Assignment 3

Sigvard Bratlie

In [1]:
```
import pandas as pd
import requests
import matplotlib.pyplot as plt
import numpy as np
from scipy.fft import dct, idct
from scipy.stats import median_abs_deviation
from scipy import signal
from sklearn.neighbors import LocalOutlierFactor
from statsmodels.tsa.seasonal import STL
from pymongo import MongoClient
import os
from typing import Literal
import plotly.graph_objects as go
from plotly.subplots import make_subplots
from scipy.signal import spectrogram
```

Fetch data

In [2]:
```
def mk_request(url: str,params: dict = None):
    try:
        response = requests.get(url, params=params)
        response.raise_for_status()
        data = response.json()
        return data
    except requests.exceptions.RequestException as e:
        print(f"Error fetching data: {e}")
        return None

#Function for the API download
def get_weather(lat : float , lon:float, year : int, ):
    params = {"latitude" : lat, "longitude": lon,
              "start_date": f"{year}-01-01",
              "end_date": f"{year}-12-31",
              "hourly": "temperature_2m,precipitation,wind_speed_10m,wind_gusts_10m_spread,wind_direction_10m",
              "models" : "era5"
              }
    base_url = "https://archive-api.open-meteo.com/v1/archive?"
    return mk_request(base_url,params=params)
```

Geocoding

Oslo, Kristiansand, Trondheim, Tromsø and Bergen

- Østlandet (NO1)
- Sørvest-Norge (NO2)
- Midt-Norge (NO3)
- Nord-Norge (NO4)
- Vestlandet (NO5)

In [3]:
```
def geocode(city : str):
    url = f"https://geocoding-api.open-meteo.com/v1/search?name={city}&count=10&language=en&format=json"
    return mk_request(url)

def extract_coordinates(city: str):
    res = geocode(city).get("results")[0]
    lat, lon = res.get("latitude"), res.get("longitude")
    return lat, lon
```

In [4]:
```
cities = {"Oslo" : "NO1",
          "Kristiansand" : "NO2",
          "Trondheim" : "NO3",
          "Tromsø" : "NO4",
          "Bergen" : "NO5"}
geocoded_cities = {}
for city,price_area in cities.items():
    lat, lon = extract_coordinates(city)
    geocoded_cities[city] = (lat, lon,price_area)

cities_df = pd.DataFrame.from_dict(geocoded_cities, orient="index", columns=["latitude", "longitude", "price_area"])
cities_df
```

Out[4]:

	latitude	longitude	price_area
Oslo	59.91273	10.74609	NO1
Kristiansand	58.14671	7.99560	NO2
Trondheim	63.43049	10.39506	NO3
Tromsø	69.64890	18.95508	NO4
Bergen	60.39299	5.32415	NO5

Weather data for Bergen

In [5]:
```
lat,lon = extract_coordinates("Bergen")
data = get_weather(lat, lon, 2019)
```

Outliers and anomalies

In [6]:
```
df = pd.DataFrame(data.get("hourly"))
df["time"] = pd.to_datetime(df["time"])
```

```
df.set_index("time", inplace=True)
df.head()
```

Out[6]:

	temperature_2m	precipitation	wind_speed_10m	wind_gusts_10m_spread	wind_direction_10m
time					
2019-01-01 00:00:00	6.6	0.5	48.0	5.0	278
2019-01-01 01:00:00	6.8	0.9	48.6	5.4	296
2019-01-01 02:00:00	6.8	0.7	52.6	5.8	310
2019-01-01 03:00:00	6.6	0.6	55.8	6.1	314
2019-01-01 04:00:00	6.2	1.0	55.5	5.8	317

In [7]:
```
df.index.min(), df.index.max()
```

Out[7]:
```
(Timestamp('2019-01-01 00:00:00'), Timestamp('2019-12-31 23:00:00'))
```

In [8]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 8760 entries, 2019-01-01 00:00:00 to 2019-12-31 23:00:00
Data columns (total 5 columns):
 #   Column                 Non-Null Count  Dtype  
---  ------                 --------------  -----  
 0   temperature_2m         8760 non-null   float64
 1   precipitation          8760 non-null   float64
 2   wind_speed_10m         8760 non-null   float64
 3   wind_gusts_10m_spread  8760 non-null   float64
 4   wind_direction_10m     8760 non-null   int64  
dtypes: float64(4), int64(1)
memory usage: 410.6 KB
```

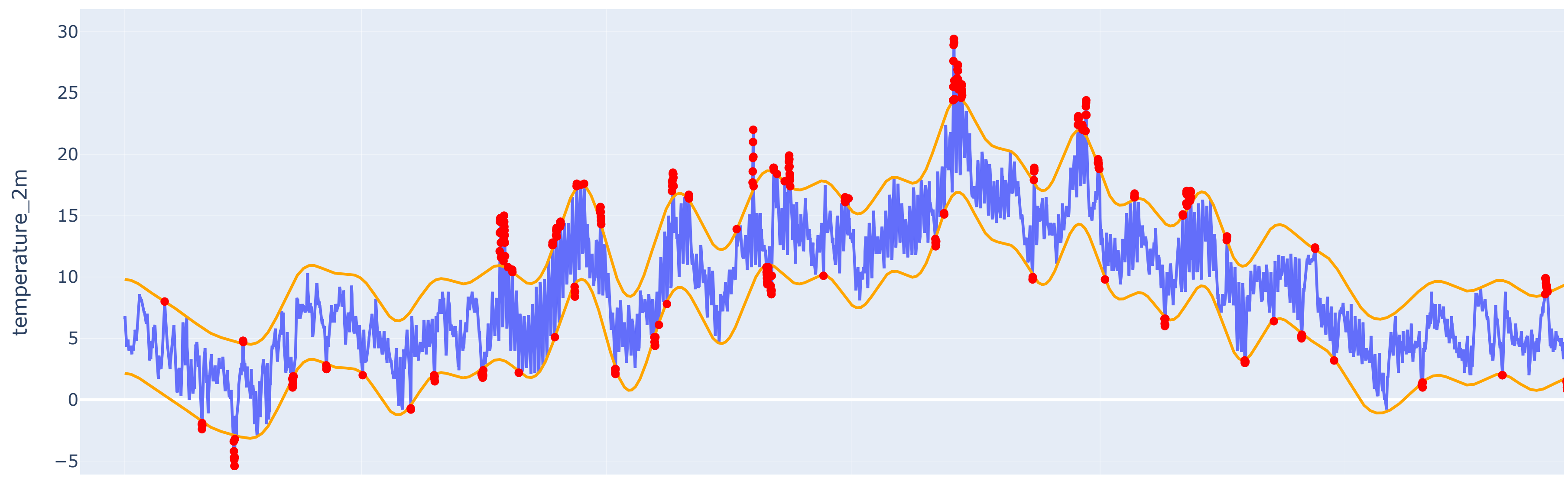
Temperature

- Perform a high-pass filtering of the temperature using Direct Cosine Transfer to create seasonally adjusted temperature variations (SATV).
- Add curves to the plot indicating Statistical Process Control boundaries between inliers and outliers based on the SATV according to robust statistics estimated from the * whole year. Colour outliers with a contrasting colour. Do not plot SATV values; only use them to find boundaries and outliers.
- Let the frequency cut-off for the DCT and the number of standard deviations be parameters with sensible defaults.
- Wrap this in a function that returns the plot and relevant summaries of the outliers, and test the function.

In [9]:
```
def calc_highpass(data, cutoff: int):
    fourier = dct(data, norm="forward")
    #Plot the temperature as a function of time
    satv = fourier.copy()
    f = np.arange(0, len(satv))
    satv[f<cutoff] = 0 #high pass filter
    return idct(satv, norm="forward")

def high_pass(df : pd.DataFrame, feature,cutoff : int = 50,nstd : float = 2.0):
    temp = df[feature].to_numpy()

    satv_reconstructed = calc_highpass(temp, cutoff)
    mean,std = satv_reconstructed.mean(), satv_reconstructed.std()

    outliers = np.where((satv_reconstructed > mean + nstd*std) | (satv_reconstructed < mean - nstd*std))
    df_outliers = df.iloc[outliers]

    low_pass_reconstructed = temp - satv_reconstructed
    print(f"Number of outliers detected: {len(outliers[0])}")

    fig = go.Figure()
    fig.add_trace(go.Scatter(x=df.index, y=df[feature], mode='lines', name='Original'))
    fig.add_trace(go.Scatter(x=df.index, y=low_pass_reconstructed + nstd*std, mode='lines', name='Upper boundary', line=dict(color='orange')))
    fig.add_trace(go.Scatter(x=df.index, y=low_pass_reconstructed - nstd*std, mode='lines', name='Lower boundary', line=dict(color='orange')))
    fig.add_trace(go.Scatter(x=df_outliers.index, y=df_outliers[feature], mode='markers', name='Outliers', marker=dict(color='red')))
    fig.update_layout(title='Temperature Data with lower and upper boundaries',
                      xaxis_title='Time',
                      yaxis_title=feature)
    return fig

fig = high_pass(df, feature="temperature_2m", cutoff=50,nstd=2.0)
fig.show(renderer = "notebook")
```

```
Number of outliers detected: 347
```

Temperature Data with lower and upper boundaries

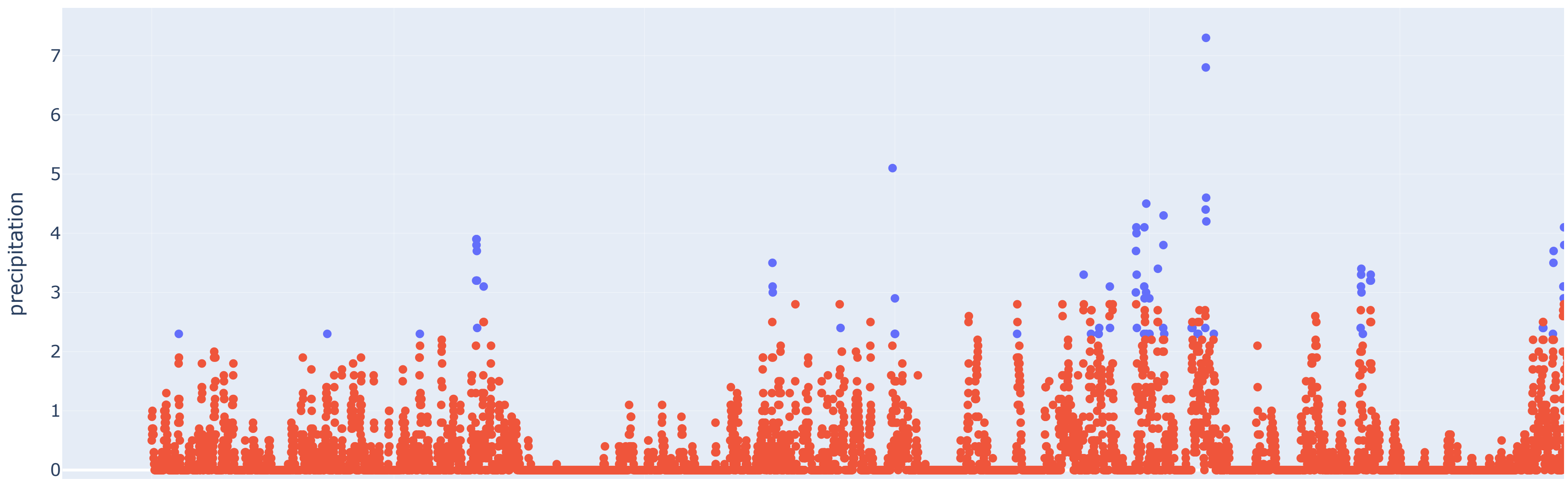


Precipitation

- Indicate anomalies according to the Local Outlier Factor method.
- Let the proportion of outliers be a parameter defaulting to 1%.
- Wrap this in a function that returns the plot and relevant summaries of the outliers, and test the function.

In [10]:
```
def lof(df,feature,n_neighbors: int = 20, contamination: float = 0.01):
    lof = LocalOutlierFactor(n_neighbors=n_neighbors, contamination=contamination)
    data = df[[feature]]
    labels = lof.fit_predict(data)
    outliers = data[labels == -1]
    inliers = data[labels == 1]

    fig = go.Figure()
    fig.add_trace(go.Scatter(x=outliers.index, y=outliers[feature], mode='markers',
                             name='Outliers'))
    fig.add_trace(go.Scatter(x=inliers.index, y=inliers[feature], mode='markers',
                             name='Inliers'))
    fig.update_layout(title='',
                      xaxis_title='Time',
                      yaxis_title=feature)
    return fig
fig = lof(df = df,feature="precipitation", contamination=0.01, n_neighbors=20)
fig.show(renderer = "notebook")
```

```
/Users/sigvardbratlie/miniconda3/envs/datsci/lib/python3.11/site-packages/sklearn/neighbors/_lof.py:322: UserWarning:

Duplicate values are leading to incorrect results. Increase the number of neighbors for more accurate results.
```



LOESS (STL)

- Perform LOESS on the production data from elbub (downloaded in part 1 of the project) and plot its decomposition.
- Let the electricity price area, production group, period length, seasonal smoother, trend smoother and robust (true/false) be parameters, and give each of them sensible defaults.
- Wrap this in a function that returns the plot, and test the function.

In [11]:
```
from dotenv import load_dotenv
load_dotenv()
```

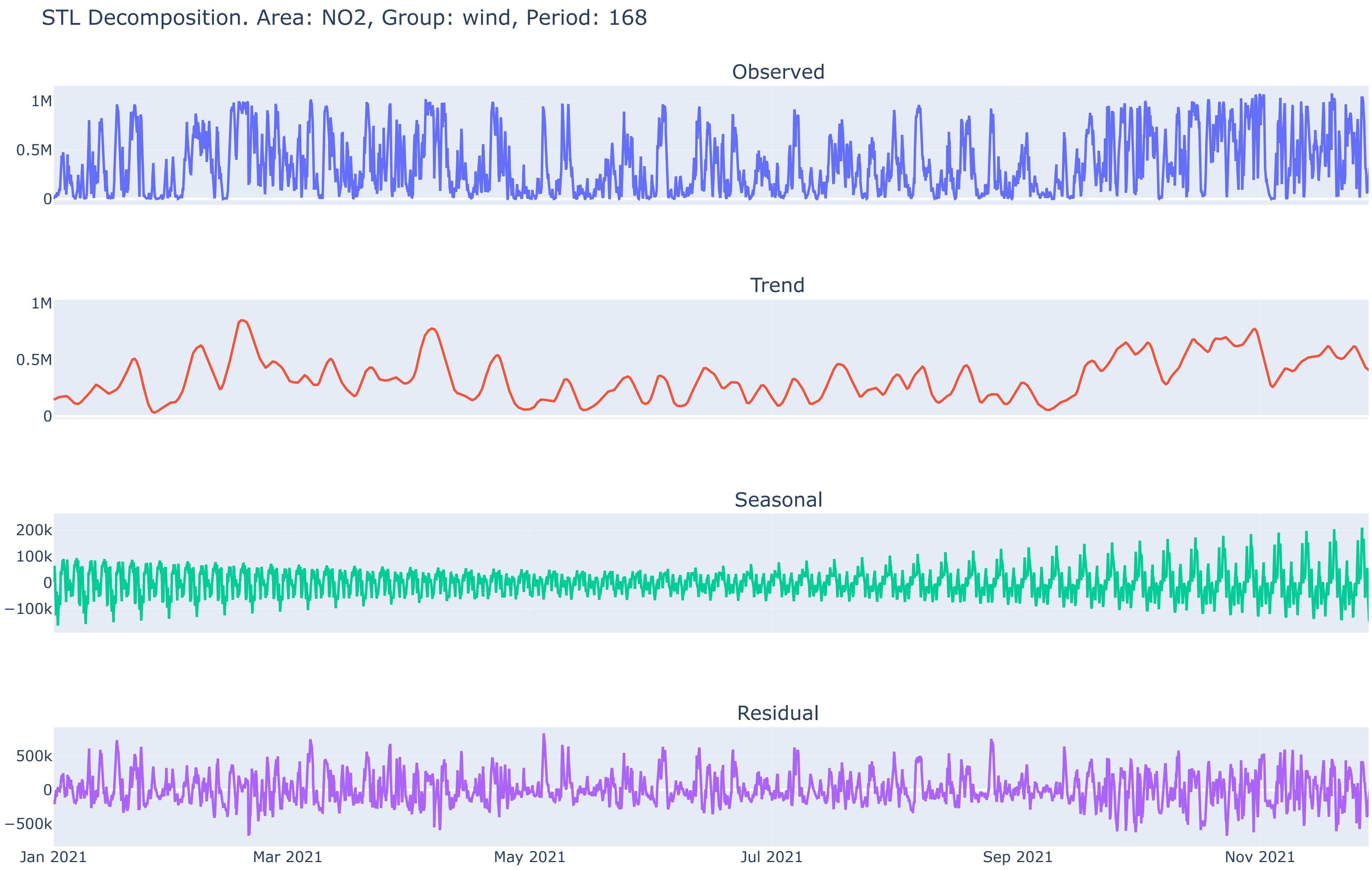
```python
def get_client():
   # Connection string from MongoDB
   CONNECTION_STRING = f"mongodb+srv://sigvardbratlie:{os.getenv('MONGODB_PASSWORD')}@cluster0.y7mplij.mongodb.net/?retryWrites=true&w=majority&a

   # Create a connection using MongoClient.
   client = MongoClient(CONNECTION_STRING)
   try:
      client.admin.command("ping")
      print(f'Everything Okay') #print if connection is successful
      return client
   except Exception as e:
     print(e)

def get_data(client):
    db = client.elhub
    items = db.prod_data.find({})
    items = list(items)  # make hashable for st.cache_data
    data = pd.DataFrame(items)
    data.set_index("starttime", inplace=True)
    data.sort_index(inplace=True)
    data.drop(columns=["_id"], inplace=True)
    return data

elhub = get_data(get_client())
elhub.info()
```

```
Everything Okay
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 215353 entries, 2021-01-01 00:00:00 to 2021-12-31 23:00:00
Data columns (total 3 columns):
 #   Column           Non-Null Count   Dtype
---  ------           --------------   -----
 0   pricearea        215353 non-null  object
 1   productiongroup  215353 non-null  object
 2   quantitykwh      215353 non-null  float64
dtypes: float64(1), object(2)
memory usage: 6.6+ MB
```

In [12]:
```python
elhub["pricearea"].unique().tolist(), elhub["productiongroup"].unique().tolist()
```

Out[12]:
```
(['NO5', 'NO3', 'NO1', 'NO4', 'NO2'],
 ['other', 'thermal', 'hydro', 'solar', 'wind'])
```

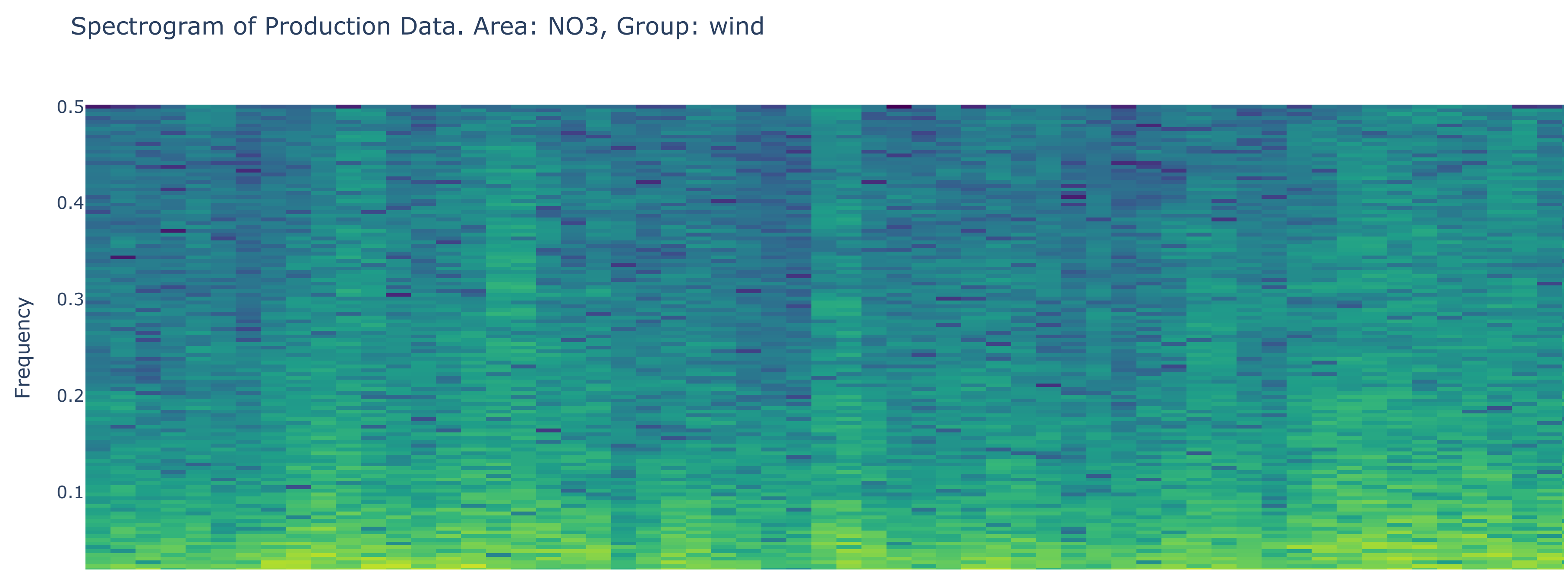
In [13]:
```python
def loess(data : pd.DataFrame,
          price_area : Literal["NO1","NO2","NO3","NO5","NO5"] = "NO2",
          production_group : Literal["hydro","wind","solar","thermal"] = "hydro",
          period : int = 24*7,
          seasonal_smoother : int = 141,
        trend_smoother : int = 141,
        robust  : bool = True,
          ):

        if period > trend_smoother:
            trend_smoother = period + 1 if period % 2 == 0 else period

        data = data.loc[(data["pricearea"] == price_area) & (data["productiongroup"] == production_group), "quantitykwh"]

        stl = STL(data,
                period = period,
                robust=robust,
                seasonal=seasonal_smoother,
                trend=trend_smoother,
                )

        res = stl.fit()

        fig = make_subplots(rows=4, cols=1, shared_xaxes=True,
                            subplot_titles=("Observed", "Trend", "Seasonal", "Residual"))

        fig.add_trace(go.Scatter(x=res.observed.index, y=res.observed, name="Observed"), row=1, col=1)
        fig.add_trace(go.Scatter(x=res.trend.index, y=res.trend, name="Trend"), row=2, col=1)
        fig.add_trace(go.Scatter(x=res.seasonal.index, y=res.seasonal, name="Seasonal"), row=3, col=1)
        fig.add_trace(go.Scatter(x=res.resid.index, y=res.resid, name="Residual"), row=4, col=1)
        fig.update_layout(height=800, width=1400,
                        title_text = f"STL Decomposition. Area: {price_area}, Group: {production_group}, Period: {period}"
                        )

        return fig

fig = loess(data = elhub,
      period = 24*7,
      production_group="wind",
      price_area="NO2") #Weekly seasonality for wind
fig.show(renderer = "notebook")
```



Spectrogram

- Create a spectrogram based on the production data from elhub.
- Let the electricity price area, production group, window length and window overlap be parameters, and give each of them sensible defaults.
- Wrap this in a function that returns the plot, and test the function.

In [14]:
```
def spectrogram(data : pd.DataFrame,
        price_area : Literal["NO1","NO2","NO3","NO5","NO5"] = "NO2",
          production_group : Literal["hydro","wind","solar","thermal"] = "hydro",
          window_length : int = 256,
          overlap : int = 128,
          ):

    data = data.loc[(data["pricearea"] == price_area) & (data["productiongroup"] == production_group), "quantitykwh"]

    fs = 1.0 #sampling frequency for hourly data
    f, t, Sxx = signal.spectrogram(data.to_numpy(),
                                   fs,
                                   nperseg=window_length,
                                   noverlap=overlap,)

    fig = go.Figure(data=go.Heatmap(
        z=10 * np.log10(Sxx),  # convert to dB scale
        x=t,
        y=f,
        colorscale='Viridis'
    ))

    fig.update_layout(
        title='Spectrogram of Production Data. Area: ' + price_area + ', Group: ' + production_group,
        xaxis_title='Time',
        yaxis_title='Frequency'
    )
    return fig

fig = spectrogram(elhub, production_group="wind",
            price_area="NO3")
fig.show(renderer = "notebook")
```



AI Usage

Used for content refinement (spelling, syntax debugging) and structuring/drafting log entries.
Especially helpful for issues related to cassandra-spark connection errors.

Specifically for this CA3, AI was used to understand and implemenent the different statistical methods in `New A` and `New B`.

Project Log (300-500 words)

Assignment 3: Advanced Time Series Analysis and Anomaly Detection

This assignment built upon previous work by introducing advanced statistical methods for analyzing Norwegian electricity production and meteorological data. The work was divided between Jupyter Notebook development and Streamlit application deployment.

Jupyter Notebook Development: The notebook development began with integrating the Open-Meteo API to fetch historical ERA5 reanalysis data. Creating a flexible geocoding system for Norwegian cities (Oslo, Kristiansand, Trondheim, Tromsø, Bergen) and mapping them to their respective price areas (NO1-NO5) provided a solid foundation. The API integration function was designed to be reusable, accepting latitude, longitude, and year parameters to fetch temperature, precipitation, wind speed, and wind direction data.

The outlier detection for temperature data using Direct Cosine Transform (DCT) proved challenging but rewarding. Implementing high-pass filtering to create seasonally adjusted temperature variations (SATV) required understanding both frequency domain analysis and statistical process control. The visual representation with upper and lower control boundaries effectively highlighted temperature anomalies while filtering out seasonal patterns. Fine-tuning the cutoff frequency and standard deviation parameters was essential to achieve meaningful results.

For precipitation anomaly detection, the Local Outlier Factor (LOF) method provided a different approach compared to the SPC method. The sklearn implementation was straightforward. The visualization clearly distinguished between normal rainfall patterns and anomalous precipitation events.

The STL (Seasonal-Trend decomposition using LOESS) implementation for electricity production data was particularly interesting. Decomposing the time series into observed, trend, seasonal, and residual components revealed underlying patterns in renewable energy production. Managing the parameter constraints (ensuring trend_smoother exceeds period) required careful validation logic.

Creating the spectrogram for frequency analysis of production data added another dimension to understanding patterns. The scipy.signal.spectrogram function, combined with Plotly heatmaps, produced insightful visualizations showing how production frequencies vary over time.

Streamlit Application: Updating the Streamlit app involved reorganizing page structure and replacing CSV imports with live API calls. The page reorganization (moving the price area selector earlier in the flow) improved user experience. Implementing st.tabs() for the new pages provided clean separation between STL/Spectrogram analysis and Outlier/Anomaly detection. Ensuring all functions worked correctly with user-selected parameters and locations required thorough testing. The interactive nature of Streamlit made the analysis tools accessible and user-friendly.

Overall, this assignment successfully combined advanced statistical methods with practical web application development, providing valuable experience in both analytical and deployment aspects of data science projects.

Links

GitHub Repository: https://github.com/sigvardbratlie/ind320 \

Streamlit App: https://ind320-h63n5qj5uc26acyzlq3x39.streamlit.app/
