Member-only story

Butterworth Digital Filter for Trading? Backtesting with Python and Backtrader

PyQuantLab Following 5 min read · Jun 17, 2025

In financial markets, price data is inherently noisy, often obscuring true trends and making accurate signal generation challenging. Traditional moving averages can smooth this noise, but they introduce lag. Digital filters, particularly the Butterworth filter, offer a sophisticated alternative, allowing for precise control over the trade-off between smoothing and lag. This article explores the theory, method, and practical application of a Butterworth low-pass filter in a simple crossover trading strategy.

🔥 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

Butterworth Filter

The Butterworth filter is a type of signal filter designed to have as flat a frequency response as possible in the passband. The digital version applies a difference equation to smooth price data.

The basic recursive filter equation is:

$$y[n] = b_0 x[n] + b_1 x[n-1] + \cdots - a_1 y[n-1] - a_2 y[n-2] \ldots$$

Where:

- $x[n]$: input data (e.g., price)
- $y[n]$: filtered output
- b, a: filter coefficients from `signal.butter(...)`
- N: filter order
- W_n: normalized cutoff frequency (relative to Nyquist)

The filter is applied using its transfer function coefficients, denoted as (numerator) and (denominator), which are derived

from the filter's design. The `scipy.signal.butter` function computes these coefficients. The filtering itself is performed using `scipy.signal.lfilter`, which applies a linear filter to data. For real-time applications like trading, it's crucial to manage the filter's internal state (`zi`) to ensure continuous, accurate filtering of new data points without re-filtering the entire historical series.

The Indicator: `ButterworthFilter`

Our `ButterworthFilter` indicator wraps this digital signal processing logic within Backtrader:

```
class ButterworthFilter(bt.Indicator):
    lines = ('filtered',)
    params = (
        ('cutoff_freq', 0.1),  # Normalized cutoff frequency (0 to
        ('order', 2),          # Filter order
        ('lookback', 100),     # Data points needed for initial sta
    )

    def __init__(self):
        self.addminperiod(self.params.lookback)
        # Design the filter (coefficients b, a)
        self.b, self.a = signal.butter(
            N=self.params.order,
            Wn=self.params.cutoff_freq,
            btype='low',       # Low-pass filter
            analog=False       # Digital filter
        )
        self.data_buffer = deque(maxlen=self.params.lookback)
        self.zi = signal.lfilter_zi(self.b, self.a) # Initial filte

    def next(self):
```

```
        current_price = self.data.close[0]
        self.data_buffer.append(current_price)

        if len(self.data_buffer) < self.params.lookback:
            self.lines.filtered[0] = current_price # Not enough dat
            return

        # Apply filter using the 'zi' (initial conditions) for cont
        filtered_point, self.zi = signal.lfilter(
            self.b, self.a, [current_price], zi=self.zi
        )
        self.lines.filtered[0] = filtered_point[0]
```

The `lookback` parameter ensures that the filter has enough initial data to stabilize its output, preventing artifacts at the beginning of the series. The `zi` (initial conditions) array is crucial for `lfilter` to process new data points sequentially, maintaining the filter's state from the previous calculation.

The Strategy: `ButterworthCrossoverStrategy`

The trading strategy employs two Butterworth filters: a “fast” filter with a higher `cutoff_freq` (less smoothing, less lag) and a "slow" filter with a lower `cutoff_freq` (more smoothing, more lag). Signals are generated based on the crossover of these two filtered lines, analogous to traditional moving average crossovers.

Additionally, the strategy incorporates:

- Trend Confirmation: A `trend_threshold` parameter demands a minimum difference between the fast and slow filters for a signal to be considered valid, aiming to filter out weak crossovers.
- Trailing Stop Loss: Dynamically adjusts a stop loss to lock in profits as a trade moves favorably.
- Fixed Stop Loss: A static percentage-based stop loss from the entry price provides a hard limit on potential losses.

```
class ButterworthCrossoverStrategy(bt.Strategy):
    params = (
        ('fast_cutoff', 0.15),
        ('slow_cutoff', 0.05),
        ('filter_order', 2),
        ('lookback', 50),
        ('trailing_stop_pct', 0.04),
        ('stop_loss_pct', 0.08),
        ('trend_threshold', 0.001), # Minimum relative difference f
        ('printlog', False),
    )

    def __init__(self):
        # Initialize fast and slow Butterworth filters
        self.fast_filter = ButterworthFilter(self.data.close, cutof
        self.slow_filter = ButterworthFilter(self.data.close, cutof

        # Crossover indicator for signals
        self.crossover = bt.indicators.CrossOver(self.fast_filter,
        self.filter_diff = self.fast_filter - self.slow_filter # Fo

        self.order = None
        self.trailing_stop_price = None
        self.entry_price = None
        self.last_signal = 0 # To prevent re-entry on same signal
```

```
    def next(self):
        if self.order: return # Only one order at a time
        current_price = self.data.close[0]
        fast_val = self.fast_filter[0]
        slow_val = self.slow_filter[0]

        # Ensure filters have enough data to be valid
        if len(self) < self.params.lookback * 2 or np.isnan(fast_va
            return

        # Entry Logic
        if not self.position:
            filter_diff_ratio = (fast_val - slow_val) / abs(slow_va

            # Long signal: Fast filter crosses above slow with suff
            if self.crossover[0] > 0 and filter_diff_ratio > self.p
                self.log(f'BUY CREATE: {current_price:.2f}')
                self.order = self.buy()
                self.last_signal = 1
            # Short signal: Fast filter crosses below slow with suf
            elif self.crossover[0] < 0 and filter_diff_ratio < -sel
                self.log(f'SELL CREATE: {current_price:.2f}')
                self.order = self.sell()
                self.last_signal = -1

        # Exit and Stop-Loss/Trailing-Stop Logic (simplified for co
        else: # If in a position
            # ... (Full code includes logic for updating trailing_s
            #     exit conditions based on crossover reversal, trai
            pass
```

The `trend_threshold` adds an interesting dimension, requiring not just a crossover but a sustained difference between the filters, potentially reducing false signals.

Backtesting and Initial Observations

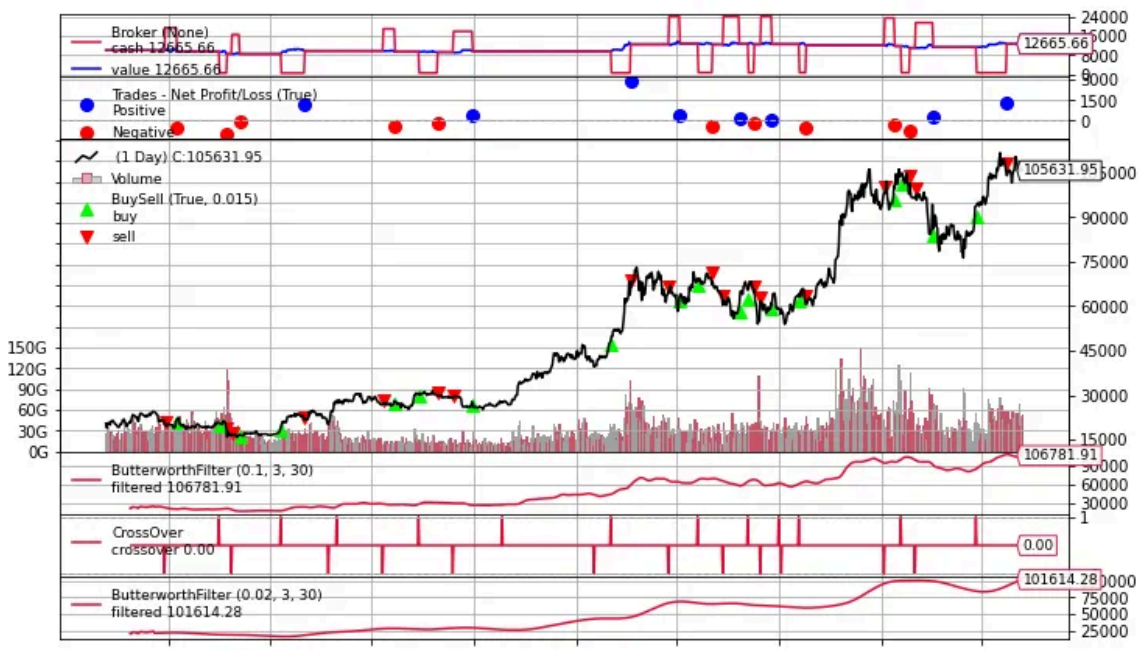
The strategy is evaluated using a Backtrader environment with historical BTC-USD data. The `run_butterworth_strategy()` function sets up the `Cerebro` engine, adds the strategy with customizable parameters, fetches data from `yfinance` (respecting the `auto_adjust=False` and `droplevel` instruction), and includes standard financial analyzers.

```
def run_butterworth_strategy():
    print("Downloading BTC-USD data...")
    # Using saved instruction: yfinance download with auto_adjust=F
    data = yf.download('BTC-USD', period='3y', auto_adjust=False).d

    cerebro = bt.Cerebro()
    cerebro.addstrategy(ButterworthCrossoverStrategy,
                        fast_cutoff=0.1, slow_cutoff=0.02, filter_o
                        trailing_stop_pct=0.05, stop_loss_pct=0.1,
    cerebro.adddata(bt.feeds.PandasData(dataname=data))
    cerebro.broker.setcash(10000.0)
    cerebro.broker.setcommission(commission=0.001)
    cerebro.addsizer(bt.sizers.PercentSizer, percents=95)

    # Add analyzers for performance evaluation
    cerebro.addanalyzer(bt.analyzers.SharpeRatio, _name='sharpe')
    cerebro.addanalyzer(bt.analyzers.DrawDown, _name='drawdown')
    cerebro.addanalyzer(bt.analyzers.TradeAnalyzer, _name='trades')
    cerebro.addanalyzer(bt.analyzers.Returns, _name='returns')

    print(f'Starting Portfolio Value: {cerebro.broker.getvalue():.2
    results = cerebro.run()
    strat = results[0]
    print(f'Final Portfolio Value: {cerebro.broker.getvalue():.2f}'
    # ... (code to print detailed analysis results and plot)
```



The backtest provides valuable metrics like Sharpe Ratio and Max Drawdown, alongside a visual representation of the equity curve. These initial results are crucial for understanding the strategy's behavior in specific historical contexts.

Conclusion: A Promising Avenue for Noise Reduction

The Butterworth digital filter offers a powerful, mathematically grounded approach to smoothing financial data, providing a compelling alternative to traditional moving averages. By precisely controlling the cutoff frequency and filter order, traders can fine-tune the balance between noise reduction and lag. The `ButterworthCrossoverStrategy` demonstrates how these filters can be integrated into a complete trading system with robust risk management. While this initial exploration shows the

potential of such an approach in filtering market noise and identifying trends, comprehensive optimization and extensive out-of-sample testing are essential next steps to validate its robustness and profitability across diverse market conditions.

Python | Algorithmic Trading | Trading Strategy | Bitcoin | Finance

Written by PyQuantLab

655 followers · 6 following

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

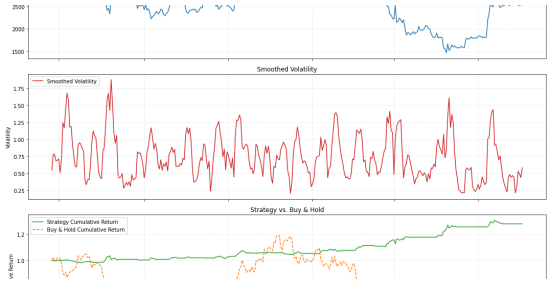
Following

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...

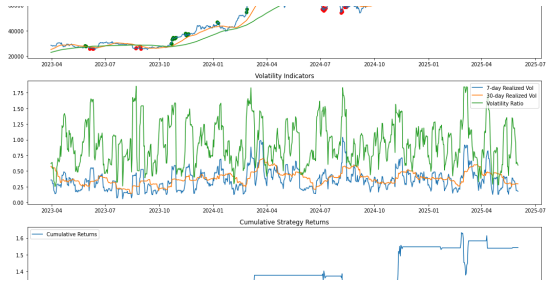
Jun 3 32 3

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

PyQuantLab

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

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

Jun 3 60

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

Jun 4 64

See all from PyQuantLab

Recommended from Medium

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

PyQuantLab

Precision Trading with Volume Profile: An Enhance...

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

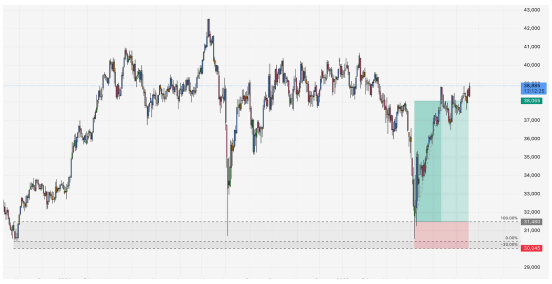
Jun 20 11

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



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

In DataDrivenInvestor by Mr. Q

Why You Should Ignore Most Backtested Trading...

To be more precise, while the title is limited in length, what I truly mean ar...

Jun 18 133 2

Unicorn Day

Why Buy-and-Hold Is Your Portfolio's Silent Killer Durin...

You're sitting in your favorite coffee shop , scrolling through financial...

Jun 23 6

See more recommendations