PyMC - Samplers

Lecture 19

Dr. Colin Rundel

Gibbs Sampler

The Gibbs sampler is an MCMC algorithm for drawing samples from a multivariate distribution by breaking it into a sequence of simpler 1D (or lower-dimensional) sampling problems.

Rather than sampling all parameters simultaneously, we cycle through each parameter one at a time (or in blocks), drawing from their distribution *conditional on the current values of all other parameters*.

Algorithm:

1. Start with an initial guess for all parameters
2. Pick a parameter and sample a new value for it, holding all others fixed
3. Repeat for every other parameter (one full cycle = one iteration)
4. Repeat iterations until the chain converges

Effective Sample Size (ESS)

Because consecutive MCMC draws are correlated, N draws are worth fewer than N independent samples.

ESS quantifies this:

$$\text{ESS} = \frac{N}{1 + 2\sum_{t=1}^{\infty} \rho_t}$$

where ρ_t is the autocorrelation at lag t. The denominator (τ) is the integrated autocorrelation time — the effective number of draws needed to produce one independent sample.

- If draws are independent ($\rho_t = 0$ for all $t > 0$) then $\text{ESS} = N$
- High autocorrelation $\Rightarrow$ large τ $\Rightarrow$ small ESS

Example 1
Banana Distribution

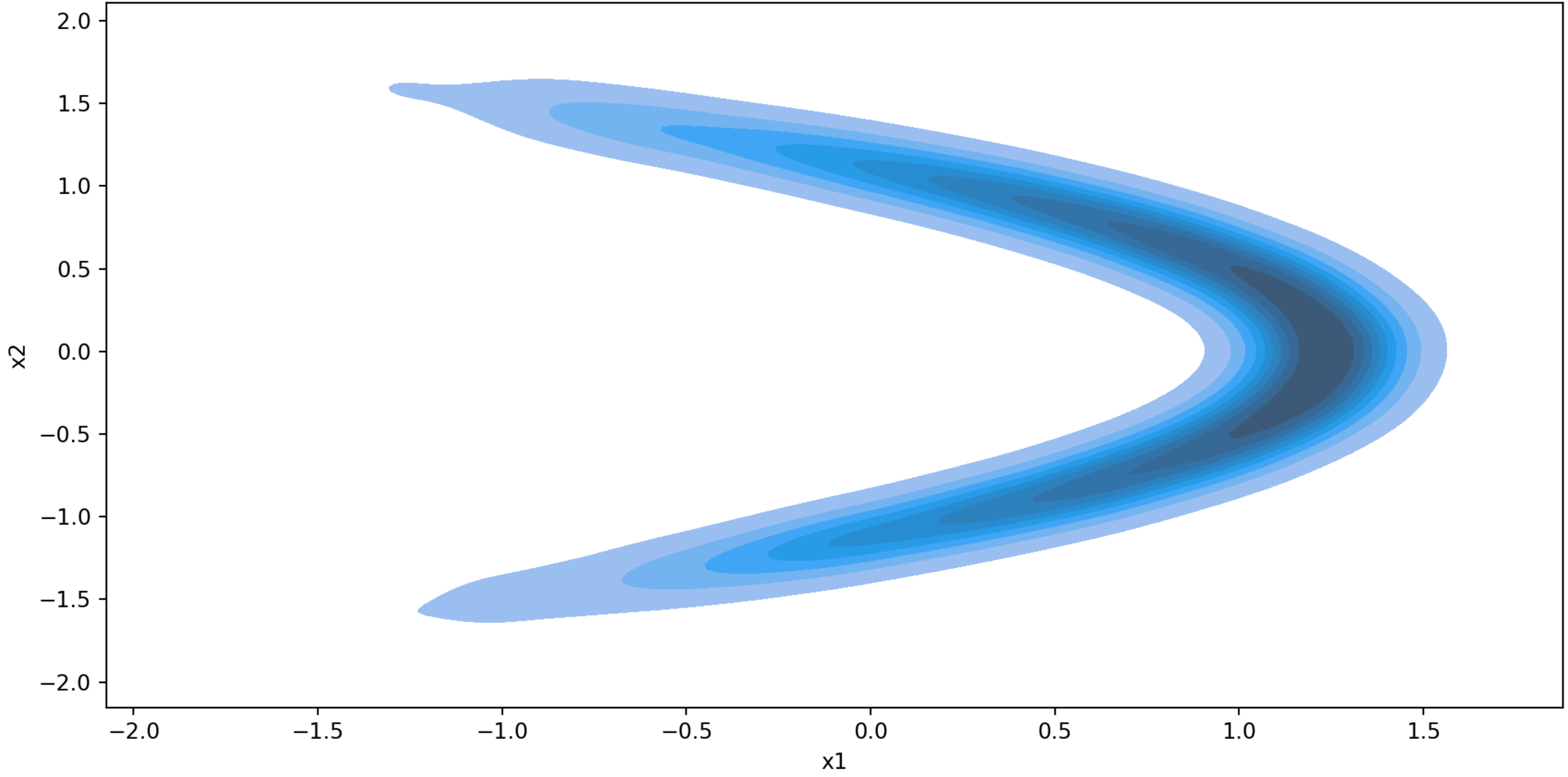
Banana Distribution

```
# Data
n = 100
x1_mu = .75
x2_mu = .75
y = pm.draw(pm.Normal.dist(mu=x1_mu+x2_mu**2, sigma=1, shape=n))

# Model
with pm.Model() as banana:
  x1 = pm.Normal("x1", mu=0, sigma=1)
  x2 = pm.Normal("x2", mu=0, sigma=1)

  y = pm.Normal("y", mu=x1+x2**2, sigma=1, observed=y)
```

Joint posterior of x1 & x2



Metropolis-Hastings

MH Algorithm

For a parameter of interest start with an initial value θ_0 then for the next sample $(t+1)$,

1. Generate a proposal value θ' from a proposal distribution $q(\theta'|\theta_t)$.
2. Calculate the acceptance probability,

$$\alpha = \min\left(1, \frac{P(\theta'|x)}{P(\theta_t|x)} \frac{q(\theta_t|\theta')}{q(\theta'|\theta_t)}\right)$$

   where $P(\theta|x)$ is the target posterior distribution.
3. Accept proposal θ' with probability α, if accepted $\theta_{t+1} = \theta'$ else $\theta_{t+1} = \theta_t$.

Some considerations:

- Choice of the proposal distribution matters a lot
- Results are for the limit as $t \to \infty$
- Concerns are around efficiency (ess / s)

Metropolis-Hastings Sampler

Model | Traces | ACF | Trajectories

```
with banana:
  mh = pm.sample(
    draws=100, tune=0,
    step=pm.Metropolis([x1,x2]),
    random_seed=1234
  )
```

Progress	Draws	Tuning	Scaling	Accept Rate	Sampling Speed
━━━━━━━━━━━━━━━━━━━━━━━━	99	False	1.00	0.00	14532.77 drawss/s
━━━━━━━━━━━━━━━━━━━━━━━━	99	False	1.00	0.44	12752.53 drawss/s
━━━━━━━━━━━━━━━━━━━━━━━━	99	False	1.00	10.09	11974.08 drawss/s
━━━━━━━━━━━━━━━━━━━━━━━━	99	False	1.00	0.01	8114.08 drawss/s

```
az.summary(mh)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
x1	0.262	0.624	-0.607	1.313	0.293	0.111	5.0	45.0	2.71
x2	0.033	0.994	-1.322	1.243	0.446	0.147	5.0	8.0	3.35

MH with Tuning (burnin + adaptation)

Model | Traces | ACF | Trajectories

```
with banana:
  mht = pm.sample(
    draws=100, tune=1000,
    step=pm.Metropolis([x1,x2]),
    random_seed=1234
  )
```

Progress	Draws	Tuning	Scaling	Accept Rate	Sampling Speed	Elapsed
━━━━━━━━	1099	False	0.59	0.00	18396.02 drawss/s	0:00:00
━━━━━━━━	1099	False	0.53	0.46	17389.62 drawss/s	0:00:00
━━━━━━━━	1099	False	0.59	0.00	15877.37 drawss/s	0:00:00
━━━━━━━━	1099	False	0.59	0.00	10220.69 drawss/s	0:00:00

```
az.summary(mht)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
x1	0.346	0.750	-0.676	1.261	0.354	0.066	5.0	34.0	2.37
x2	-0.038	0.963	-1.330	1.381	0.448	0.185	5.0	7.0	3.41

Effects of tuning / burn-in

There are two confounded effects from letting the sampler tune / burn-in:

1. We have let the sampler run for 1000 iterations - this gives it a chance to find the areas of higher density and settle in.

   This also makes each chain less sensitive to their initial starting position.

2. We have also tuned the size of the MH proposals to achieve a better acceptance rate(s) - this lets the chains better explore the target distribution.

   PyMC uses an adaptive algorithm to adjust the proposal size during the tuning phase to achieve an acceptance rate between 0.2 and 0.5.

More samples?

Model | Traces | ACF | Trajectories

```python
with banana:
  mh_more = pm.sample(
    draws=1000, tune=1000,
    step=pm.Metropolis([x1,x2]),
    random_seed=1234
  ).sel(
    draw=slice(None,None,10)
  )
```

Progress	Draws	Tuning	Scaling	Accept Rate	Sampling Speed	Elapsed
━━━━━━━━━━	1999	False	0.59	0.04	18041.91 drawss/s	0:00:00
━━━━━━━━━━	1999	False	0.53	0.52	17718.31 drawss/s	0:00:00
━━━━━━━━━━	1999	False	0.59	0.00	16890.46 drawss/s	0:00:00
━━━━━━━━━━	1999	False	0.59	0.00	10559.66 drawss/s	0:00:00

```python
az.summary(mh_more)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
x1	0.643	0.601	-0.616	1.337	0.205	0.099	11.0	71.0	1.32
x2	0.030	0.786	-1.264	1.385	0.277	0.130	7.0	16.0	1.60

Even more samples?

Model | Traces | ACF | Trajectories

```
with banana:
  mh_more2 = pm.sample(
    draws=10000, tune=1000,
    step=pm.Metropolis([x1,x2]),
    random_seed=1234
  ).sel(
    draw=slice(None,None,100)
  )
```

Progress	Draws	Tuning	Scaling	Accept Rate	Sampling Speed	Elapsed
━━━━━━━━━━	10999	False	0.59	0.00	19056.75 drawss/s	0:00:00
━━━━━━━━━━	10999	False	0.53	0.00	18440.16 drawss/s	0:00:00
━━━━━━━━━━	10999	False	0.59	1.55	17167.10 drawss/s	0:00:00
━━━━━━━━━━	10999	False	0.59	0.00	10766.02 drawss/s	0:00:01

```
az.summary(mh_more2)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
x1	0.439	0.850	-1.190	1.386	0.146	0.153	47.0	38.0	1.08
x2	-0.251	0.867	-1.666	1.142	0.124	0.057	45.0	38.0	1.07

Slice Sampling

Slice Algorithm

Given a current value θ_t, each iteration proceeds as:

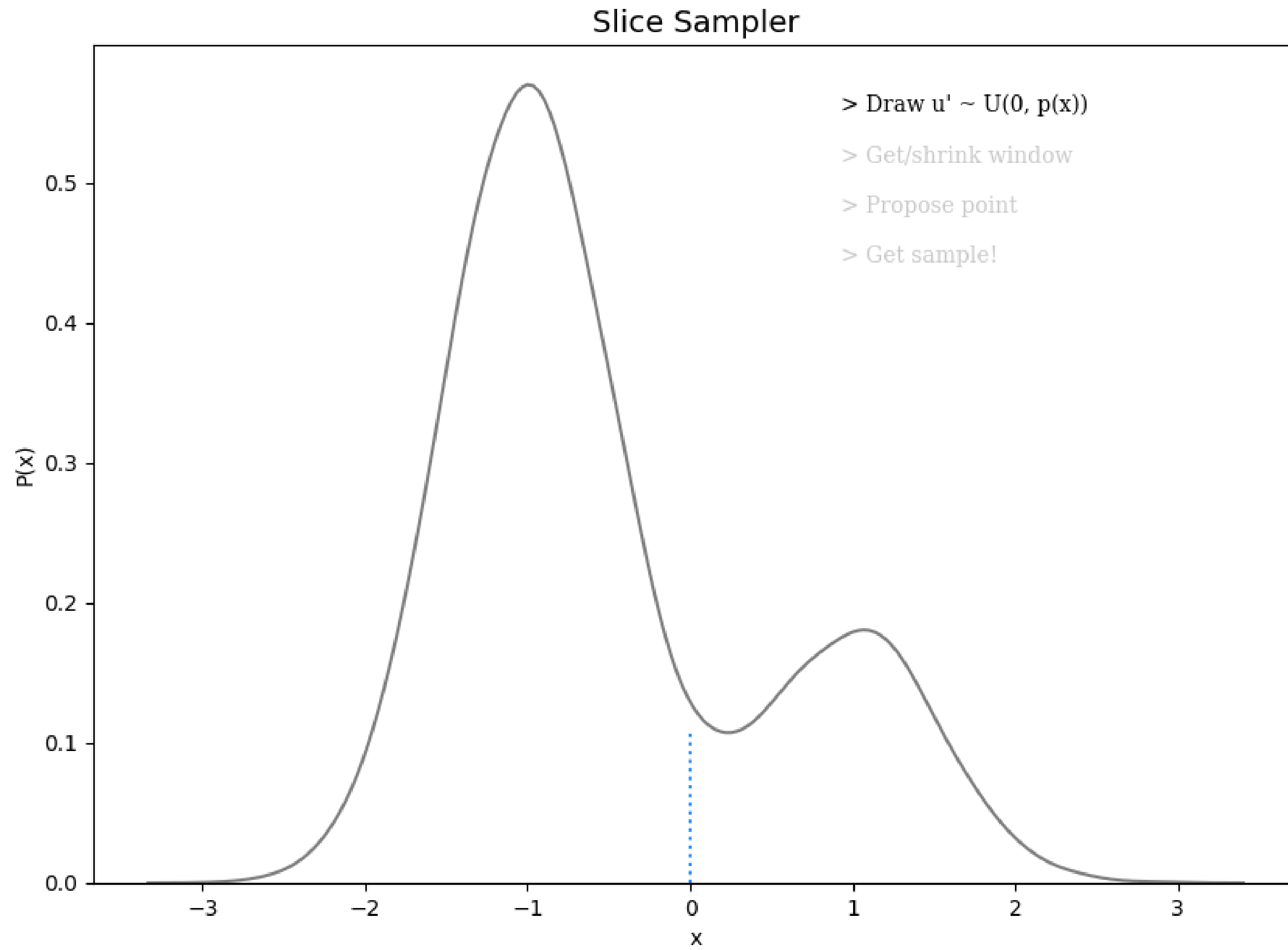
1. Draw a "height" uniformly below the current density
2. Define the "slice" — the level set at height u
3. Draw the next sample uniformly from the slice

- $u \sim \text{Uniform}(0,\ p(\theta_t \mid x))$
- $S = \{\theta : p(\theta \mid x) \geq u\}$
- $\theta_{t+1} \sim \text{Uniform}(S)$

:::

In practice, finding S exactly is rarely feasible, so a **stepping-out / shrinkage** procedure is used: bracket θ_t with an interval, expand it until both ends are outside the slice, then shrink inward whenever a proposed draw is rejected.

Animation



Slice Samples

Model | Traces | ACF | Trajectories

```
with banana:
  sl = pm.sample(
    draws=100, tune=1000,
    step=pm.Slice([x1, x2]),
    random_seed=1234
  )
```

Progress	Draws	Tuning	Steps out	Steps in	Sampling Speed	Elapsed
━━━━━━━━━━	1099	False	1	4	18961.80 drawss/s	0:00:00
━━━━━━━━━━	1099	False	0	2	17856.10 drawss/s	0:00:00
━━━━━━━━━━	1099	False	0	2	16656.00 drawss/s	0:00:00
━━━━━━━━━━	1099	False	1	3	12447.50 drawss/s	0:00:00

```
az.summary(sl)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
x1	0.400	0.569	-0.762	1.302	0.164	0.087	12.0	20.0	1.26
x2	0.257	0.887	-1.464	1.193	0.347	0.135	6.0	14.0	1.78

More Slice Samples

Model | Traces | ACF | Trajectories

```
with banana:
  sl_more = pm.sample(
    draws=1000, tune=1000,
    step=pm.Slice([x1, x2]),
    random_seed=1234
  ).sel(
    draw=slice(None,None,10)
  )
```

Progress	Draws	Tuning	Steps out	Steps in	Sampling Speed	Elapsed
━━━━━━━━━━	1999	False	0	1	18994.17 drawss/s	0:00:00
━━━━━━━━━━	1999	False	0	0	19540.84 drawss/s	0:00:00
━━━━━━━━━━	1999	False	0	3	17811.94 drawss/s	0:00:00
━━━━━━━━━━	1999	False	1	0	12997.13 drawss/s	0:00:00

```
az.summary(sl_more)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
x1	0.388	0.74	-0.926	1.431	0.198	0.048	15.0	114.0	1.21
x2	0.331	0.88	-1.449	1.463	0.234	0.083	12.0	43.0	1.30

Hamiltonian Samplers

Background

Hamiltonian Monte Carlo (HMC) conceptualizes the model parameters as a particle moving through the posterior landscape. These particles have a position (θ) and an auxiliary momentum (ρ), and their joint dynamics are governed by the **Hamiltonian**:

$$H(\theta, \rho) = V(\theta) + T(\rho \mid \theta)$$

$$\underbrace{V(\theta)}_{\text{potential energy}} = -\log p(\theta \mid x),$$

$$\underbrace{T(\rho \mid \theta)}_{\text{kinetic energy}} = -\log p(\rho \mid \theta)$$

In most applications of HMC, the auxiliary density is a multivariate normal that does not depend on the parameters, $\rho \sim \square(0, M)$.

Hamiltonian dynamics

Given this setup Hamilton's equations of motion give a set of PDEs governing the particle's trajectory.

$$\frac{d\theta}{dt} = \frac{\partial H}{\partial \rho} = \frac{\partial}{\partial \rho} - \log p(\rho \mid \theta)$$

$$\frac{d\rho}{dt} = -\frac{\partial H}{\partial \theta} = \nabla_\theta \log p(\theta \mid x)$$

The first equation implies that position changes in the direction of the momentum.

The second implies momentum is accelerated by the gradient of the log-posterior — high-density regions attract the particle.

Along any exact trajectory $H(\theta, \rho)$ is conserved, meaning the particle moves without changing its total energy.

Leapfrog Integrator

The dynamics are solved numerically using the leapfrog integrator.

Starting from (θ_t, ρ_t), each leapfrog step of size ϵ proceeds as:

$$
\begin{aligned}
\rho_{t+1/2} &= \rho_t + \frac{\epsilon}{2} \nabla \log p(\theta_t \mid x) \\
\theta_{t+1} &= \theta_t + \epsilon M^{-1} \rho_{t+1/2} \\
\rho_{t+1} &= \rho_{t+1/2} + \frac{\epsilon}{2} \nabla \log p(\theta_{t+1} \mid x)
\end{aligned}
$$

This is repeated L times to produce a proposed (θ', ρ'). A Metropolis acceptance step is then used to correct for any numerical error by accepting the proposed move with probability

$$
\alpha = \min\Big(1,\ \exp\Big(H(\theta, \rho) - H(\theta', \rho')\Big)\Big)
$$

If the integrator were exact, H would be conserved and $\alpha = 1$ always — the MH step only corrects for numerical error, so acceptance rates are typically very high.

HMC Algorithm

For the current parameter value θ_t:

1. Sample a momentum $\rho \sim \square(0, M)$
2. Run L leapfrog steps of size ϵ to obtain a proposal (θ', ρ')
3. Accept θ' with probability

$$\alpha = \min\Big(1,\ \exp\Big(H(\theta_t, \rho) - H(\theta', \rho')\Big)\Big)$$

The new momentum draw at step 1 randomizes the direction of travel each iteration, ensuring the chain is ergodic.

Algorithm parameters

- ϵ **(step size)** — controls the size of each leapfrog step. Too large $\Rightarrow$ large numerical error, low acceptance rate; too small $\Rightarrow$ fine-grained but slow exploration.
- M **(mass matrix)** — acts as a preconditioning matrix. A diagonal M rescales parameters with different variances; a full M also accounts for correlations.
- L **(number of leapfrog steps)** — controls how far the particle travels each iteration. Too few $\Rightarrow$ short trajectories; too many $\Rightarrow$ the trajectory can curve back on itself (a "U-turn"), wasting computation without improving exploration.

All three are tuned automatically during warmup, but L is particularly difficult to tune.

HamiltonianMC

Model | Traces | ACF | Trajectories

```
with banana:
  hmc = pm.sample(
    draws=100, tune=1000,
    step=pm.HamiltonianMC([x1,x2]),
    random_seed=1234
  )
```

Progress	Draws	Divergences	Grad evals	Sampling Speed	Elapsed
━━━━━━━━━━	1099	8	21	4874.74 drawss/s	0:00:00
━━━━━━━━━━	1099	3	10	6531.36 drawss/s	0:00:00
━━━━━━━━━━	1099	36	15	5428.16 drawss/s	0:00:00
━━━━━━━━━━	1099	8	16	5343.53 drawss/s	0:00:00

```
az.summary(hmc)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
x1	0.589	0.593	-0.386	1.271	0.200	0.080	11.0	6.0	1.34
x2	0.269	0.771	-0.998	1.246	0.232	0.049	9.0	73.0	1.43

More HamiltonianMC

Model | Traces | ACF | Trajectories

```
with banana:
  hmc_more = pm.sample(
    draws=1000, tune=1000,
    step=pm.HamiltonianMC([x1,x2]),
    random_seed=1234
  ).sel(
    draw=slice(None,None,10)
  )
```

Progress	Draws	Divergences	Grad evals	Sampling Speed	Elapsed
━━━━━━━━━━	1999	137	17	5480.62 drawss/s	0:00:00
━━━━━━━━━━	1999	218	9	7819.25 drawss/s	0:00:00
━━━━━━━━━━	1999	161	13	6416.78 drawss/s	0:00:00
━━━━━━━━━━	1999	103	18	6089.80 drawss/s	0:00:00

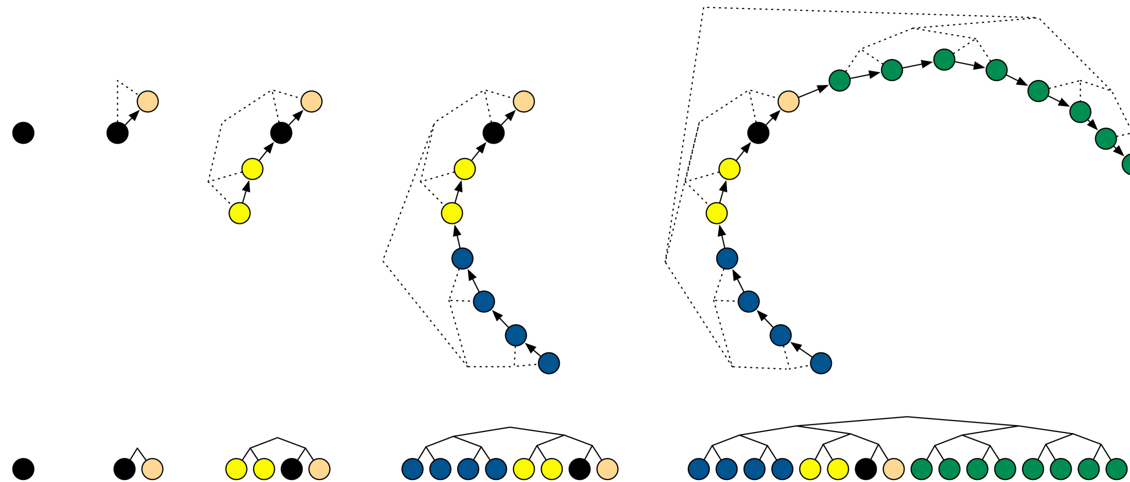
```
az.summary(hmc_more)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
x1	0.651	0.524	-0.390	1.387	0.040	0.031	210.0	123.0	1.02
x2	0.088	0.772	-1.083	1.341	0.052	0.024	189.0	116.0	1.02

No-U-turn sampler (NUTS)

This is a variation of Hamiltonian Monte Carlo that automatically tunes the number of leapfrog steps to allow more effective exploration of the parameter space.

Specifically, it uses a tree based algorithm that tracks trajectories forwards and backwards in time. The tree expands until a maximum depth is achieved or a “U-turn” is detected.



NUTS also does not use a metropolis step to select the final parameter value, instead the sample is chosen among the valid candidates along the trajectory.

NUTS

Model | Traces | ACF | Trajectories

```
with banana:
  nuts = pm.sample(
    draws=100, tune=1000,
    step=pm.NUTS([x1,x2]),
    random_seed=1234
  )
```

Progress	Draws	Divergences	Step size	Grad evals	Sampling Speed	Elapsed
━━━━━━━━━━	1099	1	0.170	15	4038.02 drawss/s	0:00:00
━━━━━━━━━━	1099	0	0.102	31	3748.27 drawss/s	0:00:00
━━━━━━━━━━	1099	2	0.139	59	3871.63 drawss/s	0:00:00
━━━━━━━━━━	1099	1	0.187	15	3634.27 drawss/s	0:00:00

```
az.summary(nuts)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
x1	0.826	0.389	0.106	1.369	0.048	0.023	60.0	51.0	1.07
x2	0.089	0.662	-0.985	1.127	0.152	0.032	19.0	83.0	1.20

More NUTS

Model | Traces | ACF | Trajectories

```
with banana:
  nuts_more = pm.sample(
    draws=1000, tune=1000,
    step=pm.NUTS([x1,x2]),
    random_seed=1234
  ).sel(
    draw=slice(None,None,10)
  )
```

Progress	Draws	Divergences	Step size	Grad evals	Sampling Speed	Elapsed
━━━━━━━━━━	1999	5	0.170	5	4236.00 drawss/s	0:00:00
━━━━━━━━━━	1999	6	0.102	11	4023.28 drawss/s	0:00:00
━━━━━━━━━━	1999	45	0.139	3	4716.16 drawss/s	0:00:00
━━━━━━━━━━	1999	28	0.187	7	4048.82 drawss/s	0:00:00

```
az.summary(nuts_more)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
x1	0.551	0.665	-0.788	1.423	0.099	0.089	67.0	31.0	1.04
x2	-0.098	0.838	-1.514	1.188	0.091	0.052	63.0	33.0	1.04

Some considerations

- Hamiltonian MC methods are all very sensitive to the choice of their tuning parameters (NUTS less so, but adds additional parameters)
- Hamiltonian MC methods require the gradient of the log density of the parameter(s) of interest for the leapfrog integrator - *this limits this method to continuous parameters*
- HMC updates are generally more expensive computationally than MH updates, but they also tend to produce chains with lower autocorrelation. Think about performance in terms of *effective samples per unit of time*.

Divergent transitions

Using Stan or PyMC with NUTS you will often see messages / warnings about divergent transitions or divergences.

This is based on the assumption of conservation of energy with regard to the Hamiltonian system - $H(\theta, \rho)$ should remain constant for the "particle" along its trajectory.

When the $H(\theta, \rho)$ resulting from the leapfrog integrator differs from the initial value then a divergence is considered to have occurred.

- The proximate cause of this is a breakdown of the first-order approximations in the leapfrog algorithm.
- The ultimate cause is usually a highly curved posterior or a posterior where the rate of curvature is changing rapidly.

Solutions?

Very much depend on the nature of the problem - typically we can reparameterize the model and/or adjust some of the tuning parameters to help the sampler deal with the problematic posterior.

For the latter the following options can be passed to `pm.sample()` or `pm.NUTS()`:

- `target_accept` - step size is adjusted to achieve the desired acceptance rate (larger values result in smaller steps which often work better for problematic posteriors). Default value is 0.8, increase for smaller steps and fewer divergences, decrease for larger steps and more exploration.
- `max_treedepth` - maximum depth of the trajectory tree. Default value is 10, increase for deeper exploration, decrease for faster sampling.
- `step_scale` - the initial guess for the step size before warmup adaptation. Default value is 0.25 which is further scaled based on dimensionality.

NUTS (adjusted)

Model | Traces | ACF | Trajectories

```python
with banana:
  nuts2 = pm.sample(
    draws=1000, tune=1000,
    step=pm.NUTS([x1,x2],
    target_accept=0.9),
    random_seed=1234
  ).sel(
    draw=slice(None,None,10)
  )
```

Progress	Draws	Divergences	Step size	Grad evals	Sampling Speed	Elapsed
━━━━━━━━━━	1999	1	0.118	7	3022.47 drawss/s	0:00:00
━━━━━━━━━━	1999	0	0.079	3	2968.01 drawss/s	0:00:00
━━━━━━━━━━	1999	0	0.081	63	2886.71 drawss/s	0:00:00
━━━━━━━━━━	1999	2	0.074	15	2918.71 drawss/s	0:00:00

```python
az.summary(nuts2)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
x1	0.483	0.676	-0.780	1.400	0.037	0.027	338.0	374.0	1.01
x2	-0.027	0.876	-1.312	1.472	0.051	0.020	301.0	292.0	1.00

Sampler Comparison

Sampler	Gradient needed	Handles discrete	Autocorrelation	Notes
Metropolis-Hastings	No	Yes	High	Simple; performance depends heavily on proposal tuning
Slice	No	No	Low–medium	Auto-adapts; more density evaluations per draw
HMC	Yes	No	Low	Efficient on correlated posteriors; L hard to tune
NUTS	Yes	No	Low	Auto-tunes L; PyMC/Stan default

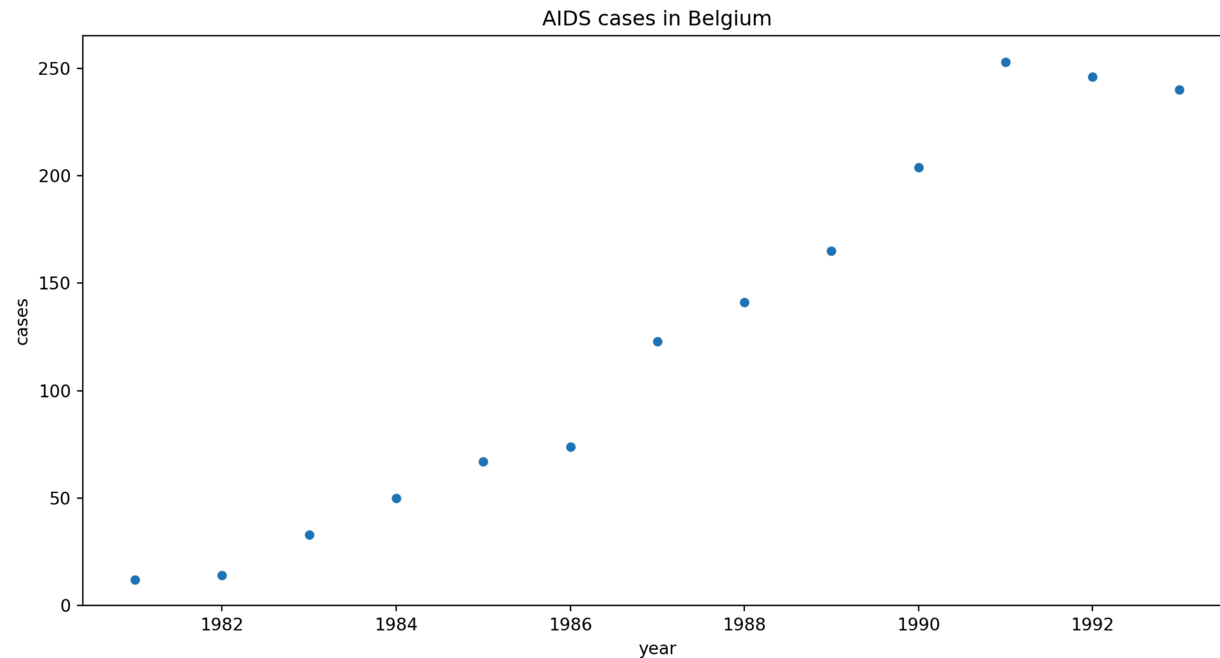
In general, prefer NUTS for continuous models — it is the most efficient in effective samples per unit time. Fall back to MH or a compound sampler only when discrete parameters are present.

Example 2
Poisson Regression

AIDS cases in Belgium from 1981 to 1993

```
1 aids
```

```
    year  cases
0   1981     12
1   1982     14
2   1983     33
3   1984     50
4   1985     67
5   1986     74
6   1987    123
7   1988    141
8   1989    165
9   1990    204
10  1991    253
11  1992    246
12  1993    240
```



Model

```
y, X = patsy.dmatrices("cases ~ year", aids)

X_lab = X.design_info.column_names
y = np.asarray(y).flatten()
X = np.asarray(X)

with pm.Model(coords = {"coeffs": X_lab}) as model:
    b = pm.Cauchy("b", alpha=0, beta=1, dims="coeffs")
    η = X @ b
    λ = pm.Deterministic("λ", np.exp(η))

    likelihood = pm.Poisson("y", mu=λ, observed=y)

    post = pm.sample(random_seed=1234)
```

Progress	Draws	Divergences	Step size	Grad evals	Sampling Speed
━━━━━━━━━━	1999	0	0.000	7	8210.92 drawss/s
━━━━━━━━━━	1999	0	0.000	1	6266.68 drawss/s
━━━━━━━━━━	1999	0	0.001	1023	171.46 draws/s
━━━━━━━━━━	1999	0	0.002	831	205.55 draws/s

Summary

```
1  az.summary(post)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcs
b[Intercept]	-9.897300e+01	1.711590e+02	-406.754	1.750000e-01	8.513900e+01	49
b[year]	1.220000e-01	7.600000e-02	0.002	2.070000e-01	3.800000e-02	0
λ[0]	2.644633e+146	4.581211e+146	21.937	1.057853e+147	2.281425e+146	
λ[1]	3.137637e+146	5.435226e+146	27.469	1.255055e+147	2.706721e+146	
λ[2]	3.722546e+146	6.448444e+146	34.395	1.489018e+147	3.211299e+146	
λ[3]	4.416491e+146	7.650543e+146	43.069	1.766596e+147	3.809940e+146	
λ[4]	5.239800e+146	9.076734e+146	53.929	2.095920e+147	4.520177e+146	
λ[5]	6.216587e+146	1.076879e+147	67.527	2.486635e+147	5.362814e+146	
λ[6]	7.375464e+146	1.277628e+147	84.555	2.950186e+147	6.362534e+146	
λ[7]	8.750375e+146	1.515799e+147	105.876	3.500150e+147	7.548618e+146	
λ[8]	1.038159e+147	1.798369e+147	132.440	4.152637e+147	8.955808e+146	
λ[9]	1.231690e+147	2.133616e+147	135.458	4.926759e+147	1.062532e+147	
λ[10]	1.461298e+147	2.531358e+147	135.780	5.845190e+147	1.260606e+147	
λ[11]	1.733708e+147	3.003246e+147	136.104	6.934832e+147	1.495604e+147	
λ[12]	2.056901e+147	3.563102e+147	136.428	8.227603e+147	1.774410e+147	

Sampler stats

```
1  print(post.sample_stats)
```

```
<xarray.Dataset> Size: 528kB
Dimensions:                (chain: 4, draw: 1000)
Coordinates:
  * chain                  (chain) int64 32B 0 1 2 3
  * draw                   (draw) int64 8kB 0 1 2 3 4 5 ... 995 996 997 998 999
Data variables: (12/18)
    energy                 (chain, draw) float64 32kB 4.669e+148 ... 97.63
    tree_depth             (chain, draw) int64 32kB 1 1 1 3 1 1 ... 1 5 10 10 10
    n_steps                (chain, draw) float64 32kB 1.0 1.0 ... 611.0 831.0
    perf_counter_start     (chain, draw) float64 32kB 3.702e+05 ... 3.702e+05
    diverging              (chain, draw) bool 4kB False False ... False False
    perf_counter_diff      (chain, draw) float64 32kB 2.325e-05 ... 0.009242
    ...                     ...
    largest_eigval         (chain, draw) float64 32kB nan nan nan ... nan nan
    step_size_bar          (chain, draw) float64 32kB 1.257e-93 ... 0.001635
    divergences            (chain, draw) int64 32kB 0 0 0 0 0 0 ... 0 0 0 0 0 0
    index_in_trajectory    (chain, draw) int64 32kB -1 -1 1 -3 ... -333 -362 290
    process_time_diff      (chain, draw) float64 32kB 2.4e-05 ... 0.009241
    lp                     (chain, draw) float64 32kB -4.669e+148 ... -96.2
Attributes:
```

Tree depth

```
1 post.sample_stats["tree_depth"].values
```

```
array([[ 1,  1,  1,  3,  1,  1,  4,  2,  1,  1,  4,  1,  1,  1,  5,  1,  1,  1,  1,  3,  1,  4,  1
         1,  4,  3,  1,  1,  1,  1,  1,  1,  1,  1,  2,  2,  3],
       [ 1,  5,  1,  1,  5,  5,  1,  5,  1,  4,  1,  1,  1,  1,  1,  1,  1,  2,  1,  1,  1,  1,  3
         4,  1,  1,  2,  2,  2,  1,  2,  1,  4,  1,  1,  2,  1],
       [10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10
        10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10],
       [ 8, 10, 10, 10, 10,  9, 10,  2,  2,  9,  9,  6, 10, 10,  2, 10,  4,  1, 10, 10, 10, 10,  7
         1, 10, 10,  3,  9,  7,  9,  2,  7,  1,  5, 10, 10, 10]], shape=(4, 1000))
```

```
1 post.sample_stats["reached_max_treedepth"].values
```

```
array([[False, False, False, False, False, False, False, False, False, False, False, False, False,
        False, False, False, ..., False, False, False, False, False, False, False, False, False, F
        False, False, False, False, False, False, False],
       [False, False, False, False, False, False, False, False, False, False, False, False, False,
        False, False, False, ..., False, False, False, False, False, False, False, False, False, F
        False, False, False, False, False, False, False],
       [ True,  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,  True,
         True,  True,  True, ...,  True,  True,  True,  True,  True,  True,  True,  True,  True,
         True,  True,  True,  True,  True,  True,  True],
       [False, False,  True,  True, False, False,  True, False, False, False, False, False, False,
         True, False,  True, ...,  True, False, False, False, False,  True,  True,  True, False, F
        False, False, False, False, False, False, False]], shape=(4, 1000))
```

Adjusting the sampler

```
1 with model:
2   post = pm.sample(
3     random_seed=1234,
4     step = pm.NUTS(max_treedepth=20)
5   )
```

Progress	Draws	Divergences	Step size	Grad evals	Sampling Speed
━━━━━━━━━━	1999	0	0.001	39	253.61 drawss/s
━━━━━━━━━━	1999	0	0.002	3	128.99 drawss/s
━━━━━━━━━━	1999	1	0.001	523	154.43 drawss/s
━━━━━━━━━━	1999	0	0.002	1999	136.02 drawss/s

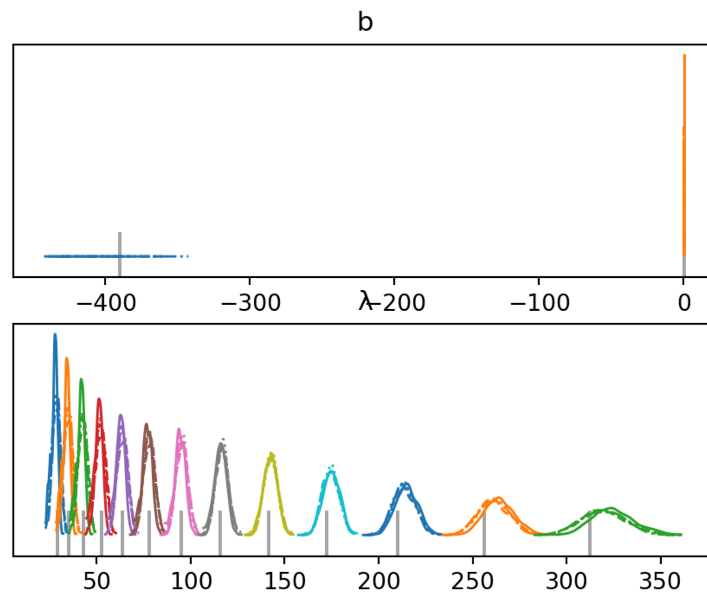
Summary

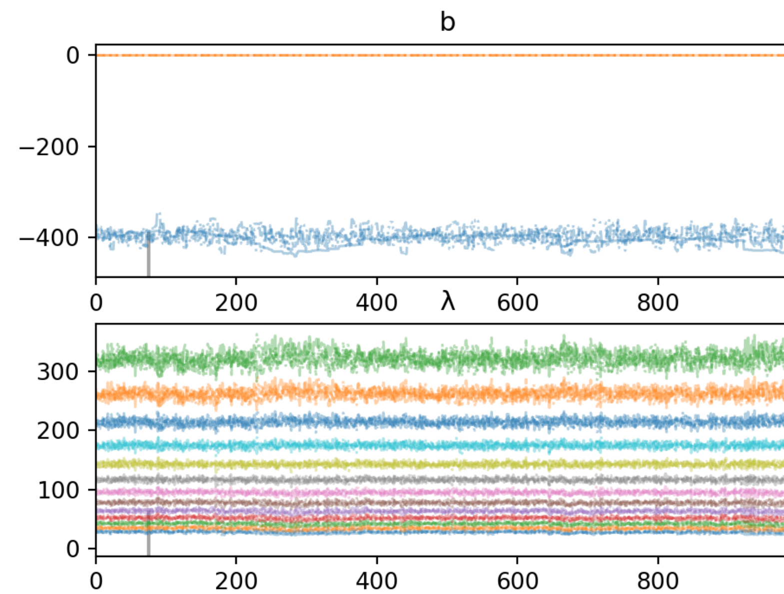
```
1  az.summary(post)
```

```
                  mean      sd    hdi_3%   hdi_97%  mcse_mean  mcse_sd  ess_bulk  ess_tai
b[Intercept]  -398.614  15.707  -429.752  -369.721      1.552    0.937     104.0      88.
b[year]          0.203   0.008     0.188     0.218      0.001    0.000     104.0      90.
λ[0]            28.180   2.023    24.344    32.020      0.185    0.098     118.0     156.
λ[1]            34.502   2.225    30.314    38.758      0.200    0.104     122.0     162.
λ[2]            42.245   2.422    37.775    46.936      0.213    0.109     128.0     184.
λ[3]            51.729   2.607    46.951    56.775      0.221    0.109     138.0     190.
λ[4]            63.345   2.773    58.221    68.665      0.221    0.102     156.0     240.
λ[5]            77.575   2.918    72.030    83.047      0.210    0.086     193.0     315.
λ[6]            95.008   3.061    89.334   100.897      0.180    0.062     293.0     550.
λ[7]           116.366   3.259   110.139   122.481      0.124    0.049     689.0    1504.
λ[8]           142.533   3.649   136.007   149.586      0.072    0.054    2559.0    2608.
λ[9]           174.596   4.452   166.243   182.497      0.086    0.063    2671.0    3080.
λ[10]          213.885   5.935   202.410   224.044      0.215    0.085     765.0    2190.
λ[11]          262.031   8.346   247.082   278.051      0.461    0.166     333.0     576.
λ[12]          321.035  11.946   298.041   343.316      0.816    0.392     220.0     293.
```

Trace plots

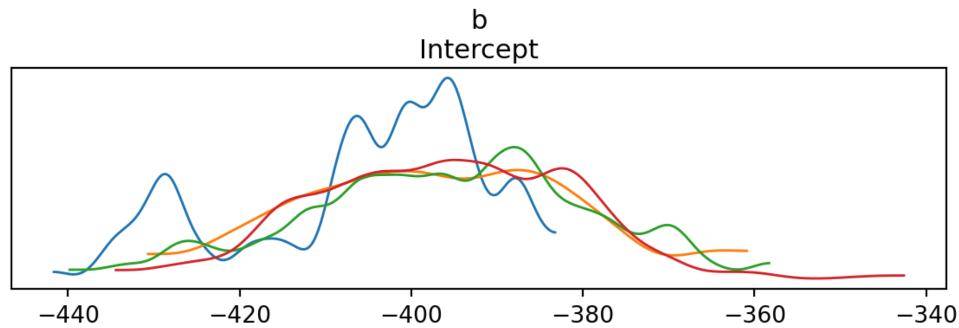
```
ax = az.plot_trace(post)
plt.show()
```

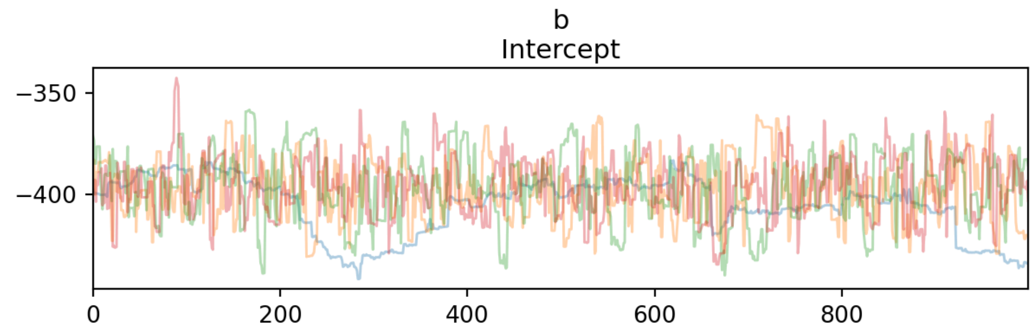


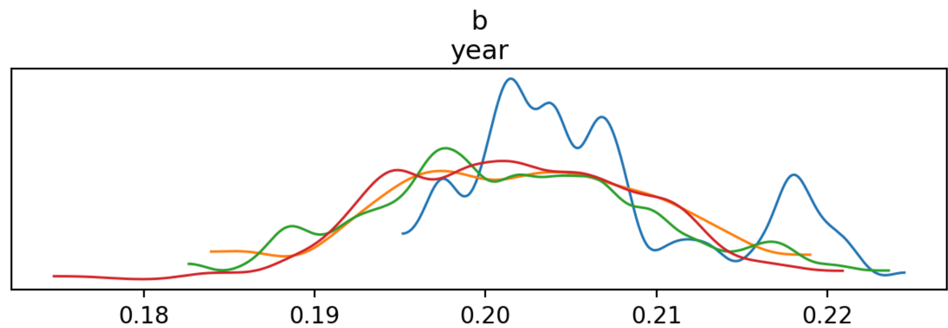


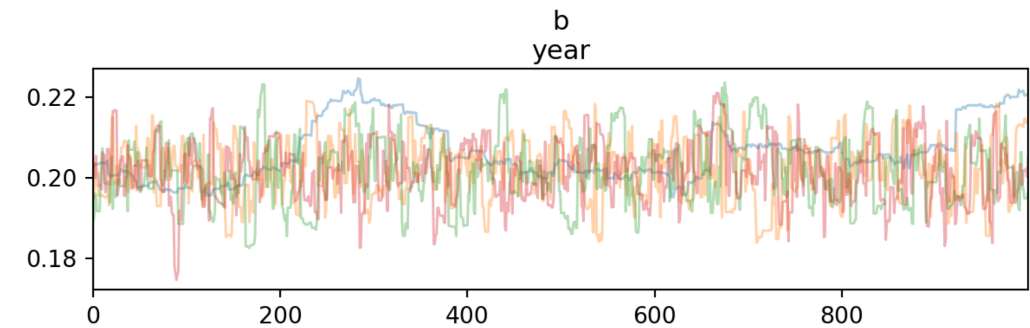
Trace plots (again)

```
1 ax = az.plot_trace(post.posterior["b"], compact=False)
2 plt.gcf().set_layout_engine("constrained")
3 plt.show()
```



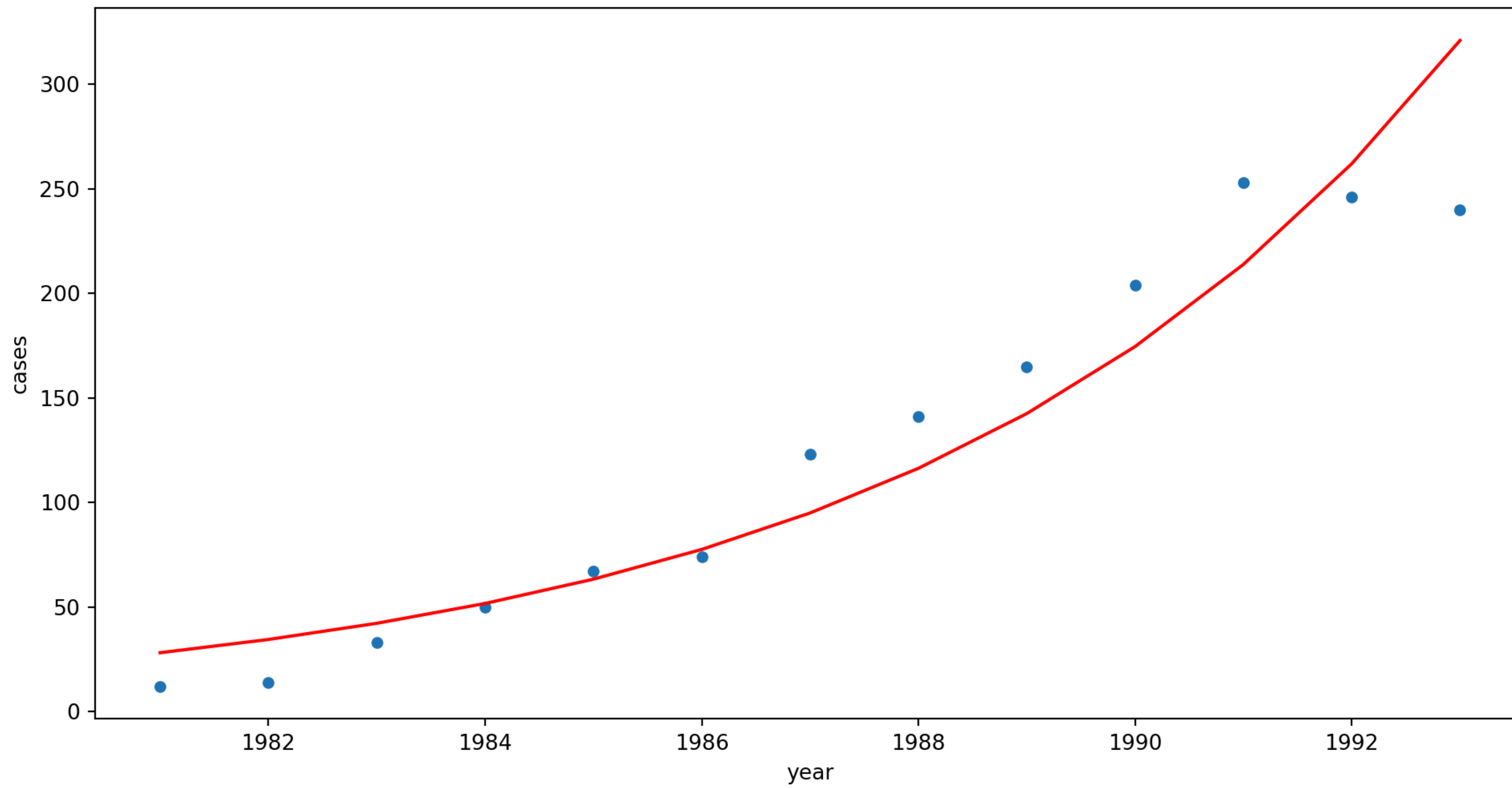






Predictions (λ)

```
plt.figure(figsize=(12,6))
sns.scatterplot(x="year", y="cases", data=aids)
sns.lineplot(
  x="year", y=post.posterior["λ"].mean(dim=["chain", "draw"]), data=aids, color='red'
)
plt.show()
```


300
250
200
150
100
50
0
cases
1982
1984
1986
1988
1990
1992
year

Revised model

```
y, X = patsy.dmatrices(
  "cases ~ year_min + I(year_min**2)",
  aids.assign(year_min = lambda x: x.year-np.min(x.year))
)

X_lab = X.design_info.column_names
y = np.asarray(y).flatten()
X = np.asarray(X)

with pm.Model(coords = {"coeffs": X_lab}) as model:
    b = pm.Cauchy("b", alpha=0, beta=1, dims="coeffs")
    η = X @ b
    λ = pm.Deterministic("λ", np.exp(η))

    likelihood = pm.Poisson("y", mu=λ, observed=y)

    post = pm.sample(random_seed=1234)
```

Progress	Draws	Divergences	Step size	Grad evals	Sampling Speed	Elapsed
━━━━━━━━	1999	0	0.090	19	2557.63 drawss/s	0:00:00
━━━━━━━━	1999	0	0.089	7	2144.12 drawss/s	0:00:00
━━━━━━━━	1999	0	0.080	25	2739.00 drawss/s	0:00:00
━━━━━━━━	1999	0	0.081	31	2280.13 drawss/s	0:00:00

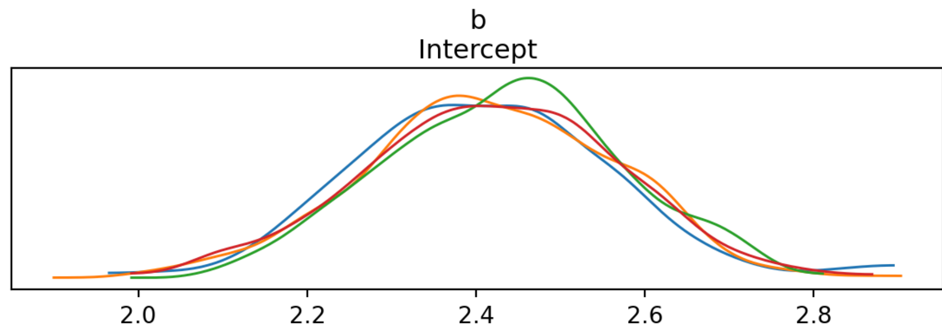
Summary

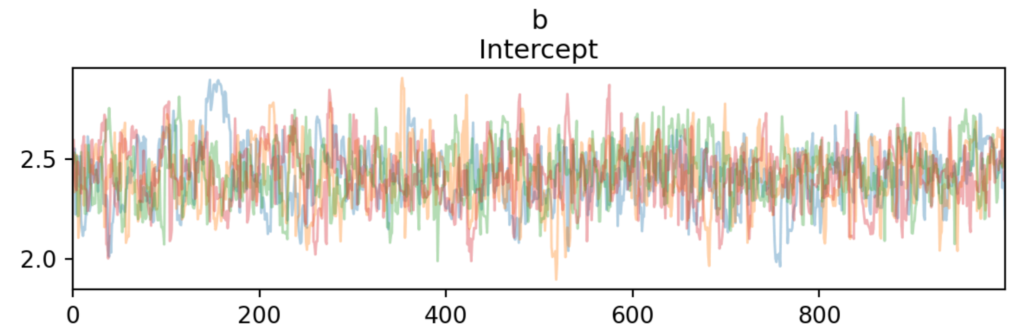
```
1  az.summary(post)
```

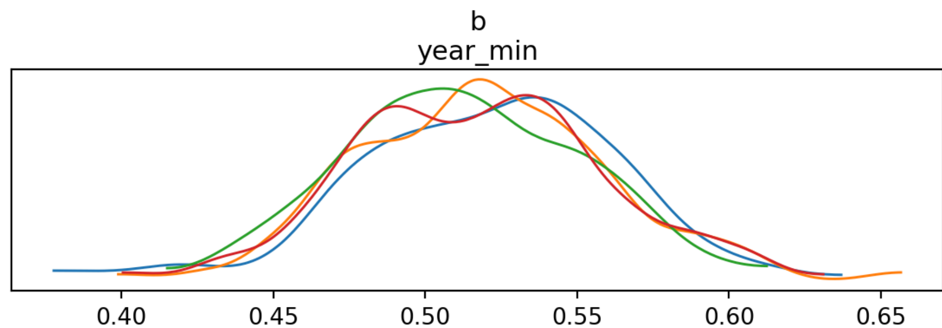
	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_
b[Intercept]	2.419	0.149	2.155	2.712	0.006	0.004	640.0	685.0	[illegible]
b[year_min]	0.517	0.041	0.442	0.595	0.002	0.001	590.0	679.0	[illegible]
b[I(year_min ** 2)]	-0.022	0.003	-0.027	-0.017	0.000	0.000	619.0	709.0	[illegible]
λ[0]	11.359	1.705	8.180	14.420	0.069	0.061	640.0	685.0	[illegible]
λ[1]	18.561	2.119	14.713	22.574	0.082	0.067	686.0	779.0	[illegible]
λ[2]	29.101	2.469	24.538	33.757	0.089	0.066	783.0	902.0	[illegible]
λ[3]	43.754	2.736	38.576	48.816	0.085	0.058	1046.0	1233.0	[illegible]
λ[4]	63.059	3.016	57.463	68.810	0.072	0.057	1755.0	1762.0	[illegible]
λ[5]	87.084	3.524	81.035	94.250	0.069	0.063	2612.0	2251.0	[illegible]
λ[6]	115.210	4.372	106.904	123.159	0.093	0.071	2191.0	2451.0	[illegible]
λ[7]	145.990	5.361	135.847	156.099	0.135	0.082	1556.0	2304.0	[illegible]
λ[8]	177.174	6.114	165.651	188.657	0.170	0.093	1285.0	2144.0	[illegible]
λ[9]	205.930	6.394	193.522	217.515	0.171	0.097	1397.0	2415.0	[illegible]
λ[10]	229.256	6.629	216.417	241.189	0.137	0.102	2379.0	2924.0	[illegible]
λ[11]	244.497	8.275	229.033	260.145	0.139	0.121	3547.0	3066.0	[illegible]
λ[12]	249.857	12.241	227.540	273.177	0.300	0.178	1667.0	2439.0	[illegible]

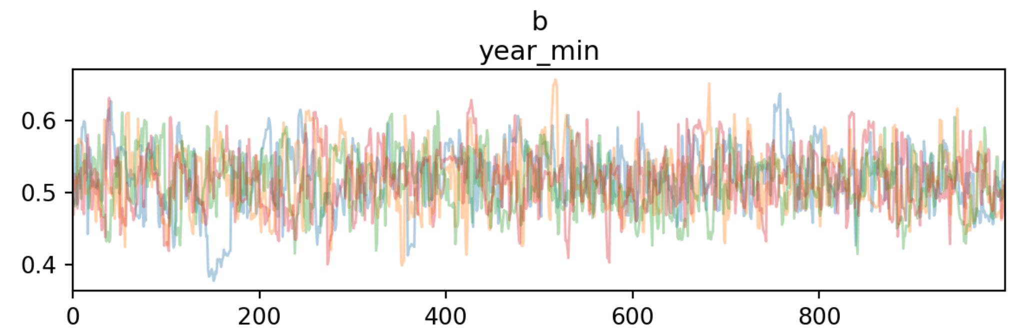
Trace plots

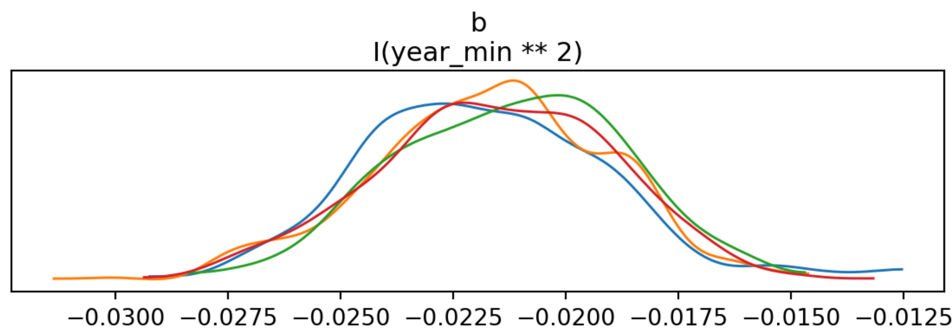
```
1 ax = az.plot_trace(post.posterior["b"], compact=False)
2 plt.gcf().set_layout_engine("constrained")
3 plt.show()
```

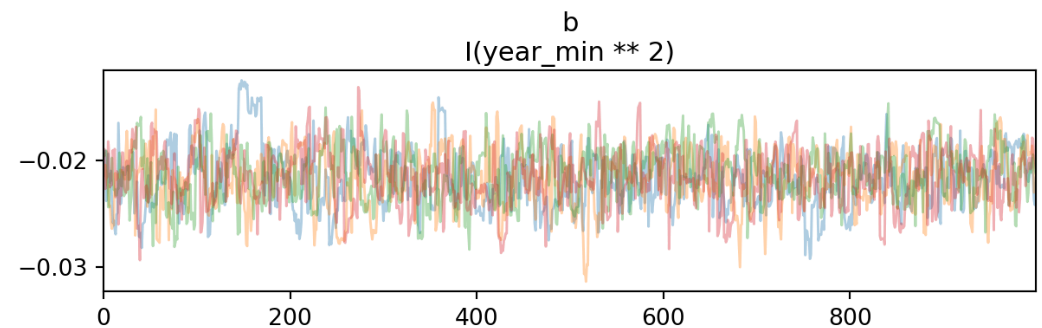






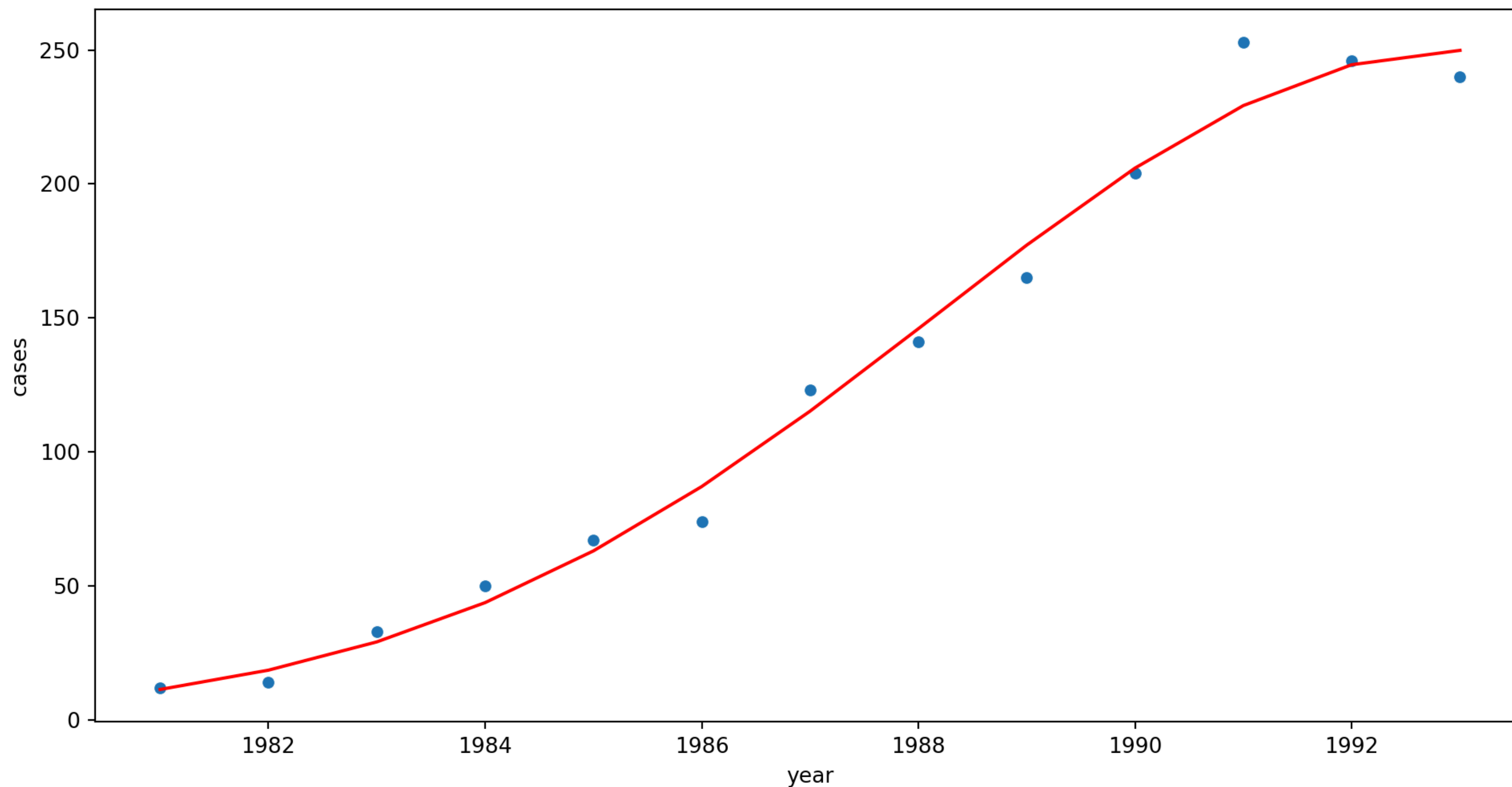






Predictions (λ)

```
1 plt.figure(figsize=(12,6))
2 sns.scatterplot(x="year", y="cases", data=aids)
3 sns.lineplot(x="year", y=post.posterior["λ"].mean(dim=["chain", "draw"]), data=aids, color='r
4 plt.show()
```

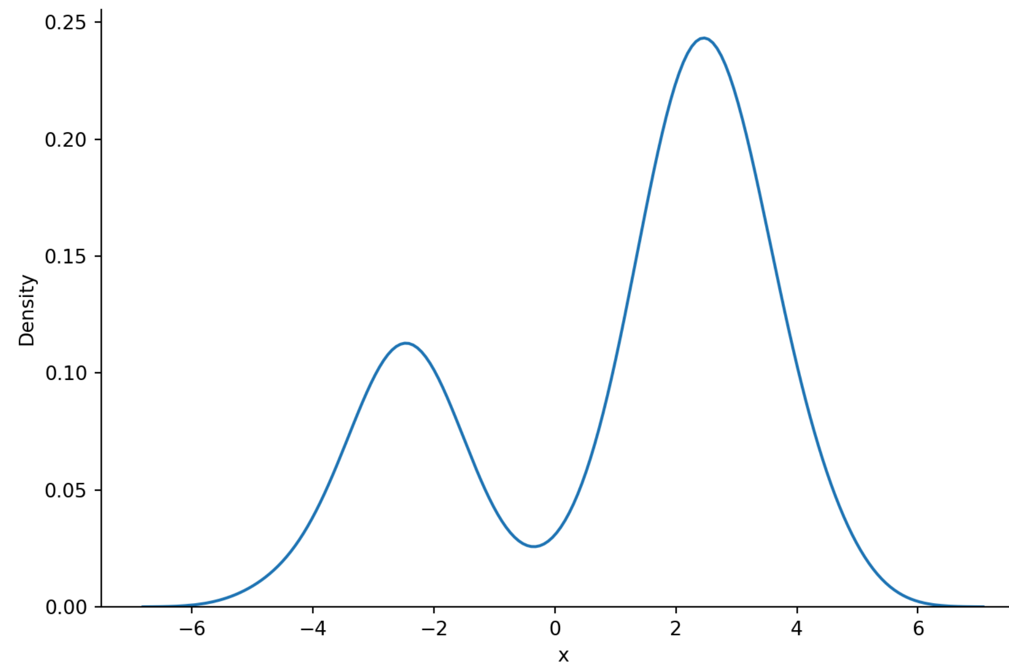


Example 3
Compound Samplers

Gaussian Mixture model

Below is a basic mixture of two Gaussians using the discrete variable `i`

```
np.random.seed(1234)
x1 = np.random.normal(-2.5, 1, size=1000)
x2 = np.random.normal( 2.5, 1, size=1000)
i = np.random.binomial(1, 0.3, size=1000)
y = np.where(i, x1, x2)
```



pymc model with a discrete parameter

```python
with pm.Model() as gmm:
    μ = pm.Normal("μ", mu=0, sigma=5, shape=2)
    σ = pm.HalfNormal("σ", sigma=3, shape=2)

    p = pm.Beta("p", 1, 1)
    i = pm.Bernoulli("i", p, shape=len(y))

    obs = pm.Normal("y", mu=μ[i], sigma=σ[i], observed=y)

    trace = pm.sample(random_seed=1234, draws=1000, chains=4)
```

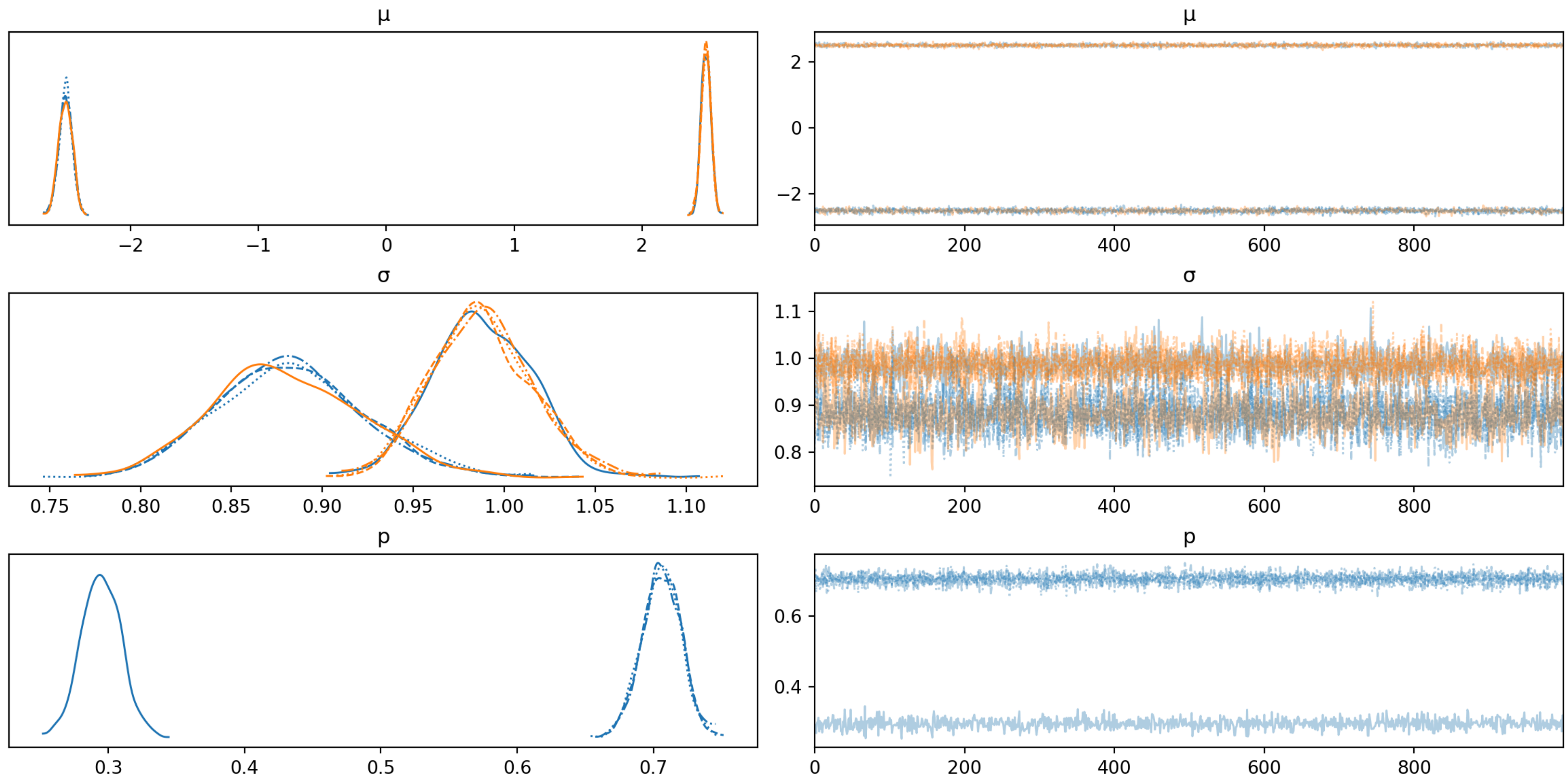
Progress	Draws	Divergences	Step size	Grad evals	Sampling Speed	Elapsed
━━━━━━━━━━	1999	0	0.987	3	58.18 drawss/s	0:00:34
━━━━━━━━━━	1999	0	1.128	3	57.42 drawss/s	0:00:34
━━━━━━━━━━	1999	0	0.779	3	57.64 drawss/s	0:00:34
━━━━━━━━━━	1999	0	1.115	3	57.51 drawss/s	0:00:34

```
/Users/rundel/Desktop/Sta663-Sp26/website/.venv/lib/python3.14/site-packages/arviz/stats/diagnost:
  rhat_value = np.sqrt(
```

```python
az.summary(trace, var_names=["~i"])
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
μ[0]	-1.255	2.169	-2.598	2.547	1.080	0.623	7.0	32.0	1.53
μ[1]	1.249	2.170	-2.560	2.575	1.080	0.624	7.0	29.0	1.53
σ[0]	0.909	0.059	0.817	1.021	0.023	0.009	8.0	37.0	1.46
σ[1]	0.961	0.056	0.841	1.043	0.023	0.013	8.0	30.0	1.48
p	0.603	0.179	0.278	0.730	0.089	0.051	7.0	28.0	1.53

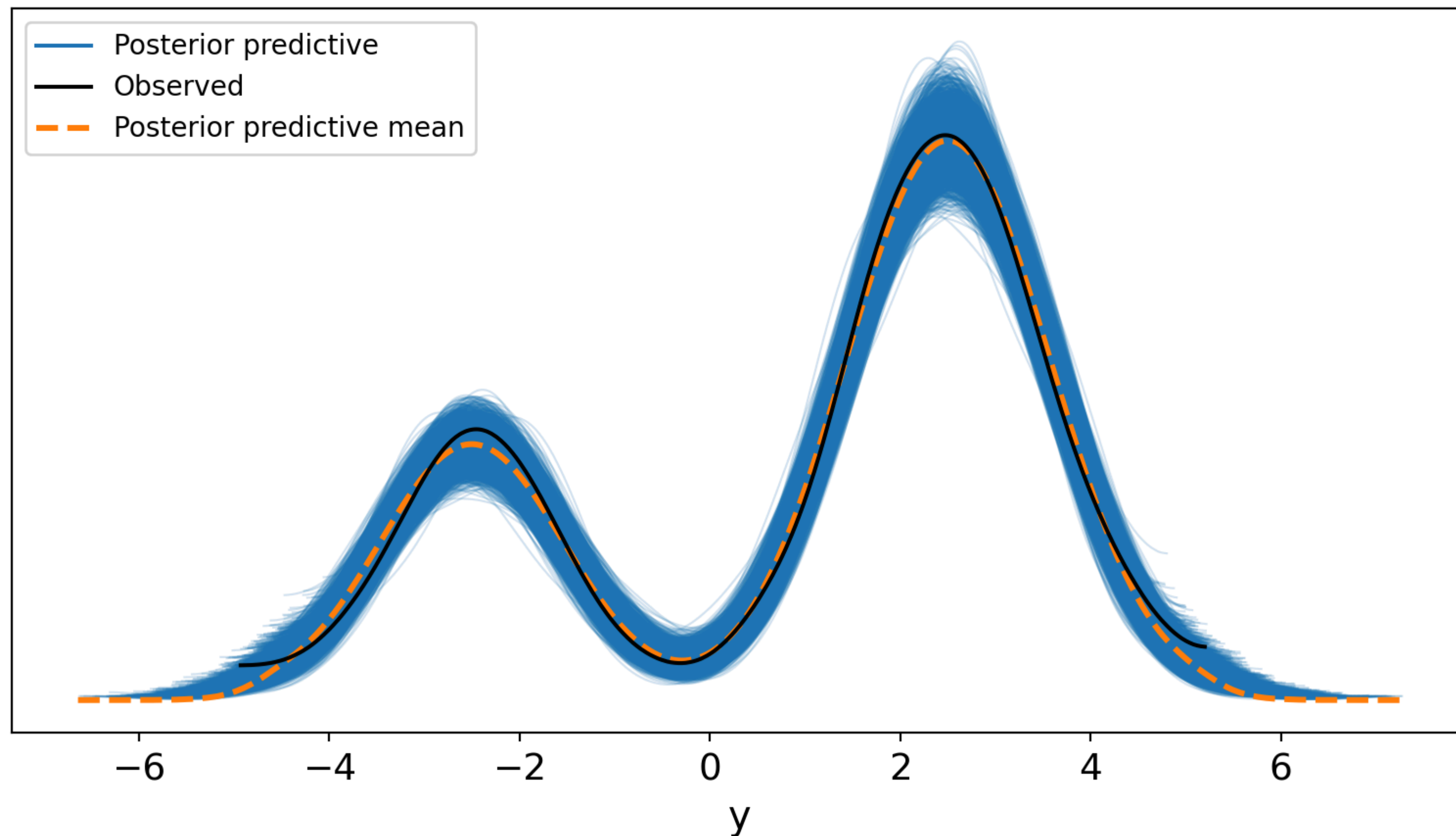
```
1 ax = az.plot_trace(trace, var_names=["~i"])
2 plt.gcf().set_layout_engine("constrained")
3 plt.show()
```



```python
with gmm:
    pp = pm.sample_posterior_predictive(trace, random_seed=1234)
```

```
Sampling ... ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 100% 0:00:00 / 0:00:00
```

```python
az.plot_ppc(pp)
```



How did that work?

Previously we mentioned that HMC methods require the gradient of the log density of the parameter of interest for the leapfrog integrator, which limits HMC samplers to continuous parameters. i is very much not a continuous parameter, so how did PyMC sample it?

While it does not show in the slides if you run the above code you will see the following message printed to the console:

```
## Multiprocess sampling (4 chains in 4 jobs)
## CompoundStep
## >NUTS: [μ, σ, p]
## >BinaryGibbsMetropolis: [i]
```

PyMC is able to use a **compound sampler** that combines NUTS for the continuous parameters and a MH sampler for the discrete parameter.

You can do this explicitly by passing a list of step methods to `pm.sample()`.

Just because you can …

… doesn't mean you should use a discrete sampler. Here we reparameterize using `pm.NormalMixture`, to marginalize out the discrete variable, keeping all parameters continuous and letting NUTS handle everything.

```
1  init_mu = np.sort(np.random.normal(size=2))
2  with pm.Model() as gmm2:
3      μ = pm.Normal(
4          "μ", mu=0, sigma=10, shape=2,
5          transform = pm.distributions.transforms.ordered,
6          initval = init_mu
7      )
8      σ = pm.HalfNormal("σ", sigma=10, shape=2)
9      weights = pm.Dirichlet("w", np.ones(2))
10
11     obs = pm.NormalMixture("y", w=weights, mu=μ, sigma=σ, observed=y)
12     trace = pm.sample(random_seed=1234, draws=1000)
```

Progress	Draws	Divergences	Step size	Grad evals	Sampling Speed	Elapsed
━━━━━━━━━━	1999	0	0.607	7	1334.03 drawss/s	0:00:01
━━━━━━━━━━	1999	0	0.631	7	1327.24 drawss/s	0:00:01
━━━━━━━━━━	1999	0	0.595	3	1327.08 drawss/s	0:00:01
━━━━━━━━━━	1999	0	0.536	7	1329.49 drawss/s	0:00:01

```python
az.summary(trace)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
μ[0]	-2.509	0.053	-2.607	-2.410	0.001	0.001	3464.0	2964.0	1.0
μ[1]	2.500	0.039	2.426	2.572	0.001	0.001	4705.0	2870.0	1.0
σ[0]	0.882	0.041	0.809	0.964	0.001	0.001	5039.0	2415.0	1.0
σ[1]	0.988	0.027	0.940	1.043	0.000	0.000	4395.0	2895.0	1.0
w[0]	0.294	0.015	0.267	0.321	0.000	0.000	5127.0	2985.0	1.0
w[1]	0.706	0.015	0.679	0.733	0.000	0.000	5127.0	2985.0	1.0