Task3:-

🧠 Objective:-

●Analyze and transform large-scale multivariate time series data efficiently by:

●Implementing vectorized rolling window statistics and EWMA.

●Performing spectral analysis via FFT and filtering.

●Comparing implementations: pure NumPy, pandas built-ins, and Numba-optimized or stride-trick-accelerated variants.

●Selecting the fastest method dynamically based on input size.

📁 Project Structure:-

```
Time_series_project/
│
├── timeseries_utils.py      # Core transformation functions (NumPy, pandas, Numba)
├── benchmark.py             # Benchmarks all implementations on synthetic datasets
├── benchmark_results.csv    # Benchmark timing and memory usage results
├── report.md / report.pdf   # Performance comparison and summary
├── plots/                   # Generated plots (speed, memory use, FFT visuals)
└── data/                    # Optionally, store large synthetic CSV datasets
```

🧩 Component 1: Rolling Window Statistics

Target Functions:-

●Rolling_mean(x, window)

●Rolling_var(x, window)

📌 Implementation Variants:-

Variant	Notes
`numpy_naive`	Simple loop/vector approach
`stride_trick`	C-contiguous memory view with `as_strided`
`numba_accel`	Just-In-Time (JIT) accelerated with `@njit`
`pandas_rolling`	`df.rolling(window).mean()` — simple but slower at scale

🧩 **Component 2: Exponentially Weighted Moving Average (EWMA)**

●Recursive definition:
$$ y_t = \alpha x_t + (1 - \alpha)y_{t-1} $$

📌**Implementation Options:-**

Variant	**Notes**
Numpy_loop.	Pure NumPy implementation (explicit recursion)

Numba_accel.	JIT-accelerated for performance
Pandas_ewm	df.ewm(alpha=α).mean() — clean but heavier on memory

Also include:-

•Ewma_cov(x, y, alpha): exponentially weighted covariance.

🧩 Component 3: FFT Spectral Analysis & Filtering

- Use np.fft.rfft or scipy.signal for:

●Power Spectral Density (PSD): visualize energy in frequency bands.

●Band-pass filtering: retain frequencies in a defined range.

Functions:-

```
Def compute_psd(signal: np.ndarray, fs: float) -> Tuple[np.ndarray, np.ndarray]:
    ...

Def bandpass_filter(signal: np.ndarray, fs: float, low: float, high: float) -> np.ndarray:
    ...
```

🧪 Component 4: Benchmarking & Auto-Selection

- Create benchmark.py to:

●Generate synthetic time series data (1M+ rows)

●Time each implementation (e.g., time.perf_counter)

●Record memory usage (e.g., tracemalloc, psutil)

●Save results to benchmark_results.csv

Also include:-

```
Def auto_select_method(data: np.ndarray, task: str) -> str:
```

"""
Returns 'numba', 'numpy', or 'pandas' depending on performance heuristics.
"""

📊 **Visualizations (in benchmark.py)**

●Runtime vs. Data Size (line plot)

●Memory usage comparison (bar plot)

●FFT spectrum plots (log scale)

•Use matplotlib or plotly.

📘 **Report Contents (report.md or .pdf)**

Sections :-

- *Introduction*

▪Task overview and motivation

- *Methodology*

▪Describe algorithms for each transformation

▪Explanation of acceleration techniques (Numba, stride tricks)

- *Performance Analysis*

▪Runtime plots

▪Memory consumption tables

▪Auto-selection strategy

- *Conclusion & Recommendations*

▪Best approach per task/data size

▪When pandas is better vs. Custom

🛠 **Dependencies:-**

Bash-

Pip install numpy pandas numba matplotlib psutil

•*Optional-*

Bash-

Pip install plotly seaborn scipy memory_profiler

✅ **1. Starter timeseries_utils.py-**

```
# timeseries_utils.py

Import numpy as np
Import pandas as pd
From numpy.lib.stride_tricks import as_strided
From numba import njit
From typing import Tuple

# ---------------------------
# Rolling Mean – NumPy naive
# ---------------------------
Def rolling_mean_numpy(x: np.ndarray, window: int) -> np.ndarray:
    Return np.convolve(x, np.ones(window)/window, mode='valid')

# ---------------------------
# Rolling Mean – Stride Trick
# ---------------------------
Def rolling_mean_stride(x: np.ndarray, window: int) -> np.ndarray:
    Shape = (x.size – window + 1, window)
```

```
    Strides = (x.strides[0], x.strides[0])
    Rolled = as_strided(x, shape=shape, strides=strides)
    Return rolled.mean(axis=1)

# ---------------------------
# Rolling Mean – Numba JIT
# ---------------------------
@njit
Def rolling_mean_numpy(x: np.ndarray, window: int) -> np.ndarray:
    Result = np.empty(x.size – window + 1)
    For i in range(result.size):
        Result[i] = np.mean(x[i:i+window])
    Return result

# ---------------------------
# EWMA – NumPy
# ---------------------------
Def ewma_numpy(x: np.ndarray, alpha: float) -> np.ndarray:
    Result = np.zeros_like(x)
    Result[0] = x[0]
    For t in range(1, len(x)):
        Result[t] = alpha * x[t] + (1 – alpha) * result[t – 1]
    Return result

# ---------------------------
# EWMA – Numba
# ---------------------------
@njit
Def ewma_numba(x: np.ndarray, alpha: float) -> np.ndarray:
    Result = np.empty_like(x)
    Result[0] = x[0]
    For t in range(1, len(x)):
        Result[t] = alpha * x[t] + (1 – alpha) * result[t – 1]
    Return result
```

✅ 2. Ready-made benchmark.py Script-

```
# benchmark.py

Import numpy as np
Import time
Import pandas as pd
```

```
Import matplotlib.pyplot as plt
from timeseries_utils.py import (
   Rolling_mean_numpy,
   Rolling_mean_stride,
   Rolling_mean_numpy,
   Ewma_numpy,
   Ewma_numba
)

Window = 50
Alpha = 0.1
Sizes = [10_000, 100_000, 1_000_000]
Methods = {
   "rolling_numpy": rolling_mean_numpy,
   "rolling_stride": rolling_mean_stride,
   "rolling_numba": rolling_mean_numba,
   "ewma_numpy": ewma_numpy,
   "ewma_numba": ewma_numba
}

Results = []

For size in sizes:
   X = np.random.rand(size)
   For name, func in methods.items():
      Try:
         T0 = time.perf_counter()
         If "rolling" in name:
            Func(x, window)
         Else:
            Func(x, alpha)
         Elapsed = time.perf_counter() – t0
         Results.append((name, size, elapsed))
      Except Exception as e:
         Results.append((name, size, np.nan))

Df = pd.DataFrame(results, columns=["Method", "DataSize", "TimeSeconds"])
Df.to_csv("benchmark_results.csv", index=False)

# Plot
Plt.figure(figsize=(10, 6))
For method in df["Method"].unique():
   Subset = df[df["Method"] == method]
   Plt.plot(subset["DataSize"], subset["TimeSeconds"], label=method)
Plt.xlabel("Data Size")
```

```
Plt.ylabel("Time (s)")
Plt.title("Benchmark: Rolling Mean & EWMA")
Plt.legend()
Plt.grid(True)
Plt.tight_layout()
Plt.savefig("plots/benchmark_plot.png")
Plt.show()
```

✅ **3. Template: report.md-**

High-Performance Time Series Transformations

📘 Overview
This report evaluates several approaches to compute rolling means and exponentially weighted moving averages (EWMA) on large time series datasets using NumPy, pandas, stride tricks, and Numba.

📊 Methods Compared

Method	Backend	Optimized For
rolling_numpy	NumPy	Simplicity
rolling_stride	NumPy	Memory efficiency
rolling_numba	Numba	Speed (JIT compiled)
ewma_numpy	NumPy	Recursive updates
ewma_numba	Numba	High performance

🧪 Benchmarking

Dataset
Synthetic univariate time series generated with:
- Sizes: 10K, 100K, 1M
- Distribution: Uniform [0, 1]

Results

`![Benchmark Plot](plots/benchmark_plot.png)`

⚖️ Observations

- **Stride tricks** outperform basic NumPy for rolling windows on medium sizes but scale poorly with massive data due to cache/memory pressure.
- **Numba** significantly speeds up recursive operations like EWMA.
- **NumPy** is sufficient for small datasets (<50K).
- **pandas** (not benchmarked yet here) is great for convenience but lags on performance.

🧠 Recommendations

- Use **Numba** for any recursive or cumulative statistics.
- Use **stride tricks** where vectorization is possible and memory alignment is good.
    - For streaming or real-time data, favor `ewma_numba()`.

📁 Files

- `timeseries_utils.py`: All methods
- `benchmark.py`: Test runner and visualizer
- `benchmark_results.csv`: Timings
- `plots/benchmark_plot.png`: Performance chart

✅ Next Steps

- Package all this into a ZIP file?
- Add FFT analysis and band-pass filtering as the next step?