Module 4: Market Impact Modeling

Introduction

Accurate market impact modeling is crucial for MEV strategy success. This module covers sophisticated price impact models, liquidity dynamics, order flow analysis, and market microstructure effects that influence trade execution in decentralized finance markets.

Learning Objectives

By the end of this module, you will be able to:
- Implement advanced price impact models for different market conditions
- Simulate liquidity dynamics and depth analysis
- Model order flow and its impact on market prices
- Analyze market microstructure effects in DeFi markets
- Build adaptive models that respond to changing market conditions

4.1 Price Impact Fundamentals

4.1.1 Quadratic Price Impact Model

```
from typing import Dict, List, Any, Optional, Tuple
from dataclasses import dataclass
from enum import Enum
import numpy as np
import pandas as pd
from datetime import datetime, timedelta

class MarketRegime(Enum):
    NORMAL = "normal"
    VOLATILE = "volatile"
    ILLIQUID = "illiquid"
    CONGESTED = "congested"

@dataclass
class MarketState:
    """Current market state snapshot"""
    timestamp: datetime
    price: float
    volume_24h: float
    liquidity: float
    spread: float
    volatility: float
    block_utilization: float
    mempool_size: int
    gas_price: int

@dataclass
class Order:
    """Individual order in the market"""
    order_id: str
    side: str  # 'buy' or 'sell'
    size: float
    price_limit: Optional[float]
    order_type: str  # 'market', 'limit', 'stop'
    timestamp: datetime
    expires_at: Optional[datetime]
    filled_size: float = 0.0

@dataclass
class TradeExecution:
    """Trade execution result"""
    order_id: str
```

```
    executed_size: float
    execution_price: float
    slippage: float
    market_impact: float
    liquidity_consumed: float
    timestamp: datetime

class PriceImpactModel:
    """Advanced price impact model for DeFi markets"""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.impact_coefficients = {
            MarketRegime.NORMAL: {'alpha': 0.5, 'beta': 1.2},
            MarketRegime.VOLATILE: {'alpha': 0.8, 'beta': 1.5},
            MarketRegime.ILLIQUID: {'alpha': 1.2, 'beta': 2.0},
            MarketRegime.CONGESTED: {'alpha': 1.0, 'beta': 1.8}
        }

    def calculate_price_impact(self, order_size: float, market_state:
MarketState,
                               market_regime: MarketRegime =
MarketRegime.NORMAL) -> Dict[str, float]:
        """Calculate price impact using quadratic model"""

        # Get coefficients for market regime
        coeffs = self.impact_coefficients.get(market_regime,
self.impact_coefficients[MarketRegime.NORMAL])
        alpha = coeffs['alpha']
        beta = coeffs['beta']

        # Base liquidity (normalized)
        normalized_liquidity = market_state.liquidity /
(market_state.volume_24h + 1)

        # Order size relative to liquidity
        size_ratio = order_size / (market_state.liquidity + 1)

        # Calculate temporary price impact (temporary)
        # Formula: ΔP_temp = α * (Q/L)^β * P
        temp_impact = alpha * (size_ratio ** beta) * market_state.price
```

```
        # Calculate permanent price impact (permanent)
        # Formula: ΔP_perm = γ * (Q/V)^δ * P
        gamma = 0.3  # Permanent impact coefficient
        delta = 0.8  # Permanent impact exponent

        volume_ratio = order_size / (market_state.volume_24h + 1)
        perm_impact = gamma * (volume_ratio ** delta) *
market_state.price

        # Calculate total price impact
        total_impact = temp_impact + perm_impact

        # Calculate slippage (percentage)
        slippage = (total_impact / market_state.price) * 100

        return {
            'temporary_impact': temp_impact,
            'permanent_impact': perm_impact,
            'total_impact': total_impact,
            'slippage_percent': slippage,
            'execution_price': market_state.price + total_impact,
            'impact_cost': order_size * total_impact
        }

    def calculate_adaptive_impact(self, order_size: float,
market_state: MarketState,
                                  recent_trades: List[TradeExecution] =
None) -> Dict[str, float]:
        """Calculate adaptive price impact based on recent market
activity"""

        base_impact = self.calculate_price_impact(order_size,
market_state)

        # Adjust impact based on recent trades
        if recent_trades:
            recent_volume = sum(trade.executed_size for trade in
recent_trades[-100:])
            recent_price_impact = np.mean([trade.market_impact for
trade in recent_trades[-20:]])

            # Adjust coefficients based on recent activity
```

```
            volume_factor = min(2.0, 1.0 + (recent_volume /
market_state.volume_24h) * 0.5)
            impact_factor = min(1.5, 0.8 + recent_price_impact /
market_state.price)

            # Apply adjustments
            base_impact['total_impact'] *= volume_factor *
impact_factor
            base_impact['slippage_percent'] =
(base_impact['total_impact'] / market_state.price) * 100
            base_impact['execution_price'] = market_state.price +
base_impact['total_impact']
            base_impact['adaptive_adjustment'] = (volume_factor - 1) *
100
        else:
            base_impact['adaptive_adjustment'] = 0

        return base_impact

    def model_order_book_depth(self, market_state: MarketState,
                               order_size: float) -> Dict[str, float]:
        """Model available liquidity at different price levels"""

        # Generate synthetic order book based on market state
        price_levels = 20
        depth_curve = []

        base_price = market_state.price
        current_depth = market_state.liquidity

        for i in range(price_levels):
            # Price moves away from mid-price
            if i < price_levels // 2:
                # Below mid-price (bids)
                price_offset = - (i + 1) * market_state.spread /
price_levels
            else:
                # Above mid-price (asks)
                price_offset = (i - price_levels // 2 + 1) *
market_state.spread / price_levels

            price = base_price * (1 + price_offset)
```

```
            # Depth decreases as we move away from mid-price
            depth_multiplier = np.exp(-0.3 * abs(i - price_levels //
2))
            depth_at_level = current_depth * depth_multiplier

            depth_curve.append({
                'price': price,
                'depth': depth_at_level,
                'cumulative_depth': sum(d['depth'] for d in
depth_curve) + depth_at_level
            })

        # Calculate how much depth is consumed by our order
        remaining_size = order_size
        consumed_depth = 0
        price_levels_used = []

        for level in depth_curve:
            if remaining_size <= 0:
                break

            available_at_level = level['depth']
            if available_at_level >= remaining_size:
                # Order fills at this level
                price_levels_used.append({
                    'price': level['price'],
                    'size': remaining_size
                })
                consumed_depth += remaining_size
                remaining_size = 0
            else:
                # Order consumes entire level
                price_levels_used.append({
                    'price': level['price'],
                    'size': available_at_level
                })
                consumed_depth += available_at_level
                remaining_size -= available_at_level

        # Calculate volume-weighted average execution price
        if consumed_depth > 0:
```

```
            total_cost = sum(level['price'] * level['size'] for level
in price_levels_used)
            avg_price = total_cost / consumed_depth
            market_impact = (avg_price - base_price) / base_price
        else:
            avg_price = base_price
            market_impact = 0

        return {
            'execution_price': avg_price,
            'market_impact': market_impact,
            'slippage': market_impact * 100,
            'depth_consumed': consumed_depth,
            'remaining_size': remaining_size,
            'price_levels_used': price_levels_used
        }

class LiquidityAnalysis:
    """Analyzes liquidity dynamics and depth"""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.liquidity_history = []

    def analyze_liquidity_provision(self, market_state: MarketState,
                                    expected_trade_size: float) ->
Dict[str, Any]:
        """Analyze liquidity provision dynamics"""

        # Liquidity response to expected trade
        liquidity_response =
self._model_liquidity_response(expected_trade_size, market_state)

        # Slippage estimation
        slippage_estimation =
self._estimate_slippage(expected_trade_size, market_state)

        # Liquidity concentration analysis
        concentration =
self._analyze_liquidity_concentration(market_state)

        # Risk assessment
```

```
        liquidity_risk =
self._assess_liquidity_risk(expected_trade_size, market_state)

        return {
            'liquidity_response': liquidity_response,
            'slippage_estimation': slippage_estimation,
            'concentration_analysis': concentration,
            'liquidity_risk': liquidity_risk
        }

    def _model_liquidity_response(self, expected_size: float,
                                  market_state: MarketState) -> Dict[str,
float]:
        """Model how liquidity providers respond to expected large
trades"""

        # Liquidity tends to increase when large trades are expected
        liquidity_boost = 1.0

        # Boost factor based on expected size relative to normal volume
        size_ratio = expected_size / (market_state.volume_24h + 1)

        if size_ratio > 0.1:  # Expected trade > 10% of daily volume
            liquidity_boost = 1.2 + (size_ratio - 0.1) * 0.5
        elif size_ratio > 0.05:  # Expected trade 5-10% of daily volume
            liquidity_boost = 1.1

        # Adjust for market regime
        if market_state.block_utilization > 0.8:  # Congested network
            liquidity_boost *= 0.8  # Liquidity providers reduce
activity

        if market_state.volatility > 0.05:  # High volatility
            liquidity_boost *= 0.9  # Reduced liquidity provision

        return {
            'boost_factor': liquidity_boost,
            'new_liquidity_level': market_state.liquidity *
liquidity_boost,
            'expected_additional_liquidity': market_state.liquidity *
(liquidity_boost - 1)
        }
```

```
    def _estimate_slippage(self, trade_size: float,
                           market_state: MarketState) -> Dict[str,
float]:
        """Estimate slippage for given trade size"""

        # Linear relationship for small trades
        base_slippage = 0.001  # 0.1% base slippage

        # Quadratic relationship for larger trades
        size_ratio = trade_size / (market_state.liquidity + 1)
        quadratic_slippage = 0.01 * (size_ratio ** 2)

        # Adjust for market conditions
        volatility_adjustment = 1 + market_state.volatility * 10
        congestion_adjustment = 1 + (market_state.block_utilization -
0.5) * 0.5

        total_slippage = (base_slippage + quadratic_slippage) *
volatility_adjustment * congestion_adjustment

        return {
            'base_slippage': base_slippage,
            'quadratic_component': quadratic_slippage,
            'volatility_adjustment': volatility_adjustment,
            'congestion_adjustment': congestion_adjustment,
            'total_slippage': total_slippage,
            'slippage_cost_usd': trade_size * market_state.price *
total_slippage
        }

    def _analyze_liquidity_concentration(self, market_state:
MarketState) -> Dict[str, Any]:
        """Analyze concentration of liquidity provision"""

        # Simulate liquidity concentration across price levels
        # In DeFi, liquidity can be concentrated at specific price
ranges


# Assume liquidity follows a normal distribution around current price
        spread = market_state.spread
```

```
        concentration_factor = np.random.uniform(0.3, 0.8)  # Random
concentration

        # Calculate Gini coefficient for liquidity distribution
        # (higher values indicate more concentration)
        liquidity_gini = 1 - concentration_factor

        # Liquidity depth at different distances from mid-price
        depth_profile = {}
        for distance_pct in [0.25, 0.5, 1.0, 2.0, 5.0]:
            depth_at_distance = market_state.liquidity *
concentration_factor * np.exp(-distance_pct)
            depth_profile[f'{distance_pct}%'] = {
                'depth': depth_at_distance,
                'percentage_of_total': (depth_at_distance /
market_state.liquidity) * 100
            }

        return {
            'concentration_factor': concentration_factor,
            'liquidity_gini': liquidity_gini,
            'depth_profile': depth_profile,
            'concentration_description':
self._describe_concentration(concentration_factor)
        }

    def _describe_concentration(self, concentration_factor: float) ->
str:
        """Provide textual description of concentration level"""

        if concentration_factor > 0.7:
            return "Highly concentrated liquidity"
        elif concentration_factor > 0.5:
            return "Moderately concentrated liquidity"
        else:
            return "Well-distributed liquidity"

    def _assess_liquidity_risk(self, trade_size: float,
                               market_state: MarketState) -> Dict[str,
Any]:
        """Assess liquidity risk for given trade size"""
```

```
        # Calculate risk metrics
        size_to_liquidity_ratio = trade_size / (market_state.liquidity
+ 1)
        size_to_volume_ratio = trade_size / (market_state.volume_24h +
1)

        # Risk categories
        if size_to_liquidity_ratio > 0.2:
            liquidity_risk_level = "High"
        elif size_to_liquidity_ratio > 0.1:
            liquidity_risk_level = "Medium"
        else:
            liquidity_risk_level = "Low"

        # Expected execution time (blocks)
        if market_state.block_utilization > 0.9:
            expected_blocks = 5
        elif market_state.block_utilization > 0.7:
            expected_blocks = 3
        else:
            expected_blocks = 1

        # Gas cost estimation
        base_gas = 150000  # Swap gas usage
        gas_cost_eth = base_gas * market_state.gas_price / 1e18
        gas_cost_usd = gas_cost_eth * (market_state.price / 2000)  #
Assuming ETH price

        return {
            'liquidity_risk_level': liquidity_risk_level,
            'size_to_liquidity_ratio': size_to_liquidity_ratio,
            'size_to_volume_ratio': size_to_volume_ratio,
            'expected_execution_blocks': expected_blocks,
            'estimated_gas_cost_usd': gas_cost_usd,
            'recommendation':
self._get_risk_recommendation(liquidity_risk_level)
        }

    def _get_risk_recommendation(self, risk_level: str) -> str:
        """Get trading recommendation based on risk level"""

        recommendations = {
```

```
            "High": "Consider breaking into smaller orders or waiting
for better liquidity",
            "Medium": "Monitor market conditions closely and consider
limit orders",
            "Low": "Proceed with execution, monitor for any market
changes"
        }

        return recommendations.get(risk_level, "Monitor market
conditions")

# Example usage
price_impact_model = PriceImpactModel({})
liquidity_analyzer = LiquidityAnalysis({})

# Example market state
market_state = MarketState(
    timestamp=datetime.now(),
    price=2000.0,
    volume_24h=100000.0,
    liquidity=10000.0,
    spread=0.01,
    volatility=0.03,
    block_utilization=0.6,
    mempool_size=5000,
    gas_price=25_000_000_000
)

# Calculate price impact for different order sizes
for size in [100, 500, 1000, 5000]:
    impact = price_impact_model.calculate_price_impact(size,
market_state, MarketRegime.NORMAL)
    print(f"Order size {size}: {impact['slippage_percent']:.2f}%
slippage")

# Analyze liquidity
liquidity_analysis =
liquidity_analyzer.analyze_liquidity_provision(market_state, 1000)
print(f"Liquidity boost factor:
{liquidity_analysis['liquidity_response']['boost_factor']:.2f}")
print(f"Slippage estimation: {liquidity_analysis['slippage_estimation']
['total_slippage']:.3f}")
```

4.1.2 Order Flow Impact Model

```
from collections import defaultdict, deque
from typing import Dict, List, Any, Optional, Tuple
from dataclasses import dataclass
import numpy as np
from datetime import datetime, timedelta

@dataclass
class OrderFlowEvent:
    """Order flow event in the market"""
    timestamp: datetime
    order_id: str
    side: str  # 'buy' or 'sell'
    size: float
    price: Optional[float]
    order_type: str
    market_impact: float
    source: str  # 'mev_bot', 'retail', 'institutional'

class OrderFlowAnalyzer:
    """Analyzes order flow and its market impact"""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.order_flow_history = deque(maxlen=1000)
        self.market_state_history = deque(maxlen=1000)

    def process_order_flow(self, orders: List[Order], market_state:
MarketState) -> List[TradeExecution]:
        """Process order flow and calculate market impact"""

        executions = []

        for order in orders:
            # Calculate market impact based on order flow context
            market_impact = self._calculate_flow_impact(order,
market_state)

            # Execute the order with calculated impact
            execution = self._execute_order_with_impact(order,
market_state, market_impact)
            executions.append(execution)
```

```
            # Record order flow event
            flow_event = OrderFlowEvent(
                timestamp=datetime.now(),
                order_id=order.order_id,
                side=order.side,
                size=order.filled_size,
                price=execution.execution_price,
                order_type=order.order_type,
                market_impact=execution.market_impact,
                source="unknown"
            )

            self.order_flow_history.append(flow_event)

        return executions

    def _calculate_flow_impact(self, order: Order, market_state:
MarketState) -> float:
        """Calculate market impact considering order flow"""

        # Get recent order flow context
        recent_trades = list(self.order_flow_history)[-50:]  # Last 50
trades
        flow_pressure = self._calculate_flow_pressure(recent_trades)

        # Calculate base impact
        price_impact_model = PriceImpactModel({})
        base_impact_result = price_impact_model.calculate_price_impact(
            order.size, market_state
        )

        # Adjust for order flow pressure
        impact_multiplier = 1.0


# Positive flow pressure (buying pressure) increases impact for sell
orders
        if order.side == 'sell' and flow_pressure['buy_pressure'] >
0.1:
            impact_multiplier *= (1 + flow_pressure['buy_pressure'])

        # Negative flow pressure (selling pressure) increases impact
```

```
for buy orders
        if order.side == 'buy' and flow_pressure['sell_pressure'] >
0.1:
            impact_multiplier *= (1 + flow_pressure['sell_pressure'])

        # Speed of execution affects impact
        execution_speed = self._estimate_execution_speed(order)
        if execution_speed > 10:  # Taking more than 10 seconds
            impact_multiplier *= 1.2  # Higher impact for slow
execution

        adjusted_impact = base_impact_result['market_impact'] *
impact_multiplier

        return adjusted_impact

    def _calculate_flow_pressure(self, recent_trades:
List[OrderFlowEvent]) -> Dict[str, float]:
        """Calculate current flow pressure in the market"""

        if not recent_trades:
            return {'buy_pressure': 0, 'sell_pressure': 0, 'net_flow':
0}

        # Calculate buy vs sell pressure
        total_buy_size = sum(trade.size for trade in recent_trades if
trade.side == 'buy')
        total_sell_size = sum(trade.size for trade in recent_trades if
trade.side == 'sell')

        total_volume = total_buy_size + total_sell_size

        if total_volume == 0:
            return {'buy_pressure': 0, 'sell_pressure': 0, 'net_flow':
0}

        buy_pressure = total_buy_size / total_volume
        sell_pressure = total_sell_size / total_volume

        net_flow = buy_pressure - sell_pressure

        # Adjust for time decay (more recent trades have more weight)
```

```
        time_weights = []
        current_time = datetime.now()

        for trade in recent_trades:
            age_seconds = (current_time -
trade.timestamp).total_seconds()
            weight = np.exp(-age_seconds / 300)  # 5-minute decay
            time_weights.append(weight)

        # Weighted flow calculation
        if time_weights:
            weighted_buy = sum(trade.size * weight for trade, weight in
zip(recent_trades, time_weights) if trade.side == 'buy')
            weighted_sell = sum(trade.size * weight for trade, weight
in zip(recent_trades, time_weights) if trade.side == 'sell')
            total_weighted = weighted_buy + weighted_sell

            if total_weighted > 0:
                buy_pressure = weighted_buy / total_weighted
                sell_pressure = weighted_sell / total_weighted
                net_flow = buy_pressure - sell_pressure

        return {
            'buy_pressure': buy_pressure,
            'sell_pressure': sell_pressure,
            'net_flow': net_flow
        }

    def _estimate_execution_speed(self, order: Order) -> float:
        """Estimate execution speed for an order"""

        # Simple estimation based on order size and market liquidity
        # This would need to be more sophisticated in practice

        base_time = 2.0  # 2 seconds base time
        size_factor = order.size / 1000  # Larger orders take more time
        complexity_factor = 2.0 if order.order_type == 'limit' else 1.0

        estimated_time = base_time + size_factor + complexity_factor
        return estimated_time

    def _execute_order_with_impact(self, order: Order, market_state:
```

```
MarketState,
                                market_impact: float) ->
TradeExecution:
        """Execute order with calculated market impact"""

        # Calculate execution price
        if order.side == 'buy':
            execution_price = market_state.price * (1 + market_impact)
        else:  # sell
            execution_price = market_state.price * (1 - market_impact)

        # Calculate slippage
        if order.side == 'buy':
            slippage = (execution_price - market_state.price) /
market_state.price
        else:
            slippage = (market_state.price - execution_price) /
market_state.price

        # Execute full order size (simplified)
        executed_size = order.size

        return TradeExecution(
            order_id=order.order_id,
            executed_size=executed_size,
            execution_price=execution_price,
            slippage=slippage,
            market_impact=market_impact,
            liquidity_consumed=executed_size,
            timestamp=datetime.now()
        )

    def analyze_market_impact_patterns(self) -> Dict[str, Any]:
        """Analyze patterns in market impact"""

        if len(self.order_flow_history) < 20:
            return {"error": "Insufficient data for pattern analysis"}

        # Extract impact data
        impacts = [event.market_impact for event in
self.order_flow_history]
        sizes = [event.size for event in self.order_flow_history]
```

```
        timestamps = [event.timestamp for event in
self.order_flow_history]

        # Calculate correlations
        size_impact_correlation = np.corrcoef(sizes, impacts)[0, 1] if
len(sizes) > 1 else 0

        # Impact statistics
        impact_stats = {
            'mean_impact': np.mean(impacts),
            'std_impact': np.std(impacts),
            'max_impact': np.max(impacts),
            'impact_percentile_95': np.percentile(impacts, 95),
            'impact_percentile_5': np.percentile(impacts, 5)
        }

        # Time-based patterns
        time_patterns = self._analyze_time_patterns(timestamps,
impacts)

        # Size-based patterns
        size_patterns = self._analyze_size_patterns(sizes, impacts)

        # Flow correlation patterns
        flow_patterns = self._analyze_flow_patterns()

        return {
            'impact_statistics': impact_stats,
            'size_impact_correlation': size_impact_correlation,
            'time_patterns': time_patterns,
            'size_patterns': size_patterns,
            'flow_patterns': flow_patterns
        }

    def _analyze_time_patterns(self, timestamps: List[datetime],
                               impacts: List[float]) -> Dict[str, Any]:
        """Analyze temporal patterns in market impact"""

        # Convert to time of day
        hours = [ts.hour for ts in timestamps]

        # Group by hour
```

```
        hourly_impacts = defaultdict(list)
        for hour, impact in zip(hours, impacts):
            hourly_impacts[hour].append(impact)

        # Calculate average impact by hour
        hourly_averages = {}
        for hour in range(24):
            if hour in hourly_impacts:
                hourly_averages[hour] = np.mean(hourly_impacts[hour])
            else:
                hourly_averages[hour] = 0

        # Find peak impact hours
        peak_hours = sorted(hourly_averages.items(), key=lambda x:
x[1], reverse=True)[:3]

        return {
            'hourly_averages': hourly_averages,
            'peak_impact_hours': [hour for hour, _ in peak_hours],
            'peak_impact_values': [impact for _, impact in peak_hours],
            'low_impact_hours': [hour for hour, _ in
sorted(hourly_averages.items(), key=lambda x: x[1])[:3]]
        }

    def _analyze_size_patterns(self, sizes: List[float],
                               impacts: List[float]) -> Dict[str, Any]:
        """Analyze relationship between order size and impact"""

        # Size bins for analysis
        size_bins = [0, 100, 500, 1000, 5000, float('inf')]
        bin_labels = ['<100', '100-500', '500-1000', '1000-5000',
'>5000']

        # Group impacts by size
        binned_impacts = [[] for _ in range(len(size_bins) - 1)]

        for size, impact in zip(sizes, impacts):
            for i in range(len(size_bins) - 1):
                if size_bins[i] <= size < size_bins[i + 1]:
                    binned_impacts[i].append(impact)
                    break
```

```
        # Calculate statistics for each bin
        bin_stats = {}
        for i, label in enumerate(bin_labels):
            if binned_impacts[i]:
                bin_stats[label] = {
                    'mean_impact': np.mean(binned_impacts[i]),
                    'count': len(binned_impacts[i]),
                    'std_impact': np.std(binned_impacts[i])
                }
            else:
                bin_stats[label] = {'mean_impact': 0, 'count': 0,
'std_impact': 0}

        return {
            'size_binned_impacts': bin_stats,
            'impact_scaling_factor':
self._calculate_impact_scaling(sizes, impacts)
        }

    def _calculate_impact_scaling(self, sizes: List[float],
                                  impacts: List[float]) -> float:
        """Calculate how impact scales with order size"""

        # Simple power law fit: impact = a * size^b
        # Take logarithm to linearize: log(impact) = log(a) +
b*log(size)

        if len(sizes) < 2 or any(s <= 0 for s in sizes) or any(i <= 0
for i in impacts):
            return 1.0  # Default scaling

        log_sizes = np.log(sizes)
        log_impacts = np.log(impacts)

        # Linear regression
        try:
            slope, intercept = np.polyfit(log_sizes, log_impacts, 1)
            return slope  # This is the scaling exponent b
        except:
            return 1.0  # Default scaling

    def _analyze_flow_patterns(self) -> Dict[str, Any]:
```

```
        """Analyze order flow patterns"""

        if len(self.order_flow_history) < 10:
            return {}

        recent_flow = list(self.order_flow_history)[-50:]

        # Calculate flow characteristics
        total_buy_volume = sum(event.size for event in recent_flow if
event.side == 'buy')
        total_sell_volume = sum(event.size for event in recent_flow if
event.side == 'sell')

        # Flow imbalance
        total_volume = total_buy_volume + total_sell_volume
        if total_volume > 0:
            flow_imbalance = (total_buy_volume - total_sell_volume) /
total_volume
        else:
            flow_imbalance = 0

        # Flow acceleration (rate of change)
        if len(recent_flow) >= 20:
            early_flow = recent_flow[:10]
            late_flow = recent_flow[-10:]

            early_volume = sum(event.size for event in early_flow)
            late_volume = sum(event.size for event in late_flow)

            flow_acceleration = (late_volume - early_volume) /
(early_volume + 1)
        else:
            flow_acceleration = 0

        return {
            'flow_imbalance': flow_imbalance,
            'flow_acceleration': flow_acceleration,
            'buy_sell_ratio': total_buy_volume / (total_sell_volume +
1),
            'total_recent_volume': total_volume
        }
```

```
# Example usage
flow_analyzer = OrderFlowAnalyzer({})

# Simulate order flow
orders = [
    Order("1", "buy", 500, None, "market", datetime.now()),
    Order("2", "sell", 300, None, "market", datetime.now()),
    Order("3", "buy", 1000, None, "market", datetime.now())
]

market_state = MarketState(
    timestamp=datetime.now(),
    price=2000.0,
    volume_24h=100000.0,
    liquidity=10000.0,
    spread=0.01,
    volatility=0.03,
    block_utilization=0.6,
    mempool_size=5000,
    gas_price=25_000_000_000
)

executions = flow_analyzer.process_order_flow(orders, market_state)

for execution in executions:
    print(f"Order {execution.order_id}: {execution.executed_size} at
{execution.execution_price:.2f}")
    print(f"  Slippage: {execution.slippage:.4f}, Impact:
{execution.market_impact:.4f}")

# Analyze patterns
patterns = flow_analyzer.analyze_market_impact_patterns()
print(f"Mean market impact: {patterns['impact_statistics']
['mean_impact']:.4f}")
print(f"Size-impact correlation: {patterns['size_impact_correlation']:.
2f}")
```

4.2 Advanced Market Microstructure Models

4.2.1 Adaptive Impact Model

```
from typing import Dict, List, Any, Optional
import numpy as np
from scipy.optimize import minimize
from dataclasses import dataclass
from datetime import datetime, timedelta

@dataclass
class MarketMicrostructureState:
    """Detailed market microstructure state"""
    timestamp: datetime
    best_bid: float
    best_ask: float
    bid_size: float
    ask_size: float
    spread: float
    volatility_1m: float
    volatility_5m: float
    volatility_15m: float
    order_flow_imbalance: float
    trade_count_1m: int
    average_trade_size: float
    liquidity_score: float

class AdaptiveImpactModel:
    """Adaptive model that learns from market conditions"""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.model_parameters = {
            'base_impact': 0.001,
            'liquidity_coefficient': 0.5,
            'volatility_coefficient': 1.5,
            'size_exponent': 1.2,
            'time_decay': 0.95,
            'regime_threshold': 0.02
        }

        self.learned_parameters = self.model_parameters.copy()
        self.training_data = []
        self.recent_performance = []

    def fit_model(self, historical_trades: List[TradeExecution],
```

```
                market_states: List[MarketMicrostructureState]) ->
Dict[str, float]:
        """Fit the model to historical data"""

        if len(historical_trades) != len(market_states):
            raise ValueError("Trade data and market states must have
same length")

        # Prepare training data
        X = []
        y = []

        for trade, state in zip(historical_trades, market_states):
            # Feature vector
            features = [
                trade.executed_size,
                state.liquidity_score,
                state.volatility_5m,
                state.order_flow_imbalance,
                state.spread,
                trade.order_id  # Placeholder for order characteristics
            ]
            X.append(features)
            y.append(abs(trade.market_impact))

        # Parameter optimization
        def objective(params):
            """Objective function for parameter optimization"""
            total_error = 0
            for i, (features, actual_impact) in enumerate(zip(X, y)):
                predicted_impact =
self._predict_impact_with_params(features, params)
                error = (predicted_impact - actual_impact) ** 2
                total_error += error
            return total_error

        # Initial parameter guess
        initial_params = list(self.model_parameters.values())

        # Bounds for parameters
        bounds = [
            (0.0001, 0.01),    # base_impact
```

```
            (0.1, 2.0),        # liquidity_coefficient
            (0.5, 5.0),        # volatility_coefficient
            (0.8, 2.0),        # size_exponent
            (0.9, 0.999),      # time_decay
            (0.001, 0.1)       # regime_threshold
        ]

        # Optimize
        result = minimize(objective, initial_params, bounds=bounds,
method='L-BFGS-B')

        if result.success:
            # Update learned parameters
            param_names = ['base_impact', 'liquidity_coefficient',
'volatility_coefficient',
                           'size_exponent', 'time_decay',
'regime_threshold']

            for i, name in enumerate(param_names):
                self.learned_parameters[name] = result.x[i]

            # Calculate training error
            training_error = result.fun / len(X)

            return {
                'optimization_success': True,
                'training_error': training_error,
                'learned_parameters': self.learned_parameters.copy(),
                'training_samples': len(X)
            }
        else:
            return {
                'optimization_success': False,
                'error': result.message
            }

    def _predict_impact_with_params(self, features: List[float],
                                    params: List[float]) -> float:
        """Predict impact using given parameters"""

        size, liquidity, volatility, flow_imbalance, spread, _ =
features
```

```
        base_impact, liq_coeff, vol_coeff, size_exp, time_decay,
regime_thresh = params

        # Base impact
        impact = base_impact

        # Liquidity adjustment
        impact *= 1 + liq_coeff * (1 / (liquidity + 0.001))

        # Volatility adjustment
        impact *= 1 + vol_coeff * volatility

        # Size scaling
        impact *= (size / 1000) ** size_exp

        # Flow imbalance adjustment
        if abs(flow_imbalance) > regime_thresh:
            impact *= 1 + abs(flow_imbalance)

        # Spread adjustment
        impact *= 1 + spread * 10

        return max(0, impact)  # Ensure non-negative

    def predict_impact(self, trade_size: float, market_state:
MarketMicrostructureState,
                       regime: str = 'normal') -> Dict[str, float]:
        """Predict market impact using learned model"""

        features = [
            trade_size,
            market_state.liquidity_score,
            market_state.volatility_5m,
            market_state.order_flow_imbalance,
            market_state.spread,
            regime
        ]

        predicted_impact = self._predict_impact_with_params(features,
list(self.learned_parameters.values()))
```

```
        # Add confidence interval based on recent model performance
        confidence_interval = self._calculate_confidence_interval()

        # Adjust for regime
        regime_adjustments = {
            'normal': 1.0,
            'volatile': 1.5,
            'illiquid': 2.0,
            'congested': 1.3
        }

        regime_multiplier = regime_adjustments.get(regime, 1.0)
        adjusted_impact = predicted_impact * regime_multiplier

        return {
            'predicted_impact': predicted_impact,
            'adjusted_impact': adjusted_impact,
            'confidence_interval': confidence_interval,
            'regime_multiplier': regime_multiplier,
            'model_quality': self._assess_model_quality()
        }

    def _calculate_confidence_interval(self) -> Dict[str, float]:
        """Calculate confidence interval based on recent performance"""

        if len(self.recent_performance) < 10:
            return {'lower': 0.8, 'upper': 1.2}  # Default wide
interval

        recent_errors = self.recent_performance[-50:]  # Last 50
predictions

        mean_error = np.mean(recent_errors)
        std_error = np.std(recent_errors)

        # 95% confidence interval
        margin = 1.96 * std_error

        return {
            'lower': max(0.1, mean_error - margin),
            'upper': mean_error + margin
        }
```

```
    def _assess_model_quality(self) -> str:
        """Assess the quality of the current model"""

        if len(self.recent_performance) < 20:
            return "insufficient_data"

        recent_errors = self.recent_performance[-50:]
        mean_error = np.mean(recent_errors)

        if mean_error < 0.01:
            return "excellent"
        elif mean_error < 0.02:
            return "good"
        elif mean_error < 0.05:
            return "fair"
        else:
            return "poor"

    def update_model(self, trade_execution: TradeExecution,
                     market_state: MarketMicrostructureState,
                     actual_impact: float):
        """Update model with new observation"""

        # Store training example
        self.training_data.append({
            'features': [
                trade_execution.executed_size,
                market_state.liquidity_score,
                market_state.volatility_5m,
                market_state.order_flow_imbalance,
                market_state.spread
            ],
            'actual_impact': actual_impact,
            'timestamp': datetime.now()
        })

        # Calculate prediction error
        features = [
            trade_execution.executed_size,
            market_state.liquidity_score,
            market_state.volatility_5m,
```

```
            market_state.order_flow_imbalance,
            market_state.spread
        ]

        predicted_impact = self._predict_impact_with_params(features,
list(self.learned_parameters.values()))
        prediction_error = abs(predicted_impact - actual_impact) /
actual_impact

        self.recent_performance.append(prediction_error)

        # Limit history size
        if len(self.training_data) > 1000:
            self.training_data = self.training_data[-800:]
# Keep last 800 examples

        if len(self.recent_performance) > 100:
            self.recent_performance = self.recent_performance[-80:]

        # Periodic retraining
        if len(self.training_data) % 100 == 0:
            self._retrain_model()

    def _retrain_model(self):
        """Retrain model with accumulated data"""

        if len(self.training_data) < 50:
            return  # Need minimum data for retraining

        # Prepare data for retraining
        X = [example['features'] for example in
self.training_data[-500:]]  # Use last 500 examples
        y = [example['actual_impact'] for example in
self.training_data[-500:]]


# Simple retraining with exponential weighting (newer data more
important)
        weights = [0.9 ** (len(self.training_data) - i - 1) for i in
range(len(self.training_data)-500, len(self.training_data))]

        # Update parameters using gradient descent
```

```
        learning_rate = 0.01

        current_params = list(self.learned_parameters.values())

        for _ in range(50):  # 50 iterations
            total_gradient = [0] * len(current_params)

            for i, (features, actual_impact, weight) in
enumerate(zip(X, y, weights)):
                predicted = self._predict_impact_with_params(features,
current_params)
                error = predicted - actual_impact

                # Simple gradient approximation
                for j, param in enumerate(current_params):
                    param_plus = current_params.copy()
                    param_plus[j] += 0.001  # Small step
                    predicted_plus =
self._predict_impact_with_params(features, param_plus)
                    gradient = (predicted_plus - predicted) / 0.001
                    total_gradient[j] += error * gradient * weight

            # Update parameters
            for j in range(len(current_params)):
                current_params[j] -= learning_rate *
total_gradient[j] / len(X)

        # Update learned parameters
        param_names = ['base_impact', 'liquidity_coefficient',
'volatility_coefficient',
                       'size_exponent', 'time_decay',
'regime_threshold']

        for i, name in enumerate(param_names):
            self.learned_parameters[name] = current_params[i]

class RegimeDetectionModel:
    """Detects market regimes for adaptive modeling"""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
        self.regime_history = []
```

```
    def detect_current_regime(self, market_state:
MarketMicrostructureState) -> Dict[str, Any]:
        """Detect current market regime"""

        # Define regime characteristics
        regimes = {
            'normal': {
                'volatility_threshold': 0.03,
                'spread_threshold': 0.02,
                'liquidity_threshold': 0.8,
                'flow_imbalance_threshold': 0.1
            },
            'volatile': {
                'volatility_threshold': 0.05,
                'spread_threshold': 0.03,
                'liquidity_threshold': 0.6,
                'flow_imbalance_threshold': 0.2
            },
            'illiquid': {
                'volatility_threshold': 0.02,
                'spread_threshold': 0.05,
                'liquidity_threshold': 0.4,
                'flow_imbalance_threshold': 0.15
            },
            'congested': {
                'volatility_threshold': 0.03,
                'spread_threshold': 0.03,
                'liquidity_threshold': 0.7,
                'flow_imbalance_threshold': 0.1
            }
        }

        # Score each regime
        regime_scores = {}
        for regime_name, criteria in regimes.items():
            score = 0

            # Volatility score
            if market_state.volatility_5m >
criteria['volatility_threshold']:
                score += 1
```

```
            # Spread score
            if market_state.spread > criteria['spread_threshold']:
                score += 1

            # Liquidity score
            if market_state.liquidity_score <
criteria['liquidity_threshold']:
                score += 1

            # Flow imbalance score
            if abs(market_state.order_flow_imbalance) >
criteria['flow_imbalance_threshold']:
                score += 1

            regime_scores[regime_name] = score

        # Determine most likely regime
        most_likely_regime = max(regime_scores, key=regime_scores.get)
        confidence = regime_scores[most_likely_regime] / 4
# Normalize to 0-1

        # Store regime detection
        regime_detection = {
            'regime': most_likely_regime,
            'confidence': confidence,
            'scores': regime_scores,
            'characteristics': {
                'volatility': 'high' if market_state.volatility_5m >
0.05 else 'normal',
                'spread': 'wide' if market_state.spread > 0.03 else
'normal',
                'liquidity': 'low' if market_state.liquidity_score <
0.5 else 'normal',
                'flow_imbalance': 'high' if
abs(market_state.order_flow_imbalance) > 0.2 else 'normal'
            }
        }

        self.regime_history.append({
            'timestamp': market_state.timestamp,
            'regime': most_likely_regime,
```

```
            'confidence': confidence
        })

        return regime_detection

    def get_regime_statistics(self, window_hours: int = 24) ->
Dict[str, Any]:
        """Get regime statistics over time window"""

        cutoff_time = datetime.now() - timedelta(hours=window_hours)
        recent_regimes = [r for r in self.regime_history if
r['timestamp'] > cutoff_time]

        if not recent_regimes:
            return {}

        # Count regime occurrences
        regime_counts = {}
        for regime_record in recent_regimes:
            regime = regime_record['regime']
            regime_counts[regime] = regime_counts.get(regime, 0) + 1

        # Calculate percentages
        total_records = len(recent_regimes)
        regime_percentages = {regime: (count / total_records) * 100
                              for regime, count in regime_counts.items()}

        # Calculate average confidence
        avg_confidence = np.mean([r['confidence'] for r in
recent_regimes])

        # Regime transitions
        transitions = []
        for i in range(1, len(recent_regimes)):
            if recent_regimes[i]['regime'] != recent_regimes[i-1]
['regime']:
                transitions.append({
                    'from': recent_regimes[i-1]['regime'],
                    'to': recent_regimes[i]['regime'],
                    'timestamp': recent_regimes[i]['timestamp']
                })
```

```
        return {
            'regime_distribution': regime_percentages,
            'regime_counts': regime_counts,
            'average_confidence': avg_confidence,
            'total_transitions': len(transitions),
            'regime_transitions': transitions[-10:],  # Last 10
transitions
            'dominant_regime': max(regime_counts,
key=regime_counts.get) if regime_counts else 'unknown'
        }

# Example usage
adaptive_model = AdaptiveImpactModel({})
regime_detector = RegimeDetectionModel({})

# Example market microstructure state
microstructure_state = MarketMicrostructureState(
    timestamp=datetime.now(),
    best_bid=1999.0,
    best_ask=2001.0,
    bid_size=5000.0,
    ask_size=3000.0,
    spread=0.002,
    volatility_1m=0.02,
    volatility_5m=0.025,
    volatility_15m=0.03,
    order_flow_imbalance=0.1,
    trade_count_1m=45,
    average_trade_size=1200.0,
    liquidity_score=0.8
)

# Detect regime
regime = regime_detector.detect_current_regime(microstructure_state)
print(f"Current regime: {regime['regime']} (confidence:
{regime['confidence']:.2f})")

# Predict impact
impact_prediction = adaptive_model.predict_impact(1000,
microstructure_state, regime['regime'])
print(f"Predicted impact: {impact_prediction['predicted_impact']:.4f}")
print(f"Adjusted impact: {impact_prediction['adjusted_impact']:.4f}")
```

```
print(f"Model quality: {impact_prediction['model_quality']}")

# Regime statistics
stats = regime_detector.get_regime_statistics(6)  # Last 6 hours
print(f"Dominant regime in last 6h: {stats.get('dominant_regime',
'unknown')}")
```

4.3 Dynamic Liquidity Modeling

4.3.1 Liquidity Pool Dynamics

```
from typing import Dict, List, Any, Optional
from dataclasses import dataclass
from enum import Enum
import numpy as np
from datetime import datetime, timedelta

class LiquidityEventType(Enum):
    ADD = "add"
    REMOVE = "remove"
    SWAP = "swap"
    HARVEST = "harvest"

@dataclass
class LiquidityEvent:
    """Liquidity pool event"""
    timestamp: datetime
    event_type: LiquidityEventType
    token0_amount: float
    token1_amount: float
    lp_tokens_minted: float
    user_address: str

@dataclass
class LiquidityPoolState:
    """Current liquidity pool state"""
    pool_address: str
    token0: str
    token1: str
    reserve0: float
    reserve1: float
    total_liquidity: float
    lp_token_supply: float
    fee_tier: int
    impermanent_loss: float
    utilization_rate: float

class LiquidityDynamicsSimulator:
    """Simulates liquidity pool dynamics and their impact on market
impact"""

    def __init__(self, config: Dict[str, Any]):
        self.config = config
```

```
        self.liquidity_history = []
        self.pool_states = {}

    def simulate_liquidity_change(self, pool_state: LiquidityPoolState,
                                  liquidity_change: LiquidityEvent) ->
Dict[str, Any]:
        """Simulate impact of liquidity change on pool"""

        old_reserve0 = pool_state.reserve0
        old_reserve1 = pool_state.reserve1
        old_total_liquidity = pool_state.total_liquidity

        # Apply liquidity change
        if liquidity_change.event_type == LiquidityEventType.ADD:
            # Add liquidity
            pool_state.reserve0 += liquidity_change.token0_amount
            pool_state.reserve1 += liquidity_change.token1_amount
            pool_state.total_liquidity +=
liquidity_change.lp_tokens_minted

        elif liquidity_change.event_type == LiquidityEventType.REMOVE:
            # Remove liquidity
            pool_state.reserve0 -= liquidity_change.token0_amount
            pool_state.reserve1 -= liquidity_change.token1_amount
            pool_state.total_liquidity -=
liquidity_change.lp_tokens_minted

        # Calculate liquidity depth changes
        liquidity_depth_change = {
            'token0': pool_state.reserve0 - old_reserve0,
            'token1': pool_state.reserve1 - old_reserve1,
            'total': pool_state.total_liquidity - old_total_liquidity
        }

        # Update utilization rate
        pool_state.utilization_rate =
self._calculate_utilization_rate(pool_state)

        # Calculate price impact of the liquidity change itself
        price_change_0_1 = (pool_state.reserve1 /
pool_state.reserve0) / (old_reserve1 / old_reserve0) - 1
        price_change_1_0 = (pool_state.reserve0 /
```

```
pool_state.reserve1) / (old_reserve0 / old_reserve1) - 1

        # Store event
        self.liquidity_history.append(liquidity_change)

        return {
            'liquidity_depth_change': liquidity_depth_change,
            'price_change_0_1': price_change_0_1,
            'price_change_1_0': price_change_1_0,
            'new_reserves': {
                'token0': pool_state.reserve0,
                'token1': pool_state.reserve1
            },
            'new_utilization_rate': pool_state.utilization_rate,
            'event_impact_score':
self._calculate_event_impact_score(liquidity_change, pool_state)
        }

    def _calculate_utilization_rate(self, pool_state:
LiquidityPoolState) -> float:
        """Calculate pool utilization rate"""

        # Utilization is based on recent volume relative to liquidity
        # This is a simplified model - in practice would use actual
volume data

        if pool_state.total_liquidity == 0:
            return 0

        # Estimate 24h volume as percentage of liquidity
        estimated_volume = pool_state.total_liquidity * 0.1  # 10%
daily turnover

        utilization = min(1.0, estimated_volume /
pool_state.total_liquidity)
        return utilization

    def _calculate_event_impact_score(self, event: LiquidityEvent,
                                      pool_state: LiquidityPoolState) ->
float:
        """Calculate impact score of liquidity event"""
```

```
        # Base impact on total liquidity
        liquidity_change_percent = (
            (event.lp_tokens_minted if event.event_type ==
LiquidityEventType.ADD else -event.lp_tokens_minted)
            / pool_state.total_liquidity
        )

        # Adjust for event type
        if event.event_type == LiquidityEventType.SWAP:
            # Swaps have minimal direct impact on total liquidity
            event_impact = 0.01 * abs(liquidity_change_percent)
        elif event.event_type == LiquidityEventType.ADD:
            # Adding liquidity generally positive for depth
            event_impact = -0.5 * liquidity_change_percent  # Negative
impact means less impact
        elif event.event_type == LiquidityEventType.REMOVE:
            # Removing liquidity reduces depth
            event_impact = liquidity_change_percent
        else:
            event_impact = 0

        return event_impact

    def predict_liquidity_response(self, expected_trade_size: float,
                                   pool_state: LiquidityPoolState,
                                   market_conditions: Dict[str, Any]) ->
Dict[str, Any]:
        """Predict how liquidity will respond to expected large
trade"""

        # Expected trade size relative to liquidity
        size_ratio = expected_trade_size / pool_state.total_liquidity

        # Base liquidity response
        if size_ratio > 0.2:  # Large trade expected
            # Liquidity providers will add liquidity
            liquidity_response = 1.3 + (size_ratio - 0.2) * 0.5
        elif size_ratio > 0.1:
            liquidity_response = 1.1
        else:
            liquidity_response = 1.0
```

```
        # Adjust for market conditions
        volatility = market_conditions.get('volatility', 0.03)
        if volatility > 0.05:
            liquidity_response *= 0.8  # High volatility reduces
liquidity provision

        congestion = market_conditions.get('network_congestion', 0.5)
        if congestion > 0.8:
            liquidity_response *= 0.7  # Network congestion reduces
activity

        # Predict timing of liquidity response
        if size_ratio > 0.2:
            response_time_minutes = np.random.randint(2, 10)  # 2-10
minutes
        elif size_ratio > 0.1:
            response_time_minutes = np.random.randint(5, 20)  # 5-20
minutes
        else:
            response_time_minutes = np.random.randint(10, 60)  # 10-60
minutes

        # Calculate expected new liquidity level
        expected_new_liquidity = pool_state.total_liquidity *
liquidity_response

        # Estimate impact reduction
        impact_reduction = (expected_new_liquidity -
pool_state.total_liquidity) / pool_state.total_liquidity * 0.5

        return {
            'liquidity_response_factor': liquidity_response,
            'expected_new_liquidity': expected_new_liquidity,
            'estimated_response_time_minutes': response_time_minutes,
            'impact_reduction_estimate': impact_reduction,
            'confidence': min(1.0, 0.5 + liquidity_response / 2)
        }

    def model_liquidity_concentration(self, pool_state:
LiquidityPoolState) -> Dict[str, Any]:
        """Model concentration of liquidity in the pool"""
```

```
        # Simulate liquidity concentration across price ranges
        # In concentrated liquidity pools, this is very important

        current_price_ratio = pool_state.reserve1 / pool_state.reserve0
        price_ranges = []

        # Define price ranges around current price
        price_multipliers = [0.8, 0.9, 0.95, 0.98, 0.99, 1.0, 1.01,
1.02, 1.05, 1.1, 1.2]

        total_liquidity = pool_state.total_liquidity

        for multiplier in price_multipliers:
            # Simulate price-specific liquidity
            # At current price (multiplier = 1.0), assume highest
concentration

            if multiplier == 1.0:
                # At-the-money liquidity
                concentration_factor = np.random.uniform(0.3, 0.5)  #
30-50% at current price
            elif 0.95 <= multiplier <= 1.05:
                # Near-the-money liquidity
                concentration_factor = np.random.uniform(0.15, 0.25)
# 15-25% near current price
            elif 0.8 <= multiplier <= 1.2:
                # Medium distance liquidity
                concentration_factor = np.random.uniform(0.05, 0.15)
# 5-15% medium distance
            else:
                # Far-from-the-money liquidity
                concentration_factor = np.random.uniform(0.01, 0.05)
# 1-5% far from current price

            liquidity_at_range = total_liquidity * concentration_factor
            price = current_price_ratio * multiplier

            price_ranges.append({
                'price_multiplier': multiplier,
                'price': price,
                'liquidity': liquidity_at_range,
                'percentage_of_total': concentration_factor * 100
```

```
            })

        # Calculate depth at different price levels
        depth_analysis = {
            'at_money_depth': price_ranges[5]['liquidity'],  # At
current price
            'near_money_depth': sum(r['liquidity'] for r in
price_ranges[3:8]),
            'total_analyzed_depth': sum(r['liquidity'] for r in
price_ranges),
            'concentration_score':
self._calculate_concentration_score(price_ranges)
        }

        return {
            'price_ranges': price_ranges,
            'depth_analysis': depth_analysis,
            'liquidity_gini_coefficient':
self._calculate_gini_coefficient([r['percentage_of_total'] for r in
price_ranges])
        }

    def _calculate_concentration_score(self, price_ranges:
List[Dict[str, Any]]) -> float:
        """Calculate liquidity concentration score"""

        # Herfindahl-Hirschman Index for concentration
        total_share = sum(r['percentage_of_total'] for r in
price_ranges) / 100
        if total_share == 0:
            return 0

        hhi = sum((r['percentage_of_total'] / 100) ** 2 for r in
price_ranges)

        # Normalize to 0-1 scale (0 = perfectly distributed, 1 =
perfectly concentrated)
        normalized_hhi = hhi / total_share if total_share > 0 else 0

        return min(1.0, normalized_hhi)

    def _calculate_gini_coefficient(self, values: List[float]) ->
```

```
float:
        """Calculate Gini coefficient for inequality measurement"""

        if not values or sum(values) == 0:
            return 0

        # Sort values
        sorted_values = sorted(values)
        n = len(sorted_values)

        # Calculate Gini coefficient
        cumsum = np.cumsum(sorted_values)
        gini = (2 * sum((i + 1) * sorted_values[i] for i in
range(n))) / (n * cumsum[-1]) - (n + 1) / n

        return gini

    def simulate_flash_loan_impact(self, loan_amount: float,
pool_state: LiquidityPoolState) -> Dict[str, Any]:
        """Simulate impact of flash loan on pool liquidity"""

        # Flash loans temporarily borrow and return within same
transaction
        # They can significantly impact pool reserves during execution

        # Calculate impact as percentage of total liquidity
        impact_ratio = loan_amount / pool_state.total_liquidity

        if impact_ratio > 0.5:
            impact_level = "high"
        elif impact_ratio > 0.2:
            impact_level = "medium"
        else:
            impact_level = "low"

        # Simulate temporary reserve changes
        borrow_ratio = min(impact_ratio, 0.4)  # Cannot borrow more
than 40% of liquidity

        temp_reserve0 = pool_state.reserve0 * (1 - borrow_ratio)
        temp_reserve1 = pool_state.reserve1 * (1 - borrow_ratio)
```

```
        # Calculate temporary price impact
        price_before = pool_state.reserve1 / pool_state.reserve0
        price_after = temp_reserve1 / temp_reserve0
        price_impact = abs(price_after - price_before) / price_before

        # Flash loan repayment effect

# In practice, flash loans are repaid, so net effect depends on trading
during the loan

        return {
            'impact_level': impact_level,
            'impact_ratio': impact_ratio,
            'temporary_reserve_reduction': borrow_ratio,
            'temporary_price_impact': price_impact,
            'estimated_duration_blocks': 1,  # Flash loans are same-
block
            'risk_assessment':
self._assess_flash_loan_risk(impact_level, price_impact)
        }

    def _assess_flash_loan_risk(self, impact_level: str, price_impact:
float) -> str:
        """Assess risk level of flash loan impact"""

        if impact_level == "high" and price_impact > 0.1:
            return "very_high"
        elif impact_level == "high" or price_impact > 0.05:
            return "high"
        elif impact_level == "medium" or price_impact > 0.02:
            return "medium"
        else:
            return "low"

# Example usage
liquidity_sim = LiquidityDynamicsSimulator({})

# Create liquidity pool state
pool_state = LiquidityPoolState(
    pool_address="0xB4e16d0168e52d35CaCD2c6185b44281Ec28C9Dc",
    token0="WETH",
    token1="USDC",
```

```
    reserve0=1000.0,
    reserve1=2_000_000.0,
    total_liquidity=2_000_000.0,
    lp_token_supply=2_000_000.0,
    fee_tier=3000,
    impermanent_loss=0.02,
    utilization_rate=0.15
)

# Simulate adding liquidity
add_liquidity = LiquidityEvent(
    timestamp=datetime.now(),
    event_type=LiquidityEventType.ADD,
    token0_amount=100.0,
    token1_amount=200_000.0,
    lp_tokens_minted=200_000.0,
    user_address="0x1234567890123456789012345678901234567890"
)

result = liquidity_sim.simulate_liquidity_change(pool_state,
add_liquidity)
print(f"Liquidity added: {result['liquidity_depth_change']['total']:.
0f}")
print(f"Price change: {result['price_change_0_1']:.4f}")

# Predict liquidity response to large trade
response = liquidity_sim.predict_liquidity_response(5000, pool_state,
{'volatility': 0.03, 'network_congestion': 0.6})
print(f"Expected liquidity response:
{response['liquidity_response_factor']:.2f}x")
print(f"Impact reduction estimate:
{response['impact_reduction_estimate']:.3f}")

# Model liquidity concentration
concentration = liquidity_sim.model_liquidity_concentration(pool_state)
print(f"Liquidity concentration score: {concentration['depth_analysis']
['concentration_score']:.3f}")
print(f"Depth at current price: {concentration['depth_analysis']
['at_money_depth']:.0f}")
```

4.4 Practical Exercise: Complete Market Impact Model

```
def build_market_impact_modeling_system():
    """Complete exercise to build a market impact modeling system"""

    print("Building Market Impact Modeling System...")

    # Initialize components
    config = {'model_config': 'advanced'}

    # 1. Basic Price Impact Model
    print("\n1. Testing Basic Price Impact Model...")
    impact_model = PriceImpactModel(config)

    market_state = MarketState(
        timestamp=datetime.now(),
        price=2000.0,
        volume_24h=100000.0,
        liquidity=10000.0,
        spread=0.01,
        volatility=0.03,
        block_utilization=0.6,
        mempool_size=5000,
        gas_price=25_000_000_000
    )

    # Test different order sizes
    for size in [100, 500, 1000, 2000]:
        result = impact_model.calculate_price_impact(size,
market_state, MarketRegime.NORMAL)
        print(f"   Order {size}: {result['slippage_percent']:.2f}%
slippage, ${result['impact_cost']:.2f} impact cost")

    # 2. Order Flow Analysis
    print("\n2. Testing Order Flow Analysis...")
    flow_analyzer = OrderFlowAnalyzer({})

    # Simulate order flow
    orders = [
        Order("1", "buy", 500, None, "market", datetime.now()),
        Order("2", "sell", 300, None, "market", datetime.now()),
        Order("3", "buy", 1000, None, "market", datetime.now()),
        Order("4", "sell", 800, None, "market", datetime.now())
    ]
```

```
    executions = flow_analyzer.process_order_flow(orders, market_state)

    print(f"   Processed {len(executions)} orders")
    for execution in executions:
        print(f"   {execution.order_id}: {execution.executed_size} @
{execution.execution_price:.2f} "
              f"({execution.slippage:.3%} slippage)")

    # Flow analysis
    flow_patterns = flow_analyzer.analyze_flow_patterns()
    print(f"   Flow imbalance: {flow_patterns.get('flow_imbalance',
0):.3f}")
    print(f"   Flow acceleration:
{flow_patterns.get('flow_acceleration', 0):.3f}")

    # 3. Adaptive Impact Model
    print("\n3. Testing Adaptive Impact Model...")
    adaptive_model = AdaptiveImpactModel({})

    # Generate synthetic training data
    training_trades = []
    training_states = []

    for i in range(100):
        trade_size = np.random.uniform(100, 5000)
        market_impact = np.random.uniform(0.001, 0.05)  # 0.1% to 5%
impact

        # Create synthetic trade execution
        trade_execution = TradeExecution(
            order_id=f"trade_{i}",
            executed_size=trade_size,
            execution_price=2000.0 + (trade_size / 1000) *
market_impact * 2000.0,
            slippage=market_impact,
            market_impact=market_impact,
            liquidity_consumed=trade_size,
            timestamp=datetime.now() - timedelta(minutes=i)
        )

        # Create synthetic market state
```

```
        market_state_synthetic = MarketMicrostructureState(
            timestamp=datetime.now() - timedelta(minutes=i),
            best_bid=1999.0,
            best_ask=2001.0,
            bid_size=5000.0,
            ask_size=5000.0,
            spread=0.002,
            volatility_1m=0.02 + np.random.normal(0, 0.01),
            volatility_5m=0.025 + np.random.normal(0, 0.01),
            volatility_15m=0.03 + np.random.normal(0, 0.01),
            order_flow_imbalance=np.random.normal(0, 0.1),
            trade_count_1m=int(np.random.uniform(20, 100)),
            average_trade_size=trade_size,
            liquidity_score=0.8 + np.random.normal(0, 0.1)
        )

        training_trades.append(trade_execution)
        training_states.append(market_state_synthetic)

    # Fit the model
    fit_result = adaptive_model.fit_model(training_trades,
training_states)
    print(f"   Model fitting: {fit_result['optimization_success']}")
    print(f"   Training error: {fit_result['training_error']:.6f}")

    # Test prediction
    microstructure_state = MarketMicrostructureState(
        timestamp=datetime.now(),
        best_bid=1999.0,
        best_ask=2001.0,
        bid_size=5000.0,
        ask_size=5000.0,
        spread=0.002,
        volatility_1m=0.02,
        volatility_5m=0.025,
        volatility_15m=0.03,
        order_flow_imbalance=0.1,
        trade_count_1m=50,
        average_trade_size=1200.0,
        liquidity_score=0.8
    )
```

```
    prediction = adaptive_model.predict_impact(1000,
microstructure_state, 'normal')
    print(f"   Predicted impact: {prediction['predicted_impact']:.4f}")
    print(f"   Adjusted impact: {prediction['adjusted_impact']:.4f}")
    print(f"   Model quality: {prediction['model_quality']}")

    # 4. Regime Detection
    print("\n4. Testing Regime Detection...")
    regime_detector = RegimeDetectionModel({})

    # Test different market conditions
    test_states = [
        # Normal market
        MarketMicrostructureState(
            timestamp=datetime.now(), best_bid=1999.0, best_ask=2001.0,
            bid_size=5000.0, ask_size=5000.0, spread=0.002,
            volatility_1m=0.02, volatility_5m=0.025,
volatility_15m=0.03,
            order_flow_imbalance=0.05, trade_count_1m=50,
average_trade_size=1000.0,
            liquidity_score=0.8
        ),
        # Volatile market
        MarketMicrostructureState(
            timestamp=datetime.now(), best_bid=1990.0, best_ask=2010.0,
            bid_size=2000.0, ask_size=2000.0, spread=0.01,
            volatility_1m=0.08, volatility_5m=0.06,
volatility_15m=0.05,
            order_flow_imbalance=0.3, trade_count_1m=150,
average_trade_size=500.0,
            liquidity_score=0.4
        ),
        # Illiquid market
        MarketMicrostructureState(
            timestamp=datetime.now(), best_bid=1995.0, best_ask=2005.0,
            bid_size=500.0, ask_size=300.0, spread=0.005,
            volatility_1m=0.025, volatility_5m=0.03,
volatility_15m=0.035,
            order_flow_imbalance=0.2, trade_count_1m=15,
average_trade_size=3000.0,
            liquidity_score=0.3
        )
```

```
    ]

    regimes_detected = []
    for i, state in enumerate(test_states):
        regime = regime_detector.detect_current_regime(state)
        regimes_detected.append(regime['regime'])
        print(f"   Market {i+1}: {regime['regime']} (confidence:
{regime['confidence']:.2f})")

    # 5. Liquidity Dynamics
    print("\n5. Testing Liquidity Dynamics...")
    liquidity_sim = LiquidityDynamicsSimulator({})

    pool_state = LiquidityPoolState(
        pool_address="0x1234",
        token0="WETH",
        token1="USDC",
        reserve0=1000.0,
        reserve1=2_000_000.0,
        total_liquidity=2_000_000.0,
        lp_token_supply=2_000_000.0,
        fee_tier=3000,
        impermanent_loss=0.02,
        utilization_rate=0.15
    )

    # Test liquidity response prediction
    liquidity_response = liquidity_sim.predict_liquidity_response(
        5000, pool_state, {'volatility': 0.03, 'network_congestion':
0.6}
    )
    print(f"   Liquidity response factor:
{liquidity_response['liquidity_response_factor']:.2f}")
    print(f"   Expected new liquidity:
{liquidity_response['expected_new_liquidity']:.0f}")
    print(f"   Response time:
{liquidity_response['estimated_response_time_minutes']} minutes")

    # Test liquidity concentration
    concentration =
liquidity_sim.model_liquidity_concentration(pool_state)
    print(f"   Concentration score: {concentration['depth_analysis']
```

```
['concentration_score']:.3f}")
    print(f"   At-money depth: {concentration['depth_analysis']
['at_money_depth']:.0f}")

    # Test flash loan impact
    flash_loan_impact = liquidity_sim.simulate_flash_loan_impact(10000,
pool_state)
    print(f"   Flash loan impact level:
{flash_loan_impact['impact_level']}")
    print(f"   Price impact:
{flash_loan_impact['temporary_price_impact']:.3%}")
    print(f"   Risk assessment:
{flash_loan_impact['risk_assessment']}")

    # 6. Comprehensive Integration Test
    print("\n6. Testing Complete Integration...")

    # Simulate a complex trading scenario
    complex_scenario = {
        'initial_liquidity': 10000.0,
        'expected_trade_size': 2000.0,
        'market_volatility': 0.04,
        'network_congestion': 0.7,
        'flow_imbalance': 0.15
    }

    # Calculate combined impact
    base_impact = impact_model.calculate_price_impact(
        complex_scenario['expected_trade_size'], market_state
    )

    # Adjust for liquidity dynamics
    adjusted_impact = base_impact['total_impact']

    # Predict liquidity response
    response_pred = liquidity_sim.predict_liquidity_response(
        complex_scenario['expected_trade_size'], pool_state,
        {'volatility': complex_scenario['market_volatility'],
         'network_congestion': complex_scenario['network_congestion']}
    )

    # Apply liquidity response adjustment
```

```
    final_impact = adjusted_impact * (1 -
response_pred['impact_reduction_estimate'])

    print(f"   Base impact: {base_impact['slippage_percent']:.2f}%")
    print(f"   Liquidity response adjustment: -
{response_pred['impact_reduction_estimate']:.2%}")
    print(f"   Final expected impact: {(final_impact /
market_state.price * 100):.2f}%")
    print(f"   Expected execution cost: $
{complex_scenario['expected_trade_size'] * market_state.price *
final_impact / market_state.price:.2f}")

    print("\n✅ Market Impact Modeling System Test Complete!")

    # Summary
    print("\n" + "="*50)
    print("MARKET IMPACT MODELING SUMMARY")
    print("="*50)
    print(f"Price Impact Models: ✅ Quadratic, Adaptive, Flow-based")
    print(f"Regime Detection: ✅ {len(set(regimes_detected))} regimes
detected")
    print(f"Liquidity Dynamics: ✅ Response prediction and
concentration analysis")
    print(f"Flash Loan Modeling: ✅ Impact simulation and risk
assessment")
    print(f"Model Quality: {prediction['model_quality']}")
    print(f"Integration Test: ✅ Complex scenario modeling successful")

    return {
        'impact_model': impact_model,
        'flow_analyzer': flow_analyzer,
        'adaptive_model': adaptive_model,
        'regime_detector': regime_detector,
        'liquidity_sim': liquidity_sim,
        'test_results': {
            'base_impact_model': base_impact['slippage_percent'],
            'flow_patterns': flow_patterns,
            'model_quality': prediction['model_quality'],
            'regimes_detected': list(set(regimes_detected)),
            'liquidity_response':
response_pred['liquidity_response_factor'],
            'final_impact': (final_impact / market_state.price * 100)
```

```
        }
    }

# Run the exercise
if __name__ == "__main__":
    modeling_results = build_market_impact_modeling_system()
    print("\nMarket impact modeling system ready for sophisticated MEV
strategy analysis!")
```

Module Summary

In this module, you have built a comprehensive market impact modeling system that includes:

- **Price Impact Models**: Quadratic, adaptive, and order flow-based impact calculations
- **Market Microstructure**: Detailed analysis of bid-ask spreads, volatility, and liquidity
- **Regime Detection**: Automatic identification of market conditions (normal, volatile, illiquid, congested)
- **Adaptive Learning**: Models that improve from historical data and market feedback
- **Liquidity Dynamics**: Pool-level liquidity changes, concentration analysis, and flash loan impacts
- **Flow Analysis**: Order flow pressure and its impact on market prices

Key Takeaways

1. **Multi-Factor Impact**: Price impact depends on size, liquidity, volatility, and market regime
2. **Adaptive Models**: Models should learn and adapt to changing market conditions
3. **Regime Awareness**: Different market regimes require different impact models
4. **Liquidity Concentration**: Concentrated liquidity significantly affects execution quality
5. **Order Flow Effects**: Recent trade flow provides valuable context for impact prediction
6. **Dynamic Adjustment**: Impact models should adjust for expected liquidity responses

Next Steps

1. **Parameter Calibration**: Calibrate models using real trading data
2. **Multi-Asset Extension**: Extend to cross-asset and cross-chain scenarios
3. **Real-Time Integration**: Connect with live market data feeds

4. **Portfolio-Level Impact**: Extend to portfolio-level impact analysis
5. **Stress Testing**: Build stress testing scenarios for extreme market conditions

This market impact modeling system provides the sophisticated analysis framework needed for optimal MEV strategy execution and risk management.