The Journal of Open Source Software

# TaylorSeries.jl: Taylor expansions in one and several variables in Julia

**Luis Benet**[1] **and David P. Sanders**[2]

**1** Instituto de Ciencias Físicas, Universidad Nacional Autónoma de México (UNAM) **2** Departamento de Física, Facultad de Ciencias, Universidad Nacional Autónoma de México (UNAM)

**DOI:** 10.21105/joss.01043

**Software**

- Review
- Repository
- Archive

**Submitted:** 20 September 2018
**Published:** 07 April 2019



## Summary

The purpose of the `TaylorSeries.jl` package is to provide a framework to exploit Taylor polynomials in one and several variables in the Julia programming language (Bezanson, Edelman, Karpinski, & Shah, 2017). It can be thought of as providing a primitive CAS (computer algebra system), which works numerically and not symbolically. The package allows the user to define dense polynomials $p(x)$ of one variable and $p(\mathbf{x})$ of several variables with a specified maximum degree, and perform operations on them, including powers and composition, as well as series expansions for elementary functions of polynomials, for example $\exp[p(x)]$, where techniques of automatic differentiation are used (Haro, Canadell, Figueras, Luque, & Mondelo, 2016; Tucker, 2011). Differentiation and integration are also implemented.

Two basic immutable types are defined, `Taylor1{T}` and `TaylorN{T}`, which represent polynomials in one and several variables, respectively; the maximum degree is a field of the types. These types are parametrized by the type `T` of the polynomial coefficients; they essentially consist of one-dimensional arrays of coefficients, ordered by increasing degree.

In the case of `TaylorN`, the coefficients are `HomogeneousPolynomials`, which in turn are vectors of coefficients representing all monomials with a given number of variables and order (total degree), ordered lexicographically. Higher degree polynomials require more memory allocation, especially for several variables; while we have not extensively tested the limits of the degree of the polynomials that can be used, `Taylor1` polynomials up to degree 80 and `TaylorN` polynomials up to degree 60 in 4 variables have been successfully used. Note that the current implementation of multi-variable series builds up extensive tables in memory which allow to speed up the index calculations.

Julia's parametrized type system allows the construction of Taylor series whose coefficient type is any subtype of the `Number` abstract type. Use cases include complex numbers, arbitrary precision `BigFloats` (Fousse, Hanrot, Lefèvre, Pélissier, & Zimmermann, 2007), `Intervals` (Sanders & Benet, 2019), `ArbFloats` (Sarnoff, 2019), as well as `Taylor1` and `TaylorN` objects themselves.

`TaylorSeries.jl` is the main component of `TaylorIntegration.jl` (Pérez-Hernández & Benet, 2019), whose aim is to perform accurate integration of ODEs using the Taylor method, including jet transport techniques, where a small region around an initial condition is integrated. It is also a key component of `TaylorModels.jl` (Benet & Sanders, 2019), whose aim is to construct rigorous polynomial approximations of functions.

## Examples

We present three examples to illustrate the use of `TaylorSeries.jl`. Other examples, as well as a detailed user guide, can be found in the documentation.

### Hermite polynomials

As a first example we describe how to generate the Hermite polynomials ("physicist's" version) up to a given maximum order. Firstly we directly exploit the recurrence relation satisfied by the polynomials.

```
In [1]: using TaylorSeries

        displayBigO(false)

        function hermite_polynomials(::Type{T}, nmax::Int) where {T <: Integer}

            x = Taylor1(T, nmax)     # Taylor variable
            H = fill(x, nmax + 1)    # vector of Taylor series to be overwritten

            H[1] = 1    # order 0
            H[2] = 2x   # order 1

            for n in 2:nmax
                # recursion relation for order n:
                H[n+1] = 2x * H[n] - 2(n-1) * H[n-1]
            end

            return H
         end

        hermite_polynomials(n) = hermite_polynomials(Int, n);

        H = hermite_polynomials(10);

        function hermite_polynomial(n::Int)
            @assert 0 ≤ n ≤ length(H) "Not enough Hermite polynomials generated"
            return H[n+1]
        end

        hermite_polynomial(6)

Out[1]:  - 120 + 720 t^2 - 480 t^4 + 64 t^6
```

The example above can be slightly modified to compute, for example, the 100th Hermite polynomial. In this case, the coefficients will be larger than $2^{63} - 1$, so the modular behavior, under overflow of the standard `Int64` type, will not suffice. Rather, the polynomials should be generated with `hermite_polynomials(BigInt, 100)` to ensure the use of arbitrary-length integers.

### Using a generating function

As a second example, we describe a numerical way of obtaining the Hermite polynomials from their generating function: the $n$th Hermite polynomial corresponds to the $n$th derivative of the function $\exp(2t\,x - t^2)$.

```
In [2]: 𝒢(x,t) = exp(2t * x - t^2)  # generating function; 𝒢 is typed as \scrG<TAB>

        xn = set_variables("x", numvars=1, order=10)

        x = xn[1]

        t = Taylor1([zero(x), one(x)], 10)  # Taylor1{TaylorN{Float64}}

        gf = 𝒢(x, t)  # Taylor1 expansion of 𝒢

        HH(n::Int) = derivative(n, gf)  # n-th derivative of `gf`

        HH(6)

Out[2]:  - 120.0 + 720.0 x₁² - 480.0 x₁⁴ + 63.99999999999999 x₁⁶
```

This example shows that the calculations are performed numerically and not symbolically, using `TaylorSeries.jl` as a polynomial manipulator; this is manifested by the fact that the last coefficient of `HH(6)` is not identical to an integer.

## Taylor method for integrating ordinary differential equations

As a final example, we give a simple implementation of Picard iteration to integrate an ordinary differential equation, which is equivalent to the Taylor method.

We consider the initial-value problem $\dot{x} = x$, with initial condition $x(0) = 1$. One step of the integration corresponds to constructing the Taylor series of the solution $x(t)$ in powers of $t$:

```
In [3]: ∫_dt(u::Taylor1) = integrate(u) # the symbol ∫ is obtained as \int<TAB>

        function taylor_step(f, u0)

            u = copy(u0)
            unew = u0 + ∫_dt(f(u))

            while unew != u
                u = unew
                unew = u0 + ∫_dt(f(u))    # Picard iteration
            end

            return u
        end

        f(x) = x  # Differential equation

        order = 20  # maximum order of the Taylor expansion for the solution

        u0 = Taylor1([1.0], order)  # initial condition given as a Taylor expansion

        solution = taylor_step(f, u0);   # solution

        solution(1.0) - exp(1.0) # compare solution with the exact value at t=1

Out[3]: 0.0
```

Thus this Taylor expansion of order 20 around $t_0 = 0$ suffices to obtain the exact solution at $t = 1$, while the error at time $t = 2$ from the same expansion is $4.53 \times 10^{-14}$. This indicates that a proper treatment should estimate the size of the required step that should be taken as a function of the solution.

## Acknowledgements

We are thankful for the additions of all contributors to this project. We acknowledge financial support from PAPIME grants PE-105911 and PE-107114, and PAPIIT grants IG-101113, IG-100616 and IN-117117. LB and DPS acknowledge support via the Cátedra Marcos Moshinsky (2013 and 2018, respectively).

## References

Benet, L., & Sanders, D. P. (2019). TaylorModels.jl: Rigorous function approximation with Taylor models in Julia. Retrieved from https://github.com/JuliaIntervals/TaylorModels.jl

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, *59*(1), 65–98. doi:10.1137/141000671

Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., & Zimmermann, P. (2007). MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, *33*(2). doi:10.1145/1236463.1236468

Haro, À., Canadell, M., Figueras, J.-L., Luque, A., & Mondelo, J. M. (2016). *The parameterization method for invariant manifolds: From rigorous results to effective computations*. Applied mathematical sciences. Switzerland: Springer International Publishing. doi:10.1007/978-3-319-29662-3

Pérez-Hernández, J. A., & Benet, L. (2019). TaylorIntegration.jl: Precise ODE integration using Taylor's method in Julia. doi:10.5281/zenodo.2562364

Sanders, D. P., & Benet, L. (2019). IntervalArithmetic.jl: Rigorous floating-point calculations using interval arithmetic in Julia. Retrieved from https://github.com/JuliaIntervals/IntervalArithmetic.jl

Sarnoff, J. (2019). ArbFloats.jl. Retrieved from https://github.com/JuliaArbTypes/ArbFloats.jl

Tucker, W. (2011). *Validated numerics: A short introduction to rigorous computations*. Princeton, NJ, USA: Princeton University Press.