

Chaotic attractors and Lyapunov exponents of fractional-order systems: a tutorial on the augmented-system algorithm, with a complete Python program

Stefano Fazzino

sfazzino@gmail.com

University of Catania <https://orcid.org/0000-0002-3015-4938>

Research Article

Keywords: Lyapunov, Fractional-Order, chaos, tutorial, python

Posted Date: October 20th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-11391039/v1>

License:   This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: The authors declare no competing interests.

Chaotic attractors and Lyapunov exponents of fractional-order systems: a tutorial on the augmented-system algorithm, with a complete Python program

Stefano Fazzino

Independent researcher, Catania, Italy

`sfazzino@gmail.com`

Abstract

This tutorial explains, from first principles and with a complete program of about one hundred and fifty lines of Python, how to simulate a chaotic fractional-order system and how to compute its Lyapunov exponents. The method is the augmented-system algorithm of the diffusive representation, introduced in a recent article by the author: the power-law memory kernel of the Caputo derivative is written as a sum of exponentials, each exponential becomes a first-order “memory mode”, and the fractional system becomes an ordinary, autonomous system in the space of its modes, to which the standard tools of dynamical systems apply. The mathematics of the original article is recalled only to the extent needed to justify each design choice of the program: which modes to keep, how to advance them in time, in which norm to orthonormalise the tangent vectors, and how to read the exponents that come out, in particular why a fractional system has no negative exponents. The program is validated on a linear system with known exponent, to better than 0.05%, and applied to the fractional Lorenz and Chen systems. Every figure of the paper is produced from the output of the program listed in the appendix by a short plotting script distributed with it.

1 Introduction

A fractional-order dynamical system is one in which the time derivative is replaced by a derivative of non-integer order $\nu \in (0, 1)$. The replacement is not a mathematical curiosity: it is the natural way to model a process whose present rate of change depends on its whole past with a power-law weight, and such processes are common, from viscoelastic materials to electrochemical cells to the adaptation of neurons [1, 2]. When the system is nonlinear it can be chaotic, and the fractional versions of the Lorenz, Rössler and Chen systems have been simulated for twenty years [3]. What has been much less clear is how to *measure* this chaos. The Lyapunov exponents, which for an ordinary system quantify the exponential separation of nearby trajectories and are computed by a classical algorithm [4], are not even well defined for a fractional system in the usual sense, because a fractional system is not a dynamical system in the usual sense: its future depends on its whole past, not on its present state, and the algorithms in use either truncate the memory or apply Benettin’s method to a scheme whose relation to the fractional dynamics is not established [9].

In a recent article [5] we gave a definition and an algorithm that resolve the difficulty, by taking seriously the idea that the state of a fractional system is its memory. The present paper is the practical companion of that article. It is written for the reader who wants to simulate a fractional chaotic system and compute its exponents correctly, and who is willing to understand why the algorithm is built the way it is, but who does not need the proofs. We therefore recall the mathematics of [5] only where it dictates a choice in the code, and we build the program step by step, each step justified by one idea. The result is a self-contained Python program, listed in

full in the appendix, which runs in a minute on a laptop and which the reader can modify for their own system by changing ten lines.

The plan is the following. Section 2 states what a fractional system is and why ordinary integrators and ordinary Lyapunov algorithms do not apply to it. Section 3 presents the one idea on which everything rests, that a power law is a sum of exponentials, and derives the augmented system. Section 4 builds the program: the choice of modes, the time step, the attractor, the tangent system and the orthonormalisation, the run. Section 5 validates it on a linear system whose exponent is known exactly, Section 6 applies it to the Lorenz and Chen systems, and Section 7 lists the mistakes that are easy to make and that the theory tells us to avoid.

2 The problem: a system whose state is its past

The Caputo derivative of order $\nu \in (0, 1)$ of a function x defined for $t \geq 0$ is

$${}^C D^\nu x(t) = \frac{1}{\Gamma(1-\nu)} \int_0^t (t-s)^{-\nu} x'(s) ds, \quad (1)$$

a weighted average of the ordinary derivative over the whole past, with a weight that decays as a power law. A fractional-order system is an equation ${}^C D^\nu x = f(x)$ with $x \in \mathbb{R}^n$ and an initial condition $x(0) = x_0$; the Lorenz, Rössler and Chen systems of this paper are obtained from the ordinary ones by this replacement, with the same vector fields. For what follows it is better to use the equivalent integral form, which one obtains by inverting (1):

$$x(t) = x_0 + \int_0^t k_\nu(t-s) f(x(s)) ds, \quad k_\nu(\tau) = \frac{\tau^{\nu-1}}{\Gamma(\nu)}. \quad (2)$$

For $\nu = 1$ the kernel is $k_1 \equiv 1$ and (2) is just $x(t) = x_0 + \int_0^t f(x) ds$, the integral form of an ordinary differential equation. For $\nu < 1$ the kernel k_ν is a decreasing power law, singular at $\tau = 0$ and with a tail that decays so slowly that it is not integrable at infinity: the vector field at every past instant keeps contributing to the present value of x , with a weight that never becomes negligible.

This is why the usual tools fail. An ordinary integrator advances the state $x(t)$ to $x(t+h)$ using only $x(t)$; in (2) there is no such thing as “the state at time t ”, because $x(t+h)$ depends on $x(s)$ for all $s \leq t$. The obvious way out, to keep the whole history and recompute the integral at every step, costs time proportional to the square of the number of steps and is what the Grünwald–Letnikov and predictor–corrector schemes do [1, 2]; in practice the history is often truncated to a finite window to make the cost bearable. The Lyapunov exponents raise a deeper problem. The exponents of an ordinary system are defined through the linearised flow, which maps a perturbation of the state at time 0 to a perturbation at time t ; a fractional system has no flow, because it has no state, and the quantities computed in the literature by applying Benettin’s algorithm to a truncated-memory scheme are, as shown in [5], exponents of a different system, the one with the truncated memory, which can differ substantially from the fractional one. What we need is a formulation in which the fractional system *has* a state and a flow. It exists, and it is simple.

3 The idea: a power law is a sum of exponentials

The kernel k_ν can be written as a superposition of decaying exponentials. The identity

$$\frac{\tau^{\nu-1}}{\Gamma(\nu)} = \int_0^\infty \mu_\nu(\omega) e^{-\omega\tau} d\omega, \quad \mu_\nu(\omega) = \frac{\sin \pi\nu}{\pi} \omega^{-\nu}, \quad (3)$$

is checked by computing the integral on the right, which is $\frac{\sin \pi \nu}{\pi} \Gamma(1 - \nu) \tau^{\nu-1}$, and using Euler's reflection formula $\Gamma(\nu) \Gamma(1 - \nu) = \pi / \sin \pi \nu$. It says that the power-law memory is the sum of infinitely many exponential memories, one for each relaxation rate ω , weighted by $\mu_\nu(\omega)$; slow rates ($\omega \rightarrow 0$) have a large weight, which is why the tail of the kernel is long, and the weight is integrable near $\omega = 0$ only because $\nu < 1$. This is the diffusive representation [6, 7], and its consequence for us is immediate. Substituting (3) into (2) and exchanging the integrals,

$$x(t) = x_0 + \int_0^\infty \mu_\nu(\omega) z(\omega, t) d\omega, \quad z(\omega, t) := \int_0^t e^{-\omega(t-s)} f(x(s)) ds. \quad (4)$$

Each $z(\omega, t)$ is the output of a first-order linear filter with rate ω driven by $f(x)$: it satisfies the ordinary differential equation

$$\partial_t z(\omega, t) = -\omega z(\omega, t) + f(x(t)), \quad z(\omega, 0) = 0, \quad (5)$$

and x is recovered from the z 's by the weighted sum in (4). Equations (4)–(5) are the *augmented system*: an ordinary, autonomous system of differential equations whose unknowns are the *memory modes* $z(\omega, t)$, one for each ω , and whose output is x . The fractional system has become a dynamical system, at the price of an infinite-dimensional state. The state is the function $\omega \mapsto z(\omega, t)$, that is, the memory of the system sorted by relaxation rate, and the initial condition $z(\cdot, 0) = 0$ says that the Caputo problem starts with empty memory.

Two facts about this system, proved in [5], are all the theory we need, because each one dictates a design choice of the program.

The first is that the memory *ages*. The slow modes, those with small ω , relax so slowly that they keep accumulating the past: even when $x(t)$ stays bounded on a chaotic attractor, the norm of the memory grows without bound, as $t^{(1-\nu)/2}$. The Caputo trajectory is a transient that never ends, and the slow modes are not a detail that one can drop: they are where the long-time behaviour of the system is decided. The design consequence is that the program must represent the slow modes down to rates ω of the order of $1/T$, where T is the length of the simulation, and must not truncate the memory, however tempting; Section 7 shows what happens if it does.

The second is that the linearised dynamics of the augmented system, the one that governs small perturbations and hence the Lyapunov exponents, has a very particular structure: it is the sum of a pure relaxation of the modes, $\delta z(\omega, t) \mapsto e^{-\omega t} \delta z(\omega, 0)$, which has norm one and contracts nothing uniformly because ω can be arbitrarily small, and of a compact operator, the coupling through the Jacobian of f . A theorem of Thieullen on linearised flows of this form [8] then says two things. The *positive* Lyapunov exponents are well defined and discrete, exactly as for an ordinary system, because the compact part behaves, as far as growth rates are concerned, like a finite matrix. The non-positive part of the spectrum, instead, is a continuous band sitting at zero, inherited from the relaxation of the modes. A fractional system has no negative exponents: perturbations that die do so algebraically, as a power of t , not exponentially. The design consequence is twofold. The norm in which the tangent vectors are measured must be the one in which this structure holds, and it is a specific weighted norm on the modes, given below; and the program should be expected to return one or more positive exponents followed by exponents that drift slowly towards zero, never a clean negative number. A reader who sees $\lambda_3 \approx -0.0003$ instead of the -14.6 of the ordinary Lorenz system is seeing the theory, not a bug.

4 Building the program

4.1 Choosing the modes

The augmented system has a continuum of modes; the program keeps finitely many, $z_i = z(\omega_i, t)$, $i = 1, \dots, J$, chosen so that the weighted sum in (4) is approximated by a quadrature rule,

$$\int_0^\infty \mu_\nu(\omega) g(\omega) d\omega \approx \sum_{i=1}^J w_i g(\omega_i), \quad (6)$$

accurate for the functions g that actually appear, which are the exponentials $e^{-\omega\tau}$ with τ between the time step h and the simulation length T . The accuracy of the whole method is the accuracy with which $\sum_i w_i e^{-\omega_i\tau}$ reproduces $k_\nu(\tau)$ on $[h, T]$, and this is the one quantity to check before anything else.

The rule we use exploits two properties of μ_ν . It is a power $\omega^{-\nu}$, singular at the origin; and the integrand $\mu_\nu(\omega)e^{-\omega\tau}$ varies over many decades of ω but is smooth in $\ln\omega$. So we split $[0, \omega_{\max}]$ into a *singular panel* $[0, \omega_{\min}]$, treated by a Gauss–Jacobi rule with weight $\omega^{-\nu}$, that is, a Gauss rule whose nodes and weights are built for integrands of the form $\omega^{-\nu}$ times a polynomial, which it integrates exactly, and *geometric panels* $[\omega_{\min}, \omega_{\max}]$, one per decade, each treated by a Gauss–Legendre rule in the variable $u = \ln\omega$, where $\mu_\nu(\omega) d\omega = \frac{\sin \pi\nu}{\pi} \omega^{1-\nu} du$. With eight nodes per panel, $\omega_{\min} = 0.1/T$ and $\omega_{\max} = 10^4/h$, this gives about a hundred modes and reproduces the kernel on $[h, T]$ to a relative error of 10^{-9} (Figure 1). The code is short:

```
def modes(nu, om_min, om_max, per_decade=8):
    c = math.sin(math.pi*nu)/math.pi
    xj, wj = roots_jacobi(per_decade, 0.0, -nu)          # singular panel [0, om_min]
    om = [om_min*(1+xj)/2]; w = [c*(om_min/2)**(1-nu)*wj]
    xl, wl = roots_legendre(per_decade)                  # one panel per decade in ln(om)
    for k in range(int(math.ceil(math.log10(om_max/om_min)))):
        u0 = math.log(om_min) + k*math.log(10); u1 = u0 + math.log(10)
        u = 0.5*(u1-u0)*xl + 0.5*(u1+u0)
        om.append(np.exp(u)); w.append(c*np.exp(u)**(1-nu)*wl*0.5*(u1-u0))
    return np.concatenate(om), np.concatenate(w)
```

The singular panel deserves a comment, because it is where the ageing of Section 3 enters the code. Its eight nodes are the slowest modes of the system, with rates between $2 \cdot 10^{-7}$ and 10^{-4} for $T = 1000$; they carry little weight individually, and it is tempting to drop them. Figure 1 shows what dropping them does to the kernel: the relative error jumps from 10^{-9} to 26% at $\tau = 0.01$, 42% at $\tau = 1$, 53% at $\tau = 10$ and 83% at $\tau = 1000$. The deficit is large at *all* lags, not only at long ones, because removing the modes below $\omega_{\min}$ removes from the kernel an amount whose ratio to $k_\nu(\tau)$ is $(\omega_{\min}\tau)^{1-\nu}/\Gamma(2-\nu)$, and since the exponent $1-\nu$ is small this ratio is 26% already at $\omega_{\min}\tau = 10^{-6}$. Removing slow modes is not a mild truncation of the memory; it changes the system.

4.2 Advancing in time

With the modes chosen, the state of the program is the matrix $z \in \mathbb{R}^{J \times n}$ of the modes, plus the constant x_0 , and the output is $x = x_0 + \sum_i w_i z_i$. The equations to integrate are the J copies of (5),

$$z'_i = -\omega_i z_i + f(x), \quad x = x_0 + \sum_j w_j z_j, \quad (7)$$

an ordinary system of dimension nJ . It is stiff: the rates ω_i range from 10^{-7} to about 10^7 , and an explicit Runge–Kutta method would need a step $h \lesssim 1/\omega_{\max}$, a million times smaller than the step the dynamics of f requires. The remedy is standard and exact for the stiff part: since the linear term $-\omega_i z_i$ is diagonal, we integrate it analytically and treat only the nonlinear term

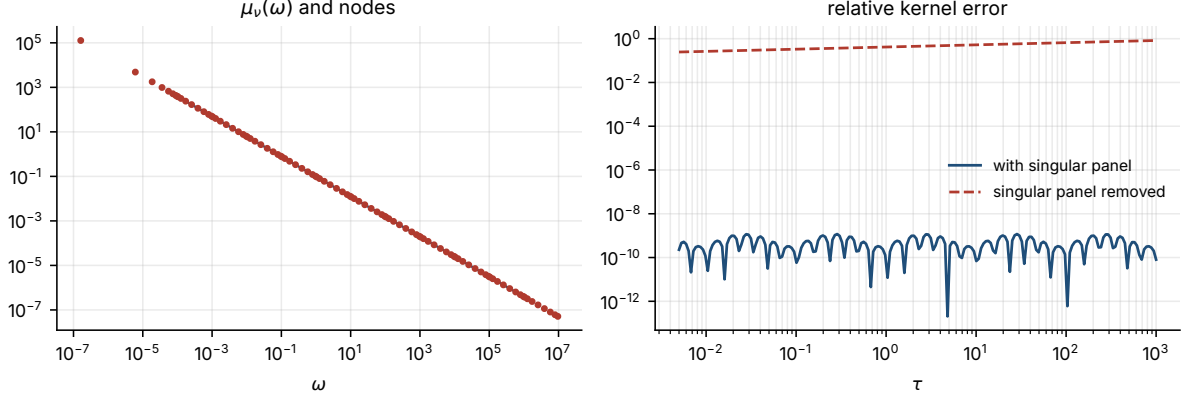


Figure 1: Left: the weight $\mu_\nu(\omega)$ for $\nu = 0.9$ and the 96 quadrature nodes ($\omega_{\min} = 10^{-4}$, eleven decades of eight nodes each up to 10^7 , plus the singular panel). Right: relative error of $\sum_i w_i e^{-\omega_i \tau}$ with respect to $\tau^{\nu-1}/\Gamma(\nu)$ on $[h, T] = [0.005, 1000]$, with the singular panel (solid, below 10^{-8}) and with the singular panel removed (dashed, tens of per cent at every lag).

$f(x)$ numerically. This is *exponential time differencing*; in its second-order Runge–Kutta form [10], writing $E_i = e^{-\omega_i h}$, $a_i = (1 - E_i)/\omega_i$ and $b_i = (E_i - 1 + \omega_i h)/(\omega_i^2 h)$, a step from t to $t + h$ is

$$z_i^{\text{pred}} = E_i z_i + a_i N_0, \quad z_i(t+h) = z_i^{\text{pred}} + b_i (N_1 - N_0), \quad N_0 = f(x(t)), \quad N_1 = f(x^{\text{pred}}), \quad (8)$$

where x^{pred} is the output computed from the predicted modes. For $\omega_i h \rightarrow 0$ the coefficients reduce to $a_i \rightarrow h$, $b_i \rightarrow h/2$ and (8) becomes Heun’s method; for $\omega_i h \rightarrow \infty$, $E_i \rightarrow 0$ and $a_i, b_i \rightarrow 1/\omega_i$, so that $z_i(t+h) \rightarrow f(x^{\text{pred}})/\omega_i$: the fast modes are slaved to $f(x)/\omega_i$, which is their exact quasi-static behaviour. The step h is then chosen for the dynamics of f alone, $h = 0.005$ for the systems of this paper, whatever $\omega_{\max}$ is. The coefficients are computed once, with a series expansion for small $\omega_i h$ to avoid cancellation:

```
def etd_coeffs(om, h):
    oh = om*h; E = np.exp(-oh)
    a = np.where(oh > 1e-6, (1-E)/np.maximum(om, 1e-300), h*(1-oh/2+oh**2/6))
    b = np.where(oh > 1e-4, (E-1+oh)/np.maximum(om**2*h, 1e-300), h*(0.5-oh/6+oh**2/24))
    return E, a, b
```

One detail concerns the start. The solution of a Caputo problem with empty memory behaves as $x_0 + ct^\nu$ for small t , with an infinite derivative at $t = 0$, and a fixed step h makes an error of order h^ν in the first step that then propagates. The program therefore divides each of the first $1/h$ steps into twenty sub-steps; after $t = 1$ the solution is smooth and the full step is used. This costs nothing and reduces the start-up error by a factor $20^\nu \approx 11$.

4.3 Drawing the attractor

With (7) and (8) the program already simulates the fractional system: initialise $z = 0$, step forward, record $x = x_0 + \sum w_i z_i$. Figure 2 shows the attractors of the fractional Chen system at $\nu = 0.9$ and of the fractional Lorenz system at $\nu = 0.995$, obtained in this way with the parameters of Table 1 and 300 time units of integration, the first hundred discarded and the trajectory sampled every 0.01 (the listed program samples it at the renormalisation times; the plotting script calls `run` with a finer sampling). They look like their integer-order relatives, with a difference that is visible once one knows to look for it: the Lorenz attractor at $\nu = 0.995$ is close to the threshold $\nu \approx 0.994$ below which, by Matignon’s theorem, the two non-trivial equilibria become stable and the chaos disappears, and the trajectory lingers near them longer

system	vector field $f(x, y, z)$	parameters	x_0	order
Lorenz	$(\sigma(y - x), x(\rho - z) - y, xy - \beta z)$	10, 28, 8/3	(1, 2, 0.1)	$\nu = 0.995$
Chen	$(a(y - x), (c - a)x - xz + cy, xy - bz)$	35, 3, 28	(-9, -5, 14)	$\nu = 0.9$
Rössler	$(-y - z, x + ay, b + z(x - c))$	0.5, 0.2, 10	(1, 1, 0.1)	$\nu = 0.9$

Table 1: The systems implemented in the program. The orders are those at which the program finds chaos; Lorenz converges to an equilibrium for $\nu \leq 0.99$ (Matignon threshold 0.994), Chen has a periodic window at $\nu = 0.99$.

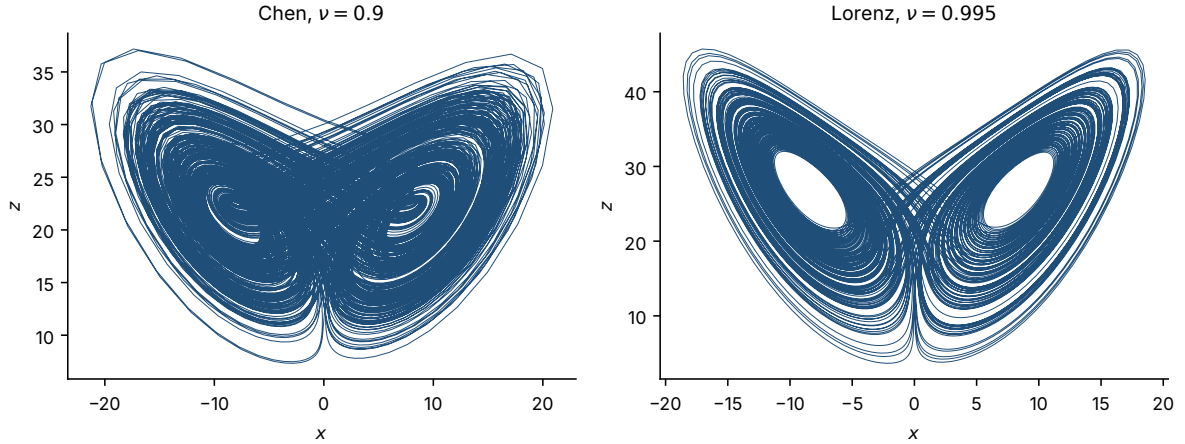


Figure 2: Attractors of the fractional Chen system at $\nu = 0.9$ (left) and of the fractional Lorenz system at $\nu = 0.995$ (right), projected on the (x, z) plane; $h = 0.005$, $t \in [100, 300]$.

than at $\nu = 1$. A reader who integrates the Lorenz system at $\nu = 0.99$ with this program will see it converge to an equilibrium, and should not conclude that the program is wrong.

4.4 The tangent system and the right norm

Lyapunov exponents measure the growth of small perturbations, and small perturbations of the augmented system obey its linearisation. Perturbing $x_0 \rightarrow x_0 + \delta a$ and $z_i \rightarrow z_i + \delta z_i$ in (7) and keeping first-order terms,

$$\delta a' = 0, \quad \delta z_i' = -\omega_i \delta z_i + J(x) \delta x, \quad \delta x = \delta a + \sum_j w_j \delta z_j, \quad (9)$$

where $J(x) = Df(x)$ is the Jacobian of the vector field. A tangent vector is thus a pair $(\delta a, \delta z)$ with $\delta a \in \mathbb{R}^n$ and $\delta z \in \mathbb{R}^{J \times n}$: a perturbation of the initial condition, which is constant in time, and a perturbation of the memory. The tangent equation has the same form as (7) with $f(x)$ replaced by $J(x)\delta x$, so the same exponential time differencing advances it, and the program integrates trajectory and tangent vectors together, with the Jacobian evaluated at the same two points of each step.

Benettin's algorithm [4] integrates m tangent vectors, orthonormalises them by a QR decomposition every Δt_{QR} time units, and accumulates the logarithms of the diagonal of R ; the exponents are the accumulated logarithms divided by the elapsed time. The one question specific to the fractional case is *in which norm* to orthonormalise. This is not a matter of taste: the structure of Section 3, relaxation of norm one plus compact coupling, holds in a specific norm, and it is in that norm that the exponents are the ones defined in [5]. The norm is

$$\|(\delta a, \delta z)\|^2 = |\delta a|^2 + \int_0^\infty \mu_\nu(\omega)(1 + \omega) |\delta z(\omega)|^2 d\omega \approx |\delta a|^2 + \sum_i w_i(1 + \omega_i) |\delta z_i|^2, \quad (10)$$

the weight $\mu_\nu(1 + \omega)$ being the one for which the output map $\delta z \mapsto \int \mu_\nu \delta z$ has norm exactly one and the linearised evolution has the structure described in Section 3. In the program this weighted norm is turned into a Euclidean one by rescaling the mode coordinates, $\delta \zeta_i := \sqrt{w_i(1 + \omega_i)} \delta z_i$: the tangent vectors are stored as columns of a matrix Q in the coordinates $(\delta a, \delta \zeta)$, the standard `numpy` QR is applied to Q , and the rescaling is undone before evaluating the right-hand side of (9). This is the whole content of the methods `_split`, `_join` and `renormalize` of the class `Augmented` in the appendix. The step of trajectory and tangent vectors together reads:

```
def step(self, h=None):
    h = self.h if h is None else h
    if h not in self.coeffs: self.coeffs[h] = etd_coeffs(self.om, h)
    E, a, b = (c[:, None] for c in self.coeffs[h])
    x = self.x(); NO = self.f(x, self.p)
    da, dz = self._split(self.Q); T0 = self._tangent_rhs(x, da, dz) # J(x) dx
    za = E*self.z + a*NO[None, :] # predictor
    dza = E[:, :, None]*dz + a[:, :, None]*T0[None, :, :]
    xa = self.x0 + self.w @ za
    N1 = self.f(xa, self.p); T1 = self._tangent_rhs(xa, da, dza)
    self.z = za + b*(N1-NO)[None, :] # corrector
    dz = dza + b[:, :, None]*(T1-T0)[None, :, :]
    self.Q = self._join(da, dz); self.t += h
```

Two remarks on this code. The component δa is carried along unchanged, as (9) says, but it matters: it is through δa that a perturbation of the initial condition enters, and a tangent vector with $\delta a \neq 0$ is the one that, in the linear test of the next section, grows at the exact rate. And the dimension of the tangent space is $n + nJ \approx 300$ for $n = 3$ and $J \approx 100$, so the QR of a 300×3 matrix every hundred steps is a negligible cost; the program is dominated by the evaluation of f and J , and runs at a few hundred thousand steps per minute in pure `numpy`.

4.5 The run, and how to read the exponents

The function `run` assembles the pieces: it builds the modes, creates the augmented system with $m = 3$ random orthonormal tangent vectors, steps forward with the sub-steps in the first time unit, renormalises every $\Delta t_{\text{QR}} = 0.5$, and returns the running exponents $\lambda_j(t) = (\text{accumulated log})/t$ together with the sampled trajectory. It also watches for divergence, because a fractional system can have a chaotic attractor at one order and an unbounded trajectory at a slightly higher one (the Rössler system of Table 1 diverges at $\nu = 0.95$), and a run that blows up prints a message and stops rather than returning `nan`.

How should the output be read? Three things are different from the integer case, and all three follow from Section 3. First, the running exponent $\lambda_1(t)$ converges to its limit as $1/t$ with a constant that depends on how the initial tangent vector happened to be aligned; since a random vector in a 300-dimensional space has a small component along the growing direction, the “cold” average from $t = 0$ underestimates λ_1 by $|\ln p|/t$ for a long time, $p < 1$ being the initial projection. The remedy is the usual one, to average from a later time T_w , which the reader can do from the returned data: with $c_j(t) = t \lambda_j(t)$ the accumulated logarithm, the delayed estimate is $(c_j(t) - c_j(T_w))/(t - T_w)$. Second, the non-positive exponents do not converge to negative numbers but drift towards zero, typically as $\ln t/t$: they are the band at zero of the theory, and the number that one reads at finite time is not an estimate of anything but a measure of how far the transient has gone. Third, $\lambda_1 \approx 0$ can mean two different things, an equilibrium, which is visible as a constant trajectory, or a periodic window, which is visible as a periodic trajectory with a long transient; the program returns the trajectory precisely so that one can look.

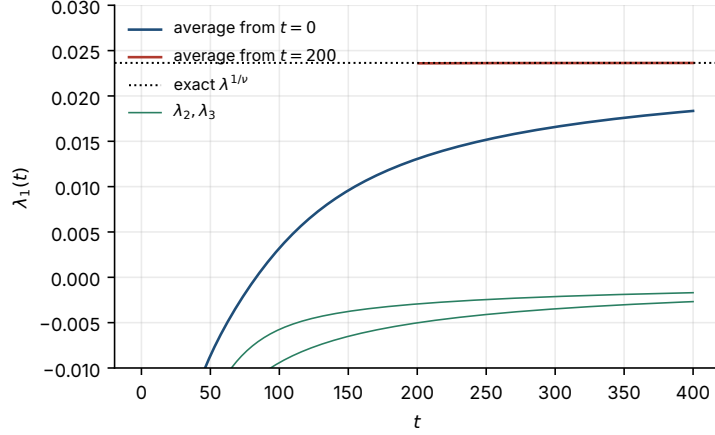


Figure 3: Linear validation, ${}^C D^\nu x = \text{diag}(0.05, -1, -2)x$, $\nu = 0.8$: running largest exponent averaged from $t = 0$ and from $t = 200$, against the exact value $0.05^{1/0.8} = 0.023644$; the two other exponents drift towards zero instead of converging to -1 and -2 .

5 Validation on a linear system

The one case in which the exponent of a fractional system is known exactly is the linear one. For ${}^C D^\nu x = \lambda x$ with $\lambda > 0$ the solution is the Mittag-Leffler function $E_\nu(\lambda t^\nu)$, which grows as $\frac{1}{\nu} \exp(\lambda^{1/\nu} t)$: the exponent is $\lambda^{1/\nu}$, not λ . For $\lambda < 0$ the solution decays as $t^{-\nu}$, algebraically, and the exponent is zero. We take $f(x) = \text{diag}(0.05, -1, -2)x$ at $\nu = 0.8$, for which the theory predicts $(\lambda_1, \lambda_2, \lambda_3) = (0.05^{1.25}, 0, 0) = (0.023644, 0, 0)$, and run the program for 400 time units (Figure 3). The cold average of λ_1 is still at 0.0183 at $t = 400$, for the alignment reason explained above; the average from $t = 200$ gives 0.023636, within 0.04% of the exact value. The other two exponents are at -0.0027 and -0.0017 and drifting towards zero as the theory says, where an ordinary integrator would return -1 and -2 . This test is worth running whenever the quadrature or the step is changed: it checks the modes, the time stepping, the tangent system and the norm at once, against an exact number.

6 Two chaotic systems

Figure 4 shows the running exponents of the Chen system at $\nu = 0.9$ over 3000 time units, of the Lorenz system at $\nu = 0.995$ over 1000, and of the Chen system at $\nu = 0.99$ over 600. The Chen system at $\nu = 0.9$ settles at $\lambda_1 = 2.33$, with fluctuations of a few per cent that are the normal statistics of a chaotic average, and with λ_2, λ_3 at $-3 \cdot 10^{-4}$; the value is a few per cent above the 2.25 of [5], obtained with a finer step and a longer run, which is the size of the step dependence discussed there. The Lorenz system at $\nu = 0.995$ gives $\lambda_1 = 0.856$, where the integer system has 0.906; the slow start of its running exponent, which stays near zero for the first thirty time units, is the trajectory lingering near the equilibria that are about to become stable, as noted in Section 4. The Chen system at $\nu = 0.99$ is the cautionary example: its running exponent decreases steadily towards zero, and the trajectory, when one plots it, is periodic after a long transient; this is a periodic window, not an equilibrium, and it is why the program returns the trajectory.

7 What to avoid

The program is simple, and most of the ways to misuse it are ways of ignoring the two facts of Section 3. The first and most consequential is to truncate the memory, either by removing

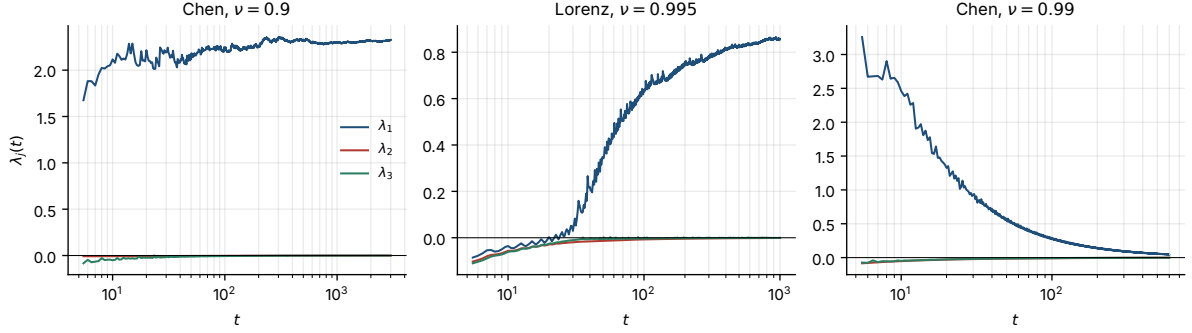


Figure 4: Running Lyapunov exponents $\lambda_j(t)$, cold average, $h = 0.005$, QR every 0.5. Left: Chen, $\nu = 0.9$, $\lambda_1 \rightarrow 2.33$. Centre: Lorenz, $\nu = 0.995$, $\lambda_1 \rightarrow 0.86$. Right: Chen, $\nu = 0.99$, a periodic window, $\lambda_1 \rightarrow 0$. In all cases λ_2, λ_3 drift to zero and never become negative.

the singular panel, or by setting $\omega_{\min}$ larger than $1/T$, or by using a fixed-memory window in a Volterra-type scheme. All three are the same operation, and Figure 1 shows what it does to the kernel; what it does to the exponents is documented in [5], where the Chen attractor computed with a window of 500 time units is inflated and its exponent reduced by twenty per cent, and the one with a window of 100 escapes to infinity. The short-memory principle of the Grünwald–Letnikov scheme, which truncates the *derivative* and not the integral, is a controlled approximation with a bounded error, of order $L^{-\nu}$ for a window L [1], rather than a change of system; it is a different operation, and the two should not be confused.

The second is to choose $\omega_{\max}$ independently of the step: the fastest modes must resolve the kernel at lag h , which means that $\omega_{\max}h$ must be large; the missing tail is of order $e^{-\omega_{\max}h}$, so $\omega_{\max} \approx 30/h$ is already enough, and the program’s $10^4/h$ is a generous margin that costs two extra panels. The error check of Figure 1 should be repeated if either is changed. The third is to expect negative exponents, or to tune the run until they appear: they will not, and a method that produces them is measuring something else. The fourth is to read $\lambda_1 \approx 0$ without looking at the trajectory. The fifth, less obvious, is to compare exponents across orders ν without remembering that the system changes with ν in more than one way: the Matignon threshold moves, periodic windows open and close, and $\lambda_1(\nu)$ is not monotone; the non-monotone dependence of the Chen exponent on ν reported in [5] is real, and is not a numerical artefact.

8 Conclusions

We have built, in about one hundred and fifty lines of Python, a program that simulates a fractional-order chaotic system and computes its Lyapunov exponents, and we have justified each of its parts by a single idea: that the state of a fractional system is its memory, and that the memory is a sum of exponential modes. The quadrature of the modes, the exponential time stepping, the tangent system and the weighted norm of its orthonormalisation are each the direct translation of one piece of the theory of [5], and the program reproduces the exact exponent of a linear system to better than 0.05%. What the reader should take away, beyond the code, is the way of reading its output: a fractional system has positive exponents that are as well defined as those of an ordinary system, and a band at zero in place of the negative ones; its memory must never be truncated; and a vanishing exponent is an invitation to look at the trajectory. The program is listed in full in the appendix and can be adapted to any system of the form ${}^C D^\nu x = f(x)$ by supplying f and its Jacobian.

A The complete program

```

"""
frac_lyap.py -- Chaotic attractors and Lyapunov exponents of fractional-order systems
(Caputo derivative of order nu in (0,1)) with the augmented-system algorithm of the
diffusive representation. Companion code of the tutorial article.


$$C-D^{\text{nu}} x = f(x), x(0) = x_0 \iff x = x_0 + \sum_i w_i z_i, \quad z_i' = -\omega_i z_i + f(x), \quad z_i(0) = 0$$


Tangent system:  $da' = 0, \quad dz_i' = -\omega_i dz_i + J(x) dx, \quad dx = da + \sum_i w_i dz_i.$ 
Norm of the memory space:  $|da|^2 + \sum_i w_i (1+\omega_i) |dz_i|^2.$ 

Usage: python frac_lyap.py --system chen --nu 0.9 --T 1000
Requires numpy and scipy (matplotlib only for the plotting script figures.py).
"""

import argparse, math, time
import numpy as np
from scipy.special import roots_jacobi, roots_legendre

# ----- 1. the systems
def lorenz(x, p):
    s, r, b = p
    return np.array([s*(x[1]-x[0]), x[0]*(r-x[2])-x[1], x[0]*x[1]-b*x[2]])
def lorenz_J(x, p):
    s, r, b = p
    return np.array([[-s, s, 0.0], [r-x[2], -1.0, -x[0]], [x[1], x[0], -b]])
def chen(x, p):
    a, b, c = p
    return np.array([a*(x[1]-x[0]), (c-a)*x[0]-x[0]*x[2]+c*x[1], x[0]*x[1]-b*x[2]])
def chen_J(x, p):
    a, b, c = p
    return np.array([[-a, a, 0.0], [c-a-x[2], c, -x[0]], [x[1], x[0], -b]])
def rossler(x, p):
    a, b, c = p
    return np.array([-x[1]-x[2], x[0]+a*x[1], b+x[2]*(x[0]-c)])
def rossler_J(x, p):
    a, b, c = p
    return np.array([[0.0, -1.0, -1.0], [1.0, a, 0.0], [x[2], 0.0, x[0]-c]])

def linear(x, p): return p @ x
def linear_J(x, p): return p
SYSTEMS = {"linear": (linear, linear_J, np.diag([0.05, -1.0, -2.0]), np.array([1.0, 1.0, 1.0])),
           "lorenz": (lorenz, lorenz_J, (10.0, 28.0, 8.0/3.0), np.array([1.0, 2.0, 0.1])),
           "chen": (chen, chen_J, (35.0, 3.0, 28.0), np.array([-9.0, -5.0, 14.0])),
           "rossler": (rossler, rossler_J, (0.5, 0.2, 10.0), np.array([1.0, 1.0, 0.1]))}

# ----- 2. the memory modes
def modes(nu, om_min, om_max, per_decade=8):
    """Nodes  $\omega_i$  and weights  $w_i$  such that  $\sum_i w_i e^{-\omega_i \tau} \sim \tau^{(\text{nu}-1)}/\Gamma(\text{nu})$ 
    for  $\tau$  in  $[h, T]$ . Singular panel  $[0, \omega_{\min}]$ : Gauss-Jacobi with weight  $\omega^{(-\text{nu})}$ .
    Geometric panels  $[\omega_{\min}, \omega_{\max}]$ , one per decade: Gauss-Legendre in  $\ln(\omega)$ ."""
    c = math.sin(math.pi*nu)/math.pi
    xj, wj = roots_jacobi(per_decade, 0.0, -nu) # singular panel
    om = [om_min*(1+xj)/2]; w = [c*(om_min/2)**(1-nu)*wj]
    xl, wl = roots_legendre(per_decade) # geometric panels
    for k in range(int(math.ceil(math.log10(om_max/om_min)))):
        u0 = math.log(om_min) + k*math.log(10); u1 = u0 + math.log(10)
        u = 0.5*(u1-u0)*xl + 0.5*(u1+u0)
        om.append(np.exp(u)); w.append(c*np.exp(u)**(1-nu)*wl*0.5*(u1-u0))
    return np.concatenate(om), np.concatenate(w)

```

```

def kernel_error(nu, om, w, taus):
    exact = taus**(nu-1)/math.gamma(nu)
    approx = np.array([np.sum(w*np.exp(-om*t)) for t in taus])
    return np.abs(approx/exact - 1.0)

# ----- 3. the time step (ETD-RK2)
def etd_coeffs(om, h):
    """For u' = -om u + N: u(t+h) = E u + a N0 + b (N1 - N0)."""
    oh = om*h; E = np.exp(-oh)
    a = np.where(oh > 1e-6, (1-E)/np.maximum(om, 1e-300), h*(1-oh/2+oh**2/6))
    b = np.where(oh > 1e-4, (E-1+oh)/np.maximum(om**2*h, 1e-300), h*(0.5-oh/6+oh**2/24))
    return E, a, b

class Augmented:
    """State: x0 (constant), z (J x n memory modes); tangent: Q (dim x n_vec) in rescaled
    coordinates."""
    def __init__(self, f, Jf, p, nu, x0, om, w, h, n_vec=3, seed=0):
        self.f, self.Jf, self.p = f, Jf, p
        self.x0 = np.array(x0, float); self.n = len(x0)
        self.om, self.w, self.h = om, w, h; self.J = len(om)
        self.z = np.zeros((self.J, self.n))
        self.s = np.sqrt(w*(1+om))[:, None] # rescaling: dzeta = s * dz ->
        Euclidean norm
        self.dim = self.n + self.n*self.J
        self.Q, _ = np.linalg.qr(np.random.default_rng(seed).standard_normal((self.dim, n_vec))
        )
        self.coeffs = {}; self.t = 0.0
    def x(self):
        return self.x0 + self.w @ self.z
    def _split(self, V):
        k = V.shape[1]
        return V[:self.n], V[self.n:].reshape(self.J, self.n, k)/self.s[:, :, None]
    def _join(self, da, dz):
        return np.vstack([da, (dz*self.s[:, :, None]).reshape(self.J*self.n, -1)])
    def _tangent_rhs(self, x, da, dz):
        dx = da + np.einsum("i,ijk->jk", self.w, dz)
        return self.Jf(x, self.p) @ dx
    def step(self, h=None):
        h = self.h if h is None else h
        if h not in self.coeffs: self.coeffs[h] = etd_coeffs(self.om, h)
        E, a, b = (c[:, None] for c in self.coeffs[h])
        x = self.x(); N0 = self.f(x, self.p)
        da, dz = self._split(self.Q); T0 = self._tangent_rhs(x, da, dz)
        za = E*self.z + a*N0[None, :] # predictor
        dza = E[:, :, None]*dz + a[:, :, None]*T0[None, :, :]
        xa = self.x0 + self.w @ za
        N1 = self.f(xa, self.p); T1 = self._tangent_rhs(xa, da, dza)
        self.z = za + b*(N1-N0)[None, :] # corrector
        dz = dza + b[:, :, None]*(T1-T0)[None, :, :]
        self.Q = self._join(da, dz); self.t += h
    def renormalize(self):
        Q, R = np.linalg.qr(self.Q); d = np.diag(R)
        self.Q = Q*np.sign(d)[None, :]
        return np.log(np.abs(d))

# ----- 4. the run
def run(system, nu, T, h=0.005, dt_qr=0.5, n_vec=3, per_decade=8, om_min=None, om_max=None,
    log_every=None):
    f, Jf, p, x0 = SYSTEMS[system]
    om_min = om_min or 0.1/T; om_max = om_max or 1e4/h
    om, w = modes(nu, om_min, om_max, per_decade)
    sim = Augmented(f, Jf, p, nu, x0, om, w, h, n_vec)
    n_qr = max(1, int(round(dt_qr/h))); n_steps = int(round(T/h))

```

```

acc = np.zeros(n_vec); times, cum, xs = [], [], []
t0 = time.time()
for k in range(1, n_steps+1):
    if sim.t < 1.0:                                     # t^nu start: 20 sub-steps in the
first time unit
        for _ in range(20): sim.step(h/20)
    else:
        sim.step()
        if not np.all(np.isfinite(sim.z)) or np.abs(sim.x()).max() > 1e6:
            print(f"divergence at t={sim.t:.2f}: the run is stopped and the partial results
returned"); break
        if k % n_qr == 0:
            acc += sim.renormalize(); times.append(sim.t); cum.append(acc.copy()); xs.append(
sim.x())
            if log_every and k % log_every == 0:
                print(f"t={sim.t:8.1f}  lambda={np.round(acc/sim.t, 4)}  ({time.time()-t0:.0f} s)",
flush=True)
times = np.array(times); cum = np.array(cum)
if len(times) == 0: return {"t": times, "lambda": cum, "x": np.array(xs), "om": om, "w": w,
"modes": len(om)}
return {"t": times, "lambda": cum/times[:, None], "x": np.array(xs), "om": om, "w": w, "
modes": len(om)}

if __name__ == "__main__":
    ap = argparse.ArgumentParser()
    ap.add_argument("--system", default="chen", choices=list(SYSTEMS))
    ap.add_argument("--nu", type=float, default=0.9)
    ap.add_argument("--T", type=float, default=1000.0)
    ap.add_argument("--h", type=float, default=0.005)
    ap.add_argument("--save", default=None)
    args = ap.parse_args()
    om, w = modes(args.nu, 0.1/args.T, 1e4/args.h)
    taus = np.logspace(math.log10(args.h), math.log10(args.T), 50)
    print(f"{len(om)} modes; max kernel error on [h,T]: {kernel_error(args.nu, om, w, taus).max
():.1e}")
    out = run(args.system, args.nu, args.T, args.h, log_every=int(100/args.h))
    print("Lyapunov exponents:", np.round(out["lambda"][-1], 4))
    if args.save: np.savez(args.save, **out)

```

References

- [1] I. Podlubny, *Fractional Differential Equations*, Academic Press, San Diego, 1999.
- [2] K. Diethelm, *The Analysis of Fractional Differential Equations*, Springer, Berlin, 2010.
- [3] I. Petráš, *Fractional-Order Nonlinear Systems: Modeling, Analysis and Simulation*, Springer, Berlin, 2011.
- [4] G. Benettin, L. Galgani, A. Giorgilli, J.-M. Strelcyn, Lyapunov characteristic exponents for smooth dynamical systems and for Hamiltonian systems; a method for computing all of them, *Meccanica* 15 (1980) 9–30.
- [5] S. Fazzino, Lyapunov exponents of fractional-order systems: a definition from the diffusive representation, the ageing of memory, and what memory truncation measures, submitted to *Commun. Nonlinear Sci. Numer. Simul.* (2026).
- [6] G. Montseny, Diffusive representation of pseudo-differential time-operators, *ESAIM: Proceedings* 5 (1998) 159–175.

- [7] D. Matignon, Stability results for fractional differential equations with applications to control processing, in *Computational Engineering in Systems Applications*, IMACS, Lille, 1996, pp. 963–968.
- [8] P. Thiellien, Fibrés dynamiques asymptotiquement compacts. Exposants de Lyapounov. Entropie. Dimension, *Ann. Inst. H. Poincaré Anal. Non Linéaire* 4 (1987) 49–97.
- [9] M.-F. Danca, N. Kuznetsov, Matlab code for Lyapunov exponents of fractional-order systems, *Int. J. Bifurc. Chaos* 28 (2018) 1850067.
- [10] S. M. Cox, P. C. Matthews, Exponential time differencing for stiff systems, *J. Comput. Phys.* 176 (2002) 430–455.