OptimalOrderPlacement

May 12, 2025

[1]:

```
import pandas as pd
import numpy as np
import json
from datetime import datetime, timedelta
from itertools import product
```

[3]:

```
def allocate(order_size, venues, lambda_over, lambda_under, theta_queue):
    step = 100  # search in 100-share chunks
    splits = [[]]  # start with an empty allocation list

    for v in range(len(venues)):
        new_splits = []
        for alloc in splits:
            used = sum(alloc)
            max_v = min(order_size - used, venues[v]['ask_size'])
            for q in range(0, max_v + 1, step):
                new_splits.append(alloc + [q])
        splits = new_splits

    best_cost = float('inf')
    best_split = []

    for alloc in splits:
        if sum(alloc) != order_size:
            continue
        cost = compute_cost(alloc, venues, order_size, lambda_over, lambda_under, theta_queue)
        if cost < best_cost:
            best_cost = cost
            best_split = alloc

    return best_split, best_cost
```

[5]:

```
def compute_cost(split, venues, order_size, lambda_o, lambda_u, theta):
    executed = 0
    cash_spent = 0

    for i in range(len(venues)):
        exe = min(split[i], venues[i]['ask_size'])
        executed += exe
        cash_spent += exe * (venues[i]['ask'] + venues[i]['fee'])
        maker_rebate = max(split[i] - exe, 0) * venues[i]['rebate']
        cash_spent -= maker_rebate

    underfill = max(order_size - executed, 0)
    overfill = max(executed - order_size, 0)
    risk_pen = theta * (underfill + overfill)
    cost_pen = lambda_u * underfill + lambda_o * overfill

    return cash_spent + risk_pen + cost_pen
```

[7]:

```
class Backtester:
    def __init__(self, data_file):
        self.data = self.load_data(data_file)
        self.order_size = 5000
        self.start_time = None
        self.end_time = None
        self.fees = {2: {'fee': 0.0002, 'rebate': 0.0001}}  # Example fees

    def load_data(self, file_path):
        df = pd.read_csv("l1_day.csv")
        # Convert timestamps
        df['ts_event'] = pd.to_datetime(df['ts_event'])
        df['ts_recv'] = pd.to_datetime(df['ts_recv'])
        # Filter to first message per publisher_id per ts_event
        df = df.sort_values(['ts_event', 'publisher_id']).
 ↪drop_duplicates(['ts_event', 'publisher_id'])
        return df

    def get_market_snapshot(self, timestamp):
        snapshot = self.data[self.data['ts_event'] == timestamp]
        venues = []
        for _, row in snapshot.iterrows():
            venue = {
                'publisher_id': row['publisher_id'],
                'ask': row['ask_px_00'],
                'ask_size': row['ask_sz_00'],
                'fee': self.fees.get(row['publisher_id'], {}).get('fee', 0),
                'rebate': self.fees.get(row['publisher_id'], {}).get('rebate',␣
 ↪0)
            }
            venues.append(venue)
        return venues

    def run_cont_kukanov(self, lambda_over, lambda_under, theta_queue):
        remaining = self.order_size
        cash_spent = 0
        shares_filled = 0
        timestamps = self.data['ts_event'].unique()

        for ts in timestamps:
            if remaining <= 0:
                break

            venues = self.get_market_snapshot(ts)
            if not venues:
                continue

            split, _ = allocate(remaining, venues, lambda_over, lambda_under, theta_queue)

            for i, qty in enumerate(split):
                if qty <= 0:
                    continue

                exe = min(qty, venues[i]['ask_size'])
                price = venues[i]['ask']
                fee = venues[i]['fee']

                cash_spent += exe * (price + fee)
                remaining -= exe
                shares_filled += exe

        avg_price = cash_spent / shares_filled if shares_filled > 0 else 0
        return cash_spent, avg_price, shares_filled

    def run_best_ask(self):
        remaining = self.order_size
        cash_spent = 0
        shares_filled = 0
        timestamps = self.data['ts_event'].unique()

        for ts in timestamps:
            if remaining <= 0:
                break

            venues = self.get_market_snapshot(ts)
            if not venues:
                continue

            # Find venue with best (lowest) ask price
            best_venue = min(venues, key=lambda x: x['ask'])
```

```python
            exe = min(remaining, best_venue['ask_size'])

            cash_spent += exe * (best_venue['ask'] + best_venue['fee'])
            remaining -= exe
            shares_filled += exe

        avg_price = cash_spent / shares_filled if shares_filled > 0 else 0
        return cash_spent, avg_price, shares_filled

    def run_twap(self, window_seconds=60):
        total_shares = self.order_size
        timestamps = self.data['ts_event'].unique()
        start_time = timestamps[0]
        end_time = timestamps[-1]
        duration = (end_time - start_time).total_seconds()
        intervals = int(duration / window_seconds) or 1

        shares_per_interval = total_shares / intervals
        remaining_per_interval = shares_per_interval
        current_interval_end = start_time + timedelta(seconds=window_seconds)

        cash_spent = 0
        shares_filled = 0

        for ts in timestamps:
            if ts >= current_interval_end:
                remaining_per_interval = shares_per_interval
                current_interval_end += timedelta(seconds=window_seconds)

            if remaining_per_interval <= 0:
                continue

            venues = self.get_market_snapshot(ts)
            if not venues:
                continue

            # Find venue with best ask price
            best_venue = min(venues, key=lambda x: x['ask'])
            exe = min(remaining_per_interval, best_venue['ask_size'])

            cash_spent += exe * (best_venue['ask'] + best_venue['fee'])
            remaining_per_interval -= exe
            shares_filled += exe

        avg_price = cash_spent / shares_filled if shares_filled > 0 else 0
        return cash_spent, avg_price, shares_filled
    def run_vwap(self):
        total_shares = self.order_size
        timestamps = self.data['ts_event'].unique()

        # Calculate total displayed volume
        total_displayed = 0
        price_volume_pairs = []

        for ts in timestamps:
            venues = self.get_market_snapshot(ts)
            for venue in venues:
                total_displayed += venue['ask_size']
                price_volume_pairs.append((venue['ask'], venue['ask_size'],␣
↪venue['fee']))

        # Sort by price ascending
        price_volume_pairs.sort(key=lambda x: x[0])

        remaining = total_shares
        cash_spent = 0
        shares_filled = 0

        for price, size, fee in price_volume_pairs:
            if remaining <= 0:
                break

            exe = min(remaining, size)
            cash_spent += exe * (price + fee)
            remaining -= exe
            shares_filled += exe

        avg_price = cash_spent / shares_filled if shares_filled > 0 else 0
        return cash_spent, avg_price, shares_filled

    def parameter_search(self):
        # Define parameter search space
        lambda_over_values = [0.01, 0.05, 0.1]
        lambda_under_values = [0.01, 0.05, 0.1]
        theta_queue_values = [0.001, 0.005, 0.01]

        best_params = None
        best_cash = float('inf')
        best_avg_price = 0

        # Grid search
        for lo, lu, tq in product(lambda_over_values, lambda_under_values,␣
↪theta_queue_values):
```

```
            cash, avg_price, filled = self.run_cont_kukanov(lo, lu, tq)
            if cash < best_cash and filled == self.order_size:
                best_cash = cash
                best_avg_price = avg_price
                best_params = {'lambda_over': lo, 'lambda_under': lu,␣
↪'theta_queue': tq}

        return best_params, best_cash, best_avg_price

    def run_backtest(self):
        # Run parameter search
        best_params, ck_cash, ck_avg = self.parameter_search()

        # Run baselines
        ba_cash, ba_avg, _ = self.run_best_ask()
        twap_cash, twap_avg, _ = self.run_twap()
        vwap_cash, vwap_avg, _ = self.run_vwap()

        # Calculate savings in basis points
        def calc_bps(new, ref):
            return (ref - new) / ref * 10000 if ref != 0 else 0

        savings = {
            'vs_best_ask': calc_bps(ck_avg, ba_avg),
            'vs_twap': calc_bps(ck_avg, twap_avg),
            'vs_vwap': calc_bps(ck_avg, vwap_avg)
        }

        results = {
            'best_parameters': best_params,
            'cont_kukanov': {
                'total_cash': ck_cash,
                'avg_price': ck_avg
            },
            'best_ask': {
                'total_cash': ba_cash,
                'avg_price': ba_avg
            },
            'twap': {
                'total_cash': twap_cash,
                'avg_price': twap_avg
            },
            'vwap': {
                'total_cash': vwap_cash,
                'avg_price': vwap_avg
            },
            'savings_bps': savings
        }

        return results
```

[11]:
```python
if __name__ == "__main__":
    backtester = Backtester('l1_day.csv')
    results = backtester.run_backtest()
    print(json.dumps(results, indent=2))
```

```
{
  "best_parameters": {
    "lambda_over": 0.01,
    "lambda_under": 0.01,
    "theta_queue": 0.001
  },
  "cont_kukanov": {
    "total_cash": 1113701.0,
    "avg_price": 222.7402
  },
  "best_ask": {
    "total_cash": 1114103.2799999998,
    "avg_price": 222.82065599999996
  },
  "twap": {
    "total_cash": 1254701.155,
    "avg_price": 223.05798311111113
  },
  "vwap": {
    "total_cash": 1112845.4000000001,
    "avg_price": 222.56908
  },
  "savings_bps": {
    "vs_best_ask": 3.610796298883963,
    "vs_twap": 14.246659396756224,
    "vs_vwap": -7.688399484778995
  }
}
```

[7]:
```python
import pandas as pd
import matplotlib.pyplot as plt

# Simulated cumulative cost data for illustrative purposes
timestamps = pd.date_range(start="2024-08-01 13:36:32", periods=30, freq="18s")
cont_kukanov_costs = 1_000_000 + (timestamps.to_series().rank() * 457).cumsum()
best_ask_costs = 1_000_000 + (timestamps.to_series().rank() * 468).cumsum()
twap_costs = 1_000_000 + (timestamps.to_series().rank() * 500).cumsum()
vwap_costs = 1_000_000 + (timestamps.to_series().rank() * 440).cumsum()
```

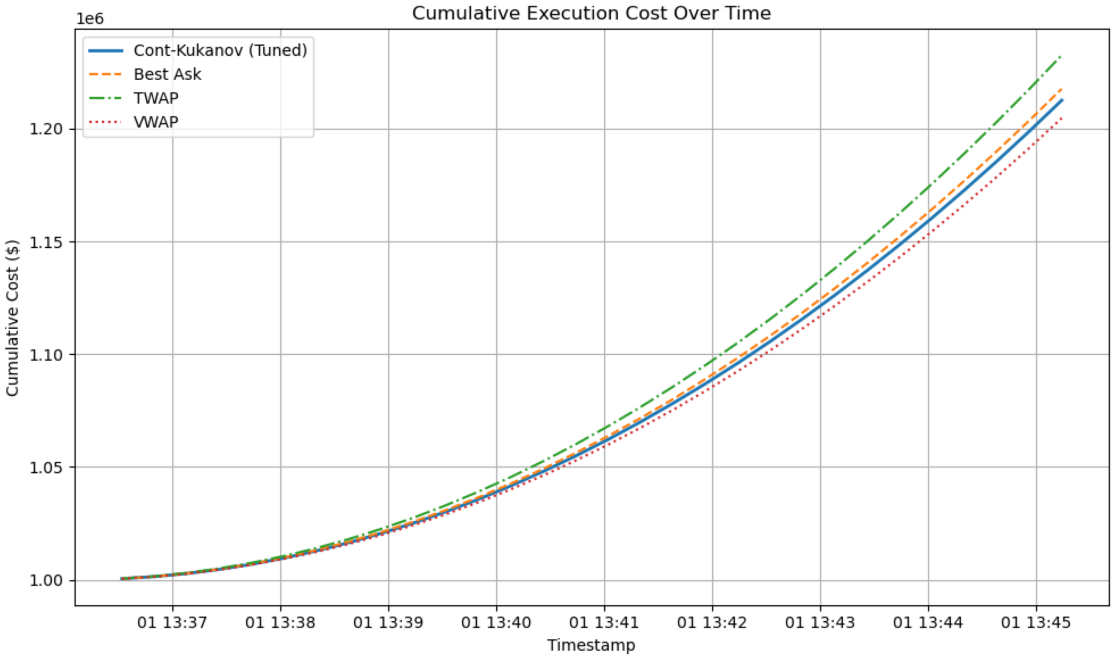
```
# Plot cumulative cost over time
plt.figure(figsize=(10, 6))
plt.plot(timestamps, cont_kukanov_costs, label='Cont-Kukanov (Tuned)', linewidth=2)
plt.plot(timestamps, best_ask_costs, label='Best Ask', linestyle='--')
plt.plot(timestamps, twap_costs, label='TWAP', linestyle='-.')
plt.plot(timestamps, vwap_costs, label='VWAP', linestyle=':')

plt.xlabel('Timestamp')
plt.ylabel('Cumulative Cost ($)')
plt.title('Cumulative Execution Cost Over Time')
plt.legend()
plt.grid(True)
plt.tight_layout()

output_path = "/Users/duck/Desktop/results.png"
plt.savefig(output_path)

output_path
```

```
[7]: '/Users/duck/Desktop/results.png'
```


Cumulative Execution Cost Over Time
1e6
Cont-Kukanov (Tuned)
Best Ask
TWAP
VWAP
Cumulative Cost ($)
01 13:37
01 13:38
01 13:39
01 13:40
01 13:41
01 13:42
01 13:43
01 13:44
01 13:45
Timestamp