The Journal of Open Source Software

cppTPSA/pyTPSA: a C++/Python package for truncated power series algebra

He Zhang[*1]

1 Thomas Jefferson National Accelerator Facility, Newport News, VA 23606, USA

DOI: 10.21105/joss.03665

Software

- Review
- Repository
- Archive

Editor: Pending Editor

Submitted: 27 August 2021
Published: 31 August 2021

Summary

The truncated power series algebra (TPSA), also referred to as differential algebra (DA), is a well-established and widely used method in particle accelerator physics and astronomy. The most straightforward usage of TPSA/DA is to calculate the Taylor expansion of a given function at a specific point up to order n, based on which more sophisticated methods have been developed, *e.g.* symplectic tracking (Berz, 1991a), normal form analysis (Berz, 1991b), verified integration (Berz & Makino, 1998), global optimization (Makino & Berz, 2003), fast multipole method for pairwise interactions between particles (Zhang & Berz, 2011) *etc.* The cppTPSA package implements the TPSA/DA in C++11 and provides the developers a convenient library to build the advanced TPSA/DA-based method (Zhang, 2021a). A Python 3 library, pyTPSA, has also been developed based on the C++ lib and is available in a separate GitHub repository (Zhang, 2021b).

Background

In the following, we give a very brief introduction on TPSA/DA from a practical perspective of computation. Please refer to (Berz, 1999) and (Chao, 2002) for the complete theory with more details.

The fundamental concept in DA is the DA vector. To make the concept easier to understand, we can take a DA vector as the Taylor expansion of a function at a specific point.

Considering a function $f(\mathbf{x})$ and its Taylor expansion $f_{\mathrm{T}}(\mathbf{x}_0)$ at the point $\mathbf{x}_0$ up to the order n, we can define an equivalence relation between the Taylor expansion and the DA vector as follows

$$[f]_n = f_{\mathrm{T}}(\mathbf{x}_0) = \sum C_{n_1,n_2,\ldots,n_v} \cdot d_1^{n_1} \cdot \cdots \cdot d_v^{n_v},$$

where $\mathbf{x} = (x_1, x_2, \ldots, x_v)$, and $n \geq n_1 + n_2 + \cdots + n_v$. Here d_i is a special number and it represents a small variance in x_i. Generally one can define a DA vector by directly setting values to respective terms, without defining the function f. The addition and multiplication of two DA vectors can be defined straightforwardly. To add two DA vectors, we simply add the coefficients of the like terms. To multiply two DA vectors, we multiply each term in the first one with all the terms in the second one and combine like terms while ignoring all terms above order n. So given two DA vectors $[a]_n$ and $[b]_n$ and a scalar c, we have the following formulas:

*corresponding author

$$\begin{aligned} [a]_n + [b]_n &:= [a+b]_n, \\ c \cdot [a]_n &:= [c \cdot a]_n, \\ [a]_n \cdot [b]_n &:= [a \cdot b]_n, \end{aligned} \tag{1}$$

According to the fixed point theorem (Berz, 1999), the inverse of a DA vector that is not infinitely small can be calculated iteratively in a limit number of iterations.

The derivation operator ∂_v with respect to the v^{th} variable can be defined as

$$\partial_v [a]_n = \left[\frac{\partial}{\partial x_v} a \right]_{n-1},$$

which can be carried out term by term on $[a]_n$. The operator ∂_v satisfies the chain rule:

$$\partial_v([a] \cdot [b]) = [a] \cdot (\partial_v [b]) + (\partial_v [a]) \cdot [b].$$

The inverse operator ∂_v^{-1} can also be defined and carried out easily in a term-by-term manner. Once the fundamental operators are defined, the DA vector can be used in calculations just as a number.

Statement of need

The TPSA/DA methods for particle beam dynamic analysis was developed in 1980s. The tools are available in several popular programs for particle accelerator design and simulations, *e.g.* COSY Infinity 9 (Makino & Berz, 2006), MAD-X (Deniau et al., 2017), PTC (Forest et al., 2002), *etc.* In recent years, the use of TPSA/DA has been extended in other fields, which motivates building TPSA/DA libraries in popular programming languages. The existing programs are not convenient for developers in other fields. MAD-X is specifically developed for the accelerator design and cannot be used as a general programming language. Although COSY Infinity can be used as a general programming languages, it lacks some convenient programming features in a modern language, such as C++ or Python. It also does not have abundant libraries and a large supporting community. PTC includes a TPSA/DA library in Fortran 90 but it does not have a user-friendly interface. The TPSA/DA library in C++ is rare. DACE Massari & Wittig (2021) is one alternative. The DACE repository on GitHub had been created but no codes had been released when the author started to develop cppTPSA. Now DACE is available to the public. DACE provides the fundamental DA operations as well as some advanced algorithms based on DA but it has not supported the complex DA vectors, which is useful in the normal form analysis. To the best knowledge of the author, there is no other TPSA/DA library in Python 3.

Features

This library is composed of a C++ library that performs the TPSA/DA calculations and a Python wrapper (in a separate repository). Users can compile the source code into a static or shared library or generate a Python library for Python 3 environment. The readme file in each repository describes how to compile the C++ library and the Python library respectively.

The C++ library is based on Lingyun Yang's TPSA code, which was included in the previous versions of MAD-X. In the development, we try to make minimal changes on the original code,

but had to revise or rewrite some functions for better efficiency and/or consistency. One big change is the memory management. In Yang's code, the pointers to all the DA vectors are stored in a vector. Each time a new DA vector is needed, the program will search in the vector to find the first empty pointer and allocate the memory. Once the DA vector is out of scope, the memory is freed. In this library, we allocate the memory pool for all DA vectors (number defined by the user) in the very beginning when we initialize the DA environment. 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 this way, the repetitive searching and allocation/deallocation operations are avoided and better efficiency can be achieved.

Some new features have been added, which are listed in the following.

1. Add a DA vector data type and define the commonly used math operators for it, so that users can use a DA vector as simple as a normal number in calculations.
2. Support the complex DA vector defined by the C++ complex template.
3. More math functions are supported. (A list of the overloaded math functions can be found in the readme file of the repository.)
4. Add new functions that perform the composition of (complex) DA vectors, which can carry out multiple compositions in a call.
5. A Python wrapper is provided.

The following C++ code shows an example of a simple TPSA/DA calculation. After initializing an environment that can contain at most 400 three dimensional DA vectors up to the 4-th order, two DA vectors x1 and x2 and a complex DA vector y1 are defined, some trigonometric functions are performed on them, and the results are output to the screen.

```
#include "da.h"
da_init(4, 3, 400);
DAVector x1, x2;
x1 = da[0] + 2*da[1] + 3*da[2];
x2 = sin(x1);
x1 = cos(x1);
auto y1 = x1 + x2*1i;
std::cout<<x1<<x2<<std::endl;
std::cout<<sin(y1)<<std::endl;
```

A Python example doing the same calculation is presented as follows.

```
import tpsa
tpsa.da_init(4, 3, 400)
da = tpsa.base()
x1 = da[0] + 2*da[1] + 3*da[2]
x2 = tpsa.sin(x1)
x1 = tpsa.cos(x1)
y1 = tpsa.complex(x1, x2)
print(x1)
print(x2)
print(tpsa.sin(y1))
```

More examples can be found in the respective repository.

Verification

This library has been verified with COSY Infinity 9.0. As an example, the outputs of calculating sin (0.3+da[0]+2×da[1]) up to the fourth order by both programs are presented in Figure 1 and Figure 2 respectively. Figure 1 shows the result by COSY Infinity, while Figure 2 shows the result by cppTPSA. The two programs give out exactly the same result. Here we want to note (1) for some functions, *e.g.* arcsin, 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 and (2) the sequence of the terms may be different when outputting a DA vector from cppTPSA and from COSY Infinity.

```
 I  COEFFICIENT            ORDER EXPONENTS
 1  0.2955202066613395       0   0 0
 2  0.9553364891256060       1   1 0
 3   1.910672978251212       1   0 1
 4  -.1477601033306698       2   2 0
 5  -.5910404133226791       2   1 1
 6  -.5910404133226791       2   0 2
 7  -.1592227481876010       3   3 0
 8  -.9553364891256060       3   2 1
 9  -1.910672978251212       3   1 2
10  -1.273781985500808       3   0 3
11  0.1231334194422248E-01   4   4 0
12  0.9850673555377985E-01   4   3 1
13  0.2955202066613395       4   2 2
14  0.3940269422151194       4   1 3
15  0.1970134711075597       4   0 4
 -----------------------------------
```

Figure 1: COSY Infinity 9.0 output.

```
 I          V [36]              Base  [ 15 / 15 ]
-------------------------------------------------
 1   2.955202066613395e-01     0 0     0
 2   9.553364891256060e-01     1 0     1
 3   1.910672978251212e+00     0 1     2
 4  -1.477601033306698e-01     2 0     3
 5  -5.910404133226791e-01     1 1     4
 6  -5.910404133226791e-01     0 2     5
 7  -1.592227481876010e-01     3 0     6
 8  -9.553364891256060e-01     2 1     7
 9  -1.910672978251212e+00     1 2     8
10  -1.273781985500808e+00     0 3     9
11   1.231334194422248e-02     4 0    10
12   9.850673555377985e-02     3 1    11
13   2.955202066613395e-01     2 2    12
14   3.940269422151194e-01     1 3    13
15   1.970134711075597e-01     0 4    14
```

Figure 2: cppTPSA output.

Acknowledgements

The author 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

Berz, M. (1991a). Symplectic tracking in circular accelerators with high order maps. *Nonlinear Problems in Future Particle Accelerators*, 288.

Berz, M. (1991b). *High-order computation and normal form analysis of repetitive systems, in: M. Month (Ed), physics of particle accelerators* (Vol. 249, p. 456). American Institute of Physics. https://doi.org/10.1063/1.41975

Berz, M. (1999). *Modern map methods in particle beam physics*. Academic Press. https://doi.org/10.1016/s1076-5670(08)x7018-1

Berz, M., & Makino, K. (1998). Verified integration of ODEs and flows using differential algebraic methods on high-order Taylor models. *Reliable Computing*, *4*(4), 361–369. https://doi.org/10.1023/A:1024467732637

Chao, A. W. (2002). *Lecture notes on topics in accelerator physics*. Stanford Linear Accelerator Center, Menlo Park, CA (US). https://doi.org/10.2172/812598

Deniau, L., Grote, H., Roy, G., & Schmidt, F. (2017). *The MAD-X program, version 5.07.00, user's reference manual*. CERN.

Forest, E., Schmidt, F., & McIntosh, E. (2002). Introduction to the polymorphic tracking code. *KEK Report*, *3*, 2002.

Makino, K., & Berz, M. (2003). Verified global optimization with Taylor model methods. *International Journal of Computer Research*, *12,2*, 245–252.

Makino, K., & Berz, M. (2006). COSY INFINITY version 9. *Nuclear Instruments and Methods*, *558*, 346–350. https://doi.org/10.1016/j.nima.2005.11.109

Massari, M., Di Lizia, P., Cavenago, F., & Wittig, A. (2018). Differential algebra software library with automatic code generation for space embedded applications. In *2018 AIAA information systems-AIAA infotech@ aerospace* (p. 0398). https://doi.org/10.2514/6.2018-0398

Massari, M., & Wittig, A. (2021). DACE: The differential algebra computational toolbox. In *GitHub repository*. GitHub. https://github.com/dacelib/dace

Zhang, H. (2021a). cppTPSA: A C++ TPSA lib. In *GitHub repository*. GitHub. https://github.com/zhanghe9704/tpsa

Zhang, H. (2021b). pyTPSA: A Python TPSA lib. In *GitHub repository*. GitHub. https://github.com/zhanghe9704/tpsa-python

Zhang, H., & Berz, M. (2011). The fast multipole method in the differential algebra framework. *Nuclear Instruments and Methods A 645*, 338–344. https://doi.org/10.1016/j.nima.2011.01.053