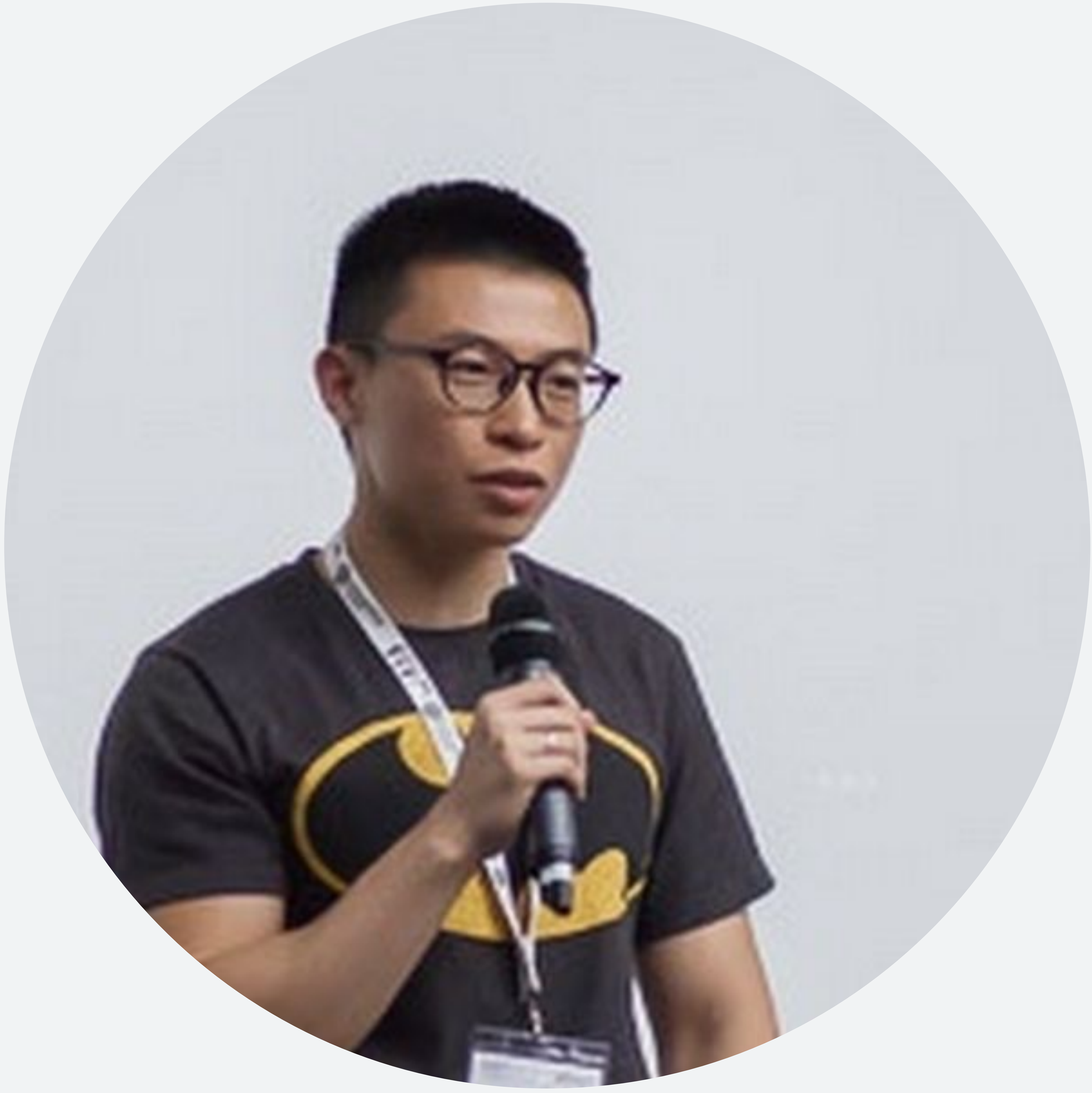
Bayesian Time Series Analysis with TFP on JAX

Junpeng Lao
June 2022

Google Research

About Me

- Psychologist turned Bayesian Statistician
- Data Scientist @ Google since 2018
  - Forecasting and time series analysis
- Open source software contributor
  - PyMC
  - Tensorflow Probability
  - BlackJAX
  - ...



Topics for today

01 Introduction to TFP on JAX

02 Time Series Analysis as Regression

03 State Space Models

04 Exercise and Wrap up

Follow along in Colab (part 1, part 2)

Material base on Chapter 6 of Bayesian Modeling and Computation in Python

Google Research

What does it even mean being Bayesian?!

To me ™:

- Gets excited talking about priors, MCMC sampling, HMC/NUTS, divergences, etc,.🤪
- Likes repeating the same thing and expecting different results 🤪
- Thinking generatively (you can simulate fake data set) 🤓
- Explicit assumptions (i.e., we spell out the Priors and Likelihood) 🧐
- Model parameters and output are represented as distributions, and we actually use the distribution (aspirational goal) 😇

01

Introduction to TFP on JAX

Google Research

JAX As Accelerated NumPy

Almost identical API to Numpy and Scipy, plus you can easily get (high order) gradient with auto differentiation; JIT, vectorization, and parallelization with great GPU and TPU support for fast computation.

```
import jax
import jax.numpy as jnp
import jax.scipy as jsp
```

Why JAX is awesome

```python
import jax
import jax.numpy as jnp

def linear_regression(beta, X): return jnp.matmul(X, beta)
def loss(beta, X, y):
  y_hat = linear_regression(beta, X)
  return jnp.mean(jnp.square(y-y_hat))
```

```python
loss_jitted = jax.jit(lambda beta: loss(beta, X, y))
%timeit loss_jitted(beta)

grads = jax.grad(loss_jitted)(beta)
```

```python
loss_vmapped = jax.vmap(lambda beta: loss(beta, X, y))
num_batch = 1000
loss_vmapped(jnp.tile(beta[None, ...], [num_batch, 1, 1]))

hessian = jax.jacfwd(jax.jacrev(loss_jitted))(beta)
```

```python
loss_pmapped = jax.pmap(lambda beta: loss(beta, X, y))

from jax.experimental import maps
from jax.experimental.pjit import pjit

mesh_shape = (4, 2)
devices = np.asarray(jax.devices()).reshape(*mesh_shape)
```

TFP: Tensor-*Friendly* Probability

State of the art library that fits all your day to day needs for Bayesian modeling and Probabilistic Deep Learning, standalone or in production.

```
import tensorflow_probability.substrates.jax as tfp

tfd = tfp.distributions
tfb = tfp.bijectors

tf = tfp.tf2jax
tfl = tfp.tf2jax.linalg
```

Example:
Linear Regression

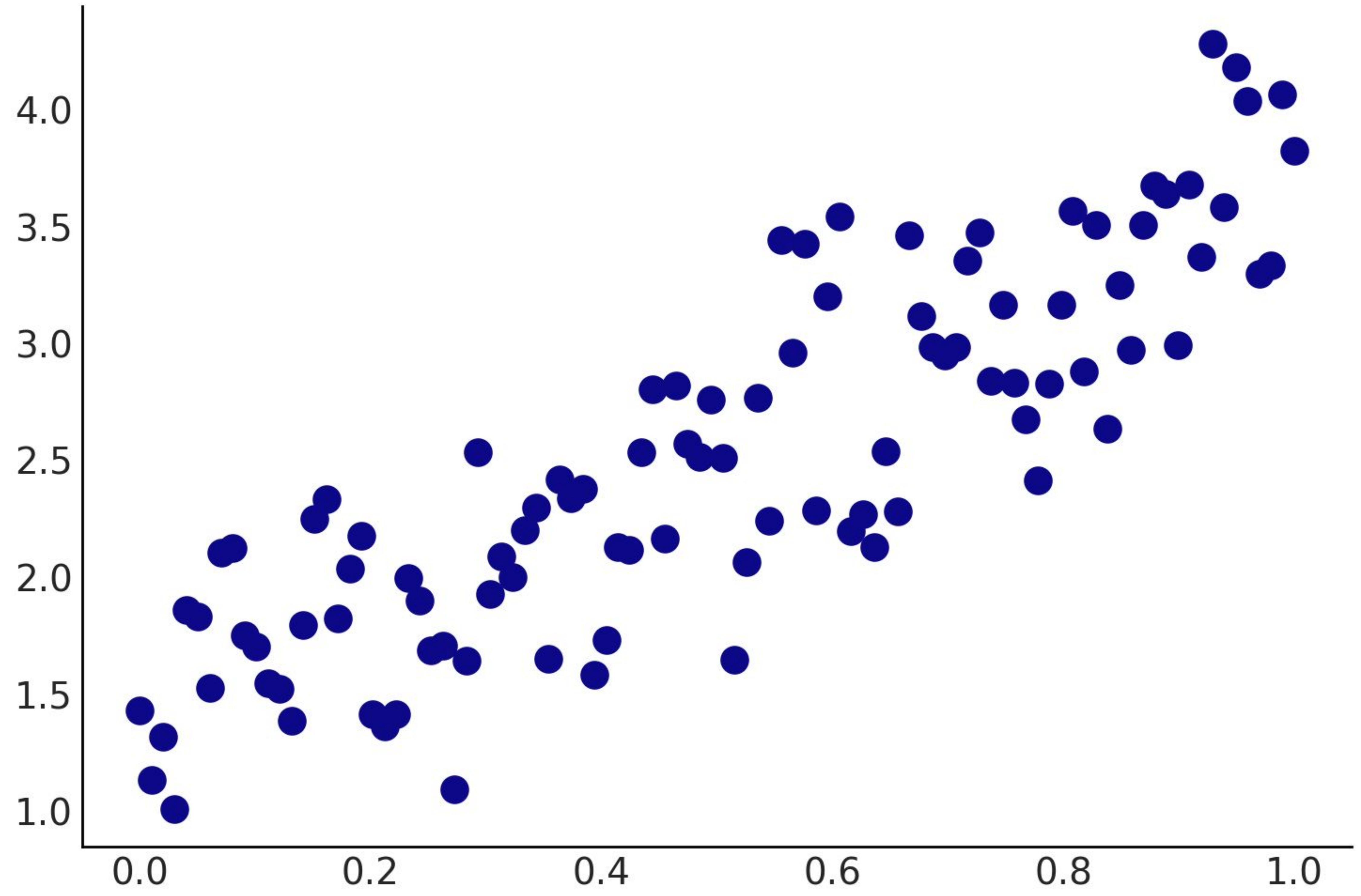
Simulate a linear regression model

```
theta0, theta1 = 1.2, 2.6
sigma = 0.4
num_timesteps = 100

x = jnp.linspace(0., 1., num_timesteps)[..., None]
yhat = theta0 + theta1 * x

rng, key = jax.random.split(rng, 2)
# y ~ Normal(yhat, σ^2)
y = tfd.Normal(yhat, sigma).sample(seed=key)
```

Note similar in terms of API compare to Numpy

Stateless Pseudo Random Numbers in JAX

TFP distribution API



Setting up the model

JAX have excellent support to (nested) Python structures, also known as Pytrees

```
from collections import namedtuple
params = namedtuple("model_params", ["w", "b", "log_sigma"])

def model(params, X):
  yhat = params.b + params.w * X
  sigma = jnp.exp(params.log_sigma)
  return tfd.Normal(yhat, sigma)

@jax.jit
def loss_fn(params, X, y):
  y_dist = model(params, X)
  return -jnp.mean(y_dist.log_prob(y))
```

Returning a tfp distribution

Jit early is encouraged!

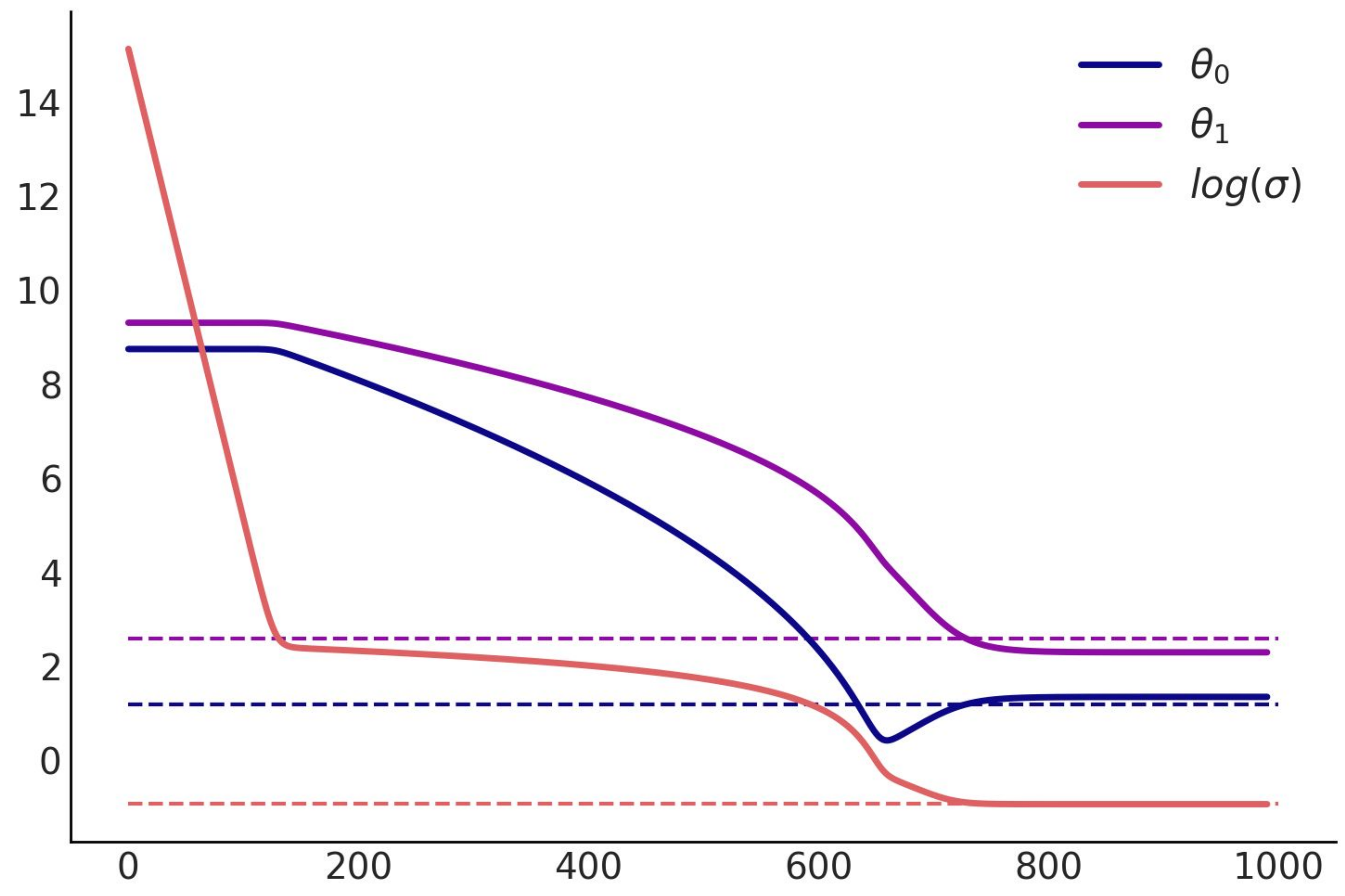
Google Research

Estimate parameter with SGD

```
@jax.jit
def update(params, X, y, lr=0.1):
  grads = jax.grad(loss_fn)(params, X, y)
  update_fn = lambda v, g: v - lr * g
  return jax.tree_map(update_fn, params, grads)

theta = params(10., 10., 0.)
hist = [theta]
for _ in range(1000):
  theta = update(theta, x, y)
  hist.append(theta)
```

Getting the gradient and perform update (SGD)

The training loop



Going Full Bayesian

```
@tfd.JointDistributionCoroutineAutoBatched
def linear_model():
  w = yield tfd.Normal(0., 10., name='w')
  b = yield tfd.Normal(0., 100., name='b')
  sigma = yield tfd.HalfNormal(5., name='sigma')
  y_hat = b + w * x
  yield tfd.Normal(y_hat, sigma, name='y')
```

Priors

Understand the output

```
rng, sample_key = jax.random.split(rng, 2)
sample = linear_model.sample(seed=sample_key)
jax.tree_map(lambda x: x.shape, sample)
```

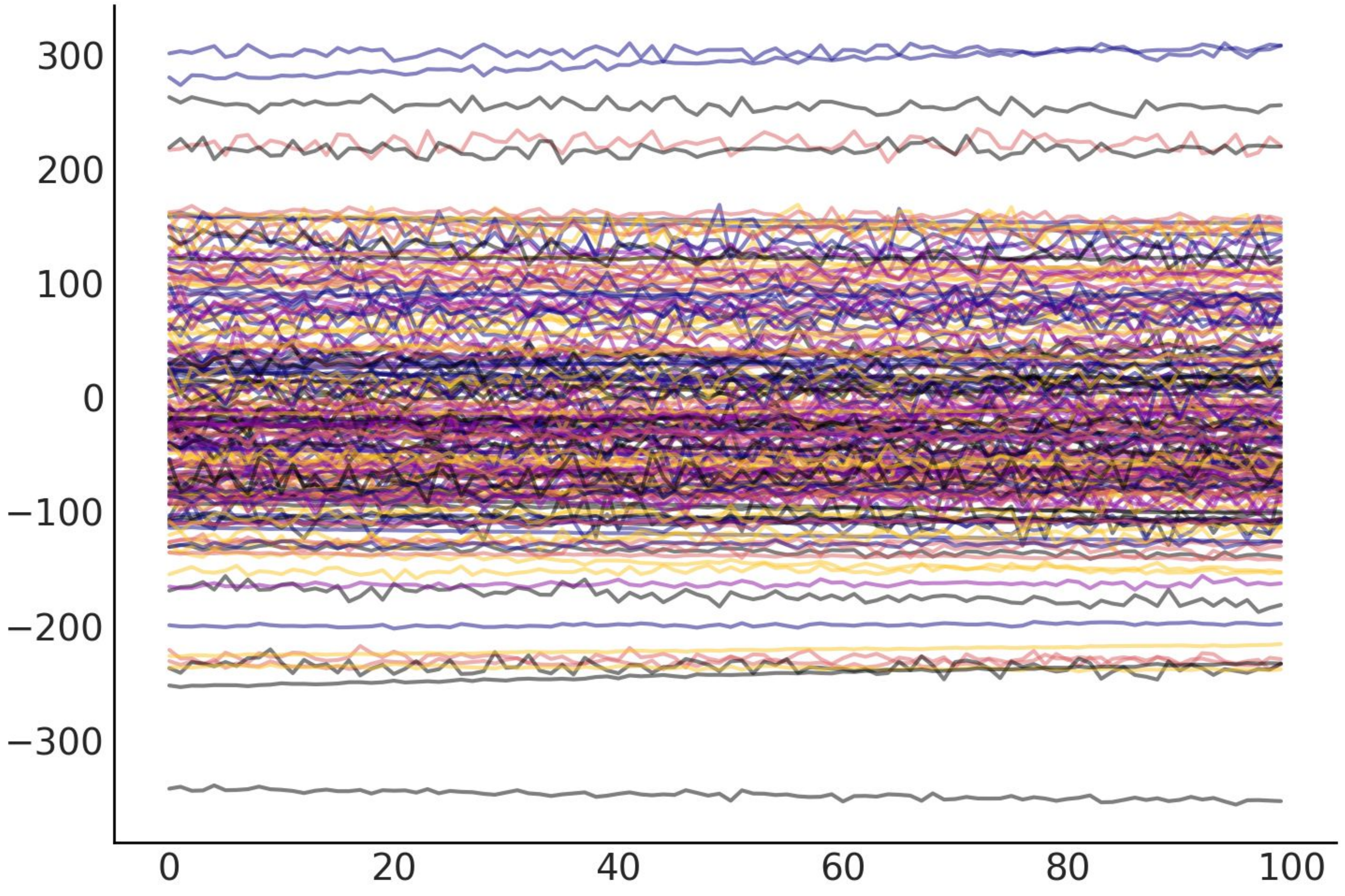
```
StructTuple(
  w=(),
  b=(),
  sigma=(),
  y=(100, 1)
)
```

```
sample = linear_model.sample(10, seed=sample_key)
jax.tree_map(lambda x: x.shape, sample)
```

```
StructTuple(
  w=(10,),
  b=(10,),
  sigma=(10,),
  y=(10, 100, 1)
)
```

Prior (Predictive) Samples

```
prior_samples = linear_model.sample(200, seed=sample_key)
prior_predictive_samples = prior_samples.y.squeeze()
plt.plot(prior_predictive_samples.T, alpha=.5);
```



Google Research

Inference with MCMC

```
n_draws = 500
n_chains = 4
run_mcmc = lambda seed: tfp.experimental.mcmc.windowed_adaptive_nuts(
   n_draws, linear_model, n_chains=n_chains, num_adaptation_steps=1000,
   seed=seed,
   y=y)
run_mcmc = jax.jit(run_mcmc)
rng, inference_key = jax.random.split(rng, 2)
mcmc_samples, sampler_stats = run_mcmc(inference_key)
```

```
CPU times: user 13.8 s, sys: 204 ms, total: 14 s
Wall time: 13.6 s
```

```
rng, inference_key = jax.random.split(rng, 2)
mcmc_samples, sampler_stats = run_mcmc(inference_key)
```

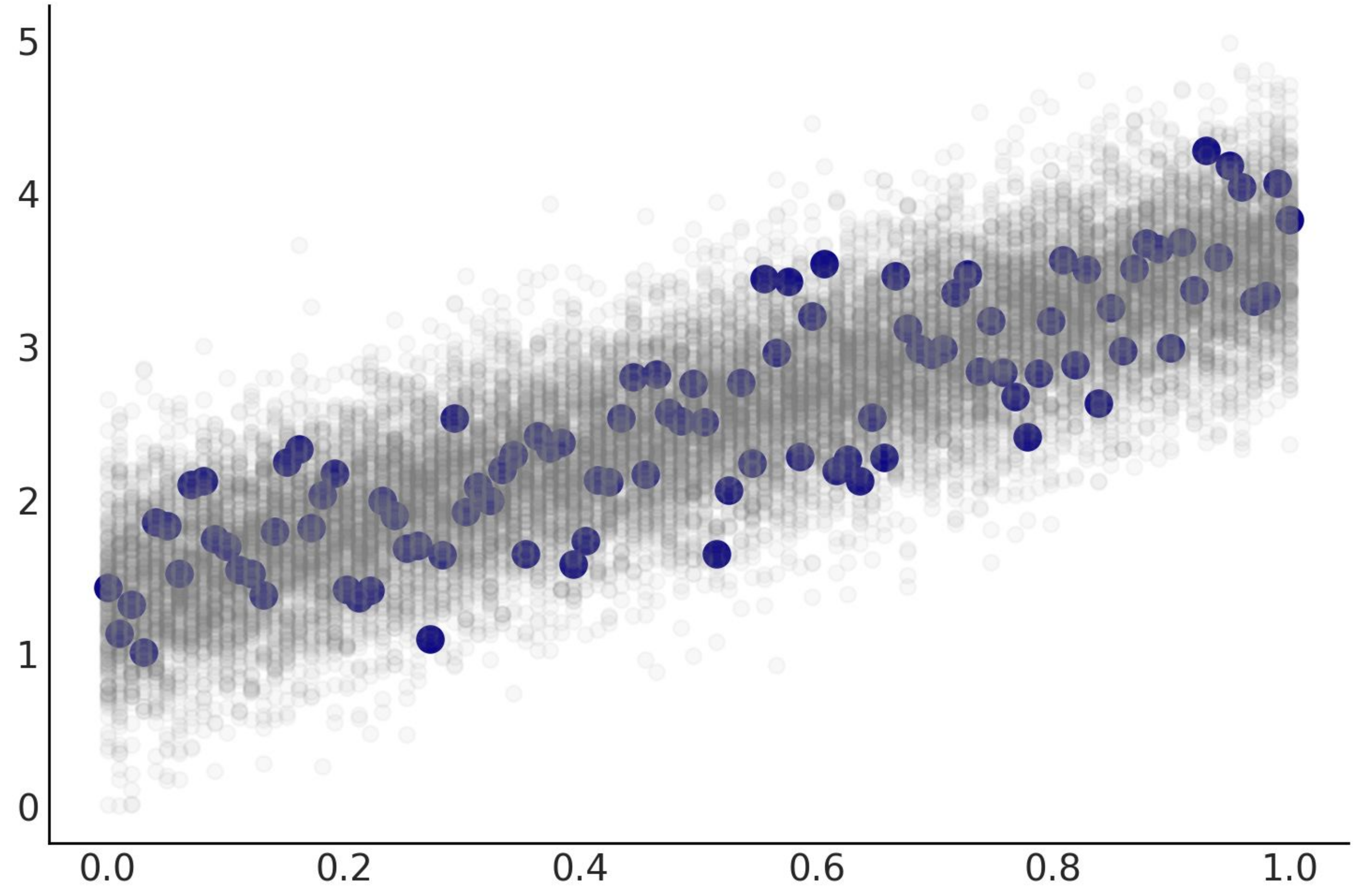
```
CPU times: user 4.29 s, sys: 4.42 ms, total: 4.29 s
Wall time: 4.41 s
```

Model diagnostic and visualization with Arviz

```
import arviz as az
regression_idata = az.from_dict(
   posterior={
        k:np.swapaxes(v, 1, 0)
        for k, v in mcmc_samples._asdict().items()},
   sample_stats={
        k:np.swapaxes(sampler_stats[k], 1, 0)
        for k in ["target_log_prob", "diverging", "accept_ratio", "n_steps"]}
)
axes = az.plot_trace(regression_idata, compact=True);
```

Posterior predictive samples

```
def draw_ppc_samples(mcmc_sample, seed):
 value = linear_model.sample(value=mcmc_sample, seed=seed)
 return value.y

rng, sample_key2 = jax.random.split(rng, 2)
sample_key2 = jax.random.split(sample_key2, n_draws * n_chains)
ppc_sample = jax.vmap(jax.vmap(draw_ppc_samples))(
   mcmc_samples,
   sample_key2.reshape(n_draws, n_chains, 2))
```



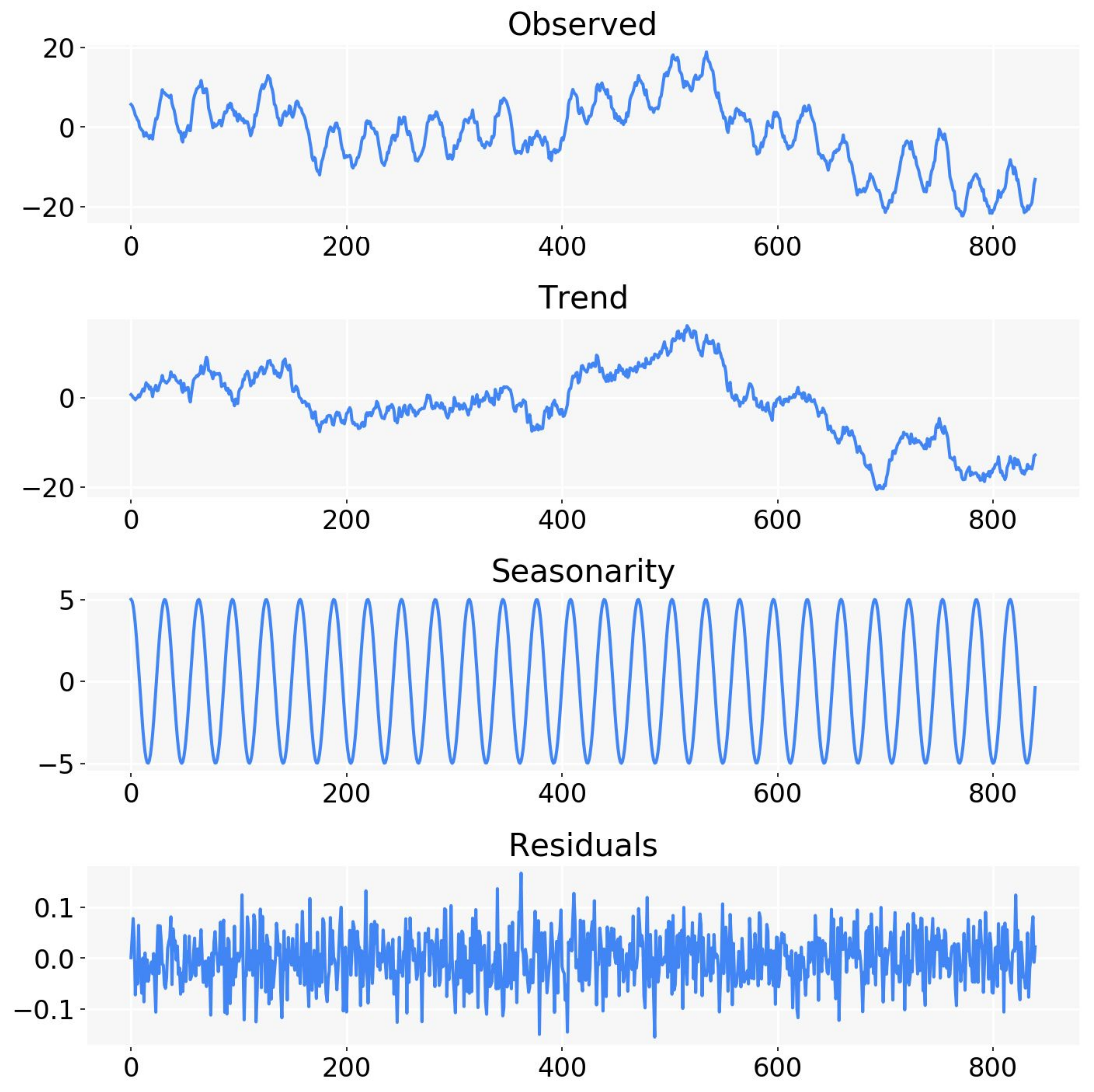
02

Time Series Analysis as Regression

Time Series Models

Classical time series models attempt to model the decomposition of:

```
Y_{t} = Trend_{t}
      + Seasonality/Holiday_{t}
      + Residuals_{t}
```

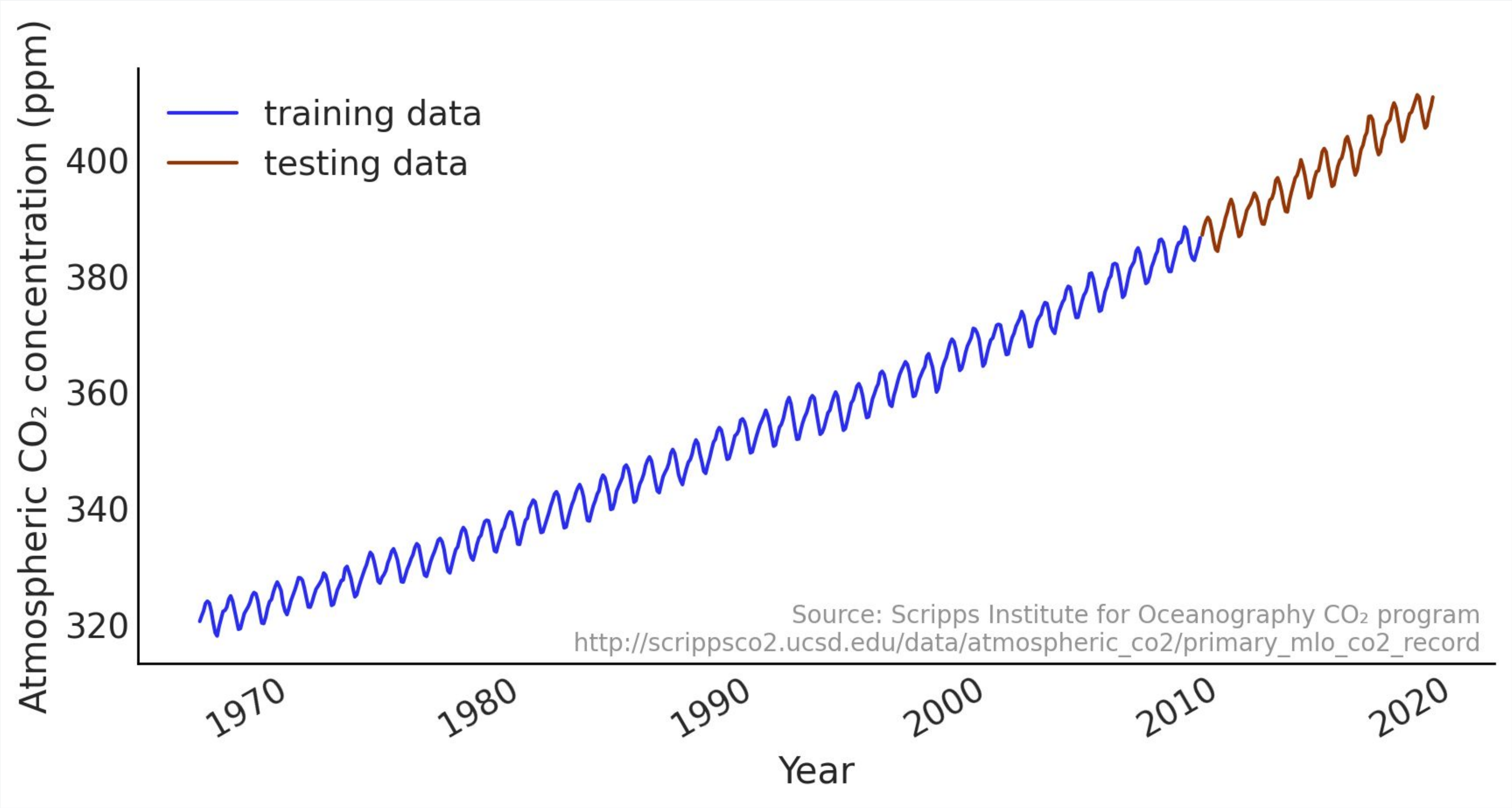


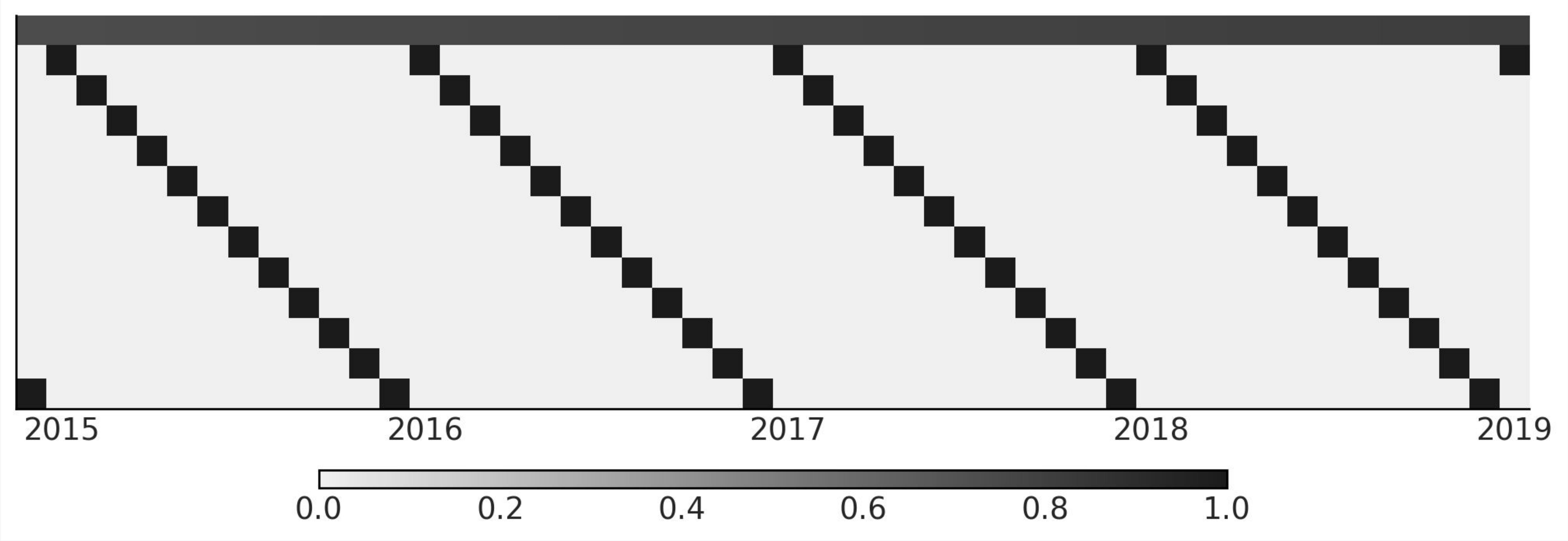
Setting up design matrix

```
Y_{t} = Trend_{t}
      + Seasonality/Holiday_{t}
      + Residuals_{t}
```

⬇

```
Y_{t} ~ Normal(X @ beta, sigma)
Y_{t} ~ Normal(\hat{Y}_t, sigma)
```





Modeling time series as a regression with a linear trend

Our model in pseudocode:

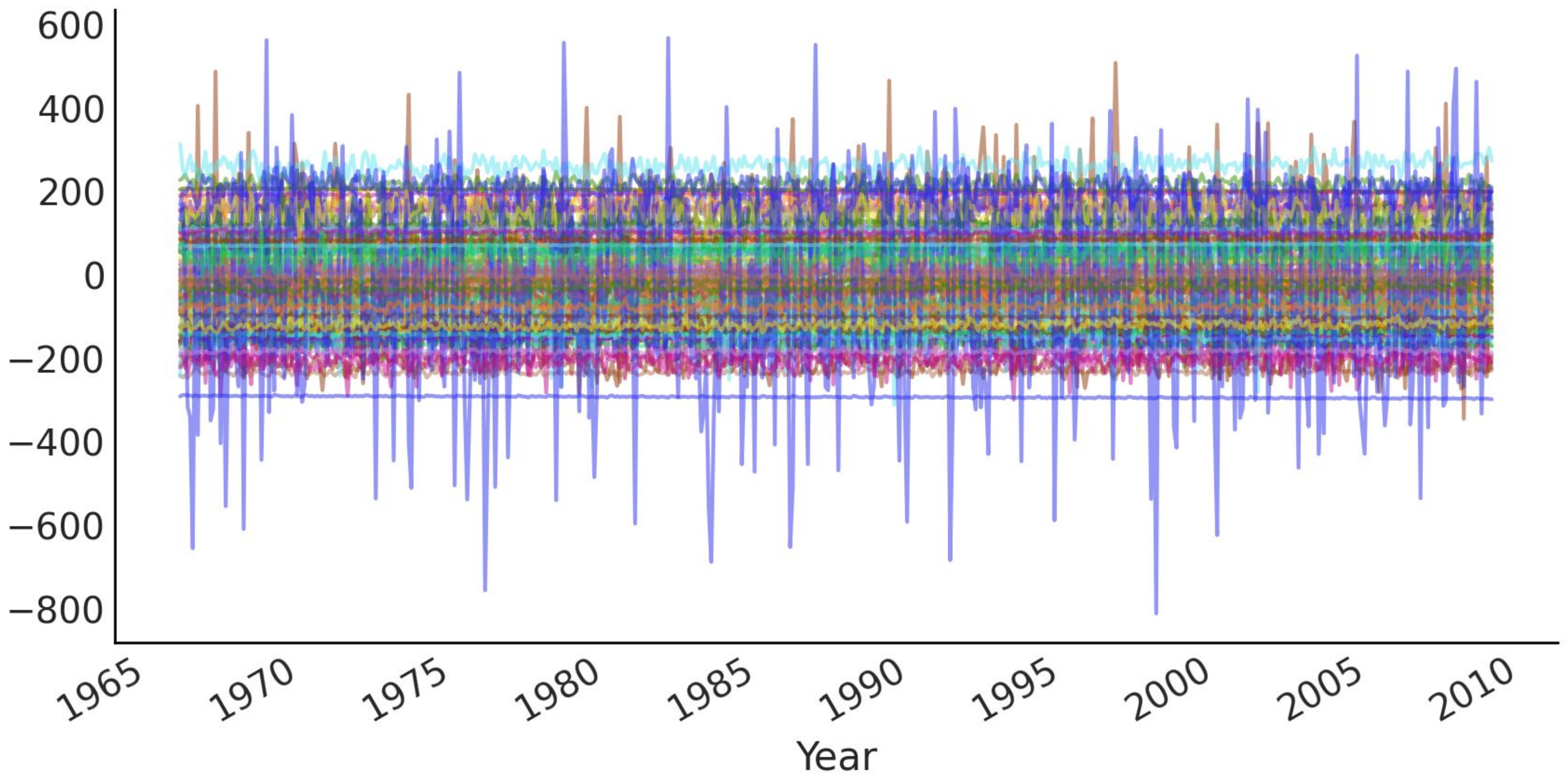
```
def ts_regression_model(...):
    intercept ~ Normal(0., 100.)
    trend_coeff ~ Normal(0., 10.)
    seasonality_coeff<12> ~ Normal(0., 1.)
    noise ~ HalfCauchy(scale=5.)

    y_hat = intercept + trend * trend_coeff + seasonality @ seasonality_coeff

    observed ~ Normal(y_hat, noise)
```

Modeling time series as a regression with a linear trend

```
@tfd.JointDistributionCoroutineAutoBatched
def ts_regression_model():
  intercept = yield tfd.Normal(0., 100., name='intercept')
  trend_coeff = yield tfd.Normal(0., 10., name='trend_coeff')
  seasonality_coeff = yield tfd.Sample(
      tfd.Normal(0., 1.),
      sample_shape=(seasonality.shape[-1], 1),
      name='seasonality_coeff')
  noise = yield tfd.HalfCauchy(loc=0., scale=5., name='noise_sigma')
  y_hat = intercept + trend * trend_coeff + seasonality @ seasonality_coeff
  observed = yield tfd.Normal(y_hat, noise[..., None], name='observed')
```

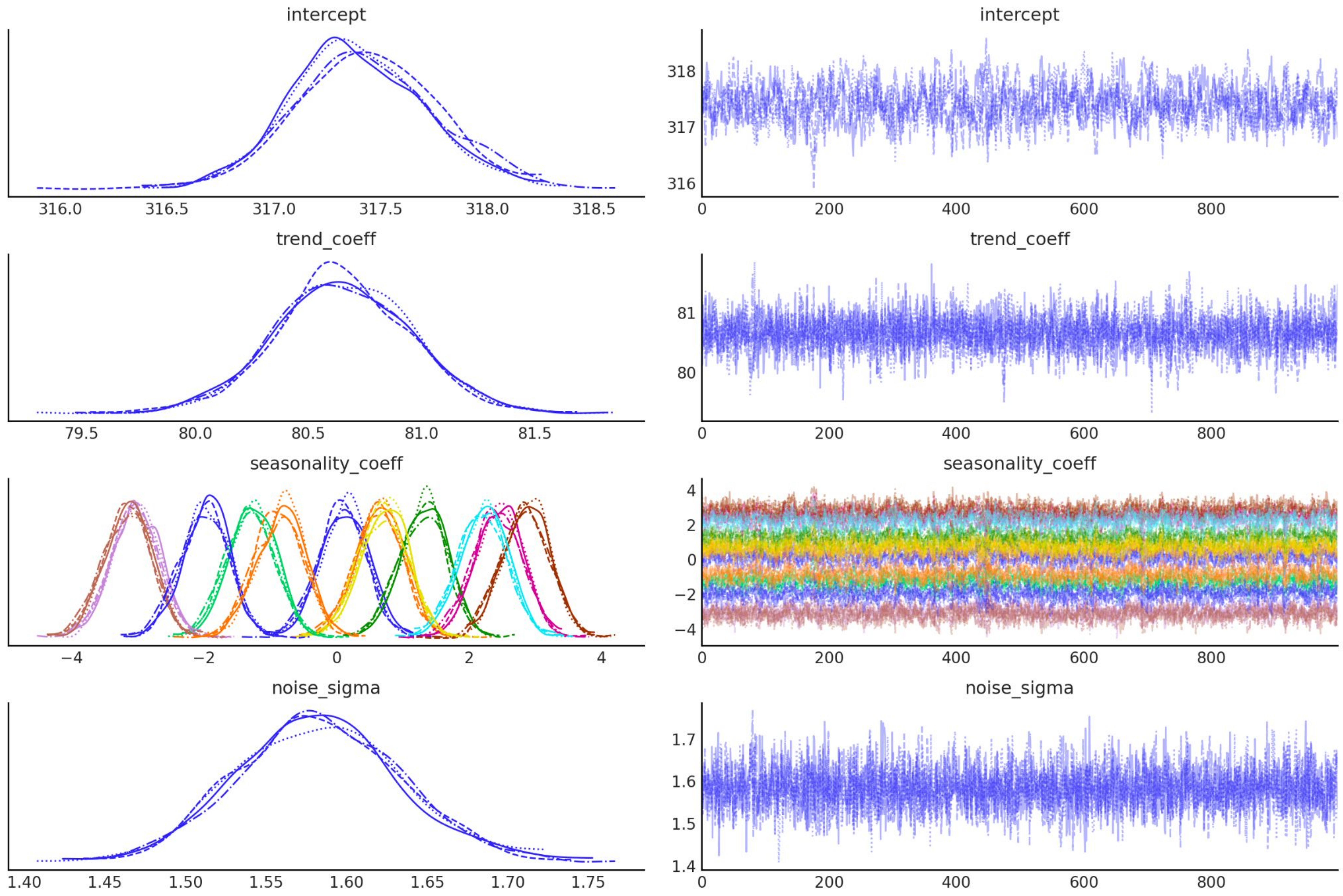
Prior predictive check

```
# Draw 100 prior and prior predictive samples
rng, key = jax.random.split(rng, 2)
prior_samples = ts_regression_model.sample(100, seed=key)
prior_predictive_timeseries = jnp.squeeze(prior_samples.observed)
```



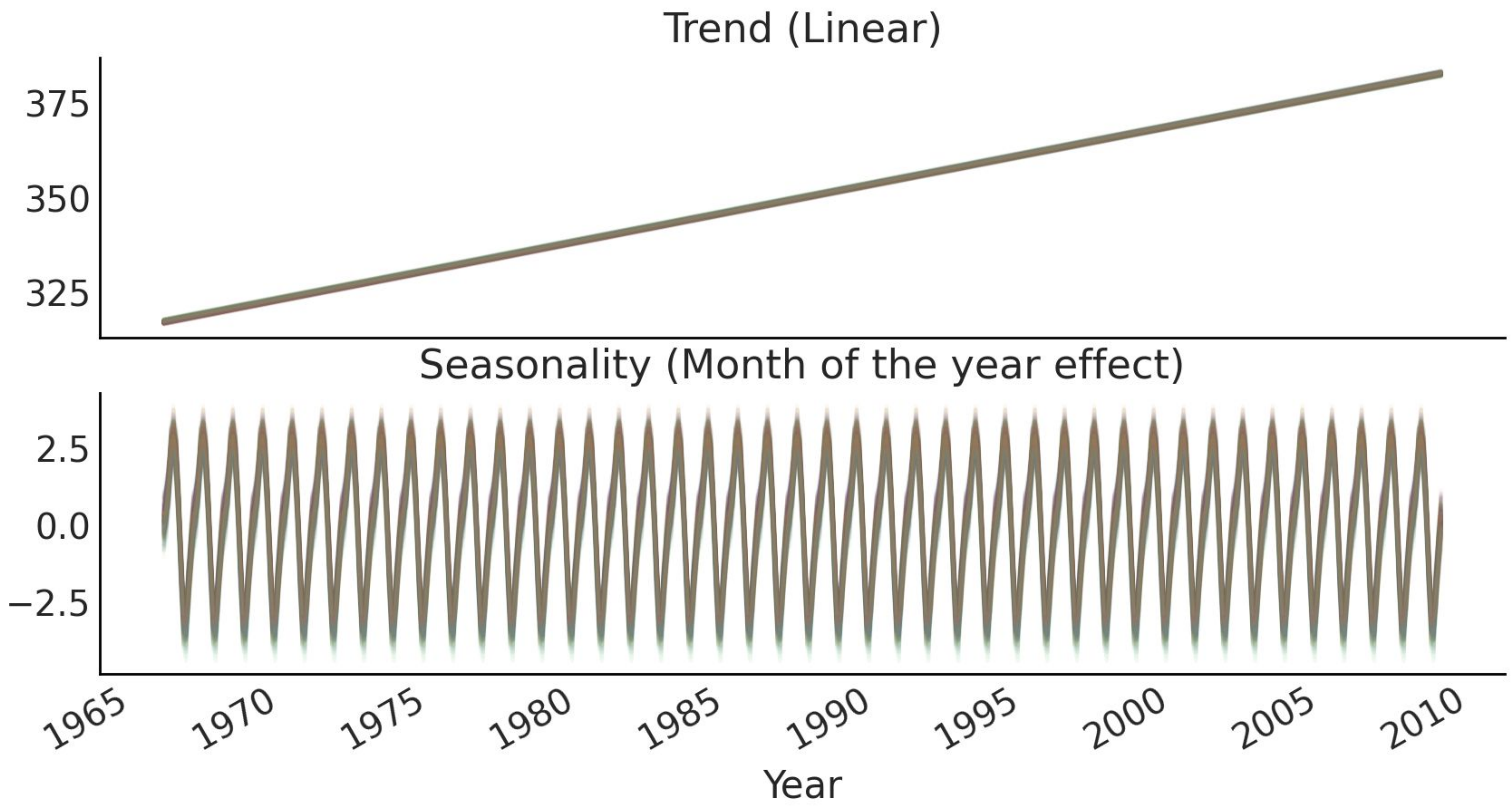
Google Research

Inference with MCMC



Visualize time series components

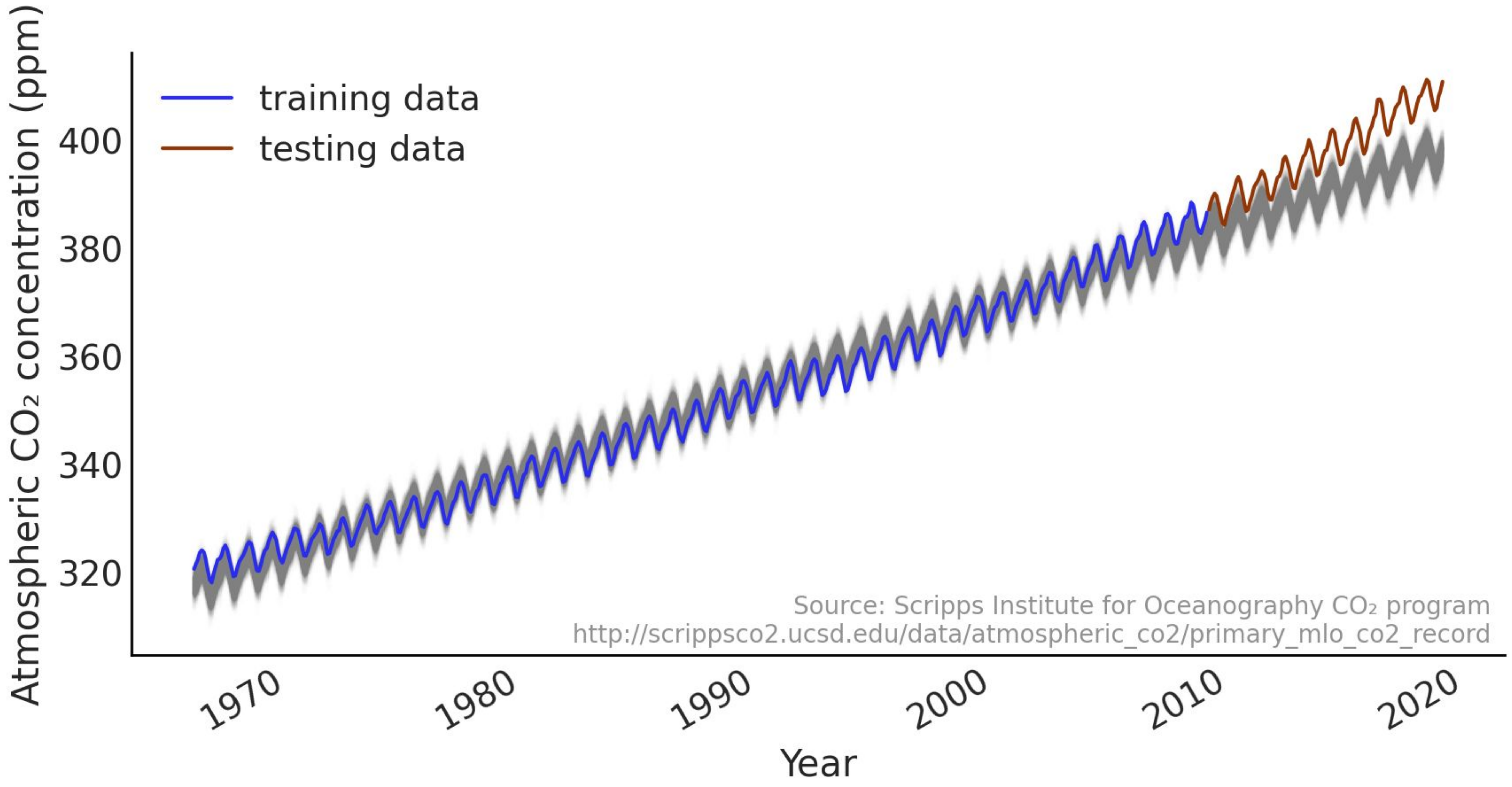
```
trend_posterior = mcmc_samples.intercept + \
    jnp.einsum('ij,...->i...', trend_all, mcmc_samples.trend_coeff)
seasonality_posterior = jnp.einsum(
    'ij,...jk->i...k', seasonality_all, mcmc_samples.seasonality_coeff)
```



Google Research

Visualize Forecast

```
y_hat = trend_posterior + seasonality_posterior.squeeze()
posterior_predictive_dist = tfd.Normal(y_hat, mcmc_samples.noise_sigma)
rng, key = jax.random.split(rng, 2)
ppc_sample = posterior_predictive_dist.sample(seed=key)
```

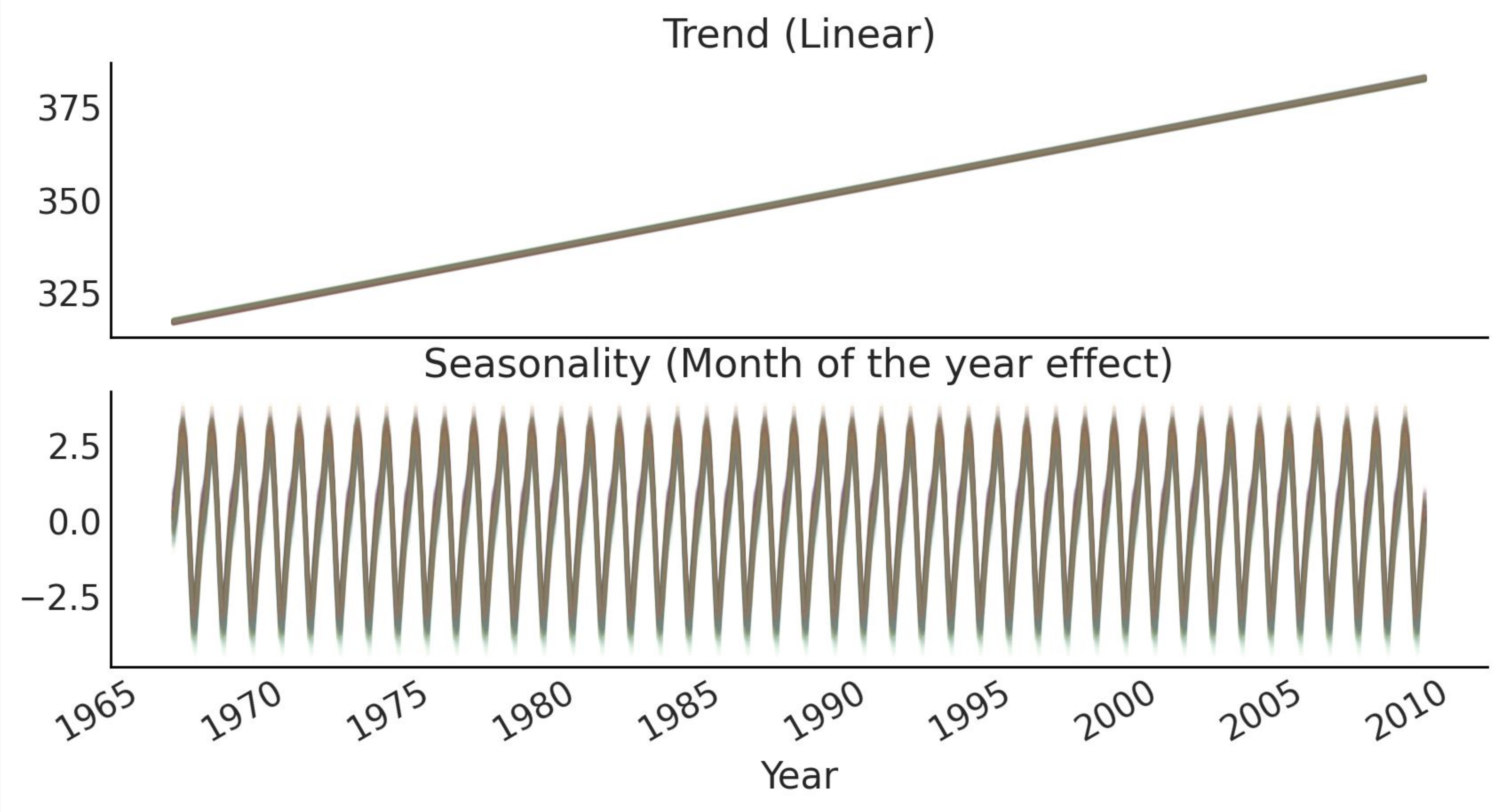


Google Research

Example:
Generalized Additive Model (GAM)

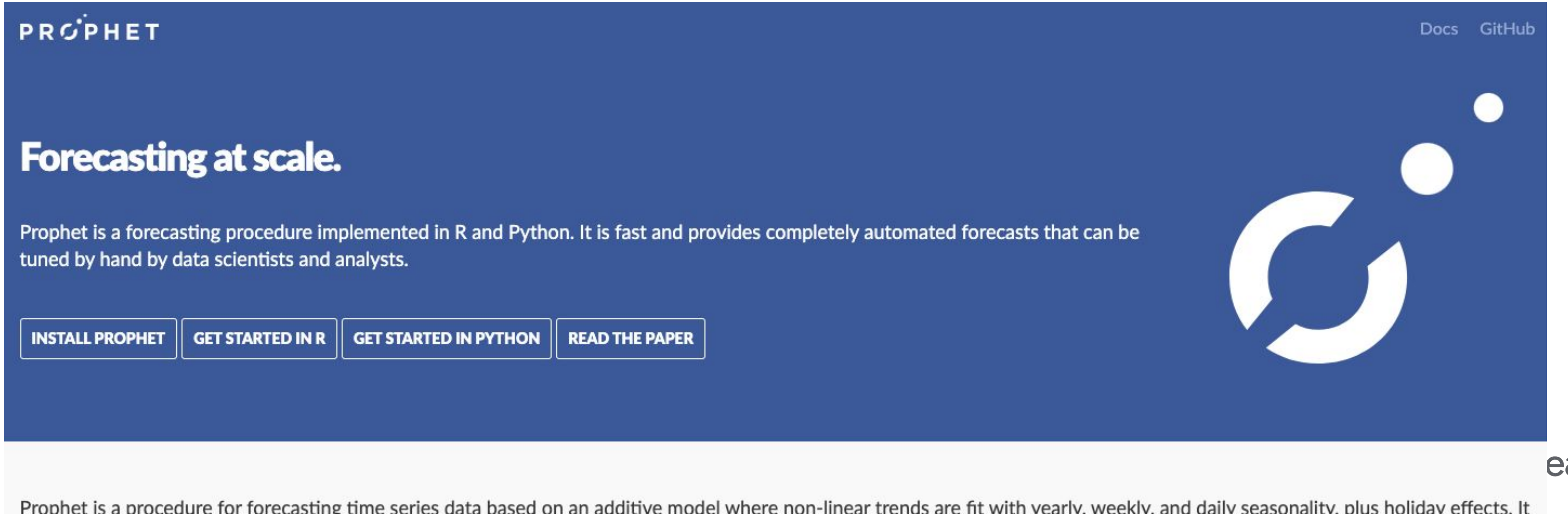
How can we capture the trend better?

```
Y_{t} = Trend_{t}
      + Seasonality/Holiday_{t}
      + Residuals_{t}
```



Generalized Additive Model (GAM)

- Example: Facebook Prophet
  - Trend: step linear function
  - Seasonality: fourier basis function

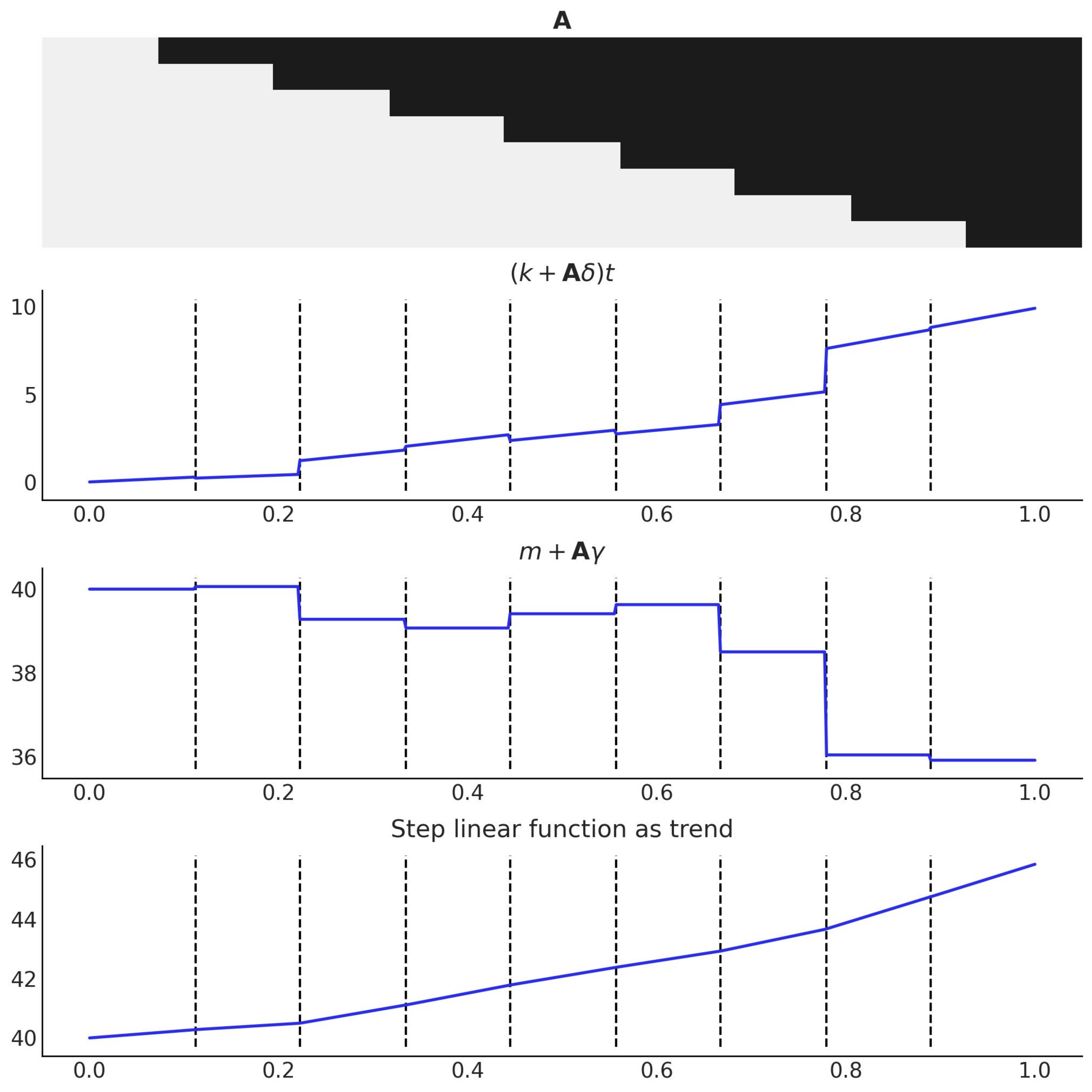


earch

Step linear function as trend

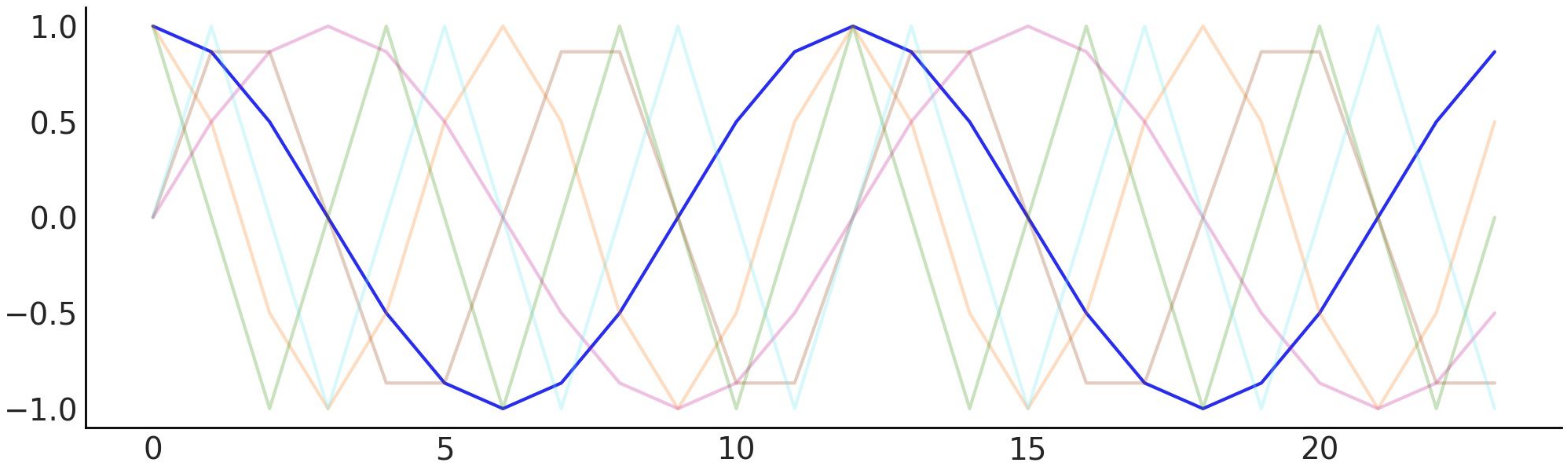
$$g(t) = (k + A\delta)t + (m + A\gamma)$$

```
n_changepoints = 8
n_tp = 500
t = np.linspace(0, 1, n_tp)
s = np.linspace(0, 1, n_changepoints + 2)[1:-1]
A = (t[:, None] > s)

k, m = 2.5, 40
delta = np.random.laplace(.1, size=n_changepoints)
growth = (k + A @ delta) * t
offset = m + A @ (-s * delta)
trend = growth + offset
```



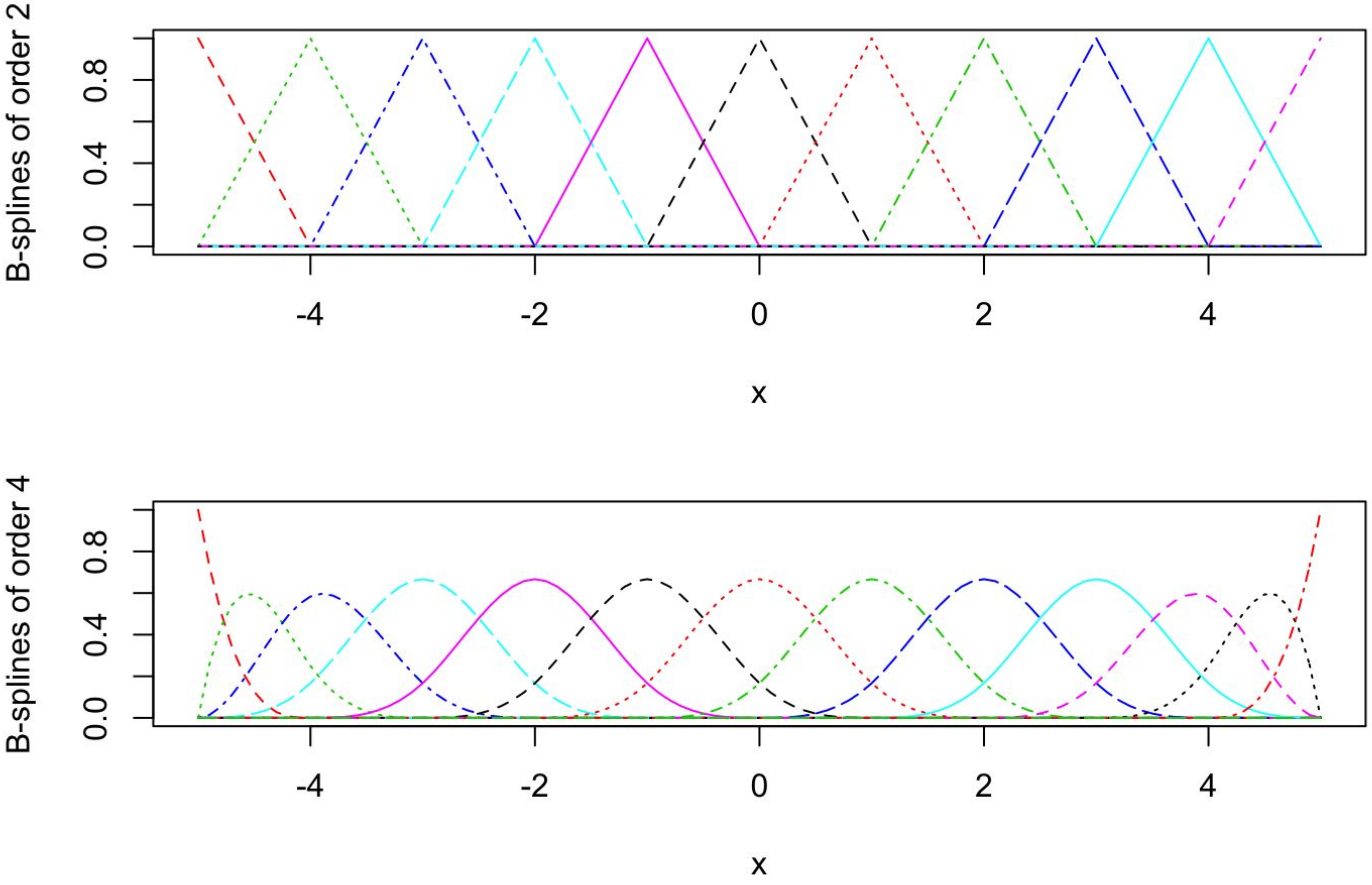
Fourier basis function as seasonality

```
def gen_fourier_basis(t, p=365.25, n=3):
  x = 2 * np.pi * (np.arange(n) + 1) * t[:, None] / p
  return np.concatenate((np.cos(x), np.sin(x)), axis=1)

n_tp = 500
p = 12
t_monthly = np.asarray([i % p for i in range(n_tp)])
monthly_X = gen_fourier_basis(t_monthly, p=p, n=3)
```



Google Research

B-spline basis functions



From https://mc-stan.org/users/documentation/case-studies/splines_in_stan.html

The Model

Wrap the JointDistribution* in a function so we can condition on new input easier

```
def gen_gam_jd(t, A, X):

  @tfd.JointDistributionCoroutineAutoBatched
  def gam():
    seasonality, trend, noise_sigma = yield from gam_prediction(t, A, X)
    y_hat = seasonality + trend
    observed = yield tfd.Normal(y_hat, noise_sigma, name='observed')

 return gam
```

Google Research

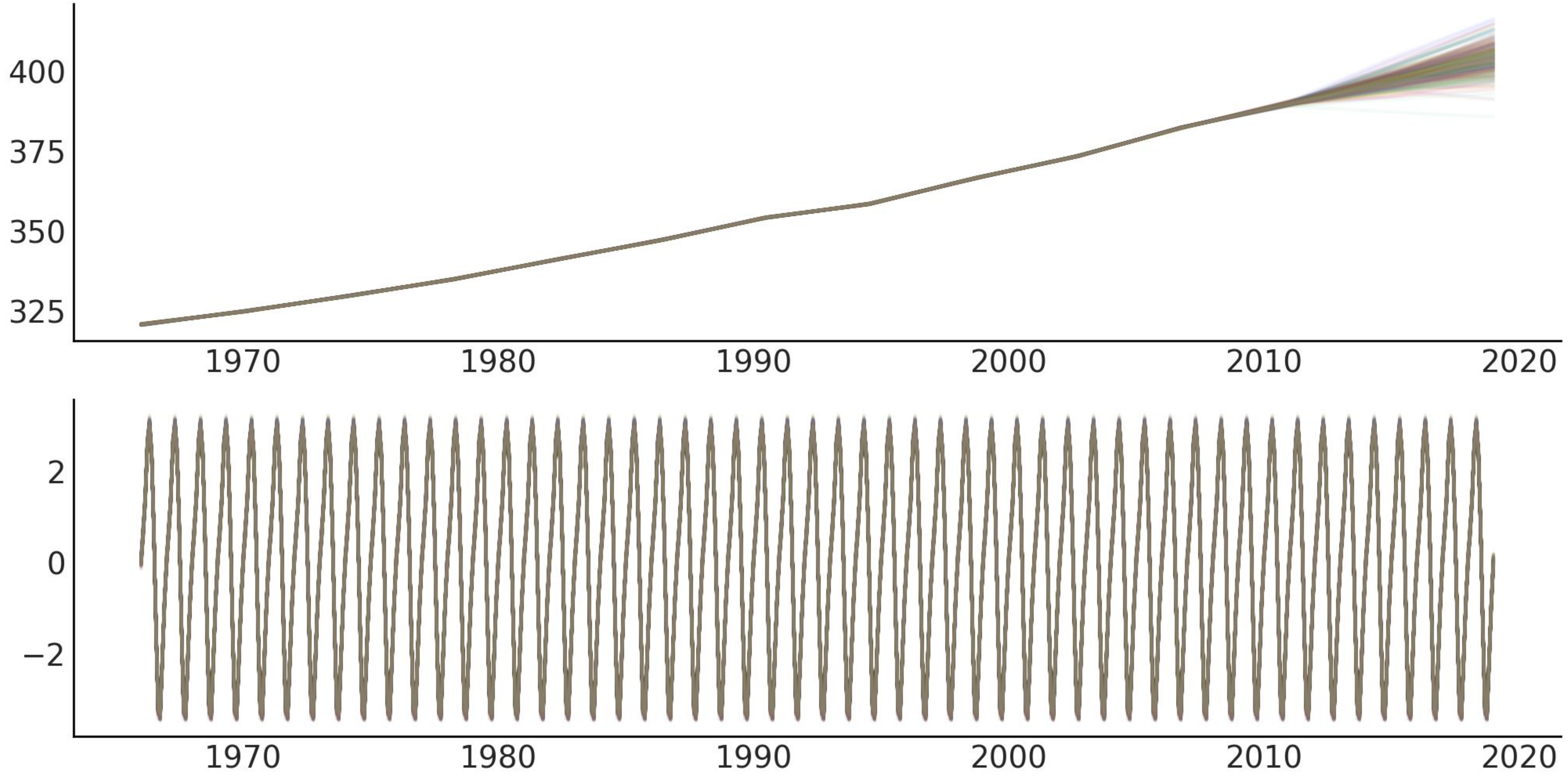
The Model

```
def gam_prediction(t, A, X):
  beta = yield tfd.Sample(tfd.Normal(0., 1.), sample_shape=X.shape[-1], name='beta')
  seasonality = jnp.einsum('ij,...j->...i', X, beta)

  k = yield tfd.HalfNormal(10., name='k')
  m = yield tfd.Normal(0., 100., name='m')
  tau = yield tfd.HalfNormal(10., name='tau')
  delta = yield tfd.Sample(tfd.Laplace(0., tau), sample_shape=A.shape[-1], name='delta')
  growth_rate = k[..., None] + jnp.einsum('ij,...j->...i', A, delta)
  offset = m[..., None] + jnp.einsum('ij,...j->...i', A, -s * delta)
  trend = growth_rate * t + offset

  noise_sigma = yield tfd.HalfNormal(scale=5., name='noise_sigma')
  return seasonality, trend, noise_sigma
```

Google Research

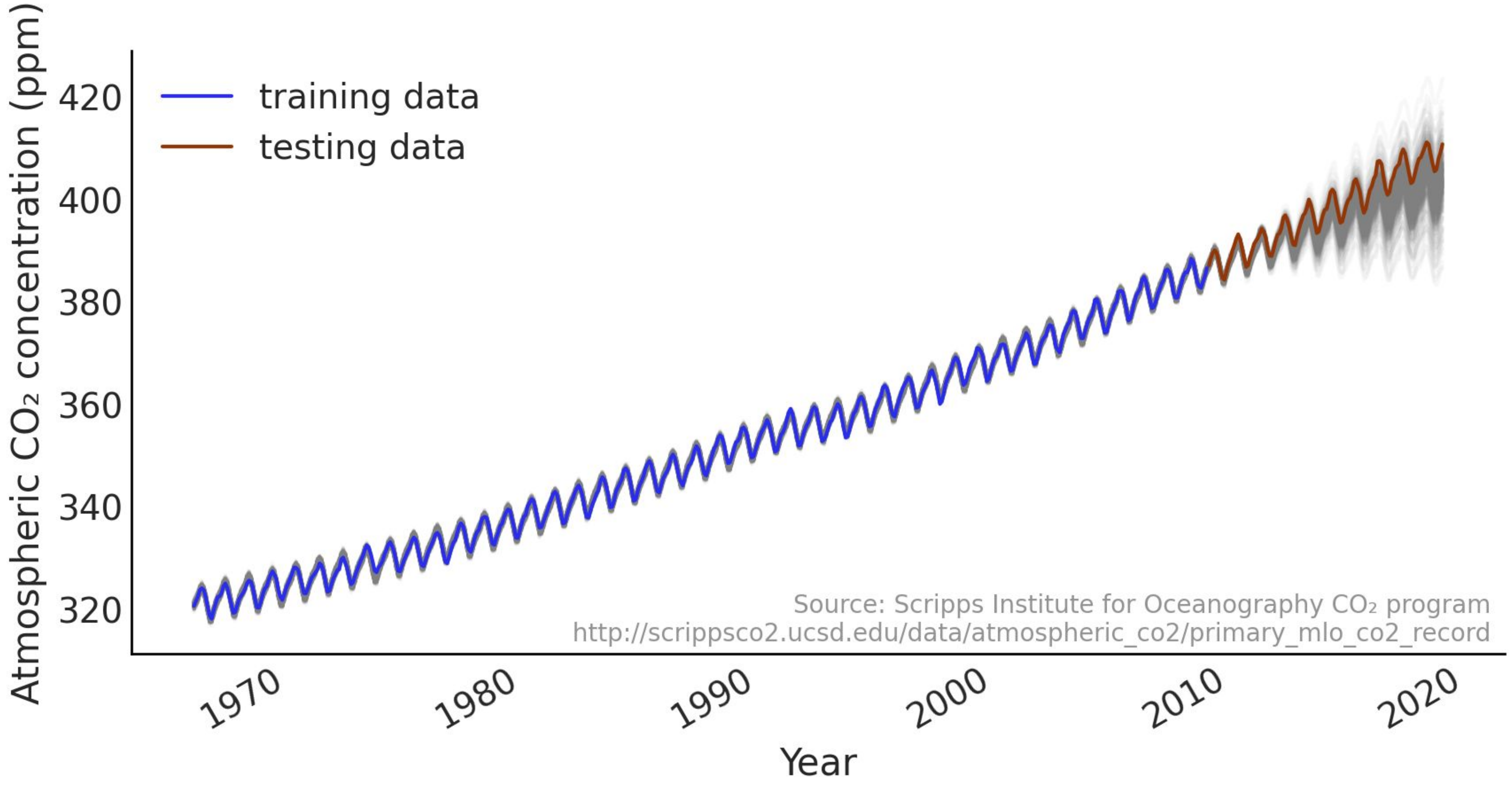
The Model and inference result

```
A_train = A[:co2_by_month_training_data.shape[0]]
X_train = X_pred[:co2_by_month_training_data.shape[0]]
t_train = t[:co2_by_month_training_data.shape[0]]

gam = gen_gam_jd(t_train, A_train, X_train)
```



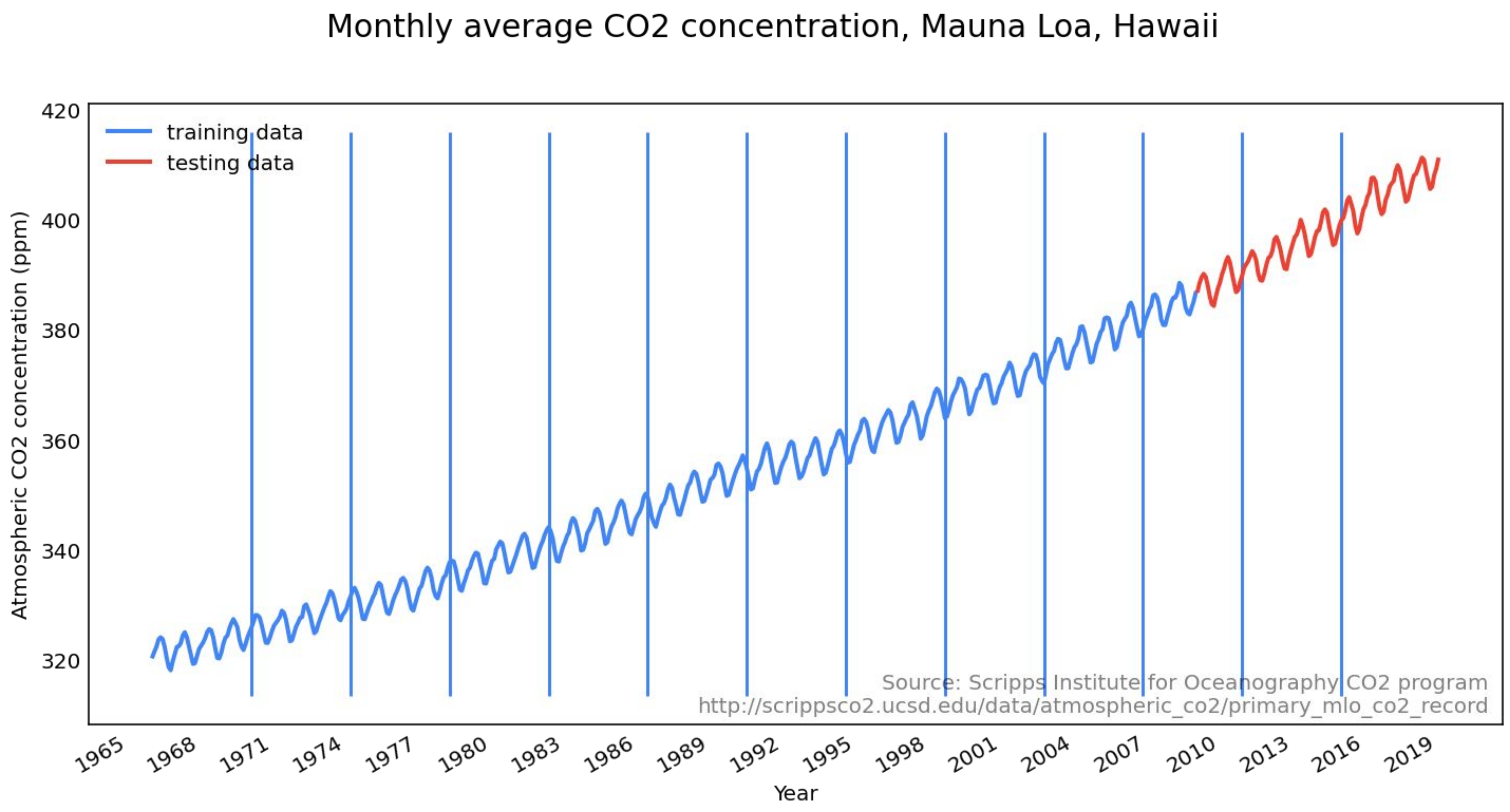
Make forecast

```
gam_full = gen_gam_jd(t, A, X_pred)
...
```



Google Research

Note on generative process of step linear model

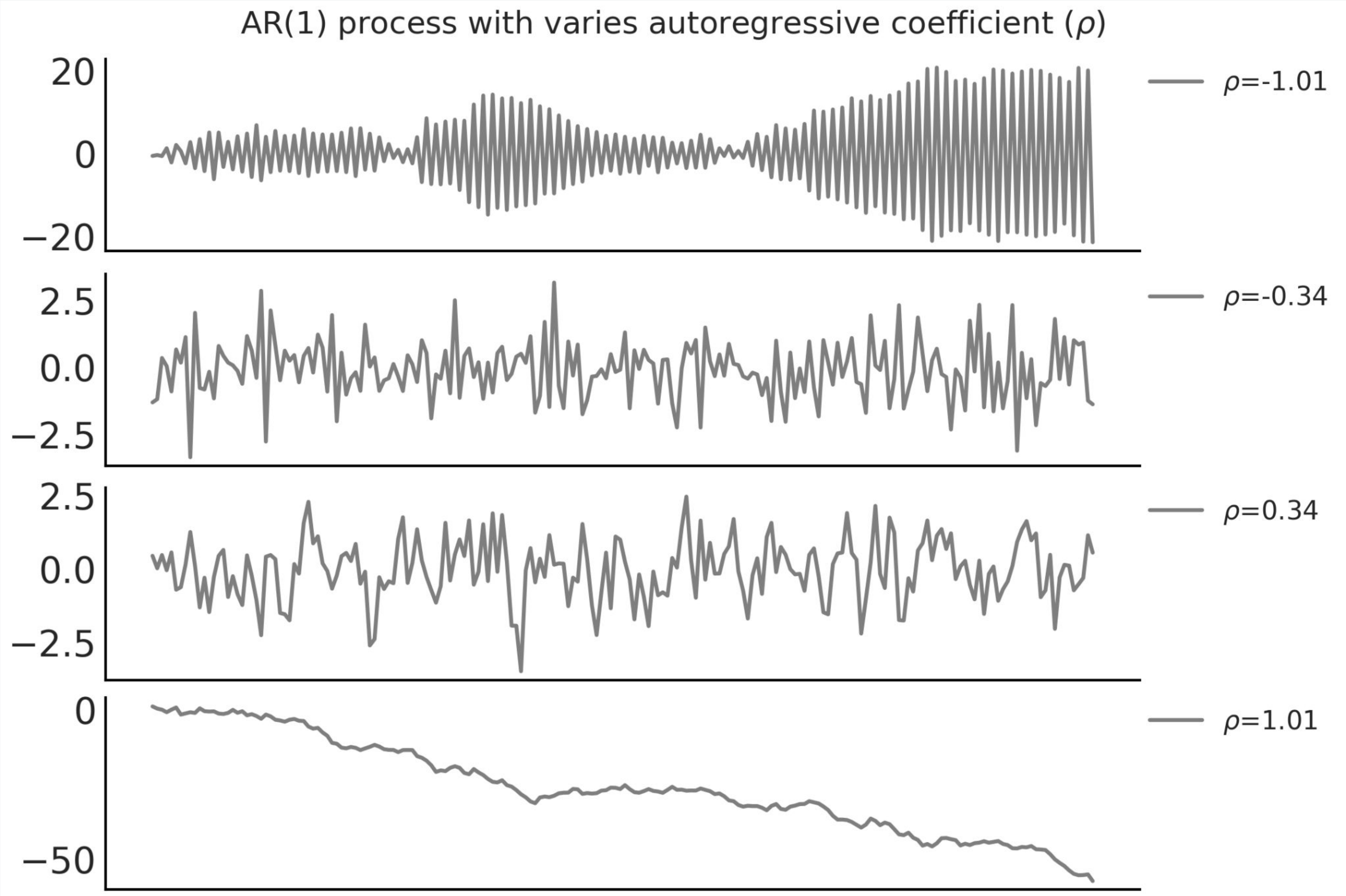


In Taylor, S. J., & Letham, B. (2018), the generative process for forecast is not identical to the generative model, as the step linear function is evenly spaced with the change point predetermined. Taylor, S. J., & Letham, B. (2018) recommend for forecasting to generate first whether that time point would be a change point, proportion to the number of change points divided by the total number of time points in the observation; and then generate a new delta from the posterior distribution $\delta_{new} \sim \text{Laplace}(0, tau)$

Better model for residual component

```
Y_{t} = Trend_{t}
      + Seasonality/Holiday_{t}
      + Residuals_{t}
```

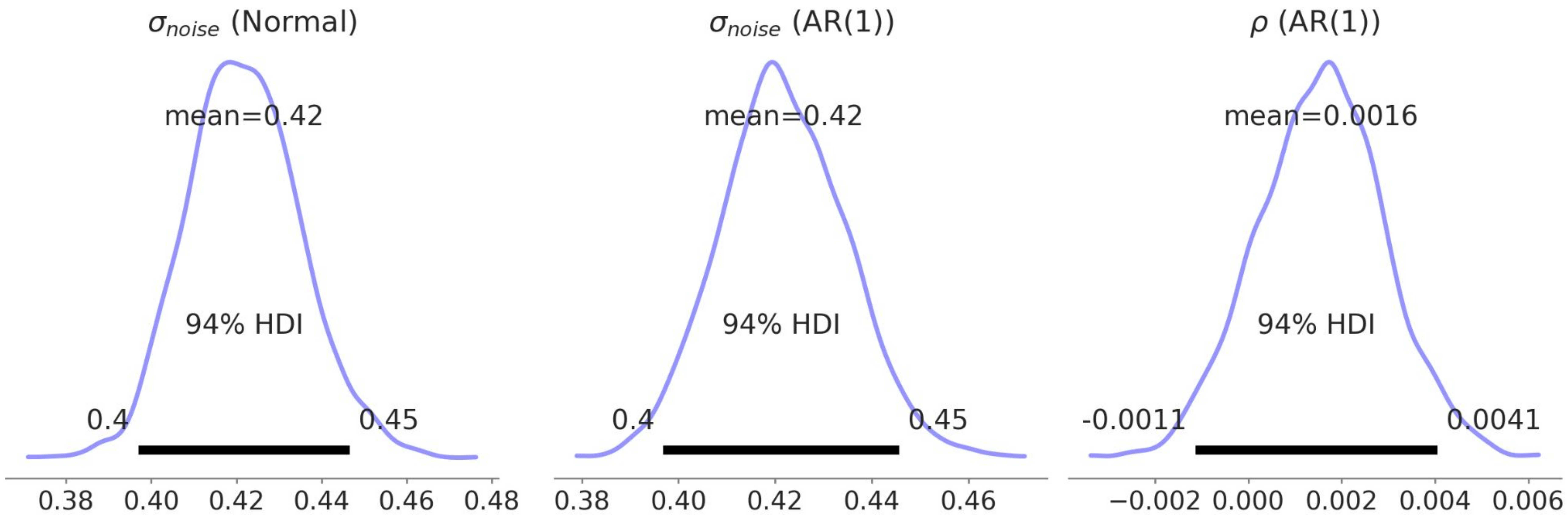
$$y_t \sim \mathrm{Normal}(\alpha + \rho y_{t-1}, \sigma)$$



AR(1) Likelihood

```
def gen_gam_ar_jd(t, A, X, y):

  @tfd.JointDistributionCoroutineAutoBatched
  def gam():
    seasonality, trend, noise_sigma = yield from gam_prediction(t, A, X)
    y_hat = seasonality + trend
    # Likelihood
    rho = yield tfd.Uniform(-1., 1., name="rho")
    ym1 = jnp.concatenate([jnp.zeros_like(y[:1]), y[:-1]], axis=0)
    observed = yield tfd.Normal(y_hat + rho * ym1, noise_sigma, name='observed')

 return gam
```

Posterior distribution of selected parameters

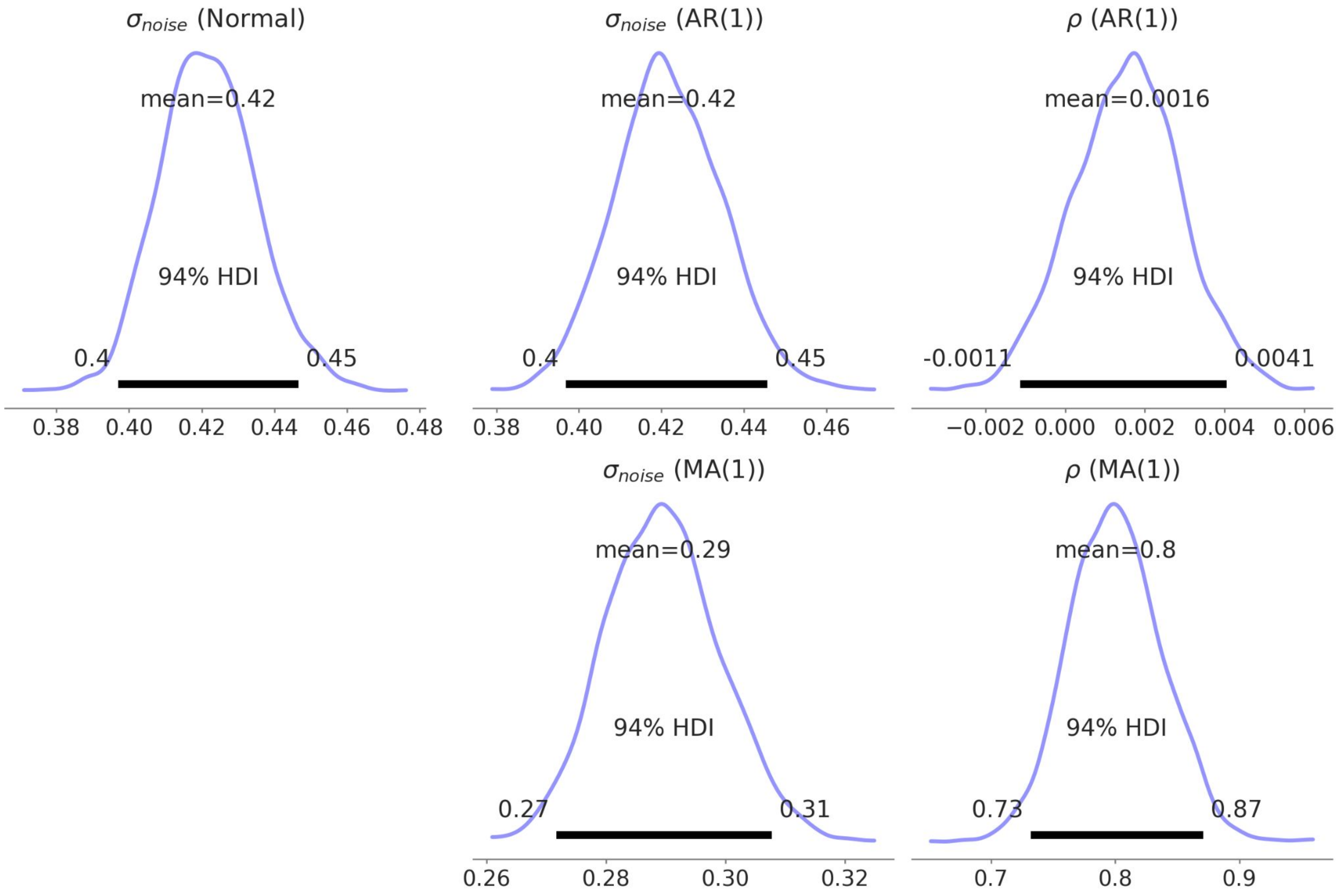


Google Research

MA(1) Likelihood

```python
def gen_gam_ma_jd(t, A, X, y):

  @tfd.JointDistributionCoroutineAutoBatched
  def gam():
    seasonality, trend, noise_sigma = yield from gam_prediction(t, A, X)
    y_hat = seasonality + trend
    # Likelihood
    rho = yield tfd.Uniform(-1., 1., name="rho")
    delta = y - y_hat
    delta_m1 = jnp.concatenate([jnp.zeros_like(delta[:1]), delta[:-1]], axis=0)
    observed = yield tfd.Normal(
        y_hat + rho * delta_m1, noise_sigma, name='observed')

 return gam
```

Posterior distribution of selected parameters



(S)AR(I)MA(X)

Seasonal AutoRegressive Integrated Moving Average with eXogenous regressors model

ARMAX

$$y_t = \alpha + \sum_{i=1}^{p} \phi_i y_{t-i} + \sum_{j=1}^{q} \theta_j \epsilon_{t-j} + \epsilon_t$$

$$\epsilon_t \sim \text{Normal}(0, \sigma^2)$$

SARMAX

$$y_t = \alpha + \sum_{i=1}^{p} \phi_i y_{t-period-i} + \sum_{j=1}^{q} \theta_j \epsilon_{t-period-j} + \epsilon_t$$

Google Research

What about the Integrated part?

I(k) is more of a data analysis trick, as you perform y_t - y_tm1 on the observed (and the predictor if you have X). It does not increase the number of parameter in your model.

Short Break

Google Research

03

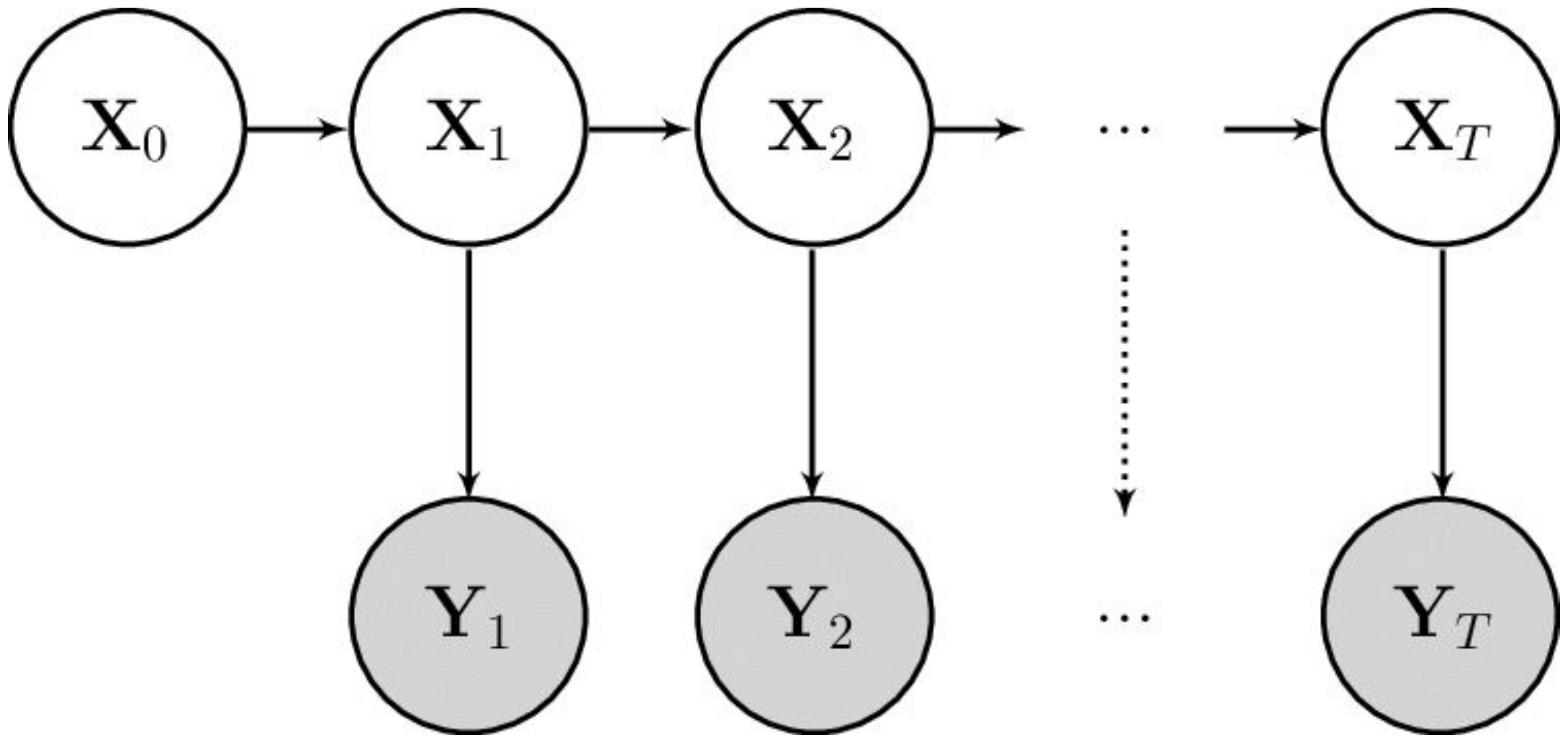
State Space Models

Time series as State Space Models

$$X_0 \sim \pi(X_0)$$

for t in 0...T:

$$Y_t \sim \pi^{\psi}(Y_t \mid X_t)$$
$$X_{t+1} \sim \pi^{\theta}(X_{t+1} \mid X_t)$$



Linear Gaussian State Space Model (LGSSM)

$$Y_t = H_t X_t + \epsilon_t$$
$$X_t = F_t X_{t-1} + \eta_t$$

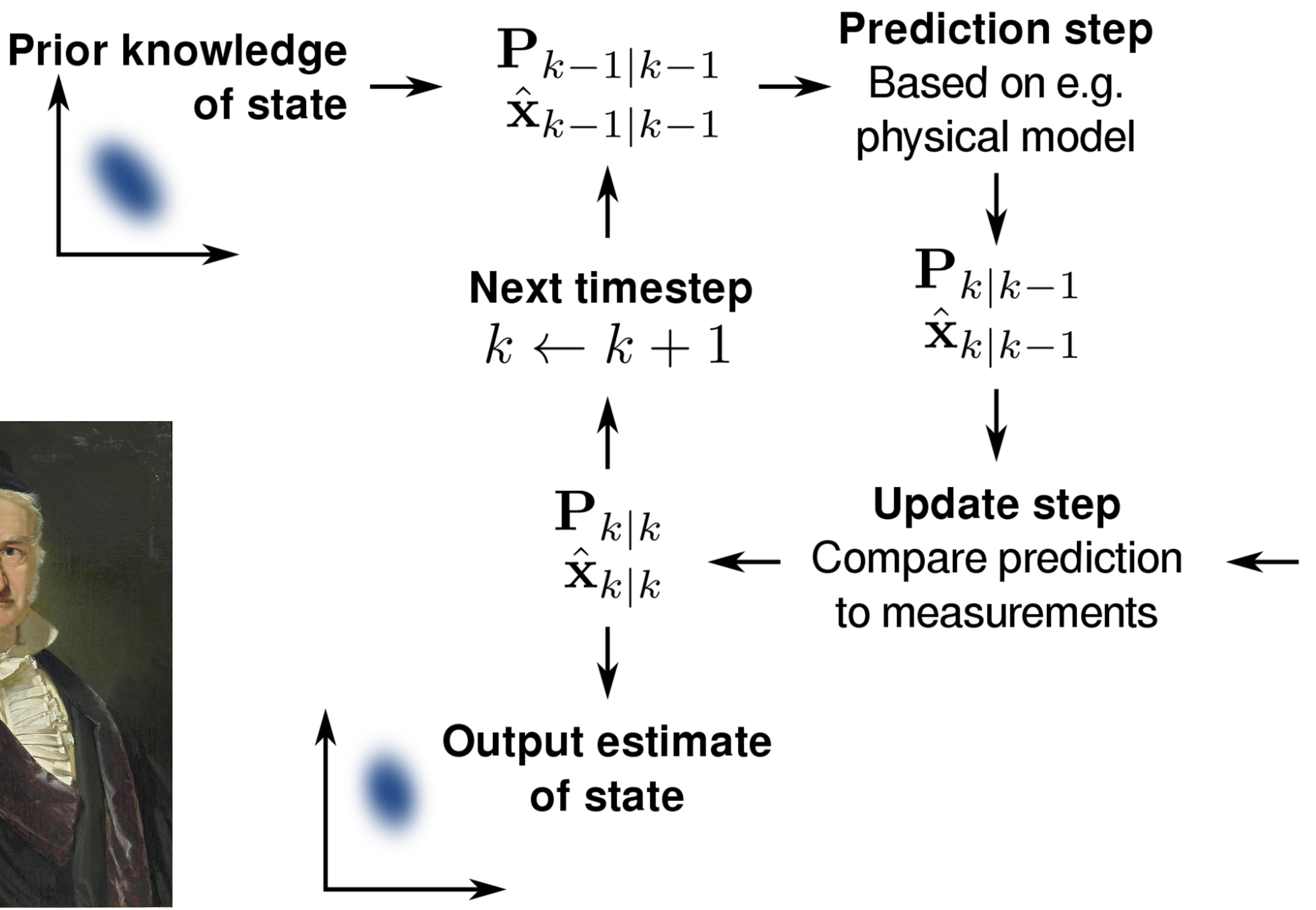
$$X_t \sim \pi(X_t \mid X_{t-1}) = \text{Normal}(F_{t-1} X_{t-1}, Q_{t-1})$$
$$Y_t \sim \pi(Y_t \mid X_t) = \text{Normal}(H_t X_t, R_t)$$

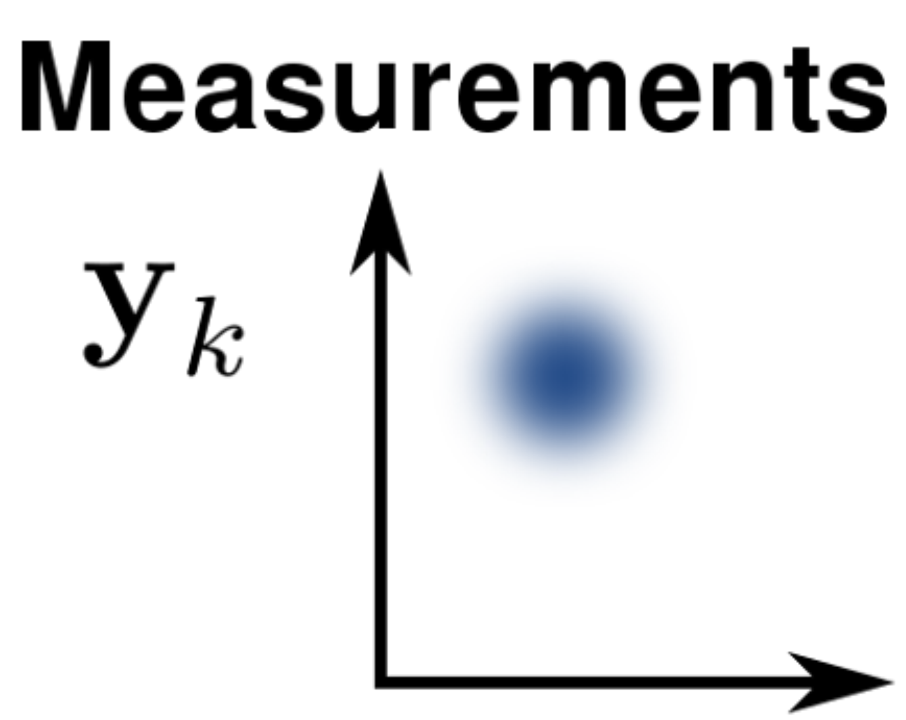
Google Research

When *everything** is Gaussian

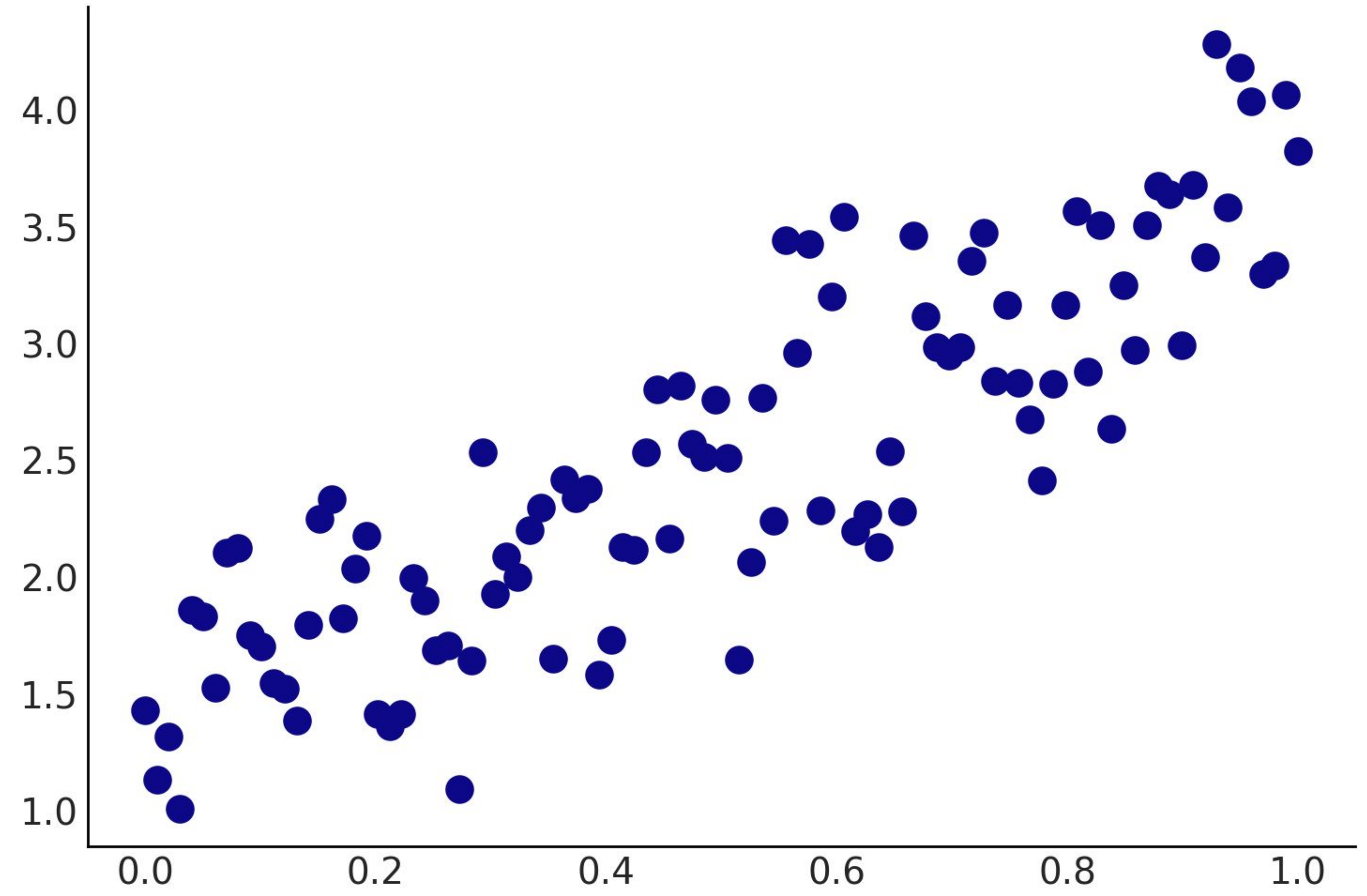
Google Research

Kalman filter





Linear regression as a `tfd.LinearGaussianStateSpaceModel`



$$X_t = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$$

$$y_t = \theta_0 + \theta_1 * t = [1, t] \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$$

$$Y_t = H_t X_t + \epsilon_t$$
$$X_t = F_t X_{t-1} + \eta_t$$

```
X_0 = tfd.MultivariateNormalDiag(loc=[0., 0.], scale_diag=[5., 5.])

H_t = lambda t: tfl.LinearOperatorFullMatrix([[1., x[t].squeeze()]])
# epsilon_t ~ Normal(0, R_t)
eps_t = lambda _: tfd.MultivariateNormalDiag(loc=[0.], scale_diag=[sigma])

F_t = lambda _: tfl.LinearOperatorIdentity(2)
# eta_t ~ Normal(0, Q_t)
eta_t = lambda _: tfd.MultivariateNormalDiag(loc=[0., 0.], scale_diag=[0., 0.])
```

Linear regression as a `tfd.LinearGaussianStateSpaceModel`

$$Y_t = H_t X_t + \epsilon_t$$
$$X_t = F_t X_{t-1} + \eta_t$$

```
linear_growth_model = tfd.LinearGaussianStateSpaceModel(
    num_timesteps=num_timesteps,
    transition_matrix=F_t,
    transition_noise=eta_t,
    observation_matrix=H_t,
    observation_noise=eps_t,
    initial_state_prior=X_0)
```

Run the Kalman filter

```
(
    log_likelihoods,
    mt_filtered, Pt_filtered,
    mt_predicted, Pt_predicted,
    observation_means, observation_cov  # observation_cov is R_t
) = linear_growth_model.forward_filter(y)
```

Kalman filter

Update:

$$z_t = Y_t - \mathbf{H}_t m_{t|t-1}$$

$$\mathbf{S}_t = \mathbf{H}_t \mathbf{P}_{t|t-1} \mathbf{H}_t^T + \mathbf{R}_t$$

$$\mathbf{K}_t = \mathbf{P}_{t|t-1} \mathbf{H}_t^T \mathbf{S}_t^{-1}$$

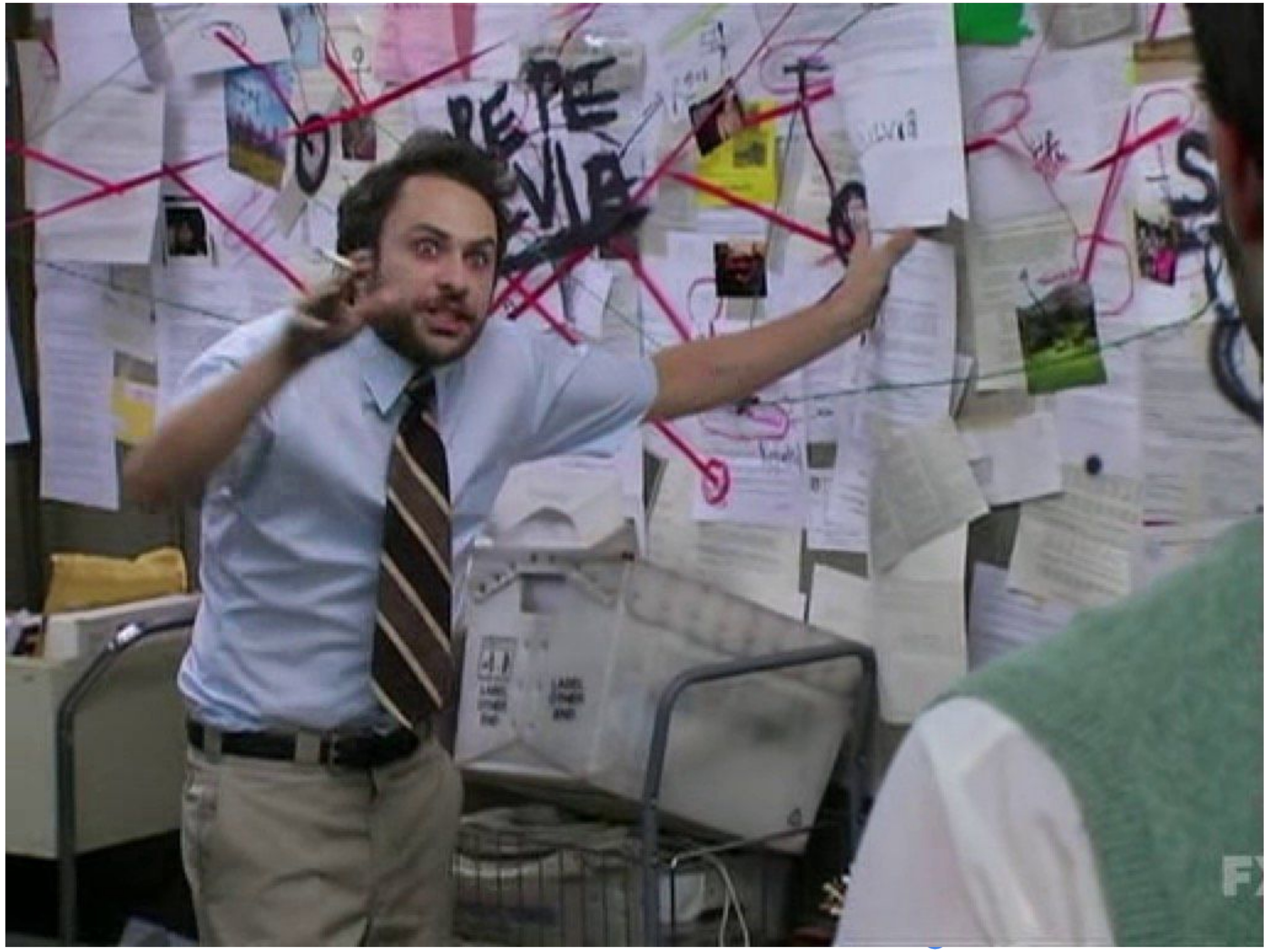
$$m_{t|t} = m_{t|t-1} + \mathbf{K}_t z_t$$

$$\mathbf{P}_{t|t} = \mathbf{P}_{t|t-1} - \mathbf{K}_t \mathbf{H}_t \mathbf{P}_{t|t-1}$$

Prediction:

$$m_{t+1|t} = \mathbf{F}_t m_{t|t}$$

$$\mathbf{P}_{t+1|t} = \mathbf{F}_t \mathbf{P}_{t|t} \mathbf{F}_t^T + \mathbf{Q}_t$$



Prediction in state space model

If there is no observed data, there is no update:

```
t = 37  # Select a random time point
```

Transition: $X_{t-1} \to X_t$

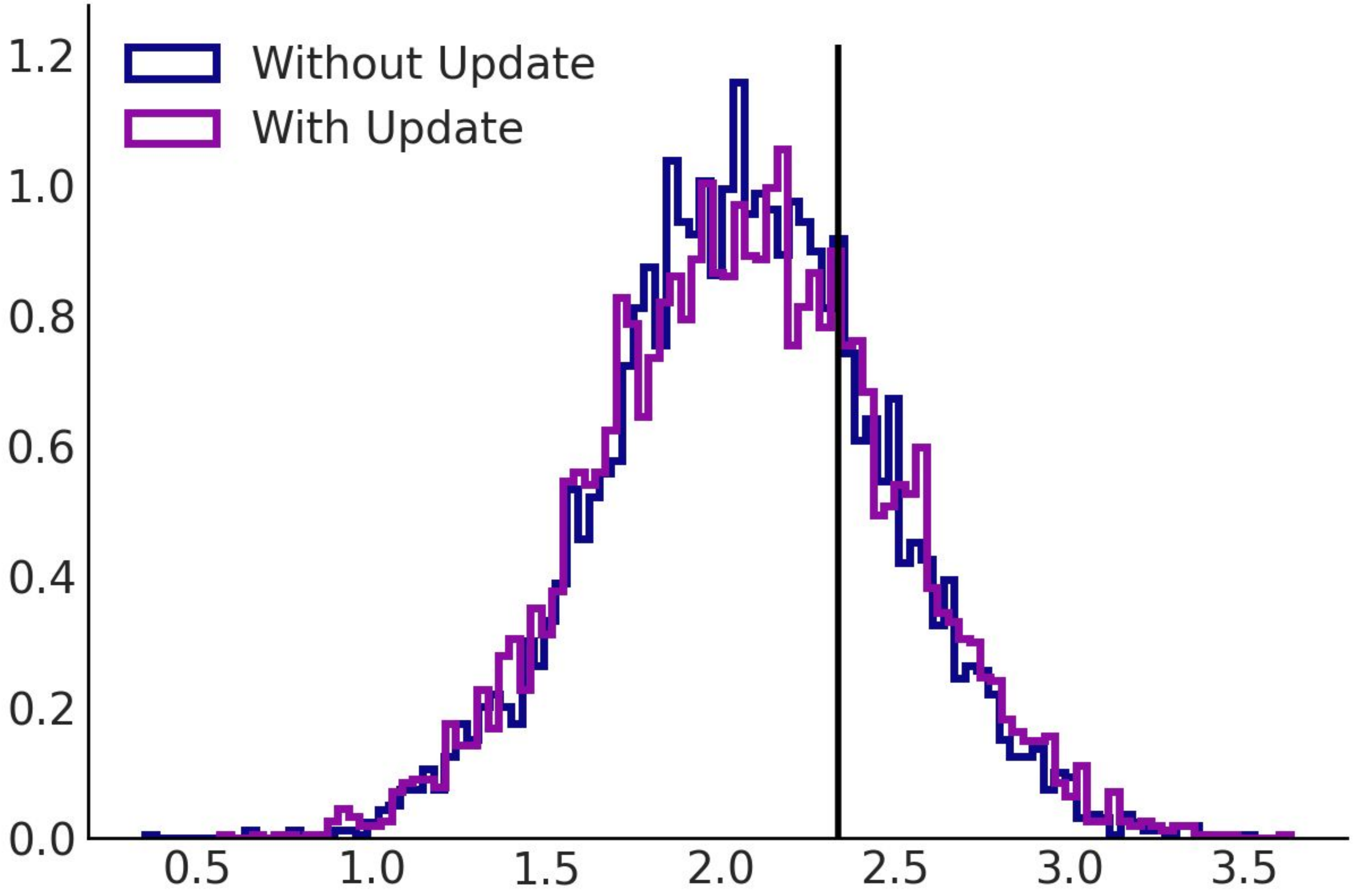
```
mu_at_t = mt_predicted[t - 1][..., None]
# Note that F_t is identity and transition noise being 0 in this case
F_at_t = linear_growth_model.transition_matrix(t)
eta_at_t = linear_growth_model.transition_noise(t)

rng, key = jax.random.split(rng, 2)
mu_at_tp1 = F_at_t.matmul(mu_at_t) + eta_at_t.sample(5000, seed=key)[..., None]
```

Observation: $X_t \to Y_t$

```
H_at_t = linear_growth_model.observation_matrix(t)
eps_at_t = linear_growth_model.observation_noise(t)
y_hat_ = H_at_t.matmul(mu_at_tp1)

rng, key = jax.random.split(rng, 2)
y_sampled = y_hat_ + eps_at_t.sample(5000, seed=key)[..., None]
```

Prediction in state space model

```
y_t = y[t]
y_hat_t = observation_means[t]
y_cov_t = observation_covs[t]

y_t_dist = tfd.Normal(y_hat_t, scale=np.sqrt(y_cov_t).squeeze())
y_sampled_ = y_t_dist.sample(5000, seed=key)
```



Google Research

Conjugate Gaussian Update and Filtering

```
mu_t_tm1 = mt_predicted[t - 1][..., None]
P_t_tm1 = Pt_predicted[t - 1]

H_at_t = linear_growth_model.observation_matrix(t).to_dense()
eps_at_t = linear_growth_model.observation_noise(t)
```

Observation: $X_{t|t-1} \to Y_t$

```
y_hat_t = H_at_t @ mu_t_tm1 + eps_at_t.mean()
S_t = H_at_t @ P_t_tm1 @ H_at_t.T + eps_at_t.covariance()
```

Update: $X_{t|t-1} \to X_{t|t}$

```
# Optimal Kalman gain K_t
K_t = P_t_tm1 @ H_at_t.T @ (S_t ** -1)
mu_t_t = mu_t_tm1 + K_t @ (y_t - y_hat_t)
# P* = P - K * H * P
P_t_t = P_t_tm1 - K_t @ H_at_t @ P_t_tm1
# P_t_t = P_t_tm1 - K_t @ S_t @ K_t.T
```

Conjugate Gaussian Update and Filtering

```
F_at_t = linear_growth_model.transition_matrix(t).to_dense()
eta_at_t = linear_growth_model.transition_noise(t)
```

Transition: $X_{t+1|t} \rightarrow X_{t|t}$

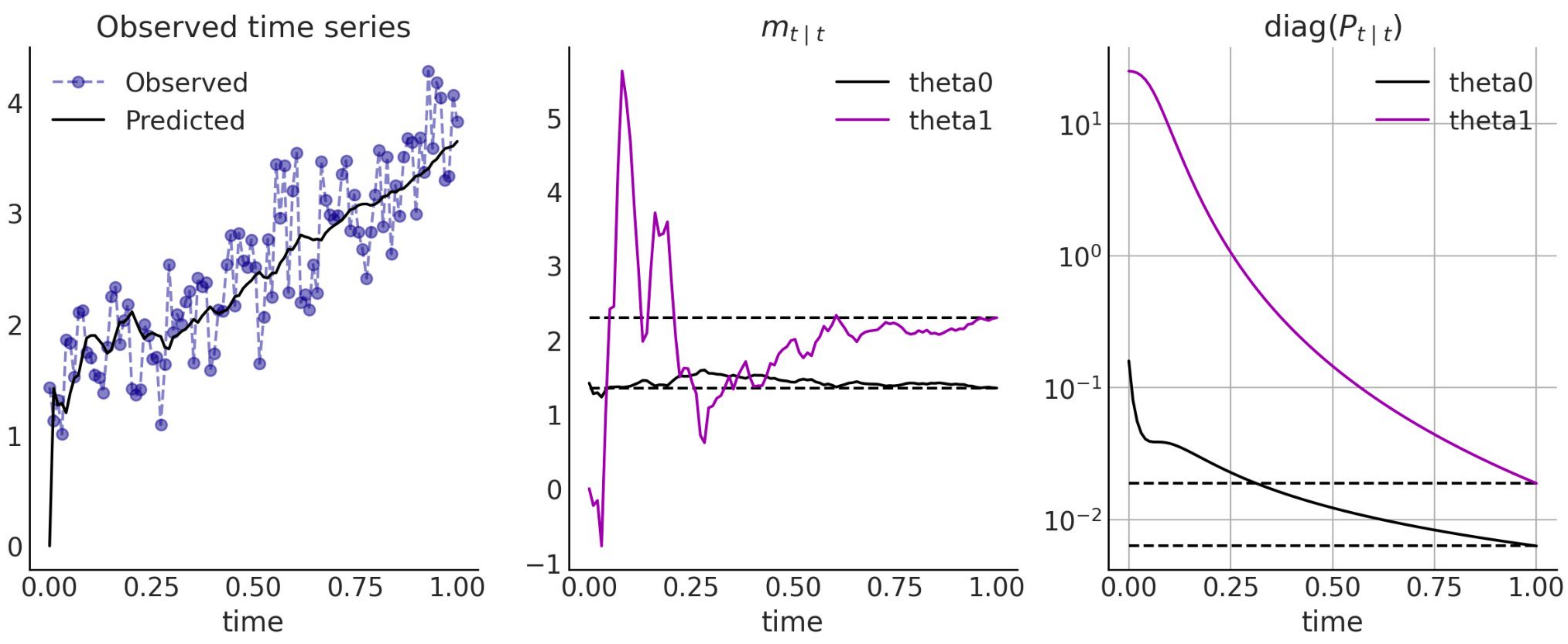
```
m_tp1_t = F_at_t @ mu_t_t + eta_at_t.mean()[..., None]
P_tp1_t = F_at_t @ P_t_t @ F_at_t.T + eta_at_t.covariance()
```

```
mu_t_t, P_t_t, y_t_dist = tfd.linear_gaussian_ssm.linear_gaussian_update(
     mu_t_tm1, P_t_tm1, H_at_t, eps_at_t, y_t)

m_tp1_t, P_tp1_t = tfd.linear_gaussian_ssm.kalman_transition(
     mu_t_t, P_t_t, F_at_t, eta_at_t)
```

Google Research

Comparison with the analytic solution

```
(
    log_likelihoods,
    mt_filtered, Pt_filtered,
    mt_predicted, Pt_predicted,
    observation_means, observation_cov  # observation_cov is R_t
) = linear_growth_model.forward_filter(y)
```



Google Research

LGSSM is a flexible extension of many classical time series models

For example:

- Adding transition noise to previous model we got local linear trend model
- Smoothing (exponential, Holt-Winters, ...)
- ARIMA express as LGSSM

Example:
Expressing ARMA as LGSSM

Google Research

ARMA(p, q)

$$y_t = \alpha + \sum_{i=1}^{p} \phi_i y_{t-i} + \sum_{j=1}^{q} \theta_j \epsilon_{t-j} + \epsilon_t$$

$$\epsilon_t \sim \text{Normal}(0, \sigma^2)$$

Google Research

ARMA(2, 1)

$$\begin{bmatrix} y_{t+1} \\ \phi_2 y_t + \theta_1 \eta'_{t+1} \end{bmatrix} = \begin{bmatrix} \phi_1 & 1 \\ \phi_2 & 0 \end{bmatrix} \begin{bmatrix} y_t \\ \phi_2 y_{t-1} + \theta_1 \eta'_t \end{bmatrix} + \begin{bmatrix} 1 \\ \theta_1 \end{bmatrix} \eta'_{t+1}$$

$$\eta'_{t+1} \sim \text{Normal}(0, \sigma^2)$$

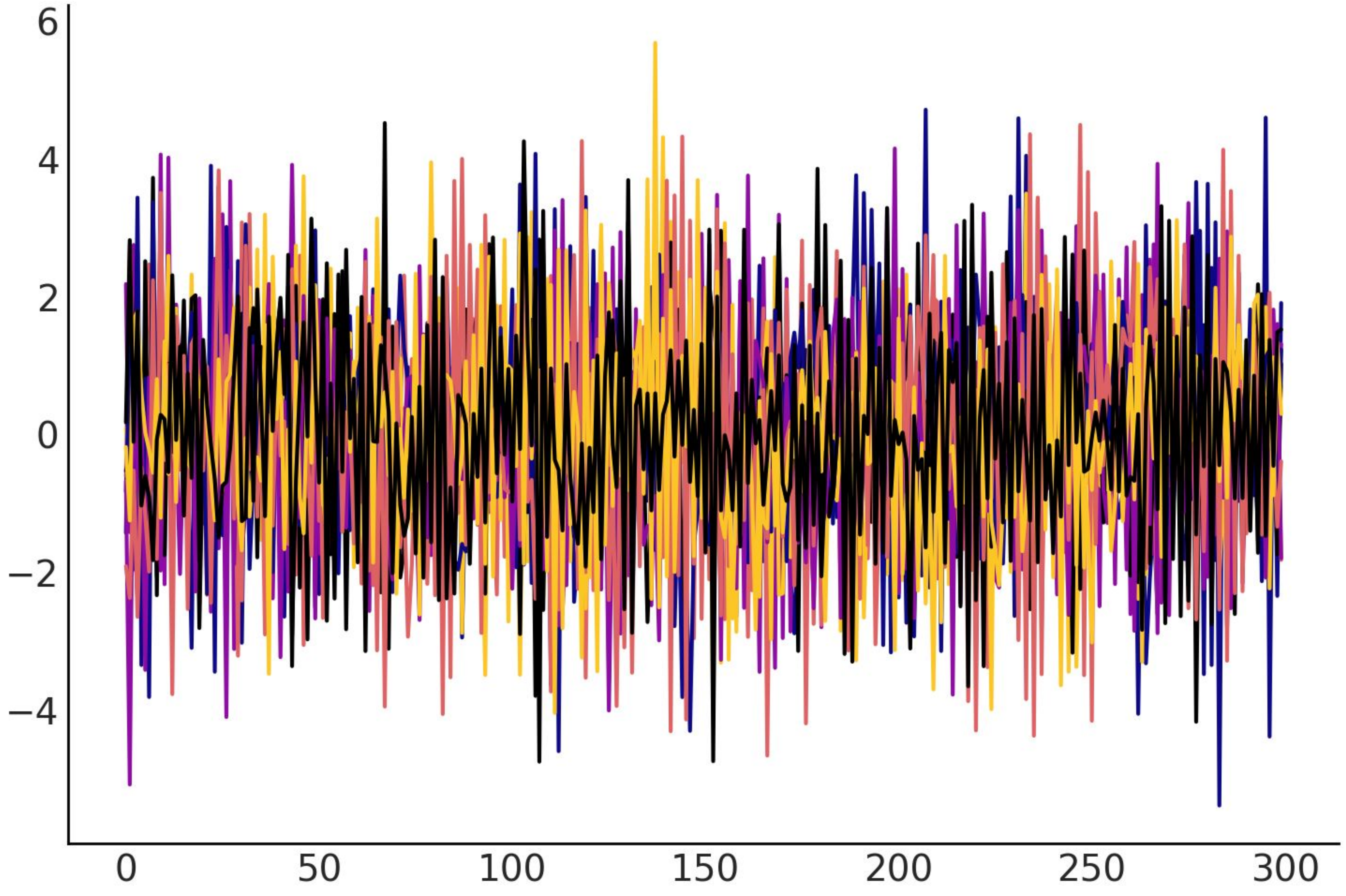
ARMA(2, 1)

```
num_timesteps = 300
phi0 = -.1
phi1 = .5
theta0 = -.25
sigma = 1.25
```

```
X_0 = tfd.MultivariateNormalDiag(scale_diag=[sigma, sigma])

F_t = jnp.asarray([[phi0, 1], [phi1, 0]])
A_t = jnp.asarray([[sigma], [sigma*theta0]])
Q_t_tril = jnp.concatenate([A_t, jnp.zeros_like(A_t)], axis=-1)
eta_t = tfd.MultivariateNormalTriL(scale_tril=Q_t_tril)

H_t = jnp.asarray([[1., 0.]])
eps_t = tfd.MultivariateNormalDiag(loc=[0.], scale_diag=[0.])

arma = tfd.LinearGaussianStateSpaceModel(
   num_timesteps, F_t, eta_t, H_t, eps_t, X_0)
```

ARMA(2, 1)

```
# Simulate from the model
rng, key = jax.random.split(rng, 2)
sim_ts = arma.sample(10, seed=key)
```



Google Research

ARMA(2, 1)

Alternative formulation:

```
arma_ = sts.AutoregressiveMovingAverageStateSpaceModel(
   num_timesteps=num_timesteps,
   ar_coefficients=[phi0, phi1],
   ma_coefficients=[theta0],
   level_scale=sigma,
   observation_noise_scale=0.,
   initial_state_prior=X_0)
```

ARMA(2, 1) as a Bayesian Structural Time Series

```
arma_sts = sts.AutoregressiveIntegratedMovingAverage(
   ar_order=2, ma_order=1, initial_state_prior=X_0)

arma_sts.parameters
arma_sts.latent_size
```

Set up a `tfd.JointDistribution` for inference

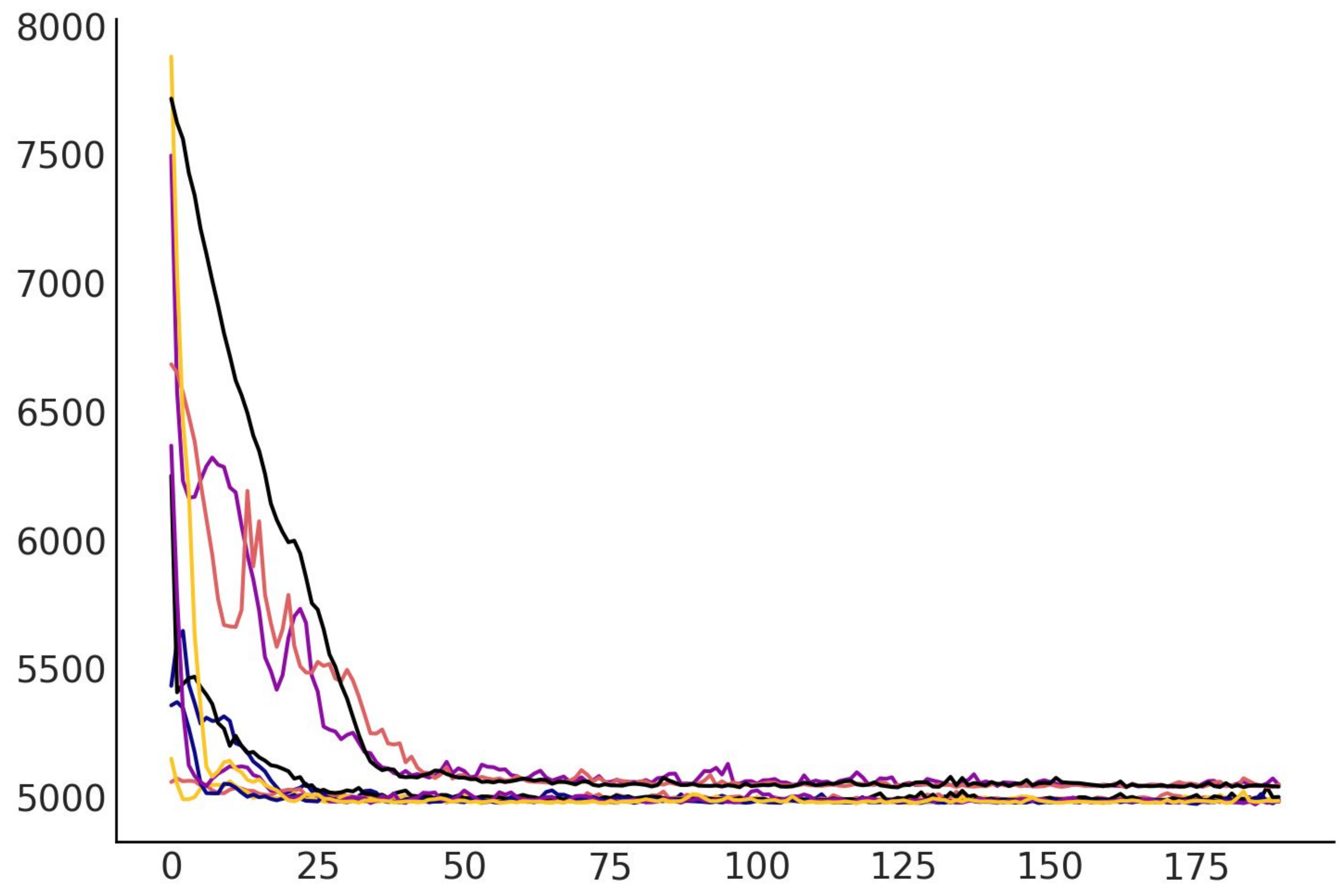
```
arma_jd_pinned = arma_sts.joint_distribution(observed_time_series=sim_ts)
```

Variational Inference

```
nb = 10
jd_batch_shape = arma_jd_pinned.batch_shape
num_variational_steps = 200

init_fn, build_surrogate_mb = tfp.experimental.vi.build_factored_surrogate_posterior_stateless(
   event_shape=arma_jd_pinned.event_shape,
   bijector=arma_jd_pinned.experimental_default_event_space_bijector(),
   batch_shape=(nb, ) + jd_batch_shape if nb is not None else jd_batch_shape
   )
initial_parameters = init_fn(jax.random.PRNGKey(1))
```

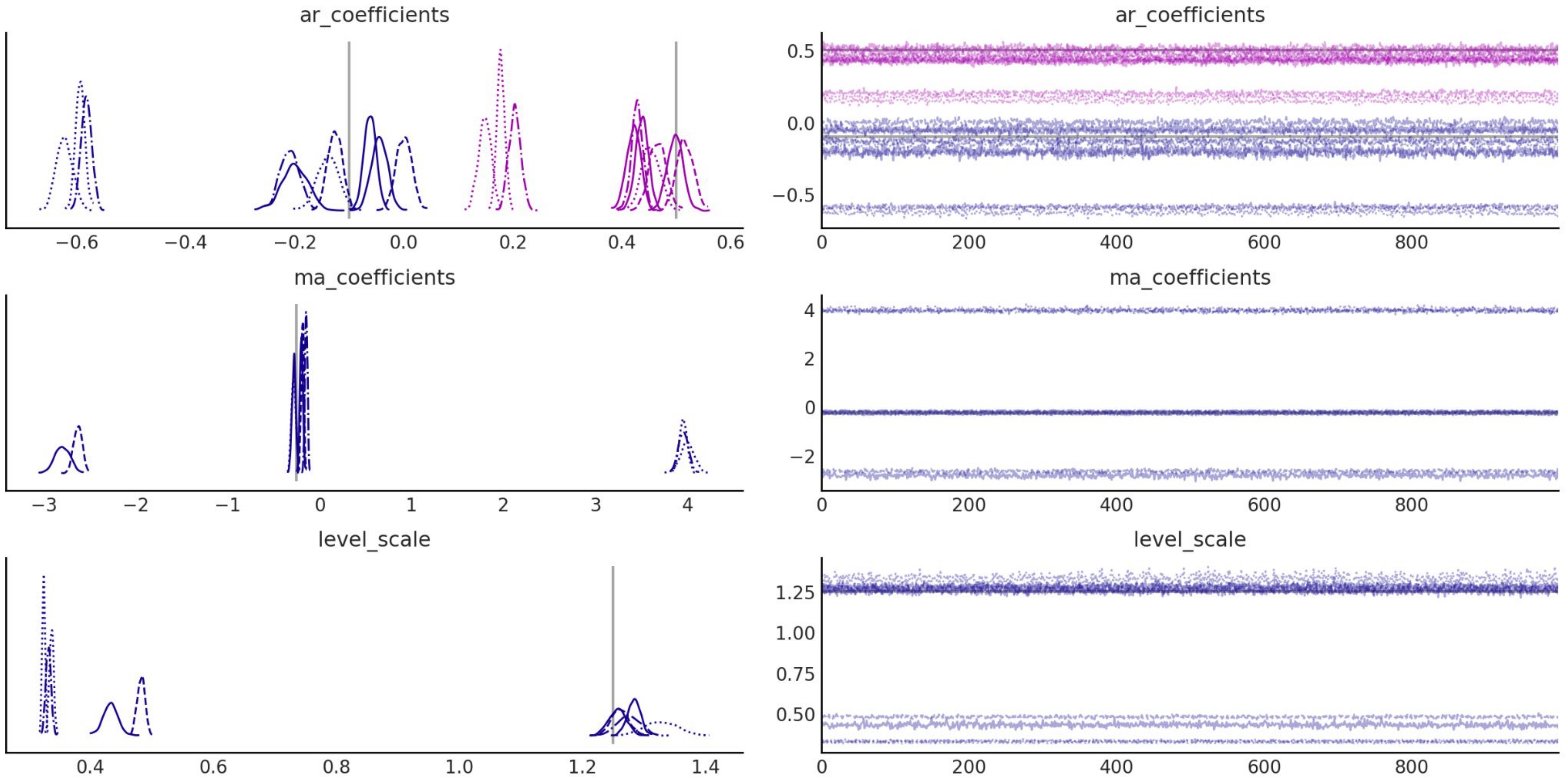
Variational Inference

```
rng, key = jax.random.split(rng, 2)
parameters_mb, elbo_loss_curve = tfp.vi.fit_surrogate_posterior_stateless(
    arma_jd_pinned.unnormalized_log_prob,  # log_prob_fn
    build_surrogate_mb,
    initial_parameters,
    optimizer=optax.chain(optax.clip(10.), optax.adam(0.1)),
    num_steps=num_variational_steps,
    seed=key,
    jit_compile=True)

plt.plot(elbo_loss_curve[10:]);
```



Draw from Surrogate Distribution

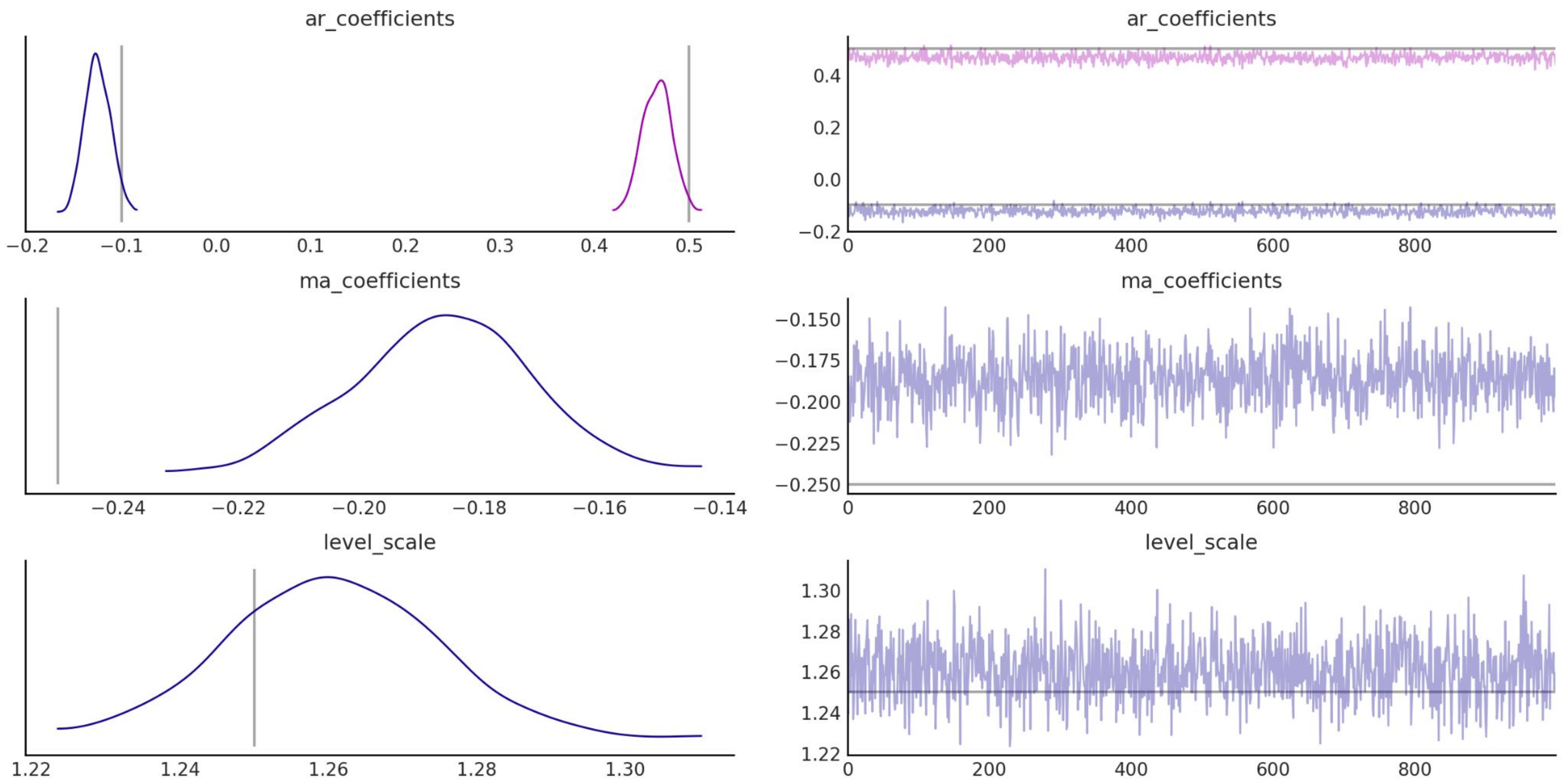
```
rng, key = jax.random.split(rng, 2)
variational_posteriors_mb = build_surrogate_mb(parameters_mb)
q_samples_mb = variational_posteriors_mb.sample(1000, seed=key)
```



Google Research

Take advantage that we trained multiple batches

```
sel_min = np.argmin(elbo_loss_curve[-20:].mean(axis=0))
parameters_mb_sel = jax.tree_map(lambda x: x[sel_min], parameters_mb)
variational_posteriors_mb = build_surrogate_mb(parameters_mb_sel)
q_samples_mb = variational_posteriors_mb.sample(1000, seed=key)
```



Treat each time series as independent

… but with the same configuration

```
arma_sts = sts.AutoregressiveIntegratedMovingAverage(
    ar_order=2, ma_order=1,
    initial_state_prior=X_0,
    observed_time_series=sim_ts
    )

arma_sts.batch_shape  # ==> TensorShape([10])
```

Now training the model you get one set of parameters per time series.

ARIMA(2, 1, 1) as LGSSM

$$\begin{bmatrix} y_{t-1} + \Delta y_t \\ \phi_1 \Delta y_t + \phi_2 \Delta y_{t-1} + \eta'_{t+1} + \theta_1 \eta'_t \\ \phi_2 \Delta y_t + \theta_1 \eta'_{t+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 \\ 0 & \phi_1 & 1 \\ 0 & \phi_2 & 0 \end{bmatrix} \begin{bmatrix} y_{t-1} \\ \Delta y_t \\ \phi_2 \Delta y_{t-1} + \theta_1 \eta'_t \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ \theta_1 \end{bmatrix} \eta'_{t+1}$$

Bayesian Structural Time Series Models

- Generalize many classical models.
  - Regression, Autoregressive (AR, ARIMA, ...), Smoothing (exponential, Holt-Winters, ...)
- Express structural assumptions in model
  - Interpretable models and predictions
  - By combining modular model components

$$F_t = \begin{bmatrix} F_t^1 & 0 \\ 0 & F_t^2 \end{bmatrix}, Q_t = \begin{bmatrix} Q_t^1 & 0 \\ 0 & Q_t^2 \end{bmatrix}, X_t = \begin{bmatrix} X_t^1 \\ X_t^2 \end{bmatrix}$$

$$H_t = [H_t^1 \quad H_t^2], R_t = R_t^1 + R_t^2$$

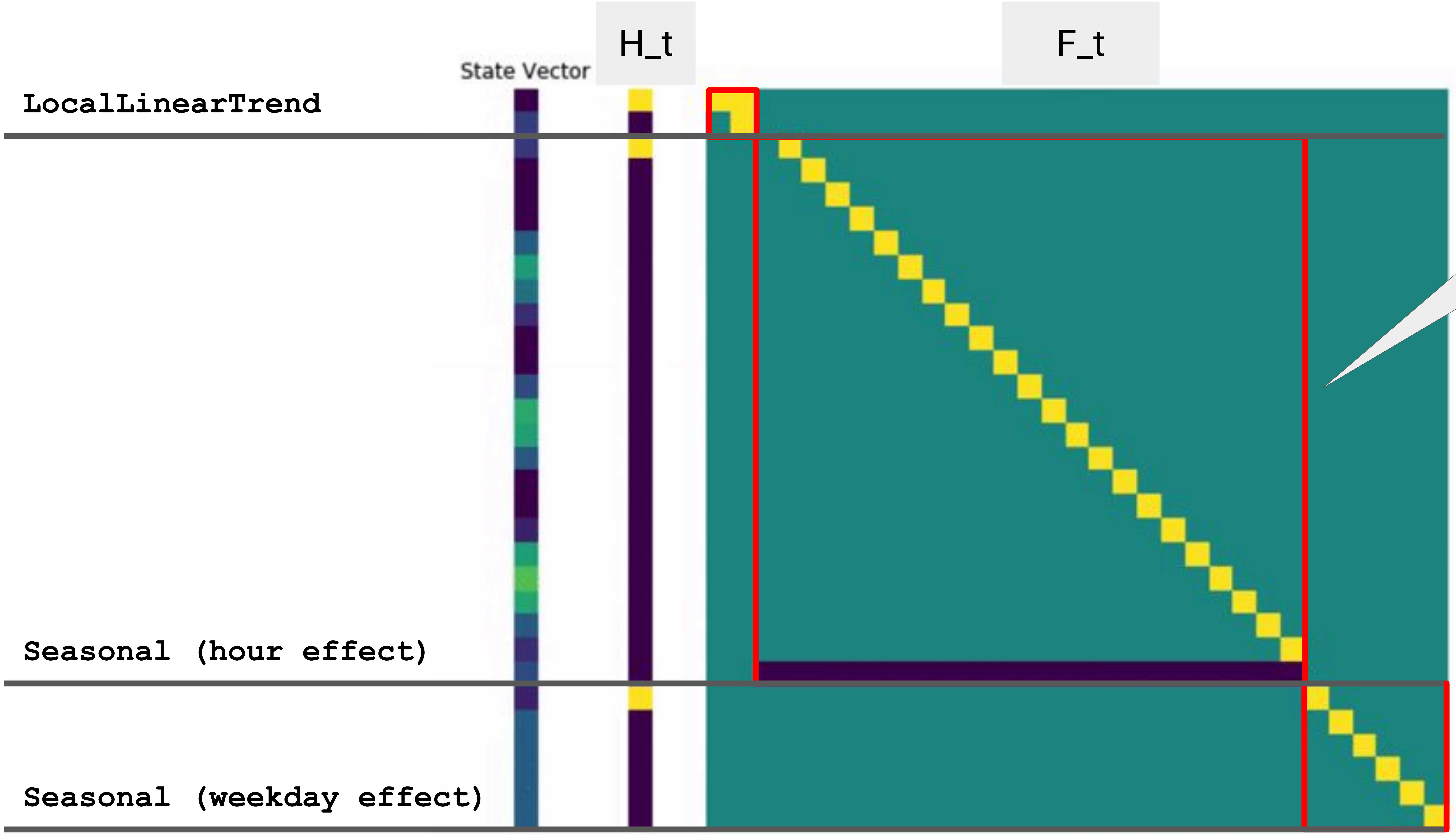
- Probabilistic inference and forecasting

Google Research

LGSSM and BSTS

Google Research

Example of BSTS components

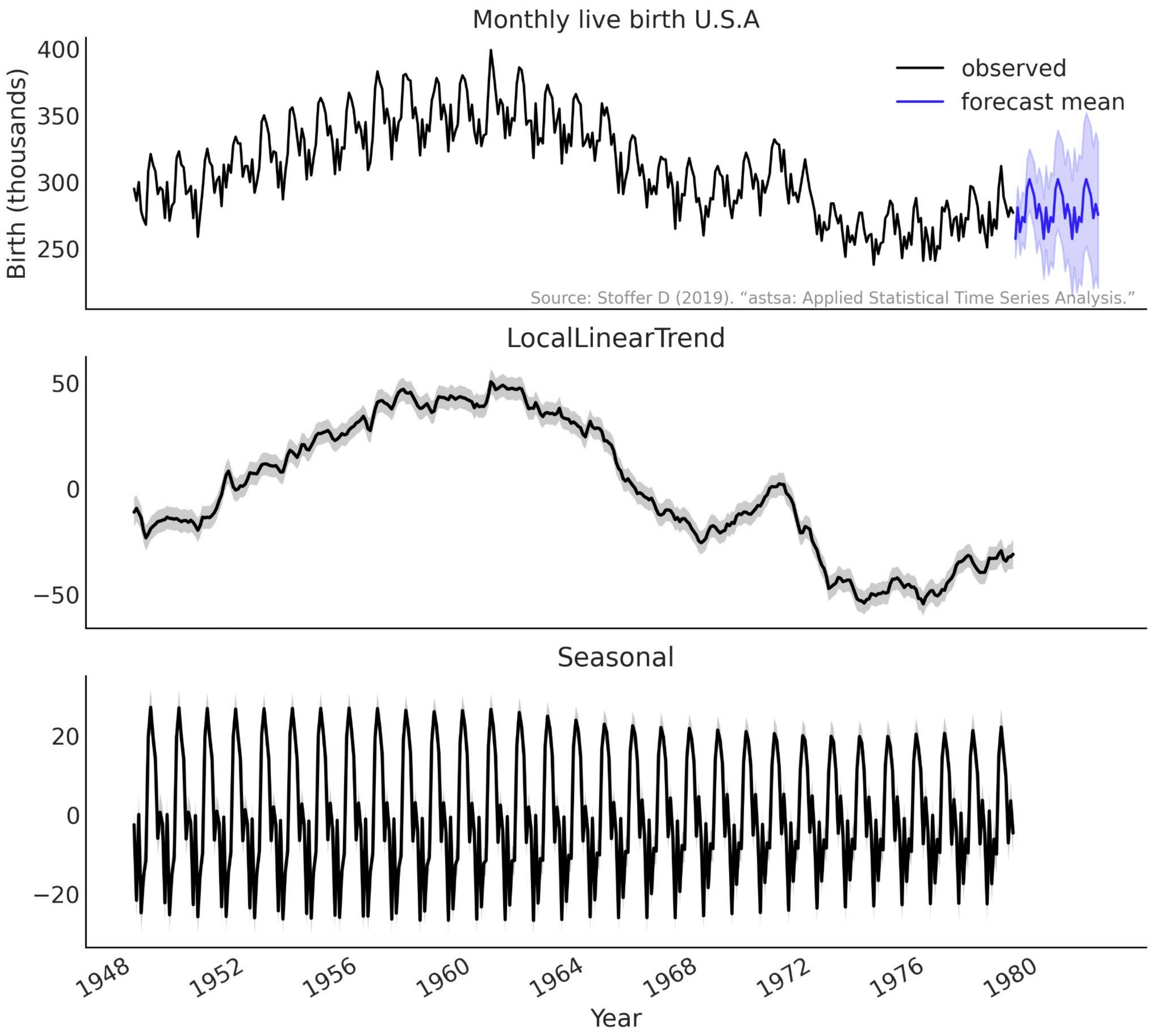


A constant permutation matrix

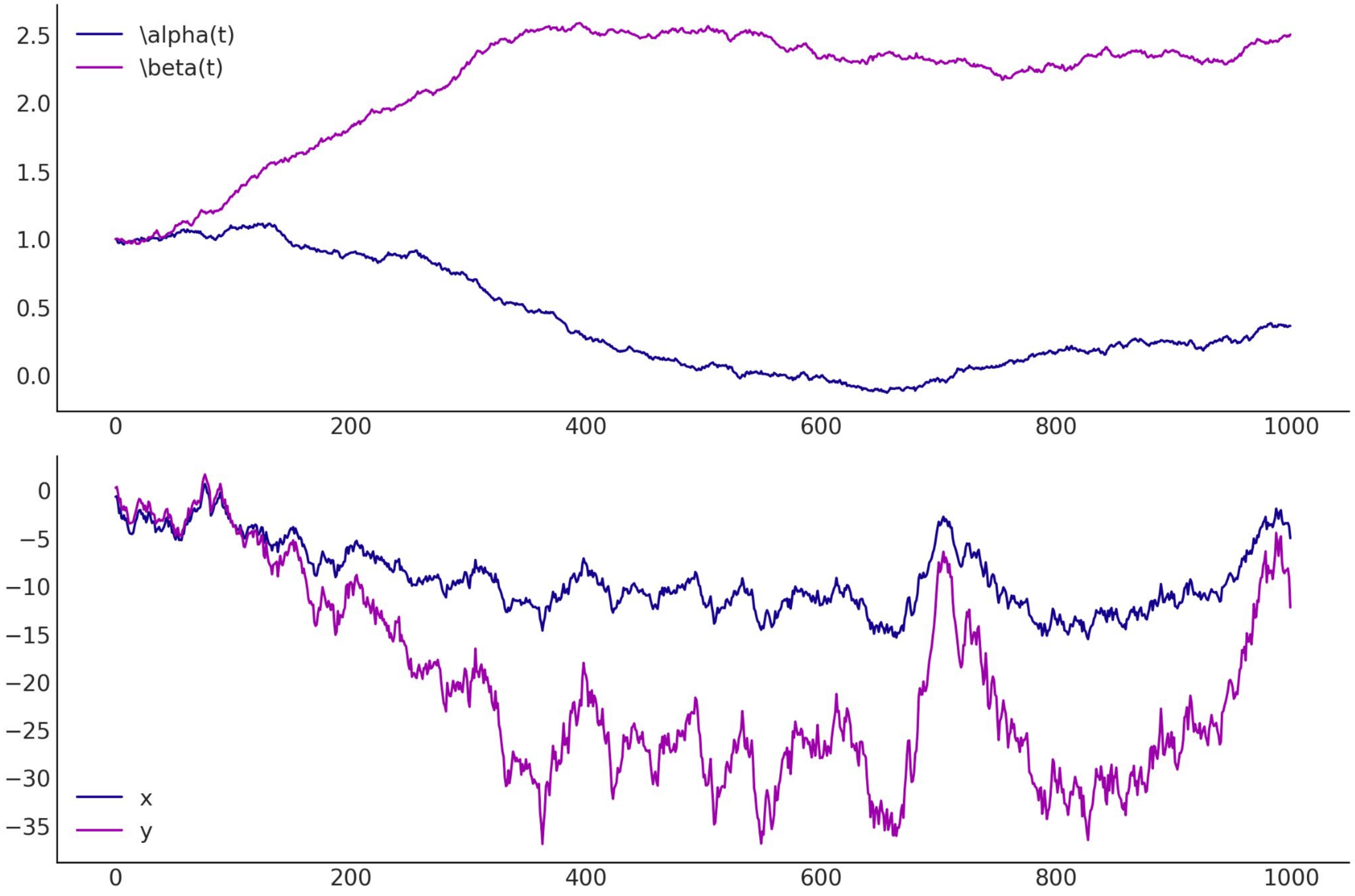
A permutation matrix at every 24 step (i.e., change to a new day), otherwise an identity matrix

Google Research

Exercise: Local Linear Trend with Seasonal effect



Exercise: Dynamic linear regression



Google Research

04

Summary

Other time series models

- Hidden Markov Model
  - SSM with discrete observation
- Gaussian Process
  - Could be reformulated as regression problem or SSM
- Differential equations (ODEs and PDEs)
- Ensemble learning methods (Gradient boosting)
- Deep Learning based approach
  - Dense and Convolution Layer → think regression
  - Recurrent Layer → think State Space Model

Google Research

Time series models

	Deterministic dynamics	**Stochastic dynamics**
Discrete time	automata / discretized ODEs	state space models
Continuous time	ODEs	SDEs

Google Research

Thank you!

Come join me on Friday if you would like some hands on practice writing while_loop/scan in JAX