# Symbolic Computations in C++

**Notes taken by Simon Badger, s.d.badger@durham.ac.uk**

Abstract: These notes are a summary of the lectures of Stephan Weinzierl given at the CAPP05 workshop in DESY-Zeuthen, 03/04/2005 - 08/04/2005. These notes closely follow the article by Weinzierl 'Computer Algebra in Particle Physics" [1].

# Contents



## 1. Introduction

As the complexity of calculations necessary for precision tests of the Standard Model (and its extensions) has increased, the need for Computer Algebra Systems (CAS) is becoming more apparent [1]. These notes describe the basic ideas behind a generic computer algebra system and discusses some of the principles behind implementing efficient algorithms.

There have been many Commercial and Open Source programs available for calculations over the years. Beginning with LISP based systems like REDUCE and MATHLAB programs migrated to C as they became more commercial with the introduction of MATHEMATICA and MAPLE. There are a number of specific languages which were developed for speed in a special areas. The most applicable in this category is FORM which has the ability to handle large amounts of data where MATHEMATICA and MAPLE would struggle. Another program which is written in C++ is GiNaC. It also has the ability to handle large expressions but also allows the user to benefit from the object-oriented features of C++ and is most suitable if the user wishes to implement their own algorithms.

## 2. Data Structures

There are four many types of container which are useful for storage of objects in the computer memory:

- **Vectors**: Also known as an *Array* this container stores information linearly in the computer memory. This means that accessing an individual element of the vector is easy:

$$a_i = a_1 + (i-1)l. \tag{2.1}$$

  This operation is cheap, $\mathcal{O}(1)$ in access speed. However adding a new element to the vector requires a shift of all the elements below the new element and goes as $\mathcal{O}(n)$.

- **Lists**: For a list one stores the information in fields containing 3 quantities, a piece of information, a pointer the next field and a pointer to the previous field. This format is very well suited for adding new elements as updating the pointers is only an $\mathcal{O}(1)$ operation. Accessing an individual element requires following the pointers sequentially from the 1st element, this is an $\mathcal{O}(n)$ operation.

- **Binary Trees**: Here elements contain two values, one value, the key, allows you to access the other. There is some 'less than' operation which orders the elements. To find a mapped value using the key takes on average $\mathcal{O}(log(n))$ while adding new elements is also $\mathcal{O}(log(n))$.

- **Hash Maps**: These also take two values in each element. Given an easy to calculate function, called the Hash function, one can each key to a specific memory address. In general the Hash function cannot be guaranteed to uniquely specify each memory address but one can find ways to implement the function correctly. The hash map is generally useful in situations where speed for both access and insertions is high and sufficient memory space is available.

## 3. Example of a Computer Algebra System in C++

We need two main ingredients to put together a basic computer algebra system. The first is a set of *atoms*, these are symbols,integers and other operands. The second is a set of containers which form the basic operations like addition and multiplication. To implement this in C++ we can use the features of modular programming and inheritance to form classes of variables which we will use in the calculations. The following example has a set of symbols and integers. Figure 1 shows the hierarchy of classes. The main structure lies in having a **basic** class from which both **symbol** and **numeric** are derived.

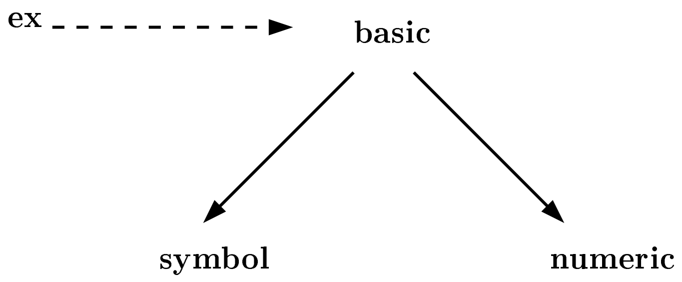


**Figure 1:** The inheritance tree for the example computer algebra system in C++. The classes **symbol** and **numeric** inherit the properties of the basic class. The **ex** class acts as a pointer to the basic class and is used for memory management.

```
class symbol : public basic
{
protected:
std::string name;
};

class numeric : public basic
{
protected:
int value;
};
```

At this stage we can also implement operations of multiplication and addition on the two atoms. The basic class could be just a common set of properties of the two sub-classes but here we consider issues of memory management and efficiency as well. To this end we introduce a new class **ex** which acts a pointer to the basic class:

```
class ex
{
public:
basic *bp;
};

class basic
{
private:
unsigned refcount;
};
```

The **basic** class has a reference counter which counts how many times that object has been pointed to by the class **ex**. This allows the object to be deleted from the memory as soon as it is no longer needed.

# 4. Efficiency

In order to create a useful computer algebra system efficient algorithms are essential. In this section we discuss two examples, multiplication and Eulcid's GCD algorithm, to demonstrate how one can improve efficiency.

## 4.1 Multiplication

We can write an arbitrary integer, $a$, with $n$ digits in a base, $B$, as

$$a = a_0 + a_1 B + \ldots + a_{n-1} B^{n-1}. \tag{4.1}$$

If we assume that $n$ is even[1] then we can write,

$$a = a_h B^{n/2} + a_l \tag{4.2}$$

where $a_h$ and $a_l$ both have $n/2$ digits. Thus the product $ab$ can be calculated by

$$ab = a_h b_h B^n + (a_h b_l + b_h a_l) B^{n/2} + a_l b_l \tag{4.3}$$

This classical method an requires 4 multiplications of integers with $n/2$ digits and 4 additions. This algorithm grows as $\mathcal{O}(n^2)$.

For large integers the efficiency can be greatly improved by writing (4.3) as

$$ab = a_h b_h B^n + \left(a_h b_h + b_l a_l - (a_h - a_l)(b_h - b_l)\right) B^{n/2} + a_l b_l. \tag{4.4}$$

This method was invented by Karatsuba and grows as $\mathcal{O}(n^{1.58})$ where $n$ is the number of digits. This is because we now have only 3 multiplications of $n/2$ digit integers.

There is an even faster method due to Schönhage and Strassen which utilises the fast Fourier transform.

Consider a polynomial of degree $n$

$$a(x) = a_0 + a_1 x + \cdots + a_n x^m. \tag{4.5}$$

It is specified uniquely by $n+1$ points $(x_0, \ldots, x_n)$. This set is called the *modular representation*. Suppose that we would like to multiply a degree $n$ polynomial, $a(x)$, with a degree $m$, $b(x)$, polynomial to get a degree $n+m$ polynomial. Suppose we know the polynomials a(x) and b(x) at $n+m$ distinct points then we can write the product as:

$$ab = (a(x_0)b(x_0), \ldots, a(x_{m+n})b(x_{n+m})) \tag{4.6}$$

[1] If $n$ is odd then we can add zero to make the number of digits even

This uniquely defines the polynomial $ab$ with only $\mathcal{O}(n+m)$ operations. However this does not include the cost of putting the original polynomials in the modular representation and converting $ab$ back to the standard representation.

Converting to the modular representation can be done very efficiently for a clever choice of points. Let us choose these points to be $\Omega = (1, \omega, \omega^2, \ldots, \omega^{n-1})$ where $\omega$ is the primitive $nth$ root of unity:

$$\omega^n = 1 \tag{4.7}$$

We can choose $n$ to be even just as we did before, and so the roots satisfy the symmetry relation

$$\omega^{i+n/2} = -\omega^i \tag{4.8}$$

since they lie on a circle of radius 1. Evaluating the polynomial at these points is nothing other than a discrete Fourier transform and by writing the polynomials as

$$a(x) = b(x^2) + xc(x^2) \tag{4.9}$$

It is sufficient to evaluate the polynomials $b$ and $c$ at the independent $n/2$ points of $\Omega$.

Using this method recursively results in a method that grows as $\mathcal{O}(nlogn)$.

## 4.2 Euclid's Algorithm for Greatest Common Divisor(gcd)

Consider simplifying the expression:

$$\frac{(x+1)(x-1)^2}{(x+1)(x^2-1)} = x - 1. \tag{4.10}$$

This is extremely easy, remembering that $x^2 - 1 = (x+1)(x-1)$. However this is not an efficient way to simplify the expression using a computer algebra program as it requires factorising of both the numerator and the denominator which is very expensive. A much better approach is to find the gcd of the numerator and the denominator which can be done more efficiently. The algorithm for computing the gcd given here is due to Euclid and forms an essential part of many algorithms important for symbolic programming.

In a Euclidean domain, $D$, (see [1] for more detailed explanation) we are able to define elements $q$ and $r$ given two elements $a$ and $b$ s.t.

$$a = bq + r. \tag{4.11}$$

It is then possible to show that

$$gcd(a, b) = gcd(b, r) \tag{4.12}$$

This can be equation can be applied recursively until $gcd(a, b) = gcd(c, 0)$ and so $c = gcd(a, b)$. This method does not quite apply to polynomials as they are only uniquely factorisable domains not Euclidean ones. However with a slight modification the algorithm can be extended to this regime. Given two polynomials $a(x)$ and $b(x)$ of order $n$ and $m$ respectively, $n \geq m$, we can write:

$$b_m^{n-m+1} a(x) = b(x)q(x) + r(x), \tag{4.13}$$

where $deg(r(x)) < deg(b(x))$. The Euclidean algorithm then applies as before.

$$gcd(a(x), b(x)) = gcd(b(x), r(x)) \tag{4.14}$$

The problem with this Euclidean algorithm is that it can lead to large intermediate expressions. Therefore it is necessary to find an improvement to the algorithm. More details on this can be found in [1].

# 5. Classical Algorithms

In this section we discuss a few a important algorithms that have been developed specifically for symbolic computations.

## 5.1 Factorisation

Factorisation of a polynomial is an extremely common task in computer algebra. Here we consider factorisation of a polynomial $u(x) \in \mathbb{Z}[x]$ over the integers $\mathbb{Z}$:

$$u(x) = u_n x^n + \ldots + u_1 x + u_0. \tag{5.1}$$

Here we outline a method which will be presented in three parts. Firstly performs a square free decomposition, secondly one uses Berlekamp's algorithm to factor the polynomial in the ring $\mathbb{Z}_p$ (where $p$ is prime), finally uses a procedure known as Hensel lifting to yield factorisation over $\mathbb{Z}$.

### Square Free Decomposition

Suppose we can find $v(x)$ s.t.

$$u(x) = (v(x))^m r(x) \tag{5.2}$$

By taking the derivative and calculating $gcd(u(x), u'(x))$ we can reduce the exponent $m$ to 1:

$$u'(x) = m(v(x))^{m-1} v'(x) r(x) + (v(x))^m r'(x). \tag{5.3}$$

Now $g(x) = gcd(u(x), u'(x))$ so that we can write:

$$u(x) = \left( \frac{u(x)}{g(x)} \right) g(x) \tag{5.4}$$

where $g(x)|u(x)$ by the definition of the gcd. Repeating this process results in a decomposition in which all factors appear with a single power.

### Berlekamp's Algorithm

We now have a representation of $u(x)$ which is square free:

$$u(x) = p_1(x)p_2(x)\ldots p_k(x). \tag{5.5}$$

The degree of $U(x)$ is $n$. We will find a factorisation of $u(x)$ in $\mathbb{Z}_p$ where $p$ is a prime number. For $0 < k < n$ one can define $q_{k,j}$ from the $modu(x)$ representation of $x^{kp}$:

$$x^{kp} = (q_{k,n-1}x^{n-1} + \ldots + q_{k,1}x + qk, 0) \bmod u(x) \tag{5.6}$$

We can therefore define the $n \times n$ matrix $Q = [q_{i,j}]$

$$\begin{pmatrix} q_{0,0} & q_{0,1} & \cdots & q_{0,n-1} \\ \vdots & & & \vdots \\ q_{n-1,0} & q_{n-1,1} & \cdots & q_{n-1,n-1} \end{pmatrix} \tag{5.7}$$

Solving for the eigenvectors of $Q$ then defines a non-trivial polynomial $v_1(x)$. Then we need to calculate $gcd(u(x), v_1(x) - s)$ for $0 < s < p$ s.t.

$$u(x) = v_1(x)w_1(x) \bmod p \tag{5.8}$$

### Hensel Lifting

This is a procedure to promote the factorisation of $u(x)$ from $\mathbb{Z}_p$ to $\mathbb{Z}$:

$$u(x) = v_r(x)w_r(x) \bmod p^r. \tag{5.9}$$

After applying Berlekamp's algorithm to a square free decomposed polynomial we know that $gcd(v_1, w_1) = 1(mod)p$. We can also compute the $gcd$ using the Euclidean algorithm to find:

$$a(x)v_1(x) + b(x)w_1(x) = 1 \bmod p. \tag{5.10}$$

Suppose that we now have $(w_r, v_r)$ then from $a(x)$ and $b(x)$ it is then possible [1] to construct $(w_{r+1}, v_{r+1})$:

$$v_{r+1}(x) = v_r(x) - p^r b_r(x) \bmod p^{r+1}, \tag{5.11}$$

$$w_{r+1}(x) = w_r(x) - p^r a_r(x) \bmod p^{r+1}, \tag{5.12}$$

where $a_r$ and $b_r$ are computed from $a(x)$, $b(x)$ and $u(x)$.

*Example*

Consider $u(x) = x^2 + 27x + 176$. This polynomial is already square free so we can try to factorise it in $\mathbb{Z}_3$ using Berlekamp's method:

$$\begin{aligned} u(x) &= x^2 + 2 \bmod 3 \\ &= (x+1)(x+2) \bmod 3 \end{aligned} \tag{5.13}$$

Then applying Hensel lifting yields:

$$\begin{aligned} u(x) &= (x+7)(x+2) \bmod 9 \\ &= (x+16)(x+11) \bmod 27 \end{aligned} \tag{5.14}$$

which now agrees with the factorisation in $\mathbb{Z}$.

### 5.2 Symbolic Integration

The development of serious computer algebra packages has prompted the formulation of systematic procedures for integrating certain classes of functions. Here we concentrate on the integration of rational functions. Consider a rational function $f(x)$:

$$f(x) = \frac{a(x)}{b(x)} = \frac{a_m x^m + \ldots + a_1 x + a_0}{b_m x^m + \ldots + b_1 x + b_0}. \tag{5.15}$$

We can then factorise the denominator over $\mathbb{C}$ using the method described in the previous section:

$$b(x) = b_n (x - c_1)^{m_1} \ldots (x - c_r)^{m_r}. \tag{5.16}$$

Using this form and using partial fractions allows to write

$$f(x) = p(x) + \sum_{i-1}^{r} \sum_{j=1}^{m_i} \frac{d_{ij}}{(x - c_i)^j}, \tag{5.17}$$

where $p(x)$ is a polynomial and $d_{ij}$ are complex numbers. This is expression is straightforward to integrate using

$$\int \frac{dx}{(x - c_i)^j} = \begin{cases} \ln(x - c_i), & j = 1, \\ \frac{1}{(j-1)(x-c_i)^{j-1}}, & j > 1. \end{cases} \tag{5.18}$$

The major problem with this method is that completely factorising the denominator can lead to unnecessary algebraic extensions. For example consider

$$f(x) = \frac{1 - x^2}{(1 + x^2)^2}. \tag{5.19}$$

Applying the above method leads to

$$f(x) = -\frac{1}{2}\frac{1}{(x+i)^2} - \frac{1}{2}\frac{1}{(x-i)^2} \tag{5.20}$$

thereby introducing a complex number, $i$, which cancels when the expression in integrated. It is therefore preferable to find a more efficient method. Such a procedure, due to Rothstein and Trager, first the square free decomposition to the denominator before computing the logarithmic part of the integral. Using Hermite's reduction method it is possible to write:

$$f(x) = p(x) + \frac{a(x)}{b(x)}, \tag{5.21}$$

where $deg(a(x)) < deg(b(x))$ and $gcd(a(x), b(x)) = 1$. Then consider the square free decomposition of $b(x)$:

$$b(x) = u(x)(w(x))^m. \tag{5.22}$$

We can then use the Eucildean algorithm to find $r(x)$ and $s(x)$ s.t.

$$r(x)u(x)v'(x) + s(x)v(x) = \frac{1}{1-m}a(x). \tag{5.23}$$

This results in the integral reducing to one with a smaller power of $v(x)$,

$$\int dx \frac{a(x)}{b(x)} = \frac{r(x)}{(v(x))^{m-1}} + \int dx \frac{(1-m)s(x) - u(x)r'(x)}{u(x)(v(x))^{m-1}}. \tag{5.24}$$

Keeping this procedure running leads to:

$$\int dx \frac{a(x)}{b(x)} = \sum_{i=1}^{n} r_i ln(x - c_i), \tag{5.25}$$

where $r_i$ are the residues of f(x) at $c_i$ - the zeros of $b(x)$.

There are extensions to this algorithm due to Lazard and Rioboo and a more complex algorithm which applies to any elementary function due to Risch.

### 5.3 Symbolic Summation

Symbolic summation is the discrete analogue of integration described in the previous section. Here we consider an indefinite sum,

$$T(k+1) - T(k) = t(k) \tag{5.26}$$

where the definite sum is therefore

$$\sum_{k=a}^{b-1} t(k) = T(b) - T(a). \tag{5.27}$$

An algorithm to calculate this sum looks for a class a functions in terms of which the indefinte sum can be expressed. Using Gosper's algorithm this can be done for the sums of 'hypergemoetric' terms for which the ratio of two consecutive terms is a rational function.

### 5.4 Gröbner Bases

Here we consider simplifications with the help of a special bases known as Gröbner bases. Consider a complicated expression $f(x)$ which is a polynomial of several variables $x_1, \ldots, x_k$. In addition assume that we have $r$ siderelations of the form,

$$s_j(x_1, \ldots, x_k) = 0, \qquad 1 \leq j \leq r. \tag{5.28}$$

The task is then to re-write $f(x)$ in the form,

$$f(x) = g(x) + \sum_{j=1}^{r} a_j s_j, \tag{5.29}$$

where $g(x)$ is simpler than f(x). That is $g$ is simpler than $f$ with respect to some order relations e.g. $y < x$ or $x < x^2$ where '$<$' is taken to mean 'less complicated'.

For multivariate polynomials in the ring $R[x_1, \ldots, x_k]$ we can write each element as a sum of monomials: $cx_1^{m_1} \ldots x_k^{m_k}$. We define the order elation to be lexicographic ordering

$$c' x_1^{m'_1} \ldots x_k^{m'_k} < c x_1^{m_1} \ldots x_k^{m_k}, \tag{5.30}$$

where the left-most, non-zero entry in $(m_1 - m'_1, \ldots, m_k - m'_k)$ is positive. We can now write the expression as a sum of monomials,

$$f(\{x_l\}) = \sum_{i=0}^{n} h_i \tag{5.31}$$

where $h_n$ is the most complicated monomial and is called the leading term, $lt(f) = h_n$. Now we use the siderelations as a basis for an ideal, $I$. The Gröbner basis is a basis for $I$ s.t. for a subset $G$ of $I$ s.t. $G$ is a basis for $I$ and $lt(G)$ is a basis for $lt(I)$. $G$ forms a natural basis for simplifying the expression, $f$, with respect to the order relation. There is an algorithm due to Buchberger which allows us to compute $G$.

## References

[1] S. Weinzierl, *Computer algebra in particle physics*, `hep-ph/0209234`.