ISBN: 978-3-95450-261-5 ISSN: 2673-7000 doi: 10.18429/JACoW-NAPAC2025-MOP044

GTPSA.jl : A SciBmad INTERFACE TO THE GENERALISED TRUNCATED POWER SERIES ALGEBRA LIBRARY*

M. G. Signorelli†, G. H. Hoffstaetter[1], D. Sagan, CLASSE, Cornell University, Ithaca, NY, USA
L. Deniau, CERN, Geneva, Switzerland
O. Beznosov, LANL, Los Alamos, NM, USA
[1]also at Brookhaven National Laboratory, Upton, NY, USA

Abstract

A full-featured interface package to the Generalised Truncated Power Series Algebra (GTPSA) library in MAD-NG has been implemented in the Julia programming language. GTPSA performs fast Taylor-mode automatic differentiation (AD) of functions to arbitrary orders in the specified variables and parameters. In particular, GTPSA excels at computing derivatives to high orders (>1) and high numbers of variables/parameters, making it an extremely powerful tool for use in optimization and in computing parametric Taylor maps. This Julia interface offers another simple way of using the GTPSA library, and will be used extensively in the SciBmad accelerator physics software ecosystem. The interface can also be easily called from Python, via the `juliacall` package. In this paper, we showcase features implemented in the interface package including performance enhancements, and present an example of integrating a GTPSA map using polymorphic integrators already implemented in Julia.

INTRODUCTION

To understand and analyze the beam dynamics in a particle accelerator, we express the motion through the accelerator as a Taylor series expanded around the "reference trajectory" - the closed orbit in a periodic ring, or the "ideal" particle in an open beamline - and optionally the parameters of the accelerator (quadrupole strengths, lengths, etc). This Taylor map is referred to as a (parametric) differential-algebraic (DA) map. Once a DA map is computed, canonical perturbation theory can then be applied to extract *all* information on the dynamics, up to the truncation order of the series; this includes the Twiss parameters, amplitude-dependent tunes, chromatic beta beat, so-called second order dispersion, resonance driving terms (RDTs), etc., and how all of these quantities depend on the parameters.

A code that can compute high order Taylor maps is a necessary tool in the design of particle accelerators. However, accurately computing all the higher order cross-derivatives is very nontrivial; finite differencing methods for example can be highly sensitive to the chosen step size for each individual variable, can suffer round-off errors that grow with the order of the partial derivative, and can include higher-order errors in the low-order derivatives, proportional to the step size. Furthermore, as the number of variables and truncation order increases, the number of partial derivatives to compute and store grows exponentially.

Automatic differentiation (AD) is a method which computes derivatives without any approximation - being exact to numerical precision - by relying on the fact that the chain rule can be repeatedly applied to a sequence of elementary arithmetic operations and math functions [1]. A "differentiable" code is one whose derivatives can be computed using AD. Truncated Power Series Algebra (TPSA) is a method of AD that is highly-optimized for computing high order derivatives, including potentially many variables and parameters; this makes TPSA ideal for accelerator physics.

Therefore, as has been recognized since the early 1990s [2–4], accelerator physics simulation codes must be differentiable via a TPSA package to properly compute and analyze DA maps. The Generalised Truncated Power Series Algebra (GTPSA) library, written in C, has been shown to outperform many well-known TPSA libraries [5]. GTPSA also uniquely allows the user to specify custom truncation orders for individual variables and parameters, enabling better flexibility without a performance cost. To harness the power of GTPSA for use both in the new SciBmad accelerator physics code and in the broader optimization and AD community, we have implemented a full featured interface package in the Julia programming language. GTPSA bindings have also been implemented in two open-source, widely-used Julia packages: `DifferentiationInterface.jl` [6, 7] which enables GTPSA to be used generically alongside any other Julia AD package, and `DiffEqBase.jl` which enables integration of GTPSA maps using any of the differentiable integrators already available in `DifferentialEquations.jl` [8]. And, with the Python package `juliacall` [9], `GTPSA.jl` can be easily called from Python. In this paper, we present all such features implemented in `GTPSA.jl`.

TRUNCATED POWER SERIES ALGEBRA

The most natural AD implementation utilizes *forward-accumulation*, which works by defining a new number type called a *dual*. The dual number stores two numbers: one which acts like a regular number, and one which is the exact partial derivative of any previously evaluated function. All elementary arithmetic operators and functions are properly overloaded for the dual to apply the chain rule and compute this derivative exactly. In this case, the chain rule is traversed from the input to the output, hence the name "forward".

In a fully polymorphic code, higher order derivatives can be computed by nesting dual numbers. However, this can

* Work supported by Brookhaven Science Associates, LLC under Contract No. DE-SC0012704 with the U.S. Department of Energy.
† mgs255@cornell.edu

ISBN: 978-3-95450-261-5 ISSN: 2673-7000 doi: 10.18429/JACoW-NAPAC2025-MOP044

be slow and memory-intensive due to the cost of repeated evaluation of the partial derivatives for higher orders, recursive data structures storing the partials, and the redundant storage of the cross derivatives. TPSA addresses these issues by using the Taylor series directly for higher order partial derivatives, storing the partials in a flattened array in dynamically-allocated memory, and storing no redundant cross terms. The GTPSA library specifically includes many optimizations beyond this, discussed in detail in [5, 10].

Using Einstein notation for the variable indices, and letting n specify the order up to a maximum truncation order MO, we can express a function f expanded around $\vec{a}$ as a truncated power series (TPS),

$$f(\vec{x}) \approx f(\vec{a}) + \sum_{n=1}^{\mathrm{MO}} \frac{1}{n!} \left. \frac{\partial^n f}{\partial x_{i_1} \dots \partial x_{i_n}} \right|_{\vec{a}} \Delta x_{i_1} \dots \Delta x_{i_n} \,. \quad (1)$$

A TPS number type stores and propagates all of the monomial coefficients up to the truncation order.

BASIC USAGE

The Julia programming language is a just-in-time (JIT) compiled, high-level, high performance computing language that, by leveraging multiple dispatch and a powerful type system, enables universally polymorphic code without any performance losses. This makes Julia ideal for writing a simulation code that by default works with both regular numbers and dual numbers/truncated power series.

First the number of variables, parameters, and truncation order(s) must be defined in a `Descriptor`. A vector of the variables Δx_i's as TPSs can then be obtained using `@vars`. Taylor maps can be computed simply by using these in place of regular numbers. E.g., to compute the multivariate Taylor expansion of the $\mathbb{R}^2 \to \mathbb{C}^1$ function $f(\vec{x}) = \cos x_1 + \mathrm{i} \sin x_2$ around $\vec{a} = (-\pi/2, \pi/2)$ to 6th order:

```
using GTPSA
d6 = Descriptor(2, 6)
Δx = @vars(d6)
f(x) = cos(x[1]) + im*sin(x[2])
ft = f([-π/2, π/2] + Δx)

# Print imaginary part:
println(imag(ft)) # Output in following block
```

```
TPS64{Descriptor(NV=2, MO=6)}:
 Coefficient                Order   Exponent
  1.0000000000000000e+00      0      0   0
 -5.0000000000000000e-01      2      0   2
  4.1666666666666664e-02      4      0   4
 -1.3888888888888887e-03      6      0   6
```

`GTPSA.jl` includes two types: `TPS64` and `ComplexTPS64`, which respectively represent 64-bit floats and complex numbers as two 64-bit floats. Promotion from regular numbers to TPSs, and from real to complex, is handled automatically. To define a `Descriptor` for custom truncation orders of the variables/parameters, see the documentation [11].

`GTPSA.jl` includes routines for TPS evaluation, composition, inversion, and translation. The syntax is natural:

```
gt = [ft, -im*ft] # New TPS vector function

gt([1, 2])   # Evaluation for Δx = [1,2]
ht = ft ∘ gt # Composition, equivalently ft(gt)
inv(ht)      # Inversion of Taylor map
translate(gt,[3,4]) # Translate expansion point
```

`GTPSA.jl` provides the following methods for getting/setting individual monomial coefficients in a TPS :

```
# 1) Flat index, sorted by order:
ft[0] # Scalar part
ft[1] # Coefficient of Δx₁
ft[2] # Coefficient of Δx₂

# 2) Monomial:
ft[[0,0]] # Scalar part
ft[[1,0]] # Coefficient of Δx₁
ft[[1,2]] # Coefficient of (Δx₁)(Δx₂)²

# 3) Sparse monomial:
ft[[1=>2, 2=>3]] # Coefficient of (Δx₁)²(Δx₂)³
```

For any of these methods, a colon (:) can be included to "slice" the TPS. For example, with monomial getting, `ft[[1,:]]` will return a new TPS including only the monomial coefficients from `ft` with Δx_1 at order 1, and all orders in Δx_2. This makes it easy to extract a particular polynomial from within the TPS. To extract a polynomial of a given order(s), the `cutord` and `getord` functions can be used. Finally, `GTPSA.jl` provides `cycle!` to easily cycle through all nonzero coefficients; this is particularly useful for normal form algorithms, and is used extensively in the SciBmad normal form package `NonlinearNormalForm.jl` [12].

`GTPSA.jl` provides `jacobian`, `gradient`, and `hessian` functions to extract the corresponding derivatives. However, if you are only interested in these quantities, then we strongly recommend using `GTPSA.jl` with the popular, open-source `DifferentiationInterface.jl` package in Julia [6, 7]. This package enables users to plug-and-play different Julia AD backends with the same interface.

INTEGRATING A GTPSA MAP

`DifferentialEquations.jl` is an open source, massive suite of integrators [8]. It is impossible to highlight all of its features here, but of most interest to the accelerator physics community are probably its symplectic, Runge Kutta, Magnus, and stochastic integrators.

To enable integration of GTPSA maps using these integrators, bindings have been added to the base package

ISBN: 978-3-95450-261-5 ISSN: 2673-7000 doi: 10.18429/JACoW-NAPAC2025-MOP044

`DiffEqBase.jl`. All one needs to do now is use TPSs as the initial conditions, and the result will be a GTPSA map. As an example, we compute a parametric 2nd order transport map through a paraxial quadrupole, where the strength is varied as a parameter. In this case, we define a `Descriptor` with 6 variables to 2nd order, and 1 parameter to 1st order. We use `@params` to get a vector of the parameters Δk_i's. The 6th order Yoshida drift-kick-drift integrator [13] in `OrdinaryDiffEq.jl` is used:

```
using GTPSA, OrdinaryDiffEq
# 2nd order in 6 variables, 1st order parameter
d = Descriptor(6, 2, 1, 1)
Δx = @vars(d)
Δk = @params(d)

# Define EOM
pdot(p,q,K1,s) = -[K1*q[1], -K1*q[2], 0]
qdot(p,q,K1,s) = [p[1]/(1+p[3]), p[2]/(1+p[3]),
                  1/2*(p[1]^2+p[2]^2)/(1+p[3])^2]

# Quad strength (as TPS) and length:
K1 = 0.36 + Δk[1] # TPS
L = 0.5

#Initial conditions as TPSs:
p0 = Δx[1:3]; q0 = Δx[4:6]

prob = DynamicalODEProblem(
        pdot, qdot, p0, q0, (0, L), K1)

# sol contains the GTPSA map at each step
sol = solve(prob, Yoshida6(), dt=0.01)
```

CALLING FROM PYTHON

The `juliacall` package in Python makes it very easy to call any Julia code within Python [9]. For example:

```
from juliacall import Main as jl
jl.seval('import Pkg; Pkg.add("GTPSA")') # once
jl.seval("using GTPSA")
d6 = jl.Descriptor(2, 6)
x = jl.vars(d6)
ft = jl.cos(x[0]) + 1j*jl.sin(x[1])
```

@FastGTPSA /@FastGTPSA! MACROS

Because TPSs are mutable, every operation with a TPS must dynamically-allocate a new TPS. This includes the temporaries computed during expression evaluation; for example if a, b, c, and d are TPSs, then the expression $(a + b) * (c + d)$ will allocate 3 TPSs: one for each sum, then one for the final product. Because Julia has garbage collection, the temporaries will be deallocated when Julia decides to. Nonetheless, dynamically-allocated memory and garbage collection can hinder performance.

To address this issue, a thread-safe buffer of TPSs for use by temporaries is preallocated in the `Descriptor`. In Julia the assignment operator cannot be overloaded. Therefore, to "tell" the code to use the pre-allocated buffer for temporaries, `GTPSA.jl` provides the `@FastGTPSA` macro which can be prepended to any expression, or block of code, that may include TPSs. By taking advantage of multiple dispatch, the macro is also *completely transparent to all other types*, so it can be prepended to any existing code and *maintain universal polymorphism*. For example:

```
foo(a, b, c, d) = @FastGTPSA (a + b)*(c + d)
a, b, c, d = @vars(Descriptor(4, 2))
r = foo(a, b, c, d) # Only 1 TPS allocated
r = foo(1, 2, 3, 4) # Transparent to non-TPSs
```

`GTPSA.jl` also provides the `@FastGTPSA!` macro, which is similarly transparent to all non-TPS types and can be used to write the result to pre-allocated TPSs for *zero* allocations:

```
r = zero(a) # Pre-allocates result
@FastGTPSA! r = (a + b)*(c + d) # Zero allocs
```

In our examples and tracking codes, we have seen a roughly 2x speedup by using these macros with TPSs.

STATIC VS. DYNAMIC RESOLUTION

`GTPSA.jl` allows for multiple `Descriptors` defined in the same program (though mixing TPSs with different `Descriptors` is not allowed). This brings the question of whether a `Descriptor` should be stored explicitly in the TPS type itself (statically) or resolved from the TPS at runtime (dynamically). `GTPSA.jl` allows users to decide between either; `@vars`/`@params` have the keyword argument `dynamic`, with default `false`. For dynamic resolution, if the `Descriptor` cannot be inferred at runtime, it can be specified with the `use` argument. `use` defaults to `GTPSA.desc_current`, a global variable always set to the last defined `Descriptor`:

```
d = Descriptor(1, 2)
# Promote 1.23 to a TPS:
x = TPS64{d}(1.23) # Static, as type parameter
y = TPS64{GTPSA.Dynamic}(1.23, use=d) # Dynamic
```

CONCLUSIONS

Polymorphic codes that are differentiable with TPSA are essential to designing particle accelerators. The GTPSA library, written in C, has been shown to provide far better performance and flexibility than other well-known TPSA libraries. This full-featured interface package `GTPSA.jl` , written in Julia, brings the full-power of GTPSA to a much wider community, both inside and outside of accelerator physics. Julia-specific performance optimizations and features have been implemented, as well as bindings to widely-used Julia packages for easy composability – this includes a generic autodifferentiation interface and a massive suite of polymorphic integrators. The new SciBmad accelerator physics code will use `GTPSA.jl` extensively.

ISBN: 978-3-95450-261-5 ISSN: 2673-7000 doi: 10.18429/JACoW-NAPAC2025-MOP044

REFERENCES

[1] Wikipedia, "Automatic differentiation — Wikipedia, the free encyclopedia", http://en.wikipedia.org/w/index.php?title=Automatic\%20differentiation&oldid=1284647437, 2025,

[2] E. Forest, *Beam dynamics*. CRC Press, 1998. doi:10.1201/9781315138176

[3] M. Berz, *Modern map methods in particle beam physics*. San Diego: Academic Press, 1999.

[4] E. Forest, "Geometric integration for particle accelerators", *J. Phys. A: Math. Gen.*, vol. 39, no. 19, p. 5321, Apr. 2006. doi:10.1088/0305-4470/39/19/S03

[5] L. Deniau and C. Tomoiagă, "Generalised Truncated Power Series Algebra for Fast Particle Accelerator Transport Maps", in *Proc. IPAC'15*, Richmond, VA, USA, May 2015, pp. 374–377. doi:10.18429/JACoW-IPAC2015-MOPJE039

[6] G. Dalle and A. Hill, "A Common Interface for Automatic Differentiation", *arXiv*, 2025. doi:10.48550/arXiv.2505.05542

[7] F. Schäfer, M. Tarek, L. White, and C. Rackauckas, "Abstract-Differentiation.jl: Backend-Agnostic Differentiable Programming in Julia", *arXiv*, 2022. doi:10.48550/arXiv.2109.12449

[8] C. Rackauckas and Q. Nie, "DifferentialEquations.jl–a performant and feature-rich ecosystem for solving differential equations in Julia", *J. Open Res. Softw.*, vol. 5, no. 1, 2017. doi:10.5334/jors.151

[9] C. Rowley, PythonCall.jl: Python and Julia in harmony, 2022. https://github.com/JuliaPy/PythonCall.jl

[10] L. Deniau, "Methodical Accelerator Design - Next Generation: Introduction to the Generalized Truncated Power Series Algebra", presented at the ABP-CAP Section Meeting, Mar. 2025, unpublished. https://indico.cern.ch/event/1513480/contributions/6368615/attachments/3036088/5368770/MADNG-GTPSA.pdf

[11] M. G. Signorelli, GTPSA.jl, 2023. https://github.com/bmad-sim/GTPSA.jl

[12] M. G. Signorelli, NonlinearNormalForm.jl, 2024. https://github.com/bmad-sim/NonlinearNormalForm.jl

[13] H. Yoshida, "Construction of higher order symplectic integrators", *Phys. Lett. A*, vol. 150, no. 5-7, pp. 262–268, 1990. doi:10.1016/0375-9601(90)90092-3