Lecture 24

Time Series Decomposition

Time series analysis begins from some very basic tools. We want to dig into the operations of these functions a bit. We're going to look at some basic time series processes such as differencing, moving averages, lags and autoregression, and time series decomposition methods.

Time series data refers to a sequence of observations recorded at specific time intervals, typically evenly spaced (e.g., daily, monthly, yearly). Analyzing time series data involves understanding the underlying patterns, trends, and seasonality to make forecasts or gain insights.

Key Concepts

1. *Stationarity*:
    - A time series is **stationary** if its statistical properties, like mean, variance, and autocorrelation, are constant over time.
    - **Non-stationary** series often exhibit trends, seasonality, or other time-dependent structures.
    - Stationarity is crucial because many time series forecasting methods, such as ARIMA, assume stationarity.
    - **Tests for Stationarity**: The Augmented Dickey-Fuller (ADF) test and the KPSS test are commonly used to check for stationarity.

2. ***Decomposition***:
    - Time series can often be decomposed into several components:
        - *Trend*: The long-term progression in the data (e.g., upward or downward movement).
        - *Seasonality*: Regular, repeating patterns in the data, usually within a year (e.g., monthly sales peaks).
        - *Cyclic*: Similar to seasonality but not of a fixed period. Cycles can be irregular and longer-term.
        - *Residual*: The remaining variability in the data after removing trend and seasonality (essentially, the noise).
    - *Additive Model*: Assumes the components add up (i.e., $y_t = Trend_t + Seasonality + t + Residual_t$).
    - *Multiplicative Model*: Assumes the components multiply (i.e., $y_t = Trend_t \times Seasonality_t \times Residual_t$).
    - Decomposition helps in understanding the structure of the data and can assist in making it stationary.

3. ***ARIMA Models***:
    - *ARIMA* stands for **AutoRegressive Integrated Moving Average**. It's a popular model for forecasting time series data, especially when the data shows evidence of non-stationarity.
    - *Components*:
        - *AR (AutoRegressive):* Models the dependency between an observation and some number of lagged observations (past values).

        - *I (Integrated):* Represents the differencing required to make the series stationary.
        - *MA (Moving Average):* Models the dependency between an observation and a residual error from a moving average model applied to lagged observations.
    - *Parameters*:
        - **p**: Number of lag observations in the model (AR part).
        - **d**: Number of times the data needs to be differenced to achieve stationarity (I part).
        - **q**: Size of the moving average window (MA part).
    - *Seasonal ARIMA (SARIMA):* An extension of ARIMA that explicitly supports univariate time series data with a seasonal component.

Example Workflow in Time Series Analysis

1. *Exploratory Data Analysis (EDA):* Plot the time series data to visually inspect for trends, seasonality, and other patterns. Check for stationarity using ADF or KPSS tests.

2. *Decomposition*: Decompose the time series to identify trend, seasonal, and residual components.

3. *Transformation*: If the series is not stationary, apply transformations (like differencing, log transformation) to achieve stationarity.

4. *Modeling*: Use models like ARIMA for forecasting. Tune model parameters (p, d, q) based on the data (using techniques like AIC/BIC for model selection).

5. *Validation*: Evaluate the model using techniques like cross-validation and check forecast accuracy metrics (e.g., RMSE, MAE).

6. *Forecasting*: Use the model to make future predictions.

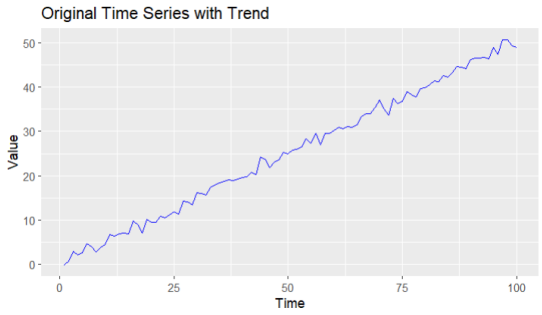
Visualization Techniques

- *Time Series Plot*: Basic plot to visualize the time series data.
- *Decomposition Plot*: Shows the trend, seasonal, and residual components.
- *ACF and PACF Plots*: Autocorrelation and partial autocorrelation plots help in identifying the order of AR and MA models.

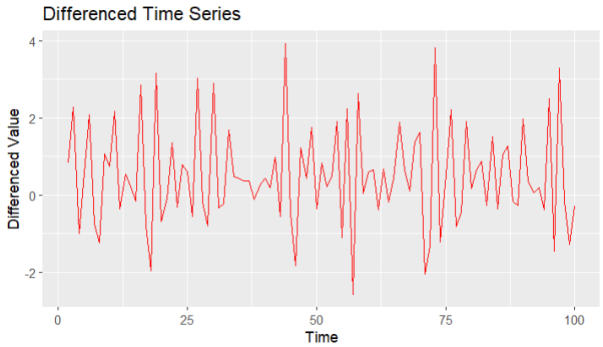
Let's start with differencing and checking for stationarity.

```
# Load necessary libraries
library(ggplot2)
library(forecast)  # For the stationarity test

# Set seed for reproducibility
set.seed(123)

# Generate a simple non-stationary time series (e.g., with a trend)
n <- 100
time <- 1:n
```

```
trend <- 0.5 * time
noise <- rnorm(n, mean = 0, sd = 1)
ts_data <- trend + noise

# Convert to a time series object
ts_data <- ts(ts_data)

# Plot the original time series
ggplot() +
  geom_line(aes(x = time, y = ts_data), color = "blue") +
  ggtitle("Original Time Series with Trend") +
  xlab("Time") +
  ylab("Value")
```




```
# Manually calculate the first difference
diff_ts <- numeric(length(ts_data) - 1)

for (i in 2:length(ts_data)) {
  diff_ts[i - 1] <- ts_data[i] - ts_data[i - 1]
}

# Plot the differenced time series
ggplot() +
  geom_line(aes(x = time[-1], y = diff_ts), color = "red") +
  ggtitle("Differenced Time Series") +
  xlab("Time") +
  ylab("Differenced Value")
```



ADF Test:

- The ADF test checks whether a unit root is present in the series (i.e., the series is non-stationary).

- If the p-value is below a certain threshold (commonly 0.05), we reject the null hypothesis of a unit root and conclude that the series is stationary.

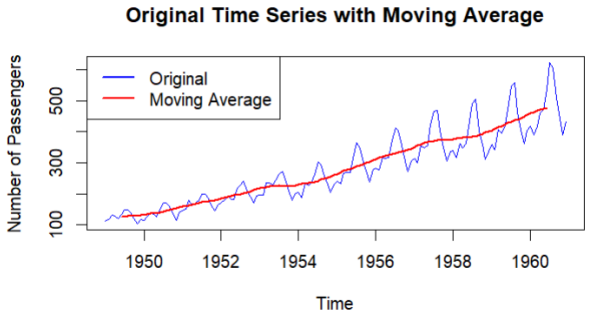
```
###############################################
# Augmented Dickey-Fuller Test Unit Root Test #
###############################################

Test regression drift


Call:
lm(formula = z.diff ~ z.lag.1 + 1 + z.diff.lag)

Residuals:
     Min       1Q   Median       3Q      Max
-3.00066 -0.61741 -0.02911  0.55117  2.96017

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.06722    0.13099   8.147 1.56e-12 ***
z.lag.1     -2.16704    0.15521 -13.962  < 2e-16 ***
z.diff.lag   0.48155    0.09093   5.296 7.77e-07 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.027 on 94 degrees of freedom
Multiple R-squared:  0.7955,    Adjusted R-squared:  0.7912
F-statistic: 182.9 on 2 and 94 DF,  p-value: < 2.2e-16


Value of test-statistic is: -13.9616 97.536

Critical values for test statistics:
      1pct  5pct 10pct
tau2 -3.51 -2.89 -2.58
phi1  6.70  4.71  3.86
```

Since the test statistic is much less than the critical value at the 5% level, we can conclude that the series is now stationary. If you need to perform a second differencing, if the series is not yet stationary, you can, just replace the differenced series where the original series was in the code. You almost never need to go past two differences.

Let’s look at finding a moving average and plotting it on our graph. Moving average is calculated from averaging together nearby values, and the window moves as the graph continues. This helps to smooth the random errors out and give us a better view of things like trends.

```
# Simulate or load a time series (for example, we'll use the AirPassengers dataset)
data("AirPassengers")
time_series <- AirPassengers

# Parameters for the moving average
window_size <- 12 # 12-month (1-year) moving average

# Calculate the moving average
moving_average <- filter(time_series, rep(1/window_size, window_size), sides = 2)

# Plot the original time series
```

```
plot(time_series, main="Original Time Series with Moving Average", col="blue", ylab="Number of Passengers", xlab="Time")

# Add the moving average to the plot
lines(moving_average, col="red", lwd=2)

# Add a legend
legend("topleft", legend=c("Original", "Moving Average"), col=c("blue", "red"), lwd=2)
```



- *filter() Function*: The filter() function is used to apply a linear filter to a univariate time series. The rep(1/window_size, window_size) creates a vector of equal weights for the window.
- *sides=2*: This option centers the moving average, using data from both sides of each point. You can set sides=1 for a trailing moving average.
- *plot() and lines():* The original series is plotted using plot(), and the moving average is added with lines().

Moving averages can also be more sophisticated than a simple average, which may be needed in particular contexts. An **Exponential Moving Average (EMA)** is a type of moving average that gives more weight to recent data points, making it more responsive to recent price changes compared to a simple moving average. The EMA is often used in time series analysis, especially in financial markets, to smooth data and identify trends while maintaining a focus on more recent observations.

How EMA Works:

1. *Weighting*: Unlike a simple moving average (SMA) which assigns equal weight to all observations within the window, the EMA assigns exponentially decreasing weights as the observations get older. The most recent data points are given the highest weight.

2. *Smoothing Factor (α):* The degree of weighting decrease is expressed as a smoothing factor, α, which lies between 0 and 1. The formula for α depends on the period (window size) N chosen for the EMA:
$$\alpha = \frac{2}{N+1}$$
Here, N is the number of time periods over which the EMA is calculated.

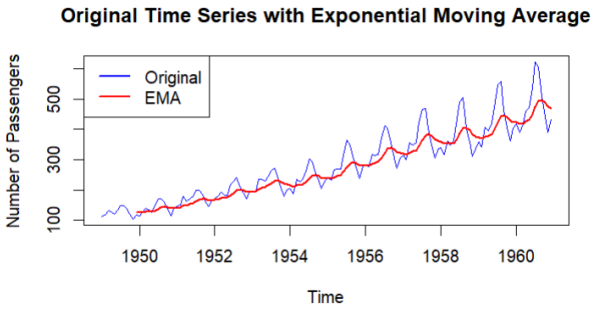
3. *EMA Formula*: The EMA at time t (EMA_t) can be calculated as:
$$EMA_t = \alpha Y_t + (1-\alpha) EMA_{t-1}$$
Where Y_t is the value of the time series at time t, and EMA_{t-1} is the EMA calculated for the previous time period.

The initial EMA value, EMA_0, is typically set to the first observation in the time series.

When is EMA Used?

- *Trend Analysis*: EMA is widely used in financial markets to analyze price trends and signals. Traders often use EMAs to generate buy and sell signals based on the crossover of different EMA periods (e.g., 50-day EMA crossing above 200-day EMA).
- *Responsive Indicator*: EMA is more responsive to recent data than the SMA, making it a better choice when the analyst wants to emphasize recent changes in the data while still smoothing out noise.
- *Time Series Forecasting*: EMA is sometimes used in time series forecasting as part of exponential smoothing techniques.

```
# Load necessary package
if(!require(TTR)) install.packages("TTR", dependencies=TRUE)
library(TTR)

# Simulate or load a time series (for example, we'll use the AirPassengers dataset)
data("AirPassengers")
time_series <- AirPassengers

# Calculate the Exponential Moving Average
ema <- EMA(time_series, n = 12)  # 12-month EMA

# Plot the original time series
plot(time_series, main="Original Time Series with Exponential Moving Average", col="blue",
ylab="Number of Passengers", xlab="Time")

# Add the Exponential Moving Average to the plot
lines(ema, col="red", lwd=2)

# Add a legend
legend("topleft", legend=c("Original", "EMA"), col=c("blue", "red"), lwd=2)
```



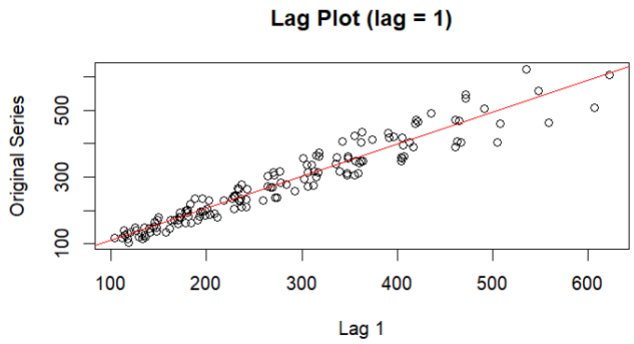
A lagged time series is simply a shifted version of the original time series. For example, a lag of 1 means that each value in the time series is shifted by one time step.

```
# Example time series data (we'll use the AirPassengers dataset)
data("AirPassengers")
time_series <- AirPassengers
```

```
# Function to calculate lags
calculate_lags <- function(series, lag = 1) {
  # Create a vector of NA values to shift the series
  lagged_series <- c(rep(NA, lag), series[1:(length(series) - lag)])
  return(lagged_series)
}

# Calculate a lag of 1 for the time series
lag_1 <- calculate_lags(time_series, lag = 1)

# Display the original and lagged series
head(cbind(time_series, lag_1), 10)
```

A lag plot is a scatter plot of the time series against a lagged version of itself. It helps in visualizing the relationship between the series and its lag.

```
# Create a lag plot (lag 1)
plot(lag_1, time_series, main="Lag Plot (lag = 1)", xlab="Lag 1", ylab="Original Series")
abline(lm(time_series ~ lag_1), col="red") # Adding a regression line
```



Autocorrelation measures the correlation between the time series and a lagged version of itself. It can be calculated manually using the following steps:

```
# Function to calculate autocorrelation for a given lag
calculate_autocorrelation <- function(series, lag) {
  lagged_series <- calculate_lags(series, lag)
  # Remove NA values
  valid_indices <- !is.na(lagged_series)
  series <- series[valid_indices]
  lagged_series <- lagged_series[valid_indices]

  # Calculate the autocorrelation
  n <- length(series)
  mean_series <- mean(series)
  mean_lagged <- mean(lagged_series)
```

```
  numerator <- sum((series - mean_series) * (lagged_series - mean_lagged))
  denominator <- sqrt(sum((series - mean_series)^2) * sum((lagged_series - mean_lagged)^2))

  autocorrelation <- numerator / denominator
  return(autocorrelation)
}

# Calculate autocorrelation for lag 1
autocorrelation_1 <- calculate_autocorrelation(time_series, lag = 1)
autocorrelation_1
```

Autoregression (AR) is a model where the current value of the series is regressed on its past values (lags).

```
# Function to fit a simple AR(1) model
fit_ar_model <- function(series, lag = 1) {
  lagged_series <- calculate_lags(series, lag)
  valid_indices <- !is.na(lagged_series)

  # Use linear regression to fit the model
  ar_model <- lm(series[valid_indices] ~ lagged_series[valid_indices])
  return(ar_model)
}

# Fit an AR(1) model
ar_model <- fit_ar_model(time_series, lag = 1)

# Display the summary of the model
summary(ar_model)

# Make predictions based on the model
predictions <- predict(ar_model)

# Plot original vs predicted values
plot(time_series, type = "l", col = "blue", main = "AR(1) Model Predictions vs Original Series")
lines(c(NA, predictions), col = "red")  # Lag causes first prediction to be NA
legend("topleft", legend=c("Original", "Predicted"), col=c("blue", "red"), lty=1)
```

When we construct an ACF plot, we do this many times, calculating lags, calculating correlations, and then plotting the correlations. Now, let's consider partial autocorrelations.

Partial autocorrelation measures the correlation between a time series and its lagged values after removing the effect of the intervening lags. It's a more precise measure of correlation for each lag, isolating the direct relationship at that lag.

The PACF can be computed using the **Yule-Walker equations** or by fitting AR models.

Steps to Calculate PACF

1. *Calculate Autocorrelations (ACF):* First, we need to calculate the autocorrelations for lags 1, 2, and 3.
2. *Solve the Yule-Walker Equations:*
    - For PACF at lag 1, it's just the ACF at lag 1.
    - For PACF at lag 2, we adjust the ACF at lag 2 using the ACF at lag 1.
    - For PACF at lag 3, we adjust the ACF at lag 3 using the ACFs at lag 1 and 2.

Yule-Walker Equations

Given a time series $\{X_t\}$ that follows an autoregressive process of order $p\big(AR(p)\big)$, the Yule-Walker equations relate the autocorrelation function (ACF) to the parameters (coefficients) of the AR model. For an $AR(p)$ process:

$$X_t = \phi_1 X_{t-1} + \phi_2 X_{t-2} + \cdots + \phi_p X_{t-p} + \varepsilon_t$$

where $\phi_1, \phi_2, \ldots, \phi_p$ are the coefficients of the AR model, and ε_t is white noise.

Autocovariance Form

The Yule-Walker equations can be expressed in terms of autocovariances γ_k of the time series. The autocovariance at lag k is defined as:

$$\gamma_k = Cov(X_t, X_{t-k})$$

The Yule-Walker equations for an $AR(p)$ model are:

1. For $k = 1$:
$$\gamma_1 = \phi_1 \gamma_0 + \phi_2 \gamma_1 + \cdots + \phi_p \gamma_{p-1}$$
2. For $k = 2$:
$$\gamma_2 = \phi_1 \gamma_1 + \phi_2 \gamma_0 + \cdots + \phi_p \gamma_{p-2}$$

…

3. p-th equation:
$$\gamma_p = \phi_1 \gamma_{p-1} + \phi_2 \gamma_{p-2} + \cdots + \phi_p \gamma_0$$

These equations can be written in matrix form:

$$\begin{bmatrix} \gamma_1 \\ \gamma_2 \\ \vdots \\ \gamma_p \end{bmatrix} = \begin{bmatrix} \gamma_0 & \gamma_1 & \cdots & \gamma_{p-1} \\ \gamma_1 & \ddots & \ddots & \gamma_{p-2} \\ \vdots & \ddots & \ddots & \vdots \\ \gamma_{p-1} & \gamma_{p-2} & \cdots & \gamma_0 \end{bmatrix} \begin{bmatrix} \phi_1 \\ \phi_2 \\ \vdots \\ \phi_p \end{bmatrix}$$

In compact notation, this can be written as:

$$\Gamma_p = \Gamma \Phi$$

Where:

- Γ is the autocovariance matrix.
- Φ is the vector of AR coefficients.
- Γ_p is the vector of autocovariances at lags 1 to p.

Autocorrelation Form

Alternatively, the Yule-Walker equations can be expressed in terms of autocorrelations $\rho_k = \frac{\gamma_k}{\gamma_0}$:

$$\rho_k = \phi_1 \rho_{k-1} + \phi_2 \rho_{k-2} + \cdots + \phi_p \rho_{k-p}$$

The matrix form is similar, but with autocorrelations instead of autocovariances.

PACF from Yule-Walker Equations

The partial autocorrelation coefficient ϕ_{kk} at lag k can be derived as the solution of the k-th Yule-Walker equation:

$$\phi_{kk} = \frac{\gamma_k - \sum_{i=1}^{k-1} \phi_{ki}\gamma_{k-i}}{\gamma_0 - \sum_{i=1}^{k-1} \phi_{ki}\gamma_i}$$

This formula is recursive, where ϕ_{ki} are the coefficients of the AR model estimated at previous lags.

Each of these components, differencing, moving average and autocorrelation go into choosing the parameters for the ARIMA model. Recall that the number of differences is the integrated component, the number of significant autocorrelations goes into the moving average component, and the number of significant partial autocorrelations goes into the autocorrelation component of the ARIMA model.

Finally, we want to look at decomposing the time series into trend, seasonal and random components. Before starting the decomposition, it is essential to identify whether your time series follows an **additive** or **multiplicative** model.

- *Additive Model*:
  The series is decomposed as:

$$X_t = T_t + S_t + R_t$$

where X_t is the observed value, T_t is the trend component, S_t is the seasonal component, and R_t is the random (residual) component.

- *Multiplicative Model:*
  The series is decomposed as:

$$X_t = T_t \times S_t \times R_t$$

This model is used when the variation in the seasonal component is proportional to the level of the trend.

Calculate the Moving Average (Trend Component)

The trend component can be estimated using a moving average. The idea is to smooth out the short-term fluctuations to reveal the long-term trend.

- *Choose the window size*: This typically corresponds to the length of one seasonal cycle. For example, if the data has a monthly seasonality, use a window size of 12.
- *Compute the moving average*: For each point in the series, calculate the average of the points in the window centered on that point.

Estimate the Seasonal Component

Once the trend component is removed, the seasonal component can be estimated by averaging the deviations of the data from the trend.

- *Detrend the series*: Subtract the trend component from the original series to obtain the detrended series (for the additive model). For the multiplicative model, divide the original series by the trend component.
- *Compute the seasonal indices*: For each season (e.g., each month if monthly data), average the detrended values. This gives an estimate of the seasonal component for each time point in the season.

Compute the Residual Component

The residual component is what's left after removing the trend and seasonal components from the original series.

- *Calculate the residual*: For the additive model, subtract the estimated trend and seasonal components from the original series. For the multiplicative model, divide the original series by the product of the trend and seasonal components.

Reconstruct the Components

To ensure the decomposition makes sense, it's often useful to visualize the original series along with the reconstructed series obtained by adding (or multiplying in the multiplicative case) the trend, seasonal, and residual components.

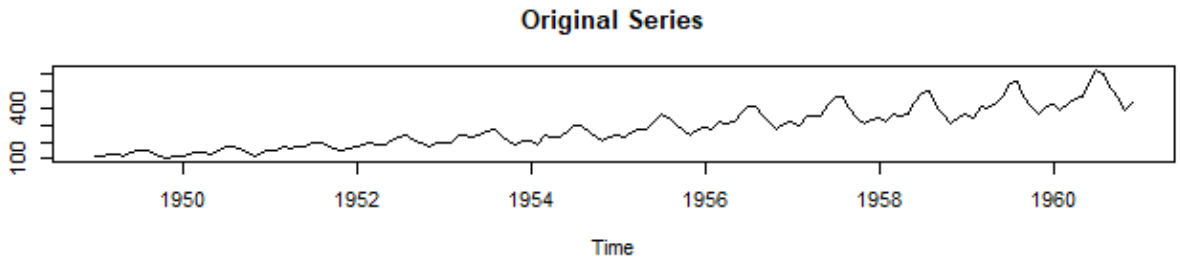
Visualize the Results

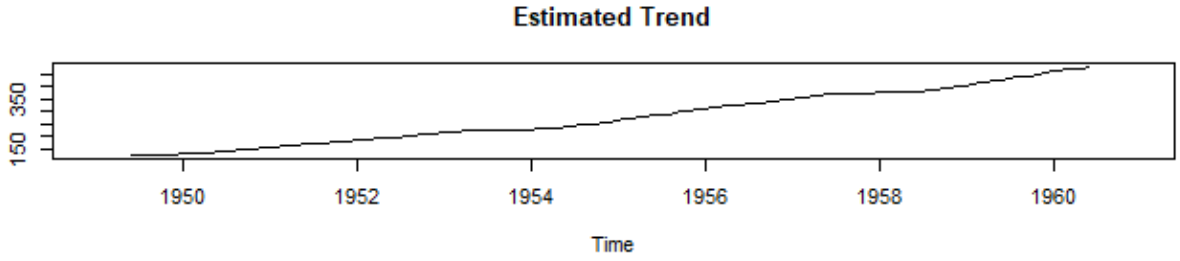
Plot the original series and its decomposed components (trend, seasonal, and residual) to examine the quality of the decomposition.

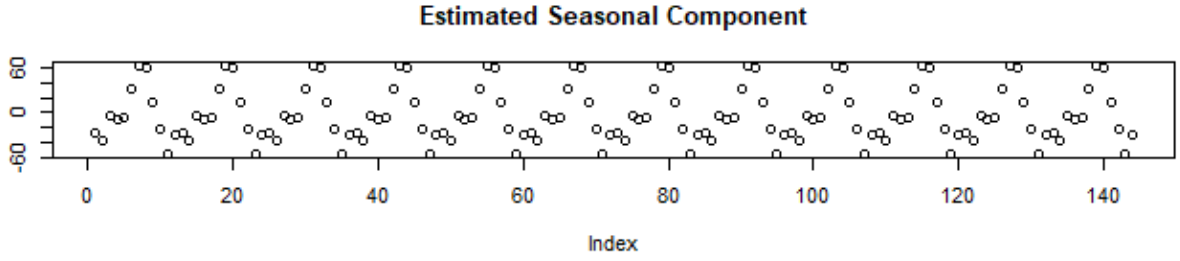
Let's look at how this works for the additive model.

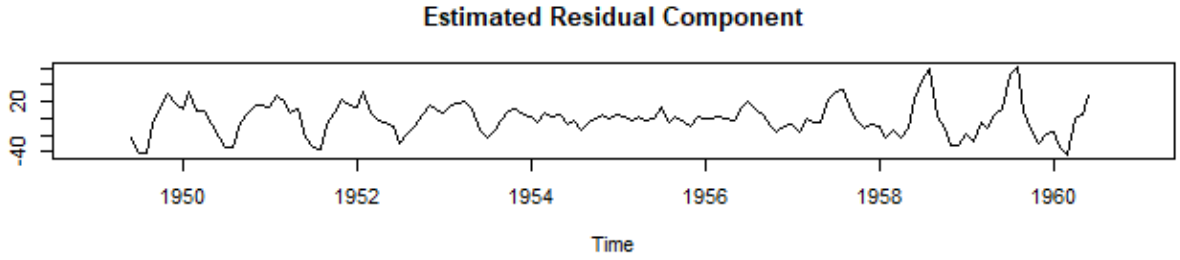
```
# Example: Resetting the Graphics Device and Resizing the Plot Panel
dev.off()  # Reset graphics device
windows()  # Opens a new plotting window (use quartz() on macOS, x11() on Linux)

# Load the AirPassengers dataset
data(AirPassengers)
time_series <- AirPassengers

# Step 1: Visualize the Original Series
plot(time_series, main = "Original AirPassengers Time Series")

# Step 2: Estimate Trend Component with Moving Average
window_size <- 12  # 12 months in a year
trend <- stats::filter(time_series, rep(1/window_size, window_size), sides=2)

# Step 3: Detrend the Series to Estimate Seasonal Component
detrended <- time_series - trend

# Calculate Seasonal Indices
seasonal <- tapply(detrended, cycle(time_series), mean, na.rm=TRUE)

# Expand the seasonal component to the length of the time series
seasonal_full <- rep(seasonal, length.out=length(time_series))

# Step 4: Calculate Residual Component
residual <- time_series - trend - seasonal_full

# Step 5: Visualize the Decomposition
par(mfrow=c(4,1))
plot(time_series, main="Original Series", ylab="")
```

```
plot(trend, main="Estimated Trend", ylab="")
plot(seasonal_full, main="Estimated Seasonal Component", ylab="")
plot(residual, main="Estimated Residual Component", ylab="")
```









Notes:

- Trend Component: The trend is estimated using a centered moving average, with a window size of 12 to account for annual seasonality.
- Seasonal Component: We detrend the series by subtracting the trend, then compute seasonal indices by averaging the detrended values within each season (month).
- Residual Component: The residuals are what remains after removing the trend and seasonal components.

Interpretation:

- The original series shows the overall pattern in the data.

- The trend component represents the long-term upward trend in the number of air passengers.
- The seasonal component highlights regular seasonal fluctuations in the data.
- The residual component captures random noise or any remaining irregularities not explained by the trend and seasonality.

Considerations for a Multiplicative Model

For a multiplicative model, the approach is similar but involves dividing instead of subtracting when removing components:

- *Detrend the series*: $detrended_t = \frac{X_t}{T_t}$
- *Compute residual*: $R_t = \frac{X_t}{T_t \times S_t}$

Built-in functions in R (e.g., decompose, stl) automate this process.

Resources:

1. https://a-little-book-of-r-for-time-series.readthedocs.io/en/latest/src/timeseries.html