Dynestyx: A probabilistic programming library for dynamical systems

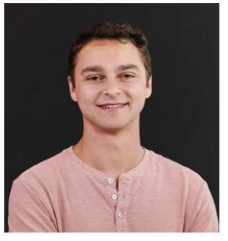
Matthieu Darcy

Basis Research Institute

Joint/ongoing work with



Daniel Waxman
Basis Research Institute & MIT



Matthew Levine
Basis Research Institute



Youssef Marzouk
MIT AeroAstro



Ricardo Baptista
University of Toronto

BASIS

Mathematical and Computational Foundations
of Digital Twins, Cargèse
November 2–6, 2026

Digital twins: unifying different tasks



observations y_t

actions u_t

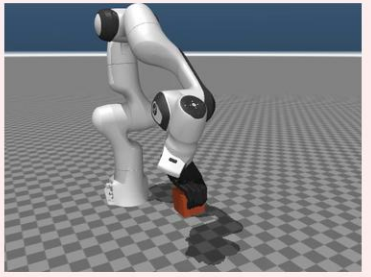
Physical system

Digital twin

Data assimilation

State estimation $\hat{p}_t(x_t)$

System identification $\hat{p}(\theta)$



Control

Model predictive control

Open-loop control

...

update x_t, θ

candidate u

Forward model

$x_t \sim p(x_t \mid x_{t-1}, u_t, \theta)$

states x_t, parameters θ

Reich, S. (2025, April 1). Digital twins: Synergy between data assimilation and optimal control. *SIAM News, 58*(3). https://www.siam.org/publications/siam-news/articles/digital-twins-synergy-between-data-assimilation-and-optimal-control/

Haddadin, S., Parusel, S., Johannsmeier, L., Golz, S., Gabl, S., Walch, F., Sabaghian, M., Jähne, C., Hausperger, L., & Haddadin, S. (2022). The Franka Emika robot: A reference platform for robotics research and education. *IEEE Robotics & Automation Magazine, 29*(2), 46–64. https://doi.org/10.1109/MRA.2021.3138382

Photo: Ims, CC BY-SA 4.0, cropped, via Wikimedia Commons.



- `dynestyx` is a probabilistic programming library for dynamical systems (DW et al., 2026)
- A unified interface: write the model once, then simulate, infer states, infer parameters, or control
- Provides state-of-the-art methods for Bayesian inference in dynamical systems
- Allows researchers to easily swap one component for another, without rebuilding the entire model
- In development: we want to understand what is useful to the community

Waxman, D., Batenkov, D., Feser, J., Zane, A., Bingham, E., Marzouk, Y., & Levine, M. E. (2026). *Dynestyx: A probabilistic programming library for dynamical systems* (arXiv:2606.16985). arXiv. https://doi.org/10.48550/arXiv.2606.16985

State space models as a formalism

- A state $x_t \in \mathbb{R}^{d_x}$, in continuous or discrete time, with $x_0 \sim p(x_0)$
- Noisy, partial observations $y_t \in \mathbb{R}^{d_y}$, in discrete time
- Unknown parameters $\theta \in \mathbb{R}^{d_\theta}$
- Controls u_t: recorded inputs, or chosen online by a policy

$$x_t \sim \overbrace{p(x_t \mid x_{t-1}, u_t, \theta)}^{\text{transition density}} \quad \text{or} \quad \mathrm{d}x_t = \overbrace{f_\theta(x_t, u_t, t)\,\mathrm{d}t}^{\text{drift}} + \overbrace{g_\theta(x_t, t)\,\mathrm{d}W_t}^{\text{diffusion}}$$

$$y_t \sim p(y_t \mid x_t, u_t, \theta)$$

- Special cases: ODEs, SDEs, discretized PDEs, and black-box simulators

Four problems, one model

Problem	Target	Methods in `dynestyx`
Simulation	$x_{0:T}, y_{1:T} \sim p(\cdot \mid u_{1:T}, \theta)$	ODE/SDE solvers, external simulators
State inference	$p(x_t \mid y_{1:t}, u_{1:t}, \theta)$	KF, EKF, UKF, EnKF, particle filters, smoothers
Parameter inference	$p(\theta \mid y_{1:T}, u_{1:T})$	NUTS, SVI, MAP estimation, pseudo-marginal MCMC, gradient-free samplers
Control	$u_t = \pi(\hat{p}_{t-1})$	Feedback, learned policies, MPPI, your own controller

- These tasks are inter-related:
  - Parameter inference can use filters for better scalability
  - A model predictive controller can call the simulator
  - A controller can use a filter for decision making under uncertainty
  - Control needs system identification

Dynestyx is a probabilistic programming language

- Like a normal programming language, but the variables are random variables...
  - Or in our case, dynamical systems
- Built on NumPyro: can leverage its inference methods
- Written in JAX (scalable + GPU + autodiff)
- “Separation of concerns”: model code describes a mathematical ideal, and all approximations come after
- Keeping the two apart enables Bayesian workflow: write a model, run inference, check it, refine



Building a `dynestyx` model

- We write down an idealized version, from the mathematical description
- Write down the uncertain parameters: θ

$$x_0 \sim \mathcal{N}(0, 1)$$

$$\mathrm{d}x_t = \sin(x_t)\,\mathrm{d}t + \mathrm{d}W_t$$

$$y_t \sim \mathcal{N}(x_t, 0.1^2)$$

observe $y_{1:T}$ at $t_{1:T}$

```
def model(obs_times=None, obs_values=None, predict_times=None):
    dynamics = dsx.DynamicalModel(
        initial_condition=dist.MultivariateNormal(
            jnp.zeros(1), jnp.eye(1)),
        state_evolution=dsx.ContinuousTimeStateEvolution(
            drift=lambda x, u, t: jnp.sin(x),
            diffusion=dsx.ScalarDiffusion(1.0, bm_dim=1),
        ),
        observation_model=dsx.LinearGaussianObservation(
            H=jnp.eye(1), R=0.1**2 * jnp.eye(1)),
    )
    dsx.sample("f", dynamics, obs_times=obs_times,
               obs_values=obs_values, predict_times=predict_times)
```

A `dynestyx` model from the ground up

- We write down an idealized version, from the mathematical description
- Write down the uncertain parameters: θ

$$\theta \sim \text{HalfNormal}(0.5)$$

$$x_0 \sim \mathcal{N}(0, 1)$$

$$\mathrm{d}x_t = \theta \sin(x_t)\,\mathrm{d}t + \mathrm{d}W_t$$

$$y_t \sim \mathcal{N}(x_t, 0.1^2)$$

observe $y_{1:T}$ at $t_{1:T}$

```
def model(obs_times=None, obs_values=None, predict_times=None):
    theta = numpyro.sample("theta", dist.HalfNormal(0.5))
    dynamics = dsx.DynamicalModel(
        initial_condition=dist.MultivariateNormal(
            jnp.zeros(1), jnp.eye(1)),
        state_evolution=dsx.ContinuousTimeStateEvolution(
            drift=lambda x, u, t: theta * jnp.sin(x),
            diffusion=dsx.ScalarDiffusion(1.0, bm_dim=1),
        ),
        observation_model=dsx.LinearGaussianObservation(
            H=jnp.eye(1), R=0.1**2 * jnp.eye(1)),
    )
    dsx.sample("f", dynamics, obs_times=obs_times,
               obs_values=obs_values, predict_times=predict_times)
```

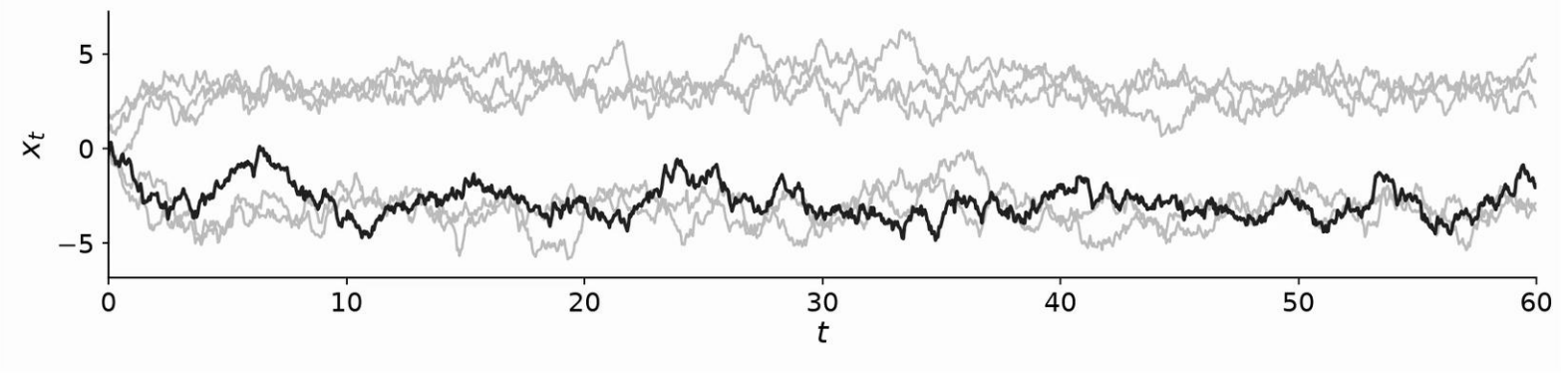
- One model, three questions of increasing difficulty:
- Simulate: $x_{0:T} \sim p(x_{0:T} \mid \theta)$
- Estimate states: $p(x_t \mid y_{1:t}, \theta)$
- System identification: $p(\theta \mid y_{1:T})$

Simulation

Step 1 of 3: θ known, draw trajectories

- Fix $\theta = 1$ and sample $x_{0:T} \sim p(x_{0:T} \mid \theta)$ and $y_k \sim \mathcal{N}(x_{t_k}, 0.1^2)$
- In the backend, we can call state-of-the-art integrators from `diffrax`

```
with dsx.Simulator():
    sim = Predictive(model, params={"theta": 1.0},
                     num_samples=6)(key, predict_times=ts)
```



The only variability is aleatoric uncertainty.

State estimation

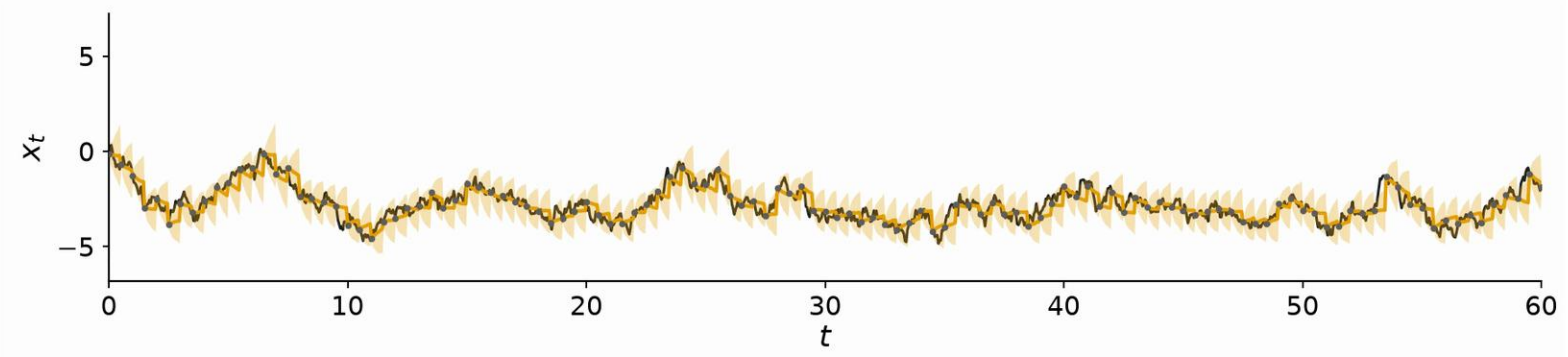
Step 2 of 3: θ known, recover the unobserved state

$$\text{predict:} \quad p(x_k \mid y_{1:k-1}, \theta) = \int p(x_k \mid x_{k-1}, \theta)\, p(x_{k-1} \mid y_{1:k-1}, \theta)\, \mathrm{d}x_{k-1}$$

$$\text{update:} \quad p(x_k \mid y_{1:k}, \theta) \propto p(y_k \mid x_k, \theta)\, p(x_k \mid y_{1:k-1}, \theta)$$

- Call state-of-the-art filters from `cuthbert` and `cd-dynamax` (Kalman filter, EnKF, ...)

```
with Filter(EnKFConfig(n_particles=400, ...)):
    with Discretizer(EulerMaruyamaConfig()):
        out = Predictive(model, posterior_samples={"theta": theta})(
            key, obs_times=ts, obs_values=ys)
```



Observations every 0.5 s. Black: truth. Orange: filter mean $\pm$ 2 sd.

System identification

Step 3 of 3: θ unknown, two ways to do inference

Sample the latent states

$$p(y_{1:T} \mid \theta) = \int p(x_0) \prod_{k=1}^{T} p(y_k \mid x_k, \theta)\, p(x_k \mid x_{k-1}, \theta)\, \mathrm{d}x_{0:T}$$

```
with LatentPathBuilder():
    with Discretizer():
        mcmc = MCMC(NUTS(model), ...)
```

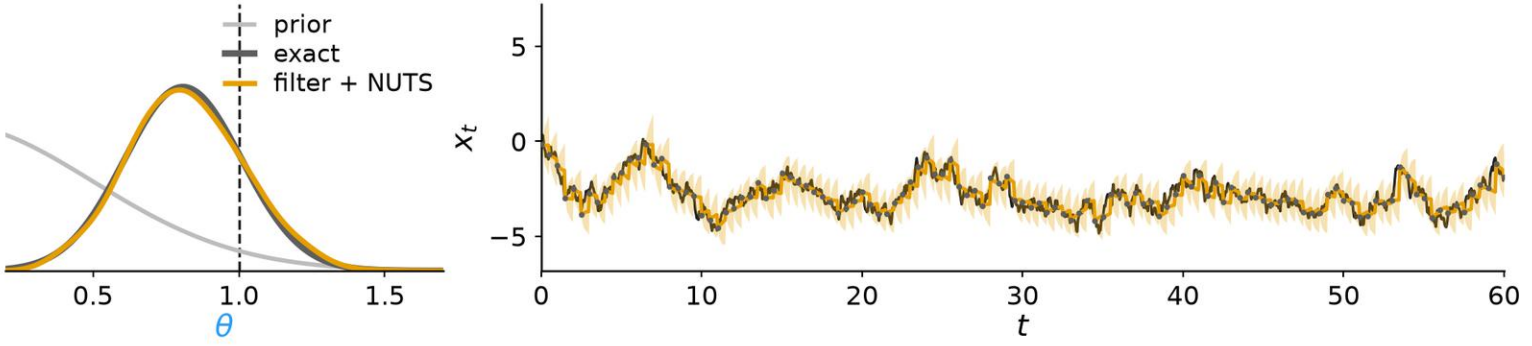
- Can work well for small problems
- Difficult to scale to high dimensions

Filter as an approximate likelihood

$$p(y_{1:T} \mid \theta) = \prod_{k=1}^{T} \int p(y_k \mid x_k, \theta)\, p(x_k \mid y_{1:k-1}, \theta)\, \mathrm{d}x_k$$

```
with Filter(EnKFConfig(...)):
    with Discretizer():
        mcmc = MCMC(NUTS(model), ...)
```

- Use the Filter to compute approximate likelihoods instead
- We get the filtered states for free



Same model, swap the method

Change the methods without rebuilding the model each time

Simulate: Euler to Runge–Kutta

$$x_{n+1} = x_n + h\, f(x_n) + \Delta W_n$$

$$x_{n+1} = x_n + \tfrac{h}{2}\big(f(x_n) + f(\tilde{x}_{n+1})\big) + \Delta W_n$$

```
solver = dfx.Euler()  # swap for dfx.Heun()
with SDESimulator(SDESimulatorConfig(solver=solver, ...)):
    sim = Predictive(model, ...)(key, predict_times=ts)
```

Filter: Gaussian or particles

$$p(x_k \mid y_{1:k}, \theta) \approx \mathcal{N}(m_k, P_k)$$

$$p(x_k \mid y_{1:k}, \theta) \approx \textstyle\sum_i w_k^i\, \delta_{x_k^i}$$

```
config = EnKFConfig()  # swap for EKFConfig(), PFConfig()
with Filter(config):
    with Discretizer():
        mcmc = MCMC(NUTS(model), ...)
```

Inference: MCMC, variational, MAP

$$\theta^{(i)} \sim p(\theta \mid y_{1:T})$$

$$\min_\phi \ \mathrm{KL}\big(q_\phi(\theta) \,\|\, p(\theta \mid y_{1:T})\big)$$

$$\arg\max_\theta \ p(\theta \mid y_{1:T})$$

```
mcmc = MCMC(NUTS(model), ...)              # MCMC
svi = SVI(model, AutoNormal(model), ...)   # variational
svi = SVI(model, AutoDelta(model), ...)    # MAP
```

State-of-the-art backends underneath: `cuthbert`, `diffrax`, NumPyro, BlackJAX

What about control?

Discrete time closed loop control in `dynestyx`

The system, with a control

$$\mathrm{d}x_t = \left(\theta \sin(x_t) + u_{t_k}\right)\mathrm{d}t + \mathrm{d}W_t, \quad t \in [t_k, t_{k+1})$$
$$y_{t_k} \sim \mathcal{N}(x_{t_k}, 0.1^2)$$

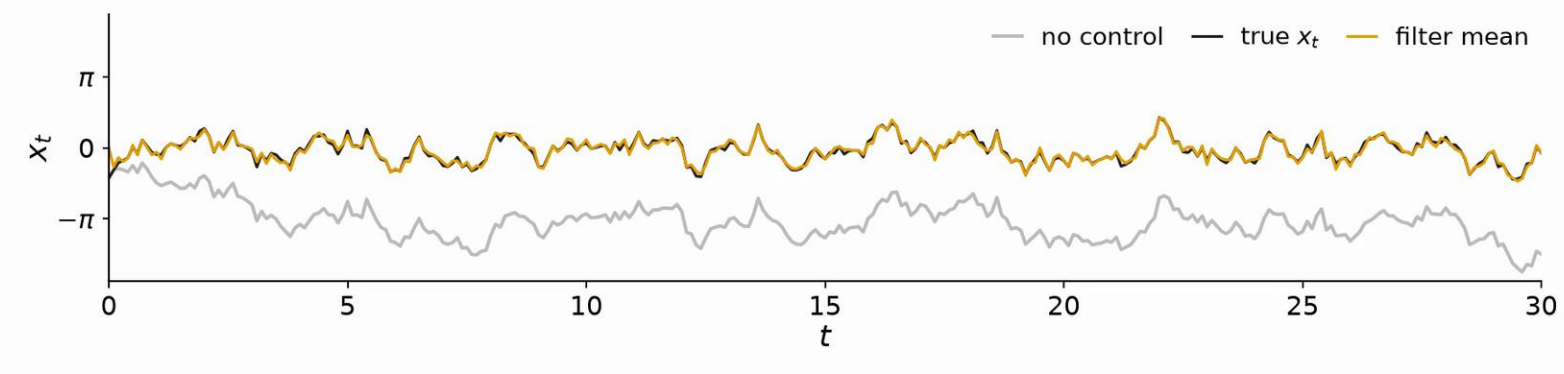
Goal: hold x_t at the unstable point 0, with $\theta = 1$

Unknown state: the policy sees the filtered state

The closed loop, $k = 0, \ldots, N-1$

$$u_{t_k} = \pi(\hat{x}_{t_k})$$
$$x_{t_{k+1}} \sim p(x_{t_{k+1}} \mid x_{t_k}, u_{t_k}, \theta)$$
$$y_{t_{k+1}} \sim p(y_{t_{k+1}} \mid x_{t_{k+1}}, \theta)$$
$$\hat{x}_{t_{k+1}} = \mathrm{FilterUpdate}(\hat{x}_{t_k}, u_{t_k}, y_{t_{k+1}})$$

```
result = dsx.simulate(dynamics, rng_key=key, predict_times=ts,
                      control_policy=policy,
                      filter_config=EnKFConfig(n_particles=400))
```



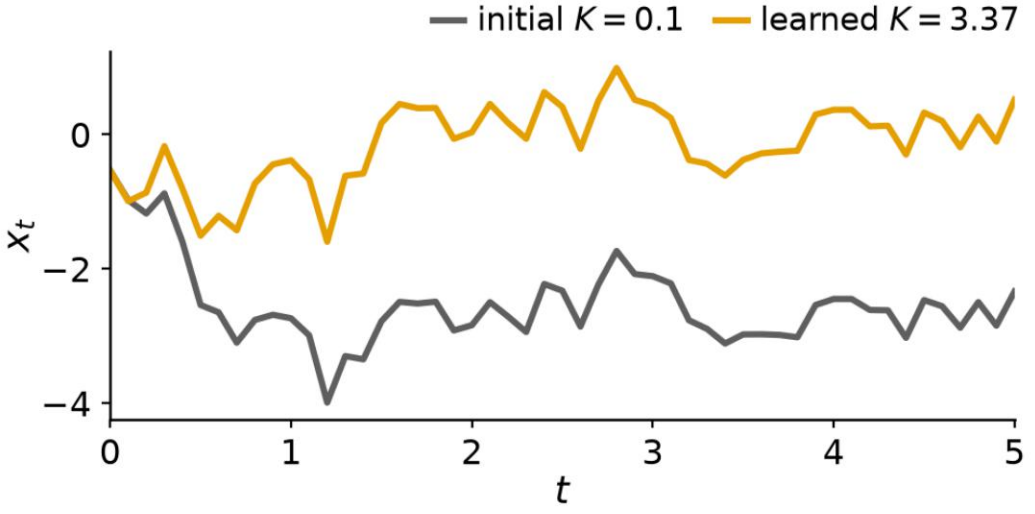
Optimizing the controller

The closed loop is differentiable, so gradients flow through

$$\min_{K} \mathbb{E}\Big[\frac{1}{N}\sum_{k} x_{t_k}^2 + 0.1\, u_{t_k}^2\Big], \quad u_{t_k} = -K\,\hat{x}_{t_k}$$

```
def cost(K, key):
    result = dsx.simulate(dynamics, rng_key=key,
        ..., control_policy=LinearPolicy(K))
    x, u = result.states, result.controls
    return jnp.mean(x**2) + 0.1 * jnp.mean(u**2)
grad = jax.grad(cost)(K, key)
```

We learn the optimal K



Going beyond toy models

High dimensional systems and robotics

PDE system identification: a hybrid model

Known diffusion, unknown reaction

True system

$$\frac{\partial u}{\partial t} = \nu \frac{\partial^2 u}{\partial x^2} + 1.1u(1-u)$$

$$x \in (0,1), \quad u(t,0) = u(t,1) = 0, \quad \nu = 0.01$$

Hybrid model

$$\dot{\mathbf{u}} = \underbrace{\nu L_h \mathbf{u}}_{\text{known diffusion}} + \underbrace{c_\theta(\mathbf{u})}_{\text{learned reaction}}$$

$$\mathbf{y}_k \mid \mathbf{u}(t_k) \sim \mathcal{N}\big(\mathbf{u}(t_k), \sigma_y^2 I\big)$$

- We learn the missing term with a Gaussian process (GP)
  - Specifically, we use a Hilbert space GP (HSGP) approximation (Solin & Särkkä, 2020)

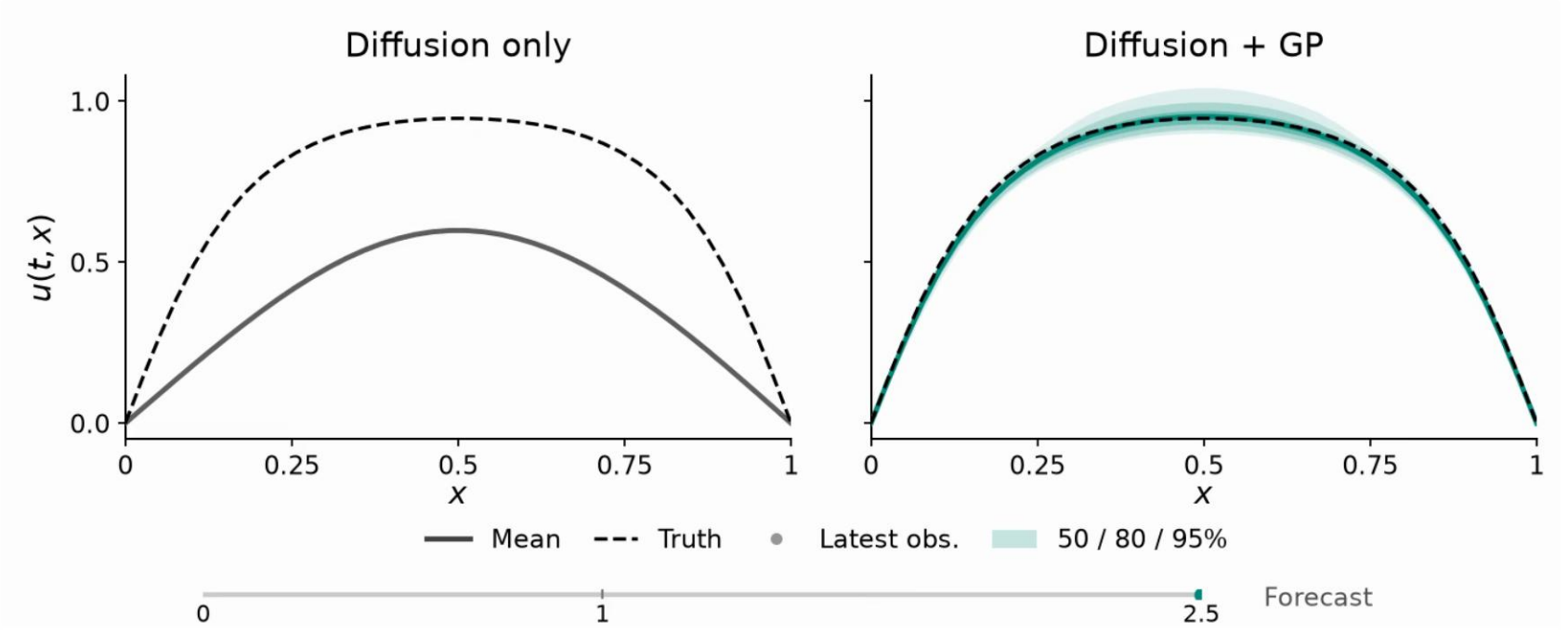
Solin, A., & Särkkä, S. (2020). Hilbert space methods for reduced-rank Gaussian process regression. *Statistics and Computing, 30*(2), 419–446. https://doi.org/10.1007/s11222-019-09886-w

PDE system identification: a hybrid model

Fitted on $t \in [0, 1]$, then forecast to $t = 2.5$ with no observations

```
c = sample_hsgp()  # GP prior for the reaction
dynamics = dsx.ContinuousTimeStateEvolution(
    drift=lambda u, _, t: nu * Lh @ u + c(u))
```

```
guide = AutoMultivariateNormal(filtered_model, **guide_init)
```



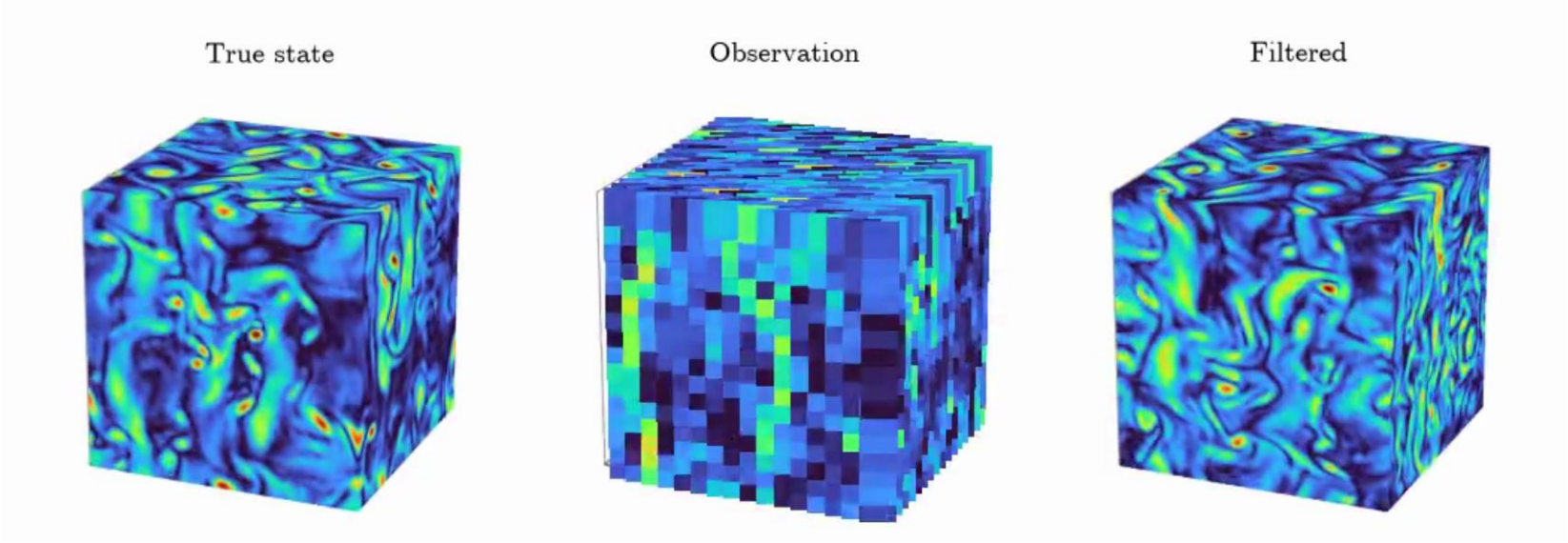
PDE filtering: 3D Navier–Stokes

Reconstructing a turbulent flow from coarse, noisy measurements

$$\partial_t \mathbf{u} = \nu \Delta \mathbf{u} + \lambda \mathbf{u} + \mathcal{P}(\mathbf{u} \times \boldsymbol{\omega}) + \mathbf{f}, \qquad \boldsymbol{\omega} = \nabla \times \mathbf{u}, \qquad \nabla \cdot \mathbf{u} = 0$$

- State: velocity $\mathbf{u}$ on a 64^3 grid, $x_t \in \mathbb{R}^{786{,}432}$
- Observed: noisy 16^3 grid, $y_t \in \mathbb{R}^{12{,}288}$
- External solver (Exponax) with EnKF

```
config = EnKFConfig(n_particles=7,
                    ensemble_subspace=True, ...)
with Filter(config):
    result = dsx.condition("u", dynamics,
        obs_times=ts, obs_values=ys)
```



Bring your own solver

- Up until now, every model was hand-written in JAX
- But what if you want to run a solver (possibly outside JAX): contact physics, CFD, legacy codes...
- We make it easy to communicate between `dynestyx` and external solvers
- `dynestyx` only needs the transition, as a callable $(x_{t-1}, u_t, t, t') \mapsto x_{t'}$
- Example: MuJoCo (Todorov et al., 2012), an external solver for rigid bodies

External Solver

$x_{t'} = S(x_{t-1}, u_t; \theta)$

```
def step(x, u, t_now, t_next):
    x, u = layout.state.unflatten(x), layout.control.unflatten(u)
    qpos, qvel = jax.pure_callback(
        mujoco_stepper, shapes, x.qpos, x.qvel, u, mu, mass,
        vmap_method="broadcast_all")
    return layout.state.flatten(MujocoState(qpos, qvel))
```

the model, as before

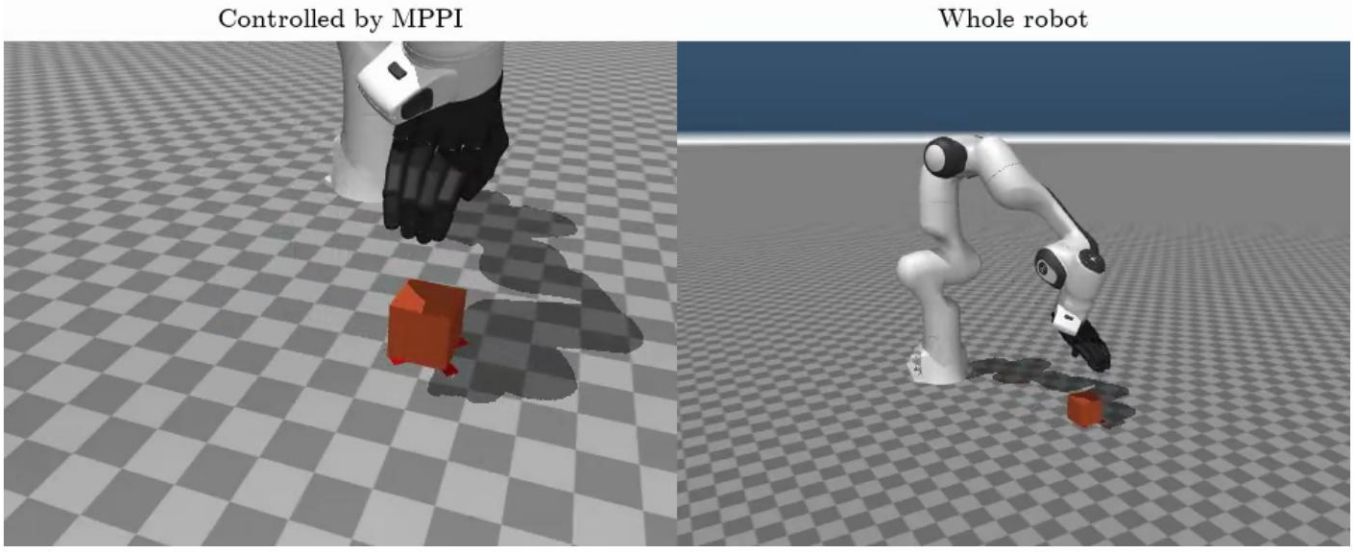
```
dynamics = DynamicalModel(
    initial_condition=dist.Normal(x0, initial_std).to_event(1),
    state_evolution=DeterministicStateEvolution(F=step),
    observation_model=observation_model,
    control_dim=layout.control.dim)
```

Todorov, E., Erez, T., & Tassa, Y. (2012). MuJoCo: A physics engine for model-based control. *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 5026–5033. https://doi.org/10.1109/IROS.2012.6386109

Doing Model Predictive control with MuJoCo

- MPPI (Williams et al., 2017) sampling-based model predictive control: sample candidates, weight them and execute the first control
- Can use the same solver or a different one internally
- You can design your own controller

```
mppi = NormalizedMPPI(dynamics=dynamics, loss_fn=loss_fn, horizon=horizon,
                      noise=AR1Noise(horizon, 4, rho=0.5), n_samples=128, ...)
result = dsx.simulate(dynamics, rng_key=key, predict_times=ts,
    control_policy=mppi, use_true_state=True,
    initial_policy_state=mppi.initial_state())
```

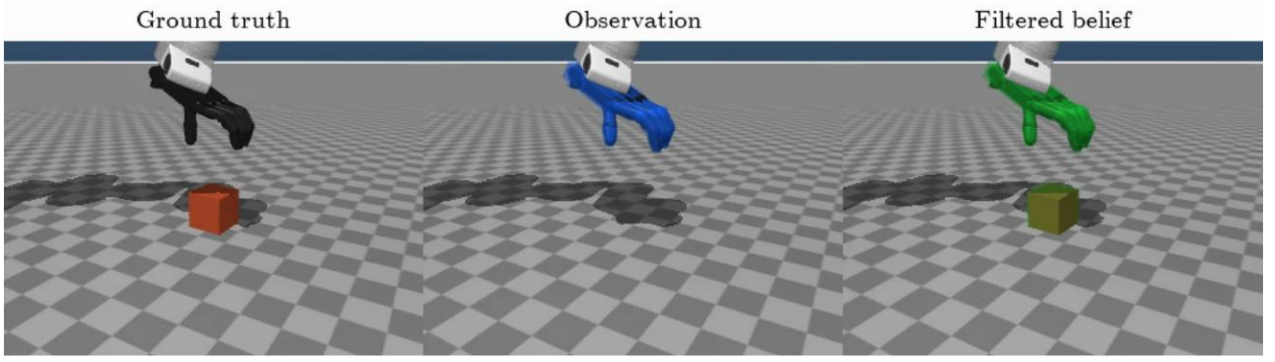


Williams, G., Aldrich, A., & Theodorou, E. A. (2017). Model predictive path integral control: From theory to parallel computation. *Journal of Guidance, Control, and Dynamics, 40*(2), 344–357. https://doi.org/10.2514/1.G001921

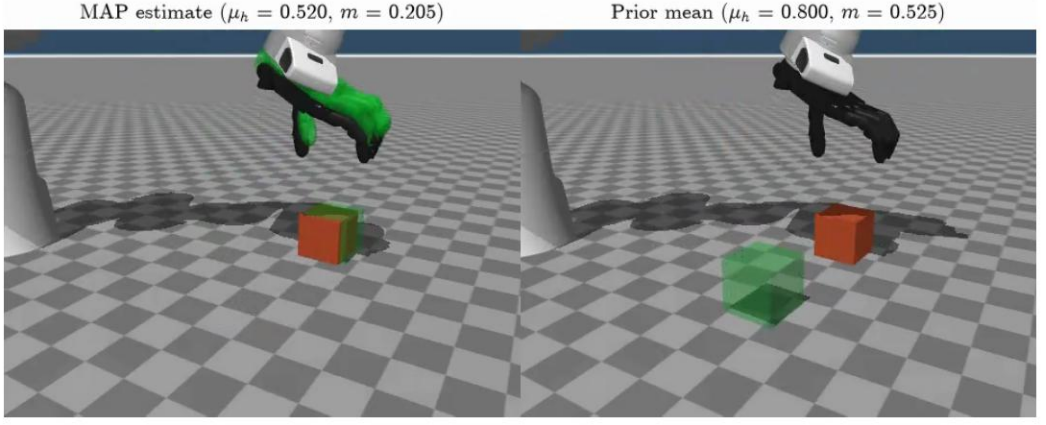
Doing state and system inference with MuJoCo

Online: u_t, x_t. *Offline:* θ

- We observe the robot's joints only, the cube is hidden and we use the EnKF to recover it



- The same filter gives the likelihood for system identification (optimized using Bayesian optimization)



	Truth	MAP
μ_h	0.5	0.520
m [kg]	0.2	0.205

Truth with green ghosts: rollout at the MAP (left) and at the prior mean (right)

Work in progress: online control + state + parameter estimation

Everything else we can do

- Ensemble NUTS: EnKF likelihood + gradient-based MCMC (Drovandi et al., 2022; Chen et al., 2022)
- Learning filters via proper scoring rules (Adrian et al., 2026; Bach et al., 2026)
- Bayesian SINDy: sparse equation discovery (Brunton et al., 2016; Hirsh et al., 2022)
- Also: neural networks in the drift, hierarchical models, HMMs with missing observations...

Compositional structure unlocks new methods

Drovandi, C., Everitt, R. G., Golightly, A., & Prangle, D. (2022). Ensemble MCMC: Accelerating pseudo-marginal MCMC for state space models using the ensemble Kalman filter. *Bayesian Analysis, 17*(1). https://doi.org/10.1214/20-BA1251

Chen, Y., Sanz-Alonso, D., & Willett, R. (2022). Autodifferentiable ensemble Kalman filters. *SIAM Journal on Mathematics of Data Science, 4*(2), 801–833. https://doi.org/10.1137/21M1434477

Adrian, M., Sanz-Alonso, D., & Willett, R. (2026). *Auto-differentiable data assimilation: Co-learning of states, dynamics, and filtering algorithms* (arXiv:2603.20891). arXiv. https://doi.org/10.48550/arXiv.2603.20891

Bach, E., Baptista, R., Bröcker, J., Chen, B., & Stuart, A. (2026). *Learning probabilistic filters with strictly proper scoring rules* (arXiv:2606.26497). arXiv. https://doi.org/10.48550/arXiv.2606.26497

Brunton, S. L., Proctor, J. L., & Kutz, J. N. (2016). Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the National Academy of Sciences, 113*(15), 3932–3937. https://doi.org/10.1073/pnas.1517384113

Hirsh, S. M., Barajas-Solano, D. A., & Kutz, J. N. (2022). Sparsifying priors for Bayesian uncertainty quantification in model discovery. *Royal Society Open Science, 9*(2), Article 211823. https://doi.org/10.1098/rsos.211823

Dynestyx is ready-to-use

- `dynestyx` provides the state-of-the-art in inference for practitioners to use, helping to close a gap between theory and application
- Algorithms are available for continuous & discrete time, deterministic & stochastic processes
- We support control, learning of filters, and lots more
- Combine multiple tasks to model a system: simulate, estimate and control
- The framework `dynestyx` creates implies new algorithms, and an avenue for applying new methods

Questions?

matthieu@basis.ai

`dynestyx`



Paper



GitHub

Waxman, D., Batenkov, D., Feser, J., Zane, A., Bingham, E., Marzouk, Y., & Levine, M. E. (2026). *Dynestyx: A probabilistic programming library for dynamical systems* (arXiv:2606.16985). arXiv. https://doi.org/10.48550/arXiv.2606.16985

Backup

There's lots left to do

- Our thesis: closing the theory-application gap will uncover a wide array of new methodological work
  - Applications will have their own difficulties, and we need methods that effectively detect and address them
- We want to provide support to people who want to use dynestyx
  - Develop benchmarks based on challenges we see
  - Develop diagnostics that can detect difficulties
  - Ultimately, use our general framework + knowledge we gain to develop new methods

The filter marginalizes out the states

$$p(\theta \mid y_{1:T}) = \int p(\theta, x_{0:T} \mid y_{1:T})\, \mathrm{d}x_{0:T} \propto p(\theta)\, p(y_{1:T} \mid \theta)$$

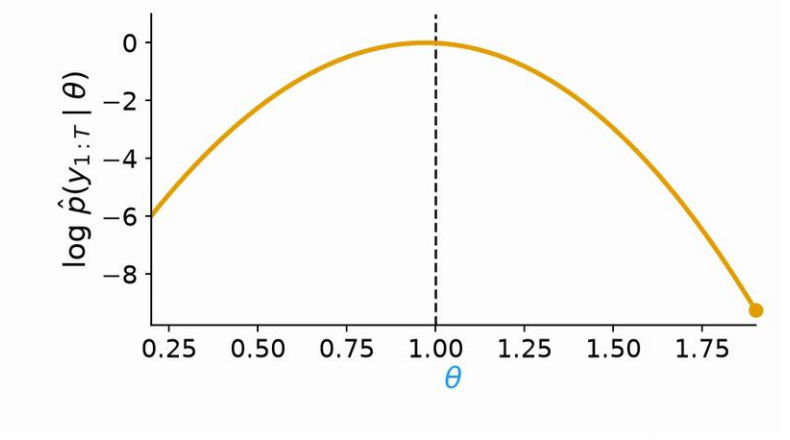
$$p(y_{1:T} \mid \theta) = \int p(x_0) \prod_{k=1}^{T} p(y_k \mid x_k, \theta)\, p(x_k \mid x_{k-1}, \theta)\, \mathrm{d}x_{0:T}$$

← one integral over every state at once

$$= \prod_{k=1}^{T} \int p(y_k \mid x_k, \theta)\, p(x_k \mid y_{1:k-1}, \theta)\, \mathrm{d}x_k$$

← the filter's predict step: one integral per observation

- Every filter run already returns $\log \hat{p}(y_{1:T} \mid \theta)$
- Each value of θ is one filter run
- The sampler now sees only θ: in this toy, 1 unknown instead of 1202



$\hat{p}$: the EnKF's approximation. Dashed: true θ.

The price of a black box: no gradients

Problem	Needs gradients	Works with a black box
State inference	EKF	EnKF, particle filter
Parameter inference	NUTS, SVI	Filter likelihood + Bayesian optimisation, gradient-free MCMC
Control	Differentiating through rollouts	MPPI

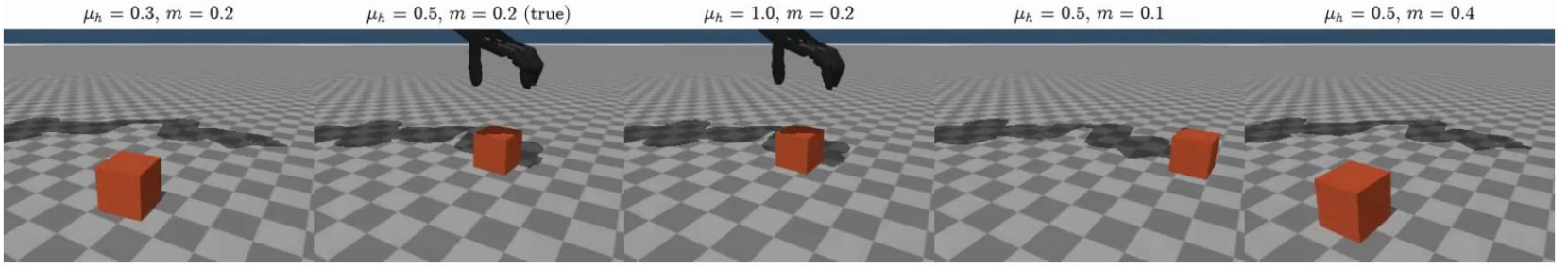
- The interface is unchanged: we swap the method, not the model
- Nothing is lost in the wrapper: on the robot hand, `dynestyx` matches a direct MuJoCo run to 10^{-7} in generalized positions

Robot: simulation

A robot hand lifts a cube

- Panda arm + Shadow Hand in MuJoCo: $x_t \in \mathbb{R}^{71}$, a new observation every 10 ms
- Unknown $\theta = (\mu_h, m)$: hand–cube friction and cube mass
- Replay the recorded torques $u_{1:T}$ at other θ: same torques, different outcome

```
result = dsx.simulate(make_dynamics(mu, mass), rng_key=key,
    predict_times=ts, ctrl_times=ts, ctrl_values=u)
```



Robot: parameter inference

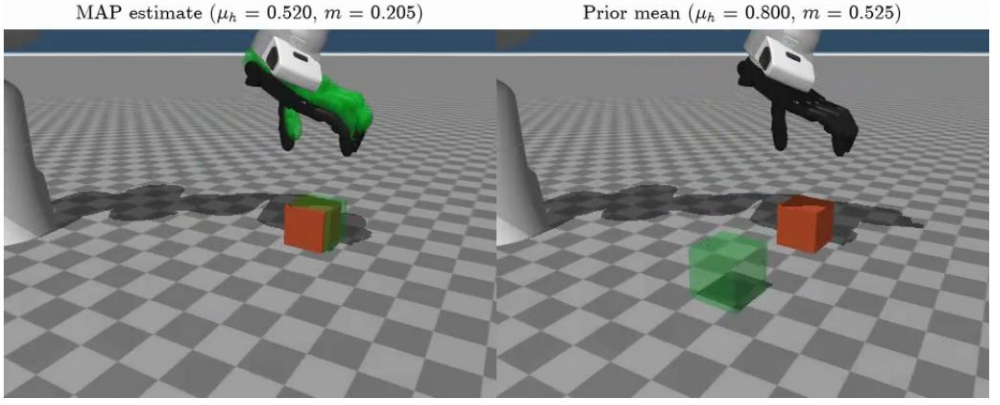
System identification given control inputs

- The same filter estimates the likelihood, with the states integrated out

$$\hat{\theta} = \arg\max_{\theta} \; \log \hat{p}(y_{1:T} \mid \theta, u_{1:T}) + \log p(\theta)$$

- No gradients, so Bayesian optimisation: 60 filter runs, about one minute

	Truth	Prior mean	MAP
μ_h	0.5	0.800	0.520
m [kg]	0.2	0.525	0.205



Truth with green ghosts: rollout at the MAP (left) and at the prior mean (right)

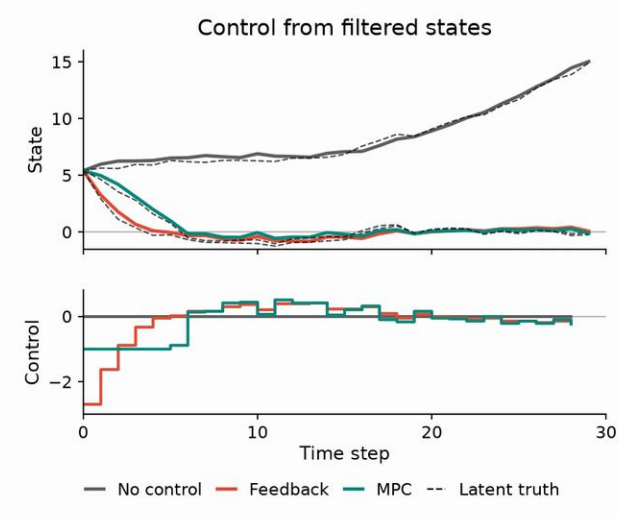
Control with uncertain states

The policy acts on a belief, updated online

$$
\begin{aligned}
u_t, s_t &= \pi(\hat{p}_{t-1}, s_{t-1}) \\
x_t &\sim p(x_t \mid x_{t-1}, u_t, \theta) \\
y_t &\sim p(y_t \mid x_t, u_t, \theta) \\
\hat{p}_t &= \mathrm{FilterUpdate}(\hat{p}_{t-1}, u_t, y_t)
\end{aligned}
$$

- One argument changes: the policy now receives the filter's belief
- Any filter, including EnKF and particle filters on black-box dynamics

```
result = dsx.simulate(dynamics, rng_key=key,
    predict_times=ts, control_policy=policy,
    filter_config=KFConfig(...))
```



Linear example from the docs: stabilizing an unstable system from filtered states

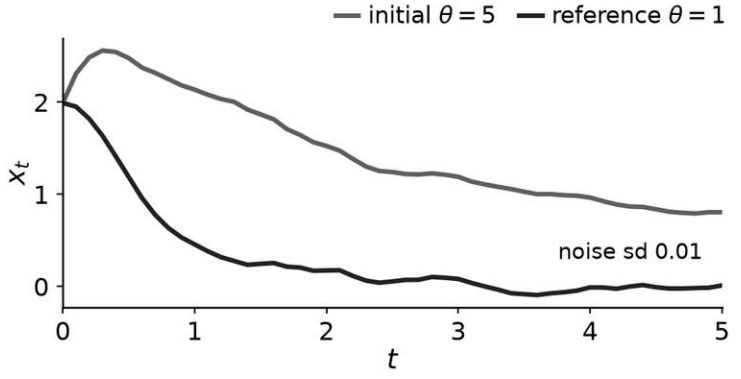
System identification and optimizing the controller

The closed loop is differentiable, so gradients flow through

1. Learning the system

$$\min_{\theta} \sum_{k} \left(x_{t_k}(\theta) - x_{t_k}^{\text{ref}}\right)^2, \quad u_{t_k} = \pi_{\text{MPPI}}(\hat{x}_{t_k}; \theta)$$

```
def residual(theta, key):
    dynamics = make_dynamics(theta)
    mppi = make_mppi(dynamics)  # uses theta
    result = dsx.simulate(dynamics, rng_key=key,
        ..., control_policy=mppi)
    return jnp.sum((result.states - x_ref) ** 2)
grad = jax.grad(residual)(theta, key)
```

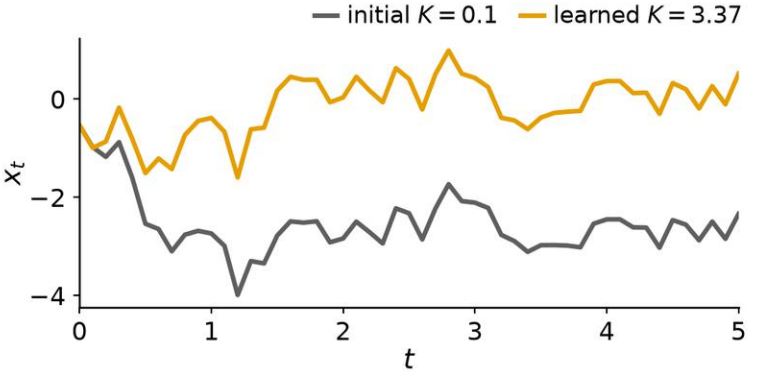


2. Learning the controller

$$\min_{K} \mathbb{E}\left[\frac{1}{N} \sum_{k} x_{t_k}^2 + 0.1\, u_{t_k}^2\right], \quad u_{t_k} = -K\, \hat{x}_{t_k}$$

```
def cost(K, key):
    result = dsx.simulate(dynamics, rng_key=key,
        ..., control_policy=LinearPolicy(K))
    x, u = result.states, result.controls
    return jnp.mean(x**2) + 0.1 * jnp.mean(u**2)
grad = jax.grad(cost)(K, key)
```

We learn the optimal K



Adapted from the `dynestyx` tutorials: control optimization, sections 4 and 5

Open problem: closing the whole loop

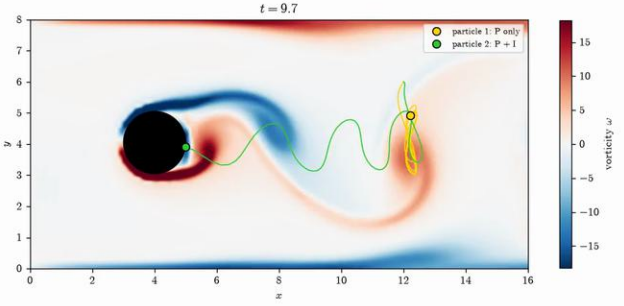
Combination	Status
Online control + state + parameter estimation	**Open**
Control + state estimation, offline system ID	Supported

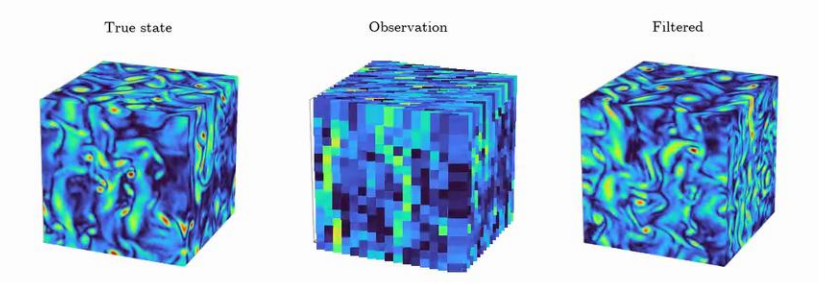
$$\hat{p}_t(x_t, \theta) \approx p(x_t, \theta \mid y_{1:t}, u_{1:t}), \qquad u_t = \pi(\hat{p}_{t-1})$$

- Dual control: each action must make progress *and* reveal θ
- A static θ collapses online filters: ensembles lose spread, particles degenerate
- Contact and nonlinearity make beliefs non-Gaussian and multimodal
- Real-time budgets: expensive solvers, possibly different fidelities for filtering and planning

The same pattern works for PDEs

- **Control** in a vortex wake: PhiFlow through the same callback (128 × 64 grid, Re = 1000)
- **Filtering** 3D Navier–Stokes: 64^3 grid, 16^3 observations, 7-member EnKF





Same model object, same `dsx.simulate` and `Filter`

To approximate or not to approximate

An example with discretization

Variational inference

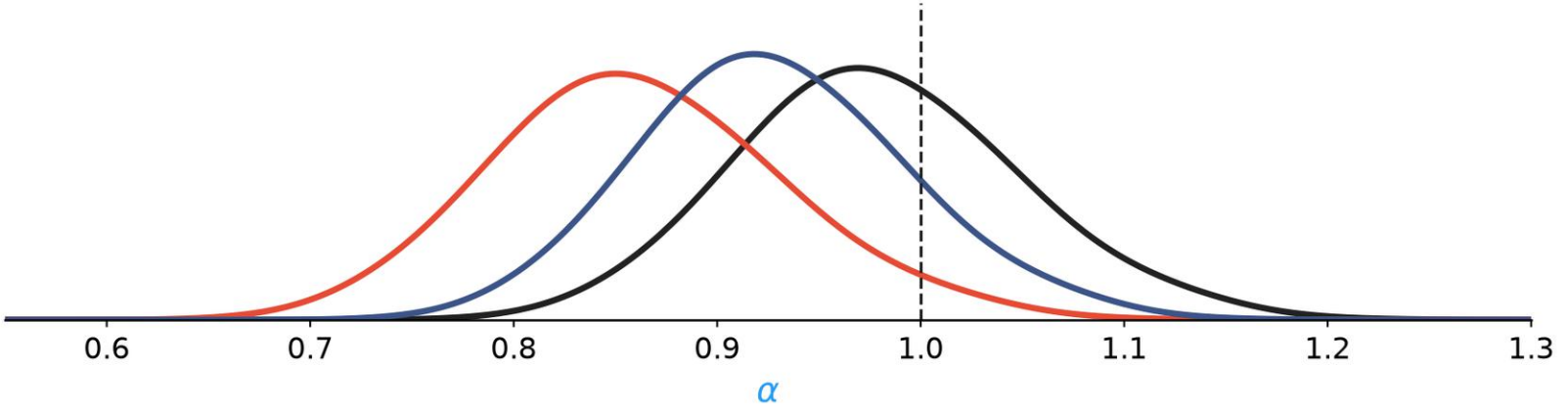
$\theta \sim \text{HalfNormal}(0.5)$

$x_0 \sim \mathcal{N}(0, 1)$

$\mathrm{d}x_t = \theta \sin(x_t)\,\mathrm{d}t + \mathrm{d}W_t$

$y_t \sim \mathcal{N}(x_t, 0.1^2)$

```
with Filter(ContinuousTimeEnKFConfig()):
    guide = AutoNormal(model)
    svi = SVI(model, guide, optax.adam(1e-2),
              loss=Trace_ELBO())
    result = svi.run(key, 300,
                     obs_times, obs_values)
```



Summary

- `dynestyx` is a unified interface to simulation, state inference, parameter inference, and control
- One model description; state-of-the-art methods swapped around it
- The filter is the link: integrating out the states gives $p(\theta \mid y_{1:T})$
- External solvers plug in as a transition, so simulators you already trust can join the loop
- A common interface gives us a place to test the open problem: online control, state, and parameter estimation together

