Open source Differentiable ODE Solving Infrastructure

Rakshit Kr. Singh[1], **Aaron Rock Menezes**[1], **Rida Irfan**[1], **Bharath Ramsundar**[1]

[1]Deep Forest Sciences
rakshit@deepforestsci.com, aaron.r.menezes@gmail.com, rida@deepforestsci.com, bharath@deepforestsci.com

Abstract

Ordinary Differential Equations (ODEs) are widely used in physics, chemistry, and biology to model dynamic systems, including reaction kinetics, population dynamics, and biological processes. In this work, we integrate GPU-accelerated ODE solvers into the open-source DeepChem framework (Ramsundar et al. 2019), making these tools easily accessible. These solvers support multiple numerical methods and are fully differentiable, enabling easy integration into more complex differentiable programs. We demonstrate the capabilities of our implementation through experiments on Lotka-Volterra predator-prey dynamics, pharmacokinetic compartment models, neural ODEs (Chen et al. 2018), and solving PDEs using reaction-diffusion equations. Our solvers achieved high accuracy with mean squared errors ranging from 10^{-4} to 10^{-6} and showed scalability in solving large systems with up to 100 compartments.

Introduction

Solving ordinary differential equations (ODEs) is fundamental to the computational sciences. Researchers rely on ODEs to model complex biological systems such as disease dynamics, cellular interactions, and drug behaviors within the body. These models help predict how these systems will evolve and respond over time, which is essential for fields like pharmacokinetics and ecology. In pharmacokinetics, for example, ODEs help simulate how drugs are absorbed, distributed, metabolized, and excreted, allowing scientists to predict concentration levels and therapeutic effects more accurately.

Numerical methods offer a practical way to solve complex ODEs when analytical solutions aren't possible. Techniques like Euler's Method and Runge-Kutta (Runge 1895; Kutta 1901) approximate solutions by breaking down higher-order equations into simpler update rules. However, selecting the best numerical method for a specific problem can be challenging (Sumon and Nurulhoque 2023).

In addition, numerical methods for solving ODE face limitations with complex biological models that involve high-dimensional or stiff systems. These computational demands are difficult to meet without significant resources, limiting accessibility and slowing research in data-intensive fields. Furthermore, the lack of GPU acceleration in many ODE-solving frameworks hinders large-scale simulations and parameter estimation tasks, particularly in applications like pharmacokinetics that require quick, iterative computations. More robust open-source support for GPU-enabled ODE solvers could enable numerous downstream applications.

DeepChem (Ramsundar et al. 2019) is an open-source Python library designed for machine learning and deep learning, with a focus on applications in drug discovery and materials science. DeepChem's modular structure enables researchers to address challenging scientific problems in fields such as drug discovery, bioinformatics, and computational physics. In this work, we expand DeepChem's capabilities by integrating a GPU-accelerated ODE-solving infrastructure designed for science, with a particular focus on pharmacokinetic modeling. Our contributions include new optimization primitives for parameter estimation and simulations in DeepChem, that allow researchers to model drug transport and dynamics efficiently. Additionally, we have contributed a tutorial to guide users through these optimizations and open-sourced our infrastructure to promote accessibility and innovation within the field.

Background

Prior Research at solving ODE Systems

The study of ODEs began in the late 1600s when Newton and Leibniz developed calculus. In the 1700s, Euler and d'Alembert expanded on these methods, applying ODEs to new problems in physics and engineering, which solidified their role in scientific modeling. In the early 1900s, iterative numerical methods were introduced to address the need for higher accuracy in complex systems, like the Runge-Kutta methods (Runge 1895; Kutta 1901), which calculate solutions through multiple evaluations within each step. The fourth-order Runge-Kutta (RK4) method remains popular today for its balance of accuracy and computational efficiency. Additionally, methods like Broyden's approach for nonlinear systems have become essential for solving complex ODE systems (Broyden 1965).

Numerical Methods for solving ODE systems

Numerical methods for solving ordinary differential equations (ODEs) are essential for addressing a wide range of problems in science and engineering where analytical solutions are not feasible. These methods approximate solutions by discretizing the equations, allowing for practical computation. Below are key numerical techniques commonly used:

Euler's Method: A straightforward technique that approximates solutions by using the slope at the current point to estimate the next value (Burden and Faires 2011).

Runge-Kutta Methods: These are iterative techniques to approximate solutions to ODEs (Butcher 1996). They improve on simpler techniques such as Euler's method, which relied solely on the slope at the beginning of each interval by evaluating slopes at multiple points. More sophisticated methods, such as the fourth-order Runge-Kutta (RK4), evaluate multiple slopes at specifically chosen points to achieve higher accuracy. In our experiments, we used RK4 and RK38 methods (Butcher 2008).

Predictor-Corrector Methods: These combine a predictor step to estimate the next value and a corrector step to refine this estimate (Gear 1971).

Multi-step Methods: These use information from several previous points to compute the next value, enhancing efficiency (Hairer, Nørsett, and Wanner 1993).

Implicit Methods: Useful for stiff ODEs, these methods require solving equations at each step but offer improved stability (Ascher and Petzold 1998).

Understanding Differentiable Programming

Differentiable programming is a paradigm that structures programs in a way that allows them to be differentiated throughout their execution. This allows optimization algorithms, especially gradient-based methods, to be applied directly to the program's outputs. By making each part of the computation differentiable, it becomes possible to compute gradients for any output with respect to any input or internal parameter (Wang et al. 2018; Izzo, Biscani, and Mereta 2017).

Differentiable programming as a paradigm extends beyond traditional neural networks to allow the integration of complex mathematical operations and domain-specific knowledge directly into learnable models. This flexibility makes it a powerful tool not only in machine learning but also in scientific fields more broadly. Differentiable physics advocates for the use of differentiable programs to model physical systems (Ramsundar, Krishnamurthy, and Viswanathan 2021). A number of proposed approaches combine neural networks with differential equations to model various physical systems (Navarro, Moreno, and Rodrigo 2023). These methods need GPU-accelerated software to conduct experiments effectively, as they require a large amount of computation.

Ordinary Differential Equations (ODEs) in Systems Biology

ODEs are mathematical equations that describe the relationship between a function and its derivatives. They model dynamic systems where changes in a quantity depend on other variables over time, making them essential tools in systems biology for studying cellular metabolism, gene regulation, and disease progression. The general form of an ODE is as follows:

$$\frac{dy}{dx} = f(x, y) \tag{1}$$

A common approach in systems biology is to use compartment models, which divide complex systems into interconnected compartments. Each compartment represents a distinct region where substances can move, accumulate, or be processed. In pharmacokinetics, for example, these models track how drugs are absorbed, distributed, metabolized, and excreted across different organs or tissues. ODEs govern the transfer rates between compartments. This enables researchers to predict the dynamics and interactions of the system over time.

However, many biological models are "stiff," requiring careful selection of numerical integration methods and hyperparameters to ensure accurate and efficient solutions. Recent research emphasizes the importance of choosing robust ODE solvers that can handle the challenges posed by stiff biological systems effectively (Städter et al. 2021). In this paper, as a case study, we use GPU-accelerated ODE solvers in DeepChem to efficiently simulate pharmacokinetic compartment models and benchmark their performance against existing solvers.

Related Work

In recent years a lot of effort has been expended to build software tools for efficient and accurate solutions of ODE Systems. For instance, SciPy has implemented an extensive set of solvers that are fast and can be easily used (Virtanen et al. 2020). Their approach, however, is limited to CPUs, which can create bottlenecks when a large number of ODEs need to be solved. More recently, Julia-based libraries like DiffeqGPU have tried to address this issue (Utkarsh et al. 2024), but as these libraries are not native to the Python ecosystem they cannot be easily integrated with popular Python machine-learning frameworks and pipelines. Our implementation of ODE solvers is entirely written in Python and integrated with the DeepChem ecosystem. These design choices allow our implementations to run efficiently on GPUs and integrate easily into pre-existing DeepChem and Torch-based scientific workflows. Torchdiffeq (Chen 2018) provides similar capabilities but does not integrate into a scientific ecosystem like DeepChem's natively.

Recent work has investigated the use of black-box ODE solvers for modeling time-series data, supervised learning tasks, and density estimation. In particular, continuous-time normalizing flows, offer a flexible trade-off between computation speed and accuracy (Chen et al. 2018). Other related work presents a vectorized algorithm using deep neural networks to solve complex ODE-related problems such as stochastic or delay differential equations (Dufera 2021). Additionally, deep learning-based ODE solvers have greatly improved computational efficiency in chemical kinetics while maintaining accuracy for stiff ODE systems (Zhang et al. 2020).

Methodology

DeepChem and Differentiable Optimizers

Embedding physics knowledge into deep neural networks can reduce data requirements for deep learning (Ramsundar, Krishnamurthy, and Viswanathan 2021). By integrating differentiable scientific functions, like ODE solvers and optimization tools, models can simulate known physical behavior accurately without needing to learn from large amounts of data. However, existing machine learning libraries often lack differentiable equation solvers essential for scientific applications. Recent work has begun to close this gap with the development of PyTorch-based libraries providing critical scientific functions along with first and higher-order derivatives (Kasim and Vinko 2020). We have incorporated similar differentiation utilities into DeepChem to support complex scientific simulations. Our implementations build on those from (Kasim and Vinko 2020), but with considerable modification and adaption to the DeepChem ecosystem.

We have added a number of optimization algorithms to DeepChem, like Anderson acceleration, Adam, and Gradient Descent, and root-finding algorithms like Broyden's First Method and Broyden's Second Method.

Using DeepChem for System Identification

Integrating differentiable solvers directly into DeepChem allowed us to straightforwardly integrate standard machine learning techniques like gradient descent and Adam optimizers with ODE solvers. For example, in the experiments, we use a combination of gradient descent and ODE solver tools to conduct system identification experiments on Lotka-Volterra and compartment models.

Using DeepChem for Parameter Estimation

Parameter estimation refers to the techniques used to derive estimates of unknown system parameters based on observed data that may contain inherent randomness. In DeepChem, we combined minimization tools like gradient descent and Adam, which we used with our ODE Solvers to conduct parameter estimation experiments.

Training Neural ODEs using DeepChem

We leverage our new differentiable utility implementations to implement a Neural ODE model within DeepChem. Neural ODEs enable continuous-time modeling of time series data. Unlike other temporal architectures like recurrent neural networks, Neural ODEs model dynamics as differential equations and provide an efficient, flexible representation of temporal systems. This is especially valuable in scientific fields like biology, where continuous-time processes, such as gene expression and population dynamics, can play a crucial role (Chen et al., 2018). DeepChem's new differentiable ODE solvers enabled a fast and effective implementation of Neural ODEs and demonstrated the powerful composability of differentiable numerical tools.

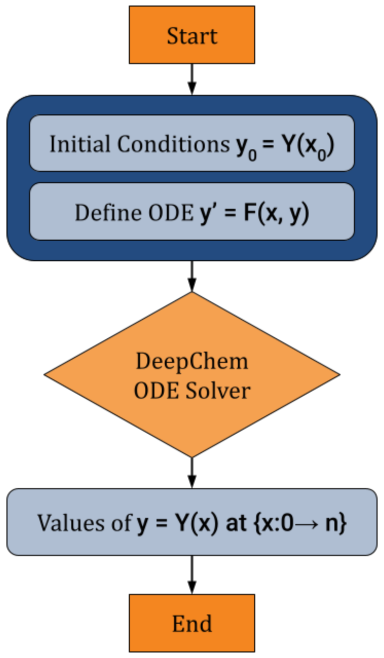
Start
Initial Conditions $y_0 = Y(x_0)$
Define ODE $y' = F(x, y)$
DeepChem ODE Solver
Values of $y = Y(x)$ at $\{x:0 \rightarrow n\}$
End

Figure 1: DeepChem ODE Solving workflow

Pharmacokinetics Simulation using DeepChem

Pharmacokinetics models use mathematical equations to describe how a drug is absorbed, distributed, metabolized, and excreted in the body over time. These models help understand the drug's concentration in different compartments of the body and ensure therapeutic levels are achieved without causing toxicity. In drug discovery, pharmacokinetics models help identify candidates with favorable ADME properties, reducing attrition rates in clinical trials. (Caldwell et al. 2003). DeepChem's ODE solvers and minimizers introduced in this work enable the resolution of multi-compartment pharmacokinetic models with remarkable flexibility. We demonstrate the solver's capabilities by successfully modeling complex scenarios involving up to 100 individual compartments.

Applications of Pharmacokinetics Model

Pharmacokinetics has diverse applications across various stages of drug development and clinical practice (Krishna 2004; Ruiz-García et al. 2008). One of the most significant applications in developing dosage regimens. By analyzing drug concentration in different body compartments over time, pharmacokinetic models provide critical insights that guide the selection of appropriate dosages. These models consider individual variability, including factors like age, weight, genetic makeup, and organ function, ensuring that the right dose is administered for maximum therapeutic effect while minimizing adverse reactions.

In toxicology, pharmacokinetic models are used to predict the thresholds at which substances may become harmful.

Algorithm 1: Parameter Estimation using Deepchem Minimizers and ODE Solvers

```
1: **Input:** Observed data $\mathbf{y}_{\text{obs}} = \{y_{\text{obs},i}\}$ at times $\{t_i\}$, initial parameter guess $\theta_0$, ODE model $\frac{d\mathbf{x}}{dt} = f(t, \mathbf{x}; \theta)$, tolerance $\epsilon$.
2: **Output:** Estimated parameter vector $\hat{\theta}$.
3: Initialize parameter $\theta \leftarrow \theta_0$
4: **repeat**
5:     Solve the ODE system $\frac{d\mathbf{x}}{dt} = f(t, \mathbf{x}; \theta)$ with current parameter $\theta$ to get predictions $\mathbf{x}_{\text{pred}}(t; \theta)$ at times $\{t_i\}$
6:     Compute the residual $\mathbf{r}(\theta) = \mathbf{y}_{\text{obs}} - \mathbf{x}_{\text{pred}}(t; \theta)$
7:     Define the cost function as $J(\theta) = \sum_i \|\mathbf{r}(\theta)\|^2$, where $\|\cdot\|$ denotes the Euclidean norm
8:     Update $\theta$ using an optimization method (e.g., Gradient Descent, Levenberg-Marquardt) to minimize $J(\theta)$
9: **until** $J(\theta) < \epsilon$ or convergence criteria are met
10: **Return** $\hat{\theta} = \theta$
```

These models are essential for establishing safety margins and designing drugs with favorable safety profiles. By simulating the complex interactions between chemicals and various tissues, pharmacokinetic studies can identify potential toxicities early in the development process. Furthermore, it helps refine experimental designs in toxicity testing by elucidating the relationships between external dosages and internal tissue exposures. (Leung 1991).

In drug development, they allow researchers to extrapolate animal data to predict human responses before clinical trials begin. This predictive capability saves time and resources by refining drug candidates early on and identifying those that are more likely to succeed in human trials. Moreover, pharmacokinetic models facilitate dose adjustments across different populations, such as children, the elderly, or individuals with compromised organ functions, enhancing personalized medicine approaches. Furthermore, leveraging pharmacokinetic principles early in discovery can significantly improve go/no-go decision-making and reduce the high costs of clinical development (Caldwell et al. 2003).

Experiments and Results

Experiments were conducted using Google Colab with 12 GB of RAM, an Nvidia T4 GPU, and an Intel Xeon CPU (2.2 GHz).

We tested our ODE solver infrastructure on four case studies:

- Lotka-Volterra (Predator-Prey Model) ODE Modeling and Parameter Optimization
- Pharmacokinetic Compartment Models
- Training Neural ODEs
- PDE Solving using ODE Solvers

Ordinary Differential Equation Solving

In this experiment, we analyze and solve two classical biological models described by systems of ordinary differential equations (ODEs): the Predator-Prey model and the Compartment model. These models are chosen because they exemplify different applications of ODEs in biology, with one focusing on population dynamics and the other on substance distribution across compartments. We solve these ODEs using our implementations of Runge-Kutta methods and analyze their accuracy and efficiency. Specifically, we are focusing on initial value problems (IVPs) of the form:

$$\frac{dy}{dt} = f(t, y), \quad y(t_0) = y_0 \tag{2}$$

where $f(t, y)$ represents the differential equation function, $y(t_0) = y_0$ is the initial condition, and t is the independent variable (often time).

ODE-solving experiments used the RK38 method from torchdiffeq and DeepChem, but since SciPy doesn't have an RK38 implementation, we use the default Runge-Kutta method of order 5(4) (RK5(4)). SciPy is partially written in C and C++ and uses adaptive stepping for their ODE solvers, while torchdiffeq and DeepChem use fixed steps and are purely Python-based, so the timing results are not directly comparable in our benchmarks but still serve as a useful comparison point.

1. Predator-Prey Model The Predator-Prey model, also known as the Lotka-Volterra model, describes the interaction between two species: a prey and a predator. The system of equations is given by:

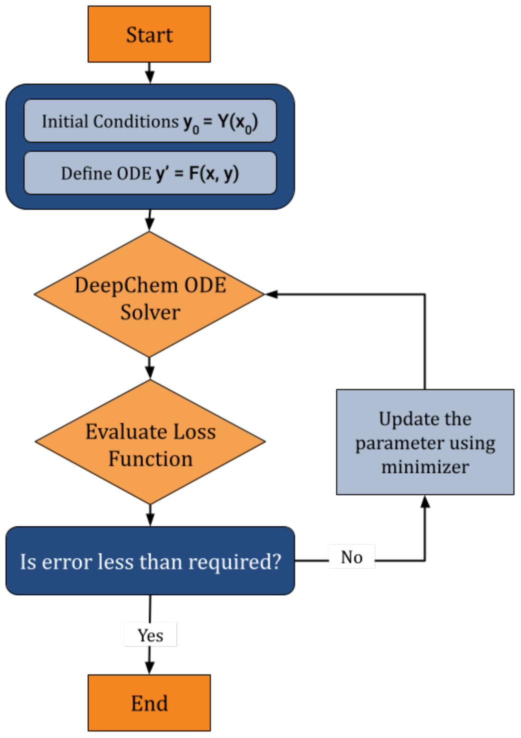
Start
Initial Conditions $\mathbf{y}_0 = \mathbf{Y}(\mathbf{x}_0)$
Define ODE $\mathbf{y' = F(x, y)}$
DeepChem ODE Solver
Evaluate Loss Function
Update the parameter using minimizer
Is error less than required?
No
Yes
End

Figure 2: DeepChem parameter estimation workflow

Table 1: Time taken (in sec) by different ODE Solvers for solving all the models. (The difference in time between SciPy and torchdiffeq/DeepCehm can be attributed to the adaptive stepping used by SciPy, which is highly optimized and uses Fortran, C, and C++ in addition to Python.)

Solvers	**torchdiffeq**	**SciPy**	**DeepChem**
Time Taken	54.4566	15.2681	52.2793

Table 2: L^1 distance between trajectories obtained Deepchem's ODE Solver and baseline trajectories obtained using SciPy's Solver. (Willmott and Matsuura 2005)

Solver	**Predator L^1**	**Prey L^1**
DeepChem vs. SciPy	0.0197	0.0275

$$\frac{dx}{dt} = \alpha x - \beta xy \tag{3}$$
$$\frac{dy}{dt} = \gamma xy - \delta y \tag{4}$$

where, x represents the prey population, y represents the predator population, α, β, γ, and δ are rate constants that dictate the interaction rates.

Experimental Setup: We solve 10 models, each for time $t : 0 \to 100$ and 10000 time steps using step size h of 0.01. These models were solved with initial values of x and y varying between $[10, 15]$ and $[5, 10]$ and the values of rate constants varying between $[0, 1]$. In Tables 1 and 2, we compare the performance of our implementation with torchdiffeq and SciPy.

2. Pharmacokinetic Compartment Model Pharmacokinetic compartment models divide the body into "compartments" that represent groups of tissues or organs with similar drug distribution characteristics. The system of equations for a three-compartment model is given by:

$$\frac{dC_1}{dt} = -(k_{10} + k_{12} + k_{13})C_1 + k_{21}C_2 + k_{31}C_3 \tag{5}$$
$$\frac{dC_2}{dt} = k_{12}C_1 - k_{21}C_2 \tag{6}$$
$$\frac{dC_3}{dt} = k_{13}C_1 - k_{31}C_3 \tag{7}$$

where, C_1 represents the central compartment while C_2 and C_3 are peripheral compartments, k_{10} is the elimination rate, k_{12} and k_{13} are rate constants for central to peripheral compartment motion, k_{21} and k_{31} are rate constants for peripheral to central compartment motions. Peripheral compartments are not connected directly.

Compartment models with more compartments can be constructed in a straightforward manner extending these equations.

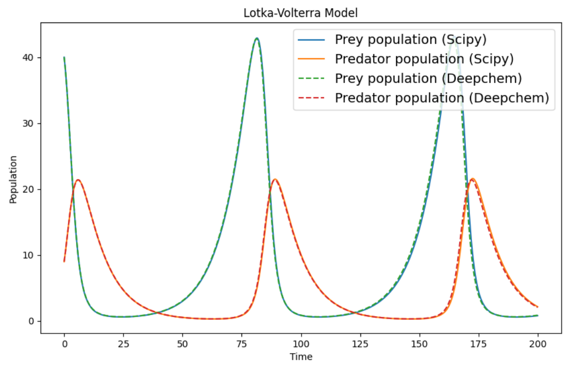

Figure 3: Predator-prey equation solutions obtained using the SciPy and DeepChem match closely.

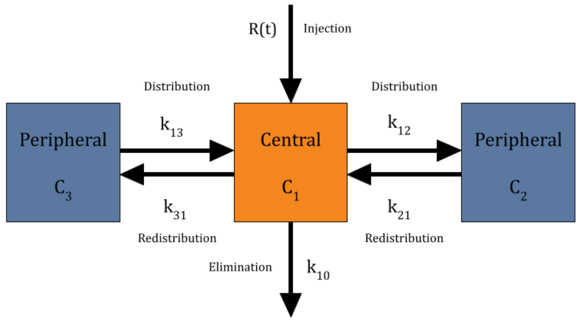

Figure 4: Schematic representation of a three-compartment model without a direct connection between peripheral compartments.

Experimental Setup: We solve 10 models, each for time $t : 0 \to 100$ and 10000 time steps with step size h of 0.01. These models were solved with initial values of C_1 at 10 and C_2, C_3 at 0 while the values of rate constants varied between $[0, 1]$. In Tables 3 and 4, we compare the performance of our implementation with torchdiffeq and SciPy, in which we simulated three different compartment models with 3, 10, and 100 compartments respectively.

3. Training Neural ODEs We trained a Neural ODE on synthetic data from a Damped Harmonic Oscillator model to capture the dynamics of oscillatory systems. The Damped Harmonic Oscillator is represented by the set of equations given below:

$$\frac{dx}{dt} = v \tag{8}$$
$$\frac{dv}{dt} = -kx - bv \tag{9}$$

where x is the position, v is the velocity, k is the spring constant and b is the damping coefficient.

This approach highlights the versatility of Neural ODEs for modeling time-varying phenomena governed by

Table 3: Comparison of time taken (in seconds) for solving models with varying numbers of compartments (Comp.) using different ODE solvers. The solvers compared include torchdiffeq, SciPy, and DeepChem.

	torchdiffeq	**SciPy**	**DeepChem**
3 Comp.	88.2022	20.2429	84.6690
10 Comp.	105.1800	23.6817	101.1345
100 Comp.	298.5121	40.4703	296.9111

Table 4: L^1 Error between simulated and predicted values of a Harmonic Oscillator by Neural ODE

Function	L^1 **Error**
Harmonic Oscillator	0.0156

continuous-time dynamics. Tables 4 and 5 provide experimental results. While the damped harmonic oscillator system is simple, our results serve as proof-of-concept validation of DeepChem's Neural ODE implementation. We will perform further validation in future studies.

Experimental Setup: We simulate a Damped Harmonic Oscillator for time $t : 0 \rightarrow 30$ and 100 time steps with parameters $b = 1$ and $v = 0.1$ and starting values of position and velocity being 0.99 and -0.99 respectively. We then train a 2 layer MLP with 32 units per layer. We use a tanh activation in the hidden layers for smoothness which helps mimic the dynamics of the continuous-time system. The forward pass is solved by the DeepChem ODE Solver using the Runge Kutta 3/8 method. We used the Adam optimizer to train a Neural ODE model on this simulated data with a learning rate of 0.01 and trained the model for 1000 epochs.

Parameter Estimation

Parameter estimation seeks to learn system parameters for dynamical systems governed by differential equations. These techniques have applications across diverse fields, from robotics and control engineering to biology, economics, and even environmental science.

For this experiment, we use DeepChem's ODE tools to estimate the parameters of the Lotka Volterra model from simulated data. While we use simulated data, the same techniques can be applied to time-series population counts or field observations. Table 5 shows comparisons between using DeepChem Solver along with the SciPy Minimizer and the DeepChem Solver along with the DeepChem Minimizer (Adam).

Solving PDEs Using ODE Solvers

This experiment investigates an innovative methodology for solving partial differential equations (PDEs) using the ODE solvers available within the DeepChem framework. Specifically, the focus is on reaction-diffusion systems (Figure

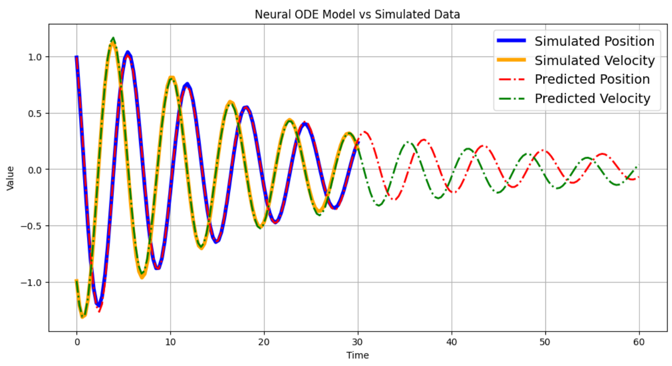

Figure 5: Harmonic Oscillator Dynamics using Neural ODEs: We compare a simulated Damped Harmonic Oscillator, solved using an ODE Solver, with the predictions of a Neural ODE and use the model to predict the dynamics of the system for the next 30 seconds.

Table 5: Lotka-Volterra Parameter Estimation. The ODE solver is DeepChem (dc). Either SciPy's parameter estimation or DeepChem's Adam implementations are used to minimize parameter error. The top row contains ground-truth values.

Solver	**Mini.**	α	β	γ	δ
-	-	1.1000	0.4000	0.1000	0.4000
dc	SciPy	0.8646	0.3447	0.1181	0.4903
dc	dc	1.0909	0.3909	0.0909	0.3909

6), which are instrumental in modeling the interactions of chemical species that undergo both reaction and spatial diffusion. Reaction-diffusion equations are used in various fields, including biology, chemistry, and physics to model phenomena such as pattern formation, population dynamics, and the spread of diseases.

The reaction-diffusion system for concentrations $U(x, y, t)$ and $V(x, y, t)$ is given by the following partial differential equations.

$$\frac{\partial U}{\partial t} = D_U \nabla^2 U - UV^2 + F(1 - U) \quad (10)$$

$$\frac{\partial V}{\partial t} = D_V \nabla^2 V + UV^2 - (F + k)V \quad (11)$$

- $U(x, y, t)$ and $V(x, y, t)$ are the concentrations of chemical species U and V, respectively.
- D_U and D_V are the diffusion coefficients for U and V.
- F is the feed rate for U.
- k is the kill rate for V.
- ∇^2 is the Laplacian operator, representing diffusion in two-dimensional space:

$$\nabla^2 Z = \frac{\partial^2 Z}{\partial x^2} + \frac{\partial^2 Z}{\partial y^2}$$

Experimental Setup We model a two-dimensional grid to represent a flat homogeneous spatial domain, wherein the concentrations of species U and V evolve over time. The system parameters are initialized as follows:

- $D_U = 0.16$: Diffusion coefficient for species U.
- $D_V = 0.08$: Diffusion coefficient for species V.
- $F = 0.04$: Feed rate for species U.
- $k = 0.06$: Kill rate for species V.

Initial concentration profiles are established to create a localized interaction zone. Specifically, within the central square region defined by $30 \leq x, y \leq 70$, the concentration of U is initialized at 0.5, indicating a higher concentration, while V is initialized at 0.25, indicating a lower concentration. Outside of this central region, both U and V are initialized at minimal baseline values, representing their negligible presence.

Using an ODE solver to integrate the discretized system over time, the spatio-temporal dynamics of U and V are observed. This evolution leads to the formation of complex spatial patterns, including spots, stripes, and wavefronts. These emergent patterns provide valuable insight into how relatively simple reaction-diffusion interactions can give rise to intricate and biologically significant structures.

We ran the experiments while varying the hyperparameters by a margin of 50% of the earlier initialized values (0.16, 0.08, 0.04, 0.06). This series of experiments gave us a more varying set of patterns.

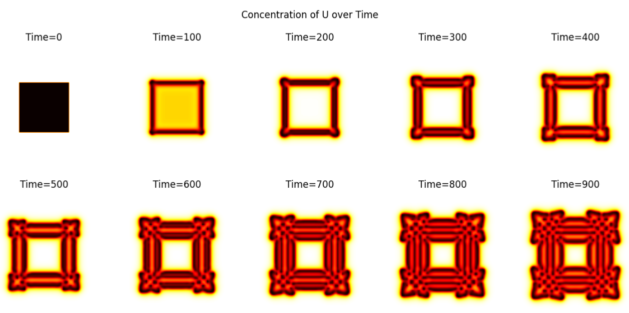

Figure 6: Reaction-Diffusion Dynamics: Simulating Pattern Formation with U and V Interactions. This visualization showcases the evolution of chemical concentrations in a reaction-diffusion system modelled using a grid of size 100x100 over 900 time steps.

Conclusion

This paper implements GPU-accelerated ODE solvers in DeepChem and studies their use in various scientific applications. We applied these solvers to solve predator-prey dynamics and pharmacokinetic compartment models. We also leveraged the solvers as building blocks to implement neural ODEs and PDE solvers in DeepChem. DeepChem solvers achieved high accuracy in parameter estimation and also demonstrated scalability by solving systems with up to 100 compartments. Additionally, Deepchem solvers are effective for modeling both simple and complex ODE systems.

In our experiments, we compared DeepChem solvers with SciPy and torchdiffeq and found DeepChem to be slightly faster than torchdiffeq but significantly slower than SciPy, likely due to the use of fixed timesteps and being written completely in Python. We aim to address this gap in future works. By open-sourcing this infrastructure, we aim to make these tools accessible and encourage innovation in systems biology and related fields.

References

Anderson, D. G. 1965. Iteration Processes for Nonlinear Equations. *Journal of the ACM (JACM)*, 12(4): 647–656.

Ascher, U. M.; and Petzold, L. R. 1998. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*. SIAM.

Broyden, C. G. 1965. A class of methods for solving nonlinear simultaneous equations. *Mathematical Computation*, 19: 577–593.

Burden, R. L.; and Faires, J. D. 2011. *Numerical Analysis*. Cengage Learning, 9th edition.

Butcher, J. 1996. A history of Runge-Kutta methods. *Applied Numerical Mathematics*, 20: 247–260.

Butcher, J. C. 2008. *Numerical Methods for Ordinary Differential Equations*. Wiley.

Caldwell, G.; Yan, Z.; Masucci, J.; Hageman, W.; Leo, G.; and Ritchie, D. 2003. Applied Pharmacokinetics in Drug Development. *Pharmaceutical Development and Regulation*, 1: 117–132.

Chen, R. T. Q. 2018. torchdiffeq.

Chen, R. T. Q.; Rubanova, Y.; Bettencourt, J.; Duvenaud, D.; of Toronto, U.; and Institute, V. 2018. Neural ordinary differential equations.

Dufera, T. T. 2021. Deep neural network for system of ordinary differential equations: Vectorized algorithm and simulation. *Machine Learning with Applications*, 5: 100058.

Gear, C. W. 1971. The Numerical Solution of Ordinary Differential Equations. *SIAM Review*, 13(1): 1–21.

Hairer, E.; Nørsett, S. P.; and Wanner, G. 1993. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer-Verlag, 2nd edition.

Izzo, D.; Biscani, F.; and Mereta, A. 2017. Differentiable Genetic Programming. In *Genetic Programming*, volume 10196 of *Lecture Notes in Computer Science*, 35–51. Springer. ISBN 978-3-319-55695-6.

Kasim, M. F.; and Vinko, S. M. 2020. ξ-torch: differentiable scientific computing library. *CoRR*, abs/2010.01921.

Kingma, D. P.; and Ba, J. L. 2015. ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION. *arXiv*.

Krishna, R. 2004. *Applications of Pharmacokinetic Principles in Drug Development*. Springer Nature Link. ISBN 978-1-4419-9216-1.

Kutta, M. W. 1901. Beitrag zur näherungsweisen Integration totaler Differentialgleichungen. *Zeitschrift für Mathematik und Physik*, 46: 435–453.

Leung, H. 1991. Development and utilization of physiologically based pharmacokinetic models for toxicological applications. *Journal of Toxicology and Environmental Health*, 32(3): 247–267. PMID: 2002511.

Navarro, L. M.; Moreno, L. M.; and Rodrigo, S. G. 2023. Solving differential equations with Deep Learning: a beginner's guide. arXiv:2307.11237.

Ramsundar, B.; Eastman, P.; Walters, P.; Pande, V.; Leswing, K.; and Wu, Z. 2019. *Deep Learning for the Life Sciences*. O'Reilly Media. https://www.amazon.com/Deep-Learning-Life-Sciences-Microscopy/dp/1492039837.

Ramsundar, B.; Krishnamurthy, D.; and Viswanathan, V. 2021. Differentiable physics: A position piece. *arXiv preprint arXiv:2109.07573*.

Ruiz-García, A.; Bermejo, M.; Moss, A.; and Casabo, V. G. 2008. Pharmacokinetics in drug discovery. *Journal of Pharmaceutical Sciences*, 97(2): 654–690.

Runge, C. 1895. Über die numerische Auflösung von Differentialgleichungen. *Mathematische Annalen*, 46: 167–178.

Städter, P.; Schälte, Y.; Schmiester, L.; Hasenauer, J.; and Stapor, P. L. 2021. Benchmarking of numerical integration methods for ODE models of biological systems. *Scientific Reports*, 11.

Sumon, M. M. I.; and Nurulhoque, M. 2023. A Comparative Study of Numerical Methods for Solving Initial Value Problems (IVP) of Ordinary Differential Equations (ODE). *American Journal of Applied Mathematics*, 11(6): 106–118.

Utkarsh, U.; Churavy, V.; Ma, Y.; Besard, T.; Srisuma, P.; Gymnich, T.; Gerlach, A. R.; Edelman, A.; Barbastathis, G.; Braatz, R. D.; et al. 2024. Automated translation and accelerated solving of differential equations on multiple GPU platforms. *Computer Methods in Applied Mechanics and Engineering*, 419: 116591.

Virtanen, P.; Gommers, R.; Oliphant, T. E.; Haberland, M.; Reddy, T.; Cournapeau, D.; Burovski, E.; Peterson, P.; Weckesser, W.; Bright, J.; van der Walt, S. J.; Brett, M.; Wilson, J.; Millman, K. J.; Mayorov, N.; Nelson, A. R. J.; Jones, E.; Kern, R.; Larson, E.; Carey, C. J.; Polat, İ.; Feng, Y.; Moore, E. W.; VanderPlas, J.; Laxalde, D.; Perktold, J.; Cimrman, R.; Henriksen, I.; Quintero, E. A.; Harris, C. R.; Archibald, A. M.; Ribeiro, A. H.; Pedregosa, F.; van Mulbregt, P.; and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17: 261–272.

Wang, F.; Decker, J.; Wu, X.; Essertel, G.; and Rompf, T. 2018. Backpropagation with Callbacks: Foundations for Efficient and Expressive Differentiable Programming. In *NIPS'18: Proceedings of the 32nd International Conference on Neural Information Processing Systems*, 10201–10212. Curran Associates.

Willmott, C. J.; and Matsuura, K. 2005. Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance. *Climate Research*, 30: 79–82.

Zhang, T.; Zhang, Y.; E, W.; and Ju, Y. 2020. A deep learning-based ODE solver for chemical kinetics.

Appendix

Anderson Acceleration Anderson acceleration is a method for improving the convergence rate of fixed-point iterations, particularly for nonlinear systems, by linearly combining previous iterations to achieve faster results without needing derivative information. (Anderson 1965)

Fixed-point iterations are fundamental in numerical analysis for solving equations of the form:

$$x = G(x) \tag{12}$$

where G is a given function. The iterative process starts with an initial guess x_0 and generates a sequence:

$$x_{k+1} = G(x_k) \tag{13}$$

Convergence depends on the properties of G and the choice of x_0. Specifically, if the spectral radius of the derivative $G'(x^*)$ at the fixed point x^* is less than 1, the iteration converges locally.

Adam Adam (Adaptive Moment Estimation) is an optimization algorithm that adjusts learning rates based on the first and second moments of gradients, improving convergence for deep learning models (Kingma and Ba 2015).

Gradient Descent Gradient Descent is used for training a vast array of machine learning models. Its performance heavily depends on factors like learning rate selection, feature scaling, and the specific variant of gradient descent employed.

Initialization

- Choose an initial set of parameters θ_0.
- Select a learning rate α, which determines the step size during each update.

Iterative Update

1. Compute the Gradient:

$$\nabla f(\theta_k) = \left[\frac{\partial f}{\partial \theta_1}, \frac{\partial f}{\partial \theta_2}, \dots, \frac{\partial f}{\partial \theta_n}\right]^T \tag{14}$$

2. Update the Parameters:

$$\theta_{k+1} = \theta_k - \alpha \nabla f(\theta_k) \tag{15}$$

3. Convergence Check:
   - If $\|\nabla f(\theta_{k+1})\|$ is below a predefined threshold, stop.
   - Otherwise, set $k = k + 1$ and repeat.

Broyden's First Method Broyden's First Method iteratively updates an approximation of the Jacobian matrix.

Mathematical Formulation

Given the current iterate $\mathbf{x}_k$, the Jacobian approximation $\mathbf{B}_k$, and the function evaluation $\mathbf{F}(\mathbf{x}_k)$, the method proceeds as follows:

1. Compute the Newton Step Solve the linear system to find the step $\mathbf{s}_k$.:

$$\mathbf{B}_k \mathbf{s}_k = -\mathbf{F}(\mathbf{x}_k) \tag{16}$$

2. Update the Solution

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k \tag{17}$$

3. Compute the Change in Function Values

$$\mathbf{y}_k = \mathbf{F}(\mathbf{x}_{k+1}) - \mathbf{F}(\mathbf{x}_k) \tag{18}$$

4. Update the Jacobian Approximation
   The Jacobian is updated using a rank-one update formula:

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{(\mathbf{y}_k - \mathbf{B}_k\mathbf{s}_k)\mathbf{s}_k^\top}{\mathbf{s}_k^\top \mathbf{s}_k} \tag{19}$$

   Here, $\mathbf{s}_k^\top$ denotes the transpose of $\mathbf{s}_k$.

Broyden's Second Method Broyden's Second Method iteratively updates an approximation of the inverse Jacobian matrix.

Algorithm Steps
Given:

- An initial guess $\mathbf{x}_0$.
- An initial inverse Jacobian approximation $\mathbf{B}_0$ (commonly the identity matrix).

The method proceeds as follows for each iteration $k = 0, 1, 2, \ldots$:

1. Compute the Newton Step:
   Solve the linear system:

$$\mathbf{B}_k\mathbf{F}(\mathbf{x}_k) = -\mathbf{s}_k \tag{20}$$

   to find the step $\mathbf{s}_k$.
2. Update the Solution:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{s}_k \tag{21}$$

3. Evaluate the Function at the New Point:

$$\mathbf{F}(\mathbf{x}_{k+1}) \tag{22}$$

4. Compute the Change in Function Values:

$$\mathbf{y}_k = \mathbf{F}(\mathbf{x}_{k+1}) - \mathbf{F}(\mathbf{x}_k) \tag{23}$$

5. Update the Inverse Jacobian Approximation:
   The inverse Jacobian is updated using a rank-one update formula:

$$\mathbf{B}_{k+1} = \mathbf{B}_k + \frac{(\mathbf{s}_k - \mathbf{B}_k\mathbf{y}_k)\mathbf{s}_k^\top \mathbf{B}_k}{\mathbf{s}_k^\top \mathbf{B}_k \mathbf{y}_k} \tag{24}$$

   Here, $\mathbf{s}_k^\top$ denotes the transpose of $\mathbf{s}_k$.
6. Convergence Check:
   If $\|\mathbf{F}(\mathbf{x}_{k+1})\|$ is below a predefined tolerance level, terminate the algorithm. Otherwise, set $k = k+1$ and repeat the iteration.

Pharmacokinetic Compartment Models Pharmacokinetic compartment models are mathematical models used to describe the way drugs are absorbed, distributed, metabolized, and eliminated by the body. These models simplify the complex processes of drug movement and interaction within the body by dividing it into compartments that represent different physiological spaces. The compartments are not necessarily anatomical structures but conceptual spaces where the drug concentration is assumed to be uniform.

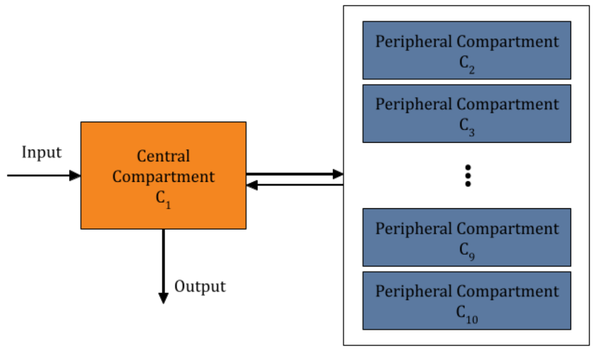
Input
Central Compartment C_1
Output
Peripheral Compartment C_2
Peripheral Compartment C_3
Peripheral Compartment C_9
Peripheral Compartment C_{10}

Figure 7: Schematic representation of a model with 10 compartments

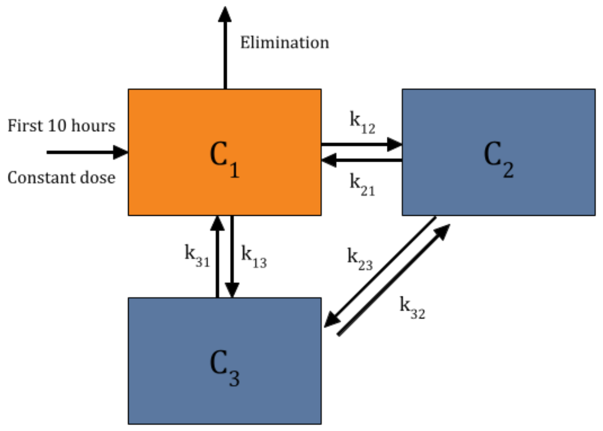

Figure 8: Compartment model showing interaction between the peripherals