SDA: a symbolic differential algebra package in C++

He Zhang [1¶]

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

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970

Published: unpublished

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. It is typically used to generate high-order map of a nonlinear dynamical system. The DA calculation is based on the DA vector that can be practically viewed as the Taylor expansion of a given function at a specific point up to a predetermined order n. Besides map generation, the DA/TPSA technique can also be used in symplectic tracking, normal form analysis, verified integration, global optimization, fast multipole method, etc. This package is the first one to combine the DA/TPSA algorithm with symbolic calculation, and it allows users to carry out symbolic DA-based calculations. The coefficient of any element in a symbolic DA vector is an explicit expression of preselected symbols, in lieu of a number in a numerical DA vector. This capability makes it possible to trace the contribution of initial conditions to the final result in a DA calculation process and improve the efficiency of repeated DA calculations. It also provides a method to obtain explicit expressions for the higher-order derivatives of an arbitrary function.

Background

In the following, we give a very brief introduction to (symbolic) TPSA/DA from a practical computational perspective. 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 this concept easier to understand, we can consider 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 order n, we can define the equivalence relation between the Taylor expansion and the DA vector as follows

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

where $\mathbf{x} = (x_1, x_2, \dots, x_v)$, and $n \geq n_1 + n_2 + \dots + n_v$. Here d_i is a special number: 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. We then 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 formulae:

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

According to the fixed point theorem (Berz, 1999), the inverse of a DA vector that is not infinitely small can be calculated in a finite 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 on a term by term basis 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 applied easily on a term-by-term basis. Once the fundamental operators are defined, the DA vector can be used in calculations just as a number.

The symbolic DA/TPSA combines the DA/TPSA with symbolic calculation. Exactly the same DA/TPSA algorithms are implemented on symbols rather than numbers. Compared to the numerical DA/TPSA, the only difference is any coefficient of a Symbolic DA (SDA) vector is an explicit expression of the symbols in lieu of a number.

Statement of need

DA/TPSA methods for particle beam dynamic analysis were developed in the 1980s. Since then, their application has been gradually extended to other fields. DA provides powerful analyzing tools, *i.e.*, map generation, symplectic tracking (Berz, 1991a), and normal form analysis (Berz, 1991b), for a dynamic system. In addition, it can also be used in verified integration (Berz & Makino, 1998), global optimization (Makino & Berz, 2005), and fast multipole method for pairwise interactions between particles (H. Zhang & Berz, 2011). DA tools are available in several popular programs for particle accelerator design and simulations, such as COSY Infinity (Makino & Berz, 2006), MAD-X (Deniau et al., 2017; Grote & Schmidt, 2003), and PTC (Forest et al., 2002). Stand-alone DA/TPSA libraries include DACE (Massari et al., 2018; Massari & Wittig, 2021) and cppTPSA/pyTPSA (He Zhang, 2024). All of them only perform numerical DA calculation. To the best of our knowledge, this library is the first and currently the only one that can carry out symbolic DA calculations.

We developed SDA to improve the efficiency of repetitive DA processes. In some scenarios, we need to implement a complicated DA method repetitively on the same object with different initial conditions, *i.e.*, different positions and velocities of particles. Some DA calculations are relatively slow. For example, calculating the inverse of a DA vector includes iterations and the number of iterations is determined by the order of the DA vector. This limits the use of DA in high-efficiency simulations. Using numerical DA, we lose track of the initial conditions in the calculation and have to repeat the same DA calculation for each case. Using symbolic DA, we only need to carry out the DA calculation once to obtain the final result as explicit expressions of the initial conditions. We then use those expressions on all the different initial conditions. By reducing the calculation on DA vectors to calculation on numbers, we can significantly enhance the efficiency.

SDA can also be used in higher-order derivative calculation. DA calculates the Taylor expansion of an arbitrary function. The partial derivatives of that function can be extracted from the coefficients of its Taylor expansion. Numerical DA calculates all the derivatives up to a predetermined order at a specific point. It is not possible to calculate a specific derivative without calculating the others. SDA provides explicit expressions for all the derivatives, allowing to calculate any specific derivative at any point. Calculating the derivatives using the expressions from SDA is usually faster. If only a few derivatives are needed, SDA can save even more time by avoiding redundant calculations on unnecessary derivatives.

Features

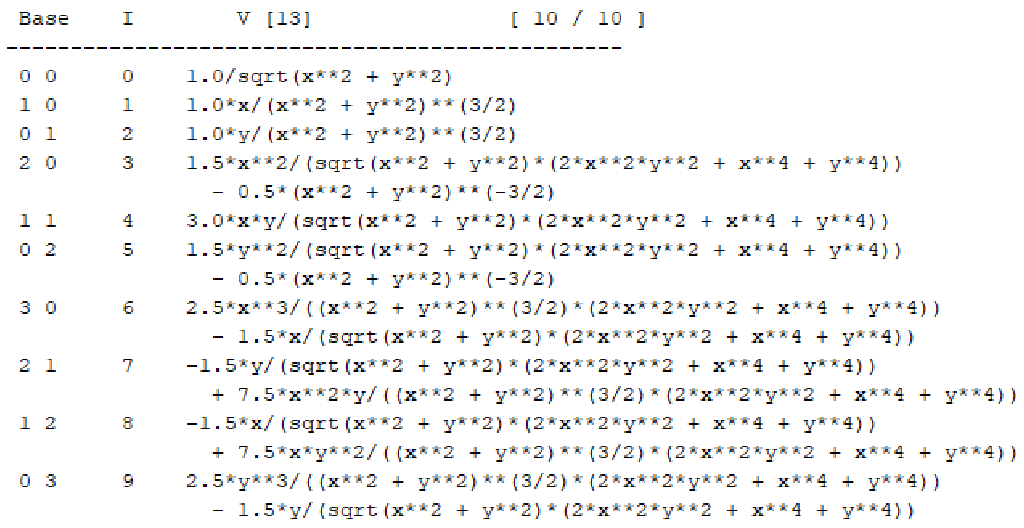
This library performs symbolic DA/TPSA calculations. It is based on the numerical DA library, cppTPSA (He Zhang, 2024). All the DA calculations are carried out using exactly the same algorithms in cppTPSA, except they are implemented on symbols by employing the SymEngine library (Fernando et al., 2024), rather than numbers. Users can compile the source code into a static or shared library and install it on their system. The main features of this library are listed as follows.

1. An SDA vector data type is defined and the commonly used math operators are overloaded for it, so that users can use an SDA vector as simple as a normal number in calculations.
2. Common mathematical functions are overloaded for the SDA vector data type. (A list of the functions supported can be found in the README file of the repository)
3. Support the composition of SDA vectors.
4. Support derivation and inverse derivation of a SDA vector.
5. Obtain the explicit expression of a partial derivative from a Taylor Expansion, *i.e.*, an SDA vector.
6. Obtain the callable function on the symbols from an SDA.
7. Obtain the numerical DA from an SDA by assigning values to all the symbols.

The following C++ code shows an example of a simple SDA calculation. This code calculates the SDA vector of $1/\sqrt{x^2+y^2}$ up to the third order. First we initialize a memory pool that can contain 400 two-dimensional DA vectors up to the third order, which is much larger than what is needed for this calculation. Then we define the symbols, x and y. Finally we calculate the SDA vector and print it out. The value of the SDA vector is shown in Figure 1. The first column shows the orders of the bases, the second column displays the index of each term in the SDA vector, and the last column lists the coefficients of each term. Each coefficient of the SDA vector is an explicit expression in x and y. The number 13 in "V [13]" means the SDA vector is saved in the 13th slot in the memory pool. "[10 / 10]" means the SDA has 10 non-zero elements and the total number of elements is also 10. This indicates a full vector. For a sparse vector, the first number is smaller than the second number because some elements are zero and not shown. More examples can be found in the repository.

```cpp
#include <iostream>
#include <sda.h>
typedef SymbDA::DAVector SDA;
using SymEngine::Expression;

int main() {
    int order{3}, dim{2}, pool{400};
    SymbDA::da_init(order, dim, pool);
    auto& sda = SymbDA::da;
    Expression sx("x"), sy("y");
    SDA f = 1/sqrt((sx+sda[0])*(sx+sda[0]) + (sy+sda[1])*(sy+sda[1]));
    std::cout<<f;
}
```

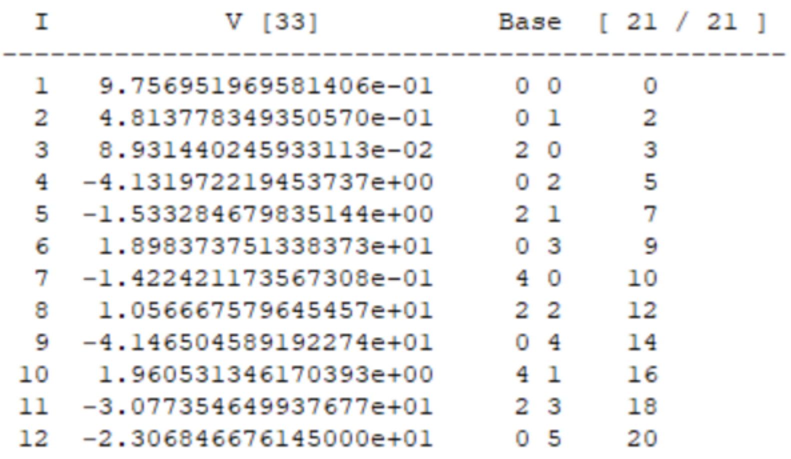


```
Base      I        V [13]                  [ 10 / 10 ]
------------------------------------------------
0 0       0     1.0/sqrt(x**2 + y**2)
1 0       1     1.0*x/(x**2 + y**2)**(3/2)
0 1       2     1.0*y/(x**2 + y**2)**(3/2)
2 0       3     1.5*x**2/(sqrt(x**2 + y**2)*(2*x**2*y**2 + x**4 + y**4))
                  - 0.5*(x**2 + y**2)**(-3/2)
1 1       4     3.0*x*y/(sqrt(x**2 + y**2)*(2*x**2*y**2 + x**4 + y**4))
0 2       5     1.5*y**2/(sqrt(x**2 + y**2)*(2*x**2*y**2 + x**4 + y**4))
                  - 0.5*(x**2 + y**2)**(-3/2)
3 0       6     2.5*x**3/((x**2 + y**2)**(3/2)*(2*x**2*y**2 + x**4 + y**4))
                  - 1.5*x/(sqrt(x**2 + y**2)*(2*x**2*y**2 + x**4 + y**4))
2 1       7     -1.5*y/(sqrt(x**2 + y**2)*(2*x**2*y**2 + x**4 + y**4))
                  + 7.5*x**2*y/((x**2 + y**2)**(3/2)*(2*x**2*y**2 + x**4 + y**4))
1 2       8     -1.5*x/(sqrt(x**2 + y**2)*(2*x**2*y**2 + x**4 + y**4))
                  + 7.5*x*y**2/((x**2 + y**2)**(3/2)*(2*x**2*y**2 + x**4 + y**4))
0 3       9     2.5*y**3/((x**2 + y**2)**(3/2)*(2*x**2*y**2 + x**4 + y**4))
                  - 1.5*y/(sqrt(x**2 + y**2)*(2*x**2*y**2 + x**4 + y**4))
```

Figure 1: Example code output.

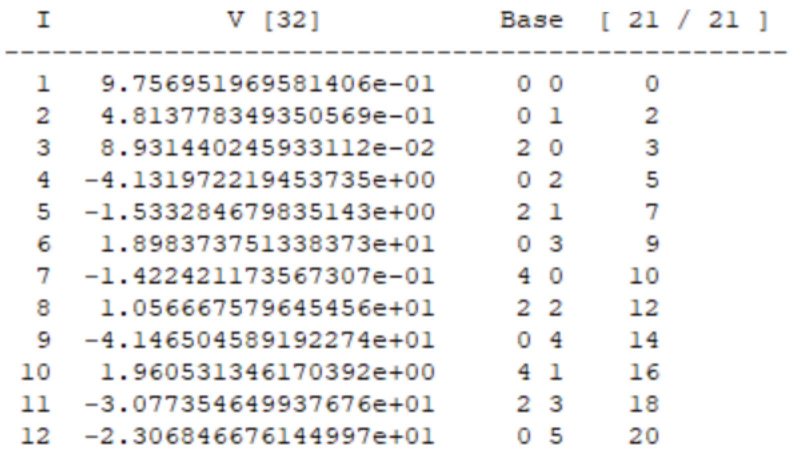
Verification

This library has been verified with the numerical DA library, cppTPSA, by assigning values to all the symbols in an SDA vector, which results in a numerical DA vector. This vector is then checked against direct calculation using cppTPSA. For example, we calculate an SDA vector $v_1 = \exp(x + d_1^2 + y \cdot d_2)$, in which x and y are symbols and d_1, d_2 are the bases of the SDA vector. By setting $x = 1.5926$ and $y = 5.3897$, we obtain the numerical DA vector shown in Figure 2 up to the fifth order. It should agree with the numerical DA $v_2 = \exp(1.5926 + d_1^2 + 5.3897 \cdot d_2)$, as shown in Figure 3. We calculate the relative error for each non-zero coefficient. This procedure is repeated 1,000 times for all the math functions in the SDA lib with randomly generated x and y. For the three inverse trigonometric functions, asin, acos, and atan, x ranges in (0, 0.5) and y ranges in (0,1), while for all the other functions, both x and y range in (0,10). In all the cases, the absolute values of the relative errors are less than 1×10^{-15}.



```
 I           V [33]             Base  [ 21 / 21 ]
------------------------------------------------
 1   9.756951969581406e-01     0 0      0
 2   4.813778349350570e-01     0 1      2
 3   8.931440245933113e-02     2 0      3
 4  -4.131972219453737e+00     0 2      5
 5  -1.533284679835144e+00     2 1      7
 6   1.898373751338373e+01     0 3      9
 7  -1.422421173567308e-01     4 0     10
 8   1.056667579645457e+01     2 2     12
 9  -4.146504589192274e+01     0 4     14
10   1.960531346170393e+00     4 1     16
11  -3.077354649937677e+01     2 3     18
12  -2.306846676145000e+01     0 5     20
```

Figure 2: SDA output.



```
   I            V [32]            Base   [ 21 / 21 ]
 ------------------------------------------------------
   1    9.756951969581406e-01     0 0       0
   2    4.813778349350569e-01     0 1       2
   3    8.931440245933112e-02     2 0       3
   4   -4.131972219453735e+00     0 2       5
   5   -1.533284679835143e+00     2 1       7
   6    1.898373751338373e+01     0 3       9
   7   -1.422421173567307e-01     4 0      10
   8    1.056667579645456e+01     2 2      12
   9   -4.146504589192274e+01     0 4      14
  10    1.960531346170392e+00     4 1      16
  11   -3.077354649937676e+01     2 3      18
  12   -2.306846676144997e+01     0 5      20
```

Figure 3: cppTPSA output.

Acknowledgements

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*, 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., Skowronski, P., Roy, G., & others. (2017). MAD-X: Methodical accelerator design. In *GitHub repository*. GitHub. https://doi.org/10.5281/zenodo.7900975

Fernando, I., Čertík, O., & others. (2024). *SymEngine*. https://github.com/symengine/symengine

Forest, E., Schmidt, F., & McIntosh, E. (2002). Introduction to the polymorphic tracking code. *KEK Report*, *3*, 2002. https://inspirehep.net/literature/591979

Grote, H., & Schmidt, F. (2003). MAD-X-an upgrade from MAD8. *Proceedings of the 2003 Particle Accelerator Conference*, *5*, 3497–3499. https://doi.org/10.1109/PAC.2003.1289960

Makino, K., & Berz, M. (2005). Verified global optimization with Taylor model based range bounders. *Transactions on Computers*, *11*(4), 1611–1618. https://www.bmtdynamics.org/pub/papers/GOM05/GOM05.pdf

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, He. (2024). cppTPSA/pyTPSA: A C++/Python package for truncated power series algebra. *Journal of Open Source Software*, *9*(94), 4818. https://doi.org/10.21105/joss.04818

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