Biostatistics 682: Applied Bayesian Inference
Lecture 9: PyMC for Posterior Computation

Jian Kang

Department of Biostatistics
University of Michigan, Ann Arbor

Outline

- Bridge from JAGS ⇒ PyMC (workflow mapping)
- PyMC Syntax Deep Dive (model, RVs, shapes, coords, data)
- Example: Bayesian Linear Regression (PyMC & JAGS)
- Diagnostics ($\hat{R}$, ESS, divergences, energy/BFMI) & PPC

Reference (continuity with JAGS lecture): 5-step workflow; diagnostics module.

Bridge from JAGS to PyMC

JAGS (BUGS)	PyMC (Python)
Model written in BUGS language; ~ for stochastic nodes and <- for deterministic nodes.	Model defined within `with pm.Model():` context; random variables declared via `pm.Normal`, `pm.Bernoulli`, etc.; deterministic transformations via `pm.Deterministic`.
Data supplied from R as a list through `R2jags`.	Data provided directly as NumPy or Pandas objects; dynamic updating supported with `pm.MutableData`.
Posterior sampling via Gibbs or Metropolis–Hastings using `jags()`.	Posterior sampling via `pm.sample()` using NUTS (Hamiltonian Monte Carlo) for continuous parameters; Metropolis or Gibbs for discrete ones.
Diagnostics produced in R: trace plots, ACF plots, $\hat{R}$.	Diagnostics and visualization handled through `ArviZ`: trace plots, forest plots, ESS, $\hat{R}$, divergences, energy/BFMI, posterior predictive checks (PPC).

Workflow comparisons:

JAGS: Model → Data → Initial values / Parameters → Sampling → Diagnostics

PyMC: Model → Data (arrays or `MutableData`) → `pm.sample()` → ArviZ diagnostics & PPC

PyMC Distributions: Overview

Random variables (RVs) in PyMC represent model parameters and latent or observed quantities. Each RV is created by calling a distribution constructor within the model context.

Syntax pattern:

```
param = pm.DistributionName("param_name", <parameters>, shape=...)
```

Common continuous distributions:

PyMC function	Mathematical form	Typical use
`pm.Normal(mu, sigma)`	$\mathcal{N}(\mu, \sigma^2)$	regression coeffs, priors
`pm.HalfNormal(sigma)`	$\lvert Z\rvert,\ Z\sim\mathcal{N}(0, \sigma^2)$	positive scale params ($\sigma > 0$)
`pm.Uniform(lower, upper)`	Uniform(a, b)	weak priors, bounded params
`pm.Exponential(lam)`	Exp(λ)	positive rates
`pm.Gamma(alpha,beta)`	Gamma(α, β)	positive scale, variance priors
`pm.InverseGamma(alpha,beta)`	IG(α, β)	variance priors (classical)
`pm.StudentT(nu, mu, sigma)`	$t_\nu(\mu, \sigma)$	heavy-tailed errors

Syntax reminder: PyMC uses **keyword arguments** (e.g., `mu=`, `sigma=`) rather than positional parameters.

PyMC Distributions: Discrete & Derived Types

Common discrete distributions:

PyMC function	Mathematical form	Typical use
`pm.Bernoulli(p)`	Bernoulli(p)	binary outcomes (0/1)
`pm.Binomial(n,p)`	Binomial(n, p)	counts of successes
`pm.Poisson(mu)`	Poisson(μ)	event counts
`pm.Categorical(p)`	categorical w/ prob vector p	multinomial choices
`pm.Dirichlet(alpha)`	Dirichlet($\boldsymbol{\alpha}$)	simplex-valued probabilities

Deterministic nodes: Functions of other RVs stored for reporting or plotting.

```
eta = alpha + pm.math.dot(X, beta)
p   = pm.Deterministic("p", pm.math.sigmoid(eta))
```

Observed variables: Attach data using the `observed=` argument.

```
y = pm.Normal("y", mu=mu, sigma=sigma, observed=y_obs)
```

Takeaway: PyMC provides a broad library of continuous, discrete, and hierarchical-friendly distributions that map directly to standard Bayesian model components.

Further reading: For a full list of available distributions and parameterizations, see

`https://www.pymc.io/projects/docs/en/stable/api/distributions.html`

Math Operators in PyMC

PyMC builds models using symbolic math operations provided by `pm.math`, a wrapper around `aesara.tensor` (the backend computational graph).

Basic arithmetic:

```
x = pm.Normal("x", 0, 1)
y = pm.Normal("y", 0, 1)
z = x + 2*y                  # addition, scalar operations
```

Elementwise functions:

```
pm.math.exp(x)               # e^x
pm.math.log(x)               # log(x)
pm.math.sqrt(x)
pm.math.abs(x)
pm.math.sin(x), pm.math.cos(x)
```

Special functions:

```
pm.math.sigmoid(x)           # 1 / (1 + exp(-x))
pm.math.softmax(X)           # normalize to probabilities
pm.math.switch(cond, a, b)   # elementwise conditional (if-else)
```

Note: These are symbolic operations (not NumPy); they build a computational graph for automatic differentiation and sampling.

Linear Algebra in PyMC (Core Operations)

PyMC uses `pm.math` (Aesara backend) for symbolic linear algebra. Always use `pm.math` inside the model so that gradients are available to NUTS.

Matrix–vector product:

```
X = np.random.randn(n, p)               # design matrix
beta = pm.Normal("beta", 0, 5, shape=p) # coefficients (p,)
mu   = pm.math.dot(X, beta)             # (n x p) dot (p,) -> (n,)
```

Basic vector/matrix ops:

```
pm.math.dot(a, b)     # dot or matrix product
pm.math.sum(x, axis=0)
pm.math.mean(x, axis=1)
pm.math.square(x)
```

Broadcasting: Automatically expands dimensions when shapes align:

```
alpha = pm.Normal("alpha", 0, 10)
mu = alpha + pm.math.dot(X, beta)  # scalar alpha + vector (n,)
```

Tip: Vectorization avoids Python loops → faster symbolic graph and sampling.

Advanced Matrix Operations in PyMC

PyMC supports most NumPy-like linear algebra operations through `pm.math` or `aesara.tensor`.

Matrix–matrix and transpose:

```
A = pm.Normal("A", 0, 1, shape=(p, p))
B = pm.Normal("B", 0, 1, shape=(p, p))
C = pm.math.dot(A, B)              # matrix product
At = pm.math.transpose(A)          # transpose
```

Stacking and concatenation:

```
Z = pm.math.concatenate([A, B], axis=1)  # join along columns
S = pm.math.stack([A, B], axis=0)        # stack along a new axis
```

Tensordot (general contraction):

```
# Example: batch design matrices X[g, n, p]
# and group-specific beta[g, p]
mu_g = pm.math.tensordot(X, beta, axes=([2],[1]))  # -> (g, n)
```

Numerical stability tips:

- Prefer `solve()` or `cholesky()` instead of matrix inverse.
- Keep matrices well-conditioned for HMC/NUTS.

Further reading: `PyMC Math API`

PyMC Syntax: Model, RVs, Shapes

Model context:

```
import pymc as pm, numpy as np, arviz as az
with pm.Model() as model:
    ...
```

Random variables (RVs):

```
alpha = pm.Normal("alpha", mu=0, sigma=10)
beta  = pm.Normal("beta",  mu=0, sigma=5, shape=p)   # vector of
    length p
sigma = pm.HalfNormal("sigma", sigma=1)
```

Vectorized linear predictor & broadcasting:

```
mu = alpha + pm.math.dot(X, beta)   # X: (n x p), beta: (p,)
y  = pm.Normal("y", mu=mu, sigma=sigma, observed=y_obs)
```

Deterministics (store transforms for reporting):

```
eta = alpha + pm.math.dot(X, beta)
p   = pm.Deterministic("p", pm.math.sigmoid(eta))   # logistic link
```

PyMC Syntax: Sampling, InferenceData, Coords

Sampling with pm.sample():

```
with model:
    idata = pm.sample(draws=2000, tune=1000, chains=4,
                      target_accept=0.9, random_seed=682)
```

InferenceData (xarray) via ArviZ:

```
az.summary(idata, var_names=["alpha","beta","sigma"])
az.plot_trace(idata, var_names=["alpha","beta","sigma"])
```

Posterior predictive:

```
with model:
    ppc = pm.sample_posterior_predictive(idata, var_names=["y"])
idata_ppc = az.from_pymc(posterior_predictive=ppc, model=model)
az.plot_ppc(idata_ppc)
```

Named dims/coords for nice output:

```
coords = {"coef": np.array(feature_names)}
with pm.Model(coords=coords) as m:
    beta = pm.Normal("beta", 0, 1, dims="coef")
```

Hamiltonian Monte Carlo (HMC)

Motivation: Traditional Metropolis–Hastings can mix slowly in high dimensions. HMC improves exploration by introducing **momentum variables** and simulating a physical system.

Key idea:

- Treat parameters $\boldsymbol{\theta}$ as particle positions and introduce momenta $\mathbf{r}$.
- Define Hamiltonian energy: $H(\boldsymbol{\theta}, \mathbf{r}) = U(\boldsymbol{\theta}) + K(\mathbf{r})$, where potential energy $U(\boldsymbol{\theta}) = -\log \pi(\boldsymbol{\theta} \mid y)$ and kinetic energy $K(\mathbf{r}) = \frac{1}{2}\mathbf{r}^\top \mathbf{M}^{-1}\mathbf{r}$.
- Simulate Hamilton's equations:

$$\frac{d\boldsymbol{\theta}}{dt} = \nabla_{\mathbf{r}} H = \mathbf{M}^{-1}\mathbf{r}, \quad \frac{d\mathbf{r}}{dt} = -\nabla_{\boldsymbol{\theta}} H = \nabla_{\boldsymbol{\theta}} \log \pi(\boldsymbol{\theta} \mid y).$$

Algorithm outline:

1. Draw initial momentum $\mathbf{r} \sim \mathcal{N}(0, \mathbf{M})$.
2. Simulate dynamics using a leapfrog integrator for L steps with step size ϵ.
3. Accept/reject proposed $(\boldsymbol{\theta}', \mathbf{r}')$ by Metropolis rule using ΔH.

Advantages: Efficient exploration with long-distance moves; fewer random-walk steps.

No-U-Turn Sampler (NUTS)

Motivation: HMC requires tuning of step size ϵ and number of leapfrog steps L. Choosing them poorly leads to either random-walk behavior (too small L) or wasted computation (too large L).

NUTS (Hoffman & Gelman, 2014):

- Automatically selects L by building a binary tree of leapfrog steps.
- Expands trajectory in both directions until a "U-turn" is detected:

  $$(\boldsymbol{\theta}^+ - \boldsymbol{\theta}^-)^\top \mathbf{r}^+ < 0,$$

  meaning further integration would start reversing direction.
- Performs slice sampling along the trajectory to choose the next state.
- Step size ϵ adapted during warm-up using dual averaging.
- Mass matrix $\mathbf{M}$ (covariance preconditioning) adapted for scaling of parameters.

In PyMC:

- Default sampler for continuous models: `pm.sample()` uses NUTS.
- Tuning phase ⇒ automatic step size and mass matrix adaptation.
- Usually gives well-mixed chains without manual tuning.

Target Acceptance Rate in NUTS

Acceptance probability:

$$\alpha = \min\left(1, e^{-\Delta H}\right), \quad \Delta H = H(\boldsymbol{\theta}', \mathbf{r}') - H(\boldsymbol{\theta}, \mathbf{r})$$

Goal: Adjust the step size ϵ so that the long-run acceptance rate $\mathbb{E}[\alpha]$ approaches the `target_accept` value.

Typical choices:

Target acceptance rate	Behavior	Effect
< 0.7	Large step sizes	Fast but may cause **divergences**
0.8–0.9	Balanced (default)	Stable, efficient sampling
> 0.95	Very small step sizes	Safe but slow exploration

In PyMC:

```
idata = pm.sample(target_accept=0.9, chains=4)
```

Guidelines:

- Start with the default (0.8–0.9).
- If you observe divergences ⇒ increase to 0.9–0.95.
- If sampling is too slow ⇒ decrease slightly (e.g., 0.7–0.8).

The Markov-chain Monte Carlo Interactive Gallery

MCMC Demo by Chi-Feng

What is ArviZ?

ArviZ = Analysis of Results of Bayesian Inference in Python

Purpose:

- A model-agnostic library for analyzing and visualizing Bayesian inference results.
- Works seamlessly with `PyMC`, `Stan`, `NumPyro`, and `TensorFlow Probability`.
- Provides a unified data structure for storing samples: **InferenceData** (built on `xarray`).

Core functionalities:

- **Summaries:** posterior means, credible intervals, $\hat{R}$, ESS.
- **Diagnostics:** divergences, energy/BFMI, autocorrelation.
- **Visualization:** trace plots, forest plots, pair plots, rank plots.
- **Model checking:** posterior predictive checks (PPC), LOO, WAIC.

Typical usage in PyMC:

```
import arviz as az
az.summary(idata, var_names=["alpha", "beta", "sigma"])
az.plot_trace(idata)
az.plot_ppc(idata)
```

Example 1: Bayesian Linear Regression — Model

$$y_i \mid \alpha, \boldsymbol{\beta}, \sigma^2 \sim \mathcal{N}\big(\alpha + \mathbf{x}_i^\top \boldsymbol{\beta},\, \sigma^2\big),$$

$$\alpha \sim \mathcal{N}(0, 10^2), \quad \beta_j \sim \mathcal{N}(0, 5^2), \quad \sigma \sim \text{Half-Normal}(1).$$

Half-Normal prior:

- The Half-Normal distribution is defined as the distribution of $X = |Z|$ where $Z \sim \mathcal{N}(0, \sigma_0^2)$.
- Probability density function:

$$\pi(\sigma) = \frac{\sqrt{2}}{\sigma_0 \sqrt{\pi}} \exp\left(-\frac{\sigma^2}{2\sigma_0^2} \right), \quad \sigma > 0.$$

- Expected value: $\mathbb{E}[\sigma] = \sigma_0 \sqrt{\frac{2}{\pi}}$.
- Provides a weakly informative prior on scale parameters: favors small σ but allows moderate spread.

Choosing Priors for Scale Parameters

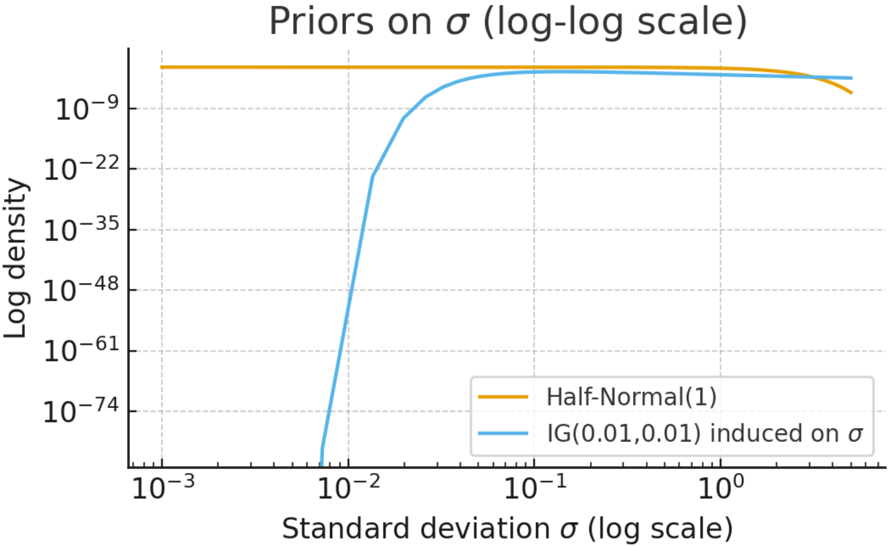
Traditional conjugate approach: Gamma prior on σ^{-2} works analytically with Gibbs sampling: $\sigma^{-2} \sim \mathrm{G}(\alpha_\sigma, \beta_\sigma)$

- Convenient for closed-form updates (Normal–Inverse-Gamma conjugacy).
- But "noninformative" choices like G$(0.001, 0.001)$ often put excessive mass near zero or too heavy right tail.
- Leads to over-shrinkage and poor mixing in gradient-based samplers.

Modern approach (recommended for HMC/NUTS):

$$\sigma \sim \text{Half-Normal}(1) \quad \text{or} \quad \text{Half-t}(\nu, 1)$$

- Places a direct prior on the **scale** (standard deviation) rather than variance.
- Easier to interpret: "typical noise is around 1 unit."
- Better numerical behavior and geometry for HMC/NUTS samplers.


Priors on σ (log-log scale)
Log density
10^{-9}
10^{-22}
10^{-35}
10^{-48}
10^{-61}
10^{-74}
10^{-3}
10^{-2}
10^{-1}
10^{0}
Standard deviation σ (log scale)
Half-Normal(1)
IG(0.01,0.01) induced on σ

PyMC syntax comparison

```
# Modern default
sigma = pm.HalfNormal("sigma", sigma=1)

# Traditional conjugate (less common in PyMC)
tau2  = pm.Gamma("tau2", alpha=0.01, beta=0.01)
sigma = pm.Deterministic("sigma", tau2**(-0.5))
```

Linear Regression — PyMC

```
with pm.Model() as linmod:
    alpha = pm.Normal("alpha", 0, 10)
    beta  = pm.Normal("beta",  0, 5, shape=X.shape[1])
    sigma = pm.HalfNormal("sigma", 1)
    mu    = alpha + pm.math.dot(X, beta)
    y     = pm.Normal("y", mu, sigma, observed=y_obs)

    idata_lin = pm.sample(target_accept=0.9, chains=4, random_seed=1)
```

Linear Regression — JAGS (general p, matching PyMC)

```
linear.reg.half.normal <- function(){
  for(i in 1:n){
    mu[i] <- alpha + inprod(beta[], X[i, ])    # X: n x p
    y[i]  ~ dnorm(mu[i], tau_eps)
  }

  # Priors (match PyMC):
  alpha ~ dnorm(0.0, 1.0/100)        # Normal(0, sd=10)
  for(j in 1:p){
    beta[j] ~ dnorm(0.0, 1.0/25)       # Normal(0, sd=5)
  }

  # Half-Normal(1) prior on sigma via truncated Normal(0,1)
  sigma  ~ dnorm(0.0, 1.0) T(0,)     # sd=1, truncated at 0
  tau_eps <- 1 / pow(sigma, 2)       # precision for likelihood
}
```

Notes: (i) `dnorm(mean, precision)`, so sd $= 10 \Rightarrow$ prec $= 1/100$; sd $= 5 \Rightarrow$ prec $= 1/25$. (ii) `T(0,)` implements a Half-Normal prior on σ. (iii) This mirrors the PyMC slide: $\alpha \sim \mathcal{N}(0, 10^2)$, $\beta_j \sim \mathcal{N}(0, 5^2)$, $\sigma \sim$ Half-Normal(1).

JAGS Run (R2jags) for the general p model

```
linear.reg.half.normal <- function(){
  for(i in 1:n){
    y[i]  ~ dnorm(alpha + inprod(beta, X[i, ]), tau_eps)
  }

  # Priors (match PyMC):
  alpha ~ dnorm(0.0, 1.0/100)       # Normal(0, sd=10)
  for(j in 1:p){
    beta[j] ~ dnorm(0.0, 1.0/25)      # Normal(0, sd=5)
  }

  # Half Normal(1) prior on sigma via truncated Normal(0,1)
  signed_sigma ~ dnorm(0.0, 1.0)           # sd=1,
  sigma <- abs(signed_sigma)
  tau_eps <- 1 / pow(sigma, 2)     # precision for likelihood
}
```

```
#Simulate y and X
n = 100, p = 10
X = matrix(rnorm(n*p),nrow=n,ncol=p)
beta = rep(c(-1,1),length=p)
alpha = 100, sigma = 10
y = as.numeric(alpha + X%*%beta + rnorm(n,sd=sigma))

# Data list (X: n x p matrix; y: length-n vector)
dat.JAGS  <- list(y = y, X = X, n = nrow(X), p = ncol(X))

# Reasonable inits
inits.JAGS <- function(){
  list(alpha = rnorm(1), beta = rnorm(dat.JAGS$p),signed_sigma = 1)
}

# Parameters to monitor
pars <- c("alpha","beta","sigma")

# Fit
fit.JAGS <- jags(data=dat.JAGS, inits=inits.JAGS,
    parameters.to.save=pars,
                 n.chains = 4, n.iter=4000, n.burnin=1000, n.thin=2,
                 model.file = linear.reg.half.normal)

print(fit.JAGS)     # summaries comparable to ArviZ az.summary()
```

MCMC Diagnostics: Linear Regression

Model setup

```
with pm.Model() as linmod:
    alpha = pm.Normal("alpha", 0, 10)
    beta  = pm.Normal("beta", 0, 5, shape=X.shape[1])
    sigma = pm.HalfNormal("sigma", 1)
    mu    = alpha + pm.math.dot(X, beta)
    yobs  = pm.Normal("y", mu, sigma, observed=y)
    idata_lin = pm.sample(1000, tune=1000, target_accept=0.9)
```

Check convergence and mixing

```
az.summary(idata_lin, var_names=["alpha","beta","sigma"])
az.plot_trace(idata_lin, var_names=["alpha","beta","sigma"])
```

- $\hat{R} \approx 1$ and large ESS $\Rightarrow$ good convergence.
- Trace plots: well-mixed chains, no long-term trends.

These ensure your posterior samples are reliable *before* evaluating model fit.

Theory of Posterior Predictive Checks (PPC)

Goal: Assess whether the model can reproduce data features seen in $\mathbf{y}$.

Posterior predictive distribution: $\pi(y^{\text{rep}} \mid y) = \int \pi(y^{\text{rep}} \mid \theta)\, \pi(\theta \mid y)\, d\theta$, where

- y: observed data,
- y^{rep}: replicated (simulated) data under the model,
- θ: parameters.

Posterior predictive check:

$$\text{Compare} \quad T(y^{\text{rep}}) \text{ vs. } T(y) \quad \text{using} \quad p_{\text{B}} = P\big\{T(y^{\text{rep}}) \geq T(y) \mid y\big\}$$

where $T(\cdot)$ is a test statistic or discrepancy measure (e.g., mean, variance, regression residuals).

Interpretation:

- If the model is well-calibrated, $T(y)$ should be typical under $\pi(y^{\text{rep}} \mid y)$.
- Extreme p_{B} values (close to 0 or 1) ⇒ model–data mismatch.

PPCs test model adequacy *without introducing new priors or parameters*. They are Bayesian analogs of classical goodness-of-fit diagnostics.

Posterior Predictive Sampling

Code example

```
with linmod:
    ppc_lin = pm.sample_posterior_predictive(
        idata_lin,
        random_seed=42,
        var_names=["y"],
        extend_inferencedata=True)
```

Step-by-step explanation

- `with linmod`: enters the fitted model context, giving access to its structure (priors, likelihood, and observed data).
- `idata_lin`: stores posterior draws $\{\theta^{(s)}\}_{s=1}^{S}$ produced by MCMC sampling.
- `pm.sample_posterior_predictive`: uses each posterior draw $\theta^{(s)}$ to generate replicated data $y^{\text{rep},(s)} \sim \pi(y \mid \theta^{(s)}, X)$ and appends these simulations to the existing `InferenceData` (`extend_inferencedata=True`).
- The resulting `idata_lin` now includes a new group `posterior_predictive` alongside the original `posterior` and `observed_data`.

Purpose: To check whether replicated outcomes y^{rep} generated by the model are consistent with the observed data y.

From PyMC to ArviZ: Plotting Posterior Predictive Checks

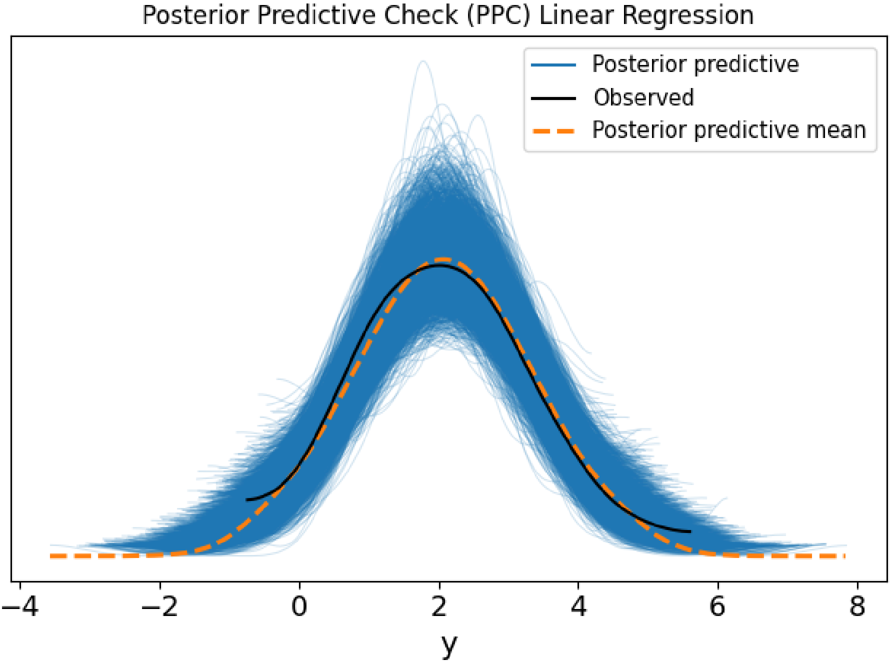
Code continuation

```
az.plot_ppc(idata_lin)
plt.title("Posterior␣Predictive␣Check␣(PPC)␣-␣Multivariate␣X")
plt.tight_layout()
plt.show()
```

What each step does

- `az.plot_ppc(...)` compares observed data to replicated samples:
  - Densities of y, y^{rep} and predictive mean
- **Visual interpretation:**
  - If y lies within the simulated range ⇒ good fit.
  - If y lies in the tails ⇒ model misfit.

Schematic: posterior draws ⇒ simulated y^{rep} ⇒ comparison with observed y.


Posterior Predictive Check (PPC) Linear Regression
Posterior predictive
Observed
Posterior predictive mean
−4
−2
0
2
4
6
8
y

Bayesian p-values for PPC

Idea: Choose a discrepancy (test) statistic $T(\cdot, \theta)$ that measures a feature of the data. Compare $T(y, \theta)$ to $T(y^{\text{rep}}, \theta)$ over posterior draws of θ.

Bayesian p-value:

$$p_B \;=\; \Pr\big[T(y^{\text{rep}}, \theta) \;\geq\; T(y, \theta) \,\big|\, y\big] \;\approx\; \frac{1}{S}\sum_{s=1}^{S} \mathbb{I}\Big\{T\big(y^{\text{rep},(s)}, \theta^{(s)}\big) \geq T\big(y, \theta^{(s)}\big)\Big\}.$$

Workflow:

1. Draw $\theta^{(s)} \sim \pi(\theta \mid y)$ (`idata.posterior`).
2. Simulate $y^{\text{rep},(s)} \sim \pi(y \mid \theta^{(s)})$ (`pm.sample_posterior_predictive`).
3. Compute $T(y, \theta^{(s)})$ and $T(y^{\text{rep},(s)}, \theta^{(s)})$.
4. Estimate p_B by the Monte Carlo proportion.

Notes: p_B is *not* a frequentist p-value; values far from 0.5 (near 0 or 1) suggest misfit for the chosen T.

Bayesian p-values: Extract Posterior & Predictive Draws

Obtain posterior samples $\theta^{(s)} = (\alpha^{(s)}, \boldsymbol{\beta}^{(s)}, \sigma^{(s)})$ and simulated replicates $y^{\text{rep},(s)}$ to evaluate discrepancy statistics.

1. Extract samples from ArviZ InferenceData:

```
post = idata_lin.posterior              # posterior draws
pp   = idata_lin.posterior_predictive    # posterior predictive draws
```

2. Observed data:

```
y_obs = idata_lin.constant_data["y"].values   # shape (n,)
```

3. Stack chains and draws into one dimension:

```
alpha_draws = post["alpha"].stack(sample=("chain","draw")).values #(S,)
beta_draws  = post["beta"].stack(sample=("chain","draw")).values #(p,S)
sigma_draws = post["sigma"].stack(sample=("chain","draw")).values #(S,)
yrep = pp["y"].stack(sample=("chain","draw")).values.T  #(S,n)
```

Notation:

- S = total posterior samples = (chains × draws).
- p = number of predictors; n = number of observations.
- Each $y^{\text{rep},(s)}$ is an n-vector simulated from $\pi(y \mid \theta^{(s)})$.

Bayesian p-values: Dimensions & Matrix Computation

Compute $\mu^{(s)} = \alpha^{(s)} + \mathbf{X}\boldsymbol{\beta}^{(s)}$ for each posterior draw to compare y^{rep} and y.

Matrix setup:

- $\mathbf{X}$: design matrix of shape (n, p)
- $\boldsymbol{\beta}^{(s)}$: coefficient vector, one per draw $\rightarrow (p, 1)$
- $\alpha^{(s)}$: scalar intercept
- $\mu^{(s)}$: fitted mean vector of length n

Vectorized computation across all draws:

```
# beta_draws: (p, S)  -> beta_draws.T: (S, p)
# X.T: (p, n)
mu = alpha_draws[:, None] + beta_draws.T @ X.T   # (S, n)
```

Shape reasoning:

Variable	Shape	Meaning
`alpha_draws[:,None]`	(S,1)	intercept per sample
`beta_draws.T`	(S,p)	coefficients per sample
`X.T`	(p,n)	transpose of design matrix
`mu`	(S,n)	predicted mean for each sample

Each row of `mu` corresponds to one posterior draw $\theta^{(s)}$, providing fitted means for all n observations.

Computing Discrepancy Statistics $T(y, \theta)$

Discrepancy 1: Residual variance

$$T(y, \theta) = \mathrm{Var}(y - \mu_\theta)$$

```
T_obs_var = ((y_obs - mu)**2).mean(axis=1)   # observed
T_rep_var = ((yrep   - mu)**2).mean(axis=1)  # replicated
p_var     = (T_rep_var >= T_obs_var).mean()
```

Discrepancy 2: Maximum absolute residual

$$T(y, \theta) = \max_i |y_i - \mu_{\theta,i}|$$

```
T_obs_max = np.abs(y_obs - mu).max(axis=1)
T_rep_max = np.abs(yrep   - mu).max(axis=1)
p_max     = (T_rep_max >= T_obs_max).mean()
```

Interpretation: $p_B \approx \Pr[T(y^{\mathrm{rep}}, \theta) \geq T(y, \theta) \mid y]$. Values near 0.5 indicate good calibration; values near 0 or 1 suggest under- or over-dispersion.

Calibration Statistic & Interpreting p_B

Discrepancy 3: Fraction above a threshold

$$T(y,\theta) = \text{fraction}(y_i > t), \quad t = \text{90th percentile of observed } y$$

```
t = np.quantile(y_obs, 0.9)
T_obs_thr = (y_obs > t).mean() * np.ones_like(alpha_draws)
T_rep_thr = (yrep  > t).mean(axis=1)
p_thr     = (T_rep_thr >= T_obs_thr).mean()
```

Reporting:

```
print(f"p_B␣(residual␣var)␣␣=␣{p_var:0.3f}")
print(f"p_B␣(max␣|resid|)␣␣␣=␣{p_max:0.3f}")
print(f"p_B␣(>90th␣quant.)␣␣=␣{p_thr:0.3f}")
```

Interpretation guidelines:

- $p_B \approx 0.5 \rightarrow$ model reproduces the feature well.
- $p_B \ll 0.1$ or $p_B \gg 0.9 \rightarrow$ systematic misfit for that discrepancy.
- Always examine several $T(\cdot)$; no single p_B is decisive.

Visual Posterior Predictive Check: Conditional Effect

Goal: Visualize the fitted regression function and its uncertainty for one covariate (X_1), while holding other covariates at their mean values.

Why:

- Examine how the model predicts y as X_1 varies, conditional on typical X_2, X_3.
- Display the posterior mean function and the 95% credible band.
- Show predictive variability across posterior draws.

Steps:

1. Extract posterior draws for $\alpha, \boldsymbol{\beta}, \sigma$.
2. Define a grid for X_1: `xg = np.linspace(min(X1), max(X1), 250)`.
3. Fix X_2, X_3 at their sample means $\bar{X}_2, \bar{X}_3$.
4. Compute $\mu(x_1) = \alpha + \beta_1 x_1 + \beta_2 \bar{X}_2 + \beta_3 \bar{X}_3$ for each posterior draw.
5. Derive the posterior mean and 95% credible interval for $\mu(x_1)$.
6. Generate a few posterior predictive draws y^{rep} to show outcome variability.

Interpretation:

- Shaded region → 95% credible band for the regression line.
- Thin lines → predictive draws y^{rep}.
- Helps assess fit, nonlinearity, and uncertainty visually.

Build Grid for X_1 and Broadcast Others

Goal: Vary X_1 over a grid; hold X_2, X_3 at their means.

```
xbar = X.mean(axis=0)                               # (p,)
xg   = np.linspace(X[:,0].min(), X[:,0].max(), 250) # (250,)
```

Construct $\mu(x_1)$ per draw:

$$\mu(x_1) = \alpha + \beta_1 x_1 + \beta_2 \bar{X}_2 + \beta_3 \bar{X}_3$$

```
mu_grid = (
    alpha_draws[:, None]             # (S,1)
  + beta_draws[[0], :].T * xg[None,:]# (S,1)*(1,250) -> (S,250)
  + beta_draws[[1], :].T * xbar[1]   # (S,1)*scalar  -> (S,1)
  + beta_draws[[2], :].T * xbar[2]   # (S,1)*scalar  -> (S,1)
)
```

Broadcasting checks:

- `alpha_draws[:,None]` makes $(S, 1)$ so it can add to $(S, 250)$.
- `beta_draws[[k],:].T` reshapes the k-th coefficient to $(S, 1)$.
- If $p > 3$: add terms $\beta_j \bar{X}_j$ similarly for all nuisance covariates.

Summaries & Predictive Draws

Credible band for the regression function:

```
lower = np.percentile(mu_grid,  2.5, axis=0)    # (250,)
upper = np.percentile(mu_grid, 97.5, axis=0)    # (250,)
mu_hat = mu_grid.mean(axis=0)                   # (250,)
```

Posterior predictive for a few sample paths:

```
rng  = np.random.default_rng(2025)
S    = mu_grid.shape[0]
keep = rng.choice(S, size=min(12, S), replace=False)    # indices of
    draws
yrep_grid = mu_grid[keep, :] \
          + rng.normal(0, sigma_draws[keep, None], size=(len(keep),
    len(xg)))
```

Why this split:

- μ-level credible band (epistemic uncertainty about the mean function).
- y^{rep} paths show outcome variability (σ) around that mean.

Plot & Interpret

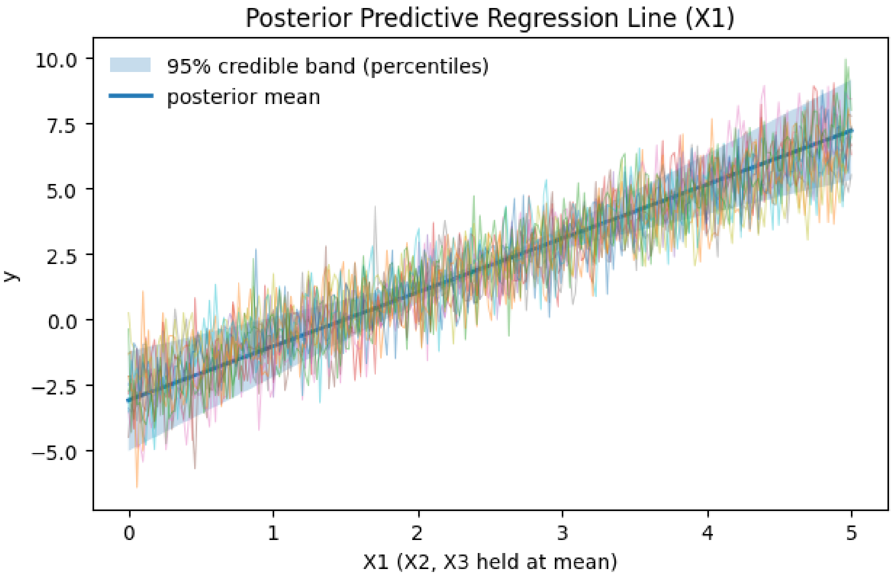
Plotting code:

```
plt.figure(figsize=(6.4,4.2))
plt.fill_between(xg, lower, upper, alpha=0.25, label="95%␣credible␣
    band")
plt.plot(xg, mu_hat, linewidth=2, label="posterior␣mean")
for k in range(yrep_grid.shape[0]):
    plt.plot(xg, yrep_grid[k], linewidth=0.7, alpha=0.45)
plt.xlabel("X1␣(X2,␣X3␣held␣at␣mean)")
plt.ylabel("y")
plt.legend(frameon=False)
plt.title("Posterior␣Predictive␣Regression␣Line␣(X1)")
plt.tight_layout()
```

Reading the figure:

- Shaded band: 95% credible interval for the regression function in x_1.
- Thick line: posterior mean function.
- Thin lines: example predictive draws y^{rep} (noise around μ).

Tips: Standardize covariates for clearer scales; repeat for each covariate or for meaningful covariate settings (e.g., quantiles).


Posterior Predictive Regression Line (X1)
95% credible band (percentiles)
posterior mean
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
y
0
1
2
3
4
5
X1 (X2, X3 held at mean)