Experiment Writeup: Flow Matching with AGHF-Constrained Vector Fields in Isaac Lab

Justin Lu
ROAHM Lab, University of Michigan

February 16, 2026

1 Overview

This document serves two purposes: firstly, to actually describe the flow matching framework with respect to robot trajectory generation (with physics constraints), and secondly, to verify that all required mathematical objects are in fact computable from Isaac Lab simulation environments. I will follow the direct notation and formulation of **writeup_v3.pdf** to minimize confusion.

2 Flow Matching Summary (in AGHF Terms)

Let Γ denote the space of trajectories $\gamma : [0,1] \to \mathbb{R}^{2N}$ satisfying the constraints $\gamma(0) = x_0$, $\gamma(1) = x_f$. We generate trajectories by integrating an ODE on Γ:

$$\frac{d\gamma}{ds}(t) = v_\theta(\gamma)(t), \quad s \in [0,1],$$

where s is a *fictitious time parameter* distinct from physical time t, and $v_\theta : \Gamma \to T\Gamma$ is a vector field on trajectory space parameterized by θ, our neural network.

2.1 Target Velocity Field Definition

Given a demonstration trajectory $\gamma_{\text{demo}} \in \Gamma$, a noise trajectory $\gamma_{\text{noise}} \sim \pi(\cdot \mid x_0, x_f)$, and flow time $s \sim \text{Uniform}[0,1]$, we can define the interpolated trajectory as follows:

$$\gamma_s(t) = (1-s)\,\gamma_{\text{demo}}(t) + s\,\gamma_{\text{noise}}(t), \quad \forall\, t \in [0,1].$$

We can then define our target velocity field as a finite difference between trajectories:

$$v^*(t) = \gamma_{\text{demo}}(t) - \gamma_{\text{noise}}(t), \quad \forall\, t \in [0,1].$$

This is the velocity that transports γ_{noise} to γ_{demo} along a straight path in Γ. Note that we are leveraging the fact that we defined $\gamma_s(t)$ as a linear interpolation.

Remark 1. Here, $\pi(\cdot \mid x_0, x_f)$ denotes a distribution over trajectories satisfying the boundary conditions $\gamma(0) = x_0$, $\gamma(1) = x_f$. In layman's terms, it's just the (linear) "starting" trajectory.

2.2 Decomposed Vector Field

Following concepts introduced in **writeup_v3.pdf**, our vector field decomposes as:

$$v_\theta(\gamma) = v_{\text{phys}}(\gamma) + v_{\text{learn}}(\gamma;\, \theta),$$

where v_{phys} is the fixed AGHF vector field and v_{learn} is a learned neural network. We will then augment our flow matching loss to incorporate both vector field definitions as follows:

$$\mathcal{L}(\theta;\, \gamma_{\text{demo}}, \gamma_{\text{noise}}, s) = \left\| v_{\text{phys}}(\gamma_s) + v_{\text{learn}}(\gamma_s;\, \theta) - v^* \right\|^2 \tag{1}$$

where the norm is L^2 over trajectory time: $\|f\|^2 = \int_0^1 \|f(t)\|^2 \, dt$.

2.3 Inference

Sampling requires a single ODE integration from noise to data:

$$\frac{d\gamma}{ds} = -v_\theta(\gamma), \quad s : 1 \to 0,$$

starting from $\gamma_1 \sim \pi(x_0, x_f)$.

3 Single Pendulum System in Isaac Lab

For the single pendulum, the configuration is $q = \theta \in \mathbb{R}$ (the pendulum angle from the downward vertical), and the full state is:

$$x(t) = \begin{bmatrix} q(t) \\ \dot{q}(t) \end{bmatrix} = \begin{bmatrix} \theta(t) \\ \dot{\theta}(t) \end{bmatrix} \in \mathbb{R}^2.$$

So $N = 1$ and the state space is $\mathbb{R}^{2N} = \mathbb{R}^2$. Using the notation from [1]:

- $x^{P_1} = q = \theta$ (position component),
- $x^{P_2} = \dot{q} = \dot{\theta}$ (velocity component).

The single control input $u = \Delta q \in \mathbb{R}$ is a joint position delta relative to the default upright configuration q_{ref}. The commanded position $q_{\text{cmd}} = q_{\text{ref}} + \Delta q$ will be tracked in simulation by a virtual PD controller.

Remark 2. Ultimately, the control input does not greatly affect the functionality of the proposed framework

3.1 Isaac Lab Environment

The environment is adapted from the existing Isaac Lab `CartpoleEnv` and `CartDoublePendulumEnv` examples. The key modifications are:

- Removing the sliding cart prismatic joint so only the revolute joint remains
- Removing the auxiliary pendulum arm
- Applying control solely to the remaining pendulum joint

The simulation runs at $\Delta t = 1/120$ s with control decimation 2, yielding a control period of $\Delta t_{\text{ctrl}} = 1/60$ s. Each episode produces a discrete trajectory:

$$\{x_k\}_{k=0}^{K}, \quad x_k \approx x(t_k), \quad t_k = k\, \Delta t_{\text{ctrl}}.$$

In the case of Isaac lab, these quantities are exposed at each control step:

Quantity	IsaacLab Equivalent
$\theta(t_k)$	`normalize_angle(robot.data.joint_pos[:, pole_dof_idx])`
$\dot{\theta}(t_k)$	`robot.data.joint_vel[:, pole_dof_idx]`

A trajectory for flow matching is the concatenation:

$$\gamma = (x_0, x_1, \ldots, x_K) \in \mathbb{R}^{K\times 2}.$$

4 Physical Action and Its Gradient from Isaac Lab Data

4.1 Formulation of A_{phys}

The physical action from **Definition 3** (**writeup_v3.pdf**) defines the *action functional with only the kinematic consistency term*:

$$A_{\text{phys}}(\gamma) = \int_0^1 k_d \left\| \dot{\gamma}^{P_1}(t) - \gamma^{P_2}(t) \right\|^2 dt.$$

This penalizes trajectories where the recorded velocity states $\gamma^{P_2}(t) = \dot{q}(t)$ are inconsistent with the time derivative of the recorded positions $\dot{\gamma}^{P_1}(t) = \frac{d}{dt}q(t)$.

4.2 Discretizing A_{phys} for Isaac Lab

Discretizing with forward finite differences (so we may pattern-match with IsaacLab), we roughly obtain:

$$A_{\text{phys}}(\gamma) \approx k_d \sum_{k=0}^{K-2} \left\| \frac{q_{k+1} - q_k}{\Delta t_{\text{ctrl}}} - \dot{q}_k \right\|^2 \Delta t_{\text{ctrl}} \tag{2}$$

where $q_k = \gamma^{P_1}(t_k) = \theta_k$ and $\dot{q}_k = \gamma^{P_2}(t_k) = \dot{\theta}_k$ are the position and velocity components of waypoint x_k.

Remark 3. Every term in A_{phys} depends only on q_k, $\dot{q}_k$, and Δt_{ctrl}. These are all provided by Isaac Lab directly. No access to the mass matrix $H(q)$, Coriolis terms $C(q, \dot{q})$, or actuation matrix B is required for computing A_{phys}.

4.3 Note on Backpropagation

At no point are we required to backpropagate through simulator states directly. This is because we formulate the problem with respect to IsaacLab solely as trajectory collection:

- Recording $[q, \dot{q}]$
- Constructing our trajectory, γ_s
- Obtaining v_{phys} via the function call `compute_v_phys(`γ_s`, dt_ctrl, kd)`
- Constructing unified vector field $v_\theta(\gamma) = v_{\text{phys}}(\gamma) + v_{\text{learn}}(\gamma; \theta)$

As a result, autograd solely flows through v_{phys}, and Isaac Lab is allowed to remain a non-differentiable black box.

4.4 AGHF Vector Field

The AGHF vector field from [1, Def. 4] is:

$$v_{\text{phys}}(\gamma) = -G(\gamma)^{-1}\nabla_\gamma A_{\text{phys}}(\gamma).$$

For the initial single-pendulum experiment, we take $G = I$ (identity matrix), so:

$$v_{\text{phys}}(\gamma) = -\nabla_\gamma A_{\text{phys}}(\gamma).$$

For our use-case, V_{phys} and its gradient will be given by the following method:

`v_phys = compute_v_phys(gamma_s, dt_ctrl, kd) # blackbox function, has grad`

5 Training

For each training iteration, we perform the following:

1. Sample γ_{demo} from the dataset, $\gamma_{\text{noise}} \sim \pi$, $s \sim \text{Unif}[0, 1]$.
2. Get interpolated trajectory $\gamma_s = (1 - s)\,\gamma_{\text{demo}} + s\,\gamma_{\text{noise}}$.
3. Compute $v^* = \gamma_{\text{demo}} - \gamma_{\text{noise}}$.
4. Compute $v_{\text{phys}}(\gamma_s) = -\nabla_\gamma A_{\text{phys}}(\gamma_s)$ via `compute_v_phys(`γ_s`, dt_ctrl, kd)`.
5. Forward pass neural net; get $v_{\text{learn}}(\gamma_s, s; \theta)$.
6. Compute loss $\mathcal{L} = \|v_{\text{phys}}(\gamma_s) + v_{\text{learn}}(\gamma_s, s; \theta) - v^*\|^2$.
7. Backprop through v_{learn} to update θ.

5.1 Training Pseudocode (Python)

To put it more concretely, I expect to do the following:

```
# 1. Sample trajectory components
gamma_demo = sample_demo_trajectory(dataset)
gamma_noise = sample_noise_trajectory(x0, xf)
s = torch.rand(1)   # randomize flow time param

# 2. Create our trajectory via lerp
gamma_s = s * gamma_noise + (1 - s) * gamma_demo

# 3. Compute target velocity v* (using finite difference; straight-path transport)
v_star = gamma_demo - gamma_noise

# 4. Get component vector fields
v_phys = compute_v_phys(gamma_s, dt_ctrl, kd)   # treat as black box w/ grad
v_learn = model.forward(gamma_s, s)             # neural net forward pass

# 6. Compute supervised loss = ||(v_phys + v_learn) - v*||^2
residual = (v_phys + v_learn - v_star)
loss = (residual ** 2).sum() * dt_ctrl

# 7. Backprop
optimizer.zero_grad()
loss.backward() # grad will only flow through v_learn (into neural net)
optimizer.step()
```

Remark 4. The gradient quantity $v_{\text{phys}} = -\nabla_\gamma A_{\text{phys}}$ participates in the loss but does not contribute parameters to optimize.

6 Checklist: Required Terms

Object	**Depends on**	**Source**
State $x_k = [\theta_k, \dot{\theta}_k]^\top$	Joint position, velocity	`joint_pos`, `joint_vel`
Demo trajectories γ_{demo}	State at each control step	Rollouts of expert RL policy
Noise trajectories γ_{noise}	Boundary states x_0, x_K	Synthetic (generated)
Interpolated γ_s	$\gamma_{\text{demo}}, \gamma_{\text{noise}}, s$	Vectorized calculation
Target velocity v^*	$\gamma_{\text{demo}}, \gamma_{\text{noise}}$	Pointwise subtraction
Physical action A_{phys}	$q_k, \dot{q}_k, \Delta t_{\text{ctrl}}$	A_{phys}
$\nabla_\gamma A_{\text{phys}}$	A_{phys} as torch graph	`torch.autograd.grad`
$v_{\text{phys}}(\gamma)$	$\nabla_\gamma A_{\text{phys}}, G$	$-G^{-1}\nabla_\gamma A_{\text{phys}}$; $G{=}I$
$v_{\text{learn}}(\gamma; \theta)$	Trajectory tensor γ	Neural net forward pass
Loss $\mathcal{L}$	All of the above	Eq. (1)

Table 1: All quantities required by the method, and a brief overview for how they'll be calculated from Isaac Lab data.