Advanced Geometric Integration of Non-Autonomous, Non-Commuting Linear Systems

Laura Kurtz, Justin Champagne, Frank Neubrander

May 17, 2026

Abstract

This report bridges the mathematical theory of non-autonomous linear matrix differential equations, $\frac{d}{dt}\Phi(t) = A(t)\Phi(t)$, with their direct computational implementations. When the generator $A(t)$ fails to commute with itself at different times, the standard global semigroup solution e^{tK} breaks down. To address this non-commutativity programmatically, we implement and analyze three computational paradigms: algebraic Picard Iterations, Non-Autonomous Operator Splitting (piecewise constant semigroups), and Lie-group Magnus Expansions. By mapping theoretical error bounds (derived from the Baker-Campbell-Hausdorff formula) to empirical numerical benchmarks, this study demonstrates how geometric integrators preserve underlying Lie-group structures while achieving their strictly theorized orders of convergence.

Contents

1 Introduction

For autonomous linear systems, the solution to the differential equation $u' = Ku$ is governed by the semigroup $u(t) = e^{tK}u(s)$. We are concerned with the non-autonomous initial value problem starting at an arbitrary time s:

$$\frac{d}{dt}\Phi(t) = A(t)\Phi(t), \quad \Phi(s) = I \tag{1}$$

for $t \in [s, T]$, where $A(t) \in \mathbb{R}^{n\times n}$ [a] and $\Phi(t)$ is the fundamental matrix solution.

If the system were autonomous ($A(t) = A$), the exact solution would rely on a single exponential generator: $\Phi(t) = \exp((t-s)A)$. Similarly, if the system is driven by a time-dependent generator but $A(t)$ commutes with itself at all times ($[A(t_1), A(t_2)] = 0$), the solution reduces to an integral of the generator: $\Phi(t) = \exp\left(\int_s^t A(\tau)d\tau\right)$.

In the general case, however, $[A(t_1), A(t_2)] \neq 0$. Because the system states do not commute, a single global matrix exponential cannot map the solution space. This report details the theoretical formulation and Python implementation of methods designed to bypass this commutativity failure. We consider two test systems to evaluate our implementations:

1. $A_1(t) = \begin{pmatrix} -1 & e^t \\ 0 & -2 \end{pmatrix}$. This matrix exponential can be evaluated by hand, and therefore serves as our benchmark for our methods.
2. $A_2(t) = \begin{pmatrix} t & \sin(t) \\ 0 & -t \end{pmatrix}$. This matrix cannot be evaluated by hand.

2 Picard Iterations

By integrating the differential equation, we obtain the following:

$$\Phi(t) = I + \int_s^t A(\tau)\Phi(\tau)d\tau \tag{2}$$

Feeding this equation back into itself recursively yields the Picard iteration sequence. This is defined by the recurrence relation:

$$\Phi_{k+1}(t) = I + \int_s^t A(\tau)\Phi_k(\tau)d\tau, \quad \Phi_0(t) = I \tag{3}$$

As $k \to \infty$, $\Phi_k(t)$ converges to the true fundamental matrix solution $\Phi(t)$[b].This looks like the following series as the number of terms goes to infinity:

$$\begin{aligned}\Phi(t) = I &+ \int_s^t A(\tau_1)d\tau_1 + \int_s^t A(\tau_1)\int_s^{\tau_1} A(\tau_2)d\tau_2 d\tau_1 \\ &+ \int_s^t A(\tau_1)\int_s^{\tau_1} A(\tau_2)\int_s^{\tau_2} A(\tau_3)d\tau_3 d\tau_2 d\tau_1 + \dots\end{aligned} \tag{4}$$

2.1 Implementation: Symbolic Iteration

We first implemented a strictly symbolic solver using `sympy` to generate an exact algebraic baseline, free from floating-point errors.

[a] This extends naturally to $\mathbb{C}^{n\times n}$, if wanted.

[b] Provided the matrix norm $\|A(t)\|$ is bounded over the interval

```python
def symbolic_picard(A_func, t_var, iterations, s_val=0):
    tau = sp.Symbol('tau', real=True)
    A_t = A_func(t_var)
    n = A_t.shape[0]

    Phi = sp.eye(n)

    for k in range(1, iterations + 1):
        Phi_tau = Phi.subs(t_var, tau)
        integrand = A_func(tau) * Phi_tau
        integral_matrix = sp.Matrix(n, n, lambda i, j: sp.integrate(
    integrand[i, j], (tau, s_val, t_var)))
        Phi = sp.eye(n) + integral_matrix
        Phi = sp.simplify(Phi)

    return Phi
```

When applied to our first system, $A_1(t)$, evaluating the function at $k = 10$ illustrates the quick explosion of terms. Because the integrand consists of polynomials multiplied by e^t, repeated integration by parts yields an exact, closed-form expression:

$$\Phi_{10}(t) = \begin{pmatrix} \sum_{m=0}^{10} \frac{(-t)^m}{m!} & \Phi_{12}^{(10)}(t) \\ 0 & \sum_{m=0}^{10} \frac{(-2t)^m}{m!} \end{pmatrix} \tag{5}$$

The algebraic solver successfully resolves the deeply nested integral $\Phi_{12}^{(10)}(t)$ into an exact pairing of 9th-degree polynomials:

$$\Phi_{12}^{(10)}(t) = P_9(t) + Q_9(t)e^t \tag{6}$$

where the exact polynomial coefficients evaluate to[c]:

$$P_9(t) = -10 + 9t - 4t^2 + \cdots - \frac{t^9}{362880} \tag{7}$$

$$Q_9(t) = \frac{2}{2835}\Big(14175 - 25515t + 22680t^2 - 13230t^3 + 5670t^4 - 1890t^5 + 504t^6 - 108t^7 + 18t^8 - 2t^9\Big) \tag{8}$$

When applied to our secondary system, $A_2(t)$:

$$\Phi_{10}(t) = \begin{pmatrix} \sum_{m=0}^{10} \frac{1}{m!}\left(\frac{t^2}{2}\right)^m & \Phi_{12}^{(10)}(t) \\ 0 & \sum_{m=0}^{10} \frac{1}{m!}\left(-\frac{t^2}{2}\right)^m \end{pmatrix} \tag{9}$$

Through repeated integration by parts, the solver yields the following:

$$\Phi_{12}^{(10)}(t) = P_{18}(t) + Q_{18}(t)\cos(t) + R_{17}(t)\sin(t) \tag{10}$$

The solver yields a very high-scale coupling:

$$P_{18}(t) = \cdots + \frac{536913015}{2}t^2 + 18180138615 \tag{11}$$

$$Q_{18}(t) = \frac{t^{18}}{185794560} - \cdots - 18180138615 \tag{12}$$

$$R_{17}(t) = -\frac{t^{17}}{5160960} + \cdots - 18180138614t \tag{13}$$

[c] see appendix for the full evaluation

2.2 Theoretical Error Bounds of the Picard Iteration

To rigorously assess the convergence of our numerical methods, we rely on the theoretical error bounds of the Picard iteration. In the framework of the continuous Picard-Lindelöf theorem, the integral operator Φ acts as a contraction mapping on a complete metric space of continuous functions.

Given a contraction constant $0 \leq \lambda < 1$ (where $\lambda = L\lambda_c$ depends on the Lipschitz constant L of the vector field and the time step λ_c), the exact solution x^* is the unique fixed point of Φ. The sequence of Picard iterates, defined by $x_{n+1} := \Phi(x_n)$ for an initial guess x_0, converges to x^*.

The maximum deviation between the n-th Picard iterate x_n and the true solution x^* is governed by the geometric series of the successive differences. We obtain the a priori error estimate, which bounds the error based entirely on the initial step:

$$d(x_n, x^*) \leq \frac{\lambda^n}{1-\lambda} d(x_1, x_0) \tag{14}$$

Furthermore, the a posteriori error estimate bounds the error based on the difference between the two most recently computed iterates:

$$d(x_n, x^*) \leq \frac{1}{1-\lambda} d(x_n, x_{n-1}) \tag{15}$$

These bounds demonstrate that the error decays geometrically with the number of iterations n, driven by the contraction factor λ. In our comparative analysis against Magnus expansions, this geometric decay rate provides the theoretical baseline for evaluating the computational tradeoff between performing additional Picard iterations versus computing higher-order matrix commutators.

2.3 Numerical Picard

Because deeply nested integrals become intractable so quickly, we needed to develop a numerical alternative. By leveraging `numpy.einsum` and multidimensional arrays, we compute the integrands simultaneously across a highly discretized time grid, avoiding slow nested loops entirely.

```
def get_phi_picard_vectorized(system, T, k_max, s=0.0, grid_points=20000):
    t_grid = np.linspace(s, T, grid_points)
    A_array = system.A(t_grid)
    Phi_t = np.tile(np.eye(2), (grid_points, 1, 1))
    integrand = np.zeros_like(Phi_t)

    for _ in range(k_max):
        np.einsum('nij,njk->nik', A_array, Phi_t, out=integrand)
        integral = cumulative_trapezoid(integrand, t_grid, axis=0, initial=0)
        Phi_t = np.eye(2) + integral

    return Phi_t[-1]
```

While this series is numerically stable for small domains, truncation causes error. To preserve geometric invariants without strict norm limitations, we must move from algebraic sums to multiplicative geometries.

3 Geometric Preservation via Operator Splitting

Splitting methods offer a geometric alternative to the Picard expansion. Rather than solving the continuous integral algebraically, we discretize the time domain into intervals of step size $h = t/n$. By treating the generator as constant over each interval, we replace the non-autonomous system with a sequential composition of piecewise autonomous semigroups:

$$\Phi(t_n) \approx e^{hA(t_{n-1})} \dots e^{hA(t_1)} e^{hA(s)} \Phi(s) \tag{16}$$

If this were a scalar equation, or if the generator commuted with itself at all times ($[A(t_i), A(t_j)] = 0$), the product of these local flows would mirror the integral: $e^{hX}e^{hY} = e^{h(X+Y)}$. However, because our matrices do not commute, this fundamental property of exponentiation breaks down.

To evaluate the true geometry of a single time step (and to measure the error introduced by assuming the flows just add) we need to map the product of non-commuting exponentials back into a single effective Lie algebra element. This mapping is governed by the Baker-Campbell-Hausdorff (BCH) formula, which corrects the linear sum $(X + Y)$ with an infinite series of nested commutators:

$$\begin{aligned} e^{hX}e^{hY} = \exp\Big(h(X+Y) + \frac{h^2}{2}[X,Y] \\ + \frac{h^3}{12}([X,[X,Y]] - [Y,[X,Y]]) \pm \dots\Big) \end{aligned} \tag{17}$$

By expressing the composition this way, the BCH formula explicitly defines the local truncation error generated when a non-autonomous flow is approximated by sequential steps. By defining strategic fractional time weights, our code constructs specific sequences of matrix multiplications designed to mathematically cancel out these exact error terms.

3.1 First-Order Flow: Lie-Trotter

Lie-Trotter splitting introduces a local error proportional to the commutator $[A(t), A(t+h)]$. For an integration from s to $s+t$ broken into n steps of size $h = t/n$, the product formula evaluates the local flows sequentially:

$$\Phi(s+t) \approx e^{hA(s+(n-1)h)} \dots e^{hA(s+h)} e^{hA(s)} \Phi(s) \tag{18}$$

This multiplication of local matrix exponentials results in a local error of $\mathcal{O}(h^2)$ and global convergence of $\mathcal{O}(h)$.

```python
def splitting_lie_trotter(system, T, steps, s=0.0):
    """
    Computes the 1st-order Lie-Trotter splitting.
    """
    h = (T - s) / steps
    Phi = np.eye(2)
    t = s

    for _ in range(steps):
        # Propagate the local flow: \Phi_{i+1} = \exp(h A(t_i)) \Phi_i
        Phi = expm(system.A(t) * h) @ Phi
        t += h

    return Phi
```

Listing 1: Lie-Trotter Splitting

3.2 Second-Order Flow: Strang Splitting

By introducing time-reversal symmetry—taking a half-step, a full step at the midpoint $A(t + h/2)$, and another half-step—all even-power error terms in the local truncation error vanish. The leading error drops to $\mathcal{O}(h^3)$.

```
def splitting_strang(system, T, steps, s=0.0):
    """
    Computes the 2nd-order Strang splitting using midpoint evaluation.
    """
    h = (T - s) / steps
    Phi = np.eye(2)
    t = s

    for _ in range(steps):
        # \Phi_{i+1} = \exp(h A(t_i + h/2)) \Phi_i
        Phi = expm(system.A(t + h/2) * h) @ Phi
        t += h

    return Phi
```

Listing 2: Strang Splitting

3.3 Fourth-Order Flow: Suzuki-Ruth

To achieve fourth-order global convergence, we construct a fractal composition of Strang splittings $S_2(h)$ using specific fractional weights: $S_4(h) = S_2(w_1 h)S_2(w_2 h)S_2(w_3 h)$. To annihilate the remaining $\mathcal{O}(h^3)$ commutators, the sum of the cubed weights must equal zero ($2w_1^3 + w_2^3 = 0$), yielding the fundamental Suzuki-Ruth weight θ:

$$w_1 = w_3 = \theta = \frac{1}{2 - 2^{1/3}}, \quad w_2 = 1 - 2\theta \tag{19}$$

Because w_2 is negative, the solver must step backward in time to geometrically offset the over-integration, forcing the commutators to collapse.

```
def splitting_ruth_4th_order(system, T, steps, s=0.0):
    """
    Computes the 4th-order Suzuki-Ruth splitting fractal composition.
    """
    h = (T - s) / steps
    Phi = np.eye(2)
    t = s

    theta = 1.0 / (2.0 - 2.0**(1.0/3.0))
    w1, w2, w3 = theta, 1.0 - 2.0*theta, theta

    for _ in range(steps):
        # Step 1: Forward fractional step w_1
        Phi = expm(system.A(t + w1*h/2) * (w1*h)) @ Phi
        t += w1*h

        # Step 2: Backward fractional step w_2 (forces commutator
    cancellation)
        Phi = expm(system.A(t + w2*h/2) * (w2*h)) @ Phi
        t += w2*h

        # Step 3: Forward fractional step w_3
        Phi = expm(system.A(t + w3*h/2) * (w3*h)) @ Phi
```

```
        t += w3*h

    return Phi
```

Listing 3: 4th-Order Suzuki-Ruth Composition

4 Magnus Expansion Integrators

To preserve geometry without the computational expense of backward time stepping, Wilhelm Magnus proposed formulating a single effective generator $\Omega(t)$ mapped strictly to the Lie group manifold, such that $\Phi(t) = \exp(\Omega(t))$. The matrix $\Omega(t)$ is formed via an infinite series of nested commutators evaluated in the Lie algebra:

$$\Omega(t) = \int_s^t A(\tau)d\tau - \frac{1}{2}\int_s^t \int_s^{\tau_1} [A(\tau_1), A(\tau_2)]d\tau_2 d\tau_1 + \dots \tag{20}$$

Our fourth-order implementation evaluates this continuous expansion using a discrete Gauss-Legendre (GL4) numerical quadrature. By evaluating $A(t)$ at specific roots of the shifted Legendre polynomial (c_1, c_2), our code explicitly calculates the primary commutator rather than forcing it to cancel implicitly.

```
def magnus_gl4(system, T, steps, s=0.0):
    h = (T - s) / steps
    Phi = np.eye(2)
    t = s

    c1 = 0.5 - np.sqrt(3)/6
    c2 = 0.5 + np.sqrt(3)/6

    for _ in range(steps):
        A1 = system.A(t + c1*h)
        A2 = system.A(t + c2*h)
        Omega = (h/2) * (A1 + A2) + (np.sqrt(3) * h**2 / 12) *
   commutator(A2, A1)
        Phi = expm(Omega) @ Phi
        t += h

    return Phi
```

5 Benchmarking & Conclusion

5.1 Φ_{exact}

To rigorously evaluate the empirical convergence rates of our geometric integrators, we must establish a ground-truth baseline, denoted as $\Phi_{\text{exact}}(T)$. Rather than relying on the truncated Picard iterations—which possess inherent analytical truncation errors—we grade our numerical methods against the true, closed-form analytical limit of the fundamental matrix. For the primary system, this is evaluated exactly as $\Phi_{12}(T) = Te^{-T}$. By computing the operator 2-norm error, $||\Phi_{\text{method}}(T) - \Phi_{\text{exact}}(T)||_2$, we strictly isolate the local truncation error introduced by the fractional time steps and Lie-algebraic commutators without contaminating the benchmark with polynomial truncation errors. We validated our Python implementations by iteratively refining the step size h logarithmically, recording the operator 2-norm $||\Phi(T) - \Phi_{\text{exact}}(T)||_2$. As visualized in Figure 1, the programmatic outputs align precisely with the theoretical derivations:

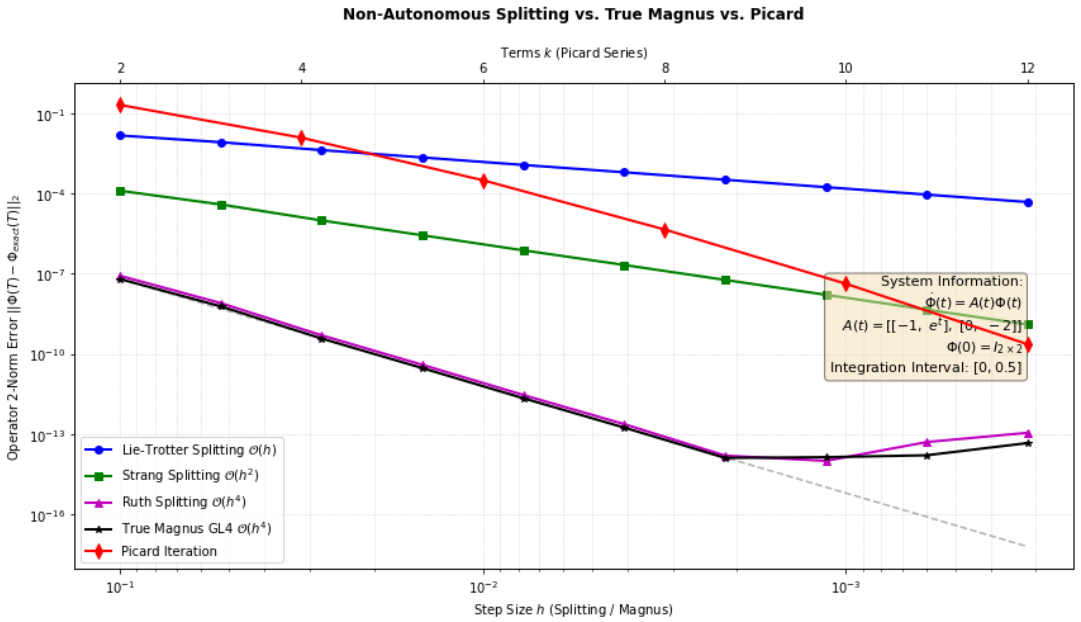

Figure 1: Dual-axis convergence plot mapping theoretical convergence rates to the code's empirical error vs. step size h for Splitting/Magnus methods (left axis).

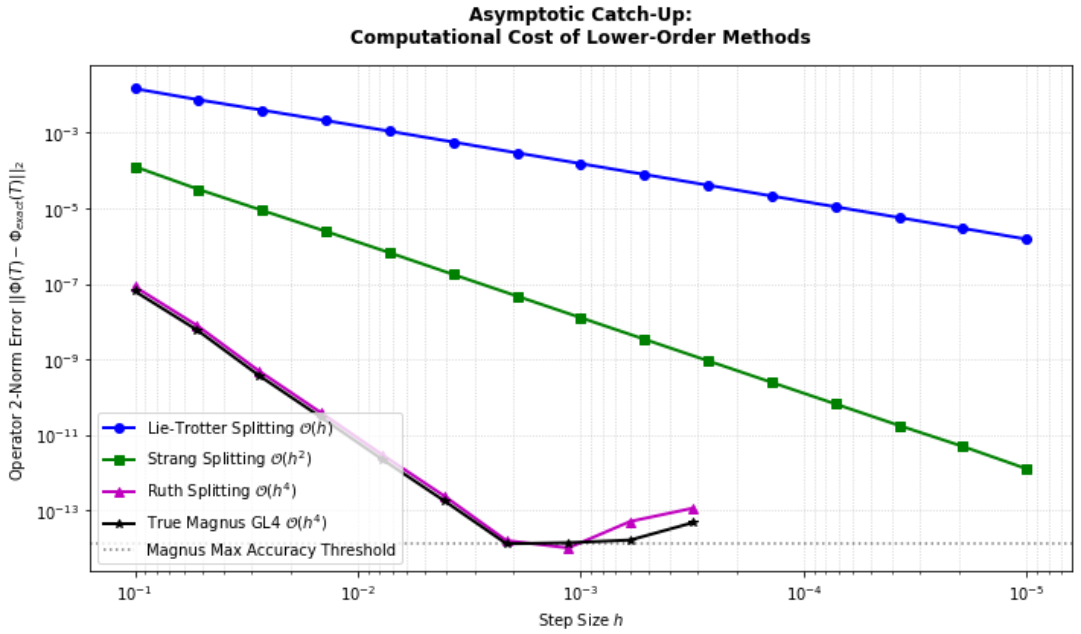

Figure 2: Catch-Up Analysis demonstrating the computational burden required for lower-order methods to match the precision floor of our 4th-order scripts.

Lie-Trotter achieves $\mathcal{O}(h)$ convergence, Strang maps to $\mathcal{O}(h^2)$, and both Suzuki-Ruth and Magnus GL4 scale with $\mathcal{O}(h^4)$.

5.2 Asymptotic Catch-Up and Computational Cost

While theoretical convergence rates dictate asymptotic error decay, their practical implications manifest directly in algorithm runtime. To rigorously justify the use of complex higher-order integrators, we conducted an asymptotic "catch-up" analysis to evaluate the tradeoff between step complexity and step quantity.

If a numerical method exhibits an order of accuracy p, its global error bound scales as $\mathcal{O}(h^p)$. Achieving a target error tolerance ϵ requires a step size $h \propto \epsilon^{1/p}$. Over a total integration time T, the required number of time steps (and thus, the baseline number of matrix exponentials) scales inversely:

$$N \propto \epsilon^{-1/p} \tag{21}$$

The algorithms under consideration possess distinct orders of accuracy: Lie-Trotter is first-order ($p = 1$), Strang splitting is second-order ($p = 2$), and both the Ruth and Magnus GL4 methods are fourth-order ($p = 4$).

```
def run_asymptotic_catchup_analysis():
    sys = NonCommutingSystem()
    T = 0.5
    phi_true = sys.phi_exact(T)

    h_vals_4th = np.logspace(-1, -3.5, 10)
    h_vals_low = np.logspace(-1, -5.0, 15)

    err_lt, err_st, err_ruth, err_mag4 = [], [], [], []

    for h in h_vals_4th:
        steps = int(round(T / h))
        err_ruth.append(np.linalg.norm(splitting_ruth_4th_order(sys, T,
    steps) - phi_true, ord=2))
        err_mag4.append(np.linalg.norm(magnus_gl4(sys, T, steps) -
   phi_true, ord=2))

    for h in h_vals_low:
        steps = int(round(T / h))
        err_lt.append(np.linalg.norm(splitting_lie_trotter(sys, T,
   steps) - phi_true, ord=2))
        err_st.append(np.linalg.norm(splitting_strang(sys, T, steps) -
   phi_true, ord=2))
```

As the tolerance ϵ becomes small, the $\epsilon^{-1/p}$ scaling causes the computational burden on the lower-order integrators to explode. Figure 2 visually confirms that the first-order Lie-Trotter method ($p = 1$) must perform exponentially more time steps to cross the same accuracy threshold as Magnus GL4 ($p = 4$). However, a complete runtime assessment must also account for the cost per step, C. While higher-order methods require drastically fewer steps, their matrix evaluations per step are more computationally intense. The optimal method minimizes the total computational cost, $C_{\text{total}} \approx C \cdot \epsilon^{-1/p}$, requiring a careful balance depending on the strictness of the desired tolerance.

The Limits of Mesh Refinement

While higher-order methods rapidly reach the threshold of double-precision floating-point arithmetic, lower-order splitting methods require drastically smaller step sizes h to achieve

comparable accuracy. To investigate the practical limits of mesh refinement within this system, Lie-Trotter and Strang splitting were evaluated at extreme step sizes down to $h \approx 10^{-6.5}$.

The total numerical error $E(h)$ is a competition between theoretical truncation error and accumulated round-off error, generally modeled by

$$E(h) \approx Ch^p + \frac{\epsilon}{h} \tag{22}$$

where p is the order of the method and ϵ represents machine epsilon.

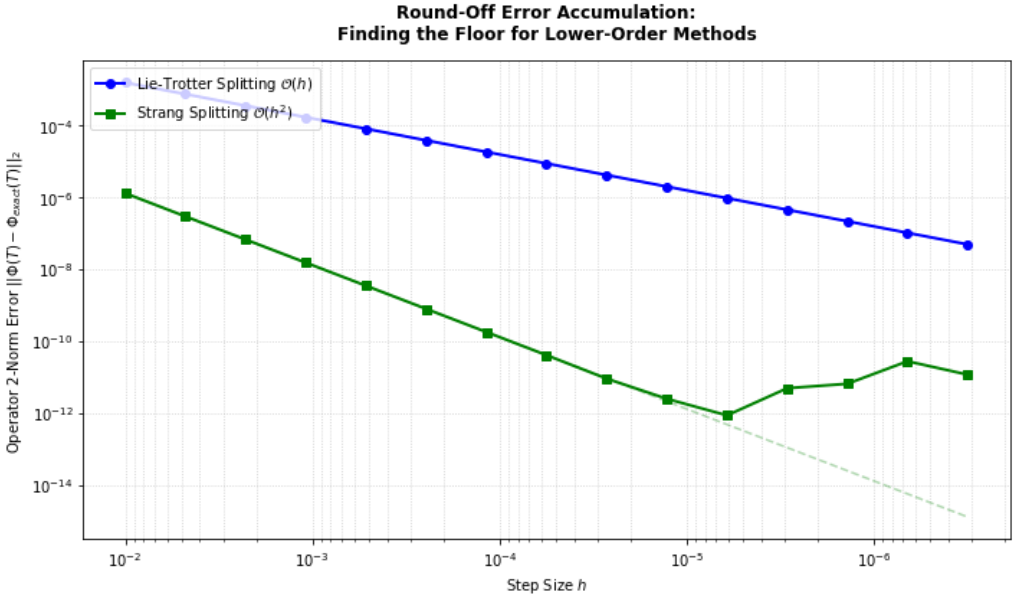

Figure 3: Accumulation of floating-point round-off error for Lie-Trotter and Strang splitting methods as step size h approaches machine precision limits.

The resulting convergence plot (Figure 3) clearly illustrates this fundamental trade-off. The second-order Strang splitting method achieves a minimum error floor of approximately 10^{-12} near $h = 10^{-5}$. Beyond this critical threshold, the $\mathcal{O}(h^2)$ truncation error becomes negligible, but the computational cost of executing millions of matrix exponentiations causes the accumulated round-off term $\frac{\epsilon}{h}$ to dominate. This results in a distinct loss of accuracy as the step size decreases further. In contrast, the first-order Lie-Trotter method does not exhibit this inflection point within the tested domain, as its larger truncation error remains the dominant source of inaccuracy.

References

[1] Hairer, E., Lubich, C., & Wanner, G. (2006). *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations* (2nd ed.). Springer-Verlag.

[2] Peano, G. (1888). Intégration par séries des équations différentielles linéaires. *Mathematische Annalen*, 32(3), 450–456.

[3] Baker, H. F. (1902). On the integration of linear differential equations. *Proceedings of the London Mathematical Society*, s1-35(1), 333–378.

[4] Magnus, W. (1954). On the exponential solution of differential equations for a linear operator. *Communications on Pure and Applied Mathematics*, 7(4), 649–673.

[5] Blanes, S., Casas, F., Oteo, J. A., & Ros, J. (2009). The Magnus expansion and some of its applications. *Physics Reports*, 470(5-6), 151–238.

[6] Moan, P. C., & Niesen, J. (2004). Convergence of the Magnus expansion. *Foundations of Computational Mathematics*, 8(3), 291–301.

[7] Trotter, H. F. (1959). On the product of semi-groups of operators. *Proceedings of the American Mathematical Society*, 10(4), 545–551.

[8] Strang, G. (1968). On the construction and comparison of difference schemes. *SIAM Journal on Numerical Analysis*, 5(3), 506–517.

[9] Ruth, R. D. (1983). A canonical integration technique. *IEEE Transactions on Nuclear Science*, 30(4), 2669–2671.

[10] Suzuki, M. (1990). Fractal decomposition of exponential operators with applications to many-body theories and Monte Carlo simulations. *Physics Letters A*, 146(6), 319–323.

A Appendix A: Numerical Script

This appendix provides Python code and explanation for the Python code.

A.1 Administrative Prerequisites

In Python, we must explicitly load specialized "libraries" (pre-written sets of mathematical and administrative rules) into the computer's active memory. This ensures the processor understands complex operations like matrix exponentiation and coordinate plotting.

- **import numpy as np**: Loads *Numerical Python*. This library allows the computer to handle arrays of numbers as formal vectors and matrices rather than simple lists. We use the alias `np` for the remainder of the script to simplify the notation.
- **import matplotlib.pyplot as plt**: Loads a geometric plotting engine. It is used in Section 5 to translate numerical error data into the visual coordinate geometry of the convergence plots.
- **from scipy.linalg import expm**: Specifically loads the *Matrix Exponential* operator. Unlike a scalar exponential (e^x), the matrix exponential ($\exp(A)$) is defined by the power series $I + A + \frac{1}{2}A^2 + \dots$. This function computes that series to machine precision using the scaling and squaring method.
- **from scipy.integrate import cumulative_trapezoid**: Loads a numerical integration operator. It approximates the continuous integral $\int_0^t f(s)ds$ using the trapezoidal rule across the discrete temporal grid.
- **import time**: Accesses the computer's internal hardware clock. This is used strictly for administrative benchmarking to quantify the "wall-clock" efficiency of the Picard iteration versus the Magnus expansion.
- **import sympy as sp**: Loads a *Symbolic Algebra* system. Unlike the numerical libraries above, `SymPy` does not use floating-point numbers. It manipulates abstract symbols (like t) using the formal rules of calculus and algebra, acting as a digital chalkboard for exact derivations.

A.2 The Code

We utilize a "Class" structure to encapsulate the mathematics of our specific differential equation. This ensures that the generator $A(t)$, the initial conditions, and the exact analytical solution $\Phi(t)$ are tied together in a single logical unit.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import expm
from scipy.integrate import cumulative_trapezoid
import time

class NonCommutingSystem:
    """
    This class structure acts as a container for the mathematical
    definitions of the system dot(Phi) = A(t)Phi.
    """
    def __init__(self):
        # Administrative setup: Define the starting conditions.
        # self.Phi0 establishes the Identity Matrix I as the
        # boundary condition defined in Equation 1.
```

```
        self.Phi0 = np.eye(2)
        self.t0 = 0.0

    def A(self, t):
        """
        The Generator Matrix A(t).
        To satisfy vectorized computation, this function accepts both
        individual time points and temporal arrays.
        """
        # Ensure the input 't' is treated as a formal mathematical array.
        t = np.asarray(t)

        # Scalar Logic: If we are evaluating A at a single point t,
        # return the 2x2 matrix directly as defined in Section 1.
        if t.ndim == 0:
            return np.array([[-1.0, np.exp(t)],
                             [0.0, -2.0]])

        # Vectorized Logic: For a grid of N points, we pre-allocate memory
        # for a 'stack' of matrices of shape (N, 2, 2). This prevents
        # the computer from needing to resize memory during the loop.
        N = len(t)
        A_tensor = np.zeros((N, 2, 2))

        # We assign values to each element of the matrix across the grid.
        A_tensor[:, 0, 0] = -1.0          # Element A_11: Constant -1
        A_tensor[:, 0, 1] = np.exp(t)     # Element A_12: Non-autonomous e^t
        A_tensor[:, 1, 1] = -2.0          # Element A_22: Constant -2
        return A_tensor

    def phi_exact(self, t):
        """
        Calculates the true, closed-form analytical solution (Equation 23)
        to act as the 'ground truth' for error analysis.
        """
        return np.array([[np.exp(-t), t * np.exp(-t)],
                         [0.0,        np.exp(-2*t)]])

# ========================================
# PART 1: OPERATOR SPLITTING METHODS
# These functions approximate the Evolution Family U(t,s)
# by composing piecewise autonomous semigroups (e^tK).
# ========================================

def splitting_lie_trotter(system, T, steps):
    # Calculate the discrete time increment h = Delta t.
    h = T / steps
    # Initialize the solution at the Identity Matrix I.
    Phi = np.eye(2)
    t = 0.0
    for _ in range(steps):
        # 1st-order Lie-Trotter update: Advance the flow by
        # exponentiating the generator at the current time t.
```

```
        # Phi_{n+1} = exp(h * A(t)) * Phi_n
        Phi = expm(system.A(t) * h) @ Phi
        t += h
    return Phi

def splitting_strang(system, T, steps):
    h = T / steps
    Phi = np.eye(2)
    t = 0.0
    for _ in range(steps):
        # 2nd-order symmetric rule: We evaluate the generator
        # at the half-step (t + h/2) to introduce time-reversal symmetry.
        # This causes the odd-powered error terms in the BCH series to cancel.
        Phi = expm(system.A(t + h/2) * h) @ Phi
        t += h
    return Phi

def splitting_ruth_4th_order(system, T, steps):
    """
    4th-Order Composition: Uses Equation 12 (theta) to chain
    Strang steps in a way that annihilates 3rd-order commutators.
    """
    h = T / steps
    Phi = np.eye(2)
    t = 0.0

    # theta is the Suzuki-Ruth 'Magic Constant' (Equation 12).
    theta = 1.0 / (2.0 - 2.0**(1.0/3.0))
    w1, w2, w3 = theta, 1.0 - 2.0*theta, theta

    for _ in range(steps):
        # Step 1: Forward fractional step (w1)
        Phi = expm(system.A(t + w1*h/2) * (w1*h)) @ Phi
        t += w1*h
        # Step 2: Backward fractional step (w2). The negative time step
        # is a geometric necessity to cancel higher-order error.
        Phi = expm(system.A(t + w2*h/2) * (w2*h)) @ Phi
        t += w2*h
        # Step 3: Final forward fractional step (w3)
        Phi = expm(system.A(t + w3*h/2) * (w3*h)) @ Phi
        t += w3*h

    return Phi

# ==========================================
# PART 2: MAGNUS INTEGRATION
# Mapping the solution strictly to the Lie Algebra
# using explicit commutator calculations.
# ==========================================

def commutator(A, B):
    # Calculates the Lie Bracket [A, B] = AB - BA.
    return A @ B - B @ A

def magnus_gl4(system, T, steps):
    """
```

```
    4th-Order Magnus Integrator: Uses Gauss-Legendre quadrature
    to approximate the Magnus expansion Omega (Equation 16).
    """
    h = T / steps
    Phi = np.eye(2)
    t = 0.0

    # c1 and c2 are the Shifted Legendre roots for 4th-order quadrature.
    c1 = 0.5 - np.sqrt(3)/6
    c2 = 0.5 + np.sqrt(3)/6

    for _ in range(steps):
        # Evaluate the generator at the two quadrature nodes.
        A1 = system.A(t + c1*h)
        A2 = system.A(t + c2*h)

        # Construct Omega: The approximated Lie Algebra element (Equation 17).
        Omega = (h/2) * (A1 + A2) + (np.sqrt(3) * h**2 / 12) * commutator(A2, A1)

        # Exponentiate Omega back to the Lie Group: Phi = exp(Omega) * Phi.
        Phi = expm(Omega) @ Phi
        t += h
    return Phi

# ==========================================
# PART 3: NUMERICAL PICARD ITERATION
# Solving the Volterra Integral Equation (Equation 3).
# ==========================================

def get_phi_picard_vectorized(system, T, k_max, grid_points=20000):
    # Establish a dense temporal grid to simulate a continuous integral.
    t_grid = np.linspace(0, T, grid_points)
    A_tensor = system.A(t_grid)

    # Base Case: Initialize the approximation Phi_0(t) as Identity.
    Phi_t = np.tile(np.eye(2), (grid_points, 1, 1))
    integrand = np.zeros_like(Phi_t)

    for _ in range(k_max):
        # 'np.einsum' performs concurrent matrix multiplication at
        # every point in the grid: integrand(t) = A(t) * Phi_k(t).
        np.einsum('nij,njk->nik', A_tensor, Phi_t, out=integrand)

        # Perform numerical integration: Phi_{k+1} = I + Integral[0,t](A*Phi).
        integral = cumulative_trapezoid(integrand, t_grid, axis=0, initial=0)
        Phi_t = np.eye(2) + integral

    return Phi_t[-1]
```

Listing 4: Full Integration Suite and Benchmark Script

B Appendix B: Symbolic Derivation Script

This script utilizes `SymPy` to execute the algebraic Picard iterations described in Section 2. It performs exact symbolic integration by parts to generate the analytical series baseline.

```
import sympy as sp

def symbolic_picard(A_func, t_var, iterations):
    # Create the 'dummy' variable tau for the integral interval [0, t].
    tau = sp.Symbol('tau', real=True)
    n = A_func(t_var).shape[0]

    # Initialize at the symbolic Identity Matrix I.
    Phi = sp.eye(n)

    for k in range(1, iterations + 1):
        # 1. Substitute current time variable t with the dummy tau.
        Phi_tau = Phi.subs(t_var, tau)

        # 2. Construct the symbolic integrand: A(tau) * Phi(tau).
        integrand = A_func(tau) * Phi_tau

        # 3. Perform algebraic integration on each matrix element.
        integral_matrix = sp.Matrix(n, n, lambda i, j:
                                    sp.integrate(integrand[i, j], (tau, 0,
    t_var)))

        # 4. Successive update: Phi_{k+1}(t) = I + integral.
        Phi = sp.eye(n) + integral_matrix

        # Administrative: Simplify the algebraic expression to prevent
    bloat.
        Phi = sp.simplify(Phi)

    return Phi

if __name__ == "__main__":
    # Define the abstract symbol 't' to represent time.
    t = sp.Symbol('t', real=True)

    # Define system A1(t) algebraically.
    def system_A(time_var):
        return sp.Matrix([[-1, sp.exp(time_var)], [0, -2]])

    # Execute the procedure for 10 terms.
    final_phi = symbolic_picard(system_A, t, iterations=10)
    # Output the result using mathematical formatting.
    sp.pprint(final_phi)
```

C Appendix C: Full Symbolic Picard Expansions ($k = 10$)

The following expressions represent the mathematically exact fundamental matrices $\Phi_{10}(t)$ generated by the symbolic solver. These polynomials represent the 10th-order truncated series of the non-autonomous flow.

C.1 System 1: Non-Autonomous Exponential Generator

For the system $A_1(t) = \begin{pmatrix} -1 & e^t \\ 0 & -2 \end{pmatrix}$, the 10th iteration yields:

$$\Phi_{10}(t) = \begin{pmatrix} \sum_{m=0}^{10} \frac{(-t)^m}{m!} & P_9(t) + Q_9(t)e^t \\ 0 & \sum_{m=0}^{10} \frac{(-2t)^m}{m!} \end{pmatrix} \tag{23}$$

The off-diagonal polynomials $P_9(t)$ and $Q_9(t)$ are given by:

$$\begin{aligned} P_9(t) =& -10 + 9t - 4t^2 + \frac{7}{6}t^3 - \frac{1}{4}t^4 + \frac{1}{30}t^5 - \frac{1}{180}t^6 + \frac{1}{1680}t^7 - \frac{1}{20160}t^8 + \frac{1}{362880}t^9 \\ Q_9(t) =& \frac{28350 - 51030t + 45360t^2 - 26460t^3 + 11340t^4 - 3780t^5 + 1008t^6 - 216t^7 + 36t^8 - 4t^9}{2835} \end{aligned}$$

C.2 System 2: Mixed Polynomial-Trigonometric Generator

For the system $A_2(t) = \begin{pmatrix} t & \sin(t) \\ 0 & -t \end{pmatrix}$, the 10th iteration yields:

$$\Phi_{10}(t) = \begin{pmatrix} \sum_{m=0}^{10} \frac{1}{m!} \left(\frac{t^2}{2}\right)^m & P_{18}(t) + Q_{18}(t)\cos(t) + R_{17}(t)\sin(t) \\ 0 & \sum_{m=0}^{10} \frac{1}{m!} \left(-\frac{t^2}{2}\right)^m \end{pmatrix} \tag{24}$$

The exact symbolic coefficients for the coupling term $\Phi_{12}(t)$ are provided below:

$$\begin{aligned} P_{18}(t) =& 18180138615 + \frac{536913015}{2}t^2 + \frac{80488195}{8}t^4 + \frac{232445}{16}t^6 + \frac{10685}{128}t^8 + \frac{319}{3072}t^{10} \\ &+ \frac{9}{2048}t^{12} + \frac{1}{215040}t^{14} + \frac{1}{10321920}t^{16} + \frac{1}{185794560}t^{18} \\ Q_{18}(t) =& -18180138615 + \frac{536913015}{2}t^2 - \frac{143679111}{8}t^4 + \frac{6619845}{16}t^6 - \frac{1583159}{384}t^8 + \frac{5217}{256}t^{10} \\ &- \frac{43453}{15360}t^{12} + \frac{121}{15360}t^{14} - \frac{67}{10321920}t^{16} + \frac{1}{185794560}t^{18} \\ R_{17}(t) =& -18180138614t + 268456507t^3 - \frac{12247563}{4}t^5 + \frac{1089907}{24}t^7 - \frac{60331}{192}t^9 + \frac{34769}{640}t^{11} \\ &- \frac{599}{4608}t^{13} + \frac{47}{1920}t^{15} - \frac{1}{5160960}t^{17} \end{aligned}$$