ADTAYL

Taylor Series, ODEs, and Lie Derivatives in MATLAB

Ned Nedialkov
McMaster University
Canada

John Pryce
Cardiff University
UK

28th EuroAD Workshop
December 8–10, 2025
CERN, Switzerland

Outline

Reference: NN & JP. Sub-ODEs simplify Taylor series algorithms for ordinary differential equations, SIAM Journal on Scientific Computing, 47 (2025), pp. A2746–A2773.

ADTAYL

ADTAYL is a MATLAB package with two main parts:

- adtayl – a class for computations with univariate Taylor series
- odets – an ODE solver based on Taylor series.

Key features

- adtayl objects behave like MATLAB arrays; each element stores the coefficients of a Taylor series.
- odets
  - interface is like the built-in MATLAB solvers ode45, etc.
  - can perform high-precision integrations
- Simple mechanism for computing Lie derivatives.
- ADTAYL works in
  - double precision by default
  - supports MATLAB's variable-precision arithmetic (vpa)
  - both real and complex.
- The "AD" means purely numeric – no symbolic algebra.

adtayl class

- An adtayl object is an array with each element storing the Taylor coefficients of a scalar function $x(t)$ to a given order p, expanded at some t_0:

$$x(t_0 + s) = x_0 + x_1 s + \cdots + x_p s^p + O(s^{p+1}), \quad x_k = \frac{x^{(k)}(t_0)}{k!}.$$

- All elements are of the same order.
- Operations perform truncated Taylor series (TTS) arithmetic, discarding terms of order $> p$.
- E.g.

```
t = adtayl(0,3);  % t = 0 + 1s + 0s^2 + 0s^3,    independent variable
x = exp(t);       % x = 1 + 1s + 0.5s^2 + 0.1667s^3
y = [x.^2;1./x];% y = [ 1 + 2s + 2s^2 + 1.3333s^3 ]
                      [ 1 - 1s + 0.5s^2 - 0.1667s^3 ]
```

(displayed to 4 digits but stored to full double precision)

adtayl: functionality highlights

- Element-wise binary operations `+`, `-`, `.*`, `./`, `.^`
- Full set of element-wise standard functions exp, log, sin, asin, ...
- Binary operations can mix adtayl with numeric arrays and scalars

  E.g. if `y` contains $\begin{bmatrix} 1+2s+2s^2+1.3333s^3 \\ 1-1s+0.5s^2-0.1667s^3 \end{bmatrix}$,

  `z = 1+y; % contains` $\begin{bmatrix} 2+2s+2s^2+1.3333s^3 \\ 2-1s+0.5s^2-0.1667s^3 \end{bmatrix}$

- Basic matrix operations. If `A` and `B` are adtayl arrays storing TTS of matrix-valued functions $A(t)$ and $B(t)$, then
  - `A*B` computes TTS of $A(t)\,B(t)$
  - `A\B` solves $A(t)\,X(t) = B(t)$ in TTS arithmetic (if constant term of $A(t)$ is ordinary nonsingular matrix).

  `A` and `B` must be of compatible sizes (in usual matrix sense) and orders (have the same order, or one has order 0).

odets solver

- odets numerically solves IVPs

$$y' = f(t, y), \qquad y(t_0) = y_0.$$

  It also recognises and accepts autonomous ODEs $y' = f(y)$.
- odets preprocesses f into a code-list object `cl`, which is interpreted to compute Taylor coefficients (TCs):
  - `cl` can be reused unchanged for different initial conditions
  - parameters $y' = f(t, y; p)$ can be handled by reusing the same `cl` and setting only the p values between runs.
- Fixed Taylor order, variable stepsize control.
- Arbitrary-precision integration by giving initial conditions as `vpa(y0)` instead of double `y0`.
  - Allows integrations for very small tolerances.
  - Built-in solvers cannot handle tolerances $\lesssim 10^{-14}$.

odets cont.

- Interface like the built-in solvers:

```
[t,y] = odets(odefun,tspan,y0)
[t,y] = odets(odefun,tspan,y0,options)
sol = odets(___)
```

- Outputs (t_i, y_i) at internally chosen points t_i, or at user-specified output times t_i^*, with $y(t_i^*)$ computed by interpolation.
- Optionally, at each t_i returns TCs of the solution and Jacobians of these TCs:
  - w.r.t. initial state y_0 at t_0 — TCs of the variational equation
  - w.r.t. current state y_i at t_i — TCs of the local variational equation
- Standard MATLAB stuff not supported yet:
  - event location (in progress)
  - solve stiff ODEs by implicit method (big project but we know how)
  - ODEs with mass matrix
  - ODEs with nonsmooth or discontinuous $f(t, y)$

Taylor coefficients

AD gives a practical scheme for computing coefficients.

Example. Consider $\dot{x} = x^2$, $x(t_0) = x_0$.

- Assume a series $x(t_0 + s) = x_0 + x_1 s + x_2 s^2 + \cdots$. Then

$$\dot{x} = x_1 + 2x_2 s + 3x_3 s^2 + \cdots$$
$$x^2 = x_0^2 + (2x_0 x_1)s + (2x_0 x_2 + x_1^2)s^2 + \cdots .$$

- Starting with x_0, and equating these term by term:

$$x_1 = x_0^2$$
$$x_2 = \tfrac{1}{2}(2x_0 x_1)$$
$$x_3 = \tfrac{1}{3}(2x_0 x_2 + x_1^2)$$
$$\vdots$$

Taylor method

A Taylor method (TM) for $\dot{x} = f(t, x),\ x(t_0) = x_0$

- computes the TS to some order of the solution at the current t, and
- sums the TS with a suitable step size to move to the next point.

- A TM requires $f(t, x)$ to be provided as code, which is executed in the form of recurrences that compute TCs.
- Recurrences are needed for:
  - basic arithmetic operations (BAOs)
  - standard functions: $\exp$, $\sin$, etc.
- Most TM implementations use individual recurrences for each standard function.
- odets implements our sub-ODE method eliminating the need for individual recurrences.
- The computational kernel is the implementation of 5 operators: for the BAOs $+, -, \times, \div$ and our sub-ODE operator $\odot$.

sub-ODE idea

- A standard function is treated as a TS operation. E.g. given the coefficients of u to order p, compute TCs of

$$v(t) = \exp\big(u(t)\big) \quad \text{to order } p.$$

- Each standard function satisfies a simple ODE. E.g.

$$v = \exp(u) \quad \text{satisfies} \quad \dot{v} = v\,\dot{u}.$$

- In the code-list, replace each call to $\exp$ by its sub-ODE $\dot{v} = v\,\dot{u}$; similarly for all standard functions.
- In general, the $\odot$ operator implements a sub-ODE of the form

$$\dot{v} = h(u, v)\,\dot{u}, \quad \text{with } h \text{ using only the four BAOs } +,\ -,\ \times,\ \div.$$

- Lets us code ~ 40 standard functions in ~ 130 lines of MATLAB.
- $\odot$ calls a standard function (e.g. $\exp$) for the TS's constant term; all higher orders come automatically from the sub-ODE.

Computing high-order Lie brackets

We compute 3 kinds of high-order Lie derivatives.
This is about high-order Lie brackets.

- Let $f, g : \mathcal{M} \subseteq \mathbb{R}^n \to \mathbb{R}^n$ be smooth vector fields.
- Lie derivative of g along f is the Lie bracket, another vector field:

$$\mathrm{ad}_f\, g(x) = [f, g](x) = g'(x)\, f(x) - f'(x)\, g(x), \qquad (' = \text{Jacobian}).$$

  ad_f notation regards it as linear operator on g's, for given f.
- Higher-order (iterated) Lie brackets are powers of this operator:

$$\mathrm{ad}_f^0\, g = g,$$
$$\mathrm{ad}_f^k\, g = \mathrm{ad}_f\, \mathrm{ad}_f^{k-1}\, g, \qquad k \geq 1.$$

Task: Given order p and point x_0, compute $\mathrm{ad}_f^k\, g(x_0)$ for $k = 1\!:\!p$.

- Solution: The $\mathrm{ad}_f^k\, g(x_0)$ satisfy

$$\sum_{k=0}^{\infty} \frac{1}{k!}\, \mathrm{ad}_f^k\, g(x_0)\, t^k = J^{-1}(t)\, g(x(t)),$$

where $x(t)$ solves $\dot{x} = f(x)$, $x(0) = x_0$ and $J(t) = \frac{\partial x(t)}{\partial x_0}$ solves

$$\dot{J}(t) = f'\big(x(t)\big)\, J(t), \qquad J(0) = I.$$

- $\frac{1}{k!}\, \mathrm{ad}_f^k\, g(x_0)$ is the kth TC of $J^{-1}(t)\, g(x(t))$ at $t = 0$.
- With `f`, `g` coded in ordinary MATLAB, the whole computation is:

```
[x,J] = taylorcoeffs(f,x0,p);
ad    = J\g(x);
```

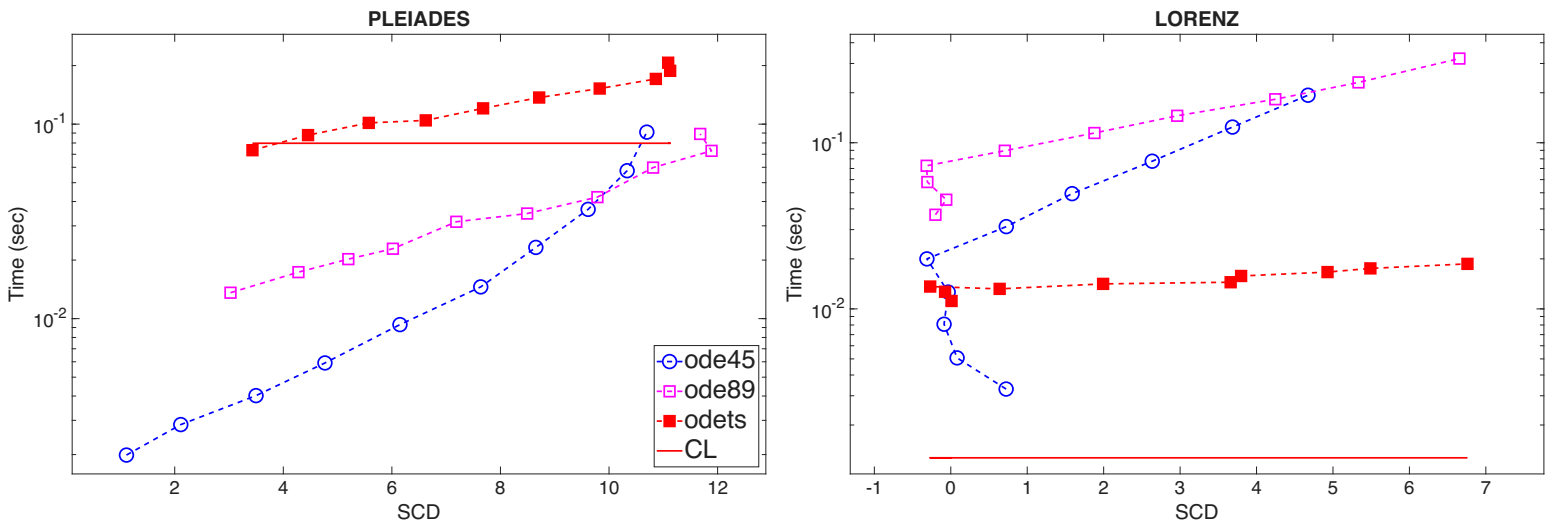
`taylorcoeffs(f,x0,p)` $\rightarrow$ TCs of $x(t)$ and $J(t)$ with $x(0) = x_0$

`J\g(x)` $\rightarrow$ TCs of $J^{-1}(t)\, g(x(t))$.

Then, `ad.gettc(k)` $\rightarrow \frac{1}{k!}\, \mathrm{ad}_f^k\, g(x_0)$.

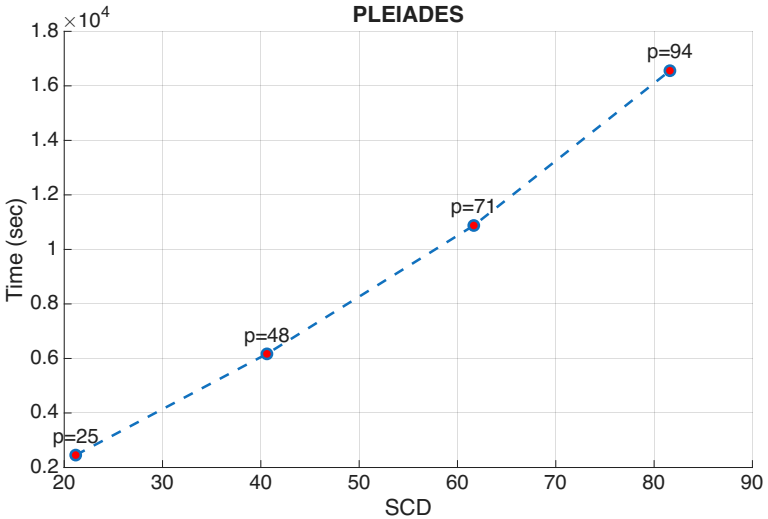
Performance: odets vs. built-in solvers

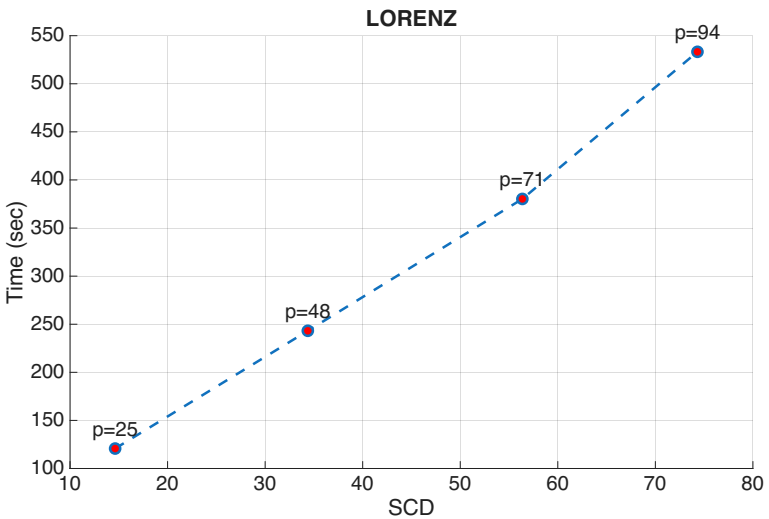
- Mac Studio, M4 Max, 64 GB RAM, MATLAB 2025b
- Plots CPU time vs. significant correct digits (SCD) at t_{end}
- Pleiades, motion of 7 stars (size 28), $t_{\text{end}} = 3$
- Lorenz system (size 3), $t_{\text{end}} = 20$



CL: time for code-list generation; odets: time for integration only

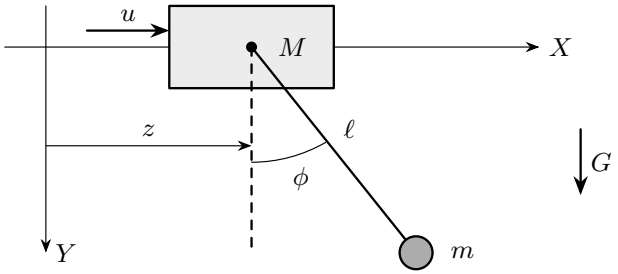
odets high-accuracy integration





- vpa integration with tolerances 10^{-20}, 10^{-40}, 10^{-60}, 10^{-80}.
- Reference solution computed with tolerance 10^{-85}.
- vpa is slow due to array indexing overhead rather than the arithmetic itself.

Gantry crane example[1]



- Cart of mass M moving horizontally
- Load of mass m, cable length ℓ, angle ϕ
- Input $u(t)$: horizontal force on cart
- Gravity G downward

Equations of motion

$$\begin{pmatrix} m+M & m\ell\cos\phi \\ m\ell\cos\phi & m\ell^2 \end{pmatrix} \begin{pmatrix} \ddot{z} \\ \ddot{\phi} \end{pmatrix} + \begin{pmatrix} -m\ell\dot{\phi}^2\sin\phi \\ Gm\ell\sin\phi \end{pmatrix} = \begin{pmatrix} u \\ 0 \end{pmatrix}$$

Setting $x = \left(z, \phi, \dot{z}, \dot{\phi}\right)$, the corresponding first-order system is of the form

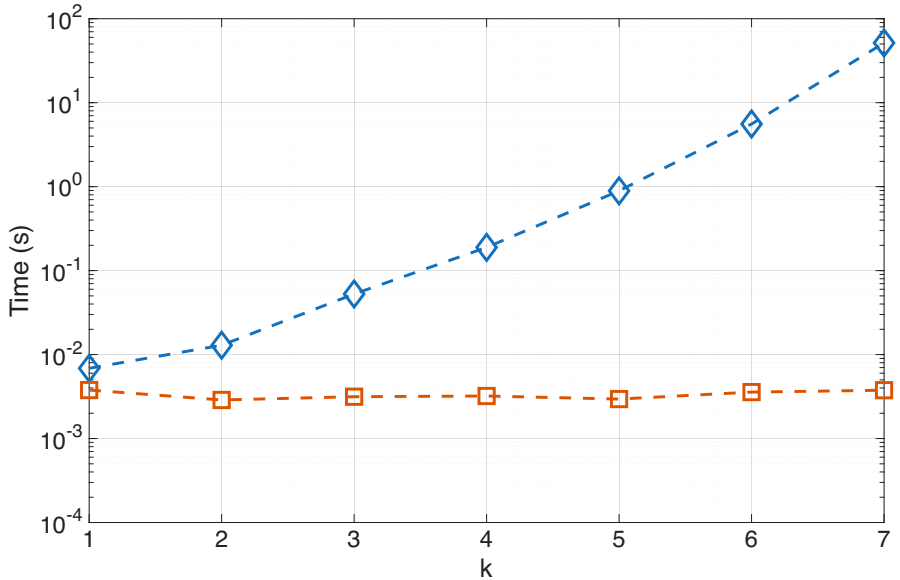
$$\dot{x} = f(x) + g(x)u.$$

Control theory needs to compute $\mathrm{ad}_f^k\, g$ for k up to some p.

[1] From K. Röbenack et al., LIEDRIVERS — Efficient Lie derivative computation using ADOL-C..., EOOLT Workshop, 2011.

Gantry crane: ADTAYL vs. symbolic Lie brackets

- With suitable values for the parameters M, m, ℓ, G and x_0:



Time for $\mathrm{ad}_f^k\, g(x_0)$ vs. order k

- Note log scale on time axis.

Conclusion

- adtayl: easy-to-use Taylor series computations.
- odets: familiar interface of built-in ODE solvers,
  and high-precision integration beyond capabilities of built-in solvers.
- ADTAYL: Lie derivatives and Lie brackets efficiently and elegantly,
  much faster than symbolic methods.

We are completing the user guide and plan to release the code.

Credit. To our knowledge, R. Neidinger (2013) is the first person to give an embedding method that can, in principle, be automated—an operator he calls `ddot`, nearly equivalent to our $\odot$.