

BayesianEstimationTool Manual

BayesianEstimationTool (BET) helps you learn from a group of related individuals and make predictions for someone new. An **individual** (also called an *entry* in some API descriptions) might be a patient, an experiment, or a simulated system. This manual uses individual for all three; the examples happen to use people. You provide a model of how the system works and the measurements you have collected; BET estimates its unknown parameters and helps you test its predictions.

Imagine measuring how quickly tumors grow in several patients. The same growth equation might describe every patient, but each patient's growth rate can differ. If you estimate each rate separately, a patient with only one or two measurements gives you little information. BET instead learns from the whole group: what growth rates are typical, how much they vary, and what the measurements tell us about each patient.

You can predict a new patient's trajectory *before seeing any measurements*, using what BET learned from the group. *After seeing a few measurements*, you can update that prediction using information from the patient. BET also helps you measure whether those predictions were accurate and how uncertain they are.

This approach is called **hierarchical Bayesian inference**. You do not need to master the terminology to use the basic workflow. Throughout the manual, **population** means the group whose shared patterns BET learns; it does not require a medical or demographic population.

Installation

BET requires Python 3.10 or newer. Install from [PyPI](#):

```
python -m pip install BayesianEstimationTool
```

Alternatively, install the latest source directly from GitHub:

```
python -m pip install "git+https://github.com/rafaelrbravo/BayesianEstimationTool.git"
```

You can also download a wheel (`.whl`) or source archive (`.tar.gz`) from [GitHub Releases](#) and install it with `python -m pip install PATH_TO_DOWNLOADED_FILE` .

Verify installation:

```
python -c "import BayesianEstimationTool as bet; print(bet.Estimator)"
```

For editable development from a source checkout, run `python -m pip install -e .` in the repository root. The conda recipe in `conda-recipe/meta.yaml` is for local builds; BET is not yet published on conda-forge.

Create a local examples and documentation folder

After installing a BET build that includes bundled resources, run:

```
python -m BayesianEstimationTool.ExampleInstaller
```

This creates a `BET-Examples/` folder in your current directory containing the six runnable Python examples and a `Documentation/` subfolder with Markdown and PDF versions of the manual and cheatsheet. For example:

```
BET-Examples/  
  1_SimpleExampleNuts.py  
  2_SimpleExampleSvi.py  
  3_SaveExampleNuts.py  
  4_LoadExampleNuts.py  
  5_SaveExampleSvi.py  
  6_LoadExampleSvi.py  
Documentation/  
  Manual.md  
  Manual.pdf  
  Cheatsheet.md  
  Cheatsheet.pdf
```

To create it in a different parent directory, provide that directory as an argument:

```
python -m BayesianEstimationTool.ExampleInstaller /path/to/parent
```

The installer refuses to overwrite an existing `BET-Examples/` folder. Run an example from the parent directory with `python BET-Examples/1_SimpleExampleNuts.py`; run each Save example before its corresponding Load example. If the installer module is missing, install a BET build that includes the bundled resources or use the latest GitHub source installation above.

The workflow at a glance

1. **Organize observations.** Put each individual's measurements and inputs into arrays.
2. **Describe the system.** Write a function that predicts what happens when its parameters are known.
3. **Describe what is unknown.** Specify which parameters are shared, which vary by individual, and how measurements differ from ideal predictions.
4. **Learn from the group.** Use `Train` to estimate the population and individual parameters.
5. **Predict for new individuals.** Use `Test` with or without their available measurements.
6. **Check the predictions.** Use scoring functions to compare predictions with observations held back from fitting.

Core components

Component	What it does
<code>data</code>	Holds measurements and inputs for each individual
<code>ModelFn</code>	Predicts what happens for one individual
<code>GlobalPriorFn</code>	Describes plausible values for unknown shared parameters before fitting
<code>localZNames</code>	Names the ways individuals can differ from one another

Component	What it does
<code>TrainLikelihoodFn</code>	Describes how training observations relate to model predictions
<code>TestLikelihoodFn</code>	Describes how available observations from new individuals update their local latent z-scores
<code>Method</code> (<code>Nuts</code> / <code>Svi</code>)	Chooses how BET estimates unknown parameters from observations
<code>Estimator</code>	Connects your model, data, and inference results
<code>Train</code> / <code>Test</code>	Learns from the training group / predicts for new individuals
<code>ScoreFn</code>	Measures how well a prediction matches observations

Part 1: Core concepts

What BET learns

A model is a rule for making predictions. For example, a growth equation uses a growth rate and a time to predict a size. BET does not choose that equation for you; you write it in `ModelFn`. BET helps determine which parameter values are consistent with your measurements.

Some parameters describe the group; others describe individuals. A population might have a typical growth rate, while each patient's tumor grows somewhat faster or slower. BET learns about both the shared pattern and the individual differences. In the API, shared unknown parameters are called **global parameters**. Individual differences are represented by **local latent z-scores** (`localZs`): standardized, unobserved values that describe how each individual differs from the group. Your model combines these z-scores with global parameters to obtain meaningful individual parameters, such as each patient's tumor growth rate.

Measurements are imperfect. Even a good model will not match every observation exactly. The **likelihood** describes how observations can differ from model predictions, including measurement noise. A **prior** describes plausible parameter values before those observations are used. BET combines the prior information with the measurements to estimate what values are plausible afterward; these updated estimates form the **posterior**.

Learning and predicting are different steps. During `Train`, BET uses the selected individuals to learn the global parameters and each training individual's local latent z-scores. During `Test`, it predicts for individuals not included in training. Without their measurements, BET draws possible local latent z-scores from the learned population. With some measurements and a test likelihood, BET updates those z-scores for each new individual. The current `Test` workflow requires at least one named local z-score (`localZNames`); a model with only global parameters can be trained but cannot use `Test` directly. To assess prediction quality fairly, keep the observations you want to predict out of the fitting step.

BET tracks uncertainty rather than returning only one answer. Different plausible parameter values can produce different predictions. Samples from the fitted distribution let you examine that variation. Predictions of the underlying model are not automatically noisy future measurements; the observation model describes that additional variation.

Designing a BET model

To implement this workflow:

1. Prepare `data` as a dictionary of arrays. Every array's first axis identifies the individual; later axes can represent times, features, or positions.

2. Write `ModelFn(globalParams, localZs, data)` to predict outputs for *one individual*. Return a dictionary of named outputs. Keep this function deterministic: do not sample random variables inside it.
3. If shared parameters are unknown, define `GlobalPriorFn(dataShapes)` to give their prior distributions. Otherwise, omit it.
4. Use `localZNames` for the individual differences you want BET to estimate. Inside `ModelFn`, transform these standardized effects into meaningful parameters such as growth rates.
5. Define `TrainLikelihoodFn` to connect observations to predictions using distributions. If measurements are missing, mask them explicitly.
6. Choose training individuals and an inference method, `Nuts(...)` or `Svi(...)`. Call `Train` and check whether the fitted model describes the data reasonably.
7. Choose held-out individuals. Call `Test` for predictions from the population alone, or provide a test likelihood and inference method to update local z-scores using measurements available at prediction time.
8. Define a scoring function and call `ScoreTrainSamples` or `ScoreTestSamples` to evaluate predictions. Keep track of which dimensions represent samples and which represent individuals.

Automatic vectorization

Write `ModelFn`, `TrainLikelihoodFn`, `TestLikelihoodFn`, and `ScoreFn` for one individual at a time. BET applies them across individuals and parameter samples using JAX vectorization, so you do not need to write those loops yourself. These functions should use JAX-compatible operations and avoid side effects. Vectorization enables efficient batched computation, although actual parallel execution depends on the operations and hardware.

Part 2 combines these steps in an analytically solvable tumor growth example. Later sections explain correlated local z-scores and inference-method configuration.

Part 2: Worked example

Predicting tumor growth for new patients

This worked example follows the runnable files [SimpleExampleNuts](#) and [SimpleExampleSvi](#). Both use the **same model and data**, changing only the inference method. We simulate eight tumor-size measurements for each of ten patients. Patients 0–4 are used for population training; patients 5–9 are held out. For the test patients, the first three observations may personalize predictions, and the final five are reserved for scoring.

Generate the synthetic data

```
import numpyro as npo
npo.set_host_device_count(4)

import jax.numpy as jnp
import numpyro.distributions as dist
from jax import random
import BayesianEstimationTool as bet

keyG, keyNoise=random.split(random.PRNGKey(2), 2)
nPatients=10
nTimes=8
times=jnp.linspace(0, 10, nTimes)
trueG=0.15+0.03*random.normal(keyG, (nPatients,))
v0=jnp.linspace(0.8, 1.2, nPatients)
trueTumorSize=v0[:, None]*jnp.exp(trueG[:, None]*times)
tumorSize=trueTumorSize+0.05*random.normal(keyNoise, trueTumorSize.shape)
```

```
data={
    "times":jnp.tile(times,(nPatients,1)),
    "v0":v0,
    "tumorSize":tumorSize,
}
```

The synthetic growth rates have mean 0.15 and standard deviation 0.03; the simulated measurement noise has standard deviation 0.05. These are **data-generating values**, not values passed to BET during inference. Each patient has a known initial size `v0`.

Define the model, priors, and likelihoods

Tumor size follows the analytical solution $v(t) = v_0 * \exp(g*t)$. Each patient's growth rate is the population mean plus a scaled standardized local effect.

```
def ModelFn(globalParams, localZs, data):
    g=globalParams["gMean"]+globalParams["gStd"]*localZs["g"]
    return {"tumorSize":data["v0"]*jnp.exp(g*data["times"])}

def GlobalPriorFn(dataShapes):
    return {
        "gMean":dist.Normal(0.15,0.1),
        "gStd":dist.HalfNormal(0.1),
        "sigma":dist.HalfNormal(0.1),
    }

def TrainLikelihoodFn(globalParams, localZs, data,modelOut):
    return {"tumorSize":(data["tumorSize"],dist.Normal(modelOut["tumorSize"],globalParams["sigma"])))}

def TestLikelihoodFn(globalParams, localZs, data,modelOut):
    nFit=3
    tumorSize=modelOut["tumorSize"]
    mask=jnp.arange(tumorSize.shape[-1])<nFit
    return {"tumorSize":(data["tumorSize"],dist.Normal(tumorSize,globalParams["sigma"]).mask(mask))}

def ScoreFn(modelOut,data):
    return jnp.mean(jnp.abs(modelOut["tumorSize"][3:]-data["tumorSize"][3:]))
```

`ModelFn` and the likelihood functions operate on **one patient at a time**; BET handles vectorization. `TrainLikelihoodFn` uses all training measurements. `TestLikelihoodFn` masks the final five measurements so only the first three test measurements affect personalized inference. `ScoreFn` evaluates those five held-out measurements; it is not used for fitting.

Train the population

```
est=bet.Estimator(data=data,ModelFn=ModelFn,localZNames=["g"],GlobalPriorFn=GlobalPriorFn,rngSeed=0)
est.Train(trainDataIdxs=jnp.arange(5),TrainLikelihoodFn=TrainLikelihoodFn,Method=bet.Svi(numSteps=10000,num
Samples=2000,stepSize=0.01,progressBar=False),printSummary=True)
trainScores=est.ScoreTrainSamples(ScoreFn=ScoreFn)
```

This is the **SVI** configuration from `2_SimpleExampleSvi.py`. The analytical model optimizes quickly, so this example explicitly uses `progressBar=False` to avoid NumPyro's potentially expensive post-optimization processing of the loss history. BET's default remains `progressBar=True` for live feedback on longer runs. For the NUTS example, replace the method with

`bet.Nuts(numWarmup=500, numSamples=500, numChains=4)` . The training scores measure fit on the training patients' last five observations; **they are not out-of-sample validation**, because training used all their observations.

Predict before seeing test observations

```
est.Test(testDataIdxs=jnp.arange(5,10), TestLikelihoodFn=None, nGlobalSamples=None, nLocalZSamples=2000)
noInferenceScores=est.ScoreTestSamples(ScoreFn=ScoreFn)
```

With no test likelihood, BET draws each test patient's local effect from the trained population. `nGlobalSamples=None` uses the median trained global state, so these examples do **not** propagate full global posterior uncertainty. `nLocalZSamples=2000` controls population-drawn local effects.

Personalize using three early observations

```
est.Test(testDataIdxs=jnp.arange(5,10), TestLikelihoodFn=TestLikelihoodFn, Method=bet.Svi, nGlobalSamples=None,
printSummary=True)
testSviScores=est.ScoreTestSamples(ScoreFn=ScoreFn)

print("Training score:", jnp.mean(trainScores))
print("Test score, no inference:", jnp.mean(noInferenceScores))
print("Test score, SVI:", jnp.mean(testSviScores))
```

Passing `bet.Svi` as a **class** reuses the saved SVI configuration, including `progressBar=False` in this example. In the NUTS script, the matching call uses `Method=bet.Nuts` and stores `testNutsScores` . Calling `Test` again replaces the estimator's current test state, but `noInferenceScores` remains available for comparison.

The last five test measurements were not used for population training or test personalization. Comparing the two test scores therefore measures how much the first three observations helped predict later values. A lower error after personalization is plausible, but not guaranteed with a small synthetic sample. These are **latent-trajectory errors**, not full posterior predictive checks including future observation noise.

For a more comprehensive uncertainty analysis, set `nGlobalSamples` to a positive integer to sample global posterior states; this costs more computation. `Save(None)` exposes the saved parameter samples and optionally model outputs, as described in Part 6. The example scripts use the faster median-global setting.

Part 3: Learning correlations between local z-scores

The introductory model has one patient-specific parameter, growth rate `g` . BET can also learn relationships between **multiple** standardized local effects. This section follows the correlated growth-and-drug-response model used in [SaveExampleNuts](#) and [SaveExampleSvi](#). Their corresponding Load scripts use the same model.

Each patient has a growth rate `g` and a drug-response effect `d` . Before treatment, $dv/dt = gv$; treatment starts at `drugStart=4` , after which $dv/dt = (g-d)v$. The exact solution is $v(t) = v_0 * \exp(g*t - d*\max(t-\text{drugStart}, 0))$. It avoids a numerical ODE solver while retaining the treatment-onset behavior.

```
def ModelFn(globalParams, localZs, data):
    g=globalParams["gMean"]+globalParams["gStd"]*localZs["g"]
    d=globalParams["dMean"]+globalParams["dStd"]*localZs["d"]
    tumorSize=data["v0"]*jnp.exp(g*data["times"]-d*jnp.maximum(data["times"]-data["drugStart"],0.0))
```

```

    return {"tumorSize": tumorSize}

def GlobalPriorFn(dataShapes):
    return {
        "gMean": dist.Normal(0.15, 0.1),
        "gStd": dist.HalfNormal(0.1),
        "dMean": dist.Normal(0.25, 0.1),
        "dStd": dist.HalfNormal(0.1),
        "sigma": dist.HalfNormal(0.1),
    }

```

These are the **same functions and parameter names** used in the runnable Save/Load examples. Use `localZNames=["g", "d"]` when constructing the estimator. Unlike the single-local model in Part 2, BET can infer the population correlation between the standardized `g` and `d` effects.

The Save examples generate ten patients with eleven time points between 0 and 10, patient-specific initial tumor sizes, and **synthetic correlation** `rho=0.8` between local effects:

```

rho=0.8
zG=random.normal(keyG, (nPatients,))
zDIndependent=random.normal(keyD, (nPatients,))
zD=rho*zG+jnp.sqrt(1.0-rho**2)*zDIndependent
trueG=0.15+0.03*zG
trueD=0.25+0.05*zD

```

This generates the example data; the true `rho` is **not** supplied to BET during training. The example uses five training patients and five held-out patients. Because this is a small dataset, the inferred correlation need not recover 0.8 accurately.

BET uses a **correlation matrix** and its **Cholesky factor** to represent the relationship among local effects. The `lkjConcentration` argument controls the LKJ prior on correlation matrices: `1` gives uniform density over valid matrices, `2` (default) mildly favors weaker correlations, and larger values favor weaker correlations more strongly. Values below `1` favor more extreme structures. Correlation refers to the standardized local effects, not necessarily to arbitrary nonlinear transformations of the physical parameters.

The runnable Save examples train this correlated model with either `bet.Nuts(...)` or `bet.Svi(...)`, score the training trajectories, and save the fit. Unlike Part 2's test scores, these Save examples' training scores use **all observations** and measure training fit rather than held-out prediction. Part 6 continues with exactly this model.

Part 4: Building blocks

Parts 2 and 3 showed how to train a model, make predictions, and represent correlated differences between individuals. This part serves as a reference for the functions and arguments used in that workflow: how to configure an `Estimator`, run `Train` and `Test`, score predictions, and define the model, priors, and likelihoods they require.

Core workflow: Estimator, Train, Test, and Score

`Estimator(data, ModelFn, localZNames=None, GlobalPriorFn=None, lkjConcentration=2.0, rngSeed=None)` sets up the model:

- `data`: A dictionary of arrays, each with individuals along the first axis. All arrays must have the same number of individuals; remaining axes can represent times, features, or spatial coordinates. For example, measurements from 10 patients at 8 times

give `data["y"]` shape `(10, 8)`; the corresponding one-patient shape, `dataShapes["y"]`, is `(8,)`.

- `ModelFn(globalParams, localZs, data)`: Receives dictionaries of global parameters, one individual's local z-scores, and that individual's data; returns a dictionary of model outputs. Use JAX-compatible operations without Python side effects or converting traced values to NumPy.
- `localZNames`: Names of the local latent z-scores to infer; omit when there are no individual-specific latent variables.
- `GlobalPriorFn(dataShapes)`: Returns a dictionary mapping shared global parameter names to NumPyro prior distributions. `dataShapes` contains each array's shape for one individual, without the leading individual axis. Optional when there are no global parameters to infer.
- `lkjConcentration`: Controls the correlation prior for multiple local z-scores (default `2.0`); see Part 3.
- `rngSeed`: Optional random seed for reproducible sampling.

`est.Train(trainDataIdxs, TrainLikelihoodFn, Method, printSummary=False, rngSeed=None, saveLocalZs=True, saveModelOut=False)` learns from the training individuals:

- `trainDataIdxs`: Indices of the individuals used for training.
- `TrainLikelihoodFn(globalParams, localZs, data, modelOut)`: Returns a dictionary mapping observation site names to `(observed_value, distribution)` tuples. Each site can contain scalar or vector observations; observation, distribution, and mask shapes must agree.
- `Method`: Configured inference method, such as `Nuts(...)` or `Svi(...)`, or a supported Method class that can reuse saved configuration.
- `printSummary`: Whether to print an inference summary.
- `rngSeed`: Optional seed for this training run.
- `saveLocalZs`: Whether to retain training local z-score samples (default `True`). For later training scores, retain either these draws or model outputs.
- `saveModelOut`: Whether to retain model outputs for the training samples.

`est.Test(testDataIdxs, TestLikelihoodFn=None, Method=None, nGlobalSamples=None, nLocalZSamples=2000, printSummary=False, rngSeed=None, saveModelOut=False)` makes predictions for test individuals:

- `testDataIdxs`: Indices of the individuals to predict.
- `TestLikelihoodFn(globalParams, localZs, data, modelOut)`: Optional function with the same return format as `TrainLikelihoodFn`, using available test observations to update local z-scores. Omit for population-only predictions.
- `Method`: Inference method for that update, when a test likelihood is supplied; it can reuse saved configuration.
- `nGlobalSamples`: Number of sampled global parameter states used for testing; `None` uses the posterior median global state; a positive integer samples training posterior states.
- `nLocalZSamples`: Number of local z-score draws per global state for population-only predictions.
- `printSummary`: Whether to print an inference summary.
- `rngSeed`: Optional seed for this test run.
- `saveModelOut`: Whether to retain test model outputs.

`est.ScoreTrainSamples(ScoreFn, nSamples=None, rngSeed=None)` and `est.ScoreTestSamples(ScoreFn, nSamples=None, rngSeed=None)` evaluate predictions:

- `ScoreFn(modelOut, data)`: Compares one individual's model output with their data and returns a scalar or array. Training scores have axes (draw, training individual, score dimensions); test scores have axes (global draw, local draw, test individual, score dimensions).
- `nSamples`: Optional number of posterior draws to score; `None` uses all available draws.
- `rngSeed`: Optional seed when selecting draws.

Additional API details

Models without local z-scores. `Estimator` and `Train` support models without local z-scores, but `Test` currently requires at least one (`Test()` requires at least one local Z). Consequently, `ScoreTestSamples` is unavailable through that workflow.

Saving and loading. `est.Save(fileName, saveTrainLocalZs=True, saveTestLocalZs=True, saveTrainModelOut=False, saveTestModelOut=False, saveAllData=False)` saves the estimator state to a compressed file. With `fileName=None`, it returns the results as a dictionary instead. `Estimator.Load(fileName, ModelFn, GlobalPriorFn=None, data=None)` reconstructs the estimator; supply `data` unless it was saved. With `ModelFn=None`, it returns the saved record without reconstructing an estimator. See Part 6 for examples and persistence options.

Custom methods. `ValidateMethod(method)` checks whether a custom inference method implements BET's required interface. See **Advanced: Adding a custom inference method** at the end of the manual for implementation details.

Part 5: How BET learns from data

After you define the model and observations, BET estimates plausible parameter values using one of two built-in inference methods. Both produce parameter samples, but their algorithms and computational costs differ.

Choosing between NUTS and SVI

NUTS explores plausible parameter values by drawing samples through a Markov chain. It is often useful when the shape and spread of uncertainty matter. `Nuts(numWarmup=2000, numSamples=2000, numChains=4, acceptProb=0.95, denseMass=False, medianSamples=50)` uses NumPyro's NUTS MCMC kernel with median-based initialization. `numWarmup` is the adaptation period; `numSamples` is the retained number of draws **per chain**; `numChains` controls chain count; `acceptProb` is the target acceptance probability; `denseMass` enables dense mass-matrix adaptation; and `medianSamples` controls the initialization strategy. BET combines the chains into one leading draw axis for downstream scoring, so a run with four chains and 2,000 retained draws per chain provides up to 8,000 training posterior draws before any subsampling. Its reported summary includes effective sample size and R-hat. Inspect divergences, mixing, and posterior predictive adequacy rather than treating a successful run as validation.

SVI takes a different approach: it fits a simpler probability distribution to approximate the range of plausible parameter values. It can be faster, although the approximation may miss some uncertainty. `Svi(numSteps=10000, numSamples=2000, stepSize=0.01, numParticles=1, progressBar=True, medianSamples=50, multivariate=False)` optimizes a variational approximation with Adam and an ELBO objective. `numSteps` is the number of optimization updates; `numSamples` controls approximate posterior draws generated after optimization; `stepSize` is Adam's learning rate; `numParticles` controls Monte Carlo particles for the ELBO; and `medianSamples` controls initialization. `multivariate=False` uses `AutoNormal`; `True` uses `AutoMultivariateNormal`, which can represent posterior covariance but may be expensive in large hierarchical models. `progressBar=True` is the default and displays optimization progress. With NumPyro's progress-bar execution path, assembling the loss history after optimization can add noticeable overhead, even when optimization itself is fast. Setting `progressBar=False` selects NumPyro's compiled execution path and can substantially reduce this overhead, at the cost of hiding live progress. The introductory and Save SVI examples explicitly disable the bar for this reason; for longer-running models, the default progress feedback may be more useful.

Method selection. NUTS is often preferable when reliable posterior uncertainty is central and runtime is manageable. SVI can be faster for large problems, but an apparently stable loss curve does not establish calibrated uncertainty. Compare posterior predictive behavior and uncertainty against NUTS or simulated ground truth when feasible. You can pass a configured method, such as `Svi(numSteps=5000)`, directly to `Train` or `Test`. After loading a saved estimator, passing the method class (such as `svi`) lets BET reuse its saved settings, if available.

Custom methods. Developers can integrate algorithms beyond NUTS and SVI through BET's Method interface. See **Advanced: Adding a custom inference method** and `ValidateMethod` .

Part 6: Saving your work

The correlated growth-and-treatment model from Part 3 is also the model used in the four runnable Save/Load examples:

- `3_SaveExampleNuts.py` trains with NUTS and writes `SimpleTumorGrowthNuts.npz` ; `4_LoadExampleNuts.py` restores it.
- `5_SaveExampleSvi.py` trains with SVI and writes `SimpleTumorGrowthSvi.npz` ; `6_LoadExampleSvi.py` restores it.

Run each Save script before its corresponding Load script, from the same working directory. The Load scripts recreate the identical synthetic dataset and supply `ModelFn` and `GlobalPriorFn` explicitly; functions are not serialized.

Saving and loading

After training the correlated model with `localZNames=["g", "d"]` , the NUTS example saves it with:

```
est.Save("SimpleTumorGrowthNuts.npz")
```

The corresponding Load example recreates `data` and the Part 3 model functions, then restores the estimator:

```
est=bet.Estimator.Load("SimpleTumorGrowthNuts.npz", ModelFn=ModelFn, GlobalPriorFn=GlobalPriorFn, data=data)
```

The SVI pair uses `SimpleTumorGrowthSvi.npz` instead. After loading, both scripts score the restored training posterior, then test five new patients without personalization, with three observations, and with seven observations. They evaluate mean absolute error and mean squared error over **entire trajectories**. These scores illustrate the API and the effects of conditioning; the first three or seven observations are part of those scores, so they should **not** be described as wholly held-out forecasting accuracy.

`save` preserves the population posterior, training/test settings, selected indices, and RNG state. By default it also retains available training local samples; model outputs and the full input dataset are not saved unless requested.

```
est.Save("fit.npz", saveAllData=True)
restored=bet.Estimator.Load("fit.npz", ModelFn=ModelFn, GlobalPriorFn=GlobalPriorFn)
```

When `saveAllData=True` , the loader can reconstruct the estimator without separately supplying `data` . This **optional variation** is not what the runnable Save/Load examples use.

The full signature is:

```
Save(fileName, saveTrainLocalZs=True, saveTestLocalZs=True,
      saveTrainModelOut=False, saveTestModelOut=False, saveAllData=False)
```

Option	What it controls
<code>fileName</code>	Destination for the compressed <code>.npz</code> file; <code>None</code> returns an in-memory record.

Option	What it controls
<code>saveTrainLocalZs</code>	Include training local-effect samples (default <code>True</code>).
<code>saveTestLocalZs</code>	Include latest test local effects and associated global/Cholesky states (default <code>True</code>).
<code>saveTrainModelOut</code>	Include retained training model outputs, if available.
<code>saveTestModelOut</code>	Include retained test model outputs, if available.
<code>saveAllData</code>	Include original data so it need not be supplied on load.

The model-output options preserve outputs **already retained** by `Train(..., saveModelOut=True)` or `Test(..., saveModelOut=True)`; they do not generate missing predictions at save time. If local samples and model outputs are omitted, later scoring may be unavailable.

For inspection without reconstructing an estimator, `bet.Estimator.Load("fit.npz", ModelFn=None)` returns the saved record. Method names and constructor arguments are stored to reconstruct inference configuration; live NUTS or SVI engines are not serialized.

Part 7: Working with incomplete data and numerical models

Individuals often have different numbers of measurements, and some measurements may be missing. BET works with arrays, so these differences need to be represented explicitly without pretending missing values were observed.

Different numbers of measurements

Use masks when entries have different observation counts, measurements are missing, or testing should reveal only part of a trajectory. A mask distinguishes *positions stored in an array* from *observations contributing to inference*. Held-out measurements can remain in the data for scoring while being excluded from the test likelihood.

Pad each time series to a common length `T`. Store `time`, `y`, and a boolean `valid` mask as `[N, T]` arrays; padded positions are placeholders, not observations. Choose finite placeholder measurements that are safe for any transformations in your model; repeating the last valid measurement is often useful. Choose padded times separately to satisfy your solver's requirements: some solvers reject repeated times and need a strictly increasing grid. Exclude all padded positions from the likelihood.

```
data = {
    "time": jnp.array([[0., 1., 2.], [0., 3., 4.]]),
    "y": jnp.array([[1.0, 1.2, 1.4], [0.9, 1.1, 1.1]]),
    "valid": jnp.array([[True, True, True], [True, True, False]]),
}
```

Use a masked NumPyro distribution and replace invalid observed values with finite, safe values before log-probability evaluation:

```
def likelihood(global_params, local_zs, data, model_out):
    valid = data["valid"]
    pred = model_out["prediction"]
    safe_y = jnp.where(valid, data["y"], pred)
    return {"y": (safe_y, dist.Normal(pred, 0.2).mask(valid))}
```

The model must produce finite predictions even at padded positions. Masking cannot make an intermediate `NaN` or infinity safe; multiplying a divergent log-probability by zero does not guarantee a finite gradient. Here the second individual's final measurement is a placeholder, and its padded time advances from 3 to 4 to keep the time array increasing. The validity mask excludes that position from inference. A real ODE model still needs a time grid compatible with its solver and finite predictions at padded positions.

For partial-observation testing, use a separate mask such as `test_observed = valid & (time <= cutoff)`. Apply it in the test likelihood, but retain the original validity mask for scoring held-out observations. Ensure hidden measurements cannot enter the model through inputs or preprocessing.

Models defined by differential equations

Some scientific models specify *rates of change* rather than values directly. A differential equation solver converts those rates into a predicted trajectory. `ModelFn` can call a JAX-compatible ODE solver such as `jax.experimental.ode.odeint`. Use JAX arrays and differentiable operations, and return outputs at observation times. In the workflows tested for this project, `odeint` has worked more reliably with BET's batching than Difffrax. Compatibility is model-dependent, so verify gradients and numerical behavior. Difffrax is an optional external dependency, not part of BET's installation requirements.

```
from jax.experimental.ode import odeint

def rhs(state, time, rate):
    return rate * state

def model_fn(global_params, local_zs, data):
    rate = jnp.exp(global_params["log_rate"] + local_zs["rate"])
    times = data["time"] # sorted, finite, compatible with odeint
    trajectory = odeint(rhs, data["initial"], times, rate)
    return {"prediction": trajectory}
```

Padded time grids must satisfy the solver's ordering requirements. One approach is to solve on a shared regular grid and interpolate to each individual's observation times. Check for duplicate or unsorted padded times.

Part 8: Evaluating predictions without data leakage

A model can reproduce its training data yet predict poorly for new individuals. Part 2 demonstrated scoring; here the focus is on designing an evaluation that measures genuine predictive performance.

Separate fitting, personalization, and evaluation

Keep test individuals out of training. For each test individual, distinguish measurements available **at prediction time** from later outcomes reserved for evaluation. In the worked example, the first three measurements can update the individual's local z-scores, while the final five are held out for scoring. Neither the model inputs nor the test likelihood should expose those future outcomes during prediction.

Training scores answer a different question: how well does the fitted model describe individuals it learned from? They can reveal poor fit but are not an unbiased estimate of performance on new individuals. Population-only test scores measure prediction without individual observations; personalized test scores measure prediction after using the permitted early observations. Compare the latter two against the **same held-out outcomes** to assess the value of personalization.

Interpret scores at the right level

The worked example uses mean absolute error (MAE), where lower is better. BET computes a score for each sampled prediction and individual. A mean across posterior draws summarizes errors under the fitted model; draws are not independent patients or repeated validation experiments. In particular, the mean of MAEs computed on sampled trajectories need not equal the MAE of the posterior-mean trajectory.

A single overall mean can hide differences between patients. To check whether personalization helps broadly rather than only for a few individuals, retain the test-individual axis:

```
before_by_person = jnp.mean(before, axis=(0, 1))
after_by_person = jnp.mean(after, axis=(0, 1))
print("MAE without calibration, per person:", before_by_person)
print("MAE with calibration, per person:", after_by_person)
```

These arrays have one value per test patient. Quantiles across posterior draws describe uncertainty conditional on the fitted model; they are **not** confidence intervals for performance across future patient populations.

Check what uncertainty is represented

Using positive `nGlobalSamples` propagates uncertainty in learned global parameters and correlations; local z-score draws represent uncertainty and variation between individuals. Deterministic model outputs do not automatically include new observation noise, and none of these draws accounts for possible errors in the model's structure.

For stronger evidence of predictive performance, evaluate more held-out individuals, repeat across data splits or random seeds, and vary how many early observations are available for personalization. These checks address different questions from simply drawing more posterior samples.

Part 9: Troubleshooting

Bayesian inference can be fragile when a model is difficult to fit. Parameters that compensate for one another, parameters that barely affect predictions, and very large or unstable gradients can make reliable fitting difficult or impossible. Complex dynamics and likelihoods can make the problem harder still.

Build complexity incrementally. Start with a simple model that fits reliably, ideally checking parameter recovery on synthetic data with known values. Add one mechanism or parameter at a time and check that inference still behaves sensibly before proceeding. If fitting breaks down, investigate the most recent addition rather than debugging the entire complex model at once. Building toward a complex model is usually easier to debug than fitting it all at once.

Keep model evaluations fast and differentiable. NUTS and SVI may evaluate the model and its gradients many thousands of times during training or test-time personalization. For differential-equation models, use a differentiable ODE solver rather than manually simulating the dynamics with a Python time-step loop; use an analytical solution when one is available. Check runtime across plausible parameter values: regimes that force a numerical solver to take tiny steps can make inference prohibitively slow.

For specific failures, use the checks below. See Part 4 for the API reference.

Common problems

Shape mismatch. Start with one entry and print array shapes before tracing. All top-level data arrays must have the same entry count, and per-entry model outputs must match likelihood observation shapes. Broadcasting can silently produce an incorrect

likelihood, so verify dimensions explicitly.

Invalid entry indices. Training and test indices must be nonempty, one-dimensional collections of unique integers within the data's entry range. BET rejects duplicate, negative, and out-of-range indices; it does not silently remove duplicates or interpret negative indices as Python indexing from the end.

Local z-scores mistaken for physical parameters. A local z-score has a standard-normal marginal prior and is not measured in units of growth rate, drug sensitivity, or observation noise. Specify each transformation to physical units explicitly, and document the resulting units.

Hidden test outcomes leaking into inference. Keeping the full `y` array for scoring is safe only if the test likelihood masks unavailable outcomes and the mechanistic model does not read them. Check preprocessing, features, and masks to prevent future measurements from influencing predictions.

Missing data and NaNs. Masked observations still require finite values at points where JAX evaluates intermediate expressions. Use safe finite placeholders and keep ODE solver time grids valid even for padded observations.

Unexpectedly narrow intervals. With `nGlobalSamples=None`, BET fixes global parameters at posterior medians and does not propagate their uncertainty. Use a positive `nGlobalSamples` to include it. Approximate SVI posteriors may also underestimate uncertainty.

Poor convergence. Check observation scaling, priors, parameter identifiability, and solver gradients. For NUTS, inspect mixing and convergence diagnostics; for SVI, inspect the loss trajectory and sensitivity to initialization.

Save/Load mismatch. Restore compatible data shapes, Method configuration, model function, and global prior function. Python functions are not serialized; stored names are metadata, not executable code.

Advanced: Adding a custom inference method

Most users can rely on NUTS or SVI. This section explains how developers can add another inference algorithm without changing BET's training, testing, or scoring workflows.

BET separates inference algorithms from its population and individual workflows. NUTS and SVI are built in; to integrate another backend, implement the minimum Method interface and validate it before fitting.

A configured Method instance must implement:

- `_Fit(ModelFn, rngKey, samplePrefixes)` : run inference and return a dictionary with `"output"` (backend-specific results or diagnostics) and `"samples"` (posterior or explicitly approximate-posterior draws).
- `_SaveArgs()` : return a dictionary of constructor keyword arguments so BET can reconstruct the Method from saved configuration.

The `samples` dictionary maps NumPyro sample-site names to arrays with *draws along the first axis*. Retain only sites whose names start with one of `samplePrefixes`; BET uses these names to identify global parameters, local z-scores, Cholesky factors, and optionally model outputs.

This minimal skeleton illustrates the interface:

```
class MyMethod:
    _printMode = None
```

```

def __init__(self, nSamples=100):
    self.nSamples = nSamples

def _Fit(self, ModelFn, rngKey, samplePrefixes):
    # Replace with an actual inference backend.
    rawSamples, backendOutput = run_my_inference(ModelFn, rngKey, self.nSamples)
    samples = {name: value for name, value in rawSamples.items()
               if any(name.startswith(prefix) for prefix in samplePrefixes)}
    return {"output": backendOutput, "samples": samples}

def _SaveArgs(self):
    return {"nSamples": self.nSamples}

```

`run_my_inference` is a placeholder, *not a BET function*. The backend must produce meaningful posterior or clearly identified approximate-posterior draws. Repeating a maximum-a-posteriori (MAP) point does not measure posterior uncertainty.

Validate the Method's structure before using it:

```

import BayesianEstimationTool as bet

method = MyMethod(nSamples=100)
bet.ValidateMethod(method)

```

`ValidateMethod` checks required callable signatures, saved constructor arguments, and printing hooks. It *does not execute* `_Fit`, inspect returned samples or shapes, or establish statistical correctness. Test those separately on small synthetic models and with posterior predictive checks.

Optional features include:

- **Diagnostics and printing:** Set `_printMode` to `"before"`, `"after"`, or `"replace"`, and implement `_Print(output, localNames, globalNames, title=None)`. With `_printMode=None`, BET uses its generic sample summary.
- **Configuration:** Accept options for initialization, optimization, sampling, and progress reporting, and return reproducible constructor options from `_SaveArgs()`.
- **Persistence:** Keep configuration reconstructible through constructor keyword arguments. The backend `output` is returned to the caller but is not automatically serialized as a live inference-engine session.
- **Inference quality:** Compare posterior summaries, predictive distributions, and computational costs against NUTS on a known model. This is behavioral testing, not structural validation.

See `NutsMethod.py` and `SviMethod.py` for complete Method implementations.