# MXYZPTLK: AN EFFICIENT, NATIVE C++ DIFFERENTIATION ENGINE *

J.-F. Ostiguy and L. Michelotti
Fermi National Accelerator Laboratory, Batavia, IL 60510, USA

*Abstract*

Mxyzptlk was among the earliest implementations of a differentiation engine reported in the literature [1, 2]. It was created with an eye to enabling accelerator related computations, especially within the realm of perturbation theories. Such computations are supported by (1) a one-to-one correspondence between original mathematical abstractions and the data types and operations used to implement them and (2) accurate computation of high order derivatives. To this day, mxyzptlk distinguishes itself by being among the few available implementations that takes full advantage of the native operator overloading capabilities of their implementation language.

Recently, significant efforts were expanded in modernizing mxyzptlk, both architecturally and algorithmically, resulting in substantially improved performance, maintainability and usability. We present an overview of the mxyzptlk internals and summarize current capabilities and performance.

## INTRODUCTION

In the early 1990's, the development of a suite of libraries dedicated to accelerator simulation with an eye on non-linear dynamics was initiated. The libraries would take advantage of Automatic Differentiation (AD), a then emerging technique. High order derivatives are the backbone of perturbation analysis in nonlinear dynamics; since AD computes such derivatives to machine precision (something that standard finite difference techniques simply cannot deliver) the technique is a natural and compelling fit.

C++ which had just arrived on the scene, was selected as the implementation language because by design (1) it allows user-defined types to have essentially the same status as native types and (2) it provides comprehensive support for operator overloading. An additional practical consideration was that as, a superset of C, C++ was well-positioned for commercial success, long term viability and availability of development tools. This certainly turned out to be a correct assumption.

This paper focuses on mxyzptlk, the automatic differentiation engine which provides the foundation for an accelerator computation framework. The status of this framework is the object of a companion report. It should be emphasized that mxyzptlk contains no specific dependencies on accelerator concepts and can be used independently in applications where accurate computation of derivatives is relevant.

By the end of the 1990s mxyzptlk had reached a plateau in stability and maturity. Until then, development emphasis had been put on correctness and was becoming painfully obvious that performance was sub-optimal. A major overhaul was started in mid-2003. Internal data structures were redesigned, architectural and algorithmic changes were introduced and, in the process, maximum advantage was taken of idioms and features (e.g. templates) of C++ not available in the early 1990s. While it would be instructive, it is outside the scope of this report to discuss the implementation details made obsolete by recent efforts.

## DATA TYPES

Mxyzptlk defines a number of data types; the principal ones are: Jet, Jet__environment, JLterm, JetVector, Mapping and LieOperator.

Jet is the fundamental type.[1] It generalizes the notion of a variable (double or complex) by carrying supplementary information about derivatives up to some specified order. Every Jet exists in the context of a specific Jet__environment (described below); furthermore, Jet algebra can involve only Jets with compatible environments. A Jet__environment is an attribute of a Jet which encapsulates properties of the work environment e.g. dimensionality of the variable space, maximum derivative order and the expansion reference point. It also acts as a central point for term (monomial) indexing facilities.

The information about numerical derivatives held within a Jet corresponds (within a constant) to the monomial coefficients of a Taylor polynomial expansion. Each monomial term is completely characterized by a numerical coefficient (double or complex) and a tuple of integers representing the monomial exponents, or equivalently, the order of the partial derivatives associated to the coefficient. The type JLterm encapsulates a monomial term. Each JLterm holds a monomial coefficient, the (integer) offset of the exponent tuple in an pre-ordered table and an integer which represents the "weight" or "total order" (the sum the individual monomial exponents).

Through operator overloading, mxzypltlk provides comprehensive support for mathematical operations on Jets including the usual transcendental functions (sine, cosine, logarithm, exponential etc.). In addition, Jets supports functional composition, an essential ingredient for accelerator related computations involving concatenation of nonlinear element maps.

* Work supported by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the United States Department of Energy.

[1] The word Jet comes from the mathematical literature [3].

A `JetVector` is simply what its name suggests: a vector of `Jet` quantities. Note that each component of a `JetVector` can be interpreted a coordinate transformation and therefore a `JetVector` can be viewed as a mapping. In addition to `JetVector` `mxyzptlk` defines the derived types `Mapping` and `LieOperator` which are used to represent mappings and operations arising in the context of Hamiltonian perturbation theory.

`mxyzptlk` types are templated on the type of the monomial coefficient, but not on maximum order since this would complicate the memory allocation recycling strategy as well as the use of multiple orders within in a single program. Templates eliminate a fair amount of code duplication since one implementation can be used for either the double and complex variant of a function. Judicious usage of implicit type conversions also contributes to reducing code duplication and code complexity by allowing both `double` and `complex` `Jets` to inter-operate. Finally, `mxyzptlk` is layered on a set of support classes that provide, among other things, support for linear algebra. These classes can be instantiated for `Jets`: one can, for example, factor a matrix of `Jets`.

## IMPLEMENTATION

As already mentioned, a `Jet` is completely described by its environment and by its monomials with non-zero coefficients. A complete monomial basis for a polynomial of order $N$ in $D$ variables is comprised of

$$M = \frac{(N+D)!}{N!D!} \tag{1}$$

monomials. Needless to say, because of this combinatoric scaling, $M$ rapidly becomes very large. For example, with $D = 6$ and $N = 10$, $M = 8008$. Any usable implementation of automatic differentiation must pay close attention to memory management.

In `mxyzptlk`, the actual `Jet` representation in memory is both sparse and ordered. By sparse, we mean that as much a possible, monomonials with null coefficients are not stored, although it is entirely permissible to store null coefficients. Monomials, that is `JLterm` objects, are always stored sequentially in memory and ordered by increasing *weight* ( total monomial order) and at equal weight, in inverse lexicographical order of exponent tuples.

### *Mathematical Operations*

Ultimately, all mathematical operations on `Jets` are reduced to sequences of additions and multiplications. Therefore, the implementation of these two basic operations plays a dominant role.

**Addition** Although some complexity is introduced by the sparse storage scheme ( one needs to account for the fact that a term in one operand may not have a corresponding term) addition is a straightforward matter: terms with identical indices are simply added together. Note that this procedure can efficiently be performed if the terms in both operands are *ordered*. Obviously, addition is order preserving.

**Multiplication** Multiplying Jets is significantly more involved than addition. `Jet` algebra is essentially isomorphic to polynomial algebra, and multiplying two `Jets` with a maximum of $M$ terms leads to a possible maximum of $M^2$ monomial multiplications. Many of these operations are vacuous (and altogether avoided) since they produce monomials of weight larger than the maximum weight allowed for the computations. As mentioned earlier, each term (monomial) exponent tuple is only implicitly stored through an integer representing an offset within an ordered table. Monomial multiplication is accomplished by table lookup. A multiplication table maps the offset of the two monomial operands to the offset of the product monomial. Note that a naively implemented lookup table becomes rapidly unwieldy with increasing order. The actual implementation uses double indirection, avoids duplicate entries as well as entries where which exceed the maximum order of the computations.

Rather than directly allocating terms present in the result, the multiplication algorithm accumulates terms in a permanent pre-ordered "scratch pad", essentially a work area containing slots for all possible monomials. Once monomial multiplications have been exhausted, *only* nonzero terms present in the scratch pad are copied *in order* into a new `Jet`.

**Other Operations** While addition and subtraction are essentially the same, the division algorithm differs substantially. However, division, as well as all other basic mathematical operations can be reduced to a sequence of additions and multiplication. Since the later both preserve monomial ordering, it follows that *all operations preserve monomial ordering*.

## PERFORMANCE CONSIDERATIONS

Keeping in mind that operations involving `Jets` involve manipulation, allocation and de-allocation of very large numbers of individual monomials, maximizing efficiency hinges on a few principles: (1) avoid searching for specific monomials (2) make the cost of comparing (the exponents of) two monomials as low as possible (3) reduce as much as possible the cost (both in terms of cpu cycles and memory utilization) of copying Jets (4) reduce as much as possible the costs of allocating and deallocating `Jets`.

(1) and (2) are neatly taken care of by the indirect monomial indexing scheme: two monomials can be compared by comparing their respective offset within an ordered table. Item (3) deserves some discussion.

While the need to *explicitly* copy variables in the course of a computation is obvious, in the context of overloaded operators, a specific issue arises: implicit temporary copies. The issue can be illustrated with the help of two

simple examples. First, consider the addition of two jets. The implementation of the addition operator accumulates the sum of both operands in new Jet variable which must then be returned to the caller. Typically, an anonymous temporary copy of the local variable (in short, a temporary), is returned. In situations involving complex expressions, many temporaries can be implicitly allocated and deallocated. Consider

$$Y = X_1 + X_2 + X_3 + X_4 \tag{2}$$

where $Y, X_1, ..., X4$ are `Jets`. In C++, this expression is evaluated as $X_1 + (X_2 + (X_3 + X_4))$. The result of $(X_3 + X_4)$ is a temporary, which is subsequently added to $X_2$ and so forth. Clearly, the temporaries are not necessary, and it would be more efficient to accumulate the sum of all three vectors $X, Y$ and $Z$ directly into $V$. The compiler must generate temporaries because it can make no a-priori assumptions about the nature of the operation performed by operator "+". Interestingly, this behavior accounts for much of the folklore about the better performance of Fortran in scientific computing. Various methods exists that mitigate almost entirely this "complex expression temporary problem"; some involve compile-time template metaprograms, others involve run-time expression trees in conjunction with delayed evaluation. In principle, these techniques can be applied to `Jet` computations. However, the additional code complexity introduced is significant.

An important observation is that temporaries are *read-only copies*. Much of the cost of copying can be eliminated through the use of reference counting. In `mxyzptlk`, `Jets` are implemented using the pointer-to-implementation (pimpl) idiom. In a nutshell, `Jet` operations are dispatched to reference-counted smart pointers. For a read-only `Jet`, the cost of a copy reduces to that of incrementing a counter and copying a single pointer. Note that the pimpl idiom has significant advantages over raw pointer manipulations: (i) there is no object ownership ambiguity (ii) `Jets` have value semantics and memory is automatically de-allocated (if necessary) whenever a `Jet` goes out of scope.

To close this discussion, we must mention that for those situations where a `Jet` deep copy (that is, a copy involving all terms) cannot be avoided, `mxyzptlk` uses a specialized fixed-size block allocator for `JLterms`. The cost of a de-allocation is made trivial through a recycling strategy which involves assigning deleted `Jets` to a special "recycling pool".

It is not possible to provide additional details here. Suffice it to say that a profiling tool reveals that in complex computations involving `Jets`, memory allocation typically accounts for less than 1% of the CPU time, even though the `JLterm` allocation overwhelmingly dominates (by orders of magnitude) in terms of number of function calls. In fact, just as one would expect, the multiplication operation consumes most of the CPU cycles, typically about 75%.

In closing, it is worth mentioning that sparsity and the concept of monomial ordering based on weight and lexicographical order have been present in `mxyzptlk` since the beginning. Reference counting was partially introduced many years ago but was not implemented in terms of modern idioms and other recent advances. Although some specifics differ, our use of table-driven monomial multiplication was influenced by previously published work e.g. Ref. [4], which credits the idea to A. Dragt from the University of Maryland.

## BENCHMARKS

We present here timing results for a sample computation that should convey a sense of the performance of `mxyzpltlk`. In this example, a one turn, $n$-th order phase-space map for a ring comprised of about 300 elements, including combined-function magnets, quadrupoles and sextupoles. All computations are performed on a standard single-core 3.0 GHz Pentium 4 CPU. Remarkably, at first order, performance matches what one would expect from a conventional matrix-based code.

| Order | Monomials | CPU time [s] |
|---|---|---|
| 1 | 7 | 0.08 |
| 2 | 28 | 0.52 |
| 3 | 84 | 1.42 |
| 4 | 210 | 4.37 |
| 5 | 462 | 16.08 |
| 6 | 924 | 51.35 |
| 7 | 1716 | 146.76 |
| 8 | 3003 | 367.58 |

## CONCLUSION

`mxyzptlk` is now a competitive AD engine. In addition to good performance, it offers: (1) a reasonable and dynamic memory footprint (no a-priori allocation) (2) a fully native C++ implementation that does not involve a specialized syntax and the associated overhead (3) a compact and maintainable implementation, resulting from extensive usage of templates and modern idioms.

## REFERENCES


[1] L. Michelotti, "Differential Algebras Without Differentials: An Easy C++ Implementation", PAC 1989 Chicago.

[2] L. Michelotti, "A C++ Hacker's Implementation of Automatic Differentiation", Automatic Differentiation of Algorithms Theory, Implementation, and Application, SIAM, Philadephia, 1991.

[3] R.L. Anderson and N. H. Ibragimov, "Lie-Bäcklund Transformations in Applications", "SIAM Studies in Applied Mathematics, Philadelphia, 1979.

[4] S. G. Shasharina and J. R. Cary, "Efficient Differential Algebra Computations", PAC 1999 New-York, p. 377