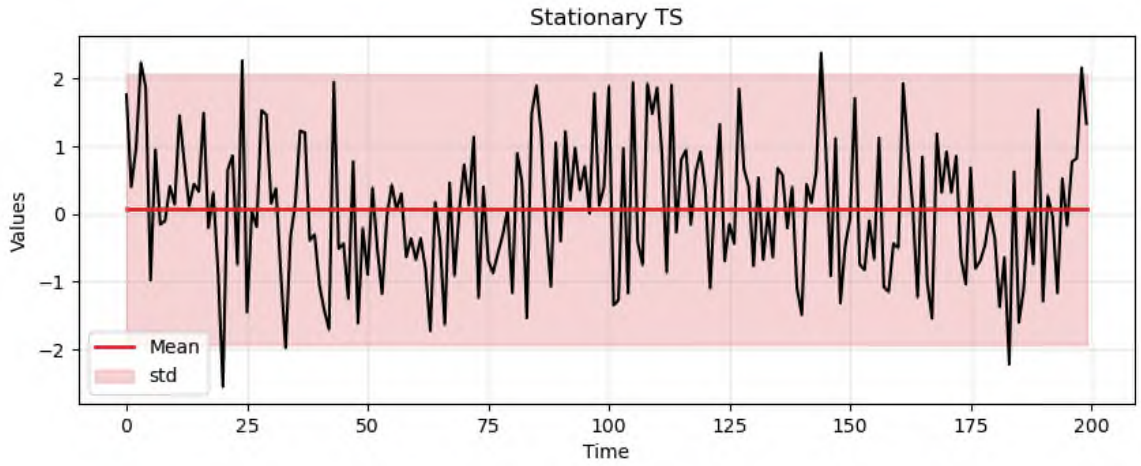
Time series analysis: Stationarity & Smoothing

ΕΠΛ 428: IOT PROGRAMMING

Dr. Panayiotis Kolios
Assistant Professor, Dept. Computer Science,
KIOS CoE for Intelligent Systems and Networks
Office: FST 01, 116
Telephone: +357 22893450 / 22892695
Web: https://www.kios.ucy.ac.cy/pkolios/

Πανεπιστήμιο Κύπρου

- Stationarity
  - Signal from a system without underlying changes
  - Strict / Weak stationarity
- Example of common stationary
  - White noise (mean 0 and variance 1)



- How to detect stationarity
- Common ways to transform nonstationary time series into stationary ones

- A stochastic process X(t): t∈T is called strictly stationary if, $\forall$ $t_1, t_2, \ldots, t_n \in T$ and $h \in T$, $t_1+h, t_2+h, \ldots, t_n+h \in T$, holds:

$$\big(X(t_1), X(t_2), \ldots, X(t_n)\big) = \big(X(t_1+h), X(t_2+h), \ldots, X(t_n+h)\big)$$

- The joint distribution of any set of observations in the series remains the same regardless of the time at which the observations are taken

- A time series is weakly stationary if:
  1. Mean constant over time $E[X(t)] = m, \forall t \in T$
  2. Variance is finite $E[X(t)^2] < \infty, \forall t \in T$
  3. Covariance of $X(t)$ and $X(t+h)$ depends only on h

- Autocorrelation measures how much the current time series measurement is correlated with a past measurement
  - e.g. today's temperature usually highly correlated with yesterday's temperature
- In general, computing the correlation of the time series with versions of the signal delayed up to N lags:

$$R_{xx}(t_1, t_2) = E[X(t_1)X(t_2)]$$

- Covariance of $X(t_1)$ and $X(t_2)$ is called autocovariance

$$C_{xx}(t_1, t_2) = E\left[(X(t_1) - \mu_{t_1})(X(t_2) - \mu_{t_2})\right]$$
$$= R_{xx}(t_1, t_2) - \mu_{t_1}\mu_{t_2}$$

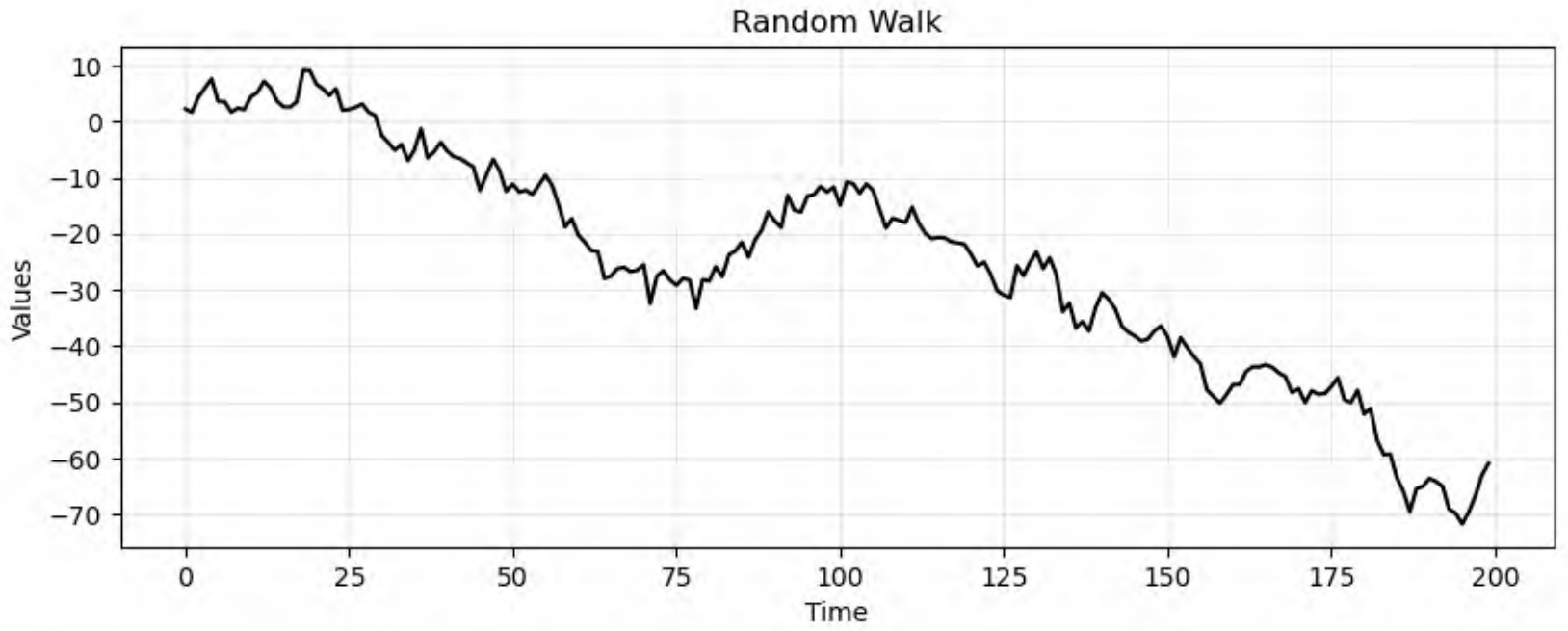
- If $X(t)$ has zero-mean, autocorrelation and autocovariance are the same

- The random walk is one of the most important nonstationary time series

$$X(t) = X(t-1) + \varepsilon_t$$

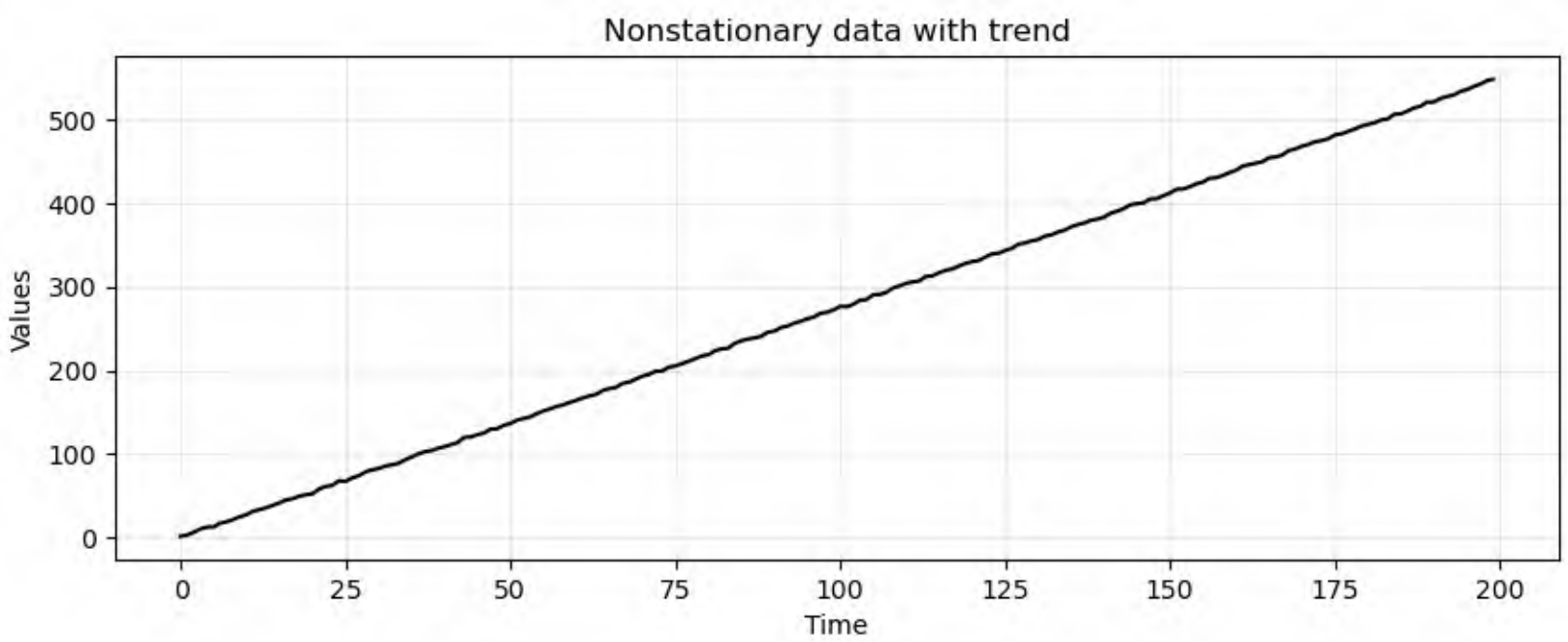
- ε_t are called innovations and are iid, e.g. $\varepsilon_t \sim N(0, \sigma^2)$

```
# seed to start series
seed = 3.14

# Random Walk
rand_walk = np.empty_like(time, dtype='float')
for t in time:
    rand_walk[t] = seed + np.random.normal(loc=0, scale=2.5, size=1)[0]
    seed = rand_walk[t]
```

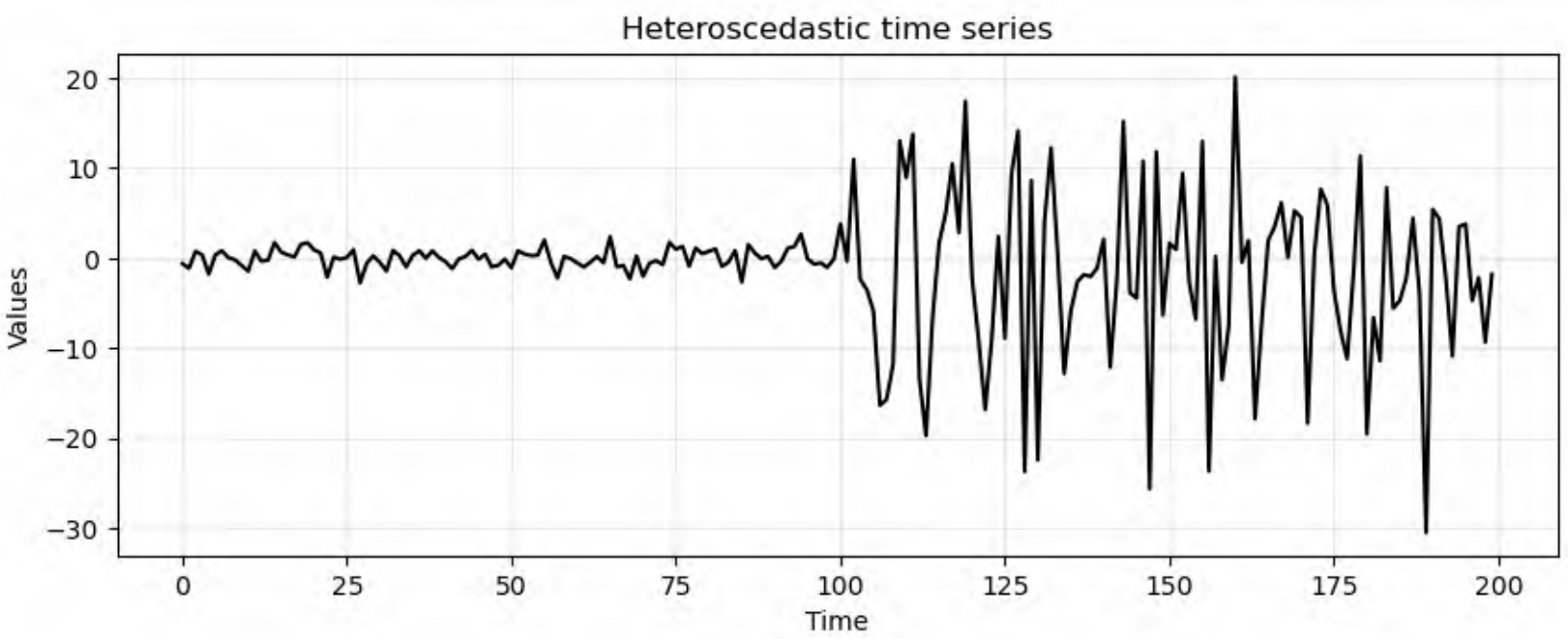


- Current value depends on its initial value and the sum of all previous innovations
- Variance changes over time (increases linearly over time)

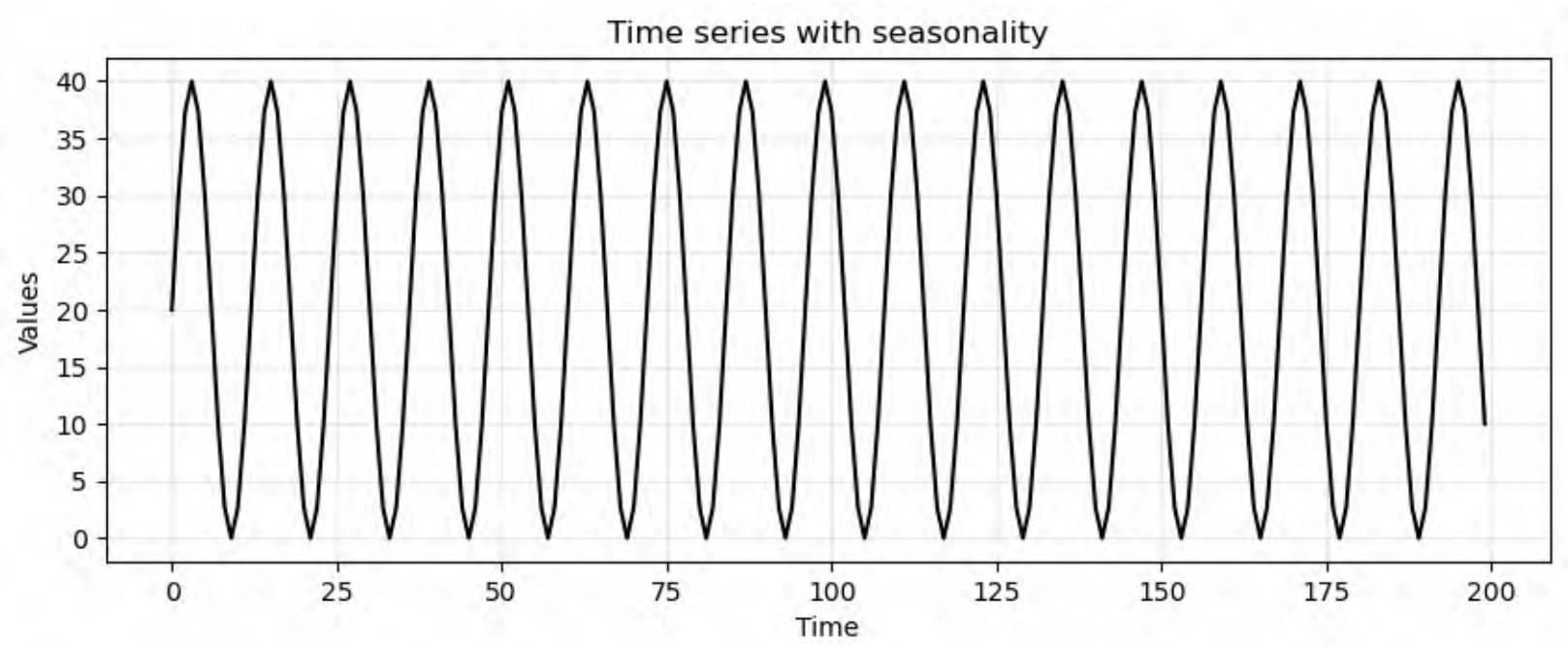
$$Var\big(X(t)\big) = Var(\varepsilon_1) + Var(\varepsilon_2) + \cdots + Var(\varepsilon_t) = t\sigma^2$$



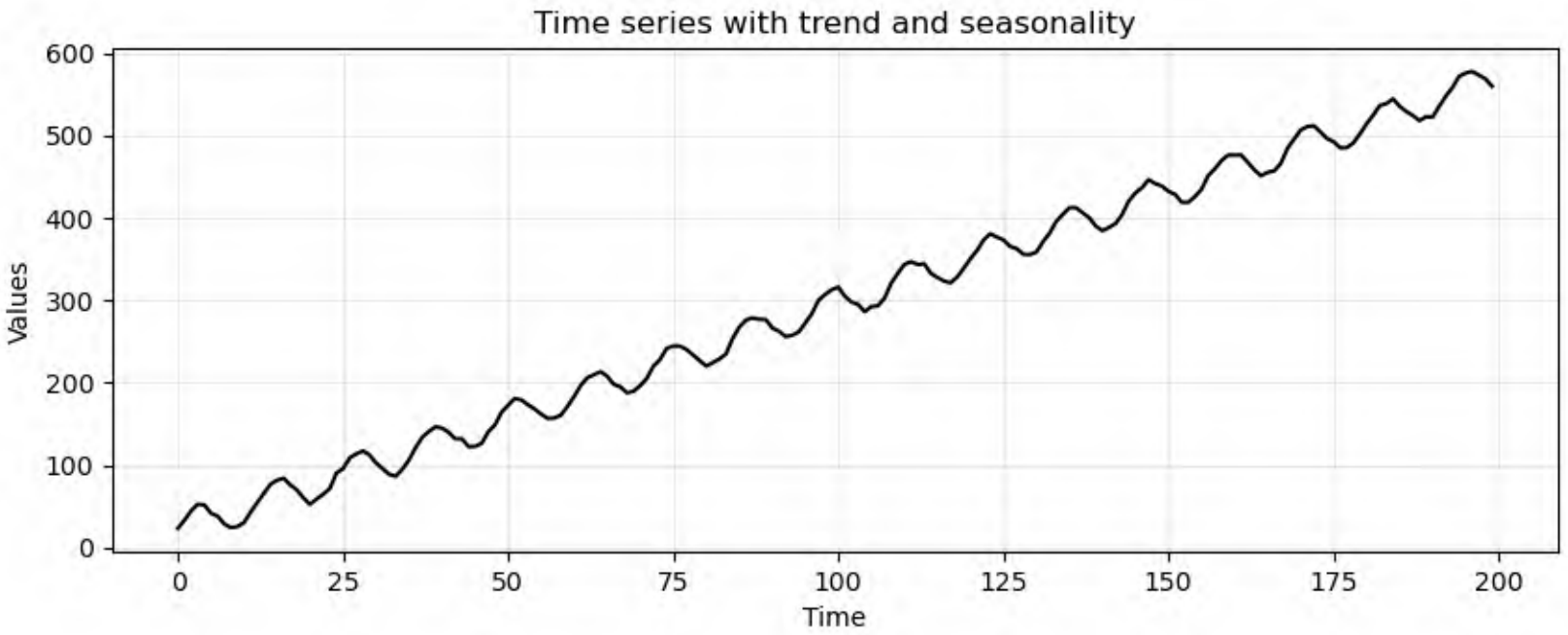
- Mean changes over time when there is a trend



- Variance changes over time



- Mean is constant for a full cycle (20 in the example)
- Any period not equal to a full cycle has a different mean
- Variance also depends on the time period that is measured
- Hence not stationary

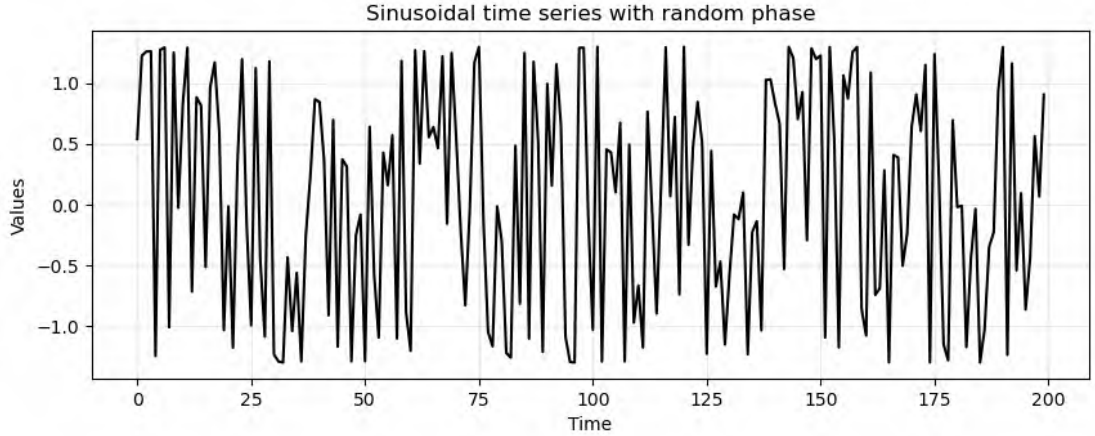


- Mean changes over time due to trend and there is a periodic component

- Sinusoidal signal

$$x_t = \mu + Rsin(\lambda t + \psi)$$

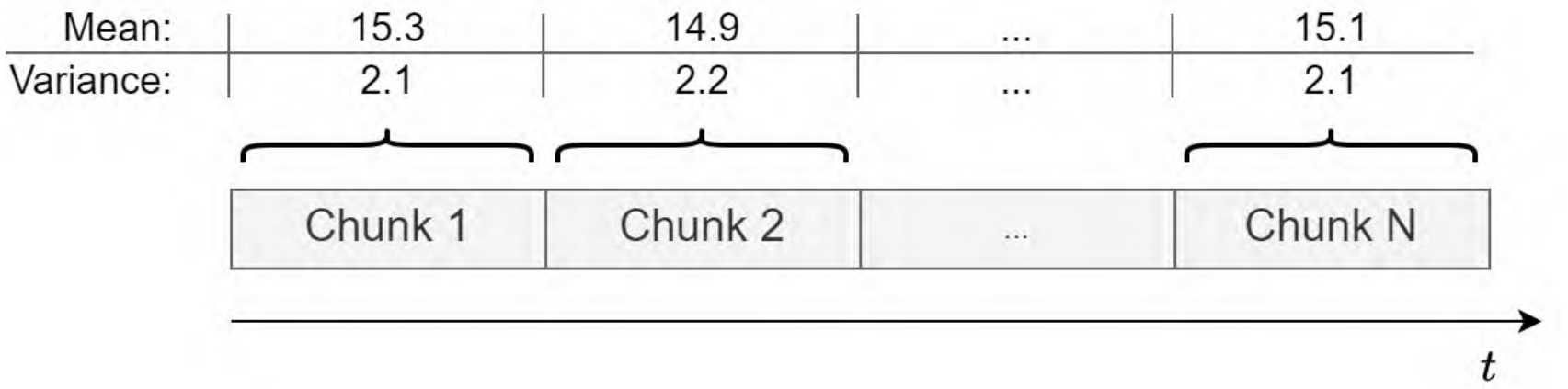
- μ is the mean, R is amplitude, λ frequency and ψ the phase
- ψ random variable distributed uniformly over $[-\pi, \pi]$



- $E[sin(\lambda t + \psi)] = 0$
- $Var(X) = E[X^2] - (E[X])^2 = Var\big(sin(\lambda t + \psi)\big) = 1/2$
- Autocovariance $Cov(x_t, x_{t+h}) = E[(sin(\lambda t + \psi)sin(\lambda(t + \tau) + \psi)] = \frac{R^2}{2}\cos(\lambda h)$, which depends only on h
- Hence weakly stationary

Πανεπιστήμιο Κύπρου

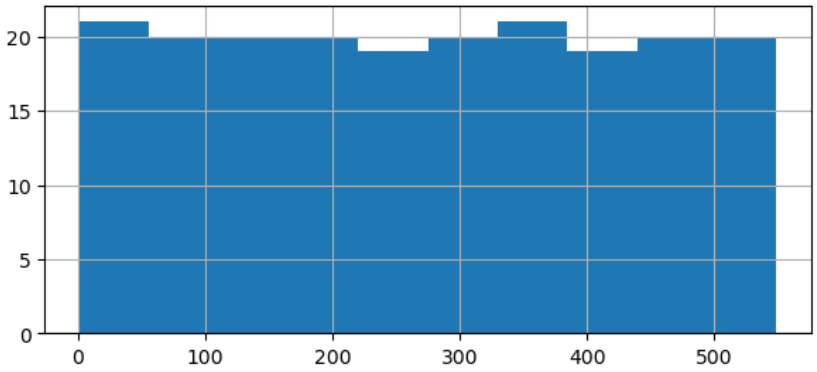
- Run-sequence plots
    - Allows visual inspection of the data
- Summary statistics
    - Cut series into separate chunks
    - Calculate statistics for each chunk and compare them
    - Large deviations might indicate non-stationarity
- Histogram plots
- Augmented Dickey-Fuller test
    - Statistical procedure to determine stationarity



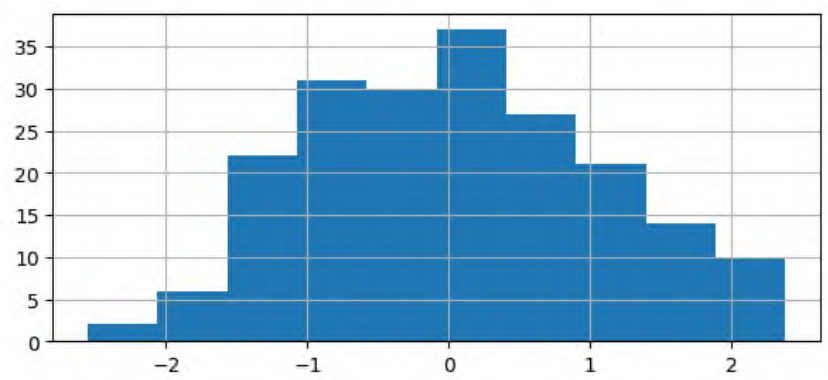

```
# split data into 10 chunks
chunks = np.split(trend, indices_or_sections=10)

# compare means and variances
print("{}\t | {}\t\t | {}".format("Chunk", "Mean", "Variance"))
print("-" * 35)
for i, chunk in enumerate(chunks, 1):
    print("{:2}\t | {:.5}\t | {:.5}".format(i, np.mean(chunk), np.var(chunk)))
```

- Statistical test can be used to check if the difference in means or the difference in variances is statistically significant

Πανεπιστήμιο Κύπρου



- histogram resembles a uniform distribution



- histogram resembles a normal distribution
- Suggest mean and variance are constant

- Statistical procedure to determine stationarity
- Consider
    1. Null hypothesis: H_0 the series is nonstationary
    2. Alternative hypothesis: H_A the series is stationary
- Set a significance level or threshold that determines whether you accept or reject the null
    - $\alpha = 0.05$ which yield a confidence of 95%
- ADF test might be inaccurate with small datasets or when heteroskedasticity is present
- Best to pair ADF with other techniques (summary statistics, etc)

```
# ADF test in Python
adf, pvalue, usedlag, nobs, critical_values, icbest = adfuller(stationary)
```

- Larger negative values provide indication of stationarity
- pvalue should be compared with the confidence level α
- Based on the comparison, we reject or fail to reject H_0

Transformation	Effect
Subtract trend	Constant mean
Apply log	Constant variance
Differencing	Remove autocorrelation
Seasonal differencing	Remove periodic component

PERIODIC SIGNALS

- Periodic signals, by their nature, have means and variances that repeat over the period of the cycle
- This implies that their statistical properties are functions of time within each period
    - i.e., the mean of a periodic signal over one cycle may be constant
- However, when considering any point in time relative to the cycle, the instantaneous mean of the signal can vary
    - Variance can also fluctuate within the cycle
- The ADF test specifically looks for a unit root
- A unit root indicates that shocks to the time series have a permanent effect, causing drifts in the level of the series
- A sinusoidal function, by contrast, is inherently mean-reverting within its cycles
    - After a peak a sinusoid reverts to its mean and any “shock” in terms of phase shift or amplitude change does not alter its oscillatory nature.
- Hence the ADF test’s conclusion of stationarity for a sinusoid does not imply that the sinusoid is stationary
- The test’s conclusion is about the absence of a unit root.
- This does not imply that the mean and variance are constant within the periodic fluctuations

```
# ADF test in Python
def adftest(series, plots=True):
    out = adfuller(series, autolag='AIC')
    print(f'ADF Statistic: {out[0]:.2f}')
    print(f'p-value: {out[1]:.3f}')
    print(f"Critical Values: {[f'{k}: {r:.2f}' for r,k in zip(out[4].values(),
out[4].keys())]}\n")

    if plots:
        # Compute rolling statistics
        rolmean = series.rolling(window=12).mean()
        rolstd = series.rolling(window=12).std()

        # Plot rolling statistics:
        plt.figure(figsize=(14, 4))
        plt.plot(series, color='tab:blue',label='Original')
        plt.plot(rolmean, color='tab:red', label='Rolling Mean')
        plt.plot(rolstd, color='black', label = 'Rolling Std')
        plt.legend(loc='best')
        plt.title('Rolling Mean and Standard Deviation')
        plt.grid();
```

SMOOTHING

- A time series consists of measurements characterized by temporal dependencies
- A time series can be decomposed into trend, seasonality, and residuals
- Many time series models require the data to be stationary in order to make forecasts
- Smoothing:
    - A data collection process is often affected by noise
    - If too strong, the noise can conceal useful patterns in the data
    - Smoothing can filter out noise
    - Also used to make forecasts by projecting the recovered patterns into the future

- Simple smoothing
    1. Simple average
    2. Moving average
    3. Weighted moving average

- Exponential smoothing
    1. Simple exponential smoothing
    2. Double exponential smoothing
    3. Triple exponential smoothing

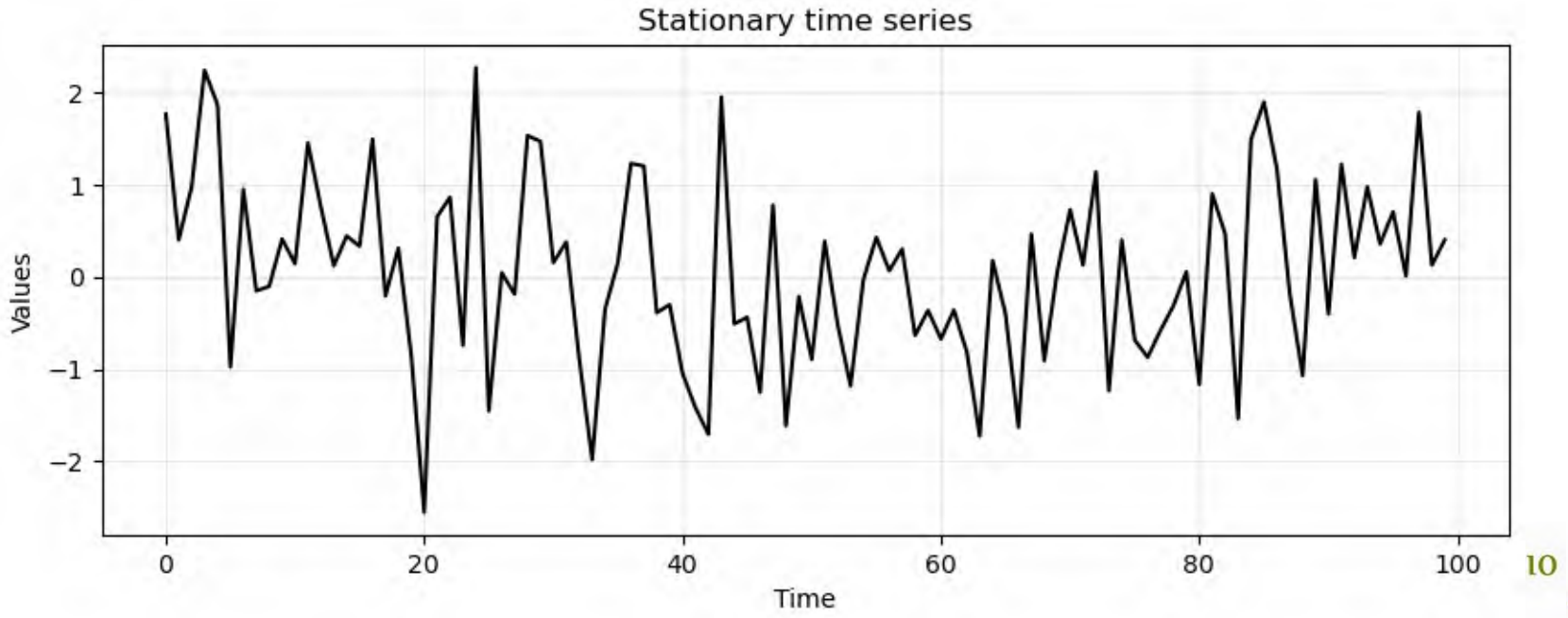
EXAMPLE TO BE USED

1. Generate stationary data

```
# Generate stationary data
time = np.arange(100)
stationary = np.random.normal(loc=0, scale=1.0, size=len(time))
```

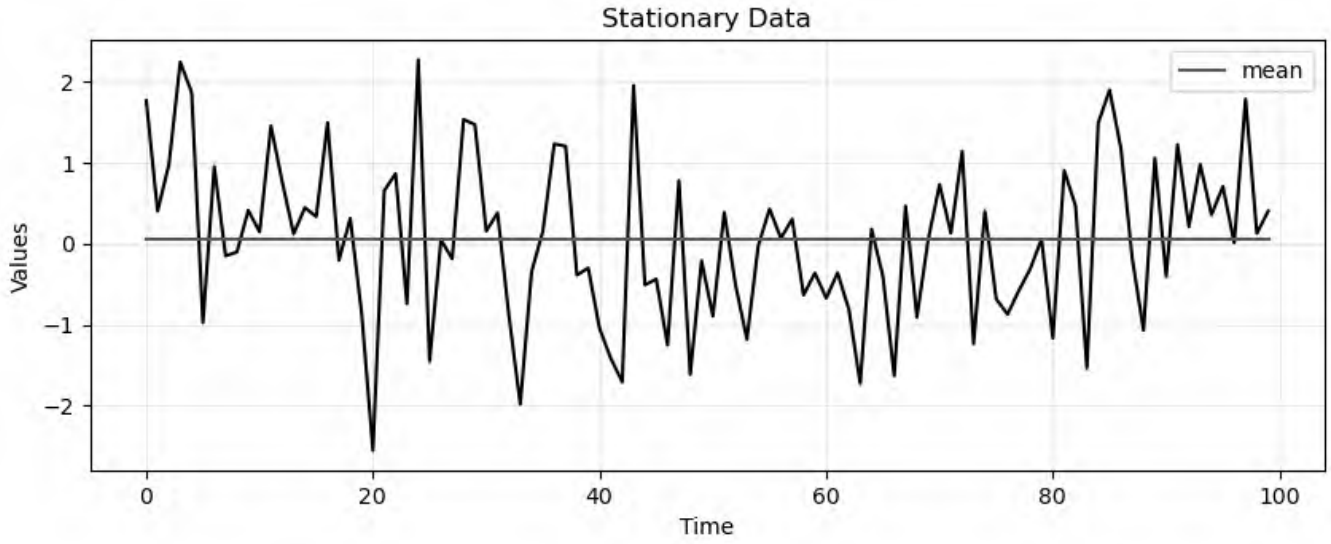
2. run-sequence plot used to visually inspect time series

```
# function to visualize data
run_sequence_plot(time, stationary, title="Stationary time series");
```



- Simple average is the most basic technique
- Mean value used to predict future values
  - Conservative way to represent time series

```
# find mean of series
stationary_time_series_avg = np.mean(stationary);

# create array composed of mean value and equal to length of time array
sts_avg = np.full(shape=len(time), fill_value=stationary_time_series_avg,
dtype='float')

# plot resulting figure
ax = run_sequence_plot(time, stationary, title="Stationary Data")
ax.plot(time, sts_avg, 'tab:red', label="mean")
plt.legend();
```



- Approximating with the mean seems reasonable
- Want to measure how far off our estimate is from reality
- One way is to calculate the Mean Squared Error (MSE)

$$MSE = \frac{1}{T}\sum_{t=1}^{T}(X(t) - \hat{X}(t))^2$$

- where $X(t)$ and $\hat{X}(t)$ are the true and estimated values at t
- Example:
  - Observed $X = [0, 1, 3, 2]$
  - Predicted by model $\hat{X}(t) = [1, 1, 2, 4]$

$$MSE = (0-1)^2 + (1-1)^2 + (3-2)^2 + (2-4)^2 = 6$$

- Smaller MSE better model!!

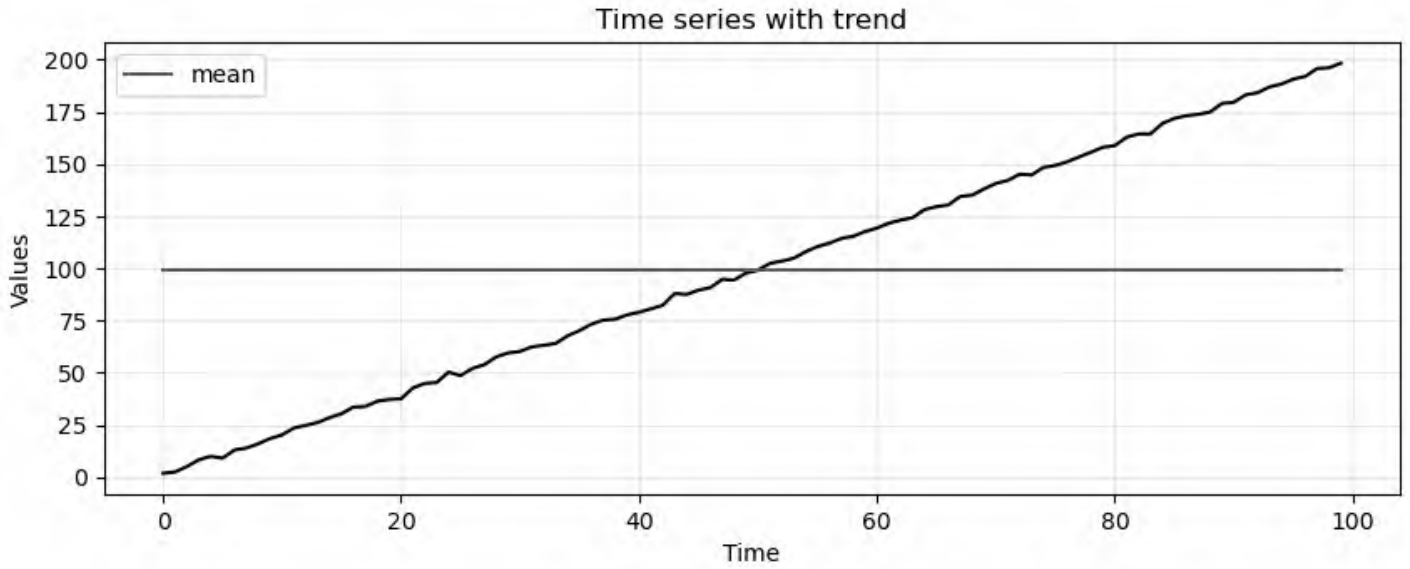
MSE FUNCTION CODE

```
def mse(observations, estimates):
    """
    INPUT:
        observations - numpy array of values indicating observed values
        estimates - numpy array of values indicating an estimate of values
    OUTPUT:
        Mean Square Error value
    """
    # check arg types
    assert type(observations) == type(np.array([])), "'observations' must be a
numpy array"
    assert type(estimates) == type(np.array([])), "'estimates' must be a numpy
array"
    # check length of arrays equal
    assert len(observations) == len(estimates), "Arrays must be of equal length"

    # calculations
    difference = observations - estimates
    sq_diff = difference ** 2
    mse = sum(sq_diff)

    return mse
```

```
# call mse function
res = mse(np.array([0, 1, 3, 2]), np.array([1, 1, 2, 4]))
print(res)
```

Πανεπιστήμιο Κύπρου

- Simple average does not work well when time series has a trend



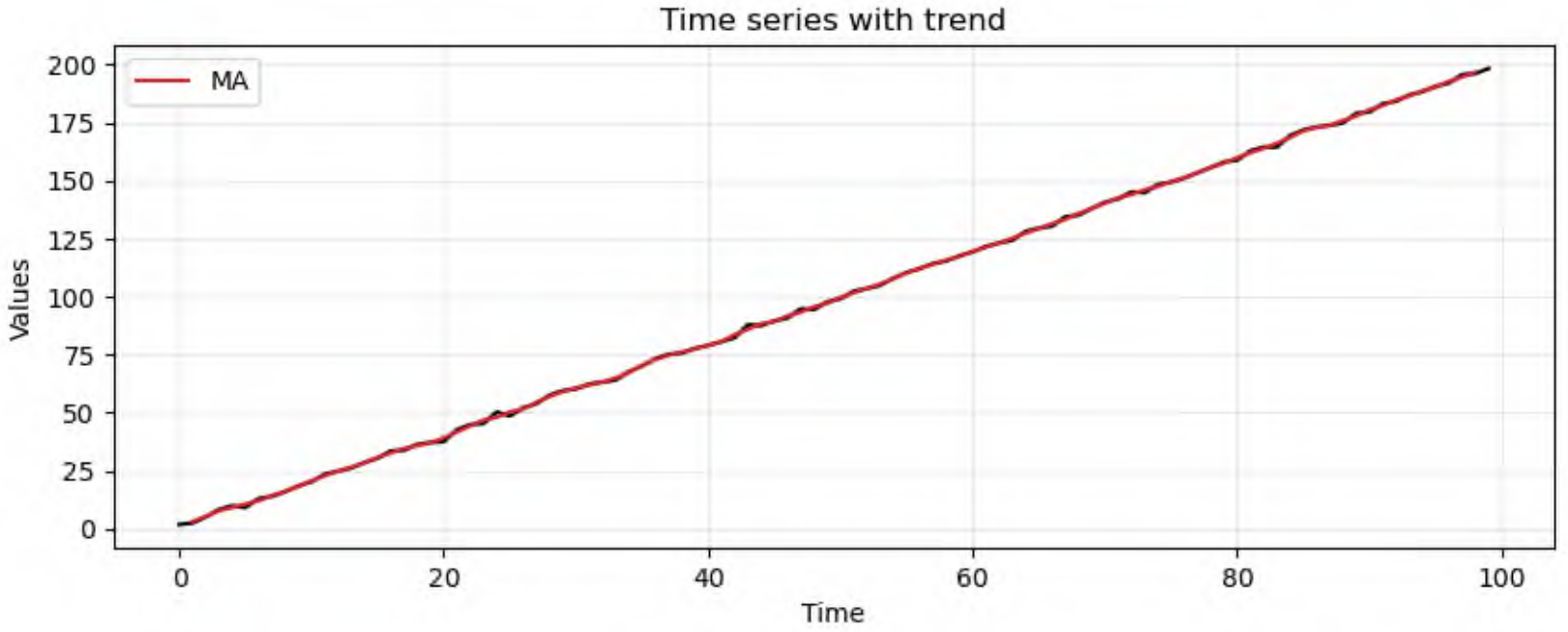
- Moving average has better sensitivity to local changes
  - Select a window size P
  - Compute average for the current window
  - Slide window by one and compute average again

- Example of MA


```
def moving_average(observations, window=3, forecast=False):
cumulative_sum = np.cumsum(observations, dtype=float)
cumulative_sum[window:] = cumulative_sum[window:] - cumulative_sum[:-window]
if forecast:
        return np.insert(cumulative_sum[window - 1:] / window, 0,
np.zeros(window-1))
else:
        return cumulative_sum[window - 1:] / window
```

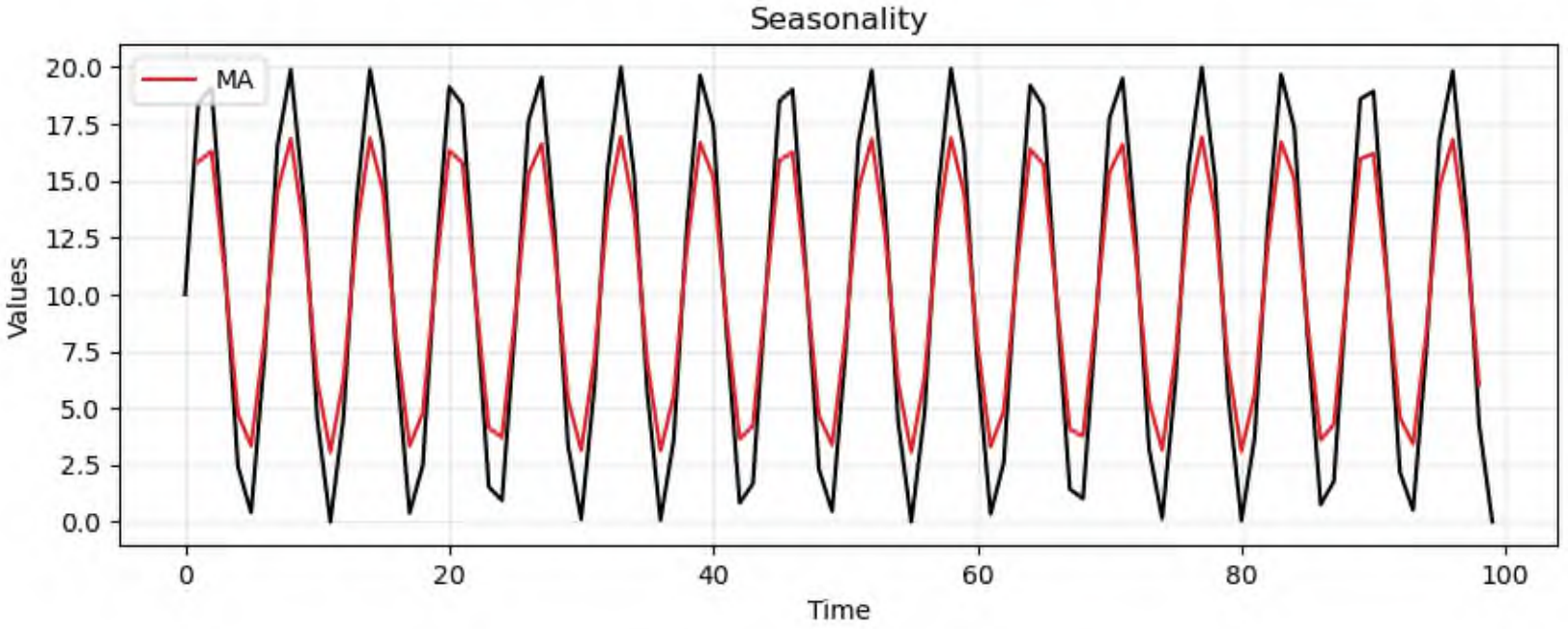
```
# create and test dataset
time = np.arange(100)
stationary = np.random.normal(loc=0, scale=1.0, size=len(time))
trend = (time * 2.0) + stationary
MA_trend = moving_average(trend, window=3)

print(f"MSE:\n--------\nsimple average: {mse(trend,
avg_trend):.2f}\nmoving_average: {mse(trend[2:], MA_trend):.2f}")
```

- Example of MA on time series with trend



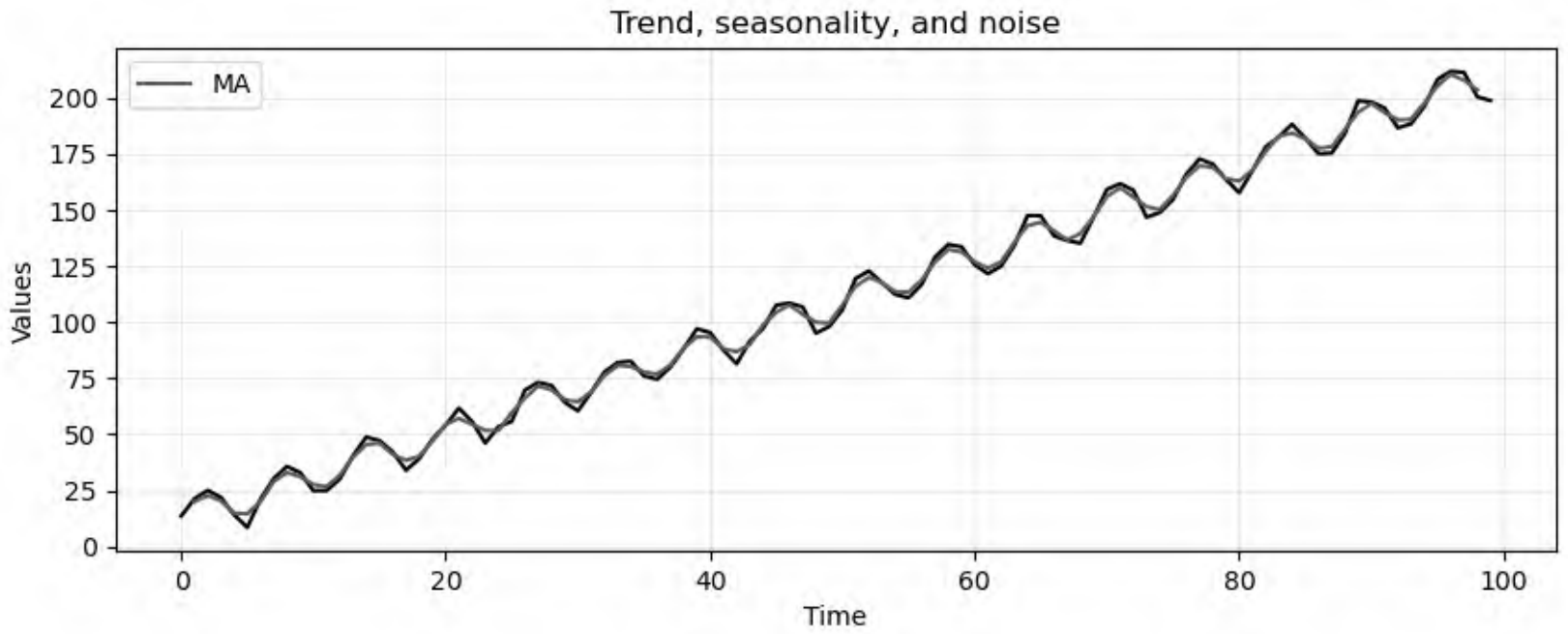
- Example of MA on time series with seasonality

```
# create and test dataset
seasonality = 10 + np.sin(time) * 10;
MA_seasonality = moving_average(seasonality, window=3);

run_sequence_plot(time, seasonality, title="Seasonality")
plt.plot(time[1:-1], MA_seasonality, 'tab:red', label="MA")
plt.legend(loc='upper left');
```



- Example of MA on time series with trend, seasonality, noise

```
# create and test dataset
trend_seasonality = trend + seasonality + stationary
MA_trend_seasonality = moving_average(trend_seasonality, window=3)

run_sequence_plot(time, trend_seasonality, title="Trend, seasonality, and noise")
plt.plot(time[1:-1], MA_trend_seasonality, 'tab:red', label="MA")
plt.legend(loc='upper left');
```



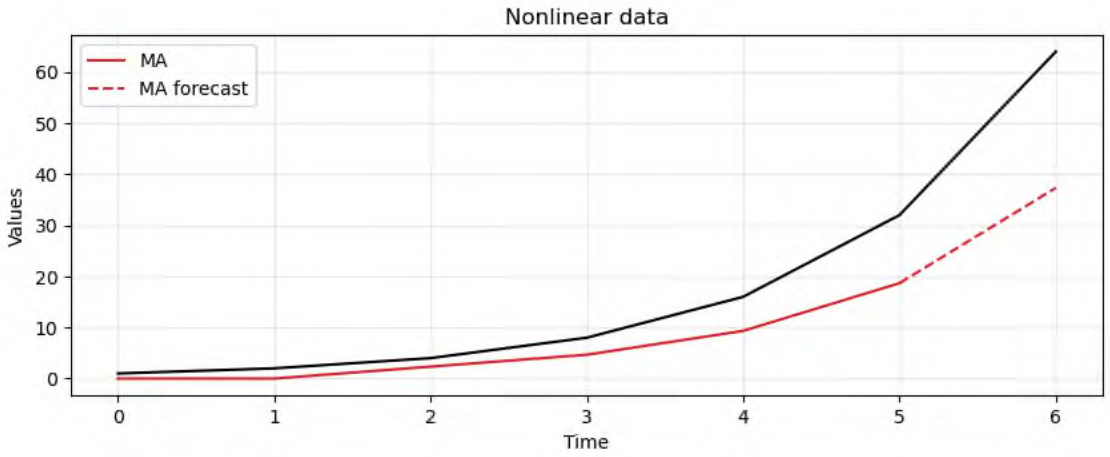
- MA has several limitations:
  - assigns equal importance to all values in the window, regardless of their chronological order
  - Hence it fails to capture trends that appear in the recent past
- MA requires the selection of an arbitrary window size
  - a small window may lead to noise
  - a too large window could oversmooth the data, missing important short-term fluctuations
- MA does not adjust for changes in trend or seasonality
  - Can lead to inaccurate predictions, especially when these components are nonlinear and time-dependent

- Instead of computing patterns within a series, the smoothing functions can be used to create forecasts
- Forecast for the next time step is computed as follows:

$$\hat{X}(t+1) = \frac{X(t) + X(t+1) + \cdots + X(t-P+1)}{P}$$

- Example

```
# create and test dataset
x = np.array([1, 2, 4, 8, 16, 32, 64])
ma_x = moving_average(x, window=3, forecast=True)

t = np.arange(len(x))
run_sequence_plot(t, x, title="Nonlinear data")
```



Πανεπιστήμιο Κύπρου

- In the previous example
  - MA can not keep up with the rate of change
  - Lagging by $\frac{(P+1)}{2}$
    - lag increases as you increase the window size
    - a window size that's too small however will chase noise in the data as opposed to extracting the pattern
    - hence tradeoff between **responsiveness** and **robustness** to noise
    - careful tuning required to determine which setup is best for a given dataset and problem at hand

- WMA weights recent observations more than older values
- By applying diminishing weights to past observations, we can control how much each value affects the forecasts
- Various methods to set the weights:
  - Could define them with the following system of equations:

$$\begin{cases} w_1 + w_2 + w_3 = 1 \\ w_2 = (w_1)^2 \\ w_3 = (w_1)^3 \end{cases}$$

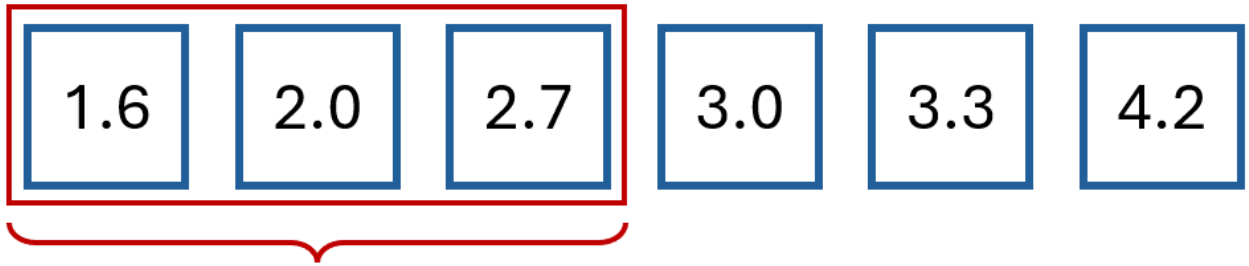
  - Resulting to

$$w_1 \approx 0.543 \; weight \; associated \; with \; t-1$$
$$w_2 \approx 0.294 \; weight \; associated \; with \; t-2$$
$$w_3 \approx 0.16 \; weight \; associated \; with \; t-3$$

$w_1 \approx 0.543$ *weight associated with* $t-1$

- $w_2 \approx 0.294$ *weight associated with* $t-2$

$w_3 \approx 0.16$ *weight associated with* $t-3$



$$w_3 \times 1.6 + w_2 \times 2.0 + w_1 \times 2.7 = 2.3$$

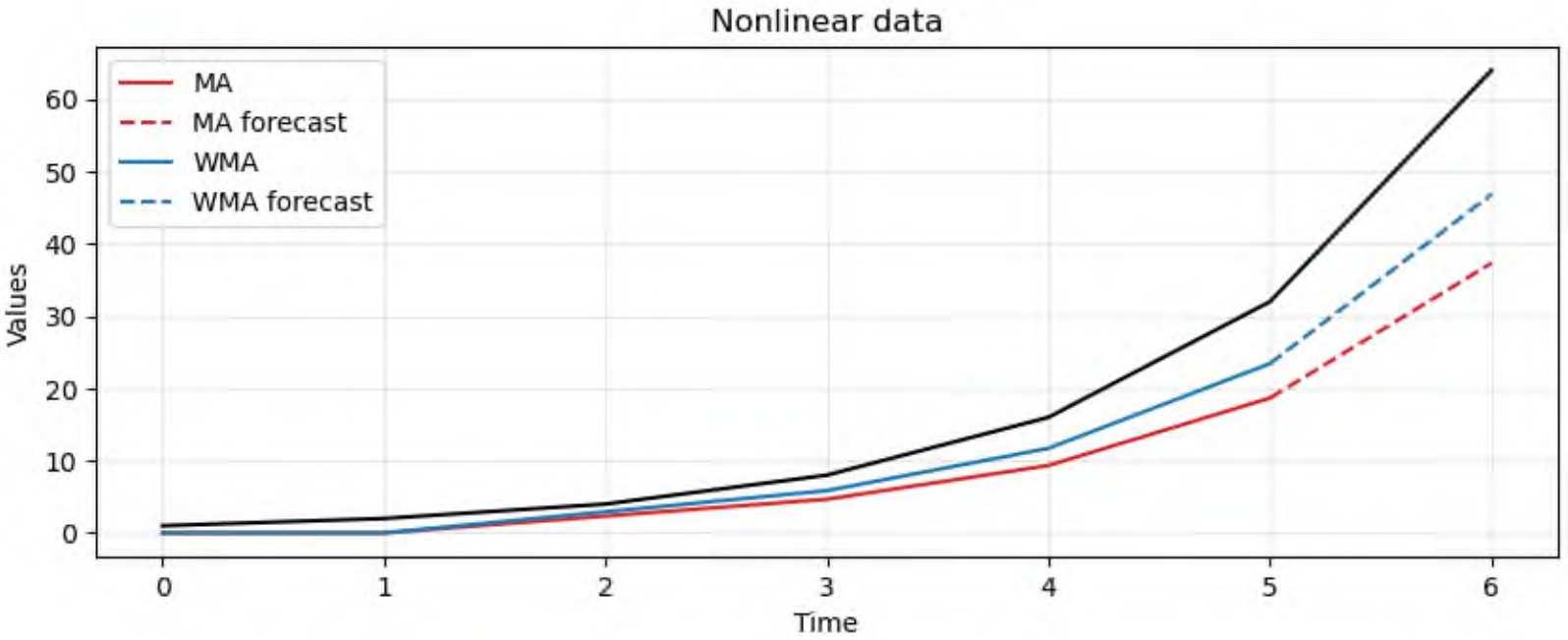
- Instead of computing patterns within a series, the smoothing functions can be used to create forecasts
- Forecast for the next time step is computed as follows:

$$\hat{X}(t+1) = \frac{w_1 X(t) + w_2 X(t+1) + \cdots + w_P X(t-P+1)}{P}$$

```
def weighted_moving_average(observations, weights, forecast=False):    """
 if len(weights) != len(observations[0:len(weights)]):
        raise ValueError("Length of weights must match the window size")

    # Normalize weights
    weights = np.array(weights) / np.sum(weights)
    # Initialize the result array
    result = np.empty(len(observations))
    # Calculate weighted moving average
    for i in range(len(weights)-1, len(observations)):
        window = observations[i-len(weights)+1:i+1]
        result[i] = np.dot(window, weights)
    # Handle forecast option
    if forecast:
        # Pad the beginning of the result array with zeros to maintain the
original length
        result[:len(weights)-1] = 0
    else:
        # Remove the start of the array that doesn't have a full window
        result = result[len(weights)-1:]

    return result
```

- Example

```
weights = np.array([0.160, 0.294, 0.543])
wma_x = weighted_moving_average(x, weights, forecast=True)

t = np.arange(len(x))
run_sequence_plot(t, x, title="Nonlinear data")
```



- WMA more sensitive to local changes
- More flexible to importance of different time steps
- Does not require a fixed window size

Πανεπιστήμιο Κύπρου

Type	Capture trend	Capture seasonality
Single Exponential Smoothing	✗	✗
Double Exponential Smoothing	✓	✗
Triple Exponential Smoothing	✓	✓

- Single Exponential Smoothing
  - Extract patterns from data without trend and seasonality
  - $S(t) = \alpha X(t) + (1 - a)S(t - 1)$ with α smoothing constant
  - At any time step τ:

$$S(t) = \sum_{t=0}^{\tau} \alpha(1 - a)^{\tau - t} X(t)$$

  - S(0) can be set to X(0) or mean of first few samples
  - α can also be computed with MSE

- Double Exponential Smoothing
  - Extract patterns from data AND trend
  - $S(t) = \alpha X(t) + (1 - a)(S(t-1) + b(t-1))$ Smoothed values
  - $b(t) = \beta\big(S(t) - S(t-1)\big) + (1-\beta)b(t-1)$ Estimated trend
  - $\hat{X}(t+1) = S(t) + b(t)$ Forecasts
  - Parameter $\beta \in [0, 1]$ controls the decay of the change
- Initialization
  - S(0) can be set to X(0) or mean of first few samples
  - $b(0) = X(1) - X(0)$
- Forecasts beyond 1-step ahead
  - $\hat{X}(t+\tau) = S(t) + \tau b(t)$

- TRIPLE Exponential Smoothing
  - Extract patterns from data AND trend AND seasonality
  - Third component added for seasonality of length L
    - Additive seasonality variant
    - Multiplicative seasonality variant
- Additive seasonality for variations that are roughly constant
  - $S(t) = \alpha(X(t) + c(t-L)) + (1-a)(S(t-1) + b(t-1))$
  - $b(t) = \beta\big(S(t) - S(t-1)\big) + (1-\beta)b(t-1)$
  - $c(t) = \gamma(X(t) - S(t-1) - b(t-1)) + (1-\gamma)c(t-L)$
  - $\hat{X}(t+\tau) = S(t) + \tau b(t) + c(t-L+1+(\tau-1)mod(L))$
- Multiplicative for changing seasonal variations
  - $S(t) = \alpha\frac{X(t)}{c(t-L)} + (1-a)\big(S(t-1) + b(t-1)\big)$
  - $b(t) = \beta\big(S(t) - S(t-1)\big) + (1-\beta)b(t-1)$
  - $c(t) = \gamma\frac{X(t)}{S(t)} + (1-\gamma)c(t-L)$
  - $\hat{X}(t+\tau) = (S(t) + \tau b(t))c(t-L+1+(\tau-1)mod(L))$

- Initialization for double / triple Exponential Smoothing
  - $b(0) = \frac{1}{L}\left(\frac{X(L+1)-X(1)}{L} + \frac{X(L+2)-X(2)}{L} + \cdots + \frac{X(L+L)-X(L)}{L}\right)$
  - $c(t) = \frac{1}{N}\sum_{J=1}^{N} \frac{X(L(j-1)+t)}{A_j}$ $\forall t$, N seasonal cycles in time series
  - $A_j = \frac{\sum_{i=1}^{L} X(2(j-1)+t)}{L}$
- All of these implemented in python using *statsmodels*

```
# Train/test split
train = trend_seasonality[:-5]
test = trend_seasonality[-5:]

# find mean of series
trend_seasonal_avg = np.mean(trend_seasonality)

# create array of mean value equal to length of time array
simple_avg_preds = np.full(shape=len(test), fill_value=trend_seasonal_avg,
dtype='float')

# mse
simple_mse = mse(test, simple_avg_preds)

# results
print("Predictions: ", simple_avg_preds)
print("MSE: ", simple_mse)
```

- Single Exponential Smoothing

```
from statsmodels.tsa.api import SimpleExpSmoothing

single = SimpleExpSmoothing(train).fit(optimized=True)
single_preds = single.forecast(len(test))
single_mse = mse(test, single_preds)
print("Predictions: ", single_preds)
print("MSE: ", single_mse)
```

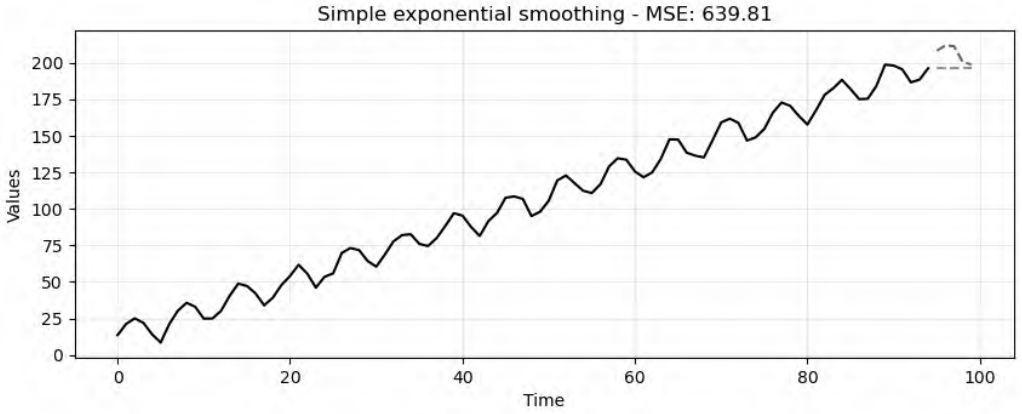
- Double Exponential Smoothing

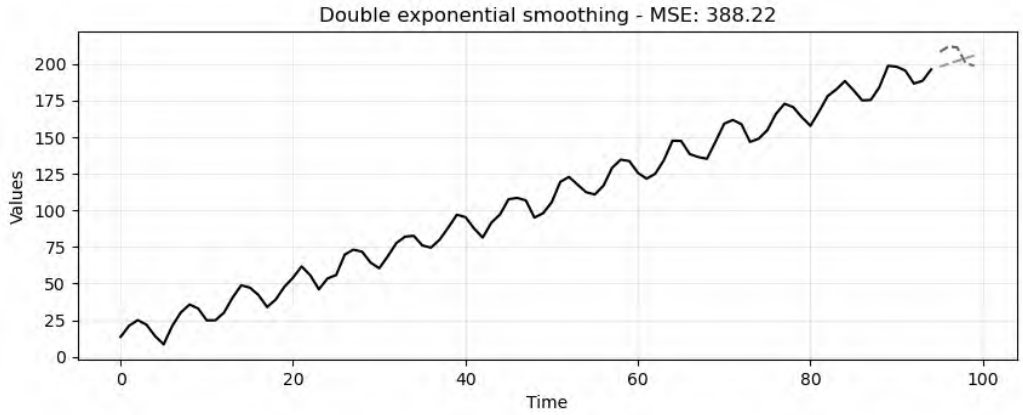
```
from statsmodels.tsa.api import Holt

double = Holt(train).fit(optimized=True)
double_preds = double.forecast(len(test))
double_mse = mse(test, double_preds)
print("Predictions: ", double_preds)
print("MSE: ", double_mse)
```

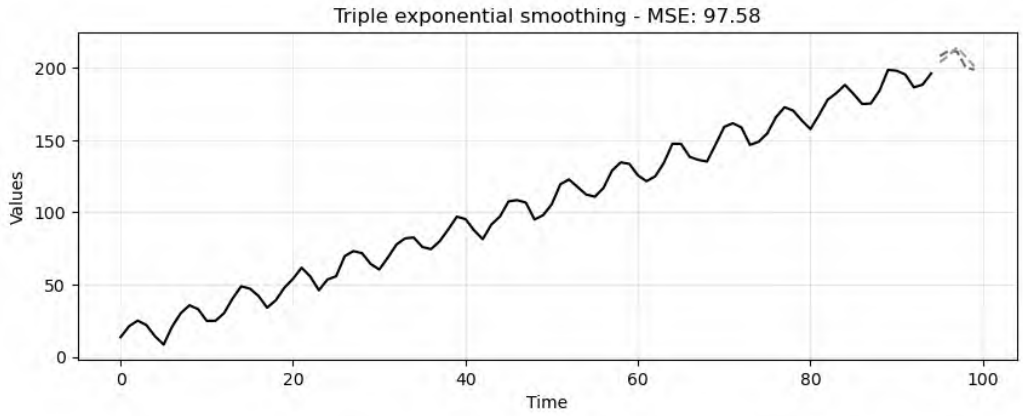
- Triple Exponential Smoothing

```
from statsmodels.tsa.api import ExponentialSmoothing

triple = ExponentialSmoothing(train, trend="additive", seasonal="additive",
                              seasonal_periods=13).fit(optimized=True)
triple_preds = triple.forecast(len(test))
triple_mse = mse(test, triple_preds)
print("Predictions: ", triple_preds)
print("MSE: ", triple_mse)
```

Exponential Smoothing Example







	MSE
simple	47197.581558
single	639.806113
double	388.215595
triple	97.583706

Πανεπιστήμιο Κύπρου