# Volatility Modelling

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import netCDF4 as nc
from datetime import datetime, timedelta
import statsmodels.api as sm
from sklearn.preprocessing import PolynomialFeatures
import scipy.interpolate as interpolate
from scipy.stats import norm, ks_2samp, anderson
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from scipy.stats import skew, kurtosis
```

Data Preparation for Volatility Modelling

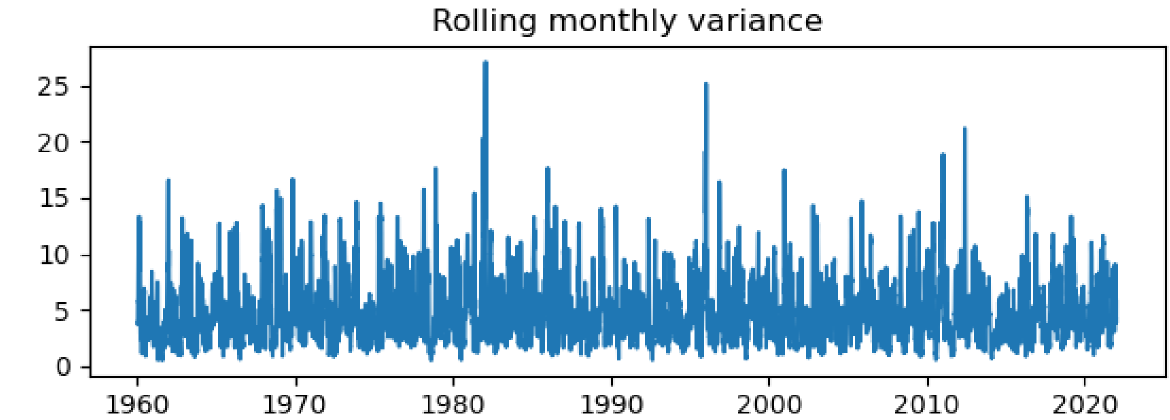
In [ ]:
```
max_daily_dataset = nc.Dataset('../../datasets/tasmax_hadukgrid_uk_region_d
min_daily_dataset = nc.Dataset('../../datasets/tasmin_hadukgrid_uk_region_d
region = 10

def convert_hours_to_datetime(hours):
    base_date = datetime(1800, 1, 1, 0, 0, 0)  # Base date for the calculat
    delta = timedelta(hours=hours)  # Create a timedelta based on the hours
    # Add the timedelta to the base date to get the resulting datetime
    result_datetime = base_date + delta
    return result_datetime.date()

region_daily_min = min_daily_dataset['tasmin'][:, region]
region_daily_max = max_daily_dataset['tasmax'][:, region]
data_dict = {'min': region_daily_min, 'max': region_daily_max, 'time': min_
df = pd.DataFrame(data=data_dict)
df['date'] = df['time'].apply(convert_hours_to_datetime)
df['avg'] = (df['min'] + df['max'])/2
pass
```

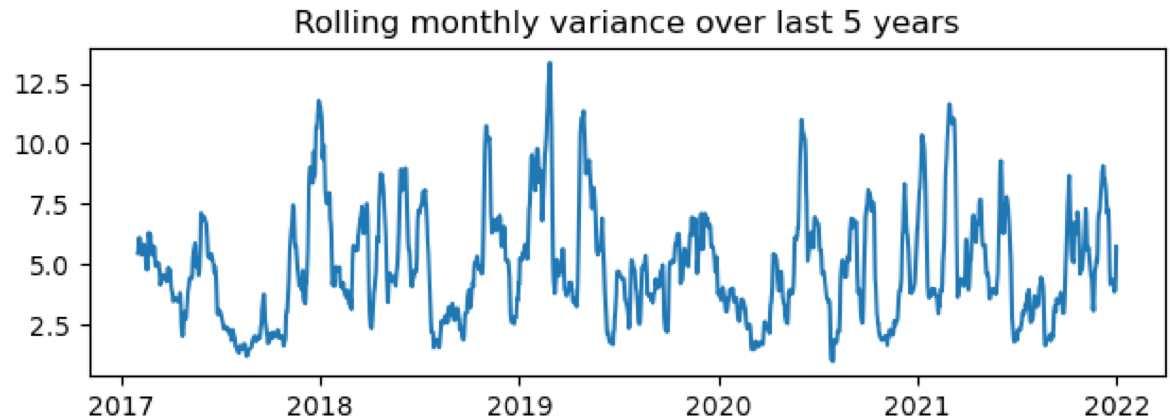
# Daily and monthly Volatility Modelling

First, we look at the monthly rolling volatility in our time series data

In [ ]:
```
fig, axs = plt.subplots(2, 1)
axs[0].plot(df['date'], df['avg'].rolling(window = 30).var())
axs[0].set_title("Rolling monthly variance")
axs[1].plot(df['date'][-365*5:], df['avg'][-365*5:].rolling(window = 30).va
axs[1].set_title("Rolling monthly variance over last 5 years")
fig.tight_layout()
fig.show()
```

```
/tmp/ipykernel_9380/166740494.py:7: UserWarning: Matplotlib is currently us
ing module://matplotlib_inline.backend_inline, which is a non-GUI backend,
so cannot show the figure.
  fig.show()
```

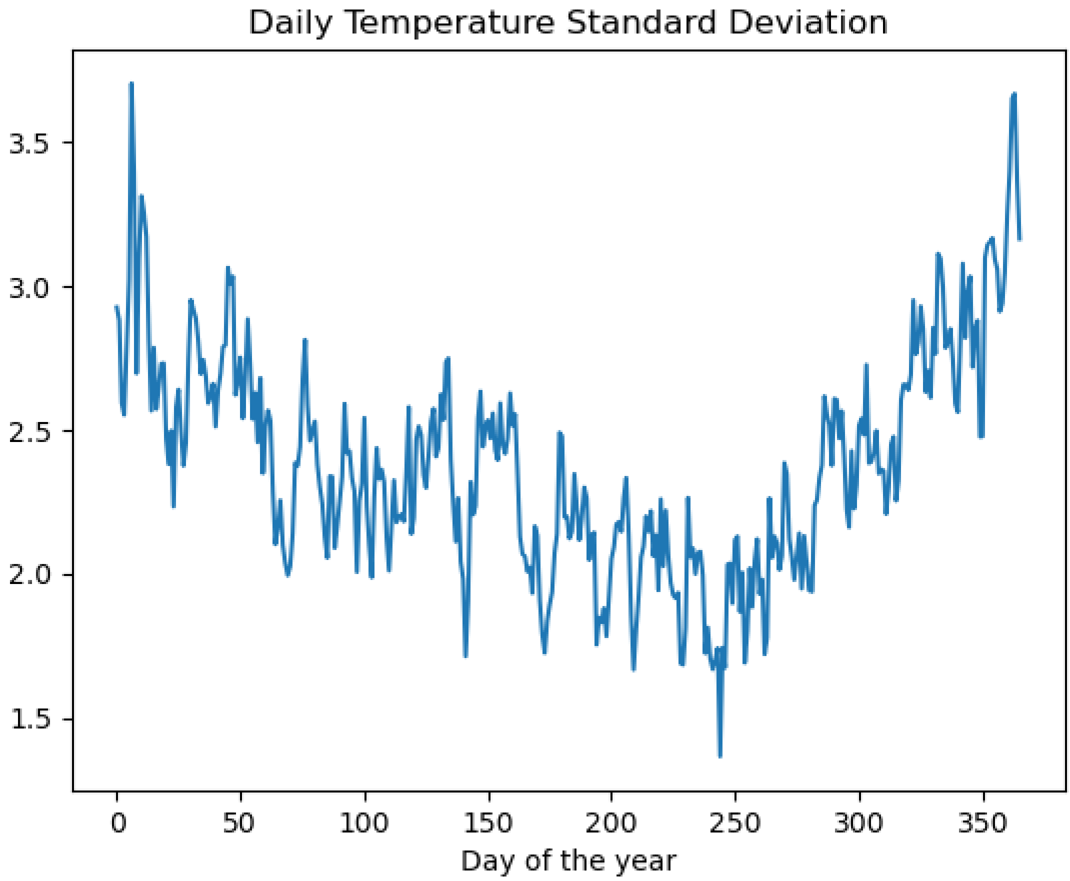



In [ ]:
```
# Convert 'date' column to datetime format
df['date'] = pd.to_datetime(df['date'])

# Extract day and month from the 'date' column
df['day'] = df['date'].dt.day
df['month'] = df['date'].dt.month

# Group by day and month, and compute variance for each day
std_by_day = df.groupby(['month', 'day'])['avg'].std()
print(std_by_day)
```

```
month  day
1      1      2.927041
       2      2.881849
       3      2.593544
       4      2.552560
       5      2.788373
                ...
12     27     3.399486
       28     3.652368
       29     3.669040
       30     3.366287
       31     3.165318
Name: avg, Length: 366, dtype: float64
```

In [ ]:
```
plt.plot(std_by_day.values)
plt.title("Daily Temperature Standard Deviation")
plt.xlabel("Day of the year")
plt.show()
```




# Model Comparison Metrics

Ideally, I would use a likelihood based metric such as the Akaike Information and Bayesian Information Criterons to compare model suitability. The Akaike Information Criteron is given by,

$$AIC = -2\log\hat{L} - 2k$$

Where $\hat{L}$ is the likelihood function of the training data under the model and $k$ is the number of parameters in the model, a regularising term.

Similarly, the Bayesian Information Criteron is given by,

$$BIIC = -2\log\hat{L} - k \cdot \log n$$

In this model, the regularisation term is not multiplied by a constant but instead by the logarthim of the number of data points in the model. Scaling by the number of training points instead of a constant is subtle however as models with more data are more prone to overfitting, scaling the regularisation penalises larger models attempting to fit more data. It also ensures the score of a model is invariant of the dataset size.

Unfortunately, in fitting the trend a single prediction is outputted for each input, therefore I cannot use likelihood based metrics as there is no probability distribution produced by my outputs, effectively our models gives a Kronker delta for the trend at each point. Therefore, I will modify these metric to depend on the RSS instead.

In [ ]:
```
def AIC_RSS(y_pred, y_true, k_parameters):
```

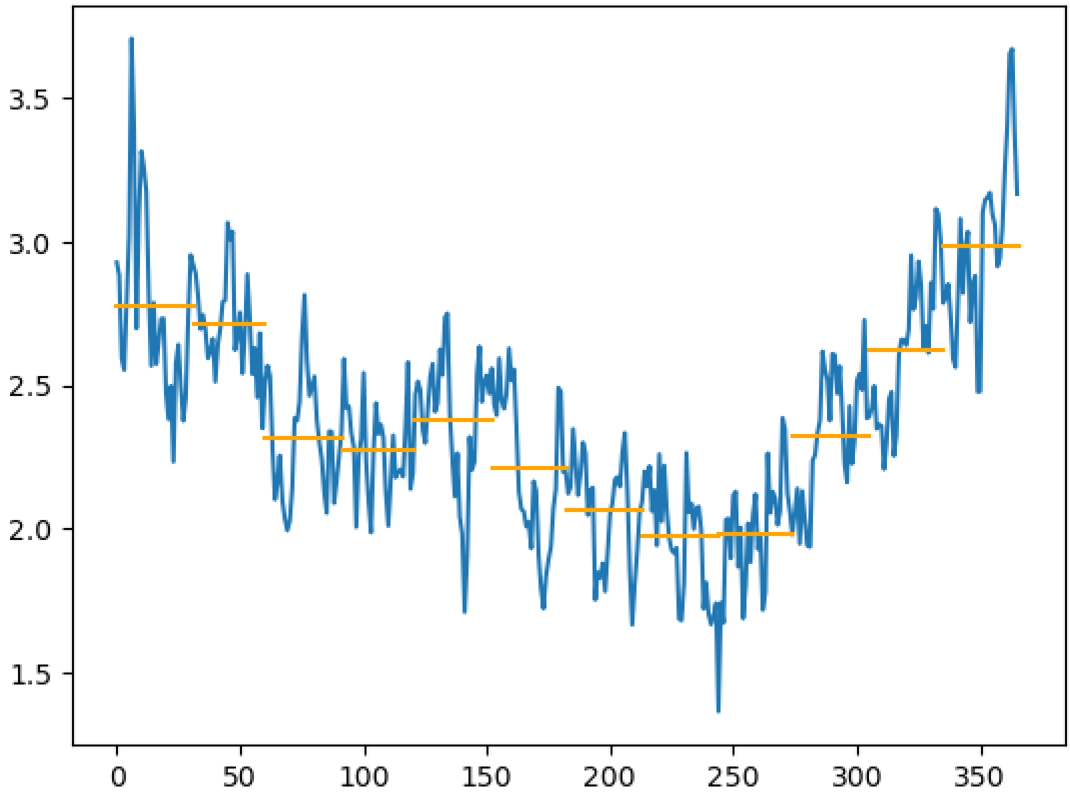
```
    return 2 * ((y_pred - y_true)**2).sum() + 2 * k_parameters

def BIC_RSS(y_pred, y_true, k_parameters):
    return ((y_pred - y_true)**2).sum() + np.log(len(y_pred)) * k_parameter
```

# Piecewise Modelling

Our first simple model will be fitting a piecewise constant function to each month of the year.

In [ ]:
```
monthly_means = np.zeros(12, dtype=np.float64)
fig, axs = plt.subplots(1)
axs.plot(std_by_day.values)
piecewise_preds = np.empty(len(std_by_day.values))
day = 0

for i in range(12):
    month = i + 1
    monthly_means[i] = np.mean(std_by_day[month].values)
    print(f"Month {month} average standard deviation: {monthly_means[i]}")
    newday = day + len(std_by_day[month].values)
    piecewise_preds[day:newday] = monthly_means[i]
    axs.plot([day, newday], [monthly_means[i], monthly_means[i]], color='or
    day = newday
```

```
Month 1 average standard deviation: 2.7763002417809917
Month 2 average standard deviation: 2.713507633203285
Month 3 average standard deviation: 2.3146612343794626
Month 4 average standard deviation: 2.2733487675745825
Month 5 average standard deviation: 2.375743823973072
Month 6 average standard deviation: 2.211228659500156
Month 7 average standard deviation: 2.0624116896451805
Month 8 average standard deviation: 1.973711600351153
Month 9 average standard deviation: 1.9786444518128625
Month 10 average standard deviation: 2.3254938100930382
Month 11 average standard deviation: 2.6240833015672775
Month 12 average standard deviation: 2.986426498518974
```



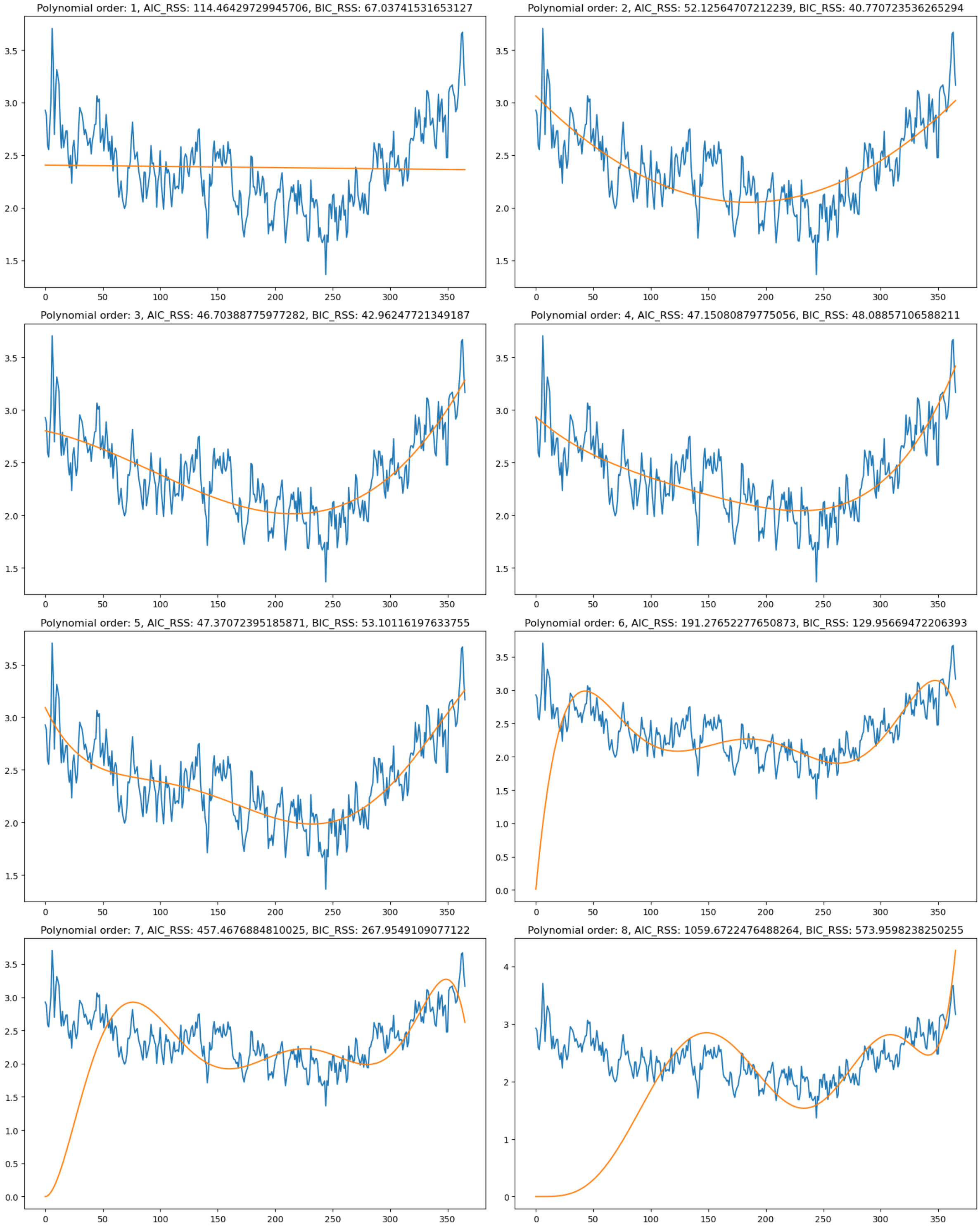
In [ ]:
```
print(f"AIC_RSS: {AIC_RSS(piecewise_preds, std_by_day.values, 12)}")
print(f"BIC_RSS: {BIC_RSS(piecewise_preds, std_by_day.values, 12)}")
```

```
AIC_RSS: 63.034736647996425
BIC_RSS: 90.3489683248146
```

# Parameteric Regression

In [ ]:
```
x = np.array(np.arange(len(std_by_day.values.flatten())))
y = np.array(std_by_day.values.flatten())

# Degrees of polynomial
polynomial_degrees = np.arange(1, 9, dtype=np.int16)

fig, axs = plt.subplots(4, 2, figsize=(16, 20))

def polynomial_regression_fit(degree, x, y):
    x= x[:,np.newaxis]
    y= y[:,np.newaxis]
    poly_feats = PolynomialFeatures(degree=degree)
    transform = poly_feats.fit_transform(x)
    model = sm.OLS(y, transform).fit()
    predictions = model.predict(transform)
    return model, predictions

def polynomial_regression(degree, x, y):
    _model, predictions = polynomial_regression_fit(degree, x, y)
    return predictions

for degree in polynomial_degrees:
    predictions = polynomial_regression(degree, x, y)
    axs[int((degree - 1) / 2), (degree + 1) % 2 ].plot(y)
    axs[int((degree - 1) / 2), (degree + 1) % 2 ].plot(predictions)
    axs[int((degree - 1) / 2), (degree + 1) % 2 ].set_title(f"Polynomial or
fig.tight_layout()
fig.show()
```

```
/tmp/ipykernel_9380/3291774492.py:28: UserWarning: Matplotlib is currently
using module://matplotlib_inline.backend_inline, which is a non-GUI backen
d, so cannot show the figure.
  fig.show()
```




# Fourier Series Volatility Modelling

We use the numpy fast fourier series function to calculate the fourier coefficients of the series. These are given analytically by,

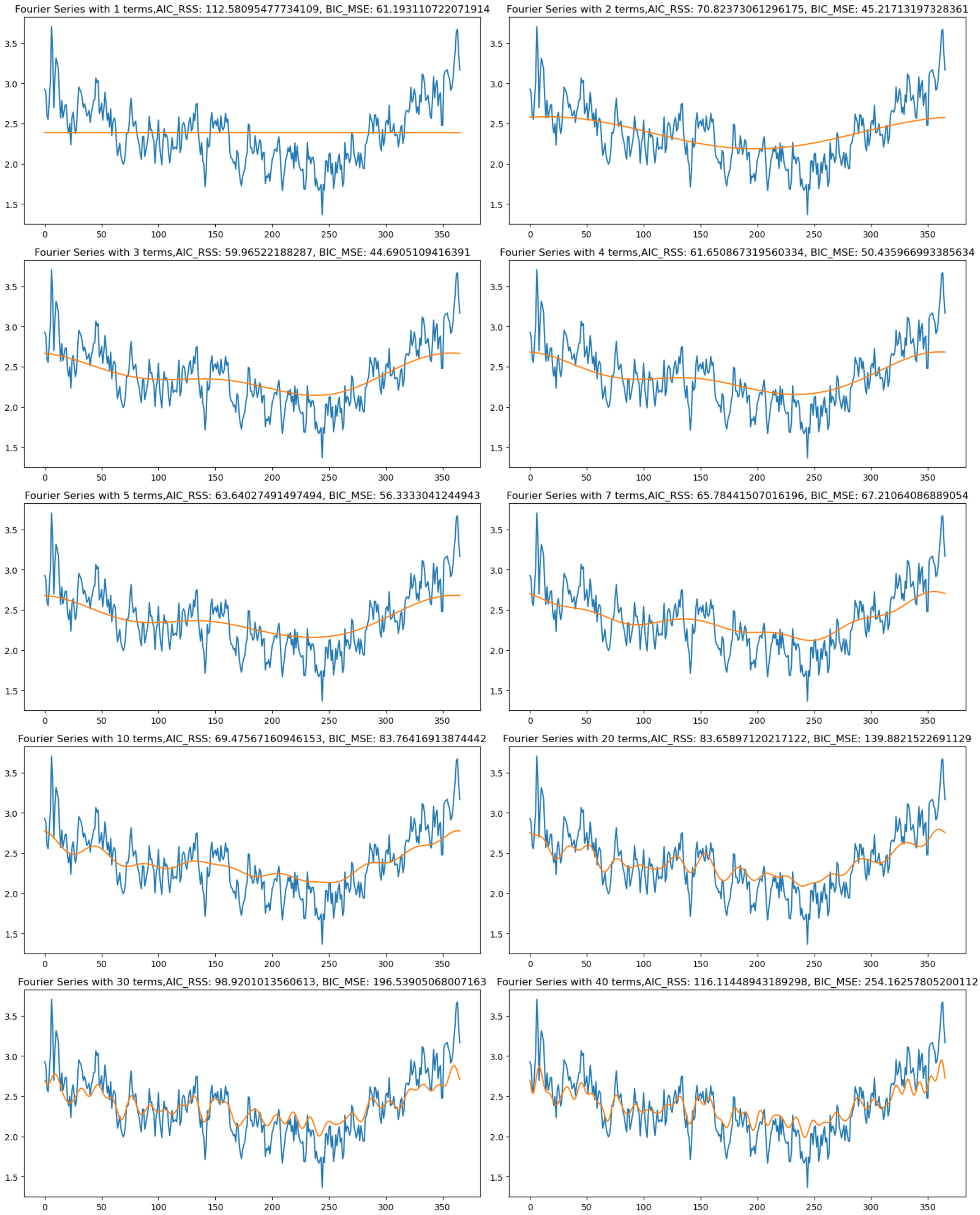
$$c_k = \int_0^1 f(x) e^{-2\pi ikx} dx$$

The inverse fourier transform is then used to reconstruct the signal using these terms, the formula for the reconstructed sequence is given by:

$$f(x) = \sum_{k=-N}^{N} c_k e^{2\pi ikx}$$

In [ ]:
```
x = np.arange(0, 366)
y = np.array(std_by_day.values.flatten())

 # Number of terms in the Fourier series
fourier_terms = np.array([1, 2, 3, 4, 5, 7, 10, 20, 30, 40])
fourier_coeffs = np.fft.fft(y) / len(y)

def fourier_series(x, coeffs, n_terms):
    y_reconstructed = np.zeros_like(x, dtype=complex)
    for k in range(n_terms):
        y_reconstructed += coeffs[k] * np.exp(2j * np.pi * k * x / 365.25)
    return y_reconstructed.real

fig, axs = plt.subplots(5, 2, figsize=(16, 20))
for fourier_index, fourier_term in enumerate(fourier_terms):
    predictions = fourier_series(x, fourier_coeffs, fourier_term)
    axs[int(fourier_index / 2), fourier_index % 2].plot(y)
    axs[int(fourier_index / 2), fourier_index % 2].plot(predictions)
    axs[int(fourier_index / 2), fourier_index % 2].set_title(f"Fourier Seri

fig.tight_layout()
fig.show()
```

```
/tmp/ipykernel_9380/1005823297.py:22: UserWarning: Matplotlib is currently
using module://matplotlib_inline.backend_inline, which is a non-GUI backen
d, so cannot show the figure.
  fig.show()
```



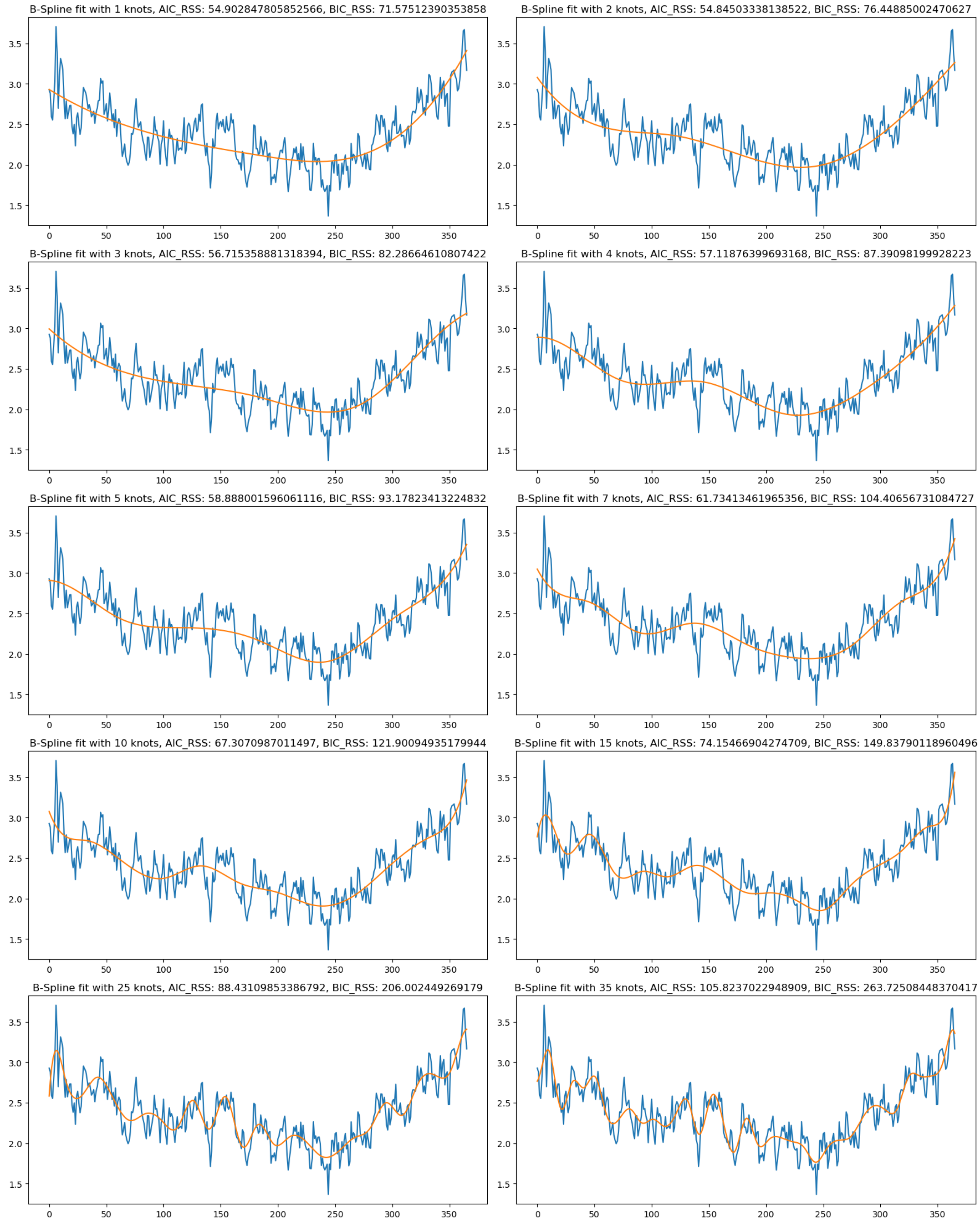

# Splines

Splines model functions by interpolating between so-called `knots', which are points lying on the tangents between data points. These curves are $\mathbf{C}^2$ continuous. The number of knots is a hyper-parameter of the model. I will be using cubic splines to interpolate between knots, a higher degree corresponds to a smoother curve, however this is at the expense of more computation.

In the `scipy.interpolate.splrep` (spline representation) function by default cubics are used to interpolate between knots. The learnable parameters are given by `c` and therefore the number of learnable parameters (a proxy for model complexity) is given by `len(c)`.

In [ ]:
```
knot_numbers = np.array([1, 2, 3, 4, 5, 7, 10, 15, 25, 35])

def spline_fit(x, y, no_knots):
    x_new = np.linspace(0, 1, no_knots+2)[1:-1]
    q_knots = np.quantile(x, x_new)
    t,c,k = interpolate.splrep(x, y, t=q_knots, s=1)
    return interpolate.BSpline(t,c,k)(x), len(c)

fig, axs = plt.subplots(5, 2, figsize=(16, 20))
for knot_index, knot in enumerate(knot_numbers):
    predictions, no_learnable_parameters = spline_fit(x, y, knot)
    axs[int(knot_index / 2), knot_index % 2].plot(y)
    axs[int(knot_index / 2), knot_index % 2].plot(predictions)
    axs[int(knot_index / 2), knot_index % 2].set_title(f"B-Spline fit with

fig.tight_layout()
fig.show()
```

```
/tmp/ipykernel_9380/3351486271.py:17: UserWarning: Matplotlib is currently
using module://matplotlib_inline.backend_inline, which is a non-GUI backen
d, so cannot show the figure.
  fig.show()
```




# Chosen Volatility Model

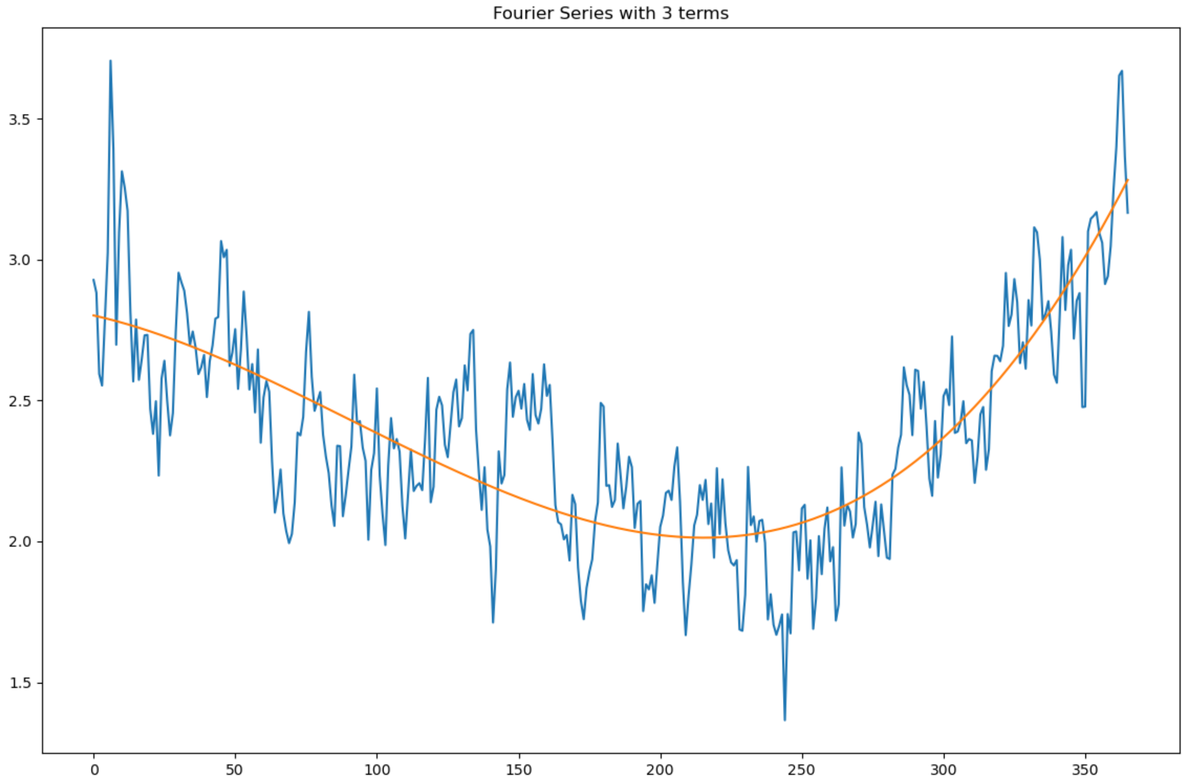
We summarise the results in the table below;

| Model | AIC_RSS | BIC_RSS |
|---|---|---|
| Piecewise | 63.035 | 90.349 |
| Polynomial (Degree 2) | 52.126 | 40.771 |
| Polynormail (Degree 3) | 46.704 | 42.962 |
| Fourier Series, 3 Terms | 59.965 | 71.673 |
| B-Spline (1-Knot) | 54.903 | 71.575 |

We see the fourier series, while being the only continuous function $\left(\lim_{t \to 0^+} f(t) = \lim_{t \to 365^-} f(t)\right)$ the discrepancy between volatilities at early in the year and late in the year means this continuity is lost on all other models.

In [ ]:
```
fig, axs = plt.subplots(1, 1, figsize=(14, 9))
# y_model = fourier_series(x, fourier_coeffs, 3)
y_model, y_predictions = polynomial_regression_fit(3, x, y)
axs.plot(y, label="Volatilities")
axs.plot(y_predictions, label="Model")
axs.set_title(f"Polynomial model of degree {3} terms")
print(y_model.params)
```

```
[ 2.80131071e+00 -2.28205148e-03 -2.98096364e-05  1.08681681e-07]
```




In [ ]:
```
def sigma_derivative(params, x):
    powers = np.arange(1, len(params))  # 1 to len(params)-1
    return np.dot(np.power.outer(x, powers-1), powers * params[1:])
```

In [ ]:
```
def poly_model(params, x):
    powers = np.arange(len(params))
    return np.dot(np.power.outer(x, powers), params)
```

# Skew and Kurtosis

## Skew

The skewness of a dataset or distribution is the measure of its symmetry (or asymmetry). The skewness of a random variable $X \sim p_X$ is given by the quantity

$$\text{Skewness: } \gamma = \mathrm{E}_{p_X}\left[\left(\frac{X-\mu}{\sigma}\right)^3\right]$$

The reason why this represents skewness in a distribution is because: the quantity $\left(\frac{X-\mu}{\sigma}\right)$ represents how many standard deviations the data is around the mean. Upon cubing these quantity, the sign of the value is preserved however the magnitude of values within one standard deviation is diminished and the magnitude of values above one standard deviation is increased. For symmetric diistributions the positive and negative contributions around the mean will cancel, resulting in $0$ skewness. For skewed, asymmetric, distributions, there will be greater contributions from one side of the mean resulting in a non-zero, directional skew.

## Kurtosis

Kurtosis measures the "tailed-ness" or the sharpness of the peak of the distribution. The Kurtosis of the normal distribution (of arbitrary parameters) is $3$. Therefore, $3$ is often subtracted from the Kurtosis to give a statistic relative to the Kurtosis, tailed-ness, of the normal distribution.

- A kurtosis close to 0 indicates a distribution that is relatively similar to the normal distribution in terms of its tails and peak.
- A positive kurtosis indicates a distribution with heavier tails and a sharper peak than the normal distribution.
- A negative kurtosis indicates a distribution with lighter tails and a less sharp peak than the normal distribution.

$$\text{Kurtosis: } \kappa = \frac{\mathrm{E}\left[(X-\mu)^4\right]}{\mathrm{E}\left[(X-\mu)^2\right]^2} = \mathrm{E}\left[\left(\frac{X-\mu}{\sigma}\right)^4\right].$$

This quantity can be interpreted as an average over the fourth power standardised distribution. For data points within one standard deviation of the mean the standardised statistic is less than one. Upon raising this to the fourth power the contribution of the standardised values in this domain are minimal. Therefore the Kurtosis effectively measures how much of the distribution is outside of one standard deviation from the mean.

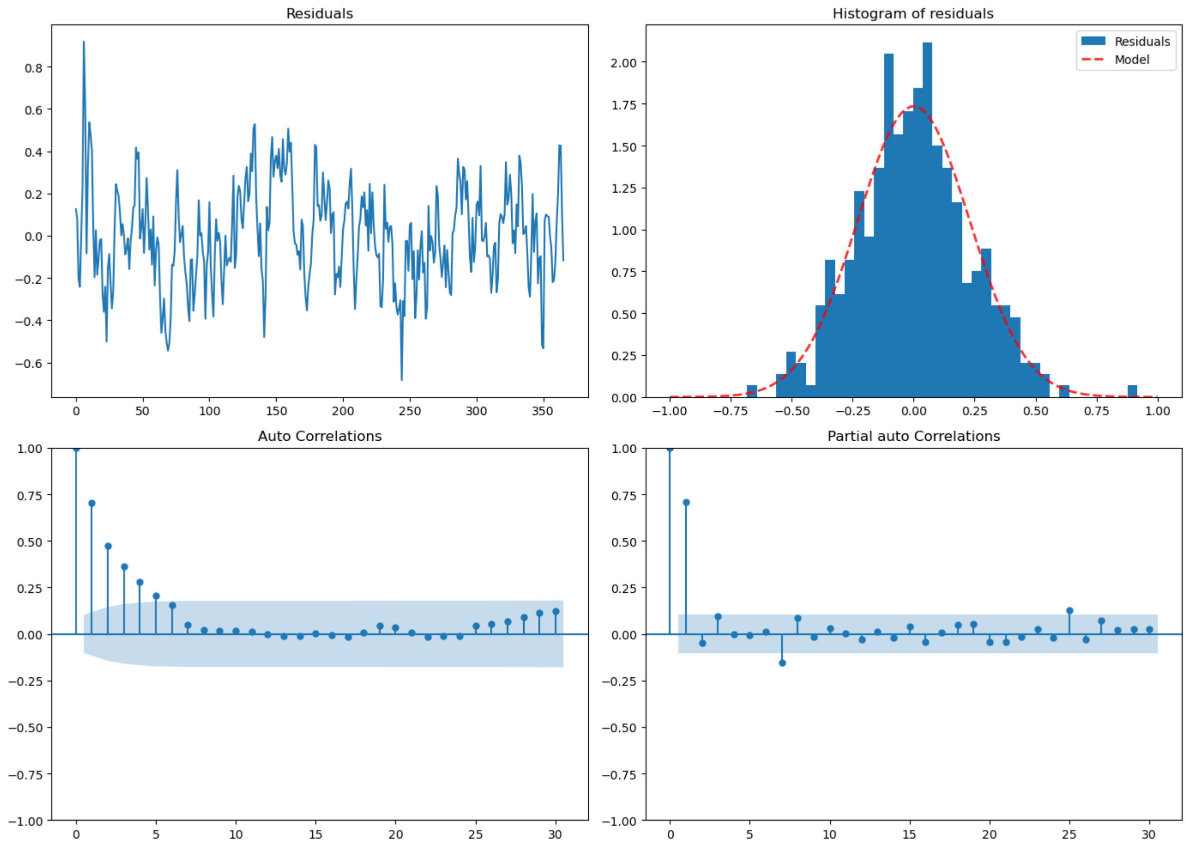
In [ ]:
```
vol_residuals = y - y_predictions

print(f"Mean of residuals of residuals: {np.mean(vol_residuals)}")
print(f"Standard deviation of residuals: {np.std(vol_residuals)}")
print(f"Skewness of residuals: {skew(vol_residuals)}")
print(f"Kurtosis of residuals: {kurtosis(vol_residuals)}")
```

```
Mean of residuals of residuals: 1.0497384989190519e-11
Standard deviation of residuals: 0.22994382706706223
Skewness of residuals: 0.16569367397639861
Kurtosis of residuals: 0.30315890490716635
```

In [ ]:
```
fig, axs = plt.subplots(2, 2, figsize=(14, 10))
axs[0, 0].plot(x, vol_residuals)
axs[0, 0].set_title("Residuals")
```

```
axs[0, 1].hist(vol_residuals, bins=40, stacked=True, density=True, label="R
p = norm.pdf(np.linspace(-1.0, 1.0, 100), np.mean(vol_residuals), np.std(vo
axs[0, 1].plot(np.linspace(-1.0, 1.0, 100), p, 'r', alpha=0.8, linestyle='-
axs[0, 1].set_title("Histogram of residuals")
axs[0, 1].legend()

plot_acf(vol_residuals, lags=30, ax=axs[1, 0])
axs[1, 0].set_title("Auto Correlations")

plot_pacf(vol_residuals, lags=30, ax=axs[1, 1])
axs[1, 1].set_title("Partial auto Correlations")

fig.tight_layout()
fig.show()
```

```
/home/peter/anaconda3/envs/urop-env/lib/python3.11/site-packages/statsmodel
s/graphics/tsaplots.py:348: FutureWarning: The default method 'yw' can prod
uce PACF values outside of the [-1,1] interval. After 0.13, the default wil
l change tounadjusted Yule-Walker ('ywm'). You can use this method now by s
etting method='ywm'.
  warnings.warn(
/tmp/ipykernel_9380/540982911.py:18: UserWarning: Matplotlib is currently u
sing module://matplotlib_inline.backend_inline, which is a non-GUI backend,
so cannot show the figure.
  fig.show()
```




# Now we aim to model the volatility of the volatility, $\chi$

First we check if the distribution of residuals this is Gaussian, if so we can use a constant to model to dispersion of volatilities around the annual trend we have modelled above.

# Kolmogorov-Smirnov and Anderson-Darling Tests

## Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov test is a nonparametric test which determines if two datasets come from a particular theoretical distribution. The K-S test quantifies the distance between the empirical distribution function (EDF) of the samples and the cumulative distribution function (CDF) of the reference distribution or between the EDFs of two samples.

The test statistic is the maximum absolute difference between the EDF and the CDF:

$$\text{K-S test statistic: } D = \max_{x \in \text{Samples}} |F_n(x) - F(x)|$$

In [ ]:
```
D, p_value = ks_2samp(vol_residuals, np.random.normal(0, np.std(vol_residua

print(f"D-statistic: {D}")
print(f"P-value: {p_value}")

if p_value > 0.99:
    print("Two samples seem to be drawn from the same distribution (fail to
else:
    print("Two samples seem to be drawn from different distributions (rejec
```

```
D-statistic: 0.047382513661202184
P-value: 0.5642800683623818
Two samples seem to be drawn from different distributions (reject H0)
```

## Anderson-Darling Test

In [ ]:
```
standardized_vol_residuals = vol_residuals / np.std(vol_residuals)

# Perform Anderson-Darling test for normality on the standardized data
result = anderson(standardized_vol_residuals)

print(f'Anderson-Darling Statistic: {result.statistic}')

for i in range(len(result.critical_values)):
    sl, cv = result.significance_level[i], result.critical_values[i]
    if result.statistic < cv:
        print(f'At {sl}% significance level, data looks normal (fail to rej
    else:
        print(f'At {sl}% significance level, data does not look normal (rej
```

```
Anderson-Darling Statistic: 0.2946336256512154
At 15.0% significance level, data looks normal (fail to reject H0)
At 10.0% significance level, data looks normal (fail to reject H0)
At 5.0% significance level, data looks normal (fail to reject H0)
At 2.5% significance level, data looks normal (fail to reject H0)
At 1.0% significance level, data looks normal (fail to reject H0)
```

Therefore, it appears the residuals of the volatility are Gaussian.

# Final model of the volatility

$$dS_t = \left(\dot{\sigma}(t) + \alpha_2(\sigma(t) - S_t)\right)dt + \chi dW_t$$

As reasoned previously, we can find a value of $\alpha_2$ by fitting an AR-1 model to the residuals.

# Modelling the rate of mean reversion, $\alpha_2$

We can model the speed of reversion by first considering the AR(1) process.

$$S_t = w + \theta S_{t-1} + \epsilon_t$$

The Ornsten-Ulhenbeck process can be interpreted as a continuous time analogue of the residuals of the AR(1) process.

Considering the Euler-Maryuama discretisation of our modified OU process over the interval $t \in [i-1, i]$, we find:

$$dS_t = S_i - S_{i-1} = \sigma(i) - \sigma(i-1) + \alpha\left(\sigma_{i-1} - S_{i-1}\right) + \chi dW_1$$
$$S_i - \sigma(t_i) = S_{i-1} - \sigma(i-1) + \alpha_2\left(\sigma(i-1) - S_{i-1}\right) + \chi dW_1$$
$$R_i = R_{i-1}\left(1 - \alpha_2\right) + \epsilon_i$$

Which is an AR(1) model of the residuals. As we saw from the partial auto correlation of our residuals and AR(1) model is appropriate for modelling the residuals. Therefore, we find that $1 - \alpha_2 = \theta$ in the AR(1) model of the residuals. This is how we can determine the rate of mean reversion, $\alpha_2$.

In [ ]:
```
model_fit = sm.tsa.AutoReg(vol_residuals, lags=1, old_names=True,trend='n')
print(model_fit.summary())
```

```
                            AutoReg Model Results
==============================================================================
===
Dep. Variable:                      y   No. Observations:
366
Model:                     AutoReg(1)   Log Likelihood                 144.
545
Method:               Conditional MLE   S.D. of innovations              0.
163
Date:                Fri, 25 Aug 2023   AIC                           -285.
090
Time:                        10:32:05   BIC                           -277.
290
Sample:                             1   HQIC                          -281.
991
                                  366
==============================================================================
===
                 coef    std err          z      P>|z|      [0.025      0.9
75]
------------------------------------------------------------------------------
---
y.L1           0.7067      0.037     19.083      0.000       0.634       0.
779
                                    Roots
==============================================================================
==
                  Real          Imaginary           Modulus         Frequen
cy
------------------------------------------------------------------------------
--
AR.1            1.4151           +0.0000j            1.4151           0.00
00
------------------------------------------------------------------------------
--
```

```
/home/peter/anaconda3/envs/urop-env/lib/python3.11/site-packages/statsmodel
s/tsa/ar_model.py:233: FutureWarning: old_names will be removed after the
0.14 release. You should stop setting this parameter and use the new names.
  warnings.warn(
```

# Final SDE model for temperature:

Finally, we can present our model for temperature modelling:

$$\begin{pmatrix} dT_t \\ dS_t \end{pmatrix} = \begin{pmatrix} \dot{\mu}(t) + \alpha_1(\mu(t) - T_t) \\ \dot{\sigma}(t) + \alpha_2(\sigma(t) - S_t) \end{pmatrix} dt + \begin{pmatrix} S_t & 0 \\ 0 & \chi \end{pmatrix} \begin{pmatrix} dW_t^{(1)} \\ dW_t^{(2)} \end{pmatrix}$$

# Monte-Carlo Simulations