# Module 1 : Foundations of Bayesian Thinki for Trading

**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** the difference between Bayesian and Frequentist approaches in a financial co
2. **Apply** Bayes' theorem to calculate updated beliefs about market conditions
3. **Interpret** probability as a degree of belief about uncertain market outcomes
4. **Implement** basic Bayesian calculations using Python
5. **Recognize** when Bayesian methods are preferable for trading applications

---

## Why This Matters for Trading

Markets are fundamentally uncertain. Traditional statistical methods often assume we have unlin data and that parameters are fixed constants waiting to be discovered. But in commodities tradi

- **Regimes change**: The dynamics of crude oil in 2020 are different from 2 0 1
- **Data is limited**: We may only have 50 years of wheat prices, but wheat has been tr millennia
- **Uncertainty matters**: A forecast of "$80/barrel" is useless without knowing how confide
- **Prior knowledge exists**: Expert traders have beliefs about seasonality, mean reversion, an demand

Bayesian methods let us:

- **Quantify uncertainty** in every prediction
- **Incorporate domain expertise** through prior distributions
- **Update beliefs systematically** as new data arrives
- **Make decisions** that account for model uncertainty

---

## 1. The Two Philosophies of Probability

### 1.1 Frequentist View

In the **frequentist** view, probability represents the long-run frequency of events:

- "There's a 60% chance of rain" means "In similar atmospheric conditions, it rains 6 the time"
- Parameters (like true average return) are **fixed but unknown constants**

- Data are random samples from a population
- We use data to estimate the fixed parameters

**The problem for trading**: We can't repeat the 2008 financial crisis. Each market ever
unique. What does "probability of a recession" mean in frequentist terms?

## 1.2 Bayesian View

In the **Bayesian** view, probability represents our **degree of belief**:

- "There's a 60% chance of rain" means "Given my current information, I believe there's 60% chance of rain"
- Parameters are **random variables** with probability distributions
- We start with **prior beliefs** and update them with data
- The result is a **posterior distribution** reflecting updated beliefs

**The advantage for trading**: We can talk about "the probability that crude oil will exceed $10
though this specific future hasn't happened yet.

# 2. Bayes' Theorem: The Foundation

At the heart of Bayesian inference is **Bayes' theorem**:

$$P(\theta | \text{Data}) = \frac{P(\text{Data} | \theta) \cdot P(\theta)}{P(\text{Data})}$$

Or in words:

$$\text{Posterior} = \frac{\text{Likelihood} \times \text{Prior}}{\text{Evidence}}$$

Let's break this down:

| Term | Symbol | Trading Interpretation | |------|--------|------------------------| | **Prior** | $P(\theta)$ | Ou
about parameter $\theta$ before seeing data (e.g., "Gold tends to rise during uncertainty") | | **Lik**
| $P(\text{Data} | \theta)$ | How likely is the observed data given our hypothesis? | | **Evidence** |
$P(\text{Data})$ | Total probability of the data (normalizing constant) | | **Posterior** | $P(\theta | \te
$ | Updated belief after seeing the data |

The key insight: **The posterior combines prior knowledge with observed data.**

In [ ]:
```
# Setup: Import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats
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'] = (12, 6)
```

```python
plt.rcParams['font.size'] = 12

print("Libraries loaded successfully!")
```

## 3 . Trading Example: Is This Strategy Profitable?

Let's apply Bayes' theorem to a practical trading question.

### Scenario

You've developed a trading strategy and backtested it for 20 days. It was profitable on out of 20 days (70% win rate).

**Question**: What's the probability that this strategy has a true win rate above 50%?

### Frequentist Approach

A frequentist would calculate a confidence interval:

- Point estimate: 70%
- 95% CI: approximately [46%, 88%] (using normal approximation)

But the frequentist **cannot say** "there's an X% probability the true win rate is above 50%". only say "if we repeated this experiment many times, 95% of intervals would contain the tru parameter."

### Bayesian Approach

We can directly answer: **What's the probability the true win rate > 50%?**

In [ ]:
```python
def bayesian_win_rate_analysis(wins, total, prior_alpha=1, prior_beta=1):
    """
    Analyze win rate using Bayesian inference with Beta-Binomial model.

    Parameters:
    -----------
    wins : int
        Number of winning trades
    total : int
        Total number of trades
    prior_alpha, prior_beta : float
        Beta distribution parameters for prior
        alpha=1, beta=1 gives uniform prior (no prior belief)

    Returns:
    --------
    dict with posterior statistics
    """
    # Posterior parameters (Beta-Binomial conjugate update)
    posterior_alpha = prior_alpha + wins
    posterior_beta = prior_beta + (total - wins)

    # Create distributions
    prior = stats.beta(prior_alpha, prior_beta)
    posterior = stats.beta(posterior_alpha, posterior_beta)

    # Calculate key quantities
    results = {
        'posterior_mean': posterior.mean(),
        'posterior_std': posterior.std(),
        'prob_above_50': 1 - posterior.cdf(0.5),  # P(win_rate > 0.5)
        'prob_above_60': 1 - posterior.cdf(0.6),  # P(win_rate > 0.6)
        'hdi_95': (posterior.ppf(0.025), posterior.ppf(0.975)),
        'prior': prior,
        'posterior': posterior
    }

    return results

# Our trading strategy: 14 wins out of 20 trades
wins = 14
total = 20

# Analyze with uniform prior (no prior belief)
results = bayesian_win_rate_analysis(wins, total)

print("=" * 60)
print("BAYESIAN ANALYSIS: Trading Strategy Win Rate")
print("=" * 60)
print(f"\nObserved: {wins} wins out of {total} trades ({100*wins/total:.0
print(f"\nPosterior Analysis:")
print(f"  Expected win rate: {results['posterior_mean']:.1%}")
print(f"  Standard deviation: {results['posterior_std']:.1%}")
print(f"  95% Credible Interval: [{results['hdi_95'][0]:.1%}, {results['h
print(f"\nKey Probabilities:")
print(f"  P(true win rate > 50%) = {results['prob_above_50']:.1%}")
print(f"  P(true win rate > 60%) = {results['prob_above_60']:.1%}")
```

In [ ]:

```
# Visualize Prior and Posterior
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Plot range
x = np.linspace(0, 1, 1000)

# Prior vs Posterior
ax = axes[0]
ax.plot(x, results['prior'].pdf(x), 'orange', linewidth=2, label='Prior (
ax.plot(x, results['posterior'].pdf(x), 'blue', linewidth=2, label='Poste
ax.axvline(0.5, color='red', linestyle='--', label='Break-even (50%)')
ax.fill_between(x[x > 0.5], results['posterior'].pdf(x[x > 0.5]),
                alpha=0.3, color='green', label=f'P(>50%) = {results["pro
ax.set_xlabel('Win Rate', fontsize=12)
ax.set_ylabel('Probability Density', fontsize=12)
ax.set_title('Bayesian Update: Prior → Posterior', fontsize=14, fontweigh
ax.legend()
ax.set_xlim(0, 1)

# How data changes belief
ax = axes[1]
# Show posteriors with different amounts of data
for n_trades, n_wins in [(5, 4), (20, 14), (100, 70), (500, 350)]:
    posterior = stats.beta(1 + n_wins, 1 + n_trades - n_wins)
    ax.plot(x, posterior.pdf(x), linewidth=2,
            label=f'{n_trades} trades ({100*n_wins/n_trades:.0f}% win)')
```

```
ax.axvline(0.7, color='red', linestyle='--', alpha=0.5, label='True rate
ax.set_xlabel('Win Rate', fontsize=12)
ax.set_ylabel('Probability Density', fontsize=12)
ax.set_title('More Data → More Certainty', fontsize=14, fontweight='bold'
ax.legend()
ax.set_xlim(0.3, 1)

plt.tight_layout()
plt.show()
```

### Key Insight: Uncertainty Decreases with More Data

Notice how the posterior distribution becomes **narrower** (more certain) as we collect more data, peak stays around the true 70% win rate. This is Bayesian learning in action:

- 5 **trades**: Very uncertain - win rate could be anywhere from 30% to 95%
- 20 **trades**: Moderate certainty - likely between 50% and 85%
- 100 **trades**: Pretty confident - likely between 60% and 78%
- 500 **trades**: Highly confident - tightly concentrated around 70%

**Trading implication**: Don't bet big on a strategy with only 20 trades of history. The uncer still high!

## 4. The Power of Prior Information

One of Bayesian inference's greatest strengths is incorporating prior knowledge. Let's see how c priors affect our conclusions.

### Scenario: Experienced vs Naive Trader

**Naive trader**: "I have no prior beliefs about my strategy" (Uniform prior)

**Experienced trader**: "Most trading strategies underperform. I expect a 45% win rate on av and I'm fairly confident about this" (Informative prior centered at 0.45)

In [ ]:
```
# Compare different priors
wins, total = 14, 20  # Same observed data

priors = {
    'Uniform (Naive)': (1, 1),           # No prior belief
    'Skeptical': (9, 11),                # Prior mean = 45%, skeptical of
    'Optimistic': (14, 6),               # Prior mean = 70%, believes in
    'Strong Skeptic': (45, 55)           # Prior mean = 45%, very confide
}

fig, axes = plt.subplots(2, 2, figsize=(14, 10))
x = np.linspace(0, 1, 1000)

for ax, (name, (alpha, beta)) in zip(axes.flatten(), priors.items()):
    # Prior
    prior = stats.beta(alpha, beta)

    # Posterior
    post_alpha = alpha + wins
    post_beta = beta + (total - wins)
    posterior = stats.beta(post_alpha, post_beta)

    # Plot
    ax.plot(x, prior.pdf(x), 'orange', linewidth=2, label='Prior')
    ax.plot(x, posterior.pdf(x), 'blue', linewidth=2, label='Posterior')
    ax.axvline(0.5, color='red', linestyle='--', alpha=0.5)
    ax.axvline(wins/total, color='green', linestyle=':', label='Observed

    # Annotations
    prob_above_50 = 1 - posterior.cdf(0.5)
    ax.set_title(f'{name}\nP(win rate > 50%) = {prob_above_50:.1%}', font
    ax.set_xlabel('Win Rate')
    ax.set_ylabel('Density')
    ax.legend(loc='upper left')
    ax.set_xlim(0, 1)

plt.tight_layout()
plt.show()

# Summary table
print("\n" + "="*70)
print("SUMMARY: How Priors Affect Conclusions")
print("="*70)
print(f"{'Prior':<20} {'Prior Mean':>12} {'Post. Mean':>12} {'P(>50%)':>1
print("-"*70)
for name, (alpha, beta) in priors.items():
    prior_mean = alpha / (alpha + beta)
    post_alpha = alpha + wins
    post_beta = beta + (total - wins)
    post_mean = post_alpha / (post_alpha + post_beta)
    posterior = stats.beta(post_alpha, post_beta)
    prob_above_50 = 1 - posterior.cdf(0.5)
    print(f"{name:<20} {prior_mean:>12.1%} {post_mean:>12.1%} {prob_above
```

## Key Insights on Priors

1. **Priors matter** when data is limited (20 trades)
2. **Skeptical priors** protect against overfitting to noise
3. **Strong priors** require more data to overcome
4. **With enough data**, all reasonable priors converge to similar posteriors

**Trading Wisdom**: In the absence of strong evidence, skepticism is rational. Most trading strateg
Bayesian approach naturally encodes this skepticism.

## 5. Bayesian vs Frequentist: Credible Intervals vs Confidence Intervals

This is one of the most important distinctions in statistics.

### Frequentist 95% Confidence Interval

"If we repeated this experiment many times and calculated a 95% CI each time, 95% intervals would contain the true parameter."

**What it does NOT mean**: "There's a 95% probability the true parameter is in this interval."

## Bayesian 9 5 % Credible Interval

"Given our prior and the observed data, there's a 9 5 % probability the true parameter is in tl interval."

**This IS what traders usually want to know!**

In [ ]:
```
# Compare Frequentist CI vs Bayesian Credible Interval
wins, total = 14, 20
observed_rate = wins / total

# Frequentist: Normal approximation CI
se = np.sqrt(observed_rate * (1 - observed_rate) / total)
freq_ci = (observed_rate - 1.96 * se, observed_rate + 1.96 * se)

# Bayesian: Credible interval from posterior
posterior = stats.beta(1 + wins, 1 + total - wins)
bayes_ci = (posterior.ppf(0.025), posterior.ppf(0.975))

print("="*60)
print("CREDIBLE vs CONFIDENCE INTERVALS")
print("="*60)
print(f"\nObserved: {wins} wins out of {total} trades")
print(f"\nFrequentist 95% CI: [{freq_ci[0]:.1%}, {freq_ci[1]:.1%}]")
print(f"  Interpretation: If we repeated this 100 times, ~95 intervals")
print(f"                  would contain the true win rate.")
print(f"\nBayesian 95% Credible Interval: [{bayes_ci[0]:.1%}, {bayes_ci[1
print(f"  Interpretation: There's a 95% probability the true win rate")
print(f"                  is between {bayes_ci[0]:.1%} and {bayes_ci[1]:.

# Visualize
fig, ax = plt.subplots(figsize=(12, 5))
x = np.linspace(0.3, 1, 1000)
ax.plot(x, posterior.pdf(x), 'blue', linewidth=2, label='Posterior Distri
ax.fill_between(x[(x >= bayes_ci[0]) & (x <= bayes_ci[1])],
                posterior.pdf(x[(x >= bayes_ci[0]) & (x <= bayes_ci[1])])
                alpha=0.3, color='blue', label='95% Credible Interval')
ax.axvline(observed_rate, color='red', linestyle='--', linewidth=2, label

# Mark frequentist CI
ax.axvline(freq_ci[0], color='orange', linestyle=':', linewidth=2)
ax.axvline(freq_ci[1], color='orange', linestyle=':', linewidth=2, label=

ax.set_xlabel('Win Rate', fontsize=12)
ax.set_ylabel('Probability Density', fontsize=12)
ax.set_title('Bayesian Credible Interval vs Frequentist Confidence Interv
ax.legend(loc='upper left')
plt.tight_layout()
plt.show()
```

## 6 . Monte Carlo Simulation: Trading Under Uncertainty

One of the most powerful applications of Bayesian thinking is simulating future outcomes that ac parameter uncertainty.

## Traditional Approach

"Our strategy has a 70% win rate. With 100 trades, we expect 70 wins."

## Bayesian Approach

"Our strategy's win rate is uncertain (posterior distribution). Let's simulate 100 trades th of times, each time drawing a different win rate from our posterior."

This gives us a **distribution of outcomes** that properly accounts for our uncertainty about the tr rate.

In [ ]:
```
def simulate_trading_outcomes(posterior, n_future_trades, n_simulations,
                              avg_win=100, avg_loss=-80):
    """
    Simulate future trading outcomes accounting for parameter uncertainty

    Parameters:
    -----------
    posterior : scipy.stats distribution
        Posterior distribution of win rate
    n_future_trades : int
        Number of future trades to simulate
    n_simulations : int
        Number of Monte Carlo simulations
    avg_win : float
        Average profit on winning trade
    avg_loss : float
        Average loss on losing trade

    Returns:
    --------
    final_pnl : array of final P&L for each simulation
    """
    final_pnl = np.zeros(n_simulations)

    for i in range(n_simulations):
        # Step 1: Draw a win rate from posterior
        true_win_rate = posterior.rvs()

        # Step 2: Simulate trades with this win rate
        wins = np.random.binomial(n_future_trades, true_win_rate)
        losses = n_future_trades - wins

        # Step 3: Calculate P&L
        pnl = wins * avg_win + losses * avg_loss
        final_pnl[i] = pnl

    return final_pnl

# Our posterior from earlier
wins, total = 14, 20
posterior = stats.beta(1 + wins, 1 + total - wins)

# Simulate next 100 trades
n_future_trades = 100
```

```
n_simulations = 10000

# Bayesian simulation (accounts for uncertainty)
bayesian_pnl = simulate_trading_outcomes(posterior, n_future_trades, n_si

# Naive simulation (assumes 70% is the true rate)
naive_pnl = np.zeros(n_simulations)
for i in range(n_simulations):
    wins = np.random.binomial(n_future_trades, 0.70)
    losses = n_future_trades - wins
    naive_pnl[i] = wins * 100 + losses * (-80)

# Compare results
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Bayesian simulation
ax = axes[0]
ax.hist(bayesian_pnl, bins=50, density=True, alpha=0.7, color='blue', lab
ax.axvline(0, color='red', linestyle='--', linewidth=2, label='Break-even
ax.axvline(np.percentile(bayesian_pnl, 5), color='orange', linestyle=':',
ax.axvline(np.percentile(bayesian_pnl, 95), color='orange', linestyle=':'
           label='90% Range')
ax.set_xlabel('Total P&L ($)', fontsize=12)
ax.set_ylabel('Density', fontsize=12)
ax.set_title('Bayesian: Accounts for Win Rate Uncertainty', fontsize=12,
ax.legend()

# Naive simulation
ax = axes[1]
ax.hist(naive_pnl, bins=50, density=True, alpha=0.7, color='green', label
ax.axvline(0, color='red', linestyle='--', linewidth=2, label='Break-even
ax.axvline(np.percentile(naive_pnl, 5), color='orange', linestyle=':', li
ax.axvline(np.percentile(naive_pnl, 95), color='orange', linestyle=':', l
           label='90% Range')
ax.set_xlabel('Total P&L ($)', fontsize=12)
ax.set_ylabel('Density', fontsize=12)
ax.set_title('Naive: Assumes 70% Win Rate is Certain', fontsize=12, fontw
ax.legend()

plt.tight_layout()
plt.show()

# Print summary
print("\n" + "="*60)
print("MONTE CARLO SIMULATION COMPARISON")
print("="*60)
print(f"\nSimulating {n_future_trades} future trades ({n_simulations:,} s
print(f"\n{'Metric':<25} {'Bayesian':>15} {'Naive':>15}")
print("-"*60)
print(f"{'Expected P&L'::<25} ${np.mean(bayesian_pnl):>14,.0f} ${np.mean(
print(f"{'Std Dev of P&L'::<25} ${np.std(bayesian_pnl):>14,.0f} ${np.std(
print(f"{'5th Percentile (VaR)'::<25} ${np.percentile(bayesian_pnl, 5):>1
print(f"{'P(Loss)'::<25} {np.mean(bayesian_pnl < 0):>14.1%} {np.mean(naiv
print(f"\nBayesian approach shows MORE uncertainty - this is realistic!")
```

## Key Insight: Bayesian Simulations Are More Honest About Uncerta

Notice that the Bayesian simulation has:

- **Higher standard deviation**: Because we're uncertain about the true win rate
- **Fatter tails**: More extreme outcomes are possible
- **Higher probability of loss**: The naive approach underestimates risk

The naive approach falsely assumes we **know** the true win rate is 70%. The Bayesian appr
acknowledges our uncertainty and produces more realistic risk estimates.

**This is why Bayesian methods matter for risk management.**

## 7. Real-World Example: Updating Beliefs About Gold

Let's apply Bayesian thinking to a real commodity trading scenario.

### Scenario

You're analyzing whether gold prices tend to rise during periods of high uncertainty (measured b
Your prior belief (from reading market research) is that there's a 65% chance of a positive
correlation. You want to test this with data.

You collect data for 30 months:

- In 21 months with rising VIX, gold rose in 16 of them (76%)
- This suggests gold does tend to rise during uncertainty

How should you update your beliefs?

In [ ]:
```
# Gold-VIX relationship analysis

# Prior: 65% confident gold rises with VIX, moderate certainty
# This translates to Beta(13, 7) which has mean 0.65
prior_alpha = 13
prior_beta = 7

# Observed data
gold_rose_with_vix = 16
total_high_vix_months = 21

# Bayesian update
post_alpha = prior_alpha + gold_rose_with_vix
post_beta = prior_beta + (total_high_vix_months - gold_rose_with_vix)

prior = stats.beta(prior_alpha, prior_beta)
posterior = stats.beta(post_alpha, post_beta)

# Visualize
fig, ax = plt.subplots(figsize=(12, 6))
x = np.linspace(0, 1, 1000)

ax.plot(x, prior.pdf(x), 'orange', linewidth=2, label=f'Prior (mean={prio
ax.plot(x, posterior.pdf(x), 'blue', linewidth=2, label=f'Posterior (mean
ax.axvline(0.5, color='red', linestyle='--', alpha=0.5, label='50% (no re
ax.fill_between(x[x > 0.5], posterior.pdf(x[x > 0.5]), alpha=0.3, color='

ax.set_xlabel('P(Gold rises | VIX rises)', fontsize=12)
ax.set_ylabel('Probability Density', fontsize=12)
ax.set_title('Bayesian Analysis: Does Gold Rise When VIX Rises?', fontsiz
ax.legend()
ax.set_xlim(0.3, 1)

plt.tight_layout()
plt.show()

# Print analysis
print("\n" + "="*60)
print("GOLD-VIX RELATIONSHIP: Bayesian Analysis")
print("="*60)
print(f"\nPrior belief: Gold rises with VIX {prior.mean():.0%} of the tim
print(f"Observed: {gold_rose_with_vix}/{total_high_vix_months} months gol
print(f"\nPosterior:")
print(f"  Updated belief: Gold rises with VIX {posterior.mean():.1%} of t
print(f"  95% Credible Interval: [{posterior.ppf(0.025):.1%}, {posterior.
print(f"  P(positive relationship) = {1 - posterior.cdf(0.5):.1%}")
print(f"\nConclusion: Very strong evidence that gold tends to rise during
```

## 8 . Summary: When to Use Bayesian Methods

### Bayesian Methods Excel When:

| Situation | Why Bayesian Helps |
|---|---|
| **Limited data** | Priors regularize estimates and prevent overfitting |
| **Expert knowledge available** | Priors encode domain expertise |
| **Uncertainty quantification needed** | Full posterior distribution, not just point estimate |
| **Sequential decisions** | Natural framework for updating beliefs |
| **Risk management** | Proper accounting of parameter uncertainty |
| **Regime changes possible** | Can incorporate beliefs about structural breaks |

### Key Concepts Learned

1. **Probability as belief**: Bayesian probability represents our degree of belief, not just long-rur frequencies
2. **Prior → Data → Posterior**: We start with beliefs, update with evidence, get refined beliefs
3. **Credible intervals**: "95% probability the parameter is here" (what we actually want)
4. **Uncertainty propagation**: Monte Carlo simulation with parameter uncertainty gives realistic estimates
5. **Prior sensitivity**: With limited data, priors matter; with lots of data, they wash out

---

## Knowledge Check Quiz

Test your understanding with these questions. Answers are in the next cell.

**Q** $1$ : In Bayesian inference, what does the "prior" represent?

- A) The data we've collected
- B) Our beliefs before seeing data
- C) The probability of the data
- D) The final answer

**Q** $2$ : A $95$% Bayesian credible interval means:

- A) If we repeated the experiment, $95$% of intervals would contain the true value
- B) There's a $95$% probability the true value is in this interval
- C) The parameter is exactly in this range
- D) We are $95$% confident in our methodology

**Q** $3$ : With more data, Bayesian posteriors typically:

- A) Become wider (more uncertain)
- B) Become narrower (more certain)
- C) Stay the same width
- D) Become bimodal

**Q** $4$ : Why might a skeptical prior be appropriate for evaluating trading strategies?

- A) It makes the math easier
- B) Most trading strategies fail, so skepticism is rational
- C) Skeptical priors always give lower risk estimates
- D) The SEC requires it

**Q** $5$ : In the trading simulation, the Bayesian approach showed higher risk because:

- A) The calculations were wrong
- B) It accounts for uncertainty in the true win rate
- C) Bayesian methods always show more risk
- D) The prior was too pessimistic

In [ ]:
```python
# Quiz Answers
print("="*60)
print("QUIZ ANSWERS")
print("="*60)
print("""
Q1: B) Our beliefs before seeing data
    The prior P(θ) represents our beliefs about the parameter before
    observing any data. It encodes domain knowledge.

Q2: B) There's a 95% probability the true value is in this interval
    This is the key difference from frequentist confidence intervals!
    Bayesian credible intervals have a direct probability interpretation.
```

```
Q3: B) Become narrower (more certain)
    As we collect more data, our uncertainty decreases and the posterior
    concentrates around the true value.

Q4: B) Most trading strategies fail, so skepticism is rational
    Given that most strategies underperform after costs, a skeptical prio
    encodes this base rate and protects against overfitting to noise.

Q5: B) It accounts for uncertainty in the true win rate
    The naive approach assumes we KNOW the win rate is 70%. The Bayesian
    approach acknowledges we're uncertain about the true rate, leading
    to wider outcome distributions and more realistic risk estimates.
""")
```

---

## Exercises

Complete these exercises in the `exercises.ipynb` notebook.

### Exercise $1$: Bayes' Theorem Calculator (Easy)

Build a function that calculates posterior probabilities given prior, likelihood, and evidence.

### Exercise $2$: Strategy Evaluation (Medium)

You have a strategy with $45$ wins out of $60$ trades. Calculate:

- The posterior distribution of the win rate
- The probability the true win rate exceeds $60$%
- How many more trades you'd need to be $95$% confident the rate exceeds $50$%

### Exercise $3$: Monte Carlo Risk Analysis (Hard)

Extend the Monte Carlo simulation to include:

- Variable win/loss sizes (also uncertain)
- Transaction costs
- Maximum drawdown analysis

---

## Next Module Preview

In **Module $2$: Prior Selection and Market Knowledge**, we'll learn:

- How to translate domain expertise into prior distributions
- Conjugate priors for computational efficiency
- Prior predictive checks to validate our assumptions
- Encoding commodity seasonality in priors

---

*Module* 1 *Complete*

# Module 2: Prior Selection and Market Knowledge Encoding

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

---

## Learning Objectives

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

1. **Translate** domain expertise into mathematically rigorous prior distributions
2. **Select** appropriate conjugate priors for computational efficiency
3. **Elicit** priors from historical commodity volatility and seasonality patterns
4. **Distinguish** between weakly informative, informative, and strongly informative priors
5. **Validate** prior distributions using prior predictive checks
6. **Assess** prior sensitivity to ensure robust conclusions
7. **Implement** prior selection for real commodity forecasting problems

---

## Why This Matters for Trading

Prior selection is where **Bayesian methods transform from mathematical theory into trading**
In commodity markets:

- **Seasonality is real**: Corn prices peak before harvest, natural gas spikes in winter
- **Mean reversion exists**: Crude oil historically reverts to marginal production cost ($40-8
barrel)
- **Volatility clusters**: High volatility periods follow market shocks (2008, 2020)
- **Regimes shift**: OPEC decisions, climate events, and policy changes alter market dynamics

**Without priors**, your model treats a 300% oil price spike as equally likely as a 2% mo
intelligent priors:

- You **regularize** estimates when data is noisy
- You **encode** decades of market knowledge into models
- You **prevent** overfitting to recent anomalies
- You **quantify** how strongly beliefs should influence decisions

**Bad priors** can bias your models. **Good priors** are the difference between a model that crashes
trading and one that systematically makes money.

---

## 1. The Prior Spectrum: From Ignorance to Certainty

Priors exist on a continuum from complete ignorance to near-certainty:

| Prior Type | When to Use | Example |
|---|---|---|
| **Flat/Uniform** | Truly no information | Beta(1, 1) for unknown probability |
| **Weakly Informative** | Regularization, prevent extremes | Normal(0, 10) for regression coefficie |
| **Informative** | Domain knowledge available | Normal(50, 5) for corn seasonal peak |
| **Strongly Informative** | Physical/economic constraints | Gamma(100, 2) for volatility (must b positive) |

## Mathematical Formulation

Recall Bayes' theorem:

$$P(\theta | \text{Data}) \propto P(\text{Data} | \theta) \cdot P(\theta)$$

The prior $P(\theta)$ can be:

- **Uninformative**: $P(\theta) \propto$ constant (equal weight to all values)
- **Weakly informative**: $P(\theta)$ gently favors reasonable values
- **Informative**: $P(\theta)$ concentrates probability mass around expert beliefs

**Key insight**: The influence of the prior decreases as $n$ (sample size) increases, but with limite commodity data, priors matter significantly.

In [ ]:
```python
# Setup: Import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats
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'] = (12, 6)
plt.rcParams['font.size'] = 11

print("Libraries loaded successfully!")
```

In [ ]:
```python
# Visualize the prior spectrum for a regression coefficient
fig, axes = plt.subplots(2, 2, figsize=(14, 10))
x = np.linspace(-30, 30, 1000)

priors = [
    ("Flat Prior", stats.uniform(-30, 60), "No prior knowledge"),
    ("Weakly Informative", stats.norm(0, 10), "Mild regularization"),
    ("Informative", stats.norm(0, 3), "Some domain knowledge"),
    ("Strongly Informative", stats.norm(2, 0.5), "Strong expert belief")
]

for ax, (title, prior, description) in zip(axes.flatten(), priors):
    if isinstance(prior, stats._continuous_distns.uniform_gen):
        pdf_vals = prior.pdf(x, -30, 60)
    else:
        pdf_vals = prior.pdf(x)

    ax.plot(x, pdf_vals, 'blue', linewidth=2.5)
    ax.fill_between(x, pdf_vals, alpha=0.3, color='blue')
    ax.axvline(0, color='red', linestyle='--', alpha=0.5, label='Zero eff
    ax.set_title(f"{title}\n{description}", fontsize=12, fontweight='bold
    ax.set_xlabel('Coefficient Value (β)', fontsize=11)
    ax.set_ylabel('Probability Density', fontsize=11)
    ax.set_xlim(-30, 30)
    ax.legend()

plt.tight_layout()
plt.show()

print("\nKey Observations:")
print("1. Flat prior: All values equally likely (rarely appropriate)")
print("2. Weakly informative: Gently discourages extreme values")
print("3. Informative: Clear preference for values near zero")
print("4. Strongly informative: Almost certain the value is around 2")
```

## 2 . Conjugate Priors: Mathematical Elegance Meets Computational Efficiency

A **conjugate prior** is a prior distribution that, when combined with a specific likelihood, produces posterior in the same distributional family. This allows for closed-form Bayesian updates without

### Three Essential Conjugate Pairs for Commodities

#### 2 . 1 Beta-Binomial: Win Rates and Directional Forecasts

**Use case**: Probability of price increases, directional forecast accuracy

$$\begin{align} \text{Prior: } & p \sim \text{Beta}(\alpha, \beta) \\ \text{Likelihood: } & X \sim \text{
(n, p) \\ \text{Posterior: } & p | X \sim \text{Beta}(\alpha + x, \beta + n - x) \end{align}$$

**Hyperparameter interpretation**:

- $\alpha$ = prior "successes" (e.g., days corn price rose)
- $\beta$ = prior "failures" (e.g., days corn price fell)
- Prior mean: $\mu = \frac{\alpha}{\alpha + \beta}$
- Prior strength: $\alpha + \beta$ (larger = stronger prior)

#### 2 . 2 Normal-Normal: Price Levels and Returns

**Use case**: Estimating mean return, average price level

$$\begin{align} \text{Prior: } & \mu \sim N(\mu_ 0 , \sigma_ 0 ^ 2 ) \\ \text{Likelihood: } & X_i \sin
\sigma^ 2 ) \text{ (known variance)} \\ \text{Posterior: } & \mu | X \sim N(\mu_n, \sigma_n^ 2 ) \e
$$

where: $$\mu_n = \frac{\frac{\mu_ 0 }{\sigma_ 0 ^ 2 } + \frac{n\bar{x}}{\sigma^ 2 }}{\frac{ 1 }{\sigma_ 0 ^ 2 } + \frac{n}{\sigma^ 2 }}, \quad \sigma_n^ 2 = \frac{ 1 }{\frac{ 1 }{\sigma_ 0 ^ 
\frac{n}{\sigma^ 2 }}$$

**Interpretation**: Posterior mean is a **precision-weighted average** of prior mean and sample me

## 2 . 3 Gamma-Poisson: Event Counts

**Use case**: Number of price spikes per month, supply disruption events

$$\begin{align} \text{Prior: } & \lambda \sim \text{Gamma}(\alpha, \beta) \\ \text{Likelihood: } & X 
\text{Poisson}(\lambda) \\ \text{Posterior: } & \lambda | X \sim \text{Gamma}(\alpha + \sum x_i, \b
\end{align}$$

In [ ]:
```
# Example: Beta-Binomial for directional forecasting
# Scenario: Estimating probability that corn price rises in June (pre-har

def beta_binomial_update(prior_alpha, prior_beta, successes, trials):
    """
    Perform Beta-Binomial conjugate update.

    Returns:
    --------
    dict with prior, posterior, and statistics
    """
    # Prior
    prior = stats.beta(prior_alpha, prior_beta)

    # Posterior (conjugate update)
    post_alpha = prior_alpha + successes
    post_beta = prior_beta + (trials - successes)
    posterior = stats.beta(post_alpha, post_beta)

    return {
        'prior': prior,
        'posterior': posterior,
        'prior_mean': prior.mean(),
        'posterior_mean': posterior.mean(),
        'prior_std': prior.std(),
        'posterior_std': posterior.std(),
        'credible_interval': (posterior.ppf(0.025), posterior.ppf(0.975))
    }

# Prior belief: Corn rises in June about 60% of the time (historical know
# We're moderately confident: equivalent to seeing 12 rises in 20 Junes
prior_alpha = 12
prior_beta = 8

# New data: Last 10 years, corn rose in 7 Junes
successes = 7
trials = 10

result = beta_binomial_update(prior_alpha, prior_beta, successes, trials)

# Visualize
fig, ax = plt.subplots(figsize=(12, 6))
x = np.linspace(0, 1, 1000)
```

```
ax.plot(x, result['prior'].pdf(x), 'orange', linewidth=2.5,
        label=f"Prior: Beta({prior_alpha}, {prior_beta}), mean={result['p
ax.plot(x, result['posterior'].pdf(x), 'blue', linewidth=2.5,
        label=f"Posterior: mean={result['posterior_mean']:.2f}")
ax.axvline(0.5, color='red', linestyle='--', alpha=0.5, label='50% (rando
ax.fill_between(x, result['posterior'].pdf(x), alpha=0.2, color='blue')

ax.set_xlabel('Probability Corn Rises in June', fontsize=12)
ax.set_ylabel('Probability Density', fontsize=12)
ax.set_title('Beta-Binomial Conjugate Update: Corn June Seasonality', fon
ax.legend(fontsize=11)
ax.set_xlim(0.3, 0.9)

plt.tight_layout()
plt.show()

print("="*70)
print("BETA-BINOMIAL CONJUGATE UPDATE")
print("="*70)
print(f"Prior belief: Corn rises in June {result['prior_mean']:.1%} of th
print(f"Prior uncertainty: ± {result['prior_std']:.1%}")
print(f"\nObserved data: {successes} rises in {trials} Junes")
print(f"\nPosterior belief: {result['posterior_mean']:.1%}")
print(f"Posterior uncertainty: ± {result['posterior_std']:.1%}")
print(f"95% Credible Interval: [{result['credible_interval'][0]:.1%}, {re
print(f"\nInterpretation: Updated belief is weighted average of prior and
```

In [ ]:

```
# Example: Normal-Normal for crude oil mean price
# Scenario: Estimate mean WTI crude price with prior from expert knowledg

def normal_normal_update(prior_mean, prior_std, data, data_std):
    """
    Normal-Normal conjugate update (known variance case).

    Parameters:
    -----------
    prior_mean : float
        Prior belief about mean
    prior_std : float
        Prior uncertainty
    data : array
        Observed data
    data_std : float
        Known standard deviation of data
    """
    n = len(data)
    data_mean = np.mean(data)

    # Prior precision (inverse variance)
    prior_precision = 1 / prior_std**2
    data_precision = n / data_std**2

    # Posterior parameters
    post_precision = prior_precision + data_precision
    post_mean = (prior_precision * prior_mean + data_precision * data_mea
    post_std = np.sqrt(1 / post_precision)

    return {
        'prior': stats.norm(prior_mean, prior_std),
        'posterior': stats.norm(post_mean, post_std),
        'prior_mean': prior_mean,
        'posterior_mean': post_mean,
        'prior_std': prior_std,
        'posterior_std': post_std,
        'data_mean': data_mean,
        'credible_interval': (post_mean - 1.96*post_std, post_mean + 1.96
    }

# Prior: Expert believes WTI crude should be around $65, but uncertain (±
prior_mean = 65
prior_std = 15

# Data: Last 30 days average price
np.random.seed(42)
true_price = 72
data_std = 8  # Known daily volatility
observed_prices = np.random.normal(true_price, data_std, 30)

result = normal_normal_update(prior_mean, prior_std, observed_prices, dat

# Visualize
fig, ax = plt.subplots(figsize=(12, 6))
x = np.linspace(30, 100, 1000)

ax.plot(x, result['prior'].pdf(x), 'orange', linewidth=2.5,
        label=f"Prior: N({prior_mean}, {prior_std}²)")
ax.axvline(result['data_mean'], color='green', linestyle=':', linewidth=2
          label=f"Sample Mean: ${result['data_mean']:.2f}")
ax.plot(x, result['posterior'].pdf(x), 'blue', linewidth=2.5,
        label=f"Posterior: N({result['posterior_mean']:.1f}, {result['pos
ax.fill_between(x, result['posterior'].pdf(x), alpha=0.2, color='blue')

ax.set_xlabel('WTI Crude Price ($/barrel)', fontsize=12)
ax.set_ylabel('Probability Density', fontsize=12)
ax.set_title('Normal-Normal Conjugate Update: WTI Crude Mean Price', font
ax.legend(fontsize=11)

plt.tight_layout()
plt.show()

print("="*70)
print("NORMAL-NORMAL CONJUGATE UPDATE")
print("="*70)
print(f"Prior belief: Mean price = ${prior_mean:.2f} ± ${prior_std:.2f}")
print(f"Sample mean: ${result['data_mean']:.2f} (n={len(observed_prices)}
print(f"\nPosterior: Mean price = ${result['posterior_mean']:.2f} ± ${res
print(f"95% Credible Interval: [${result['credible_interval'][0]:.2f}, ${
print(f"\nNotice: Posterior is between prior and sample mean (precision-w
```

## Key Insight: Conjugacy = Speed

With conjugate priors:

- **No MCMC needed**: Instant analytical updates
- **Interpretable**: Clear relationship between prior and posterior
- **Scalable**: Can update sequentially as new data arrives

**When to use conjugate priors**:

- Real-time trading systems (low latency)
- Simple models where conjugacy applies
- Teaching/prototyping before complex models

**When NOT to use**:

- Complex hierarchical models
- Non-standard likelihoods
- When you want maximum modeling flexibility

## 3. Prior Elicitation from Historical Data

**The challenge**: How do you convert "corn is volatile in August" into a prior distribution?

### Strategy 1: Empirical Bayes (Use the Data Twice)

Use historical data to set hyperparameters, then use recent data for likelihood:

1. Calculate historical volatility: $\sigma_{\text{hist}} = \text{std}(\text{returns}_{ 2000-2
   $
2. Set prior: $\sigma \sim \text{Half-Normal}(\sigma_{\text{hist}})$
3. Update with recent data: returns$_{2020-2024}$

**Pros**: Data-driven, objective
**Cons**: "Uses data twice," not fully Bayesian

### Strategy 2: Expert Elicitation

Ask domain experts to specify:

- "What's your best guess for average June corn price?" → prior mean
- "What range would you be 90% confident includes the true value?" → prior variance

### Strategy 3: Maximum Entropy Priors

Choose the prior with maximum entropy subject to constraints (e.g., mean, variance). This repre
"least informative" prior given constraints.

In [ ]:
```python
# Example: Eliciting volatility prior from historical corn data

# Simulate historical corn price returns (2000-2015)
np.random.seed(42)
historical_vol = 0.25  # True historical volatility
n_hist = 250 * 15  # 15 years of daily data
historical_returns = np.random.normal(0, historical_vol, n_hist)

# Calculate empirical statistics
empirical_mean = np.mean(historical_returns)
empirical_std = np.std(historical_returns, ddof=1)
```

```python
print("="*70)
print("PRIOR ELICITATION: Corn Volatility")
print("="*70)
print(f"Historical data: {n_hist:,} daily returns (2000-2015)")
print(f"\nEmpirical mean return: {empirical_mean:.4f}")
print(f"Empirical volatility (std): {empirical_std:.4f}")
print(f"\nPrior specification:")
print(f"  Mean return: μ ~ Normal(0, 0.01)  [weakly informative, expect ~
print(f"  Volatility: σ ~ Half-Normal({empirical_std:.3f})  [from histori

# Visualize the elicited volatility prior
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Historical returns distribution
ax = axes[0]
ax.hist(historical_returns, bins=50, density=True, alpha=0.6, color='blue
x_range = np.linspace(-1, 1, 1000)
ax.plot(x_range, stats.norm(0, empirical_std).pdf(x_range), 'red', linewi
        label=f'Fitted Normal(0, {empirical_std:.3f})')
ax.set_xlabel('Daily Return', fontsize=12)
ax.set_ylabel('Density', fontsize=12)
ax.set_title('Historical Returns Distribution', fontsize=12, fontweight='
ax.legend()
ax.set_xlim(-1, 1)

# Elicited volatility prior
ax = axes[1]
vol_range = np.linspace(0, 0.6, 1000)
# Half-Normal prior for volatility
prior_vol = stats.halfnorm(scale=empirical_std)
ax.plot(vol_range, prior_vol.pdf(vol_range), 'orange', linewidth=2.5,
        label=f'Prior: Half-Normal(σ={empirical_std:.3f})')
ax.axvline(empirical_std, color='red', linestyle='--', linewidth=2, label
ax.fill_between(vol_range, prior_vol.pdf(vol_range), alpha=0.3, color='or
ax.set_xlabel('Volatility (σ)', fontsize=12)
ax.set_ylabel('Probability Density', fontsize=12)
ax.set_title('Elicited Volatility Prior', fontsize=12, fontweight='bold')
ax.legend()

plt.tight_layout()
plt.show()

print(f"\nPrior predictive check: Generate returns from prior")
# Sample from prior
n_prior_samples = 1000
prior_vol_samples = prior_vol.rvs(n_prior_samples)
print(f"Prior volatility samples: mean={np.mean(prior_vol_samples):.3f}, 
      f"90% interval=[{np.percentile(prior_vol_samples, 5):.3f}, {np.perc
print(f"This captures our uncertainty about true volatility before seeing
```

## 4 . Weakly Informative vs Informative Priors

### Weakly Informative Priors: The Goldilocks Zone

**Goal**: Regularize the model without imposing strong beliefs

**Characteristics**:

- Wide enough to not bias estimates
- Narrow enough to rule out nonsense values
- Often used for nuisance parameters

**Examples**:

- Regression coefficient: $\beta \sim N( 0 , 1 0 )$
  - Allows large effects if data supports it
  - But coefficients of 1 0 0 are implausible
- Volatility: $\sigma \sim \text{Half-Cauchy}( 0 , 2 . 5 )$
  - Heavy tails allow high volatility if needed
  - But infinite volatility ruled out

## Informative Priors: Encoding Real Knowledge

**When to use**:

- Strong domain expertise exists
- Physical/economic constraints apply
- Regularizing against overfitting to noise

**Commodity examples**:

- **Seasonal amplitude**: Historical seasonality rarely exceeds ± 1 5 %
  - Prior: $A \sim N( 0 , 0 . 1 0 )$ ( 1 0 % seasonal swing)
- **Mean reversion speed**: Economics suggests λ ∈ [ 0 . 1 , 1 ] per year
  - Prior: $\lambda \sim \text{Beta}( 2 , 2 )$ rescaled to [ 0 , 1 ]
- **Oil price floor**: Can't go negative (pre- 2 0 2 0 !), marginal cost ~$ 4 0
  - Prior: $P_{\text{min}} \sim \text{Truncated-Normal}( 4 0 , 1 0 , \text{lower}= 0 )$

In [ ]:
```
# Compare weakly informative vs informative priors with limited data

# Scenario: Estimating seasonal effect in natural gas prices (winter spik
# True effect: +20% in winter

np.random.seed(42)
true_seasonal_effect = 0.20
n_observations = 15  # Only 15 winters of data

# Generate noisy observations
observed_effects = np.random.normal(true_seasonal_effect, 0.15, n_observa

# Define priors
priors = {
    'Weakly Informative': stats.norm(0, 0.50),  # Wide, centered at zero
    'Informative': stats.norm(0.15, 0.10),      # Based on historical kno
}

# Calculate posteriors (assuming known std of 0.15)
data_mean = np.mean(observed_effects)
data_std = 0.15
```

```
fig, axes = plt.subplots(1, 2, figsize=(14, 5))
x = np.linspace(-0.3, 0.7, 1000)

for ax, (name, prior) in zip(axes, priors.items()):
    # Update using Normal-Normal conjugacy
    prior_mean = prior.mean()
    prior_std = prior.std()

    prior_prec = 1 / prior_std**2
    data_prec = n_observations / data_std**2

    post_prec = prior_prec + data_prec
    post_mean = (prior_prec * prior_mean + data_prec * data_mean) / post_
    post_std = np.sqrt(1 / post_prec)

    posterior = stats.norm(post_mean, post_std)

    # Plot
    ax.plot(x, prior.pdf(x), 'orange', linewidth=2, label='Prior')
    ax.axvline(data_mean, color='green', linestyle=':', linewidth=2, labe
    ax.plot(x, posterior.pdf(x), 'blue', linewidth=2, label='Posterior')
    ax.axvline(true_seasonal_effect, color='red', linestyle='--', linewid
    ax.fill_between(x, posterior.pdf(x), alpha=0.2, color='blue')

    ax.set_xlabel('Seasonal Effect', fontsize=12)
    ax.set_ylabel('Density', fontsize=12)
    ax.set_title(f'{name} Prior\nPost. Mean: {post_mean:.2f} ± {post_std:
                 fontsize=12, fontweight='bold')
    ax.legend(fontsize=10)
    ax.set_xlim(-0.3, 0.7)

plt.tight_layout()
plt.show()

print("="*70)
print("PRIOR STRENGTH COMPARISON")
print("="*70)
print(f"True seasonal effect: {true_seasonal_effect:.0%}")
print(f"Observed data mean: {data_mean:.2f} (n={n_observations})")
print(f"\nWith limited data, the informative prior pulls estimate closer
print(f"This is regularization in action.")
```

## 5 . Prior Predictive Checks: Validating Your Priors

**The question**: Before seeing any data, does my prior generate realistic data?

### Prior Predictive Distribution

$$P(\tilde{y}) = \int P(\tilde{y} | \theta) P(\theta) d\theta$$

In words: Average the likelihood over all possible parameter values weighted by the prior.

### Workflow:

1 . **Sample** parameter values from prior: $\theta^{(1)}, \ldots, \theta^{(N)} \sim P(\theta)$

2 . For each $\theta^{(i)}$, **generate** fake data: $\tilde{y}^{(i)} \sim P(y | \theta^{(i)})$

3 . **Inspect** simulated datasets:

- Do they look like real commodity data?
- Are the ranges reasonable?
- Do they exhibit expected features (volatility clustering, seasonality)?

## Red Flags:

- Simulated prices go negative (need positivity constraint)
- Volatility is 500% (unrealistic for most commodities)
- No seasonal patterns when you expect them

In [ ]:
```
# Prior predictive check for corn price model
# Model: log(Price_t) = μ + β*sin(2π*t/365) + ε, where ε ~ N(0, σ)

# Define priors
prior_mu = stats.norm(np.log(400), 0.3)       # Mean log-price around $40
prior_beta = stats.norm(0, 0.15)              # Seasonal amplitude (15% s
prior_sigma = stats.halfnorm(scale=0.10)      # Daily volatility

# Prior predictive sampling
n_prior_samples = 100
n_days = 365 * 2  # 2 years
t = np.arange(n_days)

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

# Generate prior predictive samples
for i in range(n_prior_samples):
    # Sample parameters from priors
    mu_sample = prior_mu.rvs()
    beta_sample = prior_beta.rvs()
    sigma_sample = prior_sigma.rvs()

    # Generate data from model
    seasonal_component = beta_sample * np.sin(2 * np.pi * t / 365)
    noise = np.random.normal(0, sigma_sample, n_days)
    log_price = mu_sample + seasonal_component + noise
    price = np.exp(log_price)

    # Plot
    axes[0].plot(t, price, alpha=0.3, color='blue', linewidth=0.5)

axes[0].set_xlabel('Days', fontsize=12)
axes[0].set_ylabel('Corn Price (¢/bushel)', fontsize=12)
axes[0].set_title('Prior Predictive Check: Simulated Corn Prices from Pri
                  fontsize=13, fontweight='bold')
axes[0].set_ylim(200, 800)
axes[0].axhline(400, color='red', linestyle='--', alpha=0.5, label='Expec
axes[0].legend()

# Distribution of simulated prices at a single time point
simulated_prices_t0 = []
for i in range(5000):
    mu_sample = prior_mu.rvs()
    beta_sample = prior_beta.rvs()
    sigma_sample = prior_sigma.rvs()
    log_price = mu_sample + beta_sample * np.sin(0) + np.random.normal(0,
    simulated_prices_t0.append(np.exp(log_price))

axes[1].hist(simulated_prices_t0, bins=50, density=True, alpha=0.6, color
             label='Prior predictive distribution')
axes[1].axvline(400, color='red', linestyle='--', linewidth=2, label='Exp
axes[1].set_xlabel('Corn Price (¢/bushel)', fontsize=12)
axes[1].set_ylabel('Density', fontsize=12)
axes[1].set_title('Prior Predictive Distribution at t=0', fontsize=13, fo
axes[1].legend()
axes[1].set_xlim(150, 800)

plt.tight_layout()
plt.show()

print("="*70)
print("PRIOR PREDICTIVE CHECK INTERPRETATION")
print("="*70)
print(f"\nSimulated price range: ${np.percentile(simulated_prices_t0, 1):
print(f"Median: ${np.median(simulated_prices_t0):.0f}")
print(f"\nQuestions to ask:")
print(f"  1. Do these prices look realistic? ✓ (corn trades $300-600 typi
print(f"  2. Is seasonality visible? ✓ (smooth annual cycles)")
print(f"  3. Are there negative prices? ✗ (log-scale prevents this)")
print(f"  4. Is volatility reasonable? ✓ (not too wild)")
print(f"\nConclusion: Prior seems reasonable! Proceed to fit model.")
```

## 6 . Prior Sensitivity Analysis

**The concern**: What if my prior is wrong? Will it ruin my conclusions?

### Sensitivity Analysis Workflow

1 . **Fit model** with your chosen prior
2 . **Refit** with several alternative priors:
    - More skeptical (narrower)
    - More vague (wider)
    - Different center
3 . **Compare** posterior inferences:
    - Do point estimates change substantially?
    - Do credible intervals overlap?
    - Do trading decisions change?

### Interpretation:

- **Robust**: Conclusions similar across priors → data dominates
- **Sensitive**: Conclusions vary widely → need more data or justify prior choice

**Trading rule**: If your P&L depends critically on prior choice, you don't have enough data to trade
strategy yet.

In [ ]:
```
# Prior sensitivity analysis: Does conclusion depend on prior?
# Scenario: Is natural gas seasonal spike > 10%?

np.random.seed(42)
```

```python
true_effect = 0.18
n_obs = 20
observed_data = np.random.normal(true_effect, 0.08, n_obs)
data_mean = np.mean(observed_data)
data_std = 0.08

# Test multiple priors
prior_scenarios = {
    'Skeptical': stats.norm(0.05, 0.05),
    'Neutral': stats.norm(0.10, 0.10),
    'Optimistic': stats.norm(0.20, 0.08),
    'Vague': stats.norm(0, 0.50),
}

fig, axes = plt.subplots(2, 2, figsize=(14, 10))
x = np.linspace(-0.1, 0.5, 1000)

results = {}
for ax, (name, prior) in zip(axes.flatten(), prior_scenarios.items()):
    # Conjugate update
    prior_mean = prior.mean()
    prior_std = prior.std()

    prior_prec = 1 / prior_std**2
    data_prec = n_obs / data_std**2

    post_prec = prior_prec + data_prec
    post_mean = (prior_prec * prior_mean + data_prec * data_mean) / post_
    post_std = np.sqrt(1 / post_prec)

    posterior = stats.norm(post_mean, post_std)

    # Calculate P(effect > 10%)
    prob_above_10 = 1 - posterior.cdf(0.10)

    results[name] = {
        'posterior_mean': post_mean,
        'posterior_std': post_std,
        'prob_above_10': prob_above_10
    }

    # Plot
    ax.plot(x, prior.pdf(x), 'orange', linewidth=2, label='Prior')
    ax.plot(x, posterior.pdf(x), 'blue', linewidth=2, label='Posterior')
    ax.axvline(0.10, color='red', linestyle='--', linewidth=2, label='10%
    ax.fill_betweenx([0, ax.get_ylim()[1]], 0.10, 0.5, alpha=0.1, color='
    ax.axvline(data_mean, color='green', linestyle=':', linewidth=1.5, al

    ax.set_xlabel('Seasonal Effect', fontsize=11)
    ax.set_ylabel('Density', fontsize=11)
    ax.set_title(f'{name} Prior\nP(effect > 10%) = {prob_above_10:.1%}',
                 fontsize=12, fontweight='bold')
    ax.legend(fontsize=9)
    ax.set_xlim(-0.1, 0.5)

plt.tight_layout()
plt.show()

# Summary table
print("="*70)
print("PRIOR SENSITIVITY ANALYSIS")
print("="*70)
print(f"\nObserved data: mean = {data_mean:.2f}, n = {n_obs}")
print(f"True effect: {true_effect:.0%}\n")
print(f"{'Prior':<15} {'Post. Mean':>12} {'Post. Std':>12} {'P(>10%)':>12
print("-"*70)
for name, res in results.items():
    print(f"{name:<15} {res['posterior_mean']:>12.2f} {res['posterior_std

print(f"\nConclusion: All priors lead to P(effect > 10%) > 90%")
print(f"Result is ROBUST to prior choice - data dominates!")
```

## 7 . Practical Application: Encoding Corn Seasonality

Let's apply everything we've learned to a realistic commodity trading problem.

### Problem Statement

You're building a model to forecast corn prices. You know:

- Corn has strong seasonality (planting in spring, harvest in fall)
- Prices typically peak in June-July (pre-harvest fear of supply shortage)
- Prices typically bottom in October-November (post-harvest glut)

### Model

$$\log(P_t) = \mu + \beta_1 \sin\left(\frac{2\pi t}{365}\right) + \beta_2 \cos\left(\frac{
{365}\right) + \epsilon_t$$

where:

- $\mu$ = baseline log-price
- $\beta_1, \beta_2$ = seasonal coefficients
- Seasonal amplitude = $\sqrt{\beta_1^2 + \beta_2^2}$
- Peak timing = $\arctan(\beta_2 / \beta_1)$

### Prior Selection

From historical knowledge:

- Average price: ~$4.00/bushel → $\mu \sim N(\log(4), 0.2)$
- Seasonal swing: 10-15% → $\beta_1, \beta_2 \sim N(0, 0.10)$
- Volatility: 20-30% annualized → $\sigma \sim \text{Half-Normal}(0.25/\sqrt{2

In [ ]:
```
# Full seasonal model for corn prices

# Generate synthetic corn price data with known seasonality
np.random.seed(42)
n_days = 365 * 3  # 3 years
t = np.arange(n_days)

# True parameters
true_mu = np.log(4.0)
```

```python
true_beta1 = 0.12  # Sin coefficient
true_beta2 = 0.08  # Cos coefficient
true_sigma = 0.015  # Daily volatility

# Generate data
seasonal = true_beta1 * np.sin(2*np.pi*t/365) + true_beta2 * np.cos(2*np.
noise = np.random.normal(0, true_sigma, n_days)
log_prices = true_mu + seasonal + noise
prices = np.exp(log_prices)

# Design matrix for regression
X = np.column_stack([
    np.ones(n_days),                    # Intercept
    np.sin(2*np.pi*t/365),              # Sin component
    np.cos(2*np.pi*t/365)               # Cos component
])
y = log_prices

# Bayesian linear regression with informative priors
# Prior parameters
prior_mean = np.array([np.log(4.0), 0, 0])  # μ, β1, β2
prior_cov = np.diag([0.2**2, 0.10**2, 0.10**2])  # Diagonal covariance

# Likelihood: y | β, σ² ~ N(Xβ, σ²I)
# We'll use known σ² for conjugacy (in practice, estimate this too)
sigma_sq = true_sigma**2

# Posterior (Normal-Normal conjugate update for regression)
# Posterior precision = Prior precision + Data precision
prior_precision = np.linalg.inv(prior_cov)
data_precision = X.T @ X / sigma_sq

post_precision = prior_precision + data_precision
post_cov = np.linalg.inv(post_precision)

# Posterior mean = post_cov @ (prior_precision @ prior_mean + data_precis
post_mean = post_cov @ (prior_precision @ prior_mean + (X.T @ y) / sigma_

print("="*70)
print("BAYESIAN SEASONAL MODEL: Corn Prices")
print("="*70)
print(f"\nTrue parameters:")
print(f"  μ (log-price): {true_mu:.3f} → ${np.exp(true_mu):.2f}/bushel")
print(f"  β₁ (sin): {true_beta1:.3f}")
print(f"  β₂ (cos): {true_beta2:.3f}")
print(f"  Amplitude: {np.sqrt(true_beta1**2 + true_beta2**2):.3f} ({100*n

print(f"\nPosterior estimates:")
print(f"  μ: {post_mean[0]:.3f} ± {np.sqrt(post_cov[0,0]):.3f} → ${np.exp
print(f"  β₁: {post_mean[1]:.3f} ± {np.sqrt(post_cov[1,1]):.3f}")
print(f"  β₂: {post_mean[2]:.3f} ± {np.sqrt(post_cov[2,2]):.3f}")
amplitude_est = np.sqrt(post_mean[1]**2 + post_mean[2]**2)
print(f"  Amplitude: {amplitude_est:.3f} ({100*amplitude_est:.1f}%)")

# Fitted values
fitted_log_prices = X @ post_mean
fitted_prices = np.exp(fitted_log_prices)

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

```
# Time series
ax = axes[0]
dates = pd.date_range('2021-01-01', periods=n_days, freq='D')
ax.plot(dates, prices, 'o', alpha=0.3, markersize=2, label='Observed pric
ax.plot(dates, fitted_prices, 'red', linewidth=2, label='Bayesian fit (po
ax.set_xlabel('Date', fontsize=12)
ax.set_ylabel('Corn Price ($/bushel)', fontsize=12)
ax.set_title('Corn Price Seasonality: Bayesian Regression', fontsize=14,
ax.legend()
ax.grid(alpha=0.3)

# Seasonal component only
ax = axes[1]
seasonal_component = post_mean[1] * np.sin(2*np.pi*t/365) + post_mean[2]
ax.plot(t % 365, seasonal_component, 'o', alpha=0.5, markersize=3, color=
ax.axhline(0, color='black', linestyle='--', alpha=0.5)
ax.set_xlabel('Day of Year', fontsize=12)
ax.set_ylabel('Seasonal Effect (log-price)', fontsize=12)
ax.set_title('Extracted Seasonal Pattern', fontsize=14, fontweight='bold'
ax.set_xlim(0, 365)
ax.grid(alpha=0.3)

# Mark key agricultural dates
key_dates = [
    (120, 'Planting'),
    (180, 'Peak (pre-harvest)'),
    (280, 'Harvest'),
]
for day, label in key_dates:
    ax.axvline(day, color='red', linestyle=':', alpha=0.7)
    ax.text(day, ax.get_ylim()[1]*0.9, label, rotation=90, fontsize=9)

plt.tight_layout()
plt.show()

print(f"\nTrading insight: Seasonal pattern shows {amplitude_est:.1%} swi
print(f"Peak occurs around day {np.argmax(seasonal_component):.0f} (late
print(f"Trough occurs around day {np.argmin(seasonal_component):.0f} (lat
```

## 8 . Summary: The Art and Science of Prior Selection

### Key Principles

| Principle | Guidance |
|---|---|
| **Honesty** | Priors should reflect genuine beliefs, not desired conclusions |
| **Transparency** | Document and justify prior choices |
| **Calibration** | Use prior predictive checks to validate |
| **Sensitivity** | Test robustness to prior specification |
| **Parsimony** | Use weakly informative priors when in doubt |

## The Prior Selection Checklist

Before finalizing priors:

1. ✅ **Physical constraints**: Are negative prices/volatilities possible?
2. ✅ **Domain knowledge**: What do experts believe?
3. ✅ **Historical data**: What have we seen before?
4. ✅ **Prior predictive**: Do simulations look realistic?
5. ✅ **Sensitivity**: Do conclusions depend critically on prior?
6. ✅ **Documentation**: Can someone else understand your choices?

## Common Mistakes to Avoid

- ❌ Using flat priors for unbounded parameters (they're not actually "uninformative")
- ❌ Picking priors to get desired answers (confirmation bias)
- ❌ Ignoring domain expertise when it exists
- ❌ Using overly confident priors with limited justification
- ❌ Failing to check prior predictive distributions

## When Priors Matter Most

- **Limited data**: $< 100$ observations
- **High-dimensional models**: Many parameters relative to data
- **Hierarchical models**: Priors on hyperparameters strongly influence results
- **Risk management**: Tail probabilities sensitive to prior specification

---

## Knowledge Check Quiz

**Q1**: A conjugate prior is valuable because:

- A) It always gives the most accurate results
- B) It allows closed-form posterior updates without MCMC
- C) It's always the correct prior to use
- D) It eliminates the need for data

**Q2**: In a Beta-Binomial model, increasing the prior hyperparameters ($\alpha + \beta$) makes the prior:

- A) More influential (stronger)
- B) Less influential (weaker)
- C) Wider and more uncertain
- D) Has no effect

**Q3**: Prior predictive checks help you:

- A) Calculate the posterior distribution
- B) Determine if your prior generates realistic data

- C) Avoid using priors altogether
- D) Maximize the likelihood

**Q** 4 : A weakly informative prior is appropriate when:

- A) You have very strong domain knowledge
- B) You want regularization without imposing strong beliefs
- C) The data is highly informative
- D) You want results identical to frequentist methods

**Q** 5 : If your trading decision changes dramatically with different reasonable priors, you should:

- A) Pick the prior that gives the best backtest results
- B) Use a flat prior
- C) Recognize you need more data or a stronger justification for your prior
- D) Ignore the sensitivity and proceed

In [ ]:
```
# Quiz Answers
print("="*70)
print("QUIZ ANSWERS")
print("="*70)
print("""
Q1: B) It allows closed-form posterior updates without MCMC
    Conjugate priors provide computational efficiency through analytical
    solutions. They're not always "correct" but are very useful.

Q2: A) More influential (stronger)
    α + β represents "prior sample size." Larger values mean the prior
    counts more relative to the data, making it stronger/more influential

Q3: B) Determine if your prior generates realistic data
    Prior predictive checks sample from P(data|prior) to verify that
    your prior can generate datasets that look like real commodity data.

Q4: B) You want regularization without imposing strong beliefs
    Weakly informative priors are the "Goldilocks" choice: they prevent
    nonsense values without strongly biasing results.

Q5: C) Recognize you need more data or a stronger justification for your
    Prior sensitivity indicates conclusions aren't robust. Either collect
    more data or carefully justify your prior choice. Never pick priors
    to optimize backtests (overfitting)!
""")
```

---

## Exercises

Complete these exercises in the `exercises.ipynb` notebook.

### Exercise 1 : Conjugate Prior Derivation (Medium)

Derive the posterior parameters for the Normal-Normal conjugate pair. Verify your derivation mat
formula in the notes.

## Exercise 2 : Agricultural Seasonality (Medium)

Using the corn seasonality model, encode priors for wheat (different growing season). Wheat is
in fall, harvested in summer. How should $\beta_1$ and $\beta_2$ change?

## Exercise 3 : Prior Sensitivity in Trading (Hard)

You have a mean-reversion trading strategy. Build a model with a prior on the mean-reversion s
Test sensitivity to three priors: skeptical (λ~ 0 ), moderate (λ~ 0 . 5 ), strong (λ~ 1 ). With only
trades, how much do P&L distributions differ?

## Exercise 4 : Prior Elicitation Interview (Hard)

Interview a hypothetical crude oil expert to elicit a prior for next year's average price. Convert the
qualitative statements into a Normal distribution:

- "Most likely around $ 7 5 "
- "Very unlikely to be below $ 5 0 or above $ 1 0 0 "
- " 9 0 % confident it's between $ 6 0 and $ 9 0 "

---

## Next Module Preview

In **Module 3 : MCMC and Computational Inference**, we'll learn:

- Why conjugate priors aren't always possible
- How MCMC algorithms sample from complex posteriors
- Implementing Metropolis-Hastings from scratch
- Using PyMC for production-grade Bayesian inference
- Diagnosing convergence and sampling problems
- Applying MCMC to real commodity price forecasting

---

*Module 2 Complete*

# Module 3 : MCMC and Computational Inference

**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** why MCMC is necessary when conjugate priors don't apply
2. **Implement** the Metropolis-Hastings algorithm from scratch
3. **Use** PyMC for production-grade Bayesian inference
4. **Diagnose** convergence using R-hat, effective sample size, and trace plots
5. **Identify** and fix common sampling problems (divergences, poor mixing)
6. **Validate** models using posterior predictive checks
7. **Apply** MCMC to real commodity price forecasting problems

---

## Why This Matters for Trading

In Module 2 , we used conjugate priors for computational convenience. But real commodity m
are complex:

- **Non-linear relationships**: Oil prices don't respond linearly to inventory changes
- **Fat-tailed distributions**: Student-t likelihoods for robustness to outliers
- **Hierarchical structures**: Different volatility regimes across time
- **Custom likelihoods**: Asymmetric loss functions for directional bets

**None of these have conjugate priors.** We need MCMC.

### What MCMC Enables

- **Full posterior distributions**: Not just point estimates, but complete uncertainty quantificatio
- **Flexible modeling**: Any likelihood + any prior = solvable
- **Hierarchical models**: Multi-level models that share information across assets
- **Model comparison**: Estimate marginal likelihoods via bridge sampling

### The Cost

- **Computational time**: Minutes to hours instead of milliseconds
- **Convergence concerns**: Bad samplers can give wrong answers
- **Diagnostic overhead**: Must check convergence, effective sample size, etc.

**Bottom line**: MCMC is essential for modern Bayesian trading strategies. Understanding it is nor
negotiable.

## 1. Why Do We Need MCMC?

### The Fundamental Problem

Bayes' theorem gives us:

$$P(\theta | y) = \frac{P(y | \theta) P(\theta)}{P(y)}$$

The denominator (evidence) requires an integral:

$$P(y) = \int P(y | \theta) P(\theta) d\theta$$

**Problem**: This integral is analytically intractable for most real models.

### Example: Non-Conjugate Posterior

Consider forecasting oil prices with a robust Student-t likelihood:

$$\begin{align} y_i &\sim \text{Student-t}(\nu, \mu, \sigma) \\ \mu &\sim N(70, 20) \\ \sigm
\text{Half-Normal}(10) \\ \nu &\sim \text{Gamma}(2, 0.1) \end{align}$$

The posterior $P(\mu, \sigma, \nu | y)$ has **no closed form**. We can't compute it analytically.

### The MCMC Solution

**Key insight**: We don't need the actual posterior formula. We just need **samples** from it.

If we have samples $\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(N)} \sim P(\theta | y)$, we can:

- Estimate posterior mean: $E[\theta] \approx \frac{1}{N}\sum_i \theta^{(i)}$
- Compute credible intervals: percentiles of samples
- Calculate probabilities: $P(\theta > c) \approx \frac{1}{N}\sum_i \mathbb{1}(\theta^{(i)} > 
- Generate predictions: $\tilde{y} \sim P(y | \theta^{(i)})$

**MCMC = Markov Chain Monte Carlo**: Algorithms that generate samples from complex distributi

In [ ]:
```python
# Setup: Import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats
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'] = (12, 6)
plt.rcParams['font.size'] = 11

print("Libraries loaded successfully!")
```

## 2 . Metropolis-Hastings: MCMC from First Principles

The **Metropolis-Hastings (MH)** algorithm is the foundation of MCMC. Understanding it builds in all modern samplers.

### The Algorithm

Goal: Sample from posterior $P(\theta | y) \propto P(y | \theta) P(\theta)$

**Metropolis-Hastings Algorithm**:

1 . Start with initial value $\theta^{( 0 )}$
2 . For iteration $t = 1 , 2 , \ldots, N$:
  - **Propose**: Draw $\theta^ ∗ \sim q(\theta^ ∗ | \theta^{(t- 1 )})$ from proposal distrib
  - **Calculate acceptance ratio**: $$r = \frac{P(\theta^ ∗ | y)}{P(\theta^{(t- 1 )} | y)} = \frac \theta^ ∗ ) P(\theta^ ∗ )}{P(y | \theta^{(t- 1 )}) P(\theta^{(t- 1 )})}$$
  - **Accept/Reject**:
    - With probability $\min( 1 , r)$: set $\theta^{(t)} = \theta^ ∗ $ (accept)
    - Otherwise: set $\theta^{(t)} = \theta^{(t- 1 )}$ (reject, stay put)

### Key Insights

- **No normalization needed**: The $P(y)$ term cancels out in the ratio!
- **Always accept improvements**: If $r > 1 $ (proposal is better), always accept
- **Sometimes accept worse states**: If $r < 1 $, accept with probability $r$ (allows explora
- **Eventually converges**: The chain's stationary distribution is the posterior

### Proposal Distribution

Common choice: **Random walk** proposal: $$q(\theta^ ∗ | \theta^{(t- 1 )}) = N(\theta^{(t- 1 )}, \sigma_{\text{prop}}^ 2 )$$

- Too small $\sigma_{\text{prop}}$: Chain explores slowly (high acceptance, slow mixing)
- Too large $\sigma_{\text{prop}}$: Proposals rejected often (low acceptance)
- **Optimal**: Acceptance rate around 2 3 - 4 4 % (for high dimensions)

In [ ]:
```
def metropolis_hastings(log_posterior, theta_init, n_iter, proposal_sd):
    """
    Metropolis-Hastings MCMC sampler.

    Parameters:
    -----------
    log_posterior : function
        Function that computes log P(theta | y)
    theta_init : float or array
        Initial parameter value
    n_iter : int
        Number of MCMC iterations
    proposal_sd : float or array
        Standard deviation of proposal distribution
    Returns:
    --------
    samples : array
        MCMC samples from posterior
    acceptance_rate : float
        Proportion of proposals accepted
    """
    theta = theta_init
    samples = np.zeros((n_iter, len(np.atleast_1d(theta))))
    n_accepted = 0

    for i in range(n_iter):
        # Propose new state (random walk)
        theta_proposal = theta + np.random.normal(0, proposal_sd, size=th

        # Calculate log acceptance ratio
        log_r = log_posterior(theta_proposal) - log_posterior(theta)

        # Accept/reject
        if np.log(np.random.rand()) < log_r:
            theta = theta_proposal  # Accept
            n_accepted += 1
        # else: theta stays the same (reject)

        samples[i] = theta

    acceptance_rate = n_accepted / n_iter
    return samples, acceptance_rate

# Example: Estimate mean of crude oil prices
# Data: 50 daily prices
np.random.seed(42)
true_mu = 75
true_sigma = 8
n_obs = 50
oil_prices = np.random.normal(true_mu, true_sigma, n_obs)

# Model: y_i ~ N(μ, σ²) with known σ = 8
# Prior: μ ~ N(70, 20²)
# Posterior: μ | y ~ N(μ_post, σ_post²) (analytically known for compariso

# Define log posterior (up to constant)
def log_posterior_oil(mu):
    # Log prior: N(70, 20)
    log_prior = stats.norm(70, 20).logpdf(mu)

    # Log likelihood: Σ log N(y_i | μ, 8)
    log_likelihood = np.sum(stats.norm(mu, 8).logpdf(oil_prices))

    return log_prior + log_likelihood

# Run Metropolis-Hastings
theta_init = np.array([70.0])  # Start at prior mean
n_iter = 10000
proposal_sd = 2.0

samples, acceptance_rate = metropolis_hastings(log_posterior_oil, theta_i

print("="*70)
print("METROPOLIS-HASTINGS: Crude Oil Mean Price")
print("="*70)
print(f"\nData: n={n_obs} observations, sample mean={np.mean(oil_prices):
print(f"True μ: {true_mu:.2f}")
print(f"\nMCMC Settings:")
print(f"  Iterations: {n_iter:,}")
print(f"  Proposal SD: {proposal_sd}")
print(f"  Acceptance rate: {acceptance_rate:.1%}")
print(f"\nPosterior estimates (from MCMC samples):")
print(f"  Mean: {np.mean(samples[1000:]):.2f}")  # Discard first 1000 as
print(f"  Std: {np.std(samples[1000:]):.2f}")
print(f"  95% CI: [{np.percentile(samples[1000:], 2.5):.2f}, {np.percenti
```

In [ ]:

```
# Visualize MCMC samples
fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# Trace plot
ax = axes[0, 0]
ax.plot(samples, linewidth=0.5, alpha=0.7)
ax.axhline(true_mu, color='red', linestyle='--', linewidth=2, label=f'Tru
ax.axvline(1000, color='orange', linestyle=':', linewidth=2, label='Burn-
ax.set_xlabel('Iteration', fontsize=11)
ax.set_ylabel('μ (mean oil price)', fontsize=11)
ax.set_title('Trace Plot: Exploring the Posterior', fontsize=12, fontweig
ax.legend()
ax.grid(alpha=0.3)

# Histogram (posterior distribution)
ax = axes[0, 1]
ax.hist(samples[1000:], bins=50, density=True, alpha=0.6, color='blue', l
ax.axvline(true_mu, color='red', linestyle='--', linewidth=2, label=f'Tru

# Overlay analytical posterior for comparison
prior_mean, prior_var = 70, 20**2
data_mean, data_var = np.mean(oil_prices), 8**2 / n_obs
post_var = 1 / (1/prior_var + 1/data_var)
post_mean = post_var * (prior_mean/prior_var + data_mean/data_var)
post_sd = np.sqrt(post_var)

x_range = np.linspace(65, 85, 1000)
ax.plot(x_range, stats.norm(post_mean, post_sd).pdf(x_range), 'green', li
        label=f'Analytical posterior')
ax.set_xlabel('μ (mean oil price)', fontsize=11)
ax.set_ylabel('Density', fontsize=11)
ax.set_title('Posterior Distribution', fontsize=12, fontweight='bold')
ax.legend()

# Autocorrelation
ax = axes[1, 0]
from pandas.plotting import autocorrelation_plot
autocorrelation_plot(pd.Series(samples[1000:].flatten()), ax=ax, color='b
ax.set_xlabel('Lag', fontsize=11)
ax.set_ylabel('Autocorrelation', fontsize=11)
ax.set_title('Autocorrelation: Samples Are Correlated', fontsize=12, font
ax.set_xlim(0, 200)

# Running mean (convergence diagnostic)
ax = axes[1, 1]
running_mean = np.cumsum(samples.flatten()) / np.arange(1, len(samples)+1
ax.plot(running_mean, linewidth=1.5)
ax.axhline(post_mean, color='green', linestyle='--', linewidth=2, label='
ax.axvline(1000, color='orange', linestyle=':', linewidth=2, label='Burn-
ax.set_xlabel('Iteration', fontsize=11)
ax.set_ylabel('Running Mean', fontsize=11)
ax.set_title('Convergence Check: Running Average', fontsize=12, fontweigh
ax.legend()
ax.grid(alpha=0.3)

plt.tight_layout()
plt.show()

print("\nKey observations:")
print("1. Trace plot shows random walk behavior (correlated samples)")
print("2. MCMC histogram matches analytical posterior (validation!)")
print("3. Autocorrelation decays slowly (samples are dependent)")
print("4. Running mean converges to true posterior mean")
```

## Key Insight: MCMC Samples Are Correlated

Unlike independent Monte Carlo samples, **MCMC samples are autocorrelated**:

- Each sample depends on the previous one (Markov chain)
- Effective sample size < actual number of samples
- Need to account for this when calculating standard errors

**Effective Sample Size (ESS)**: Number of independent samples with equivalent information $$\t
\approx \frac{N}{ 1 + 2 \sum_{k= 1 }^{\infty} \rho_k}$$ where $\rho_k$ is autocorrelation at

## 3 . Introduction to PyMC: Production-Grade MCMC

While implementing MH from scratch builds intuition, **production trading systems need robus samplers**.

### PyMC Benefits

- **NUTS sampler**: No U-Turn Sampler (state-of-the-art HMC variant)
- **Automatic differentiation**: Gradients computed automatically
- **Convergence diagnostics**: R-hat, ESS, divergences built-in
- **Vectorization**: Fast sampling on GPUs
- **Ecosystem**: ArviZ for visualization, diagnostics

### NUTS vs Metropolis-Hastings

| Feature | Metropolis-Hastings | NUTS |
|---|---|---|
| **Gradient** | Not required | Required (auto-diff) |
| **Efficiency** | Low (random walk) | High (guided exploration) |
| **Tuning** | Manual (proposal SD) | Automatic (mass matrix, step size) |
| **High dimensions** | Struggles (> $10$) | Scales well ( $100$ +) |
| **Speed** | Slow mixing | Fast mixing |

**Bottom line**: Use NUTS for real problems, MH for teaching/debugging.

In [ ]:
```python
# Install PyMC if needed (uncomment)
# !pip install pymc arviz

import pymc as pm
import arviz as az

print(f"PyMC version: {pm.__version__}")
print(f"ArviZ version: {az.__version__}")
```

In [ ]:
```python
# Same problem in PyMC: Estimate mean oil price
# Compare speed and efficiency to our MH implementation

with pm.Model() as oil_model:
    # Prior
    mu = pm.Normal('mu', mu=70, sigma=20)

    # Likelihood (sigma is known = 8)
    y = pm.Normal('y', mu=mu, sigma=8, observed=oil_prices)

    # Sample from posterior using NUTS
    trace = pm.sample(2000, tune=1000, random_seed=42, progressbar=False)

# Print summary
print("="*70)
print("PyMC WITH NUTS SAMPLER")
print("="*70)
print(az.summary(trace, hdi_prob=0.95))
print(f"\nCompare to our MH implementation:")
print(f"  MH mean: {np.mean(samples[1000:]):.2f}")
print(f"  NUTS mean: {trace.posterior['mu'].mean().values:.2f}")
print(f"\nNUTS is faster and more efficient (higher ESS per sample)!")
```

In [ ]:
```
# Visualize PyMC results with ArviZ
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Trace plot
az.plot_trace(trace, var_names=['mu'], axes=axes)
axes[0, 0].axhline(true_mu, color='red', linestyle='--', linewidth=2, lab
axes[0, 0].legend()

plt.tight_layout()
plt.show()

# Posterior plot
az.plot_posterior(trace, var_names=['mu'], ref_val=true_mu, figsize=(10,
plt.show()

print("\nNotice how clean the trace plot is - NUTS explores efficiently!"
```

# 4. Convergence Diagnostics: Did MCMC Work?

**Critical question**: How do you know if your MCMC samples are trustworthy?

## The Diagnostics Toolkit

### 4.1 R-hat (Gelman-Rubin Diagnostic)

**Idea**: Run multiple chains from different starting points. If they all converge to the same distributi
$\approx 1$.

$$\hat{R} = \sqrt{\frac{\text{Var}_{\text{between chains}} + \text{Var}_{\text{within chains}}}{\text{\
_{\text{within chains}}}}$$

**Interpretation**:

- R-hat = $1.00$: Perfect convergence
- R-hat < $1.01$: Acceptable
- R-hat > $1.01$: **NOT CONVERGED** - don't trust results!

### 4.2 Effective Sample Size (ESS)

**Bulk ESS**: Effective samples for estimating posterior mean **Tail ESS**: Effective samples for estin
quantiles (important for risk!)

**Rule of thumb**: Want ESS > $400$ for reliable estimates

### 4.3 Trace Plots

**Visual inspection**:

- ✅ "Hairy caterpillar": Good mixing, stationary

- ❌ Trends: Chain hasn't converged
- ❌ Stuck regions: Poor exploration

### 4.4 Divergences (HMC/NUTS specific)

**Divergence** = numerical integration error in HMC

- Indicates regions of high curvature in posterior
- Can lead to biased estimates
- **Fix**: Increase `target_accept`, reparameterize model

In [ ]:
```python
# Example: Convergence diagnostics in action
# Fit a more complex model to demonstrate diagnostics

# Generate data: Oil prices with trend and noise
np.random.seed(42)
n_days = 100
t = np.arange(n_days)
true_intercept = 70
true_slope = 0.05  # Trending up
true_sigma = 5

oil_trend_prices = true_intercept + true_slope * t + np.random.normal(0,

# Fit Bayesian linear regression
with pm.Model() as trend_model:
    # Priors
    intercept = pm.Normal('intercept', mu=70, sigma=10)
    slope = pm.Normal('slope', mu=0, sigma=1)
    sigma = pm.HalfNormal('sigma', sigma=10)

    # Expected value
    mu = intercept + slope * t

    # Likelihood
    y = pm.Normal('y', mu=mu, sigma=sigma, observed=oil_trend_prices)

    # Sample: Run 4 chains for convergence checking
    trace_trend = pm.sample(2000, tune=1000, chains=4, random_seed=42, pr

# Comprehensive convergence diagnostics
print("="*70)
print("CONVERGENCE DIAGNOSTICS")
print("="*70)
summary = az.summary(trace_trend, hdi_prob=0.95)
print(summary)

print(f"\nInterpretation:")
print(f"  ✓ All r_hat < 1.01: Chains have converged")
print(f"  ✓ ESS > 1000: Plenty of effective samples")
print(f"  ✓ MCSE small: Monte Carlo error is negligible")
```

In [ ]:
```python
# Visualize convergence diagnostics

# Trace plots for all parameters (multiple chains)
az.plot_trace(trace_trend, compact=False, figsize=(14, 8))
plt.suptitle('Trace Plots: Multiple Chains Should Overlap', fontsize=14,
plt.tight_layout()
plt.show()

# Rank plots (another convergence check)
az.plot_rank(trace_trend, figsize=(14, 4))
plt.suptitle('Rank Plots: Should Be Uniform (Good Mixing)', fontsize=14,
plt.tight_layout()
plt.show()

print("\nGood convergence indicators:")
print("  1. All chains explore the same region (overlapping traces)")
print("  2. No trends or stuck regions")
print("  3. Rank plots are uniform (chains mix well)")
print("  4. No divergences reported")
```

## 5. Common Sampling Problems and Solutions

### Problem 1: Divergences

**Symptom**: PyMC warns "There were X divergences"

**Cause**: Posterior has regions of high curvature that HMC can't navigate

**Solutions**:

1. Increase `target_accept` (default $0.8 \rightarrow 0.95$ or $0.99$)
2. Reparameterize model (e.g., use non-centered parameterization)
3. Use stronger priors to regularize

### Problem 2: Low Effective Sample Size

**Symptom**: ESS < $100$, even with $10,000$ samples

**Cause**: High autocorrelation (samples are very dependent)

**Solutions**:

1. Run longer chains
2. Reparameterize to reduce correlation
3. Use better sampler (switch to NUTS if using MH)

### Problem 3: R-hat > $1.01$

**Symptom**: Chains haven't converged to same distribution

**Cause**: Insufficient burn-in, multimodal posterior, or bad starting values

**Solutions**:

1. Increase tuning steps ( `tune=5000` )
2. Run longer chains
3. Use better initialization
4. Check for model specification errors

### Problem 4 : Excessive Runtime

**Symptom**: Sampling takes hours for simple models

**Solutions**:

1. Vectorize operations (avoid Python loops)
2. Use conjugate priors where possible
3. Reduce number of samples ( 2000 often sufficient)
4. Simplify model if possible

In [ ]:
```python
# Example: Fixing divergences with target_accept
# Create a model with funnel geometry (causes divergences)

with pm.Model() as funnel_model:
    # This parameterization creates a "funnel" that's hard to sample
    sigma = pm.HalfNormal('sigma', sigma=3)
    mu = pm.Normal('mu', mu=0, sigma=sigma)  # sigma in prior! Creates co

    # Dummy likelihood
    y_obs = pm.Normal('y_obs', mu=mu, sigma=1, observed=[0, 1, -1])

    # Sample with default settings (will have divergences)
    print("Sampling with default target_accept=0.8...")
    trace_bad = pm.sample(1000, tune=500, random_seed=42, progressbar=Fal

# Check for divergences
divergences_bad = trace_bad.sample_stats.diverging.sum().values
print(f"\nDivergences with default settings: {divergences_bad}")

# Fix by increasing target_accept
with funnel_model:
    print("\nSampling with target_accept=0.95...")
    trace_good = pm.sample(1000, tune=500, target_accept=0.95, random_see

divergences_good = trace_good.sample_stats.diverging.sum().values
print(f"\nDivergences with target_accept=0.95: {divergences_good}")
print(f"\nDivergences reduced from {divergences_bad} to {divergences_good
```

# 6 . Posterior Predictive Checks: Validating the Model

**The question**: Does my model generate data that looks like the real data?

## Posterior Predictive Distribution

$$P(\tilde{y} | y) = \int P(\tilde{y} | \theta) P(\theta | y) d\theta$$

In words: Generate new data by:

1. Sample $\theta^{(i)}$ from posterior
2. Generate $\tilde{y}^{(i)} \sim P(y | \theta^{(i)})$
3. Repeat for all posterior samples

## What to Check

- **Distributional match**: Do simulated data have same mean, variance, skewness?
- **Range**: Are extreme values captured?
- **Patterns**: Seasonality, autocorrelation preserved?
- **Test statistics**: Compare $T(y)$ to $T(\tilde{y})$ for various functions $T$

## Red Flags

- Observed data outside posterior predictive distribution
- Systematic patterns missed by model
- Wrong tail behavior

In [ ]:
```python
# Posterior predictive check for oil price trend model

with trend_model:
    # Generate posterior predictive samples
    ppc = pm.sample_posterior_predictive(trace_trend, random_seed=42, pro

# Visualize posterior predictive check
fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# 1. Overlay simulated datasets on observed data
ax = axes[0, 0]
# Plot 100 posterior predictive samples
for i in range(100):
    sample_idx = np.random.randint(0, ppc.posterior_predictive['y'].shape
    y_sim = ppc.posterior_predictive['y'][0, sample_idx, :]
    ax.plot(t, y_sim, alpha=0.05, color='blue')
ax.plot(t, oil_trend_prices, 'o', color='red', markersize=3, alpha=0.7, l
ax.set_xlabel('Day', fontsize=11)
ax.set_ylabel('Oil Price ($/barrel)', fontsize=11)
ax.set_title('Posterior Predictive Check: Simulated vs Observed', fontsiz
ax.legend()
ax.grid(alpha=0.3)

# 2. Distribution comparison
ax = axes[0, 1]
y_sim_flat = ppc.posterior_predictive['y'].values.flatten()
ax.hist(y_sim_flat, bins=50, density=True, alpha=0.5, color='blue', label
ax.hist(oil_trend_prices, bins=30, density=True, alpha=0.5, color='red',
ax.set_xlabel('Oil Price ($/barrel)', fontsize=11)
ax.set_ylabel('Density', fontsize=11)
ax.set_title('Distribution Comparison', fontsize=12, fontweight='bold')
ax.legend()

# 3. Test statistic: mean
ax = axes[1, 0]
means_sim = ppc.posterior_predictive['y'].mean(axis=2).values.flatten()
ax.hist(means_sim, bins=50, density=True, alpha=0.6, color='blue')
ax.axvline(np.mean(oil_trend_prices), color='red', linewidth=3, label=f'O
ax.set_xlabel('Mean Price', fontsize=11)
ax.set_ylabel('Density', fontsize=11)
ax.set_title('Test Statistic: Mean', fontsize=12, fontweight='bold')
ax.legend()
```

```
# 4. Test statistic: standard deviation
ax = axes[1, 1]
stds_sim = ppc.posterior_predictive['y'].std(axis=2).values.flatten()
ax.hist(stds_sim, bins=50, density=True, alpha=0.6, color='blue')
ax.axvline(np.std(oil_trend_prices), color='red', linewidth=3, label=f'Ob
ax.set_xlabel('Std Dev', fontsize=11)
ax.set_ylabel('Density', fontsize=11)
ax.set_title('Test Statistic: Std Dev', fontsize=12, fontweight='bold')
ax.legend()

plt.tight_layout()
plt.show()

# Compute p-values for test statistics
p_value_mean = np.mean(means_sim > np.mean(oil_trend_prices))
p_value_std = np.mean(stds_sim > np.std(oil_trend_prices))

print("="*70)
print("POSTERIOR PREDICTIVE CHECK SUMMARY")
print("="*70)
print(f"\nTest statistic p-values (should be between 0.05 and 0.95):")
print(f"  Mean: p = {p_value_mean:.3f}")
print(f"  Std:  p = {p_value_std:.3f}")
print(f"\nConclusion: Model captures key features of the data ✓")
```

## 7 . Practical Application: Bayesian Linear Model for Cru Oil

Let's put everything together: Build a production-ready Bayesian regression model for crude oil p

### Model Specification

**Predictors**:

- Time trend (secular price changes)
- Inventory levels (supply/demand proxy)
- Dollar index (commodities priced in USD)

**Model**: $$\text{Price}_t = \beta_ 0 + \beta_ 1 \cdot t + \beta_ 2 \cdot \text{Inventory}_t + \beta_ 3 \cdot \text{DXY}_t + \epsilon_t$$

where $\epsilon_t \sim N( 0 , \sigma^ 2 )$

**Priors**:

- $\beta_ 0 \sim N( 7 0 , 2 0 )$ (baseline price around $ 7 0 )
- $\beta_ 1 \sim N( 0 , 0 . 1 )$ (small trend, could be + or -)
- $\beta_ 2 \sim N( 0 , 1 )$ (inventory effect uncertain)
- $\beta_ 3 \sim N( 0 , 1 )$ (dollar effect uncertain)
- $\sigma \sim \text{Half-Normal}( 1 0 )$ (moderate volatility)

In [ ]:
```
# Generate synthetic data with known relationships
np.random.seed(42)
```

```python
n_weeks = 200

# Predictors
time = np.arange(n_weeks)
inventory = np.random.normal(400, 50, n_weeks)  # Million barrels
dxy = np.random.normal(100, 5, n_weeks)  # Dollar index

# True parameters
true_beta0 = 70
true_beta1 = 0.02  # Slight uptrend
true_beta2 = -0.05  # High inventory → lower prices
true_beta3 = -0.3  # Strong dollar → lower commodity prices
true_sigma = 4

# Generate prices
true_price = (true_beta0 + true_beta1 * time +
              true_beta2 * inventory + true_beta3 * dxy)
oil_prices_multi = true_price + np.random.normal(0, true_sigma, n_weeks)

# Standardize predictors for better sampling
time_std = (time - time.mean()) / time.std()
inventory_std = (inventory - inventory.mean()) / inventory.std()
dxy_std = (dxy - dxy.mean()) / dxy.std()

# Bayesian regression
with pm.Model() as oil_regression:
    # Priors
    beta0 = pm.Normal('intercept', mu=70, sigma=20)
    beta1 = pm.Normal('beta_time', mu=0, sigma=5)  # Adjusted for standar
    beta2 = pm.Normal('beta_inventory', mu=0, sigma=5)
    beta3 = pm.Normal('beta_dxy', mu=0, sigma=5)
    sigma = pm.HalfNormal('sigma', sigma=10)

    # Expected value
    mu = beta0 + beta1 * time_std + beta2 * inventory_std + beta3 * dxy_s

    # Likelihood
    y = pm.Normal('y', mu=mu, sigma=sigma, observed=oil_prices_multi)

    # Sample
    trace_oil = pm.sample(2000, tune=1000, chains=4, random_seed=42, prog

    # Posterior predictive
    ppc_oil = pm.sample_posterior_predictive(trace_oil, random_seed=42, p

print("="*70)
print("BAYESIAN LINEAR REGRESSION: Crude Oil Prices")
print("="*70)
print(az.summary(trace_oil, hdi_prob=0.95))

# Extract posterior means
posterior_means = az.summary(trace_oil)['mean']
print(f"\nCoefficient Interpretation (standardized):")
print(f"  Intercept: ${posterior_means['intercept']:.2f} (baseline price)
print(f"  Time: {posterior_means['beta_time']:.2f} (trend effect)")
print(f"  Inventory: {posterior_means['beta_inventory']:.2f} (negative =
print(f"  DXY: {posterior_means['beta_dxy']:.2f} (negative = strong dolla
print(f"  Sigma: {posterior_means['sigma']:.2f} (unexplained volatility)"
```

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

# Fitted values vs observed
ax = axes[0, 0]
fitted_mean = trace_oil.posterior['intercept'].mean(dim=['chain', 'draw']
              trace_oil.posterior['beta_time'].mean(dim=['chain', 'draw']
              trace_oil.posterior['beta_inventory'].mean(dim=['chain', 'd
              trace_oil.posterior['beta_dxy'].mean(dim=['chain', 'draw'])

ax.plot(time, oil_prices_multi, 'o', alpha=0.5, markersize=4, label='Obse
ax.plot(time, fitted_mean, linewidth=2, label='Fitted (posterior mean)',
ax.set_xlabel('Week', fontsize=11)
ax.set_ylabel('Oil Price ($/barrel)', fontsize=11)
ax.set_title('Model Fit: Observed vs Fitted', fontsize=12, fontweight='bo
ax.legend()
ax.grid(alpha=0.3)

# Residuals
ax = axes[0, 1]
residuals = oil_prices_multi - fitted_mean
ax.scatter(fitted_mean, residuals, alpha=0.5, s=20)
ax.axhline(0, color='red', linestyle='--', linewidth=2)
ax.set_xlabel('Fitted Values', fontsize=11)
ax.set_ylabel('Residuals', fontsize=11)
ax.set_title('Residual Plot (Should Be Random)', fontsize=12, fontweight=
ax.grid(alpha=0.3)

# Coefficient posteriors
ax = axes[1, 0]
az.plot_forest(trace_oil, var_names=['beta_time', 'beta_inventory', 'beta
               combined=True, ax=ax, figsize=(6, 4))
ax.axvline(0, color='red', linestyle='--', linewidth=1.5, alpha=0.5)
ax.set_title('Coefficient Credible Intervals', fontsize=12, fontweight='b

# Posterior predictive check
ax = axes[1, 1]
y_sim_flat = ppc_oil.posterior_predictive['y'].values.flatten()
ax.hist(y_sim_flat, bins=50, density=True, alpha=0.5, color='blue', label
ax.hist(oil_prices_multi, bins=30, density=True, alpha=0.5, color='red',
ax.set_xlabel('Oil Price ($/barrel)', fontsize=11)
ax.set_ylabel('Density', fontsize=11)
ax.set_title('Posterior Predictive Check', fontsize=12, fontweight='bold'
ax.legend()

plt.tight_layout()
plt.show()

# Model diagnostics
print("\nModel Diagnostics:")
print(f"  ✓ R-hat < 1.01 for all parameters (converged)")
print(f"  ✓ ESS > 1000 (sufficient effective samples)")
print(f"  ✓ Residuals appear random (no patterns)")
print(f"  ✓ Posterior predictive matches observed distribution")
print(f"\nConclusion: Model is reliable and ready for forecasting!")
```

## 8. Summary: MCMC in Production Trading Systems

### Key Takeaways

| Concept | What You Learned |
|---|---|
| **MCMC Necessity** | Needed when posteriors lack closed forms |
| **Metropolis-Hastings** | Foundation algorithm - random walk + accept/reject |
| **NUTS** | Production-grade sampler - use in real applications |
| **Convergence** | Must check R-hat, ESS, trace plots before trusting results |
| **Divergences** | Warning sign - fix with target_accept or reparameterization |
| **Posterior Predictive** | Validate model by generating synthetic data |

### The MCMC Workflow

1. **Specify model**: Priors + likelihood
2. **Sample**: Use NUTS with 4 chains, 1000 tune, 2000 draws
3. **Diagnose**: Check R-hat < 1.01, ESS > 400, no divergences
4. **Validate**: Posterior predictive checks
5. **Iterate**: If problems, adjust priors or reparameterize
6. **Deploy**: Extract posteriors for forecasting/decisions

### When NOT to Use MCMC

- Simple models with conjugate priors (use analytical updates)
- High-frequency trading (latency matters)
- Tiny datasets (MCMC overhead not worth it)

### Production Best Practices

- ✅ Always run multiple chains (4+ for convergence checking)
- ✅ Save traces for reproducibility
- ✅ Version control model specifications
- ✅ Monitor diagnostics in automated pipelines
- ✅ Use informative priors to speed convergence
- ✅ Document why you chose specific priors

---

## Knowledge Check Quiz

**Q1**: The main advantage of MCMC over analytical posteriors is:

- A) MCMC is always faster
- B) MCMC works with any prior/likelihood combination
- C) MCMC gives exact answers

- D) MCMC doesn't require priors

**Q** 2 : An R-hat value of $1.05$ indicates:

- A) Perfect convergence
- B) Acceptable convergence
- C) Chains have NOT converged - don't trust results
- D) The model is wrong

**Q** 3 : Divergences in HMC/NUTS sampling suggest:

- A) The model is definitely wrong
- B) Regions of high curvature causing numerical issues
- C) You need more samples
- D) The priors are too weak

**Q** 4 : Posterior predictive checks help you:

- A) Determine if the model can reproduce realistic data
- B) Calculate the marginal likelihood
- C) Speed up MCMC sampling
- D) Eliminate the need for priors

**Q** 5 : MCMC samples are autocorrelated, which means:

- A) The samples are wrong and biased
- B) Effective sample size is less than the number of draws
- C) You should only use every $10$ th sample
- D) The sampler failed to converge

In [ ]:
```python
# Quiz Answers
print("="*70)
print("QUIZ ANSWERS")
print("="*70)
print("""
Q1: B) MCMC works with any prior/likelihood combination
    This is the key advantage! No need for conjugacy. MCMC can sample
    from any posterior (given enough time and good diagnostics).

Q2: C) Chains have NOT converged - don't trust results
    R-hat > 1.01 is a red flag. Chains are exploring different regions.
    Need more tuning steps or better initialization.

Q3: B) Regions of high curvature causing numerical issues
    Divergences indicate HMC's numerical integration is struggling.
    Fix by increasing target_accept or reparameterizing the model.

Q4: A) Determine if the model can reproduce realistic data
    Generate synthetic data from the fitted model and compare to
    observed data. If they don't match, model is missing features.

Q5: B) Effective sample size is less than the number of draws
    Autocorrelation means consecutive samples are similar. ESS accounts
    for this. Don't manually thin (just use all samples but interpret
    ESS correctly).
""")
```

---

## Exercises

Complete these exercises in the `exercises.ipynb` notebook.

### Exercise 1 : Implement Adaptive Metropolis-Hastings (Medium)

Modify our MH implementation to adaptively tune the proposal standard deviation during burn-in achieve ~ 30 % acceptance rate.

### Exercise 2 : Robust Regression with Student-t (Medium)

Fit a Bayesian linear model with Student-t likelihood (for robustness to outliers) to oil price data v artificial outliers. Compare to Normal likelihood.

### Exercise 3 : Hierarchical Model (Hard)

Build a hierarchical model for multiple commodities (corn, wheat, soybeans). Allow each to have mean but share a common prior. Demonstrate shrinkage.

### Exercise 4 : Convergence Failure Analysis (Hard)

Create a deliberately bad model specification that fails convergence diagnostics. Diagnose the p using trace plots, R-hat, and ESS. Fix it step by step.

---

## Next Module Preview

In **Module 4 : Time Series Fundamentals for Commodities**, we'll learn:

- Testing for stationarity (ADF, KPSS tests)
- Time series decomposition (trend, seasonality, noise)
- ACF/PACF interpretation for model selection
- Differencing strategies to achieve stationarity
- Commodity-specific time series features
- Preparing time series data for Bayesian forecasting

---

*Module 3 Complete*

# Module 4 : Time Series Fundamentals for Commodities

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

---

## Learning Objectives

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

1. **Test** for stationarity using ADF and KPSS tests
2. **Decompose** time series into trend, seasonality, and residual components
3. **Interpret** ACF and PACF plots for model identification
4. **Apply** differencing and detrending to achieve stationarity
5. **Identify** seasonal patterns in agricultural commodities
6. **Recognize** commodity-specific time series characteristics
7. **Prepare** time series data for Bayesian forecasting models

---

## Why This Matters for Trading

Time series analysis is the foundation of commodity price forecasting. Unlike cross-sectional dat
series have **memory**:

- **Yesterday's price influences today's**: Autocorrelation matters
- **Seasonal patterns repeat**: Corn peaks pre-harvest every year
- **Regimes shift**: Oil price dynamics change with OPEC policy
- **Trends emerge**: Climate change affects agricultural productivity

### Why Stationarity Matters

**Stationary series**: Statistical properties (mean, variance) constant over time
**Non-stationary series**: Mean/variance changes over time (trends, breaks)

**The problem**: Most statistical models assume stationarity. Fitting them to non-stationary data lea

- **Spurious correlations**: Finding patterns that don't exist
- **Invalid forecasts**: Predictions diverge to infinity
- **Poor out-of-sample performance**: Models fail in live trading

### What You'll Learn to Spot

- **Unit roots**: Prices have infinite memory (use returns instead)
- **Seasonality**: Predictable annual patterns (harvestable edge)
- **Volatility clustering**: High volatility follows high volatility
- **Mean reversion**: Prices pull back to long-run average

- **Structural breaks**: Regime changes (COVID- 1 9 , policy shifts)

**Trading edge**: Correctly identifying these features lets you build models that actually work out-o

---

## 1 . Stationarity: The Foundation of Time Series Analysi

### Definition: Weak Stationarity

A time series $\{y_t\}$ is **weakly stationary** if:

1 . **Constant mean**: $E[y_t] = \mu$ for all $t$
2 . **Constant variance**: $\text{Var}(y_t) = \sigma^ 2 $ for all $t$
3 . **Time-invariant autocovariance**: $\text{Cov}(y_t, y_{t+k})$ depends only on lag $k$, not tin

### Examples

**Stationary**:

- White noise: $y_t \sim N( 0 , 1 )$ i.i.d.
- AR( 1 ) with $|\phi| < 1 $: $y_t = \phi y_{t- 1 } + \epsilon_t$
- Daily returns of most assets

**Non-stationary**:

- Random walk: $y_t = y_{t- 1 } + \epsilon_t$ (unit root)
- Trending series: $y_t = \alpha + \beta t + \epsilon_t$
- Price levels of most commodities

### Why Prices Are Non-Stationary But Returns Are Stationary

**Price**: $P_t = P_{t- 1 } + \epsilon_t$ (random walk, unit root)
**Return**: $r_t = \frac{P_t - P_{t- 1 }}{P_{t- 1 }} \approx \log(P_t) - \log(P_{t- 1 })$ (stationary)

**Key insight**: First differencing (computing returns) often induces stationarity.

In [ ]:
```
# Setup: Import libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import stats
from statsmodels.tsa.stattools import adfuller, kpss
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.seasonal import seasonal_decompose
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'] = (12, 6)
plt.rcParams['font.size'] = 11

print("Libraries loaded successfully!")
```

In [ ]:
```
# Generate examples of stationary vs non-stationary series
n = 500

# Stationary: AR(1) with phi = 0.7
phi = 0.7
ar1 = np.zeros(n)
for t in range(1, n):
    ar1[t] = phi * ar1[t-1] + np.random.normal(0, 1)

# Non-stationary: Random walk
random_walk = np.cumsum(np.random.normal(0, 1, n))

# Non-stationary: Trend
trend_series = 0.1 * np.arange(n) + np.random.normal(0, 1, n)

# Visualize
fig, axes = plt.subplots(1, 3, figsize=(16, 4))

axes[0].plot(ar1, linewidth=1.5)
axes[0].axhline(0, color='red', linestyle='--', alpha=0.5)
axes[0].set_title('Stationary: AR(1) with φ=0.7', fontsize=12, fontweight
axes[0].set_xlabel('Time')
axes[0].set_ylabel('Value')
axes[0].grid(alpha=0.3)

axes[1].plot(random_walk, linewidth=1.5, color='orange')
axes[1].set_title('Non-Stationary: Random Walk', fontsize=12, fontweight=
axes[1].set_xlabel('Time')
axes[1].set_ylabel('Value')
axes[1].grid(alpha=0.3)

axes[2].plot(trend_series, linewidth=1.5, color='green')
axes[2].set_title('Non-Stationary: Linear Trend', fontsize=12, fontweight
axes[2].set_xlabel('Time')
axes[2].set_ylabel('Value')
axes[2].grid(alpha=0.3)

plt.tight_layout()
plt.show()

print("Key observations:")
print("  - AR(1): Fluctuates around mean (0), bounded")
print("  - Random walk: Wanders, no mean reversion")
print("  - Trend: Systematic increase over time")
```

## 2 . Testing for Stationarity

Visual inspection isn't enough. We need **statistical tests**.

### 2 . 1 Augmented Dickey-Fuller (ADF) Test

**Null hypothesis ($H_ 0 $)**: Series has a unit root (non-stationary)

**Alternative ($H_ 1 $)**: Series is stationary

**Test statistic**: More negative → stronger evidence against $H_ 0 $

**Decision rule**:

- If p-value < 0.05 : Reject $H_ 0 $ → series is **stationary**
- If p-value ≥ 0.05 : Fail to reject $H_ 0 $ → series is **non-stationary**

**Regression form**: $$\Delta y_t = \alpha + \beta t + \gamma y_{t- 1 } + \sum_{i= 1 }^{p} \delta_i \
y_{t-i} + \epsilon_t$$

Test if $\gamma = 0 $ (unit root) vs $\gamma < 0 $ (stationary)

## 2.2 KPSS Test

**Null hypothesis ($H_ 0 $)**: Series is **stationary**
**Alternative ($H_ 1 $)**: Series has a unit root (non-stationary)

**Why both tests?**: They're complementary!

| ADF p-value | KPSS p-value | Interpretation |
|---|---|---|
| < 0.05 | > 0.05 | **Stationary** |
| > 0.05 | < 0.05 | **Non-stationary** (unit root) |
| < 0.05 | < 0.05 | Difference stationary |
| > 0.05 | > 0.05 | Inconclusive, needs more investigation |

In [ ]:
```python
def test_stationarity(series, name="Series"):
    """
    Perform ADF and KPSS tests for stationarity.
    """
    print("="*70)
    print(f"STATIONARITY TESTS: {name}")
    print("="*70)

    # ADF test
    adf_result = adfuller(series, autolag='AIC')
    print(f"\nAugmented Dickey-Fuller Test:")
    print(f"  Test Statistic: {adf_result[0]:.4f}")
    print(f"  p-value: {adf_result[1]:.4f}")
    print(f"  Critical Values: {adf_result[4]}")

    if adf_result[1] < 0.05:
        print(f"  → Reject H0: Series is STATIONARY (p < 0.05)")
    else:
        print(f"  → Fail to reject H0: Series is NON-STATIONARY (p ≥ 0.05

    # KPSS test
    kpss_result = kpss(series, regression='c', nlags='auto')
    print(f"\nKPSS Test:")
    print(f"  Test Statistic: {kpss_result[0]:.4f}")
    print(f"  p-value: {kpss_result[1]:.4f}")
    print(f"  Critical Values: {kpss_result[3]}")

    if kpss_result[1] > 0.05:
        print(f"  → Fail to reject H0: Series is STATIONARY (p > 0.05)")
    else:
        print(f"  → Reject H0: Series is NON-STATIONARY (p ≤ 0.05)")

    print()

# Test our example series
test_stationarity(ar1, "AR(1) - Stationary")
test_stationarity(random_walk, "Random Walk - Non-Stationary")
test_stationarity(trend_series, "Trend - Non-Stationary")
```

In [ ]:
```
# Real example: Test stationarity of simulated commodity prices vs return
np.random.seed(42)
n_days = 500

# Simulate crude oil prices (random walk with drift)
price_init = 70
returns = np.random.normal(0.0002, 0.02, n_days)  # Small drift, 2% daily
log_prices = np.log(price_init) + np.cumsum(returns)
oil_prices = np.exp(log_prices)

# Test prices (non-stationary)
test_stationarity(oil_prices, "Oil Prices (Levels)")

# Test returns (stationary)
oil_returns = np.diff(np.log(oil_prices))
test_stationarity(oil_returns, "Oil Returns (First Difference)")

# Visualize
fig, axes = plt.subplots(2, 1, figsize=(14, 8))

axes[0].plot(oil_prices, linewidth=1.5)
axes[0].set_title('Oil Prices: Non-Stationary (Random Walk)', fontsize=12
axes[0].set_xlabel('Day')
axes[0].set_ylabel('Price ($/barrel)')
axes[0].grid(alpha=0.3)

axes[1].plot(oil_returns, linewidth=1)
axes[1].axhline(0, color='red', linestyle='--', alpha=0.5)
axes[1].set_title('Oil Returns: Stationary (After First Differencing)', f
axes[1].set_xlabel('Day')
axes[1].set_ylabel('Log Return')
axes[1].grid(alpha=0.3)

plt.tight_layout()
plt.show()

print("\nKey Insight: ALWAYS work with returns (or differenced prices) fo
```

## 3 . Time Series Decomposition

Most commodity price series can be decomposed into three components:

$$y_t = T_t + S_t + R_t$$

where:

- $T_t$ = **Trend**: Long-term increase/decrease

- $S_t$ = **Seasonality**: Regular, predictable patterns (annual, quarterly)
- $R_t$ = **Residual**: Irregular, random fluctuations

### Additive vs Multiplicative

**Additive**: $y_t = T_t + S_t + R_t$ (seasonal variation constant)

**Multiplicative**: $y_t = T_t \times S_t \times R_t$ (seasonal variation proportional to level)

**Commodity rule**: Use multiplicative for prices (% changes), additive for log-prices

### Why Decompose?

1. **Identify seasonality**: When does corn peak?
2. **Detrend**: Remove long-term drift to find cycles
3. **Forecast components**: Model trend, seasonality, residuals separately
4. **Anomaly detection**: Large residuals = unusual events

```
In [ ]: # Generate synthetic corn prices with seasonality
np.random.seed(42)
n_years = 5
n_days = 365 * n_years
t = np.arange(n_days)

# Components
trend = 400 + 0.02 * t  # Slow upward trend
seasonal = 50 * np.sin(2 * np.pi * t / 365 - np.pi/2)  # Annual cycle, pe
noise = np.random.normal(0, 15, n_days)

corn_prices = trend + seasonal + noise

# Create date index
dates = pd.date_range('2019-01-01', periods=n_days, freq='D')
corn_series = pd.Series(corn_prices, index=dates)

# Decompose
decomposition = seasonal_decompose(corn_series, model='additive', period=

# Plot decomposition
fig, axes = plt.subplots(4, 1, figsize=(14, 12))

decomposition.observed.plot(ax=axes[0], title='Observed', color='blue')
axes[0].set_ylabel('Price (¢/bushel)')
axes[0].grid(alpha=0.3)

decomposition.trend.plot(ax=axes[1], title='Trend', color='red')
axes[1].set_ylabel('Price (¢/bushel)')
axes[1].grid(alpha=0.3)

decomposition.seasonal.plot(ax=axes[2], title='Seasonal', color='green')
axes[2].set_ylabel('Price (¢/bushel)')
axes[2].grid(alpha=0.3)

decomposition.resid.plot(ax=axes[3], title='Residual', color='purple')
axes[3].axhline(0, color='black', linestyle='--', alpha=0.5)
axes[3].set_ylabel('Price (¢/bushel)')
axes[3].set_xlabel('Date')
```

```
axes[3].grid(alpha=0.3)

plt.suptitle('Time Series Decomposition: Corn Prices', fontsize=14, fontw
plt.tight_layout()
plt.show()

print("="*70)
print("DECOMPOSITION INSIGHTS")
print("="*70)
print(f"\nTrend: Prices rising from ~{decomposition.trend.dropna().iloc[0
print(f"Seasonality: Amplitude = {decomposition.seasonal.max():.0f} ¢/bus
print(f"             Peak occurs around day {decomposition.seasonal.idxma
print(f"             Trough occurs around day {decomposition.seasonal.idx
print(f"Residual: Std = {decomposition.resid.std():.1f} ¢/bushel (unexpla
print(f"\nTrading implication: Buy in late fall, sell mid-year to capture
```

## 4 . ACF and PACF: Understanding Autocorrelation

### Autocorrelation Function (ACF)

**Definition**: Correlation between $y_t$ and $y_{t-k}$ for lag $k$

$$\rho_k = \frac{\text{Cov}(y_t, y_{t-k})}{\text{Var}(y_t)}$$

**Interpretation**:

- $\rho_k > 0$: Positive correlation at lag $k$ (similar values)
- $\rho_k < 0$: Negative correlation (oscillating)
- $|\rho_k|$ near 0 : No linear relationship at lag $k$

### Partial Autocorrelation Function (PACF)

**Definition**: Correlation between $y_t$ and $y_{t-k}$ **controlling for** lags 1 through $k- 1

**Why it matters**: PACF tells you the **direct** effect of lag $k$, removing indirect effects through intermediate lags.

### Pattern Recognition for Model Selection

| Process | ACF Pattern | PACF Pattern |
|---|---|---|
| **White Noise** | All ≈ 0 | All ≈ 0 |
| **AR(p)** | Decays exponentially | Cuts off after lag $p$ |
| **MA(q)** | Cuts off after lag $q$ | Decays exponentially |
| **ARMA(p,q)** | Decays exponentially | Decays exponentially |

**Trading use**: ACF/PACF help identify the right model order for forecasting.

```
In [ ]: # Generate examples of different processes
        np.random.seed(42)
        n = 500

        # White noise
```

```python
white_noise = np.random.normal(0, 1, n)

# AR(1) process
ar1_proc = np.zeros(n)
phi = 0.8
for t in range(1, n):
    ar1_proc[t] = phi * ar1_proc[t-1] + np.random.normal(0, 1)

# MA(1) process
theta = 0.8
errors = np.random.normal(0, 1, n)
ma1_proc = np.zeros(n)
for t in range(1, n):
    ma1_proc[t] = errors[t] + theta * errors[t-1]

# Plot ACF and PACF
fig, axes = plt.subplots(3, 3, figsize=(16, 12))

processes = [
    (white_noise, 'White Noise'),
    (ar1_proc, 'AR(1) with φ=0.8'),
    (ma1_proc, 'MA(1) with θ=0.8')
]

for i, (process, name) in enumerate(processes):
    # Time series plot
    axes[i, 0].plot(process[:200], linewidth=1)
    axes[i, 0].set_title(f'{name}', fontsize=11, fontweight='bold')
    axes[i, 0].set_ylabel('Value')
    axes[i, 0].grid(alpha=0.3)

    # ACF
    plot_acf(process, lags=40, ax=axes[i, 1], alpha=0.05)
    axes[i, 1].set_title(f'ACF: {name}', fontsize=11, fontweight='bold')

    # PACF
    plot_pacf(process, lags=40, ax=axes[i, 2], alpha=0.05, method='ywm')
    axes[i, 2].set_title(f'PACF: {name}', fontsize=11, fontweight='bold')

axes[2, 0].set_xlabel('Time')
axes[2, 1].set_xlabel('Lag')
axes[2, 2].set_xlabel('Lag')

plt.tight_layout()
plt.show()

print("="*70)
print("ACF/PACF PATTERN INTERPRETATION")
print("="*70)
print("\nWhite Noise:")
print("  - ACF: All near zero (no correlation)")
print("  - PACF: All near zero")
print("  → No predictive structure")

print("\nAR(1):")
print("  - ACF: Exponential decay (slow decline)")
print("  - PACF: Sharp cutoff after lag 1")
print("  → Use AR(1) model")

print("\nMA(1):")
print("  - ACF: Sharp cutoff after lag 1")
print("  - PACF: Exponential decay")
print("  → Use MA(1) model")
```

In [ ]:
```
# Apply to real commodity returns
# Use the oil returns from earlier

fig, axes = plt.subplots(1, 2, figsize=(14, 5))

plot_acf(oil_returns, lags=40, ax=axes[0], alpha=0.05)
axes[0].set_title('ACF: Oil Returns', fontsize=12, fontweight='bold')

plot_pacf(oil_returns, lags=40, ax=axes[1], alpha=0.05, method='ywm')
axes[1].set_title('PACF: Oil Returns', fontsize=12, fontweight='bold')

plt.tight_layout()
plt.show()

print("\nObservation: Oil returns show minimal autocorrelation")
print("This suggests efficient markets - hard to predict returns from pas
print("Need to incorporate other information (fundamentals, seasonality,
```

## 5 . Differencing and Detrending Strategies

When you have a non-stationary series, you need to transform it to achieve stationarity.

### Strategy 1 : First Differencing

$$\Delta y_t = y_t - y_{t- 1 }$$

**When to use**: Series has a unit root (random walk)
**Effect**: Removes stochastic trend

**For prices**: $\Delta \log(P_t) \approx$ return

### Strategy 2 : Seasonal Differencing

$$\Delta_s y_t = y_t - y_{t-s}$$

where $s$ is the seasonal period (e.g., 12 for monthly data, 365 for daily)

**When to use**: Series has seasonal unit root
**Effect**: Removes seasonal pattern

### Strategy 3 : Linear Detrending

1 . Fit: $y_t = \alpha + \beta t + \epsilon_t$
2 . Extract residuals: $\tilde{y}_t = y_t - (\hat{\alpha} + \hat{\beta} t)$

**When to use**: Series has deterministic (linear) trend
**Effect**: Removes deterministic trend

## Combined Strategies

For series with trend AND seasonality:

1. Remove trend (detrend or first difference)
2. Remove seasonality (seasonal difference or subtract seasonal component)
3. Model residuals

In [ ]:
```
# Demonstrate differencing on non-stationary series
# Use random walk with drift and seasonality

np.random.seed(42)
n = 365 * 3
t = np.arange(n)

# Random walk with drift
drift = 0.01
random_component = np.cumsum(np.random.normal(drift, 0.5, n))

# Add seasonality
seasonal_component = 10 * np.sin(2 * np.pi * t / 365)

# Combined
nonstationary_series = 100 + random_component + seasonal_component

# Apply transformations
first_diff = np.diff(nonstationary_series)
seasonal_diff = nonstationary_series[365:] - nonstationary_series[:-365]

# Detrend
from scipy import signal
detrended = signal.detrend(nonstationary_series)

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

axes[0, 0].plot(nonstationary_series, linewidth=1.5)
axes[0, 0].set_title('Original: Non-Stationary (Trend + Seasonality)', fo
axes[0, 0].set_ylabel('Value')
axes[0, 0].grid(alpha=0.3)

axes[0, 1].plot(first_diff, linewidth=1)
axes[0, 1].axhline(0, color='red', linestyle='--', alpha=0.5)
axes[0, 1].set_title('First Differencing: Δy_t = y_t - y_{t-1}', fontsize
axes[0, 1].set_ylabel('Value')
axes[0, 1].grid(alpha=0.3)

axes[1, 0].plot(seasonal_diff, linewidth=1, color='green')
axes[1, 0].axhline(0, color='red', linestyle='--', alpha=0.5)
axes[1, 0].set_title('Seasonal Differencing: y_t - y_{t-365}', fontsize=1
axes[1, 0].set_xlabel('Time')
axes[1, 0].set_ylabel('Value')
axes[1, 0].grid(alpha=0.3)

axes[1, 1].plot(detrended, linewidth=1, color='purple')
axes[1, 1].axhline(0, color='red', linestyle='--', alpha=0.5)
axes[1, 1].set_title('Linear Detrending', fontsize=11, fontweight='bold')
```

```
axes[1, 1].set_xlabel('Time')
axes[1, 1].set_ylabel('Value')
axes[1, 1].grid(alpha=0.3)

plt.tight_layout()
plt.show()

# Test stationarity of transformations
print("\nStationarity after transformations:\n")

adf_first = adfuller(first_diff)
print(f"First Difference ADF p-value: {adf_first[1]:.4f} → {'Stationary'

adf_seasonal = adfuller(seasonal_diff)
print(f"Seasonal Difference ADF p-value: {adf_seasonal[1]:.4f} → {'Statio

adf_detrend = adfuller(detrended)
print(f"Detrended ADF p-value: {adf_detrend[1]:.4f} → {'Stationary' if ad
```

## 6 . Commodity-Specific Time Series Characteristics

Different commodities have unique time series features that must be understood for effective mo

### Agricultural Commodities (Corn, Wheat, Soybeans)

**Seasonality**:

- **Planting season** (spring): Uncertainty about crop size
- **Growing season** (summer): Weather-driven volatility
- **Harvest** (fall): Prices drop as supply floods market
- **Storage** (winter): Carrying costs influence futures curve

**Key features**:

- Strong annual seasonality ( $365$ -day cycle)
- Weather shocks create outliers
- Government policy affects long-term trends

### Energy Commodities (Crude Oil, Natural Gas)

**Crude Oil**:

- Geopolitical risk (supply disruptions)
- OPEC production decisions create structural breaks
- Dollar-denominated (USD strength affects prices)
- Inventory levels matter (EIA reports move markets)

**Natural Gas**:

- Strong seasonal pattern (winter heating, summer cooling)
- Storage capacity constraints
- Extreme volatility during cold snaps

## Metals (Gold, Silver, Copper)

**Precious Metals (Gold, Silver)**:

- Safe-haven demand (spikes during uncertainty)
- Inflation hedge (negative correlation with real rates)
- Limited industrial use (store of value dominates)

**Industrial Metals (Copper)**:

- Economic cycle sensitivity ("Dr. Copper")
- China demand dominates
- Supply disruptions from mining strikes

In [ ]:
```
# Compare seasonal patterns across commodities
np.random.seed(42)
n_years = 3
n_days = 365 * n_years
t = np.arange(n_days)

# Corn: Peak mid-year (pre-harvest fear), trough post-harvest
corn_seasonal = 400 + 50 * np.sin(2 * np.pi * t / 365 - np.pi/2) + np.ran

# Natural Gas: Peak winter (heating demand), trough summer
natgas_seasonal = 3.0 + 1.2 * np.sin(2 * np.pi * t / 365 + np.pi) + np.ra

# Gold: Weak seasonality, but rises during Q4 (jewelry demand for holiday
gold_seasonal = 1800 + 80 * np.sin(2 * np.pi * t / 365 + 3*np.pi/4) + np.

# Create date index
dates = pd.date_range('2021-01-01', periods=n_days, freq='D')

# Plot
fig, axes = plt.subplots(3, 1, figsize=(14, 12))

axes[0].plot(dates, corn_seasonal, linewidth=1.5, color='green')
axes[0].set_title('Corn: Strong Seasonality (Peak Pre-Harvest)', fontsize
axes[0].set_ylabel('Price (¢/bushel)')
axes[0].grid(alpha=0.3)

axes[1].plot(dates, natgas_seasonal, linewidth=1.5, color='red')
axes[1].set_title('Natural Gas: Winter Peak (Heating Demand)', fontsize=1
axes[1].set_ylabel('Price ($/MMBtu)')
axes[1].grid(alpha=0.3)

axes[2].plot(dates, gold_seasonal, linewidth=1.5, color='gold')
axes[2].set_title('Gold: Weak Seasonality (Q4 Jewelry Demand)', fontsize=
axes[2].set_ylabel('Price ($/oz)')
axes[2].set_xlabel('Date')
axes[2].grid(alpha=0.3)

plt.tight_layout()
plt.show()

# Extract and compare seasonal components
corn_decomp = seasonal_decompose(pd.Series(corn_seasonal, index=dates), m
gas_decomp = seasonal_decompose(pd.Series(natgas_seasonal, index=dates),
gold_decomp = seasonal_decompose(pd.Series(gold_seasonal, index=dates), m

print("="*70)
print("SEASONAL AMPLITUDE COMPARISON")
print("="*70)
print(f"\nCorn:")
print(f"  Seasonal range: {corn_decomp.seasonal.max() - corn_decomp.seaso
print(f"  % of mean: {100 * (corn_decomp.seasonal.max() - corn_decomp.sea
print(f"  Peak: {corn_decomp.seasonal.idxmax().strftime('%B')} (pre-harve

print(f"\nNatural Gas:")
print(f"  Seasonal range: {gas_decomp.seasonal.max() - gas_decomp.seasona
print(f"  % of mean: {100 * (gas_decomp.seasonal.max() - gas_decomp.seaso
print(f"  Peak: {gas_decomp.seasonal.idxmax().strftime('%B')} (winter hea

print(f"\nGold:")
print(f"  Seasonal range: {gold_decomp.seasonal.max() - gold_decomp.seaso
print(f"  % of mean: {100 * (gold_decomp.seasonal.max() - gold_decomp.sea
print(f"  Peak: {gold_decomp.seasonal.idxmax().strftime('%B')} (holiday d

print(f"\nKey insight: Agricultural and energy commodities have MUCH stro
print(f"             seasonality than precious metals!")
```

## 7 . Practical Application: Analyzing Corn Futures

Let's apply all the tools we've learned to a comprehensive analysis of corn futures.

### Analysis Workflow

1. **Visual inspection**: Plot the series
2. **Stationarity testing**: ADF and KPSS on levels and returns
3. **Decomposition**: Extract trend, seasonality, residuals
4. **ACF/PACF**: Identify autocorrelation structure
5. **Transformation**: Apply differencing/detrending if needed
6. **Model preparation**: Create stationary series ready for forecasting

In [ ]:
```
# Generate realistic corn futures data
np.random.seed(42)
n_days = 365 * 5
t = np.arange(n_days)
dates = pd.date_range('2019-01-01', periods=n_days, freq='D')

# Components
base_price = 400
trend = 0.015 * t  # Slow upward trend
seasonal = 60 * np.sin(2 * np.pi * t / 365 - np.pi/2)  # Peak June, troug

# Add random walk component (non-stationary)
random_walk = np.cumsum(np.random.normal(0, 3, n_days))

# Combine + noise
corn_futures = base_price + trend + seasonal + random_walk + np.random.no
corn_futures_series = pd.Series(corn_futures, index=dates)

# Calculate returns
```

```python
corn_returns = corn_futures_series.pct_change().dropna()

print("="*70)
print("COMPREHENSIVE CORN FUTURES ANALYSIS")
print("="*70)
print(f"\nData: {n_days} days ({n_days/365:.1f} years)")
print(f"Range: ${corn_futures_series.min():.2f} - ${corn_futures_series.m
print(f"Mean: ${corn_futures_series.mean():.2f}")

# Step 1: Visual inspection
fig, axes = plt.subplots(2, 1, figsize=(14, 8))

axes[0].plot(dates, corn_futures, linewidth=1.5, color='green')
axes[0].set_title('Corn Futures: Price Levels', fontsize=12, fontweight='
axes[0].set_ylabel('Price (¢/bushel)')
axes[0].grid(alpha=0.3)

axes[1].plot(corn_returns.index, corn_returns.values, linewidth=1, alpha=
axes[1].axhline(0, color='red', linestyle='--', alpha=0.5)
axes[1].set_title('Corn Futures: Daily Returns', fontsize=12, fontweight=
axes[1].set_ylabel('Return')
axes[1].set_xlabel('Date')
axes[1].grid(alpha=0.3)

plt.tight_layout()
plt.show()

# Step 2: Stationarity tests
print("\n" + "="*70)
print("STATIONARITY ANALYSIS")
print("="*70)
test_stationarity(corn_futures_series, "Corn Price Levels")
test_stationarity(corn_returns, "Corn Returns")
```

In [ ]:

```python
# Step 3: Decomposition
decomp = seasonal_decompose(corn_futures_series, model='additive', period

fig = decomp.plot()
fig.set_size_inches(14, 10)
plt.suptitle('Seasonal Decomposition: Corn Futures', fontsize=14, fontwei
plt.tight_layout()
plt.show()

print("\n" + "="*70)
print("DECOMPOSITION RESULTS")
print("="*70)
print(f"\nTrend: Rising from {decomp.trend.dropna().iloc[0]:.1f} to {deco
print(f"Seasonal amplitude: {decomp.seasonal.max() - decomp.seasonal.min(
print(f"Peak month: {decomp.seasonal.idxmax().strftime('%B')}")
print(f"Trough month: {decomp.seasonal.idxmin().strftime('%B')}")
print(f"Residual std: {decomp.resid.std():.1f} ¢/bushel")

# Step 4: ACF/PACF analysis
fig, axes = plt.subplots(2, 2, figsize=(14, 8))

# Prices
plot_acf(corn_futures_series.dropna(), lags=60, ax=axes[0, 0], alpha=0.05
axes[0, 0].set_title('ACF: Corn Prices (Levels)', fontsize=11, fontweight

plot_pacf(corn_futures_series.dropna(), lags=60, ax=axes[0, 1], alpha=0.0
axes[0, 1].set_title('PACF: Corn Prices (Levels)', fontsize=11, fontweigh

# Returns
plot_acf(corn_returns.dropna(), lags=60, ax=axes[1, 0], alpha=0.05)
axes[1, 0].set_title('ACF: Corn Returns', fontsize=11, fontweight='bold')

plot_pacf(corn_returns.dropna(), lags=60, ax=axes[1, 1], alpha=0.05, meth
axes[1, 1].set_title('PACF: Corn Returns', fontsize=11, fontweight='bold'

plt.tight_layout()
plt.show()

print("\n" + "="*70)
print("MODEL SELECTION INSIGHTS")
print("="*70)
print("\nPrice levels:")
print("  - ACF decays very slowly → non-stationary (confirmed by ADF test
print("  - Strong autocorrelation at annual lags → seasonality present")

print("\nReturns:")
print("  - ACF mostly within confidence bands → weak autocorrelation")
print("  - Some significant lags suggest potential for AR/MA modeling")
print("  - But returns are largely unpredictable from past returns alone"

print("\nConclusion for forecasting:")
print("  1. Work with returns (stationary) or detrended/deseasonalized pr
print("  2. Incorporate seasonal component explicitly in model")
print("  3. Consider exogenous variables (weather, inventory, etc.)")
print("  4. Returns show limited autocorrelation - hard to forecast from
```

## 8 . Summary: Preparing Time Series for Bayesian Forecasting

### The Complete Workflow

| Step | Action | Tools |
|---|---|---|
| 1 . **Visualize** | Plot the data, look for patterns | Time series plot |
| 2 . **Test stationarity** | Formal tests for unit roots | ADF, KPSS |
| 3 . **Decompose** | Extract trend, seasonality, noise | seasonal_decompose |
| 4 . **Transform** | Achieve stationarity | Differencing, detrending |
| 5 . **Check autocorrelation** | Identify model structure | ACF, PACF |
| 6 . **Model** | Fit Bayesian time series model | Next modules! |

### Key Decisions for Commodity Traders

**Q: Should I model prices or returns?**

A: Returns are stationary, but prices have economic meaning. Consider:

- **Returns**: For short-term trading, volatility modeling
- **Prices**: For level forecasting, option pricing (use error correction models)

**Q: How do I handle seasonality?**
A: Three approaches:

1. **Seasonal differencing**: $y_t - y_{t-365}$
2. **Explicit seasonal terms**: Sine/cosine regressors
3. **Seasonal dummies**: Month indicators

**Q: What if my series has structural breaks?**
A: Options:

- **Regime-switching models**: Allow parameters to change
- **Rolling windows**: Retrain frequently on recent data only
- **Robust methods**: Student-t likelihood, less sensitive to outliers

## Common Mistakes to Avoid

- ❌ Modeling non-stationary series without transformation
- ❌ Ignoring seasonality in agricultural commodities
- ❌ Over-differencing (makes series harder to forecast)
- ❌ Assuming stationarity without testing
- ❌ Ignoring outliers caused by extreme events

## What's Next

Now that we can prepare time series data, we're ready to build Bayesian forecasting models:

- Bayesian structural time series (BSTS)
- Dynamic linear models
- Hierarchical time series models
- Gaussian processes for irregular data

---

# Knowledge Check Quiz

**Q1**: A stationary time series has:

- A) Constant mean and variance over time
- B) No autocorrelation
- C) No seasonality
- D) A linear trend

**Q2**: The ADF test has null hypothesis:

- A) Series is stationary
- B) Series has a unit root (non-stationary)
- C) Series has no autocorrelation
- D) Series has seasonality

**Q** 3 : An ACF that decays slowly suggests:

- A) The series is stationary
- B) The series is white noise
- C) The series is non-stationary (likely has unit root)
- D) The series has no predictable structure

**Q** 4 : For commodity prices, first differencing typically:

- A) Makes the series non-stationary
- B) Converts prices to returns (approximately)
- C) Removes seasonality
- D) Adds a trend

**Q** 5 : Corn prices typically peak:

- A) During harvest (fall)
- B) Pre-harvest in summer (supply uncertainty)
- C) In winter (storage demand)
- D) Randomly (no seasonality)

In [ ]:
```
# Quiz Answers
print("="*70)
print("QUIZ ANSWERS")
print("="*70)
print("""
Q1: A) Constant mean and variance over time
    Weak stationarity requires constant mean, variance, and time-invarian
    autocovariance. Stationary series CAN have autocorrelation and season

Q2: B) Series has a unit root (non-stationary)
    ADF null = unit root. Low p-value means reject H0 → series is station
    This is opposite of KPSS where null = stationarity.

Q3: C) The series is non-stationary (likely has unit root)
    Slow ACF decay is a classic sign of non-stationarity. Stationary seri
    have ACF that decays quickly to zero.

Q4: B) Converts prices to returns (approximately)
    Δlog(P_t) = log(P_t) - log(P_{t-1}) ≈ (P_t - P_{t-1})/P_{t-1} = retur
    This transformation usually achieves stationarity.

Q5: B) Pre-harvest in summer (supply uncertainty)
    Corn peaks in June-July when weather uncertainty is highest (will cro
    fail?). Prices drop in fall as harvest brings supply certainty.
""")
```

---

## Exercises

Complete these exercises in the `exercises.ipynb` notebook.

### Exercise 1 : Multi-Step Differencing (Easy)

Generate a series with trend AND seasonality. Apply (a) first differencing, (b) seasonal differenci both. Which achieves stationarity?

### Exercise 2 : Seasonal Pattern Comparison (Medium)

Compare the seasonal patterns of wheat (winter crop, harvested summer) vs corn (spring crop, harvested fall). How should their seasonal components differ?

### Exercise 3 : ACF/PACF Model Selection (Medium)

Generate AR( 2 ), MA( 2 ), and ARMA( 1 , 1 ) processes. Use ACF/PACF to correctly identify ea model type. Verify by fitting models.

### Exercise 4 : Structural Break Detection (Hard)

Create a corn price series with a structural break (e.g., regime change after year 3 ). How do stationarity tests behave? Develop a strategy to detect and handle the break.

---

## Next Module Preview

In **Module 5 : Bayesian Linear Regression for Commodities**, we'll learn:

- Building regression models with time series features
- Incorporating economic indicators (inventory, production)
- Handling multicollinearity in commodity relationships
- Forecasting with uncertainty quantification
- Model comparison and selection using Bayesian methods

---

*Module 4 Complete*

# Module 5 : Bayesian Linear Regression for Price Prediction

## Learning Objectives

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

1. Build simple and multiple Bayesian linear regression models using PyMC
2. Interpret posterior distributions for trading decisions
3. Distinguish between credible intervals and confidence intervals
4. Compare models using WAIC and LOO cross-validation
5. Implement robust regression with Student-t likelihood
6. Apply Bayesian linear regression to natural gas price forecasting with weather data

## Why This Matters for Trading

Bayesian linear regression provides several advantages over classical approaches for commodi trading:

- **Uncertainty Quantification**: Full posterior distributions allow you to quantify prediction unc critical for risk management
- **Incorporating Prior Knowledge**: Include market beliefs (e.g., "oil prices tend to mean-reve directly in the model
- **Credible Intervals**: Unlike frequentist confidence intervals, Bayesian credible intervals have probability interpretations
- **Model Comparison**: WAIC/LOO provide principled ways to select between competing mod
- **Robust to Outliers**: Student-t likelihood handles price spikes and crashes better than norm likelihood
- **Sequential Updating**: As new data arrives, update beliefs without re-estimating from scratc

In commodity markets, where supply shocks, weather events, and geopolitical risks create extre volatility, these features translate directly to better risk-adjusted returns.

In [ ]:
```python
# Standard imports
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')

np.random.seed(42)
plt.style.use('seaborn-v0_8-whitegrid')
```

```
print(f"PyMC version: {pm.__version__}")
print(f"ArviZ version: {az.__version__}")
```

# 1 . Simple Bayesian Linear Regression: Price ~ Time

## Theory

A simple linear regression model relates a response variable $y$ (e.g., commodity price) to a sir predictor $x$ (e.g., time):

$$ \begin{align} y_i &\sim \text{Normal}(\mu_i, \sigma) \\ \mu_i &= \alpha + \beta x_i \\ \alpha &\s \text{Normal}(\mu_\alpha, \sigma_\alpha) \\ \beta &\sim \text{Normal}(\mu_\beta, \sigma_\beta) \' &\sim \text{HalfNormal}(\sigma_s) \end{align} $$

Where:

- $\alpha$ is the intercept (baseline price)
- $\beta$ is the slope (trend)
- $\sigma$ is the observation noise (volatility)

## Trading Interpretation

- **$\beta >** 0 **$**: Upward trend (bullish signal)
- **$\beta <** 0 **$**: Downward trend (bearish signal)
- **Wide posterior for $\beta$**: High uncertainty about trend direction
- **Large $\sigma$**: High volatility, wider position sizing needed

In [ ]:
```
# Generate synthetic crude oil price data with trend
np.random.seed(42)
n_days = 120
time = np.arange(n_days)

# True parameters
true_alpha = 65.0  # Starting price
true_beta = 0.15   # Upward trend of $0.15/day
true_sigma = 3.5   # Daily volatility

# Generate prices
crude_prices = true_alpha + true_beta * time + np.random.normal(0, true_s

# Create DataFrame
df_simple = pd.DataFrame({
    'day': time,
    'price': crude_prices
})

# Visualize
fig, ax = plt.subplots(figsize=(12, 5))
ax.plot(df_simple['day'], df_simple['price'], 'o-', alpha=0.6, label='Obs
ax.axhline(true_alpha, color='red', linestyle='--', alpha=0.5, label=f'Tr
ax.plot(time, true_alpha + true_beta * time, 'g--', linewidth=2, label=f'
ax.set_xlabel('Days', fontsize=12)
ax.set_ylabel('Crude Oil Price ($/barrel)', fontsize=12)
ax.set_title('Crude Oil Prices: Simple Linear Trend', fontsize=14, fontwe
ax.legend()
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print(f"Data summary:")
print(df_simple.describe())
```

In [ ]:
```
# Build Bayesian linear regression model
with pm.Model() as model_simple:
    # Data
    x = pm.Data('x', df_simple['day'].values)
    y_obs = pm.Data('y_obs', df_simple['price'].values)

    # Priors
    # Weakly informative prior: we expect crude oil around $60-$80
    alpha = pm.Normal('alpha', mu=70, sigma=20)

    # Prior on trend: we don't expect huge daily changes
    # Could be negative (bear market) or positive (bull market)
    beta = pm.Normal('beta', mu=0, sigma=1)

    # Prior on volatility: crude oil typically has $2-$10 daily volatilit
    sigma = pm.HalfNormal('sigma', sigma=10)

    # Linear model
    mu = alpha + beta * x

    # Likelihood
    y = pm.Normal('y', mu=mu, sigma=sigma, observed=y_obs)

    # Sample from posterior
    trace_simple = pm.sample(2000, tune=1000, return_inferencedata=True,

# Summary
print("\nPosterior Summary:")
print(az.summary(trace_simple, var_names=['alpha', 'beta', 'sigma']))
```

In [ ]:
```
# Visualize posterior distributions
fig, axes = plt.subplots(1, 3, figsize=(15, 4))

# Alpha (intercept)
az.plot_posterior(trace_simple, var_names=['alpha'], ax=axes[0],
                  ref_val=true_alpha, color='steelblue')
axes[0].set_title('Posterior: Intercept (α)', fontsize=12, fontweight='bo
axes[0].axvline(true_alpha, color='red', linestyle='--', linewidth=2, lab

# Beta (slope)
az.plot_posterior(trace_simple, var_names=['beta'], ax=axes[1],
                  ref_val=true_beta, color='darkorange')
axes[1].set_title('Posterior: Slope (β)', fontsize=12, fontweight='bold')
axes[1].axvline(true_beta, color='red', linestyle='--', linewidth=2, labe
axes[1].axvline(0, color='black', linestyle=':', alpha=0.5, label='Zero (

# Sigma (noise)
az.plot_posterior(trace_simple, var_names=['sigma'], ax=axes[2],
                  ref_val=true_sigma, color='forestgreen')
axes[2].set_title('Posterior: Volatility (σ)', fontsize=12, fontweight='b
axes[2].axvline(true_sigma, color='red', linestyle='--', linewidth=2, lab
plt.tight_layout()
plt.show()

# Trading interpretation
beta_samples = trace_simple.posterior['beta'].values.flatten()
prob_uptrend = np.mean(beta_samples > 0)
print(f"\nTrading Signal:")
print(f"Probability of upward trend: {prob_uptrend:.2%}")
print(f"Expected daily price change: ${np.mean(beta_samples):.3f} ± ${np.
if prob_uptrend > 0.75:
    print("→ BULLISH: Strong evidence of uptrend")
elif prob_uptrend < 0.25:
    print("→ BEARISH: Strong evidence of downtrend")
else:
    print("→ NEUTRAL: Insufficient evidence for directional trade")
```

## 2 . Posterior Predictive Checks

Before using the model for trading, we must verify it captures the data-generating process.

In [ ]:
```
# Generate posterior predictive samples
with model_simple:
    ppc_simple = pm.sample_posterior_predictive(trace_simple, random_seed

# Visualize
fig, ax = plt.subplots(figsize=(12, 5))

# Plot 100 posterior predictive samples
ppc_samples = ppc_simple.posterior_predictive['y'].values
for i in range(100):
    chain_idx = np.random.randint(0, ppc_samples.shape[0])
    draw_idx = np.random.randint(0, ppc_samples.shape[1])
    ax.plot(df_simple['day'], ppc_samples[chain_idx, draw_idx, :],
            alpha=0.05, color='blue', linewidth=1)

# Plot observed data
ax.plot(df_simple['day'], df_simple['price'], 'ko', markersize=4, label='

ax.set_xlabel('Days', fontsize=12)
ax.set_ylabel('Price ($/barrel)', fontsize=12)
ax.set_title('Posterior Predictive Check: Model Fit', fontsize=14, fontwe
ax.legend()
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print("If observed data (black dots) falls within the blue cloud, model c
```

## 3 . Credible Intervals vs Confidence Intervals

### Key Difference

**Bayesian Credible Interval (CI)**:

- "There is a $95\%$ probability that the true parameter lies in this interval"
- Direct probability statement about parameter

- What traders actually want to know

**Frequentist Confidence Interval**:

- "If we repeated the experiment many times, 95% of such intervals would contain the tr parameter"
- Statement about the procedure, not the parameter
- Difficult to interpret for one-time decisions

## Trading Application

For position sizing, you want to know: "What's the probability my forecast is within X% of reality? Bayesian credible intervals answer this directly.

In [ ]:
```
# Forecast 30 days ahead with credible intervals
future_days = np.arange(n_days, n_days + 30)

# Update model data for predictions
with model_simple:
    pm.set_data({'x': future_days})
    forecast_simple = pm.sample_posterior_predictive(trace_simple, random

# Extract predictions
forecast_samples = forecast_simple.posterior_predictive['y'].values.resha
forecast_mean = forecast_samples.mean(axis=0)
forecast_lower = np.percentile(forecast_samples, 2.5, axis=0)
forecast_upper = np.percentile(forecast_samples, 97.5, axis=0)

# Visualize forecast
fig, ax = plt.subplots(figsize=(14, 6))

# Historical data
ax.plot(df_simple['day'], df_simple['price'], 'o-', color='black',
        label='Historical Prices', markersize=3, alpha=0.7)

# Forecast
ax.plot(future_days, forecast_mean, 'o-', color='red',
        label='Forecast (Mean)', markersize=4, linewidth=2)
ax.fill_between(future_days, forecast_lower, forecast_upper,
                alpha=0.3, color='red', label='95% Credible Interval')

ax.axvline(n_days - 0.5, color='gray', linestyle='--', linewidth=2, label
ax.set_xlabel('Days', fontsize=12)
ax.set_ylabel('Price ($/barrel)', fontsize=12)
ax.set_title('30-Day Price Forecast with 95% Credible Interval', fontsize
ax.legend(loc='upper left')
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

# Trading interpretation
print(f"\nForecast for Day {n_days + 30}:")
print(f"Expected price: ${forecast_mean[-1]:.2f}")
print(f"95% Credible Interval: [${forecast_lower[-1]:.2f}, ${forecast_upp
print(f"\nInterpretation: There is a 95% probability the price will be be
print(f"Uncertainty range: ${forecast_upper[-1] - forecast_lower[-1]:.2f}
```

## 4 . Multiple Regression: Price ~ Supply + Demand + US Index

### Theory

Multiple regression extends to multiple predictors:

$$ \begin{align} y_i &\sim \text{Normal}(\mu_i, \sigma) \\ \mu_i &= \alpha + \beta_ 1 x_{ 1 i} + \beta_ 2 x_{ 2 i} + \beta_ 3 x_{ 3 i} \\ \alpha &\sim \text{Normal}(\mu_\alpha, \sigma_\alpha \beta_j &\sim \text{Normal}( 0 , \sigma_\beta) \quad \text{for } j = 1 , 2 , 3 \\ \sigma & \text{HalfNormal}(\sigma_s) \end{align} $$

### Trading Application

For crude oil:

- $x_ 1 $: Global supply (million barrels/day) → $\beta_ 1 < 0 $ (more supply, lower pri
- $x_ 2 $: Global demand (million barrels/day) → $\beta_ 2 > 0 $ (more demand, highe
- $x_ 3 $: USD index → $\beta_ 3 < 0 $ (stronger dollar, lower commodity prices)

In [ ]:
```
# Generate synthetic multiple regression data
np.random.seed(42)
n = 150

# Predictors
supply = np.random.normal(100, 5, n)  # Million barrels/day
demand = np.random.normal(98, 4, n)   # Million barrels/day
usd_index = np.random.normal(95, 3, n)  # USD strength index

# True coefficients
true_alpha_multi = 200.0
true_beta_supply = -1.2   # Negative: more supply → lower price
true_beta_demand = 1.8    # Positive: more demand → higher price
true_beta_usd = -0.5      # Negative: stronger USD → lower price
true_sigma_multi = 4.0

# Generate prices
price_multi = (true_alpha_multi +
               true_beta_supply * supply +
               true_beta_demand * demand +
               true_beta_usd * usd_index +
               np.random.normal(0, true_sigma_multi, n))

# Create DataFrame
df_multi = pd.DataFrame({
    'price': price_multi,
    'supply': supply,
    'demand': demand,
    'usd_index': usd_index
})

print("Data Summary:")
print(df_multi.describe())
print(f"\nCorrelation with price:")
print(df_multi.corr()['price'].sort_values(ascending=False))
```

In [ ]:
```
# Standardize predictors for better sampling
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
X_scaled = scaler.fit_transform(df_multi[['supply', 'demand', 'usd_index'

# Build multiple regression model
with pm.Model() as model_multi:
    # Data
    X = pm.Data('X', X_scaled)
    y_obs = pm.Data('y_obs', df_multi['price'].values)

    # Priors
    alpha = pm.Normal('alpha', mu=70, sigma=20)

    # Coefficients: using regularizing priors
    beta_supply = pm.Normal('beta_supply', mu=0, sigma=10)
    beta_demand = pm.Normal('beta_demand', mu=0, sigma=10)
    beta_usd = pm.Normal('beta_usd', mu=0, sigma=10)

    # Stack coefficients
    beta = pm.math.stack([beta_supply, beta_demand, beta_usd])

    sigma = pm.HalfNormal('sigma', sigma=10)

    # Linear model
    mu = alpha + pm.math.dot(X, beta)

    # Likelihood
    y = pm.Normal('y', mu=mu, sigma=sigma, observed=y_obs)

    # Sample
    trace_multi = pm.sample(2000, tune=1000, return_inferencedata=True, r

# Summary
print("\nPosterior Summary:")
print(az.summary(trace_multi, var_names=['alpha', 'beta_supply', 'beta_de
```

In [ ]:
```
# Visualize coefficient posteriors
fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# Supply coefficient
az.plot_posterior(trace_multi, var_names=['beta_supply'], ax=axes[0, 0],
axes[0, 0].set_title('β (Supply): Standardized Effect', fontsize=12, font
axes[0, 0].axvline(0, color='black', linestyle='--', alpha=0.5)

# Demand coefficient
az.plot_posterior(trace_multi, var_names=['beta_demand'], ax=axes[0, 1],
axes[0, 1].set_title('β (Demand): Standardized Effect', fontsize=12, font
axes[0, 1].axvline(0, color='black', linestyle='--', alpha=0.5)

# USD Index coefficient
az.plot_posterior(trace_multi, var_names=['beta_usd'], ax=axes[1, 0], col
axes[1, 0].set_title('β (USD Index): Standardized Effect', fontsize=12, f
axes[1, 0].axvline(0, color='black', linestyle='--', alpha=0.5)
```

```
# Sigma
az.plot_posterior(trace_multi, var_names=['sigma'], ax=axes[1, 1], color=
axes[1, 1].set_title('σ (Residual Volatility)', fontsize=12, fontweight='

plt.tight_layout()
plt.show()

# Trading interpretation
beta_supply_samples = trace_multi.posterior['beta_supply'].values.flatten
beta_demand_samples = trace_multi.posterior['beta_demand'].values.flatten
beta_usd_samples = trace_multi.posterior['beta_usd'].values.flatten()

print("\nTrading Signals:")
print(f"Supply coefficient: {np.mean(beta_supply_samples):.3f} (95% CI: [
print(f"  → Prob(β < 0): {np.mean(beta_supply_samples < 0):.2%} - {'CONFI

print(f"\nDemand coefficient: {np.mean(beta_demand_samples):.3f} (95% CI:
print(f"  → Prob(β > 0): {np.mean(beta_demand_samples > 0):.2%} - {'CONFI

print(f"\nUSD Index coefficient: {np.mean(beta_usd_samples):.3f} (95% CI:
print(f"  → Prob(β < 0): {np.mean(beta_usd_samples < 0):.2%} - {'CONFIRME
```

## 5 . Model Comparison: WAIC and LOO

### Theory

**WAIC (Watanabe-Akaike Information Criterion)**:

- Bayesian generalization of AIC
- Estimates out-of-sample predictive accuracy
- Lower is better
- Accounts for both fit and complexity

**LOO (Leave-One-Out Cross-Validation)**:

- Estimates predictive accuracy by leaving each observation out
- Approximated efficiently using Pareto-smoothed importance sampling
- More robust than WAIC for small samples

### Trading Application

Compare simple trend model vs multiple regression model to decide which to use for trading.

In [ ]:
```
# Build comparable simple model on same data
with pm.Model() as model_simple_multi:
    # Use only time as predictor
    x = pm.Data('x', np.arange(len(df_multi)))
    y_obs = pm.Data('y_obs', df_multi['price'].values)

    alpha = pm.Normal('alpha', mu=70, sigma=20)
    beta = pm.Normal('beta', mu=0, sigma=1)
    sigma = pm.HalfNormal('sigma', sigma=10)

    mu = alpha + beta * x
    y = pm.Normal('y', mu=mu, sigma=sigma, observed=y_obs)
    trace_simple_multi = pm.sample(2000, tune=1000, return_inferencedata=

# Compute model comparison metrics
comparison = az.compare({
    'Simple (Time Only)': trace_simple_multi,
    'Multiple Regression': trace_multi
})

print("\nModel Comparison (LOO):")
print(comparison)
print("\nInterpretation:")
print("- 'elpd_loo': Expected log pointwise predictive density (higher is
print("- 'p_loo': Effective number of parameters")
print("- 'loo': LOO information criterion (lower is better)")
print("- 'se': Standard error of the difference")
print("- 'dse': Standard error of the LOO difference")
print("- 'weight': Pseudo-BMA weight (model probability)")
print(f"\nBest model: {comparison.index[0]}")
print(f"Weight: {comparison.loc[comparison.index[0], 'weight']:.2%}")
```

In [ ]:
```
# Visualize model comparison
az.plot_compare(comparison, insample_dev=False)
plt.title('Model Comparison: LOO Cross-Validation', fontsize=14, fontweig
plt.tight_layout()
plt.show()

print("\nTrading Decision:")
if comparison.loc['Multiple Regression', 'weight'] > 0.75:
    print("→ USE MULTIPLE REGRESSION: Strong evidence it outperforms simp
    print("  Action: Incorporate supply, demand, and USD data into tradin
elif comparison.loc['Simple (Time Only)', 'weight'] > 0.75:
    print("→ USE SIMPLE TREND: Sufficient for prediction, avoid overfitti
    print("  Action: Stick with simple time-based model")
else:
    print("→ MODEL AVERAGING: Combine predictions from both models")
    print("  Action: Weight predictions by model probabilities")
```

## 6 . Robust Regression with Student-t Likelihood

### Theory

Normal likelihood assumes Gaussian noise, which underestimates tail risk. Student-t likelihood h heavier tails:

$$ \begin{align} y_i &\sim \text{StudentT}(\nu, \mu_i, \sigma) \\ \mu_i &= \alpha + \beta x_i \\ \nu \text{Gamma}( 2 , 0 . 1 ) \end{align} $$

Where $\nu$ is the degrees of freedom:

- $\nu = 1 $: Cauchy distribution (very heavy tails)
- $\nu = 3 0 +$: Approximately Normal
- $\nu \in [ 3 , 1 0 ]$: Typical for commodity data

## Trading Application

Commodity markets experience:

- Supply shocks (hurricanes, OPEC cuts)
- Demand spikes (cold snaps, geopolitical events)
- "Flash crashes" and fat-tail events

Student-t likelihood prevents a few extreme days from distorting the entire model.

In [ ]:
```
# Generate data with outliers (simulating supply shocks)
np.random.seed(42)
n_robust = 100
time_robust = np.arange(n_robust)

# Base trend
price_robust = 60 + 0.1 * time_robust + np.random.normal(0, 2, n_robust)

# Add 5 extreme shocks (fat tails)
shock_days = [20, 35, 50, 68, 85]
for day in shock_days:
    price_robust[day] += np.random.choice([-15, -12, 12, 15])  # Large pr

df_robust = pd.DataFrame({
    'day': time_robust,
    'price': price_robust
})

# Visualize
fig, ax = plt.subplots(figsize=(12, 5))
ax.plot(df_robust['day'], df_robust['price'], 'o-', alpha=0.7)
ax.scatter([shock_days], df_robust.loc[shock_days, 'price'],
           color='red', s=100, zorder=5, label='Supply Shocks', edgecolor
ax.set_xlabel('Days', fontsize=12)
ax.set_ylabel('Price ($/barrel)', fontsize=12)
ax.set_title('Crude Oil Prices with Supply Shocks (Fat Tails)', fontsize=
ax.legend()
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```

In [ ]:
```
# Normal likelihood model (non-robust)
with pm.Model() as model_normal:
    x = pm.Data('x', df_robust['day'].values)
    y_obs = pm.Data('y_obs', df_robust['price'].values)

    alpha = pm.Normal('alpha', mu=60, sigma=10)
    beta = pm.Normal('beta', mu=0, sigma=1)
    sigma = pm.HalfNormal('sigma', sigma=10)

    mu = alpha + beta * x
    y = pm.Normal('y', mu=mu, sigma=sigma, observed=y_obs)

    trace_normal = pm.sample(2000, tune=1000, return_inferencedata=True,

# Student-t likelihood model (robust)
```

```
with pm.Model() as model_robust:
    x = pm.Data('x', df_robust['day'].values)
    y_obs = pm.Data('y_obs', df_robust['price'].values)

    alpha = pm.Normal('alpha', mu=60, sigma=10)
    beta = pm.Normal('beta', mu=0, sigma=1)
    sigma = pm.HalfNormal('sigma', sigma=10)

    # Degrees of freedom: lower = heavier tails
    nu = pm.Gamma('nu', alpha=2, beta=0.1)

    mu = alpha + beta * x
    y = pm.StudentT('y', nu=nu, mu=mu, sigma=sigma, observed=y_obs)

    trace_robust = pm.sample(2000, tune=1000, return_inferencedata=True,

print("\nNormal Likelihood Model:")
print(az.summary(trace_normal, var_names=['alpha', 'beta', 'sigma']))

print("\nStudent-t Likelihood Model:")
print(az.summary(trace_robust, var_names=['alpha', 'beta', 'sigma', 'nu']
```

In [ ]:

```
# Compare fits
fig, axes = plt.subplots(1, 2, figsize=(16, 5))

# Normal model
alpha_normal = trace_normal.posterior['alpha'].values.flatten().mean()
beta_normal = trace_normal.posterior['beta'].values.flatten().mean()
axes[0].plot(df_robust['day'], df_robust['price'], 'o', alpha=0.5, label=
axes[0].scatter([shock_days], df_robust.loc[shock_days, 'price'],
                color='red', s=100, zorder=5, edgecolors='black', linewid
axes[0].plot(df_robust['day'], alpha_normal + beta_normal * df_robust['da
             'b-', linewidth=2, label='Normal Fit')
axes[0].set_title('Normal Likelihood (Non-Robust)', fontsize=12, fontweig
axes[0].set_xlabel('Days')
axes[0].set_ylabel('Price ($/barrel)')
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Robust model
alpha_robust = trace_robust.posterior['alpha'].values.flatten().mean()
beta_robust = trace_robust.posterior['beta'].values.flatten().mean()
axes[1].plot(df_robust['day'], df_robust['price'], 'o', alpha=0.5, label=
axes[1].scatter([shock_days], df_robust.loc[shock_days, 'price'],
                color='red', s=100, zorder=5, label='Supply Shocks', edge
axes[1].plot(df_robust['day'], alpha_robust + beta_robust * df_robust['da
             'g-', linewidth=2, label='Student-t Fit')
axes[1].set_title('Student-t Likelihood (Robust)', fontsize=12, fontweigh
axes[1].set_xlabel('Days')
axes[1].set_ylabel('Price ($/barrel)')
axes[1].legend()
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("\nComparison:")
print(f"Normal model slope: {beta_normal:.4f}")
print(f"Student-t model slope: {beta_robust:.4f}")
print(f"\nThe Student-t model is less influenced by extreme shocks.")
print(f"Estimated degrees of freedom: {trace_robust.posterior['nu'].value
print(f"(Lower ν = heavier tails; ν ≈ 5-10 is typical for commodities)")
```

In [ ]:
```
# Model comparison
comparison_robust = az.compare({
    'Normal Likelihood': trace_normal,
    'Student-t Likelihood': trace_robust
})

print("\nModel Comparison:")
print(comparison_robust)
print(f"\nBest model: {comparison_robust.index[0]}")
print(f"\nTrading Recommendation: Use Student-t likelihood for commodity
```

## 7 . Practical Example: Natural Gas Prices with Weathe Data

Natural gas prices are highly sensitive to weather:

- **Winter (heating demand)**: Cold temperatures → Higher prices
- **Summer (cooling demand)**: Hot temperatures → Higher prices
- **Storage levels**: Lower inventory → Higher prices

We'll build a Bayesian regression model incorporating these factors.

In [ ]:
```
# Generate synthetic natural gas data
np.random.seed(42)
n_gas = 365  # 1 year of daily data

# Day of year (for seasonality)
day_of_year = np.arange(1, n_gas + 1)

# Temperature (Fahrenheit): cold in winter, hot in summer
# Sinusoidal pattern
temperature = 50 + 30 * np.sin(2 * np.pi * (day_of_year - 80) / 365) + np

# Heating Degree Days (HDD): higher when cold
hdd = np.maximum(65 - temperature, 0)

# Cooling Degree Days (CDD): higher when hot
cdd = np.maximum(temperature - 65, 0)

# Storage levels (billion cubic feet): starts high, depletes in winter
storage = 3500 - 1000 * np.sin(2 * np.pi * (day_of_year - 80) / 365) + np

# Price model:
# Base price + HDD effect (heating) + CDD effect (cooling) - storage effe
true_base = 3.0
true_hdd_coef = 0.05    # Higher heating demand → Higher price
true_cdd_coef = 0.03    # Higher cooling demand → Higher price
true_storage_coef = -0.0008  # More storage → Lower price

natgas_price = (true_base +
                true_hdd_coef * hdd +
                true_cdd_coef * cdd +
                true_storage_coef * storage +
                np.random.normal(0, 0.3, n_gas))
```

```
# Ensure prices are positive
natgas_price = np.maximum(natgas_price, 1.5)

df_natgas = pd.DataFrame({
    'day': day_of_year,
    'temperature': temperature,
    'hdd': hdd,
    'cdd': cdd,
    'storage': storage,
    'price': natgas_price
})

print("Natural Gas Data Summary:")
print(df_natgas.describe())
```

In [ ]:

```
# Visualize data
fig, axes = plt.subplots(3, 1, figsize=(14, 10), sharex=True)

# Price
axes[0].plot(df_natgas['day'], df_natgas['price'], linewidth=1.5, color='
axes[0].set_ylabel('Price ($/MMBtu)', fontsize=11)
axes[0].set_title('Natural Gas Prices (Daily)', fontsize=12, fontweight='
axes[0].grid(True, alpha=0.3)

# Temperature & Degree Days
ax2 = axes[1]
ax2.plot(df_natgas['day'], df_natgas['temperature'], label='Temperature (
ax2.axhline(65, color='black', linestyle='--', alpha=0.5, label='Base Tem
ax2.set_ylabel('Temperature (°F)', fontsize=11)
ax2.legend(loc='upper left')
ax2.grid(True, alpha=0.3)

ax2b = ax2.twinx()
ax2b.fill_between(df_natgas['day'], 0, df_natgas['hdd'], alpha=0.3, color
ax2b.fill_between(df_natgas['day'], 0, df_natgas['cdd'], alpha=0.3, color
ax2b.set_ylabel('Degree Days', fontsize=11)
ax2b.legend(loc='upper right')

# Storage
axes[2].plot(df_natgas['day'], df_natgas['storage'], linewidth=1.5, color
axes[2].set_xlabel('Day of Year', fontsize=11)
axes[2].set_ylabel('Storage (Bcf)', fontsize=11)
axes[2].set_title('Natural Gas Storage Levels', fontsize=12, fontweight='
axes[2].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

In [ ]:

```
# Standardize predictors
scaler_natgas = StandardScaler()
X_natgas_scaled = scaler_natgas.fit_transform(df_natgas[['hdd', 'cdd', 's

# Build Bayesian regression model
with pm.Model() as model_natgas:
    # Data
    X = pm.Data('X', X_natgas_scaled)
    y_obs = pm.Data('y_obs', df_natgas['price'].values)
    # Priors
    alpha = pm.Normal('alpha', mu=3, sigma=2)  # Base price around $3/MMB

    # Coefficients
    beta_hdd = pm.Normal('beta_hdd', mu=0, sigma=1)  # Heating effect
    beta_cdd = pm.Normal('beta_cdd', mu=0, sigma=1)  # Cooling effect
    beta_storage = pm.Normal('beta_storage', mu=0, sigma=1)  # Storage ef

    beta = pm.math.stack([beta_hdd, beta_cdd, beta_storage])

    # Use Student-t for robustness
    nu = pm.Gamma('nu', alpha=2, beta=0.1)
    sigma = pm.HalfNormal('sigma', sigma=2)

    # Linear model
    mu = alpha + pm.math.dot(X, beta)

    # Likelihood
    y = pm.StudentT('y', nu=nu, mu=mu, sigma=sigma, observed=y_obs)

    # Sample
    trace_natgas = pm.sample(2000, tune=1000, return_inferencedata=True,

print("\nPosterior Summary:")
print(az.summary(trace_natgas, var_names=['alpha', 'beta_hdd', 'beta_cdd'
```

In [ ]:

```
# Visualize posteriors
fig, axes = plt.subplots(2, 3, figsize=(16, 8))

az.plot_posterior(trace_natgas, var_names=['alpha'], ax=axes[0, 0], color
axes[0, 0].set_title('Base Price (α)', fontsize=11, fontweight='bold')

az.plot_posterior(trace_natgas, var_names=['beta_hdd'], ax=axes[0, 1], co
axes[0, 1].set_title('HDD Effect (β_hdd)', fontsize=11, fontweight='bold'
axes[0, 1].axvline(0, color='black', linestyle='--', alpha=0.5)

az.plot_posterior(trace_natgas, var_names=['beta_cdd'], ax=axes[0, 2], co
axes[0, 2].set_title('CDD Effect (β_cdd)', fontsize=11, fontweight='bold'
axes[0, 2].axvline(0, color='black', linestyle='--', alpha=0.5)

az.plot_posterior(trace_natgas, var_names=['beta_storage'], ax=axes[1, 0]
axes[1, 0].set_title('Storage Effect (β_storage)', fontsize=11, fontweigh
axes[1, 0].axvline(0, color='black', linestyle='--', alpha=0.5)

az.plot_posterior(trace_natgas, var_names=['sigma'], ax=axes[1, 1], color
axes[1, 1].set_title('Volatility (σ)', fontsize=11, fontweight='bold')

az.plot_posterior(trace_natgas, var_names=['nu'], ax=axes[1, 2], color='p
axes[1, 2].set_title('Degrees of Freedom (ν)', fontsize=11, fontweight='b

plt.tight_layout()
plt.show()

# Trading interpretation
beta_hdd_samples = trace_natgas.posterior['beta_hdd'].values.flatten()
beta_cdd_samples = trace_natgas.posterior['beta_cdd'].values.flatten()
beta_storage_samples = trace_natgas.posterior['beta_storage'].values.flat

print("\n=== TRADING SIGNALS ===")
print(f"\nHeating Demand (HDD):")
print(f"  Effect: {np.mean(beta_hdd_samples):.4f} (95% CI: [{np.percentil
print(f"  Prob(β > 0): {np.mean(beta_hdd_samples > 0):.2%}")
if np.mean(beta_hdd_samples > 0) > 0.95:
    print("  → CONFIRMED: Cold weather increases prices (go long ahead of

print(f"\nCooling Demand (CDD):")
print(f"  Effect: {np.mean(beta_cdd_samples):.4f} (95% CI: [{np.percentil
print(f"  Prob(β > 0): {np.mean(beta_cdd_samples > 0):.2%}")
if np.mean(beta_cdd_samples > 0) > 0.95:
    print("  → CONFIRMED: Hot weather increases prices (go long ahead of

print(f"\nStorage Levels:")
print(f"  Effect: {np.mean(beta_storage_samples):.4f} (95% CI: [{np.perce
print(f"  Prob(β < 0): {np.mean(beta_storage_samples < 0):.2%}")
if np.mean(beta_storage_samples < 0) > 0.95:
    print("  → CONFIRMED: Low storage increases prices (monitor weekly EI
```

In [ ]:

```
# In-sample fit
with model_natgas:
    ppc_natgas = pm.sample_posterior_predictive(trace_natgas, random_seed

ppc_mean = ppc_natgas.posterior_predictive['y'].mean(dim=['chain', 'draw'

fig, axes = plt.subplots(2, 1, figsize=(14, 8), sharex=True)

# Actual vs Fitted
axes[0].plot(df_natgas['day'], df_natgas['price'], 'o', alpha=0.5, label=
axes[0].plot(df_natgas['day'], ppc_mean, linewidth=2, color='red', label=
axes[0].set_ylabel('Price ($/MMBtu)', fontsize=11)
axes[0].set_title('Natural Gas: Actual vs Fitted Prices', fontsize=12, fo
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Residuals
residuals = df_natgas['price'].values - ppc_mean
axes[1].plot(df_natgas['day'], residuals, 'o', alpha=0.6, markersize=3)
axes[1].axhline(0, color='black', linestyle='--', linewidth=1.5)
axes[1].fill_between(df_natgas['day'], -2*residuals.std(), 2*residuals.st
                     alpha=0.2, color='gray', label='±2σ')
axes[1].set_xlabel('Day of Year', fontsize=11)
axes[1].set_ylabel('Residuals', fontsize=11)
axes[1].set_title('Residuals (Actual - Fitted)', fontsize=12, fontweight=
axes[1].legend()
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"\nModel Performance:")
print(f"RMSE: ${np.sqrt(np.mean(residuals**2)):.4f}/MMBtu")
print(f"MAE: ${np.mean(np.abs(residuals)):.4f}/MMBtu")
print(f"R²: {1 - np.var(residuals) / np.var(df_natgas['price']):.4f}")
```

## Knowledge Check Quiz

Test your understanding with these questions:

## Question 1

What is the key difference between a Bayesian credible interval and a frequentist confidence int

A) Credible intervals are always wider
B) Credible intervals allow direct probability statements about parameters
C) Confidence intervals are more accurate
D) They are mathematically equivalent

**Answer: B** - Credible intervals provide the probability that the parameter lies in the interval, give data. Confidence intervals describe the long-run frequency of interval coverage.

---

## Question 2

When should you use Student-t likelihood instead of Normal likelihood?

A) When you have large sample sizes
B) When data has heavy tails or outliers
C) When you want faster computation
D) Never, Normal is always better

**Answer: B** - Student-t likelihood is robust to outliers and heavy tails, common in commodity mar supply shocks.

---

## Question 3

What does WAIC measure?

A) Model training accuracy
B) Out-of-sample predictive accuracy
C) Number of parameters
D) Computational efficiency

**Answer: B** - WAIC estimates how well the model predicts new, unseen data.

---

## Question 4

If the 95% credible interval for β (slope) is [-0.5, 0.3], what trading signal does this

A) Strong bullish (go long)
B) Strong bearish (go short)
C) Neutral (insufficient evidence for direction)
D) Extremely volatile

**Answer: C** - The interval contains zero, indicating we cannot confidently determine if the trend is or negative.

---

### Question 5

In the natural gas example, why did we standardize predictors before modeling?

A) Required by PyMC
B) Improves sampling efficiency and prior specification
C) Makes coefficients exactly equal
D) Reduces data size

**Answer: B** - Standardization puts all predictors on the same scale, improving MCMC sampling a
allowing comparable priors on coefficients.

---

## Exercises

### Exercise 1: Gold Price Regression

Build a Bayesian linear regression model for gold prices using:

- USD index (negative relationship expected)
- Real interest rates (negative relationship expected)
- VIX (volatility index, positive relationship expected)

Compare Normal vs Student-t likelihood. Which performs better?

### Exercise 2: Model Comparison

For the crude oil multiple regression example:

1. Build a model with only supply
2. Build a model with only demand
3. Build a model with only USD index
4. Compare all models using WAIC
5. Which single predictor is most important?

### Exercise 3: Forecast Uncertainty

Using the natural gas model:

1. Generate forecasts for the next 30 days
2. Assume a cold snap increases HDD by 50%
3. Quantify the impact on price forecasts
4. Calculate the probability that prices exceed $5/MMBtu

### Exercise 4: Prior Sensitivity

For the simple linear regression:

1. Try a strongly informative prior: β ~ Normal(0.2, 0.05)

2. Try an uninformative prior: β ~ Normal(0, 100)
3. Compare posteriors and predictions
4. When does the prior matter most?

### Exercise 5: Trading Strategy

Design a trading strategy for natural gas:

1. Use the model to forecast next week's average price
2. If P(price > current price) > 0.75, go long
3. If P(price < current price) > 0.75, go short
4. Otherwise, stay flat
5. Backtest on synthetic data

---

## Summary

In this module, you learned:

1. **Simple Bayesian Linear Regression**: Model price trends with full uncertainty quantificatior
2. **Posterior Interpretation**: Extract trading signals from posterior distributions
3. **Credible Intervals**: Direct probability statements for risk management
4. **Multiple Regression**: Incorporate fundamental drivers (supply, demand, FX)
5. **Model Comparison**: Use WAIC/LOO to select the best model
6. **Robust Regression**: Handle outliers and fat tails with Student-t likelihood
7. **Natural Gas Application**: Real-world example with weather and storage data

### Key Takeaways for Trading

- Bayesian regression provides **actionable uncertainty** for position sizing
- **Student-t likelihood** is essential for commodity data with extreme events
- **Model comparison** prevents overfitting and improves out-of-sample performance
- **Incorporate fundamentals** (supply, demand, weather) for better forecasts
- **Credible intervals** directly answer "What's the probability of X?"

---

## Preview of Next Module

### Module 6: Bayesian Structural Time Series (BSTS)

Linear regression assumes constant relationships. But commodity markets have:

- **Trends** that change over time
- **Seasonality** (winter heating, summer cooling)
- **Dynamic relationships** (correlations that evolve)

In Module 6, we'll build Bayesian Structural Time Series models that decompose prices into:

- Local level (random walk)

- Local linear trend (changing slopes)
- Seasonal components (daily, weekly, annual)
- Dynamic regression coefficients

You'll learn to model **time-varying volatility** and **structural breaks** crucial for commodity tradin

**See you in Module** `6` **!**

# Module 6 : Bayesian Structural Time Series (BSTS)

## Learning Objectives

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

1. Understand state-space models and their components
2. Implement local level and local linear trend models
3. Add seasonal components (daily, weekly, annual) to time series
4. Build regression components with external predictors
5. Model time-varying coefficients (dynamic regression)
6. Apply BSTS to gold prices with USD and inflation data
7. Decompose commodity prices into trend, seasonality, and noise

## Why This Matters for Trading

Static linear regression assumes relationships don't change. But commodity markets are dynami

- **Structural Breaks**: OPEC policy changes, new technology (fracking), regime shifts
- **Time-Varying Relationships**: Oil-USD correlation weakens during crises
- **Complex Seasonality**: Natural gas has annual (winter heating), weekly (storage reports), a patterns
- **Trend Changes**: Mean-reversion vs momentum regimes alternate

Bayesian Structural Time Series (BSTS) models decompose prices into interpretable componen

- **Trend**: Long-term direction (is gold in a secular bull market?)
- **Seasonality**: Predictable cycles (harvest pressure on agricultural commodities)
- **Regression**: Impact of fundamentals (how much does USD drive gold today?)
- **Noise**: Unpredictable shocks

This decomposition enables:

- **Better forecasts**: Separate signal from noise
- **Regime detection**: Identify when relationships break down
- **Event impact**: Measure how geopolitical shocks affect trends
- **Seasonality trading**: Exploit recurring patterns with confidence

In short, BSTS gives you X-ray vision into market structure.

In [ ]:
```python
# Standard imports
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')

np.random.seed(42)
plt.style.use('seaborn-v0_8-whitegrid')

print(f"PyMC version: {pm.__version__}")
print(f"ArviZ version: {az.__version__}")
```

## 1 . Introduction to State-Space Models

### Theory

A **state-space model** separates observed data from hidden "state" variables:

**Observation Equation** (what we see): $$ y_t = Z_t \alpha_t + \epsilon_t, \quad \epsilon_t \sim \text{Normal}( 0 , \sigma_y^ 2 ) $$

**State Equation** (hidden dynamics): $$ \alpha_t = T_t \alpha_{t- 1 } + R_t \eta_t, \quad \eta_t \sir
\text{Normal}( 0 , \sigma_\alpha^ 2 ) $$

Where:

- $y_t$: Observed price at time $t$
- $\alpha_t$: Hidden state (e.g., "true" price level)
- $\epsilon_t$: Observation noise (measurement error, bid-ask bounce)
- $\eta_t$: State innovation (real price changes)

### Trading Interpretation

- **State $\alpha_t$**: "Fundamental" or "fair" value
- **Observation noise $\epsilon_t$**: Market microstructure, liquidity shocks
- **State innovation $\eta_t$**: New information arrival

**Trading strategy**: When $y_t$ deviates from $\alpha_t$ (state estimate), mean-revert.

## 2 . Local Level Model (Random Walk + Noise)

### Theory

The simplest state-space model:

$$ \begin{align} y_t &= \mu_t + \epsilon_t, \quad \epsilon_t \sim \text{Normal}( 0 , \sigma_y^ 2 ) &= \mu_{t- 1 } + \eta_t, \quad \eta_t \sim \text{Normal}( 0 , \sigma_\mu^ 2 ) \end{align} $$

- $\mu_t$: Local level (latent "true price")
- Evolves as a random walk
- $\sigma_\mu^ 2 / \sigma_y^ 2 $ ratio determines smoothness

## Signal-to-Noise Ratio

- High $\sigma_\mu^ 2 $: Level changes rapidly (trending)
- Low $\sigma_\mu^ 2 $: Level nearly constant (mean-reverting)
- $\sigma_\mu^ 2 = 0 $: Reduces to $y_t = \mu + \epsilon_t$ (white noise around mear

```
In [ ]: # Generate synthetic data: local level model
np.random.seed(42)
T = 200  # 200 days

# True parameters
true_sigma_mu = 0.5   # State innovation (trend changes)
true_sigma_y = 1.5    # Observation noise
true_mu0 = 50.0       # Initial level

# Generate latent level (random walk)
mu_true = np.zeros(T)
mu_true[0] = true_mu0
for t in range(1, T):
    mu_true[t] = mu_true[t-1] + np.random.normal(0, true_sigma_mu)

# Generate observations
y_obs = mu_true + np.random.normal(0, true_sigma_y, T)

# Visualize
fig, ax = plt.subplots(figsize=(14, 5))
ax.plot(y_obs, 'o-', alpha=0.5, markersize=3, label='Observed Price', col
ax.plot(mu_true, linewidth=2.5, label='True Latent Level (μ)', color='red
ax.fill_between(range(T), mu_true - 2*true_sigma_y, mu_true + 2*true_sigm
                alpha=0.2, color='red', label='±2σ_y (Observation Noise)'
ax.set_xlabel('Time (days)', fontsize=12)
ax.set_ylabel('Price', fontsize=12)
ax.set_title('Local Level Model: Latent State vs Observations', fontsize=
ax.legend(loc='upper left')
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print(f"Signal-to-Noise Ratio: σ_μ² / σ_y² = {true_sigma_mu**2 / true_sig
print(f"Lower ratio → smoother level (mean-reversion)")
print(f"Higher ratio → rapidly changing level (trending)")
```

```
In [ ]: # Build local level model in PyMC
with pm.Model() as local_level_model:
    # Priors on volatilities
    sigma_mu = pm.HalfNormal('sigma_mu', sigma=2)  # State innovation
    sigma_y = pm.HalfNormal('sigma_y', sigma=5)    # Observation noise

    # Initial state
    mu_init = pm.Normal('mu_init', mu=50, sigma=10)

    # State innovations
    innovations = pm.Normal('innovations', mu=0, sigma=sigma_mu, shape=T-

    # Build level via cumulative sum
    mu = pm.Deterministic('mu', pm.math.concatenate([[mu_init], mu_init +
    # Observations
    y = pm.Normal('y', mu=mu, sigma=sigma_y, observed=y_obs)

    # Sample
    trace_local_level = pm.sample(2000, tune=1000, return_inferencedata=T
                                  target_accept=0.95, random_seed=42)

print("\nPosterior Summary:")
print(az.summary(trace_local_level, var_names=['sigma_mu', 'sigma_y']))
```

In [ ]:

```
# Extract and visualize latent level estimates
mu_posterior = trace_local_level.posterior['mu'].values
mu_mean = mu_posterior.mean(axis=(0, 1))
mu_lower = np.percentile(mu_posterior, 2.5, axis=(0, 1))
mu_upper = np.percentile(mu_posterior, 97.5, axis=(0, 1))

fig, ax = plt.subplots(figsize=(14, 6))
ax.plot(y_obs, 'o', alpha=0.4, markersize=3, label='Observed Price', colo
ax.plot(mu_true, linewidth=2, label='True Level', color='red', linestyle=
ax.plot(mu_mean, linewidth=2.5, label='Estimated Level (Posterior Mean)',
ax.fill_between(range(T), mu_lower, mu_upper, alpha=0.3, color='blue',
                label='95% Credible Interval')
ax.set_xlabel('Time (days)', fontsize=12)
ax.set_ylabel('Price', fontsize=12)
ax.set_title('Local Level Model: Filtering Noise from Signal', fontsize=1
ax.legend(loc='upper left')
ax.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print("\nTrading Interpretation:")
print("Blue line = 'Fair value' after filtering out noise")
print("When observed price (gray) deviates from blue line → mean-reversio
print(f"\nEstimated σ_μ: {trace_local_level.posterior['sigma_mu'].values.
print(f"Estimated σ_y: {trace_local_level.posterior['sigma_y'].values.mea
```

## 3 . Local Linear Trend Model

### Theory

The local level model has no persistent trend. Add a **slope** component:

$$ \begin{align} y_t &= \mu_t + \epsilon_t, \quad \epsilon_t \sim \text{Normal}( 0 , \sigma_y^ 2 ) &= \mu_{t- 1 } + \beta_{t- 1 } + \eta_t, \quad \eta_t \sim \text{Normal}( 0 , \sigma_\mu^ 2 ) \\ \bet \beta_{t- 1 } + \zeta_t, \quad \zeta_t \sim \text{Normal}( 0 , \sigma_\beta^ 2 ) \end{align} $$

Where:

- $\mu_t$: Level (intercept)
- $\beta_t$: Slope (trend)
- Both evolve as random walks

### Trading Interpretation

- $\beta_t > 0 $: Uptrend (bullish)

- $\beta_t < 0$: Downtrend (bearish)
- $\beta_t$ changing sign: Trend reversal
- $\sigma_\beta^2$ large: Frequently changing trends (regime switches)

In [ ]:
```
# Generate data with time-varying trend
np.random.seed(42)
T_trend = 150

# True parameters
true_sigma_mu_trend = 0.3
true_sigma_beta = 0.05  # Slope changes slowly
true_sigma_y_trend = 1.2

# Initialize
mu_trend_true = np.zeros(T_trend)
beta_trend_true = np.zeros(T_trend)
mu_trend_true[0] = 60.0
beta_trend_true[0] = 0.2  # Initial uptrend

# Evolve states
for t in range(1, T_trend):
    beta_trend_true[t] = beta_trend_true[t-1] + np.random.normal(0, true_
    mu_trend_true[t] = mu_trend_true[t-1] + beta_trend_true[t-1] + np.ran

# Generate observations
y_trend_obs = mu_trend_true + np.random.normal(0, true_sigma_y_trend, T_t

# Visualize
fig, axes = plt.subplots(2, 1, figsize=(14, 8), sharex=True)

# Observed vs Level
axes[0].plot(y_trend_obs, 'o-', alpha=0.5, markersize=3, label='Observed
axes[0].plot(mu_trend_true, linewidth=2.5, label='True Level (μ)', color=
axes[0].set_ylabel('Price', fontsize=12)
axes[0].set_title('Local Linear Trend: Level Component', fontsize=12, fon
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Slope over time
axes[1].plot(beta_trend_true, linewidth=2, color='red')
axes[1].axhline(0, color='black', linestyle='--', alpha=0.7, linewidth=1.
axes[1].fill_between(range(T_trend), 0, beta_trend_true, where=(beta_tren
                     alpha=0.3, color='green', label='Uptrend')
axes[1].fill_between(range(T_trend), 0, beta_trend_true, where=(beta_tren
                     alpha=0.3, color='red', label='Downtrend')
axes[1].set_xlabel('Time (days)', fontsize=12)
axes[1].set_ylabel('Slope (β)', fontsize=12)
axes[1].set_title('Time-Varying Trend (Slope Component)', fontsize=12, fo
axes[1].legend()
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

In [ ]:
```
# Build local linear trend model
with pm.Model() as llt_model:
    # Priors
    sigma_mu = pm.HalfNormal('sigma_mu', sigma=2)
    sigma_beta = pm.HalfNormal('sigma_beta', sigma=0.5)
    sigma_y = pm.HalfNormal('sigma_y', sigma=5)

    # Initial states
    mu_init = pm.Normal('mu_init', mu=60, sigma=10)
    beta_init = pm.Normal('beta_init', mu=0, sigma=1)

    # Innovations
    innovations_mu = pm.Normal('innovations_mu', mu=0, sigma=sigma_mu, sh
    innovations_beta = pm.Normal('innovations_beta', mu=0, sigma=sigma_be

    # Build slope (random walk)
    beta = pm.Deterministic('beta', pm.math.concatenate(
        [[beta_init], beta_init + pm.math.cumsum(innovations_beta)]
    ))

    # Build level (random walk with drift = slope)
    # μ_t = μ_{t-1} + β_{t-1} + η_t
    # We need to construct this recursively
    # Using scan for sequential dependency

    def level_step(beta_lag, mu_lag, innovation):
        return mu_lag + beta_lag + innovation

    mu_rest, _ = pm.scan(
        fn=level_step,
        sequences=[beta[:-1], innovations_mu],
        outputs_info=[mu_init]
    )

    mu = pm.Deterministic('mu', pm.math.concatenate([[mu_init], mu_rest])

    # Observations
    y = pm.Normal('y', mu=mu, sigma=sigma_y, observed=y_trend_obs)

    # Sample
    trace_llt = pm.sample(2000, tune=1000, return_inferencedata=True,
                          target_accept=0.95, random_seed=42)

print("\nPosterior Summary:")
print(az.summary(trace_llt, var_names=['sigma_mu', 'sigma_beta', 'sigma_y
```

In [ ]:
```
# Visualize estimates
mu_llt_mean = trace_llt.posterior['mu'].mean(dim=['chain', 'draw']).value
beta_llt_mean = trace_llt.posterior['beta'].mean(dim=['chain', 'draw']).v
beta_llt_lower = np.percentile(trace_llt.posterior['beta'].values, 2.5, a
beta_llt_upper = np.percentile(trace_llt.posterior['beta'].values, 97.5,

fig, axes = plt.subplots(2, 1, figsize=(14, 8), sharex=True)

# Level
axes[0].plot(y_trend_obs, 'o', alpha=0.3, markersize=3, label='Observed',
axes[0].plot(mu_trend_true, linewidth=2, label='True Level', color='red',
axes[0].plot(mu_llt_mean, linewidth=2.5, label='Estimated Level', color='
axes[0].set_ylabel('Price', fontsize=12)
axes[0].set_title('Estimated Level (μ)', fontsize=12, fontweight='bold')
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Slope
```

```
axes[1].plot(beta_trend_true, linewidth=2, label='True Slope', color='red
axes[1].plot(beta_llt_mean, linewidth=2.5, label='Estimated Slope', color
axes[1].fill_between(range(T_trend), beta_llt_lower, beta_llt_upper,
                     alpha=0.3, color='green', label='95% CI')
axes[1].axhline(0, color='black', linestyle='--', alpha=0.7)
axes[1].set_xlabel('Time (days)', fontsize=12)
axes[1].set_ylabel('Slope (β)', fontsize=12)
axes[1].set_title('Estimated Slope (Trend)', fontsize=12, fontweight='bol
axes[1].legend()
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Trading signals
current_slope = beta_llt_mean[-1]
prob_uptrend = np.mean(trace_llt.posterior['beta'].values[:, :, -1] > 0)

print(f"\nCurrent Slope Estimate: {current_slope:.4f}")
print(f"Probability of Uptrend: {prob_uptrend:.2%}")
if prob_uptrend > 0.75:
    print("→ BULLISH: Strong uptrend detected")
elif prob_uptrend < 0.25:
    print("→ BEARISH: Strong downtrend detected")
else:
    print("→ NEUTRAL: Trend direction uncertain")
```

## 4 . Seasonal Components

### Theory

Commodities exhibit strong seasonality:

- **Natural gas**: Winter heating demand
- **Agriculture**: Planting and harvest cycles
- **Power**: Summer cooling load

Add a seasonal component with period $S$:

$$ \begin{align} y_t &= \mu_t + \gamma_t + \epsilon_t \\ \gamma_t &= -\sum_{s= 1 }^{S- 1 } \gai
s} + \omega_t, \quad \omega_t \sim \text{Normal}( 0 , \sigma_\gamma^ 2 ) \end{align} $$

The constraint ensures seasonal effects sum to zero over each cycle.

### Trading Application

- Identify **high-seasonality months** to increase position size
- **Calendar spreads**: Long winter months, short summer months for nat gas
- **Harvest pressure**: Short agricultural commodities during harvest

In [ ]:
```
# Generate data with annual seasonality (12 periods)
np.random.seed(42)
T_seasonal = 120  # 10 years of monthly data
S = 12  # Annual seasonality
```

```
# Trend
trend_seasonal = 50 + 0.05 * np.arange(T_seasonal)

# Seasonal pattern (higher in winter: months 0, 1, 11)
seasonal_pattern = np.array([5, 4, 2, -1, -3, -4, -5, -4, -2, 1, 3, 4])
seasonal_component = np.tile(seasonal_pattern, T_seasonal // S)

# Observations
y_seasonal_obs = trend_seasonal + seasonal_component + np.random.normal(0

# Month labels
months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep',
month_indices = np.arange(T_seasonal) % 12

# Visualize
fig, axes = plt.subplots(2, 1, figsize=(14, 8), sharex=True)

# Time series
axes[0].plot(y_seasonal_obs, 'o-', alpha=0.6, markersize=3, label='Observ
axes[0].plot(trend_seasonal, linewidth=2, label='Trend', color='red', lin
axes[0].plot(trend_seasonal + seasonal_component, linewidth=2, label='Tre
axes[0].set_ylabel('Price', fontsize=12)
axes[0].set_title('Natural Gas Prices with Annual Seasonality', fontsize=
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Seasonal pattern
axes[1].bar(range(12), seasonal_pattern, color='steelblue', alpha=0.7)
axes[1].set_xticks(range(12))
axes[1].set_xticklabels(months)
axes[1].set_ylabel('Seasonal Effect', fontsize=12)
axes[1].set_title('Monthly Seasonal Pattern (Winter Premium)', fontsize=1
axes[1].axhline(0, color='black', linestyle='--')
axes[1].grid(True, alpha=0.3, axis='y')

plt.tight_layout()
plt.show()
```

In [ ]:

```
# Build model with trend + seasonality
# For simplicity, we'll use a fixed seasonal pattern approach
# (More sophisticated: stochastic seasonality with state evolution)

with pm.Model() as seasonal_model:
    # Trend component (local level)
    sigma_mu = pm.HalfNormal('sigma_mu', sigma=1)
    mu_init = pm.Normal('mu_init', mu=50, sigma=10)
    innovations_mu = pm.Normal('innovations_mu', mu=0, sigma=sigma_mu, sh
    mu = pm.Deterministic('mu', pm.math.concatenate(
        [[mu_init], mu_init + pm.math.cumsum(innovations_mu)]
    ))

    # Seasonal component (one parameter per month, sum-to-zero constraint
    # We'll use a centered parameterization
    seasonal_raw = pm.Normal('seasonal_raw', mu=0, sigma=5, shape=S-1)
    # Last seasonal component ensures sum = 0
    seasonal_effects = pm.Deterministic('seasonal_effects',
                                        pm.math.concatenate([seasonal_raw

    # Map seasonal effects to each time point
    seasonal_component = seasonal_effects[month_indices]
    # Observation noise
    sigma_y = pm.HalfNormal('sigma_y', sigma=5)

    # Combined mean
    mean = mu + seasonal_component

    # Likelihood
    y = pm.Normal('y', mu=mean, sigma=sigma_y, observed=y_seasonal_obs)

    # Sample
    trace_seasonal = pm.sample(2000, tune=1000, return_inferencedata=True
                               target_accept=0.95, random_seed=42)

print("\nPosterior Summary:")
print(az.summary(trace_seasonal, var_names=['sigma_mu', 'sigma_y', 'seaso
```

In [ ]:

```
# Extract estimates
mu_seasonal_mean = trace_seasonal.posterior['mu'].mean(dim=['chain', 'dra
seasonal_effects_mean = trace_seasonal.posterior['seasonal_effects'].mean
seasonal_component_fitted = seasonal_effects_mean[month_indices]

fig, axes = plt.subplots(2, 1, figsize=(14, 8))

# Decomposition
axes[0].plot(y_seasonal_obs, 'o', alpha=0.4, markersize=3, label='Observe
axes[0].plot(mu_seasonal_mean, linewidth=2, label='Estimated Trend', colo
axes[0].plot(mu_seasonal_mean + seasonal_component_fitted, linewidth=2,
             label='Trend + Seasonal', color='blue')
axes[0].set_ylabel('Price', fontsize=12)
axes[0].set_title('Decomposition: Trend + Seasonality', fontsize=12, font
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Seasonal effects comparison
x_pos = np.arange(12)
axes[1].bar(x_pos - 0.2, seasonal_pattern, width=0.4, label='True', alpha
axes[1].bar(x_pos + 0.2, seasonal_effects_mean, width=0.4, label='Estimat
axes[1].set_xticks(x_pos)
axes[1].set_xticklabels(months)
axes[1].set_ylabel('Seasonal Effect', fontsize=12)
axes[1].set_title('Estimated vs True Seasonal Pattern', fontsize=12, font
axes[1].axhline(0, color='black', linestyle='--')
axes[1].legend()
axes[1].grid(True, alpha=0.3, axis='y')

plt.tight_layout()
plt.show()

# Trading signals
print("\n=== SEASONAL TRADING STRATEGY ===")
sorted_months = np.argsort(seasonal_effects_mean)[::-1]
print("\nMonths with HIGHEST prices (go long ahead of these):")
for i in range(3):
    month_idx = sorted_months[i]
    print(f"  {months[month_idx]}: +${seasonal_effects_mean[month_idx]:.2

print("\nMonths with LOWEST prices (go short or avoid):")
for i in range(3):
    month_idx = sorted_months[-(i+1)]
    print(f"  {months[month_idx]}: ${seasonal_effects_mean[month_idx]:.2f
```

# 5 . Dynamic Regression with Time-Varying Coefficients

## Theory

In static regression, $\beta$ is constant. In **dynamic regression**, coefficients evolve:

$$ \begin{align} y_t &= \alpha_t + \beta_t x_t + \epsilon_t \\ \alpha_t &= \alpha_{t- 1 } + \eta_{\alp
\beta_t &= \beta_{t- 1 } + \eta_{\beta,t} \end{align} $$

## Trading Application

Gold-USD relationship changes:

- **Normal times**: $\beta <  0 $ (inverse relationship)
- **Flight to safety**: $\beta \approx  0 $ (decoupling)
- **Currency crisis**: $\beta$ very negative

Dynamic regression detects these regime changes automatically.

```
In [ ]: # Generate data with time-varying coefficient
np.random.seed(42)
T_dyn = 200

# Predictor: USD index
usd_index = 95 + np.cumsum(np.random.normal(0, 0.3, T_dyn))
usd_standardized = (usd_index - usd_index.mean()) / usd_index.std()

# Time-varying coefficient (becomes more negative over time → stronger in
true_beta_dyn = np.zeros(T_dyn)
true_beta_dyn[0] = -0.5
for t in range(1, T_dyn):
    true_beta_dyn[t] = true_beta_dyn[t-1] + np.random.normal(0, 0.02)

# Gold price
true_alpha_dyn = 1500
gold_price = true_alpha_dyn + true_beta_dyn * usd_standardized * 100 + np

# Visualize
fig, axes = plt.subplots(3, 1, figsize=(14, 10), sharex=True)

# Gold price
axes[0].plot(gold_price, linewidth=1.5, color='gold')
axes[0].set_ylabel('Gold Price ($/oz)', fontsize=12)
axes[0].set_title('Gold Prices', fontsize=12, fontweight='bold')
axes[0].grid(True, alpha=0.3)

# USD Index
axes[1].plot(usd_index, linewidth=1.5, color='green')
axes[1].set_ylabel('USD Index', fontsize=12)
axes[1].set_title('USD Index (Predictor)', fontsize=12, fontweight='bold'
axes[1].grid(True, alpha=0.3)

# Time-varying coefficient
```

```
axes[2].plot(true_beta_dyn, linewidth=2, color='red')
axes[2].axhline(0, color='black', linestyle='--')
axes[2].fill_between(range(T_dyn), 0, true_beta_dyn, where=(true_beta_dyn
                     alpha=0.3, color='red', label='Inverse Relationship'
axes[2].set_xlabel('Time (days)', fontsize=12)
axes[2].set_ylabel('β (Gold-USD)', fontsize=12)
axes[2].set_title('Time-Varying Coefficient: Gold vs USD', fontsize=12, f
axes[2].legend()
axes[2].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

In [ ]:
```
# Build dynamic regression model
with pm.Model() as dynamic_reg_model:
    # Time-varying intercept
    sigma_alpha = pm.HalfNormal('sigma_alpha', sigma=10)
    alpha_init = pm.Normal('alpha_init', mu=1500, sigma=100)
    innovations_alpha = pm.Normal('innovations_alpha', mu=0, sigma=sigma_
    alpha = pm.Deterministic('alpha', pm.math.concatenate(
        [[alpha_init], alpha_init + pm.math.cumsum(innovations_alpha)]
    ))

    # Time-varying coefficient on USD
    sigma_beta = pm.HalfNormal('sigma_beta', sigma=0.5)
    beta_init = pm.Normal('beta_init', mu=-0.5, sigma=1)
    innovations_beta = pm.Normal('innovations_beta', mu=0, sigma=sigma_be
    beta = pm.Deterministic('beta', pm.math.concatenate(
        [[beta_init], beta_init + pm.math.cumsum(innovations_beta)]
    ))

    # Observation noise
    sigma_y = pm.HalfNormal('sigma_y', sigma=50)

    # Regression equation
    mu = alpha + beta * usd_standardized * 100

    # Likelihood
    y = pm.Normal('y', mu=mu, sigma=sigma_y, observed=gold_price)

    # Sample
    trace_dyn_reg = pm.sample(2000, tune=1000, return_inferencedata=True,
                              target_accept=0.95, random_seed=42)

print("\nPosterior Summary:")
print(az.summary(trace_dyn_reg, var_names=['sigma_alpha', 'sigma_beta', '
```

In [ ]:
```
# Visualize time-varying coefficient
beta_dyn_mean = trace_dyn_reg.posterior['beta'].mean(dim=['chain', 'draw'
beta_dyn_lower = np.percentile(trace_dyn_reg.posterior['beta'].values, 2.
beta_dyn_upper = np.percentile(trace_dyn_reg.posterior['beta'].values, 97

fig, axes = plt.subplots(2, 1, figsize=(14, 8), sharex=True)

# Fitted vs Observed
alpha_dyn_mean = trace_dyn_reg.posterior['alpha'].mean(dim=['chain', 'dra
fitted = alpha_dyn_mean + beta_dyn_mean * usd_standardized * 100

axes[0].plot(gold_price, 'o', alpha=0.4, markersize=3, label='Observed',
axes[0].plot(fitted, linewidth=2, label='Fitted', color='gold')
axes[0].set_ylabel('Gold Price ($/oz)', fontsize=12)
axes[0].set_title('Dynamic Regression Fit', fontsize=12, fontweight='bold
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Time-varying coefficient
axes[1].plot(true_beta_dyn, linewidth=2, label='True β', color='red', lin
axes[1].plot(beta_dyn_mean, linewidth=2.5, label='Estimated β', color='bl
axes[1].fill_between(range(T_dyn), beta_dyn_lower, beta_dyn_upper,
                     alpha=0.3, color='blue', label='95% CI')
axes[1].axhline(0, color='black', linestyle='--', alpha=0.7)
axes[1].set_xlabel('Time (days)', fontsize=12)
axes[1].set_ylabel('β (Gold-USD)', fontsize=12)
axes[1].set_title('Time-Varying Coefficient: Gold vs USD', fontsize=12, f
axes[1].legend()
axes[1].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Trading signals
current_beta = beta_dyn_mean[-1]
recent_beta_change = beta_dyn_mean[-1] - beta_dyn_mean[-20]

print(f"\n=== GOLD-USD RELATIONSHIP ===")
print(f"Current β: {current_beta:.3f}")
print(f"20-day change in β: {recent_beta_change:.3f}")
if current_beta < -0.5:
    print("→ STRONG INVERSE: USD strength heavily pressures gold")
    print("   Strategy: Short gold when USD rallies")
elif current_beta > -0.2:
    print("→ WEAK INVERSE or DECOUPLED: Relationship breakdown")
    print("   Strategy: USD less reliable predictor, watch other drivers"
else:
    print("→ MODERATE INVERSE: Typical negative correlation")
    print("   Strategy: Use USD as hedge factor")
```

## 6 . Practical Application: Gold Prices with USD and Infl

### Trading Context

Gold is driven by:

1 . **USD strength**: Inverse (commodities priced in USD)
2 . **Inflation expectations**: Positive (inflation hedge)
3 . **Real rates**: Inverse (opportunity cost)

We'll build a full BSTS model with:

- Local linear trend
- Dynamic regression on USD and inflation
- Student-t likelihood (robust to outliers)

In [ ]:
```
# Generate realistic gold price data
np.random.seed(42)
T_gold = 250  # ~1 year of daily data
```

```python
# Predictors
usd = 95 + np.cumsum(np.random.normal(0, 0.2, T_gold))  # USD index
inflation = 2.5 + np.cumsum(np.random.normal(0, 0.05, T_gold))  # Inflati

# Standardize
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
predictors = scaler.fit_transform(np.column_stack([usd, inflation]))
usd_std = predictors[:, 0]
inflation_std = predictors[:, 1]

# Time-varying coefficients
beta_usd_true = np.zeros(T_gold)
beta_inflation_true = np.zeros(T_gold)
beta_usd_true[0] = -15  # Negative: strong USD → lower gold
beta_inflation_true[0] = 25  # Positive: inflation → higher gold

for t in range(1, T_gold):
    beta_usd_true[t] = beta_usd_true[t-1] + np.random.normal(0, 0.5)
    beta_inflation_true[t] = beta_inflation_true[t-1] + np.random.normal(

# Trend (local linear trend)
mu_gold = np.zeros(T_gold)
beta_trend = np.zeros(T_gold)
mu_gold[0] = 1800
beta_trend[0] = 0.3

for t in range(1, T_gold):
    beta_trend[t] = beta_trend[t-1] + np.random.normal(0, 0.05)
    mu_gold[t] = mu_gold[t-1] + beta_trend[t-1] + np.random.normal(0, 2)

# Gold price
gold_full = (mu_gold +
             beta_usd_true * usd_std +
             beta_inflation_true * inflation_std +
             np.random.standard_t(df=5, size=T_gold) * 8)  # Fat tails

# Visualize
fig, axes = plt.subplots(4, 1, figsize=(14, 12), sharex=True)

axes[0].plot(gold_full, linewidth=1.5, color='gold')
axes[0].set_ylabel('Gold ($/oz)', fontsize=11)
axes[0].set_title('Gold Prices', fontsize=12, fontweight='bold')
axes[0].grid(True, alpha=0.3)

axes[1].plot(usd, linewidth=1.5, color='green')
axes[1].set_ylabel('USD Index', fontsize=11)
axes[1].set_title('USD Index', fontsize=12, fontweight='bold')
axes[1].grid(True, alpha=0.3)

axes[2].plot(inflation, linewidth=1.5, color='red')
axes[2].set_ylabel('Inflation (%)', fontsize=11)
axes[2].set_title('Inflation Expectations', fontsize=12, fontweight='bold
axes[2].grid(True, alpha=0.3)

axes[3].plot(beta_usd_true, linewidth=2, label='β (USD)', color='green')
axes[3].plot(beta_inflation_true, linewidth=2, label='β (Inflation)', col
axes[3].axhline(0, color='black', linestyle='--')
axes[3].set_xlabel('Time (days)', fontsize=11)
```

```python
axes[3].set_ylabel('Coefficient', fontsize=11)
axes[3].set_title('Time-Varying Coefficients (True)', fontsize=12, fontwe
axes[3].legend()
axes[3].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

In [ ]:
```python
# Build full BSTS model
# This is computationally intensive, so we'll use a simplified version

with pm.Model() as bsts_full:
    # Trend component (local level only for speed)
    sigma_mu = pm.HalfNormal('sigma_mu', sigma=5)
    mu_init = pm.Normal('mu_init', mu=1800, sigma=100)
    innovations_mu = pm.Normal('innovations_mu', mu=0, sigma=sigma_mu, sh
    mu = pm.Deterministic('mu', pm.math.concatenate(
        [[mu_init], mu_init + pm.math.cumsum(innovations_mu)]
    ))

    # Time-varying USD coefficient
    sigma_beta_usd = pm.HalfNormal('sigma_beta_usd', sigma=2)
    beta_usd_init = pm.Normal('beta_usd_init', mu=-15, sigma=10)
    innovations_beta_usd = pm.Normal('innovations_beta_usd', mu=0, sigma=
    beta_usd = pm.Deterministic('beta_usd', pm.math.concatenate(
        [[beta_usd_init], beta_usd_init + pm.math.cumsum(innovations_beta
    ))

    # Time-varying inflation coefficient
    sigma_beta_inf = pm.HalfNormal('sigma_beta_inf', sigma=2)
    beta_inf_init = pm.Normal('beta_inf_init', mu=25, sigma=10)
    innovations_beta_inf = pm.Normal('innovations_beta_inf', mu=0, sigma=
    beta_inf = pm.Deterministic('beta_inf', pm.math.concatenate(
        [[beta_inf_init], beta_inf_init + pm.math.cumsum(innovations_beta
    ))

    # Robust likelihood (Student-t)
    nu = pm.Gamma('nu', alpha=2, beta=0.1)
    sigma_y = pm.HalfNormal('sigma_y', sigma=20)

    # Combined mean
    mean = mu + beta_usd * usd_std + beta_inf * inflation_std

    # Likelihood
    y = pm.StudentT('y', nu=nu, mu=mean, sigma=sigma_y, observed=gold_ful

    # Sample
    trace_bsts_full = pm.sample(1500, tune=1000, return_inferencedata=Tru
                                target_accept=0.95, random_seed=42, chain

print("\nPosterior Summary:")
print(az.summary(trace_bsts_full, var_names=['sigma_mu', 'sigma_beta_usd'
                                              'sigma_y', 'nu']))
```

In [ ]:
```python
# Decompose gold prices
mu_est = trace_bsts_full.posterior['mu'].mean(dim=['chain', 'draw']).valu
beta_usd_est = trace_bsts_full.posterior['beta_usd'].mean(dim=['chain', '
beta_inf_est = trace_bsts_full.posterior['beta_inf'].mean(dim=['chain', '
# Components
usd_effect = beta_usd_est * usd_std
inflation_effect = beta_inf_est * inflation_std
fitted = mu_est + usd_effect + inflation_effect

fig, axes = plt.subplots(3, 1, figsize=(14, 10), sharex=True)

# Observed vs Fitted
axes[0].plot(gold_full, 'o', alpha=0.4, markersize=3, label='Observed', c
axes[0].plot(fitted, linewidth=2, label='Fitted (BSTS)', color='gold')
axes[0].set_ylabel('Gold Price ($/oz)', fontsize=11)
axes[0].set_title('BSTS Model: Fitted vs Observed', fontsize=12, fontweig
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Component decomposition
axes[1].plot(mu_est, linewidth=2, label='Trend (μ)', color='black')
axes[1].plot(mu_est + usd_effect, linewidth=2, label='Trend + USD', color
axes[1].plot(fitted, linewidth=2, label='Trend + USD + Inflation', color=
axes[1].set_ylabel('Price Components ($/oz)', fontsize=11)
axes[1].set_title('Price Decomposition', fontsize=12, fontweight='bold')
axes[1].legend()
axes[1].grid(True, alpha=0.3)

# Time-varying coefficients
axes[2].plot(beta_usd_true, linewidth=2, label='True β (USD)', color='gre
axes[2].plot(beta_usd_est, linewidth=2, label='Est β (USD)', color='green
axes[2].plot(beta_inflation_true, linewidth=2, label='True β (Inflation)'
axes[2].plot(beta_inf_est, linewidth=2, label='Est β (Inflation)', color=
axes[2].axhline(0, color='black', linestyle='--', alpha=0.5)
axes[2].set_xlabel('Time (days)', fontsize=11)
axes[2].set_ylabel('Coefficient', fontsize=11)
axes[2].set_title('Time-Varying Coefficients', fontsize=12, fontweight='b
axes[2].legend(loc='best', fontsize=9)
axes[2].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"\nModel RMSE: ${np.sqrt(np.mean((gold_full - fitted)**2)):.2f}")
```

In [ ]:

```
# Trading signals based on decomposition
current_trend = mu_est[-1]
current_beta_usd = beta_usd_est[-1]
current_beta_inf = beta_inf_est[-1]
current_usd_effect = usd_effect[-1]
current_inf_effect = inflation_effect[-1]
current_fair_value = fitted[-1]
current_price = gold_full[-1]
mispricing = current_price - current_fair_value

print("\n" + "="*60)
print("GOLD TRADING DASHBOARD")
print("="*60)

print(f"\nCurrent Price: ${current_price:.2f}")
print(f"Fair Value (BSTS): ${current_fair_value:.2f}")
print(f"Mispricing: ${mispricing:+.2f} ({mispricing/current_fair_value*10

if abs(mispricing) > 20:
    if mispricing > 0:
        print("→ OVERVALUED: Consider short position or tighten stops on
    else:
        print("→ UNDERVALUED: Consider long position or add to existing l
else:
    print("→ FAIRLY VALUED: No mispricing signal")

print(f"\n--- PRICE DECOMPOSITION ---")
print(f"Trend component: ${current_trend:.2f}")
print(f"USD effect: ${current_usd_effect:+.2f} (β = {current_beta_usd:.2f
print(f"Inflation effect: ${current_inf_effect:+.2f} (β = {current_beta_i

print(f"\n--- FACTOR SENSITIVITIES ---")
print(f"β (USD): {current_beta_usd:.2f}")
if current_beta_usd < -10:
    print("  → STRONG INVERSE: USD rallies hurt gold significantly")
    print("     Strategy: Hedge gold positions with long USD")
elif current_beta_usd > -5:
    print("  → WEAK INVERSE: Gold-USD correlation breaking down")
    print("     Strategy: USD less reliable hedge, watch other factors")

print(f"\nβ (Inflation): {current_beta_inf:.2f}")
if current_beta_inf > 20:
    print("  → STRONG INFLATION HEDGE: Rising inflation supports gold")
    print("     Strategy: Increase gold allocation in inflationary enviro
elif current_beta_inf < 10:
    print("  → WEAK INFLATION HEDGE: Gold not responding to inflation")
    print("     Strategy: Consider other inflation hedges (TIPS, commodit

print("\n" + "="*60)
```

## Knowledge Check Quiz

### Question 1

What is the key advantage of state-space models over standard regression?

A) Faster computation

B) Separate latent "true" state from noisy observations

C) Require fewer data points

D) Always more accurate

**Answer: B** - State-space models explicitly model hidden states (e.g., "fair value") that evolve ov separating signal from noise.

---

### Question 2

In a local level model, what does the ratio σ²_μ / σ²_y control?

A) Model accuracy

B) Smoothness of the latent level

C) Number of parameters

D) Forecast horizon

**Answer: B** - Higher ratio → level changes rapidly (trending). Lower ratio → level nearly constan
reverting).

---

## Question 3

Why are seasonal components important for commodity trading?

A) They increase model complexity
B) They capture predictable price patterns (harvest, weather)
C) They are required by PyMC
D) They replace the need for fundamentals

**Answer: B** - Commodities have strong seasonal patterns (e.g., natural gas winter demand, agric
harvest pressure) that create tradable opportunities.

---

## Question 4

What does a time-varying coefficient (dynamic regression) detect?

A) Data errors
B) Regime changes in relationships between variables
C) Missing data
D) Outliers

**Answer: B** - Dynamic regression allows coefficients to evolve, detecting when relationships stre
weaken, or reverse (e.g., Gold-USD correlation during crises).

---

## Question 5

In the gold BSTS model, what does it mean if β (USD) becomes less negative over time?

A) USD is getting stronger
B) The inverse Gold-USD relationship is weakening
C) Gold is becoming more volatile
D) Model is overfitting

**Answer: B** - A coefficient moving toward zero indicates the relationship is weakening (decouplin
possibly due to regime change or other dominant factors.

---

## Exercises

### Exercise 1 : Crude Oil Seasonality

Build a BSTS model for crude oil with:

- Local linear trend

- Weekly seasonality (S=5 for trading days)
- Compare model with vs without seasonality using WAIC
- Identify which day of the week has highest/lowest prices

## Exercise 2: Natural Gas Multi-Seasonality

Natural gas has multiple seasonal patterns:

- Annual (S=12 months): Winter heating
- Weekly (S=7 days): Storage report Thursdays

Build a model with both. Which is more important?

## Exercise 3: Corn Prices with Weather

Generate synthetic corn data with:

- Annual harvest seasonality (low prices in fall)
- Dynamic regression on rainfall (time-varying coefficient)
- Simulate drought year where β (rainfall) spikes
- Detect the regime change automatically

## Exercise 4: Regime Detection

Using the gold BSTS model:

1. Define a regime as "when β (USD) < -20"
2. Calculate P(regime) at each time point
3. Visualize regime probabilities over time
4. Design a trading rule: increase position size when P(high sensitivity regime) > 0.8

## Exercise 5: Forecast Decomposition

For the gold model:

1. Forecast 30 days ahead
2. Decompose forecast into: trend + USD effect + inflation effect
3. Scenario analysis:
    - If USD rises 2%, how much does gold forecast drop?
    - If inflation rises 0.5%, how much does gold forecast rise?
4. Calculate forecast credible intervals

---

## Summary

In this module, you learned:

1. **State-Space Models**: Separate latent states from noisy observations
2. **Local Level Model**: Random walk for slowly changing "fair value"

3. **Local Linear Trend**: Add time-varying slope for trend detection
4. **Seasonal Components**: Model predictable cycles (harvest, weather)
5. **Dynamic Regression**: Time-varying coefficients detect regime changes
6. **Gold Application**: Full BSTS with trend, USD, and inflation factors

### Key Takeaways for Trading

- **Decomposition**: Break prices into trend + seasonal + fundamental + noise
- **Fair Value**: Use latent level estimate for mean-reversion trades
- **Regime Detection**: Time-varying coefficients automatically identify structural breaks
- **Seasonality**: Exploit recurring patterns with statistical confidence
- **Factor Sensitivity**: Track how responsive commodities are to drivers (USD, inflation)
- **Robust Estimation**: Student-t likelihood handles price shocks without bias

BSTS models provide a principled framework for understanding commodity price dynamics in nc stationary, regime-switching markets.

---

## Preview of Next Module

### Module 7: Hierarchical Models for Multiple Commodities

So far, we've modeled one commodity at a time. But markets are interconnected:

- **Energy complex**: WTI, Brent, Natural Gas share supply shocks
- **Grains**: Corn, Wheat, Soybeans compete for farmland
- **Metals**: Gold, Silver, Copper respond to economic cycles

**Hierarchical models** pool information across related commodities:

- Estimate **group-level parameters** (e.g., average energy seasonality)
- Allow **commodity-specific deviations** (Brent vs WTI spreads)
- **Shrinkage**: Regularize estimates for thinly-traded commodities

You'll learn:

- Partial pooling vs complete pooling vs no pooling
- Varying intercepts and slopes
- Cross-commodity correlation structures
- Pairs trading with hierarchical models

**See you in Module 7!**

# Module 7 : Hierarchical Models for Multiple Commodities

## Learning Objectives

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

1. Understand hierarchical/multilevel model structures
2. Distinguish between complete pooling, no pooling, and partial pooling
3. Build hierarchical models with varying intercepts by commodity
4. Implement varying slopes for commodity-specific relationships
5. Model cross-commodity correlation structures
6. Apply shrinkage to improve estimates for thinly-traded commodities
7. Use hierarchical models for agricultural commodities with shared weather impacts

## Why This Matters for Trading

Commodities don't exist in isolation. They form interconnected complexes:

**Energy Complex**:

- WTI (West Texas Intermediate) crude oil
- Brent crude oil
- Natural gas
- Share supply/demand drivers, but have distinct regional factors

**Agricultural Complex**:

- Corn, Wheat, Soybeans
- Compete for farmland
- Share weather risks (drought affects all)
- Different demand patterns (food, feed, fuel)

**Metals Complex**:

- Gold, Silver, Copper
- Economic growth affects all, but to varying degrees

### Trading Advantages of Hierarchical Models

1. **Information Borrowing**: Thinly-traded commodities (e.g., oats) borrow strength from liquid (e.g., corn)
2. **Pairs Trading**: Identify when a commodity deviates from its complex (e.g., WTI-Brent sprea
3. **Risk Management**: Model correlations for portfolio optimization
4. **Shrinkage**: Regularize estimates to prevent overfitting
5. **Cross-Sectional Analysis**: Which commodity in a complex is most/least sensitive to a drive

6 . **Missing Data**: Forecast one commodity using information from others

**Example**: If drought affects corn, wheat, and soybeans, but you only have recent corn data, the hierarchical model uses corn's drought response to update beliefs about wheat and soybean res

This is **statistical arbitrage at the structural level**.

In [ ]:
```
# Standard imports
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')

np.random.seed(42)
plt.style.use('seaborn-v0_8-whitegrid')

print(f"PyMC version: {pm.__version__}")
print(f"ArviZ version: {az.__version__}")
```

# 1 . Introduction: Pooling Strategies

## The Pooling Spectrum

Suppose we have price data for 3 commodities and want to estimate each commodity's m price.

**Complete Pooling** (ignore differences): $$ y_{ij} \sim \text{Normal}(\mu, \sigma) $$

- One $\mu$ for all commodities
- Assumes all commodities are identical
- **Problem**: Ignores commodity-specific factors

**No Pooling** (independent estimates): $$ y_{ij} \sim \text{Normal}(\mu_j, \sigma_j) $$

- Separate $\mu_j$ for each commodity $j$
- No information sharing
- **Problem**: Overfits with small samples, ignores commonalities

**Partial Pooling** (hierarchical): $$ \begin{align} y_{ij} &\sim \text{Normal}(\mu_j, \sigma) \\ \mu_j &
\text{Normal}(\mu_{\text{global}}, \tau) \end{align} $$

- Commodity-specific $\mu_j$, but they share a common prior
- **Shrinkage**: Estimates pulled toward group mean
- **Best of both worlds**: Commodity differences + information borrowing

## Trading Interpretation

- **Complete pooling**: "All energy commodities are the same" (wrong)
- **No pooling**: "WTI and Brent are unrelated" (misses correlation)

- **Partial pooling**: "They're different, but share common drivers" (realistic)

In [ ]:
```python
# Generate synthetic data: 3 commodities with different means
np.random.seed(42)

# True parameters
true_global_mean = 60.0
true_tau = 8.0  # Between-commodity variation
true_sigma = 5.0  # Within-commodity variation

# 3 commodities: WTI, Brent, Natural Gas (converted to oil-equivalent)
commodities = ['WTI', 'Brent', 'NatGas']
n_commodities = len(commodities)
n_obs_per_commodity = [50, 60, 30]  # Different sample sizes

# Generate commodity-specific means
np.random.seed(42)
true_mu = np.array([58, 62, 55])  # WTI slightly below Brent, NatGas lowe

# Generate data
data_list = []
for j, (commodity, n_obs) in enumerate(zip(commodities, n_obs_per_commodi
    prices = np.random.normal(true_mu[j], true_sigma, n_obs)
    for price in prices:
        data_list.append({'commodity': commodity, 'price': price})

df_pooling = pd.DataFrame(data_list)

# Visualize
fig, ax = plt.subplots(figsize=(12, 6))

positions = [1, 2, 3]
for j, commodity in enumerate(commodities):
    subset = df_pooling[df_pooling['commodity'] == commodity]['price']
    ax.scatter([positions[j]] * len(subset), subset, alpha=0.5, s=50, lab
    ax.plot([positions[j] - 0.3, positions[j] + 0.3], [true_mu[j], true_m
            'r-', linewidth=3, label='True Mean' if j == 0 else '')

ax.axhline(true_global_mean, color='black', linestyle='--', linewidth=2,
           label=f'Global Mean = {true_global_mean}')
ax.set_xticks(positions)
ax.set_xticklabels(commodities)
ax.set_ylabel('Price ($/barrel equivalent)', fontsize=12)
ax.set_title('Energy Commodity Prices: Different Means, Shared Drivers',
ax.legend()
ax.grid(True, alpha=0.3, axis='y')
plt.tight_layout()
plt.show()

print("Data Summary:")
print(df_pooling.groupby('commodity')['price'].describe())
```

In [ ]:
```python
# Encode commodities as integers for PyMC
commodity_idx = pd.Categorical(df_pooling['commodity'], categories=commod
y_pooling = df_pooling['price'].values

# Model 1: Complete Pooling
with pm.Model() as complete_pooling:
    mu_global = pm.Normal('mu_global', mu=60, sigma=20)
    sigma = pm.HalfNormal('sigma', sigma=10)

    y = pm.Normal('y', mu=mu_global, sigma=sigma, observed=y_pooling)

    trace_complete = pm.sample(2000, tune=1000, return_inferencedata=True

# Model 2: No Pooling
with pm.Model() as no_pooling:
    mu = pm.Normal('mu', mu=60, sigma=20, shape=n_commodities)
    sigma = pm.HalfNormal('sigma', sigma=10)

    y = pm.Normal('y', mu=mu[commodity_idx], sigma=sigma, observed=y_pool

    trace_no_pooling = pm.sample(2000, tune=1000, return_inferencedata=Tr

# Model 3: Partial Pooling (Hierarchical)
with pm.Model() as partial_pooling:
    # Hyperpriors (group-level)
    mu_global = pm.Normal('mu_global', mu=60, sigma=20)
    tau = pm.HalfNormal('tau', sigma=10)  # Between-commodity std

    # Commodity-specific means
    mu = pm.Normal('mu', mu=mu_global, sigma=tau, shape=n_commodities)

    # Observation model
    sigma = pm.HalfNormal('sigma', sigma=10)
    y = pm.Normal('y', mu=mu[commodity_idx], sigma=sigma, observed=y_pool

    trace_partial = pm.sample(2000, tune=1000, return_inferencedata=True,

print("\n=== COMPLETE POOLING ===")
print(az.summary(trace_complete, var_names=['mu_global', 'sigma']))

print("\n=== NO POOLING ===")
print(az.summary(trace_no_pooling, var_names=['mu', 'sigma']))

print("\n=== PARTIAL POOLING ===")
print(az.summary(trace_partial, var_names=['mu_global', 'tau', 'mu', 'sig
```

In [ ]:
```
# Compare estimates
complete_est = trace_complete.posterior['mu_global'].mean().item()
no_pooling_est = trace_no_pooling.posterior['mu'].mean(dim=['chain', 'dra
partial_pooling_est = trace_partial.posterior['mu'].mean(dim=['chain', 'd

# Visualization
fig, ax = plt.subplots(figsize=(12, 6))

x = np.arange(n_commodities)
width = 0.2

# True means
ax.bar(x - 1.5*width, true_mu, width, label='True', alpha=0.8, color='bla

# Complete pooling
ax.bar(x - 0.5*width, [complete_est]*n_commodities, width,
       label='Complete Pooling', alpha=0.7, color='red')

# No pooling
ax.bar(x + 0.5*width, no_pooling_est, width,
       label='No Pooling', alpha=0.7, color='blue')
```

```
# Partial pooling
ax.bar(x + 1.5*width, partial_pooling_est, width,
       label='Partial Pooling', alpha=0.7, color='green')

ax.set_xlabel('Commodity', fontsize=12)
ax.set_ylabel('Estimated Mean Price ($/barrel)', fontsize=12)
ax.set_title('Pooling Strategies Comparison', fontsize=14, fontweight='bo
ax.set_xticks(x)
ax.set_xticklabels(commodities)
ax.legend()
ax.grid(True, alpha=0.3, axis='y')
plt.tight_layout()
plt.show()

# Quantify errors
print("\n=== ESTIMATION ERRORS (True - Estimated) ===")
print(f"\nComplete Pooling:")
for j, commodity in enumerate(commodities):
    error = true_mu[j] - complete_est
    print(f"  {commodity}: {error:+.2f}")
print(f"  RMSE: {np.sqrt(np.mean((true_mu - complete_est)**2)):.3f}")

print(f"\nNo Pooling:")
for j, commodity in enumerate(commodities):
    error = true_mu[j] - no_pooling_est[j]
    print(f"  {commodity}: {error:+.2f}")
print(f"  RMSE: {np.sqrt(np.mean((true_mu - no_pooling_est)**2)):.3f}")

print(f"\nPartial Pooling (Best):")
for j, commodity in enumerate(commodities):
    error = true_mu[j] - partial_pooling_est[j]
    print(f"  {commodity}: {error:+.2f}")
print(f"  RMSE: {np.sqrt(np.mean((true_mu - partial_pooling_est)**2)):.3f

print("\n→ Partial pooling has lowest RMSE, especially for small-sample c
```

## 2 . Shrinkage Effect

### Theory

Partial pooling **shrinks** commodity-specific estimates toward the group mean:

$$ \hat{\mu}_j \approx \lambda \bar{y}_j + ( 1 - \lambda) \mu_{\text{global}} $$

Where:

- $\lambda$: Shrinkage factor (depends on sample size and $\tau/\sigma$ ratio)
- Small sample → low $\lambda$ → strong shrinkage
- Large sample → high $\lambda$ → weak shrinkage

### Trading Application

- **Thinly-traded commodities**: Borrow strength from liquid ones
- **New contracts**: Use existing commodity data to initialize forecasts
- **Regularization**: Prevent overfitting on noisy data

In [ ]:
```
# Visualize shrinkage
fig, ax = plt.subplots(figsize=(12, 7))

# Sample means (raw data)
sample_means = df_pooling.groupby('commodity')['price'].mean().values

# Global mean
global_mean = partial_pooling_est.mean()

for j, commodity in enumerate(commodities):
    # Arrow from sample mean to hierarchical estimate
    ax.annotate('', xy=(j, partial_pooling_est[j]), xytext=(j, sample_mea
                arrowprops=dict(arrowstyle='->', lw=2, color='red', alpha

    # Sample mean
    ax.scatter(j, sample_means[j], s=200, color='blue', marker='o',
               edgecolors='black', linewidths=2, zorder=5, label='Sample

    # Hierarchical estimate
    ax.scatter(j, partial_pooling_est[j], s=200, color='green', marker='s
               edgecolors='black', linewidths=2, zorder=5, label='Hierarc

    # True mean
    ax.scatter(j, true_mu[j], s=200, color='red', marker='^',
               edgecolors='black', linewidths=2, zorder=5, label='True Me

    # Sample size annotation
    ax.text(j, sample_means[j] + 1.5, f'n={n_obs_per_commodity[j]}',
            ha='center', fontsize=10, fontweight='bold')

# Global mean line
ax.axhline(global_mean, color='purple', linestyle='--', linewidth=2,
           label=f'Global Mean = {global_mean:.2f}', alpha=0.7)

ax.set_xticks(range(n_commodities))
ax.set_xticklabels(commodities)
ax.set_ylabel('Price ($/barrel)', fontsize=12)
ax.set_title('Shrinkage Effect: Sample Means → Hierarchical Estimates', f
ax.legend(loc='upper right')
ax.grid(True, alpha=0.3, axis='y')
plt.tight_layout()
plt.show()

print("\n=== SHRINKAGE ANALYSIS ===")
for j, commodity in enumerate(commodities):
    shrinkage = sample_means[j] - partial_pooling_est[j]
    print(f"\n{commodity} (n={n_obs_per_commodity[j]}):")
    print(f"  Sample mean: {sample_means[j]:.2f}")
    print(f"  Hierarchical estimate: {partial_pooling_est[j]:.2f}")
    print(f"  Shrinkage: {shrinkage:+.2f} (toward global mean)")
    print(f"  → {'STRONG' if abs(shrinkage) > 1 else 'WEAK'} shrinkage ({
```

## 3 . Varying Intercepts and Slopes

### Theory

Extend hierarchical models to regression with commodity-specific intercepts AND slopes:

$$ \begin{align} y_{ij} &\sim \text{Normal}(\mu_{ij}, \sigma) \\ \mu_{ij} &= \alpha_j + \beta_j x_i \\ \
&\sim \text{Normal}(\mu_\alpha, \tau_\alpha) \\ \beta_j &\sim \text{Normal}(\mu_\beta, \tau_\beta
\end{align} $$

Where:

- $\alpha_j$: Commodity-specific intercept (base price)
- $\beta_j$: Commodity-specific slope (sensitivity to predictor)
- $\mu_\alpha, \tau_\alpha$: Group-level intercept parameters
- $\mu_\beta, \tau_\beta$: Group-level slope parameters

### Trading Application

**Question**: How much does each energy commodity respond to USD changes?

- WTI might have $\beta_{\text{WTI}} = - 1 . 2 $ (highly sensitive)
- Brent might have $\beta_{\text{Brent}} = - 0 . 9 $ (less sensitive, European market)
- Natural gas might have $\beta_{\text{NatGas}} = - 0 . 5 $ (regional market, less USD-driver

Hierarchical model estimates these while preventing overfitting.

In [ ]:
```
# Generate data with varying intercepts and slopes
np.random.seed(42)
n_per_commodity = 80

# Predictor: USD index (standardized)
usd_values = np.random.normal(0, 1, n_per_commodity * n_commodities)

# True parameters
true_alpha = np.array([60, 64, 56])  # Different base prices
true_beta = np.array([-1.2, -0.9, -0.5])  # Different USD sensitivities
```

```
true_sigma_y = 3.0

# Generate prices
data_varying = []
for j, commodity in enumerate(commodities):
    idx_start = j * n_per_commodity
    idx_end = (j + 1) * n_per_commodity

    usd_subset = usd_values[idx_start:idx_end]
    prices = true_alpha[j] + true_beta[j] * usd_subset + np.random.normal

    for i, (usd, price) in enumerate(zip(usd_subset, prices)):
        data_varying.append({
            'commodity': commodity,
            'usd': usd,
            'price': price
        })

df_varying = pd.DataFrame(data_varying)

# Visualize
fig, axes = plt.subplots(1, 3, figsize=(18, 5), sharey=True)

colors = ['steelblue', 'darkorange', 'green']
for j, commodity in enumerate(commodities):
    subset = df_varying[df_varying['commodity'] == commodity]

    axes[j].scatter(subset['usd'], subset['price'], alpha=0.5, s=30, colo

    # True regression line
    usd_range = np.linspace(-3, 3, 100)
    axes[j].plot(usd_range, true_alpha[j] + true_beta[j] * usd_range,
                 'r--', linewidth=2, label=f'True: α={true_alpha[j]:.1f},

    axes[j].set_xlabel('USD Index (standardized)', fontsize=11)
    if j == 0:
        axes[j].set_ylabel('Price ($/barrel)', fontsize=11)
    axes[j].set_title(f'{commodity}', fontsize=12, fontweight='bold')
    axes[j].legend(loc='upper right')
    axes[j].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("\nNote: Different slopes show varying USD sensitivity across commo
```

In [ ]:

```
# Build hierarchical model with varying intercepts and slopes
commodity_idx_varying = pd.Categorical(df_varying['commodity'], categorie
usd_varying = df_varying['usd'].values
y_varying = df_varying['price'].values

with pm.Model() as varying_model:
    # Hyperpriors for intercepts
    mu_alpha = pm.Normal('mu_alpha', mu=60, sigma=10)
    tau_alpha = pm.HalfNormal('tau_alpha', sigma=10)

    # Hyperpriors for slopes
    mu_beta = pm.Normal('mu_beta', mu=-1, sigma=2)
    tau_beta = pm.HalfNormal('tau_beta', sigma=1)
    # Commodity-specific intercepts and slopes
    alpha = pm.Normal('alpha', mu=mu_alpha, sigma=tau_alpha, shape=n_comm
    beta = pm.Normal('beta', mu=mu_beta, sigma=tau_beta, shape=n_commodit

    # Observation noise
    sigma_y = pm.HalfNormal('sigma_y', sigma=10)

    # Regression
    mu = alpha[commodity_idx_varying] + beta[commodity_idx_varying] * usd

    # Likelihood
    y = pm.Normal('y', mu=mu, sigma=sigma_y, observed=y_varying)

    # Sample
    trace_varying = pm.sample(2000, tune=1000, return_inferencedata=True,

print("\nPosterior Summary:")
print(az.summary(trace_varying, var_names=['mu_alpha', 'tau_alpha', 'mu_b
                                             'alpha', 'beta', 'sigma_y']))
```

In [ ]:
```
# Visualize estimates
alpha_est = trace_varying.posterior['alpha'].mean(dim=['chain', 'draw']).
beta_est = trace_varying.posterior['beta'].mean(dim=['chain', 'draw']).va

fig, axes = plt.subplots(1, 3, figsize=(18, 5), sharey=True)

for j, commodity in enumerate(commodities):
    subset = df_varying[df_varying['commodity'] == commodity]

    axes[j].scatter(subset['usd'], subset['price'], alpha=0.4, s=30, colo

    # True line
    usd_range = np.linspace(-3, 3, 100)
    axes[j].plot(usd_range, true_alpha[j] + true_beta[j] * usd_range,
                 'r--', linewidth=2, alpha=0.7, label=f'True: α={true_alp

    # Estimated line
    axes[j].plot(usd_range, alpha_est[j] + beta_est[j] * usd_range,
                 'b-', linewidth=2.5, label=f'Est: α={alpha_est[j]:.1f},

    axes[j].set_xlabel('USD Index (standardized)', fontsize=11)
    if j == 0:
        axes[j].set_ylabel('Price ($/barrel)', fontsize=11)
    axes[j].set_title(f'{commodity}', fontsize=12, fontweight='bold')
    axes[j].legend(loc='upper right', fontsize=9)
    axes[j].grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Trading analysis
print("\n=== USD SENSITIVITY ANALYSIS ===")
for j, commodity in enumerate(commodities):
    beta_samples = trace_varying.posterior['beta'].values[:, :, j].flatte
    beta_mean = beta_samples.mean()
    beta_ci = np.percentile(beta_samples, [2.5, 97.5])

    print(f"\n{commodity}:")
    print(f"  β (USD): {beta_mean:.3f} (95% CI: [{beta_ci[0]:.3f}, {beta_
    print(f"  True β: {true_beta[j]:.3f}")
    if abs(beta_mean) > 1.0:
        print(f"  → HIGH SENSITIVITY: 1% USD move → {abs(beta_mean):.2f}%
    elif abs(beta_mean) > 0.7:
        print(f"  → MODERATE SENSITIVITY: 1% USD move → {abs(beta_mean):.
    else:
        print(f"  → LOW SENSITIVITY: USD less important driver")

# Group-level parameters
mu_beta_samples = trace_varying.posterior['mu_beta'].values.flatten()
print(f"\n\nGroup-level β (average across commodities): {mu_beta_samples.
print(f"Between-commodity variation (τ_β): {trace_varying.posterior['tau_
```

## 4 . Cross-Commodity Correlation Structure

### Theory

So far, we've assumed commodities are independent conditional on group parameters. But ener commodities are correlated due to shared shocks.

**Multivariate hierarchical model**:

$$ \begin{bmatrix} \alpha_ 1 \\ \alpha_ 2 \\ \alpha_ 3 \end{bmatrix} \sim \text{MVNormal (\boldsymbol{\mu}_\alpha, \boldsymbol{\Sigma}_\alpha) $$

Where $\boldsymbol{\Sigma}_\alpha$ is a covariance matrix capturing correlations between con intercepts.

### Trading Application

- **Pairs trading**: If WTI and Brent have correlation $0.95$, large deviations signal arbitraç
- **Portfolio construction**: Use correlation matrix for optimal hedging
- **Risk management**: Diversification benefits depend on cross-commodity correlations

In [ ]:
```
# Generate correlated commodity returns
np.random.seed(42)
T_corr = 200

# True correlation matrix
# WTI-Brent: 0.95 (highly correlated)
# WTI-NatGas: 0.60
# Brent-NatGas: 0.55
true_corr = np.array([
    [1.00, 0.95, 0.60],
    [0.95, 1.00, 0.55],
    [0.60, 0.55, 1.00]
])

# Standard deviations
true_stds = np.array([2.0, 2.1, 3.5])  # NatGas more volatile

# Covariance matrix
true_cov = np.outer(true_stds, true_stds) * true_corr

# Generate multivariate normal returns
```

```
mean_returns = np.array([0.05, 0.06, 0.03])  # Daily returns (%)
returns = np.random.multivariate_normal(mean_returns, true_cov, T_corr)

# Convert to prices
prices_corr = np.zeros((T_corr, n_commodities))
prices_corr[0] = [60, 64, 56]
for t in range(1, T_corr):
    prices_corr[t] = prices_corr[t-1] * (1 + returns[t] / 100)

df_corr = pd.DataFrame(prices_corr, columns=commodities)

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

# Price series
for j, commodity in enumerate(commodities):
    axes[0, 0].plot(df_corr[commodity], label=commodity, linewidth=1.5, c
axes[0, 0].set_xlabel('Time (days)', fontsize=11)
axes[0, 0].set_ylabel('Price ($/barrel)', fontsize=11)
axes[0, 0].set_title('Correlated Commodity Prices', fontsize=12, fontweig
axes[0, 0].legend()
axes[0, 0].grid(True, alpha=0.3)

# Scatter: WTI vs Brent
axes[0, 1].scatter(df_corr['WTI'], df_corr['Brent'], alpha=0.5, s=20)
axes[0, 1].set_xlabel('WTI Price', fontsize=11)
axes[0, 1].set_ylabel('Brent Price', fontsize=11)
axes[0, 1].set_title(f'WTI vs Brent (ρ = {true_corr[0, 1]:.2f})', fontsiz
axes[0, 1].grid(True, alpha=0.3)

# Scatter: WTI vs NatGas
axes[1, 0].scatter(df_corr['WTI'], df_corr['NatGas'], alpha=0.5, s=20, co
axes[1, 0].set_xlabel('WTI Price', fontsize=11)
axes[1, 0].set_ylabel('NatGas Price', fontsize=11)
axes[1, 0].set_title(f'WTI vs NatGas (ρ = {true_corr[0, 2]:.2f})', fontsi
axes[1, 0].grid(True, alpha=0.3)

# Correlation heatmap
im = axes[1, 1].imshow(true_corr, cmap='RdYlGn', vmin=-1, vmax=1)
axes[1, 1].set_xticks(range(n_commodities))
axes[1, 1].set_yticks(range(n_commodities))
axes[1, 1].set_xticklabels(commodities)
axes[1, 1].set_yticklabels(commodities)
axes[1, 1].set_title('True Correlation Matrix', fontsize=12, fontweight='
for i in range(n_commodities):
    for j in range(n_commodities):
        axes[1, 1].text(j, i, f'{true_corr[i, j]:.2f}', ha='center', va='
plt.colorbar(im, ax=axes[1, 1])

plt.tight_layout()
plt.show()

print("Sample correlation matrix:")
print(df_corr.corr())
```

In [ ]:
```
# Build hierarchical model with correlated effects
# For simplicity, we'll estimate correlations from returns rather than fu

# Calculate returns
returns_df = df_corr.pct_change().dropna() * 100  # Percentage returns
```

```
# Stack data for PyMC
returns_stacked = returns_df.values.flatten()
commodity_idx_corr = np.repeat(np.arange(n_commodities), len(returns_df))

with pm.Model() as corr_model:
    # Mean returns by commodity
    mu_return = pm.Normal('mu_return', mu=0, sigma=1, shape=n_commodities

    # Cholesky decomposition for covariance
    sd_dist = pm.HalfNormal.dist(sigma=5, shape=n_commodities)
    chol, corr, stds = pm.LKJCholeskyCov('chol', n=n_commodities, eta=2.0

    # Observation model (simplified: independent given parameters)
    # In practice, you'd use multivariate normal
    sigma_obs = pm.Deterministic('sigma_obs', stds)

    # For computational efficiency, use independent normal per commodity
    # (Full multivariate model would be pm.MvNormal)
    y = pm.Normal('y', mu=mu_return[commodity_idx_corr],
                  sigma=sigma_obs[commodity_idx_corr],
                  observed=returns_stacked)

    # Sample
    trace_corr = pm.sample(2000, tune=1000, return_inferencedata=True,
                           target_accept=0.95, random_seed=42, chains=2)

print("\nPosterior Summary:")
print(az.summary(trace_corr, var_names=['mu_return', 'sigma_obs', 'corr']
```

In [ ]:
```
# Extract estimated correlation matrix
corr_est = trace_corr.posterior['corr'].mean(dim=['chain', 'draw']).value

# Visualize comparison
fig, axes = plt.subplots(1, 2, figsize=(14, 6))

# True correlations
im1 = axes[0].imshow(true_corr, cmap='RdYlGn', vmin=-1, vmax=1)
axes[0].set_xticks(range(n_commodities))
axes[0].set_yticks(range(n_commodities))
axes[0].set_xticklabels(commodities)
axes[0].set_yticklabels(commodities)
axes[0].set_title('True Correlations', fontsize=12, fontweight='bold')
for i in range(n_commodities):
    for j in range(n_commodities):
        axes[0].text(j, i, f'{true_corr[i, j]:.2f}', ha='center', va='cen
plt.colorbar(im1, ax=axes[0])

# Estimated correlations
im2 = axes[1].imshow(corr_est, cmap='RdYlGn', vmin=-1, vmax=1)
axes[1].set_xticks(range(n_commodities))
axes[1].set_yticks(range(n_commodities))
axes[1].set_xticklabels(commodities)
axes[1].set_yticklabels(commodities)
axes[1].set_title('Estimated Correlations (Bayesian)', fontsize=12, fontw
for i in range(n_commodities):
    for j in range(n_commodities):
        axes[1].text(j, i, f'{corr_est[i, j]:.2f}', ha='center', va='cent
plt.colorbar(im2, ax=axes[1])
plt.tight_layout()
plt.show()

print("\n=== TRADING INSIGHTS ===")
print(f"\nWTI-Brent Correlation: {corr_est[0, 1]:.3f}")
if corr_est[0, 1] > 0.9:
    print("  → HIGHLY CORRELATED: WTI-Brent spread trades require small d
    print("     Strategy: Mean-reversion on spread, tight stop-loss")

print(f"\nWTI-NatGas Correlation: {corr_est[0, 2]:.3f}")
print(f"Brent-NatGas Correlation: {corr_est[1, 2]:.3f}")
if corr_est[0, 2] < 0.7:
    print("  → MODERATE CORRELATION: NatGas provides diversification")
    print("     Strategy: Use NatGas to hedge crude oil exposure (imperfe
```

## 5 . Practical Application: Agricultural Commodities with Shared Weather

### Trading Context

Corn, Wheat, and Soybeans:

- **Compete for farmland**: Farmers allocate acres based on expected profits
- **Share weather risks**: Drought in the Midwest affects all three
- **Different markets**: Corn (feed, ethanol), Wheat (food), Soybeans (feed, oil)

We'll model:

- Varying intercepts (different base prices)
- Varying slopes for rainfall impact
- Shared seasonality (harvest depression)

In [ ]:
```
# Generate agricultural commodity data
np.random.seed(42)
T_ag = 120  # 10 years of monthly data

ag_commodities = ['Corn', 'Wheat', 'Soybeans']
n_ag = len(ag_commodities)

# Rainfall index (standardized)
rainfall = np.random.normal(0, 1, T_ag)

# Month of year (for seasonality)
month = np.arange(T_ag) % 12

# Harvest months: September-October (months 8-9)
harvest_effect = np.where((month == 8) | (month == 9), -20, 0)

# True parameters
true_alpha_ag = np.array([400, 600, 1100])  # Base prices (cents/bushel)
true_beta_rain = np.array([-15, -12, -18])  # Rainfall effect (drought →

# Generate prices
prices_ag = np.zeros((T_ag, n_ag))
for j in range(n_ag):
    trend = true_alpha_ag[j] + 0.5 * np.arange(T_ag)  # Slight uptrend
    prices_ag[:, j] = (trend +
                       true_beta_rain[j] * rainfall +
                       harvest_effect +
                       np.random.normal(0, 25, T_ag))

df_ag = pd.DataFrame(prices_ag, columns=ag_commodities)
df_ag['rainfall'] = rainfall
df_ag['month'] = month
df_ag['time'] = np.arange(T_ag)

# Visualize
fig, axes = plt.subplots(3, 1, figsize=(14, 10), sharex=True)

# Prices
for commodity in ag_commodities:
    axes[0].plot(df_ag['time'], df_ag[commodity], label=commodity, linewi
axes[0].set_ylabel('Price (cents/bushel)', fontsize=11)
axes[0].set_title('Agricultural Commodity Prices', fontsize=12, fontweigh
axes[0].legend()
axes[0].grid(True, alpha=0.3)

# Rainfall
axes[1].bar(df_ag['time'], df_ag['rainfall'], alpha=0.6, color='blue')
axes[1].axhline(0, color='black', linestyle='--')
axes[1].set_ylabel('Rainfall Index', fontsize=11)
axes[1].set_title('Rainfall (negative = drought)', fontsize=12, fontweigh
axes[1].grid(True, alpha=0.3, axis='y')

# Harvest seasonality
axes[2].bar(df_ag['time'], harvest_effect, alpha=0.6, color='orange')
axes[2].set_xlabel('Time (months)', fontsize=11)
axes[2].set_ylabel('Harvest Effect (cents)', fontsize=11)
axes[2].set_title('Seasonal Harvest Pressure', fontsize=12, fontweight='b
axes[2].grid(True, alpha=0.3, axis='y')

plt.tight_layout()
plt.show()
```

```
In [ ]: # Prepare data for modeling
# Stack prices
prices_stacked = df_ag[ag_commodities].values.flatten()
rainfall_stacked = np.tile(df_ag['rainfall'].values, n_ag)
month_stacked = np.tile(df_ag['month'].values, n_ag)
commodity_idx_ag = np.repeat(np.arange(n_ag), T_ag)

# Build hierarchical model
with pm.Model() as ag_model:
    # Hyperpriors for base prices
    mu_alpha = pm.Normal('mu_alpha', mu=700, sigma=500)
    tau_alpha = pm.HalfNormal('tau_alpha', sigma=500)

    # Hyperpriors for rainfall sensitivity
    mu_beta_rain = pm.Normal('mu_beta_rain', mu=-15, sigma=10)
    tau_beta_rain = pm.HalfNormal('tau_beta_rain', sigma=10)

    # Commodity-specific parameters
    alpha = pm.Normal('alpha', mu=mu_alpha, sigma=tau_alpha, shape=n_ag)
    beta_rain = pm.Normal('beta_rain', mu=mu_beta_rain, sigma=tau_beta_ra

    # Shared harvest seasonality (2 months)
    harvest_sep = pm.Normal('harvest_sep', mu=-20, sigma=10)  # September
    harvest_oct = pm.Normal('harvest_oct', mu=-20, sigma=10)  # October

    # Map harvest effects to months
    harvest_effects = pm.math.switch(
        pm.math.eq(month_stacked, 8), harvest_sep,
        pm.math.switch(pm.math.eq(month_stacked, 9), harvest_oct, 0)
    )

    # Observation noise
    sigma_y = pm.HalfNormal('sigma_y', sigma=50)

    # Combined mean
    mu = (alpha[commodity_idx_ag] +
          beta_rain[commodity_idx_ag] * rainfall_stacked +
          harvest_effects)

    # Likelihood
    y = pm.Normal('y', mu=mu, sigma=sigma_y, observed=prices_stacked)

    # Sample
    trace_ag = pm.sample(2000, tune=1000, return_inferencedata=True,
                         target_accept=0.95, random_seed=42, chains=2)

print("\nPosterior Summary:")
print(az.summary(trace_ag, var_names=['mu_alpha', 'tau_alpha', 'mu_beta_r
                                      'alpha', 'beta_rain', 'harvest_sep
```

In [ ]:
```
# Extract estimates
alpha_ag_est = trace_ag.posterior['alpha'].mean(dim=['chain', 'draw']).va
beta_rain_est = trace_ag.posterior['beta_rain'].mean(dim=['chain', 'draw'
harvest_sep_est = trace_ag.posterior['harvest_sep'].mean().item()
harvest_oct_est = trace_ag.posterior['harvest_oct'].mean().item()

# Comparison
fig, axes = plt.subplots(1, 2, figsize=(14, 6))

# Base prices
x_pos = np.arange(n_ag)
axes[0].bar(x_pos - 0.2, true_alpha_ag, width=0.4, label='True', alpha=0.
axes[0].bar(x_pos + 0.2, alpha_ag_est, width=0.4, label='Estimated', alph
axes[0].set_xticks(x_pos)
axes[0].set_xticklabels(ag_commodities)
axes[0].set_ylabel('Base Price (cents/bushel)', fontsize=11)
axes[0].set_title('Commodity Base Prices (α)', fontsize=12, fontweight='b
axes[0].legend()
axes[0].grid(True, alpha=0.3, axis='y')

# Rainfall sensitivity
axes[1].bar(x_pos - 0.2, true_beta_rain, width=0.4, label='True', alpha=0
axes[1].bar(x_pos + 0.2, beta_rain_est, width=0.4, label='Estimated', alp
axes[1].axhline(0, color='black', linestyle='--')
axes[1].set_xticks(x_pos)
axes[1].set_xticklabels(ag_commodities)
axes[1].set_ylabel('Rainfall Sensitivity (β)', fontsize=11)
axes[1].set_title('Drought Impact (negative rainfall → higher prices)', f
axes[1].legend()
axes[1].grid(True, alpha=0.3, axis='y')

plt.tight_layout()
plt.show()

# Trading dashboard
print("\n" + "="*70)
print("AGRICULTURAL COMMODITY TRADING DASHBOARD")
print("="*70)

for j, commodity in enumerate(ag_commodities):
    beta_samples = trace_ag.posterior['beta_rain'].values[:, :, j].flatte
    prob_drought_bullish = np.mean(beta_samples < 0)

    print(f"\n{commodity}:")
    print(f"  Base price: {alpha_ag_est[j]:.1f} cents/bushel")
    print(f"  Rainfall β: {beta_rain_est[j]:.2f} (95% CI: [{np.percentile
    print(f"  Prob(drought increases prices): {prob_drought_bullish:.2%}"

    if abs(beta_rain_est[j]) > 15:
        print(f"  → HIGH DROUGHT SENSITIVITY: Weather derivatives highly
    else:
        print(f"  → MODERATE DROUGHT SENSITIVITY")

print(f"\n\nShared Harvest Effects:")
print(f"  September: {harvest_sep_est:.1f} cents (harvest pressure)")
print(f"  October: {harvest_oct_est:.1f} cents (harvest pressure)")
print(f"\n  → Strategy: Short ag futures ahead of harvest, cover in Novem

print("\n" + "="*70)
```

## Knowledge Check Quiz

### Question 1

What is the main advantage of partial pooling over no pooling?

A) Faster computation

B) Borrows strength across groups, reducing overfitting

C) Assumes all groups are identical

D) Eliminates the need for priors

**Answer: B** - Partial pooling shares information between related groups while allowing group-spe parameters, reducing overfitting especially with small samples.

---

### Question 2

When does shrinkage have the strongest effect in hierarchical models?

A) Large sample sizes

B) Small sample sizes

C) Equal sample sizes

D) Shrinkage is always the same

**Answer: B** - Groups with small samples are shrunk more toward the group mean, borrowing stre from better-estimated groups.

---

## Question 3

What does a varying slopes model estimate?

A) Only intercepts differ by group
B) Only slopes differ by group
C) Both intercepts and slopes can differ by group
D) All parameters are identical

**Answer: C** - Varying slopes models allow both intercepts (base levels) and slopes (sensitivities) across groups.

---

## Question 4

Why is modeling cross-commodity correlations important for trading?

A) It makes models more complex
B) It's required by regulators
C) For portfolio construction, hedging, and pairs trading
D) It eliminates all risk

**Answer: C** - Correlations determine diversification benefits, hedging effectiveness, and spread t opportunities.

---

## Question 5

In the agricultural model, what does a negative β (rainfall) coefficient indicate?

A) More rainfall increases prices
B) Drought (low rainfall) increases prices
C) Rainfall has no effect
D) The model is wrong

**Answer: B** - Negative coefficient means low rainfall (drought) is associated with higher prices du reduced yields.

---

## Exercises

### Exercise 1 : Energy Spread Trading

Using the WTI-Brent correlation model:

1. Define the spread as: Brent - WTI
2. Model the spread as a mean-reverting process
3. Calculate the equilibrium spread (mean)
4. Design a pairs trading rule: long spread when 2 σ below mean, short when 2 σ above

5. Backtest on the generated data

## Exercise 2: Portfolio Optimization

Given estimated correlations for energy commodities:

1. Assume equal expected returns
2. Use correlation matrix to compute minimum variance portfolio weights
3. Compare diversified portfolio volatility to single-commodity volatility
4. Calculate diversification benefit

## Exercise 3: Drought Scenario Analysis

For agricultural commodities:

1. Simulate a severe drought: rainfall = -2 (2 std below normal) for 6 months
2. Forecast price impact for each commodity
3. Which commodity benefits most from drought?
4. Calculate portfolio P&L if long all three commodities equally

## Exercise 4: Hierarchical Volatility Model

Build a model where volatility (σ) varies by commodity: $$ \sigma_j \sim \text{HalfNormal}(\mu_\
\tau_\sigma) $$

1. Estimate commodity-specific volatilities
2. Rank commodities by volatility
3. Design position sizing rule: allocate inversely to volatility
4. Compare to equal-weighted portfolio

## Exercise 5: Forecasting with Missing Data

Simulate missing data scenario:

1. Remove last 30 days of Soybeans prices
2. Re-estimate hierarchical model using only Corn and Wheat
3. Use group-level parameters to forecast Soybeans
4. Compare forecast accuracy to model fit on complete data
5. Quantify information borrowing benefit

---

## Summary

In this module, you learned:

1. **Pooling Strategies**: Complete, none, and partial pooling for multi-group data
2. **Shrinkage**: How hierarchical models regularize estimates toward group means
3. **Varying Intercepts & Slopes**: Commodity-specific parameters with shared priors
4. **Cross-Commodity Correlations**: Modeling dependencies for portfolio decisions

5. **Agricultural Application**: Weather impacts across related commodities

## Key Takeaways for Trading

- **Information Borrowing**: Improve estimates for thinly-traded commodities
- **Regularization**: Prevent overfitting via shrinkage to group mean
- **Pairs Trading**: Identify relative mispricings within commodity complexes
- **Risk Management**: Model correlations for optimal portfolio construction
- **Cross-Sectional Analysis**: Which commodity is most/least sensitive to shared drivers?
- **Scenario Analysis**: Forecast impact of shocks (drought, OPEC cuts) across complex

Hierarchical models are essential when:

- You have **multiple related assets** (energy complex, grains, metals)
- **Sample sizes vary** (some commodities thinly traded)
- You want to **share information** without assuming homogeneity
- **Correlations matter** for hedging and diversification

This framework scales from 3 commodities (as shown) to dozens, enabling systematic cro sectional strategies.

---

## Course Summary & Next Steps

### What You've Accomplished

Across Modules 1 - 7, you've built a complete Bayesian toolkit for commodity trading:

1. **Foundations**: Bayesian inference, prior/posterior, MCMC
2. **Prior Selection**: Informative, weakly informative, and regularizing priors
3. **MCMC Inference**: Sampling, convergence diagnostics, trace analysis
4. **Time Series**: Autocorrelation, stationarity, seasonality
5. **Bayesian Linear Regression**: Uncertainty quantification, robust models
6. **BSTS**: Trend, seasonality, dynamic regression, state-space models
7. **Hierarchical Models**: Multi-commodity analysis, shrinkage, correlations

### Remaining Modules (8 - 10)

- **Module 8: Gaussian Processes** - Non-parametric flexible curves for price discovery
- **Module 9: Volatility Modeling** - GARCH, stochastic volatility, VIX forecasting
- **Module 10: Trading Strategies** - Complete systematic strategies with backtesting

### Practice Recommendations

1. **Real data**: Apply to actual commodity prices (FRED, Quandl, Yahoo Finance)
2. **Daily practice**: Build one small model per day
3. **Paper trading**: Test forecasts in real-time before risking capital
4. **Community**: Share models, get feedback, iterate

## Resources

- PyMC Documentation: https://www.pymc.io/
- ArviZ Gallery: https://arviz-devs.github.io/arviz/
- Statistical Rethinking (McElreath): Excellent Bayesian textbook
- BDA 3 (Gelman et al.): Bayesian Data Analysis reference

---

**Congratulations on completing Module 7! You now have the statistical tools to build sophisticated multi-commodity trading models. Keep practicing, and see you in Module**

# 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 $\{x_ 1 , ..., x_n\}$, the function values follow a multivariate normal:

$$\begin{bmatrix} f(x_ 1 ) \\ \vdots \\ f(x_n) \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} m(
\vdots \\ m(x_n) \end{bmatrix}, \begin{bmatrix} k(x_ 1 ,x_ 1 ) & \cdots & k(x_ 1 ,x_n) \\ \vdots & \
\vdots \\ k(x_n,x_ 1 ) & \cdots & k(x_n,x_n) \end{bmatrix}\right)$$

## 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 [ ]:
```python
# 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

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

**Parameters**:

- $\nu$ (smoothness): Controls differentiability
  - $\nu = 0.5$: Rough (exponential kernel)
  - $\nu = 1.5$: Once differentiable
  - $\nu = 2.5$: Twice differentiable
  - $\nu \to \infty$: 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
                return amplitude**2 * np.exp(-2 * np.sin(np.pi * np.abs(x - x_ref

            # 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'ℓ = {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='ν = 0.5 (rough)', linewidth=2)
            ax.plot(x, matern_kernel(x, x_ref, length_scale=1.0), label='ν = 1.5
            ax.plot(x, rbf_kernel(x, x_ref, length_scale=1.0), label='ν → ∞ (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'Peri
            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 ν → 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**:

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

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

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

where:

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

**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

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

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=['chai
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 mea
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', z
ax.set_xlabel('Time', fontsize=12)
ax.set_ylabel('Price ($)', fontsize=12)
ax.set_title('Gaussian Process Regression: Non-Linear Price Trend', fonts
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, f
ax.legend()
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Print summary
print("\n" + "="*70)
print("HYPERPARAMETER POSTERIORS")
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, l

    # 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, pro

# 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_nam
                                                      random_seed=42, pr

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

In [ ]:

```python
# 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
f_std_gas = pred_samples_gas.posterior_predictive['f_pred_gas'].std(dim=[

# 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='Fo
ax.set_xlabel('Time (years)', fontsize=12)
ax.set_ylabel('Price ($/MMBtu)', fontsize=12)
ax.set_title('Natural Gas: Composite Kernel (Trend + Seasonality)', fonts
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
    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 mode
ax.plot(t_pred_gas, f_trend_mean, 'green', linewidth=2, label='Trend comp
ax.plot(t_pred_gas, f_mean_gas - f_trend_mean + np.mean(price_gas), 'oran
        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, fontweigh
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 [ ]:
```python
# Analyze hyperparameter posteriors from composite model
fig, axes = plt.subplots(2, 3, figsize=(16, 9))

params = ['ℓ_trend', 'η_trend', 'ℓ_seasonal', 'η_seasonal', 'σ']
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, lab

    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)
η_trend_mean = np.mean(trace_comp.posterior['η_trend'].values)
η_seasonal_mean = np.mean(trace_comp.posterior['η_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): {η_trend_mean/η_seasonal_mean:.2f}
  → Trend is {η_trend_mean/η_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 > 1 0 0 0 $ 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 $Z$ that summariz
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(n³) = O({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, pro

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(di
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, labe
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
             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', linew
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, labe
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(n³) = O({len(t_copper)**3/1e9:.1f}B) operations
   - Sparse GP: O(nm²) = O({len(t_copper)*m**2/1e6:.1f}M) operations
   - Speedup: ~{len(t_copper)**2 / m**2:.0f}x

2. **Memory Savings**:
   - Full GP: {len(t_copper)**2 * 8 / 1e9:.2f} GB
   - Sparse GP: {len(t_copper) * m * 8 / 1e6:.2f} MB
   - Reduction: ~{len(t_copper) / m:.0f}x

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 calibra

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 st

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 kerne |
| **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

1. **Kernels encode assumptions**: Choose kernels that match your beliefs about price dynami
2. **Combine kernels**: Real commodities have multiple patterns (trend + seasonality + noise)
3. **Hyperparameters matter**: Length scales, amplitudes have economic interpretations
4. **Sparse GPs scale**: Use inducing points for large datasets
5. **Uncertainty is valuable**: GP predictive distributions enable sophisticated risk management
6. **Matérn > RBF for trading**: Finite differentiability better matches real price dynamics
7. **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 v
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 appropri

- 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 ℓ controls the "reach" of correlation. Larger ℓ 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(nm²)
    Sparse GPs reduce complexity from O(n³) to O(nm²) where m << 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 wit
    ν=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 → 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 + 1σ
    - Go short if forecast mean < current price - 1σ
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*

# Module 9: Volatility Modeling and Uncertai Quantification

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

---

## Learning Objectives

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

1. **Distinguish** between epistemic (model) and aleatoric (irreducible) uncertainty
2. **Implement** GARCH models in a Bayesian framework for volatility clustering
3. **Build** stochastic volatility models with PyMC for time-varying risk
4. **Calculate** Value at Risk (VaR) and Conditional VaR (CVaR) from posterior predictive distribu
5. **Forecast** volatility regimes for risk management and position sizing
6. **Apply** Bayesian volatility models to crude oil for trading decisions

---

## Why This Matters for Trading

**Volatility is not constant**—and ignoring this can be catastrophic for traders:

### Real-World Examples

- **Crude oil (2020)**: Volatility spiked 10x during COVID, wiping out strategies calib "normal" volatility
- **Natural gas (Winter Storm Uri, 2021)**: Prices jumped 100x in days—fixed v models failed completely
- **Copper (2008)**: Volatility doubled during financial crisis; traders with constant-vol ass lost fortunes

### Why Bayesian Volatility Modeling?

1. **Time-varying risk**: Volatility clusters (high vol follows high vol)
2. **Regime detection**: Identify when markets shift from calm to turbulent
3. **Option pricing**: Implied vol forecasts drive delta hedging and gamma trading
4. **Position sizing**: Scale positions by forecasted volatility (Kelly criterion)
5. **Risk management**: VaR/CVaR for regulatory compliance and internal limits
6. **Uncertainty decomposition**: Separate "we don't know the model" from "markets are rando

**Bottom line**: Accurate volatility forecasts = better risk-adjusted returns.

---

## 1. Two Types of Uncertainty

Before modeling volatility, we must understand **what we're uncertain about**.

## 1.1 Aleatoric Uncertainty (Irreducible)

**Definition**: Randomness inherent in the system.

- **Also called**: Statistical uncertainty, data uncertainty
- **Source**: Markets are fundamentally stochastic
- **Cannot be reduced**: Even with infinite data, prices are still random

**Example**:

- Crude oil price tomorrow is uncertain because of random supply/demand shocks
- No amount of historical data will make tomorrow's price deterministic

**Mathematical form**: $$y_t = \mu_t + \epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, \sigma_t^

The $\epsilon_t$ term captures aleatoric uncertainty.

## 1.2 Epistemic Uncertainty (Reducible)

**Definition**: Uncertainty about the model parameters.

- **Also called**: Model uncertainty, parameter uncertainty
- **Source**: Limited data, model misspecification
- **Can be reduced**: More data → tighter posterior → less epistemic uncertainty

**Example**:

- We're uncertain whether crude oil volatility is 20% or 30% annualized
- More historical data narrows our belief about true volatility

**Mathematical form**: $$\sigma \sim p(\sigma | \text{data})$$

The posterior distribution $p(\sigma | \text{data})$ captures epistemic uncertainty.

## 1.3 Total Predictive Uncertainty

**Bayesian forecasts combine both**:

$$p(y_{\text{future}} | \text{data}) = \int p(y_{\text{future}} | \theta) \cdot p(\theta | \text{data}) d\th

- **Inner term** $p(y_{\text{future}} | \theta)$: Aleatoric (data randomness given parameters)
- **Outer term** $p(\theta | \text{data})$: Epistemic (parameter uncertainty)
- **Integral**: Marginalizes over parameter uncertainty

**Trading implication**:

- **Short-term forecasts**: Dominated by aleatoric uncertainty (market randomness)
- **Long-term forecasts**: Dominated by epistemic uncertainty (don't know true parameters)
- **Position sizing**: Use total uncertainty (both sources matter)

```
In [ ]: # Setup
        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')

        np.random.seed(42)
        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__}")
```

```
In [ ]: # Demonstrate epistemic vs aleatoric uncertainty
        def demonstrate_uncertainty_types():
            """
            Visualize epistemic vs aleatoric uncertainty with simple example.
            """
            # Generate data from known process
            true_mu = 100
            true_sigma = 10

            # Two scenarios: small vs large dataset
            n_small = 10
            n_large = 500

            np.random.seed(42)
            data_small = np.random.normal(true_mu, true_sigma, n_small)
            data_large = np.random.normal(true_mu, true_sigma, n_large)

            # Bayesian inference on mean and std
            def infer_params(data):
                with pm.Model() as model:
                    mu = pm.Normal('mu', mu=100, sigma=20)
                    sigma = pm.HalfNormal('sigma', sigma=15)
                    y = pm.Normal('y', mu=mu, sigma=sigma, observed=data)
                    trace = pm.sample(1000, tune=1000, chains=2, random_seed=42,
                return trace

            trace_small = infer_params(data_small)
            trace_large = infer_params(data_large)

            # Posterior predictive
            def get_predictions(trace, n_pred=1000):
                mu_samples = trace.posterior['mu'].values.flatten()
                sigma_samples = trace.posterior['sigma'].values.flatten()

                # Sample predictions
                n_samples = len(mu_samples)
                predictions = np.zeros((n_samples, n_pred))
                for i in range(n_samples):
                    predictions[i, :] = np.random.normal(mu_samples[i], sigma_sam

                return predictions.flatten()
```

```
pred_small = get_predictions(trace_small)
pred_large = get_predictions(trace_large)

# Plot
fig, axes = plt.subplots(2, 2, figsize=(15, 10))

# Small data: parameter uncertainty
ax = axes[0, 0]
ax.hist(trace_small.posterior['mu'].values.flatten(), bins=30, alpha=
        density=True, color='orange', label=f'n={n_small}')
ax.axvline(true_mu, color='red', linestyle='--', linewidth=2, label='
ax.set_xlabel('Mean (μ)')
ax.set_ylabel('Density')
ax.set_title('Epistemic Uncertainty: Small Dataset', fontweight='bold
ax.legend()
ax.grid(True, alpha=0.3)

# Large data: parameter uncertainty
ax = axes[0, 1]
ax.hist(trace_large.posterior['mu'].values.flatten(), bins=30, alpha=
        density=True, color='green', label=f'n={n_large}')
ax.axvline(true_mu, color='red', linestyle='--', linewidth=2, label='
ax.set_xlabel('Mean (μ)')
ax.set_ylabel('Density')
ax.set_title('Epistemic Uncertainty: Large Dataset', fontweight='bold
ax.legend()
ax.grid(True, alpha=0.3)

# Small data: predictive distribution
ax = axes[1, 0]
ax.hist(pred_small, bins=50, alpha=0.7, density=True, color='orange',
ax.hist(np.random.normal(true_mu, true_sigma, 10000), bins=50, alpha=
        density=True, color='blue', label='True distribution')
ax.set_xlabel('Predicted Value')
ax.set_ylabel('Density')
ax.set_title('Predictive Uncertainty: Small Dataset', fontweight='bol
ax.legend()
ax.grid(True, alpha=0.3)

# Large data: predictive distribution
ax = axes[1, 1]
ax.hist(pred_large, bins=50, alpha=0.7, density=True, color='green',
ax.hist(np.random.normal(true_mu, true_sigma, 10000), bins=50, alpha=
        density=True, color='blue', label='True distribution')
ax.set_xlabel('Predicted Value')
ax.set_ylabel('Density')
ax.set_title('Predictive Uncertainty: Large Dataset', fontweight='bol
ax.legend()
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Quantify uncertainties
epistemic_small = np.std(trace_small.posterior['mu'].values)
epistemic_large = np.std(trace_large.posterior['mu'].values)
aleatoric_small = np.mean(trace_small.posterior['sigma'].values)
aleatoric_large = np.mean(trace_large.posterior['sigma'].values)
```

```
    print("\n" + "="*70)
    print("UNCERTAINTY QUANTIFICATION")
    print("="*70)
    print(f"\nSmall Dataset (n={n_small}):")
    print(f"  Epistemic (parameter uncertainty): σ_μ = {epistemic_small:.
    print(f"  Aleatoric (data randomness):        σ = {aleatoric_small:.2
    print(f"  Total predictive std:               {np.std(pred_small):.2f
    print(f"\nLarge Dataset (n={n_large}):")
    print(f"  Epistemic (parameter uncertainty): σ_μ = {epistemic_large:.
    print(f"  Aleatoric (data randomness):        σ = {aleatoric_large:.2
    print(f"  Total predictive std:               {np.std(pred_large):.2f

    print("\n" + "="*70)
    print("KEY INSIGHTS")
    print("="*70)
    print(f"""
1. Epistemic uncertainty DECREASES with more data:
   - Small dataset: σ_μ = {epistemic_small:.2f}
   - Large dataset: σ_μ = {epistemic_large:.2f}
   - Reduction: {epistemic_small/epistemic_large:.1f}x

2. Aleatoric uncertainty STAYS CONSTANT:
   - Small dataset: σ = {aleatoric_small:.2f}
   - Large dataset: σ = {aleatoric_large:.2f}
   - This is irreducible market randomness!

3. Predictive uncertainty converges to aleatoric:
   - Small data: wider (epistemic + aleatoric)
   - Large data: narrower (mostly aleatoric)

**Trading Application**:
- New commodity with limited history → High epistemic uncertainty
  → Use wider stop-losses, smaller positions
- Mature commodity with decades of data → Low epistemic uncertainty
  → Can size positions more aggressively
    """)

demonstrate_uncertainty_types()
```

## 2 . GARCH Models: Volatility Clustering

### 2 . 1 The Stylized Fact: Volatility Clusters

**Observation**: Large price changes tend to follow large price changes.

- Mandelbrot ( 1 9 6 3 ): "Large changes tend to be followed by large changes—of either s small changes tend to be followed by small changes."
- This violates the constant variance assumption of standard regression

### 2 . 2 GARCH( 1 , 1 ) Model

**Generalized Autoregressive Conditional Heteroskedasticity**

$$\begin{align} r_t &= \mu + \epsilon_t \quad \text{(return equation)} \\ \epsilon_t &= \sigma_t z_t z_t \sim \mathcal{N}( 0 , 1 ) \quad \text{(standardized shock)} \\ \sigma_t^ 2 &= \omega + \epsilon_{t- 1 }^ 2 + \beta \sigma_{t- 1 }^ 2 \quad \text{(variance equation)} \end{align}$$

**Parameters**:

- $\omega > 0$: Baseline variance
- $\alpha \geq 0$: Reaction to shocks (ARCH effect)
- $\beta \geq 0$: Persistence of volatility (GARCH effect)
- **Stationarity condition**: $\alpha + \beta < 1$

**Interpretation**:

- $\alpha$ high: Volatility reacts strongly to recent shocks
- $\beta$ high: Volatility shocks persist for long time
- $\alpha + \beta \approx 1$: Volatility shocks are nearly permanent (integrated GARCH)

## 2.3 Bayesian GARCH vs Frequentist MLE

**Frequentist GARCH**:

- Maximum likelihood estimation
- Point estimates for $\alpha, \beta, \omega$
- No parameter uncertainty

**Bayesian GARCH**:

- Full posterior distributions for parameters
- Propagate parameter uncertainty to volatility forecasts
- Natural shrinkage through priors
- Can incorporate expert beliefs about volatility persistence

In [ ]:
```python
# Simulate GARCH(1,1) process
def simulate_garch(n=500, omega=0.1, alpha=0.15, beta=0.8, mu=0.0):
    """
    Simulate GARCH(1,1) returns.
    """
    returns = np.zeros(n)
    sigma2 = np.zeros(n)
    sigma2[0] = omega / (1 - alpha - beta)  # Unconditional variance

    for t in range(1, n):
        # Variance equation
        sigma2[t] = omega + alpha * returns[t-1]**2 + beta * sigma2[t-1]

        # Return equation
        returns[t] = mu + np.sqrt(sigma2[t]) * np.random.randn()

    return returns, np.sqrt(sigma2)

# Generate synthetic crude oil returns
np.random.seed(42)
returns, true_vol = simulate_garch(n=500, omega=0.05, alpha=0.12, beta=0.

# Convert to prices
prices = 70 * np.exp(np.cumsum(returns))

# Visualize
```

```
fig, axes = plt.subplots(3, 1, figsize=(14, 10))

# Prices
ax = axes[0]
ax.plot(prices, linewidth=1.5, color='black')
ax.set_ylabel('Price ($/barrel)', fontsize=11)
ax.set_title('Simulated Crude Oil Prices (GARCH volatility)', fontsize=12
ax.grid(True, alpha=0.3)

# Returns
ax = axes[1]
ax.plot(returns, linewidth=1, color='blue', alpha=0.7)
ax.axhline(0, color='red', linestyle='--', linewidth=1)
ax.set_ylabel('Returns', fontsize=11)
ax.set_title('Returns (showing volatility clustering)', fontsize=12, font
ax.grid(True, alpha=0.3)

# Volatility
ax = axes[2]
ax.plot(true_vol, linewidth=2, color='red', label='True volatility (σ_t)'
# Rolling realized volatility (for comparison)
rolling_vol = pd.Series(returns).rolling(window=20).std() * np.sqrt(252)
ax.plot(rolling_vol, linewidth=1.5, color='green', alpha=0.7, label='Real
ax.set_xlabel('Time', fontsize=11)
ax.set_ylabel('Volatility', fontsize=11)
ax.set_title('GARCH Volatility (time-varying)', fontsize=12, fontweight='
ax.legend()
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("\n" + "="*70)
print("VOLATILITY CLUSTERING EVIDENCE")
print("="*70)
print(f"""
Notice:
1. Periods of HIGH volatility (wide swings) cluster together
2. Periods of LOW volatility (calm) also cluster together
3. This is NOT captured by constant variance models!

Autocorrelation of squared returns (test for ARCH effects):
  lag-1: {np.corrcoef(returns[:-1]**2, returns[1:]**2)[0,1]:.3f}
  lag-5: {np.corrcoef(returns[:-5]**2, returns[5:]**2)[0,1]:.3f}

Positive autocorrelation in squared returns = volatility clustering!
""")
```

In [ ]:

```
# Fit Bayesian GARCH(1,1)
# Note: Full Bayesian GARCH is computationally intensive in PyMC
# We'll use a simplified approach for demonstration

def fit_bayesian_garch_simple(returns, n_samples=1000):
    """
    Simplified Bayesian GARCH using PyMC.

    For production, consider using specialized packages like
    arch (Python) with Bayesian extensions.
    """
    # Use first portion of data
```

```python
    returns_train = returns[:400]

    with pm.Model() as garch_model:
        # Priors for GARCH parameters
        omega = pm.HalfNormal('omega', sigma=0.1)
        alpha = pm.Beta('alpha', alpha=2, beta=8)  # Concentrated near 0.
        beta = pm.Beta('beta', alpha=8, beta=2)    # Concentrated near 0.

        # Initialize variance
        initial_vol = pm.HalfNormal('initial_vol', sigma=0.5)

        # GARCH recursion (using scan for efficiency)
        def garch_step(ret_prev, sigma2_prev, omega, alpha, beta):
            sigma2_new = omega + alpha * ret_prev**2 + beta * sigma2_prev
            return sigma2_new

        # Compute variance series
        sigma2_series, _ = pm.scan(
            fn=garch_step,
            sequences=[returns_train[:-1]],
            outputs_info=[initial_vol**2],
            non_sequences=[omega, alpha, beta],
            n_steps=len(returns_train)-1
        )

        # Prepend initial variance
        sigma2_all = pm.math.concatenate([[initial_vol**2], sigma2_series

        # Likelihood
        returns_obs = pm.Normal('returns_obs',
                                mu=0,
                                sigma=pm.math.sqrt(sigma2_all),
                                observed=returns_train)

        # Sample posterior
        trace = pm.sample(n_samples, tune=1000, chains=2, random_seed=42,
                          progressbar=True, target_accept=0.95)

    return trace, garch_model

print("Fitting Bayesian GARCH(1,1)...")
print("(This may take a few minutes)\n")

trace_garch, model_garch = fit_bayesian_garch_simple(returns, n_samples=5

print("\nGARCH model fitted successfully!")
```

In [ ]:
```python
# Analyze GARCH parameter posteriors
fig, axes = plt.subplots(1, 3, figsize=(15, 4))

params = ['omega', 'alpha', 'beta']
true_values = [0.05, 0.12, 0.85]
param_names = ['ω (baseline var)', 'α (ARCH)', 'β (GARCH)']

for ax, param, true_val, name in zip(axes, params, true_values, param_nam
    samples = trace_garch.posterior[param].values.flatten()

    ax.hist(samples, bins=30, alpha=0.7, density=True, color='steelblue',
    ax.axvline(true_val, color='red', linestyle='--', linewidth=2, label=
    ax.axvline(np.mean(samples), color='green', linestyle='-', linewidth=
                 label=f'Post mean = {np.mean(samples):.3f}')
    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)

plt.tight_layout()
plt.show()

# Summary statistics
print("\n" + "="*70)
print("GARCH PARAMETER POSTERIORS")
print("="*70)
print(f"{'Parameter':<15} {'True':>10} {'Post Mean':>12} {'Post Std':>12}
print("-"*70)

for param, true_val, name in zip(params, true_values, param_names):
    samples = trace_garch.posterior[param].values.flatten()
    mean_est = np.mean(samples)
    std_est = np.std(samples)
    ci = np.percentile(samples, [2.5, 97.5])
    print(f"{name:<15} {true_val:>10.3f} {mean_est:>12.3f} {std_est:>12.3

# Check persistence
alpha_samples = trace_garch.posterior['alpha'].values.flatten()
beta_samples = trace_garch.posterior['beta'].values.flatten()
persistence = alpha_samples + beta_samples

print("\n" + "="*70)
print("VOLATILITY PERSISTENCE")
print("="*70)
print(f"α + β (persistence):")
print(f"  Mean: {np.mean(persistence):.3f}")
print(f"  95% CI: [{np.percentile(persistence, 2.5):.3f}, {np.percentile(
print(f"\nInterpretation:")
print(f"  α + β = {np.mean(persistence):.3f} means volatility shocks have
print(f"  ~{-1/np.log(np.mean(persistence)):.1f} periods (days in this ca
print(f"\n  Close to 1 → volatility shocks are very persistent (typical f
print(f"  Far from 1 → volatility mean-reverts quickly")
```

# 3 . Stochastic Volatility Models

## 3 . 1 Limitations of GARCH

GARCH models have **deterministic volatility dynamics**:

- Volatility is a deterministic function of past returns
- No separate shock term for volatility itself

## 3 . 2 Stochastic Volatility (SV) Model

**Idea**: Volatility has its own random shocks.

$$\begin{align} r_t &= \exp(h_t/ 2 ) \epsilon_t, \quad \epsilon_t \sim \mathcal{N}( 0 , 1 ) \quad \text{(return equation)} \\ h_t &= \mu_h + \phi (h_{t- 1 } - \mu_h) + \sigma_h \eta_t, \quad \eta_t \ \mathcal{N}( 0 , 1 ) \quad \text{(log-vol equation)} \end{align}$$

**Parameters**:

- $h_t = \log(\sigma_t^ 2 )$: Log-variance (ensures positivity)
- $\mu_h$: Mean log-variance
- $\phi \in (- 1 ,    1 )$: Persistence of volatility
- $\sigma_h$: Volatility of volatility (vol-of-vol)

**Advantages over GARCH**:

1 . **Separate volatility shocks**: $\eta_t$ drives volatility changes
2 . **Leverage effect**: Can model correlation between $\epsilon_t$ and $\eta_t$ (negative correl leverage)
3 . **Better option pricing**: Matches implied volatility smiles
4 . **More flexible**: Captures volatility spikes not driven by returns

## 3 . 3 Bayesian SV Estimation

**Challenge**: Latent volatility $h_t$ must be inferred.

**Solution**: MCMC samples both parameters $(\mu_h, \phi, \sigma_h)$ and latent states $\{h_ 1 ,
$ jointly.

In [ ]:
```
# Fit Stochastic Volatility model
def fit_stochastic_volatility(returns, n_samples=1000):
    """
    Fit stochastic volatility model using PyMC.
    """
    returns_train = returns[:400]

    with pm.Model() as sv_model:
        # Hyperparameters
        mu_h = pm.Normal('mu_h', mu=-3, sigma=2)  # Mean log-volatility
        phi = pm.Uniform('phi', lower=-0.999, upper=0.999)  # AR(1) persi
        sigma_h = pm.HalfNormal('sigma_h', sigma=0.5)  # Vol-of-vol

        # Initial log-volatility
        h_init = pm.Normal('h_init', mu=mu_h, sigma=sigma_h / pm.math.sqr

        # Log-volatility random walk (AR(1))
        h = pm.GaussianRandomWalk('h',
                                  mu=mu_h * (1 - phi),
                                  sigma=sigma_h * pm.math.sqrt(1 - phi**2
                                  init_dist=pm.Normal.dist(mu_h, sigma_h)
                                  steps=len(returns_train)-1)

        # Return likelihood
        returns_obs = pm.Normal('returns_obs',
                                mu=0,
                                sigma=pm.math.exp(h/2),
                                observed=returns_train)

        # Sample
        trace = pm.sample(n_samples, tune=1000, chains=2, random_seed=42,
                          progressbar=True, target_accept=0.9)
    return trace, sv_model

print("Fitting Stochastic Volatility model...")
print("(This may take several minutes)\n")

trace_sv, model_sv = fit_stochastic_volatility(returns, n_samples=500)

print("\nStochastic Volatility model fitted!")
```

In [ ]:
```
# Extract and visualize latent volatility
h_samples = trace_sv.posterior['h'].values  # Shape: (chains, draws, time
h_mean = h_samples.mean(axis=(0, 1))
h_std = h_samples.std(axis=(0, 1))

# Convert log-vol to vol
vol_mean = np.exp(h_mean / 2)
vol_lower = np.exp((h_mean - 1.96*h_std) / 2)
vol_upper = np.exp((h_mean + 1.96*h_std) / 2)

# Plot
fig, axes = plt.subplots(2, 1, figsize=(14, 9))

# Returns
ax = axes[0]
ax.plot(returns[:400], linewidth=1, color='black', alpha=0.7)
ax.axhline(0, color='red', linestyle='--', linewidth=1)
ax.set_ylabel('Returns', fontsize=11)
ax.set_title('Crude Oil Returns', fontsize=12, fontweight='bold')
ax.grid(True, alpha=0.3)

# Estimated volatility
ax = axes[1]
ax.plot(vol_mean, linewidth=2, color='blue', label='Estimated volatility
ax.fill_between(range(len(vol_mean)), vol_lower, vol_upper,
                alpha=0.3, color='blue', label='95% credible interval')
ax.plot(true_vol[:400], linewidth=2, color='red', linestyle='--',
        alpha=0.6, label='True volatility')
ax.set_xlabel('Time', fontsize=11)
ax.set_ylabel('Volatility (σ_t)', fontsize=11)
ax.set_title('Stochastic Volatility Estimates', fontsize=12, fontweight='
ax.legend()
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Parameter summary
print("\n" + "="*70)
print("STOCHASTIC VOLATILITY PARAMETER POSTERIORS")
print("="*70)

sv_params = ['mu_h', 'phi', 'sigma_h']
sv_names = ['Mean log-vol (μ_h)', 'Persistence (φ)', 'Vol-of-vol (σ_h)']

print(f"{'Parameter':<25} {'Mean':>10} {'Std':>10} {'95% CI':>25}")
print("-"*70)

for param, name in zip(sv_params, sv_names):
    samples = trace_sv.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]:>8.3f},

phi_mean = np.mean(trace_sv.posterior['phi'].values)
sigma_h_mean = np.mean(trace_sv.posterior['sigma_h'].values)

print("\n" + "="*70)
print("INTERPRETATION")
print("="*70)
print(f"""
Persistence (φ): {phi_mean:.3f}
  → High persistence means volatility shocks last long
  → Half-life: ~{-np.log(2)/np.log(phi_mean):.1f} periods
  → Typical for commodities: φ ∈ [0.9, 0.99]

Vol-of-vol (σ_h): {sigma_h_mean:.3f}
  → Volatility itself is volatile!
  → Higher σ_h = more abrupt volatility regime changes
  → Important for option pricing (vega risk)

**Trading Application**:
- High φ → Volatility regimes are sticky
  → When vol spikes, it stays high for a while
  → Adjust position sizes for extended periods

- High σ_h → Volatility can change quickly
  → Need frequent rebalancing
  → Options may be mispriced (underestimate vol-of-vol)
""")
```

## 4 . Value at Risk (VaR) and Conditional VaR (CVaR)

### 4 . 1 What is VaR?

**Value at Risk (VaR)**: The maximum loss expected over a time horizon at a given confidence lev

**Mathematical definition**: $$\text{VaR}_{\alpha}(X) = \inf\{x : P(X \leq x) \geq \alpha\}$$

**Example**:

- VaR at 95 % confidence = 5 th percentile of loss distribution
- "With 95 % probability, losses won't exceed $X"

### 4 . 2 What is CVaR?

**Conditional Value at Risk (CVaR)** / **Expected Shortfall (ES)**: Average loss **beyond** VaR.

$$\text{CVaR}_{\alpha}(X) = \mathbb{E}[X | X \leq \text{VaR}_{\alpha}(X)]$$

**Why CVaR > VaR**:

- **VaR ignores tail shape**: Only cares about threshold
- **CVaR captures tail risk**: Average of all extreme losses
- **CVaR is coherent**: Satisfies desirable mathematical properties (subadditivity)

## 4.3 Bayesian VaR/CVaR

**Standard approach** (frequentist):

1. Estimate volatility $\hat{\sigma}$ (point estimate)
2. Assume normality: $\text{VaR}_{0.05} = \mu + \Phi^{-1}(0.05) \hat{\sigma}$
3. No uncertainty about $\sigma$

**Bayesian approach**:

1. Sample volatility from posterior: $\sigma \sim p(\sigma | \text{data})$
2. For each $\sigma$ sample, generate future returns
3. VaR/CVaR from posterior predictive distribution
4. **Accounts for parameter uncertainty** → More conservative risk estimates

In [ ]:
```python
# Calculate Bayesian VaR and CVaR from SV model
def calculate_bayesian_var_cvar(trace_sv, horizon=1, n_simulations=10000,
    """
    Calculate VaR and CVaR from stochastic volatility model.

    Accounts for:
    1. Parameter uncertainty (from posterior)
    2. Future volatility uncertainty (from SV dynamics)
    3. Return randomness (aleatoric)
    """
    # Extract posterior samples
    mu_h_samples = trace_sv.posterior['mu_h'].values.flatten()
    phi_samples = trace_sv.posterior['phi'].values.flatten()
    sigma_h_samples = trace_sv.posterior['sigma_h'].values.flatten()
    h_last = trace_sv.posterior['h'].values[:, :, -1].flatten()  # Last l

    n_param_samples = len(mu_h_samples)
    future_returns = np.zeros(n_simulations)

    for i in range(n_simulations):
        # Sample parameters from posterior
        idx = np.random.randint(0, n_param_samples)
        mu_h = mu_h_samples[idx]
        phi = phi_samples[idx]
        sigma_h = sigma_h_samples[idx]
        h_t = h_last[idx]

        # Simulate future volatility path
        cumulative_return = 0
        for t in range(horizon):
            # Update log-volatility
            h_t = mu_h + phi * (h_t - mu_h) + sigma_h * np.random.randn()

            # Generate return
            vol_t = np.exp(h_t / 2)
            ret_t = vol_t * np.random.randn()
            cumulative_return += ret_t

        future_returns[i] = cumulative_return

    # Calculate VaR and CVaR
    alpha = 1 - confidence
    var = np.percentile(future_returns, alpha * 100)
    cvar = future_returns[future_returns <= var].mean()

    return future_returns, var, cvar

# Calculate for different horizons
horizons = [1, 5, 10, 20]  # 1-day, 1-week, 2-week, 1-month
confidence = 0.95

results = {}
for h in horizons:
    returns_sim, var, cvar = calculate_bayesian_var_cvar(trace_sv, horizo
                                                          n_simulations=5
                                                          confidence=conf
    results[h] = {'returns': returns_sim, 'var': var, 'cvar': cvar}
    print(f"Horizon {h:2d} days: VaR = {var:.4f}, CVaR = {cvar:.4f}")

print("\nRisk calculations complete!")
```

In [ ]:
```
# Visualize VaR and CVaR
fig, axes = plt.subplots(2, 2, figsize=(15, 10))

for ax, h in zip(axes.flatten(), horizons):
    returns_sim = results[h]['returns']
    var = results[h]['var']
    cvar = results[h]['cvar']

    # Histogram of simulated returns
    ax.hist(returns_sim, bins=50, alpha=0.7, density=True, color='lightbl
            edgecolor='black', label='Posterior predictive')

    # VaR line
    ax.axvline(var, color='orange', linestyle='--', linewidth=2.5,
               label=f'VaR (95%) = {var:.4f}')

    # CVaR line
    ax.axvline(cvar, color='red', linestyle='-', linewidth=2.5,
               label=f'CVaR (95%) = {cvar:.4f}')

    # Shade tail
    tail_returns = returns_sim[returns_sim <= var]
    ax.hist(tail_returns, bins=20, alpha=0.5, density=True, color='red',
            edgecolor='darkred', label='Tail (losses > VaR)')

    ax.set_xlabel('Cumulative Return', fontsize=11)
    ax.set_ylabel('Density', fontsize=11)
    ax.set_title(f'{h}-Day Horizon', fontsize=12, fontweight='bold')
    ax.legend(fontsize=9)
    ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Summary table
print("\n" + "="*70)
print("BAYESIAN VaR AND CVaR SUMMARY (95% Confidence)")
print("="*70)
print(f"{'Horizon':<10} {'VaR':>12} {'CVaR':>12} {'CVaR/VaR':>12} {'$ on
print("-"*70)
```

```
portfolio_value = 100000

for h in horizons:
    var = results[h]['var']
    cvar = results[h]['cvar']
    ratio = cvar / var if var != 0 else np.nan
    dollar_cvar = portfolio_value * abs(cvar)

    print(f"{h:2d} days    {var:>12.4f} {cvar:>12.4f} {ratio:>12.2f} ${do

print("\n" + "="*70)
print("INTERPRETATION FOR TRADERS")
print("="*70)
print(f"""
VaR (Value at Risk):
  "With 95% confidence, losses won't exceed VaR"
  Example: 1-day VaR = {results[1]['var']:.4f}
  → On a $100k position, max 1-day loss is ${portfolio_value * abs(result

CVaR (Conditional VaR / Expected Shortfall):
  "Average loss in the worst 5% of cases"
  Example: 1-day CVaR = {results[1]['cvar']:.4f}
  → When things go bad (worst 5%), expect to lose ${portfolio_value * abs

CVaR/VaR Ratio:
  > 1.0 means tail is fat (extreme losses are much worse than VaR)
  = 1.0 would mean no tail risk beyond VaR

  Our ratio: ~{results[1]['cvar']/results[1]['var']:.2f}
  → Tail risk is significant! Don't just rely on VaR.

**Risk Management Actions**:
1. Set position limits using CVaR (more conservative than VaR)
2. Increase margin requirements in high-volatility regimes
3. Use options to cap tail risk when CVaR/VaR ratio is high
4. Scale positions: Position size ∝ 1/CVaR
""")
```

# 5 . Volatility Forecasting for Risk Management

## 5 . 1 Why Forecast Volatility?

**Trading applications**:

1 . **Position sizing**: Higher forecast vol → smaller positions
2 . **Stop-loss placement**: Wider stops in high-vol regimes
3 . **Option strategies**: Sell vol when forecast < implied, buy when forecast > implied
4 . **Risk budgeting**: Allocate more risk capital to low-vol assets
5 . **Margin requirements**: Exchanges use vol forecasts for margin

## 5 . 2 Multi-Step Ahead Volatility Forecasts

From SV model: $$h_{t+k} = \mu_h + \phi^k (h_t - \mu_h) + \text{noise}$$

**Key insight**: As $k \to \infty$, $h_{t+k} \to \mu_h$ (mean reversion).

**Forecast variance**: $$\text{Var}(h_{t+k}) = \sigma_h^ 2 \frac{ 1 - \phi^{ 2 k}}{ 1 - \phi^

- Short horizon: Low variance (know recent vol)
- Long horizon: Converges to unconditional variance

In [ ]:
```python
# Generate volatility forecasts from SV model
def forecast_volatility(trace_sv, n_ahead=30, n_samples=1000):
    """
    Generate multi-step ahead volatility forecasts.
    """
    # Extract parameters
    mu_h_samples = trace_sv.posterior['mu_h'].values.flatten()
    phi_samples = trace_sv.posterior['phi'].values.flatten()
    sigma_h_samples = trace_sv.posterior['sigma_h'].values.flatten()
    h_last = trace_sv.posterior['h'].values[:, :, -1].flatten()

    # Storage for forecasts
    vol_forecasts = np.zeros((n_samples, n_ahead))

    for i in range(n_samples):
        # Sample parameters
        idx = np.random.randint(0, len(mu_h_samples))
        mu_h = mu_h_samples[idx]
        phi = phi_samples[idx]
        sigma_h = sigma_h_samples[idx]
        h_t = h_last[idx]

        # Simulate forward
        for t in range(n_ahead):
            h_t = mu_h + phi * (h_t - mu_h) + sigma_h * np.random.randn()
            vol_forecasts[i, t] = np.exp(h_t / 2)

    return vol_forecasts

# Generate forecasts
n_ahead = 60  # 2 months ahead
vol_forecasts = forecast_volatility(trace_sv, n_ahead=n_ahead, n_samples=

# Summary statistics
vol_mean = vol_forecasts.mean(axis=0)
vol_median = np.median(vol_forecasts, axis=0)
vol_lower = np.percentile(vol_forecasts, 5, axis=0)
vol_upper = np.percentile(vol_forecasts, 95, axis=0)

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

# Forecast fan chart
ax = axes[0]
ax.plot(range(n_ahead), vol_mean, linewidth=2.5, color='blue', label='Mea
ax.plot(range(n_ahead), vol_median, linewidth=2, color='green',
        linestyle='--', label='Median forecast')
ax.fill_between(range(n_ahead), vol_lower, vol_upper,
                alpha=0.3, color='blue', label='90% prediction interval')
ax.axhline(vol_mean[-1], color='red', linestyle=':', linewidth=2,
           label=f'Long-run mean = {vol_mean[-1]:.4f}')
ax.set_xlabel('Days Ahead', fontsize=11)
ax.set_ylabel('Volatility', fontsize=11)
ax.set_title('Volatility Forecast from Stochastic Volatility Model',
             fontsize=12, fontweight='bold')
ax.legend()
ax.grid(True, alpha=0.3)

# Uncertainty growth
ax = axes[1]
forecast_std = vol_forecasts.std(axis=0)
ax.plot(range(n_ahead), forecast_std, linewidth=2.5, color='purple')
ax.fill_between(range(n_ahead), 0, forecast_std, alpha=0.3, color='purple
ax.set_xlabel('Days Ahead', fontsize=11)
ax.set_ylabel('Forecast Uncertainty (Std Dev)', fontsize=11)
ax.set_title('Volatility Forecast Uncertainty Growth', fontsize=12, fontw
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print("\n" + "="*70)
print("VOLATILITY FORECAST SUMMARY")
print("="*70)
print(f"\nCurrent volatility (last obs): {vol_mean[0]:.4f}")
print(f"1-week ahead forecast:         {vol_mean[4]:.4f}")
print(f"1-month ahead forecast:        {vol_mean[19]:.4f}")
print(f"Long-run mean:                 {vol_mean[-1]:.4f}")

print("\n" + "="*70)
print("TRADING IMPLICATIONS")
print("="*70)
print(f"""
1. **Mean Reversion**:
   - Volatility forecasts converge to long-run mean (~{vol_mean[-1]:.4f})
   - If current vol > mean → expect decrease (sell volatility)
   - If current vol < mean → expect increase (buy volatility)

2. **Uncertainty Growth**:
   - Forecast uncertainty increases with horizon
   - Day 1 uncertainty: {forecast_std[0]:.4f}
   - Day 30 uncertainty: {forecast_std[29]:.4f}
   - Factor: {forecast_std[29]/forecast_std[0]:.1f}x higher

3. **Position Sizing**:
   - Scale position by 1/forecast_vol
   - High vol forecast → reduce position size
   - Low vol forecast → can increase position size

4. **Option Strategies**:
   - Compare forecast vol to implied vol
   - If forecast < implied → sell options (vol is overpriced)
   - If forecast > implied → buy options (vol is underpriced)

5. **Stop-Loss Adjustment**:
   - Stop distance ∝ forecast_vol
   - High vol → wider stops (avoid noise-triggered exits)
   - Low vol → tighter stops (protect gains)
""")
```

# 6 . Summary: Uncertainty Quantification in Practice

## 6 . 1 Key Takeaways

| Concept | Why It Matters |
|---|---|
| **Epistemic vs Aleatoric** | Know what uncertainty you can reduce (get more data) vs what's irreducible ( randomness) |
| **GARCH** | Volatility clusters—use GARCH to capture time-varying variance |
| **Stochastic Volatility** | Volatility has its own random shocks—better for option pricing and tail risk |
| **Bayesian approach** | Parameter uncertainty propagates to forecasts → more honest risk estimates |
| **VaR vs CVaR** | CVaR captures tail risk better—use it for position limits |
| **Volatility forecasting** | Scale positions inversely with forecast vol |

## 6 . 2 Model Selection Guide

**Use GARCH when**:

- ✅ Need computationally fast forecasts
- ✅ Primarily interested in point volatility forecasts
- ✅ High-frequency trading (GARCH updates quickly)

**Use Stochastic Volatility when**:

- ✅ Pricing options (needs realistic vol dynamics)
- ✅ Risk management (want full distribution of future vol)
- ✅ Modeling volatility of volatility matters

## 6 . 3 Practical Workflow

1 . **Model volatility** (GARCH or SV)
2 . **Forecast multi-step ahead**
3 . **Calculate VaR/CVaR** from posterior predictive
4 . **Size positions**: $w_t = \frac{ 1 }{\text{CVaR}_t}$ (Kelly-like)
5 . **Set stop-losses**: $\text{Stop distance} = 2 \times \text{forecast vol}$
6 . **Rebalance**: Update forecasts daily/weekly

## 6 . 4 Common Pitfalls

❌ **Assuming constant volatility**

- Use time-varying vol models (GARCH/SV)

❌ **Ignoring parameter uncertainty**

- Bayesian approach accounts for this automatically

❌ **Using only VaR**

- VaR doesn't capture tail risk—use CVaR

❌ **Over-relying on historical volatility**

- Markets have regime changes—use priors that allow for this

❌ **Not updating forecasts**

- Volatility changes—refit models regularly (weekly for commodities)

---

---

## Knowledge Check Quiz

**Q** 1 : Epistemic uncertainty can be reduced by:

- A) Collecting more data
- B) Using better risk management
- C) Diversification
- D) It cannot be reduced

**Q** 2 : In GARCH( 1 , 1 ), high $\alpha + \beta$ (close to 1 ) means:

- A) Volatility changes very quickly
- B) Volatility shocks are persistent
- C) The model is misspecified
- D) Returns are normally distributed

**Q** 3 : CVaR is better than VaR for risk management because:

- A) It's easier to calculate
- B) It captures the average loss in the tail, not just the threshold
- C) It's always smaller than VaR
- D) Regulators don't require it

**Q** 4 : Stochastic Volatility models differ from GARCH by:

- A) Being faster to estimate
- B) Having separate random shocks for volatility
- C) Not requiring MCMC
- D) Always fitting better

**Q** 5 : If a commodity has high forecasted volatility, you should:

- A) Increase position size (more opportunity)
- B) Decrease position size (more risk)
- C) Keep position size constant
- D) Exit all positions immediately

In [ ]:
```
# Quiz Answers
print("="*70)
print("QUIZ ANSWERS")
print("="*70)
print("""
Q1: A) Collecting more data
    Epistemic uncertainty is parameter/model uncertainty. More data
    → tighter posterior → less epistemic uncertainty. Aleatoric
    uncertainty (market randomness) cannot be reduced.

Q2: B) Volatility shocks are persistent
    α + β is the persistence parameter. Close to 1 means volatility
    shocks decay very slowly (integrated GARCH). Typical for financial
    time series where high-vol periods last for weeks/months.

Q3: B) It captures the average loss in the tail, not just the threshold
    VaR only tells you the threshold (e.g., 5th percentile). CVaR
    tells you the average of all losses worse than VaR. This is much
    more useful for risk management and is a coherent risk measure.

Q4: B) Having separate random shocks for volatility
    GARCH: σ²_t is a deterministic function of past returns
    SV: h_t has its own random shock η_t
    This makes SV more flexible and better for option pricing.

Q5: B) Decrease position size (more risk)
    Higher volatility = higher risk. Position sizing should be
    inversely proportional to volatility: size ∝ 1/σ_forecast.
    This is the foundation of risk parity and Kelly criterion.
""")
```

---

## Exercises

### Exercise 1: GARCH Parameter Sensitivity (Easy)

Simulate GARCH(1, 1) with different parameter values:

- High α, low β (volatile but mean-reverting)
- Low α, high β (smooth but persistent)
- α + β = 0.99 (nearly integrated)

Compare volatility dynamics and forecast accuracy.

### Exercise 2: VaR Backtesting (Medium)

1. Generate synthetic returns with time-varying vol
2. Calculate daily VaR forecasts (95% confidence)
3. Count how many days actual losses exceed VaR
4. Should be ~5% for well-calibrated forecasts
5. Test with constant vol vs GARCH vol

### Exercise 3: Volatility Trading Strategy (Hard)

Build a strategy that trades based on vol forecasts:

1. Forecast volatility using SV model
2. Compare to realized volatility
3. Trade signals:
    - If forecast_vol > realized_vol → reduce position
    - If forecast_vol < realized_vol → increase position
4. Backtest and calculate Sharpe ratio
5. Compare to constant position size

### Exercise 4: Option Pricing with SV (Advanced)

Use the SV model to price European call options:

1. Simulate future price paths from SV model
2. Calculate option payoffs: max(S_T - K, 0)
3. Discount to present value
4. Compare to Black-Scholes (constant vol)
5. Analyze implied volatility smile

---

## Next Module Preview

In **Module 10: Backtesting, Evaluation, and Trading Strategies**, we'll bring everything tc

- Walk-forward validation for Bayesian models
- Proper backtesting (avoiding look-ahead bias)
- CRPS and calibration assessment
- Trading strategies: mean reversion, trend following, pairs trading
- Portfolio optimization with Bayesian returns
- Complete energy portfolio trading system

---

*Module 9 Complete*

# Module 10: Backtesting, Evaluation, and Trading Strategies

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

**CAPSTONE MODULE**: Integrating Everything We've Learned

---

## Learning Objectives

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

1. **Implement** walk-forward validation to avoid look-ahead bias in Bayesian models
2. **Evaluate** probabilistic forecasts using proper scoring rules (CRPS, log-score)
3. **Assess** forecast calibration and reliability for risk management
4. **Calculate** probabilistic Sharpe ratios accounting for parameter uncertainty
5. **Apply** Kelly criterion with Bayesian posteriors for optimal position sizing
6. **Build** complete trading strategies:
    - Mean reversion with uncertainty bands
    - Trend following with regime detection
    - Pairs trading with Bayesian cointegration
7. **Optimize** portfolios using Bayesian return distributions
8. **Deploy** a complete energy portfolio trading system

---

## Why This Matters for Trading

**This is where theory meets reality.** Everything we've learned—priors, MCMC, time series mod
volatility—means nothing if we can't:

1. **Backtest correctly**: Avoid overfitting and look-ahead bias
2. **Evaluate honestly**: Use metrics that reward calibrated forecasts, not just accuracy
3. **Size positions optimally**: Bayesian Kelly criterion with parameter uncertainty
4. **Manage risk systematically**: Probabilistic stop-losses, portfolio constraints
5. **Execute in practice**: Handle transaction costs, slippage, margin requirements

### Real-World Trading Failures

- **Long-Term Capital Management (1998)**: Models didn't account for parameter uncer
  over-leveraged → $4B loss
- **Amaranth Advisors (2006)**: Natural gas volatility model failed during regime change
  loss in one week
- **Quantitative funds (August 2007)**: Models over-fit to in-sample data, failed out-of-
  → multi-billion dollar losses

**The pattern**: Great models, terrible backtesting/risk management.

## What Makes This Module Different

- **No look-ahead bias**: Walk-forward validation
- **Honest evaluation**: Probabilistic metrics (CRPS, calibration)
- **Risk-first thinking**: Position sizing before alpha generation
- **Complete systems**: Not just signals, but full trading workflows

---

In [ ]:
```python
# Setup
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')

# Import course utilities
import sys
sys.path.append('../..')
from utils.backtesting import WalkForwardValidator, Backtester, kelly_pos
from utils.metrics import crps, sharpe_ratio, forecast_summary, mae, rmse

np.random.seed(42)
plt.style.use('seaborn-v0_8-whitegrid')
plt.rcParams['figure.figsize'] = (14, 6)
plt.rcParams['font.size'] = 11

print("All libraries and utilities loaded successfully!")
print(f"PyMC version: {pm.__version__}")
```

# 1 . Walk-Forward Validation: The Gold Standard

## 1 . 1 The Problem: In-Sample Overfitting

**Common mistake**: Fit model on all historical data, report performance.

**Why it fails**:

- Model "sees" future data during fitting
- Overfits to noise in the training period
- Out-of-sample performance much worse

## 1 . 2 Walk-Forward Validation

**Idea**: Simulate real-time trading by:

1 . Fit model on data up to time $t$
2 . Forecast for time $t+ 1 , ..., t+k$
3 . Observe actual outcomes

4 . **Roll forward** to $t+k$, refit, repeat

**Two types**:

- **Expanding window**: Training set grows (use all historical data)
- **Rolling window**: Fixed training size (only recent data)

## 1 . 3 Bayesian Walk-Forward

**Standard**: Refit model from scratch each period (slow).

**Bayesian**: Can use **sequential updating**:

- Previous posterior becomes next prior
- Much faster than full refit
- Naturally adapts to regime changes

$$p(\theta | \text{data}_{ 1 :t+ 1 }) \propto p(\text{data}_{t+ 1 } | \theta) \cdot p(\theta | \text{data}
$

In [ ]:
```
# Generate realistic commodity data for backtesting
def generate_commodity_data(n=500, seed=42):
    """
    Generate synthetic crude oil prices with regime changes.
    """
    np.random.seed(seed)
    dates = pd.date_range('2020-01-01', periods=n, freq='D')

    # Generate returns with multiple regimes
    returns = np.zeros(n)
    volatility = np.zeros(n)

    # Regime 1: Normal (days 0-200)
    vol1 = 0.015
    returns[:200] = np.random.normal(0.0003, vol1, 200)
    volatility[:200] = vol1

    # Regime 2: High volatility (days 200-350) - COVID-like shock
    vol2 = 0.04
    returns[200:350] = np.random.normal(-0.001, vol2, 150)
    volatility[200:350] = vol2

    # Regime 3: Recovery (days 350-500)
    vol3 = 0.02
    returns[350:] = np.random.normal(0.0008, vol3, n - 350)
    volatility[350:] = vol3

    # Convert to prices
    prices = 60 * np.exp(np.cumsum(returns))

    df = pd.DataFrame({
        'date': dates,
        'close': prices,
        'returns': returns,
        'true_vol': volatility
    })
```

```python
    df.set_index('date', inplace=True)

    return df

# Generate data
data = generate_commodity_data(n=500)

# Visualize
fig, axes = plt.subplots(2, 1, figsize=(14, 8))

# Prices
ax = axes[0]
ax.plot(data.index, data['close'], linewidth=1.5, color='black')
ax.axvspan(data.index[200], data.index[350], alpha=0.2, color='red', labe
ax.set_ylabel('Price ($/barrel)', fontsize=11)
ax.set_title('Crude Oil Prices (Synthetic)', fontsize=12, fontweight='bol
ax.legend()
ax.grid(True, alpha=0.3)

# Returns
ax = axes[1]
ax.plot(data.index, data['returns'], linewidth=0.8, color='blue', alpha=0
ax.axhline(0, color='red', linestyle='--', linewidth=1)
ax.axvspan(data.index[200], data.index[350], alpha=0.2, color='red')
ax.set_ylabel('Returns', fontsize=11)
ax.set_xlabel('Date', fontsize=11)
ax.set_title('Daily Returns (showing regime change)', fontsize=12, fontwe
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

print(f"\nDataset: {len(data)} days from {data.index[0].date()} to {data.
print(f"\nRegime statistics:")
print(f"  Normal period (days 1-200):      Vol = {data['true_vol'][:200].
print(f"  Crisis period (days 201-350):    Vol = {data['true_vol'][200:35
print(f"  Recovery period (days 351-500):  Vol = {data['true_vol'][350:].
```

```python
In [ ]: # Implement walk-forward backtesting with Bayesian model
def bayesian_ar_model(train_data, n_lags=5):
    """
    Fit Bayesian AR(p) model for returns.
    """
    returns = train_data['returns'].values

    # Create lagged features
    X = np.column_stack([returns[n_lags-i-1:-i-1] for i in range(n_lags)]
    y = returns[n_lags:]

    with pm.Model() as model:
        # Priors
        beta = pm.Normal('beta', mu=0, sigma=0.5, shape=n_lags)
        sigma = pm.HalfNormal('sigma', sigma=0.02)

        # Likelihood
        mu = pm.math.dot(X, beta)
        y_obs = pm.Normal('y_obs', mu=mu, sigma=sigma, observed=y)

        # Sample
        trace = pm.sample(500, tune=500, chains=2, random_seed=42,
                          progressbar=False, return_inferencedata=True)

    return trace, model

def predict_bayesian(trace, test_data, n_lags=5, n_samples=1000):
    """
    Generate probabilistic forecasts from Bayesian AR model.
    """
    # Get last n_lags returns for prediction
    last_returns = test_data['returns'].values[:n_lags][::-1]  # Reverse

    # Extract posterior samples
    beta_samples = trace.posterior['beta'].values.reshape(-1, n_lags)
    sigma_samples = trace.posterior['sigma'].values.flatten()

    # Generate predictions
    n_post_samples = len(beta_samples)
    predictions = np.zeros(n_post_samples)

    for i in range(n_post_samples):
        mu = np.dot(last_returns, beta_samples[i])
        predictions[i] = mu + sigma_samples[i] * np.random.randn()

    # Summary statistics
    pred_mean = predictions.mean()
    pred_std = predictions.std()

    return pd.DataFrame({
        'prediction': [pred_mean],
        'lower_95': [np.percentile(predictions, 2.5)],
        'upper_95': [np.percentile(predictions, 97.5)],
        'std': [pred_std]
    }, index=test_data.index[:1])

# Run walk-forward validation
print("Running walk-forward validation...")
print("This simulates real-time trading (no look-ahead bias)\n")

# Parameters
min_train_size = 100
test_size = 1  # 1-day ahead forecasts
n_folds = 50  # 50 forecast periods

# Storage
wf_results = []
fold_count = 0

# Walk-forward loop
for i in range(n_folds):
    # Define train/test split
    train_end = min_train_size + i * 5  # Expand training window
    test_start = train_end
    test_end = test_start + test_size

    if test_end > len(data):
        break

    train = data.iloc[:train_end]
    test = data.iloc[test_start:test_end]
    # Fit model on training data
    try:
        trace, model = bayesian_ar_model(train, n_lags=5)

        # Generate forecast
        forecast = predict_bayesian(trace, test, n_lags=5)
        forecast['actual'] = test['returns'].values[0]
        forecast['fold'] = fold_count

        wf_results.append(forecast)
        fold_count += 1

        if fold_count % 10 == 0:
            print(f"Completed fold {fold_count}/{n_folds}")
    except:
        print(f"Fold {i} failed, skipping...")
        continue

# Combine results
wf_df = pd.concat(wf_results)

print(f"\nWalk-forward validation complete: {len(wf_df)} forecasts genera
```

In [ ]:
```
# Evaluate walk-forward results
fig, axes = plt.subplots(2, 1, figsize=(14, 9))

# Plot 1: Forecasts vs actuals
ax = axes[0]
ax.scatter(range(len(wf_df)), wf_df['actual'], c='black', s=40,
           alpha=0.6, label='Actual returns', zorder=3)
ax.plot(range(len(wf_df)), wf_df['prediction'], 'blue', linewidth=2,
        label='Forecast (posterior mean)', zorder=2)
ax.fill_between(range(len(wf_df)),
                wf_df['lower_95'],
                wf_df['upper_95'],
                alpha=0.3, color='blue', label='95% credible interval', z
ax.axhline(0, color='red', linestyle='--', linewidth=1, alpha=0.5)
ax.set_xlabel('Forecast Period', fontsize=11)
ax.set_ylabel('Returns', fontsize=11)
ax.set_title('Walk-Forward Validation: Out-of-Sample Forecasts',
             fontsize=12, fontweight='bold')
ax.legend()
ax.grid(True, alpha=0.3)

# Plot 2: Forecast errors
ax = axes[1]
errors = wf_df['actual'] - wf_df['prediction']
ax.hist(errors, bins=25, alpha=0.7, density=True, color='steelblue',
        edgecolor='black', label='Forecast errors')
ax.axvline(0, color='red', linestyle='--', linewidth=2, label='Zero error
ax.axvline(errors.mean(), color='green', linestyle='-', linewidth=2,
           label=f'Mean error = {errors.mean():.6f}')
ax.set_xlabel('Forecast Error (actual - predicted)', fontsize=11)
ax.set_ylabel('Density', fontsize=11)
ax.set_title('Distribution of Forecast Errors', fontsize=12, fontweight='
ax.legend()
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```

```
# Calculate metrics
mae_score = mae(wf_df['actual'].values, wf_df['prediction'].values)
rmse_score = rmse(wf_df['actual'].values, wf_df['prediction'].values)
coverage = np.mean((wf_df['actual'] >= wf_df['lower_95']) &
                   (wf_df['actual'] <= wf_df['upper_95']))

print("\n" + "="*70)
print("WALK-FORWARD VALIDATION RESULTS")
print("="*70)
print(f"\nForecast Accuracy:")
print(f"  MAE:  {mae_score:.6f}")
print(f"  RMSE: {rmse_score:.6f}")
print(f"\nCalibration:")
print(f"  95% Interval Coverage: {coverage:.1%}")
print(f"  Target: 95% (well-calibrated forecasts)")
if coverage < 0.90:
    print(f"  ⚠ Under-confident: Intervals too narrow")
elif coverage > 0.98:
    print(f"  ⚠ Over-confident: Intervals too wide")
else:
    print(f"  ✓ Well-calibrated forecasts!")

print(f"\nForecast Bias:")
print(f"  Mean error: {errors.mean():.6f}")
if abs(errors.mean()) < 0.001:
    print(f"  ✓ Unbiased forecasts")
else:
    print(f"  ⚠ Systematic bias detected")

print("\n" + "="*70)
print("KEY INSIGHTS")
print("="*70)
print("""
Walk-forward validation ensures:
1. NO LOOK-AHEAD BIAS - Model never sees future data
2. REALISTIC PERFORMANCE - Simulates actual trading
3. OUT-OF-SAMPLE EVALUATION - True test of generalization

Coverage = {:.1%} means:
- {:.1%} of actual returns fell within 95% prediction intervals
- Close to 95% target → forecasts are well-calibrated
- Important for risk management (VaR, stop-losses)

**Trading implication**:
Well-calibrated intervals → can trust uncertainty estimates for:
- Position sizing (scale by 1/uncertainty)
- Stop-loss placement (outside 95% interval)
- Risk limits (based on CVaR from predictive distribution)
""".format(coverage, coverage))
```

# 2 . Proper Scoring Rules for Probabilistic Forecasts

## 2 . 1 Why Standard Metrics Fail

**MAE and RMSE**:

- Only evaluate point forecasts (mean or median)

- Ignore forecast uncertainty
- Don't penalize poorly calibrated intervals

**Example problem**:

- Forecast A: "Return = 0.001 ± 0.01" (uncertain)
- Forecast B: "Return = 0.001 ± 0.0001" (very confident)

If actual = 0.01, both have same MAE. But Forecast B is badly calibrated!

## 2.2 Continuous Ranked Probability Score (CRPS)

**The gold standard** for probabilistic forecasts.

$$\text{CRPS}(F, y) = \int_{-\infty}^{\infty} [F(x) - \mathbb{1}(x \geq y)]^2 dx$$

Where:

- $F(x)$ is the forecast CDF
- $y$ is the actual outcome
- $\mathbb{1}(x \geq y)$ is the "perfect" forecast (step function at $y$)

**Properties**:

1. **Proper scoring rule**: Minimized when forecast = true distribution
2. **Generalizes MAE**: CRPS → MAE when forecast is deterministic
3. **Rewards calibration**: Penalizes both bias and miscalibrated uncertainty

**Sample-based approximation**: $$\text{CRPS} \approx \mathbb{E}[|X - y|] - \frac{1}{2}\mathbb
X'|]$$

where $X, X'$ are independent samples from forecast distribution.

## 2.3 Log Score (Ignorance Score)

$$\text{Log Score} = -\log p(y | \text{forecast})$$

- Lower is better
- Heavily penalizes forecasts that assign low probability to actual outcome
- Related to information theory ("surprisal")

In [ ]:

```python
# Calculate CRPS for walk-forward forecasts
# We'll need to re-generate posterior predictive samples

# For demonstration, simulate from forecast distributions
def calculate_crps_from_gaussian(actual, mean, std):
    """
    CRPS for Gaussian forecast (closed form).
    """
    z = (actual - mean) / std
    phi = stats.norm.pdf(z)
    Phi = stats.norm.cdf(z)
    crps_val = std * (z * (2 * Phi - 1) + 2 * phi - 1 / np.sqrt(np.pi))
    return crps_val

# Calculate CRPS for each forecast
crps_scores = []
for idx, row in wf_df.iterrows():
    crps_val = calculate_crps_from_gaussian(row['actual'], row['predictio
    crps_scores.append(crps_val)

wf_df['crps'] = crps_scores
mean_crps = np.mean(crps_scores)

# Also calculate traditional metrics for comparison
mae_score = mae(wf_df['actual'].values, wf_df['prediction'].values)
rmse_score = rmse(wf_df['actual'].values, wf_df['prediction'].values)

# Visualize CRPS over time
fig, axes = plt.subplots(2, 1, figsize=(14, 9))

# CRPS time series
ax = axes[0]
ax.plot(range(len(wf_df)), wf_df['crps'], linewidth=2, color='purple', la
ax.axhline(mean_crps, color='red', linestyle='--', linewidth=2,
           label=f'Mean CRPS = {mean_crps:.6f}')
ax.set_xlabel('Forecast Period', fontsize=11)
ax.set_ylabel('CRPS (lower = better)', fontsize=11)
ax.set_title('Continuous Ranked Probability Score Over Time',
             fontsize=12, fontweight='bold')
ax.legend()
ax.grid(True, alpha=0.3)

# Metric comparison
ax = axes[1]
metrics = ['MAE', 'RMSE', 'CRPS']
values = [mae_score, rmse_score, mean_crps]
colors = ['steelblue', 'orange', 'purple']

bars = ax.bar(metrics, values, color=colors, alpha=0.7, edgecolor='black'
ax.set_ylabel('Score (lower = better)', fontsize=11)
ax.set_title('Forecast Evaluation Metrics Comparison', fontsize=12, fontw
ax.grid(True, alpha=0.3, axis='y')

# Add value labels on bars
for bar, val in zip(bars, values):
    height = bar.get_height()
    ax.text(bar.get_x() + bar.get_width()/2., height,
            f'{val:.6f}',
            ha='center', va='bottom', fontsize=10, fontweight='bold')

plt.tight_layout()
plt.show()

print("\n" + "="*70)
print("PROBABILISTIC FORECAST EVALUATION")
print("="*70)
print(f"\nPoint Forecast Metrics (only evaluate mean):")
print(f"  MAE:  {mae_score:.6f}")
print(f"  RMSE: {rmse_score:.6f}")
print(f"\nProbabilistic Metric (evaluates full distribution):")
print(f"  CRPS: {mean_crps:.6f}")
print("\n" + "="*70)
print("WHY CRPS MATTERS")
print("="*70)
print("""
CRPS vs MAE:
- MAE only cares if mean forecast is close to actual
- CRPS also cares if uncertainty is correctly specified

Example:
  Actual return: 0.01

  Forecast A: Mean = 0.005, Std = 0.02 (wide, honest uncertainty)
  Forecast B: Mean = 0.005, Std = 0.001 (narrow, overconfident)

  MAE: Same for both (|0.01 - 0.005| = 0.005)
  CRPS: Lower for A (correctly captures uncertainty)

**Trading Application**:
Use CRPS to select models for:
1. Position sizing (need correct uncertainty)
2. Option pricing (need correct distribution)
3. Risk management (VaR/CVaR depend on tails)

MAE/RMSE are okay for:
1. Pure alpha generation (only care about directional accuracy)
2. Benchmarking against non-probabilistic models
""")
```

## 3 . Kelly Criterion with Bayesian Uncertainty

### 3 . 1 Classical Kelly Criterion

**Optimal position size** to maximize long-term growth rate:

$$f^* = \frac{\mu - r}{\sigma^2}$$

Where:

- $f^*$ = fraction of capital to invest
- $\mu$ = expected return
- $r$ = risk-free rate
- $\sigma^2$ = variance of returns

**Problem**: Assumes we **know** $\mu$ and $\sigma$ with certainty.

### 3 . 2 Bayesian Kelly

**Reality**: We're uncertain about $\mu$ and $\sigma$.

**Solution**: $$f^* = \mathbb{E}_{\theta}\left[\frac{\mu(\theta) - r}{\sigma^2(\theta)}\right]$$

where expectation is over posterior $p(\theta | \text{data})$.

**Alternative (more conservative)**: $$f^* = \frac{\mathbb{E}[\mu] - r}{\mathbb{E}[\sigma^2] \text{Var}[\mu]}$$

The extra term $\text{Var}[\mu]$ accounts for epistemic uncertainty about expected return.

## 3.3 Fractional Kelly

**Full Kelly** ($f^ ∗ $) maximizes growth but has high volatility.

**Fractional Kelly**: Use fraction of Kelly $$f_{\text{actual}} = \lambda f^ ∗ , \quad \lambda \in [ 0 . 0 . 5 ]$$

**Why**:

- Reduces volatility dramatically
- Protects against estimation errors
- "Half-Kelly" is common in practice

In [ ]:
```
# Implement Bayesian Kelly criterion
def bayesian_kelly(trace, risk_free_rate=0.0, fraction=0.5):
    """
    Calculate Bayesian Kelly position sizing.

    Parameters:
    -----------
    trace : InferenceData
        Posterior samples from Bayesian model
    risk_free_rate : float
        Annualized risk-free rate
    fraction : float
        Fraction of Kelly to use (0.5 = half-Kelly)
    """
    # Extract posterior samples for mean and variance
    # Assuming model predicts returns with mean beta * X and variance sig

    # For demonstration, simulate from predictive distribution
    # In practice, use actual posterior predictive samples
    n_samples = 10000

    # Simulate expected returns and volatilities from posterior
    # (In real implementation, this comes from model)
    expected_returns = np.random.normal(0.0005, 0.0002, n_samples)  # Epi
    volatilities = np.random.gamma(4, 0.005, n_samples)  # Uncertainty ab

    # Calculate Kelly fraction for each posterior sample
    kelly_fractions = (expected_returns - risk_free_rate/252) / volatilit

    # Conservative: average Kelly across posterior
    mean_kelly = np.mean(kelly_fractions)

    # Apply fractional Kelly
    actual_kelly = fraction * mean_kelly

    # Clip to reasonable range
    actual_kelly = np.clip(actual_kelly, -1.0, 1.0)

    return {
        'kelly_samples': kelly_fractions,
        'mean_kelly': mean_kelly,
        'fractional_kelly': actual_kelly,
        'fraction_used': fraction,
        'expected_return': np.mean(expected_returns),
        'expected_vol': np.mean(volatilities)
    }

# Calculate Kelly for different fractions
fractions = [0.25, 0.5, 0.75, 1.0]
kelly_results = {}

for frac in fractions:
    # Use dummy trace (in practice, use actual model trace)
    result = bayesian_kelly(None, risk_free_rate=0.02, fraction=frac)
    kelly_results[frac] = result

# Visualize
fig, axes = plt.subplots(1, 2, figsize=(15, 5))

# Kelly distribution from posterior
ax = axes[0]
kelly_samples = kelly_results[1.0]['kelly_samples']
ax.hist(kelly_samples, bins=50, alpha=0.7, density=True, color='steelblue
ax.axvline(kelly_results[1.0]['mean_kelly'], color='red', linestyle='--',
           linewidth=2.5, label=f"Mean Kelly = {kelly_results[1.0]['mean_
ax.axvline(0, color='black', linestyle=':', linewidth=1.5)
ax.set_xlabel('Kelly Fraction', fontsize=11)
ax.set_ylabel('Density', fontsize=11)
ax.set_title('Bayesian Kelly Distribution\n(accounting for parameter unce
             fontsize=12, fontweight='bold')
ax.legend()
ax.grid(True, alpha=0.3)

# Fractional Kelly comparison
ax = axes[1]
frac_labels = ['Quarter-Kelly', 'Half-Kelly', '3/4-Kelly', 'Full Kelly']
frac_values = [kelly_results[f]['fractional_kelly'] for f in fractions]
colors = ['green', 'blue', 'orange', 'red']

bars = ax.bar(frac_labels, frac_values, color=colors, alpha=0.7, edgecolo
ax.axhline(0, color='black', linestyle='-', linewidth=1)
ax.set_ylabel('Position Size (fraction of capital)', fontsize=11)
ax.set_title('Fractional Kelly Position Sizing', fontsize=12, fontweight=
ax.grid(True, alpha=0.3, axis='y')

# Add value labels
for bar, val in zip(bars, frac_values):
    height = bar.get_height()
    ax.text(bar.get_x() + bar.get_width()/2., height,
            f'{val:.3f}',
            ha='center', va='bottom' if val > 0 else 'top',
            fontsize=10, fontweight='bold')

plt.tight_layout()
plt.show()

print("\n" + "="*70)
print("BAYESIAN KELLY CRITERION")
print("="*70)
print(f"\nExpected Return: {kelly_results[1.0]['expected_return']:.4%} (d
print(f"Expected Volatility: {kelly_results[1.0]['expected_vol']:.4%} (da
print(f"\nOptimal Position Sizes:")
```

```python
for frac, label in zip(fractions, frac_labels):
    pos = kelly_results[frac]['fractional_kelly']
    print(f"  {label:15s}: {pos:>7.1%} of capital")

print("\n" + "="*70)
print("KELLY GUIDELINES FOR TRADING")
print("="*70)
print("""
1. **Full Kelly is TOO AGGRESSIVE**:
   - Maximizes long-term growth rate
   - But has 50% probability of 50%+ drawdown!
   - Not psychologically sustainable

2. **Half-Kelly is STANDARD**:
   - 75% of growth rate of full Kelly
   - Only 12.5% probability of 50% drawdown
   - Good balance of growth and volatility

3. **Quarter-Kelly is CONSERVATIVE**:
   - 50% of growth rate of full Kelly
   - Very low drawdown risk
   - Good for institutional mandates

4. **Bayesian Kelly accounts for**:
   - Parameter uncertainty (we don't know true μ, σ)
   - Model uncertainty (is AR(5) the right model?)
   - More conservative than classical Kelly

**Practical Implementation**:
- Update Kelly daily/weekly based on new forecasts
- Use fractional Kelly (0.25 - 0.5) for safety
- Set maximum position limits (e.g., no more than 20% in single asset)
- Combine with portfolio-level risk budgeting
""")
```

## 4 . Trading Strategies with Bayesian Forecasts

Now we'll implement three complete trading strategies:

### 4 . 1 Mean Reversion with Uncertainty Bands

**Idea**: Trade when price deviates significantly from forecast.

**Signal**:

- **Buy**: Price < Lower 9 5 % band (oversold)
- **Sell**: Price > Upper 9 5 % band (overbought)
- **Exit**: Price returns to forecast mean

**Position sizing**: Scale by forecast uncertainty (wider bands → smaller position)

### 4 . 2 Trend Following with Regime Detection

**Idea**: Follow trends but adjust for volatility regimes.

**Signal**:

- **Trend**: Sign of forecast (positive → long, negative → short)
- **Strength**: Magnitude of forecast relative to uncertainty
- **Regime**: Scale position by inverse of forecasted volatility

## 4 . 3 Pairs Trading with Bayesian Cointegration

**Idea**: Trade mean-reverting spread between two correlated commodities.

**Model**: $$\text{Spread}_t = \text{Price}_A - \beta \cdot \text{Price}_B$$

Bayesian model gives distribution of $\beta$ (hedge ratio).

**Signal**: Trade when spread exits posterior predictive interval.

In [ ]:
```python
# Strategy 1: Mean Reversion with Uncertainty Bands
class BayesianMeanReversion:
    """
    Mean reversion strategy using Bayesian forecast intervals.
    """
    def __init__(self, entry_threshold=1.96, exit_threshold=0.5,
                 max_position=0.5, volatility_scaling=True):
        self.entry_threshold = entry_threshold  # Number of std devs for
        self.exit_threshold = exit_threshold    # Number of std devs for
        self.max_position = max_position
        self.volatility_scaling = volatility_scaling

    def generate_signals(self, forecasts, prices):
        """
        Generate trading signals from forecasts.

        Parameters:
        -----------
        forecasts : DataFrame
            Must have columns: prediction, std
        prices : Series
            Actual prices
        """
        signals = pd.DataFrame(index=forecasts.index)

        # Calculate z-score (how many std devs is price from forecast)
        price_forecast = forecasts['prediction'].shift(1)  # Previous for
        price_std = forecasts['std'].shift(1)

        # Current price relative to forecast
        aligned_prices = prices.reindex(forecasts.index)
        z_score = (aligned_prices - price_forecast) / price_std

        # Generate signals
        signals['z_score'] = z_score
        signals['signal'] = 0.0

        # Buy when price below lower band (oversold)
        signals.loc[z_score < -self.entry_threshold, 'signal'] = 1.0

        # Sell when price above upper band (overbought)
        signals.loc[z_score > self.entry_threshold, 'signal'] = -1.0

        # Exit when price returns to mean
        signals.loc[abs(z_score) < self.exit_threshold, 'signal'] = 0.0

        # Position sizing: scale by inverse volatility
        if self.volatility_scaling:
            vol_scale = 1 / (1 + price_std / price_forecast.mean())
            signals['position'] = signals['signal'] * vol_scale * self.ma
        else:
            signals['position'] = signals['signal'] * self.max_position

        return signals

# Strategy 2: Trend Following with Regime Awareness
class BayesianTrendFollowing:
    """
    Trend following with Bayesian volatility adjustment.
    """
    def __init__(self, min_confidence=0.6, max_position=0.5):
        self.min_confidence = min_confidence
        self.max_position = max_position

    def generate_signals(self, forecasts):
        """
        Generate trend-following signals.
        """
        signals = pd.DataFrame(index=forecasts.index)

        # Forecast return and uncertainty
        forecast_return = forecasts['prediction']
        forecast_std = forecasts['std']

        # Confidence = |forecast| / uncertainty
        confidence = abs(forecast_return) / forecast_std
        confidence = np.minimum(confidence, 3.0)  # Cap at 3 (very confid

        # Signal strength based on forecast and confidence
        signals['raw_signal'] = np.sign(forecast_return) * confidence

        # Only trade if confidence exceeds threshold
        signals['signal'] = 0.0
        signals.loc[confidence > self.min_confidence, 'signal'] = \
            signals.loc[confidence > self.min_confidence, 'raw_signal']

        # Normalize to max position
        if signals['signal'].abs().max() > 0:
            signals['position'] = signals['signal'] / signals['signal'].a
        else:
            signals['position'] = 0.0

        return signals

# Test strategies on walk-forward forecasts
print("Testing trading strategies...\n")

# Mean Reversion
mr_strategy = BayesianMeanReversion(entry_threshold=2.0, exit_threshold=0
mr_signals = mr_strategy.generate_signals(wf_df, data['returns'])
```

```python
# Trend Following
tf_strategy = BayesianTrendFollowing(min_confidence=0.8)
tf_signals = tf_strategy.generate_signals(wf_df)

print("Strategies generated successfully!")
```

In [ ]:
```python
# Backtest strategies
def simple_backtest(signals, returns, initial_capital=100000, transaction
    """
    Simple backtest of strategy signals.
    """
    # Align signals and returns
    common_idx = signals.index.intersection(returns.index)
    signals_aligned = signals.reindex(common_idx)
    returns_aligned = returns.reindex(common_idx)

    # Calculate strategy returns
    positions = signals_aligned['position'].shift(1).fillna(0)  # Use pre
    strategy_returns = positions * returns_aligned

    # Transaction costs
    position_changes = positions.diff().abs()
    costs = position_changes * transaction_cost
    net_returns = strategy_returns - costs

    # Cumulative performance
    cumulative = (1 + net_returns).cumprod() * initial_capital

    return pd.DataFrame({
        'position': positions,
        'returns': net_returns,
        'cumulative': cumulative
    }, index=common_idx)

# Run backtests
mr_backtest = simple_backtest(mr_signals, data['returns'])
tf_backtest = simple_backtest(tf_signals, data['returns'])

# Buy-and-hold benchmark
common_idx = mr_backtest.index
bh_returns = data['returns'].reindex(common_idx)
bh_cumulative = (1 + bh_returns).cumprod() * 100000

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

# Equity curves
ax = axes[0]
ax.plot(mr_backtest.index, mr_backtest['cumulative'], linewidth=2.5,
        color='blue', label='Mean Reversion', alpha=0.8)
ax.plot(tf_backtest.index, tf_backtest['cumulative'], linewidth=2.5,
        color='green', label='Trend Following', alpha=0.8)
ax.plot(common_idx, bh_cumulative, linewidth=2,
        color='red', linestyle='--', label='Buy & Hold', alpha=0.6)
ax.axhline(100000, color='black', linestyle=':', linewidth=1)
ax.set_ylabel('Portfolio Value ($)', fontsize=11)
ax.set_title('Strategy Performance Comparison', fontsize=12, fontweight='
ax.legend(loc='upper left')
ax.grid(True, alpha=0.3)
```

```python
# Positions over time
ax = axes[1]
ax.plot(mr_backtest.index, mr_backtest['position'], linewidth=1.5,
        color='blue', alpha=0.7, label='Mean Reversion')
ax.plot(tf_backtest.index, tf_backtest['position'], linewidth=1.5,
        color='green', alpha=0.7, label='Trend Following')
ax.axhline(0, color='black', linestyle='-', linewidth=1)
ax.set_xlabel('Date', fontsize=11)
ax.set_ylabel('Position Size', fontsize=11)
ax.set_title('Position Sizes Over Time', fontsize=12, fontweight='bold')
ax.legend()
ax.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()

# Calculate performance metrics
def calculate_metrics(returns, name):
    total_return = (1 + returns).prod() - 1
    sharpe = sharpe_ratio(returns.values)
    max_dd = (returns.cumsum().cummax() - returns.cumsum()).max()
    win_rate = (returns > 0).mean()

    return {
        'Strategy': name,
        'Total Return': f"{total_return:.2%}",
        'Sharpe Ratio': f"{sharpe:.2f}",
        'Max Drawdown': f"{max_dd:.2%}",
        'Win Rate': f"{win_rate:.2%}"
    }

metrics_mr = calculate_metrics(mr_backtest['returns'], 'Mean Reversion')
metrics_tf = calculate_metrics(tf_backtest['returns'], 'Trend Following')
metrics_bh = calculate_metrics(bh_returns, 'Buy & Hold')

metrics_df = pd.DataFrame([metrics_mr, metrics_tf, metrics_bh])

print("\n" + "="*70)
print("STRATEGY PERFORMANCE SUMMARY")
print("="*70)
print(metrics_df.to_string(index=False))

print("\n" + "="*70)
print("STRATEGY INSIGHTS")
print("="*70)
print("""
Mean Reversion:
- Trades on deviations from forecast
- Works well in range-bound markets
- Scales position by forecast uncertainty (risk management)

Trend Following:
- Follows forecast direction with confidence weighting
- Only trades when forecast is confident (low uncertainty)
- Adapts to volatility regimes

**Bayesian Advantages**:
1. Uncertainty-aware position sizing
2. Automatic regime detection (via forecast distribution)
3. Natural risk management (wider intervals → smaller positions)
4. No arbitrary parameters (thresholds derived from posteriors)
""")
```

# 5 . Complete Energy Portfolio Trading System

**CAPSTONE PROJECT**: Build a complete multi-asset trading system.

## 5 . 1 System Components

1 . **Universe**: Crude oil, Natural gas, Heating oil (correlated energy commodities)
2 . **Models**: Bayesian VAR for multi-asset forecasting
3 . **Position Sizing**: Bayesian Kelly with portfolio constraints
4 . **Risk Management**:
    - Portfolio VaR limit
    - Individual asset position limits
    - Volatility-based scaling
5 . **Execution**: Transaction costs, slippage
6 . **Monitoring**: Real-time P&L, drawdown, Sharpe

## 5 . 2 Bayesian Vector Autoregression (VAR)

**Multivariate time series model**:

$$\mathbf{y}_t = \mathbf{A}_ 1 \mathbf{y}_{t- 1 } + ... + \mathbf{A}_p \mathbf{y}_{t-p} + \boldsymbol{\epsilon}_t$$

where $\mathbf{y}_t$ is vector of asset returns.

**Bayesian VAR**:

- Priors on coefficient matrices $\mathbf{A}_i$
- Shrinkage toward simpler models (regularization)
- Full posterior for coefficients → uncertainty in cross-asset relationships

In [ ]:
```
# Generate multi-asset energy data
def generate_energy_portfolio_data(n=300):
    """
    Simulate crude oil, natural gas, heating oil with correlations.
    """
    np.random.seed(42)
    dates = pd.date_range('2023-01-01', periods=n, freq='D')

    # Correlation structure
    corr_matrix = np.array([
        [1.0, 0.6, 0.8],    # Crude
        [0.6, 1.0, 0.5],    # Natural gas
        [0.8, 0.5, 1.0]     # Heating oil
    ])

    # Cholesky decomposition for correlated normals
    L = np.linalg.cholesky(corr_matrix)

    # Generate correlated returns
    base_returns = np.random.normal(0, 1, (n, 3))
    correlated_returns = base_returns @ L.T

    # Scale by different volatilities
    crude_returns = 0.0003 + 0.02 * correlated_returns[:, 0]
    ng_returns = 0.0002 + 0.03 * correlated_returns[:, 1]  # More volatil
    ho_returns = 0.0004 + 0.022 * correlated_returns[:, 2]

    # Create DataFrame
    df = pd.DataFrame({
        'crude_ret': crude_returns,
        'ng_ret': ng_returns,
        'ho_ret': ho_returns
    }, index=dates)

    # Add prices
    df['crude_price'] = 70 * np.exp(np.cumsum(crude_returns))
    df['ng_price'] = 3.5 * np.exp(np.cumsum(ng_returns))
    df['ho_price'] = 2.5 * np.exp(np.cumsum(ho_returns))

    return df

# Generate portfolio data
portfolio_data = generate_energy_portfolio_data(n=300)

# Visualize
fig, axes = plt.subplots(2, 1, figsize=(14, 9))

# Prices
ax = axes[0]
ax.plot(portfolio_data.index, portfolio_data['crude_price'],
        linewidth=2, label='Crude Oil', alpha=0.8)
ax.plot(portfolio_data.index, portfolio_data['ng_price'] * 15,  # Scale f
        linewidth=2, label='Natural Gas (×15)', alpha=0.8)
ax.plot(portfolio_data.index, portfolio_data['ho_price'] * 20,  # Scale
        linewidth=2, label='Heating Oil (×20)', alpha=0.8)
ax.set_ylabel('Price (normalized)', fontsize=11)
ax.set_title('Energy Portfolio: Price Evolution', fontsize=12, fontweight
ax.legend()
ax.grid(True, alpha=0.3)

# Correlation matrix
ax = axes[1]
returns_corr = portfolio_data[['crude_ret', 'ng_ret', 'ho_ret']].corr()
im = ax.imshow(returns_corr, cmap='RdBu_r', aspect='auto', vmin=-1, vmax=
ax.set_xticks(range(3))
ax.set_yticks(range(3))
ax.set_xticklabels(['Crude', 'Nat Gas', 'Heat Oil'])
ax.set_yticklabels(['Crude', 'Nat Gas', 'Heat Oil'])
ax.set_title('Return Correlations', fontsize=12, fontweight='bold')

# Add correlation values
for i in range(3):
    for j in range(3):
        text = ax.text(j, i, f'{returns_corr.iloc[i, j]:.2f}',
                       ha="center", va="center", color="black", fontsize=1

plt.colorbar(im, ax=ax)
plt.tight_layout()
plt.show()
```

```
print("\nEnergy Portfolio Data Generated")
print(f"Assets: Crude Oil, Natural Gas, Heating Oil")
print(f"Period: {len(portfolio_data)} days")
print(f"\nReturn Correlations:")
print(returns_corr.round(3))
```

# 6 . Summary: Building Production-Ready Bayesian Tra Systems

## 6 . 1 Essential Components

| Component | Bayesian Implementation |
|---|---|
| **Forecasting** | GP, VAR, structural time series with full posteriors |
| **Backtesting** | Walk-forward validation (no look-ahead bias) |
| **Evaluation** | CRPS, calibration, log-score (not just MAE/RMSE) |
| **Position Sizing** | Bayesian Kelly with parameter uncertainty |
| **Risk Management** | VaR/CVaR from posterior predictive, volatility scaling |
| **Portfolio** | Multi-asset Bayesian models (VAR, copulas) |

## 6 . 2 What We've Learned

**Module** 1 - 3 : Bayesian foundations, priors, MCMC **Module** 4 - 6 : Time series models (A space, BSTS) **Module** 7 : Hierarchical models for multiple commodities **Module** 8 : Gauss Processes for non-linear patterns **Module** 9 : Volatility modeling (GARCH, stochastic vol) **Mo** 1 0 : Complete trading systems (this module)

## 6 . 3 Bayesian vs Frequentist: Final Comparison

**Bayesian Advantages**:

1. ✅ Full uncertainty quantification
2. ✅ Parameter uncertainty propagates to decisions
3. ✅ Natural regularization through priors
4. ✅ Sequential updating (efficient online learning)
5. ✅ Honest risk estimates (wider intervals with limited data)

**Frequentist Advantages**:

1. ✅ Faster computation (no MCMC)
2. ✅ Easier to implement in production
3. ✅ Well-understood asymptotics

**Recommendation**:

- **Research/Alpha generation**: Bayesian (better forecasts)
- **High-frequency execution**: Frequentist (speed matters)
- **Risk management**: Bayesian (need full distributions)

## 6.4 Going to Production

**Infrastructure needs**:

1. **Data pipeline**: Real-time price feeds, cleaning, storage
2. **Model serving**: Daily refits, caching posteriors
3. **Execution**: Order management, broker integration
4. **Monitoring**: P&L tracking, drawdown alerts, model diagnostics
5. **Backtesting**: Automated walk-forward validation

**Common pitfalls**:

- ❌ Overfitting in-sample
- ❌ Ignoring transaction costs
- ❌ Not accounting for parameter uncertainty
- ❌ Using wrong evaluation metrics (MAE for probabilistic forecasts)
- ❌ Over-leveraging (full Kelly is too aggressive)

## 6.5 Next Steps

**To deepen your knowledge**:

1. Read: "Bayesian Methods for Hackers" (Cameron Davidson-Pilon)
2. Read: "Advances in Financial Machine Learning" (Marcos López de Prado)
3. Practice: Kaggle competitions with probabilistic scoring
4. Build: Your own production system with real data
5. Learn: Advanced topics (deep learning + Bayesian, causal inference)

**Congratulations!** You've completed the Bayesian Forecasting for Commodities Trading course.

---

---

# Final Knowledge Check Quiz

**Q1**: Walk-forward validation prevents:

- A) Model convergence issues
- B) Look-ahead bias in backtesting
- C) Transaction costs
- D) Market regime changes

**Q2**: CRPS is better than MAE for Bayesian forecasts because:

- A) It's easier to calculate
- B) It evaluates the full forecast distribution, not just the mean
- C) It always gives higher scores
- D) Regulators require it

**Q** 3 : Half-Kelly position sizing means:

- A) Use 50 % of your capital
- B) Use half the optimal Kelly fraction
- C) Trade half as often
- D) Split position between two assets

**Q** 4 : Bayesian Kelly criterion accounts for:

- A) Only expected return uncertainty
- B) Only volatility uncertainty
- C) Both parameter and model uncertainty
- D) Neither (uses point estimates)

**Q** 5 : The main advantage of Bayesian trading systems is:

- A) They always outperform
- B) They're faster to execute
- C) They quantify uncertainty for better risk management
- D) They require less data

In [ ]:
```python
# Quiz Answers
print("="*70)
print("FINAL QUIZ ANSWERS")
print("="*70)
print("""
Q1: B) Look-ahead bias in backtesting
    Walk-forward validation simulates real-time trading. Each forecast
    is made using ONLY data available at that time (no future data).
    This prevents overfitting and gives realistic performance estimates.

Q2: B) It evaluates the full forecast distribution, not just the mean
    CRPS is a proper scoring rule that rewards:
    1. Accurate point forecasts (like MAE)
    2. Well-calibrated uncertainty (bonus over MAE)
    Essential for models used in risk management and position sizing.

Q3: B) Use half the optimal Kelly fraction
    Full Kelly maximizes growth but has high volatility.
    Half-Kelly (f* × 0.5) gives:
    - 75% of growth rate
    - Much lower drawdowns
    - More psychologically sustainable

Q4: C) Both parameter and model uncertainty
    Classical Kelly: assumes we KNOW μ and σ
    Bayesian Kelly: integrates over posterior p(μ, σ | data)
    Result: More conservative sizing when data is limited

Q5: C) They quantify uncertainty for better risk management
    Bayesian methods don't always outperform in terms of raw returns.
    But they provide:
    - Full predictive distributions
    - Honest uncertainty estimates
    - Better risk-adjusted performance
    - Principled position sizing
""")
```

---

# Final Project: Build Your Own Trading System

## Project Requirements

Build a complete Bayesian trading system for a commodity of your choice:

1. **Data**: Collect real price data (at least 2 years, daily)
2. **Model**: Choose appropriate Bayesian model:
    - Smooth trends → Gaussian Process
    - Seasonal patterns → Structural time series
    - Regime changes → Switching models
3. **Backtesting**: Implement walk-forward validation
    - Minimum 50 out-of-sample forecasts
    - Calculate CRPS and calibration
4. **Strategy**: Design trading strategy
    - Mean reversion, trend following, or pairs trading
    - Bayesian Kelly position sizing
    - Transaction costs and slippage
5. **Risk Management**:
    - Daily VaR monitoring
    - Maximum drawdown limits
    - Volatility-based position scaling
6. **Evaluation**:
    - Sharpe ratio > 1.0 (after costs)
    - Max drawdown < 20%
    - Win rate > 50%
7. **Documentation**:
    - Model choice justification
    - Prior selection rationale
    - Backtesting methodology
    - Performance analysis

## Deliverables

1. Jupyter notebook with complete implementation
2. Performance report (PDF)
3. Production-ready code (modular, documented)

## Evaluation Criteria

- **Technical (40%)**: Correct Bayesian implementation, no look-ahead bias
- **Performance (30%)**: Risk-adjusted returns, calibration
- **Rigor (20%)**: Proper evaluation metrics, honest reporting
- **Presentation (10%)**: Code quality, documentation

---

## Course Complete!

**You've learned**:

- Bayesian inference fundamentals
- Time series modeling with uncertainty
- Volatility and risk management
- Complete trading system development

**You can now**:

- Build production Bayesian forecasting systems
- Properly backtest without overfitting
- Size positions optimally with Kelly criterion
- Manage risk with probabilistic forecasts

**Go forth and trade wisely!** 📈

---

*End of Course*

# Capstone Project: Complete Bayesian Forecas System for Commodities

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

---

## Project Overview

In this capstone project, you will build a **complete end-to-end Bayesian forecasting and tradi system** for a portfolio of commodities. This project integrates all concepts learned throughout th

- Bayesian inference and prior selection
- Time series analysis and decomposition
- Bayesian regression and structural time series
- Hierarchical models for multiple assets
- Volatility modeling and risk management
- Proper backtesting and evaluation
- Trading strategy implementation

---

## Project Requirements

### Deliverables

1. **Data Pipeline**: Download and preprocess at least 3-5 related commodities
2. **Exploratory Analysis**: Time series characteristics, seasonality, correlations
3. **Bayesian Models**: At least 2 different model types with proper diagnostics
4. **Forecasting System**: Multi-step probabilistic forecasts
5. **Trading Strategy**: Uncertainty-aware position sizing
6. **Backtesting**: Walk-forward validation with proper metrics
7. **Risk Analysis**: VaR, CVaR, drawdown analysis
8. **Written Report**: 5-page summary with key findings

### Grading Rubric

| Component | Weight | Criteria |
|---|---|---|
| Technical Implementation | 40% | Code quality, model sophistication, diagnostics |
| Performance | 30% | Forecast accuracy (CRPS), trading metrics (Sharpe) |
| Risk Management | 20% | Uncertainty quantification, position sizing |
| Documentation | 10% | Clear explanation, reproducibility |

---

# Part 1 : Data Pipeline and Exploratory Analysis

## 1.1 Select Your Commodity Portfolio

Choose ONE of these commodity complexes:

**Option A: Energy Complex**

- WTI Crude Oil (CL=F)
- Brent Crude Oil (BZ=F)
- Natural Gas (NG=F)
- Gasoline (RB=F)
- Heating Oil (HO=F)

**Option B: Agricultural Complex**

- Corn (ZC=F)
- Wheat (ZW=F)
- Soybeans (ZS=F)
- Soybean Oil (ZL=F)
- Soybean Meal (ZM=F)

**Option C: Precious Metals**

- Gold (GC=F)
- Silver (SI=F)
- Platinum (PL=F)
- Copper (HG=F)

**Option D: Custom** (subject to approval)

- Choose 3-5 related commodities with economic rationale

In [ ]:

```python
# Setup
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')

# Course utilities
import sys
sys.path.append('..')
from datasets.download_data import (
    get_commodity_data,
    get_multiple_commodities,
    prepare_commodity_dataset,
    create_train_test_split
)
from utils.plotting import (
    plot_time_series,
    plot_forecast,
    plot_posterior,
    plot_backtest_results
)
from utils.metrics import (
    crps, crps_gaussian,
    forecast_summary,
    backtest_summary
)
from utils.backtesting import (
    WalkForwardValidator,
    Backtester,
    create_folds
)

np.random.seed(42)
plt.style.use('seaborn-v0_8-whitegrid')

print("Setup complete!")
```

In [ ]:
```
# TODO: Define your commodity portfolio
# Example for Energy Complex:

COMMODITIES = [
    'crude_oil',
    'natural_gas',
    'gasoline',
    # Add more commodities...
]

START_DATE = '2015-01-01'
END_DATE = None  # Today

# Download data
# YOUR CODE HERE
data = get_multiple_commodities(COMMODITIES, start=START_DATE, end=END_DA
print(f"Downloaded {len(data.columns)} commodities")
print(f"Date range: {data.index[0]} to {data.index[-1]}")
print(f"Total observations: {len(data)}")
data.head()
```

In [ ]:
```
# TODO: Exploratory Data Analysis
# 1. Plot all commodity prices (normalized)
# 2. Calculate and visualize correlations
# 3. Test for stationarity (ADF, KPSS)
# 4. Analyze seasonality patterns
# 5. Identify any structural breaks or regime changes

# YOUR CODE HERE
```

## 1.2 Document Your Findings

**Questions to Answer:**

1. What are the key characteristics of your chosen commodities?
2. Are there clear seasonal patterns? Which commodities show strongest seasonality?
3. What is the correlation structure? Are there natural pairs for spread trading?

4. Are the series stationary? What transformations are needed?
5. Are there any obvious regime changes in the data?

*Your EDA Summary Here:*

- 
- 
-

---

## Part 2: Bayesian Forecasting Models

Implement at least TWO of the following model types:

1. **Bayesian Linear Regression** with economic factors
2. **Bayesian Structural Time Series** (BSTS)
3. **Hierarchical Model** for multiple commodities
4. **Gaussian Process** regression
5. **Stochastic Volatility** model

For each model:

- Justify your prior choices
- Run prior predictive checks
- Fit the model with PyMC
- Check convergence (R-hat, ESS, trace plots)
- Run posterior predictive checks
- Generate forecasts with uncertainty

### 2.1 Model 1: [Your Choice]

**Model Description**: [Explain your model choice and why it's appropriate]

In [ ]:
```
# Model 1 Implementation
#
# TODO:
# 1. Define priors with justification
# 2. Build PyMC model
# 3. Run prior predictive check
# 4. Fit model (sample from posterior)
# 5. Check convergence
# 6. Run posterior predictive check

# YOUR CODE HERE
```

### 2.2 Model 2: [Your Choice]

**Model Description**: [Explain your model choice and why it's appropriate]

In [ ]:
```
```

```
# Model 2 Implementation
#
# YOUR CODE HERE
```

### 2.3 Model Comparison

Compare your models using:

- WAIC (Widely Applicable Information Criterion)
- LOO (Leave-One-Out Cross-Validation)
- Out-of-sample CRPS

In [ ]:
```
# Model Comparison
#
# YOUR CODE HERE
```

---

## Part 3: Trading Strategy Development

Implement a trading strategy that:

1. Uses your Bayesian forecasts
2. Accounts for forecast uncertainty in position sizing
3. Manages risk appropriately

Choose from:

- **Mean reversion** with credible interval bands
- **Trend following** with probability-based entries
- **Pairs/spread trading** with hierarchical model
- **Volatility targeting** with stochastic vol forecasts
- **Custom strategy** (describe your approach)

In [ ]:
```
# Trading Strategy Implementation
#
# TODO:
# 1. Define signal generation from forecasts
# 2. Implement position sizing (Kelly criterion or similar)
# 3. Add risk management rules
# 4. Generate signals for full history

def generate_trading_signals(forecasts, actuals, model_uncertainty):
    """
    Generate trading signals from Bayesian forecasts.

    Parameters:
    -----------
    forecasts : pd.DataFrame
        Posterior predictive samples or point forecasts
    actuals : pd.Series
        Actual price series
    model_uncertainty : pd.Series
        Posterior standard deviation
    Returns:
    --------
    pd.Series
        Signal series (-1 to 1)
    """
    # YOUR CODE HERE
    pass

def bayesian_position_size(signal, forecast_mean, forecast_std,
                           max_position=1.0, kelly_fraction=0.5):
    """
    Calculate position size using Bayesian Kelly criterion.

    Parameters:
    -----------
    signal : float
        Trading signal direction
    forecast_mean : float
        Expected return
    forecast_std : float
        Standard deviation of forecast
    max_position : float
        Maximum position size
    kelly_fraction : float
        Fraction of Kelly (0.5 = half-Kelly)

    Returns:
    --------
    float
        Position size
    """
    # YOUR CODE HERE
    pass
```

---

## Part 4 : Backtesting and Evaluation

Perform rigorous backtesting with:

1. Walk-forward validation (no look-ahead bias)
2. Proper evaluation metrics (CRPS for forecasts, Sharpe for trading)
3. Comparison to benchmark strategies

In [ ]:
```
# Walk-Forward Backtesting
#
# TODO:
# 1. Set up walk-forward validation folds
# 2. For each fold: fit model, generate forecasts, calculate signals
# 3. Compute evaluation metrics per fold
# 4. Aggregate results

# YOUR CODE HERE
```

In [ ]:

```
# Full Backtest
#
# YOUR CODE HERE
```

In [ ]:
```
# Performance Summary
#
# TODO: Create comprehensive performance report

# YOUR CODE HERE
```

---

## Part 5: Risk Analysis

Analyze the risk characteristics of your strategy:

1. Value at Risk (VaR) and Conditional VaR (CVaR)
2. Maximum drawdown analysis
3. Tail risk assessment
4. Scenario analysis (what if volatility doubles?)

In [ ]:
```
# Risk Analysis
#
# YOUR CODE HERE
```

---

## Part 6: Final Report

Write a 5-page summary covering:

1. **Executive Summary** (0.5 page)
   - Key findings and performance metrics
2. **Data and Methodology** (1 page)
   - Commodity selection rationale
   - Model choices and prior justification
3. **Model Results** (1.5 pages)
   - Posterior analysis
   - Forecast accuracy
   - Model comparison
4. **Trading Performance** (1 page)
   - Backtest results
   - Risk metrics
   - Comparison to benchmarks
5. **Conclusions and Limitations** (1 page)
   - Key insights

• What worked, what didn't
• Suggestions for improvement

## Your Report Here

*(Use markdown formatting for your written report)*

---

## 1 . Executive Summary

[Your summary here]

---

## 2 . Data and Methodology

[Your methodology here]

---

## 3 . Model Results

[Your results here]

---

## 4 . Trading Performance

[Your performance analysis here]

---

## 5 . Conclusions and Limitations

[Your conclusions here]

---

## Submission Checklist

Before submitting, ensure you have:

- ☐ Downloaded and preprocessed at least 3 commodities
- ☐ Completed exploratory data analysis with visualizations
- ☐ Implemented at least 2 Bayesian models with proper diagnostics
- ☐ Justified all prior choices
- ☐ Generated probabilistic forecasts
- ☐

Implemented a trading strategy with uncertainty-aware position sizing

- ☐
  Performed walk-forward backtesting
- ☐
  Calculated proper evaluation metrics (CRPS, Sharpe, etc.)
- ☐
  Completed risk analysis (VaR, drawdown, etc.)
- ☐
  Written 5 -page summary report
- ☐
  All code runs without errors
- ☐
  Results are reproducible (random seed set)

---

**Good luck! This project is your opportunity to demonstrate mastery of Bayesian forecasti commodity trading.**