magi_msvgd

Jamie Liu

August 20, 2025

Contents

1 Introduction

This document describes the installation and usage of `magi_msvgd`, a Python package that implements Mitotic Stein Variational Gradient Descent (MSVGD) for Manifold-constrained Gaussian Process Inference (MAGI) [6]. `magi_msvgd` accompanies Jamie Liu's undergraduate thesis, in which MSVGD was developed based on Stein Variational Gradient Descent [2].

For motivations and theory behind the project, see the full thesis, which is included in the package's GitHub repository, https://github.com/jamieliu2/magi_msvgd.

2 Installation

To install the package from GitHub via PIP, enter the following command in the Shell (Linux), Command Line (Windows), or Terminal (Mac). If using the TensorFlow front-end, Linux is strongly recommended for GPU compatibility. Note that the TensorFlow front-end is functional, though it has some minor incompatibilities and is slower than PyTorch. The known incompatibilities are noted throughout the documentation.

```
pip install git+https://github.com/jamieliu2/magi_msvgd.git@main
```

To uninstall, use:

```
pip uninstall magi_msvgd.git
```

There are currently no plans to upload the package to PyPI.

3 Usage

3.1 Preliminary Math and Notation

Before coding, we first define the ODE and its gradients. In our setting, we have a D-dimensional ODE system $\boldsymbol{X}$, governed by a p-dimensional parameter vector $\boldsymbol{\theta}$. The noisy data $\boldsymbol{y}$ are observed at times $\boldsymbol{\tau}$ with observation standard deviation vector $\boldsymbol{\sigma}$. For MAGI computation, we discretize the time set with an index $\boldsymbol{I} \supseteq \boldsymbol{\tau}$, which has length $n = |\boldsymbol{I}|$.

For the example in Section 4, we use the HIV model from Ref. [1]. The three-component system consists of $\boldsymbol{X} = (T_U, T_I, V)$, where T_U, T_I are the concentrations of uninfected and infected cells, respectively, and V is the viral load.

$$\boldsymbol{f}(\boldsymbol{X}, \boldsymbol{\theta}, t) = \begin{pmatrix} \lambda - \rho T_U - \eta(t) T_U V \\ \eta(t) T_U V - \delta T_I \\ N\delta T_I - cV \end{pmatrix} \tag{1}$$

where $\eta(t) = 9 \times 10^{-5} \times (1 - 0.9\cos(\pi t/1000))$ and $\boldsymbol{\theta} = (\lambda, \rho, \delta, N, c)$ are the associated parameters. To facilitate computation, we derive the gradients of $\boldsymbol{f}$ with respect to $\boldsymbol{X}$ and $\boldsymbol{\theta}$.

$$\frac{\partial \boldsymbol{f}(\boldsymbol{X}, \boldsymbol{\theta}, t)}{\partial \boldsymbol{X}} = \begin{pmatrix} -\rho - \eta(t)V & 0 & -\eta(t)T_U \\ \eta(t)V & -\delta & \eta(t)T_U \\ 0 & N\delta & -c \end{pmatrix} \tag{2}$$

$$\frac{\partial \boldsymbol{f}(\boldsymbol{X}, \boldsymbol{\theta}, t)}{\partial \boldsymbol{\theta}} = \begin{pmatrix} 1 & -T_U & 0 & 0 & 0 \\ 0 & 0 & -T_I & 0 & 0 \\ 0 & 0 & NT_I & \delta T_I & -V \end{pmatrix} \tag{3}$$

3.2 Object: MAGISolver

`MAGISolver` is the only object required by the user to interact with the package. The PyTorch version may be imported as follows.

```
from magi_msvgd.torch.magi import MAGISolver
```

The TensorFlow front-end can be imported in the same way from the `magi_msvgd.tf` sub-package.

```
magisolver = MAGISolver(
    ode,
    dfdx,
    dfdtheta,
    data,
    theta_guess,
    theta_conf=0,
    sigmas=None,
    X_guess=1
    mu=None, mu_dot=None,
    pos_X=False, pos_theta=False,
    prior_temperature=None
    bayesian_sigma=True
)
```

3.2.1 Parameters

- **ode** (*function*) – A Python function that encodes the ODE. Takes three arguments {`X` : (n, D), `theta` : $(p,)$, `t` : $(n, 1)$} and returns an $n \times D$ matrix. Must be automatically differentiable.
- **dfdx** (*function*) – A Python function that encodes the gradient of the ODE with respect to $\boldsymbol{X}$. Takes three arguments {`X` : (n, D), `theta` : $(p,)$, `t` : $(n, 1)$} and returns an $n \times D \times D$ array.
- **dfdtheta** (*function*) – A Python function that encodes the gradient of the ODE with respect to $\boldsymbol{\theta}$. Takes three arguments {`X` : (n, D), `theta` : $(p,)$, `t` : $(n, 1)$} and returns an $n \times p \times D$ array.
- **data** (*array or tensor*) – An $n \times (D+1)$ NumPy array or PyTorch / TensorFlow tensor containing the discretized data at time points $\boldsymbol{I}$. Column 0 contains the discretization index $\boldsymbol{I}$. Unobserved values should be filled with null type values.
- **theta_guess** (*array or tensor*) – A flattened $p \times 1$ NumPy array or PyTorch / TensorFlow tensor containing the user guess for $\boldsymbol{\theta}$. This is required for determining p and to provide a starting state for computing the initialization state, but may be chosen arbitrarily if necessary.
- **theta_conf** (optional*: float or array or tensor*) – A flattened $p \times 1$ NumPy array or PyTorch / TensorFlow tensor containing the confidence levels of `theta_guess`. If 0.0 (default), `theta_guess` does not affect the loss function to solve for $\boldsymbol{\theta}$ initialization. Otherwise, the elements of `theta_conf` scales the attraction strength of the corresponding elements of $\boldsymbol{\theta}$'s initializaiton state toward `theta_guess`. If *float*, the attraction strength is the same for all elements of $\boldsymbol{\theta}$.
- **sigmas** (optional*: None or array or tensor*) – A flattened $D \times 1$ NumPy array or PyTorch / TensorFlow tensor containing sampling standard deviations. If `None` (default),

all sampling standard deviations are treated as unknown. Individual entries may be set to null type values if $\boldsymbol{\sigma}$ is partially unknown.

- **X_guess** (optional*: int*) – Number of times to recursively run the initialization procedure of unobserved components in $\boldsymbol{X}$. May yield more stable initializations and escape local modes. (default: 1). **Warning:** setting `X_guess>1` does not currently work and for the TensorFlow front-end and will cause an error, due to TensorFlow's restrictions on creating `tf.Variables`.
- **mu** (optional*: None or array or tensor*) – An $n \times D$ NumPy array or PyTorch / TensorFlow tensor containing the GP prior mean function of each component, evaluated at the times in $\boldsymbol{I}$. If `None` (default), use the flat zero prior $\mu(x) = 0\ \forall x \in \mathbb{R}$ for all components.
- **mu_dot** (optional*: None or array or tensor*) – An $n \times D$ NumPy array or PyTorch / TensorFlow tensor containing the derivative of the GP prior mean function of each component, evaluated at the times in $\boldsymbol{I}$. If `None` (default), use the flat zero prior $\dot{\mu}(x) = 0\ \forall x \in \mathbb{R}$ for all components.
- **pos_X, pos_theta** (optional*: bool*) – Flag if $\boldsymbol{X}$ or $\boldsymbol{\theta}$ is strictly positive. Setting to `True` will sample initial particles from $|q_0|$ and restrict mSVGD to positive particles. (default: `False`). **Warning:** these settings are not currently working for the TensorFlow front-end due to TensorFlow's immutable tensors, but will not cause an error.
- **prior_temperature** (optional*: float*) – The tempering factor $1/\beta$ by which to scale the contribution of the GP prior. If `None` (default), use $1/\beta = N/(Dn)$, the recommended setting by Ref. [6].
- **bayesian_sigma** (optional*: boolean*) – If `True` (default), give Bayesian treatment to unknown sampling standard deviations σ_d, leaning alongside $\boldsymbol{X}$ and $\boldsymbol{\theta}$. Otherwise, hold fixed at initialized value, which is usually a significantly worse estimate of σ_d.

3.2.2 Helpers

Included in our package is some helpful code for pre-processing tasks. Unfortunately, these are written using PyTorch and are thus inaccessible to the TensorFlow front-end. Using the helper code, we can numerically solve systems, construct observation matrices, sample dummy data, and discretize data sets. We can also use automatic differentiation to check that `dfdx` and `dfdtheta` correctly encode the gradients of `ode`. See Section 4.2 for an example of how these functions may be used.

3.3 MAGISolver Method: `initialize_particles`

```
magisolver.initialize_particles(k_0, dtype,
    device=None, init_sd=0.2, random_seed=None)
```

3.3.1 Parameters

- **k_0** (*int*) – The number of initial particles to use for mSVGD. If `k_0` is set to 1, automatically configure MAP estimation; mitosis can still be used normally.
- **dtype** (*type*) – The data type to use for mSVGD computation. This should be a PyTorch dtype if using the PyTorch front-end, and a TensorFlow, NumPy, or Python dtype if using the TensorFlow front-end. Note: 32-bit floats are much faster (depending on the hardware) and very stable, but sometimes 64-bit floats are needed, usually when gradients are too large to represent in 32 bits.

- **device** (optional*: None or device*) – The device to use for mSVGD computation. If `None` (default), allow front-end to choose device. For PyTorch, the default device is typically CPU. TensorFlow manages device usage automatically, regardless of `device` setting.
- **init_sd** (optional*: float*) – The standard deviation for the mSVGD initial Gaussian distribution q_0. Note: $|q_0|$ is used to sample $\boldsymbol{\sigma}$ and if `pos_X` or `pos_theta` is flagged. (default: 0.2).
- **random_seed** (optional*: None or int*) – The random seed to use when sampling particles from the Gaussian initial distribution q_0. The initial states of $\boldsymbol{X}$, $\boldsymbol{\theta}$, and $\boldsymbol{\sigma}$ are sampled separately, with the seed reset to `random_seed` for each. If `None` (default), sample without setting a seed.
- **mitosis** (*not for user*) – This setting allows this function to be used to re-define batched matrices during mitosis splits of the mSVGD particles. Users should leave it set to `False` (default).

3.3.2 Returns

- `initialize_particles` returns `None`.

3.4 MAGISolver Method: `solve`

```
magisolver.solve(optimizer, optimizer_kwargs=dict(),
    max_iter=10_000, mitosis_splits=0, atol=1e-2, rtol=1e-8,
    bandwidth=-1, monitor_convergence=False)
```

3.4.1 Parameters

- **optimizer** (*optimizer class*) – A PyTorch or TensorFlow optimizer object class to use for mSVGD, e.g. from `torch.optim` or `tf.keras.optimizers`. To set a different optimizer for each mitosis split, `optimizer` may be an iterable of optimizers of length $M + 1$.
- **optimizer_kwargs** (optional*: dict*) – A Python dictionary containing keyword arguments to for `optimizer`. `optimizer_kwargs` should not contain the reserved keyword "params." (default: empty `dict()`). To set different optimizer parameters for each mitosis split, `optimizer_kwargs` may be an iterable of dictionaries of length $M + 1$.
- **max_iter** (optional*: int or list*) – The maximum number of mSVGD iterations per mitosis split. (default: $10,000$). To set a different maximum for each mitosis split, `max_iter` may be an iterable of integers of length $M + 1$.
- **mitosis_splits** (optional*: int*) – The number of mitosis splits M to perform for mSVGD, such that the final number of particles $k = k_0 \cdot 2^M$. (default: 0). Gradient descent is effectively run $M + 1$ times.
- **atol** (optional*: float*) – Absolute tolerance parameter for early stopping criterion, defined below. (default: 10^{-2}). To set a different absolute tolerance for each mitosis split, `atol` may be an iterable of floats of length $M + 1$.
- **rtol** (optional*: float*) – Relative tolerance parameter for early stopping criterion, defined below. (default: 10^{-8}). To set a different relative tolerance for each mitosis split, `rtol` may be an iterable of floats of length $M + 1$.

$$\left|\hat{\phi}^*(x_i)_j\right| \leq \texttt{atol} + \texttt{rtol} \times |x_{i,j}| \text{ for all particles } i \text{ in all inferred variables } j \tag{4}$$

- **bandwidth** (optional*: float*) – Bandwidth parameter h for SVGD radial basis function kernel. If non-positive (default: -1), automatically tune bandwidth at each iteration, $h^\ell = \text{med}_\ell^2 / \log k$. To set a different bandwidth for each mitosis split, `bandwidth` may be an iterable of floats of length $M + 1$.
- **monitor_convergence** (optional*: int*) – If falsy (default), do not monitor convergence. Otherwise, `monitor_convergence` specifies how often to print $\max\left(\left|\hat{\phi}^*(x_i)_j\right|\right)$. It also specifies how often to record the current state of the $\boldsymbol{\theta}$ components of the particles, which are returned upon completion. **Warning:** this setting may incur significant memory cost.

3.4.2 Returns

- **Xs** (*tensor*) – A $k \times n \times D$ dimensional tensor containing the resulting state for the $\boldsymbol{X}$ components of the particles. The 0th axis indexes over the particles, the 1st axis indexes over the time steps in $\boldsymbol{I}$, the 2nd axis indexes over the ODE components.
- **thetas** (*tensor*) – A $k \times p$ dimensional tensor containing the resulting state for the $\boldsymbol{\theta}$ components of the particles.
- **sigmas** (*tensor*) – A $k \times |\boldsymbol{\sigma}_{\text{unknown}}|$ dimensional tensor containing the resulting state for the $\boldsymbol{\sigma}$ components of the particles. Note that the ith column corresponds to the ith unknown σ_d, ordered by d, which may not correspond to the ith ODE system component. If all sampling standard deviations are known, `solve` still returns an empty tensor for `sigmas`.
- optional**: trajectories** (*list[tensor]*) – This is returned if and only if `solve` is configured to monitor convergence. A list of $k \times p$ tensors that represent the state of the $\boldsymbol{\theta}$ components of the particles, at iteration intervals encoded by the `monitor_convergence` parameter. If `monitor_convergence` is falsy, only the first three items (`Xs`, `thetas`, `sigmas`) are returned.

3.4.3 Saving Results

Note that we can extract mSVGD results without having to re-run `solve`, since `MAGISolver` saves the state of the particles in the `magisolver.particles` attribute.

```
X_result, theta_result, sigma_result =
↪  magisolver.from_svgd_vector(magisolver.particles)
```

If we want continue to run mSVGD on partially-transformed particles, we can simply re-call the `solve`. We can also force a split at the current state by running `solve` with `max_iter=1` and `mitosis_splits=1`.

At any point, we can save the `MAGISolver` object by writing its byte stream to a file using the `dill` library, which extends PYTHON's `pickle` library [3].

```
import dill
with open('magisolver.obj', 'wb') as file:
    dill.dump(magisolver, file)
```

Then, in some later session, after importing the necessary libraries, we only need to load the byte stream to have access to the original `magisolver` object. This allows us to avoid re-running the initialization procedure and to save computed results.

```
with open('magisolver.obj', 'rb') as file:
    magisolver = dill.load(file)
```

4 Example

4.1 Preliminary Math and ODE Setup

In a notebook file, we first import NumPy and PyTorch for working with data objects and defining the ODE and its gradients. Then, we import the `MAGISolver` object.

```
import numpy as np
import torch
from magi_msvgd.torch.magi import MAGISolver
```

As a running example, recall the HIV model in Section 3.1, from Ref. [1]. The three-component system consists of $\boldsymbol{X} = (T_U, T_I, V)$, where T_U, T_I are the concentrations of uninfected and infected cells, respectively, and V is the viral load.

$$\boldsymbol{f}(\boldsymbol{X}, \boldsymbol{\theta}, t) = \begin{pmatrix} \lambda - \rho T_U - \eta(t) T_U V \\ \eta(t) T_U V - \delta T_I \\ N\delta T_I - cV \end{pmatrix} \tag{5}$$

where $\eta(t) = 9 \times 10^{-5} \times (1 - 0.9\cos(\pi t/1000))$ and $\boldsymbol{\theta} = (\lambda, \rho, \delta, N, c)$ are the associated parameters. We also derive the gradients of $\boldsymbol{f}$ with respect to $\boldsymbol{X}$ and $\boldsymbol{\theta}$.

$$\frac{\partial \boldsymbol{f}(\boldsymbol{X}, \boldsymbol{\theta}, t)}{\partial \boldsymbol{X}} = \begin{pmatrix} -\rho - \eta(t)V & 0 & -\eta(t)T_U \\ \eta(t)V & -\delta & \eta(t)T_U \\ 0 & N\delta & -c \end{pmatrix} \tag{6}$$

$$\frac{\partial \boldsymbol{f}(\boldsymbol{X}, \boldsymbol{\theta}, t)}{\partial \boldsymbol{\theta}} = \begin{pmatrix} 1 & -T_U & 0 & 0 & 0 \\ 0 & 0 & -T_I & 0 & 0 \\ 0 & 0 & NT_I & \delta T_I & -V \end{pmatrix} \tag{7}$$

When encoding the ODE $\boldsymbol{f}$ in Python, we must write a function that can be automatically differentiated. This is necessary for the procedure used to compute initialization states. The functions encoding $\partial \boldsymbol{f}/\partial \boldsymbol{X}$ and $\partial \boldsymbol{f}/\partial \boldsymbol{\theta}$ need not be automatically differentiable.

The input `X` is a matrix with $n = |\boldsymbol{I}|$ rows and D columns. The input `theta` is a row vector with p components. The input `t` is a column vector with n components. The return value is an $n \times D$ matrix describing the ODE in all of the components at each time in $\boldsymbol{I}$.

```
def ode(X, theta, t=None):
    n = X.shape[0]
    T_U, T_I, V = X.T
    lam, rho, delta, N, c = theta.repeat([n, 1]).T
    t = t.flatten()
    eta = 9e-5 * (1 - 0.9*torch.cos(torch.pi*t/1000))

    return torch.stack([lam - rho*T_U - eta*T_U*V,
                        eta*T_U*V - delta*T_I,
                       N*delta*T_I - c*V], axis=1)
```

Next, we encode the gradient of the ODE with respect to $\boldsymbol{X}$. This function takes the same inputs as `ode` and outputs an $n \times D \times D$ array, where the array slice `[:,i,j]` is the partial derivative of the jth system component with respect to the ith system component.

```
def dfdx(X, theta, t=None):
    n = X.shape[0]
    T_U, T_I, V = X.T
```

```
    lam, rho, delta, N, c = theta.repeat([n, 1]).T
    t = t.flatten()
    eta = 9e-5 * (1 - 0.9*torch.cos(torch.pi*t/1000))

    zero = torch.zeros(n, device=theta.device, dtype=theta.dtype)

    return torch.stack([torch.stack([-rho - eta*V, zero, -eta*T_U], axis=1),
                        torch.stack([eta*V, -delta, eta*T_U], axis=1),
                        torch.stack([zero, N*delta, -c], axis=1)], axis=2)
```

Finally, the encoding of the gradient of the ODE with respect to $\boldsymbol{\theta}$ takes the same arguments as the other two and returns an $n \times p \times D$ array, where the array slice `[:,i,j]` is the partial derivative of the jth system component with respect to the ith parameter in $\boldsymbol{\theta}$.

```
def dfdtheta(X, theta, t=None):
    n = X.shape[0]
    T_U, T_I, V = X.T
    lam, rho, delta, N, c = theta.repeat([n, 1]).T
    t = t.flatten()
    eta = 9e-5 * (1 - 0.9*torch.cos(torch.pi*t/1000))

    zero = torch.zeros(n, device=theta.device, dtype=theta.dtype)
    one = torch.ones(n, device=theta.device, dtype=theta.dtype)

    return torch.stack([torch.stack([one, -T_U, zero, zero, zero], axis=1),
                        torch.stack([zero, zero, -T_I, zero, zero], axis=1),
                       torch.stack([zero, zero, N*T_I, delta*T_I, -V], axis=1)],
                       ↪  axis=2)
```

Remark. The encodings and dimensionalities of all of these functions are consistent with those of the existing MAGI package [5].

4.2 Helpers and Pre-processing

Included in our package is some helpful code for pre-processing tasks. Unfortunately, these are written using PyTorch and are thus inaccessible to the TensorFlow front-end. Using the helper code, we can numerically solve systems, construct observation matrices, sample dummy data, and discretize data sets. We can also use automatic differentiation to check that `dfdx` and `dfdtheta` correctly encode the gradients of `ode`.

```
from magi_msvgd.tests import test_helpers
# n = arbitrary data size, D = dimensions in X, p = dimensions in theta
test_helpers.check_gradients(ode, dfdx, dfdtheta, n=201, D=3, p=5, trials=100)
```

For each of `trials` loops, this will randomly generate an $n \times D$ matrix as a test $\boldsymbol{X}$, a p-dimensional vector as a test $\boldsymbol{\theta}$, and a scalar as a test time t. Then, at these values, it will compare the gradients given by automatic differentiation of `ode` with the gradients given by `dfdx` and `dfdtheta`. It will return a tuple `(x_score, theta_score)` containing the proportion of trials that were correct.

Remark. Correctly encoded gradients may not return perfect 1.0 scores due to floating point errors. Tolerance for "closeness" may be adjusted, but generally if the score is greater than 0.75, the encoding is probably correct.

Now, we need to generate a dataset. For this example, we set the ground truth values $\boldsymbol{\theta} = (36, 0.108, 0.5, 1000, 3)$ and initial conditions $\boldsymbol{x}(0) = (600, 30, 100000)$. We sample data with sampling standard deviations $\boldsymbol{\sigma} = (\sqrt{10}, \sqrt{10}, 10)$, which are unknown. All three components

are observed at 0.2 minute increments from time 0 to time $T = 20$ minutes.

Our package provides code for generating test cases, which we will demonstrate here. Users can also use the `ODEmodel` object demonstrated below by creating a PYTHON file that defines a model of interest, following the template of the example models in the `tests/models` folder. This example uses the HIV model included in `tests/models/hiv.py`. Note that the `test_helpers` functions can be used without defining the whole model, and only the gradient-checking function requires PYTORCH.

We set the random NUMPY seed 2025 for reproducibility. Note that the argument `obs_times` is a matrix that encodes the times at which to sample observations, where the 0th column contains the observation times, and the rest of the columns contain truthy/falsy values that indicate whether the corresponding component was observed at that time.

```
from magi_msvgd.tests.make_test import ODEmodel
from magi_msvgd.tests.models import hiv
# load HIV model
HIV = ODEmodel(hiv)
# solve ODE via numerical integration
HIV.get_ode_solution(X0=np.array([600, 30, 100000]), T=20, step=1e-4)

# define observation times (0, 0.2, 0.4, ..., 20)
tau = [np.linspace(0, 20, 101), np.linspace(0, 20, 101)]
# define observation matrix, col 0=times, rows in other cols: 1 if observed, 0
↪  otherwise
obs_times = test_helpers.obs_matrix(tau)

# generate data
data = HIV.generate_sample(obs_times=obs_times, sigma=np.array([10**0.5,
↪  10**0.5, 10]), random_seed=2025)
```

Let's inspect the ground truth solution (orange lines) and the sample data (red points).

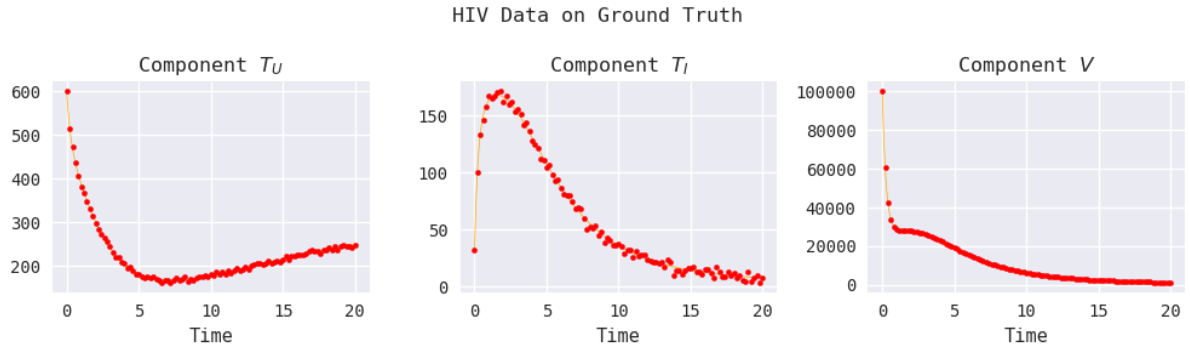

Figure 1: HIV model ground truth trajectories and dataset with NUMPY seed 2025.

The last preparation step is to define the discretization set $\boldsymbol{I}$ and to construct the discretized data matrix, which is done manually by the user. This is the means by which the `MAGISolver` object interacts with the data. The discretized data matrix may be a NUMPY array or a PYTORCH/TENSORFLOW tensor, and should have dimensionality $n \times (D+1)$. The 0th column should contain the times in the discretization set $\boldsymbol{I}$, and the remaining columns should be filled with the data of the corresponding components. The unobserved points, whether due to discretization or being unobserved in the original dataset, should be filled with null types, e.g. `None` or `np.nan` or `torch.nan`.

Remark. Entries containing non-null falsy values, e.g. `False`, will be treated as observed zeroes, not as unobserved.

For this example, we use 2× discretization, adding 1 equally spaced discretization time between every pair of observed data points. So, we set $\boldsymbol{I} = (0, 0.1, 0.2, \ldots, 20)$. This gives us $n = |\boldsymbol{I}| = 201$ discretization time points. We also provide code for discretizing data.

```
# define discretization set
I = np.linspace(0, 20, 201)
data_disc = test_helpers.discretize_data(data, I)
```

4.3 Object: MAGISolver

Now we have all we need to use the `MAGISolver` object to conduct mSVGD for MAGI. Instantiating this object stores all of the supplied variables, computes initialization states for $(\boldsymbol{X}, \boldsymbol{\theta}, \boldsymbol{\sigma})$, computes the Matern kernel parameters $\boldsymbol{\phi}$, and pre-computes the GP matrices C_d^{-1}, m_d, K_d^{-1} for each component d. Behind the scenes, it also configures the library-specific polymorphic representations, depending on the front-end being used. The existing MAGI package uses spares matrices for the GP matrices [4], which we do not.

```
magisolver = MAGISolver(
    ode=ode,
    dfdx=dfdx,
    dfdtheta=dfdtheta,
    data=data_disc,
    theta_guess=np.array([50., 0., 0., 1000., 5.]),
    theta_conf=np.array([0., 0., 0., 0., 0.]),
    sigmas=None,
    X_guess=1
    mu=None, mu_dot=None,
    pos_X=False, pos_theta=False,
    prior_temperature=None
    bayesian_sigma=True
)
```

Here, we can manually adjust the initialization states, if necessary. In this example, we correct the bad σ_V initialization, as recommended by Ref. [5], which will help stabilize the learning algorithm.

```
magisolver.sigmas[2] = 10
```

We can edit the initialization states of $\boldsymbol{X}, \boldsymbol{\theta}, \boldsymbol{\sigma}$ in the attributes `x_init`, `theta_init`, and `sigmas`, respectively. We can also edit the optimized GP prior hyperparameters $\boldsymbol{\phi}$ via the attribute `phis`, which is a $D \times 2$ array. However, if we change $\boldsymbol{\phi}$, we have to re-compute the GP matrices C_d^{-1}, m_d, K_d^{-1} with the `build_matrices` helper function.

```
from magi_msvgd._helpers import build_matrices
build_matrices(magisolver)
```

4.4 MAGISolver Method: `initialize_particles`

Now, we use the `initialize_particles` method to prepare for mSVGD. This function defines some variables necessary for the mSVGD algorithm and stores batched matrices to facilitate computation. It also samples the initial set of k_0 particles from q_0, which is a Gaussian distribution with means at the previously solved initialization states and a user-defined initial standard

deviation. In this example, we use $k_0 = 400$.

```
magisolver.initialize_particles(k_0=400, dtype=torch.float64, device='cuda',
↪  init_sd=1, random_seed=2025)
```

4.5 MAGISolver Method: `solve`

Finally, we run mSVGD using the `solve` method.

```
# define optimizer object class and parameters
optimizer = torch.optim.Adam
optimizer_kwargs = {'lr':1e-1}

# run solver
X_result, theta_result, sigma_result, trajectories =
↪  magisolver.solve(optimizer=optimizer, optimizer_kwargs=optimizer_kwargs,
                max_iter=500, mitosis_splits=1, atol=0.85, rtol=0,
                ↪  monitor_convergence=20)
```

4.6 Results

Below, we plot the evolution the distribution of $\boldsymbol{\theta}$ as mSVGD progressed, which is contained by the information returned by `solve` in `trajectories`. We can also inspect the final learned distributions of $\boldsymbol{\theta}$, which is encoded in the returned variable `thetas`

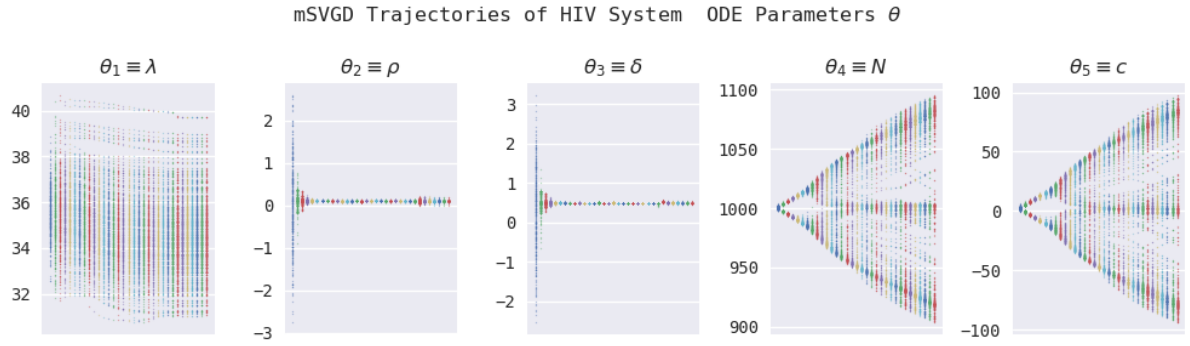

Figure 2: HIV model system parameters $\boldsymbol{\theta}$ evolving during mSVGD for dataset with NumPy seed 2025.

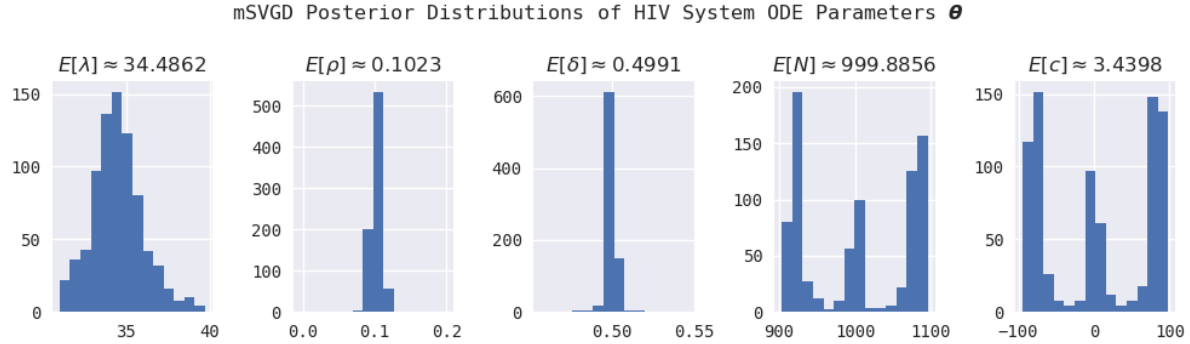

Figure 3: Posterior distribution of HIV model system parameters $\boldsymbol{\theta}$ learned from mSVGD for dataset with NumPy seed 2025. Ground truth: $\boldsymbol{\theta} = (36, 0.108, 0.5, 1000, 3)$.

Next, we plot the learned distributions of the trajectories $\boldsymbol{X}$, which we visualize using the inferred posterior mean and 95% credible interval, plotted on top of the true trajectory.

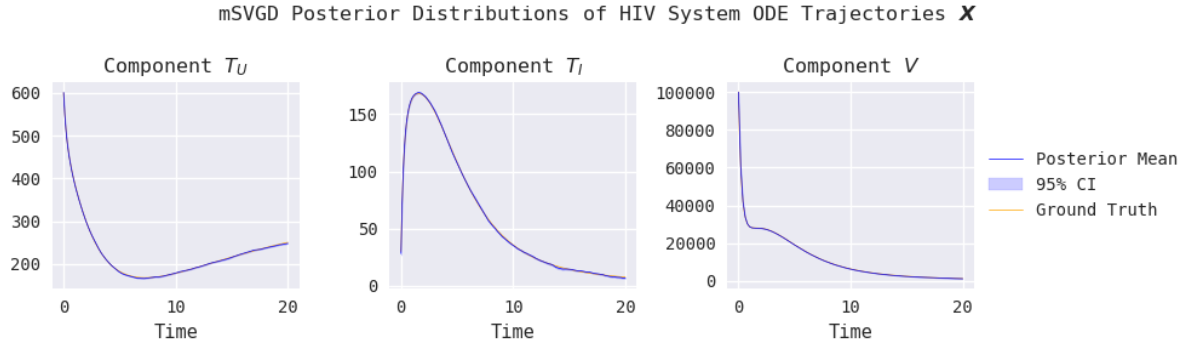

Figure 4: Posterior distribution of HIV model system trajectories $\boldsymbol{X}$ learned from mSVGD for dataset with NumPy seed 2025.

Finally, similar to $\boldsymbol{\theta}$, we also plot a histogram depicting the learned distributions of any unknown $\boldsymbol{\sigma}$, which were relatively poorly estimated in this example.

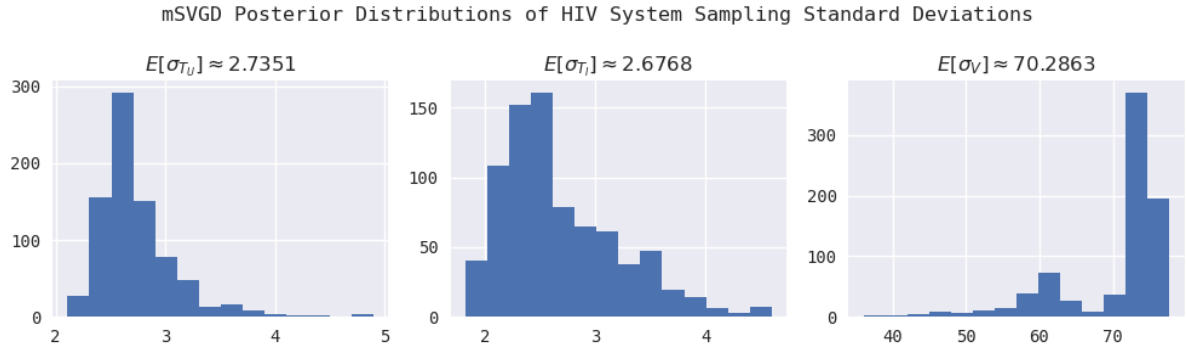

Figure 5: Posterior distribution of HIV model sampling standard deviations $\boldsymbol{\sigma}$ learned from mSVGD for dataset with NumPy seed 2025. Ground truth: $\boldsymbol{\sigma} \approx (3.162, 3.162, 10)$.

Recall that we set the ground truth values $\boldsymbol{\theta} = (36, 0.108, 0.5, 1000, 3)$. mSVGD learned posterior mean estimates $\hat{\boldsymbol{\theta}} \approx (34.486, 0.102, 0.499, 999.886, 3.440)$, accurately inferring the ODE parameters. Notice also that the plotted inferred trajectories (blue) almost exactly cover the ground truth trajectories (orange), showing that mSVGD also accurately recovered the system trajectories. In this example, the learned distributions of the parameters N and c were somewhat strange, and the standard deviations $\boldsymbol{\sigma}$ were relatively poorly inferred due to the scale of the V component, but such inaccuracies are not universal phenomena.

The initialization procedure when instantiating the `MAGISolver` object took about 20 seconds — longer than most models — the particle preparation in `initialize_particles` took about 0.01 seconds, and the mSVGD run in `solve` took about 8 seconds, giving a total runtime of less than 30 seconds. More rigorous and detailed performance results are discussed in the full thesis.

References

[1] Hua Liang, Hongyu Miao, and Hulin Wu. "Estimation of constant and time-varying dynamic parameters of HIV infection in a nonlinear differential equation model". In: *The Annals of Applied Statistics* 4.1 (Mar. 2010). DOI: 10.1214/09-aoas290.

[2] Qiang Liu and Dilin Wang. *Stein Variational Gradient Descent: A General Purpose Bayesian Inference Algorithm*. 2016. arXiv: 1608.04471v1 [stat.ML]. URL: https://arxiv.org/abs/1608.04471v1.

[3] uqfoundation. *dill*. https://github.com/uqfoundation/dill. 2025. URL: https://github.com/uqfoundation/dill.

[4] Samuel W. K. Wong. *MAnifold-constrained Gaussian process Inference (MAGI)*. https://github.com/wongswk/magi. 2024. URL: https://github.com/wongswk/magi.

[5] Samuel W. K. Wong, Shihao Yang, and S.C. Kou. "magi: A Package for Inference of Dynamic Systems from Noisy and Sparse Data via Manifold-constrained Gaussian Processes". In: *Journal of Statistical Software* 109.4 (2024), pp. 1–47. DOI: 10.18637/jss.v109.i04.

[6] Shihao Yang, Samuel W. K. Wong, and S. C. Kou. "Inference of dynamic systems from noisy and sparse data via manifold-constrained Gaussian processes". In: *Proceedings of the National Academy of Sciences* 118.15 (2021), e2020397118. DOI: 10.1073/pnas.2020397118. eprint: https://www.pnas.org/doi/pdf/10.1073/pnas.2020397118. URL: https://www.pnas.org/doi/abs/10.1073/pnas.2020397118.