ISBN: 978-3-95450-223-3 ISSN: 2673-7000 doi:10.18429/JACoW-NAPAC2019-WEPLS14

A C++ TPSA/DA LIBRARY WITH PYTHON WRAPPER*

H. Zhang#, Y. Zhang
Jefferson Lab, Newport News, VA 23606, USA

Abstract

Truncated power series algebra (TPSA) or differential algebra (DA) is often used by accelerator physicists to generate a transfer map of a dynamic system. The map then can be used in dynamic analysis of the system or in particle tracking study. TPSA/DA can also be used in some fast algorithms, *e.g.* the fast multipole method, for collective effect simulation. This paper reports a new TPSA/DA library written in C++. This library is developed based on Dr. Lingyun Yang's TPSA code, which has been used in MAD-X and PTC. Compared with the original code, the updated version has the following changes: (1) The memory management has been revised to improve the efficiency; (2) A new data type of DA vector is defined and supported by most frequently used operators; (3) Support of inverse trigonometric functions and hyperbolic functions for the DA vector has been added; (4) the composition function is revised for better efficiency; (5) a python wrapper is provided. This library is open-source and the code is published on its github repository.

INTRODUCTION

The truncated power series algebra (TPSA) [1] or differential algebra (DA) [2] is a widely used tool in accelerator physics study. It is often used to generate a transfer map for a section of an accelerator. Once the map is created, it can be used directly to analyse the dynamic property of the section or perform map-based tracking for it. TPSA/DA has also been introduced into the fast multipole method (FMM) [3], which is a fast algorithm to calculate pairwise interactions between particles. The TPSA/DA method has been implemented in many accelerator simulation programs, such as MAD-X [4], COSY Infinity [5], *etc*. However, a library outside the simulation programs is not easily available for developers. The purpose of this work is to provide a stand-alone library with good efficiency for C++ and Python code developers.

THE NEW LIBRARY

The new library is composed of a C++ library that performs the TPSA/DA calculations and a Python wrapper. The C++ code is developed based on Dr. Lingjun Yang's TPSA code [6], which was included in previous versions of MAD-X. Now, our code, Yang's code, and documentations of our code are all available on our github repository[7]. Following the documentation, the users can compile the source code into a static or a shared library. They can also download the source files and use them directly in their projects. In our development, we tried to make minimal changes on the orignal code, but we had to revise or rewrite some functions for better efficiency or consistency.

We added some new features, which are listed as follows. 1. More math functions are supported. 2. Add a DA vector data type and defined the popular math operators for it, so that users can use a DA vector as simple as a normal number in calculations. 3. Revised the function for composition of two DA vectors for better performance. 4. Provide bunch processing of the composition function, since in accelerator studies one usually has to deal with multiple dimensional problems and the composition needs to be carried out on multiple DA vectors. These features will be demonstrated in the flowing sections.

Besides the new features added, one big change is made on the memory management (see Fig. 1). In Yang's code, the pointers to all the DA vectors are stored in a vector. Although the maximum number of DA vectors in the run is defined, the memory is not allocated. Each time when a new DA vector is needed, the program will search in the vector to find the first empty pointer and allocate the memory to it. Once the DA vector is out of scope, the memory is freed. This approach is good enough for a normal usage in accelerator study, which usually only needs a light DA calculation. However, in some cases an intense DA calculation may be needed. For example, when DA is used in FMM, to perform the calculation once, there may be millions of DA vectors created and

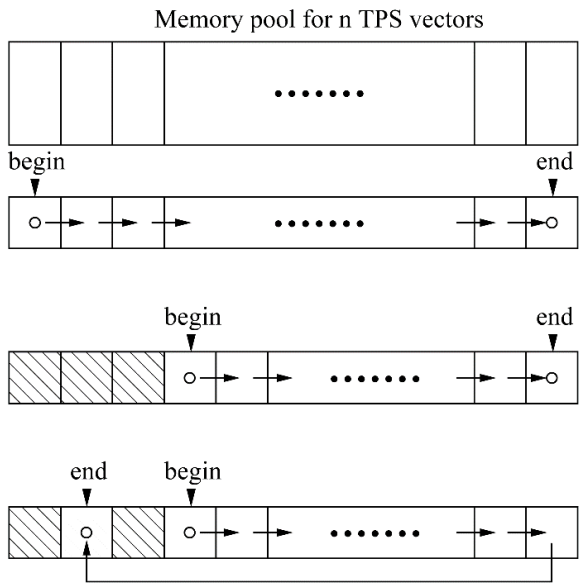

Figure 1: A schematic of memory management.

* Work supported by the Department of Energy, Laboratory Directed Research and Development Funding, under Contract No. DE-AC05-06OR23177.
#hezhang@jlab.org

ISBN: 978-3-95450-223-3 ISSN: 2673-7000 doi:10.18429/JACoW-NAPAC2019-WEPLS14

millions of DA operations carried out. In such a case, the search for address and the allocation/deallocation of memory for each DA vector will be a big burden. To solve this problem, we use a linked-list to manage the memory. The memory pool for all the DA vectors are allocated when the DA environment is initialized. The address for the slots, each for one DA vector, in the pool are saved in a linked-list. Whenever we need to create a new DA vector, we take out a slot from the beginning of the list. Whenever a DA vector goes out of the scope, its destructor will set all value in the slot to zero and put it back to the end of the list. The memory pool is managed simply by manipulating the two pointers that points to the beginning and the end of the list. In such a way, the repetitive searching and allocation/deallocation operations are avoided and a better efficiency can be achieved.

A python wrapper has also been developed based on pybind11 [8] for python 3. The source code and the documentation are available on its github repository [9]. Once compiled, a shared library will be generated, which can be imported into the python environment. It provides an access to the C++ library through python and supports almost all the functions in the C++ library. The calculations and memory management are still carried out by the C++ library.

VERIFICATION AND BENCHMARK

This library has been verified with COSY Infinity 9.0. As an example, the outputs of calculating asin(0.3+da[0]+2×da[1]), where da[0] and the da[1] are the first and the second DA base, up to the fourth order by both programs are presented in Fig. 2 and Fig. 3 respectively. Figure 2 shows the result by COSY Infinity, while figure 3 shows the result by our library called from python 3. The two programs give out exactly the same result. Here we want to note that for some functions, *e.g.* asin(), one may observe difference in the results at orders of 10^{-15} or 10^{-16}, which is due to the different algorithms used in the calculation and is considered acceptable in practice.

We also compared our lib with Yang's original lib to

```
 I   COEFFICIENT             ORDER EXPONENTS
 1   0.2914567944778671        0   0 0
 2   0.9174311926605504        1   1 0
 3    1.834862385321101        1   0 1
 4   -.2525039979799680        2   2 0
 5   -1.010015991919872        2   1 1
 6   -1.010015991919872        2   0 2
 7   -.1878979801481923        3   3 0
 8   -1.127387880889154        3   2 1
 9   -2.254775761778307        3   1 2
10   -1.503183841185538        3   0 3
11   0.1934000826207986        4   4 0
12    1.547200660966389        4   3 1
13    4.641601982899166        4   2 2
14    6.188802643865555        4   1 3
15    3.094401321932778        4   0 4
-------------------------------------
```

Figure 2: COSY Infinity 9.0 Output.

```
        V [69]                Base  [ 15 / 15 ]
-----------------------------------------------
 2.914567944778671e-01      0 0      0
 9.174311926605504e-01      1 0      1
 1.834862385321101e+00      0 1      2
-2.525039979799680e-01      2 0      3
-1.010015991919872e+00      1 1      4
-1.010015991919872e+00      0 2      5
-1.878979801481923e-01      3 0      6
-1.127387880889154e+00      2 1      7
-2.254775761778307e+00      1 2      8
-1.503183841185538e+00      0 3      9
 1.934000826207986e-01      4 0     10
 1.547200660966389e+00      3 1     11
 4.641601982899166e+00      2 2     12
 6.188802643865555e+00      1 3     13
 3.094401321932778e+00      0 4     14
```

Figure 3: Output by the new TPSA library.

see how much we gain in efficiency. The test runs presented here were carried out in a Windows 10 desktop with Xeon E5-1620 processor running at 3.60 GHz. Since in accelerator physics it is quite often to deal with a 3D dynamic system with six variables of positions and momentum, we first test substituting six DA vectors into one DA vector with six bases. Table 1 shows the computation cost by our lib and Yang's lib for orders of 2 to 10. With a moderate order 4 or 6, our lib is about 20 times faster. At order 10, the time cost of our lib is still only 1/3 of the original code. The second test we made is the bunch-processing composition of six DA vectors with another six DA vectors, which is a mimic of generating the transfer map of a 3D dynamic system with the maps of its two composites. Yang's code does not have the bunch processing feature, so we have to repeat the computation in the first test six times. We simply multiply six to the column in Table 1 to estimate of the time cost.

Table 1: Time (in seconds) for DA Vector Composition

Order	# of terms	This lib	Yang's lib
2	28	7.57×10^{-6}	6.25×10^{-6}
4	210	7.50×10^{-4}	1.44×10^{-2}
6	924	4.48×10^{-3}	8.39×10^{-2}
8	3003	0.99	2.55
10	8008	15.49	44.60

Table 2: Time (in seconds) for Bunch Processing of DA Vector Composition

Order	# of terms	This lib	Yang's lib
2	28	1.51×10^{-5}	3.75×10^{-5}
4	210	1.04×10^{-3}	8.64×10^{-2}
6	924	4.42×10^{-2}	5.03×10^{-1}
8	3003	1.05	15.3
10	8008	16.04	267.6

ISBN: 978-3-95450-223-3 ISSN: 2673-7000 doi:10.18429/JACoW-NAPAC2019-WEPLS14

The time cost of our lib is listed in Table 2. It is good to see for orders above six, the time cost only increases slightly when compared with that in the first test. It means the bunch processing does save a lot of time when we need to repeat the composition, which helps to improve the efficiency.

EXAMPLES

The sample codes in figure 4 and figure 5 show how to use the library in C++ and Python 3. More examples are available in the github repositories [7,9].

```cpp
#include "../include/da.h"
#include <cmath>
#include <iostream>
#include <vector>

int main() {

    unsigned int da_dim = 3;
    unsigned int da_order = 4;
    unsigned int n_vec = 400;

    //Initialize the DA domain.
    da_init(da_order, da_dim, n_vec);

    //Fundamental calculations of DA vectors.
    DAVector x = 1 + da[0] + 2*da[1] + 5*da[2];
    x.print();

    DAVector y = exp(x);
    y.print();

    //Substitute multiple DA vectors for bases at once.
    std::vector<DAVector> lv(2);
    lv.at(0) = sin(x);
    lv.at(1) = cos(x);
    std::vector<unsigned int> idx{0,1};
    DAVector z;
    da_substitute(y, idx, lv, z);
    z.print();

    //Composition of DA vectors with DA vectors.
    std::vector<DAVector> lx(3);
    std::vector<DAVector> ly(3);
    lx.at(0) = x;
    lx.at(1) = y;
    lx.at(2) = sinh(x);

    std::vector<DAVector> lu(3);
    lu.at(0) = sin(x);
    lu.at(1) = cos(x);
    lu.at(2) = tan(x);

    da_composition(lx, lu, ly);
    ly.at(0).print();
    ly.at(1).print();
    ly.at(2).print();

    return 0;
}
```

Figure 4: C++ sample code.

SUMMARY

A stand-alone library for TPSA/DA calculation has been developed and released. The library is based on Yang's TPSA code and has been improved for the following perspectives: 1. support for more mathematic functions, 2. more convenient usage in C++ and Python 3, and 3. better efficiency. The library has been verified with COSY Infinity 9.0. Benchmarking with the original code has shown significant improvement of efficiency for composition of DA vectors. Source code, documentation, and examples are available online [7,9].

```python
import tpsa

tpsa.da_init(4,2,100)
da = tpsa.base()

print("g(f(x)) with f(x) floats: ")
x = tpsa.DAVectorList()
x.assign(2)
x[0] = 1 + da[0] + 2*da[1]
x[1] = 0.5 + 3*da[0] + da[1]

y = [1.0, 2.0]

z = tpsa.da_composition(x, y)
print(z)

print("g(f(x)) with f(x) DA vectors: ")
y = tpsa.DAVectorList()
y.assign(2)
y[0] = 1 + 2*da[0] + da[1]
y[1] = 2 + da[0] + 0.5*da[1]

z = tpsa.DAVectorList()
z.assign(2)

tpsa.da_composition(x, y, z)
z[0].print()
z[1].print()
```

Figure 5: Python sample code.

ACKNOWLEDGEMENT

The authors would like to thank Dr. Lingyun Yang for providing his source code.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Nuclear Physics under contract DE-AC05-06OR23177.

REFERENCES

[1] A. Chao, "Lecture Notes on Topics in Accelerator Physics", *SLAC-PUB-9574*, (2002).

[2] M. Berz, "Modern Map Methods in Particle Beam Physics", *Academic Press*, (1999)

[3] H. Zhang, M. Berz, "The Fast Multipole Method in the Differential Algebra Framework", *Nuclear Instruments & Methods in Physics Research*, A, pp. 338-344 (2011)

[4] F. Schmidt and H. Grote, "MAD-X -- An Upgrade from MAD8", in *Proc. PAC'03*, Portland, OR, USA, May 2003, paper FPAG014, pp. 3497-3499.

[5] K. Makino, M. Berz, "COSY Infinity Version 9", *Nuclear Instruments & Methods in Physics Research,* A, pp. 346-350, (2006). doi:10.1016/j.nima.2005.11.109

[6] L. Yang, "Array Based Truncated Power Series Package", in *Proc. ICAP'09*, San Francisco, CA, USA, Aug.-Sep. 2009, paper THPSC059, pp. 371-373.

[7] https://github.com/zhanghe9704/tpsa

[8] https://github.com/pybind/pybind11

[9] https://github.com/zhanghe9704/tpsa-python