ARRAY BASED TRUNCATED POWER SERIES PACKAGE*

Lingyun Yang[†], NSLS-II, BNL, Upton, NY 11973, USA

Abstract

Truncated Power Series Algebra (TPSA) or Differential Algebra (DA) package has been a fundamental component for many accelerator physics simulation code, including FPP/PTC, MAD-X, BMAD and Tracy. We have developed a new algorithm to extend the ability of TPAS to handle problems with both large number of variables and high order. This package is implemented in C++ language with operator overloading, and has been integrated into PTC and MAD-X.

INTRODUCTION

Truncated Power Series Algebra (TPSA) or Differential Algebra (DA) package is a tool for Taylor series manipulation [1, 2]. It follows certain mathematical rules to do arithmatics among the Taylor series and can be used for map generating, sensitivity study, automatic differentiation and many other applications. Since it was implemented by Dr. Berz, it has been a fundamental component for many accelerator physics simulation codes, including FPP/PTC [4], MAD-X [3], BMAD and Tracy.

The package in Ref. [1] has two limitations:

- Limited index space at hign v, e.g. 39 variables can only handle up to first order[1].
- Without operator overloading, Applying DA to an existing code will need many careful translation, and introduce temporary variable. (think about translate $x = (a + b) * c/d$).

We have developed a new TPSA package in C++ recently, with a new algorithm which can expand the ability to both high number of variables and high orders. It is also more user friendly and the speed are better than the FORTRAN 77 implementation.

ALGORITHM

The Taylor series of an given function $f(x_1, \cdots, x_d)$ is given as:

$$f(x_1, \cdots, x_d) = \sum_{n_1=0}^{\infty} \cdots \sum_{n_d=0}^{\infty} \frac{(x_1 - a_1)^{n_1} \cdots (x_d - a_d)^{n_d}}{n_1! \cdots n_d!} \left(\frac{\partial^{n_1+\cdots+n_d} f}{\partial x_1^{n_1} \cdots \partial x_d^{n_d}} \right) \quad (1)$$

Given the analytic form of $f(x_1, \cdots, x_d)$, we can calculate the coefficient of Taylor series expansion, f_{x_1}, f_{x_2}, $f_{x_1 x_2}$, up to any order precisely. This can be done with finite difference method, but when going to high order, the truncation error and linear approximation may be a main obstacle. TPSA package uses a set of rule, which is complete for a field of real number and basic arithmetics. The rules can be found in Ref. [2], and we summerize here:

- Addition/Substraction. It is done between only corresponding terms (coefficients with same order or pattern of differentiation).
- Multiplication with constant. It scales all the coefficients.
- Multiplication with another series. The order of differentiation are added up for each variable. The new coefficient are added up to the new location with new pattern of order.
- Reciprocal. The coefficient is gained from the pattern of Taylor expansion of $1/x$.
- Basic functions. Such as $\sin(x)$,$\cos(x)$,$\sqrt{x}$ are just deduced from their coefficients of Taylor series expansion together with the multiplication of two series.

As we can see that +/- are trivial, and every other calculation is depending on multiplication of two TPAS series. While for multiplication the key part is how to arrange the storage of each coefficients and locate them quickly.

IMPLEMENTATION

A TPSA vector with v independent variables, each up to d degrees, will have $C(v + d, d)$ or $C(v + d, v)$ elements, where $C(n + v, v)$ are binomial coefficients defined as

$$C(v + d, v) \equiv \binom{v + d}{v} = \binom{v + d}{d} = \frac{(v + d)!}{v!d!}$$

From this we can see that, for the length of a TPSA vector, the order and number of variables are symmetric. A code can do high order with low number of variables should be

* Work supported by the Director, Office of Science, U. S. Department of Energy under Contract No. DE-AC02-05CH11231.

[†] lyyang@bnl.gov

[1] We can use more index function again, 3 instead of 2, to relax the limit from $(n + 1)^{v/2}$ to $(n + 1)^{v/3}$, but the performance will be lower.

Table 1: (v, d) Limited by Certain Number of Monomials N, e.g. $1k, 1M, 1G, 1T$.

	d				
v	$< 2^{10}$ (k)	$< 2^{20}$ (M)	$< 2^{30}$ (G)	$< 2^{40}$ (T)	$< 2^{50}$ (P)
4	10	68	398	$\approx 2^{40/4} * 4!$	$\approx 2^{50/4} * 4!$
6	6	26	92	300	962
8	4	16	46	115	282
10	4	12	30	67	139
12	3	10	23	46	88
14	3	9	19	36	64
16	3	8	16	30	51
18	2	7	15	26	42
20	2	7	13	23	36
30	2	5	10	16	23
40	2	4	8	13	18
50	1	4	7	11	16
60	1	4	7	10	14
70	1	3	6	9	13
80	1	3	6	9	12
90	1	3	6	8	11
100	1	3	5	8	11

able to do low order with high number of variables. Some examples are shown in Table 1. Where given the maximal number of terms in Taylor series, the table tells how many variables and upto which order the code can store.

The Taylor series expansion of a multi-variable function up to certain order has a *polynomial* structure, we write as (v, d), where v is the number of variables, and d is the highest order considered. It is stored as a vector, where each element represents one term in Taylor Series Expansion.

In order to speed up the multiplication, we build up a look up table P. The element $P[i, j]$ is the location to save $v[i] \times v[j]$. A look up table for the above multiplication, 2 variables, order up to 3rd order looks like Table. 2

Table 2: Hash table for TPSA multiplication $(2, 3)$

		0	1	2	3	4	5	6	7	8	9
1	0	0	1	2	3	4	5	6	7	8	9
$\frac{\partial}{\partial x}$	1	1	3	4	6	7	8				
$\frac{\partial}{\partial y}$	2	2	4	5	7	8	9				
$\frac{\partial^2}{\partial x \partial x}$	3	3	6	7							
$\frac{\partial^2}{\partial x \partial y}$	4	4	7	8							
$\frac{\partial^2}{\partial y \partial y}$	5	5	8	9							
$\frac{\partial^3}{\partial x \partial x \partial x}$	6	6									
$\frac{\partial^3}{\partial x \partial x \partial y}$	7	7									
$\frac{\partial^3}{\partial x \partial y \partial y}$	8	8									
$\frac{\partial^3}{\partial y \partial y \partial y}$	9	9									

We should first notice that Table. 2 is symmetric, i.e. $P[i, j] = P[j, i]$, therefore only half of the storage is needed. Secondly, since we are truncating at certain order, the table is not squared but a step-like. The real size of this matrix $T(v, d)$ is

$$\begin{aligned} T(v,d) &= \sum_{k=0}^{d} \binom{v+k-1}{k} N(v, d-k) \\ &\equiv \sum_{k=0}^{d} \binom{v+k-1}{k} \binom{v+d-k}{d-k} \\ &= \sum_{k=0}^{d} \binom{v-1+k}{v-1} \binom{v+d-k}{v} \\ &= \binom{2v+d}{d} \end{aligned} \tag{2}$$

where $N(v, d)$ is the length of TPSA vector (v, d). From Table. 1 we can easily find the limit on (v, d) when using this hash table.

The above hash table can solve problems both low order but high number of variables and high order but small number of variables, but for both high order and large number of variables, the hash table becomes too large to store.

We can arrange the Taylor series in a way such that, the new location of multiplication of two TPSA coefficients can be identified recursively. The number of recursion is only linear with number of variables in this TPSA. Therefore we used a small hash table H which has a size of $v \times v$. This is not a constraint at all when comparing with the size of TPSA vectors. The location of derivative pattern b_k is

$$C(b_k) = \sum_{i=1}^{v} H(i, \sum_{j=v}^{i} b_{k,j})$$

where $H(v, d)$ is the hash function. One example is shown in Table 3. The derivative pattern b_k is a vector represents the order of derivative to each variable. For example, the index of $(1\,2\,1\,0)$ is 46 from Table 3:

$$\begin{aligned} & C[\,(0\,0\,1\,0) * (1\,2\,0\,0)\,] = C(1\,2\,1\,0) \\ =& H(1,0) + H(2,(0+1)) + \\ & H(3,(0+1+2)) + H(4,(0+1+2+1)) \\ =& H(1,0) + H(2,1) + H(3,3) + H(4,4) \\ =& 0 + 1 + 10 + 35 \\ =& 46 \end{aligned} \tag{3}$$

The overall complexity is $O(v)$ when $v \approx d$ and $\binom{d}{v+d}$ is too large to fit in memory.

Once the multiplication is well implemented, the rest is just a trivial application of it to Taylor series expansion. More detailed formula can be found in Ref. [2, 1].

BENCHMARK

This new package is compared with F77 implementation of TPSA (DA), and shown in Fig. 1. This speed test is very coarse, and only used system tools of Linux platform. Our conclusion here is that the new TPSA in C++ is faster than the F77 implementation, and with more features and abilities.

Table 3: **Hash Table** $H(i, j)$

i	0	j 1	2	3	4	5
1	0	1	2	3	4	5
2	0	1	3	6	10	15
3	0	1	4	10	20	35
4	0	1	5	15	35	70

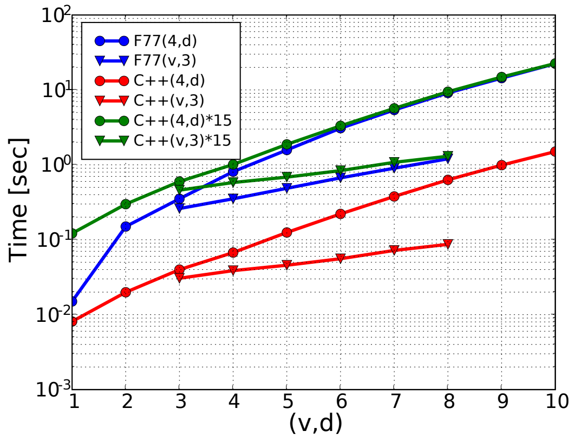

Figure 1: Performance comparison of F77 and C++ version.

CODE EXAMPLE

The following code shows a simple example calculating Taylor series of $f(x_1, x_2) = 1/(x_1 + x_2)$. With operator overloading the code is very straight forward. The package introduces a new type called `TPSA`, and trys to mimic the native C++ types, such as `double`, `string`. I am trying to make it transparent to users.

```
#include <iostream>
#include <ctime>
#include "tpsa.hh"

using namespace std;

int main()
{
    // number of variable, degree
    const unsigned char cnv = 3, cnd = 10;

    // first independent variable, with index 0
    // initialized as 1.0
    TPSA<double, cnv, cnd> a(1.0L, 0);
    // 2nd, as 2.0
    TPSA<double, cnv, cnd> b(2.0L, 1);
    // 3rd, as 3.0, not used
    // TPSA<double, cnv, cnd> c(3.0L, 2);

    TPSA<double, cnv, cnd> aa; // normal variable

    aa = 1.0/(a+b);

    cout << aa << endl; // print out the result.

    // more example:
    // sin(aa), cos(aa), sqrt(aa), tan(aa).

    return 0;
}
```

The output of FORTRAN version `TPSALib.f` and C++ version is in Table. 4.

Table 4: Output of FORTRAN and C++ version

FORTRAN				C++	
I	COEFFICIENT	ORDER	EXPONENTS	V	Base
1	0.33333333333333E+00	0	0 0 0	3.3333333333e-01	0 0 0
2	-0.11111111111111E+00	1	1 0 0	-1.1111111111e-01	1 0 0
3	-0.11111111111111E+00	1	0 1 0	-1.1111111111e-01	0 1 0
4	0.37037037037037E-01	2	2 0 0	3.7037037037e-02	2 0 0
5	0.74074074074074E-01	2	1 1 0	7.4074074074e-02	1 1 0
6	0.37037037037037E-01	2	0 2 0	3.7037037037e-02	0 2 0
7	-0.12345679012346E-01	3	3 0 0	-1.2345679012e-02	3 0 0
8	-0.37037037037037E-01	3	2 1 0	-3.7037037037e-02	2 1 0
9	-0.37037037037037E-01	3	1 2 0	-3.7037037037e-02	1 2 0
10	-0.12345679012346E-01	3	0 3 0	-1.2345679012e-02	0 3 0
11	0.41152263374486E-02	4	4 0 0	4.1152263374e-03	4 0 0
12	0.16460905349794E-01	4	3 1 0	1.6460905350e-02	3 1 0
13	0.24691358024691E-01	4	2 2 0	2.4691358025e-02	2 2 0
14	0.16460905349794E-01	4	1 3 0	1.6460905350e-02	1 3 0
15	0.41152263374486E-02	4	0 4 0	4.1152263374e-03	0 4 0
	⋮			⋮	
56	0.56450292694768E-05	10	10 0 0	5.6450292695e-06	10 0 0
57	0.56450292694768E-04	10	9 1 0	5.6450292695e-05	9 1 0
58	0.25402631712645E-03	10	8 2 0	2.5402631713e-04	8 2 0
59	0.67740351233721E-03	10	7 3 0	6.7740351234e-04	7 3 0
60	0.11854561465901E-02	10	6 4 0	1.1854561466e-03	6 4 0
61	0.14225473759081E-02	10	5 5 0	1.4225473759e-03	5 5 0
62	0.11854561465901E-02	10	4 6 0	1.1854561466e-03	4 6 0
63	0.67740351233721E-03	10	3 7 0	6.7740351234e-04	3 7 0
64	0.25402631712645E-03	10	2 8 0	2.5402631713e-04	2 8 0
65	0.56450292694768E-04	10	1 9 0	5.6450292695e-05	1 9 0
66	0.56450292694768E-05	10	010 0	5.6450292695e-06	010 0

ACKNOWLEDGEMENT

The author would like to thank ALS accelerator physics group at LBNL for their support, Etienne Forest for integration with FPP/PTC, Frank Schmidt for integration with MAD-X.

REFERENCES

[1] Martin Berz. Differential algebraic description of beam dynamics to very high orders. Particle Accelerators, pages 109–124, 1989.

[2] Alex Chao. Notes on Topics in Accelerator Physics. 2002.

[3] MAD-X website, http://mad.web.cern.ch/mad/.

[4] E. Forest, Y. Nogiwa, F. schmidt. The FPP and PTC Libraries.