

# Module 8 : Gaussian Processes for Non-Lir Forecasting

**Course:** Bayesian Regression and Time Series Forecasting for Commodities Trading

---

## Learning Objectives

By the end of this module, you will be able to:

- 1 . **Understand** Gaussian Process regression as a distribution over functions
  - 2 . **Implement** different kernel functions (RBF, Matérn, Periodic) for various patterns
  - 3 . **Build** GP models with PyMC for non-linear commodity price forecasting
  - 4 . **Combine** multiple kernels to capture complex patterns (trend + seasonality + noise)
  - 5 . **Optimize** hyperparameters and interpret their economic meaning
  - 6 . **Apply** sparse approximations for computational efficiency with large datasets
- 

## Why This Matters for Trading

Commodity prices often exhibit **complex non-linear patterns** that linear models cannot capture

- **Copper prices:** Gradual growth during economic expansion, sharp drops during recessions
- **Natural gas:** Seasonal patterns with different amplitudes across years
- **Coffee:** Multi-year cycles driven by planting and harvest dynamics
- **Crude oil:** Regime changes from geopolitical events

Gaussian Processes (GPs) offer several advantages for commodity trading:

- 1 . **Non-parametric flexibility:** Learn complex patterns without assuming functional form
- 2 . **Uncertainty quantification:** Full predictive distributions, not just point estimates
- 3 . **Kernel composition:** Combine seasonal, trend, and cyclical components naturally
- 4 . **Automatic smoothness:** Regularization through kernel hyperparameters
- 5 . **Prior knowledge:** Encode beliefs about smoothness, periodicity, and length scales

**Real-world application:** A copper trader using GPs can:

- Detect regime changes (e.g., China demand shocks)
  - Forecast with uncertainty bands for risk management
  - Identify when patterns deviate from historical behavior
- 

## 1 . What is a Gaussian Process?

### 1 . 1 Intuition: A Distribution Over Functions

**Standard regression:** We assume a functional form (linear, polynomial, etc.) and estimate para

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \epsilon$$

**Gaussian Process:** We place a prior directly over the **space of all possible functions**.

A Gaussian Process is defined by:

1. **Mean function**  $m(x)$ : Expected value of function at each point (often  $m(x) = 0$ )
2. **Covariance function (kernel)**  $k(x, x')$ : How correlated are function values at  $x$  and  $x'$

$$f(x) \sim \mathcal{GP}(m(x), k(x, x'))$$

For any finite set of points  $\{x_1, \dots, x_n\}$ , the function values follow a multivariate normal:

$$\begin{bmatrix} f(x_1) \\ \vdots \\ f(x_n) \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} m(x_1) \\ \vdots \\ m(x_n) \end{bmatrix}, \begin{bmatrix} k(x_1, x_1) & \dots & k(x_1, x_n) \\ \vdots & \ddots & \vdots \\ k(x_n, x_1) & \dots & k(x_n, x_n) \end{bmatrix}\right)$$

## 1.2 The Kernel's Role

The kernel  $k(x, x')$  encodes our **assumptions about smoothness**:

- **Large  $k(x, x')$** : Function values at  $x$  and  $x'$  are highly correlated
- **Small  $k(x, x')$** : Function values are nearly independent

**Trading interpretation:**

- High correlation → Smooth price evolution (gradual trends)
- Low correlation → Rapid price changes (volatile markets)

```
In [ ]: # Setup: Import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats
import pymc as pm
import arviz as az
import warnings
warnings.filterwarnings('ignore')

# Set random seed for reproducibility
np.random.seed(42)

# Plotting style
plt.style.use('seaborn-v0_8-whitegrid')
plt.rcParams['figure.figsize'] = (14, 6)
plt.rcParams['font.size'] = 11

print("Libraries loaded successfully!")
print(f"PyMC version: {pm.__version__}")
```

## 2. Kernel Functions: Encoding Prior Beliefs

Kernels are the heart of GP modeling. Different kernels capture different patterns.

## 2 . 1 Radial Basis Function (RBF) / Squared Exponential

$$k_{\text{RBF}}(x, x') = \sigma^2 \exp\left(-\frac{(x - x')^2}{2\ell^2}\right)$$

**Parameters:**

- $\sigma^2$  (amplitude): Variance of function values
- $\ell$  (length scale): How quickly correlation decays with distance

**Properties:**

- Infinitely differentiable (very smooth)
- Universal approximator
- Good for smooth trends

**Trading use:** Modeling gradual price trends in commodities like copper, gold

## 2 . 2 Matérn Kernel

$$k_{\text{Matérn}}(x, x') = \sigma^2 \frac{2^{1-\nu} \Gamma(\nu)}{\sqrt{2\nu}} \left(\sqrt{2\nu} \frac{|x-x'|}{\ell}\right)^{\nu} K_{\nu}\left(\sqrt{2\nu} \frac{|x-x'|}{\ell}\right)$$

**Parameters:**

- $\nu$  (smoothness): Controls differentiability
  - $\nu = 0.5$ : Rough (exponential kernel)
  - $\nu = 1.5$ : Once differentiable
  - $\nu = 2.5$ : Twice differentiable
  - $\nu \rightarrow \infty$ : Converges to RBF

**Trading use:** More realistic than RBF for commodity prices (finite differentiability)

## 2 . 3 Periodic Kernel

$$k_{\text{Periodic}}(x, x') = \sigma^2 \exp\left(-\frac{2 \sin^2(\pi |x-x'|/p)}{\ell^2}\right)$$

**Parameters:**

- $p$  (period): Length of one cycle
- $\ell$  (length scale): Smoothness within period

**Trading use:** Natural gas seasonality, agricultural commodity cycles

## 2 . 4 Rational Quadratic

$$k_{\text{RQ}}(x, x') = \sigma^2 \left(1 + \frac{(x-x')^2}{2\alpha\ell^2}\right)^{-\alpha}$$

**Properties:** Mixture of RBF kernels with different length scales

**Trading use:** Multi-scale patterns (short-term noise + long-term trends)

```

In [ ]: def visualize_kernels():
    """
    Visualize different kernel functions and their properties.
    """
    # Create x points
    x = np.linspace(0, 10, 200)
    x_ref = 5.0 # Reference point

    # Define kernels
    def rbf_kernel(x, x_ref, length_scale=1.0, amplitude=1.0):
        return amplitude**2 * np.exp(-0.5 * (x - x_ref)**2 / length_scale)

    def matern_kernel(x, x_ref, length_scale=1.0, nu=1.5, amplitude=1.0):
        # Simplified Matérn 3/2
        r = np.abs(x - x_ref)
        sqrt3_r = np.sqrt(3) * r / length_scale
        return amplitude**2 * (1 + sqrt3_r) * np.exp(-sqrt3_r)

    def periodic_kernel(x, x_ref, period=2.0, length_scale=1.0, amplitude=1.0):
        return amplitude**2 * np.exp(-2 * np.sin(np.pi * np.abs(x - x_ref) / period))

    # Plot
    fig, axes = plt.subplots(2, 3, figsize=(16, 9))

    # RBF with different length scales
    ax = axes[0, 0]
    for ls in [0.5, 1.0, 2.0]:
        ax.plot(x, rbf_kernel(x, x_ref, length_scale=ls), label=f' $\ell = \{ls\}$ ')
    ax.axvline(x_ref, color='red', linestyle='--', alpha=0.3)
    ax.set_title('RBF Kernel: Length Scale Effect', fontweight='bold')
    ax.set_xlabel('Distance from reference point')
    ax.set_ylabel('Correlation')
    ax.legend()
    ax.grid(True, alpha=0.3)

    # Matérn with different smoothness
    ax = axes[0, 1]
    # Matérn 1/2 (exponential)
    r = np.abs(x - x_ref)
    ax.plot(x, np.exp(-r / 1.0), label='v = 0.5 (rough)', linewidth=2)
    ax.plot(x, matern_kernel(x, x_ref, length_scale=1.0), label='v = 1.5')
    ax.plot(x, rbf_kernel(x, x_ref, length_scale=1.0), label='v → ∞ (RBF)')
    ax.axvline(x_ref, color='red', linestyle='--', alpha=0.3)
    ax.set_title('Matérn Kernel: Smoothness Parameter', fontweight='bold')
    ax.set_xlabel('Distance from reference point')
    ax.set_ylabel('Correlation')
    ax.legend()
    ax.grid(True, alpha=0.3)

    # Periodic kernel
    ax = axes[0, 2]
    for period in [1.0, 2.0, 3.0]:
        ax.plot(x, periodic_kernel(x, x_ref, period=period), label=f'Period = {period}')
    ax.axvline(x_ref, color='red', linestyle='--', alpha=0.3)
    ax.set_title('Periodic Kernel: Period Effect', fontweight='bold')
    ax.set_xlabel('Distance from reference point')
    ax.set_ylabel('Correlation')
    ax.legend()
    ax.grid(True, alpha=0.3)

```

```

# Sample functions from GP priors
n_samples = 5
x_sample = np.linspace(0, 10, 100)

# RBF samples
ax = axes[1, 0]
K = rbf_kernel(x_sample[:, None], x_sample[None, :], length_scale=1.0)
samples = np.random.multivariate_normal(np.zeros(len(x_sample)), K)
for i in range(n_samples):
    ax.plot(x_sample, samples[i], alpha=0.7, linewidth=1.5)
ax.set_title('Functions from RBF Prior', fontweight='bold')
ax.set_xlabel('x')
ax.set_ylabel('f(x)')
ax.grid(True, alpha=0.3)

# Matérn samples
ax = axes[1, 1]
K = matern_kernel(x_sample[:, None], x_sample[None, :], length_scale=
samples = np.random.multivariate_normal(np.zeros(len(x_sample)), K)
for i in range(n_samples):
    ax.plot(x_sample, samples[i], alpha=0.7, linewidth=1.5)
ax.set_title('Functions from Matérn Prior', fontweight='bold')
ax.set_xlabel('x')
ax.set_ylabel('f(x)')
ax.grid(True, alpha=0.3)

# Periodic samples
ax = axes[1, 2]
K = periodic_kernel(x_sample[:, None], x_sample[None, :], period=2.0,
samples = np.random.multivariate_normal(np.zeros(len(x_sample)), K)
for i in range(n_samples):
    ax.plot(x_sample, samples[i], alpha=0.7, linewidth=1.5)
ax.set_title('Functions from Periodic Prior', fontweight='bold')
ax.set_xlabel('x')
ax.set_ylabel('f(x)')
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

visualize_kernels()

print("\n" + "="*70)
print("KEY INSIGHTS")
print("="*70)
print("""
1. RBF: Very smooth functions. Good for gradual trends.
    - Larger length scale → slower correlation decay → smoother functions

2. Matérn: More realistic than RBF. Finite differentiability.
    - Lower  $\nu$  → rougher functions → better for volatile prices

3. Periodic: Repeating patterns. Perfect for seasonal commodities.
    - Period matches the cycle (e.g., 12 months for nat gas)

**Trading Application**:
    - Copper: Matérn (economic cycles are somewhat rough)
    - Natural Gas: Periodic (winter/summer demand)

```

```
- Gold: RBF (smooth flight-to-safety trends)
""")
```

## 3 . GP Regression with PyMC

### 3 . 1 The GP Regression Model

**Generative model:**

$$\begin{aligned} f(x) &\sim \text{GP}(0, k(x, x')) \quad \text{\textit{(latent function)}} \\ y_i &= f(x_i) + \epsilon_i \quad \text{\textit{(observations)}} \\ \epsilon_i &\sim \mathcal{N}(0, \sigma^2) \quad \text{\textit{(noise)}} \end{aligned}$$

**Posterior predictive** at new points  $x_*$ :

$$p(f_* | x_*, X, y) = \mathcal{N}(\mu_*, \Sigma_*)$$

where:

$$\begin{aligned} \mu_* &= K(x_*, X)[K(X, X) + \sigma^2 I]^{-1} y \\ \Sigma_* &= K(x_*, x_*) - K(x_*, X)[K(X, X) + \sigma^2 I]^{-1} K(X, x_*) \end{aligned}$$

**Key insight:** Posterior mean is a weighted average of training data, with weights determined by similarity.

### 3 . 2 Implementing GP Regression

```
In [ ]: # Generate synthetic data: non-linear price trend
def generate_nonlinear_price_data(n=100, noise=0.5):
    """
    Generate synthetic commodity price data with non-linear trend.
    """
    np.random.seed(42)
    t = np.linspace(0, 4*np.pi, n)

    # True function: smooth non-linear trend
    f_true = 100 + 10*np.sin(t) + 0.5*t**2 + 2*np.cos(2*t)

    # Observed prices (with noise)
    y = f_true + np.random.normal(0, noise, n)

    return t, y, f_true

# Generate data
t_train, y_train, f_true = generate_nonlinear_price_data(n=80, noise=2.0)
t_test = np.linspace(0, 4*np.pi, 200)

# Fit GP model with PyMC
with pm.Model() as gp_model:
    # Hyperparameters (priors)
    l = pm.Gamma("l", alpha=2, beta=1) # Length scale
    η = pm.HalfNormal("η", sigma=5.0) # Amplitude
    σ = pm.HalfNormal("σ", sigma=2.0) # Noise

    # Covariance function
    cov_func = η**2 * pm.gp.cov.ExpQuad(1, ls=l)
```

```

# GP
gp = pm.gp.Marginal(cov_func=cov_func)

# Likelihood
y_obs = gp.marginal_likelihood("y_obs", X=t_train[:, None], y=y_train)

# Sample posterior
trace = pm.sample(1000, tune=1000, chains=2, random_seed=42, progress_bar=True)

# Posterior predictive
with gp_model:
    f_pred = gp.conditional("f_pred", t_test[:, None])
    pred_samples = pm.sample_posterior_predictive(trace, var_names=["f_pred"])

print("\nGP model fitted successfully!")

```

```

In [ ]: # Visualize results
fig, axes = plt.subplots(1, 2, figsize=(16, 6))

# Extract predictions
f_pred_mean = pred_samples.posterior_predictive['f_pred'].mean(dim=['chain'])
f_pred_std = pred_samples.posterior_predictive['f_pred'].std(dim=['chain'])

# Plot 1: GP fit with uncertainty
ax = axes[0]
ax.scatter(t_train, y_train, c='black', s=40, alpha=0.6, label='Observed')
ax.plot(t_test, f_pred_mean, 'blue', linewidth=2, label='GP posterior mean')
ax.fill_between(t_test,
                f_pred_mean - 1.96*f_pred_std,
                f_pred_mean + 1.96*f_pred_std,
                alpha=0.3, color='blue', label='95% credible interval', zorder=2)
ax.set_xlabel('Time', fontsize=12)
ax.set_ylabel('Price ($)', fontsize=12)
ax.set_title('Gaussian Process Regression: Non-Linear Price Trend', fontsize=12)
ax.legend(loc='upper left')
ax.grid(True, alpha=0.3)

# Plot 2: Hyperparameter posteriors
ax = axes[1]
params = ['l', 'η', 'σ']
param_names = ['Length Scale (l)', 'Amplitude (η)', 'Noise (σ)']
for i, (param, name) in enumerate(zip(params, param_names)):
    samples = trace.posterior[param].values.flatten()
    ax.hist(samples, bins=30, alpha=0.6, label=name, density=True)
ax.set_xlabel('Parameter Value', fontsize=12)
ax.set_ylabel('Density', fontsize=12)
ax.set_title('Posterior Distributions of Hyperparameters', fontsize=13, fontweight='bold')
ax.legend()
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Print summary
print("\n" + "="*70)
print("HYPERPARAMETER POSTERIOR")
print("="*70)
for param, name in zip(params, param_names):
    samples = trace.posterior[param].values.flatten()

```

```

print(f"{name:20s}: {np.mean(samples):.3f} ± {np.std(samples):.3f}")
print(f" 95% CI: [{np.percentile(samples, 2.5):.3f}, {np.percentile(

print("\n" + "="*70)
print("INTERPRETATION")
print("="*70)
print(f"""
Length Scale (ℓ): {np.mean(trace.posterior['ℓ'].values):.2f}
    → Prices are correlated over ~{np.mean(trace.posterior['ℓ'].values):.2f}
    → Shorter length scale = more flexible (fits local variations)
    → Longer length scale = smoother (global trends)

Amplitude (η): {np.mean(trace.posterior['η'].values):.2f}
    → Typical deviation from mean is ${np.mean(trace.posterior['η'].values):.
    → Controls vertical scale of fluctuations

Noise (σ): {np.mean(trace.posterior['σ'].values):.2f}
    → Typical observation error is ${np.mean(trace.posterior['σ'].values):.
    → Higher noise → wider uncertainty bands
""")

```

## 4 . Combining Kernels for Complex Patterns

Real commodity prices exhibit **multiple patterns simultaneously**:

- Long-term trends (economic growth)
- Seasonal cycles (weather, demand patterns)
- Short-term fluctuations (supply shocks)

We can **compose kernels** to capture these patterns:

### 4 . 1 Kernel Addition

$$k_{\text{sum}}(x, x') = k_1(x, x') + k_2(x, x')$$

**Effect:** Model captures patterns from **both** kernels independently.

**Example:**  $k_{\text{trend}} + k_{\text{seasonal}}$  → trend plus seasonality

### 4 . 2 Kernel Multiplication

$$k_{\text{product}}(x, x') = k_1(x, x') \times k_2(x, x')$$

**Effect:** Model captures patterns that are **modulated** by each other.

**Example:**  $k_{\text{RBF}} \times k_{\text{periodic}}$  → locally periodic (changing amplitude)

### 4 . 3 Practical Example: Natural Gas Prices

Natural gas exhibits:

- 1 . **Long-term trend:** Growing demand over years
- 2 . **Seasonal pattern:** Winter heating, summer cooling
- 3 . **Noise:** Short-term supply/demand imbalances

**Model:**  $k_{\text{total}} = k_{\text{RBF}}(\text{trend}) + k_{\text{Periodic}}(\text{seasonal}) + k_{\text{White Noise}}$

```
In [ ]: # Generate synthetic natural gas price data
def generate_gas_prices(n=200):
    """
    Simulate natural gas prices with trend + seasonality.
    """
    np.random.seed(42)
    t = np.linspace(0, 4, n) # 4 years, monthly

    # Components
    trend = 3.0 + 0.5*t # Growing demand
    seasonal = 1.5*np.sin(2*np.pi*t) # Annual cycle
    noise = np.random.normal(0, 0.3, n)

    price = trend + seasonal + noise

    return t, price

t_gas, price_gas = generate_gas_prices(n=150)
t_pred_gas = np.linspace(0, 5, 300) # Forecast 1 year ahead

# Fit composite kernel GP
with pm.Model() as composite_model:
    # Trend component (RBF)
    l_trend = pm.Gamma("l_trend", alpha=2, beta=0.5) # Long length scale
    η_trend = pm.HalfNormal("η_trend", sigma=3.0)
    cov_trend = η_trend**2 * pm.gp.cov.ExpQuad(1, ls=l_trend)

    # Seasonal component (Periodic)
    period = 1.0 # Annual (in years)
    l_seasonal = pm.Gamma("l_seasonal", alpha=2, beta=2)
    η_seasonal = pm.HalfNormal("η_seasonal", sigma=2.0)
    cov_seasonal = η_seasonal**2 * pm.gp.cov.Periodic(1, period=period, ls=l_seasonal)

    # Combine kernels (additive)
    cov_total = cov_trend + cov_seasonal

    # Noise
    σ = pm.HalfNormal("σ", sigma=0.5)

    # GP
    gp = pm.gp.Marginal(cov_func=cov_total)

    # Likelihood
    y_obs = gp.marginal_likelihood("y_obs", X=t_gas[:, None], y=price_gas)

    # Sample
    trace_comp = pm.sample(1000, tune=1000, chains=2, random_seed=42, progress_bar=True)

# Predictions
with composite_model:
    f_pred_gas = gp.conditional("f_pred_gas", t_pred_gas[:, None])
    pred_samples_gas = pm.sample_posterior_predictive(trace_comp, var_names=["f_pred_gas"],
                                                       random_seed=42, progress_bar=True)

print("\nComposite kernel GP fitted successfully!")
```

```

In [ ]: # Visualize composite kernel results
fig, axes = plt.subplots(2, 1, figsize=(14, 10))

# Extract predictions
f_mean_gas = pred_samples_gas.posterior_predictive['f_pred_gas'].mean(dim=0)
f_std_gas = pred_samples_gas.posterior_predictive['f_pred_gas'].std(dim=0)

# Plot 1: Full model forecast
ax = axes[0]
ax.scatter(t_gas, price_gas, c='black', s=30, alpha=0.5, label='Observed')
ax.plot(t_pred_gas, f_mean_gas, 'blue', linewidth=2.5, label='GP forecast')
ax.fill_between(t_pred_gas,
                f_mean_gas - 1.96*f_std_gas,
                f_mean_gas + 1.96*f_std_gas,
                alpha=0.25, color='blue', label='95% CI', zorder=1)
ax.axvline(t_gas[-1], color='red', linestyle='--', linewidth=2, label='Forecast time')
ax.set_xlabel('Time (years)', fontsize=12)
ax.set_ylabel('Price ($/MMBtu)', fontsize=12)
ax.set_title('Natural Gas: Composite Kernel (Trend + Seasonality)', fontsize=12)
ax.legend(loc='upper left')
ax.grid(True, alpha=0.3)

# Plot 2: Decomposition (approximate)
# Extract trend and seasonal components separately
ax = axes[1]

# For visualization, we'll refit with individual components
# Trend only
with pm.Model() as trend_only:
    l_t = pm.Gamma("l_t", alpha=2, beta=0.5)
    η_t = pm.HalfNormal("η_t", sigma=3.0)
    cov_t = η_t**2 * pm.gp.cov.ExpQuad(1, ls=l_t)
    σ_t = pm.HalfNormal("σ_t", sigma=1.0)
    gp_t = pm.gp.Marginal(cov_func=cov_t)
    y_t = gp_t.marginal_likelihood("y_t", X=t_gas[:, None], y=price_gas,
                                   trace_t=pm.sample(500, tune=500, chains=1, random_seed=42, progress_bar=False))
    f_trend = gp_t.conditional("f_trend", t_pred_gas[:, None])
    pred_t = pm.sample_posterior_predictive(trace_t, var_names=["f_trend"])

f_trend_mean = pred_t.posterior_predictive['f_trend'].mean(dim=['chain'],

# Plot components
ax.plot(t_pred_gas, f_mean_gas, 'purple', linewidth=2.5, label='Full model forecast')
ax.plot(t_pred_gas, f_trend_mean, 'green', linewidth=2, label='Trend component')
ax.plot(t_pred_gas, f_mean_gas - f_trend_mean + np.mean(price_gas), 'orange',
        linewidth=2, label='Seasonal component (approx)', linestyle='-.')
ax.axvline(t_gas[-1], color='red', linestyle='--', linewidth=2, alpha=0.5)
ax.set_xlabel('Time (years)', fontsize=12)
ax.set_ylabel('Price ($/MMBtu)', fontsize=12)
ax.set_title('Decomposition: Trend + Seasonality', fontsize=13, fontweight='bold')
ax.legend(loc='upper left')
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("\n" + "="*70)
print("COMPOSITE KERNEL INSIGHTS")
print("="*70)

```

```
print("""
1. Trend Component (RBF):
  - Captures long-term growth in demand
  - Smooth evolution over years

2. Seasonal Component (Periodic):
  - Annual cycle (winter peaks, summer lows)
  - Repeating pattern with consistent period

3. Combined Model:
  - Forecasts BOTH trend and seasonality
  - Uncertainty grows in forecast period (as expected)
  - Can detect when seasonality amplitude changes

**Trading Application**:
  - Buy in summer (seasonal low), sell in winter (seasonal high)
  - Adjust positions based on trend direction
  - Size trades using uncertainty bands (wider bands = smaller positions)
""")
```

# 5 . Hyperparameter Optimization and Interpretation

## 5 . 1 The Role of Hyperparameters

GP hyperparameters control the **prior** over functions. They determine:

### 1 . Length scale ( $\ell$ ): How far we extrapolate correlations

- Small  $\ell$  → wiggly functions (overfitting risk)
- Large  $\ell$  → smooth functions (underfitting risk)

### 2 . Amplitude ( $\eta$ ): Vertical scale of variation

- Controls prior variance of function values

### 3 . Noise ( $\sigma$ ): Observation error

- High  $\sigma$  → less trust in individual observations
- Low  $\sigma$  → interpolate through points

## 5 . 2 Learning Hyperparameters

Two approaches:

### Maximum Likelihood (Type-II ML):

- Maximize marginal likelihood  $p(y|X, \theta)$  w.r.t. hyperparameters  $\theta$
- Fast but gives point estimates (no uncertainty)

### Full Bayes:

- Place priors on hyperparameters
- Sample posterior  $p(\theta | X, y)$  using MCMC
- Slower but accounts for hyperparameter uncertainty

PyMC uses **full Bayes** by default, giving us uncertainty about hyperparameters.

```
In [ ]: # Analyze hyperparameter posteriors from composite model
fig, axes = plt.subplots(2, 3, figsize=(16, 9))

params = ['l_trend', 'eta_trend', 'l_seasonal', 'eta_seasonal', 'sigma']
param_names = ['Trend Length Scale', 'Trend Amplitude', 'Seasonal Length',
               'Seasonal Amplitude', 'Noise']

for i, (param, name) in enumerate(zip(params, param_names)):
    ax = axes.flatten()[i]

    samples = trace_comp.posterior[param].values.flatten()

    # Histogram
    ax.hist(samples, bins=30, alpha=0.7, density=True, color='steelblue',

    # Add mean line
    mean_val = np.mean(samples)
    ax.axvline(mean_val, color='red', linestyle='--', linewidth=2, label=
```

```

# 95% CI
ci_low, ci_high = np.percentile(samples, [2.5, 97.5])
ax.axvline(ci_low, color='orange', linestyle=':', linewidth=1.5)
ax.axvline(ci_high, color='orange', linestyle=':', linewidth=1.5, label='')

ax.set_xlabel('Value', fontsize=11)
ax.set_ylabel('Density', fontsize=11)
ax.set_title(name, fontsize=12, fontweight='bold')
ax.legend(fontsize=9)
ax.grid(True, alpha=0.3)

# Hide extra subplot
axes.flatten()[5].axis('off')

plt.tight_layout()
plt.show()

# Print summary table
print("\n" + "="*70)
print("HYPERPARAMETER POSTERIOR SUMMARY")
print("="*70)
print(f"{'Parameter':<25} {'Mean':>10} {'Std':>10} {'95% CI':>20}")
print("-"*70)

for param, name in zip(params, param_names):
    samples = trace_comp.posterior[param].values.flatten()
    mean_val = np.mean(samples)
    std_val = np.std(samples)
    ci = np.percentile(samples, [2.5, 97.5])
    print(f"{'name':<25} {'mean_val':>10.3f} {'std_val':>10.3f} [{ci[0]:>6.3f},

print("\n" + "="*70)
print("ECONOMIC INTERPRETATION")
print("="*70)

l_trend_mean = np.mean(trace_comp.posterior['l_trend'].values)
l_seasonal_mean = np.mean(trace_comp.posterior['l_seasonal'].values)
eta_trend_mean = np.mean(trace_comp.posterior['eta_trend'].values)
eta_seasonal_mean = np.mean(trace_comp.posterior['eta_seasonal'].values)

print(f"""
Trend Length Scale: {l_trend_mean:.2f} years
    → Trend changes smoothly over ~{l_trend_mean:.2f} years
    → Long length scale = persistent directional moves

Seasonal Length Scale: {l_seasonal_mean:.2f}
    → Controls smoothness WITHIN each season
    → Higher value = smoother seasonal curves

Amplitude Ratio (Trend/Seasonal): {eta_trend_mean/eta_seasonal_mean:.2f}
    → Trend is {eta_trend_mean/eta_seasonal_mean:.2f}x more important than seas
    → Useful for position sizing (trend vs mean-reversion strategies)

**Trading Insight**:
If trend amplitude >> seasonal amplitude:
    → Focus on trend-following strategies
    → Use seasonality for entry/exit timing

If seasonal amplitude >> trend amplitude:

```

```
→ Focus on seasonal trading (buy summer, sell winter)
→ Trend is less reliable for directional bets
"""
```

## 6 . Sparse Gaussian Processes for Large Datasets

### 6 . 1 The Computational Challenge

Standard GP regression requires:

- **Covariance matrix:**  $K \in \mathbb{R}^{n \times n}$
- **Matrix inversion:**  $O(n^3)$  complexity
- **Memory:**  $O(n^2)$  storage

For  $n > 1000$  observations, this becomes prohibitive.

### 6 . 2 Sparse Approximations

**Idea:** Use  $m \ll n$  **inducing points** to approximate the full GP.

**Inducing points**  $\mathbf{u}$  are a set of latent function values at locations  $\mathbf{Z}$  that summarize GP.

**Complexity reduction:**

- Standard GP:  $O(n^3)$
- Sparse GP:  $O(nm^2)$  where  $m \ll n$

**Popular methods:**

- 1 . **FITC** (Fully Independent Training Conditional)
- 2 . **SVGP** (Stochastic Variational GP)
- 3 . **VFE** (Variational Free Energy)

PyMC supports sparse GPs through `pm.gp.Marginal` with inducing points.

### 6 . 3 Practical Example: High-Frequency Commodity Data

```
In [ ]: # Generate large dataset (daily data for several years)
def generate_large_copper_data(n=2000):
    """
    Simulate daily copper prices with trend and cycles.
    """
    np.random.seed(42)
    t = np.linspace(0, 10, n) # 10 years daily

    # Components
    trend = 6000 + 200*t + 50*np.sin(0.5*t) # Long-term trend
    cycle = 500*np.sin(2*np.pi*t/2.5) # Multi-year cycle
    noise = np.random.normal(0, 100, n)

    price = trend + cycle + noise

    return t, price
```

```

# Generate data
t_copper, price_copper = generate_large_copper_data(n=1500)

print(f"Dataset size: {len(t_copper)} observations")
print(f"Standard GP would require: {len(t_copper)**2 * 8 / 1e9:.2f} GB fo
print(f"Complexity:  $O(n^3) = O(\text{len}(t\_copper)**3 / 1e9:.2f)$  billion operat
print("\nUsing sparse approximation with m=50 inducing points...\n")

# Sparse GP with inducing points
m = 50 # Number of inducing points
Z = np.linspace(t_copper.min(), t_copper.max(), m)[: , None] # Inducing p

with pm.Model() as sparse_gp:
    # Hyperparameters
    l = pm.Gamma("l", alpha=2, beta=0.5)
    η = pm.HalfNormal("η", sigma=500)
    σ = pm.HalfNormal("σ", sigma=150)

    # Covariance function
    cov = η**2 * pm.gp.cov.Matern52(1, ls=l)

    # Sparse GP (using inducing points approximation)
    gp_sparse = pm.gp.MarginalSparse(cov_func=cov, approx="FITC")

    # Likelihood with inducing points
    y_obs = gp_sparse.marginal_likelihood(
        "y_obs",
        X=t_copper[:, None],
        Xu=Z, # Inducing points
        y=price_copper,
        sigma=σ
    )

    # Sample (much faster than full GP!)
    trace_sparse = pm.sample(500, tune=500, chains=2, random_seed=42, progres

print("\nSparse GP fitted successfully!")
print(f"Complexity reduction: {len(t_copper)**3 / (len(t_copper) * m**2):

```

```

In [ ]: # Predictions with sparse GP
t_pred_copper = np.linspace(0, 11, 500)

with sparse_gp:
    f_pred_copper = gp_sparse.conditional("f_pred_copper", t_pred_copper[
    pred_copper = pm.sample_posterior_predictive(trace_sparse, var_names=
        random_seed=42, progres

# Visualize
fig, axes = plt.subplots(2, 1, figsize=(14, 10))

f_mean_copper = pred_copper.posterior_predictive['f_pred_copper'].mean(dim=
f_std_copper = pred_copper.posterior_predictive['f_pred_copper'].std(dim=

# Plot 1: Full view
ax = axes[0]
ax.scatter(t_copper[:10], price_copper[:10], c='black', s=15, alpha=0.3
ax.scatter(Z.flatten(), price_copper[np.searchsorted(t_copper, Z.flatten(
    c='red', s=80, marker='x', linewidths=2, label=f'{m} Inducing
ax.plot(t_pred_copper, f_mean_copper, 'blue', linewidth=2, label='Sparse

```

```

ax.fill_between(t_pred_copper,
                f_mean_copper - 1.96*f_std_copper,
                f_mean_copper + 1.96*f_std_copper,
                alpha=0.2, color='blue', label='95% CI')
ax.axvline(t_copper[-1], color='green', linestyle='--', linewidth=2, label='')
ax.set_xlabel('Time (years)', fontsize=12)
ax.set_ylabel('Price ($/ton)', fontsize=12)
ax.set_title(f'Sparse GP: Copper Prices ({len(t_copper)} observations, {m} inducing points)',
             fontsize=13, fontweight='bold')
ax.legend(loc='upper left')
ax.grid(True, alpha=0.3)

# Plot 2: Zoom on forecast region
ax = axes[1]
forecast_mask = t_copper > 8
pred_mask = t_pred_copper > 8

ax.scatter(t_copper[forecast_mask], price_copper[forecast_mask], c='black',
           label='Recent observations')
ax.plot(t_pred_copper[pred_mask], f_mean_copper[pred_mask], 'blue', linewidth=2)
ax.fill_between(t_pred_copper[pred_mask],
                f_mean_copper[pred_mask] - 1.96*f_std_copper[pred_mask],
                f_mean_copper[pred_mask] + 1.96*f_std_copper[pred_mask],
                alpha=0.3, color='blue', label='95% CI')
ax.axvline(t_copper[-1], color='green', linestyle='--', linewidth=2, label='')
ax.set_xlabel('Time (years)', fontsize=12)
ax.set_ylabel('Price ($/ton)', fontsize=12)
ax.set_title('Forecast Region (Last 2 years + 1 year ahead)', fontsize=13)
ax.legend()
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("\n" + "="*70)
print("SPARSE GP ADVANTAGES")
print("="*70)
print(f"""
1. Computational Efficiency:
    - Full GP:  $O(n^3) = O(\{len(t\_copper)\}^3 / 1e9 : .1f)B$  operations
    - Sparse GP:  $O(nm^2) = O(\{len(t\_copper)\} * m^2 / 1e6 : .1f)M$  operations
    - Speedup:  $\sim \{len(t\_copper)\}^2 / m^2 : .0f \times$ 

2. Memory Savings:
    - Full GP:  $\{len(t\_copper)\}^2 * 8 / 1e9 : .2f$  GB
    - Sparse GP:  $\{len(t\_copper)\} * m * 8 / 1e6 : .2f$  MB
    - Reduction:  $\sim \{len(t\_copper)\} / m : .0f \times$ 

3. Accuracy:
    - With {m} well-placed inducing points, captures main patterns
    - Minor loss in fit quality vs full GP
    - Inducing point placement matters!

When to Use Sparse GPs:
    - Daily/hourly commodity price data ( $n > 1000$ )
    - Real-time trading systems (need fast inference)
    - Limited computational resources

Inducing Point Selection:
    - Uniform spacing (simple, works well)

```

```
- K-means clustering of inputs (adaptive)
- Learn locations (more complex, marginal benefit)
""")
```

## 7 . Practical Example: Copper Price Forecasting with Regime Detection

Let's apply everything we've learned to a realistic copper trading scenario:

**Challenge:** Copper prices exhibit:

- 1 . Long-term trends (economic cycles)
- 2 . Regime changes (China demand shocks, supply disruptions)
- 3 . Non-linear patterns

**Solution:** Use GP with Matérn kernel to capture regime changes and forecast with uncertainty.

```
In [ ]: # Generate realistic copper price data with regime change
def generate_copper_with_regimes(n=300):
    """
    Simulate copper prices with a regime change (China boom).
    """
    np.random.seed(42)
    t = np.linspace(0, 10, n)

    # Regime 1 (years 0-5): Moderate growth
    regime1 = 6000 + 100*t[:n//2] + np.random.normal(0, 200, n//2)

    # Regime 2 (years 5-10): China boom - sharp rise then volatility
    t2 = t[n//2:]
    regime2 = 6500 + 800*np.tanh((t2 - 5)*0.8) + 300*np.sin(2*t2) + np.ra

    price = np.concatenate([regime1, regime2])

    return t, price

# Generate data
t_cu, price_cu = generate_copper_with_regimes(n=250)

# Split: Train on first 80%, forecast last 20%
split_idx = int(0.8 * len(t_cu))
t_train_cu = t_cu[:split_idx]
price_train_cu = price_cu[:split_idx]
t_test_cu = t_cu[split_idx:]
price_test_cu = price_cu[split_idx:]

# Fit GP with Matérn kernel (captures regime changes better than RBF)
with pm.Model() as copper_model:
    # Hyperparameters
    l = pm.Gamma("l", alpha=2, beta=1)
    η = pm.HalfNormal("η", sigma=800)
    σ = pm.HalfNormal("σ", sigma=300)

    # Matérn 3/2 kernel (less smooth than RBF, captures regime changes)
    cov = η**2 * pm.gp.cov.Matern32(1, ls=l)

# GP
```

```

gp_cu = pm.gp.Marginal(cov_func=cov)

# Likelihood
y_obs = gp_cu.marginal_likelihood("y_obs", X=t_train_cu[:, None], y=p

# Sample
trace_cu = pm.sample(1000, tune=1000, chains=2, random_seed=42, progr

# Forecast
t_forecast_cu = np.linspace(t_cu.min(), t_cu.max() + 1, 400)

with copper_model:
    f_forecast_cu = gp_cu.conditional("f_forecast_cu", t_forecast_cu[:, N
    pred_cu_final = pm.sample_posterior_predictive(trace_cu, var_names=["
                                                random_seed=42, progr

print("\nCopper forecasting model fitted!")

```

```

In [ ]: # Visualize final copper forecast
fig, axes = plt.subplots(2, 1, figsize=(14, 10))

f_mean_cu = pred_cu_final.posterior_predictive['f_forecast_cu'].mean(dim=
f_std_cu = pred_cu_final.posterior_predictive['f_forecast_cu'].std(dim=[

# Plot 1: Full history + forecast
ax = axes[0]
ax.scatter(t_train_cu, price_train_cu, c='black', s=40, alpha=0.6, label=
ax.scatter(t_test_cu, price_test_cu, c='red', s=40, alpha=0.8, label='Hel
ax.plot(t_forecast_cu, f_mean_cu, 'blue', linewidth=2.5, label='GP foreca
ax.fill_between(t_forecast_cu,
                f_mean_cu - 1.96*f_std_cu,
                f_mean_cu + 1.96*f_std_cu,
                alpha=0.25, color='blue', label='95% CI', zorder=1)
ax.axvline(t_train_cu[-1], color='green', linestyle='--', linewidth=2,
            label='Train/test split', alpha=0.6)
ax.axvline(5, color='orange', linestyle=':', linewidth=2,
            label='Regime change (China boom)', alpha=0.6)
ax.set_xlabel('Time (years)', fontsize=12)
ax.set_ylabel('Price ($/ton)', fontsize=12)
ax.set_title('Copper Price Forecast: GP with Matérn Kernel', fontsize=13,
ax.legend(loc='upper left', fontsize=10)
ax.grid(True, alpha=0.3)

# Plot 2: Prediction intervals and uncertainty growth
ax = axes[1]
ax.plot(t_forecast_cu, f_std_cu, 'purple', linewidth=2.5, label='Uncertai
ax.axvline(t_train_cu[-1], color='green', linestyle='--', linewidth=2,
            label='Train/test split', alpha=0.6)
ax.fill_between(t_forecast_cu, 0, f_std_cu, alpha=0.3, color='purple')
ax.set_xlabel('Time (years)', fontsize=12)
ax.set_ylabel('Predictive Uncertainty ($/ton)', fontsize=12)
ax.set_title('Forecast Uncertainty Growth', fontsize=13, fontweight='bold
ax.legend()
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Calculate forecast accuracy on test set
test_mask = (t_forecast_cu >= t_test_cu[0]) & (t_forecast_cu <= t_test_cu

```

```

f_test_mean = f_mean_cu[test_mask]
f_test_std = f_std_cu[test_mask]

# Interpolate predictions to match test times
from scipy.interpolate import interp1d
interp_mean = interp1d(t_forecast_cu, f_mean_cu, kind='linear')
interp_std = interp1d(t_forecast_cu, f_std_cu, kind='linear')

pred_test_mean = interp_mean(t_test_cu)
pred_test_std = interp_std(t_test_cu)

mae = np.mean(np.abs(price_test_cu - pred_test_mean))
rmse = np.sqrt(np.mean((price_test_cu - pred_test_mean)**2))
coverage = np.mean((price_test_cu >= pred_test_mean - 1.96*pred_test_std)
                    (price_test_cu <= pred_test_mean + 1.96*pred_test_std))

print("\n" + "="*70)
print("FORECAST EVALUATION ON HELD-OUT TEST SET")
print("="*70)
print(f"MAE:  ${mae:.2f}/ton")
print(f"RMSE: ${rmse:.2f}/ton")
print(f"95% Interval Coverage: {coverage:.1%} (should be ~95% for calibration)")

print("\n" + "="*70)
print("TRADING INSIGHTS")
print("="*70)
print(f"""
1. **Regime Detection**:
    - GP automatically adapts to regime change at year 5
    - Matérn kernel allows for "kinks" that RBF would smooth over

2. **Uncertainty Quantification**:
    - Uncertainty grows as we forecast further (as it should!)
    - Narrower bands in stable periods, wider in volatile periods
    - Use for position sizing: wide bands → reduce position

3. **Forecast Calibration**:
    - {coverage:.1%} of actual prices fall in 95% interval
    - Well-calibrated forecasts enable proper risk management

4. **Trading Strategies**:
    - **Trend following**: When forecast slope is positive, go long
    - **Mean reversion**: When price exits upper band, consider shorting
    - **Volatility trading**: Wider bands → higher implied vol → option strategy

5. **Risk Management**:
    - VaR from predictive distribution: 5th percentile of forecast
    - Position sizing: Inverse to forecast uncertainty
    - Stop-losses: Place outside 95% CI to avoid noise-triggered exits
""")

```

## 8 . Summary: When to Use Gaussian Processes

### 8 . 1 GP Strengths

Strength	Why It Matters for Trading
Non-parametric	Don't need to assume functional form (linear, quadratic, etc.)

Strength	Why It Matters for Trading
Full distributions	Get predictive uncertainty for every forecast
Kernel flexibility	Encode domain knowledge (smoothness, periodicity)
Composability	Combine trend + seasonality + noise naturally
Bayesian	Hyperparameter uncertainty propagates to forecasts
Interpolation	Excellent at filling gaps in historical data

## 8 . 2 GP Limitations

Limitation	Workaround
Computational cost	Use sparse approximations (FITC, SVGP)
Stationary kernels	Most kernels assume stationary processes (can use non-stationary kernels)
High dimensions	GPs struggle with many input features (use ARD or feature selection)
Extrapolation	Uncertainty grows quickly beyond data (by design!)

## 8 . 3 Decision Guide: GP vs Other Methods

Use GP when:

- ✓ Small to medium datasets ( $n < 10,000$ )
- ✓ Non-linear patterns expected
- ✓ Uncertainty quantification critical
- ✓ Prior knowledge about smoothness/seasonality
- ✓ Interpolation important (missing data)

Consider alternatives when:

- ✗ Very large datasets ( $n > 100,000$ ) → Neural networks, ensemble methods
- ✗ Many input features ( $p > 50$ ) → Random forests, boosting
- ✗ Only point predictions needed → Linear models, ARIMA
- ✗ Real-time constraints → Faster methods (local models)

## 8 . 4 Key Takeaways

- Kernels encode assumptions:** Choose kernels that match your beliefs about price dynamics
- Combine kernels:** Real commodities have multiple patterns (trend + seasonality + noise)
- Hyperparameters matter:** Length scales, amplitudes have economic interpretations
- Sparse GPs scale:** Use inducing points for large datasets
- Uncertainty is valuable:** GP predictive distributions enable sophisticated risk management
- Matérn > RBF for trading:** Finite differentiability better matches real price dynamics
- Regime changes:** GPs adapt automatically if length scale is not too large

**Next steps:** In Module 9, we'll learn how to model **time-varying volatility** using stochastic volatility models and GARCH, building on the uncertainty quantification skills from GPs.

---

## Knowledge Check Quiz

Test your understanding. Answers in the next cell.

**Q 1 :** What does the length scale parameter  $\ell$  in a GP kernel control?

- A) The vertical scale of the function
- B) How far apart points can be and still be correlated
- C) The noise level in observations
- D) The period of oscillations

**Q 2 :** For modeling seasonal commodity prices (e.g., natural gas), which kernel is most appropriate?

- A) RBF only
- B) White noise only
- C) RBF + Periodic
- D) Linear kernel

**Q 3 :** Sparse GPs with  $m$  inducing points reduce complexity from  $O(n^3)$  to:

- A)  $O(n^2)$
- B)  $O(nm^2)$
- C)  $O(m^3)$
- D)  $O(n + m)$

**Q 4 :** Why is the Matérn kernel often better than RBF for commodity prices?

- A) It's faster to compute
- B) It allows for regime changes and finite differentiability
- C) It requires fewer hyperparameters
- D) It always gives narrower uncertainty bands

**Q 5 :** If a GP forecast shows very wide uncertainty bands in the future, you should:

- A) Increase position size (more opportunity)
- B) Reduce position size (more risk)
- C) Ignore the forecast
- D) Trade only at the mean prediction

```
In [ ]: # Quiz Answers
print("="*70)
print("QUIZ ANSWERS")
print("="*70)
print("""
Q1: B) How far apart points can be and still be correlated
      The length scale  $\ell$  controls the "reach" of correlation. Larger  $\ell$  mean
```

points further apart are still correlated (smoother functions).

Q2: C) RBF + Periodic

RBF captures the trend, Periodic captures the annual cycle. Combining them via kernel addition gives trend + seasonality.

Q3: B)  $O(nm^2)$

Sparse GPs reduce complexity from  $O(n^3)$  to  $O(nm^2)$  where  $m \ll n$  is the number of inducing points. This enables scaling to thousands of obser

Q4: B) It allows for regime changes and finite differentiability

RBF is infinitely differentiable (unrealistically smooth). Matérn with  $\nu=1.5$  or  $2.5$  is only finitely differentiable, matching real price dyn with occasional kinks and regime changes.

Q5: B) Reduce position size (more risk)

Wide uncertainty bands mean high forecast uncertainty. This translate to higher risk. Bayesian risk management says: uncertainty  $\rightarrow$  smaller Use Kelly criterion or volatility-adjusted sizing.

""")

---

## Exercises

Complete these in `exercises.ipynb`:

### Exercise 1 : Kernel Exploration (Easy)

Generate synthetic data with known structure (linear trend + sine wave + noise). Fit GPs with:

- RBF kernel only
- Periodic kernel only
- RBF + Periodic

Compare forecast accuracy. Which kernel composition works best?

### Exercise 2 : Hyperparameter Sensitivity (Medium)

Using the copper price data:

- 1 . Fit GPs with different prior choices for length scale
- 2 . Plot how posterior predictions change
- 3 . Calculate forecast coverage on test set
- 4 . What happens with too-small vs too-large length scale priors?

### Exercise 3 : Trading Strategy with GPs (Hard)

Implement a GP-based trading strategy:

- 1 . Generate synthetic commodity prices with trend + noise
- 2 . Use rolling window GP forecasts

3 . Trade signals:

- Go long if forecast mean  $>$  current price +  $1 \sigma$
- Go short if forecast mean  $<$  current price -  $1 \sigma$

4 . Position sizing: Inverse to forecast uncertainty

5 . Backtest and calculate Sharpe ratio

Compare to:

- Buy-and-hold
- Simple moving average crossover

## Exercise 4 : Multi-Output GP (Advanced)

Build a multi-output GP for **pairs trading**:

- Model two correlated commodities (e.g., WTI crude + Brent crude)
- Use cross-covariance between outputs
- Trade the spread when it deviates from GP forecast

---

## Next Module Preview

In **Module 9 : Volatility Modeling and Uncertainty Quantification**, we'll learn:

- Time-varying volatility (GARCH, stochastic volatility)
- Epistemic vs aleatoric uncertainty
- VaR and CVaR from Bayesian posteriors
- Volatility forecasting for risk management
- Practical example: Crude oil volatility regime detection

---

Module 8 Complete