

A Parallel Regression Approach to ODE Integration: for Application to Relativistic Stellar Oscillations and Beyond

Martin Veselský^{1,*}

¹*Institute of Experimental and Applied Physics, Czech Technical University in Prague, Husova 240/5, 11000 Prague, Czech Republic*
(Dated: October 18, 2025)

A non-sequential, massively parallel approach to the numerical integration of ordinary differential equations (ODEs) is proposed and demonstrated. The method constructs a global regression representation of the dependent variable $F(y, t)$ in terms of orthogonal basis functions over the domain $(y_0 - q, y_0 + q) \times (t_i, t_f)$. Coefficients of this expansion are determined by minimizing the residual between the analytical time derivative $\partial F / \partial t$ and the known right-hand side $f(y, t)$. This transforms the ODE integration problem into a global optimization solvable on GPUs with large-scale parallelism. A toy model $dy/dt = -y^3 + \sin(t)$ solved using this method reproduces a reference RK4 solution with relative error of 3×10^{-4} while avoiding sequential time stepping. Including the full differential residual $\partial_t F + f \partial_y F - f = 0$ in the collocation equations yields coefficients identical (to machine precision) to those obtained by the simpler formulation thus confirming that the fixed-point polynomial approach inherently enforces the dynamical consistency without explicitly imposing it. The method offers a new path toward fully parallel ODE solvers, potentially accelerating computation in various domains such as chemical kinetics, control, climate modeling, plasma physics, modern machine learning and nuclear physics and astrophysics, in particular concerning perturbation and oscillation calculations in relativistic stellar models.

I. INTRODUCTION

Integration of ordinary differential equations (ODEs) is a fundamental problem in computational physics. Standard solvers such as Runge–Kutta (RK) or implicit methods [1, 2] are inherently sequential, limiting the exploitation of modern massively parallel hardware such as GPUs. Parallel-in-time algorithms such as Parareal attempt partial parallelization [3], but only an approach that removes time-marching altogether enables maximal concurrency on GPUs. This limitation resulting from sequential time-marching is particularly severe in problems where many ODE systems must be solved repeatedly, in typical domains such as chemical kinetics, control, climate modeling, plasma physics, and modern machine learning (ML) featuring Neural ODEs. In nuclear physics and astrophysics such methods are used among other in the calculation of non-radial oscillation spectra of relativistic stars [5, 6], which is a topic of interest at present, following the observation of gravitational waves [7, 8] and construction of next generation of gravitational wave observatories such as Einstein telescope [9] or LISA [10]. In the present work, we propose and test a novel approach where the ODE integration problem is reformulated as a global regression task. The resulting “parallel regression integrator” evaluates the full time-evolution map without sequential stepping and is thus naturally suited to GPU execution.

II. METHOD: POLYNOMIAL FIXED-POINT COLLOCATION (PARODE)

We consider the general first-order ordinary differential equation

$$\frac{dy}{dt} = f(y, t), \quad y(t_0) = y_0, \quad (1)$$

and seek its solution in the form of an explicit algebraic mapping

$$F(y, t) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} a_{mn} y^m t^n, \quad (2)$$

with coefficients a_{mn} determined from collocation conditions. The physical trajectory of the system is then defined by the fixed-point condition

$$F(y, t) = y, \quad (3)$$

which, when satisfied for all t , represents a self-consistent solution of the ODE.

* Martin.Veselsky@cvut.cz

A. Collocation formulation

At a set of K random collocation points (y_k, t_k) within a sampling domain $(y_0 - \Delta y, y_0 + \Delta y) \times (t_0, t_f)$, the temporal derivative of the polynomial ansatz satisfies

$$\frac{\partial F}{\partial t}(y_k, t_k) = \sum_{m,n} a_{mn} y_k^m n t_k^{n-1} \approx f(y_k, t_k). \quad (4)$$

This defines a linear system

$$B a \approx f, \quad (5)$$

where the rectangular collocation matrix B has dimensions $(K, M \times N)$ and encodes the derivatives of the basis functions with respect to t .

To guarantee the correct boundary condition, the algebraic constraint

$$F(y_0, t_0) = \sum_{m,n} a_{mn} y_0^m t_0^n = y_0 \quad (6)$$

is added as a weighted equation to the normal system. The final linear problem to be solved for the coefficient vector a reads

$$(B^T B + \lambda I + w_{\text{IC}} \phi_{\text{IC}} \phi_{\text{IC}}^T) a = B^T f + w_{\text{IC}} \phi_{\text{IC}} y_0, \quad (7)$$

where ϕ_{IC} is the vector of basis functions evaluated at (y_0, t_0) , λ is a small Tikhonov regularization parameter, and w_{IC} is a large weight enforcing the initial condition.

B. Fixed-point Newton iteration

Once the coefficients a_{mn} are obtained, the polynomial map $F(y, t)$ is known analytically for all (y, t) . The physical trajectory $y(t)$ is recovered as the root of the fixed-point equation $F(y, t) - y = 0$. This is achieved by a parallel Newton iteration over the entire t -grid:

$$G(y, t) = F(y, t) - y, \quad (8)$$

$$y^{(i+1)}(t) = y^{(i)}(t) - \alpha \frac{G(y^{(i)}, t)}{\partial_y G(y^{(i)}, t)}, \quad (9)$$

where $\alpha < 1$ is a damping factor. All grid points t_j are updated simultaneously on the GPU, which eliminates the sequential dependency of traditional time-stepping solvers.

C. Interpretation

The collocation step ensures that the partial derivative $\partial_t F$ matches the prescribed right-hand side $f(y, t)$ in a least-squares sense throughout the sampled domain. By fitting $\frac{\partial F}{\partial t} \approx \frac{dy}{dt}$, we implicitly constrain $\frac{\partial F}{\partial y} \approx 1$ through the shared coefficient structure, mimicking the values of total derivations of the true solution along its trajectory. Because $\partial_t F$ and $\partial_y F$ are coupled through the same coefficient tensor a_{mn} , the method implicitly satisfies

$$\frac{\partial F}{\partial t} = f(y, t), \quad \frac{\partial F}{\partial y} \approx 1, \quad (10)$$

near the trajectory, guaranteeing that $F(y, t) \approx y$ is maintained across the domain once the initial condition holds. The fixed-point iteration then enforces global consistency between F and y . This explains why the simplified formulation—based solely on the $\partial_t F = f$ collocation—produces results identical to a full residual minimization of $\partial_t F + f \partial_y F - f = 0$. Including the full differential residual in the collocation equations yields coefficients identical (to machine precision) to those obtained by the simpler formulation. This is a confirmation that the fixed-point polynomial approach inherently enforces the dynamical consistency without explicitly imposing it.

D. Algorithm summary

1. Choose polynomial orders (M, N) , number of collocation points K , and time grid $\{t_j\}$.
2. Generate random samples (y_k, t_k) and evaluate $f(y_k, t_k)$.
3. Build basis matrices and assemble the normal equations.
4. Add the boundary constraint $F(y_0, t_0) = y_0$ with large weight w_{IC} .
5. Solve for coefficients a_{mn} using GPU linear algebra.
6. Evaluate $F(y, t)$ and apply parallel Newton iteration for $F(y, t) - y = 0$.

E. Computational characteristics

The method eliminates sequential integration entirely: all operations (sampling, linear solve, evaluation, and Newton updates) are expressed as dense GPU tensor operations of dimension (K, M, N) . For moderate basis sizes $(M, N \leq 15)$ and up to 10^6 samples, the full solution of the ODE over hundreds of time points is obtained within milliseconds on a single RTX 4080 GPU.

III. RESULTS

The method was implemented in `PyTorch` framework [11] with double precision arithmetic and executed on an NVIDIA GPU. We tested the prototype on the equation $dy/dt = -y^3 + \sin(t)$, $y(0) = 0.5$, over $t \in [0, 0.1]$. A monomial $M \times N$ basis was used. In particular, five configurations were subsequently run:

1. 5×5 basis, $K = 3000$ random samples,
2. 10×10 basis, $K = 3000$ random samples,
3. 5×5 basis, $K = 50000$ random samples,
4. 5×5 basis, $K = 200000$ random samples,
5. 10×10 basis, $K = 200000$ random samples.

The regression coefficients converged within 15 Newton iterations. Figure 1 shows the RK4 reference and the three regression solutions approaching RK4 solution. Increasing basis size and sample count consistently improves agreement with RK4. This reflects the crucial point of this method which is precision of the parallel regression.

Table I compares the fixed-point regression results with a reference RK4 solver.

t	y_{RK4}	F_{reg}
0.000	0.5000	0.5000
0.010	0.4988	0.4989
0.050	0.4951	0.4955
0.090	0.4931	0.4936

TABLE I. Comparison of regression-based solver with 10×10 basis and 200000 random samples with RK4 reference for $dy/dt = -y^3 + \sin(t)$. Relative L_2 error: 3×10^{-4} .

The runtime for the regression method on GPU was 0.04 s, significantly shorter than any sequential RK integration over comparable sampling points. The approach scales trivially with batch size, enabling simultaneous solution of thousands of IVPs.

Table II shows convergence behavior. For this purpose the number of collocation samples was increased to 800000 for 10×10 basis and subsequently convergence for this number of samples was again achieved using 15×15 basis. For each base there appears to exist optimum set of collocation samples, where the method converges fast, while further increase of the number leads to problem with convergence, possibly due to imbalance with the number of available coefficients a_{ij} . As the Table II demonstrates, increasing the basis dimension (from 5×5 to 15×15) while finding optimum sample set dramatically improves accuracy: with 15×15 and $K=8 \times 10^5$ the solver converges in 15 Newton iterations and achieves relative L_2 error 3×10^{-5} . Slow or oscillatory Newton behaviour (200 iterations, i.e. the iteration cap) indicates an under-expressive basis for the sampled region; increasing basis size or improving sampling brings fast convergence and small residuals.

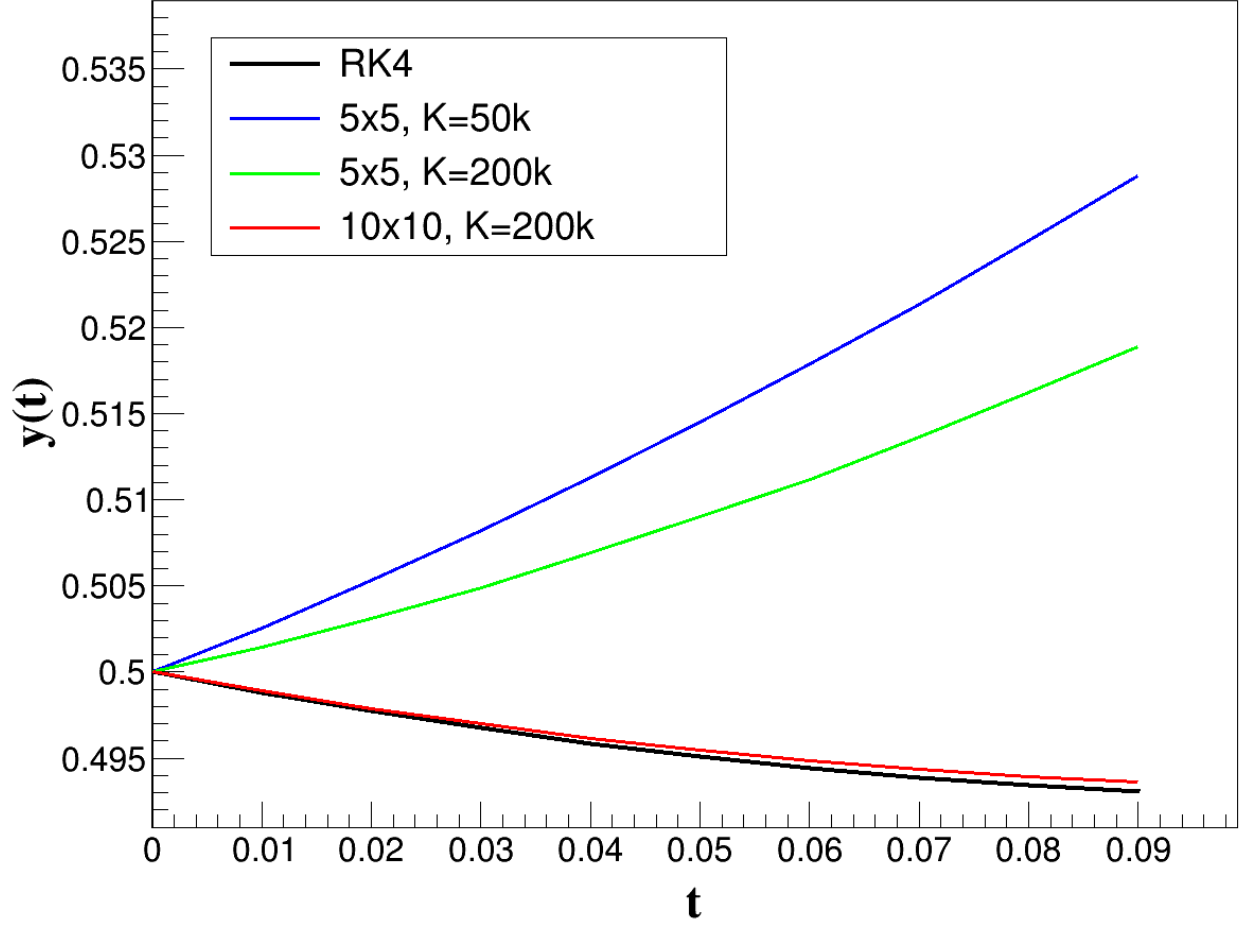


FIG. 1. Convergence: RK4 reference (black) and three regression-based solutions (colored). Larger basis and larger sample budgets progressively reduce discrepancy.

TABLE II. Convergence diagnostics for the ParODE fixed-point solver (toy ODE $\dot{y} = -y^3 + \sin t$, $y(0) = 0.5$). Shown are basis sizes $M \times N$ (number of coefficients $N_b = M \cdot N$), number of collocation samples K , initial condition enforcement weight, Newton iterations (or max iters), wall-clock time for the Newton loop, relative L_2 error vs RK4, finite-difference residual $\|dF/dt - f\|$ and the enforced IC residual.

$M \times N$	N_b	K	IC wgt	Newton iters	Time (s)	Rel. L_2 error	Residual $\ dF/dt - f\ $	IC residual
5×5	25	3×10^3	10^3	200 (max)	0.122	4.28×10^{-1}	4.55×10^{-1}	~ 0
5×5	25	5×10^4	10^3	200 (max)	0.124	1.168×10^{-1}	2.9074	~ 0
10×10	100	2×10^5	10^3	15 (conv.)	0.0396	3.28×10^{-4}	5.167×10^{-2}	2.22×10^{-16}
10×10	100	8×10^5	10^3	200 (max)	0.154	3.235×10^{-2}	3.6785	1.11×10^{-16}
15×15	225	8×10^5	10^3	15 (conv.)	0.0380	3.02×10^{-5}	3.9813×10^{-2}	1.53×10^{-13}

IV. APPLICATIONS

This method is particularly attractive for problems involving large ensembles of ODEs, occurring in many scientific fields as listed above. In nuclear physics and astrophysics can be mentioned :

- relativistic stellar perturbations and oscillation spectra (f-, p-, g-modes),
- slow-rotation corrections in the Hartle–Thorne framework,

- tidal deformability calculations in binary neutron stars.

By removing the sequential dependence in time, the entire system can be mapped onto GPU-accelerated tensor operations. With increasing performance of parallel hardware this method has potential to outperform and replace existing methods. Performance of the method can be further improved by using Chebyshev basis.

V. CONCLUSIONS

We presented a proof-of-concept for a global regression formulation of ODE integration, avoiding time marching and thus allowing massively parallel evaluation of dynamical systems. The method reproduces RK4 results for a simple test equation with high accuracy while leveraging full GPU parallelism. Extension to vector ODEs governing relativistic stellar perturbations is underway. Applications in other fields such as chemical kinetics, control, climate modeling, plasma physics, modern ML are foreseen.

ACKNOWLEDGMENTS

The author used OpenAI’s ChatGPT (GPT-5, 2025) to assist with language refinement, code translation, and discussion of numerical methods. All conceptual and scientific content is the author’s own. Computations reported here were performed using an NVIDIA GeForce RTX 4080 SUPER.

-
- [1] J. C. Butcher, Numerical Methods for Ordinary Differential Equations (Wiley, 2008).
 - [2] E. Hairer, S. P. Nørsett, and G. Wanner, Solving Ordinary Differential Equations I: Nonstiff Problems (Springer, 1993).
 - [3] J.-L. Lions, Y. Maday, and G. Turinici, A ‘parareal’ in time discretization of pdes, Comptes Rendus de l’Académie des Sciences (2001).
 - [4] T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. K. Menon, Neural ordinary differential equations, in NeurIPS (2018).
 - [5] S. Detweiler and L. Lindblom, Astrophys. J. **292**, 12 (1985).
 - [6] C. Krüger, Ph.D. thesis, University of Southampton (2015).
 - [7] LIGO Scientific, Virgo Collaboration, B. P. Abbott et al., “Observation of Gravitational Waves from a Binary Black Hole Merger,” Phys. Rev. Lett. 116 no. 6, (2016) 061102, arXiv:1602.03837 [gr-qc].
 - [8] B. P. Abbott et al., “GW170817: Observation of Gravitational Waves from a Binary Neutron Star Inspiral,” Phys. Rev. Lett. 119 no. 16, (2017) 161101, arXiv:1710.05832 [gr-qc].
 - [9] M. Punturo et al., “The Einstein Telescope: A third-generation gravitational wave observatory,” Class. Quant. Grav. 27 (2010) 194002.
 - [10] Monica Colpi et al., “LISA Definition Study Report”, arXiv:2402.07571 [astro-ph.CO].
 - [11] Paszke, A. et al., 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Advances in Neural Information Processing Systems 32. Curran Associates, Inc., pp. 8024–8035.

Appendix A: Appendix: Python implementation (monomial basis, M=N=10, K=200000)

The script below is the version used to produce the final (10×10, 200k) run. It uses monomial basis expansions, Monte-Carlo sampling in the domain $y \in [y_0 - \delta, y_0 + \delta]$, and parallel Newton solves for all time nodes. Class ParODE represents the simpler method of parallel ODE solution while ParODE2 is PDE-residual augmented. Results of both methods are identical.

```
# parode_class_demo2.py
# ParODE: GPU-parallel fixed-point ODE solver
# ParODE2: GPU-parallel fixed-point ODE solver (PDE-residual augmented)
# Author: Martin Veselsky
# Date: 2025
# See also https://github.com/mv000001/ParODE

import torch, time

class ParODE:
    def __init__(self, M=10, N=10, K=200000, ic_weight=1e3, reg_lambda=1e-8,
                 max_newton_iters=200, newton_tol=1e-10, damping=0.8,
                 device=None, dtype=torch.float64):
        """
        Initialize ParODE solver.
        Args:
            M, N          : polynomial basis orders in y and t
            K              : number of random collocation samples
```

```

        ic_weight      : weight for enforcing  $F(y_0, t_0) = y_0$ 
        reg_lambda     : Tikhonov regularization parameter
        max_newton_iters, newton_tol, damping : Newton solver controls
        device, dtype : torch device and dtype
    """
    self.M, self.N, self.K = M, N, K
    self.ic_weight = ic_weight
    self.reg_lambda = reg_lambda
    self.max_newton_iters = max_newton_iters
    self.newton_tol = newton_tol
    self.damping = damping
    self.device = device or torch.device("cuda" if torch.cuda.is_available() else "cpu")
    self.dtype = dtype

def _build_basis(self, ys, ts):
    """Build polynomial basis and its partial derivatives."""
    M, N, K = self.M, self.N, self.K
    Phi = torch.stack([ys**m for m in range(M)], dim=1)
    Tpow = torch.stack([ts**n for n in range(N)], dim=1)
    dPsi = torch.zeros((K, N), device=self.device, dtype=self.dtype)
    if N > 1:
        for n in range(1, N):
            dPsi[:, n] = n * ts**(n-1)
    dPhi = torch.zeros((K, M), device=self.device, dtype=self.dtype)
    if M > 1:
        for m in range(1, M):
            dPhi[:, m] = m * ys**(m-1)
    return Phi, Tpow, dPhi, dPsi

def _assemble_matrix(self, ys, ts, s_k, Phi, Tpow, dPhi, dPsi):
    """Assemble collocation matrix B."""
    M, N, K = self.M, self.N, self.K
    B = torch.zeros((K, M*N), device=self.device, dtype=self.dtype)
    col = 0
    for m in range(M):
        pm = Phi[:, m]
        dpm = dPhi[:, m]
        for n in range(N):
            B[:, col] = pm * dPsi[:, n] + dpm * Tpow[:, n] * s_k
            col += 1
    return B

def _evaluate_F_and_dFdy(self, y_vec, t_vec, A):
    """Evaluate  $F(y, t)$  and  $dF/dy$  for vectors  $y\_vec, t\_vec$ ."""
    M, N = self.M, self.N
    y_pows = torch.stack([y_vec**m for m in range(M)], dim=1)
    t_pows = torch.stack([t_vec**n for n in range(N)], dim=1)
    YT = y_pows.unsqueeze(2) * t_pows.unsqueeze(1)
    F = (YT * A.unsqueeze(0)).sum(dim=(1, 2))
    if M > 1:
        y_pows_m1 = torch.stack([y_vec**(m-1) if m>=1 else torch.zeros_like(y_vec)
                                for m in range(M)], dim=1)
        m_idx = torch.arange(M, device=A.device, dtype=A.dtype).unsqueeze(1)
        mA = (m_idx * A)
        dF = (y_pows_m1.unsqueeze(2) * t_pows.unsqueeze(1) * mA.unsqueeze(0)).sum(dim=(1, 2))
    else:
        dF = torch.zeros_like(y_vec)
    return F, dF

def solve(self, func, y0, t_grid):
    """
    Solve ODE  $dy/dt = func(y, t)$  from  $t\_grid[0]$  to  $t\_grid[-1]$ .
    Returns  $y(t\_grid)$  predicted by the fixed-point polynomial approach.
    """
    device, dtype = self.device, self.dtype
    y0 = torch.as_tensor(y0, device=device, dtype=dtype)
    t0, tf = t_grid[0].item(), t_grid[-1].item()
    print(f"Device: {device}, dtype: {dtype}")
    print(f"Basis: {self.M} x {self.N}, samples K={self.K}")

    # --- Collocation sampling ---
    torch.manual_seed(0)
    delta_y = 0.8
    ys_samples = (y0.item() - delta_y) + 2*delta_y * torch.rand(self.K, device=device, dtype=dtype)
    ts_samples = t0 + (tf - t0) * torch.rand(self.K, device=device, dtype=dtype)
    s_k = func(ys_samples, ts_samples)

    # --- Basis and collocation matrix ---
    Phi, Tpow, dPhi, dPsi = self._build_basis(ys_samples, ts_samples)
    B = self._assemble_matrix(ys_samples, ts_samples, s_k, Phi, Tpow, dPhi, dPsi)
    BTB = B.T @ B
    rhs = B.T @ s_k

```

```

# --- Add initial condition constraint F(y0,t0)=y0 ---
phi_ic = torch.zeros((self.M*self.N), device=device, dtype=dtype)
col = 0
for m in range(self.M):
    for n in range(self.N):
        phi_ic[col] = (y0**m) * (t0**n)
        col += 1
BTB = BTB + self.ic_weight * (phi_ic.unsqueeze(1) @ phi_ic.unsqueeze(0))
rhs = rhs + self.ic_weight * phi_ic * y0

# --- Regularize and solve for coefficients a ---
I = torch.eye(self.M*self.N, device=device, dtype=dtype)
a = torch.linalg.solve(BTB + self.reg_lambda * I, rhs)
A = a.view(self.M, self.N)

# --- Newton fixed-point iteration (fully parallel over t_grid) ---
L = len(t_grid)
y_batch = torch.full((L,), y0.item(), device=device, dtype=dtype)
start = time.time()
for it in range(self.max_newton_iters):
    Fv, dFdy = self._evaluate_F_and_dFdy(y_batch, t_grid, A)
    G = Fv - y_batch
    dG = dFdy - 1.0
    denom = torch.where(dG.abs() < 1e-12, torch.sign(dG)*1e-12 + 1e-12, dG)
    delta = G / denom
    y_new = y_batch - self.damping * delta
    max_change = torch.max(torch.abs(y_new - y_batch)).item()
    rms_resid = torch.sqrt(torch.mean(G**2)).item()
    y_batch = y_new
    if (it % 10) == 0 or it == 0:
        print(f"Iter {it:3d}: max_change={max_change:.3e}, rms_resid={rms_resid:.3e}")
    if max_change < self.newton_tol:
        print(f"Converged at iter {it}, max_change={max_change:.3e}")
        break
print(f"Newton loop time: {time.time()-start:.3f}s")

return y_batch, A

def compare_to_rk4(self, func, y0, t_grid, y_pred):
    """Compare the ParODE solution to RK4 reference and print diagnostics."""
    def rk4_integrate(f, y0, t_grid):
        y = y0.clone()
        ys = [y.clone()]
        for i in range(len(t_grid) - 1):
            t = t_grid[i]
            dt = t_grid[i + 1] - t_grid[i]
            k1 = f(y, t)
            k2 = f(y + 0.5 * dt * k1, t + 0.5 * dt)
            k3 = f(y + 0.5 * dt * k2, t + 0.5 * dt)
            k4 = f(y + dt * k3, t + dt)
            y = y + dt * (k1 + 2*k2 + 2*k3 + k4) / 6.0
            ys.append(y.clone())
        return torch.stack(ys, dim=0)

    y_ref = rk4_integrate(func, torch.tensor(y0, device=self.device, dtype=self.dtype), t_grid)
    rel_err = torch.norm(y_ref - y_pred) / torch.norm(y_ref)
    dt = t_grid[1] - t_grid[0]
    dydt_fd = torch.empty_like(y_pred)
    dydt_fd[:-1] = (y_pred[1:] - y_pred[:-1]) / dt
    dydt_fd[-1] = dydt_fd[-2]
    residual = torch.norm(dydt_fd - func(y_pred, t_grid))

    print("\nDiagnostics vs RK4:")
    print(f"Relative L2 error: {rel_err.item():.3e}")
    print(f"Residual norm ||d/dt F - f||: {residual.item():.3e}")

    print("\n t      RK4_ref      F_fixedpt")
    for i in range(0, min(10, len(t_grid))):
        print(f"{t_grid[i].item():6.3f} {y_ref[i].item():16.10e} {y_pred[i].item():16.10e}")

    return rel_err.item(), residual.item()

class ParODE2:
    def __init__(self, M=10, N=10, K=200000, ic_weight=1e3, reg_lambda=1e-8,
                  max_newton_iters=200, newton_tol=1e-10, damping=0.8,
                  device=None, dtype=torch.float64):
        self.M, self.N, self.K = int(M), int(N), int(K)
        self.ic_weight = float(ic_weight)
        self.reg_lambda = float(reg_lambda)
        self.max_newton_iters = int(max_newton_iters)
        self.newton_tol = float(newton_tol)
        self.damping = float(damping)
        self.device = device or torch.device("cuda" if torch.cuda.is_available() else "cpu")

```

```

self.dtype = dtype

def _sample_collocation(self, y0, t0, tf):
    """Return ys_samples (K,), ts_samples (K,)."""
    # deterministic seed for reproducibility
    torch.manual_seed(0)
    delta_y = 0.8
    ys = (float(y0) - delta_y) + 2.0 * delta_y * torch.rand(self.K, device=self.device, dtype=self.dtype)
    ts = float(t0) + (float(tf) - float(t0)) * torch.rand(self.K, device=self.device, dtype=self.dtype)
    return ys, ts

def _build_power_matrices(self, ys, ts):
    """
    Build y_pows: (K, M) with  $y^m$  and t_pows: (K, N) with  $t^n$ ,
    and their partial derivatives dPhi (K,M) and dTpow (K,N) for  $m \geq 1, n \geq 1$ .
    """
    M, N = self.M, self.N
    # powers
    y_pows = torch.stack([ys**m for m in range(M)], dim=1) # (K, M)
    t_pows = torch.stack([ts**n for n in range(N)], dim=1) # (K, N)
    # derivatives: d/dy  $y^m = m y^{m-1}$ , with  $m=0 \rightarrow 0$ 
    dPhi = torch.zeros_like(y_pows)
    if M > 1:
        m_idx = torch.arange(M, device=self.device, dtype=self.dtype)
        # for  $m \geq 1$ :  $m * y^{m-1}$ 
        dPhi[:, 1:] = (m_idx[1:].unsqueeze(0) * y_pows[:, :-1])
    # derivatives: d/dt  $t^n = n t^{n-1}$ ,  $n=0 \rightarrow 0$ 
    dTpow = torch.zeros_like(t_pows)
    if N > 1:
        n_idx = torch.arange(N, device=self.device, dtype=self.dtype)
        dTpow[:, 1:] = (n_idx[1:].unsqueeze(0) * t_pows[:, :-1])
    return y_pows, t_pows, dPhi, dTpow

def _assemble_pde_matrix(self, y_pows, t_pows, dPhi, dTpow, f_k):
    """
    Build PDE collocation matrix P (K, M*N) where each row is:
    row = (partial_t basis flattened) + f_k * (partial_y basis flattened)
    and RHS b_pde = f_k.
    Implementation is fully vectorized using broadcasting.
    """
    K = y_pows.shape[0]
    M, N = self.M, self.N

    # B_dt entries:  $y^m * (n * t^{n-1}) \rightarrow \text{shape} (K, M, N)$ 
    B_dt = (y_pows.unsqueeze(2) * dTpow.unsqueeze(1)) # (K, M, N)

    # B_dy entries:  $(m * y^{m-1}) * t^n \rightarrow \text{shape} (K, M, N)$ 
    B_dy = (dPhi.unsqueeze(2) * t_pows.unsqueeze(1)) # (K, M, N)

    # flatten columns in same ordering as A.view(M,N): m-major then n
    P = (B_dt + f_k.unsqueeze(1).unsqueeze(1) * B_dy).reshape(K, M * N) # (K, M*N)

    b = f_k # (K,)
    return P, b

def _assemble_ic_row(self, y0, t0):
    """Return phi_ic (M*N) with entries  $y0^m * t0^n$ """
    M, N = self.M, self.N
    phi_ic = torch.empty((M * N), device=self.device, dtype=self.dtype)
    col = 0
    for m in range(M):
        ym = (y0**m)
        for n in range(N):
            phi_ic[col] = ym * (t0**n)
            col += 1
    return phi_ic

def _evaluate_F_and_dFdy(self, y_vec, t_vec, A):
    """Vectorized eval of F(y,t) and dF/dy for vectors y_vec, t_vec and coefficient matrix A (M,N)."""
    M, N = self.M, self.N
    # y_vec shape (L,), t_vec (L,)
    y_pows = torch.stack([y_vec**m for m in range(M)], dim=1) # (L,M)
    t_pows = torch.stack([t_vec**n for n in range(N)], dim=1) # (L,N)
    # F:  $\sum_{m,n} a_{m,n} y^m t^n$ 
    F = (y_pows.unsqueeze(2) * t_pows.unsqueeze(1) * A.unsqueeze(0)).sum(dim=(1,2))
    # dF/dy:  $\sum_{m \geq 1, n} m a_{m,n} y^{m-1} t^n$ 
    if M > 1:
        # build dPhi (L,M) for current y_vec
        dPhi_local = torch.zeros_like(y_pows)
        m_idx = torch.arange(M, device=self.device, dtype=self.dtype)
        if M > 1:
            dPhi_local[:, 1:] = m_idx[1:].unsqueeze(0) * y_pows[:, :-1]
        dF = (dPhi_local.unsqueeze(2) * t_pows.unsqueeze(1) * A.unsqueeze(0)).sum(dim=(1,2))

```

```

else:
    dF = torch.zeros_like(y_vec)
    return F, dF

def solve(self, func, y0, t_grid):
    """
    Solve ODE dy/dt = func(y,t) on t_grid by the ParODE method augmented
    with the PDE residual term. Returns y_pred (L,) and coefficient matrix A (M,N).
    """
    device, dtype = self.device, self.dtype
    y0 = torch.as_tensor(float(y0), device=device, dtype=dtype)
    t0, tf = float(t_grid[0].item()), float(t_grid[-1].item())

    print(f"Device: {device}, dtype: {dtype}")
    print(f"Basis: {self.M} x {self.N}, samples K={self.K}")

    # --- sampling collocation points and evaluate f ---
    ys_samples, ts_samples = self._sample_collocation(y0, t0, tf)
    # ensure func accepts tensors and returns tensor of shape (K,)
    f_k = func(ys_samples, ts_samples).to(device=device, dtype=dtype)

    # --- power matrices and PDE matrix P ---
    y_pows, t_pows, dPhi, dTpow = self._build_power_matrices(ys_samples, ts_samples)
    P, b_pde = self._assemble_pde_matrix(y_pows, t_pows, dPhi, dTpow, f_k)

    # Normal equations for PDE-fitting: P^T P a = P^T b
    PT_P = P.t() @ P # (Nb, Nb)
    PT_b = P.t() @ b_pde # (Nb,)

    # --- IC constraint: add weighted outer-product phi_ic phi_ic^T and rhs contribution
    phi_ic = self._assemble_ic_row(y0, t0) # (Nb,)
    PT_P = PT_P + self.ic_weight * (phi_ic.unsqueeze(1) @ phi_ic.unsqueeze(0))
    PT_b = PT_b + self.ic_weight * (phi_ic * y0)

    # --- regularize and solve linear system ---
    Nb = self.M * self.N
    I = torch.eye(Nb, device=device, dtype=dtype)
    a = torch.linalg.solve(PT_P + self.reg_lambda * I, PT_b) # (Nb,)
    A = a.view(self.M, self.N)

    # --- Newton fixed-point iteration (parallel over time grid) ---
    L = len(t_grid)
    # initialize y_batch with constant initial value y0
    y_batch = torch.full((L,), float(y0.item()), device=device, dtype=dtype)
    start_time = time.time()
    for it in range(self.max_newton_iters):
        Fv, dFdy = self._evaluate_F_and_dFdy(y_batch, t_grid, A)
        G = Fv - y_batch
        dG = dFdy - 1.0
        denom = torch.where(dG.abs() < 1e-12, torch.sign(dG) * 1e-12 + 1e-12, dG)
        delta = G / denom
        y_new = y_batch - self.damping * delta
        max_change = torch.max(torch.abs(y_new - y_batch)).item()
        rms_resid = torch.sqrt(torch.mean(G**2)).item()
        y_batch = y_new
        if (it % 10) == 0 or it == 0:
            print(f"Iter {it:3d}: max_change={max_change:.3e}, rms_resid={rms_resid:.3e}")
        if max_change < self.newton_tol:
            print(f"Converged at iter {it}, max_change={max_change:.3e}")
            break
    print(f"Newton loop time: {time.time()-start_time:.3f}s")

    return y_batch, A

def compare_to_rk4(self, func, y0, t_grid, y_pred):
    """Compare ParODE solution to RK4 reference (same as in your original)."""
    def rk4_integrate(f, y0, t_grid):
        y = y0.clone()
        ys = [y.clone()]
        for i in range(len(t_grid) - 1):
            t = t_grid[i]
            dt = t_grid[i + 1] - t_grid[i]
            k1 = f(y, t)
            k2 = f(y + 0.5 * dt * k1, t + 0.5 * dt)
            k3 = f(y + 0.5 * dt * k2, t + 0.5 * dt)
            k4 = f(y + dt * k3, t + dt)
            y = y + dt * (k1 + 2*k2 + 2*k3 + k4) / 6.0
            ys.append(y.clone())
        return torch.stack(ys, dim=0)

    y_ref = rk4_integrate(func, torch.tensor(float(y0), device=self.device, dtype=self.dtype), t_grid)
    rel_err = torch.norm(y_ref - y_pred) / torch.norm(y_ref)
    dt = t_grid[1] - t_grid[0]

```

```

dydt_fd = torch.empty_like(y_pred)
dydt_fd[:-1] = (y_pred[1:] - y_pred[:-1]) / dt
dydt_fd[-1] = dydt_fd[-2]
residual = torch.norm(dydt_fd - func(y_pred, t_grid))
print("\nDiagnostics vs RK4:")
print(f"Relative L2 error: {rel_err.item():.3e}")
print(f"Residual norm ||d/dt F - f||: {residual.item():.3e}")
print("\n t      RK4_ref      F_fixedpt")
for i in range(0, min(10, len(t_grid))):
    print(f"{t_grid[i].item():6.3f} {y_ref[i].item():16.10e} {y_pred[i].item():16.10e}")
return rel_err.item(), residual.item()

# Example usage if run as script
if __name__ == "__main__":
    def f_rhs(y, t):
        # expect y and t are tensors (vectorized call)
        return -y**3 + torch.sin(t)

    solver = ParODE(M=10, N=10, K=200000)
    t_grid = torch.linspace(0.0, 2.0, 201, device=solver.device, dtype=solver.dtype)
    y0 = 0.5
    y_pred, A = solver.solve(f_rhs, y0, t_grid)
    solver.compare_to_rk4(f_rhs, y0, t_grid, y_pred)

    solver2 = ParODE2(M=10, N=10, K=200000, ic_weight=1e3, reg_lambda=1e-8)
    # t_grid = torch.linspace(0.0, 2.0, 201, device=solver.device, dtype=solver.dtype)
    # y0 = 0.5
    y_pred2, A2 = solver2.solve(f_rhs, y0, t_grid)
    solver2.compare_to_rk4(f_rhs, y0, t_grid, y_pred2)

```