PYTHON FOR QUANT TRADERS · PART II

Math Tools — เครื่องมือคณิตศาสตร์สำหรับ Quant

บทที่ 11–17: Statistics · Time Series · OLS Regression · Cointegration · Optimization · Linear Programming · Signals

จบ Part นี้ → สร้าง pair trading strategy ตั้งแต่ต้นจนได้ signal พร้อม trade

สิ่งที่จะทำได้เมื่อจบ PART II

- หาคู่หุ้นที่ cointegrate กัน ด้วย ADF test และ Johansen test
- ประมาณ hedge ratio β ด้วย OLS regression (statsmodels)
- สร้าง spread และคำนวณ z-score สำหรับ entry/exit signal
- Maximize Sharpe ratio ด้วย `scipy.optimize`
- กำหนด portfolio weight constraints ด้วย Linear Programming

ทุกอย่างนี้คือ core engine ของ Statistical Arbitrage

Running Example ตลอด Part II — Pair Trading: BTC Perp Bybit ↔ BTC Perp Binance

เราจะสร้าง pair trading strategy ทีละขั้นตอนตลอด 7 บท:
บท 11 → ตรวจสอบ distribution ของ spread · บท 12 → ทดสอบ stationarity · บท 13 → ประมาณ β ด้วย OLS
บท 14 → ยืนยัน cointegration · บท 15 → optimize position size · บท 16 → ใส่ capital constraint · บท 17 → สร้าง signal pipeline

บทที่ 11: Statistics & Probability เชิงลึก

แก่นของบทนี้

สถิติใน Part 0 เป็นแนวคิด — บทนี้แปลงเป็น Python จริง ด้วย `scipy.stats` และ `numpy` เน้น 3 เรื่องที่ Quant ใช้จริง: hypothesis testing (กลยุทธ์นี้ดีจริงหรือแค่โชค?), VaR/CVaR (ความเสี่ยงสูงสุดคือเท่าไหร่?), และ fat tail (ทำไมต้องระวัง Black Swan?)

11.1 Hypothesis Testing — กลยุทธ์ดีจริงหรือแค่โชค?

สมมติฐาน: H_0 (null) = "กลยุทธ์ไม่ดีกว่าสุ่ม" vs H_1 = "กลยุทธ์ดีกว่าสุ่มจริงๆ"

```python
from scipy import stats
import numpy as np

# ผลตอบแทนรายวัน 252 วัน จากกลยุทธ์
np.random.seed(42)
strategy_returns = np.random.normal(loc=0.0008, scale=0.015, size=252)

# t-test: H0 = mean return = 0 (ไม่ดีกว่าสุ่ม)
t_stat, p_value = stats.ttest_1samp(strategy_returns, popmean=0)

print(f"Mean Return: {strategy_returns.mean():.4f}
({strategy_returns.mean()*252:.1%}/year)")
print(f"t-statistic: {t_stat:.3f}")
print(f"p-value:     {p_value:.4f}")

if p_value < 0.05:
    print("✓ มีนัยสำคัญ — กลยุทธ์น่าจะดีกว่าสุ่ม (p < 5%)")
else:
    print("✗ ไม่มีนัยสำคัญ — อาจเป็นแค่โชค")
```

▶ OUTPUT

```
Mean Return: 0.0008 (20.2%/year)
t-statistic: 0.893
p-value:     0.3726
✗ ไม่มีนัยสำคัญ — อาจเป็นแค่โชค
```

ทำไม Return ดูดีแต่ p-value ยังสูง?

20% ต่อปีดูน่าสนใจ แต่ถ้า σ สูง (1.5%/วัน) — noise มากกว่า signal มาก
ต้องมีข้อมูลมากพอ (หลาย 100 วัน) ถึงจะ "แยก" signal ออกจาก noise ได้
$t = \frac{\bar{r}}{\sigma/\sqrt{n}}$ → ยิ่ง n มาก t ยิ่งใหญ่ → p-value ยิ่งเล็ก

11.2 VaR และ CVaR — ความเสี่ยงสูงสุดคือเท่าไหร่?

$$\mathrm{VaR}_{95\%} = \mathrm{percentile}_{5\%}(r) \qquad \mathrm{CVaR}_{95\%} = E[r \mid r \leq \mathrm{VaR}_{95\%}]$$

```
# VaR = ขาดทุนสูงสุดใน 95% ของวันทั้งหมด
# CVaR = ขาดทุนเฉลี่ยในวันที่แย่ที่สุด 5%

returns = strategy_returns  # จากข้างบน
capital = 1_000_000         # ทุน 1 ล้านบาท

var_95  = np.percentile(returns, 5)
cvar_95 = returns[returns <= var_95].mean()

print(f"VaR  95%: {var_95:.2%}  → ขาดทุนไม่เกิน {abs(var_95)*capital:,.0f} บาท/วัน (95% ของเวลา)")
print(f"CVaR 95%: {cvar_95:.2%} → เฉลี่ยขาดทุน  {abs(cvar_95)*capital:,.0f} บาท ในวันที่แย่ที่สุด 5%")
```

> **หมายเหตุ CVaR: ต้องการ sample เพียงพอ**
>
> CVaR 95% ใช้เฉพาะ return ที่อยู่ใน worst 5% — ถ้ามีข้อมูล 100 วัน จะใช้แค่ 5 วัน
> ถ้าข้อมูลน้อยกว่า 60 วัน CVaR อาจไม่น่าเชื่อถือ ควรใช้ `if len(tail) >= 5: cvar = tail.mean()`
> หรือเพิ่ม `n_tail = max(1, int(len(returns) * 0.05))` เพื่อรับประกัน minimum sample

11.3 Fat Tail — ทำไม Normal Distribution ถึงอันตราย?

```
from scipy import stats

# เปรียบ Normal กับ t-distribution (fat tail)
normal_99 = stats.norm.ppf(0.01)   # -2.33σ
t5_99     = stats.t.ppf(0.01, df=5) # -3.36σ (df=5 = fat tail)

print(f"Normal 1% VaR: {normal_99:.2f}σ")
print(f"t(df=5) 1% VaR: {t5_99:.2f}σ")
print(f"ใช้ Normal จะ underestimate ความเสี่ยง {t5_99/normal_99:.1f}x")
```

> **อย่าเชื่อ Normal Distribution สำหรับ crypto และช่วง crisis**
>
> BTC crash 50%+ เกิดหลายครั้ง — ถ้าใช้ Normal Distribution จะบอกว่า "แทบเป็นไปไม่ได้" (เกิน 10σ)
> ในความจริง: ราคาสินทรัพย์มี "fat tail" = เหตุการณ์สุดขั้วเกิดบ่อยกว่า Normal คาด
> **ใช้ t-distribution, Cornish-Fisher, หรือ historical simulation แทน Normal สำหรับ risk management**

ตัวอย่าง: สรุป Return Statistics แบบมืออาชีพ

```
def return_stats(returns, name="Strategy", capital=1_000_000):
    """สรุป statistics สำหรับ return series"""
    r = np.array(returns)
    ann = 252

    mean_daily  = r.mean()                  # return เฉลี่ยต่อวัน
    std_daily   = r.std()                   # ความผันผวนต่อวัน
    sharpe      = mean_daily / std_daily * np.sqrt(ann)  # Sharpe รายปี (คูณ √
252)
    skew        = stats.skew(r)             # ความเบ้ (ลบ = หางซ้ายยาว = อันตราย)
    kurt        = stats.kurtosis(r)         # excess kurtosis (บวก = fat
tail)
    var95       = np.percentile(r, 5)       # จุดตัด worst 5% ของวัน
    cvar95      = r[r <= var95].mean()      # ค่าเฉลี่ยของวันที่แย่กว่า VaR

    equity      = capital * np.cumprod(1 + r)        # เส้นเงินทุนสะสมรายวัน
    rolling_max = np.maximum.accumulate(equity)      # จุดสูงสุดที่เคยไปถึง ณ แต่ละวัน
    max_dd      = ((equity - rolling_max) / rolling_max).min()  # % ร่วงจากยอดที่
ลึกที่สุด

    print(f"=== {name} ===")
    print(f"Annual Return: {mean_daily*ann:.1%}   Sharpe: {sharpe:.2f}")
    print(f"Annual Vol:    {std_daily*np.sqrt(ann):.1%}   Max DD: {max_dd:.
1%}")
    print(f"Skewness: {skew:.2f}  Kurtosis: {kurt:.2f}  (Normal: 0, 0)")
    print(f"VaR 95%: {var95:.2%}  CVaR 95%: {cvar95:.2%}")

return_stats(strategy_returns)
```

▶ แบบฝึกหัด: เปรียบ Sharpe ของสองกลยุทธ์

```
# กลยุทธ์ A: return ต่ำ volatility ต่ำ
ret_a = np.random.normal(0.0003, 0.005, 252)

# กลยุทธ์ B: return สูง volatility สูง
ret_b = np.random.normal(0.002, 0.04, 252)

sharpe_a = ret_a.mean() / ret_a.std() * np.sqrt(252)
sharpe_b = ret_b.mean() / ret_b.std() * np.sqrt(252)

print(f"Sharpe A: {sharpe_a:.2f}  ({ret_a.mean()*252:.0%}/yr,
σ={ret_a.std()*np.sqrt(252):.0%}/yr)")
print(f"Sharpe B: {sharpe_b:.2f}  ({ret_b.mean()*252:.0%}/yr,
σ={ret_b.std()*np.sqrt(252):.0%}/yr)")
```

บทเรียน: B อาจให้ return ต่อปีสูงกว่า แต่ถ้า Sharpe ต่ำกว่า risk ต่อหน่วย return แย่กว่า

บทที่ 12: Time Series Basics

แก่นของบทนี้

ข้อมูลราคาเรียงตามเวลา มีคุณสมบัติพิเศษที่ข้อมูลทั่วไปไม่มี: แต่ละจุดสัมพันธ์กับจุดก่อนหน้า (autocorrelation) และอาจ "ล่องลอย" ไปเรื่อยๆ โดยไม่กลับ (non-stationary) การเข้าใจ 2 เรื่องนี้คือกุญแจของ Statistical Arbitrage ทั้งหมด

12.1 Stationarity — ล่องลอยหรือวนกลับ?

Stationary series: mean และ variance คงที่ตลอดเวลา — "วนกลับหาค่ากลาง"
Non-stationary series: mean เปลี่ยนตามเวลา — "ล่องลอยไม่มีทิศทาง" (random walk)

```
import numpy as np

# Non-stationary: Random Walk (ราคา BTC)
np.random.seed(0)
random_walk = np.cumsum(np.random.normal(0, 1, 300))

# Stationary: Spread ระหว่างสองตลาด
spread = np.random.normal(0, 1, 300)  # OU-like (mean-reverting)

# สังเกต: random_walk drift ออกไปเรื่อยๆ
#          spread วนอยู่รอบ 0
print(f"Random Walk range: {random_walk.min():.1f} to {random_walk.max():.1f}")
print(f"Spread range:      {spread.min():.1f} to {spread.max():.1f}")
```

ทำไม Stationarity ถึงสำคัญสำหรับ Pair Trading?

ถ้า spread ไม่ stationary → ไม่มีแรงดึงกลับ → "z-score สูง" อาจสูงต่อไปเรื่อยๆ → trade แล้วขาดทุนไม่จำกัด
ถ้า spread stationary → มีแรงดึงกลับ → z-score สูงแล้ว "ต้องลง" ในที่สุด → เป็น basis ของ arb
เราต้องพิสูจน์ก่อนเสมอว่า spread เป็น stationary จริง — ไม่ใช่แค่ดูกราฟ

12.2 ADF Test — ทดสอบ Stationarity

Augmented Dickey-Fuller (ADF) test: H_0 = non-stationary (random walk) vs H_1 = stationary

```
from statsmodels.tsa.stattools import adfuller

def test_stationarity(series, name="Series"):
    """รัน ADF test และแสดงผลแบบเข้าใจง่าย"""
    result = adfuller(series, autolag="AIC")
    adf_stat  = result[0]
    p_value   = result[1]
    crit_vals = result[4]

    print(f"\n--- ADF Test: {name} ---")
    print(f"ADF Statistic: {adf_stat:.4f}")
    print(f"p-value:       {p_value:.4f}")
    print(f"Critical (1%): {crit_vals['1%']:.4f}")
    print(f"Critical (5%): {crit_vals['5%']:.4f}")

    if p_value < 0.05:
        print(f"✓ STATIONARY (p={p_value:.3f} < 0.05) — กลับหาค่ากลาง")
    else:
        print(f"✗ NON-STATIONARY (p={p_value:.3f} > 0.05) — random walk")
    return p_value < 0.05

# ทดสอบกับข้อมูลจาก pair
test_stationarity(random_walk, "BTC Price (Random Walk)")
test_stationarity(spread, "BTC Spread (Mean-Reverting)")
```

12.3 Autocorrelation — ราคาวันนี้สัมพันธ์กับเมื่อวานแค่ไหน?

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
import matplotlib.pyplot as plt

fig, axes = plt.subplots(2, 2, figsize=(12, 8))

# ACF/PACF ของ returns (stationary)
plot_acf(spread, lags=30, ax=axes[0][0], title="ACF — Spread (Stationary)")
plot_pacf(spread, lags=30, ax=axes[0][1], title="PACF — Spread")

# ACF/PACF ของ raw price (non-stationary)
plot_acf(random_walk, lags=30, ax=axes[1][0], title="ACF — Price (Non-stationary)")
plot_pacf(random_walk, lags=30, ax=axes[1][1], title="PACF — Price")

plt.tight_layout()
plt.savefig("acf_comparison.png", dpi=150)
plt.show()
```

อ่าน ACF plot อย่างไร?

Pattern ที่เห็น	แปลว่า	ทำอะไรต่อ
ค่าลด exponentially ใน ACF	AR process (mean-reverting) ✓	ใช้เป็น spread ได้
ค่าสูงทุก lag ไม่ลด	Non-stationary (random walk)	ต้อง difference ก่อน
ค่าตัดที่ lag 1 ใน ACF	MA(1) process	ต้องใส่ MA term ใน model

▶ **แบบฝึกหัด: ทดสอบ stationarity ของ spread จริง**

```
import numpy as np
from statsmodels.tsa.stattools import adfuller

# จำลอง Ornstein-Uhlenbeck process (spread จริง)
np.random.seed(1)
n, kappa, mu, sigma = 500, 0.1, 0, 0.5
ou = np.zeros(n)
for t in range(1, n):
    ou[t] = ou[t-1] + kappa*(mu - ou[t-1]) + sigma*np.random.normal()

result = adfuller(ou)
print(f"ADF p-value: {result[1]:.4f}")
print("✓ Stationary" if result[1] < 0.05 else "✗ Non-stationary")

# OU process ควร stationary เสมอ (kappa > 0)
```

บทที่ 13: OLS Regression

แก่นของบทนี้

OLS (Ordinary Least Squares) เป็นเครื่องมือหลักสำหรับประมาณ hedge ratio β — อัตราส่วนที่บอกว่าต้อง long/short ในสัดส่วนเท่าไหร่เพื่อสร้าง spread ที่ stationary สูตร matrix form:

$$\hat{\beta} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

13.1 OLS ด้วย statsmodels

```
import numpy as np
import pandas as pd
import statsmodels.api as sm

# จำลองราคา BTC สองตลาด
np.random.seed(42)
n = 252
btc_bybit   = 65000 + np.cumsum(np.random.normal(0, 100, n))
btc_binance = 65000 + np.cumsum(np.random.normal(0, 100, n)) + 0.003 * btc_bybit

# OLS: log(P_binance) = alpha + beta * log(P_bybit) + epsilon
log_bybit   = np.log(btc_bybit)
log_binance = np.log(btc_binance)

X = sm.add_constant(log_bybit)   # เพิ่ม intercept column
y = log_binance

model  = sm.OLS(y, X).fit()

alpha  = model.params['const']
beta   = model.params['log_bybit']
resid  = model.resid               # epsilon (spread)
r2     = model.rsquared
p_beta = model.pvalues['log_bybit']

print(f"alpha (intercept): {alpha:.6f}")
print(f"beta  (hedge ratio): {beta:.6f}")
print(f"R²:   {r2:.4f}")
print(f"p-value (beta): {p_beta:.6f}")
print(f"\nSpread (residuals) stats:")
print(f"  Mean: {resid.mean():.6f}")
print(f"  Std:  {resid.std():.6f}")
```

13.2 ตีความผล OLS

Output	ความหมาย	ค่าที่ดี
β	Hedge ratio — Long 1 บาท Bybit ต้อง Short β บาท Binance	ใกล้ 1.0 สำหรับ same asset
α	Constant spread ระหว่างสองตลาด	ขึ้นอยู่กับ pair
R^2	สัดส่วน variance ที่ X อธิบายได้	> 0.85 สำหรับ good pair
p-value (β)	มั่นใจแค่ไหนว่า β ≠ 0	< 0.001 สำหรับ cointegrated pair
Residuals	Spread = ส่วนที่ไม่สัมพันธ์กับ X — นี่คือสิ่งที่เราจะ trade	Mean ≈ 0, Stationary

13.3 Rolling OLS — β ที่ Adaptive

β ไม่คงที่ตลอดเวลา ต้องอัปเดตสม่ำเสมอ

```
from statsmodels.regression.rolling import RollingOLS

# Rolling OLS window 60 วัน
window = 60
X = sm.add_constant(log_bybit)

rolling_model = RollingOLS(log_binance, X, window=window).fit()
rolling_beta  = rolling_model.params['log_bybit']

# สร้าง rolling spread
rolling_alpha = rolling_model.params['const']
spread = log_binance - (rolling_alpha + rolling_beta * log_bybit)

print(f"Beta range: {rolling_beta.dropna().min():.4f} to
{rolling_beta.dropna().max():.4f}")
print(f"Spread Std: {spread.std():.6f}")
```

window เลือกอย่างไร?

- **สั้นเกินไป** (เช่น 10 วัน): β volatile มาก, noise สูง → signal ผิดพลาดบ่อย
- **ยาวเกินไป** (เช่น 500 วัน): β ตอบสนองช้ามาก → พลาด regime change
- **ดี**: window ≈ 3–5 เท่าของ half-life ของ spread — เรียนวิธีหา half-life ในบทที่ 14

ตัวอย่าง: สร้าง Z-score จาก OLS Residuals

```
def build_spread(price_a, price_b, window=60):
    """
  สร้าง spread และ z-score จาก OLS rolling regression
  ทำ 3 ขั้น: (1) หา beta → (2) สร้าง spread → (3) คำนวณ z-score
    Returns: DataFrame ของ spread, beta, zscore
    """
    log_a = np.log(price_a)            # แปลงเป็น log price ทั้งคู่
    log_b = np.log(price_b)

    # ── ขั้น 1: หา hedge ratio (beta) ด้วย rolling OLS ──
    X = sm.add_constant(log_b)         # เติมคอลัมน์ค่าคงที่ (intercept) ให้
regression
    roll = RollingOLS(log_a, X, window=window).fit()
    beta  = roll.params.iloc[:, 1]     # สัมประสิทธิ์ของ log_b = hedge ratio
    alpha = roll.params['const']       # intercept

    # ── ขั้น 2: spread = ส่วนที่ "เหลือ" หลังหักความสัมพันธ์ออก ──
    predicted = alpha + beta * log_b   # ราคา A ที่ "ควรจะเป็น" ตามราคา B
    spread = log_a - predicted         # ของจริง - ที่ควรเป็น = ความเพี้ยน

    # ── ขั้น 3: z-score ของ spread (สูตรเดิมจาก Part I) ──
    roll_mean = spread.rolling(window).mean()
    roll_std  = spread.rolling(window).std()
    zscore    = (spread - roll_mean) / roll_std

    return pd.DataFrame({
        "spread": spread,
        "beta":   beta,
        "zscore": zscore
    })

df_pair = build_spread(
    pd.Series(btc_bybit),
    pd.Series(btc_binance),
    window=60
)
print(df_pair.tail())
print(f"\nCurrent Z-score: {df_pair['zscore'].iloc[-1]:.2f}")
```

▶ แบบฝึกหัด: interpret OLS output

สมมติ OLS ของ AAPL บน MSFT ได้: $\alpha=0.5$, $\beta=0.72$, $R^2=0.81$, $p(\beta)=0.000$

แปลว่า:

- $\beta=0.72$: MSFT ขึ้น 1% → AAPL มักขึ้น 0.72% (AAPL volatile น้อยกว่า)
- $R^2=0.81$: 81% ของความผันผวน AAPL อธิบายได้ด้วย MSFT — 19% ที่เหลือคือ spread
- Hedge: Long AAPL $100K → Short MSFT $72K
- $p=0.000$: $\beta \neq 0$ แน่นอน (มีนัยสำคัญสูงมาก)
- Residual mean ≈ 0 → spread ไม่ drift → น่าจะ stationary (ต้องยืนยันด้วย ADF)

บทที่ 14: Cointegration & Z-score Signal

แก่นของบทนี้

Correlation บอกว่าสองสิ่งเดินไปทิศเดียวกัน — Cointegration บอกว่าสองสิ่ง ผูกกันระยะยาว แม้ระยะสั้นจะแยกออกจากกันได้ Cointegration = ข้อพิสูจน์ว่า spread มีแรงดึงกลับจริง = basis ของ Statistical Arbitrage ทั้งหมด

14.1 Correlation vs Cointegration

	Correlation สูง	Cointegrated
หมายถึง	เดินทิศเดียวกันในระยะสั้น	ผูกกันในระยะยาว มี "แรงดึง"
Spread	อาจ drift ออกไปเรื่อยๆ	กลับหาค่ากลางเสมอ
ตัวอย่าง	หุ้นสอง sector เดียวกัน	BTC สองตลาด, ADR vs local
Trade ได้ไหม?	เสี่ยง — spread อาจไม่กลับ	ใช่ — spread ต้องกลับ (ในทฤษฎี)

14.2 Engle-Granger Test — 2 ขั้นตอน

วิธี Engle-Granger: (1) OLS หา residuals → (2) ADF test บน residuals

```
from statsmodels.tsa.stattools import adfuller, coint

# วิธีที่ 1: Engle-Granger ด้วย coint() โดยตรง
score, pvalue, crit_vals = coint(np.log(btc_bybit), np.log(btc_binance))

print(f"Cointegration test:")
print(f"  Test statistic: {score:.4f}")
print(f"  p-value:        {pvalue:.4f}")
print(f"  Critical (5%):  {crit_vals[1]:.4f}")

if pvalue < 0.05:
    print(f"✓ COINTEGRATED — spread มีแรงดึงกลับจริง")
else:
    print(f"✗ NOT COINTEGRATED — อย่า trade pair นี้!")

# วิธีที่ 2: ADF บน residuals (เข้าใจกระบวนการมากขึ้น)
log_a = np.log(btc_bybit)
log_b = np.log(btc_binance)
X = sm.add_constant(log_b)
residuals = sm.OLS(log_a, X).fit().resid
adf_resid = adfuller(residuals)
print(f"\nADF on residuals p-value: {adf_resid[1]:.4f}")
```

14.3 Johansen Test — สำหรับ 3+ Pairs

```
from statsmodels.tsa.vector_ar.vecm import coint_johansen

# Johansen ทดสอบ cointegration ระหว่าง 3 series พร้อมกัน
btc_okx = 65000 + np.cumsum(np.random.normal(0, 100, n))  # ตลาดที่ 3

data = np.column_stack([np.log(btc_bybit), np.log(btc_binance), np.log(btc_okx)])
result = coint_johansen(data, det_order=0, k_ar_diff=1)

print("Johansen Trace Statistics:")
for i in range(3):
    trace_stat = result.lr1[i]
    crit_5pct  = result.cvt[i, 1]
    print(f"  r≤{i}: trace={trace_stat:.2f}, crit(5%)={crit_5pct:.2f} "
          f"{'✓' if trace_stat > crit_5pct else '✗'}")

# r=0: ทดสอบว่ามีอย่างน้อย 1 cointegrating vector
# r≤1: ทดสอบว่ามีอย่างน้อย 2
```

14.4 Half-Life — Spread กลับเร็วแค่ไหน?

Half-life คือเวลาที่ spread ใช้กลับมาครึ่งทางจากค่าสุดขั้ว — ใช้เลือก window และ holding period

$$\text{Half-life} = \frac{-\ln(2)}{\ln(\hat{\phi})} \approx \frac{-\ln(2)}{\hat{\phi} - 1}$$

```python
def calc_halflife(spread):
    """คำนวณ half-life ของ mean-reverting spread"""
    spread_lag = pd.Series(spread).shift(1)
    delta      = pd.Series(spread) - spread_lag
    df_hl = pd.DataFrame({"delta": delta, "lag": spread_lag}).dropna()

    X = sm.add_constant(df_hl["lag"])
    phi = sm.OLS(df_hl["delta"], X).fit().params["lag"]

    if phi >= 0:
        return np.inf  # ไม่ mean-reverting (random walk หรือ explosive)
    if phi <= -1:
        return np.inf  # Oscillating — ไม่ใช่ mean-reverting จริง
    half_life = -np.log(2) / np.log(1 + phi)  # สูตรที่ถูกต้องสำหรับ AR(1)
    return half_life

hl = calc_halflife(resid)
print(f"Half-life: {hl:.1f} วัน")
print(f"แนะนำ: rolling window ≈ {hl*3:.0f} วัน, hold ไม่เกิน {hl*2:.0f} วัน")
```

ตัวอย่าง: Signal Generation Pipeline ครบวงจร

```
def pair_signal(price_a, price_b, window=60,
                entry_z=2.0, exit_z=0.5):
    """
  สร้าง trading signal สำหรับ pair trading
    Returns: DataFrame พร้อม signal column
    """
    df = build_spread(price_a, price_b, window)
    z  = df["zscore"]

    signals = pd.Series("HOLD", index=z.index)
    signals[z >  entry_z] = "SHORT_A_LONG_B"   # spread แพงเกิน
    signals[z < -entry_z] = "LONG_A_SHORT_B"   # spread ถูกเกิน
    signals[abs(z) < exit_z] = "EXIT"

    df["signal"] = signals
    # ⚠ สำคัญมาก: signal ที่ได้จาก df.index[t] ใช้ข้อมูลถึงวัน t
    # ต้อง .shift(1) ใน backtest ก่อน multiply ด้วย return เสมอ
    return df

result = pair_signal(
    pd.Series(btc_bybit),
    pd.Series(btc_binance),
    window=60, entry_z=2.0, exit_z=0.5
)
print(result[["spread","beta","zscore","signal"]].tail(10))
```

บทที่ 15: Optimization

แก่นของบทนี้

Optimization คือการหาค่าตัวแปร (เช่น portfolio weights, position size) ที่ทำให้ objective function (เช่น Sharpe ratio) ดีที่สุด ภายใต้ constraints (เช่น weights รวม = 1) Python ทำได้ด้วย `scipy.optimize.minimize`

$$\max_{\mathbf{w}} \text{Sharpe}(\mathbf{w}) = \frac{\mathbf{w}^\top \boldsymbol{\mu} - r_f}{\sqrt{\mathbf{w}^\top \boldsymbol{\Sigma} \mathbf{w}}} \quad \text{s.t.} \sum w_i = 1,\ w_i \geq 0$$

15.1 scipy.optimize.minimize — API

```
from scipy.optimize import minimize
import numpy as np

# สมมติว่าต้องการหา x ที่ทำให้ f(x) = (x-3)^2 น้อยที่สุด
def f(x):
    return (x[0] - 3)**2

result = minimize(f, x0=[0], method="SLSQP")
print(f"Optimal x: {result.x[0]:.4f}  (คาดหวัง: 3.0)")
print(f"Min f(x):  {result.fun:.6f}")
print(f"Success:   {result.success}")
```

15.2 Sharpe Ratio Maximization

```
def sharpe_maximize(returns_df, risk_free=0.0, periods=252):
    """
  หา portfolio weights ที่ maximize Sharpe ratio
    returns_df: DataFrame ของ daily returns (cols = assets)
    """
    n = len(returns_df.columns)
    mu    = returns_df.mean().values * periods        # annualized mean
    sigma = returns_df.cov().values * periods         # annualized cov

    def neg_sharpe(weights):
        """คืนค่า negative Sharpe (minimize = maximize Sharpe)"""
        port_ret = weights @ mu
        port_var = weights @ sigma @ weights
        return -(port_ret - risk_free) / np.sqrt(port_var)

    # Constraints
    constraints = [{"type": "eq", "fun": lambda w: np.sum(w) - 1}]  # sum = 1
    bounds      = [(0, 1)] * n                                       # no short
    x0          = np.ones(n) / n                                     # equally
weighted start

    result = minimize(neg_sharpe, x0,
                      method="SLSQP",
                      bounds=bounds,
                      constraints=constraints)

    if not result.success:
        print("Warning: optimizer did not converge")

    weights = result.x
    port_ret = weights @ mu
    port_vol = np.sqrt(weights @ sigma @ weights)

    print("=== Max Sharpe Portfolio ===")
    for name, w in zip(returns_df.columns, weights):
        print(f"  {name}: {w:.1%}")
    print(f"  Expected Return: {port_ret:.1%}/yr")
    print(f"  Volatility:      {port_vol:.1%}/yr")
    print(f"  Sharpe Ratio:    {(port_ret-risk_free)/port_vol:.2f}")
    return weights

# ทดสอบ
np.random.seed(0)
rets = pd.DataFrame(np.random.multivariate_normal(
    [0.001, 0.0007, 0.0012],
    [[0.0004, 0.0002, 0.0001],
     [0.0002, 0.0003, 0.0001],
     [0.0001, 0.0001, 0.0005]], 252),
    columns=["BTC","ETH","SOL"])

optimal_w = sharpe_maximize(rets)
```

ฝึกอ่าน 2 ท่อนที่ดูลึกลับที่สุดในโค้ดข้างบน

```
# ท่อนที่ 1
port_ret = weights @ mu
# ท่อนที่ 2
constraints = [{"type": "eq", "fun": lambda w: np.sum(w) - 1}]
```

ท่อนที่ 1: เครื่องหมาย `@` คือ "คูณแบบเวกเตอร์/เมทริกซ์" — `weights @ mu` อ่านว่า "เอาน้ำหนักแต่ละตัว คูณ return คาดหวังของ asset นั้น แล้วบวกรวมกัน" เท่ากับเขียน loop เองว่า:

```
port_ret = 0
for w, m in zip(weights, mu):
    port_ret = port_ret + w * m
```

ท่อนที่ 2: อ่านจากในออกนอก:

1. `lambda w: np.sum(w) - 1` — ฟังก์ชันจิ๋ว: "รับ w มา แล้วคืนค่า ผลรวมของ w ลบ 1"
2. `"type": "eq"` — บอก optimizer ว่าค่าที่ฟังก์ชันนี้คืน **ต้องเท่ากับ 0** → แปลว่า sum(w) - 1 = 0 → sum(w) = 1 = "ลงทุนเต็มพอดี ไม่ขาดไม่เกิน"
3. `[ { ... } ]` — เป็น list ของ dict เพราะใส่ได้หลายเงื่อนไข (scipy กำหนดรูปแบบนี้)

นี่คือ "ภาษาเฉพาะ" ของ scipy — เจอครั้งแรกทุกคนงง พออ่านออกแล้วจะเห็นว่ามันแค่บอกว่า "เงื่อนไข: น้ำหนักรวมกันต้องเป็น 1"

15.3 Kelly Criterion — Optimal Position Size

$$f^* = \frac{\mu - r_f}{\sigma^2} \qquad \text{(continuous Kelly)}$$

```
def kelly_fraction(mean_ret, std_ret, risk_free=0.0, fraction=0.25):
    """
  คำนวณ Kelly fraction สำหรับ continuous distribution
    fraction: ใช้ 0.25 (Quarter-Kelly) เพื่อความปลอดภัย
    """
    kelly_full = (mean_ret - risk_free) / (std_ret ** 2)
    kelly_frac = kelly_full * fraction

    print(f"Full Kelly:    {kelly_full:.2f}x")
    print(f"Quarter Kelly: {kelly_frac:.2f}x  ← แนะนำ")
    return kelly_frac

# ตัวอย่าง: กลยุทธ์ที่มี Sharpe = 1.5
daily_mean = 0.001      # 0.1% ต่อวัน
daily_std  = 0.015      # 1.5% ต่อวัน
f = kelly_fraction(daily_mean, daily_std)
```

Kelly ที่ใช้ที่นี่ = Continuous Kelly (สำหรับ return กระจาย Normal)

สูตร $f^* = \frac{\mu - r_f}{\sigma^2}$ เหมาะกับ asset ที่ return กระจายแบบ Normal distribution ต่อเนื่อง
สำหรับ **discrete bet** เช่น Binary outcome (win ได้ b เท่า, lose 1): $f^* = \frac{pb-(1-p)}{b}$

ใน Quant Trading ส่วนใหญ่ใช้ continuous Kelly หรือบ่อยกว่านั้นใช้ **Half Kelly (f*/2) หรือ Quarter Kelly (f*/4)** เพราะ:

- Return จริงไม่ Normal (fat tail)
- Estimate μ และ σ มี error เสมอ
- Full Kelly มี drawdown รุนแรงมากในช่วง streak ของ losing trades

อย่าใช้ Full Kelly ในทางปฏิบัติ

Full Kelly maximize geometric growth ในทฤษฎี แต่ในทางปฏิบัติ:

- ประมาณ parameters จากข้อมูลมีความผิดพลาด → Full Kelly อาจ oversize มาก
- Drawdown จาก Full Kelly ใหญ่มาก (50%+ เป็นปกติ) → ทนไม่ไหว

ใช้ Quarter-Kelly (f*/4) เป็นจุดเริ่มต้น — เรียนเพิ่มใน Part IV

ตัวอย่าง: Mean-Variance Optimization (Efficient Frontier)

```python
def efficient_frontier(returns_df, n_points=100):
    """
  สร้าง efficient frontier:
  สำหรับ return เป้าหมายแต่ละระดับ → หาพอร์ตที่ vol ต่ำสุด
    """
    n = len(returns_df.columns)
    mu    = returns_df.mean().values * 252   # expected return รายปี
    sigma = returns_df.cov().values * 252    # covariance รายปี

    # ไล่ return เป้าหมาย จากต่ำสุดถึงสูงสุด เป็น n_points จุด
    target_returns = np.linspace(mu.min(), mu.max(), n_points)

    def portfolio_variance(w):
        """สิ่งที่ต้องการ minimize: ความแปรปรวนของพอร์ต"""
        return w @ sigma @ w

    frontier_vols = []                        # เก็บ vol ต่ำสุดของแต่ละเป้า
    for target in target_returns:
        # เงื่อนไข 1: น้ำหนักรวม = 1 (ลงทุนเต็มพอดี)
        sum_to_one = {"type": "eq", "fun": lambda w: np.sum(w) - 1}

        # เงื่อนไข 2: return ของพอร์ต = target ของรอบนี้
        # (t=target จำเป็น! ดูกล่องกับดักด้านล่าง)
        hit_target = {"type": "eq", "fun": lambda w, t=target: w @ mu - t}

        result = minimize(portfolio_variance,
                          x0=np.ones(n) / n,         # เริ่มจากน้ำหนักเท่ากันทุกตัว
                          method="SLSQP",
                          bounds=[(0, 1)] * n,       # ห้าม short (0 ≤ w ≤ 1)
                          constraints=[sum_to_one, hit_target])

        if result.success:
            min_vol = np.sqrt(result.fun)     # ได้ variance ต่ำสุด → ถอดรากเป็น vol
        else:
            min_vol = np.nan                  # แก้ไม่ได้ (เป้าสูง/ต่ำเกิน) → เว้นจุดนี้
        frontier_vols.append(min_vol)

    return target_returns, np.array(frontier_vols)

rets_arr, vols_arr = efficient_frontier(rets)

import matplotlib.pyplot as plt
fig, ax = plt.subplots(figsize=(8, 5))
ax.plot(vols_arr, rets_arr, color="#0d9488", linewidth=2, label="Efficient
Frontier")
ax.set_xlabel("Annual Volatility")
ax.set_ylabel("Annual Return")
ax.set_title("Efficient Frontier (BTC, ETH, SOL)")
ax.legend()
plt.tight_layout()
plt.show()
```

⚠ กับดัก lambda ใน loop — ทำไมต้องเขียน `lambda w, t=target`

ถ้าเขียน `lambda w: w @ mu - target` เฉย ๆ — lambda จะ "จำชื่อตัวแปร" ไม่ใช่ "จำค่า" พอ loop จบ `target` ชี้ไปที่ค่าสุดท้ายแล้ว ทุก lambda ที่สร้างไว้จะใช้ค่าเดียวกันหมด → frontier ผิดทั้งเส้นแบบเงียบ ๆ ไม่มี error

ท่า `t=target` คือการ "ถ่ายสำเนาค่า ณ รอบนั้น" เก็บไว้ใน t ของ lambda ตัวนั้นเอง — นี่เป็นกับดัก Python คลาสสิกที่แม้แต่ AI ยังเขียนพลาดเป็นครั้งคราว **เจอ lambda ใน loop เมื่อไหร่ ให้มองหา `=` แบบนี้เสมอ**

▶ แบบฝึกหัด: Minimize Portfolio Variance (Min-Vol Portfolio)

```
def min_variance_portfolio(returns_df):
    """หา portfolio ที่มี variance ต่ำที่สุด"""
    n = len(returns_df.columns)
    sigma = returns_df.cov().values * 252

    result = minimize(
        lambda w: w @ sigma @ w,
        x0=np.ones(n)/n,
        method="SLSQP",
        bounds=[(0, 1)]*n,
        constraints=[{"type": "eq", "fun": lambda w: np.sum(w) - 1}]
    )

    w = result.x
    vol = np.sqrt(w @ sigma @ w)
    print("Min Variance Portfolio:")
    for name, wi in zip(returns_df.columns, w):
        print(f"  {name}: {wi:.1%}")
    print(f"  Annual Vol: {vol:.1%}")
    return w

min_variance_portfolio(rets)
```

บทที่ 16: Linear Programming

แก่นของบทนี้

Linear Programming (LP) คือ optimization ที่ทั้ง objective function และ constraints เป็น linear — เร็วกว่า nonlinear optimization มาก ใช้ใน Quant สำหรับ: กำหนด portfolio weight ภายใต้หลาย constraint พร้อมกัน, minimize transaction cost, sizing หลาย position พร้อมกัน

$$\min_{\mathbf{w}} \ \mathbf{c}^\top \mathbf{w} \quad \text{s.t. } \mathbf{A}\mathbf{w} \leq \mathbf{b},\ \mathbf{A}_{eq}\mathbf{w} = \mathbf{b}_{eq},\ \mathbf{lb} \leq \mathbf{w} \leq \mathbf{ub}$$

16.1 scipy.linprog — API

```
from scipy.optimize import linprog

# ตัวอย่างง่าย: minimize cost ของการซื้อ 3 assets
# minimize: 2*w1 + 3*w2 + 1.5*w3
# subject to: w1+w2+w3 = 1 (fully invested)
#             w1 >= 0.1 (min 10% BTC)
#             w2 >= 0.1 (min 10% ETH)

c   = [2.0, 3.0, 1.5]                   # objective: cost coefficients
A_eq = [[1, 1, 1]]                       # equality: sum = 1
b_eq = [1]
bounds = [(0.1, 0.6), (0.1, 0.5), (0, 0.8)]  # min/max ของแต่ละ w

result = linprog(c, A_eq=A_eq, b_eq=b_eq,
                 bounds=bounds, method="highs")

print(f"Optimal weights: {result.x.round(3)}")
print(f"Minimum cost:    {result.fun:.4f}")
print(f"Success: {result.success}")
```

16.2 Portfolio LP — ใส่ Constraints หลายอย่างพร้อมกัน

```python
from scipy.optimize import linprog
import numpy as np

def lp_portfolio(expected_returns, min_return=0.10,
                 max_weights=None, min_weights=None,
                 max_sector=None, sectors=None):
    """
    Minimize portfolio variance (approximated as linear)
  ด้วย LP — เหมาะสำหรับ constraint-heavy problem

    expected_returns: array ของ expected annual return แต่ละ asset
    min_return: minimum portfolio return ที่ต้องการ
    """
    n = len(expected_returns)

    # Objective: minimize negative return (= maximize return)
    # Linear approximation — ใช้สำหรับ constraint-heavy problems
    c = -np.array(expected_returns)

    # Equality: sum(w) = 1
    A_eq = [np.ones(n)]
    b_eq = [1.0]

    # Inequality: min return constraint
    # w @ returns >= min_return  →  -w @ returns <= -min_return
    A_ub = [-np.array(expected_returns)]
    b_ub = [-min_return]

    # Bounds: default 0–1 (long only)
    if min_weights is None: min_weights = [0.0] * n
    if max_weights is None: max_weights = [1.0] * n
    bounds = list(zip(min_weights, max_weights))

    result = linprog(c, A_ub=A_ub, b_ub=b_ub,
                     A_eq=A_eq, b_eq=b_eq,
                     bounds=bounds, method="highs")

    return result

# ตัวอย่าง: 5 assets
assets  = ["BTC", "ETH", "SOL", "AAPL", "GOLD"]
returns = [0.35,  0.25,  0.40,  0.12,   0.08]

result = lp_portfolio(
    returns,
    min_return  = 0.20,
    min_weights = [0.05]*5,      # min 5% ทุก asset
    max_weights = [0.40]*5,      # max 40% ทุก asset
)

print("LP Portfolio Allocation:")
for asset, w in zip(assets, result.x):
    print(f"  {asset:5s}: {w:.1%}")
print(f"Expected Return: {np.dot(result.x, returns):.1%}")
```

16.3 PuLP — สำหรับ Integer Programming

```
# pip install pulp
from pulp import *

# Integer LP: เลือก k หุ้นดีที่สุดจาก n ตัว
scores = {"BTC": 0.85, "ETH": 0.70, "SOL": 0.60,
          "AAPL": 0.55, "GOLD": 0.40}
costs  = {"BTC": 5000, "ETH": 2000, "SOL": 500,
          "AAPL": 1000, "GOLD": 800}
budget = 8000   # งบ 8000 บาท

prob = LpProblem("SelectAssets", LpMaximize)

# ตัวแปร binary: 1=เลือก, 0=ไม่เลือก
x = {a: LpVariable(f"x_{a}", cat="Binary") for a in scores}

# Objective: maximize total score
prob += lpSum(scores[a] * x[a] for a in scores)

# Constraints
prob += lpSum(costs[a] * x[a] for a in scores) <= budget  # budget
prob += lpSum(x[a] for a in scores) <= 3                   # เลือกได้ max 3 ตัว

prob.solve(PULP_CBC_CMD(msg=0))

print("Selected assets:")
for a in scores:
    if x[a].value() == 1:
        print(f"  {a}: score={scores[a]}, cost={costs[a]}")
```

ตัวอย่าง: Position Sizing LP สำหรับ Pair Trading

```
# กำหนด: มี 3 pairs พร้อม trade, capital จำกัด
# objective: maximize expected PnL
# constraints: total capital, max drawdown per pair, correlation limit

pairs       = ["BTC-Bybit/Binance", "ETH-OKX/Bybit", "SOL-Binance/OKX"]
exp_pnl     = [0.15, 0.12, 0.20]   # expected annual return
max_dd      = [0.05, 0.08, 0.10]   # max expected drawdown
capital     = 1_000_000             # total capital บาท

# Minimize negative PnL = maximize PnL
c   = [-e for e in exp_pnl]         # negate for minimization

# sum(allocation) = 1.0 (fully allocated)
A_eq = [[1, 1, 1]]
b_eq = [1.0]

# เงื่อนไข: drawdown ที่คาดของแต่ละ pair ≤ 5% ของ capital
# allocation_i × max_dd_i ≤ 0.05 — แยกกันทีละ pair = matrix แนวทแยง
A_ub = np.diag(max_dd)      # [[0.05, 0,    0   ],
                            #  [0,    0.08, 0   ],
                            #  [0,    0,    0.10]]
b_ub = [0.05] * 3

bounds = [(0.0, 0.5)] * 3   # max 50% ต่อ pair

result = linprog(c, A_ub=A_ub, b_ub=b_ub,
                 A_eq=A_eq, b_eq=b_eq,
                 bounds=bounds, method="highs")

print("Optimal Position Sizing:")
for pair, alloc in zip(pairs, result.x):
    print(f"  {pair}: {alloc:.1%} = ฿{alloc*capital:,.0f}")
print(f"Expected Portfolio Return: {-result.fun:.1%}")
```

AI มักสร้าง matrix แนวทแยงด้วย nested comprehension — ฝึกอ่าน

```
A_ub = [[max_dd[i] if j==i else 0 for j in range(3)] for i in range(3)]
```

comprehension ซ้อนสองชั้น — อ่าน**จากขวา (ชั้นนอก) เข้าซ้าย (ชั้นใน)**:

1. `for i in range(3)` (ชั้นนอก, ขวาสุด) — สร้าง 3 แถว แถวละหนึ่ง i
2. `for j in range(3)` (ชั้นใน) — แต่ละแถวมี 3 ช่อง ช่องละหนึ่ง j
3. `max_dd[i] if j==i else 0` — แต่ละช่อง: "เอาค่า max_dd **ถ้า**อยู่บนเส้นทแยง (แถว=คอลัมน์) **ไม่งั้น**ใส่ 0"

ผลคือ matrix เดียวกับ `np.diag(max_dd)` เป๊ะ — เวอร์ชัน np.diag สั้นกว่า อ่านง่ายกว่า และผิดยากกว่า ถ้า AI เขียนแบบ nested มา ขอให้มันแก้เป็น np.diag ได้เลย

▶ แบบฝึกหัด: LP Constraint สำหรับ Sector Limits

```
# 6 assets ใน 2 sectors: Crypto (BTC,ETH,SOL) และ Tech (AAPL,MSFT,NVDA)
assets  = ["BTC","ETH","SOL","AAPL","MSFT","NVDA"]
returns = [0.40, 0.30, 0.50, 0.15, 0.18, 0.25]

# Crypto = assets 0,1,2 / Tech = assets 3,4,5
crypto = [1,1,1,0,0,0]
tech   = [0,0,0,1,1,1]

c    = [-r for r in returns]
A_eq = [[1]*6]
b_eq = [1.0]

# Sector limits: Crypto ≤ 60%, Tech ≤ 60%
A_ub = [crypto, tech]
b_ub = [0.60, 0.60]
bounds = [(0.0, 0.4)] * 6   # max 40% per asset

res = linprog(c, A_ub=A_ub, b_ub=b_ub,
              A_eq=A_eq, b_eq=b_eq,
              bounds=bounds, method="highs")

for a, w in zip(assets, res.x):
    print(f"  {a}: {w:.1%}")
print(f"Crypto total: {sum(res.x[:3]):.1%}  Tech total: {sum(res.x[3:]):.1%}")
```

บทที่ 17: Signals & Filters

แก่นของบทนี้

Signal คือ "เมื่อไหร่ควร trade" — บทนี้รวม 3 เครื่องมือ: **Bollinger Bands** (signal จาก volatility), **Kalman Filter** (track β แบบ adaptive), และ **Signal Pipeline** (รวมหลาย signal เข้าด้วยกัน) ซึ่งเป็น boilerplate ที่ใช้ได้กับทุก strategy

17.1 Bollinger Bands — Signal จาก Volatility

```
def bollinger_bands(series, window=20, n_std=2):
    """คำนวณ Bollinger Bands"""
    s    = pd.Series(series)
    mean = s.rolling(window).mean()
    std  = s.rolling(window).std()
    upper = mean + n_std * std
    lower = mean - n_std * std
    return pd.DataFrame({"mid": mean, "upper": upper, "lower": lower,
                         "pct_b": (s - lower) / (upper - lower)})

# pct_b: 0 = ที่ lower band, 1 = ที่ upper band
# signal: SHORT เมื่อ pct_b > 1, LONG เมื่อ pct_b < 0
bb = bollinger_bands(spread, window=20, n_std=2)
signal = pd.Series("HOLD", index=bb.index)
signal[bb["pct_b"] > 1.0] = "SHORT"
signal[bb["pct_b"] < 0.0] = "LONG"
```

17.2 Kalman Filter — Track β แบบ Adaptive

Kalman Filter ประมาณ β แบบ real-time โดยอัปเดตทุกครั้งที่มีข้อมูลใหม่ — ดีกว่า rolling OLS ตรงที่ไม่มี window cutoff และ smooth กว่า

> **Kalman Filter ทำงานสองจังหวะ — "เดา แล้วแก้เดา" ซ้ำไปเรื่อย ๆ**
>
> ลองนึกถึงการเดาน้ำหนักเพื่อนทุกครั้งที่เจอกัน:
> **จังหวะ 1 — เดา (Predict):** "วันนี้น่าจะหนักเท่าเดิมจากครั้งก่อน แต่เวลาผ่านไปแล้ว ความมั่นใจของเราลดลงนิดหน่อย"
> **จังหวะ 2 — แก้เดา (Update):** เห็นข้อมูลใหม่ (เพื่อนตัวจริง) → เทียบกับที่เดา → ถ้าต่างกัน ขยับคำเดาเข้าหาความจริง ตามสัดส่วนความมั่นใจ — มั่นใจคำเดาเดิมมากก็ขยับนิดเดียว ไม่มั่นใจก็ขยับเยอะ
>
> ตัวเลข "สัดส่วนการขยับ" นี้คือ **Kalman Gain (K)** — หัวใจของทั้ง algorithm มีแค่นี้

```python
def kalman_hedge_ratio(price_a, price_b,
                       delta=1e-4, vt=1e-3):
    """
    Track hedge ratio beta ด้วย Kalman Filter
    delta: process noise (ใหญ่ = ยอมให้ beta เปลี่ยนเร็ว)
    vt:    observation noise (ใหญ่ = ราคามี noise มาก เชื่อข้อมูลใหม่น้อยลง)
    Returns: array ของ beta ณ แต่ละเวลา
    """
    n       = len(price_a)
    beta    = np.zeros(n)     # เตรียมที่เก็บ beta ของทุกวัน
    P       = np.ones(n)      # P = "ความไม่มั่นใจ" ในค่า beta (variance ของคำเดา)
    beta[0] = 1.0             # วันแรก: เดาไว้ก่อนว่า beta = 1

    for t in range(1, n):
        # ── จังหวะ 1: เดา (Predict) ──
        # beta วันนี้น่าจะเท่าเมื่อวาน แต่ความไม่มั่นใจเพิ่มขึ้นตาม delta
        P_pred = P[t-1] + delta

        # ── จังหวะ 2: แก้เดา (Update) ด้วยข้อมูลใหม่ ──
        x = price_b[t]               # ราคาตลาด B วันนี้ (ตัวพยากรณ์)

        y_pred = beta[t-1] * x       # ราคา A "ที่ควรเป็น" ตามคำเดาเดิม
        innov  = price_a[t] - y_pred  # ของจริง - ที่เดา = "เดาพลาดเท่าไหร่"

        S = x**2 * P_pred + vt       # ความแปรปรวนรวมของความพลาด
                                     # (จากความไม่มั่นใจใน beta + noise ของราคา)
        K = P_pred * x / S           # Kalman Gain: "ควรขยับคำเดากี่ %"
                                     # ไม่มั่นใจมาก (P สูง) → K ใหญ่ → ขยับเยอะ
                                     # ราคา noise มาก (vt สูง) → K เล็ก → ขยับนิดเดียว

        beta[t] = beta[t-1] + K * innov   # ขยับ beta เข้าหาความจริง ตามสัดส่วน K
        P[t]    = (1 - K * x) * P_pred    # แก้เดาแล้ว → ความมั่นใจเพิ่มขึ้น (P ลดลง)

    return beta

# ทดสอบ
log_a = np.log(btc_bybit)
log_b = np.log(btc_binance)
kf_beta = kalman_hedge_ratio(log_a, log_b)

print(f"Kalman Beta (latest 5): {kf_beta[-5:].round(6)}")
```

17.3 Signal Pipeline — รวมทุกอย่างเข้าด้วยกัน

```python
class PairTradingSignal:
    """
    Signal pipeline สำหรับ Pair Trading
  รวม: OLS hedge ratio, z-score, Bollinger filter
    """
    def __init__(self, window=60, entry_z=2.0, exit_z=0.5):
        self.window   = window
        self.entry_z  = entry_z
        self.exit_z   = exit_z

    def fit(self, price_a: pd.Series, price_b: pd.Series):
        """ประมาณ parameters จากข้อมูลประวัติ"""
        self.price_a = price_a
        self.price_b = price_b

        # OLS spread
        self.df = build_spread(price_a, price_b, self.window)

        # Half-life
        self.halflife = calc_halflife(self.df["spread"].dropna())
        print(f"Half-life: {self.halflife:.1f} วัน")

        # ADF test
        adf_p = adfuller(self.df["spread"].dropna())[1]
        self.is_stationary = adf_p < 0.05
        print(f"Stationary: {self.is_stationary} (p={adf_p:.3f})")
        return self

    def signal(self) -> pd.Series:
        """สร้าง trading signal"""
        z = self.df["zscore"]
        sig = pd.Series("HOLD", index=z.index)
        sig[z >  self.entry_z] = "SHORT_A"
        sig[z < -self.entry_z] = "LONG_A"
        sig[abs(z) < self.exit_z] = "EXIT"
        return sig

    def summary(self):
        """แสดง summary ของ current state"""
        current_z = self.df["zscore"].iloc[-1]
        current_b = self.df["beta"].iloc[-1]
        sig = self.signal().iloc[-1]
        print(f"=== Pair Signal Summary ===")
        print(f"Current Z-score: {current_z:.3f}")
        print(f"Current Beta:    {current_b:.6f}")
        print(f"Signal:          {sig}")
        print(f"Is Stationary:   {self.is_stationary}")

# ใช้งาน
pipeline = PairTradingSignal(window=60, entry_z=2.0, exit_z=0.5)
pipeline.fit(pd.Series(btc_bybit), pd.Series(btc_binance))
pipeline.summary()
```

จบ Part II — สิ่งที่ทำได้แล้ว

- ทดสอบ stationarity ด้วย ADF test ✓
- ยืนยัน cointegration ด้วย Engle-Granger และ Johansen ✓
- ประมาณ β ด้วย OLS และ Kalman Filter ✓
- สร้าง z-score signal สำหรับ entry/exit ✓
- Maximize Sharpe ratio ด้วย `scipy.optimize` ✓
- กำหนด portfolio constraint ด้วย Linear Programming ✓
- รวมทุกอย่างเป็น `PairTradingSignal` class ✓

ใน **Part III** เราจะ refactor code นี้ให้เป็น OOP ที่ maintainable และ extensible

⚠ คำเตือน: ตัวเลขใน Part II คือ In-Sample — อย่าเชื่อ 100%

Sharpe 2.0+ และ Return 15%+ ที่เห็นใน Part II ล้วน **คำนวณจากข้อมูลชุดเดียวกับที่ optimize parameter** (in-sample)

ในชีวิตจริง (out-of-sample) ผลลัพธ์มักต่ำกว่า **50–80%** เพราะ:

- ตลาดเปลี่ยน — pattern ในอดีตไม่ซ้ำแบบในอนาคต
- Parameter ที่ดีในอดีต = overfit กับ noise ในอดีต
- Transaction cost ที่ลืมใส่ กินไป 20–50% ของ gross PnL

Part IV จะสอน Walk-Forward Validation — วิธีเดียวที่ยืนยันได้ว่า strategy มี edge จริง ไม่ใช่แค่ overfit
อย่าใช้เงินจริงจนกว่าจะผ่านการทดสอบใน Part IV