Member-only story

Trading with the Fourier Transform Strategy: A Rolling Backtest

PyQuantLab Following 7 min read · Jun 20, 2025

Financial markets are often perceived as random or chaotic, yet underlying cyclic patterns and trends can sometimes be discerned from the noise. The Fourier Transform, a powerful mathematical tool from signal processing, offers a unique way to decompose a price series into its constituent frequencies, potentially revealing these hidden cycles. This article introduces a trading strategy that leverages the Fast Fourier Transform (FFT) to identify dominant market cycles and uses them for signal generation, evaluated through a robust rolling backtest.

🔥 Unlock the Power of Profitable Trading with Code

🚀 The Ultimate Algorithmic Trading Bundle

> ✅ Over 80 ready-to-run strategies
> ✅ Across 5 powerful categories: momentum, trend-following, mean reversion, ML-based, and volatility
> ✅ Includes step-by-step PDF guides with code explanations, visuals & real-world insights
> ✅ Perfect for traders, quant enthusiasts, and developers

📦 Everything you need to go from zero to confident algo trader — in one complete package.

👉 Start building & backtesting proven strategies today:

🛒 Get the Ultimate Bundle

💡 Just Starting Out? Grab the Algo Trading Value Pack

> ⚡ 3 concise eBooks
> 📘 30+ Python-coded strategies
> 🧠 Simple, practical, and actionable — ready to use in Backtrader and real trading platforms

🎯 Ideal if you're looking to learn fast and see results in just a few days.

🛒 Start with the Value Pack

The Method & Theory: Signal Processing in Financial Markets

At its core, the Fourier Transform converts a time-domain signal (like a price series) into its frequency-domain representation. This means it breaks down a complex waveform into a sum of simple sine and cosine waves, each with a specific frequency, amplitude, and phase.

The underlying theory applied here is that by identifying and reconstructing the most dominant (highest amplitude) frequency components of a price series, one can filter out random noise and reveal the true underlying momentum or cyclic behavior.

The process within the strategy involves:

1. Detrending: A linear trend is first removed from the price data. This is crucial because FFT is designed for stationary signals, and removing the trend isolates the oscillatory components.

$$\text{Detrended}_t = \text{Price}_t - (\text{Slope} \times t + \text{Intercept})$$

2. Fast Fourier Transform (FFT): The detrended series is transformed into the frequency domain. This yields a set of complex numbers, where the magnitude of each number represents the amplitude of a particular frequency, and the argument (angle) represents its phase.

$$F(k) = \sum_{n=0}^{N-1} x_n \cdot e^{-2\pi i k n / N}$$

3. Dominant Frequency Selection: Only a specified number of frequencies with the largest amplitudes (`num_components`) are retained. These are assumed to represent the most significant and stable cycles in the price data.

4. Signal Reconstruction: An Inverse FFT (IFFT) is then applied to these selected dominant frequencies. This reconstructs a smoothed, synthetic price signal that highlights the most prominent cyclical patterns, stripped of high-frequency noise.

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} F(k) \cdot e^{2\pi i k n / N}$$

5. Signal Interpretation: The current value and the slope (change) of this reconstructed signal are used to infer directional bias. A rising reconstructed signal indicates an upward trend, and a falling signal indicates a downward trend.

The Strategy: `FourierStrategy`

The `FourierStrategy` in Backtrader implements this methodology. It maintains a rolling window of historical prices, performs the Fourier analysis on this window, and then uses the properties of the reconstructed signal to generate trading signals.

```
import backtrader as bt
import yfinance as yf
import numpy as np

class FourierStrategy(bt.Strategy):
    params = (
        ('lookback', 30),        # Window size for FFT analysis
        ('num_components', 3),   # Number of dominant frequencies to
        ('trend_period', 30),    # Period for an external SMA trend
        ('stop_loss_pct', 0.02),# Percentage for stop loss
    )

    def __init__(self):
        self.price_history = [] # Buffer to store prices for FFT
        self.trend_ma = bt.indicators.SMA(self.data.close, period=s
        self.fft_signal = 0     # Stores the last reconstructed sig
        self.fft_trend = 0      # Stores the trend/slope of the rec
        self.order = None
        self.stop_order = None

    def fourier_analysis(self, prices):
```

```
        """Performs FFT analysis and returns the last reconstructed
        if len(prices) < self.params.lookback: return 0, 0

        # Detrend the data using linear regression
        x = np.arange(len(prices))
        coeffs = np.polyfit(x, prices, 1)
        trend = np.polyval(coeffs, x)
        detrended = prices - trend

        # Apply FFT and select dominant components
        fft_values = np.fft.fft(detrended)
        magnitude = np.abs(fft_values)
        # Select indices of 'num_components' largest magnitudes (ex
        dominant_indices = np.argsort(magnitude)[-self.params.num_c

        # Reconstruct signal using only dominant frequencies
        reconstructed_fft = np.zeros_like(fft_values, dtype=complex
        reconstructed_fft[dominant_indices] = fft_values[dominant_i
        reconstructed = np.real(np.fft.ifft(reconstructed_fft)) # B

        current_signal = reconstructed[-1]
        signal_trend = reconstructed[-1] - reconstructed[-2] if len

        return current_signal, signal_trend

    def notify_order(self, order):
        # Manages order lifecycle, including placing and cancelling
        # This is a critical component for risk management.
        if order.status == order.Completed:
            if order.isbuy() and self.position.size > 0:
                stop_price = order.executed.price * (1 - self.param
                self.stop_order = self.sell(exectype=bt.Order.Stop,
            elif order.issell() and self.position.size < 0:
                stop_price = order.executed.price * (1 + self.param
                self.stop_order = self.buy(exectype=bt.Order.Stop,
        # ... (logic to reset self.order and self.stop_order after
        self.order = None # Important to free up the order slot

    def next(self):
        if self.order is not None: return # Avoid multiple pending

        self.price_history.append(self.data.close[0])
        if len(self.price_history) > self.params.lookback:
            self.price_history = self.price_history[-self.params.lo
```

```
        if len(self.price_history) < self.params.lookback: return #

        # Perform Fourier analysis
        signal_val, signal_trend = self.fourier_analysis(np.array(s

        prev_signal_val = self.fft_signal # Value from previous bar
        self.fft_signal = signal_val      # Update current signal
        self.fft_trend = signal_trend     # Update current trend of

        # Trading logic:
        # Long Entry: Reconstructed signal turns up AND price is ab
        if (signal_trend > 0 and prev_signal_val < 0 and self.data.
            if self.position.size < 0: # Close short first
                if self.stop_order is not None: self.cancel(self.st
                self.order = self.close()
            elif not self.position: # Then go long
                self.order = self.buy()

        # Short Entry: Reconstructed signal turns down AND price is
        elif (signal_trend < 0 and prev_signal_val > 0 and self.dat
            if self.position.size > 0: # Close long first
                if self.stop_order is not None: self.cancel(self.st
                self.order = self.close()
            elif not self.position: # Then go short
                self.order = self.sell()
```

The `next` method's logic is designed to capture turning points in the dominant cycle. It initiates a long position when the reconstructed signal shows an upward turn (`signal_trend > 0` and `prev_signal_val < 0`) and the actual price confirms an uptrend by being above its `trend_ma` . A similar logic applies for short entries.

Here's a sample backtest result for trading Bitcoin:

Rolling Backtest: Assessing Market Adaptability

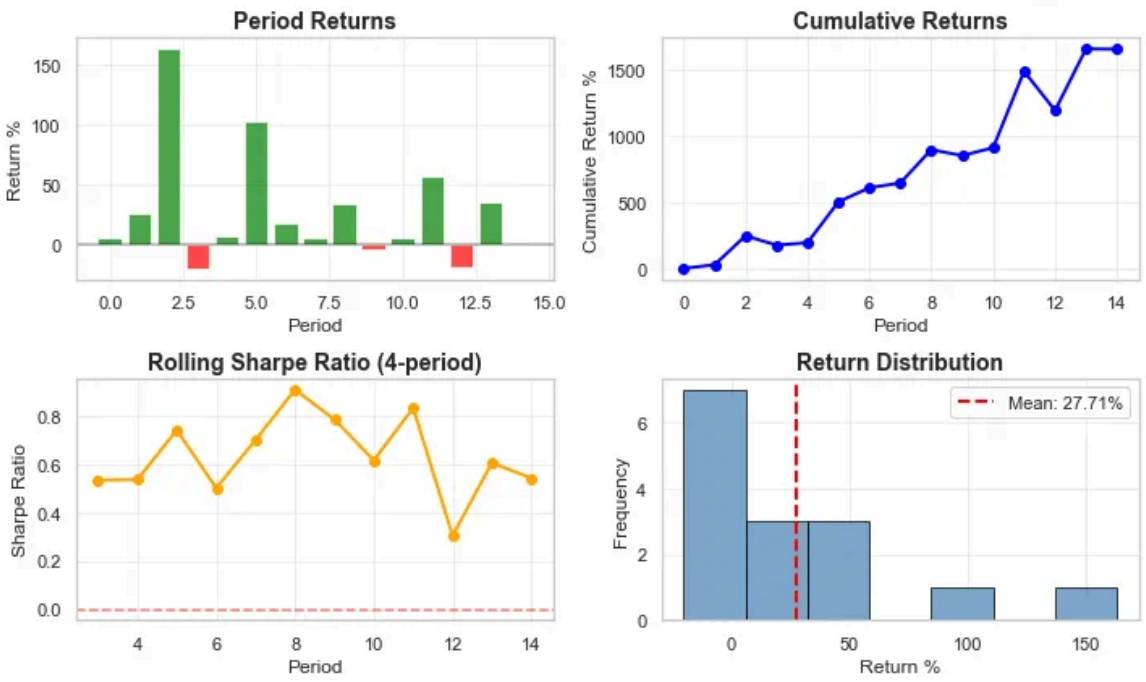
Given the Fourier Strategy's reliance on historical data patterns, a rolling backtest is essential to evaluate its performance consistency across different market periods. This method involves running the strategy repeatedly on successive, fixed-length historical windows (e.g., 6-month periods).

```
import pandas as pd
import dateutil.relativedelta as rd
import matplotlib.pyplot as plt
import seaborn as sns

def run_rolling_backtest(
    ticker="BTC-USD",
    start="2018-01-01",
    end="2025-12-31",
    window_months=6, # 6-month rolling windows
```

```
    strategy_params=None
):
    strategy_params = strategy_params or {}
    all_results = []
    start_dt = pd.to_datetime(start)
    end_dt = pd.to_datetime(end)
    current_start = start_dt

    while True:
        current_end = current_start + rd.relativedelta(months=windo
        if current_end > end_dt: break

        print(f"\nROLLING BACKTEST: {current_start.date()} to {curr
        # Data download respects the saved instruction: auto_adjust
        data = yf.download(ticker, start=current_start, end=current
        if data.empty or len(data) < 90: # Ensure sufficient data f
            print("Not enough data.")
            current_start += rd.relativedelta(months=window_months)
            continue

        cerebro = bt.Cerebro()
        cerebro.addstrategy(FourierStrategy, **strategy_params)
        cerebro.adddata(bt.feeds.PandasData(dataname=data))
        cerebro.broker.setcash(100000)
        cerebro.broker.setcommission(commission=0.001)
        cerebro.addsizer(bt.sizers.PercentSizer, percents=95)

        start_val = cerebro.broker.getvalue()
        cerebro.run()
        final_val = cerebro.broker.getvalue()
        ret = (final_val - start_val) / start_val * 100

        all_results.append({'start': current_start.date(), 'end': c
                            'return_pct': ret, 'final_value': final
        print(f"Return: {ret:.2f}% | Final Value: {final_val:.2f}")
        current_start += rd.relativedelta(months=window_months)

    return pd.DataFrame(all_results)

def report_stats(df):
    # Calculates and prints mean, median, std dev, min/max returns,
    # ... (function body)

def plot_four_charts(df, rolling_sharpe_window=4):
```

```
    # Visualizes rolling backtest results across four subplots.
    # ... (function body)

if __name__ == '__main__':
    # Run the rolling backtest with default parameters for BTC-USD
    df_results = run_rolling_backtest(window_months=6, ticker="BTC-
                                      strategy_params={'lookback':
                                                       'trend_perio

    print("\n=== ROLLING BACKTEST RESULTS ===")
    print(df_results)
    report_stats(df_results)
    plot_four_charts(df_results)
```

The `run_rolling_backtest` function iterates through historical data, conducting separate backtests on `window_months` periods. The `plot_four_charts` function then provides a comprehensive visual summary of these results, illustrating period-by-period returns, cumulative performance, rolling Sharpe ratios, and the distribution of returns.



Conclusion: A Fresh Perspective on Price Analysis

The Fourier Strategy offers a novel approach to technical analysis by applying signal processing techniques to financial data. By filtering out noise and focusing on dominant cyclical patterns, it aims to provide cleaner, more reliable trend and turning point signals. The rolling backtest is a crucial step in evaluating the practical viability of such a strategy, demonstrating its consistency (or lack thereof) across diverse market conditions. While mathematically elegant, the Fourier Transform's effectiveness in predictive trading heavily depends on the stationarity of market cycles and careful parameter tuning. Further research, including robust optimization and out-of-sample testing, would be necessary to establish its long-term profitability and adaptability to ever-evolving market dynamics.

Python | Trading | Backtesting | Algorithmic Trading | Bicoin

Written by PyQuantLab

655 followers · 6 following

Following

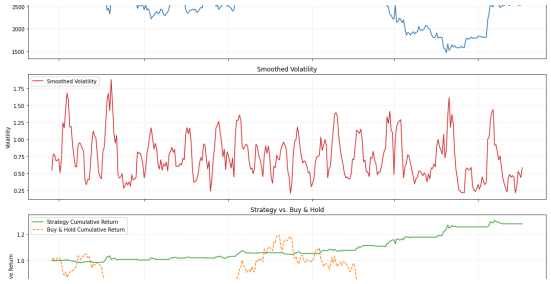
Your go-to place for Python-based quant tutorials, strategy deep-dives, and reproducible code. For more visit our website: www.pyquantlab.com

No responses yet

Steven Feng CAI

What are your thoughts?

More from PyQuantLab



PyQuantLab

Volatility Clustering Trading Strategy with Python

Ultimate Algorithmic Strategy Bundle has you covered with over 80 Python...

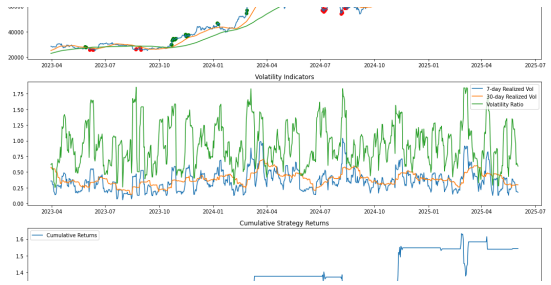
Jun 3 32 3

PyQuantLab

An Algorithmic Exploration of Volume Spread Analysis...

Note: You can read all articles for free on our website: pyquantlab.com

Jun 9 54 1

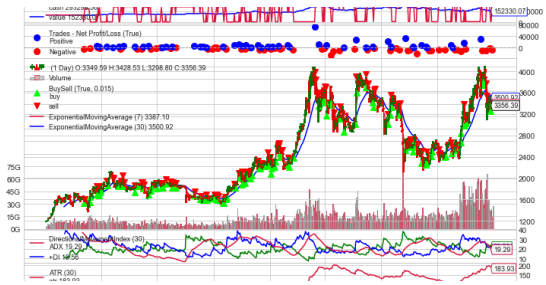


PyQuantLab

Trend-Volatility Confluence Trading Strategy

The Ultimate Algorithmic Strategy Bundle has you covered with over 80...

Jun 3 60



PyQuantLab

Building an Adaptive Trading Strategy with Backtrader: A...

Note: You can read all articles for free on our website: pyquantlab.com

Jun 4 64

See all from PyQuantLab

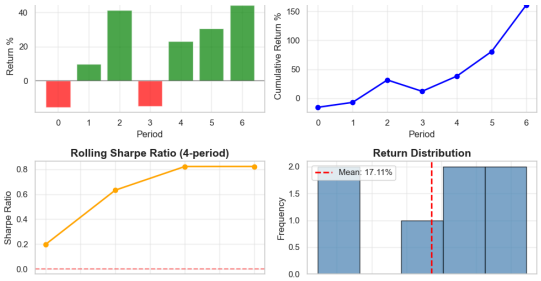
Recommended from Medium

Swapnilphutane

How I Built a Multi-Market Trading Strategy That Pass...

When I first got into trading, I had no plans of building a full-blown system....

6d ago 16 1

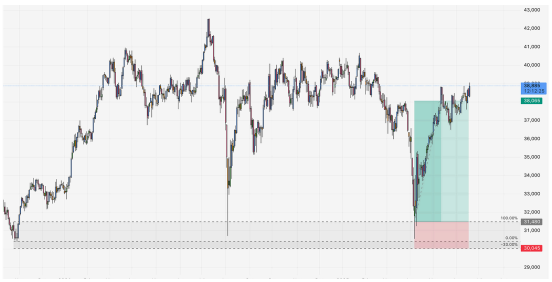


PyQuantLab

Precision Trading with Volume Profile: An Enhance...

In the intricate landscape of financial markets, understanding where...

Jun 20 11



Candence

Exposing Bernd Skorupinski Strategy: How I Profited ove...

I executed a single Nikkei Futures trade that banked $16,400 with a...

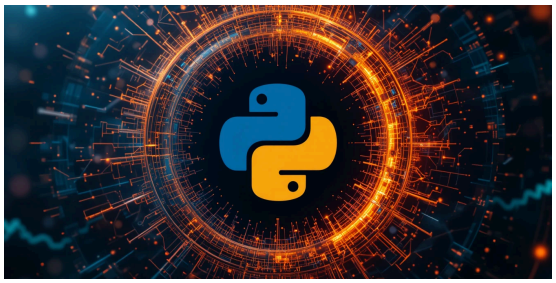
Jun 19 2

MarketMuse

“I Let an AI Bot Trade for Me for 7 Days—It Made $8,000...

Subtitle: While you’re analyzing candlestick patterns, AI bots are fron...

Jun 3 75 3



Unicorn Day

The Quest for the Perfect Trading Score: Turning...

Navigating the financial markets... it feels like being hit by a tsunami of da...

3d ago 43

Navnoor Bawa

QOADRS Mathematical Implementation: A Step-by-...

How we achieved 90.7% accuracy processing 9,462 real options...

Jun 13 2

See more recommendations