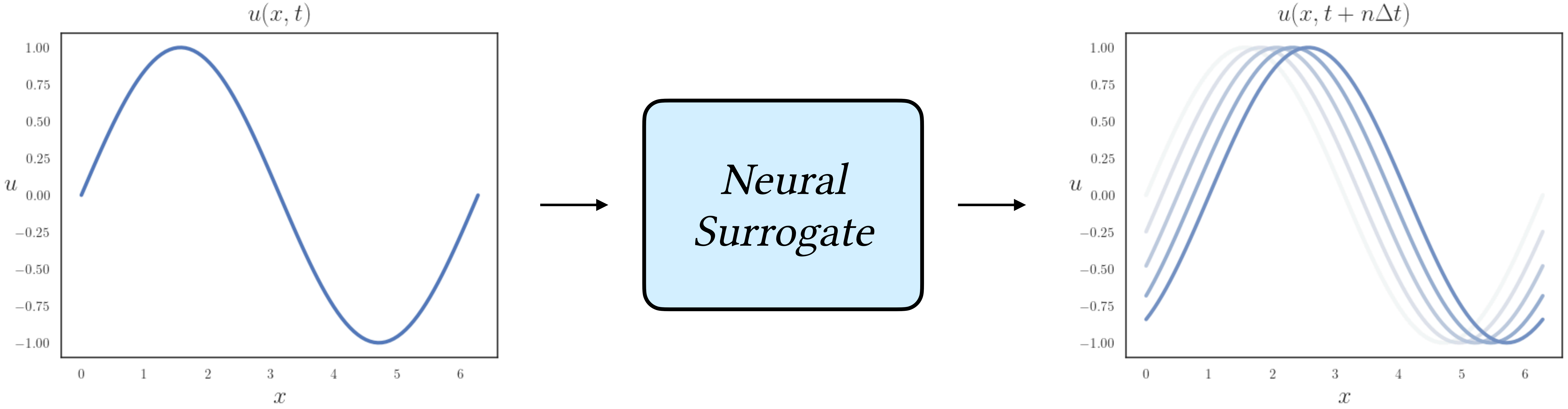
Predicting Change, Not States: An Alternate Framework for Neural PDE Surrogates

Anthony Zhou, Amir Barati Farimani
Carnegie Mellon University, 2/14/2025

Carnegie Mellon University

What is the common framework for applying neural surrogates?



Input: Solution at time T, ***u(t)***

Output: Solution at time T+1, ***u*(t+1)***

- Trained to minimize next-step loss: ***u*(t+1) - u(t+1)***. Simple to train/implement.
- Generally treats solution updates as a black box by directly outputting solution field.

Is this the most effective framework?

How do numerical solvers update solution values?

Governing Equation: $\boldsymbol{u_t = F(u_x, u_{xx}, \ldots)}$ | Approximate Spatial Derivatives | Apply Time-Stepping Scheme

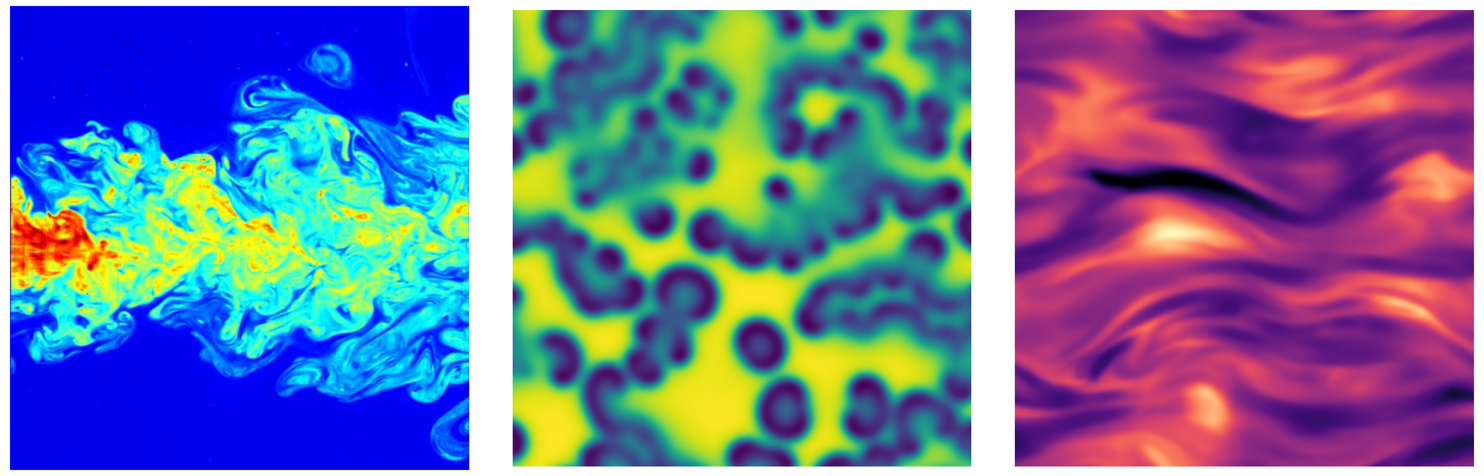
$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0. \longrightarrow -c\frac{u^j_{i+1} - u^j_{i-1}}{2\triangle x} \longrightarrow y_{n+1} = y_n + hf(t_n, y_n)$$

Example: 1D Advection Equation | Example: Central Differences | Example: Forward Euler

- Both spatial/temporal schemes highly influences solution speed/accuracy.
- Can be very complex. Many different schemes, each with tradeoffs and designed for different purposes. Still a highly active area of research.

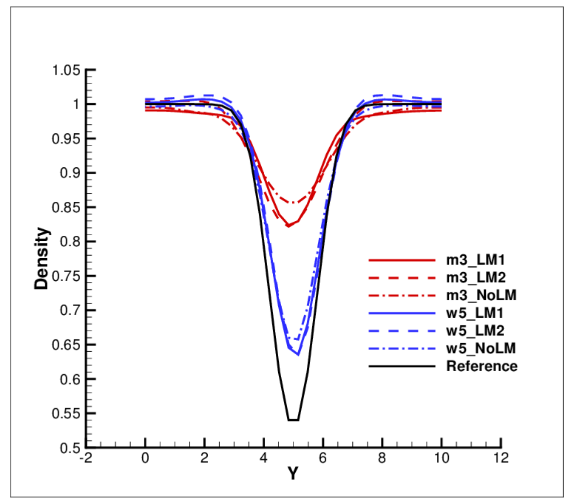
Are neural surrogates an over-simplification?

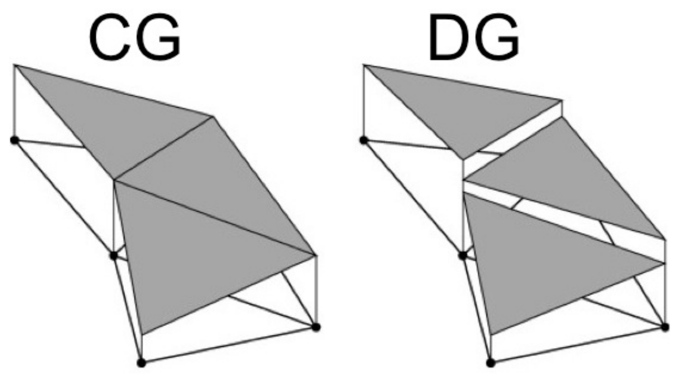
Governing Equation: $u_t = F(u_x, u_{xx}, \ldots)$



Turbulent Flow
Reaction-Diffusion
Plasma Physics
… etc.

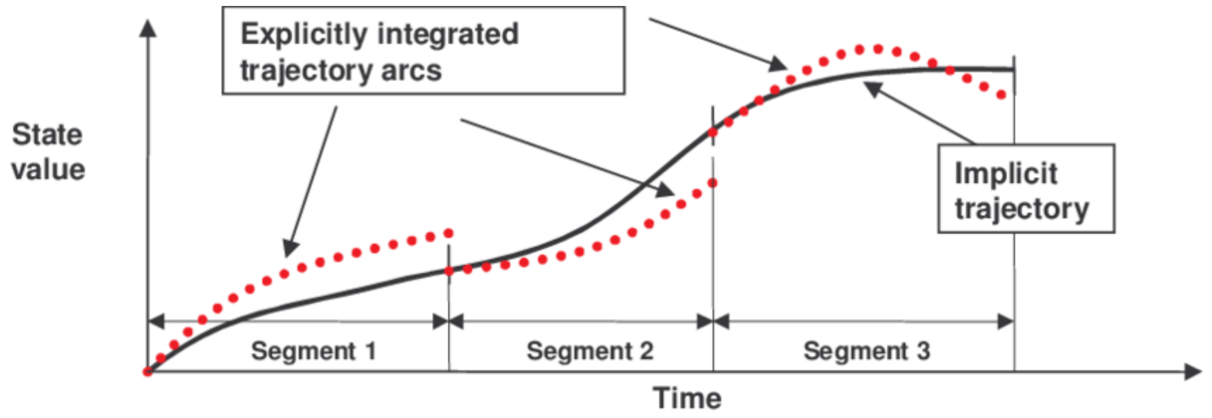
Approximate Spatial Derivatives





FVM/FEM/FDM, etc.
Shock-capturing Schemes
Discontinuous Galerkin
…etc.

Apply Time-Stepping Scheme



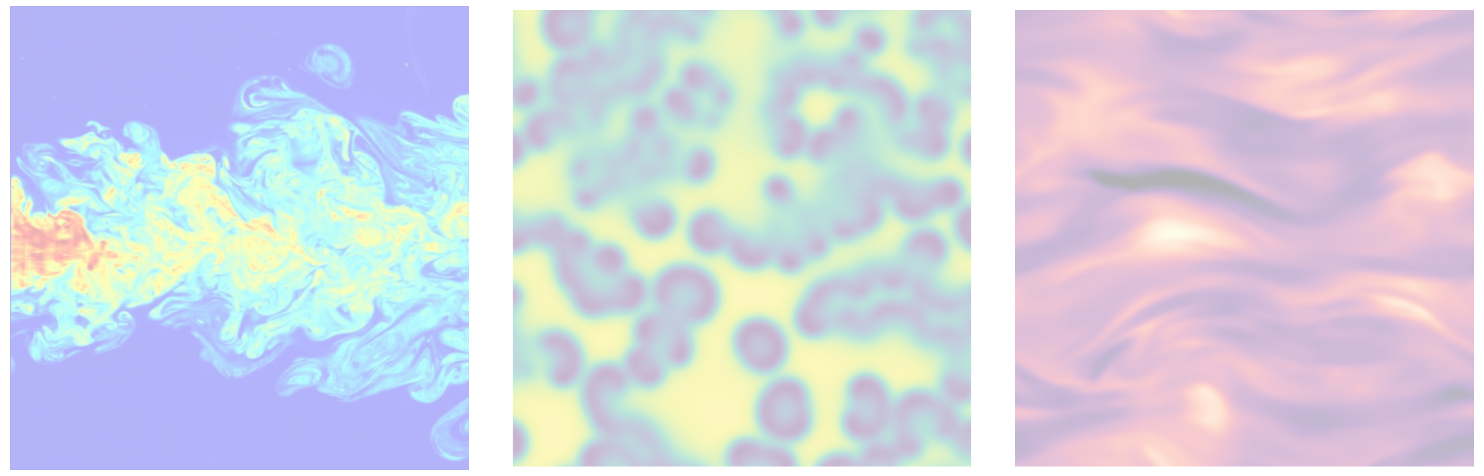
Implicit/Explicit
Higher-order Integrators
Operator Splitting
…etc.

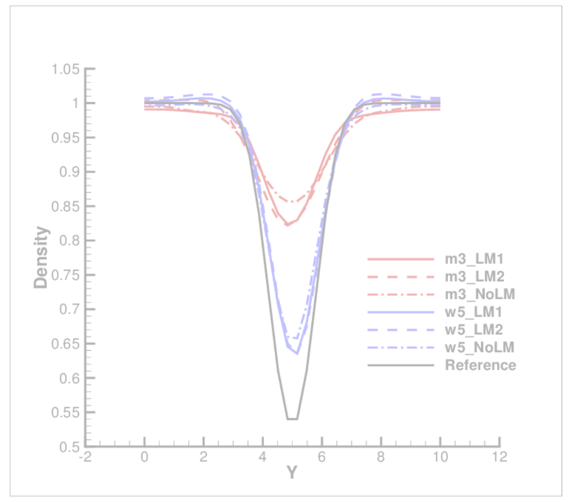
Usually bundled together using the same framework. Neural surrogates generally rely on architecture/training to handle complexity, rather than specialized numerical methods.

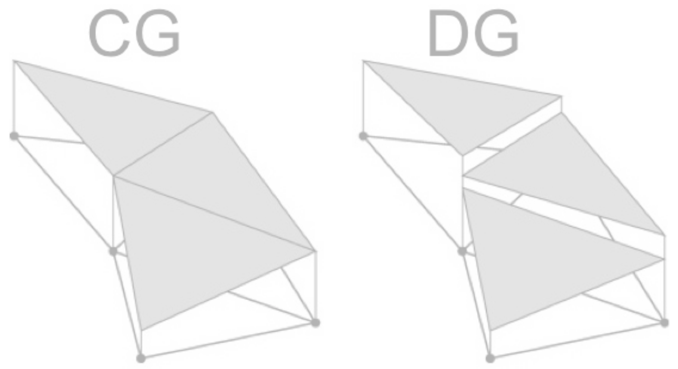
Are neural surrogates an over-simplification?

Re-introduce time integrators into neural surrogates

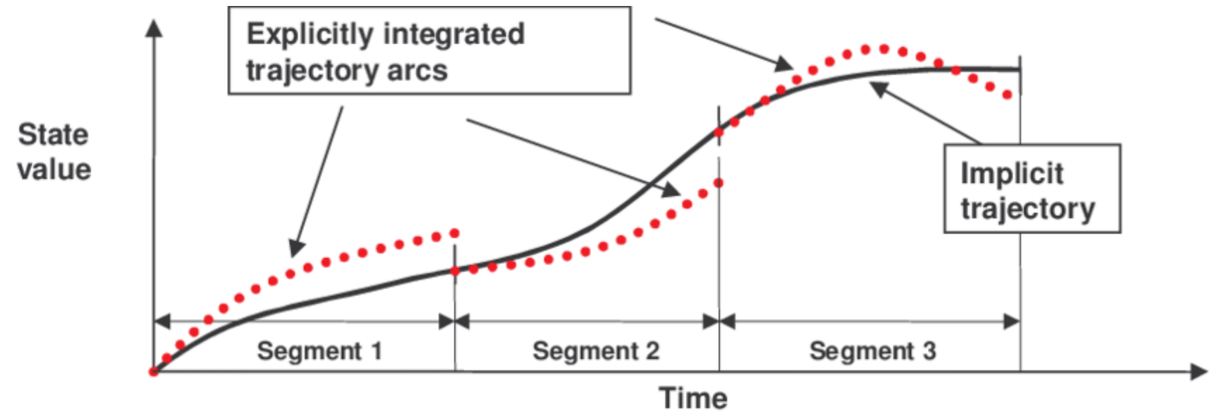
Governing Equation: $u_t = F(u_x, u_{xx}, \dots)$







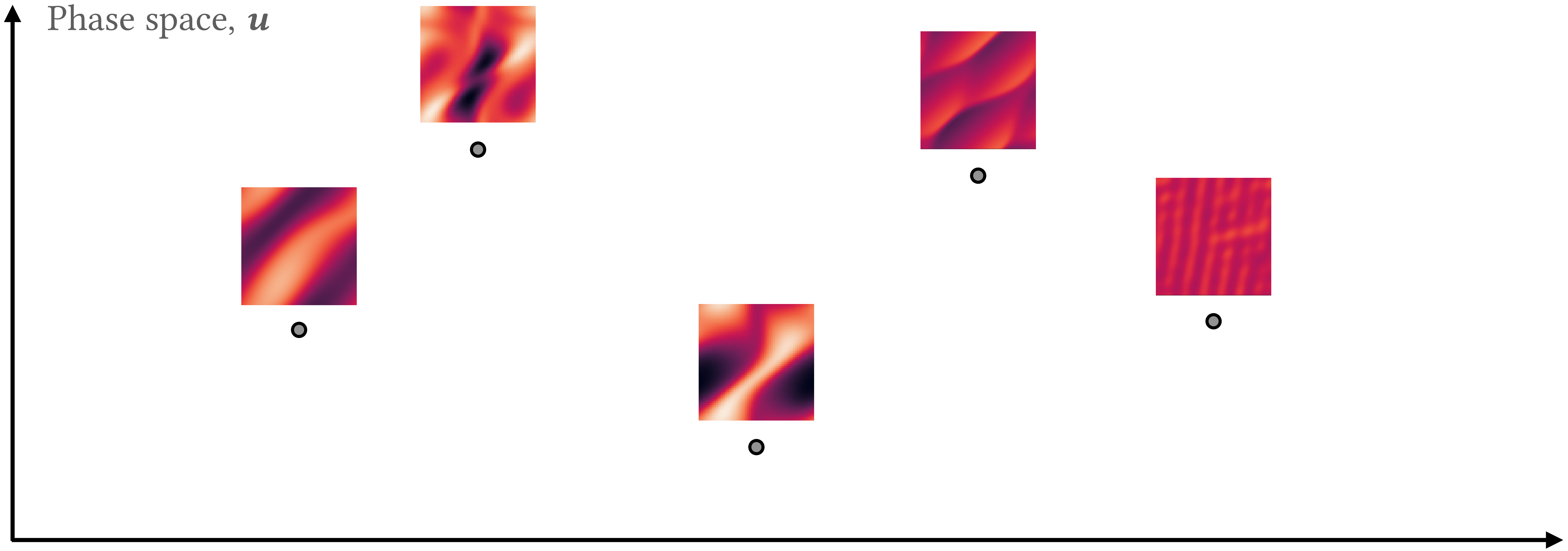
Apply Time-Stepping Scheme



Implicit/Explicit
Higher-order Integrators
Operator Splitting
...etc.

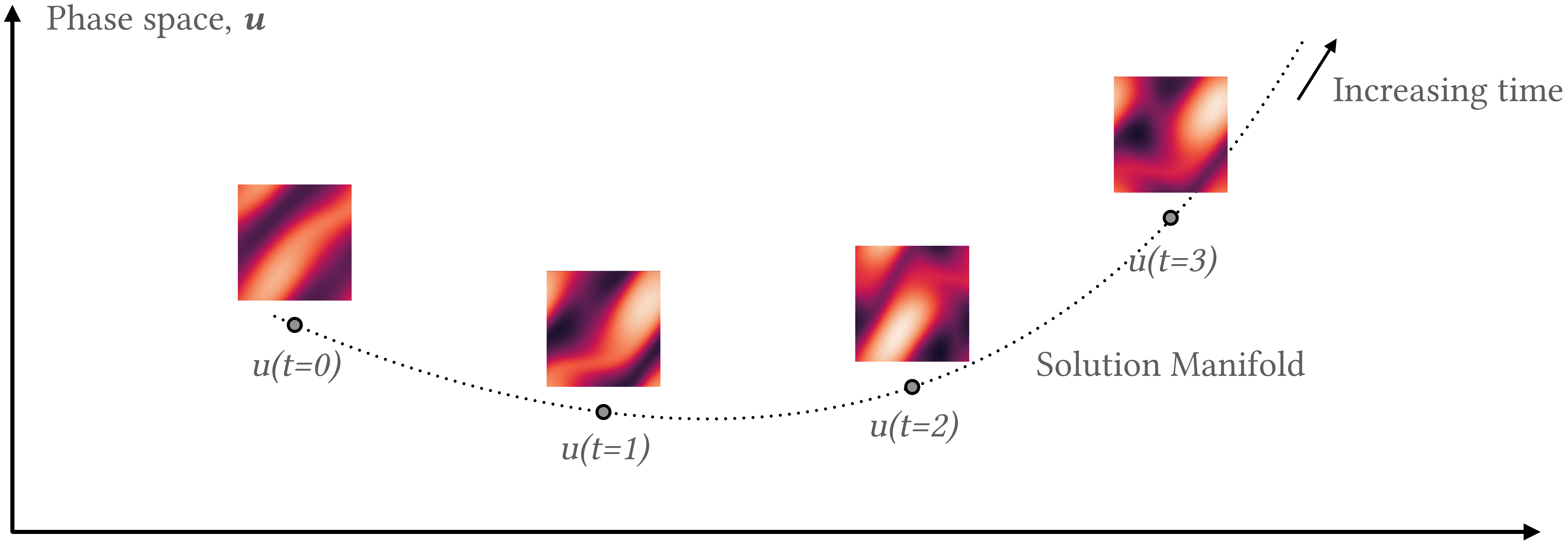
How do we introduce time-integration into neural surrogates?

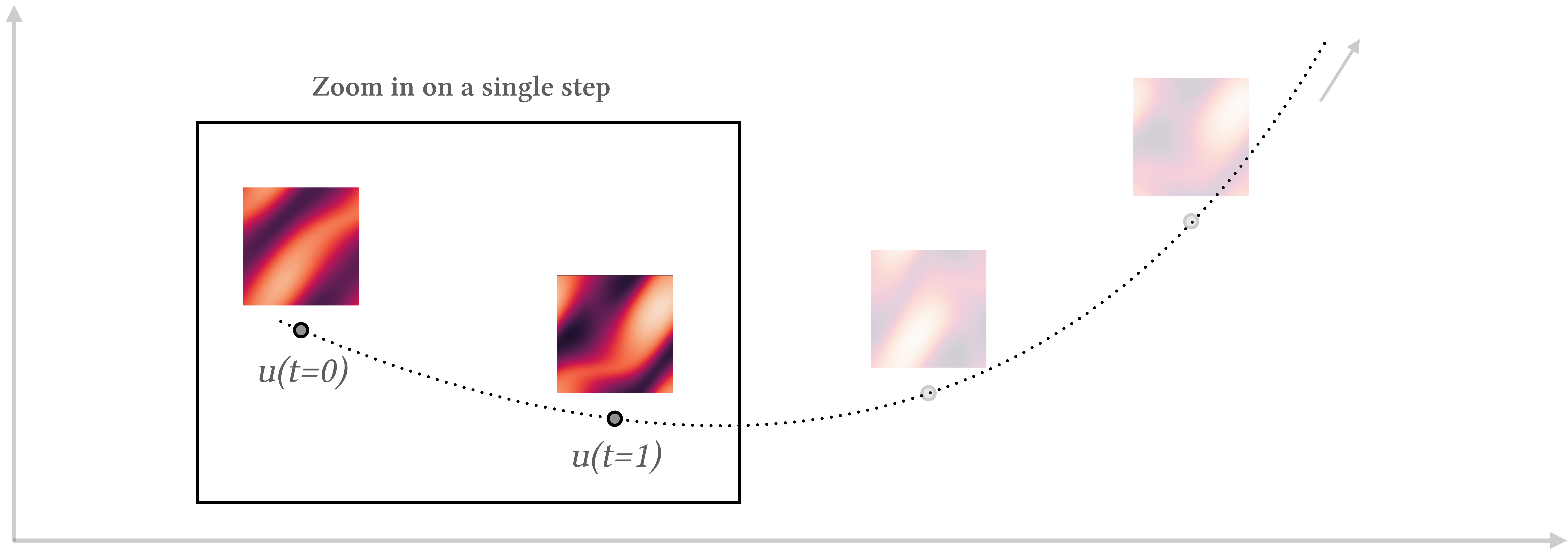
Only 2 changes: 1) Predict ***u'(t)*** rather than ***u(t+1)*** during training. 2) Use time-stepping scheme during inference.



How do we introduce time-integration into neural surrogates?

Only 2 changes: 1) Predict ***u'(t)*** rather than ***u(t+1)*** during training. 2) Use time-stepping scheme during inference.

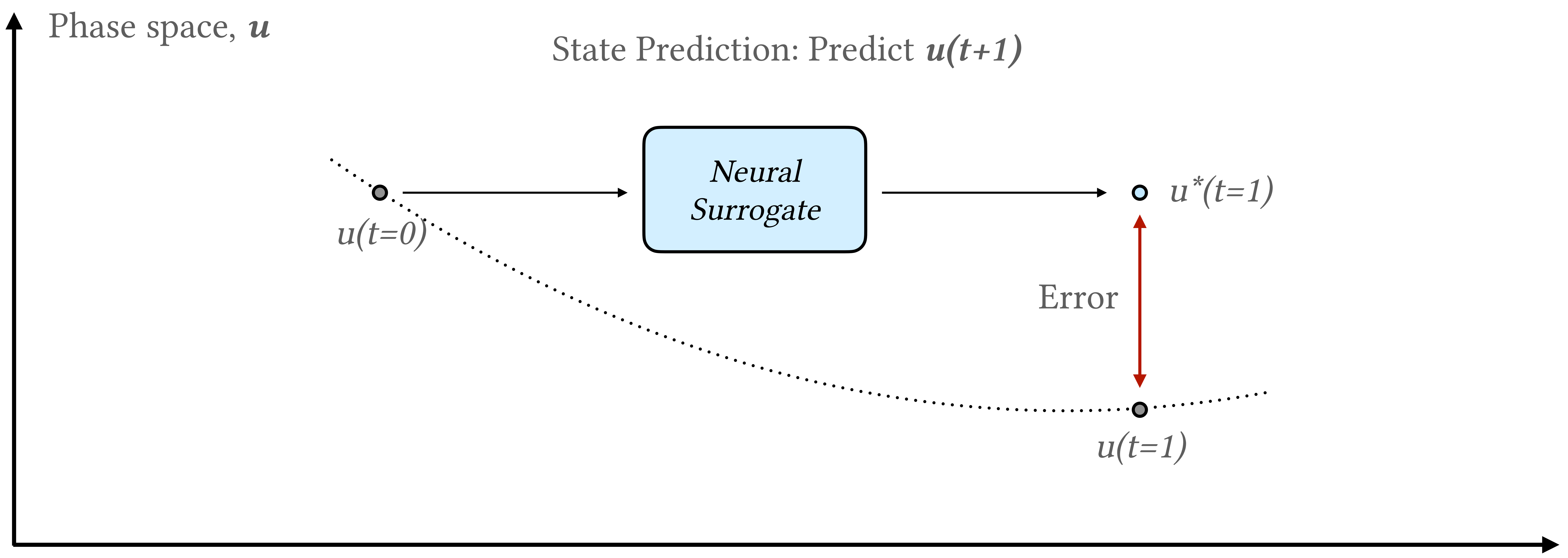




Zoom in on a single step
u(t=0)
u(t=1)

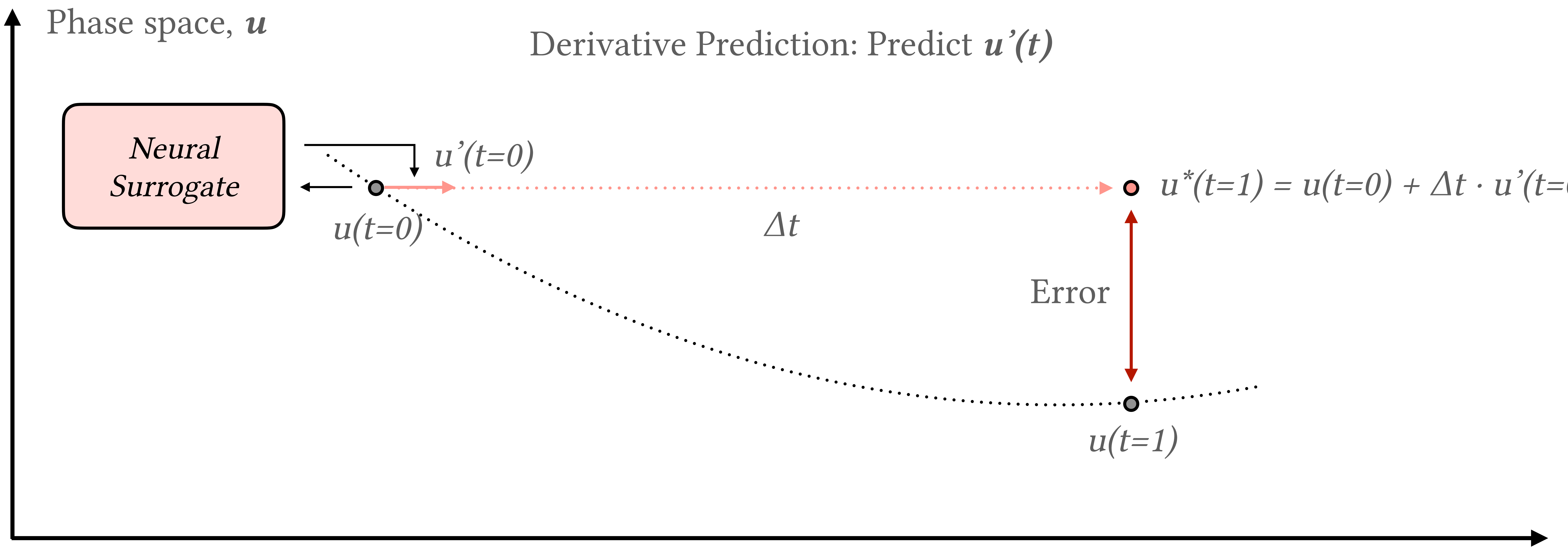
How do we introduce time-integration into neural surrogates?

Only 2 changes: 1) Predict ***u'(t)*** rather than ***u(t+1)*** during training. 2) Use time-stepping scheme during inference.



How do we introduce time-integration into neural surrogates?

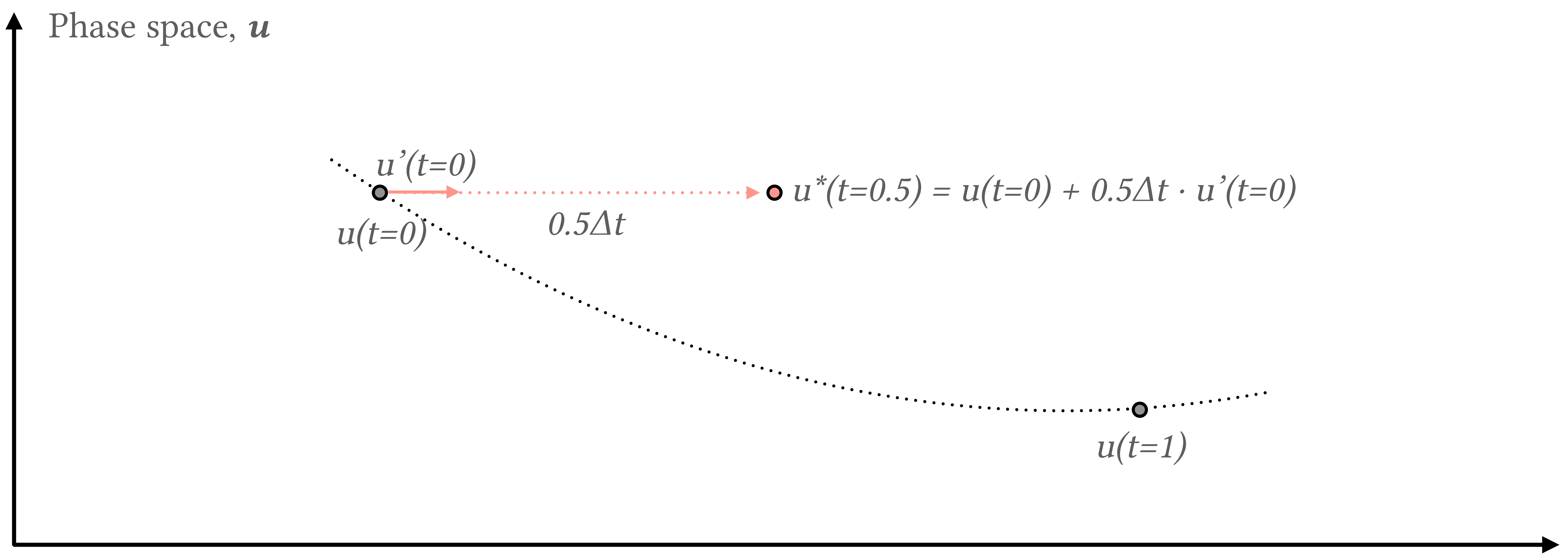
Only 2 changes: 1) Predict ***u'(t)*** rather than ***u(t+1)*** during training. 2) Use time-stepping scheme during inference.



How can this be beneficial?

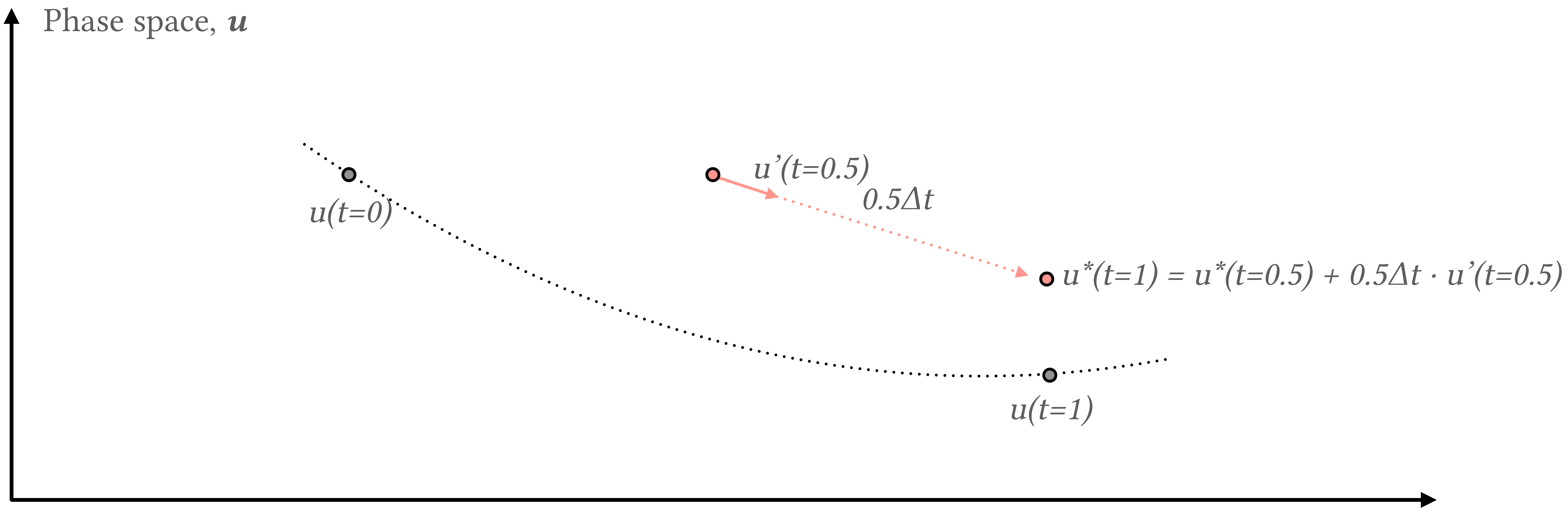
1) Flexible step sizing during inference

2) Better accuracy with higher-order integrators



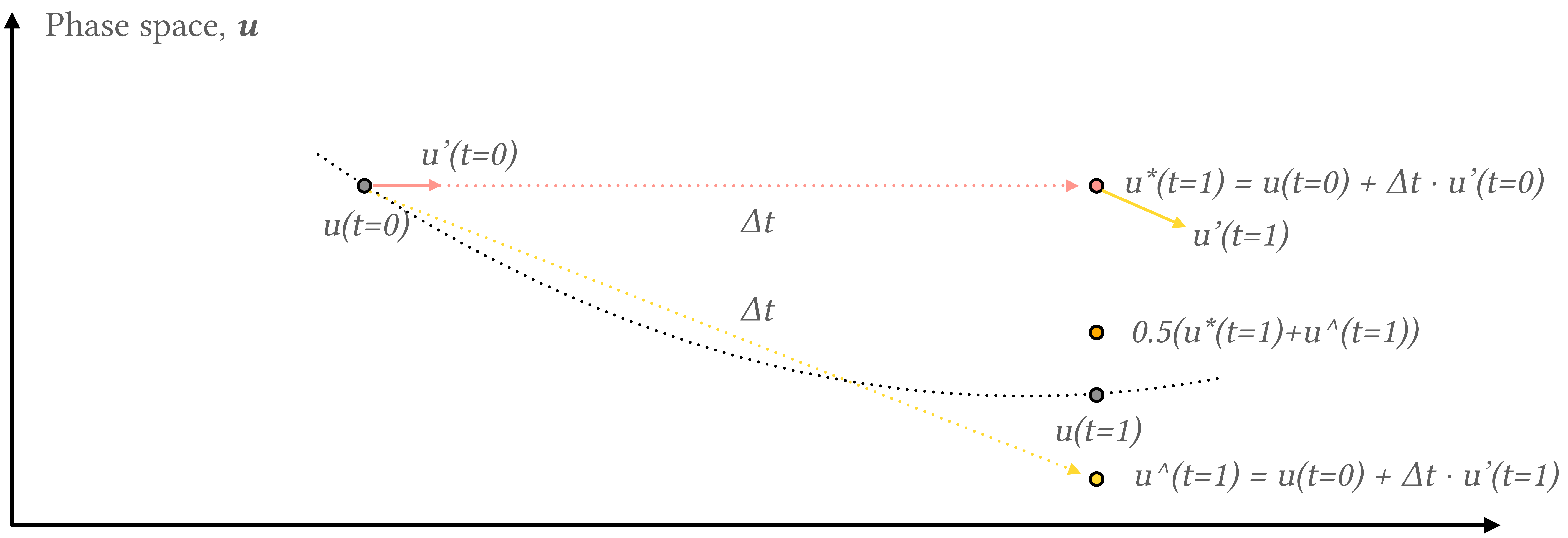
How can this be beneficial?

1) Flexible step sizing during inference



How can this be beneficial?

2) Better accuracy with higher-order integrators



A More Formal Definition

Training

$$\mathcal{L}_\theta(\mathbf{u}(t_n), t_n, \mathbf{y}) = ||F_\theta(\mathbf{u}(t_n), t_n) - \mathbf{y}||_2^2,$$

$$\mathbf{y} = \begin{cases} \mathbf{u}(t_{n+1}) & \text{state-prediction} \\ \frac{\partial \mathbf{u}}{\partial t}\big|_{t=t_n} & \text{derivative-prediction} \end{cases}$$

Learn current derivative. Calculated with FD schemes from data

Small change to loss formulation - No architecture/data changes

Inference

$$\tilde{\mathbf{u}}(t_{n+1}) = \mathbf{u}(t_n) + \Delta t F_\theta(\mathbf{u}(t_n), t_n)$$

$$\mathbf{u}(t_{n+1}) = \mathbf{u}(t_n) + \frac{\Delta t}{2}(F_\theta(\mathbf{u}(t_n), t_n) + F_\theta(\tilde{\mathbf{u}}(t_{n+1}), t_{n+1})$$

Heun's Method

Solve next step by integrating predicted derivative

Can be changed without retraining model

Remark: This is not a novel approach

- Residual Prediction: $F(\mathbf{u}(t)) = \mathbf{u}(t+1) - \mathbf{u}(t)$
  - Often used in climate applications/GNN-based surrogates
  - Can be seen as a scaled Forward Euler approximation
- Derivative Prediction
  - Two works use an RK2 integrator [1, 2]
- Hybrid Solvers
  - Surrogates often predict spatial derivatives or portions of PDE update
  - Ex. Convective flux approximation in Navier-Stokes Equation
- Hamiltonian/Lagrangian NNs, Neural ODEs
  - Need an ODE integrator since HNNs/LNNs/Neural ODEs only predict derivatives

1. *Sanchez-Gonzalez, A., Bapst, V., Cranmer, K., Battaglia, P.: Hamiltonian Graph Networks with ODE Integrators (2019). https://arxiv.org/abs/1909.1279,*
2. *Zeng, B., Wang, Q., Yan, M., Liu, Y., Chengze, R., Zhang, Y., Liu, H., Wang, Z., Sun, H.: PhyMPGN: Physics-encoded Message Passing Graph Network for spatiotemporal PDE systems (2024). https://arxiv.org/abs/2410.01337*

Experimental Setup

- Models considered (FNO/Unet)
- PDEs considered
  - 1D: Advection, Heat, KS
  - 2D: Burgers, NS, Kolmogorov Flow
- Training: use a 4-th order FD scheme to approximate spatial derivatives from dataset
- Inference
  - Forward Euler: 1st-order, simple/fast
  - Adams-Bashforth: 2nd-order, linear multistep method
  - Heun: 2nd-order, predictor-corrector method
  - RK4: 4-th order, Runge-Kutta method

Results: Prediction Accuracy

PDE: Metric:	Adv Roll. Err. ↓	Heat Roll. Err. ↓	KS Corr. Time ↑	Burgers Roll. Err. ↓	NS Roll. Err. ↓	Kolm. Flow Corr. Time ↑
FNO (State Pred.)	0.498	0.589	139.75	0.437	0.715	51.4
FNO (FwdEuler)	0.048	0.141	79.25	0.196	0.159	29.0
FNO (Adams)	0.032	0.141	183	0.174	0.100	45.5
FNO (Heun)	0.032	0.141	197.75		0.100	81.6
FNO (RK4)	0.033	0.141	198		0.100	82.3
Unet (State Pred.)	0.044	0.149	153.5	0.666	0.099	35.1
Unet (FwdEuler)	0.032	0.139	76.5	0.280	0.093	23.3
Unet (Adams)	0.011	0.139	167.75	0.264	0.048	27.9
Unet (Heun)	0.010		173.5	0.263	0.049	53.7
Unet (RK4)	0.010		174.5	0.264	0.049	88.9

Higher-order schemes benefit chaotic systems

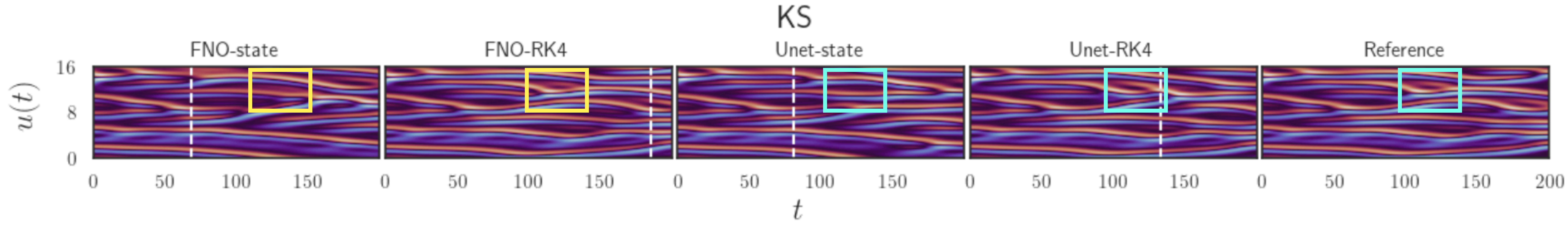
No increase in accuracy w/ increasing order

Higher-order schemes become more important

Table 1: **Prediction Accuracy.** Results on prediction accuracy across different PDEs, models, and training/inference frameworks.

- Higher order integrators are only needed for more complex equations (KS, Kolm. Flow)
  - More compute is needed during inference to evaluate higher-order schemes
- Overall, derivative prediction can help stabilize rollouts and improve performance

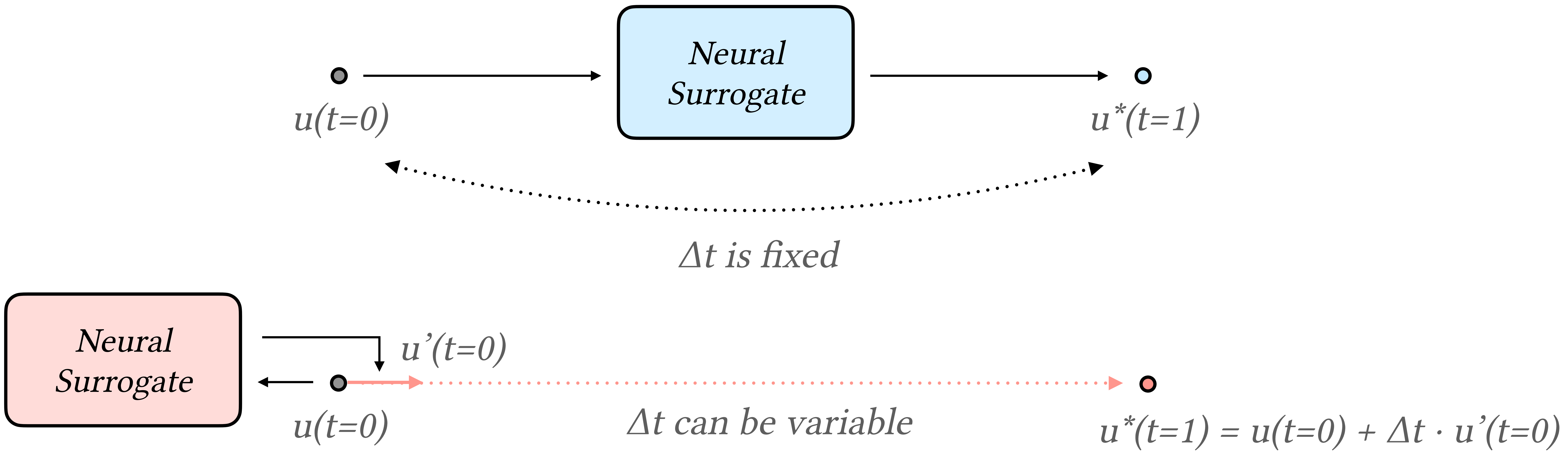
Results: Prediction Accuracy



White lines denote correlation time, after which solution diverges

Results: Inference Modifications

Derivative prediction can offer additional flexibility, since it does not fix the resolution of the surrogate.



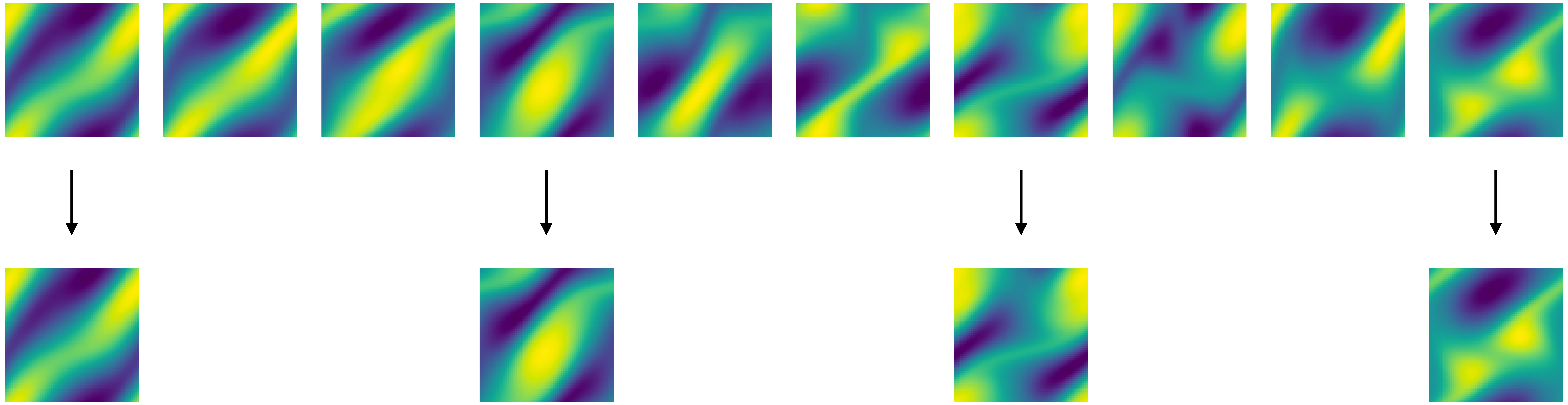
How to use this flexibility?

Can train on more finely discretized data

Can adaptively change step size during inference

Results: Inference Modifications

1) Solve Trajectory w/ Numerical Solver. Δt is usually very small.



2) Downsample by 10-1000x to form dataset. Discards 90-99% of data.

Train on high-res data to obtain accurate derivative estimates. Inference on large Δt for fast time-stepping

Results: Inference Modifications

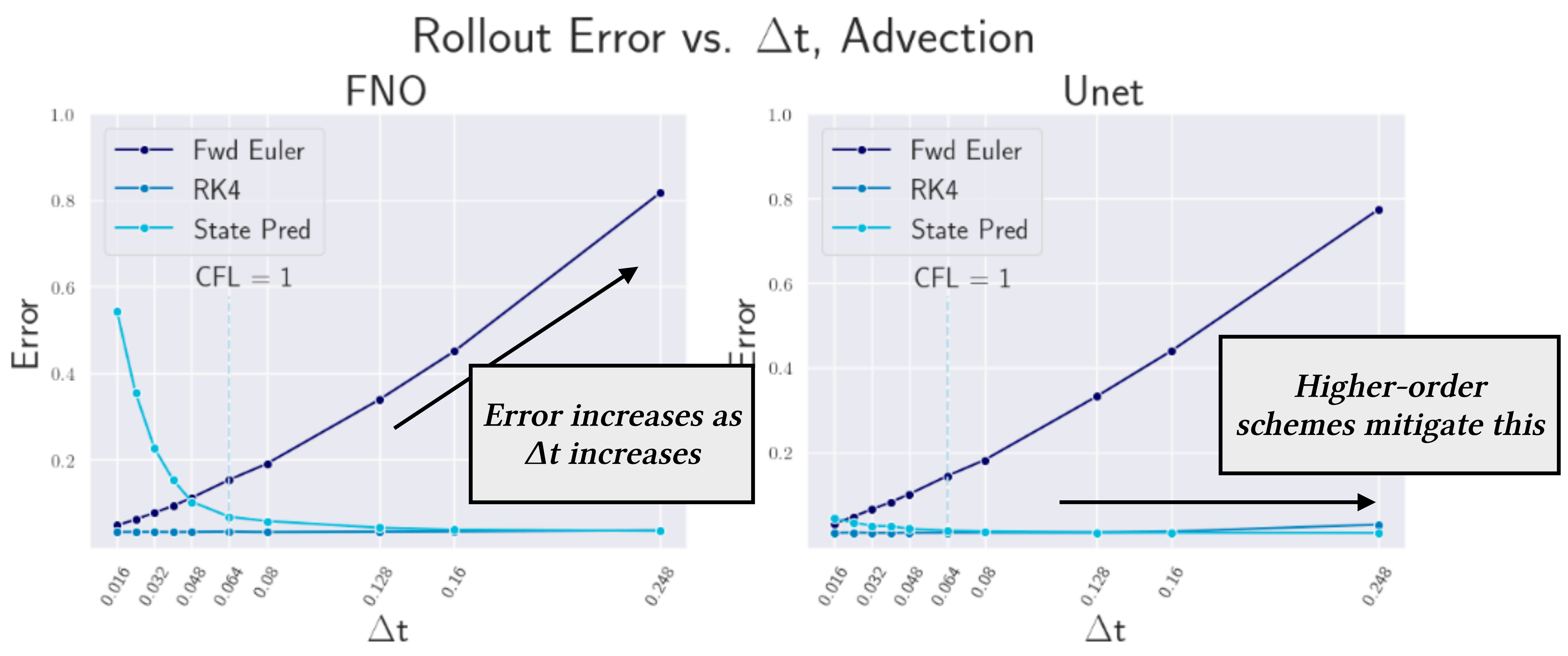
PDE: Metric:	Adv Roll. Err. ↓	NS Roll. Err. ↓	PDE: Metric:	Adv Roll. Err. ↓	NS Roll. Err. ↓
FNO (FwdEuler)	0.048	0.159	Unet (FwdEuler)	0.032	0.093
FNO (FwdEuler + 2x data)	0.045	0.139	Unet (FwdEuler + 2x data)	0.033	0.091
FNO (FwdEuler + 2x steps)	0.037	0.120	Unet (FwdEuler + 2x steps)	0.019	0.059
FNO (Heun)	0.033	0.100	Unet (Heun)	0.010	0.049

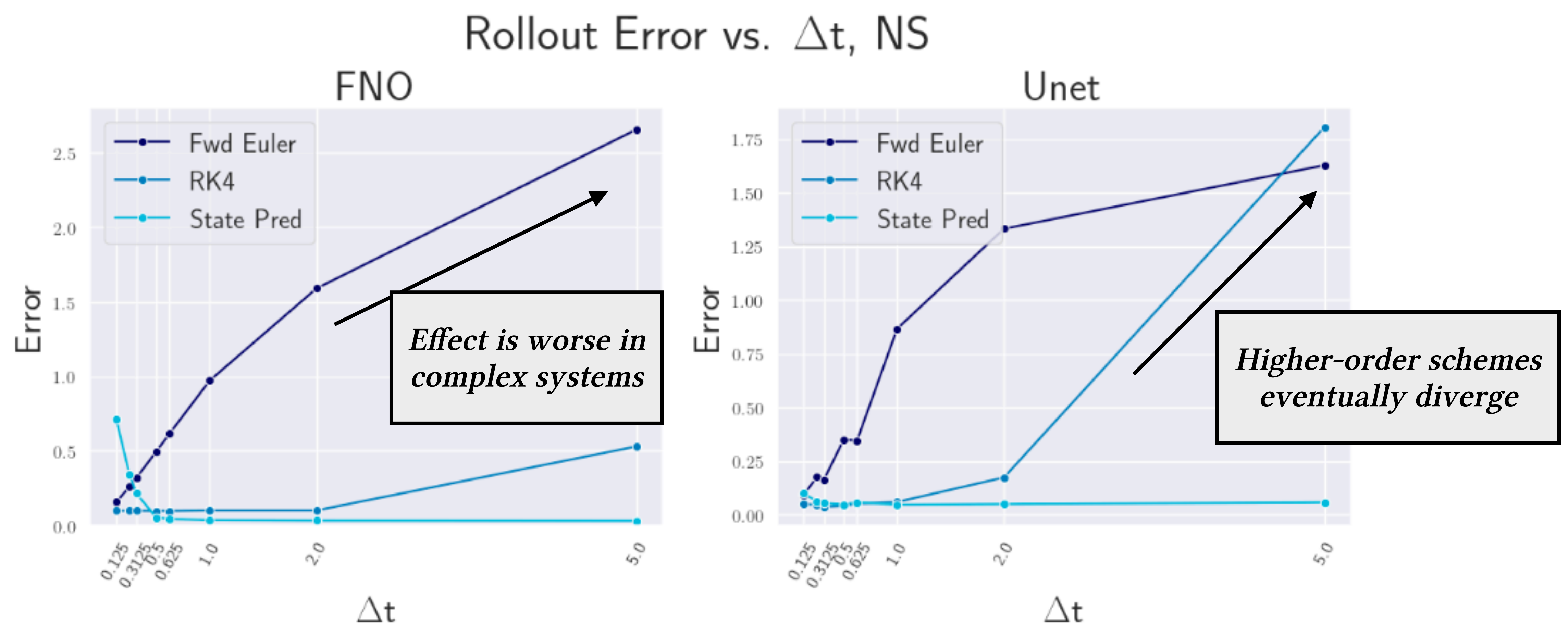
Table 3: **Inference Modifiers.** Comparing different inference modifications across various models and PDEs.

- Extra data is more beneficial for complex systems.
  - Additional data is very similar to existing data, but is free.
- Extra steps during inference can improve accuracy. Still better to use higher-order schemes.
  - Opportunity to use adaptive step sizing with high-order schemes. (Adaptive RK4 is SoA*)

**used in MATLAB's ode45(): https://www.eng.auburn.edu/~tplacek/courses/3600/ode45berkley.pdf*

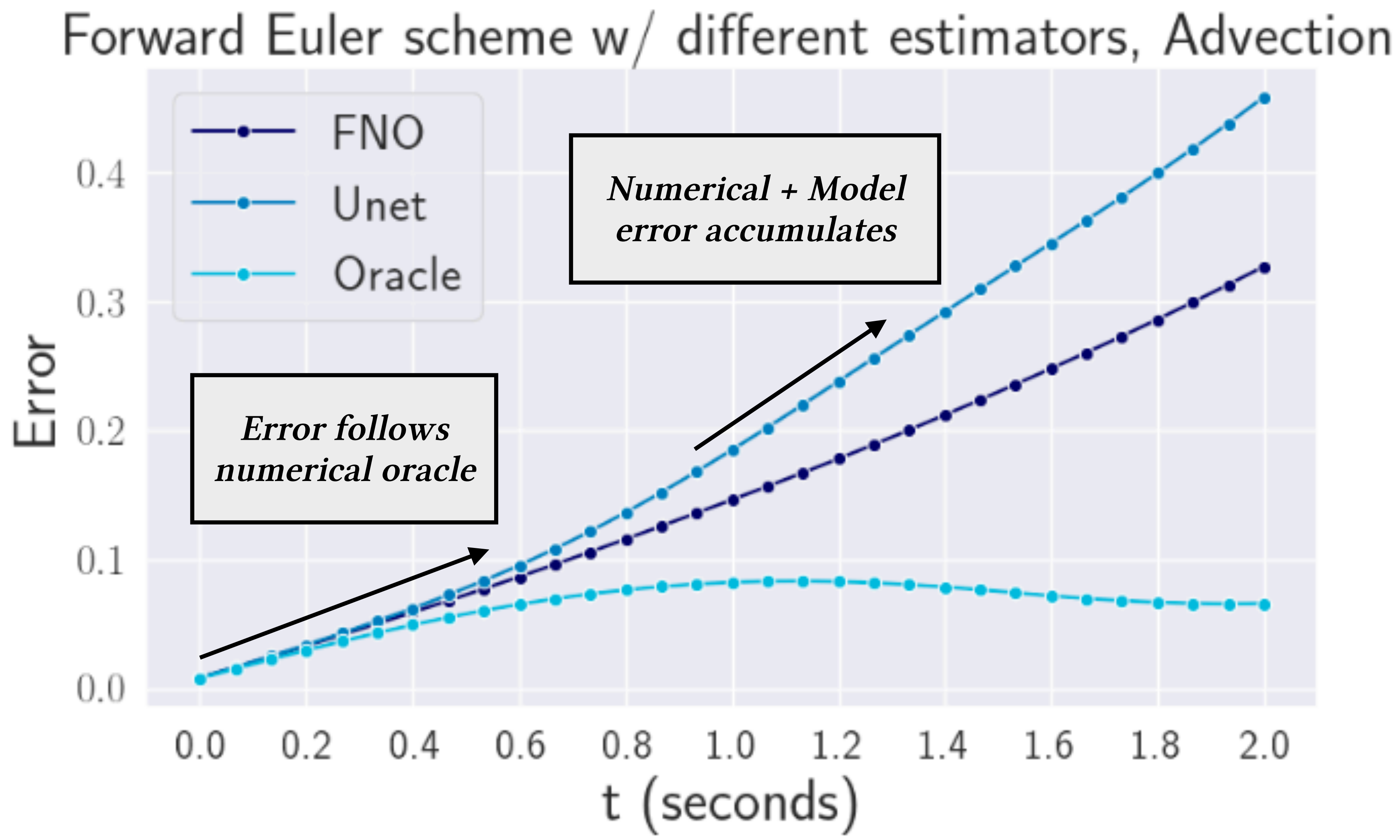
Limitations: Performance depends on Δt

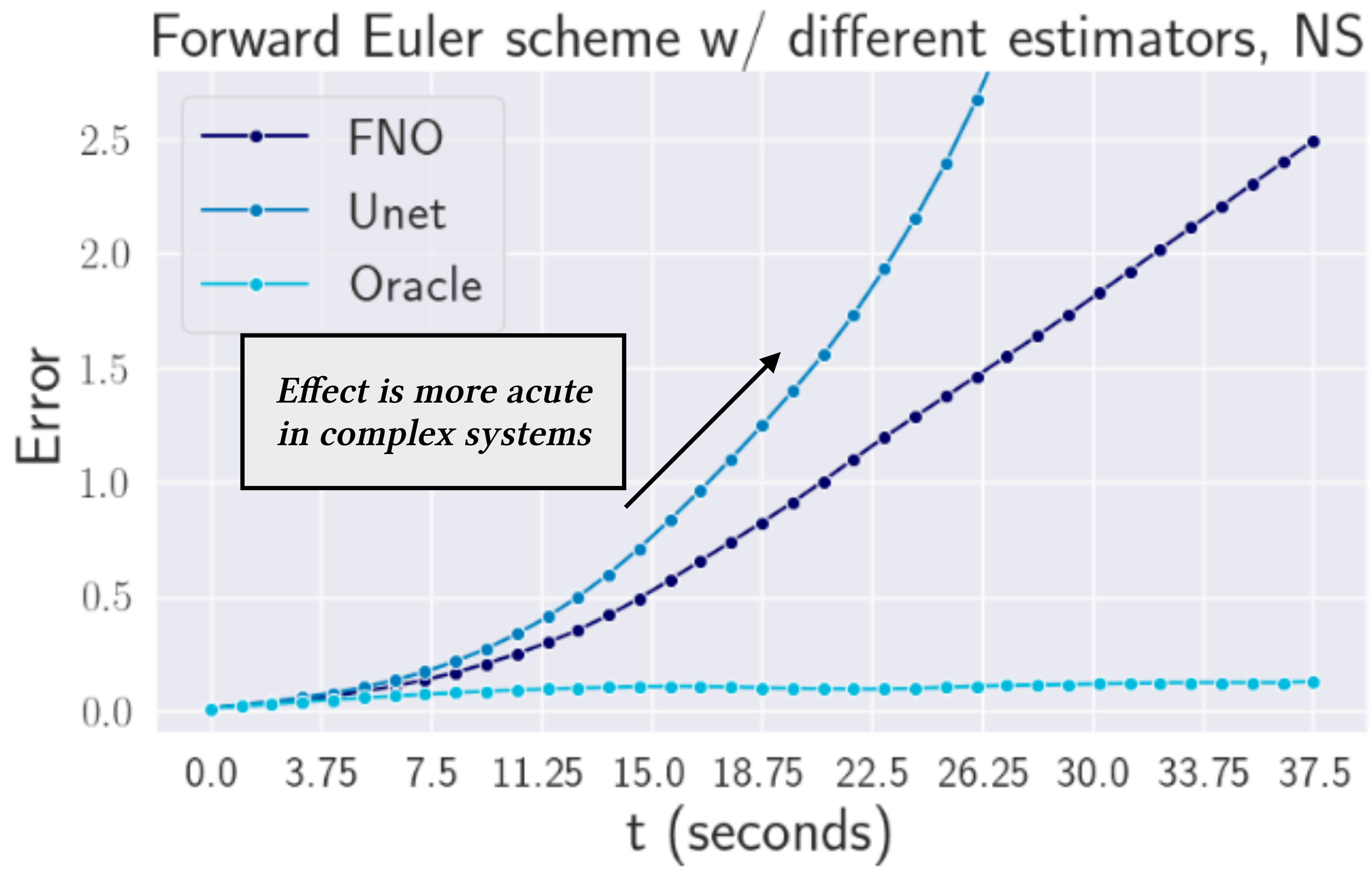




- Rollout error is dependent on step size Δt.
- Complex systems require higher-order integrators and smaller step size.
  - Re-introduces discretization constraints to neural surrogates
- Step size can still be much larger than numerically stable Δt. (i.e., CFL > 1)
- Steady-state problems (Darcy flow, statics) are incompatible.

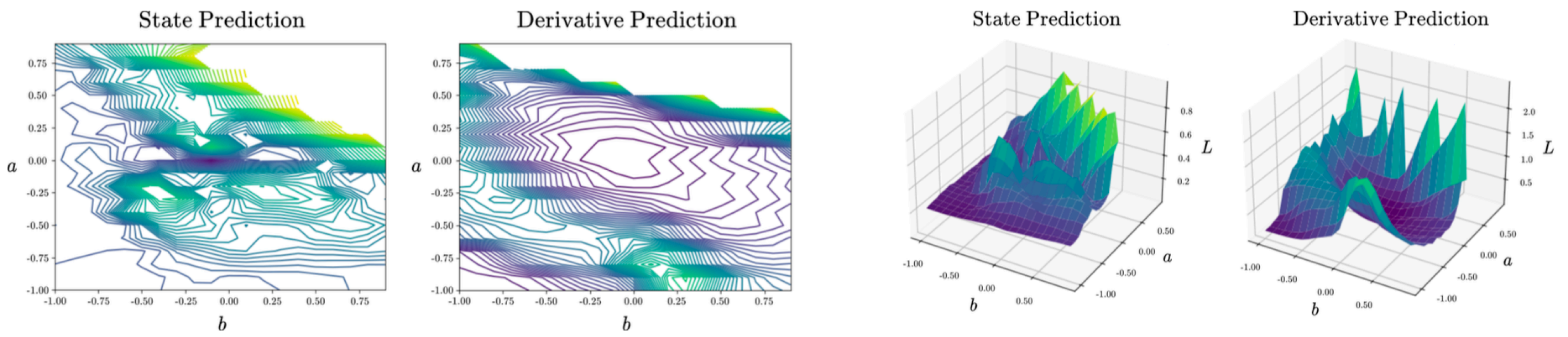
What drives this dependence on Δt?





- Even with perfect derivatives, numerical error still accumulates (truncation error)
- In early stages, numerical error drives error accumulation. Larger Δt exacerbates this.
- In later stages, model error contributes to error propagation + instability
  - Majority of performance gains to be made by improving model

Why does derivative prediction work well? **assuming sufficiently small Δt*



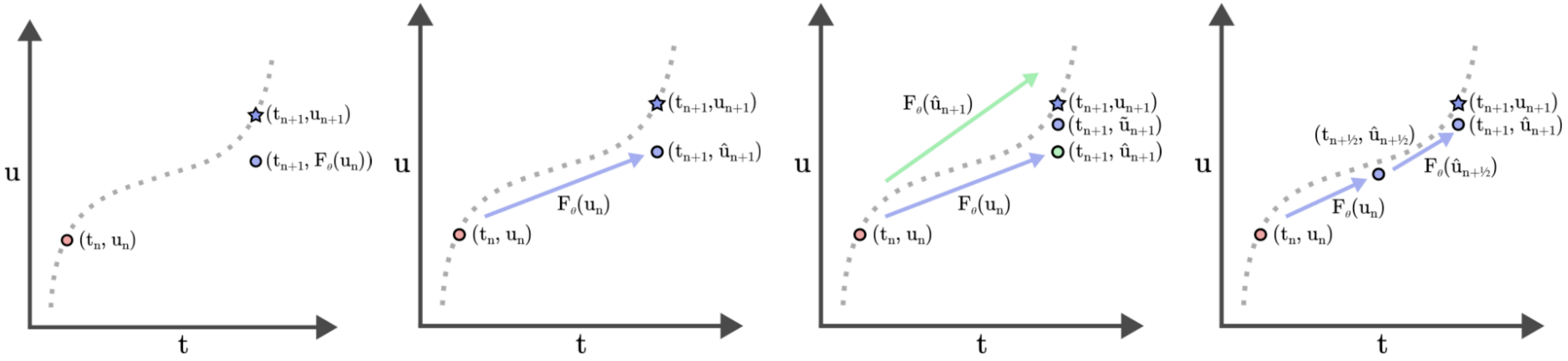
- Plotting rollout loss vs. perturbed, trained weights: $\theta^* + a\delta + b\gamma$
- Hypotheses:
  - Predicting the change in the state is more informative than the state itself
  - Better stability from only adding a small change $\Delta t \cdot u'(t)$ vs. predicting an entirely new state $u(t+1)$, assuming previous state is accurate.

Thank you for listening!

Questions/Comments?

Carnegie Mellon University

Appendix: Overview



(a) State prediction. Models are trained to directly predict $\mathbf{u}_{n+1} = F_\theta(\mathbf{u}_n)$.

(b) Derivative prediction. Models predict $\frac{\partial \mathbf{u}}{\partial t}|_{t=t_n} = F_\theta(\mathbf{u}_n)$ to solve $\hat{\mathbf{u}}_{n+1}$.

(c) Derivative prediction with higher order integrators can be more accurate.

(d) Derivative prediction with variable step sizes can be more flexible.

Fig. 1: A comparison of state prediction and derivative prediction, where models are either trained to predict $\mathbf{u}_{n+1}$ or $\frac{\partial \mathbf{u}}{\partial t}|_{t=t_n}$. During inference, models are given an initial solution $\mathbf{u}_n$, and predict future solutions along the dashed trajectory. By predicting the temporal derivatives rather than the future solution, derivative prediction can learn spatial updates while an ODE integrator updates the solution in time, which can improve accuracy. Furthermore, derivative prediction can use higher-order integrators or variable timesteps, which further improves its accuracy and flexibility, while being applicable across model architectures and datasets.

Appendix: Integration Schemes

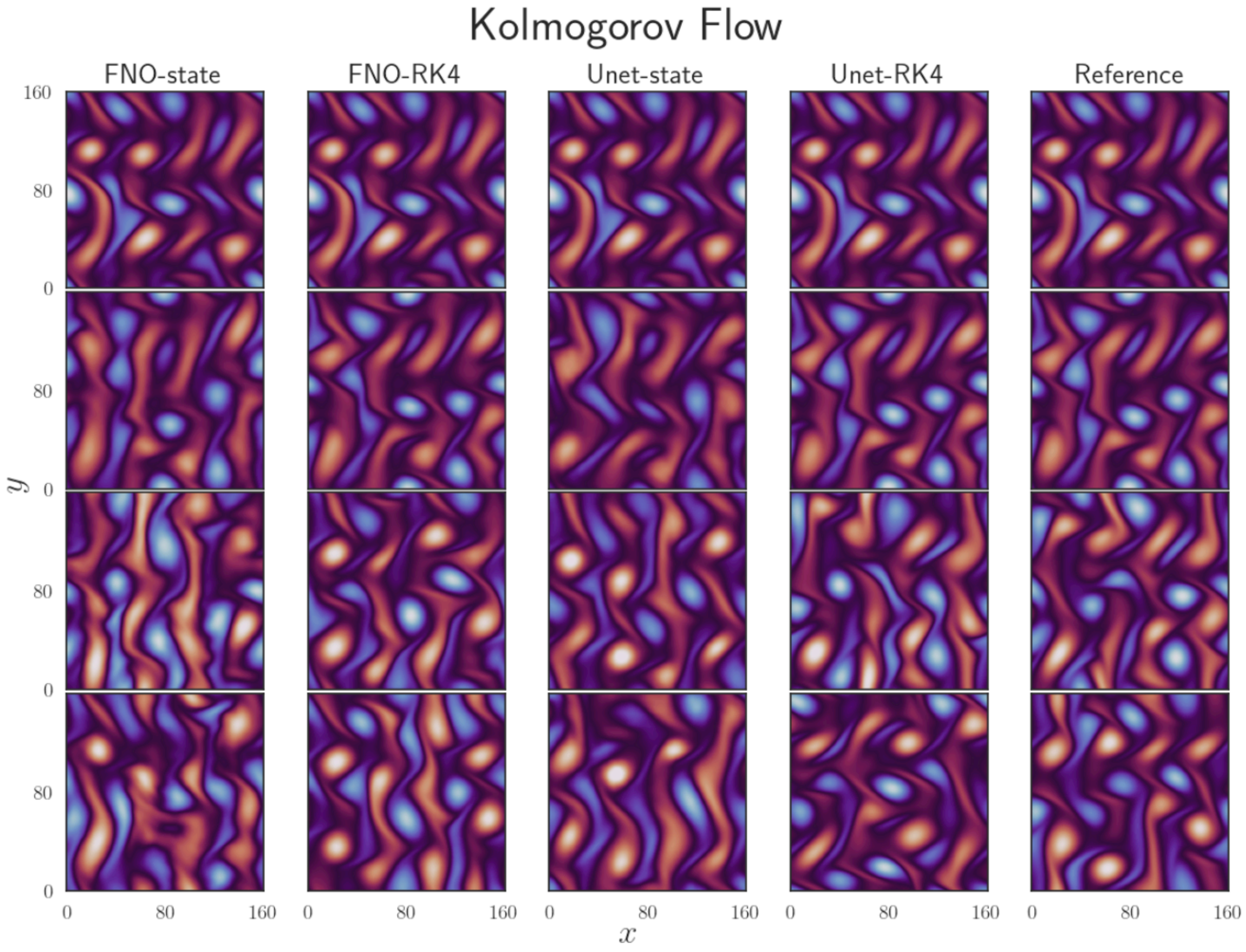
$$\mathbf{u}(t_{n+1}) = \mathbf{u}(t_n) + \Delta t F_\theta(\mathbf{u}(t_n), t_n) \qquad \text{(Forward Euler)}$$

$$\mathbf{u}(t_{n+1}) = \mathbf{u}(t_n) + \frac{3\Delta t}{2} F_\theta(\mathbf{u}(t_n), t_n) - \frac{\Delta t}{2} F_\theta(\mathbf{u}(t_{n-1}), t_{n-1}) \qquad \text{(Adams-Bashforth)}$$

$$\begin{aligned} \tilde{\mathbf{u}}(t_{n+1}) &= \mathbf{u}(t_n) + \Delta t F_\theta(\mathbf{u}(t_n), t_n) \\ \mathbf{u}(t_{n+1}) &= \mathbf{u}(t_n) + \frac{\Delta t}{2}(F_\theta(\mathbf{u}(t_n), t_n) + F_\theta(\tilde{\mathbf{u}}(t_{n+1}), t_{n+1}) \end{aligned} \qquad \text{(Heun's Method)}$$

$$\begin{aligned} k_1 &= F_\theta(\mathbf{u}(t_n), t_n) \\ k_2 &= F_\theta(\mathbf{u}(t_n) + \Delta t \frac{k_1}{2}, t_n + \frac{\Delta t}{2}) \\ k_3 &= F_\theta(\mathbf{u}(t_n) + \Delta t \frac{k_2}{2}, t_n + \frac{\Delta t}{2}) \\ k_4 &= F_\theta(\mathbf{u}(t_n) + \Delta t k_3, t_n + \Delta t) \\ \mathbf{u}(t_{n+1}) &= \mathbf{u}(t_n) + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4) \end{aligned} \qquad \text{(4th-order Runge-Kutta)}$$

Appendix: Kolmogorov Flow



Appendix: Training Modifications

PDE: Metric:	Adv Roll. Err. ↓	Heat Roll. Err. ↓	KS Corr. Time ↑	Burgers Roll. Err. ↓	NS Roll. Err. ↓	Kolm. Flow Corr. Time ↑
FNO (State Pred.)	0.498	0.589	139.75	0.437	0.715	51.4
FNO (4x params)	0.826	0.991	147.75	0.276	0.599	67.4
FNO (Pushforward)	0.357	0.486	141	0.397	0.090	53.3
FNO (Refiner)	0.036	0.167	159.75	1.141	0.596	37.5
FNO (RK4)	0.033	0.141	197.25	0.175	0.100	82.3
FNO (RK4+Pushforward)	0.023	0.140	194.25	0.322	0.058	83.8
Unet (State Pred.)	0.044	0.149	153.5	0.666	0.099	35.1
Unet (4x params)	0.036	0.144	145.25	0.222	0.053	52.1
Unet (Pushforward)	0.048	0.149	145	0.746	0.078	36.0
Unet (Refiner)	0.082	0.176	145.5	0.217	0.180	51.1
Unet (RK4)	0.010	0.139	174.5	0.264	0.049	88.9
Unet (RK4+Pushforward)	0.010	0.139	177	0.264	0.034	90.6

Table 2: **Training Modifiers.** Comparing different training modifications on various PDEs, models and frameworks.

Appendix: Noised Trajectories

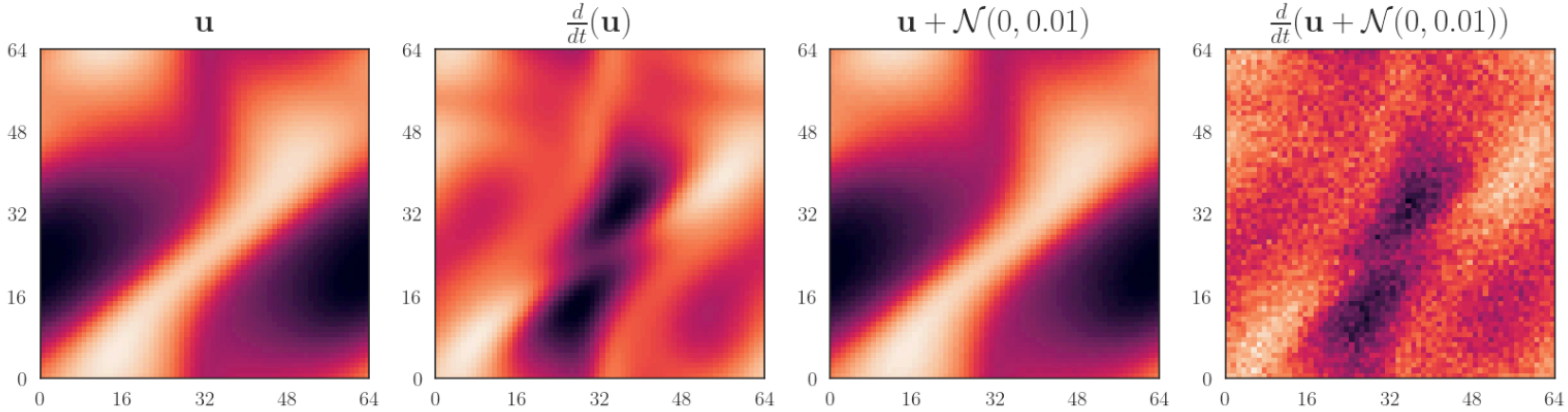


Fig. 6: **Noised Trajectories.** Snapshots of the 2D Navier-Stokes equation, along with their temporal derivatives. Adding noise to a trajectory creates a similar sample, however this effect is noticeable when taking its temporal derivative.

Appendix: Next-Step Error

Model	Adv	Heat	KS	Burgers	NS	Kolm. Flow
FNO-State (Next-Step Error)	0.0048	0.0053	0.0012	0.0382	0.0016	0.0214
FNO-Derivative (Next-Step Error)	0.0004	0.0020	0.0003	0.0045	0.0005	0.0038
FNO-Derivative (Derivative Error)	0.0204	0.2665	0.0047	0.1104	0.0234	0.0522
Unet-State (Next-Step Error)	0.0041	0.0044	0.0013	0.0291	0.0018	0.0254
Unet-Derivative (Next-Step Error)	0.0003	0.0018	0.0008	0.0068	0.0008	0.0094
Unet-Derivative (Derivative Error)	0.0101	0.3065	0.0142	0.1700	0.0402	0.1256

Table 4: **Next-Step Error.** Results on validation next-step error across different PDEs, models, and training/inference frameworks. Derivative error is calculated with an RK4 integrator. Next-step error for state-prediction models is given by: $\mathcal{L}(\mathbf{u}(t_{n+1}), F_\theta(\mathbf{u}(t_n)))$, while for derivative-prediction models both the derivative error: $\mathcal{L}(\frac{d\mathbf{u}}{dt}|_{t=t_n}, F_\theta(\mathbf{u}(t_n)))$ and next-step error: $\mathcal{L}(\mathbf{u}(t_{n+1}), \int F_\theta(\mathbf{u}(t_n)))$ are given.

Appendix: Timing Experiments

Model:	FNO				Unet				Solver		
Setup:	State	Adams	Heun	RK4	State	Adams	Heun	RK4	64x64	32x32	16x16
Runtime (s)	0.163	0.158	0.301	0.622	0.359	0.359	0.691	1.379	3.508	2.692	2.549
Rollout Error	0.715	0.100	0.100	0.100	0.099	0.048	0.049	0.049	0.000	0.096	0.285

Table 5: **Computational Cost.** Comparison of computational cost of different models and a baseline solver on the Navier-Stokes equations. Runtimes are reported in seconds (s) for a full rollout, averaged for each sample in the validation set. Rollout errors are given as relative L2 error.