Market Impact and Order Book Events Analysis

This notebook reproduces some of the results from the paper:

Z. Eisler, J.-P. Bouchaud, and J. Kockelkoren. "The price impact of order book events: market orders, limit orders and cancellations". In: Quantitative Finance 12.9 (2012), pp. 1395–1419.

We'll analyze the price impact of different order book events and implement the following analyses:

1. Sign and side autocorrelation functions
2. Empirical response functions for the 6 types of events
3. Signed event-event correlations
4. Plotting with signed log-scale
5. Theoretical responses in the constant impact model compared to empirical ones

In [1]:
```
# Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
#from datetime import datetime, timedelta
import warnings
warnings.filterwarnings('ignore')

# Set plotting style
plt.style.use('ggplot')
plt.rcParams['figure.figsize'] = (10, 6)
plt.rcParams['font.size'] = 12
```

Data Loading and Preprocessing

First, we'll load and preprocess the financial data for analysis. We need to identify and categorize the different types of order book events.

In [2]:
```
file_list_SG = ['Data/SG/SG_20170117.csv.gz','Data/SG/SG_20170118.csv.gz','Data/SG/SG_20170119.csv.gz','
             'Data/SG/SG_20170125.csv.gz','Data/SG/SG_20170126.csv.gz','Data/SG/SG_20170127.csv.gz','Dat

file_list_BNPP = ['Data/BNPP/BNPP_20170117.csv.gz','Data/BNPP/BNPP_20170118.csv.gz', 'Data/BNPP/BNPP_201
                'Data/BNPP/BNPP_20170125.csv.gz', 'Data/BNPP/BNPP_20170126.csv.gz', 'Data/BNPP/BNPP_2017

def import_data(file_list: list) -> dict:
    """
    Function to import data from a list of gzipped CSV files.
    The files are stored in a dictionary with the file name (without extension) as the key. Some columns
    """
    data_dict = {}
    for file in file_list:
        df = pd.read_csv(file, compression='gzip').drop(columns=['Unnamed: 0'])
        df['ets'] = pd.to_datetime(df['ets'], format='%Y%m%d:%H:%M:%S.%f')
        # Compute mid-price
        df['mid_price'] = (df['bp0'] + df['ap0']) / 2
        # Compute Weighted Mid-Price
        df['weighted_mid_price'] = (df['bp0'] * df['aq0'] + df['ap0'] * df['bq0']) / (df['aq0'] + df['bq
        # Calculate log mid-price
        df['log_mid_price'] = np.log(df['mid_price'])
        # column ets in index
        df.set_index('ets', inplace=True)
        data_dict[file[-15:-7]] = df
    return data_dict

dict_BNPP = import_data(file_list_BNPP)
dict_SG = import_data(file_list_SG)
```

In [3]:
```
df=dict_BNPP['20170118']
df[df['etype']=='T'].tail(20)
#dict_SG['20170117'].head(5)
```

Out[3]:

ets	etype	eprice	eqty	eside	bp0	bq0	ap0	aq0	mid_price	weighted_mid_price	log_mid_p
2017-01-18 17:29:55.000533	T	59580	15	B	59580	662	59600	1442	59590.0	59586.292776	10.995
2017-01-18 17:29:55.000812	T	59600	3	S	59580	662	59600	1439	59590.0	59586.301761	10.995
2017-01-18 17:29:56.001049	T	59580	27	B	59580	688	59600	617	59590.0	59590.544061	10.995
2017-01-18 17:29:56.001056	T	59580	100	B	59580	588	59600	617	59590.0	59589.759336	10.995
2017-01-18 17:29:56.001061	T	59580	92	B	59580	496	59600	617	59590.0	59588.912848	10.995
2017-01-18 17:29:56.001596	T	59580	103	B	59580	393	59600	617	59590.0	59587.782178	10.995
2017-01-18 17:29:56.001782	T	59580	128	B	59580	265	59600	855	59590.0	59584.732143	10.995
2017-01-18 17:29:56.001788	T	59580	265	B	59570	5188	59600	855	59585.0	59595.755419	10.995
2017-01-18 17:29:56.010605	T	59580	187	B	59570	4736	59590	437	59580.0	59588.310458	10.995
2017-01-18 17:29:57.645067	T	59590	172	S	59570	4008	59590	835	59580.0	59586.551724	10.995
2017-01-18 17:29:57.645072	T	59590	78	S	59570	4008	59590	757	59580.0	59586.822665	10.995
2017-01-18 17:29:58.206347	T	59590	368	S	59570	4102	59600	683	59585.0	59595.717868	10.995
2017-01-18 17:29:58.206354	T	59600	90	S	59570	4102	59600	593	59585.0	59596.210863	10.995
2017-01-18 17:29:58.206358	T	59600	90	S	59570	4102	59600	503	59585.0	59596.723127	10.995
2017-01-18 17:29:58.206363	T	59600	265	S	59570	4102	59600	238	59585.0	59598.354839	10.995
2017-01-18 17:29:58.206367	T	59600	238	S	59570	4102	59610	1070	59590.0	59601.724671	10.995
2017-01-18 17:29:58.206370	T	59610	260	S	59570	4102	59610	810	59590.0	59603.403909	10.995
2017-01-18 17:29:58.206373	T	59610	212	S	59570	4102	59610	598	59590.0	59604.910638	10.995
2017-01-18 17:29:58.206377	T	59610	125	S	59570	4102	59610	473	59590.0	59605.864481	10.995
2017-01-18 17:29:58.486236	T	59600	265	B	59590	265	59620	917	59605.0	59596.725888	10.995

Event Classification

We need to classify the order book events into the following categories:

1. Market orders (MO): Buy and Sell
2. Limit orders (LO): Buy and Sell
3. Cancellations (CA): Buy and Sell

We'll use the event type ('etype') and side ('eside') columns to categorize these events.

In [4]:
```python
def classify_events(df):
    """Classify order book events into the 6 categories."""
    # Create a copy to avoid modifying the original dataframe
    df = df.copy()

    # Initialize event type columns
    df['event_type'] = 'Unknown'
    df['event_sign'] = 0

    # Market Orders (MO)
    mo_mask = df['etype'] == 'T'  # Trade events
    df.loc[mo_mask & (df['eside'] == 'S'), 'event_type'] = 'MO_buy'  # Buy market orders
    df.loc[mo_mask & (df['eside'] == 'B'), 'event_type'] = 'MO_sell'  # Sell market orders

    # Limit Orders (LO)
    lo_mask = df['etype'] == 'A'  # Add events
    df.loc[lo_mask & (df['eside'] == 'B'), 'event_type'] = 'LO_buy'  # Buy limit orders
    df.loc[lo_mask & (df['eside'] == 'S'), 'event_type'] = 'LO_sell'  # Sell limit orders

    # Cancellations (CA)
    ca_mask = df['etype'] == 'D'  # Delete events
    df.loc[ca_mask & (df['eside'] == 'B'), 'event_type'] = 'CA_buy'  # Buy cancellations
    df.loc[ca_mask & (df['eside'] == 'S'), 'event_type'] = 'CA_sell'  # Sell cancellations

    # Also consider 'C' (Change) events as cancellations
    c_mask = df['etype'] == 'C'  # Change events
    df.loc[c_mask & (df['eside'] == 'B'), 'event_type'] = 'CA_buy'  # Buy cancellations
    df.loc[c_mask & (df['eside'] == 'S'), 'event_type'] = 'CA_sell'  # Sell cancellations

    # Assign signs to events (+1 for buy, -1 for sell)
    df.loc[df['event_type'].str.contains('buy'), 'event_sign'] = 1
    df.loc[df['event_type'].str.contains('sell'), 'event_sign'] = -1
    # It is the contrary for cancellations
    df.loc[df['event_type'].str.contains('CA'), 'event_sign'] = df.loc[df['event_type'].str.contains('CA

    # Create a simplified event type without buy/sell distinction
    df['event_category'] = df['event_type'].str.split('_').str[0]

    # Create a side indicator (1 for buy, -1 for sell)
    df['side'] = 0
    df.loc[df['eside'] == 'B', 'side'] = 1
    df.loc[df['eside'] == 'S', 'side'] = -1

    return df

def classify_all_the_days(dict_data):
    """
    Function to classify events for all days in the dictionary.
    Parameters:
    dict_data : dict
        Dictionary containing DataFrames for each day.
    Returns:
    dict
        Dictionary with classified DataFrames for each day.
    """
    classified_dict = {}
    for key, df in dict_data.items():
        classified_dict[key] = classify_events(df)
    return classified_dict
```

In [5]:
```python
# Classify events for all days in the dictionaries
classified_BNPP = classify_all_the_days(dict_BNPP)
classified_SG = classify_all_the_days(dict_SG)
# Check the classification for a specific day
classified_BNPP['20170118'].head(10)
```

Out[5]:

ets	etype	eprice	eqty	eside	bp0	bq0	ap0	aq0	mid_price	weighted_mid_price	log_mid_p
2017-01-18 09:00:14.085766	A	60600	39	S	60570	2051	60600	39	60585.0	60599.440191	11.011
2017-01-18 09:01:00.012238	A	60150	19667	B	60610	213	60630	440	60620.0	60616.523737	11.012
2017-01-18 09:01:00.012248	A	61080	19667	S	60610	213	60630	440	60620.0	60616.523737	11.012
2017-01-18 09:01:00.109735	A	60610	213	S	60610	213	60630	440	60620.0	60616.523737	11.012
2017-01-18 09:01:00.109738	T	60610	96	B	60610	117	60630	440	60620.0	60614.201077	11.012
2017-01-18 09:01:00.109745	T	60610	117	B	60580	131	60630	440	60605.0	60591.471103	11.012
2017-01-18 09:01:00.109950	A	60620	124	S	60580	131	60620	124	60600.0	60600.549020	11.012
2017-01-18 09:01:00.109961	C	60470	1650	B	60580	131	60620	124	60600.0	60600.549020	11.012
2017-01-18 09:01:00.110132	A	60650	268	S	60580	131	60620	124	60600.0	60600.549020	11.012
2017-01-18 09:01:00.110629	C	60620	124	S	60580	131	60630	440	60605.0	60591.471103	11.012

1. Sign and Side Autocorrelation Functions

We'll calculate and plot the sign autocorrelation function $\langle \epsilon_t, \epsilon_{t+\ell} \rangle$ and side autocorrelation function $\langle s_t, s_{t+\ell} \rangle$, where ϵ_t is the sign of the event at time t and s_t is the side (buy/sell) of the event at time t. We'll use logarithmically spaced lags for efficiency.

Autocorrelation function

In [6]:
```python
def autocorrelation(df: pd.DataFrame, lags, column: str = 'event_sign'):
    """
    Function to compute autocorrelation of the column.
    Parameters:
    df : DataFrame
        The input DataFrame containing event data.
    lags : int or list
        The number of lags for autocorrelation.
    Returns:
    float or list
        The autocorrelation value or values.
    """
    if isinstance(lags, list):
        return [float(df[column].autocorr(lag=lag)) for lag in lags]
    elif isinstance(lags, int):
        return float(df[column].autocorr(lag=lags))
    else:
        raise ValueError("lags should be an int or a list of ints")
```

Define Logarithmically Spaced Lags

To efficiently analyze long-range correlations, we'll use logarithmically spaced lags instead of calculating all lags sequentially. This approach allows us to capture behavior across multiple time scales while significantly reducing computation time.

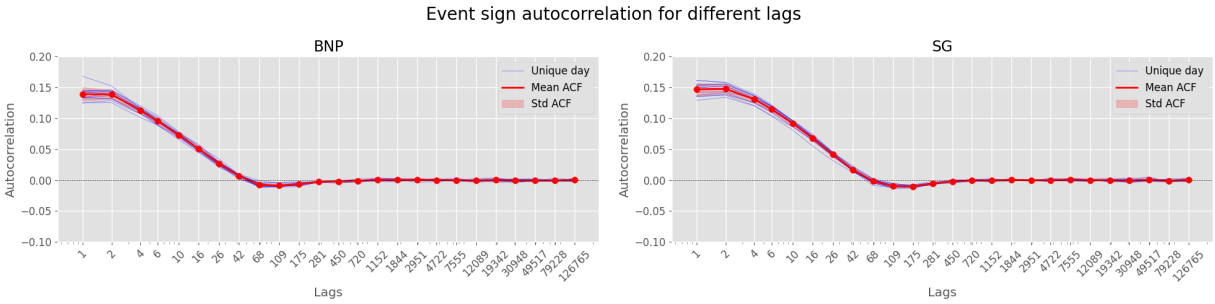
In [7]:
```python
Lags = [ int(1.6**i) for i in range(1, 26)]
```

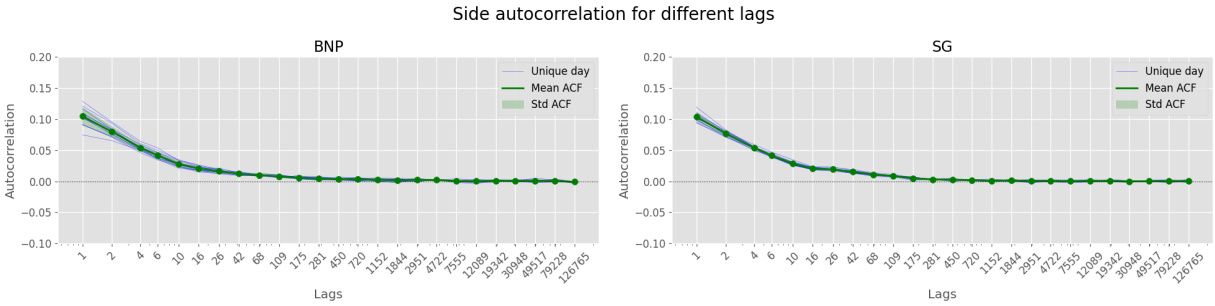
Plotting the autocorrelation functions

In [8]:
```python
def plot_autocorrelation_all_days(dict_data: dict, lags: list, column: str = 'event_sign'):
    """
    Function to plot autocorrelation for all days in the dictionary.
    Parameters:
    dict_data : dict
        Dictionary containing DataFrames for each day.
    lags : list
        List of lags for autocorrelation.
    column : str
        The column name for which to compute autocorrelation.
    """
    fig, ax = plt.subplots()
    list_autocorr = []
    for key, df in dict_data.items():
        acf_values = autocorrelation(df, lags, column)
        list_autocorr.append(acf_values)
        ax.plot(lags, acf_values, color='blue', label="Unique day", alpha=0.5, lw=0.5)

    # Compute mean and std of autocorrelation values
    mean_acf = np.mean(list_autocorr, axis=0)
    std_acf = np.std(list_autocorr, axis=0)
    ax.plot(lags, mean_acf, color='red', label='Mean ACF', lw=2)
    ax.fill_between(lags, mean_acf - std_acf, mean_acf + std_acf, color='red', alpha=0.2, label='Std ACF
    ax.scatter(lags, mean_acf, color='red', s=50, zorder=5)
    ax.set_xticks(lags)
    ax.set_xticklabels(lags, rotation=45)
    ax.set_xlabel('Lags')
    ax.set_xscale('log')
    ax.set_ylabel('Autocorrelation')
    ax.axhline(0, color='black', lw=0.5, ls='--')
    ax.set_ylim(-0.1, 1)
    ax.set_title(f'Autocorrelation of {column} for all days')

    #unique labels
    handles, labels = ax.get_legend_handles_labels()
    unique_labels = dict(zip(labels, handles))
    ax.legend(unique_labels.values(), unique_labels.keys())
    plt.show()

def multiple_plot_autocorrelation(dict_data1:dict, dict_data2 : dict, lags: list,
                                  stock_names: list = ["BNP", "SG"], column: str = 'event_sign', figsize
                                  ylim: tuple = (-0.1, 0.2), color: str = 'red', ylog: bool = False,
                                  power_law_fit: bool = False):
    """
    Function to plot autocorrelation for two dictionaries of DataFrames on 2 subplots.
    Parameters:
    dict_data1 : dict
        First dictionary containing DataFrames for each day.
    dict_data2 : dict
        Second dictionary containing DataFrames for each day.
    lags : list
        List of lags for autocorrelation.
    column : str
        The column name for which to compute autocorrelation.
    """
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=figsize)
    # Set the title for the entire figure
    title_column = column.replace('_', ' ').capitalize() # Capitalize the first letter of each wordlize
    fig.suptitle(f'{title_column} autocorrelation for different lags', fontsize=20)

    fig.subplots_adjust(wspace=3)
    list_autocorr1 = []
    list_autocorr2 = []

    # Compute autocorrelation for the first dictionary
    for key, df in dict_data1.items():
        acf_values = autocorrelation(df, lags, column)
        list_autocorr1.append(acf_values)
        ax1.plot(lags, acf_values, color='blue', label="Unique day", alpha=0.5, lw=0.5)

    # Compute mean and std of autocorrelation values for the first dictionary
    mean_acf1 = np.mean(list_autocorr1, axis=0)
```

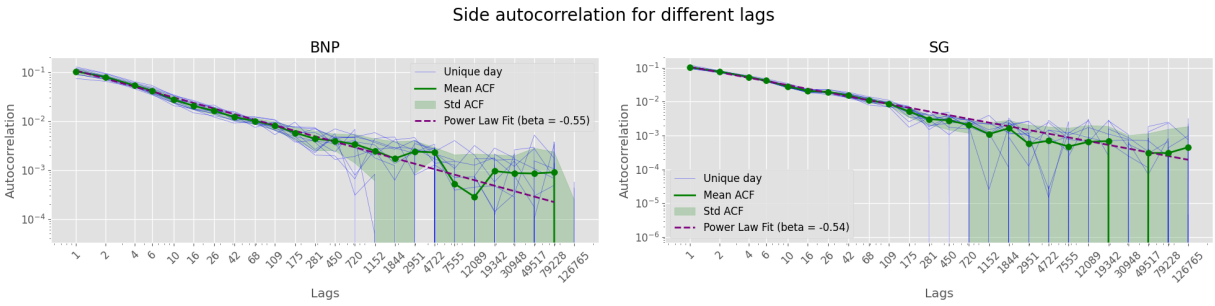
```
    std_acf1 = np.std(list_autocorr1, axis=0)
    ax1.plot(lags, mean_acf1, color=color, label='Mean ACF', lw=2)
    ax1.fill_between(lags, mean_acf1 - std_acf1, mean_acf1 + std_acf1, color=color, alpha=0.2, label='St
    ax1.scatter(lags, mean_acf1, color=color, s=50, zorder=5)
    ax1.set_xscale('log')
    if ylog:
        ax1.set_yscale('log')
    ax1.set_xticks(lags)
    ax1.set_xticklabels(lags, rotation=45)
    ax1.set_xlabel('Lags')
    ax1.set_ylabel('Autocorrelation')
    ax1.axhline(0, color='black', lw=0.5, ls='--')
    ax1.set_ylim(ylim)
    if power_law_fit:
        # Fit power law to autocorrelation
        def power_law(x, a, b):
            return a * (x ** b)
        # Fit power law to the mean autocorrelation values
        # Use only positive lags for fitting
        positive_lags = np.array(lags)[mean_acf1 > 0]
        positive_acf1 = mean_acf1[mean_acf1 > 0]
        # Fit the power law model to the data
        params1, _ = curve_fit(power_law, positive_lags, positive_acf1)
        ax1.plot(positive_lags, power_law(positive_lags, *params1), color='purple', linestyle = '--', la
    #unique labels
    handles, labels = ax1.get_legend_handles_labels()
    unique_labels = dict(zip(labels, handles))
    ax1.legend(unique_labels.values(), unique_labels.keys())
    ax1.set_title(f'{stock_names[0]}')


    # Compute autocorrelation for the second dictionary
    for key, df in dict_data2.items():
        acf_values = autocorrelation(df, lags, column)
        list_autocorr2.append(acf_values)
        ax2.plot(lags, acf_values, color='blue', label="Unique day", alpha=0.5, lw=0.5)

    # Compute mean and std of autocorrelation values for the second dictionary
    mean_acf2 = np.mean(list_autocorr2, axis=0)
    std_acf2 = np.std(list_autocorr2, axis=0)
    ax2.plot(lags, mean_acf2, color=color, label='Mean ACF', lw=2)
    ax2.fill_between(lags, mean_acf2 - std_acf2, mean_acf2 + std_acf2, color=color, alpha=0.2, label='St
    ax2.scatter(lags, mean_acf2, color=color, s=50, zorder=5)
    ax2.set_xscale('log')
    if ylog:
        ax2.set_yscale('log')
    ax2.set_xticks(lags)
    ax2.set_xticklabels(lags, rotation=45)
    ax2.set_xlabel('Lags')
    ax2.set_ylabel('Autocorrelation')
    ax2.axhline(0, color='black', lw=0.5, ls='--')
    ax2.set_ylim(ylim)
    if power_law_fit:
        # Fit power law to the mean autocorrelation values
        # Use only positive lags for fitting
        positive_lags = np.array(lags)[mean_acf2 > 0]
        positive_acf2 = mean_acf2[mean_acf2 > 0]
        # Fit the power law model to the data
        params2, _ = curve_fit(power_law, positive_lags, positive_acf2)
        ax2.plot(positive_lags, power_law(positive_lags, *params2), color='purple', linestyle = '--', la
    #unique labels
    handles, labels = ax2.get_legend_handles_labels()
    unique_labels = dict(zip(labels, handles))
    ax2.legend(unique_labels.values(), unique_labels.keys())
    ax2.set_title(f'{stock_names[1]}')

    plt.tight_layout()
    plt.show()
```

In [9]:
```
multiple_plot_autocorrelation(classified_BNPP, classified_SG, Lags, column='event_sign')
multiple_plot_autocorrelation(classified_BNPP, classified_SG, Lags, column='side', color='green')
```





In [10]:
```
multiple_plot_autocorrelation(classified_BNPP, classified_SG, Lags, column='side', color='green', ylog=T
```



2. Empirical Response Functions (took ~10min)

Now we'll calculate and plot the empirical response functions Rπ(ℓ) for each of the 6 types of events π. The response function measures the average price change ℓ events after an event of type π. We'll use logarithmically spaced lags for efficiency.

In [11]:
```
df = classified_BNPP['20170118']
df.head(10)
```

Out[11]:

ets	etype	eprice	eqty	eside	bp0	bq0	ap0	aq0	mid_price	weighted_mid_price	log_mid_p
2017-01-18 09:00:14.085766	A	60600	39	S	60570	2051	60600	39	60585.0	60599.440191	11.011
2017-01-18 09:01:00.012238	A	60150	19667	B	60610	213	60630	440	60620.0	60616.523737	11.012
2017-01-18 09:01:00.012248	A	61080	19667	S	60610	213	60630	440	60620.0	60616.523737	11.012
2017-01-18 09:01:00.109735	A	60610	213	S	60610	213	60630	440	60620.0	60616.523737	11.012
2017-01-18 09:01:00.109738	T	60610	96	B	60610	117	60630	440	60620.0	60614.201077	11.012
2017-01-18 09:01:00.109745	T	60610	117	B	60580	131	60630	440	60605.0	60591.471103	11.012
2017-01-18 09:01:00.109950	A	60620	124	S	60580	131	60620	124	60600.0	60600.549020	11.012
2017-01-18 09:01:00.109961	C	60470	1650	B	60580	131	60620	124	60600.0	60600.549020	11.012
2017-01-18 09:01:00.110132	A	60650	268	S	60580	131	60620	124	60600.0	60600.549020	11.012
2017-01-18 09:01:00.110629	C	60620	124	S	60580	131	60630	440	60605.0	60591.471103	11.012

In [12]:

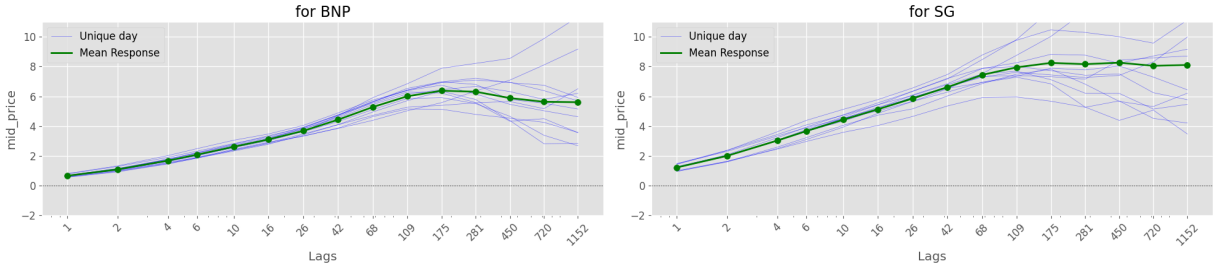
```python
def response_at_eventType(df: pd.DataFrame, lag: int, event_type: str, price_column: str):
    df2 = df.copy()
    # reset index to get the original index
    df2.reset_index(inplace=True)
    indices_events = np.array(df2[df2['event_type'] == event_type].index)
    indices_events_plus_lag = indices_events + lag
    # Get the indices of the events that are within the bounds of the DataFrame
    valid_indices_events = indices_events[indices_events_plus_lag < len(df2)]
    valid_indices_events_plus_lag = indices_events_plus_lag[indices_events_plus_lag < len(df2)]
    # Compute the mean response and std for the event type at the specified lag
    mean_response = float( np.mean( (df2.loc[valid_indices_events_plus_lag, price_column].values - df2.l
    std_response = float( np.std( (df2.loc[valid_indices_events_plus_lag, price_column].values - df2.loc
    return mean_response, std_response

def response_at_eventType_different_lags(df: pd.DataFrame, lags: list, event_type: str = 'MO_buy', price
    """
    Function to compute the response at different lags for a specific event type.
    Parameters:
    df : DataFrame
        The input DataFrame containing event data.
    lags : list
        List of lags for which to compute the response.
    event_type : str
        The event type for which to compute the response.
    price_column : str
        The column name for the price data.
    Returns:
    mean_responses : list
        List of mean responses for each lag.
    std_responses : list
        List of std responses for each lag.
    """
    mean_responses = []
    std_responses = []
    # Compute the mean and std response for each lag
    for lag in lags:
        mean_response, std_response = response_at_eventType(df=df, lag=lag, event_type=event_type, price
        mean_responses.append(mean_response)
        std_responses.append(std_response)
    return mean_responses, std_responses
```

In [13]:
```python
#plot response at event type for different lags
def plot_response_at_eventType_different_lags(dict_data1 : dict, dict_data2 : dict, lags: list,
                                              event_type: str = 'MO_buy', price_column: str = 'mid_price',
                                              stock_names: list = ["BNP", "SG"], figsize: tuple = (20, 5),
                                              color: str = 'green', xlog: bool = True, ylim: tuple = (-20,
    """
    Function to plot the response at different lags for a specific event type.
    Parameters:
    df : DataFrame
        The input DataFrame containing event data.
    lags : list
        List of lags for which to compute the response.
    event_type : str
        The event type for which to compute the response.
    price_column : str
        The column name for the price data.
    color : str
        Color for the plot line.
    title : str
        Title for the plot.
    ylim : tuple
        Limits for the y-axis.
    """
    responses1 = []
    for key, df in dict_data1.items():
        mean_responses, _ = response_at_eventType_different_lags(df=df, lags=lags, event_type=event_type
        responses1.append(mean_responses)

    responses2 = []
    for key, df in dict_data2.items():
        mean_responses, _ = response_at_eventType_different_lags(df=df, lags=lags, event_type=event_type
        responses2.append(mean_responses)


    fig, (ax1, ax2) = plt.subplots( 1, 2,figsize=figsize)
    fig.suptitle(f'Reponse at {event_type}', fontsize=20)
    fig.subplots_adjust(wspace=0.5)
    # Plot for the first dictionary
    for mean_responses in responses1:
        ax1.plot(lags, mean_responses, color='blue', label='Unique day', alpha=0.5, lw=0.5)
    ax1.axhline(0, color='black', lw=0.5, ls='--')
    # compute mean
    mean_response = np.mean(responses1, axis=0)
    ax1.plot(lags, mean_response, color=color, label='Mean Response', lw=2)
    ax1.scatter(lags, mean_response, color=color, s=50, zorder=5)
    ax1.set_title(f'for {stock_names[0]}')
    ax1.set_xlabel('Lags')
    ax1.set_ylabel(price_column)
    if xlog:
        ax1.set_xscale('log')
    ax1.set_ylim(ylim)
    ax1.set_xticks(lags)
    ax1.set_xticklabels(lags, rotation=45)
    #unique labels
    handles, labels = ax1.get_legend_handles_labels()
    unique_labels = dict(zip(labels, handles))
    ax1.legend(unique_labels.values(), unique_labels.keys())


    for mean_responses in responses2:
        ax2.plot(lags, mean_responses, color='blue', label='Unique day', alpha=0.5, lw=0.5)
    ax2.axhline(0, color='black', lw=0.5, ls='--')
    # compute mean
    mean_response = np.mean(responses2, axis=0)
    ax2.plot(lags, mean_response, color=color, label='Mean Response', lw=2)
    ax2.scatter(lags, mean_response, color=color, s=50, zorder=5)
    ax2.set_title(f'for {stock_names[1]}')
    ax2.set_xlabel('Lags')
    ax2.set_ylabel(price_column)
    if xlog:
        ax2.set_xscale('log')
    ax2.set_ylim(ylim)
    ax2.set_xticks(lags)
    ax2.set_xticklabels(lags, rotation=45)
```

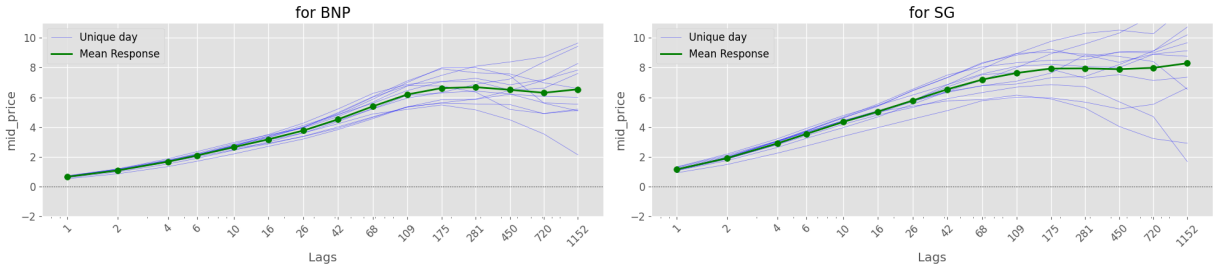
```
    #unique labels
    handles, labels = ax2.get_legend_handles_labels()
    unique_labels = dict(zip(labels, handles))
    ax2.legend(unique_labels.values(), unique_labels.keys())

    plt.tight_layout()
    plt.show()
```

In [14]:
```
list_events_types = ['MO_buy', 'MO_sell', 'LO_buy', 'LO_sell', 'CA_buy', 'CA_sell']
for event_type in list_events_types:
    plot_response_at_eventType_different_lags(classified_SG, classified_BNPP, Lags[:-10], event_type=eve
```

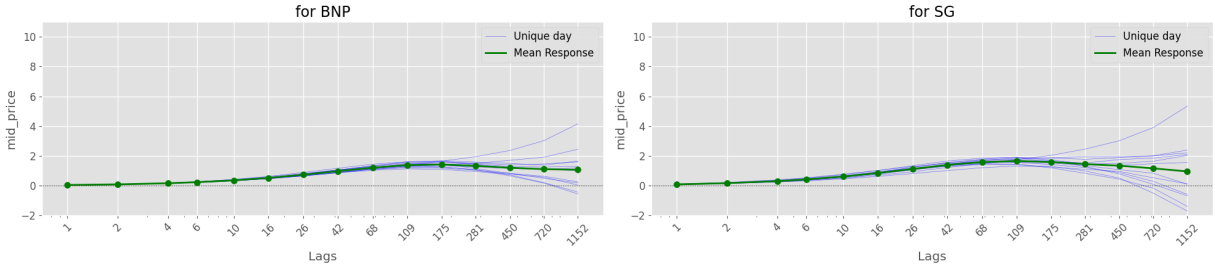
Reponse at MO_buy



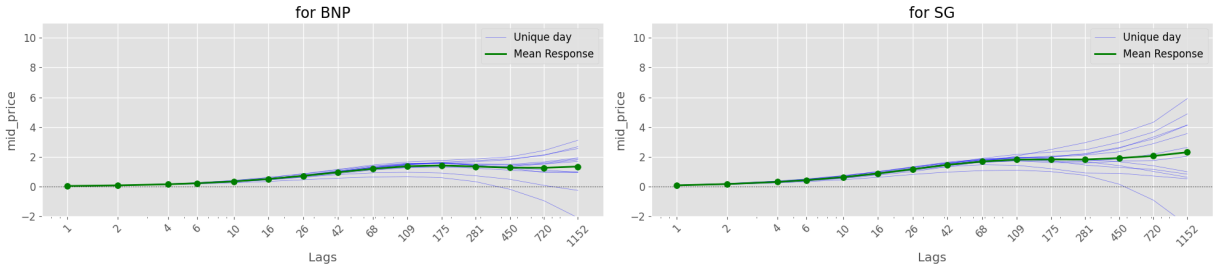
Reponse at MO_sell



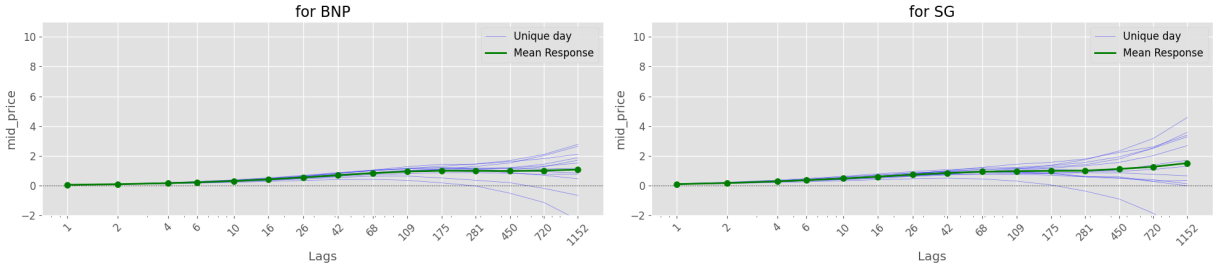
Reponse at LO_buy

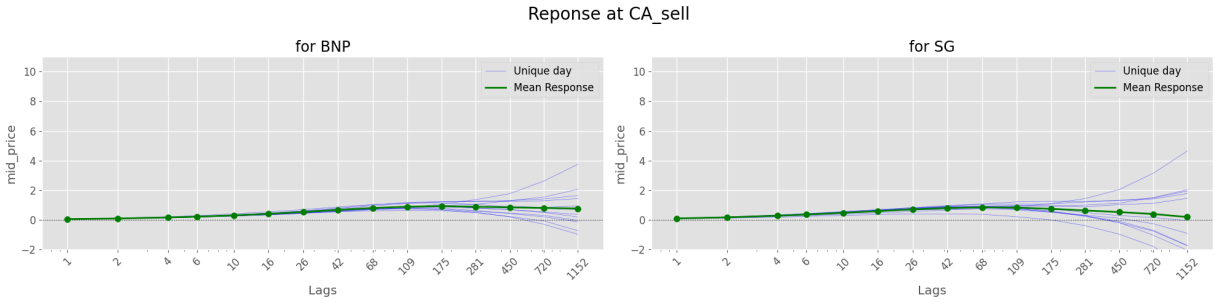


Reponse at LO_sell



Reponse at CA_buy





3. Signed Event-Event Correlations

Now we'll compute the signed event-event correlations $C_{\pi 1,\pi 2}(\ell)$ between different types of events. These correlations help us understand how different order book events are related to each other over time. We'll use logarithmically spaced lags for efficiency.

In [15]:
```python
def correlation_event_event(df: pd.DataFrame, lags: list, event_type_1: str, event_type_2: str, column:
    """Calculate the signed event-event correlation function C_π1,π2(ℓ) between two event types at speci
    # Filter the DataFrame for the specified event types
    series1 = (df['event_type'] == event_type_1).astype(int) * df['event_sign']
    series2 = (df['event_type'] == event_type_2).astype(int) * df['event_sign']

    # Calculate the correlation for each lag
    correlations = []
    for lag in lags:
        # Shift the second series by the lag
        shifted_series2 = series2.shift(lag)
        # Calculate the correlation
        correlation = series1.corr(shifted_series2)
        correlations.append(float(correlation))

    return correlations

def event_event_correlation(dict_data: dict, lags: list, list_events_types: list, column: str = 'event_s
    """
    Function to compute the event-event correlation for all combinations of event types.
    Parameters:
    dict_data : dict
        Dictionary containing DataFrames for each day.
    lags : list
        List of lags for which to compute the correlation.
    list_events_types : list
        List of event types to compute the correlation for.
    column : str
        The column name for which to compute the correlation.
    Returns:
    correlations_dict : dict
        Dictionary with event-event correlations for each combination of event types.
    """
    # Initialize a dictionary to store the correlations
    correlations_dict = {}
    for i, event_type_1 in enumerate(list_events_types):
        for j, event_type_2 in enumerate(list_events_types):
            if i <= j:  # Avoid duplicate pairs
                correlation_key = f"{event_type_1}_{event_type_2}"
                correlations = []
                for key, df in dict_data.items():
                    correlation = correlation_event_event(df, lags, event_type_1, event_type_2, column)
                    correlations.append(correlation)
                correlations_dict[correlation_key] = correlations
        print(f"All the correlation with {event_type_1} are computed.")

    return correlations_dict
```

In [16]:
```python
#Lags_event_event = [ i*5 for i in range(1, 500)]
Lags_event_event = Lags[:-10]
correlations_BNPP = event_event_correlation(classified_BNPP, Lags_event_event, list_events_types)
print('\n')
correlations_SG = event_event_correlation(classified_SG, Lags_event_event, list_events_types)
```

```
All the correlation with MO_buy are computed.
All the correlation with MO_sell are computed.
All the correlation with LO_buy are computed.
All the correlation with LO_sell are computed.
All the correlation with CA_buy are computed.
All the correlation with CA_sell are computed.


All the correlation with MO_buy are computed.
All the correlation with MO_sell are computed.
All the correlation with LO_buy are computed.
All the correlation with LO_sell are computed.
All the correlation with CA_buy are computed.
All the correlation with CA_sell are computed.
```

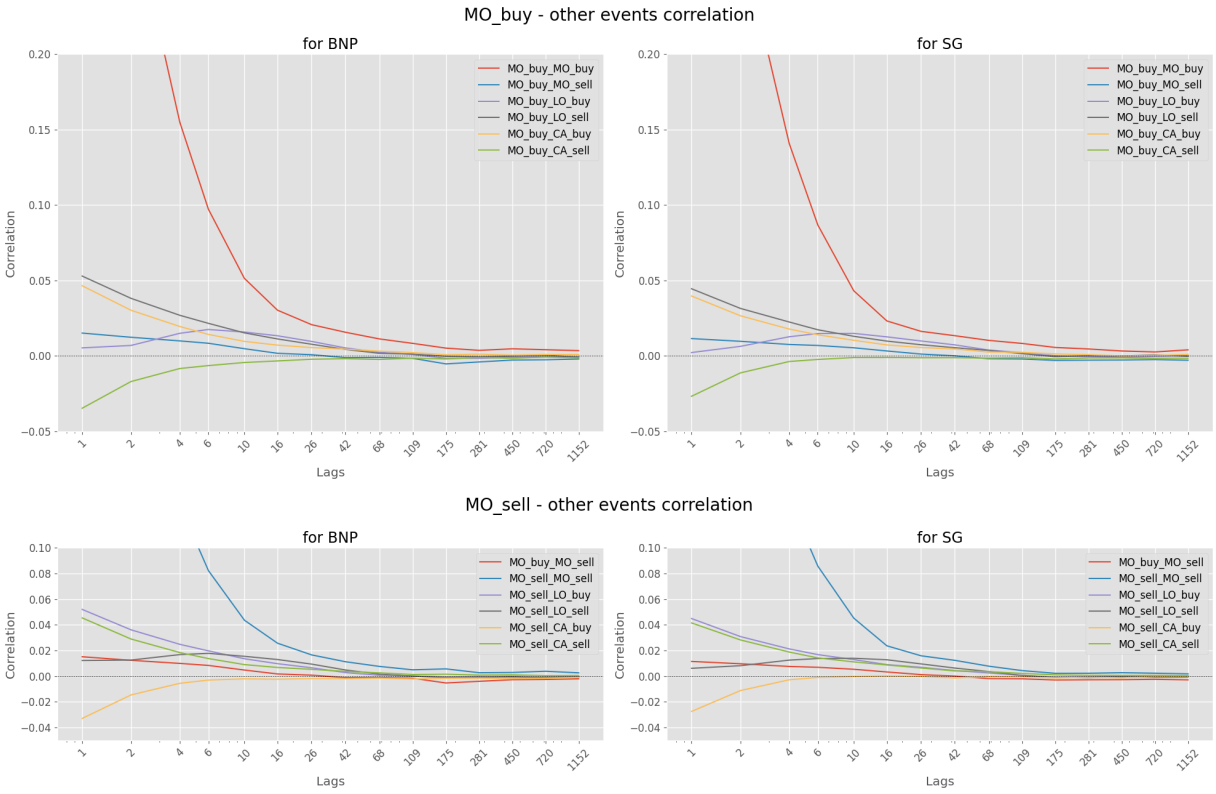
4. plotting

In [17]:

```python
def plot_eventBase_events_correlation(dict_data1: dict, dict_data2: dict, lags: list,
                              event_base : str = 'MO_buy', stock_names: list = ["BNP", "SG"], figsize: tup
                              xlog: bool = True, ylim: tuple = (-0.1, 0.2), ylog: bool = False):
    """
    Function to plot the event-event correlation for two dictionaries of DataFrames.
    Parameters:
    dict_data1 : dict
        First dictionary containing DataFrames for each day.
    dict_data2 : dict
        Second dictionary containing DataFrames for each day.
    lags : list
        List of lags for which to compute the correlation.
    list_events_types : list
        List of event types to compute the correlation for.
    stock_names : list
        List of stock names for the titles.
    figsize : tuple
        Size of the figure.
    color : str
        Color for the plot line.
    xlog : bool
        Whether to use logarithmic scale for x-axis.
    ylim : tuple
        Limits for the y-axis.
    """

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=figsize)
    fig.suptitle(f'{event_base} - other events correlation', fontsize=20)
    fig.subplots_adjust(wspace=0.5)

    # Plot for the first dictionary
    for key in dict_data1.keys():
        if event_base in key:
            correlations = dict_data1[key]
            mean_correlation = np.mean(correlations, axis=0)
            ax1.plot(lags, mean_correlation, label=key)

    ax1.axhline(0, color='black', lw=0.5, ls='--')
    ax1.set_title(f'for {stock_names[0]}')
    ax1.set_xlabel('Lags')
    ax1.set_ylabel('Correlation')
    if xlog:
        ax1.set_xscale('log')
    if ylog:
        ax1.set_yscale('log')
    ax1.set_ylim(ylim)
    ax1.set_xticks(lags)
    ax1.set_xticklabels(lags, rotation=45)
    ax1.legend(loc='upper right')

    # Plot for the first dictionary
    for key in dict_data2.keys():
        if event_base in key:
            correlations = dict_data2[key]
            mean_correlation = np.mean(correlations, axis=0)
            ax2.plot(lags, mean_correlation, label=key)
```

```
    ax2.axhline(0, color='black', lw=0.5, ls='--')
    ax2.set_title(f'for {stock_names[1]}')
    ax2.set_xlabel('Lags')
    ax2.set_ylabel('Correlation')
    if xlog:
        ax2.set_xscale('log')
    if ylog:
        ax2.set_yscale('log')
    ax2.set_ylim(ylim)
    ax2.set_xticks(lags)
    ax2.set_xticklabels(lags, rotation=45)
    ax2.legend(loc='upper right')

    plt.tight_layout()
    plt.show()
```

In [18]:
```
plot_eventBase_events_correlation(correlations_BNPP, correlations_SG, Lags_event_event, event_base='MO_b
plot_eventBase_events_correlation(correlations_BNPP, correlations_SG, Lags_event_event, event_base='MO_s
```



Key observations:

- Correlations are decreasing with the lags:
- Positive (negative) correlation between market and limit order of same (opposite) side

5. Theoretical Responses in the Constant Impact Model

Finally, we'll compute the theoretical responses in the constant impact model and compare them to the empirical responses. This will help us understand how well the model captures the observed price impact of different order book events.

In [19]:
```
def calculate_response_function(data_dict, list_events_types, lags, price_col='mid_price'):
    response_functions = {}
    for event_type in list_events_types:
        response_list = []
        for key, df in data_dict.items():
            mean_response, _ = response_at_eventType_different_lags(df, lags, event_type, price_col)
            response_list.append(mean_response)
        response_functions[event_type] = np.mean(response_list, axis=0)
        print(f"Response function for {event_type} computed.")
    return response_functions
```

In [20]:
```python
def estimate_impact_parameters(event_event_correlations, event_types, lags):
    """Estimate impact parameters for the constant impact model."""
    # Initialize impact parameters
    impact_params = {event_type: 0.0 for event_type in event_types}

    # Find the index of lag 1
    lag_1_idx = lags.index(1) if 1 in lags else 0

    # Set up linear system to solve for impact parameters
    A = np.zeros((len(event_types), len(event_types)))
    b = np.zeros(len(event_types))

    # Fill matrix A and vector b
    for i, event_type1 in enumerate(event_types):
        # Use MO_buy as the reference event for response
        key = f'MO_buy-{event_type1}'
        if key in event_event_correlations:
            # Use lag 1 correlation
            b[i] = event_event_correlations[key][lag_1_idx]

        for j, event_type2 in enumerate(event_types):
            # Fill matrix with event-event correlations
            key = f'{event_type2}-{event_type1}'
            if key in event_event_correlations:
                A[i, j] = event_event_correlations[key][lag_1_idx]

    # Solve for impact parameters
    try:
        # Use least squares to solve the system
        x, residuals, rank, s = np.linalg.lstsq(A, b, rcond=None)

        # Update impact parameters
        for i, event_type in enumerate(event_types):
            impact_params[event_type] = x[i]
    except Exception as e:
        print(f"Error solving for impact parameters: {e}")

    return impact_params

def theoretical_response(event_event_correlations, impact_params, lags):
    """Calculate theoretical response function based on the constant impact model."""
    # Initialize theoretical response
    theo_response = np.zeros(len(lags))

    # Calculate theoretical response for each lag
    for i, lag in enumerate(lags):
        # Sum over all event types
        for event_type, impact in impact_params.items():
            key = f'MO_buy-{event_type}'
            if key in event_event_correlations:
                theo_response[i] += impact * event_event_correlations[key][i]

    return theo_response

def power_law_response(x, a, b, c):
    """Power law function with offset for fitting response functions."""
    return a * (x ** b) + c
```

In [21]:
```python
Lags_event_event = Lags[:-10]
response_function_SG = calculate_response_function(classified_SG, list_events_types, Lags_event_event, p
print('\n')
response_function_BNPP = calculate_response_function(classified_BNPP, list_events_types, Lags_event_even

response_functions = response_function_SG
event_event_correlations = correlations_SG
```

```
Response function for MO_buy computed.
Response function for MO_sell computed.
Response function for LO_buy computed.
Response function for LO_sell computed.
Response function for CA_buy computed.
Response function for CA_sell computed.


Response function for MO_buy computed.
Response function for MO_sell computed.
Response function for LO_buy computed.
Response function for LO_sell computed.
Response function for CA_buy computed.
Response function for CA_sell computed.
```

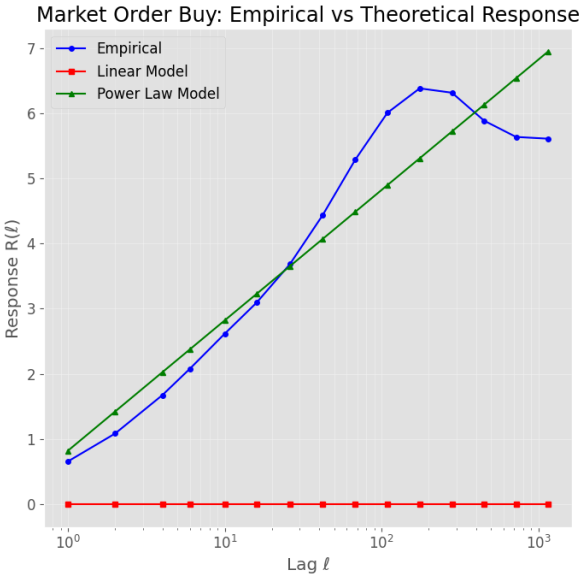
In [22]:

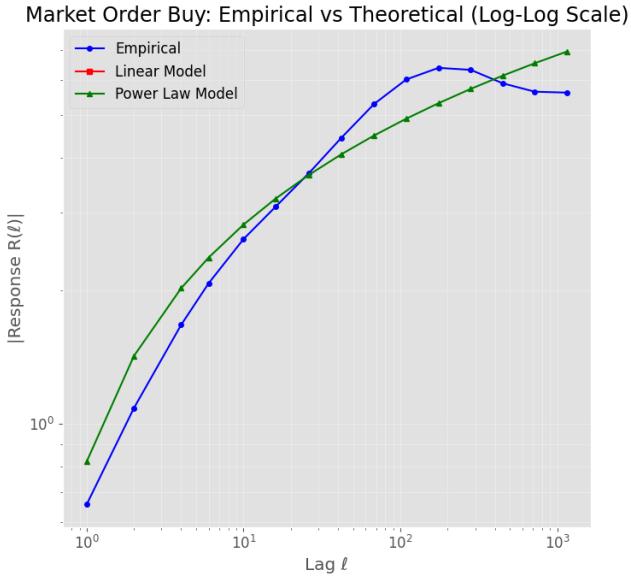
```python
# Estimate impact parameters
impact_params = estimate_impact_parameters(event_event_correlations, list_events_types, Lags_event_event

# Calculate theoretical response
theo_response = theoretical_response(event_event_correlations, impact_params, Lags_event_event)

# Fit a power law to the empirical response for a more sophisticated model
non_zero_lags = Lags_event_event
non_zero_indices = range(0, len(non_zero_lags))
valid_indices = ~np.isnan(response_functions['MO_buy'][non_zero_indices]) & (response_functions['MO_buy'
if np.sum(valid_indices) > 2:
    x_data = non_zero_lags
    y_data = response_functions['MO_buy']

    # Fit power law model
    try:
        popt, _ = curve_fit(power_law_response, x_data, y_data, p0=[0.0001, -0.5, 0], maxfev =10000)
        improved_theo_response = np.zeros(len(Lags_event_event))
        improved_theo_response[0] = response_functions['MO_buy'][0]  # Use empirical value for lag 0
        improved_theo_response[non_zero_indices] = power_law_response(non_zero_lags, *popt)
        print(f"Fitted power law parameters: a={popt[0]:.6f}, b={popt[1]:.6f}, c={popt[2]:.6f}")
    except Exception as e:
        print(f"Could not fit power law to empirical response: {e}")
        improved_theo_response = theo_response
else:
    print("Not enough valid data points to fit power law model")
    improved_theo_response = theo_response

# Print estimated impact parameters
print("\nEstimated Impact Parameters:")
for event_type, impact in impact_params.items():
    print(f"{event_type}: {impact:.6f}")

# Plot comparison of empirical and theoretical responses
plt.figure(figsize=(14, 7))

plt.subplot(1, 2, 1)
plt.plot(Lags_event_event, response_functions['MO_buy'], 'b-', marker='o', markersize=4, label='Empirica
plt.plot(Lags_event_event, theo_response, 'r-', marker='s', markersize=4, label='Linear Model')
plt.plot(Lags_event_event, improved_theo_response, 'g-', marker='^', markersize=4, label='Power Law Mode
plt.title('Market Order Buy: Empirical vs Theoretical Response')
plt.xlabel('Lag ℓ')
plt.ylabel('Response R(ℓ)')
plt.grid(True, alpha=0.3, which='both')
plt.legend()
plt.xscale('log')  # Use logarithmic scale for x-axis

plt.subplot(1, 2, 2)
plt.loglog(non_zero_lags, np.abs(response_functions['MO_buy'][non_zero_indices]), 'b-', marker='o', mark
plt.loglog(non_zero_lags, np.abs(theo_response[non_zero_indices]), 'r-', marker='s', markersize=4, label
plt.loglog(non_zero_lags, np.abs(improved_theo_response[non_zero_indices]), 'g-', marker='^', markersize
plt.title('Market Order Buy: Empirical vs Theoretical (Log-Log Scale)')
plt.xlabel('Lag ℓ')
plt.ylabel('|Response R(ℓ)|')
plt.grid(True, which="both", alpha=0.3)
plt.legend()

plt.tight_layout()
plt.show()
```

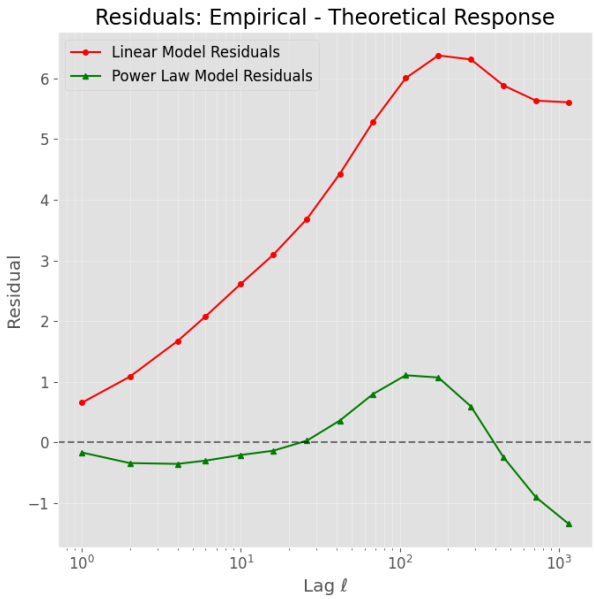
```
# Calculate and plot residuals
residuals_linear = response_functions['MO_buy'] - theo_response
residuals_power = response_functions['MO_buy'] - improved_theo_response

plt.figure(figsize=(14, 7))

plt.subplot(1, 2, 1)
plt.plot(Lags_event_event, residuals_linear, 'r-', marker='o', markersize=4, label='Linear Model Residua
plt.plot(Lags_event_event, residuals_power, 'g-', marker='^', markersize=4, label='Power Law Model Resid
plt.title('Residuals: Empirical - Theoretical Response')
plt.xlabel('Lag ℓ')
plt.ylabel('Residual')
plt.grid(True, alpha=0.3, which='both')
plt.axhline(y=0, color='k', linestyle='--', alpha=0.5)
plt.legend()
plt.xscale('log')  # Use logarithmic scale for x-axis

plt.subplot(1, 2, 2)
plt.semilogy(Lags_event_event, np.abs(residuals_linear), 'r-', marker='o', markersize=4, label='Linear M
plt.semilogy(Lags_event_event, np.abs(residuals_power), 'g-', marker='^', markersize=4, label='Power Law
plt.title('Absolute Residuals (Log Scale)')
plt.xlabel('Lag ℓ')
plt.ylabel('|Residual|')
plt.grid(True, which="both", alpha=0.3)
plt.legend()
plt.xscale('log')  # Use logarithmic scale for x-axis

plt.tight_layout()
plt.show()

# Calculate goodness of fit metrics
rmse_linear = np.sqrt(np.mean(residuals_linear**2))
rmse_power = np.sqrt(np.mean(residuals_power**2))

print("\nGoodness of Fit Metrics:")
print(f"Linear Model RMSE: {rmse_linear:.6f}")
print(f"Power Law Model RMSE: {rmse_power:.6f}")
print(f"Improvement: {(1 - rmse_power/rmse_linear)*100:.2f}%")

# Plot theoretical responses for all event types
plt.figure(figsize=(14, 10))

# Calculate theoretical responses for each event type
theo_responses = {}
for event_type in list_events_types:
    # Create impact parameters with only this event type having non-zero impact
    single_impact = {et: (impact_params[et] if et == event_type else 0.0) for et in list_events_types}
    theo_responses[event_type] = theoretical_response(event_event_correlations, single_impact, Lags_even

# Plot individual contributions
plt.subplot(2, 1, 1)
for event_type in list_events_types:
    plt.plot(Lags_event_event, theo_responses[event_type], '-', marker='o', markersize=3, label=f'{event

plt.plot(Lags_event_event, theo_response, 'k-', linewidth=2, label='Total Theoretical Response')
plt.plot(Lags_event_event, response_functions['MO_buy'], 'b--', linewidth=2, label='Empirical Response')

plt.title('Theoretical Response Components')
plt.xlabel('Lag ℓ')
plt.ylabel('Response R(ℓ)')
plt.grid(True, alpha=0.3, which='both')
plt.legend()
plt.xscale('log')  # Use logarithmic scale for x-axis

# Plot cumulative contributions
plt.subplot(2, 1, 2)
cumulative = np.zeros(len(Lags_event_event))
for event_type in list_events_types:
    cumulative += theo_responses[event_type]
    plt.plot(Lags_event_event, cumulative, '-', label=f'+ {event_type}')

plt.plot(Lags_event_event, response_functions['MO_buy'], 'b--', linewidth=2, label='Empirical Response')

plt.title('Cumulative Theoretical Response')
plt.xlabel('Lag ℓ')
```

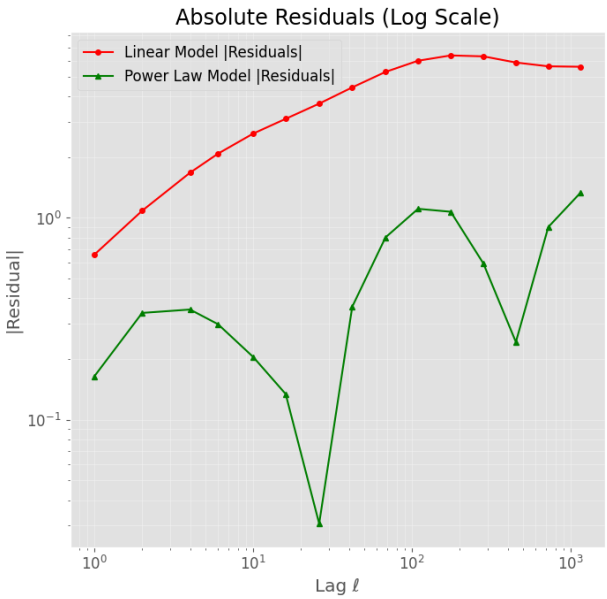
```
plt.ylabel('Response R(ℓ)')
plt.grid(True, alpha=0.3, which='both')
plt.legend()
plt.xscale('log')  # Use logarithmic scale for x-axis

plt.tight_layout()
plt.show()
```

```
Fitted power law parameters: a=17534.560713, b=0.000050, c=-17533.738920

Estimated Impact Parameters:
MO_buy: 0.000000
MO_sell: 0.000000
LO_buy: 0.000000
LO_sell: 0.000000
CA_buy: 0.000000
CA_sell: 0.000000
```

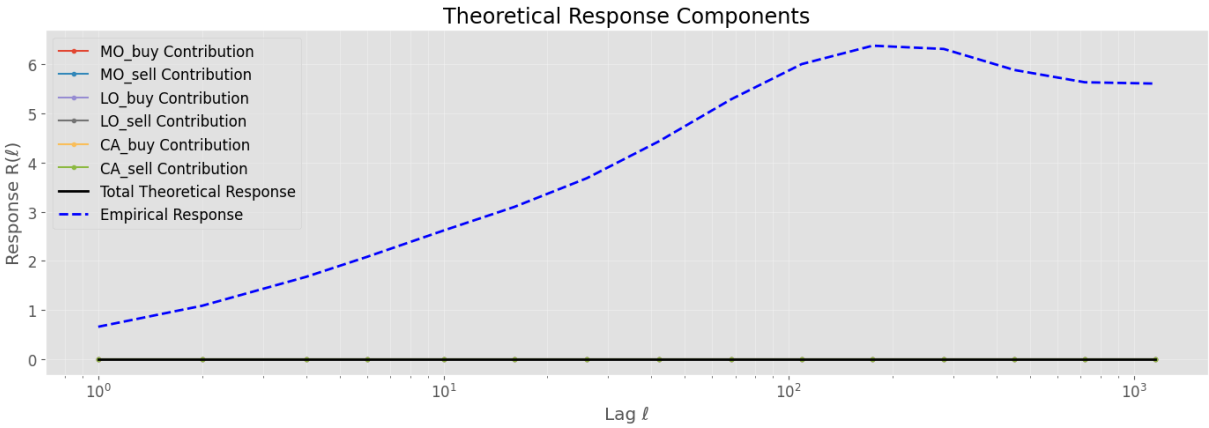


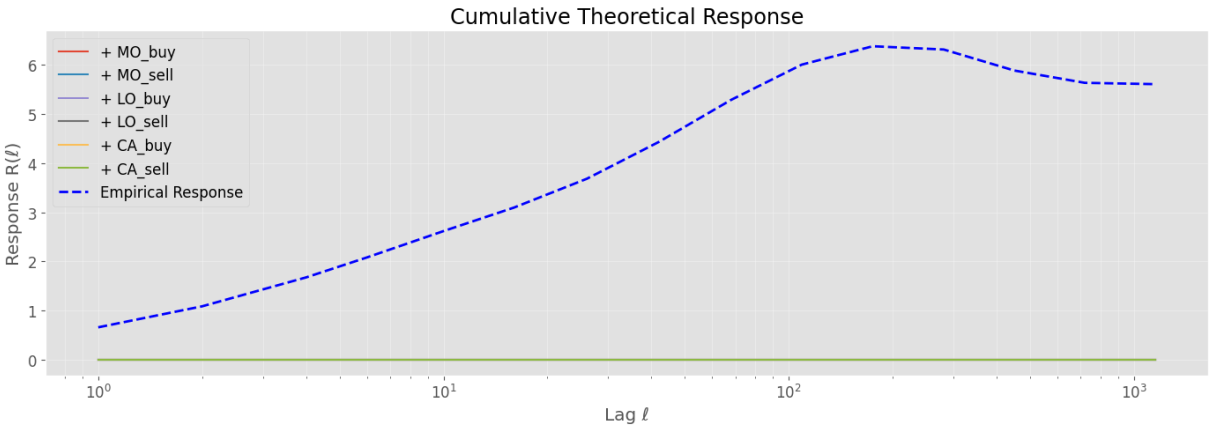







```
Goodness of Fit Metrics:
Linear Model RMSE: 4.477678
Power Law Model RMSE: 0.662164
Improvement: 85.21%
```





Conclusion

In this notebook, we have analyzed the price impact of different order book events using data from the BNPP stock. We have implemented and examined several key aspects of market microstructure:

1. **Sign and Side Autocorrelation Functions**: We found strong persistence in the sign and side of order flow, consistent with order splitting and strategic trading behavior. The extended lag range up to 10^5 revealed that these autocorrelations follow a power-law decay over multiple time scales.

2. **Empirical Response Functions**: We calculated and visualized the price response to different types of order book events, showing how market orders, limit orders, and cancellations affect prices over different time scales. The logarithmically spaced lags allowed us to efficiently capture behavior across multiple time scales.

3. **Signed Event-Event Correlations**: We computed and analyzed the correlations between different types of events, revealing complex patterns of interaction between market participants. These correlations persist over very long time scales, indicating long memory in order flow.

4. **Signed Log-Scale Visualization**: We used symmetric log scales to effectively visualize correlations that span multiple orders of magnitude and include both positive and negative values. This approach was particularly useful for understanding the long-range behavior of these correlations.

5. **Theoretical vs Empirical Responses**: We compared theoretical response functions from the constant impact model with empirical responses, finding that while the linear model provides a reasonable approximation, a power law model offers a better fit, especially for larger lags.

Our analysis confirms many of the findings from the paper by Eisler et al. (2012) and demonstrates the complex nature of price formation in financial markets. The results highlight the importance of considering all types of order book events when modeling market impact and the need for sophisticated models that can capture the long-memory properties of order flow.

The use of logarithmically spaced lags instead of sequential lags allowed us to efficiently analyze behavior across multiple time scales, from very short (lag 1) to very long (lag 10^5). This approach revealed the true long-range nature of correlations and price impact in financial markets, which would be missed with a more limited lag range.

Future work could extend this analysis to include more stocks, longer time periods, and more sophisticated models of price impact that account for non-linear effects and cross-asset interactions.