DCC-analyse med EGARCH-modell

Denne anaylsen undersøker den dynamiske samvariasjonen mellom to finansielle tidsserier ved bruk av EGARCH og DCC. Modellene estimeres basert på daglige strømpriser fra Norge og Tyskland i peiroden 2019 til 2024.

Analysen retter fokus mot mulige strukturelle endringer rundt Russlands invasjon av Ukraina 24. februar 2022, som er markert i figurene.

Importer biblioteker og definer konstanter

Denne seksjonen laster inn nødvendige Python-biblioteker for dataanalyse, statistisk modellering, visualisering og eksport til Excel.
I tillegg defineres konstanter som brukes gjennom hele analysen:

- Filbaner for input og output
- Tidsperiode for analyse
- Navnemapping for figurer
- Datoe for invasjonen
- Fargepalett for visualiseringer

Mapper for lagring av resultater opprettes automatisk dersom de ikke eksisterer.

In [ ]:
```
# --- Importer nødvendige biblioteker ---

# Standardbibliotek
import os
import re
import inspect
from pathlib import Path
from datetime import datetime
from itertools import product

# Tredjepartsbibliotek: Data og analyse
import numpy as np
import pandas as pd
from scipy.optimize import minimize
from scipy.stats import skew, kurtosis, gaussian_kde, probplot

# Tredjepartsbibliotek: Modellering og statistikk
from arch import arch_model
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.stats.diagnostic import het_arch, acorr_ljungbox
from statsmodels.tsa.stattools import adfuller, acf, pacf

# Tredjepartsbibliotek: Visualisering
import matplotlib.pyplot as plt
import plotly.express as px
import plotly.graph_objects as go
from IPython.display import display
```

```
# Tredjepartsbibliotek: Excel-eksport
from pandas import ExcelWriter
from openpyxl import Workbook, load_workbook
from openpyxl.utils import get_column_letter
from openpyxl.utils.dataframe import dataframe_to_rows
from openpyxl.styles import Font, Border, Side

# --- Konstanter: Filbaner ---
INPUT_DIR = Path("input/daily_aggregate")
OUTPUT_DIR = Path("output")
EXCEL_DIR = OUTPUT_DIR / "excel"

# Sørg for at output-mapper eksisterer
OUTPUT_DIR.mkdir(parents=True, exist_ok=True)
EXCEL_DIR.mkdir(parents=True, exist_ok=True)

# --- Konstanter: Tidsperiode og parametere ---
YEARS = range(2019, 2025)
ROLLING_WINDOW = 30
FIGURE_EXPORT_SCALE = 25

# --- Konstanter: Navnemapping for figurer ---
NAME_MAP = {
    "GER": "Tyskland",
    "NO2": "Norge"
}

# --- Viktige datoer (for analyse og figurer) ---
BREAK_DATE = "2022-02-24"
BREAK_DATE_LABEL = "24. feb 2022"
BREAK_DATE_DT = pd.to_datetime(BREAK_DATE)

# --- Fargepalett for visualisering ---
COLOR_1 = "#1f77b4"  # blå
COLOR_2 = "#ff7f0e"  # oransje
COLOR_3 = "#2ca02c"  # grønn
COLORS = [COLOR_1, COLOR_2, COLOR_3]
```

In [ ]:

```
common_layout: dict = {
    "template": "plotly_white",
    "font": {
        "family": "Times New Roman",
        "size": 16,
        "color": "black",
    },
    "margin": {
        "l": 80,
        "r": 40,
        "t": 80,
        "b": 70,
    },
    "legend": {
        "title": {"text": ""},
        "orientation": "h",
        "yanchor": "bottom",
        "y": 1.02,
        "xanchor": "right",
        "x": 1,
    },
```

```
    "hovermode": "x unified",
    "xaxis": {
        "showgrid": True,
        "title_font": {
            "size": 16
        },
        "tickfont": {
            "size": 14
        },
    },
    "yaxis": {
        "showgrid": True,
        "title_font": {
            "size": 16
        },
        "tickfont": {
            "size": 14
        },
    }
}
```

Støttefunksjoner

Her defineres nødvendige støttefunksjoner for analyse og presentasjon.

In [ ]:
```
def add_break_line(
    fig: go.Figure,
    x: float,
    label: str,
    color: str = "red",
    dash: str = "dash",
    line_width: int = 2
) -> None:
    """
    Legger til en vertikal linje i en Plotly-figur for å indikere et bruddpunkt.

    Parametre:
    ----------
    fig : go.Figure
        Plotly-figur som linjen skal legges til i.
    x : float
        X-posisjon for linjen (f.eks. dato eller tallverdi).
    label : str
        Navn som vises i figurens legend.
    color : str, optional
        Farge på linjen, som Plotly-fargenavn eller hex-kode. Standard er "red".
    dash : str, optional
        Linjetype ("solid", "dash", "dot"). Standard er "dash".
    line_width : int, optional
        Tykkelse på linjen. Standard er 2.

    Returnerer:
    -----------
    None
    """

    # --- Tegn vertikal linje ---
```

```
    fig.add_shape(
        type="line",
        x0=x, x1=x,
        y0=0, y1=1,
        xref="x", yref="paper",
        line=dict(color=color, dash=dash, width=line_width),
        layer="above"
    )

    # --- Legg til dummy-trace for legend ---
    fig.add_trace(
        go.Scatter(
            x=[None],
            y=[None],
            mode="lines",
            line=dict(color=color, dash=dash, width=line_width),
            name=label,
            hoverinfo="skip",
            showlegend=True
        )
    )
```

In [ ]:

```
def plot_timeseries(
    data: pd.DataFrame,
    title: str,
    y_title: str,
    filename: str = None,
    show: bool = True,
    show_break: bool = False,
    reference_value: float = None,
    margin: float = 0,
    reference_label: str = "± margin"
) -> None:
    """
    Lager en linjegraf for én eller flere tidsserier.

    Parametre:
    ----------
    data : pd.DataFrame
        DataFrame med datetime-indeks og én eller flere kolonner.
    title : str
        Tittel på figuren.
    y_title : str
        Y-akse tittel.
    filename : str, optional
        Filnavn uten filtype (brukes som base for lagring). Hvis None, genereres
    show : bool, optional
        Om figuren skal vises etter lagring. Standard er True.
    show_break : bool, optional
        Om en vertikal bruddlinje skal legges til på BREAK_DATE_DT. Standard er
    reference_value : float, optional
        Senterverdi for horisontale linjer. Hvis None, tegnes ikke linjene.
    margin : float, optional
        Avstand fra senterverdien for de horisontale linjene.
    reference_label : str, optional
        Tekst som vises i legenden for marginlinjene.

    Returnerer:
    -----------
    None
```

```
    """

    # --- Initialiser figur ---
    fig = go.Figure()

    # --- Sjekk om data er tom ---
    if data.empty:
        print("Advarsel: Data er tom – ingen figur genereres.")
        return

    # --- Legg til dataserier ---
    if data.shape[1] == 1:
        col = data.columns[0]
        name = NAME_MAP.get(col, col)
        fig.add_trace(
            go.Scatter(
                x=data.index,
                y=data[col],
                name=name,
                line=dict(color=COLORS[0]),
            )
        )
    else:
        for i, col in enumerate(data.columns):
            name = NAME_MAP.get(col, col)
            fig.add_trace(
                go.Scatter(
                    x=data.index,
                    y=data[col],
                    name=name,
                    line=dict(color=COLORS[i % len(COLORS)]),
                )
            )

    # --- Tegn senterlinje og marginlinjer ---
    if reference_value is not None:
        # Senterlinje (sort, stiplet og tykk)
        fig.add_shape(
            type="line",
            x0=data.index.min(),
            x1=data.index.max(),
            y0=reference_value,
            y1=reference_value,
            line=dict(color="black", width=2, dash="dash"),
            layer="above"
        )

        fig.add_trace(
            go.Scatter(
                x=[None], y=[None],
                mode='lines',
                line=dict(color="black", width=2, dash="dash"),
                name="Senterverdi",
                showlegend=True
            )
        )

        # Marginlinjer (grå) hvis margin > 0
        if margin > 0:
            for offset in [-margin, margin]:
```

```
                y = reference_value + offset
                fig.add_shape(
                    type="line",
                    x0=data.index.min(),
                    x1=data.index.max(),
                    y0=y,
                    y1=y,
                    line=dict(color="dimgray", width=1, dash="dash"),
                    layer="below"
                )

            fig.add_trace(
                go.Scatter(
                    x=[None], y=[None],
                    mode='lines',
                    line=dict(color="dimgray", width=2, dash="dash"),
                    name=reference_label,
                    showlegend=True
                )
            )

    # --- Legg til bruddlinje hvis ønskelig ---
    if show_break:
        add_break_line(fig, BREAK_DATE_DT, BREAK_DATE_LABEL)

    # --- Oppdater layout ---
    fig.update_layout(
        title=title,
        yaxis_title=y_title,
        **common_layout
    )

    # --- Generer filnavn hvis ikke angitt ---
    if filename is None:
        base = title.lower()
        base = base.replace(" ", "_").replace("-", "").replace(".", "").replace(
        filename = f"plot_line_{base}"

    # --- Lagre figur ---
    save_figure(fig, filename)

    # --- Vis figur ---
    if show:
        fig.show()
```

In [ ]:

```
def plot_scatter(
    data: pd.DataFrame,
    title: str,
    y_title: str,
    filename: str = None,
    show: bool = True,
    show_break: bool = False,
    reference_value: float = None,
    margin: float = 0,
    reference_label: str = "± margin"
) -> None:
    """
    Lager et scatter-plot for én eller flere tidsserier.

    Parametre:
```

```
    ----------
    data : pd.DataFrame
        DataFrame med datetime-indeks og én eller flere kolonner.
    title : str
        Tittel på figuren.
    y_title : str
        Y-akse tittel.
    filename : str, optional
        Filnavn uten filtype (brukes som base for lagring). Hvis None, genereres
    show : bool, optional
        Om figuren skal vises etter lagring. Standard er True.
    show_break : bool, optional
        Om dataserien skal deles opp før/etter BREAK_DATE_DT. Standard er False.
    reference_value : float, optional
        Midtpunkt for horisontale hjelpelinjer.
    margin : float, optional
        Avstand fra referanseverdien for hjelpelinjene.
    reference_label : str, optional
        Tekst som vises i legenden for hjelpelinjene.

    Returnerer:
    -----------
    None
    """

    # --- Initialiser figur ---
    fig = go.Figure()

    # --- Sjekk om data er tom ---
    if data.empty:
        print("Advarsel: Data er tom – ingen figur genereres.")
        return

    # --- Legg til dataserier ---
    if show_break and data.shape[1] == 1:
        col = data.columns[0]
        name = NAME_MAP.get(col, col)

        pre_break = data.loc[data.index < BREAK_DATE_DT, col]
        post_break = data.loc[data.index >= BREAK_DATE_DT, col]

        fig.add_trace(
            go.Scatter(
                x=pre_break.index,
                y=pre_break.values,
                name=f"{name} (før)",
                mode="markers",
                marker=dict(color=COLORS[0]),
            )
        )
        fig.add_trace(
            go.Scatter(
                x=post_break.index,
                y=post_break.values,
                name=f"{name} (etter)",
                mode="markers",
                marker=dict(color=COLORS[1]),
            )
        )
    else:
```

```
        for i, col in enumerate(data.columns):
            name = NAME_MAP.get(col, col)
            fig.add_trace(
                go.Scatter(
                    x=data.index,
                    y=data[col],
                    name=name,
                    mode="markers",
                    marker=dict(color=COLORS[i % len(COLORS)]),
                )
            )

    # --- Legg til horisontale hjelpelinjer ---
    if reference_value is not None:
        for offset in [-margin, margin]:
            y = reference_value + offset
            fig.add_shape(
                type="line",
                x0=data.index.min(),
                x1=data.index.max(),
                y0=y,
                y1=y,
                line=dict(color="gray", width=1, dash="dash"),
                layer="below"
            )

        fig.add_trace(
            go.Scatter(
                x=[None],
                y=[None],
                mode="lines",
                line=dict(color="gray", width=1, dash="dash"),
                name=reference_label,
                showlegend=True
            )
        )

    # --- Legg til bruddlinje hvis ønskelig ---
    if show_break:
        add_break_line(fig, BREAK_DATE_DT, BREAK_DATE_LABEL)

    # --- Oppdater layout ---
    fig.update_layout(
        title=title,
        yaxis_title=y_title,
        **common_layout
    )

    # --- Generer filnavn hvis ikke angitt ---
    if filename is None:
        base = title.lower()
        base = base.replace(" ", "_").replace("-", "").replace(".", "").replace(
        filename = f"plot_scatter_{base}"

    # --- Lagre figur ---
    save_figure(fig, filename)

    # --- Vis figur ---
    if show:
        fig.show()
```

In [ ]:

```python
def plot_rolling_average(
    data: pd.DataFrame,
    title: str,
    y_title: str,
    filename: str = None,
    window: int = ROLLING_WINDOW,
    show: bool = True,
    show_break: bool = True,
    reference_value: float = None,
    margin: float = 0,
    reference_label: str = "± margin"
) -> None:
    """
    Plotter glidende gjennomsnitt av én eller flere tidsserier.

    Parametre:
    ----------
    data : pd.DataFrame
        DataFrame med datetime-indeks og én eller flere kolonner.
    title : str
        Tittel på figuren.
    y_title : str
        Y-akse tittel.
    filename : str, optional
        Filnavn uten filtype (brukes som base for lagring). Hvis None, genereres
    window : int, optional
        Lengde på det glidende vinduet. Standard er ROLLING_WINDOW.
    show : bool, optional
        Om figuren skal vises etter lagring. Standard er True.
    show_break : bool, optional
        Om en bruddlinje skal legges til. Standard er True.
    reference_value : float, optional
        Midtpunkt for horisontale hjelpelinjer.
    margin : float, optional
        Avstand fra referanseverdien for hjelpelinjene.
    reference_label : str, optional
        Tekst som vises i legenden for hjelpelinjene.

    Returnerer:
    -----------
    None
    """

    # --- Sjekk om data er tom ---
    if data.empty:
        print("Advarsel: Data er tom – glidende gjennomsnitt ikke generert.")
        return

    # --- Beregn glidende gjennomsnitt ---
    rolling_data = data.rolling(window=window, min_periods=1).mean()

    # --- Generer filnavn hvis ikke angitt ---
    if filename is None:
        base = title.lower()
        base = base.replace(" ", "_").replace("-", "").replace(".", "").replace(
        filename = f"plot_rolling_{base}"

    # --- Plot glidende gjennomsnitt ---
    plot_timeseries(
```

```
        rolling_data,
        title=title,
        y_title=y_title,
        filename=filename,
        show=show,
        show_break=show_break,
        reference_value=reference_value,
        margin=margin,
        reference_label=reference_label
    )
```

In [ ]:

```
def plot_histogram(
    data: pd.DataFrame,
    title: str,
    x_title: str,
    filename: str = None,
    show: bool = True,
    bins: int = 40,
    reference_value: float = None,
    margin: float = 0,
    reference_label: str = "± margin"
) -> None:
    """
    Lager histogram og KDE for én eller flere variabler.

    Parametre:
    ----------
    data : pd.DataFrame
        DataFrame med én eller flere kolonner.
    title : str
        Tittel på figuren.
    x_title : str
        X-akse tittel.
    filename : str, optional
        Filnavn uten filtype (brukes som base for lagring). Hvis None, genereres
    show : bool, optional
        Om figuren skal vises etter lagring. Standard er True.
    bins : int, optional
        Antall søyler i histogrammet. Standard er 40.
    reference_value : float, optional
        Midtpunkt for vertikale hjelpelinjer.
    margin : float, optional
        Avstand fra referanseverdien for hjelpelinjene.
    reference_label : str, optional
        Tekst som vises i legenden for hjelpelinjene.

    Returnerer:
    -----------
    None
    """

    # --- Initialiser figur ---
    fig = go.Figure()

    # --- Sjekk om data er tom ---
    if data.empty:
        print("Advarsel: Data er tom – ingen histogram genereres.")
        return

    # --- Samle all data for x-akse range ---
```

```
    combined = data.values.flatten()
    mean_comb = np.mean(combined)
    std_comb = np.std(combined)
    x_min, x_max = mean_comb - 3 * std_comb, mean_comb + 3 * std_comb

    # --- Definer x-akse for KDE ---
    x_range = np.linspace(x_min, x_max, 500)

    # --- Legg til histogramspor og KDE-linjer ---
    for i, col in enumerate(data.columns):
        series = data[col].dropna()
        name = NAME_MAP.get(col, col)
        color = COLORS[i % len(COLORS)]

        # Histogram
        fig.add_trace(
            go.Histogram(
                x=series,
                name=name,
                marker=dict(color=color),
                opacity=0.6,
                nbinsx=bins,
                histnorm="probability density"
            )
        )

        # KDE
        kde = gaussian_kde(series)
        mean = series.mean()
        fig.add_trace(
            go.Scatter(
                x=x_range,
                y=kde(x_range),
                mode="lines",
                name=f"KDE {name} (μ = {mean:.2f})",
                line=dict(color=color)
            )
        )

        # Gjennomsnittslinje
        fig.add_vline(
            x=mean,
            line=dict(color=color, dash="dash")
        )

    # --- Legg til vertikale hjelpelinjer ± margin ---
    if reference_value is not None:
        for offset in [-margin, margin]:
            x = reference_value + offset
            fig.add_vline(
                x=x,
                line=dict(color="gray", width=1, dash="dash"),
                layer="below"
            )

        fig.add_trace(
            go.Scatter(
                x=[None],
                y=[None],
                mode="lines",
```

```
                line=dict(color="gray", width=1, dash="dash"),
                name=reference_label,
                showlegend=True
            )
        )

    # --- Oppdater layout ---
    layout = common_layout.copy()
    layout.update({
        "title": title,
        "xaxis_title": x_title,
        "yaxis_title": "Tetthet",
        "barmode": "overlay",
        "xaxis": {**common_layout.get("xaxis", {}), "range": [x_min, x_max]},
        "legend": dict(
            orientation="h",
            x=0.5,
            y=-0.3,
            xanchor="center",
            yanchor="top",
            font=dict(size=13)
        ),
        "margin": dict(l=80, r=40, t=80, b=120)
    })
    fig.update_layout(**layout)

    # --- Generer filnavn hvis ikke angitt ---
    if filename is None:
        base = title.lower()
        base = base.replace(" ", "_").replace("-", "").replace(".", "").replace(
        filename = f"plot_histogram_{base}"

    # --- Lagre figur ---
    save_figure(fig, filename)

    # --- Vis figur ---
    if show:
        fig.show()
```

In [ ]:

```
def plot_histogram_comparison(
    series1: pd.Series,
    series2: pd.Series,
    label1: str,
    label2: str,
    title: str,
    xlabel: str,
    filename: str = None,
    bins: int = 40,
    show: bool = True,
    reference_value: float = None,
    margin: float = 0,
    reference_label: str = "± margin"
) -> None:
    """
    Plotter to distribusjoner med histogram, KDE og gjennomsnittslinjer.
    Legenden viser gjennomsnitt og plasseres under plottet.

    Parametre:
    ----------
    series1, series2 : pd.Series
```

```
    Pandas Series med verdier.
label1, label2 : str
    Navn for dataseriene.
title : str
    Tittel på figuren.
xlabel : str
    Navn på x-aksen.
filename : str, optional
    Filnavn uten filtype (brukes som base for lagring). Hvis None, genereres
bins : int, optional
    Antall søyler i histogrammet. Standard er 40.
show : bool, optional
    Om figuren skal vises etter lagring. Standard er True.
reference_value : float, optional
    Midtpunkt for vertikale hjelpelinjer.
margin : float, optional
    Avstand fra referanseverdien for hjelpelinjene.
reference_label : str, optional
    Tekst som vises i legenden for hjelpelinjene.

Returnerer:
-----------
None
"""

# --- Bruk mapping på etiketter ---
label1_mapped = NAME_MAP.get(label1, label1)
label2_mapped = NAME_MAP.get(label2, label2)

# --- Beregn statistikk ---
mean1, mean2 = series1.mean(), series2.mean()
kde1, kde2 = gaussian_kde(series1), gaussian_kde(series2)

# --- Definer x-akse for KDE ---
x_range = np.linspace(
    min(series1.min(), series2.min()),
    max(series1.max(), series2.max()),
    500
)

# --- Definer visningsområde (±3 std) ---
combined = np.concatenate([series1, series2])
mean_comb = combined.mean()
std_comb = combined.std()
x_min, x_max = mean_comb - 3 * std_comb, mean_comb + 3 * std_comb

# --- Initialiser figur ---
fig = go.Figure()

# --- Legg til histogrammer ---
for series, label, color in [
    (series1, label1_mapped, COLOR_1),
    (series2, label2_mapped, COLOR_2)
]:
    fig.add_trace(
        go.Histogram(
            x=series,
            name=label,
            marker_color=color,
            opacity=0.6,
```

```
            nbinsx=bins,
            histnorm="probability density"
        )
    )

# --- Legg til KDE-linjer ---
for kde, mean, label, color in [
    (kde1, mean1, label1_mapped, COLOR_1),
    (kde2, mean2, label2_mapped, COLOR_2)
]:
    fig.add_trace(
        go.Scatter(
            x=x_range,
            y=kde(x_range),
            mode="lines",
            name=f"KDE {label} (μ = {mean:.2f})",
            line=dict(color=color)
        )
    )

# --- Legg til vertikale gjennomsnittslinjer ---
for mean, color in [(mean1, COLOR_1), (mean2, COLOR_2)]:
    fig.add_vline(
        x=mean,
        line=dict(color=color, dash="dash")
    )

# --- Legg til vertikale hjelpelinjer ± margin ---
if reference_value is not None:
    for offset in [-margin, margin]:
        x = reference_value + offset
        fig.add_vline(
            x=x,
            line=dict(color="gray", width=1, dash="dash"),
            layer="below"
        )

    fig.add_trace(
        go.Scatter(
            x=[None],
            y=[None],
            mode="lines",
            line=dict(color="gray", width=1, dash="dash"),
            name=reference_label,
            showlegend=True
        )
    )

# --- Sett opp layout ---
layout = common_layout.copy()
layout.update({
    "title": title,
    "xaxis_title": xlabel,
    "yaxis_title": "Tetthet",
    "barmode": "overlay",
    "xaxis": {**common_layout["xaxis"], "range": [x_min, x_max]},
    "legend": dict(
        orientation="h",
        x=0.5,
        y=-0.3,
```

```
            xanchor="center",
            yanchor="top",
            font=dict(size=13)
        ),
        "margin": dict(l=80, r=40, t=80, b=120)
    })

    fig.update_layout(**layout)

    # --- Generer filnavn hvis ikke angitt ---
    if filename is None:
        base = title.lower()
        base = base.replace(" ", "_").replace("-", "").replace(".", "").replace(
        filename = f"plot_histogram_comparison_{base}"

    # --- Lagre figur ---
    save_figure(fig, filename)

    # --- Vis figur ---
    if show:
        fig.show()
```

In [ ]:

```
def plot_qq(
    data: pd.Series,
    color: str,
    title: str,
    filename: str = None,
    show: bool = True
) -> None:
    """
    Lager QQ-plott mot normalfordeling.

    Parametre:
    ----------
    data : pd.Series
        Pandas Series med data som skal sammenlignes med normalfordeling.
    color : str
        Farge på datapunktene.
    title : str
        Tittel på figuren.
    filename : str, optional
        Filnavn uten filtype (brukes som base for lagring). Hvis None, genereres
    show : bool, optional
        Om figuren skal vises etter lagring. Standard er True.

    Returnerer:
    -----------
    None
    """

    # --- Sjekk om data er tom ---
    if data.empty:
        print("Advarsel: Data er tom – QQ-plot ikke generert.")
        return

    # --- Beregn teoretiske og observerte kvantiler ---
    (osm, osr), (slope, intercept, _) = probplot(data, dist="norm")
    line_x = np.array([osm.min(), osm.max()])
    line_y = slope * line_x + intercept
```

```
    # --- Initialiser figur ---
    fig = go.Figure()

    # --- Legg til datapunkter ---
    fig.add_trace(
        go.Scatter(
            x=osm,
            y=osr,
            mode="markers",
            name="Observasjoner",
            marker=dict(color=color)
        )
    )

    # --- Legg til referanselinje ---
    fig.add_trace(
        go.Scatter(
            x=line_x,
            y=line_y,
            mode="lines",
            name="Normal linje",
            line=dict(color="black", dash="dash")
        )
    )

    # --- Oppdater layout ---
    fig.update_layout(
        title=title,
        xaxis_title="Teoretiske kvantiler",
        yaxis_title="Observerte verdier",
        showlegend=False,
        **common_layout
    )

    # --- Generer filnavn hvis ikke angitt ---
    if filename is None:
        base = title.lower().replace(" ", "_").replace("-", "").replace(".", "")
        filename = f"plot_qq_{base}"

    # --- Lagre figur ---
    save_figure(fig, filename)

    # --- Vis figur ---
    if show:
        fig.show()
```

In [ ]:

```
def plot_acf_pacf(
    series: pd.Series,
    title_prefix: str = "",
    lags: int = 20,
    show: bool = True
) -> None:
    """
    Lager ACF- og PACF-plot for en gitt tidsserie, med 95 % konfidensintervall.

    Parametre:
    ----------
    series : pd.Series
        Tidsserie som skal analyseres.
    title_prefix : str, optional
```

```
    Beskrivende tekst som brukes i figurens tittel og filnavn, f.eks. "strøm
lags : int, optional
    Antall lag som skal beregnes. Standard er 20.
show : bool, optional
    Om figurene skal vises etter lagring. Standard er True.

Returnerer:
-----------
None
"""

# --- Beregn konfidensintervall ---
n = len(series.dropna())
conf_int = 1.96 / np.sqrt(n)

# --- Beregn ACF og PACF ---
acf_vals = acf(series, nlags=lags)
pacf_vals = pacf(series, nlags=lags)
x_vals = list(range(len(acf_vals)))

# --- Definer annotasjon ---
annotation_text = "Streken viser 95 % konfidensintervall for nullhypotesen (
annotation = dict(
    text=annotation_text,
    xref="paper",
    yref="paper",
    x=0,
    y=-0.20,
    showarrow=False,
    font=dict(size=12, color="gray"),
    align="left"
)

# --- Rens tittel for bruk i filnavn ---
base = title_prefix.lower()
base = base.replace(" ", "_").replace("-", "").replace(".", "").replace("–",

# --- Lag ACF-figur ---
acf_fig = go.Figure()
acf_fig.add_trace(go.Bar(x=x_vals, y=acf_vals, name="ACF"))
acf_fig.add_shape(
    type="rect",
    x0=-0.5, x1=lags + 0.5,
    y0=-conf_int, y1=conf_int,
    fillcolor="lightblue",
    opacity=0.3,
    layer="below",
    line_width=0
)
acf_fig.add_hline(y=conf_int, line=dict(dash="dash", color="blue"), opacity=
acf_fig.add_hline(y=-conf_int, line=dict(dash="dash", color="blue"), opacity
acf_fig.update_layout(
    title=f"ACF for {title_prefix}",
    xaxis_title="Lag",
    yaxis_title="Autokorrelasjon",
    annotations=[annotation],
    **common_layout
)

# --- Lag PACF-figur ---
```

```
    pacf_fig = go.Figure()
    pacf_fig.add_trace(go.Bar(x=x_vals, y=pacf_vals, name="PACF"))
    pacf_fig.add_shape(
        type="rect",
        x0=-0.5, x1=lags + 0.5,
        y0=-conf_int, y1=conf_int,
        fillcolor="lightblue",
        opacity=0.3,
        layer="below",
        line_width=0
    )
    pacf_fig.add_hline(y=conf_int, line=dict(dash="dash", color="blue"), opacity
    pacf_fig.add_hline(y=-conf_int, line=dict(dash="dash", color="blue"), opacit
    pacf_fig.update_layout(
        title=f"PACF for {title_prefix}",
        xaxis_title="Lag",
        yaxis_title="Partial autokorrelasjon",
        annotations=[annotation],
        **common_layout
    )

    # --- Lagre figurer som HTML ---
    save_figure(acf_fig, f"plot_acf_{base}")
    save_figure(pacf_fig, f"plot_pacf_{base}")

    # --- Vis figurer ---
    if show:
        acf_fig.show()
        pacf_fig.show()
```

In [ ]:

```
def save_figure(fig: go.Figure, filename_base: str) -> None:
    """
    Lagrer en Plotly-figur som HTML i en organisert undermappe.

    Parametre:
    ----------
    fig : go.Figure
        Plotly-figur som skal lagres.
    filename_base : str
        Filnavn uten filtype (brukes som base for lagrede filer).

    Returnerer:
    -----------
    None
    """

    # --- Sørg for at base-navnet ikke har filtype ---
    filename_base = os.path.splitext(filename_base)[0]

    # --- Definer format og sti ---
    fmt = "html"
    subfolder = os.path.join(OUTPUT_DIR, fmt)
    path = os.path.join(subfolder, f"{filename_base}.{fmt}")

    # --- Opprett mappe hvis den ikke finnes ---
    os.makedirs(subfolder, exist_ok=True)

    # --- Trygg lagring ---
    try:
        fig.write_html(path)
```

```
        # print(f"Lagret figur: {path}")
    except Exception as e:
        print(f"Kunne ikke lagre HTML for {filename_base}: {e}")

    # --- Registrer figur i oversikt (hvis støttet) ---
    try:
        add_figure(filename_base, fig)
    except NameError:
        pass
```

In [ ]:

```
def save_to_excel(
    excel_path: Path = Path("output/excel/data_series.xlsx"),
    **kwargs
) -> None:
    """
    Lagrer ett eller flere objekter til en Excel-fil.

    - DataFrames lagres på egne arkfaner med variabelnavn som arknavn.
    - Andre objekter lagres i et sammendrag i et ark kalt "variables".

    Parametre:
    ----------
    excel_path : Path, optional
        Filsti til Excel-filen. Standard er "output/excel/data_series.xlsx".
    kwargs : key-value
        Navn og objekter som skal lagres.

    Returnerer:
    -----------
    None
    """

    # --- Sørg for at mappe eksisterer ---
    excel_path.parent.mkdir(parents=True, exist_ok=True)

    # --- Opprett fil om den ikke eksisterer ---
    if not excel_path.exists():
        with pd.ExcelWriter(excel_path, engine="openpyxl") as writer:
            pd.DataFrame([["Midlertidig ark, kan slettes."]]).to_excel(writer, s

    # --- Registrer eksisterende ark ---
    existing_sheets = set()
    if excel_path.exists():
        wb = load_workbook(excel_path)
        existing_sheets = set(wb.sheetnames)

    variables_info = {}

    # --- Lagre objektene ---
    with pd.ExcelWriter(excel_path, mode="a", engine="openpyxl", if_sheet_exists
        for var_name, obj in kwargs.items():
            sheet_name = var_name[:31]  # Excel-begrensning på arkfanenavn

            if isinstance(obj, pd.DataFrame):
                try:
                    obj.to_excel(writer, sheet_name=sheet_name, index=False)
                except Exception as e:
                    print(f"Kunne ikke lagre DataFrame '{var_name}': {e}")
            else:
                # Lag et sammendrag for ikke-DataFrame objekter
```

```
                summary = str(obj)[:100]
                variables_info[var_name] = {
                    "Variable": var_name,
                    "Type": type(obj).__name__,
                    "Value": summary
                }

        # --- Lagre sammendrag dersom andre objekter finnes ---
        if variables_info:
            var_df = pd.DataFrame(variables_info.values())
            try:
                var_df.to_excel(writer, sheet_name="variables", index=False)
            except Exception as e:
                print(f"Kunne ikke lagre 'variables'-arket: {e}")

    # --- Ferdig ---
    print(f"Alt er lagret i Excel: {excel_path}")
```

In [ ]:
```
def save_garch_summaries_txt(
    garch_models: dict,
    txt_path: str = "output/txt/summary_garch_models.txt"
) -> None:
    """
    Lagrer sammendrag fra GARCH-modeller til en tekstfil.

    Parametre:
    ----------
    garch_models : dict
        Ordbok med nøkler (str) og verdier (fitted GARCH-modeller med .summary()
    txt_path : str, optional
        Filsti for lagring. Standard er "output/txt/summary_garch_models.txt".

    Returnerer:
    -----------
    None
    """

    try:
        os.makedirs(os.path.dirname(txt_path), exist_ok=True)

        with open(txt_path, "w", encoding="utf-8") as f:
            for key, model in garch_models.items():
                f.write(f"{'=' * 40}\n")
                f.write(f"GARCH Model for: {key}\n")
                f.write(f"{'-' * 40}\n")
                f.write(model.summary().as_text())
                f.write("\n\n")

        print(f"GARCH-sammendrag lagret til: {txt_path}")

    except Exception as e:
        print(f"Kunne ikke lagre GARCH-sammendrag: {e}")
```

In [ ]:
```
def export_garch_results_to_excel(
    results_dict: dict,
    filename: str = "summary_garch_results.xlsx"
) -> None:
    """
    Eksporterer GARCH-modellresultater til en Excel-fil med formatert tabell på
```

```
    Parametre:
    ----------
    results_dict : dict
        Dictionary med modellresultater, f.eks. {"GER": result1, "NO2": result2}
    filename : str, optional
        Navn på Excel-filen. Standard er "summary_garch_results.xlsx".

    Returnerer:
    -----------
    None
    """

    # --- Definer filsti ---
    file_path = EXCEL_DIR / "garch" / filename
    os.makedirs(file_path.parent, exist_ok=True)


    summary_rows = {}
    temp_writer_data = {}

    # --- Behandle hver modell ---
    for code, result in results_dict.items():
        country = NAME_MAP.get(code, code)
        sheet_name = f"EGARCH-modell for {country}"

        # Hent modellresultater
        params = result.params
        stderr = result.std_err
        tvals = result.tvalues
        pvals = result.pvalues
        conf_int = result.conf_int()

        # Formater rader for Excel
        formatted_rows = []
        for param in params.index:
            # Kategoriser parameter
            if param == "mu":
                section = "Gjennomsnittsmodell"
            elif param.startswith("nu"):
                section = "Distribusjon"
            else:
                section = "Volatilitetsmodell"

            coef = params[param]
            se = stderr[param]
            tval = tvals[param]
            pval = pvals[param]
            ci_low, ci_high = conf_int.loc[param]

            formatted_rows.append([
                section,
                param,
                f"{coef:.4f}",
                f"{se:.4f}",
                f"{tval:.4f}",
                f"{pval:.4f}",
                f"[{ci_low:.4f}, {ci_high:.4f}]"
            ])
```

```
        df_formatted = pd.DataFrame(
            formatted_rows,
            columns=[
                "Modellkomponent", "Parameter", "Estimat",
                "Standardfeil", "t-verdi", "p-verdi", "95 % konfidensintervall"
            ]
        )

        temp_writer_data[sheet_name[:31]] = df_formatted

        # Legg til informasjon for sammendrag
        model = result.model
        summary_rows[code] = {
            "Land": country,
            "Volatilitetsmodell": model.volatility.__class__.__name__,
            "Distribusjon": model.distribution.name,
            "AIC": round(result.aic, 4),
            "BIC": round(result.bic, 4),
            "Log-likelihood": round(result.loglikelihood, 4)
        }

    # --- Opprett sammendrag DataFrame ---
    df_summary = pd.DataFrame(summary_rows.values())
    df_summary = df_summary[[
        "Land", "Volatilitetsmodell", "Distribusjon", "AIC", "BIC", "Log-likelih
    ]]

    # --- Skriv til Excel ---
    with pd.ExcelWriter(file_path, engine="openpyxl", mode="w") as writer:
        for sheet, df in temp_writer_data.items():
            df.to_excel(writer, sheet_name=sheet, index=False)
        df_summary.to_excel(writer, sheet_name="GARCH-sammendrag", index=False)

    # --- Lagret til filbane ---
    print(f"GARCH-resultater lagret som faner i: {file_path.resolve()}")
```

In [ ]:

```
def test_egarch_variants(
    series: pd.Series,
    series_name: str = "serie",
    distributions: list = None,
    p_q_combos: list = None,
    save: bool = False,
    filename: str = None
) -> pd.DataFrame:
    """
    Estimerer EGARCH-modeller for en gitt serie over ulike (p, q)-kombinasjoner
    og utfører ARCH-test på standardiserte residualer.

    Parametre:
    ----------
    series : pd.Series
        Stasjonær inputserie.
    series_name : str
        Navn som brukes i filnavn dersom resultater lagres.
    distributions : list, optional
        Liste over fordelinger som skal testes. Default = ["t"].
    p_q_combos : list, optional
        Liste over (p, q)-kombinasjoner. Default = [(1,1)-(3,3)].
    save : bool, optional
        Om resultatene skal lagres til Excel. Default = False.
```

```
    filename : str, optional
        Filnavn for eksport. Hvis None, genereres automatisk.

    Returnerer:
    -----------
    pd.DataFrame
        Resultattabell med p, q, fordeling, AIC, BIC, log-likelihood og ARCH-tes
    """

    # --- Sett standardverdier ---
    if distributions is None:
        distributions = ["t"]
    if p_q_combos is None:
        p_q_combos = [(p, q) for p in range(1, 4) for q in range(1, 4)]

    series = series.dropna()

    results = []
    failed_models = []

    for p, q in p_q_combos:
        for dist in distributions:
            try:
                model = arch_model(series, vol="EGARCH", p=p, q=q, dist=dist)
                res = model.fit(disp="off")

                # Standardiserte residualer
                standardized = res.std_resid.dropna()

                # ARCH-test etter modellering
                arch_stat_post, arch_pval_post, *_ = het_arch(standardized)

                results.append({
                    "p": p,
                    "q": q,
                    "dist": dist,
                    "loglikelihood": res.loglikelihood,
                    "aic": res.aic,
                    "bic": res.bic,
                    "arch_stat": arch_stat_post,
                    "arch_pval": arch_pval_post
                })
            except Exception as e:
                failed_models.append((p, q, dist, str(e)))

    # --- Logg eventuelle feil ---
    if failed_models:
        print("\nFølgende modellkombinasjoner feilet:")
        for p, q, dist, msg in failed_models:
            print(f"  EGARCH({p},{q}) med fordeling '{dist}': {msg}")

    # --- Returner resultater ---
    df = pd.DataFrame(results)

    # Forbedrede kolonnenavn for Excel
    df = df.rename(columns={
        "dist": "Fordeling",
        "loglikelihood": "Log-likelihood",
        "aic": "AIC",
        "bic": "BIC",
```

```
        "arch_stat": "LM-statistikk",
        "arch_pval": "LM p-verdi"
    })

    if save and not df.empty:
        safe_name = series_name.lower().replace(" ", "_").replace("-", "").repla
        if filename is None:
            filename = f"results_egarch_gridsearch_{safe_name}.xlsx"
        file_path = EXCEL_DIR / "garch" / filename
        os.makedirs(file_path.parent, exist_ok=True)
        df.to_excel(file_path, index=False)
        print(f"Resultater lagret til: {file_path.resolve()}")

    return df
```

In [ ]:

```
def load_daily_prices(
    years: list[int],
    input_dir: Path,
    columns_to_keep: list[str]
) -> pd.DataFrame:
    """
    Leser inn og samler daglige prisdata fra CSV-filer for spesifiserte år.

    Parametre:
    ----------
    years : list[int]
        Årstall som skal leses inn (f.eks. [2020, 2021, 2022]).
    input_dir : Path
        Mappe som inneholder CSV-filene.
    columns_to_keep : list[str]
        Kolonner som skal beholdes (inkl. "Delivery Date CET").

    Returnerer:
    -----------
    pd.DataFrame
        Kombinert DataFrame med dato som indeks og sortert kronologisk.
    """

    dataframes = []
    missing_years = []

    # --- Gå gjennom alle spesifiserte år ---
    for year in years:
        file_path = input_dir / f"daily_aggregate_{year}.csv"

        if file_path.exists():
            df = pd.read_csv(
                file_path,
                delimiter=";",
                decimal=",",
                thousands="."
            )
            df["Delivery Date CET"] = pd.to_datetime(df["Delivery Date CET"])
            dataframes.append(df)
        else:
            missing_years.append(year)

    # --- Feilhåndtering ved manglende filer ---
    if not dataframes:
        raise FileNotFoundError("Ingen CSV-filer ble funnet i input-mappen.")
```

```
    if missing_years:
        print(f"Følgende år manglet filer og ble hoppet over: {missing_years}")

    # --- Slå sammen filer ---
    combined = pd.concat(dataframes, ignore_index=True)

    # --- Rens kolonnenavn og behold ønskede kolonner ---
    combined.columns = [col.replace(" (EUR)", "") for col in combined.columns]
    combined = combined[columns_to_keep].dropna()

    # --- Sett dato som indeks og sorter ---
    combined.set_index("Delivery Date CET", inplace=True)
    combined.sort_index(inplace=True)

    return combined
```

In [ ]:
```
# --- Velg hvilke kolonner som skal beholdes fra datakilden ---
columns_to_keep = ["Delivery Date CET", "GER", "NO2"]

# --- Les inn og kombiner daglige prisdata for spesifiserte år ---
daily_prices = load_daily_prices(
    years=YEARS,
    input_dir=INPUT_DIR,
    columns_to_keep=columns_to_keep
)
```

In [ ]:
```
def descriptive_analysis(
    data: pd.DataFrame,
    filnavn: str = "output/excel/descriptive_statistics.xlsx",
    prefix: str = None,
    break_date: pd.Timestamp = pd.Timestamp("2022-02-24")
) -> None:
    """
    Utfører deskriptiv analyse og lagrer resultatene i én Excel-fane,
    med separate overskrifter for hele perioden, før og etter bruddet.

    Parametre:
    ----------
    data : pd.DataFrame
        Inndata med datetime-indeks.
    filnavn : str, optional
        Sti til Excel-filen hvor resultatene skal lagres.
    prefix : str, optional
        Prefix for arknavnet. Hvis None, brukes variabelnavnet automatisk.
    break_date : pd.Timestamp, optional
        Dato for bruddpunktet. Standard er 2022-02-24.

    Returnerer:
    -----------
    None
    """

    # --- Foreslå prefix basert på variabelnavn hvis ikke oppgitt ---
    if prefix is None:
        callers_local_vars = inspect.currentframe().f_back.f_locals.items()
        prefix = next((name for name, val in callers_local_vars if val is data),
        prefix += "_stats"
```

```python
# --- Del opp data i tre perioder ---
parts = {
    "HELE PERIODEN": data,
    "FØR INVASJONEN": data[data.index < break_date],
    "ETTER INVASJONEN": data[data.index >= break_date],
}

output_path = Path(filnavn)
output_path.parent.mkdir(parents=True, exist_ok=True)

# --- Lag tom Excel-fil om den ikke finnes ---
if not output_path.exists():
    with pd.ExcelWriter(output_path, engine="openpyxl") as writer:
        pd.DataFrame([["Midlertidig ark"]]).to_excel(writer, sheet_name="tem

arkfane = prefix[:31]  # Maks 31 tegn
startrow = 0

# --- Skriv statistikk til Excel ---
with pd.ExcelWriter(output_path, mode="a", engine="openpyxl", if_sheet_exist
    for delnavn, subset in parts.items():
        rows = []

        for col in subset.columns:
            serie = subset[col].dropna()
            if serie.empty:
                continue

            rows.append({
                "Serie": col,
                "Minimum": serie.min(),
                "1. kvartil": serie.quantile(0.25),
                "Median": serie.median(),
                "3. kvartil": serie.quantile(0.75),
                "Maksimum": serie.max(),
                "Gjennomsnitt": serie.mean(),
                "Standardavvik": serie.std(),
                "Skjevhet": skew(serie, bias=False),
                "Kurtosis": kurtosis(serie, fisher=True, bias=False),
            })

            print(f"- Serie: {col:<10} | Periode: {delnavn:<13} | Antall obs

        df_stats = pd.DataFrame(rows).round(2)

        # Skriv deloverskrift
        pd.DataFrame([[delnavn]]).to_excel(
            writer, sheet_name=arkfane, startrow=startrow, startcol=0, index
        )
        # Skriv tabell
        df_stats.to_excel(
            writer, sheet_name=arkfane, startrow=startrow, startcol=1, index
        )

        startrow += len(df_stats) + 3

# --- Formater resultatarket ---
wb = load_workbook(output_path)
ws = wb[arkfane]
bold_font = Font(bold=True)
```

```
    for row in ws.iter_rows(min_row=1, max_row=startrow):
        if row[0].value in parts:
            row[0].font = bold_font

    # Juster kolonnebredder
    for col in ws.columns:
        max_length = max((len(str(cell.value)) for cell in col if cell.value is
        col_letter = col[0].column_letter
        ws.column_dimensions[col_letter].width = max_length + 2

    # --- Lagre ---
    wb.save(output_path)

    print(f"\nDeskriptiv analyse er lagret i én fane: '{arkfane}' i filen '{filn
```

DCC-modellen: Fra teori til kode

Vi viser her hvordan variablene i DCC-modellen (slik den er definert i V-Lab-dokumentasjonen) samsvarer med variablene i vårt eget datasett og kode.

📄 Teori og dokumentasjon:
https://vlab.stern.nyu.edu/docs/correlation/GARCH-DCC

Differensiert serie brukt i modellen

Differansen Δs_t brukes som erstatning for log-avkastning:

$$\Delta s_t = s_t - s_{t-1}$$

Kode:

`transformed_diff = transformed_prices.diff().dropna()`
Dette tilsvarer modellen:

$$r_t = \mu_t + \varepsilon_t$$

Hvor $\mu_t \approx 0$, og ε_t estimeres videre med GARCH.

EGARCH-estimering per serie

For hver tidsserie i, estimeres residualer og betinget volatilitet:

$$\varepsilon_{i,t} \sim \text{EGARCH}(1,1)$$

Kode:

```
res = arch_model(...).fit()
res.resid # → ε_{i,t}
res.conditional_volatility # → √{h_{i,t}}
```

Standardiserte residualer

Standardisering gir vektor z_t, som er input til DCC-modellen:

$$z_{i,t} = \frac{\varepsilon_{i,t}}{\sqrt{h_{i,t}}}$$

Kode:

```
standardized_resid[col] = res.resid / res.conditional_volatility
```

Estimering av DCC-parametere

Parametrene α og β estimeres ved å minimere DCC log-likelihood loss:

$$\min_{\alpha,\beta} \sum_t \left(\log\det R_t + z_t^\top R_t^{-1} z_t \right)$$

Kode:

```
opt_result = minimize(dcc_loss, ...)
alpha, beta = opt_result.x
```

Dynamisk kovarians

Den dynamiske kovariansmatrisen Q_t beskriver samvariasjonen mellom de standardiserte residualene over tid. Den beregnes rekursivt som:

$$Q_t = (1 - \alpha - \beta)\bar{Q} + \alpha z_{t-1} z_{t-1}^\top + \beta Q_{t-1}$$

hvor:

α, β er estimert via optimering

$\bar{Q}$ er gjennomsnittlig kovariansmatrise for residualene:

$$\bar{Q} = \frac{1}{T} \sum_{t=1}^{T} z_t z_t^\top$$

Kode:

```
Q_bar = np.cov(standardized_resid.T)
Q_list = [...]      # alle Q_t
```

Dynamisk korrelasjonsmatrise

For å få en gyldig korrelasjonsmatrise, normaliserer vi Q_t til R_t slik:

$$R_t = D_t^{-1} Q_t D_t^{-1}$$

$$D_t = \text{diag}\left(\sqrt{Q_{11,t}}, \sqrt{Q_{22,t}}, \ldots, \sqrt{Q_{nn,t}}\right)$$

Kode:

```
R_list = [...]      # alle R_t
```

Validering og diagnostikk

Loss over tid og samlet DCC-loss benyttes til evaluering:

- **Total loss**:

$$\sum_t \left(\log\det R_t + z_t^\top R_t^{-1} z_t\right)$$

- **Loss per tidssteg** gir innsikt i modellens svakheter i tid.

Kode:

```
loss_values = [...]
total_loss = sum(loss_values)
```

Variabeloversikt

Teoretisk symbol	Kodevariabel
s_t	`transformed_prices`
Δs_t	`transformed_diff`
$\varepsilon_{i,t}$	`res.resid`
$\sqrt{h_{i,t}}$	`res.conditional_volatility`
$z_{i,t}$	`standardized_resid`
Q_t	
,	`Q_list` , `Q_bar`
$\bar{Q}$	

Teoretisk symbol	Kodevariabel
R_t	`R_list` , `R_array`
α, β	`alpha` , `beta` (fra `opt_result`)
DCC log-likelihood loss	`loss_values` , `total_loss`

In [ ]:
```python
# --- Beregn daglige endringer i prisene (tillater null og negative verdier) ---
daily_prices_diff = daily_prices.diff().dropna()

# --- Estimer modeller for GER og NO2 ---
p_q_combos = [(p, q) for p in range(1, 4) for q in range(1, 4)]
results_ger = test_egarch_variants(daily_prices_diff["GER"], series_name="GER",
results_ger["serie"] = "GER"

results_no2 = test_egarch_variants(daily_prices_diff["NO2"], series_name="NO2",
results_no2["serie"] = "NO2"

# --- Kombiner og sorter etter laveste AIC ---
df_all = pd.concat([results_ger, results_no2], ignore_index=True)
df_all_sorted = df_all.sort_values("AIC").reset_index(drop=True)

# --- Lag kolonner for visning ---
df_all_sorted["modell"] = df_all_sorted.apply(
    lambda row: f"({row['p']},{row['q']}) t", axis=1
)
df_all_sorted["serie_navn"] = df_all_sorted["serie"].map(NAME_MAP)

# --- Vis tabeller ---
df_ger = df_all_sorted[df_all_sorted["serie"] == "GER"].reset_index(drop=True).r
df_no2 = df_all_sorted[df_all_sorted["serie"] == "NO2"].reset_index(drop=True).r

display(df_ger.style.set_caption(f"EGARCH-modeller for {NAME_MAP['GER']} (t-ford
display(df_no2.style.set_caption(f"EGARCH-modeller for {NAME_MAP['NO2']} (t-ford

# --- Plot AIC ---
fig_aic = px.bar(
    df_all_sorted,
    x="modell",
    y="AIC",
    color="serie_navn",
    barmode="group",
    title="AIC for EGARCH-modeller (t-fordeling)",
    labels={"modell": "Modell (p,q)", "AIC": "AIC"},
    hover_data=["p", "q", "Log-likelihood"]
)
fig_aic.update_layout(title_font_size=18, legend_title_text="Serie", xaxis_ticka
fig_aic.show()

# --- Plot BIC ---
fig_bic = px.bar(
    df_all_sorted,
    x="modell",
    y="BIC",
    color="serie_navn",
```

```
    barmode="group",
    title="BIC for EGARCH-modeller (t-fordeling)",
    labels={"modell": "Modell (p,q)", "BIC": "BIC"},
    hover_data=["p", "q", "Log-likelihood"]
)
fig_bic.update_layout(title_font_size=18, legend_title_text="Serie", xaxis_ticka
fig_bic.show()

# --- Plot Log-likelihood ---
fig_ll = px.bar(
    df_all_sorted,
    x="modell",
    y="Log-likelihood",
    color="serie_navn",
    barmode="group",
    title="Log-likelihood for EGARCH-modeller (t-fordeling)",
    labels={"modell": "Modell (p,q)", "Log-likelihood": "Log-likelihood"},
    hover_data=["p", "q", "AIC"]
)
fig_ll.update_layout(title_font_size=18, legend_title_text="Serie", xaxis_tickan
fig_ll.show()

# --- Eksport til Excel ---
excel_path = EXCEL_DIR / "data.xlsx"
sheet_name = "EGARCH-varianter"
os.makedirs(EXCEL_DIR, exist_ok=True)

if excel_path.exists():
    with ExcelWriter(excel_path, engine="openpyxl", mode="a", if_sheet_exists="r
        df_all_sorted.round(2).to_excel(writer, sheet_name=sheet_name, index=Fal
else:
    with ExcelWriter(excel_path, engine="openpyxl", mode="w") as writer:
        df_all_sorted.round(2).to_excel(writer, sheet_name=sheet_name, index=Fal

print(f"Resultater for EGARCH-modeller lagret til arket '{sheet_name}' i filen '
```

In [ ]:

```
# --- Beregn daglige endringer i prisene ---
daily_prices_diff = daily_prices.diff().dropna()

# --- Forbered datastrukturer ---
standardized_resid = pd.DataFrame(index=daily_prices_diff.index)
egarch_volatility = pd.DataFrame(index=daily_prices_diff.index)
garch_models = {}
adf_results = []
arch_results = []
ljungbox_resid_results = []
ljungbox_resid_sq_results = []

# --- Estimer EGARCH(1,1) og utfør tester ---
for col in daily_prices_diff.columns:

    # Hent original og differensiert serie
    orig_series = daily_prices[col].dropna()
    diff_series = daily_prices_diff[col]  # allerede uten NaN

    # Bestem land basert på kolonnenavn
    if "NO" in col:
        land = "Norge"
    elif "GER" in col:
        land = "Tyskland"
```

```
else:
    land = "Ukjent"

 # --- ADF-tester for stasjonaritet ---
adf_stat_orig, adf_pval_orig, _, _, crit_vals_orig, _ = adfuller(orig_series
adf_stat_diff, adf_pval_diff, _, _, crit_vals_diff, _ = adfuller(diff_series

# Kritiske verdier for 5 % nivå
crit_5_orig = crit_vals_orig.get('5%')
crit_5_diff = crit_vals_diff.get('5%')

# Lagre resultatene
adf_results.append({
    "Serie": col,
    "Land": land,

    # Før differensiering
    "ADF-statistikk (før)": adf_stat_orig,
    "p-verdi (før)": adf_pval_orig,
    "5 % grense (før)": crit_5_orig,
    "Stasjonær (før)": "Ja" if adf_stat_orig < crit_5_orig else "Nei",

    # Etter differensiering
    "ADF-statistikk (etter)": adf_stat_diff,
    "p-verdi (etter)": adf_pval_diff,
    "5 % grense (etter)": crit_5_diff,
    "Stasjonær (etter)": "Ja" if adf_stat_diff < crit_5_diff else "Nei"
})
# --- ARCH-test før modellering ---
arch_stat_pre, arch_pval_pre, *_ = het_arch(diff_series)

# --- Estimer EGARCH(1,1) med t-fordeling ---
model = arch_model(diff_series, vol="EGARCH", p=1, o=1, q=1, dist="t")
result = model.fit(disp="off")
garch_models[col] = result

# --- Hent residualer og betinget volatilitet ---
resid = result.resid
cond_vol = result.conditional_volatility
standardized = resid / cond_vol

standardized_resid[col] = standardized
egarch_volatility[col] = cond_vol

# --- ARCH-test etter modellering ---
arch_stat_post, arch_pval_post, *_ = het_arch(standardized)

arch_results.append({
    "Serie": col,
    "LM-statistikk (før)": arch_stat_pre,
    "p-verdi (før)": arch_pval_pre,
    "LM-statistikk (etter)": arch_stat_post,
    "p-verdi (etter)": arch_pval_post
})

# --- Ljung-Box-tester (nivå og kvadrat) ---
lb_resid = acorr_ljungbox(standardized, lags=10, return_df=True)
lb_resid_sq = acorr_ljungbox(standardized**2, lags=10, return_df=True)

for lag in range(1, 11):
```

```
        ljungbox_resid_results.append({
            "Serie": col,
            "Lag": lag,
            "LB-statistikk": lb_resid.loc[lag, "lb_stat"],
            "p-verdi": lb_resid.loc[lag, "lb_pvalue"]
        })
        ljungbox_resid_sq_results.append({
            "Serie": col,
            "Lag": lag,
            "LB-statistikk": lb_resid_sq.loc[lag, "lb_stat"],
            "p-verdi": lb_resid_sq.loc[lag, "lb_pvalue"]
        })

# --- Lag DataFrames med testresultater ---
adf_df = pd.DataFrame(adf_results)  # Beholder 'Serie' og 'Land' som vanlige kol
arch_df = pd.DataFrame(arch_results).set_index("Serie")
ljungbox_resid_df = pd.DataFrame(ljungbox_resid_results)
ljungbox_resid_sq_df = pd.DataFrame(ljungbox_resid_sq_results)

# --- Grupper Ljung-Box-resultater per serie ---
ljungbox_summary = {}
for serie in ljungbox_resid_df["Serie"].unique():
    resid = ljungbox_resid_df[ljungbox_resid_df["Serie"] == serie].set_index("La
    resid_sq = ljungbox_resid_sq_df[ljungbox_resid_sq_df["Serie"] == serie].set_
    summary = pd.DataFrame({
        "Lag": resid.index,
        "LB statistikk (nivå)": resid["LB-statistikk"],
        "p-verdi (nivå)": resid["p-verdi"],
        "LB statistikk (kvadrat)": resid_sq["LB-statistikk"],
        "p-verdi (kvadrat)": resid_sq["p-verdi"]
    }).reset_index(drop=True)
    ljungbox_summary[serie] = summary

# --- Grupper ARCH-resultater per serie ---
arch_summary = {}
for serie in arch_df.index:
    summary = pd.DataFrame({
        "Test": ["LM-statistikk", "p-verdi"],
        "Før modellering": [
            arch_df.loc[serie, "LM-statistikk (før)"],
            arch_df.loc[serie, "p-verdi (før)"]
        ],
        "Etter modellering": [
            arch_df.loc[serie, "LM-statistikk (etter)"],
            arch_df.loc[serie, "p-verdi (etter)"]
        ]
    })
    arch_summary[serie] = summary

# --- DCC-tapsfunksjon ---
def dcc_loss(params, residuals):
    """
    Tapsfunksjon for DCC-optimalisering. Returnerer log-likelihood-verdi gitt (a
    """
    alpha, beta = params
    if alpha < 0 or beta < 0 or (alpha + beta >= 1):
        return np.inf

    T = residuals.shape[0]
    Q_bar = np.cov(residuals.T)
```

```
    Q = Q_bar.copy()
    loss = 0.0

    for t in range(T):
        z_t = residuals.iloc[t].values.reshape(-1, 1)
        Q = (1 - alpha - beta) * Q_bar + alpha * (z_t @ z_t.T) + beta * Q
        D_inv = np.diag(1 / np.sqrt(np.diag(Q)))
        R_t = D_inv @ Q @ D_inv

        sign, logdet = np.linalg.slogdet(R_t)
        if sign <= 0:
            return np.inf

        e_t = residuals.iloc[t].values
        loss += logdet + e_t.T @ np.linalg.inv(R_t) @ e_t

    return loss

# --- Estimer DCC-parametere ---
opt_result = minimize(
    dcc_loss,
    [0.01, 0.98],
    args=(standardized_resid.dropna(),),
    method="SLSQP",
    constraints=[
        {"type": "ineq", "fun": lambda x: x[0]},
        {"type": "ineq", "fun": lambda x: x[1]},
        {"type": "ineq", "fun": lambda x: 1.0 - x[0] - x[1]}
    ],
    options={"disp": False}
)
alpha, beta = opt_result.x

# --- Beregn dynamiske korrelasjoner og kovarianser ---
T = len(standardized_resid.dropna())
Q_bar = np.cov(standardized_resid.dropna().T)
Q = Q_bar.copy()
R_list, Q_list = [], []

for t in range(T):
    z_t = standardized_resid.dropna().iloc[t].values.reshape(-1, 1)
    Q = (1 - alpha - beta) * Q_bar + alpha * (z_t @ z_t.T) + beta * Q
    D_inv = np.diag(1 / np.sqrt(np.diag(Q)))
    R_t = D_inv @ Q @ D_inv
    Q_list.append(Q.copy())
    R_list.append(R_t)

# --- Lag DataFrames for DCC-resultater ---
dates = standardized_resid.dropna().index
series_names = standardized_resid.columns.tolist()

dcc_covariances = pd.DataFrame({
    f"{series_names[0]}-{series_names[1]}": [Q[0, 1] for Q in Q_list]
}, index=dates)

correlation_data = {}
for i in range(len(series_names)):
    for j in range(i + 1, len(series_names)):
        navn = f"{series_names[i]}-{series_names[j]}"
        correlation_data[navn] = [R[i, j] for R in R_list]
```

```
dcc_correlations = pd.DataFrame(correlation_data, index=dates)

# --- Beregn total log-likelihood fra DCC ---
loss_values = []
for t, R_t in enumerate(R_list):
    sign, logdet = np.linalg.slogdet(R_t)
    if sign <= 0:
        continue
    e_t = standardized_resid.dropna().iloc[t].values
    quad_form = e_t.T @ np.linalg.inv(R_t) @ e_t
    loss_values.append(logdet + quad_form)
total_loss = sum(loss_values)

# --- Klargjør og formatter ADF-resultater for eksport ---
adf_formatted = adf_df.copy()
adf_formatted["ADF-statistikk (før)"] = adf_formatted["ADF-statistikk (før)"].ma
adf_formatted["p-verdi (før)"] = adf_formatted["p-verdi (før)"].map(lambda x: f"
adf_formatted["ADF-statistikk (etter)"] = adf_formatted["ADF-statistikk (etter)"
adf_formatted["p-verdi (etter)"] = adf_formatted["p-verdi (etter)"].map(lambda x

# --- Klargjør for eksport til Excel ---
save_kwargs = {
    "Rådata": daily_prices,
    "Differensiert": daily_prices_diff,
    "Volatilitet": egarch_volatility,
    "Standardiserte residualer": standardized_resid,
    "Korrelasjoner": dcc_correlations,
    "Kovarianser": dcc_covariances,
    "ADF_tester": adf_formatted,
    "DCC_parametere": pd.DataFrame({
        "alpha": [alpha],
        "beta": [beta],
        "Total log-likelihood loss": [total_loss]
    })
}

for serie, df in arch_summary.items():
    save_kwargs[f"ARCH_{serie}"] = df

for serie, df in ljungbox_summary.items():
    save_kwargs[f"LjungBox_{serie}"] = df

# --- Eksporter alle resultater ---
save_to_excel(
    excel_path=Path("output/excel/data_series.xlsx"),
    **save_kwargs
)


export_garch_results_to_excel(
    results_dict=garch_models,
    filename="garch_results.xlsx"
)
```

In [ ]:
```
# --- Deskriptiv analyse av daglige priser ---
descriptive_analysis(daily_prices)

# --- Deskriptiv analyse av EGARCH-volatilitet ---
descriptive_analysis(egarch_volatility)
```

```
# --- Deskriptiv analyse av standardiserte residualer ---
descriptive_analysis(standardized_resid)

# --- Deskriptiv analyse av DCC-korrelasjoner ---
descriptive_analysis(dcc_correlations)

# --- Deskriptiv analyse av DCC-kovarianser ---
descriptive_analysis(dcc_covariances)

# --- Filtrer DCC-korrelasjoner for perioden 2021–2024 og analyser ---
dcc_correlations.index = pd.to_datetime(dcc_correlations.index)
dcc_correlations_2021_2024 = dcc_correlations.loc["2021-01-01":"2024-12-31"]
descriptive_analysis(dcc_correlations_2021_2024)
```

Plotting

Daglige strømpriser

In [ ]:
```
# --- Plott tidsserie for daglige strømpriser med vertikal linje ved 24. februar
plot_timeseries(
    data=daily_prices,
    title="Daglige strømpriser",
    y_title="Pris (EUR/MWh)",
    show_break=True
)

# --- Plott histogram for å vise fordelingen av daglige strømpriser ---
plot_histogram(
    data=daily_prices,
    title="Daglige strømpriser",
    x_title="Pris (EUR/MWh)"
)

# --- Plott scatter-plot for visuell inspeksjon av trender og ekstremverdier ---
plot_scatter(
    data=daily_prices,
    title="Daglige strømpriser",
    y_title="Pris (EUR/MWh)",
    show_break=True
)

# --- Plott 7-dagers glidende gjennomsnitt for å jevne ut kortsiktig støy ---
plot_rolling_average(
    data=daily_prices,
    window=7,
    title="Strømpriser (7-dagers glidende gjennomsnitt)",
    y_title="Pris (EUR/MWh)",
    show=True,
    show_break=True
)

# --- Sammenlign fordeling av strømpriser i Tyskland og Norge ---
plot_histogram_comparison(
    series1=daily_prices["GER"],
    series2=daily_prices["NO2"],
```

```
    label1="Tyskland",
    label2="Norge",
    title="Strømpriser i Tyskland og Norge",
    xlabel="Pris (EUR/MWh)"
)
```

Sammenligning av daglige priser før og etter invasjonen

In [ ]:
```
# --- Plott histogram-sammenligning av strømpriser før og etter invasjonen ---
for pair in daily_prices.columns:
    name = NAME_MAP.get(pair, pair)  # Bruk visningsnavn hvis tilgjengelig

    plot_histogram_comparison(
        series1=daily_prices.loc[daily_prices.index < BREAK_DATE, pair],
        series2=daily_prices.loc[daily_prices.index >= BREAK_DATE, pair],
        label1="Før 24. feb 2022",
        label2="Etter 24. feb 2022",
        title=f"Strømpriser i {name} før og etter invasjonen",
        xlabel="Pris (EUR/MWh)",
    )
```

ACF og PACF

In [ ]:
```
# --- ACF og PACF for strømpriser (nivådata) ---
plot_acf_pacf(daily_prices["GER"], title_prefix="strømprisene i Tyskland", lags=
plot_acf_pacf(daily_prices["NO2"], title_prefix="strømprisene i Norge", lags=20)

# --- ACF og PACF for differensierte priser ---
plot_acf_pacf(daily_prices_diff["GER"], title_prefix="differensierte strømpriser
plot_acf_pacf(daily_prices_diff["NO2"], title_prefix="differensierte strømpriser
```

EGARCH-volatilitet

In [ ]:
```
# --- Hent og lag DataFrame med betinget volatilitet fra EGARCH-modellene ---
egarch_volatility = pd.DataFrame({
    col: garch_models[col].conditional_volatility
    for col in daily_prices_diff.columns
})

# --- Plott tidsserie for EGARCH-volatilitet med bruddpunkt markert ---
plot_timeseries(
    data=egarch_volatility,
    title="EGARCH-volatilitet",
    y_title="Volatilitet",
    show_break=True
)

# --- Plott histogram for å sammenligne volatilitet før og etter invasjonen ---
for col in ["GER", "NO2"]:
    plot_histogram_comparison(
        series1=egarch_volatility.loc[egarch_volatility.index < BREAK_DATE, col]
        series2=egarch_volatility.loc[egarch_volatility.index >= BREAK_DATE, col
        label1="Før 24. feb 2022",
```

```
        label2="Etter 24. feb 2022",
        title=f"EGARCH-volatilitet for {NAME_MAP[col]} før og etter invasjonen",
        xlabel="Volatilitet",
    )
```

Residualer

In [ ]:
```
# --- Plott tidsserie for differensierte strømpriser ---
plot_timeseries(
    data=daily_prices_diff,
    title="Differensierte strømpriser",
    y_title="Endring i pris (EUR/MWh)",
    show_break=True  # Marker 24. februar 2022 i figuren
)

# --- Hent residualer fra EGARCH-modellene og lag DataFrame ---
residuals_df = pd.DataFrame({
    col: garch_models[col].resid
    for col in daily_prices_diff.columns
})

# --- Plott tidsserie for residualer ---
plot_timeseries(
    data=residuals_df,
    title="Residualer",
    y_title="Residualer",
    show_break=True
)

# --- Plott tidsserie for standardiserte residualer ---
plot_timeseries(
    data=standardized_resid,
    title="Standardiserte residualer",
    y_title="Standardiserte residualer",
    show_break=True
)
```

QQ-plot for residualer og standardiserte residualer

In [ ]:
```
# --- Plott QQ-plot for residualer og standardiserte residualer ---

# Kartlegg adjektivnavn for land
ADJECTIVE_NAME_MAP = {
    "GER": "tyske",
    "NO2": "norske"
}

for col in daily_prices_diff.columns:
    # Sett farge basert på kolonnenavn
    if "GER" in col:
        color = COLOR_1
    elif "NO2" in col:
        color = COLOR_2
    else:
        color = COLOR_3

    # Definer datasett
```

```
    datasets = [
        (garch_models[col].resid.dropna(), "residualer", color),
        (standardized_resid[col].dropna(), "standardiserte residualer", color)
    ]

    for data, label, color in datasets:
        adjektiv = ADJECTIVE_NAME_MAP.get(col, col)
        title = f"QQ-plot for {label} fra den {adjektiv} EGARCH-modellen"

        plot_qq(
            data=data,
            color=color,
            title=title
        )
```

DCC kovarians

In [ ]:
```
# --- Plott DCC tidsvarierende kovarians ---
dcc_covariances = dcc_covariances.rename(columns={"GER-NO2": "DCC-estimert kovar
plot_timeseries(
    data=dcc_covariances,
    title="Tidsvarierende kovarians",
    y_title="Kovarians",
    show_break=True
)
```

Sammenligning av DCC-kovarians før og etter invasjonen

In [ ]:
```
# --- Plott histogram-sammenligning av DCC-kovarianser før og etter invasjonen -
for pair in dcc_covariances.columns:
    plot_histogram_comparison(
        series1=dcc_covariances.loc[dcc_covariances.index < BREAK_DATE, pair],
        series2=dcc_covariances.loc[dcc_covariances.index >= BREAK_DATE, pair],
        label1="Før 24. feb 2022",
        label2="Etter 24. feb 2022",
        title=f"Tidsvarierende kovarians før og etter invasjonen",
        xlabel="DCC-kovarians",
    )
```

DCC korrelasjon

In [ ]:
```
# --- Gi kolonnen et tydelig navn for figurer og tabeller ---
dcc_correlations = dcc_correlations.rename(columns={"GER-NO2": "DCC-estimert kor

# --- Plott tidsserie av DCC-estimert korrelasjon ---
plot_timeseries(
    data=dcc_correlations,
    title="Tidsvarierende korrelasjon",
    y_title="Korrelasjon",
    show_break=True  # Marker 24. februar 2022 i figuren
)

# --- Beregn gjennomsnittlig korrelasjon per år ---
```

```
avg_per_year = dcc_correlations.groupby(dcc_correlations.index.year).mean()

# --- Velg ut relevante år for analyse ---
years_of_interest = [2020, 2021, 2022, 2023, 2024]
avg_filtered = avg_per_year.loc[avg_per_year.index.isin(years_of_interest)]

# --- Skriv resultatene til konsoll ---
print("Gjennomsnittlig korrelasjon per år:")
print(avg_filtered)
```

Sammenligning av DCC-korrelasjon før og etter invasjonen

In [ ]:
```
# --- Plott histogram-sammenligning av DCC-korrelasjoner før og etter invasjonen
for pair in dcc_correlations.columns:
    plot_histogram_comparison(
        series1=dcc_correlations.loc[dcc_correlations.index < BREAK_DATE, pair],
        series2=dcc_correlations.loc[dcc_correlations.index >= BREAK_DATE, pair]
        label1="Før 24. feb 2022",
        label2="Etter 24. feb 2022",
        title=f"Tidsvarierende korrelasjon før og etter invasjonen",
        xlabel="DCC-korrelasjon",
    )
```

In [ ]:
```
# --- Definer periodeavgrensning ---
START_DATE = "2021-01-01"
END_DATE = "2024-12-31"

# --- Plott histogram-sammenligning av DCC-korrelasjoner før og etter invasjonen
for pair in dcc_correlations.columns:
    # Filtrer data til angitt periode (2021–2024)
    filtered_series = dcc_correlations.loc[
        (dcc_correlations.index >= START_DATE) &
        (dcc_correlations.index <= END_DATE), pair
    ]

    # Del opp i før og etter 24. februar 2022
    series1 = filtered_series.loc[filtered_series.index < BREAK_DATE]
    series2 = filtered_series.loc[filtered_series.index >= BREAK_DATE]

    # Plott histogram med KDE
    plot_histogram_comparison(
        series1=series1,
        series2=series2,
        label1="Før 24. feb 2022",
        label2="Etter 24. feb 2022",
        title=f"Tidsvarierende korrelasjon før og etter invasjonen (2021–2024)",
        xlabel="DCC-korrelasjon",
    )
```

Plotting og analyse av NordLink

In [ ]:
```
def load_daily_flow(
    years: list[int],
    input_dir: Path,
    columns_to_keep: list[str]
```

```
) -> pd.DataFrame:
    dataframes = []
    missing_years = []

    for year in years:
        file_path = input_dir / f"AuctionFlow_{year}_DayAhead_GER_Daily.csv"

        if file_path.exists():
            df = pd.read_csv(
                file_path,
                delimiter=";",    # semikolon separator
                decimal=".",      # desimaler med punktum
            )
            df.columns = df.columns.str.strip()
            df["Delivery Date CET"] = pd.to_datetime(df["Delivery Date CET"])
            dataframes.append(df)
        else:
            missing_years.append(year)

    if not dataframes:
        raise FileNotFoundError("Ingen CSV-filer ble funnet i input-mappen.")

    if missing_years:
        print(f"Følgende år manglet filer og ble hoppet over: {missing_years}")

    combined = pd.concat(dataframes, ignore_index=True)
    combined = combined[columns_to_keep].dropna()

    combined.set_index("Delivery Date CET", inplace=True)
    combined.sort_index(inplace=True)

    return combined
```

In [ ]:

```
# --- Kolonnenavn uten ---
column_orig = "GER GER-NO2 Total Net Position (MWh)"
columns = ["Delivery Date CET", column_orig]
dagligkapasitet = "Maksimal kapasitet"
file_tag = "no2_ger"
column_name = "Nettoflyt"  # fjernet (NordLink)
column = column_name

# --- Last data ---
df_flow = load_daily_flow(
    years=[2020, 2021, 2022, 2023, 2024],
    input_dir=Path("input/auction_flow"),
    columns_to_keep=columns
)

# --- Konverter og behold retning: positiv flyt = fra Norge til Tyskland gjennom
df_flow[column_orig] = pd.to_numeric(df_flow[column_orig], errors='coerce')

# --- Filtrer bort alt før første positiv nettoflyt ---
first_valid_index = df_flow[df_flow[column_orig] > 0].index.min()
df_flow = df_flow.loc[df_flow.index >= first_valid_index]

# --- Gi nytt kolonnenavn ---
df_flow.rename(columns={column_orig: column_name}, inplace=True)
y_label = f"{column_name} (MWh)"

# --- Tidsserieplot ---
```

```
plot_timeseries(
    data=df_flow,
    title="Aggregert daglig netto kraftflyt fra Norge til Tyskland via NordLink"
    y_title=y_label,
    show_break=True,
    reference_value=0,
    margin=33600,
    reference_label=dagligkapasitet
)

# --- Histogram ---
plot_histogram(
    data=df_flow,
    title="Fordeling av aggregert daglig netto kraftflyt fra Norge til Tyskland
    x_title=y_label,
    reference_value=0,
    margin=33600,
    reference_label=dagligkapasitet
)

# --- Scatterplot ---
plot_scatter(
    data=df_flow,
    title="Aggregert daglig netto kraftflyt fra Norge til Tyskland via NordLink"
    y_title=y_label,
    show_break=True,
    reference_value=0,
    margin=33600,
    reference_label=dagligkapasitet
)

# --- Glidende gjennomsnitt ---
plot_rolling_average(
    data=df_flow,
    window=7,
    title="Netto kraftflyt fra Norge til Tyskland via NordLink (7-dagers glidend
    y_title=y_label,
    show=True,
    show_break=True,
    reference_value=0,
    margin=33600,
    reference_label=dagligkapasitet
)

# --- Histogram før og etter invasjonen ---
series1 = df_flow.loc[df_flow.index < BREAK_DATE, column]
series2 = df_flow.loc[df_flow.index >= BREAK_DATE, column]

plot_histogram_comparison(
    series1=series1,
    series2=series2,
    label1="Før 24. feb 2022",
    label2="Etter 24. feb 2022",
    title="Fordeling av kraftflyt fra Norge til Tyskland via NordLink før og ett
    xlabel="Nettoflyt (MWh)",
    reference_value=0,
    margin=33600,
    reference_label=dagligkapasitet
)
```

```python
# --- Histogram for 2021–2023 ---
df_sub = df_flow.loc["2021-01-01":"2023-12-31"]
series1 = df_sub.loc[df_sub.index < BREAK_DATE, column]
series2 = df_sub.loc[df_sub.index >= BREAK_DATE, column]

plot_histogram_comparison(
    series1=series1,
    series2=series2,
    label1="Før 24. feb 2022",
    label2="Etter 24. feb 2022",
    title="Fordeling av kraftflyt fra Norge til Tyskland via NordLink før og ett
    xlabel="Nettoflyt (MWh)",
    reference_value=0,
    margin=33600,
    reference_label=dagligkapasitet
)

colnavn = df_flow[[column]].columns[0]
descriptive_analysis(df_flow[[column]], prefix=f"{colnavn}_stromflyt")
```

In [ ]:
```python
# --- Terskelverdi for ekstremflyt ---
terskel = 33000

# --- Sørg for at BREAK_DATE er et tidsstempel ---
if isinstance(BREAK_DATE, str):
    BREAK_DATE = pd.Timestamp(BREAK_DATE)

# --- Del datasettet før og etter invasjonen ---
df_before = df_flow[df_flow.index < BREAK_DATE]
df_after = df_flow[df_flow.index >= BREAK_DATE]

# --- Filtrer datoer med ekstrem flyt ---
datoer_til_tyskland_før = df_before[df_before[column] > terskel].index.date
datoer_til_norge_før = df_before[df_before[column] < -terskel].index.date
datoer_til_tyskland_etter = df_after[df_after[column] > terskel].index.date
datoer_til_norge_etter = df_after[df_after[column] < -terskel].index.date

# --- Telling ---
til_tyskland_før = len(datoer_til_tyskland_før)
til_norge_før = len(datoer_til_norge_før)
til_tyskland_etter = len(datoer_til_tyskland_etter)
til_norge_etter = len(datoer_til_norge_etter)

# --- Konsollutskrift ---
print(f"Før invasjonen (før {BREAK_DATE.date()}):")
print(f"  > {terskel} MWh til Tyskland: {til_tyskland_før} dager")
print(f"  > {terskel} MWh til Norge: {til_norge_før} dager\n")

print(f"Etter invasjonen (fra og med {BREAK_DATE.date()}):")
print(f"  > {terskel} MWh til Tyskland: {til_tyskland_etter} dager")
print(f"  > {terskel} MWh til Norge: {til_norge_etter} dager")

# --- Resultattabell med tellinger ---
flyt_data = pd.DataFrame({
    "Retning": ["Til Tyskland", "Til Norge"] * 2,
    "Periode": ["Før invasjonen"] * 2 + ["Etter invasjonen"] * 2,
    "Antall dager": [
        til_tyskland_før,
        til_norge_før,
        til_tyskland_etter,
```

```
        til_norge_etter
    ]
})

# --- Tabell med datoer uten klokkeslett ---
dato_data = pd.DataFrame({
    "Til Tyskland (før)": pd.Series(datoer_til_tyskland_før),
    "Til Norge (før)": pd.Series(datoer_til_norge_før),
    "Til Tyskland (etter)": pd.Series(datoer_til_tyskland_etter),
    "Til Norge (etter)": pd.Series(datoer_til_norge_etter)
})

# --- Lagring til Excel med din funksjon ---
save_to_excel(
    excel_path=Path("output/excel/data_series.xlsx"),
    ekstremflyt=flyt_data,
    ekstremdatoer=dato_data
)
```