SSCL-659
SLAC-PUB-10040

ZLIB++: Object-Oriented Numerical Library for Differential Algebra

N. Malitsky and A. Reshetov

Superconducting Super Collider Laboratory*
2550 Beckleymeade Avenue
Dallas, TX 75237

and

Y. Yan

Stanford Linear Accelerator Center
Stanford, CA 94309

January 1994

* Operated by the Universities Research Association, Inc., for the U.S. Department of Energy under Contract No. DE-AC35-89ER40486.

Work supported in part by Department of Energy Contract DE-AC03-76SF00515

1.0 INTRODUCTION

New software engineering tools and object-oriented design have a great impact on the software development process. But in high energy physics all major packages were implemented in FORTRAN and porting of these codes to another language is rather complicated, primarily because of their huge size and heavy use of FORTRAN mathematical libraries. But some intrinsic accelerator concepts, such as nested structure of modern accelerators, look very pretty when implemented with the object-oriented approach. In this paper we present the object-oriented version of ZLIB,[1] numerical library for differential algebra,[2] and show how the modern approaches can simplify the development and support of accelerator codes, decrease code size, and allow description of complex mathematical transformations by simple language.

2.0 TRUNCATED POWER SERIES

The truncated power series (TPS) expansion of the arbitrary function $U(\vec{\mathbf{z}})$ is defined as:[1]

$$U(\vec{\mathbf{z}}) = \sum_{k=0}^{\Omega} u(\vec{\mathbf{k}}) \vec{\mathbf{z}}^{\vec{\mathbf{k}}} \ , \tag{1}$$

where

$$\vec{\mathbf{z}}^T = [z_1, z_2, ..., z_n] \ ,$$

$$\vec{\mathbf{z}}^{\vec{\mathbf{k}}} \equiv z_1^{k_1} z_2^{k_2} ... z_n^{k_n} \ ,$$

$$k = \sum_{i=1}^{n} k_i, \ \text{for } 0 \le k_i \le \Omega \ .$$

In the new object-oriented version of ZLIB, TPS is considered as C++ class **ZSeries**, which includes overloaded assignment, additive, multiplicative operators and two additional functions **dif** and **poisson** (Appendix A). To simplify the form of the TPS transformation equations we include the additional private member **order** [order Ω of TPS expansion in Eq. (1)] and several rules for its usage. Below we describe the main elements of class **ZSeries** and their relationship with the subroutines of the previous FORTRAN version of ZLIB. Full description of fundamental TPS operations may be found in a users guide for ZLIB 1.0.[1]

2.1 Definitions

In expressions and examples in this report we will use the following notation:

U, V, W	are	the instances of class **ZSeries**;
c	is	constant or variable of the **double** type;

ZLIB_ORDER	is	global variable, which determines the maximum TPS order;
ZLIB_DIM	is	global variable, which determines the phase-space dimension.

The parameters **ZLIB_ORDER** and **ZLIB_DIM** must be defined by user before all assignment operators, because they determine memory allocation for members of class **ZSeries**.

2.2 Assignment Operators

An assignment operator is used to set a **ZSeries** variable to constant or another **ZSeries** value. This operator returns **void** and so a vague statement $(\mathbf{a} = \mathbf{b}) = \mathbf{c}$ will result in a syntax error. Here is the summary of different cases of assignment operator:

2.2.1 Operator =

1. $\mathbf{W} = \mathbf{c}$;

$$w(\vec{\mathbf{k}}) = \mathbf{c} \qquad \text{for } 0 \le k \le \mathbf{W.order}$$
$$\mathbf{W.order} = 0 \ .$$

2. $\mathbf{W} = \mathbf{U}$;

$$w(\vec{\mathbf{k}}) = u(\vec{\mathbf{k}}) \qquad \text{for } 0 \le k \le \mathbf{U.order}$$
$$\mathbf{W.order} = \mathbf{U.order} \ .$$

2.2.2 Operators += and −=

1. $\mathbf{W} \mathrel{+}= \mathbf{c}$;

$$w(\vec{\mathbf{k}}) = w(\vec{\mathbf{k}}) + \mathbf{c} \qquad \text{for } \ k = 0 \ ,$$
$$w(\vec{\mathbf{k}}) = w(\vec{\mathbf{k}}) \qquad \text{for } 0 < k \le \mathbf{W.order} \ .$$

2. $\mathbf{W} \mathrel{+}= \mathbf{U}$;

$$W(\vec{\mathbf{z}}) = \sum_{k=0}^{\mathbf{W.order}} w(\vec{\mathbf{k}})\vec{\mathbf{z}}^{\vec{\mathbf{k}}} + \sum_{k=0}^{\mathbf{U.order}} u(\vec{\mathbf{k}})\vec{\mathbf{z}}^{\vec{\mathbf{k}}} \ ,$$

$$\mathbf{W.order} = \max(\mathbf{W.order}, \mathbf{U.order}) \ .$$

3. $\mathbf{W} \mathrel{-}= \mathbf{c}$; and $\mathbf{W} \mathrel{-}= \mathbf{U}$;

These subtraction operators are determined similar to addition (Section 2.2.2).

2.2.3 Operator *=

1. $\mathbf{W} \mathrel{*}= \mathbf{c}$;

$$w(\vec{\mathbf{k}}) = w(\vec{\mathbf{k}}) * \mathbf{c} \qquad \text{for } 0 \le k \le \mathbf{W.order} \ .$$

2. $\mathbf{W}\ *= \mathbf{U}$;

$$W(\vec{\mathbf{z}}) = \sum_{k=0}^{\mathbf{W.order}} w(\vec{\mathbf{k}})\vec{\mathbf{z}}^{\vec{\mathbf{k}}} * \sum_{k=0}^{\mathbf{U.order}} u(\vec{\mathbf{k}})\vec{\mathbf{z}}^{\vec{\mathbf{k}}} ,$$

$$\mathbf{W.order} = \min(\mathbf{W.order} + \mathbf{U.order.ZLIB\_ORDER}) .$$

2.2.4 Operator /=

1. $\mathbf{W}\ /= \mathbf{c}$;

$$w(\vec{\mathbf{k}}) = w(\vec{\mathbf{k}})/\mathbf{c} \qquad \text{for } 0 \le k \le \mathbf{W.order} .$$

2. $\mathbf{W}\ /= \mathbf{U}$;

The implementation of this operator is based on two functions: multiplication (Section 2.2.3) and inversion $\mathbf{1/U}$ (Section 2.4) and defined as:

$$\mathbf{W}* = (\mathbf{1/U}) .$$

2.3 Additive and Multiplicative Operators

Additive (+ and −) and multiplicative (* and /) operators are similar to corresponding assignment operators (Sections 2.2.2, 2.2.3 and 2.2.4), but unlike assignment operators they return reference to **tmpZSeries**, temporary instance of class **ZSeries**, which is created as a result of the expression. This allows user to write usual mathematical expressions as:

$$\mathbf{W}* = (\mathbf{U} - \mathbf{c})/\mathbf{V} * \mathbf{c} + \ldots$$

2.4 Inverse Operator

Inversion of the object $\mathbf{U}$ is expressed as a Taylor expansion of $1/(1 + \mathbf{V})$, where $\mathbf{V} = \Big(\mathbf{U} - u(0)\Big)/u(0)$:

$$\mathbf{W} = 1./\mathbf{U} = \frac{1}{u(0)\left(1 + \frac{\mathbf{U}-u(0)}{u(0)}\right)}$$

$$= \frac{1}{u(0)} * \sum_{n=0}^{\mathbf{U.tmpOder}} (-1)^n \left(\frac{\mathbf{U} - u(0)}{u(0)}\right)^n .$$

The implementation of this expression is based on **ZSeries** multiplicative and additive operators (Appendix B).

2.5 Functions

This object-oriented version of ZLIB supports two functions **dif** and **poisson**. As in the case of additive and multiplicative operators (Section 2.3), these functions return reference to temporary object **tmpZSeries** and may be combined with other operators in the complex expression.

2.5.1 Derivative

The function **dif**$(\mathbf{U}, iv)$ returns the partial derivative $(\partial/\partial z_{iv})U(\vec{\mathbf{z}})$.

2.5.2 Poisson Bracket

Poisson bracket is the main operator of differential algebra. It associated with a Lie operator : $U(\vec{\mathbf{z}})$: as in Reference 2:

$$\mathbf{poisson}(\mathbf{U},\mathbf{V}) \equiv: U(\vec{\mathbf{z}}) : V(\vec{\mathbf{z}}) \equiv [U(\vec{\mathbf{z}}), V(\vec{\mathbf{z}})] = -\left(\frac{\partial U(\vec{\mathbf{z}})}{\partial \vec{\mathbf{z}}}\right)^T S\left(\frac{\partial V(\vec{\mathbf{z}})}{\partial \vec{\mathbf{z}}}\right),$$

where S is the symplectic identity.

2.6 Access Operators

Access operators may be used to change directly the private members of class **ZSeries** (coefficients of the TPS and its order **order**) and some specific rules of the transformation of **order**, described in Section 2.3. These rules may be generalized and expressed as the following:*

Rule 1 *The order of the object* **W**, *created by constructor* **ZSeries** :: **ZSeries**(), *is equal to zero, i.e., for the new object* **W.order** $= 0$;

Rule 2 *The order of the object* **W** *in the left side of an assignment operator* $\mathbf{W} = \mathbf{U}$ *is determined by the order of* **U**, *i.e.,* **W.order** $=$ **U.order**.

Rule 3 *The order of the object* **W**, *which is obtained as a result of additive operations* $(+=, -=, +$ and $-)$ *is equal to the maximum order of items* **U** *and* **V**, *i.e.,* **W.order** $=$ $\max($**U.order**, **V.order**$)$.

Rule 4 *The order of the object* **W**, *which is obtained as a result of multiplicative operations* $(*=, /=, *$ and $/)$ *or binary functions* (**poisson(U,V)**) *is equal to* $\min($**U.order** $+$ **V.order**, **ZLIB_ORDER**$)$.

* These rules are valid also for the expressions with the constant **c**, regarding its order to be zero.

2.6.1 Operator()(int)

This operator (round brackets) allows user to temporarily change the order **W.order** of the object **W** in the intermediate expression. To assign new order **newOrder** "permanently" user could use the following construction:

$$\mathbf{U} = \mathbf{U}(\mathbf{newOrder}) \ .$$

2.6.2 Operator()(int, int)

The second parameter in **operator()** allows user to bypass **Rule 4** for multiplicative operators (Section 2.6) in accordance with the following definition:

$$\mathbf{W} = \mathbf{U} * \mathbf{V}(\mathbf{V.order}, \mathbf{V.mltOrder})$$
$$\mathbf{W.order} = \min(\mathbf{V.mltOrder}, \mathbf{W.mltOrder}(\text{from } \mathbf{Rule\ 4})) \ .$$

We used **operator()** in the implementation of the **inverse operator** for the order of $\mathbf{1/U}$ not to exceed the order of **U** (Section 2.4 and Appendix B).

2.6.3 Operator[ ]

The subscripting operator[ ](*int* i) of the object **U** returns the reference to its private member **U.z[i]**, which represents the i-th coefficient $u(i)$ of the TPS expansion in Eq. (1).

3.0 MAPS

Map $\vec{\mathbf{U}}(\vec{\mathbf{z}})$ is the development of concept of the truncated power series (TPS) and defined as the m-dimensional vector of TPS expansions, Eq. (1):

$$\vec{\mathbf{U}}(\vec{\mathbf{z}}) = \sum_{k=0}^{\Omega} \vec{\mathbf{u}}(\vec{\mathbf{k}})\vec{\mathbf{z}}^{\vec{\mathbf{k}}} \ , \tag{2}$$

where

$$\vec{\mathbf{u}}(\vec{\mathbf{k}})^T = [u_1(\vec{\mathbf{k}}), u_2(\vec{\mathbf{k}}), \ldots, u_m(\vec{\mathbf{k}})] \ ,$$

$$k = \sum_{i=1}^{n} k_i, \text{ for } 0 \le k_i \le \Omega \ .$$

In the object-oriented version of ZLIB the map is considered as an object of C++ class **ZMap**, which naturally was derived from the class **ZSeries** (Appendix C). Moreover, all arithmetical operators of class **Zmap** follow the same **Rules** (Section 2.6) and are based on **ZSeries** multiplicative and additive operators. This leads to similar implementation of member functions for these two classes. For example, compare inverse operator for **ZMap** (Appendix D) and **ZSeries** (Appendix B). Below we describe only the essential distinction between these classes and the additional **ZMap** functions.

3.1 Unit Map

Unlike the truncated power series the unit map **I** is defined as a vector:

1. **I** = 1 ;

$$\vec{\mathbf{I}}(\vec{\mathbf{z}}) = \vec{\mathbf{z}},$$

$$\mathbf{I.order} = 1 \ .$$

3.2 Operator[ ]

The subscripting operator[ ](*int* i) of the object **M** (the instance of class **ZMap**) returns the reference to its private member **M.z[i]** the instance of class **ZSeries**) which represents the i-th member u_i of the m-dimensional vector $\vec{\mathbf{u}}$ in Eq. (2).

3.3 Function Poisson (ZSeries&, ZMap&)

Function **poisson(V, M)** returns the Poisson bracket: $V(\vec{\mathbf{z}}) : \vec{\mathbf{M}}(\vec{\mathbf{z}}) \equiv [V(\vec{\mathbf{z}}), \vec{\mathbf{M}}(\vec{\mathbf{z}})]$.[2] Its implementation is based on the similar **ZSeries** function as:

```
ZMap& poisson(ZSeries& V, ZMap& M)
{
                ...
  for(int i = 1; i <= ZLIB_DIM; i++)
    bracket[i] = poisson(V, M[i]);
  return(bracket);
}
```

where **bracket** is the temporary instance of class **ZMap**.

4.0 TRACKING

Tracking is one of the most important procedures in accelerator codes. In our "language" it is defined simply and naturally:

$$\mathbf{y} = \mathbf{M} * \mathbf{x} \ ; \tag{3}$$

where **M** is the **ZMap** object, and **x** and **y**, the instances of class **Particle** (Appendix D), consist of the particle coordinates correspondingly before and after one turn. For multi-particle tracking user may use usual C++ operators:

```
main()
{
   ZMap M;
   Particle ** x;
                ...
   for(int i = 1; i <= numberParticles; i++)
      for(int j = 0; j < numberTurns; j++)
         x[i][j + 1] = M * x[i][j];
                ...
}
```

5.0 CONCLUSION

In this report we have described a new object-oriented version of the ZLIB package. ZLIB++ defines two classes (**ZSeries** and **ZMap**) to represent specific accelerator objects. We argued that simple C operators (such as $=, +, -, /, *$) are naturally suited to implement different mathematical algorithms with these objects. It enables one to write simple, self-documented programs for applications of numerical methods of differential algebra in high energy physics.

ZLIB++ was designed to be a foundation for further developments. New classes could be naturally derived from the base ones, enabling expansion of the package in the open architecture style.

ACKNOWLEDGEMENTS

We would like to thank Dr. G. Bourianoff for his strong support and for various helpful discussions.

REFERENCES

1. Y. Yan and Chiung-Ying Yan, "ZLIB—A Numerical Library for Differential Algebra," SSC Laboratory Report SSCL-300, (1990).
2. Y. Yan, "Applications of Differential Algebra to Single-Particle Dynamics in Storage Rings," SSC Laboratory Report SSCL-500, (1991).

APPENDIX A

```
// File          : ZSeries.hh
// Description   : This file contains the definition of ZSeries class (TPS -
//                 truncated power series).
// Created       : February 1, 1994
// Authors       : Nikolay Malitsky (malitsky@ivory.ssc.gov )
//                 Alexander Reshetov(reshetov@vernon.ssc.gov)
//
//
// (C) Copyright
// SSC Laboratory
// 2550 Beckleymeade Ave.
// Dallas, TX, 75237
//

#define ZSERIES_H
#define ZSERIES_H

#include "Zdef.hh"

class ZSeries
{
 public :

                ZSeries();
                ZSeries(ZSeries& V);

// Access operators & function

                ZSeries& operator()(int i1);
                ZSeries& operator()(int i1, int i2);
                double&  operator[](int number);

// Assignment operators

                void operator= (double  c);
                void operator= (ZSeries& V);
                void operator+=(ZSeries& V);
                void operator+=(double  c);
                void operator-=(ZSeries& V);
                void operator-=(double  c);
                void operator*=(ZSeries& V);
                void operator*=(double  c);
                void operator/=(ZSeries& V);
                void operator/=(double  c);

// Additive & Multiplicative Operators

                ZSeries& operator+(ZSeries& V);
```

```
        ZSeries& operator-(ZSeries& V);
        ZSeries& operator*(ZSeries& V);
        ZSeries& operator/(ZSeries& V);

// Friend operators

        friend ZSeries& operator-(ZSeries& V);
        friend ZSeries& operator+(ZSeries& V, double  c);
        friend ZSeries& operator+(double  c, ZSeries& V);
        friend ZSeries& operator-(ZSeries& V, double  c);
        friend ZSeries& operator-(double  c, ZSeries& V);
        friend ZSeries& operator*(ZSeries& V, double  c);
        friend ZSeries& operator*(double  c, ZSeries& V);
        friend ZSeries& operator/(ZSeries& V, double  c);
        friend ZSeries& operator/(double  c, ZSeries& V);
        friend int ZSeriesSize(ZSeries& V);
        friend ostream& operator<<(ostream& out, ZSeries& V);

// Functions

        friend ZSeries&     dif(ZSeries& V, int iv);
        friend ZSeries&     poisson(ZSeries& V1, ZSeries& V2);

        ~ZSeries();
private:
        . . .
};

#endif
```

APPENDIX B

```
// File          : ZSeries.cc
// Description   : This file contains implementation of ZSeries class
// Created       : February 1, 1994
// Authors       : Nikolay Malitsky (malitsky@ivory.ssc.gov )
//                 Alexander Reshetov(reshetov@vernon.ssc.gov)
//
//
// (C) Copyright
// SSC Laboratory
// 2550 Beckleymeade Ave.
// Dallas, TX, 75237
//

 . . .

ZSeries& operator/(double c, ZSeries& V)
{
 ZSeries El;
 ZSeries sum;

 El = V;
 Int tN = El.prepareTmpZSeries();

 if (fabs(El[1]) < ZTINY)
 {
  cerr << "Error: ZSeries:Binary operator c/V : fabs(V[1]) = ";
  cerr << fabs(El[1]) << " < " << ZTINY << "\n";
  exit(1);
 }

 double linearInv = 1/El[1];
 sum  = linearInv;

 El -= El[1];
 El *= -1;

 int now = ZSeriesOrder(El);
 for (int i=1; i <= now; i++)
 {
   sum  *= El(i, i);
   sum  += 1;
   sum  *= linearInv;
  }

 sum *= c;

 *sum.tmpZSeries[tN] = sum;
```

```
 return (*sum.tmpZSeries[tN]);
}

...
```

APPENDIX C

```
// File          : ZMap.hh
// Description   : This file contains the definition of ZMap class
//                 (one-turn Maps)
// Created       : February 1, 1994
// Authors       : Nikolay Malitsky (malitsky@ivory.ssc.gov )
//                 Alexander Reshetov(reshetov@vernon.ssc.gov)
//
//
// (C) Copyright
// SSC Laboratory
// 2550 Beckleymeade Ave.
// Dallas, TX, 75237
//

#ifndef ZMAP_H
#define ZMAP_H

#include "ZSeries.hh"
#include "Particle.hh"

class ZMap: public ZSeries
{
 public :

                ZMap();
                ZMap(ZMap& M);

// Access operators & function

                ZMap&   operator()(int in1);
                ZMap&   operator()(int in1, int in2);
                ZSeries& operator[](int number);

// Assignment operators

                void operator= (double  c);
                void operator= (ZMap& M);
                void operator+=(ZMap& M);
                void operator+=(double  c);
                void operator-=(ZMap& M);
                void operator-=(double  c);
                void operator*=(ZMap& M);
                void operator*=(double  c);
                void operator/=(ZMap& M);
                void operator/=(double  c);

// Additive & Multiplicative Operators
```

```
                ZMap& operator+(ZMap& M);
                ZMap& operator-(ZMap& M);
                ZMap& operator*(ZMap& M);
                ZMap& operator/(ZMap& M);

// Friend operators & function

                friend ZMap& operator-(ZMap& M);
                friend ZMap& operator+(ZMap& M, double  c);
                friend ZMap& operator+(double  c, ZMap& M);
                friend ZMap& operator-(ZMap& M, double  c);
                friend ZMap& operator-(double  c, ZMap& M);
                friend ZMap& operator*(ZMap& M, double  c);
                friend ZMap& operator*(double  c, ZMap& M);
                friend ZMap& operator/(ZMap& M, double  c);
                friend ZMap& operator/(double  c, ZMap& M);
                friend int ZMapSize(ZMap& M);
                friend ostream& operator<<(ostream& out, ZMap& M);

// Tracking

                Particle& operator*(Particle& P);

// Functions

                friend ZMap&      poisson(ZSeries& V, ZMap& M);

                ~ZMap();
private:
                . . .
};

#endif
```

APPENDIX D

```
// File          : ZMap.cc
// Description   : This file contains implementation of ZMap class
// Created       : February 1, 1994
// Authors       : Nikolay Malitsky (malitsky@ivory.ssc.gov)
//                 Alexander Reshetov(reshetov@vernon.ssc.gov)
//
//
// (C) Copyright
// SSC Laboratory
// 2550 Beckleymeade Ave.
// Dallas, TX, 75237
//

...

ZMap& operator/(double c, ZMap& M)
{
 ZMap El;
 ZMap sum;

 El = M;
 int tN = El.prepareTmpZMap();

 for (int i=1; i <= ZLIB_DIM; i++)
 {
  if (fabs(El[i][1]) > ZTINY)
  {
   cerr << "Error: ZMap:Binary operator c/M : ";
   cerr << "fabs(M.z[" << i << "][1] = ";
   cerr << fabs(M.z[i][1]) << " > " << ZTINY << "\n";
   exit(1);
  }
 }

 ZMap linearInv;
 linearInv.linearInversion(El);
 sum = linearInv;

 El -= El(1);
 El *= -1;

 int now = ZMapOrder(El);
 for (i=2; i <= now; i++)
 {
   sum *= El(i, i);
   sum += 1;
   sum *= linearInv;
```

```
    }

  sum *= c;

  *sum.tmpZMap[tN] = sum;
  return (*sum.tmpZMap[tN]);
}
...
```

APPENDIX E

```
// File          : Particle.hh
// Description   : This file contains the definition of Particle class
//                 (ZLIB_DIM-dimensional particle coordinates.)
// Created       : February 1, 1994
// Authors       : Nikolay Malitsky (malitsky@ivory.ssc.gov )
//                 Alexander Reshetov(reshetov@vernon.ssc.gov)
//
//
// (C) Copyright
// SSC Laboratory
// 2550 Beckleymeade Ave.
// Dallas, TX, 75237
//
#ifndef PARTICLE_H
#define PARTICLE_H

#include "Zdef.hh"

class Particle
{
 public :

                Particle();
                Particle(Particle& P);

// Access operators

                double& operator[](int number);

// Assignment operators

                void operator= (Particle& P);

// Friend operators

                friend ostream& operator<<(ostream& out, Particle& P);

                ~Particle();

 private:

                double* z;      // ZLIB_DIM-dimensional particle coordinates.

};

#endif
```