPE value);

};  // end CAtom class

#define _CATOM_BMD
#endif

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// end Atom.h
```

Appendix B – Atom.cpp

```
// ----------------------------------------------------------------------------
// Atom.cpp
// Public Version 2004.04.21
// ©opyright 2004 Brent M. Dingle
//
// Here is defined the ATOM - the most basic part of a polynomial
//
// Notice we CANNOT add ATOMS together nor multiply atoms of different
// variable names, for that we create another class called CParticle
// where such functions (multiplies anyway) are defined.
// Additions will require another class above CParticle called CPolynom
// ----------------------------------------------------------------------------
#include "atom.h"

// ----------------------------------------------------------------------------
// Constructor  - default initialized coeff = 0, exp = 0, name = @
// ----------------------------------------------------------------------------
CAtom::CAtom()
{
   m_coeff = 0;
   m_exp = 0;
   m_name = DEF_VAR_NAME;
}

// ----------------------------------------------------------------------------
// Constructor - if just a number is sent then exponent should
// default to 0 and name to x, thus the CAtom is just a number
// because x^0 = 1
// ----------------------------------------------------------------------------
CAtom::CAtom(A_COEFF_TYPE coefficient)
{
   m_coeff = coefficient;
   m_exp = 0;
   m_name = DEF_VAR_NAME;
}

// ----------------------------------------------------------------------------
// Constructor
// - if a coeff AND a name is sent then exponent should default to 1
// ----------------------------------------------------------------------------
CAtom::CAtom(A_COEFF_TYPE coefficient, char name)
{
   m_coeff = coefficient;
   m_exp = 1;
   m_name = name;
}

// ----------------------------------------------------------------------------
// Constructor - all 3 items specified, no defaults
// ----------------------------------------------------------------------------
CAtom::CAtom(A_COEFF_TYPE coefficient, char name, A_EXP_TYPE exponent)
{
   m_coeff = coefficient;
   m_exp = exponent;
   m_name = name;
}

// ----------------------------------------------------------------------------
// copy constructor
//
// This relies the assignment operator = being overridden
// ----------------------------------------------------------------------------
CAtom::CAtom(const CAtom &rhs)
{
   *this = rhs;
}
```

```
// ---------------------------------------------------------------------------
// Destructor
// ---------------------------------------------------------------------------
CAtom::~CAtom()
{
   // do nothing
}

// ---------------------------------------------------------------------------
// Set
//
// This function works pretty much exactly like the constructors.
// ---------------------------------------------------------------------------
bool CAtom::Set(A_COEFF_TYPE coefficient)
{
   m_coeff = coefficient;
   m_exp = 0;
   m_name = DEF_VAR_NAME;

   return true;
} // end Set - 1 param

// ---------------------------------------------------------------------------
bool CAtom::Set(A_COEFF_TYPE coefficient, char name)
{
   m_coeff = coefficient;
   m_exp = 1;
   m_name = name;

   return true;
} // end Set - 2 params

// ---------------------------------------------------------------------------
bool CAtom::Set(A_COEFF_TYPE coefficient, char name, A_EXP_TYPE exponent)
{
   m_coeff = coefficient;
   m_exp = exponent;
   m_name = name;

   return true;
} // end Set - 3 params

// ---------------------------------------------------------------------------
CAtom& CAtom::operator= (const CAtom &rhs)
{
   m_coeff = rhs.m_coeff;
   m_name = rhs.m_name;
   m_exp = rhs.m_exp;

   return *this;
}

// ---------------------------------------------------------------------------
CAtom& CAtom::operator= (const char *rhs_str)
{
   ParseAndSet(rhs_str);
   return *this;
}

// ---------------------------------------------------------------------------
// ParseAndSet
// Parse a string pulling out the coefficient, variable name
// and exponent.
//
// Coeff defaults to 1 if not found
// Variable name defaults to @ if not found
// Exponent defaults to 0 if it AND var name are not found
// Exponent defaults to 1 if it is not found but a var name is
//
// String MUST be in the format [coeff][var_name]^[(exponent)]
// exponent is in parentheses.
```

```
// String MUST be NULL terminated.
//
// Function returns the index of the first NOT used character
// This is likely to be whatever comes after the end paren
// of an exponent - often NULL terminator
//
// start_index defaults to zero;
// ---------------------------------------------------------------------------
long CAtom::ParseAndSet(const char *str, long start_index)
{
   long length, index;

   if (str == NULL) { return false; }

   length = (long)strlen(str);

   if (start_index >= length) { return false; }

   index = start_index;
   m_coeff = 1;
   GetCoeff(&m_coeff, str, &index, length); // only sets m_coeff if number
                                            // found starting at str[index]
                                            // index will point at first
                                            // NON-numeric character
   m_name = DEF_VAR_NAME;  // '@'
   m_exp = 0;

   SkipSpacesAndStar(str, &index, length);
   if (index < length)
   {
      if (isalpha(str[index]))
      {
         m_name = str[index];
         m_exp = 1;
         index++;
         if (index < length)
         {
            if (str[index] == ' ')   // have something maybe like x ^7, or x *
            {
               SkipSpacesAndStar(str, &index, length);
            }

            if (str[index] == '^')
            {
               GetExp(&m_exp, str, &index, length);
            }
            else if ((!isalpha(str[index])) && (str[index] != '*') &&
                     (!isspace(str[index])) && (str[index] != '+') &&
                     (str[index] != '-') && (str[index] != ')'))
            {
               // The ) is accepted to accomodate CPolyNom allowing
               // parens around the entire poly - this to allow
               // CPolyfrac to parse easier
               PostError("Error", "CAtom::ParseAndSet",
                         "Atom is not followed by another var_name nor a *, +, - or
space");

               index = length + 1;   // this should stop any further actions
            }
         }
      }
   } // end if index < length

   return index;
} // end ParseAndSet

// ---------------------------------------------------------------------------
// Rename
// Changes this atom's variable name to m_name.
// Note - the exp and coeff stay whatever they are.
// (zero and 1 unless they have been set)
```

```
// Returns false if current name is = DEF_VAR_NAME
// ----------------------------------------------------------------------------
bool CAtom::Rename(char new_name)
{
   bool ret_val;

   ret_val = true;
   if (m_name == DEF_VAR_NAME) {   ret_val = false;   }
   m_name = new_name;

   return ret_val;
} // end Rename

// ----------------------------------------------------------------------------
// Inverse
// Set this atom to its inverse.
// Effectively we make the exponent of the atom negative what
// it currently is and divide 1 by the coeff.
// Notice this implies that our coeff types can be inverted in this way.
// (so integers are out).
// ----------------------------------------------------------------------------
bool CAtom::InvertMe()
{
   m_coeff = 1 / m_coeff;   // may need to alter to m_coeff.Inverse()
   m_exp = -m_exp;

   return true;
} // end inverse

// ----------------------------------------------------------------------------
// Eval
// Evaluate the value of the atom if its variable has value sent.
// This assumes that A_EXP_TYPE was an integer of some kind
// ----------------------------------------------------------------------------
A_COEFF_TYPE CAtom::Eval(A_COEFF_TYPE value)
{
   A_COEFF_TYPE ret_val;

   if (m_coeff == 0)         // this atom is zero no matter what
   {
      ret_val = 0;
   }
   else if (value == 1)
   {
      ret_val = m_coeff;     // m_coeff * (1)^anything
   }
   else if (m_exp == 0)
   {
      //ret_val = value;     // x^0 = 1
      ret_val = m_coeff;     // m_coeff * (value)^0
   }
   else if (value == 0)      // zero to any power is zero, except 0^0 = 1
   {                         // so put this check AFTER check on m_exp == 0
      ret_val = 0;           // m_coeff * 0^(any non-zero)
   }
   else if (value == -1)
   {
      if (m_exp % 2 == 0)
      {
         ret_val = m_coeff;    // m_coeff * (-1)^(even power)
      }
      else
      {
         ret_val = -m_coeff;    // m_coeff * (-1)^(odd power)
      }
   }
   else if (m_exp == 1)
   {
      ret_val = m_coeff * value;
   }
   else if (m_exp == 2)
```

```
   {
      ret_val = m_coeff * value * value;
   }
   else if (m_exp == 3)
   {
      ret_val = m_coeff * value * value * value;
   }
   else
   {
      ret_val = m_coeff * EvalPower(value);
   }

   return ret_val;
} // end Eval

// ---------------------------------------------------------------------------
// EvalPower
// Return value raised to m_exp
// Notice we never do more than 1 multiply on a single line.
// This is in case CAtomNumber does not have support for multiple
// multiplies in a single line.
//
// This also assumes A_EXP_TYPE to be an integer type
// ---------------------------------------------------------------------------
A_COEFF_TYPE CAtom::EvalPower(A_COEFF_TYPE value)
{
   A_COEFF_TYPE square, cube, four;
   A_COEFF_TYPE five, eight, nine, ten;
   A_COEFF_TYPE twenty, thirty;
   A_COEFF_TYPE ret_val;
   bool had_neg_exp;
   int  i;

   ret_val = 0;

   had_neg_exp = false;
   if (m_exp < 0)
   {
      had_neg_exp = true;
      m_exp = -m_exp;
   }

   if (m_exp == 0)
   {
      ret_val = 1;
   }
   else if (m_exp == 1)
   {
      ret_val = value;
   }
   else if (m_exp == 2)
   {
      ret_val = value * value;
   }
   else if (m_exp == 3)
   {
      ret_val = value * value * value;
   }
   else
   {
      square = value * value;
      cube = square * value;
      four = square * square;
      switch (m_exp)
      {
         case 4: ret_val = four;
                 break;

         case 5: ret_val = four * value;
                 break;
```

```
case 6: ret_val = four * square;
        break;

case 7: ret_val = four * cube;
        break;

case 8: ret_val = four * four;
        break;

case 9: ret_val = cube * cube;
        ret_val = ret_val * cube;
        break;

case 10: ret_val = cube * cube;
         ret_val = ret_val * four;
         break;

case 11: ret_val = four * four;
         ret_val = ret_val * cube;
         break;

case 12: ret_val = four * four;
         ret_val = ret_val * four;
         break;

case 13: ret_val = cube * cube;
         ret_val = ret_val * cube;
         ret_val = ret_val * four;
         break;

case 14: ret_val = cube * cube;
         ret_val = ret_val * four;
         ret_val = ret_val * four;
         break;

default:
   five = four * value;
   eight = four * four;
   nine = eight * value;
   ten = nine * value;
   twenty = ten * ten;
   thirty = ten * twenty;
   switch (m_exp)
   {
      case 15: ret_val = five * ten;
               break;
      case 16: ret_val = five * value;
               ret_val = ret_val * ten;
               break;
      case 17: ret_val = cube * four;
               ret_val = ret_val * ten;
               break;
      case 18: ret_val = eight * ten;
               break;
      case 19: ret_val = nine * ten;
               break;
      case 20: ret_val = twenty;
               break;
      case 21: ret_val = twenty * value;
               break;
      case 22: ret_val = twenty * square;
               break;
      case 23: ret_val = twenty * cube;
               break;
      case 24: ret_val = twenty * four;
               break;
      case 25: ret_val = twenty * five;
               break;
      case 26: ret_val = five * value;
               ret_val = ret_val * twenty;
               break;
```

```
                case 27: ret_val = four * cube;
                         ret_val = ret_val * twenty;
                         break;
                case 28: ret_val = twenty * eight;
                         break;
                case 29: ret_val = twenty * nine;
                         break;
                case 30: ret_val = thirty;
                         break;
                case 31: ret_val = thirty * value;
                         break;
                case 32: ret_val = thirty * square;
                         break;
                case 33: ret_val = thirty * cube;
                         break;
                case 34: ret_val = thirty * four;
                         break;
                case 35: ret_val = thirty * five;
                         break;
                case 36: ret_val = five * value;
                         ret_val = ret_val * thirty;
                         break;
                case 37: ret_val = four * cube;
                         ret_val = ret_val * thirty;
                         break;
                case 38: ret_val = thirty * eight;
                         break;
                case 39: ret_val = thirty * nine;
                         break;
                case 40: ret_val = twenty * twenty;
                         break;
                default:
                   ret_val = twenty * twenty;
                   for (i=41; i <= m_exp; i++)   // was just < until 3-25-04
                   {
                      ret_val= ret_val * value;  // in case A_COEFF_TYPE doesn't like *=
                   }
             } // end switch for m_exp = 15 to anything
      } // end switch for m_exp = 4 to 14

   } // end if m_exp >= 4

   if (had_neg_exp)
   {
      m_exp = -m_exp;
      if (ret_val != 0)
      {
         ret_val = 1 / ret_val;
      }
      else
      {
         PostError("Error", "CAtom::EvalPower", "Divide by Zero");
      }
   }

   return ret_val;
} // end EvalPower

// ---------------------------------------------------------------------------
// Outputting stuff:
// Invoked with cout << an_atom
// which issues the call operator<< (cout, an_atom)
// ---------------------------------------------------------------------------
ostream& operator<< (ostream& output, const CAtom& an_atom)
{
   if (an_atom.m_coeff == 0)
   {
      output << "0";
   }
   else if (an_atom.m_exp == 0)
   {
```

```
      output << an_atom.m_coeff;
   }
   else if (an_atom.m_coeff == 1)
   {
      if (an_atom.m_exp != 1)
      {
         output << an_atom.m_name << "^(" << an_atom.m_exp << ")";
      }
      else
      {
         output << an_atom.m_name;
      }
   }
   else //if (an_atom.m_coeff != 1) and != 0
   {
      if (an_atom.m_exp != 1)
      {
         output << an_atom.m_coeff << "*" << an_atom.m_name << "^(" << an_atom.m_exp <<
")";
      }
      else
      {
         output << an_atom.m_coeff << "*" << an_atom.m_name;
      }
   }

   return output;
}

// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
//                        GLOBAL FUNCTIONS FOLLOW
// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// SkipSpacesAndStar
// Skip from index to max_len any spaces or asterix characters.
// Alter index so it is on the first NON-space and NON-star char.
//
// function returns the number of chars skipped
// ----------------------------------------------------------------------------
long SkipSpacesAndStar(const char *str, long *index, long max_len)
{
   long count;
   bool done;

   done = false;
   count = 0;
   while (!done)
   {
      if ( (isspace(str[*index])) || (str[*index] == '*') )
      {
         // done stays false
         *index = *index + 1;  // don't use ++ it does goofy things with pointers =)
         count++;
      }
      else
      {
         done = true;
      }

      if (*index >= max_len)
      {
         done = true;
      }
   }
   return count;
} // end SkipSpacesAndStar
```

```
// ----------------------------------------------------------------------------
// GetExp
// This assumes A_EXP_TYPE is an integer.
// If str is long enough and formed correctly exp is set.
//
// We should be at a point in the str where the first character
// is a caret = ^ followed by an open paren = ( followed by
// a number of type corresponding to A_EXP_TYPE then ending with a
// closed paren = ).
//
// The parentheses are required in case the exponent type is NOT
// just a numeric.
//
// This will still process an exponent without parentheses
// HOWEVER it will do so assuming the exponent to be an integer.
//
// If not, nothing is done to exp, however value of index
// may be altered and FALSE is returned.
// ----------------------------------------------------------------------------
bool GetExp(A_EXP_TYPE *exp, const char *str, long *index, long max_len)
{
   char num_str[200];
   long local_index;
   bool done;
   bool ret_val;

   ret_val = true;
   if (*index < max_len)
   {
      if (str[*index] == '^')
      {
         *index = *index + 1;     // again don't use ++, with ptrs odd things happen =)

         if (str[*index] == ' ')  // have something like x^ 12
         {
            SkipSpacesAndStar(str, index, max_len);
         }

         if (str[*index] == '(')
         {
            local_index = 0;
            done = false;
            // the -1 on max_len should be okay as last char on exp should be closed
paren
            while ( (!done) && (local_index < 200) && (*index < max_len-1))
            {
               *index = *index + 1;
               if ( (isdigit(str[*index])) ||
                    (str[*index] == '-') || (str[*index] == '.') )
               {
                  num_str[local_index] = str[*index];
               }
               else
               {
                  done = true;
               }
               local_index++;
            }
            num_str[local_index] = '\0';
            // *exp = atol(num_str);
            *exp = AsciiToExponent(num_str);  // AsciiToExponent is defined in atom.h

            // And for nicety if we found an exponent we should
            // step index past that end paren
            if (str[*index] == ')')
            {
               *index = *index + 1;
            }

         }
         else // character after ^ is NOT a parentheses --> str[*index] != '('
```

```
         {
            // ASSUME exponent is a number and will terminate at first NON-numeric
            local_index = 0;
            done = false;
            if (str[*index] == '-') // only allow first char after ^ to be -,  e.g. x^-1
            {
               num_str[local_index] = str[*index];
               local_index++;
               *index = *index + 1;
            }

            while ( (!done) && (local_index < 200) && (*index < max_len))
            {
               if ( (isdigit(str[*index])) || (str[*index] == '.') )
               {
                  num_str[local_index] = str[*index];
                  local_index++;
                  *index = *index + 1;
               }
               else
               {
                  done = true;
               }
            }

            num_str[local_index] = '\0';
            // *exp = atol(num_str);
            *exp = AsciiToExponent(num_str);  // AsciiToExponent is defined in atom.h
         }
      } // if (str[*index] == '^')
      else
      {
         PostError("Error", "GetExp()", "Caret = ^ not found");
         ret_val = false;
      }
   }
   return ret_val;
} // end GetExp

// ----------------------------------------------------------------------------
// GetCoeff
// It is assumed that index points to a spot in the string where a
// number (of type corresponding to coeff type) begins.
//
// If no number is found coeff is set to be 1, else
// we parse the string from index until nonnumeric characters are
// encountered and then attempt to convert that string into a
// number using the defined function AsciiToCoeff
//
// When complete index should point at the first NON-numeric character
// ----------------------------------------------------------------------------
bool GetCoeff(A_COEFF_TYPE *coeff, const char *str, long *index, long max_len)
{
   char num_str[200];
   long local_index;
   bool done;

   *coeff = 1;  // default to 1

   if ( (str[*index] != '-') && (!isdigit(str[*index])) )
   {
      return true;
   }
   else
   {
      num_str[0] = str[*index];
      *index = *index + 1;        // don't use ++, odd things happen with ptrs
   }

   // Here we deal with the case of spaces between the sign
   // of the coeff and the numbers e.g. "- 34*b*d*e"
```

```
   if ((num_str[0] == '-') || (num_str[0] == '+'))
   {
      SkipSpacesAndStar(str, index, max_len);
   }

   // Now we need to check that we actually have numbers and
   // we are NOT dealing with something like "-x*y*z"
   if ((num_str[0] == '-') && (isalpha(str[*index])) )
   {
      *coeff = -1;
      return true;
   }

   // So we should now be on a number or a character (variable name)
   local_index = 1;
   done = false;
   while ( (*index < max_len) && (!done) )
   {
      if ( (!isdigit(str[*index])) && (str[*index] != '.'))
      {
         done = true;
      }
      else
      {
         num_str[local_index] = str[*index];
         *index = *index + 1;
         local_index++;
      }
   }

   num_str[local_index] = '\0';

   *coeff = AsciiToCoeff(num_str);  // most likely *coeff = atof(num_str)

   return true;
} // end GetCoeff

// ---------------------------------------------------------------------------
// PostError
// ---------------------------------------------------------------------------
void PostError(const char *type, const char *func_name, const char *message)
{
   cout << type << " - " << func_name << endl;
   cout << "     " << message << endl;
   cout << endl;
}

// ---------------------------------------------------------------------------
// ---------------------------------------------------------------------------
// end Atom.cpp
```