Tianning Wang

Novel LLM Algorithms for Time Series Signal Processing

School of Technology and Innovation

Master's Programme in Computing Sciences

Sustainable and Autonomous Systems

Vaasa 2025

VAASAN YLIOPISTO
School of Technology and Innovation
Author: Tianning Wang
Thesis title: Novel LLM Algorithms for Time Series Signal Processing
Degree: Master of Science in Technology
Supervisor: Mohammed Elmusrati
Evaluator: Petri Välisuo
Year of graduation: 2025 **Number of pages:** 78

ABSTRACT:

Large language models (LLM) have recently demonstrated remarkable capabilities in sequence modeling tasks, particularly in human speech and text. This research investigates whether similar principles can be applied to the domain of time series signal processing, especially under nonlinear and non-Gaussian conditions. A novel signal estimation framework has been developed based on a Transformer architecture, in which multi-dimensional time series inputs are treated as structured sequential data, enabling the modeling of both temporal dependencies and inter-channel correlations through self-attention mechanisms.

Unlike traditional filtering approaches that rely on system equations or linear assumptions, the proposed model operates without explicit knowledge of the underlying dynamics. The estimation process is formulated as a denoising task, where the model learns to recover clean signals from noisy observations through a combination of attention-based encoding and confidence-weighted fusion. Each time step is predicted multiple times via a sliding window mechanism, and these overlapping predictions are aggregated using learned confidence scores, allowing the model to assign lower weight to unreliable outputs near signal boundaries or under high noise.

To evaluate the framework, experiments have been conducted using synthetically generated three-dimensional time series signals governed by nonlinear dynamics and subject to heavy-tailed, non-Gaussian noise distributions. The proposed method has been compared against a classical extended Kalman filter (EKF), which assumes local linearity and Gaussian noise. The LLM-based model is evaluated by multiple quantitative metrics which includes mean squared error (MSE), mean absolute error (MAE), signal-to-noise ratio (SNR), and frequency-domain error, and the results represent the novel model consistently outperforms the EKF by a wide margin. Visual analysis of residual distributions and reconstructed signals further confirms the model's ability to preserve signal structure while suppressing irregular noise components.

This study demonstrates that language-inspired modeling techniques can be effectively transferred to the domain of physical signal estimation. By treating time series as structured sequences and leveraging Transformer architectures, it becomes possible to handle highly nonlinear and noisy systems without relying on handcrafted models or tuning procedures. The framework introduced here lays the groundwork for future research in interpretable, modular, and multimodal time series modeling.

Keywords: Large language Model (LLM), Time Series Signals, Kalman Filter, Confidence-Weighted aggregation, Transformer, Global Navigation Satellite Systems (GNSS).

Contents

Figures

Tables

Abbreviations

GNSS Global Navigation Satellite Systems

LLM Large Language Model

EKF Extended Kalman Filter

UKF Unscented Kalman Filter

NLP Natural Language Processing

FFN Feed-forward Neural Network

MSE Mean Squared Error

MAE Mean Absolute Error

SNR Signal-to-Noise Ratio

DFT Discrete Fourier Transform

1 Introduction

In recent years, with the growing trend of automation and digitalization, time series signal processing has played an increasingly vital role in various practical industrial applications, such as Global Navigation Satellite Systems (GNSS), financial stock markets, industrial control systems, and environmental monitoring (Proakis & Manolakis, 2013). Clearly, accurate signal estimation and filtering are crucial for ensuring correct system operation, supporting effective decision-making, and guaranteeing the accuracy of modeling and prediction. However, real-world time series signals are often affected by dynamically disturbances from nonlinear systems and are inherently mixed with various types of noise, particularly non-Gaussian noise. This makes time series signal processing a highly challenging task. Generally, linear filters, especially the Kalman filter, have been widely applied for estimating the true state of time series signals, because it can achieve the minimum variance unbiased estimator under the linear-Gaussian assumption and its recursive algorithm characteristic, which enables handling dynamic changes. Unfortunately, nonlinear time series signals with non-Gaussian noise are prevalent in real-world engineering scenarios, and in some cases, different time series signals exhibit interdependencies or correlations across multiple sources, sensors, or modalities. But the Kalman filter and its standard extensions primarily operate under the assumption of independent observation streams and cannot directly leverage cross-series relationships for state estimation. As a result, ignoring such latent dependencies may lead to suboptimal filtering performance or even estimation bias and distortion in complex, interconnected systems.

Meanwhile, the rapid advancements in artificial intelligence (AI) and deep learning have been revolutionizing various fields, including computer vision, natural language processing, healthcare, and autonomous systems. Among these developments, large language models (LLMs) based on the transformer architecture (Vaswani et al., 2017) have demonstrated remarkable capabilities in modeling sequential data by capturing long-range dependencies, contextual patterns, and latent structures. Although originally designed for human language, the underlying mechanisms of LLMs suggest a broader applicability to

other types of sequential data, offering a novel perspective for time series signal processing.

1.1 Research Problem

Although the Kalman filter and its nonlinear extensions, such as the Extended Kalman Filter (EKF) and Unscented Kalman Filter (UKF), have been widely used for time series signal estimation, these methods inherently rely on explicit system models and Gaussian noise assumptions (Wan & Van Der Merwe, 2000). In practice, system dynamical changes are often partially unknown or difficult to model accurately, and noise characteristics frequently deviate from the ideal Gaussian distribution. These discrepancies can result in degraded filtering performance or biased state estimation. Besides, multiple time series signals describing the same event over the same time period are often interrelated. Traditional Kalman filtering methods are unable to leverage these unknown latent dependencies across correlated sequences, leading to potential information loss when processing multi-sequence signals.

Therefore, there is an urgent need for an alternative approach that can learn hidden dependencies directly from data without requiring explicit system equations. Inspired by recent advances in natural language processing, this study proposes to utilize LLM as a data-driven method for complex time series signal processing. By treating multiple time series as sequential data with inherent logical patterns, analogous to natural language sequences, the proposed approach aims to leverage the transformer architecture to learn latent interdependencies and improve state estimation. The estimated signals will then be compared with the ground truth to evaluate whether the LLM-based method achieves better performance than the Kalman filter in processing multiple nonlinear, non-Gaussian time series signals simultaneously.

1.2 Research Objectives

The primary objective of this research is to explore the feasibility and effectiveness of using LLM for time series signal processing, particularly under conditions involving nonlinear system dynamics and non-Gaussian noise. To achieve this objective, the study will first construct a simulated dataset including multiple time series signal data which incorporates nonlinear dynamics and non-Gaussian noise, based on realistic scenarios from GNSS and financial markets, serving as a testbed for controlled performance evaluation. Then, EKF and a transformer-based LLM will be developed to learn from multiple time series signals within the dataset and generate estimated signals. Finally, the study will analyze the advantages and limitations of the LLM-based time series signal processing method compared to the EKF, focusing on aspects such as estimation accuracy, capability to capture long-term trends and robustness to the outliers especially non-Gaussian noise.

1.3 Research Significance

This research introduces a novel perspective by treating time series signals as sequences with inherent logical structures, analogous to sentences in human language, thereby contributing to the convergence of natural language processing and time series signal processing. By applying LLM to time series data, the study explores an alternative data-driven approach for signal estimation that does not require explicit system modeling. The findings are expected to provide new solutions for handling nonlinear and non-Gaussian scenarios, potentially improving performance in applications such as GNSS positioning, financial forecasting, and multiple sensor data analysis. Furthermore, this research opens new interdisciplinary directions for the application of LLMs in the analysis of sequential data types.

1.4 Thesis Structure

This thesis is organized into five chapters. The first chapter introduces the research background, motivation, objectives, and significance. The second chapter reviews the theoretical background and related work, including the generation of time series signals, the principles of Kalman filters, basic principle of LLM and its recent developments in time series signal processing, and research gap. Then the third chapter presents the methodology, describing the simulated dataset generation, the details of the construction of the EKF and the transformer-based LLM, the evaluation benchmark, and related experimental setup. The fourth chapter reports and analyzes the experimental results, comparing the performance of the LLM and the EKF on processing the multiple time series signal from the simulated dataset. Finally, the last chapter discusses the implications of the findings in this experiment, summarizes the limitations of the proposed LLM-based approach, and suggests directions for future research.

2 Theoretical Background and Related Work

This chapter provides an in-depth review of the theoretical foundations underlying this research. Section 2.1 presents the characteristics and challenges of time series signals, with a focus on nonlinearities, non-Gaussian noise and latent interdependencies among multiple time series signals in GNSS and financial applications. Section 2.2 introduces the principles of Kalman filtering and its extensions, while Section 2.3 review recent development in LLM and related work on sequence data processing

2.1 Time Series Signals

Time series signals refer to sequences of data points indexed in temporal order, used to describe dynamic systems across various fields such as engineering, finance, and environmental monitoring. A key characteristic of time series signals is temporal dependency, meaning that the signal value at any given time is typically correlated with past values. In practice, time series data may exhibit complex patterns such as trends, seasonality, abrupt changes, and random fluctuations, which pose challenges for signal processing and prediction. In real-world applications, for instance in GNSS, time series signals represent measurements of position, velocity, or satellite observations over time. GNSS signals are affected by various sources of interference, including ionospheric and tropospheric delays, multipath effects, satellite clock errors, and receiver noise (Misra & Enge, 2006). These factors introduce both nonlinearities and non-Gaussian noise into the system. For example, the relationship between the measured pseudorange and the true distance involves nonlinear equations due to satellite geometry and signal propagation paths. Besides, multipath effects or anomalous measurements introduce errors that deviate from the ideal Gaussian distribution, leading to non-Gaussian characteristics. In financial markets, time series signals typically refer to asset prices, returns, or trading volumes over time. Financial time series are often characterized by volatility clustering, regime switching, heavy tails, and asymmetric distributions (Cont, 2001). Empirical studies have shown that return distributions in financial markets commonly exhibit fat tails and skewness,

violating Gaussian assumptions. Moreover, market shocks and sudden regime changes introduce nonlinear dependencies that are difficult for linear models to capture.

Another important characteristic of time series signals, especially in complex dynamic systems, is the presence of latent interdependencies among multiple sequences measured simultaneously. In many real-world applications, such as GNSS positioning and financial market analysis, it is necessary to observe multiple time series signals in parallel to fully capture the system's dynamics. These signals are often not independent at the same time step, instead, they exhibit latent correlations or interactions that reflect underlying physical, economic, or systemic relationships.

Overall, the presence of nonlinear system dynamics, non-Gaussian noise characteristics, and latent dependencies across multiple time series signals poses significant challenges for traditional Kalman Filters. Accurately estimating the true state of such systems requires approaches capable of handling deviations from linear-Gaussian assumptions, while also leveraging unknown latent dependencies to assist in state estimation, and maintaining robustness against noises and uncertainties.

2.2 Kalman Filters

The Kalman filter is a recursive linear estimator designed to estimate the state of a dynamic system from a series of noisy observations (Kalman, 1960). It operates under the assumptions of linear system dynamics and Gaussian noise, providing the optimal minimum variance estimate when these conditions hold. The observed system is typically modeled using a state-space representation,

$$x_k = F_{k-1}x_{k-1} + B_{k-1}u_{k-1} + w_{k-1}, \tag{1}$$

where x_k is the state vector at time k, representing the quantities to be estimated (e.g.,

position, velocity). The matrix F_{k-1} defines how the previous state x_{k-1} propagates to the current time step, incorporating the system's dynamic evolution. The matrix B_{k-1} models how the control input u_{k-1} affects the system; u_{k-1} represents any external input or intervention applied between time $k-1$ and k. The parameter w_{k-1} is the process noise, accounting for unmodeled dynamics or uncertainties, which is assumed to follow the Gaussian distribution.

In GNSS applications, F_{k-1} typically models the relationship between previous and current position and velocity under Newtonian motion equations, B_{k-1} can include effects from known acceleration inputs, and u_{k-1} corresponds to these known accelerations or corrections from inertial sensors. In financial markets, F_{k-1} captures autoregressive dependencies between past and current asset prices or returns, while $B_{k-1}u_{k-1}$ models external economic shocks or policy interventions impacting the state evolution.

The observation equation is written as,

$$z_k = H_k x_k + v_k, \tag{2}$$

where z_k is the observation at time step k, H_k is the observation matrix mapping the true state x_k into the measurement space, and v_k is the observation noise, which is also assumed to follow Gaussian distribution.

At each iteration, the Kalman filter performs two stages: prediction and update. In the prediction step, the prior state estimate $\hat{x}_{k|k-1}$ and its covariance $P_{k|k-1}$ are computed by propagating the previous estimate $\hat{x}_{k-1|k-1}$ through the state transition model F_{k-1} and incorporating the effect of control input u_{k-1} via B_{k-1}. In the update step, the predicted state is corrected by comparing the predicted measurement $H_k\hat{x}_{k|k-1}$ with the actual observation z_k.

This recursive formulation enables the Kalman filter to iteratively refine the state estimation by combining model-based predictions and noisy measurements, using only matrix operations at each step without retaining the entire measurement history. As a result, the algorithm achieves computational efficiency suitable for real-time applications (Grewal & Andrews, 2001).

Because of its efficiency and optimality under linear-Gaussian assumptions, the Kalman filter has been widely used in navigation, tracking, and signal estimation problems. However, real-world applications often exhibit nonlinear dynamics or non-Gaussian noise characteristics, limiting the filter's performance under such conditions.

To solve these problems, nonlinear extensions were developed. The first generally used version is the EKF which linearizes the nonlinear system by performing a first-order Taylor expansion around the current estimation, enabling the application of Kalman equations to the linearized model (Maybeck, 1994). This linearization is expressed as,

$$f(x) \approx f(\hat{x}) + F(\Delta x), \tag{3}$$

where $f(x)$ is the nonlinear function, $\hat{x}$ is the current estimate, F is the Jacobian matrix evaluated at $\hat{x}$, and Δx is the deviation from $\hat{x}$.

And the other version is the UKF which propagates a set of sigma points through the nonlinear function, achieving a more accurate approximation of the mean and covariance without requiring explicit linearization (Julier & Uhlmann, 1997).

Although the UKF generally offers better estimation accuracy for strongly nonlinear systems, it requires higher computational cost due to the generation and propagation of multiple sigma points. In GNSS applications, the nonlinearity level is moderate, and the EKF achieves satisfactory performance with simpler computation and well-established

implementation frameworks (Crassidis & Junkins, 2011). Therefore, the EKF remains the standard nonlinear extension in GNSS, balancing accuracy and computational efficiency. So based on these practical considerations and its prevalence in real-world engineering, the EKF is selected as the baseline method for comparison with the LLM approach in this research.

2.3 Large Language Models

LLMs refer to deep learning models trained on large-scale textual datasets with the objective of learning statistical patterns, syntactic structures, and semantic relationships in natural language (Bommasani et al., 2021). From the current academic perspective, LLMs are typically based on transformer architectures and contain billions of parameters, enabling them to generate human-like text and perform complex NLP tasks such as language understanding, translation, and summarization.

With the development of models such as GPT-3 (Brown et al., 2020) and PaLM (Chowdhery et al., 2022), research has shown that the generalization capabilities of LLMs significantly improve as model size and training data increase. This characteristic allows the models to exhibit creative abilities and handle multiple tasks without explicit task-specific supervision. This advancement has facilitated a transition from task-specific models to general-purpose language models and even toward integrated or fusion models. At the same time, there has been growing interest in open-source LLMs that aim to maintain high understanding capabilities while minimizing training and deployment costs, further fostering the development of an active LLM community like HuggingFace.

Although it is originally developed for NLP tasks, the core mechanism of LLMs, so called sequence modeling and contextual dependency learning, which also suggests potential applications in other types of sequential data domains, such as bioinformatics, code generation, and time series analysis (Ramesh et al., 2023). And this capability is fundamentally enabled by the transformer architecture and its attention mechanism, which will be

described in detail in the following subsections.

2.3.1 Transformer

The Transformer architecture was originally proposed by Vaswani (Vaswani et al., 2017) and has become the foundational structure of modern LLMs. Transformer adopt an encoder–decoder architecture. The encoder maps the input sequence into continuous latent representations, while the decoder generates the output sequence step by step based on these representations and the previously generated outputs. Both the encoder and the decoder are composed of multiple identical stacked layers. Each layer contains several key components, including multi-head self-attention, position-wise feed-forward neural networks, residual connections, and layer normalization.

Each encoder layer first passes the input through a multi-head self-attention sublayer, allowing the tokens in the sequence (i.e., the basic units of the input sequence, which may be words, subwords like -ly, or numerical values depending on the data type) to attend to other positions and aggregate contextual information. The detailed principles of the attention mechanism will be discussed in the next section.

Then, the representation of each token is processed through a position-wise FFN, which applies two linear transformations separated by a non-linear activation at each position independently. The FFN is formally defined as:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{4}$$

where x represents the input vector for a token at a given position, with dimensionality d_{model}. In this formulation, $W_1 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$ and $W_2 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$ are learnable weight matrices for the first and second linear transformations, respectively, while $b_1 \in \mathbb{R}^{d_{\text{ff}}}$ and $b_2 \in \mathbb{R}^{d_{\text{model}}}$ are the corresponding bias vectors. The function $\max(0, \cdot)$ denotes a ReLU

activation applied element-wise. In essence, the FFN first projects the representation of each token into a higher-dimensional latent space via the transformation $xW_1 + b_1$, enabling the model to capture richer intermediate features. The ReLU activation introduces non-linearity, allowing the network to learn complex mappings beyond linear operations. Subsequently, the activated output is projected back to the original model dimension through $W_2 + b_2$, ensuring that the output dimensionality remains consistent with the input, facilitating residual connections. This design makes the model have additional capacity to perform token-wise non-linear transformations after the attention mechanism, enhancing the expressiveness and modeling capability of each layer.

And each sublayer in the transformer is surrounded by a residual connection, a design borrowed from the residual network (ResNet) framework (He, Zhang, Ren, & Sun, 2016). Given an input x to a sublayer, the output of the residual connection is formulated as:

$$\text{Output} = x + \text{Sublayer}(x) \tag{5}$$

where Sublayer(x) represents the output of the corresponding sublayer operation (either the multi-head attention module or the feed-forward neural network), and the addition is performed element-wise. This mechanism allows the original input x to bypass the sublayer computation and be directly added to its output. The primary motivation for incorporating residual connections is to facilitate the flow of gradients during backpropagation, especially in deep networks. In traditional deep neural networks, as the number of layers increases, gradients can either diminish (vanishing gradient) or explode, making training unstable or ineffective. By adding a shortcut path for the gradient to directly propagate from later layers to earlier layers, residual connections mitigate these issues and enable the training of substantially deeper architectures. Moreover, the residual addition acts as a form of identity mapping, preserving the original input signal while allowing the sublayer to learn perturbations relative to the input. This design helps stabilize learning and accelerates convergence by preventing significant distortion of the input

representation at each layer. In the transformer, residual connections are applied around both the multi-head attention sublayer and the feed-forward sublayer to ensure consistent training dynamics across the entire encoder and decoder stacks.

After each residual connection, a layer normalization operation is applied to stabilize the output and improve training dynamics. Layer normalization (Ba, Kiros, & Hinton, 2016), normalizes the activations of a layer across its feature dimension, ensuring consistent scaling and shifting. The mathematical formulation is given by:

$$\text{LayerNorm}(x) = \gamma \cdot \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{6}$$

where x is the input vector, μ and σ^2 denote the mean and variance computed over the feature dimension of x, ϵ is a small constant added for numerical stability, and γ, β are learnable scaling and shifting parameters with the same dimensionality as x. Unlike batch normalization, which normalizes across a batch of samples, layer normalization performs normalization independently for each training example. This property is particularly advantageous for transformer models, as the variable-length nature of sequences and lack of recurrence make batch statistics less consistent across different inputs. So the primary purpose of layer normalization is to reduce internal covariate shift by maintaining a stable distribution of activations throughout training. This normalization helps accelerate convergence, prevent activation explosions, and promote more stable gradient updates across deep architectures. In the transformer, layer normalization is applied after each residual addition in both the encoder and decoder to ensure that every output of sublayer maintains a normalized scale before entering subsequent processing steps.

One inherent limitation of the transformer architecture is its lack of a built-in mechanism to capture the sequential order of tokens, because the self-attention mechanism is permutation-invariant and does not encode position information by default. To solve this vital problem, positional encoding (Vaswani et al., 2017) is introduced, which injects

position-dependent signals into the input embeddings in order to provide the model with information about the relative or absolute position of tokens in the sequence. In general, the positional encoding vector for each position pos and dimension i is defined as:

$$\mathsf{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\mathsf{model}}}}\right), \quad \mathsf{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\mathsf{model}}}}\right) \tag{7}$$

where pos represents the position index (starting from zero), i denotes the dimension index within the embedding vector, and d_{model} is the dimensionality of the input embeddings. The encoding alternates between sine and cosine functions at different frequencies across dimensions, creating a unique positional vector for each position. This sinusoidal design ensures that the positional encodings exhibit linear relationships between positions, which enables the model to learn relative positions by linear transformations. Additionally, these encodings are fixed and deterministic, allowing the model to extrapolate to longer sequences beyond those seen during training. The positional encoding vectors are added element-wise to the input embeddings before being fed into the encoder stack:

$$x_{\mathsf{input}} = x_{\mathsf{embedding}} + \mathsf{PE} \tag{8}$$

where $x_{\mathsf{embedding}}$ is the token embedding vector, and PE is the corresponding positional encoding vector. By incorporating positional encoding in this additive manner, the transformer maintains the same dimensionality of inputs while embedding position-dependent patterns into the representation space. In total, positional encoding provides an ingenious solution for transformers to model sequence order without relying on recurrence or convolution, preserving the ability to process inputs in parallel while enabling it to capture positional information through learned attention patterns.

2.3.2 Attention Mechanism

Attention mechanism plays the key role in transformer architecture, which enables the model to selectively focus on different parts of the input sequence when processing each token. The attention mechanism was first introduced in the context of neural machine translation (Bahdanau, Cho, & Bengio, 2014), and later generalized in the transformer model (Vaswani et al., 2017) as a scaled dot-product attention. The scaled dot-product attention is defined as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{9}$$

where Q, K, and V denote the query, key, and value matrices, respectively, each derived from the input sequence through learned linear projections. Specifically, $Q, K, V \in \mathbb{R}^{n \times d_k}$ if the input sequence has length n and each projection has dimensionality d_k. The term $QK^\top$ computes a similarity score between the query and all keys, resulting in an $n \times n$ matrix of attention logits. To prevent excessively large dot products, the logits are scaled by $1/\sqrt{d_k}$. Applying the softmax function across each row normalizes these scores into a probability distribution over the input tokens. Finally, the weighted sum of the value vectors, according to these probabilities, yields the attended output representation for each query. The intuition behind attention is that each token in the sequence dynamically attends to other tokens based on learned similarity, allowing the model to aggregate relevant contextual information. Unlike recurrent architectures, where dependencies propagate step by step, attention enables direct interactions between any two positions, regardless of their distance. To further enrich the representational capacity, the transformer employs multi-head attention, which runs multiple attention functions in parallel. Each attention head learns a separate projection of Q, K, and V, allowing the model to attend to different aspects of the input. The outputs of all heads are concatenated and projected back to the original dimension:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{10}$$

where each head is computed as:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{11}$$

with W_i^Q, W_i^K, W_i^V, and W^O being learned projection matrices. The multi-head design allows the transformer to capture diverse types of relationships across different subspaces of the representation. For example, one head may focus on syntactic dependencies while another attends to semantic similarity. In the transformer, self-attention is applied within the encoder, meaning $Q = K = V$, as all tokens attend to each other within the same sequence. In the decoder, masked self-attention prevents attending to future positions during training, ensuring autoregressive generation. Additionally, cross-attention in the decoder enables each target token to attend to all source tokens, incorporating encoder outputs into the generation process. By relying on attention mechanisms, the transformer eliminates recurrence and enables parallel computation across sequence positions, facilitating both efficient training and effective long-range dependency modeling.

2.3.3 Why Choosing LLM for Time Series Signal Processing

LLMs are capable of capturing complex dependencies in sequential data through their self-attention mechanism, which enables direct interactions between any positions in a sequence regardless of their distance. This property aligns well with the intrinsic characteristics of time series signals, where dependencies may span short-term patterns, seasonal fluctuations, or long-term trends. Moreover, the self-attention mechanism of the transformer inherently supports cross-sequence interactions. Specifically, it allows each

token—here representing a multi-dimensional observation at a given time step containing all sequences or channels (when different time series are treated as separate feature dimensions)—to attend to tokens from other time steps. Through its embedding representation, the model can thus simultaneously capture both temporal dependencies and latent correlations across sequences, a capability not directly available in traditional Kalman filters or their extensions. In addition, the parallel computation enabled by the transformer facilitates efficient training and inference on large datasets, overcoming the sequential bottleneck of recurrent neural networks. Combined with positional encoding and multi-head attention, LLMs provide both order-awareness and multi-perspective relational modeling, which are critical for accurately estimating hidden system states from noisy, nonlinear, and interrelated time series data. In summary, LLMs offer several distinctive advantages for time series signal processing: the ability to learn complex nonlinear relationships without explicit models, the capacity to capture long-range dependencies, the ability to model latent correlations across different sequences, and scalability through parallel computation. These characteristics motivate their exploration as a promising alternative to traditional filtering methods in complex multimodal, non-Gaussian, and nonlinear time series environments.

2.3.4 Related Work on LLMs in Time Series Signal Processing

In recent years, researchers have begun exploring the potential of LLMs for time series signal processing, leveraging their powerful sequence modeling and attention-based relational reasoning capabilities. This emerging research area has produced a range of approaches, demonstrating both opportunities and limitations of applying LLMs to numerical and temporal data. One prominent line of research focuses on transforming time series signals into tokenized textual representations, allowing pretrained LLMs to process them directly without architectural modifications. The SigLLM framework (News, 2024) exemplifies this approach, encoding numerical sequences as text prompts and using frozen LLMs for anomaly detection and forecasting tasks. This text-to-time-series mapping enables zero-shot or few-shot inference but raises questions about information

loss during discretization and encoding. Another research direction involves adapting transformer-based architectures to time series forecasting through hybrid models. For example, LeMoLE (Zhang et al., 2024) integrates LLM embeddings as meta-features into a mixture of linear experts, achieving improved prediction accuracy and computational efficiency. Similarly, the Timer model (Liu et al., 2024) treats large pretrained transformers as unified generative models for multiple time series tasks, including forecasting, imputation, and anomaly detection, showcasing strong downstream task performance. Recent evaluation studies have also questioned the necessity of pretrained LLM components for time series tasks. A critical analysis (Tan, Merrill, Gupta, Althoff, & Hartvigsen, 2024) shows that in several benchmark datasets, removing the LLM or replacing it with simpler attention layers did not degrade prediction accuracy and even improved efficiency. This suggests that the benefits of LLMs for time series forecasting may depend on the specific task formulation and data characteristics. Moreover, efforts have been made to benchmark the interpretability and feature understanding abilities of LLMs in time series contexts. A study (Fons et al., 2024) designed synthetic datasets with textual descriptions linked to temporal patterns, revealing that LLMs can partially infer certain statistical properties but struggle with more complex feature interpretations. Overall, these studies highlight the versatility of LLMs for time series applications while also pointing out current challenges, such as computational overhead, data representation alignment, and domain adaptation. This emerging body of work motivates further investigation into specialized architectures, data encoding strategies, and hybrid modeling frameworks to unlock the full potential of LLMs in time series signal processing.

2.4 Research Gap and Contribution

Despite growing interest in applying LLMs to time series data, current research still faces several notable limitations. First, many existing approaches focus primarily on single time series forecasting or univariate signal analysis, lacking the ability to jointly model multiple correlated sequences that arise in practical applications such as GNSS positioning or financial market analysis. Capturing latent interdependencies across multiple time series

remains an underexplored area. Second, while some studies leverage pretrained LLMs or transformer variants for time series tasks, they often either discretize continuous signals into tokens or require specialized architectures, raising concerns about information loss, additional complexity, and scalability. There is limited investigation on directly applying LLMs to raw multi-sequence time series signals in a numerically continuous domain without extensive preprocessing. Third, existing benchmarks and evaluation frameworks mostly target standard forecasting accuracy metrics, without systematically comparing LLM-based methods to classical filtering algorithms like the Kalman filter and its nonlinear extensions. Few works have directly evaluated whether LLMs can achieve superior signal estimation under nonlinear, non-Gaussian noise, and correlated multi-sequence conditions, which is a scenario frequently encountered in engineering and financial projects.

Given these gaps, this research aims to explore the feasibility of using LLMs for time series signal estimation in a controlled simulated environment characterized by nonlinear system dynamics, non-Gaussian noise, and latent correlations among multiple sequences. Specifically, I propose viewing multiple correlated time series signals as multi-dimensional sequential data, enabling an LLM to model both temporal and cross-sequence dependencies through self-attention mechanisms. I design and implement an LLM-based signal estimation framework that directly processes multi-sequence time series with nonlinear dynamics and non-Gaussian noise, without requiring explicit model equations or discretization into tokenized text. As part of the inference pipeline, a novel confidence-weighted aggregation mechanism is introduced to fuse overlapping window predictions based on model-learned uncertainty estimates, improving robustness and reducing edge artifacts. I further provide an empirical comparison between the proposed LLM method and the EKF, evaluating their relative accuracy and efficiency under simulated conditions to gain insights into the advantages and limitations of LLM-based time series signal processing. In addition, all experiments were conducted using a single consumer-level GPU (NVIDIA RTX 3060 with 8GB VRAM), demonstrating the feasibility of implementing the proposed LLM-based signal estimation framework on modest computational resources.

3 Methodology

This study aims to investigate whether LLM can provide effective signal estimation for multi-sequence time series affected by nonlinear dynamics, non-Gaussian noise, and latent inter-sequence correlations. To explore this, an experimental framework was designed to directly compare the performance of an LLM-based signal estimation method against an EKF baseline under identical conditions.

The experimental process starts by generating synthetic multi-sequence time series data that incorporate shared low-frequency components, unique high-frequency patterns, nonlinear transformations, and both Gaussian and non-Gaussian noise. This dataset creates a controlled yet complex scenario, allowing known ground truth for evaluation while introducing signal characteristics that challenge traditional linear filtering. The EKF is applied independently to each sequence, leveraging known model parameters for state estimation but lacking mechanisms to model latent correlations between sequences. In contrast, the LLM is trained to simultaneously capture temporal dependencies within each sequence and inter-sequence relationships by processing the multi-dimensional sequence jointly through self-attention mechanisms. This approach treats the multi-sequence time series as a structured input, enabling the model to learn both within-sequence and cross-sequence patterns without requiring explicit system equations. Both EKF and LLM are tested on the testing set which is the last 20% of the dataset, with outputs compared against the basic signals using quantitative evaluation metrics such as mean squared error (MSE), mean absolute error (MAE), and signal-to-noise ratio (SNR) to evaluate the general performance, and adopt other evaluation metrics to further analyze the capabilities of processing signals. Computational efficiency, including inference time, is also measured to assess practical feasibility.

The experimental design provides a direct empirical comparison between a classical filtering method based on explicit system modeling and a data-driven deep learning approach capable of modeling latent dependencies. This framework enables not only a perfor-

mance comparison but also insights into the potential of LLMs to serve as an alternative paradigm for time series signal estimation under challenging conditions.

3.1 Dataset Generation

The dataset used in this research was synthetically generated to simulate multiple correlated time series signals under controlled conditions. A key motivation for generating synthetic data was to ensure the availability of known ground truth signals, enabling quantitative evaluation of estimation accuracy. Moreover, synthetic generation allows explicit incorporation of nonlinear system dynamics, non-Gaussian noise, and latent inter-sequence dependencies, which are difficult to isolate or control in real-world datasets.

And the first 3000 time step of sequence 1 in generated data is shown below, I attach the images of the other two sequences in Appendix 2.

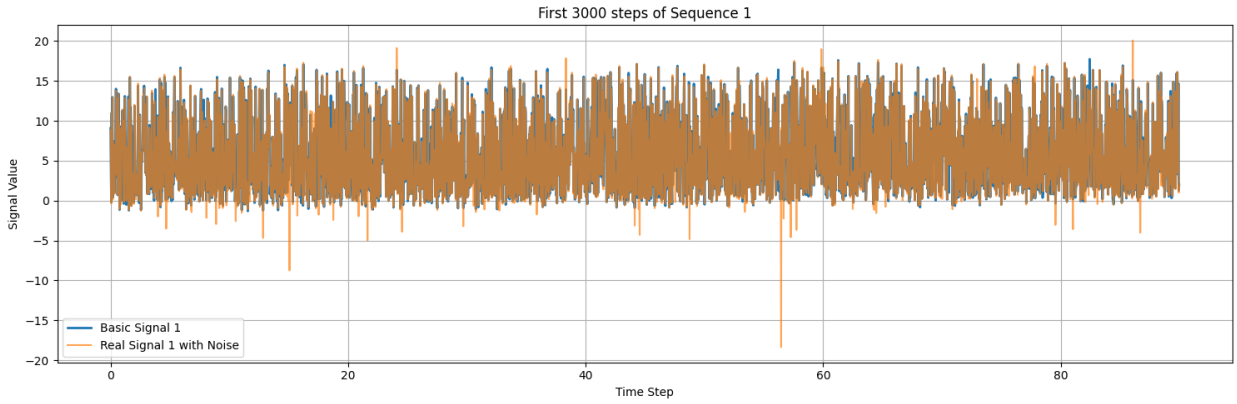

Figure 1. The first 3000 time step signals of the sequence 1.

3.1.1 Data Simulation Design

Each simulated time series signal consists of a combination of low-frequency shared components, sequence-specific high-frequency variations, nonlinear distortions, and noise contributions. Mathematically, the basic signal (which is the ground truth) for sequence i at time t is defined as:

$$x_i(t) = d_{shared}(t) + f_{low}(t) + f_{high,i}(t) + f_{nonlinear,i}(t), \tag{12}$$

where $d_{shared}(t)$ represents a cumulative drift term common to all sequences, $f_{low}(t)$ denotes a low-frequency sinusoidal signal shared across sequences, $f_{high,i}(t)$ is a unique high-frequency sinusoidal perturbation per sequence, and $f_{nonlinear,i}(t)$ is a nonlinear transformation applied to each sequence individually. Additionally, latent correlations between sequences are induced through shared drift and low-frequency components. While each sequence also has unique higher frequency patterns and nonlinear distortions, their correlation structure reflects underlying hidden factors, which is a common scenario in engineering and economic systems.

3.1.2 Noise Characteristics

Noise is incorporated into the observation by adding two types of noise: Gaussian noise and non-Gaussian spike noise modeled by a Pareto distribution. The observed noisy signal is defined as:

$$y_i(t) = x_i(t) + \epsilon_{gaussian,i}(t) + \epsilon_{nonGaussian,i}(t), \tag{13}$$

where $\epsilon_{gaussian,i}(t) \sim \mathcal{N}(0, \sigma^2)$ and $\epsilon_{nonGaussian,i}(t)$ represents occasional extreme noise

drawn from a Pareto(α) distribution.

3.1.3 Implementation and Parameter Settings

The dataset generator was implemented as a Python script, parameterized by total length, sampling frequency, noise levels, and nonlinear transformation functions. The complete codes are provided in the appendix.

This Python implementation builds each sequence by adding shared low-frequency signals and drift components, followed by unique high-frequency and nonlinear components. Then Gaussian noise and Pareto-distributed outliers are added to each sequence to simulate realistic noisy conditions.

The synthetic dataset was generated using fixed parameter values to ensure experimental reproducibility and controlled evaluation of signal estimation methods. The key parameters and their values are summarized in Table 1. These values were selected based on signal characteristics commonly observed in real-world GNSS and financial time series. All parameters except drift are fixed across dataset generation to ensure consistent evaluation and reproducibility. Randomness is internally introduced via the sampling of Gaussian noise, spike events, and phase shifts. The drift component is computed by cumulatively summing values drawn from the specified uniform range. A sequence length of 10,000 corresponds to approximately 2 hours and 46 minutes of continuous GNSS signal monitoring at a 1 Hz sampling rate, or proportionally shorter durations under higher-frequency sampling settings. Additionally, I must consider the capability of my GPU. Three parallel sequences were generated to reflect multivariate time series signals where shared low-frequency drift and cross-series latent correlations are prevalent in real-world scenarios.

Table 1. Simulation parameter settings and the rationale..

Parameter	Value	Rationale
sequence length (seq_len)	10000	Provides sufficient temporal resolution for long-range dependency learning and filtering evaluation.
number of sequences	3	Simulates multivariate time series from multiple sensors or modalities with latent correlations.
nonlinear_strength	0.4	Introduces strong nonlinear dynamics, modeling GNSS signal bending or market regime switches.
gaussian_std	0.2	Represents regular measurement noise common in GNSS receivers or financial tick data.
spike_probability	0.05	Simulates sparse but extreme outliers like GNSS spikes or market shocks.
pareto_alpha	2.0	Controls the heavy-tail behavior of spike amplitude, consistent with extreme-value noise.
pareto_scale	2.0	Sets the base magnitude scale of spike values.
skew_neg_prob	0.7	Simulates asymmetry in outlier direction, allowing more negative spikes (e.g., signal dropout).
drift_strength_range	(0.0001, 0.0005)	Defines the uniform range from which every time step drift values are sampled to simulate shared trend.

3.1.4 Design Rationale

Each design element in the synthetic dataset was deliberately chosen to mimic real-world sources of noise and variation observed in GNSS and financial time series signal. The shared drift term represents systematic errors like ionospheric delays in GNSS (Misra & Enge, 2006) or macroeconomic trends in financial markets (Cont, 2001). Shared low-frequency components simulate long-term correlated errors or slow-varying factors affecting multiple channels simultaneously. Unique high-frequency variations capture individual signal fluctuations, such as satellite-specific multipath interference (Hegarty & Kaplan, 2005) or asset-specific volatility (Pagan & Schwert, 1990). The inclusion of nonlinear distortions reflects nonlinear signal propagation paths in GNSS ranging models (Hegarty & Kaplan, 2005) or nonlinear price formation mechanisms in financial markets (Engle, 1982). Gaussian noise models typical measurement errors in both domains, while Pareto-

distributed spike noise was specifically selected to capture rare but extreme disturbances. The Pareto distribution, defined by:

$$f(x; \alpha, x_m) = \frac{\alpha x_m^{\alpha}}{x^{\alpha+1}}, \quad x \geq x_m > 0, \tag{14}$$

It is a well-known heavy-tailed distribution, meaning that extreme values have non-negligible probability (Mandelbrot, 1967). This property makes it suitable for modeling the heavy-tailed behavior empirically observed in financial returns and the sporadic extreme measurement errors in GNSS (Ehrlich, Callot, & Aubet, 2022). Compared to other heavy-tailed distributions such as Cauchy or t-distributions, the Pareto distribution offers a balance between tail heaviness and parameter controllability through α, enabling flexible tuning of spike severity while maintaining interpretability and simplicity. Both positive and negative spikes were included to reflect over- and underestimation scenarios in measurement processes. In total, these components ensure the dataset provides a challenging yet realistic benchmark for testing signal estimation methods under nonlinear, non-Gaussian, and correlated conditions.

3.2 EKF Implementation

This section describes the implementation of an EKF used as a baseline signal estimation method in this study. The EKF was independently applied to each time series signal to estimate the latent clean signals from the noisy observations generated in the dataset described earlier. This implementation follows the standard recursive filtering framework, using prior knowledge about system noise characteristics, and serves as a comparison point against the proposed LLM-based signal estimation method under identical data and evaluation conditions.

3.2.1 EKF Algorithm Design

The EKF is employed as a model-based approach for estimating the real state of multi-sequences time series signals in this experiment and it will be compared with the LLM-based method. EKF extends the standard Kalman Filter to handle nonlinear system dynamics by linearizing the nonlinear function around the current estimate using a first-order Taylor expansion.

In this experiment, the EKF uses a state transition function matching the nonlinear dynamics defined in the dataset generation:

$$x_k = f(x_{k-1}) + w_{k-1} \tag{15}$$

where

$$f(x) = \alpha(x \bmod 2\pi)^2 \tag{16}$$

with $\alpha = 0.4$ as the nonlinear strength parameter consistent with the simulation.

The observation model is linear:

$$z_k = x_k + v_k \tag{17}$$

where $w_{k-1} \sim \mathcal{N}(0, Q)$ and $v_k \sim \mathcal{N}(0, R)$ represent process and observation noise, respectively.

The Jacobian matrix of the nonlinear function is computed as:

$$F_k = \frac{df}{dx} = 2\alpha(x \bmod 2\pi) \tag{18}$$

The EKF recursively predicts and updates the state estimation using:

$$x_{k|k-1} = f(x_{k-1|k-1}) \tag{19}$$

$$P_{k|k-1} = F_{k-1} P_{k-1|k-1} F_{k-1}^T + Q \tag{20}$$

$$K_k = P_{k|k-1} H^T (H P_{k|k-1} H^T + R)^{-1} \tag{21}$$

$$x_{k|k} = x_{k|k-1} + K_k (z_k - x_{k|k-1}) \tag{22}$$

$$P_{k|k} = (I - K_k H) P_{k|k-1} \tag{23}$$

with $H = 1$ as the derivative of the observation function $h(x) = x$.

This formulation preserves consistency with the known system dynamics used in data generation, making itself an upper-bound baseline for model-driven filtering approach when the model assumptions are completely correct. This design is an EKF fuction for processing a single sequence, as no explicit multi-sequence dynamic function was defined in generated simulation dataset. So, potential latent inter-series dependencies cannot be directly incorporated into the EKF state model.

3.2.2 Implementation

The EKF was implemented in Python using NumPy to process the simulated datasets. The complete codes are provided in the appendix. Each time series signal was filtered independently by an EKF function following the algorithm in previous section. While EKF theoretically supports multivariate state-space models, it still treats each sequence independently in this experiment due to the lack of explicit functional relationships which can model latent interdependencies across sequences. Without a known multi-sequence dynamic function, the EKF cannot incorporate cross-series dependencies in the implementation.

The filter was configured with fixed hyperparameters: $Q = 0.01$, $R = 0.04$, $x_0 = z_0$, and $P_0 = 1.0$. The process noise variance Q was chosen to reflect relatively low process un-

certainty, while the observation noise variance R matches the Gaussian noise variance used in the dataset ($\sigma^2 = 0.04$). The initial state x_0 was set equal to the first observation z_0, assuming no prior knowledge of the true initial state to maintain comparable initial conditions with the LLM inference process, which also does not access ground truth during testing. The initial covariance P_0 was set to 1.0, representing moderate initial uncertainty.

The output state estimates were stored independently for each sequence to facilitate later evaluation. This per-sequence EKF implementation thus provides a model-based approach for estimating time series signals under ideal model knowledge but without leveraging latent inter-sequence dependencies.

3.3 LLM-based Signal Estimation Framework

This section introduces the LLM-based signal estimation approach, which is implemented as a Transformer-based neural network. The LLM is designed to process multivariate time series signals and directly learn to estimate latent basic signals from noisy observations (real signals) under conditions of nonlinear dynamics, non-Gaussian noise, and latent inter-series dependencies.

Unlike the EKF method, which independently filters each sequence based on a predefined state-space model, the LLM-based approach here adopts a data-driven method without requiring an explicit system model. Instead, the LLM leverages self-attention mechanisms to automatically capture both temporal dependencies and latent correlations across multiple sequences during training like processing the human speech.

There are three core components in this design, the first one is a neural network built upon a Transformer encoder to model complex temporal and inter-series relationships, and the second is a multi-objective loss function designed to enhance robustness against noise, outliers, and nonlinear distortions, the last one is a training and inference frame-

work that applies sliding window learning and confidence-based output aggregation to improve estimation reliability and reduce the consumption on a consumer grade GPU.

A schematic illustration of the model structure is provided in Figure 2. The Figure 25 shows the process and time of training stage and inference stage. The following subsections introduce the model design, loss function construction, and experimental procedure.

3.3.1 Transformer Architecture

This transformer architecture is named `TransformerDenoise` in codes which is attched as the appendix, and the scheme is illustrated in Figure 2

The architecture consists of an input projection layer, a stack of Transformer encoder layers, an output projection layer, and a confidence estimation head. The input projection transforms each 3-dimensional input vector at each time step into a $d_{model} = 64$-dimensional embedding. This embedding is then passed through two layers of Transformer encoders, each with 4 attention heads. Positional encoding is implicitly integrated in the PyTorch `TransformerEncoderLayer` to maintain temporal order information.

The encoded representation is fed into two parallel branches: one for signal estimation and the other for confidence prediction. The output projection maps the encoded features back to a 3-dimensional space which is consistent with the input shape, yielding the denoised signal estimate. The confidence head, implemented as a two-layer feedforward network with ReLU and sigmoid activations, outputs a confidence score between 0 and 1 for each time step and sequence (I will discuss the details in section 3.3.3), which is used during inference to aggregate overlapping predictions.

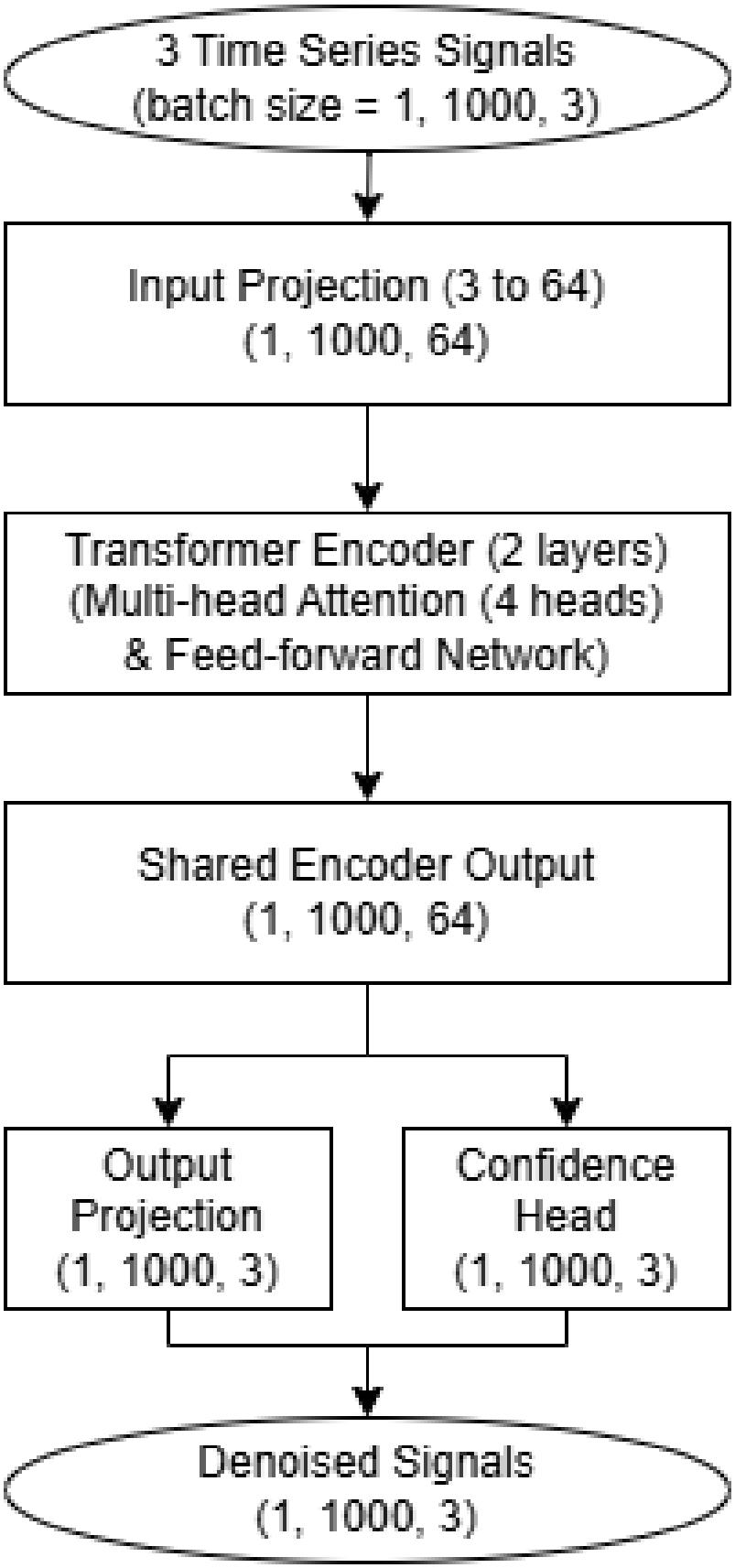

Figure 2. Architecture of the TransformerDenoise model.

The choice of model hyperparameters included $d_{model} = 64$, $n_{head} = 4$, and $num\_layers = 2$, they are not only constrained by the use of a consumer-grade GPU (NVIDIA RTX 3060, 8GB VRAM), but more importantly, reflects the relative simplicity and regularity of the time series signals compared to natural language. The time series signals exhibit smooth temporal variation, low dimensionality, and consistent patterns like shared drift and nonlinear perturbations. Consequently, a compact model is sufficient to capture the essential temporal dynamics and latent inter-sequence dependencies without excessive model depth or parameter count. If I choose some larger hyperparameters for this model, it would lead to a kind of computing resources waste due to there could be many attention heads work for the repeated task.

This design leverages the self-attention mechanism of Transformer to model long-range dependencies (time in this study) and cross-sequence interactions without relying on explicit system dynamics or observation models.

3.3.2 Loss Function Design

The proposed LLM-based signal estimation model is trained using a composite loss function tailored to the challenges of denoising nonlinear, non-Gaussian, and multivariate time series data. The simulation environment reflects realistic conditions found in GNSS and financial markets, such as shared low-frequency drift, sequence-specific nonlinear patterns, and both Gaussian and heavy-tailed noise components (e.g., Pareto spikes). Thus, the loss function must be designed to balance five desirable properties, including directional accuracy, spectral fidelity, smoothness, outlier robustness, and asymmetry awareness.

First, the vital component is a directional MSE, which penalizes not only magnitude error but also trend disagreement between estimates and the ground truth. For each channel

i, the directional MSE is computed as:

$$\mathcal{L}_{\text{dir}}^{i} = \left\| \hat{Y}_i - Y_i \right\|^2 \cdot (1 + w \cdot (1 - \cos(\theta_i))) \tag{24}$$

where $\cos(\theta_i) = \frac{\hat{Y}_i^\top Y_i}{\|\hat{Y}_i\| \cdot \|Y_i\|}$ is the cosine similarity and w is a weighting factor. This encourages trend-aligned predictions, especially important when filtering GNSS trajectories or market trends, without this directional setting, the model could learn the opposite trend (Yin, 2023).

Second, a frequency domain loss is adopted to suppress high frequency distortions:

$$\mathcal{L}_{\text{freq}} = \text{MSE}\left(|\mathcal{F}(\hat{Y})|, |\mathcal{F}(Y)| \right) \tag{25}$$

This ensures that the denoised output retains the spectral characteristics of the original signal, which is especially useful in the presence of high frequency spikes or periodicities. Prior research has shown that frequency-aware loss functions can significantly improve denoising quality in audio and biomedical signals (Peng et al., 2025).

Third, a total variation (TV) loss is applied as:

$$\mathcal{L}_{\text{TV}} = \frac{1}{T-1} \sum_{t=1}^{T-1} \left| \hat{Y}_{t+1} - \hat{Y}_t \right| \tag{26}$$

TV loss is widely used in image and signal processing to enforce local smoothness while allowing sharp edges, making it well-suited to real-valued temporal sequences with underlying continuity (Rudin, Osher, & Fatemi, 1992).

Fourth, the Huber loss is introduced as:

$$\mathcal{L}_{\text{Huber}} = \begin{cases} \frac{1}{2}(\hat{Y} - Y)^2 & \text{if } |\hat{Y} - Y| \leq \delta \\ \delta \cdot (|\hat{Y} - Y| - \frac{1}{2}\delta) & \text{otherwise} \end{cases} \tag{27}$$

Huber loss performances well when dealing with medium-scale noise or small signal oc-

clusions such as GNSS shadowing or financial anomaly windows (Wen, Gao, Song, Sun, & Tan, 2019).

Fifth, to capture asymmetric and heavy-tailed errors, a quantile regression loss is used follow the formulation below:

$$\mathcal{L}_{\text{quantile}} = \mathbb{E}\left[\max\left(q(Y-\hat{Y}), (q-1)(Y-\hat{Y})\right)\right] \tag{28}$$

This formulation is aligned with noise generated by Pareto distributions in the simulation and is proven effective in modeling skewed distributions in financial econometrics (Nirei & Aoki, 2016).

The final loss function is the weighted sum as below:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{dir}} + \alpha\mathcal{L}_{\text{freq}} + \beta\mathcal{L}_{\text{TV}} + \gamma\mathcal{L}_{\text{Huber}} + \delta\mathcal{L}_{\text{quantile}} \tag{29}$$

with the weights $\alpha = 0.1, \beta = 0.01, \gamma = 0.05, \delta = 0.1$. Because the directional MSE and frequency loss is vital for processing the nonlinear characteristics, and Huber loss and quantile loss are vital for processing non-Gaussian noise especially there are a few spike noise in the simulated dataset, so the weight of quantile loss is also set to 0.1. While tv loss would suppress nonlinearity if the weight is set too big.

In summary, this multi-term loss formulation directly reflects both the statistical structure of real-world time series signals and the synthetic noises modeled in this study. This formulation not only support a comprehensive learning strategy that goes beyond simple pointwise regression, but also make it suitable for complex time series signals processing tasks.

3.3.3 Sliding Window and Data Preprocessing

I adopt a sliding window approach with window size $W = 512$ and stride $S = 256$ in order to achieving 50% overlap between segments. This overlap ensures that each time step is covered by at least two windows, which reducs information lost and keep the suitable attention length for time series processing. Each input window is standardized using zero-mean and unit-variance normalization via StandardScaler to improve training stability, especially under non-Gaussian noise.

3.3.4 Training Stage

The model is trained on standardized noisy signals $\mathbf{X}_{\text{real}}$ using sliding windows. The target is the corresponding clean signal $\mathbf{Y}_{\text{basic}}$. This Transformer-based model outputs both denoised predictions $\hat{\mathbf{Y}}$ and confidence scores $\hat{\mathbf{C}}$ for each time step:

$$\hat{\mathbf{Y}}, \hat{\mathbf{C}} = \text{Transformer}(\mathbf{X}_{\text{real}}) \tag{30}$$

The training objective is to minimize the loss function which is described in previous section. And the optimizer here I choose Adam with learning rate 0.001, which is suitable for adaptive learning in deep sequence models.

3.3.5 Inference Stage and Confidence-Weighted Fusion

In the inference stage, each test segment is also processed by the sliding window mechanism. Predictions are aggregated at the time-step level using confidence-weighted fusion as follow:

$$\hat{\mathbf{Y}}(t) = \frac{\sum_i \hat{\mathbf{Y}}_i(t) \cdot \hat{\mathbf{C}}_i(t)}{\sum_i \hat{\mathbf{C}}_i(t)} \tag{31}$$

This fusion strategy is better than others generally used method because it acts as a locally normalized weighted expectation over overlapping predictions, aligning with soft uncertainty modeling in Bayesian filtering while being end-to-end differentiable, and the comparison with other methods is shown in Table 2.

Table 2. Comparison of Fusion Strategies for Overlapping Window Predictions.

Method	**Mechanism**	**Pros**	**Drawbacks**
Simple Average	Uniform averaging across overlapping predictions	Easy to implement and need no extra computation	Treats all predictions equally, which potentially amplify boundary artifacts
Softmax Fusion	Apply softmax over a score vector and weight predictions	Highlights relatively strong predictions	If all predictions are poor, weights are still normalized, leading to miscalibrated output.
Confidence-Weighted Fusion	Like the formulation showed above	Captures absolute prediction reliability, which is robust to noisy windows.	Requires time to learn confidence scores during training.

3.4 Evaluation Design

This section presents the evaluation framework used to assess and interpret the performance of the proposed LLM-based time series signal estimation method in comparison with the EKF. Rather than relying solely on conventional accuracy metrics such as MSE, MAE, and SNR, I adopt a more comprehensive evaluation approach that incorporates error structure, temporal dependencies, and inter-sequence relationships, which enable not only performance comparison but also insight into the underlying modeling capabilities of each method.

3.4.1 MSE, MAE, SNR

To quantify the estimation accuracy from a general view, I use three widely used metrics including MSE, MAE, and SNR. Each metric offers a different sensitivity to noise characteristics, which is critical for evaluating performance on the simulated data for both EKF and LLM.

The MSE is defined as:

$$\text{MSE} = \frac{1}{N}\sum_{t=1}^{N}(\hat{x}(t) - x(t))^2, \tag{32}$$

where $x(t)$ is the basic signal and $\hat{x}(t)$ is the estimated output. MSE is especially sensitive to large errors because of the square, which makes it suitable for detecting outlier vulnerability, especially those non-Gaussian noise set in this study because those noise distribution would make a larger difference with the ground truth, and MSE amplifies this gap so that MSE looks like paying more attention on evaluating the accuracy of denoising by different methods.

MAE provides a more robust measure in the presence of heavy-tailed noise:

$$\text{MAE} = \frac{1}{N}\sum_{t=1}^{N}|\hat{x}(t) - x(t)|. \tag{33}$$

Unlike MSE, there is no a square which makes MAE less sensitive to those spike and other non-Gaussian noise, paying more attention on the overall accuracy of the estimates.

SNR is one of the most commonly used metric in signal processing (Oppenheim, Schafer, & Buck, 1999), it is defined as:

$$\text{SNR} = 10\log_{10}\left(\frac{\sum x(t)^2}{\sum(\hat{x}(t) - x(t))^2}\right). \tag{34}$$

If the capabilities of two models are similar, it can still clearly show which one is better

due to the logarithmic operation.

3.4.2 Error Spectrum

To find which parts of frequency in signals are not estimated well, I compute the error spectrum using the Discrete Fourier Transform (DFT) of the residual signal $e(t) = \hat{x}(t) - x(t)$. The spectral magnitude is:

$$E(f) = |\text{DFT}(e(t))|, \tag{35}$$

where f is the normalized frequency. This spectrum can show whether low-frequency trends (e.g., shared low frequency caused by shared drift) or high-frequency trends (e.g., unique spikes) are captured correctly by models. This frequency domain residual analysis method is widely used in system identification and signal denoising (Widrow & Stearns, 1975).

3.4.3 Error Histogram

To understand the statistical properties of the estimation error, I examine the empirical distribution of residuals using histograms. Unlike MSE or MAE which only provide scalar summaries, the histogram reveals whether the errors are symmetrically distributed, centered near zero, or exhibit skewness or heavy tails.

Let the residual at time t be $e(t) = \hat{x}(t) - x(t)$. Over N samples, the histogram provides a discretized approximation of the probability density function (PDF) of $e(t)$. Formally, if $\mathcal{B}i$ is the i-th bin in a partition of the error range, the empirical PDF is:

$$p_i = \frac{1}{N} \sum t = 1^N \mathbb{I}[e(t) \in \mathcal{B}_i], \tag{36}$$

where $\mathbb{I}[\cdot]$ is the indicator function. This statistical representation is particularly useful for evaluating the effect of heavy-tailed noise introduced in the simulation via Pareto

distributions.

An ideal model will produce residuals that are symmetrically centered around zero and sharply peaked. Deviations from this shape (e.g., skewness or heavy tails) can indicate model bias or sensitivity to outliers. This kind of distributional diagnostics are common in robust statistics and regression analysis (Rousseeuw & Leroy, 1987).

To show the rationale behind this analysis, Table 3 summarizes key histogram features and their diagnostic implications:

Table 3. Interpretation of histogram features in residual analysis.

Feature	**Indication**	**Histogram Behavior**
Centered at 0	Unbiased estimates	Peak at zero
Symmetry	No directional bias	Left and right sides balanced
Sharp peak	High confidence, low error spread	Tall narrow shape
Heavy tails	Sensitivity to outliers	Wide flat edges
Skewness	Systematic over/underestimation	Asymmetric sides

3.4.4 Autocorrelation

Temporal dependencies in the residuals are examined using the autocorrelation function as follow:

$$R_e(\tau) = \frac{1}{N} \sum_{t=1}^{N-\tau} e(t)e(t+\tau), \tag{37}$$

where τ is the time lag. Ideally, estimation errors should be temporally uncorrelated, suggesting no systematic drift or lag remains. If autocorrelation decays slowly, this indicates that the model fails to track long-term trends, such as the shared drift term in the simulated data. Residual autocorrelation is a classic method for validating time series models (Box & Jenkins, 1976).

3.4.5 Error Correlation

To test whether a model captures shared dynamics among multiple time series sequences, I compute Pearson correlation coefficients between residuals of different sequences:

$$\rho_{ij} = \frac{\text{Cov}(e_i, e_j)}{\sigma_{e_i}\sigma_{e_j}}, \tag{38}$$

where $e_i(t)$ and $e_j(t)$ are the residuals from sequences i and j, respectively. A model that jointly learns from all sequences may reflect weak correlations due to shared latent components (e.g., shared drift and shared low-frequency trends). In contrast, independently filtered sequences (e.g., via EKF) should show near-zero correlation. This type of residual cross-correlation analysis is grounded in multivariable system identification theory (Ljung, 1999).

4 Results and Analysis

This chapter presents the experimental results and interprets them in terms of accuracy, residual structure, and modeling behavior. All evaluations are conducted using the synthetic dataset described in Chapter 3, with identical conditions for both the EKF and the proposed LLM-based model.

4.1 General Performance Comparison (MSE, MAE, SNR)

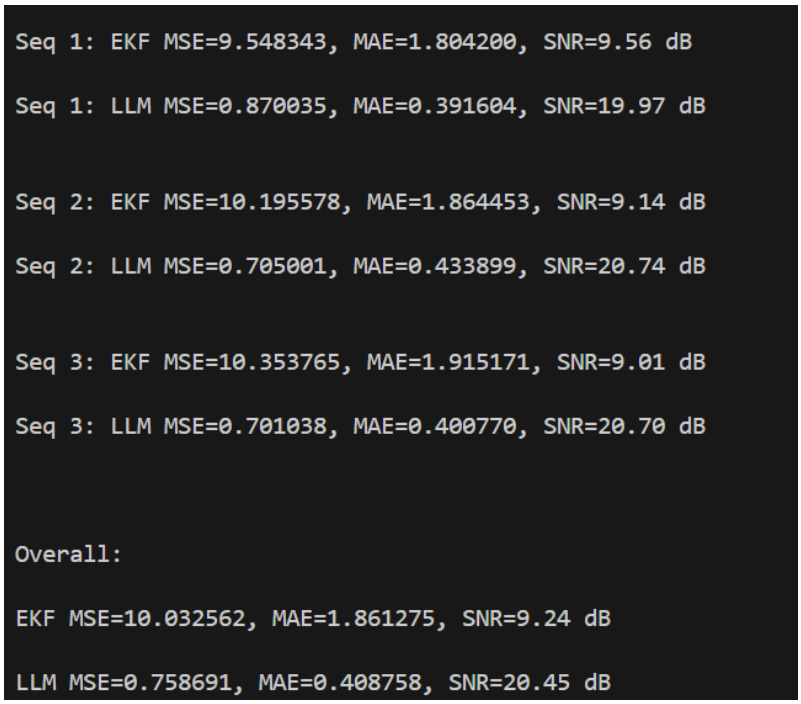

Figure 3. MSE, MAE and SNR Comparison based on EKF and LLM.

The Figure 3 shows the LLM outperforms the EKF across all three sequences in every metric. On average, the LLM reduces MSE by more than 90%, MAE by more than 75%, and improves SNR by over 11 dB. And the Figure 4 shows the comparision results between EKF and LLM in sequence1, and the other two are attached to the Appendix 2.

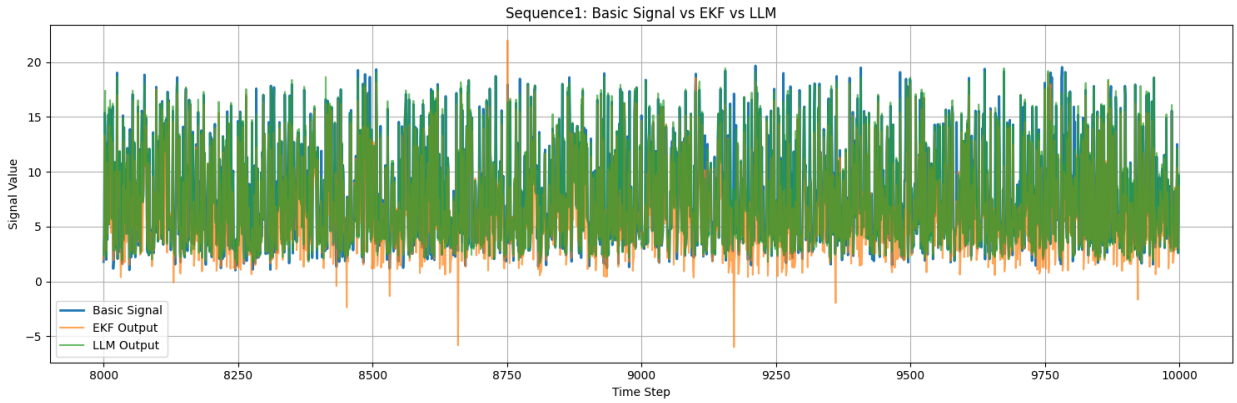

Figure 4. Comparision between EKF and LLM in sequence 1.

These results reveal the LLM has more superior ability to denoise the nonlinear, non-Gaussian time series which exhibit interdependent or structurally linked dynamics than EKF.

4.2 Frequency-Domain Residual Analysis

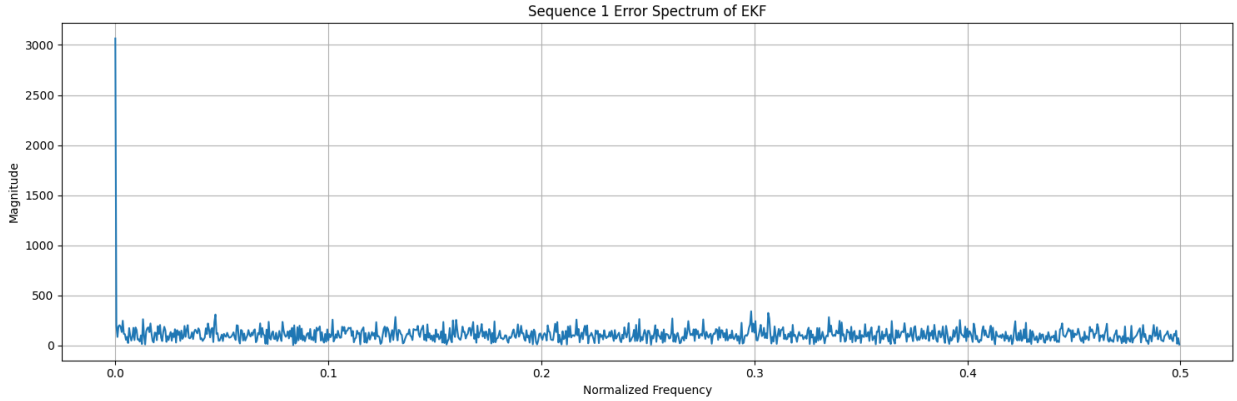

Figure 5. Error spectrum of the sequence 1 generated by EKF.

Figure 5 and Figure 6 compares the error spectrum of EKF and LLM for Sequence 1. The EKF spectrum exhibits a strong peak at low frequencies, indicating residual drift or slowly varying structure that the model failed to capture. In contrast, the LLM spectrum is flat-

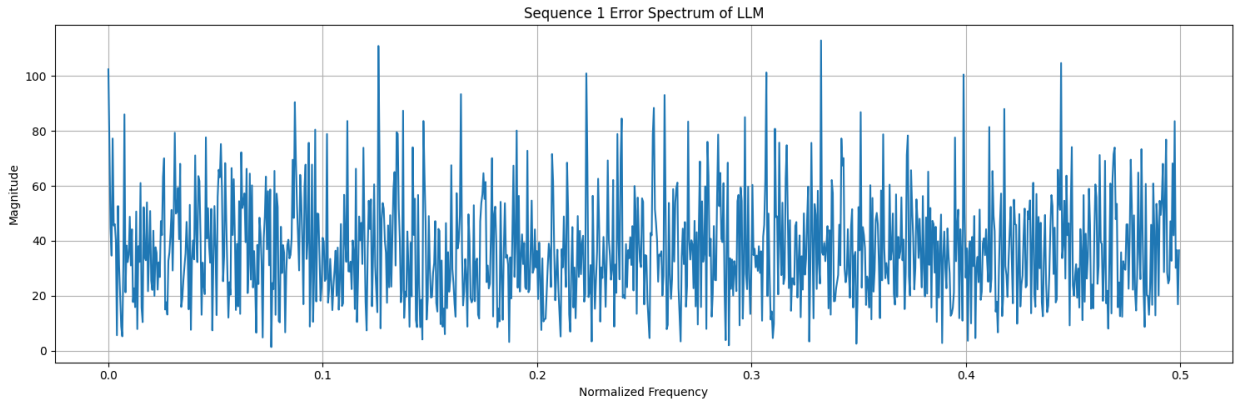

Figure 6. Error spectrum of the sequence 1 generated by LLM.

ter. This pattern is consistent across other sequences (see Appendix 2). The strong low-frequency residual energy in EKF outputs suggests that it fails to remove shared drift and low-frequency sinusoidal structures. The LLM, by contrast, shows uniform spectral suppression, implying a more effective modeling of both global trends and localized fluctuations.

The frequency-domain residual analysis thus supports the hypothesis that the LLM learns richer temporal structures and nonlinear dependencies that the EKF cannot represent due to its linearized formulation and independence across sequences.

4.3 Distributional Characteristics of Residuals

Figure 7 and Figure 8 shows the residual histograms for Sequence 1. The EKF residuals exhibit strong left-skewness and heavy tails, including extreme values below -20, indicating susceptibility to large errors. As my expectation, EKF is not designed to handle non-Gaussian spikes or shared structural components. In contrast, the LLM residuals are sharply peaked around zero and nearly symmetric, with much shorter tails.

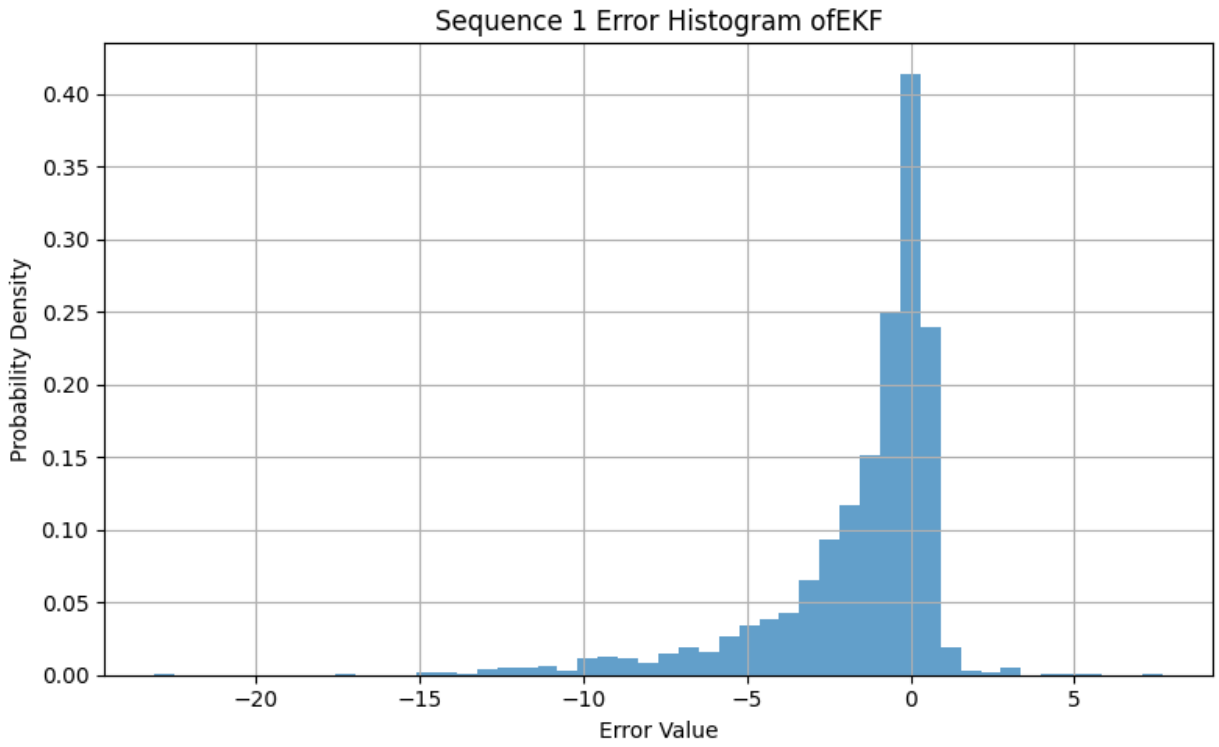

Figure 7. Error histogram of the sequence 1 generated by EKF.

These results validate the robustness of the LLM under non-Gaussian and heavy-tailed noise distributions. By producing residuals that are concentrated and unbiased, the LLM shows a more effective rejection of both impulsive disturbances and structural modeling errors compared with EKF.

4.4 Temporal Structure in Residuals

Figure 9 and Figure 10 presents the autocorrelation functions of EKF and LLM residuals for Sequence 1. The EKF residuals display a characteristic triangular pattern with slowly decaying tails, suggesting long-term correlations. This implies the EKF fails to remove certain trend-like or periodic behaviors from the data, likely due to its limited capacity to model

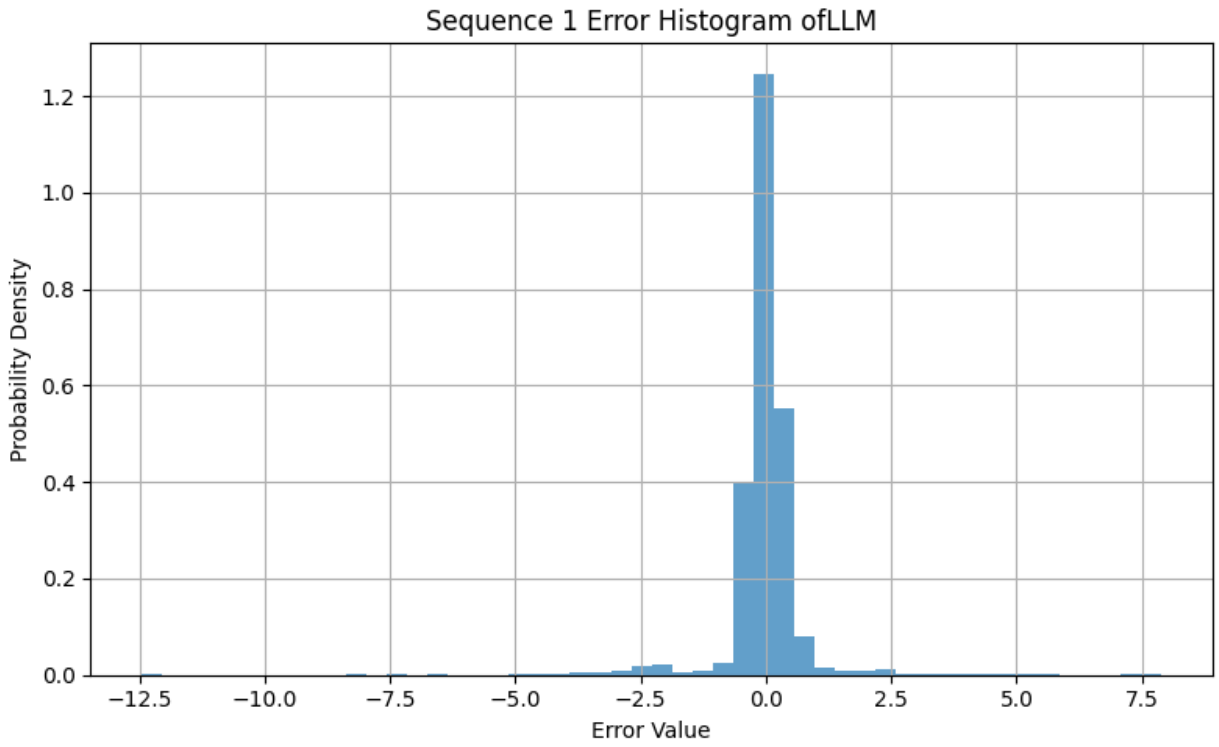

Figure 8. Error histogram of the sequence 1 generated by LLM.

nonlinear dynamics and those shared components designed in dataset. In contrast, the LLM residuals show near-zero autocorrelation across all lags except at lag zero, indicating a lack of temporal dependence. This suggests the LLM has successfully learned both local and global temporal structures, leaving behind only unstructured, random noise.

This temporal whitening of the residuals is a strong indicator of modeling completeness. And this shows LLM has stronger ability of modeling multiple time series signals which have latent correlation, while EKF cannot do this.

4.5 Inter-sequence Error Dependency

Figure 11 summarizes the residual correlation coefficients between each pair of sequences. As expected, the EKF residuals are nearly uncorrelated, with values close to zero, consis-

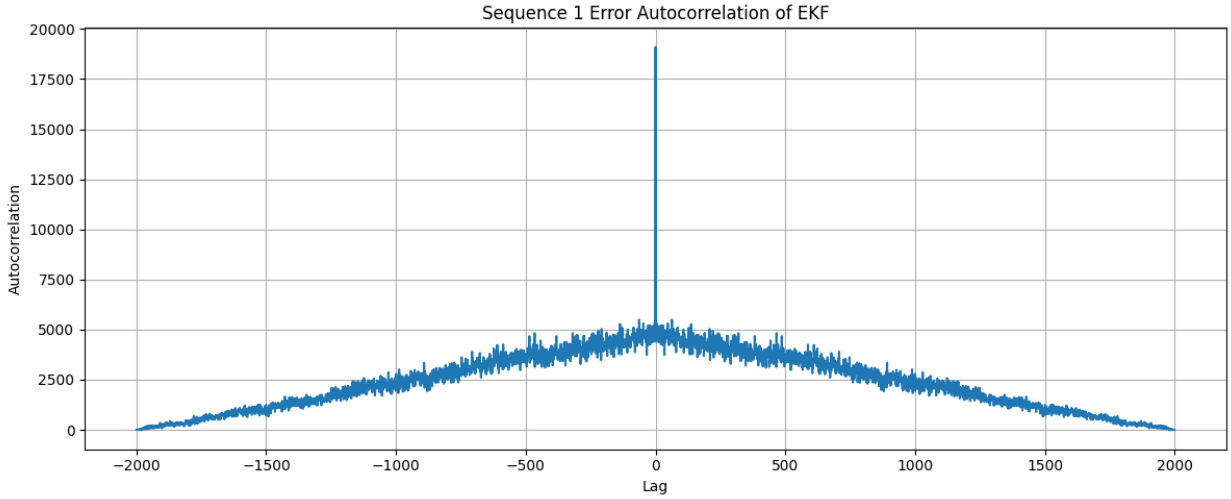

Figure 9. Error autocorrelation of the sequence 1 generated by EKF.

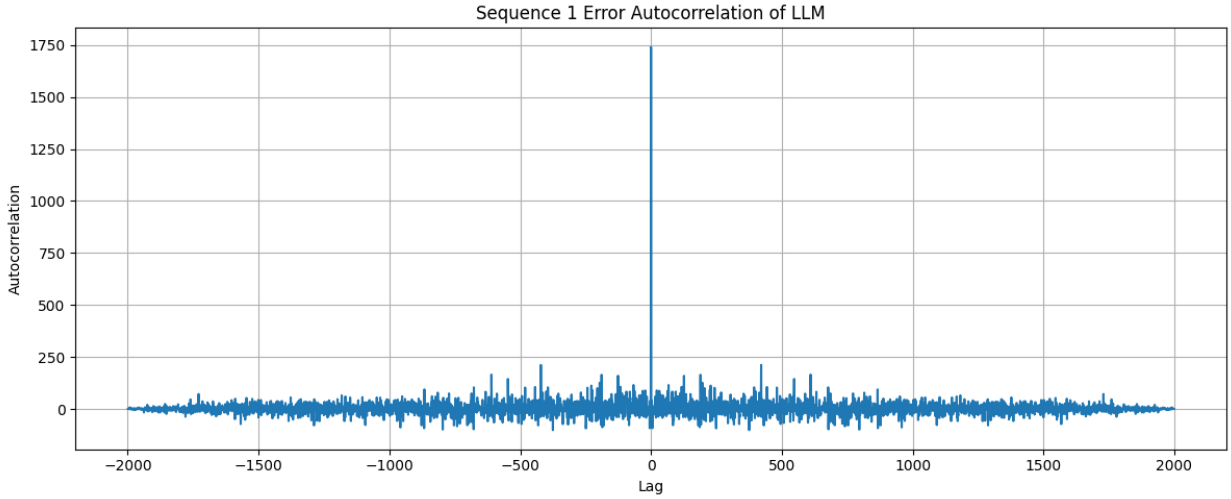

Figure 10. Error autocorrelation of the sequence 1 generated by LLM.

tent with its independent filtering process. In contrast, the LLM residuals exhibit weak but positive correlations, indicating that the model has leveraged shared information across sequences, even if imperfectly removed from the final outputs.

Although the ideal residuals should be independent under the assumption of complete modeling, the observed weak positive correlations in the LLM output are likely due to the model encoding shared latent trends across sequences. This aligns with the dataset design, which includes shared drift and low-frequency components across all signals. The EKF lack this modeling capacity, so it fails to exploit or reflect such relationships.

```
Error Correlations Across Sequences
EKF error correlations: seq1-seq2=-0.010, seq1-seq3=0.034, seq2-seq3=-0.024
LLM error correlations: seq1-seq2=0.023, seq1-seq3=0.019, seq2-seq3=0.043
```

Figure 11. Error correlation across different sequences.

This result shows LLM not only improves point-wise estmation accuracy, but also captures and utilizes structural dependencies in multi-sequence senarios, which is a big problem to EKF or other Kalman Filters.

4.6 Summary of Findings

In total, all the results shown above represent that the proposed LLM-based model not only reduces absolute error magnitudes but also better captures the latent interdependencies of multiple long time series signals, outperforming the EKF in both accuracy and model completeness. However, it is important to note that this advantage comes at a computational cost. While EKF operates with low per-step complexity and is easily deployable in real-time systems, the LLM involves attention mechanisms and deep sequence modeling, which require significantly more computational resources. These trade-offs must be considered when selecting a model for deployment, especially in resource-constrained or real-time environments.

5 Discussion and Future Work

This section presents a broader discussion of the proposed LLM-based approach adopted in the experiment. While the model demonstrates strong denoising performance on complex non-Gaussian and nonlinear time series signals, it is important to reflect on the design choices, identify limitations, and explore future research directions that could enhance its robustness and applicability in real-world scenarios.

5.1 Strengths and Novel Design Aspects

The proposed method introduces several design innovations that collectively contribute to its strong performance in complex signal processing tasks. These innovations are carefully tailored to the characteristics of non-Gaussian, nonlinear, and multi-sequence time series data.

There is a key novelty lies in the confidence-weighted fusion mechanism applied during inference. Unlike prior works that adopt direct averaging or softmax-based weighting across overlapping predictions, my approach assigns explicit, model-learned confidence scores to each time step, which are used to weight the predictions in overlapping windows. This strategy enables the model to reflect uncertainty in a calibrated manner, ensuring that predictions from unreliable or boundary regions have reduced influence. To the best of my knowledge, such a mechanism has not been applied in Transformer-based time series denoising. I also formulate the training objective as a weighted combination of multiple loss functions, including directional mean squared error, frequency domain consistency, total variation smoothness, Huber robustness, and quantile-aware deviation. This multi-objective loss ensures that the model simultaneously captures structural fidelity, statistical consistency, and robustness to outliers, which are all critical in realistic time series signal processing scenarios.

Collectively, these design aspects distinguish our method from conventional Transformer-

based sequence models and provide a solid foundation for both practical deployment and theoretical extension.

5.2 Limitations and Challenges

While the proposed LLM-based signal estimation framework shows promising performance in denoising multi-sequence time series with nonlinear and non-Gaussian characteristics, its current implementation exhibits several practical limitations rooted in the design of the architecture and execution of the code.

First, although the inference is performed on fixed-length windows (size 512) using a sliding approach with 50% overlap (stride 256), the aggregation of outputs and confidence scores requires pre-allocated tensors that span the entire test sequence length. As a result, GPU memory usage scales linearly with the sequence length, which may lead to memory exhaustion on longer signals. Moreover, PyTorch defaultly retains the computation graph during inference, further increasing memory consumption unintentionally.

Second, the inference loop is implemented as a sequential for-loop over overlapping windows, which is simple and memory-safe but causes runtime to grow linearly with the number of windows. On very long sequences, this leads to noticeable latency. Additionally, all windows are processed independently on the GPU without batching, which misses potential acceleration via parallel execution or vectorization.

Finally, the model has so far only been tested on synthetic three-sequences time series data. While the architecture is theoretically extensible to higher-dimensional signals, its generalization to real-world sensor data with heterogeneous structures and distributions remains unvalidated. Dynamic adaptation to time-varying signal properties or distributional shifts is also not supported in the current static training pipeline.

5.3 Future Work

Future work should focus on extending the proposed framework toward models that not only estimate denoised time series but also generate structured, interpretable scene-level outputs from multimodal inputs. Instead of producing dense sequences aligned with input resolution, the goal would be to derive higher-order representations such as diagnostic event logs, financial summaries, or operational insights directly from raw signal streams. This presents both architectural and semantic challenges, requiring the model to abstract, align, and structure multiple types of temporal information.

One envisioned direction is to refine the Transformer architecture to operate as a multimodal scene understanding system. Inputs could consist of time-synchronized sensor readings, control signals, or even auxiliary modalities such as images or texts. Through modality-aware encoding layers followed by shared self-attention, the model could learn to represent temporal patterns and cross-modality interactions. The key innovation lies in how outputs are structured, rather than returning time-aligned sequences, the model would decode slot-based latent representations that correspond to abstract elements of the underlying system, such as error modes, behavior primitives, or financial states.

Inspired by recent works on slot attention (Locatello et al., 2020) and set-based prediction (Carion et al., 2020), these abstractions could support output formats that match real-world reasoning processes. For example, a control system might generate structured event reports as a JSON document; a financial agent might summarize recent market dynamics and generate next-step suggestions. These outputs are interpretable, non-sequential, and often variable in size, which makes them suitable for set-based decoding architectures. This represents a shift from conventional dense modeling to sparse, modular, and semantically meaningful signal-to-structure translation.

To enable such capabilities, attention mechanisms may require redesign to better reflect local structure, causality, and uncertainty. Gated cross-modality attention, temporal rout-

ing, or adaptive slot binding are potential directions. In addition, pretraining strategies inspired by masked modeling or contrastive representation learning could improve the model's ability to generalize across domains and recognize reusable latent patterns.

Ultimately, this direction envisions time series modeling systems that serve not only as denoisers or predictors but as interpretable agents, which should be capable of understanding and reporting structured scene-level information from diverse, noisy, and multi-source signals. This paradigm aligns with emerging needs in industrial diagnostics, autonomous systems, and financial decision-making, where transparent and actionable reasoning from signal data is essential.

Bibliography

Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.

Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Bommasani, R., et al. (2021). On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*.

Box, G., & Jenkins, G. M. (1976). *Time series analysis: Forecasting and control*. Holden-Day.

Brown, T. B., et al. (2020). Language models are few-shot learners. *Advances in neural information processing systems*, *33*, 1877–1901.

Carion, N., Massa, F., Synnaeve, G., Usunier, N., Kirillov, A., & Zagoruyko, S. (2020). *End-to-end object detection with transformers*. Retrieved from https://arxiv.org/abs/2005.12872

Chowdhery, A., et al. (2022). Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

Cont, R. (2001). Empirical properties of asset returns: stylized facts and statistical issues. *Quantitative Finance*, *1*(2), 223–236.

Crassidis, J. L., & Junkins, J. L. (2011). *Optimal estimation of dynamic systems*. CRC Press.

Ehrlich, E., Callot, L., & Aubet, F.-X. (2022). *Spliced binned-pareto distribution for robust modeling of heavy-tailed time series*. Retrieved from https://arxiv.org/abs/2106.10952

Engle, R. F. (1982). Autoregressive conditional heteroscedasticity with estimates of the variance of united kingdom inflation. *Econometrica*, *50*(4), 987–1007. Retrieved from [2025-05-06]http://www.jstor.org/stable/1912773

Fons, E., Kaur, R., Palande, S., Zeng, Z., Balch, T., Veloso, M., & Vyetrenko, S. (2024). *Evaluating large language models on time series feature understanding: A comprehensive taxonomy and benchmark*. Retrieved from https://arxiv.org/abs/2404.16563

Grewal, M. S., & Andrews, A. P. (2001). *Kalman filtering: Theory and practice using*

matlab. John Wiley & Sons.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 770–778).

Hegarty, C., & Kaplan, E. (2005).

Julier, S. J., & Uhlmann, J. K. (1997). A new extension of the kalman filter to nonlinear systems. *Proceedings of AeroSense: The 11th International Symposium on Aerospace/Defense Sensing, Simulation, and Controls*, 182–193.

Kalman, R. E. (1960). A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, *82*(1), 35–45.

Liu, Y., Zhang, H., Li, C., Huang, X., Wang, J., & Long, M. (2024). *Timer: Generative pre-trained transformers are large time series models*. Retrieved from https://arxiv.org/abs/2402.02368

Ljung, L. (1999). *System identification (2nd ed.): theory for the user*. USA: Prentice Hall PTR.

Locatello, F., Weissenborn, D., Unterthiner, T., Mahendran, A., Heigold, G., Uszkoreit, J., ... Kipf, T. (2020). *Object-centric learning with slot attention*. Retrieved from https://arxiv.org/abs/2006.15055

Mandelbrot, B. (1967). The variation of some other speculative prices. *The Journal of Business*, *40*(4), 393–413. Retrieved from [2025-05-06]http://www.jstor.org/stable/2351623

Maybeck, P. S. (1994). *Stochastic models, estimation, and control* (Vol. 1). Academic Press.

Misra, P., & Enge, P. (2006). *Global Positioning System: Signals, Measurements, and Performance* (2nd edition ed.). Ganga-Jamuna Press, Lincoln MA.

News, M. (2024). Researchers use large language models to flag problems in time series data. *MIT News*. Retrieved from https://news.mit.edu/2024/researchers-use-large-language-models-to-flag-problems-0814

Nirei, M., & Aoki, S. (2016). Pareto distribution of income in neoclassical growth models. *Review of Economic Dynamics*, *20*, 25-42. https://doi.org/https://doi.org/10.1016/j.red.2015.11.002

Oppenheim, A. V., Schafer, R. W., & Buck, J. R. (1999). *Discrete-time signal processing* (Second ed.). Prentice-hall Englewood Cliffs.

Pagan, A. R., & Schwert, G. (1990). Alternative models for conditional stock volatility. *Journal of Econometrics*, *45*(1), 267-290. https://doi.org/https://doi.org/10.1016/0304-4076(90)90101-X

Peng, J., Lu, M., Li, B., Wang, J., Hu, W., & Liu, X. (2025). Frequency-aware denoising using a diffusion model for enhanced band-limited and white noise removal in x-ray acoustic computed tomography. *Medical physics*. Retrieved from https://api.semanticscholar.org/CorpusID:276257466

Proakis, J., & Manolakis, D. (2013). *Digital Signal Processing* . Pearson Deutschland. Retrieved from https://elibrary.pearson.de/book/99.150005/9781292038162

Ramesh, S., et al. (2023). Transformer models beyond language: Applications in other domains. *Journal of AI Research*, *74*, 1–23.

Rousseeuw, P., & Leroy, A. (1987). *Robust regression and outlier detection*. New York [u.a.]: Wiley. Retrieved from http://gso.gbv.de/DB=2.1/CMD?ACT=SRCHA&SRT=YOP&IKT=1016&TRM=ppn+025179764&sourceid=fbw_bibsonomy

Rudin, L. I., Osher, S., & Fatemi, E. (1992). Nonlinear total variation based noise removal algorithms. *Physica D: Nonlinear Phenomena*, *60*(1), 259-268. https://doi.org/https://doi.org/10.1016/0167-2789(92)90242-F

Tan, M., Merrill, M. A., Gupta, V., Althoff, T., & Hartvigsen, T. (2024). *Are language models actually useful for time series forecasting?* Retrieved from https://arxiv.org/abs/2406.16964

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, *30*.

Wan, E., & Van Der Merwe, R. (2000). The unscented kalman filter for nonlinear estimation. , 153-158.

Wen, Q., Gao, J., Song, X., Sun, L., & Tan, J. (2019). *Robusttrend: A huber loss with a combined first and second order difference regularization for time series trend filtering*. Retrieved from https://arxiv.org/abs/1906.03751

Widrow, B., & Stearns, S. D. (1975). *Adaptive signal processing*. Prentice-Hall.

Yin, H. (2023). Enhancing directional accuracy in stock closing price value prediction using a direction-integrated mse loss function. In *Proceedings of the 1st international conference on data analysis and machine learning (daml)* (pp. 119–126). SCITEPRESS. https://doi.org/10.5220/0012810200003885

Zhang, L., Shen, L., Zheng, Y., Piao, S., Li, Z., & Tsung, F. (2024). *Lemole: Llm-enhanced mixture of linear experts for time series forecasting*. Retrieved from https://arxiv.org/abs/2412.00053

Appendices

Appendix 1. Codes

```python
import numpy as np
import matplotlib.pyplot as plt

def generate_multi_tsdatasets(seq_len=10000,
                              nonlinear_strength=0.4,
                              gaussian_std=0.2,
                              spike_probability=0.05,
                              pareto_alpha=2.0,
                              pareto_scale=2.0,
                              skew_neg_prob=0.7,
                              drift_strength_range=(0.0001, 0.0005),
                              random_seed=0):
    """
    To generate a large simulated dataset under controlled conditions, incorporating specific nonlinearities and noise characteristics

    parameters:
        seq_len: Numbers of sequences
        nonlinear_strength: The scale of nonlinear curve
        gaussian_std: sigma value in gaussian distribution
        pareto_alpha: shape parameter of Pareto distribution controlling tail heaviness
        pareto_scale: scale parameter adjusting the base magnitude of spike amplitude
        skew_neg_prob: the probabilty of postive or negative of spike
        drift_strength_range: The accuracy will be dynamically changed as time goes
        random_seed: Make sure the result can be reproducible

    return:
        t: time
        basic_signal: The basic signal we would like to acheive without outliers
        real_signal: The real signal with outliers
    """
    np.random.seed(random_seed)

    t = np.linspace(0, 300, seq_len)

    #shared components(reserve kind of similarity or relationship between these 3 seequences)
    shared_phase_shift = np.random.uniform(0, 2*np.pi, size=seq_len)
    shared_drift = np.cumsum(np.random.uniform(*drift_strength_range, size=seq_len))
    shared_low_freq = np.sin(t + shared_phase_shift)

    basics = []
    reals = []
```

Figure 12. The first part codes of Generator.

```
    for i in range(3):
        #each sequence has unique phase shift for high frequency and nonlinear curve (unique nonlinear and non-gaussian)
        phase_shift_high = np.random.uniform(0, 2*np.pi, size=seq_len)
        phase_shift_nonlinear = np.random.uniform(0, 2*np.pi, size=seq_len)
        nonlinear_curve = nonlinear_strength * ((t + phase_shift_nonlinear) % (2*np.pi))**2
        high_freq = 0.5 * np.sin(5 * t + phase_shift_high)

        basic_signal = shared_drift + shared_low_freq + high_freq + nonlinear_curve

        #different noises inclues gaussian and non-gaussian noises
        gaussian_noise = gaussian_std * np.random.randn(seq_len)
        spikes = np.random.choice([0, 1], size=seq_len, p=[1 - spike_probability, spike_probability])
        spike_amplitude = (np.random.pareto(pareto_alpha, size=seq_len) + 1) * pareto_scale
        spike_sign = np.random.choice([-1, 1], size=seq_len, p=[skew_neg_prob, 1 - skew_neg_prob])
        non_gaussian_noise = spikes * spike_amplitude * spike_sign
        real_signal = basic_signal + gaussian_noise + non_gaussian_noise
        basics.append(basic_signal)
        reals.append(real_signal)

    return t, basics[0], reals[0], basics[1], reals[1], basics[2], reals[2]

#main
if __name__ == "__main__":
    t1, basic1, real1, basic2, real2, basic3, real3 = generate_multi_tsdatasets()
    np.savez('simulated_multi_data.npz',
             t1=t1, basic1=basic1, real1=real1,
             t2=t1, basic2=basic2, real2=real2,
             t3=t1, basic3=basic3, real3=real3)

    for i, (b, r) in enumerate(zip([basic1, basic2, basic3], [real1, real2, real3])):
        plt.figure(figsize=(15,5))
        plt.plot(t1[:3000], b[:3000], label=f'Basic Signal {i+1}', linewidth=2)
        plt.plot(t1[:3000], r[:3000], label=f'Real Signal {i+1} with Noise', alpha=0.7)
        plt.xlabel('Time Step')
        plt.ylabel('Signal Value')
        plt.title(f'First 3000 steps of Sequence {i+1}')
        plt.legend()
        plt.grid(True)
        plt.tight_layout()
        plt.savefig(f'sequence{i+1}_first3000.png')
        plt.show()
```

Figure 13. The second part codes of Generator.

```python
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import time

#load the data
data = np.load('simulated_multi_data.npz')

t1 = data['t1']
basic1 = data['basic1']
real1 = data['real1']

t2 = data['t2']
basic2 = data['basic2']
real2 = data['real2']

t3 = data['t3']
basic3 = data['basic3']
real3 = data['real3']

seq_len = len(t1)


#(seq_len, 3)
signals_real = np.stack([real1, real2, real3], axis=1)
signals_basic = np.stack([basic1, basic2, basic3], axis=1)


#standardization
scaler = StandardScaler()
signals_real_scaled = scaler.fit_transform(signals_real)
signals_basic_scaled = scaler.transform(signals_basic)


#convert to torch tensors, shape like(1, seq_len, 3)
real_tensor = torch.tensor(signals_real_scaled, dtype=torch.float32).unsqueeze(0)
basic_tensor = torch.tensor(signals_basic_scaled, dtype=torch.float32).unsqueeze(0)
```

Figure 14. The first part codes of LLM.

```python
#split train/test
train_len = int(0.8 * seq_len)
test_len = seq_len - train_len

real_tensor_train = real_tensor[:, :train_len, :] #(1, train_len, 3)
basic_tensor_train = basic_tensor[:, :train_len, :]
real_tensor_test = real_tensor[:, train_len:, :]
basic_tensor_test = basic_tensor[:, train_len:, :]


#transformer
class TransformerDenoise(nn.Module):
    def __init__(self, d_model=64, nhead=4, num_layers=2):
        super().__init__()
        self.input_proj = nn.Linear(3, d_model)
        encoder_layer = nn.TransformerEncoderLayer(d_model=d_model, nhead=nhead, batch_first=True)
        self.encoder = nn.TransformerEncoder(encoder_layer, num_layers=num_layers)
        self.output_proj = nn.Linear(d_model, 3)
        self.confidence_head = nn.Sequential(nn.Linear(d_model, d_model//2), nn.ReLU(), nn.Linear(d_model//2, 3))

    def forward(self, x):
        x = self.input_proj(x) #(batch, seq_len, d_model)
        enc_out = self.encoder(x)
        out = self.output_proj(enc_out) #(batch, seq_len, 3)
        conf = torch.sigmoid(self.confidence_head(enc_out)) #(batch, seq_len, 3)
        return out, conf

model = TransformerDenoise()


#loss functions
def frequency_domain_loss(pred, target):
    pred_fft = torch.fft.fft(pred, dim=1)
    target_fft = torch.fft.fft(target, dim=1)
    return nn.MSELoss()(torch.abs(pred_fft), torch.abs(target_fft))

def total_variation_loss(pred):
    return torch.mean(torch.abs(pred[:, 1:, :] - pred[:, :-1, :]))

def huber_loss(pred, target):
    return nn.SmoothL1Loss()(pred, target)
```

Figure 15. The second part codes of LLM.

```python
def quantile_loss(pred, target, q=0.5):
    return torch.mean(torch.max(q * diff, (q-1) * diff))

def directional_mse_single(pred, target, direction_weight=1.0):
    cos_sim = torch.nn.functional.cosine_similarity(pred.flatten(1), target.flatten(1), dim=1) #(batch,)
    penalty_factor = 1 + direction_weight * (1 - cos_sim).unsqueeze(-1).unsqueeze(-1) #(batch,1,1)
    return torch.mean(((pred - target) ** 2) * penalty_factor)

def combined_loss(pred, target, alpha=0.1, beta=0.01, gamma=0.05, delta=0.1):
    loss_seq1 = directional_mse_single(pred[:,:,0].unsqueeze(-1), target[:,:,0].unsqueeze(-1))
    loss_seq2 = directional_mse_single(pred[:,:,1].unsqueeze(-1), target[:,:,1].unsqueeze(-1))
    loss_seq3 = directional_mse_single(pred[:,:,2].unsqueeze(-1), target[:,:,2].unsqueeze(-1))
    freq = frequency_domain_loss(pred, target)
    tv = total_variation_loss(pred)
    huber = huber_loss(pred, target)
    quantile = quantile_loss(pred, target)
    return loss_seq1 + loss_seq2 + loss_seq3 + alpha*freq + beta*tv + gamma*huber + delta*quantile


#optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)


#training stage
window_size = 512
stride = 256
epochs = 100
train_start = time.time()

for epoch in range(epochs):
    epoch_loss = 0
    for i in range(0, train_len - window_size + 1, stride):
        input_win = real_tensor_train[:, i:i+window_size, :] #(1, window, 3)
        target_win = basic_tensor_train[:, i:i+window_size, :]
        optimizer.zero_grad()
        output_win, conf_win = model(input_win)
        loss = combined_loss(output_win, target_win)
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()
    if epoch % 5 == 0:
        print(f"Epoch {epoch}: Loss={epoch_loss:.4f}")
```

Figure 16. The third part codes of LLM.

```
train_end = time.time()
print(f"Complete training stage in {train_end - train_start:.2f} seconds")


#inference stage
inference_start = time.time()
denoised_result = torch.zeros_like(real_tensor_test)
conf_sum = torch.zeros_like(real_tensor_test)

for i in range(0, test_len, stride):
    if i + window_size <= test_len:
        input_win = real_tensor_test[:, i:i+window_size, :]
        valid_len = window_size
    else:
        pad_len = i + window_size - test_len
        input_win = torch.zeros((1, window_size, 3))
        input_win[:, :test_len-i, :] = real_tensor_test[:, i:test_len, :]
        valid_len = test_len - i

    output_win, conf_win = model(input_win)

    denoised_result[:, i:i+valid_len, :] += output_win[:, :valid_len, :] * conf_win[:, :valid_len, :]
    conf_sum[:, i:i+valid_len, :] += conf_win[:, :valid_len, :]

conf_sum[conf_sum == 0] = 1
agg_output = denoised_result / conf_sum

inference_end = time.time()
print(f"Inference completed in {inference_end - inference_start:.2f} seconds")

#save the output
agg_output_np = agg_output.squeeze(0).detach().numpy() #(seq_len_test, 3)
np.savez('llm_multi_output.npz', llm=agg_output_np)
torch.save(model.state_dict(), 'llm_multi_model.pt')

#plot
for idx in range(3):
    true_signal = signals_real_scaled[train_len:, idx] * scaler.scale_[idx] + scaler.mean_[idx]
    basic_signal = signals_basic_scaled[train_len:, idx] * scaler.scale_[idx] + scaler.mean_[idx]
    denoised_signal = agg_output_np[:, idx] * scaler.scale_[idx] + scaler.mean_[idx]

    plt.figure(figsize=(15,5))
```

Figure 17. The fourth part codes of LLM.

```
    plt.plot(t1[train_len:], true_signal, label='Real Signal')
    plt.plot(t1[train_len:], basic_signal, label='Basic Signal')
    plt.plot(t1[train_len:], denoised_signal, label='Denoised Signal')
    plt.xlabel('Time Step')
    plt.ylabel('Signal Value')
    plt.title(f'LLM Filtered Sequence {idx+1} Result')
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(f'sequence{idx+1}_denoised.png')
    plt.show()
```

Figure 18. The last part codes of LLM.

```python
import numpy as np
import matplotlib.pyplot as plt
import time

def ekf_filter(z, nonlinear_strength=0.4, Q=0.01, R=0.04, x_init=0, P_init=1):
    """
    EKF for 1D time series (signal smoothing).

    z: observed noises
    nonlinear_strength: assumption of the nonlinear strength in obeserved system
    Q: process noise variance
    R: observation noise variance
    x_init: initial state estimate
    P_init: initial covariance estimate
    """
    N = len(z)
    x_est = np.zeros(N)
    P = P_init
    x = x_init

    for t in range(N):
        #predict
        x_mod = x % (2 * np.pi)
        x_pred = nonlinear_strength * (x_mod ** 2) #f(x)
        F = 2 * nonlinear_strength * x_mod #df/dx

        P_pred = F * P * F + Q

        #update
        H = 1 #observation function derivative (since z = x)
        K = P_pred * H / (H * P_pred * H + R)
        x = x_pred + K * (z[t] - x_pred) #h(x) = x_pred
        P = (1 - K * H) * P_pred

        x_est[t] = x

    return x_est
```

Figure 19. The first part codes of EKF.

```python
def main():
    data = np.load('simulated_multi_data.npz')
    t1 = data['t1']
    real1 = data['real1']
    t2 = data['t2']
    real2 = data['real2']
    t3 = data['t3']
    real3 = data['real3']

    #sequence1
    ekf_start1 = time.time()
    ekf_output1 = ekf_filter(real1, Q=0.01, R=0.04, x_init=real1[0])
    ekf_end1 = time.time()
    print(f"EKF on sequence 1 completed in {ekf_end1 - ekf_start1:.2f} seconds")

    #sequence2
    ekf_start2 = time.time()
    ekf_output2 = ekf_filter(real2, Q=0.01, R=0.04, x_init=real2[0])
    ekf_end2 = time.time()
    print(f"EKF on sequence 2 completed in {ekf_end2 - ekf_start2:.2f} seconds")

    #sequence3
    ekf_start3 = time.time()
    ekf_output3 = ekf_filter(real3, Q=0.01, R=0.04, x_init=real3[0])
    ekf_end3 = time.time()
    print(f"EKF on sequence 3 completed in {ekf_end3 - ekf_start3:.2f} seconds")

    np.savez('ekf_multi_output_new.npz', ekf1=ekf_output1, ekf2=ekf_output2, ekf3=ekf_output3)

if __name__ == "__main__":
    main()
```

Figure 20. The second part codes of EKF.

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from scipy.stats import pearsonr

#use MSE, MAE and SNR to show the capability of processing nonlinear time series signal with non-gaussian noise
def evaluate(true_signal, estimated_signal):
    mse = np.mean((true_signal - estimated_signal) ** 2)
    mae = np.mean(np.abs(true_signal - estimated_signal))
    signal_power = np.mean(true_signal ** 2)
    noise_power = mse
    snr = 10 * np.log10(signal_power / noise_power)
    return mse, mae, snr

#error spectrum diagram shows the energy distribution of the sequence error at different frequencies
def plot_error_spectrum(error, label, seq_idx):
    fft_vals = np.fft.fft(error)
    freqs = np.fft.fftfreq(len(error))
    magnitude = np.abs(fft_vals)

    plt.figure(figsize=(15,5))
    plt.plot(freqs[:len(freqs)//2], magnitude[:len(freqs)//2])
    plt.xlabel('Normalized Frequency')
    plt.ylabel('Magnitude')
    plt.title(f'Sequence {seq_idx+1} Error Spectrum of {label}')
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(f'sequence{seq_idx+1}_error_spectrum_{label}.png')
    plt.show()

#the error histogram shows the distribution pattern of the error, used to observe whether there is a systematic error
def plot_error_histogram(error, label, seq_idx):
    plt.figure(figsize=(8,5))
    plt.hist(error, bins=50, density=True, alpha=0.7)
    plt.xlabel('Error Value')
    plt.ylabel('Probability Density')
    plt.title(f'Sequence {seq_idx+1} Error Histogram of{label}')
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(f'sequence{seq_idx+1}_error_histogram_{label}.png')
    plt.show()
```

Figure 21. The first part codes of Evaluation.

```python
#autocorrelation reflects whether the error is memorable over time
def plot_error_autocorrelation(error, label, seq_idx):
    corr = np.correlate(error, error, mode='full')
    lags = np.arange(-len(error)+1, len(error))
    plt.figure(figsize=(12,5))
    plt.plot(lags, corr)
    plt.xlabel('Lag')
    plt.ylabel('Autocorrelation')
    plt.title(f'Sequence {seq_idx+1} Error Autocorrelation of {label}')
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(f'sequence{seq_idx+1}_error_autocorrelation_{label}.png')
    plt.show()


#main
def main():
    data = np.load('simulated_multi_data.npz')
    ekf_data = np.load('ekf_multi_output_new.npz')
    llm_data = np.load('llm_multi_output.npz')

    basic_signal = np.stack([data['basic1'], data['basic2'], data['basic3']], axis=0) #(3, total_len)
    real_signal = np.stack([data['real1'], data['real2'], data['real3']], axis=0) #(3, total_len)
    ekf_output = np.stack([ekf_data['ekf1'], ekf_data['ekf2'], ekf_data['ekf3']], axis=0) #(3, total_len)
    llm_output = llm_data['llm'].T  #need transpose to (3, test_len) to keep consistent with others

    total_len = basic_signal.shape[1]
    train_len = int(0.8 * total_len)
    test_index = np.arange(train_len, total_len)
    t_test = test_index  #use index as x-axis

    #independent scaler for each sequence
    scalers = []
    real_scaled = np.zeros_like(real_signal)
    basic_scaled = np.zeros_like(basic_signal)
    for i in range(3):
        scaler = StandardScaler()
        real_scaled[i] = scaler.fit_transform(real_signal[i].reshape(-1,1)).flatten()
        basic_scaled[i] = scaler.transform(basic_signal[i].reshape(-1,1)).flatten()
        scalers.append(scaler)
```

Figure 22. The second part codes of Evaluation.

```
basic_test = basic_signal[:, test_index]
ekf_test = ekf_output[:, test_index]

#inverse transform llm output to each sequence
llm_outputs = []
for i in range(3):
    llm_outputs.append(scalers[i].inverse_transform(llm_output[i].reshape(-1,1)).flatten())

#evaluate each sequence
for idx in range(3):
    mse_ekf, mae_ekf, snr_ekf = evaluate(basic_test[idx], ekf_test[idx])
    mse_llm, mae_llm, snr_llm = evaluate(basic_test[idx], llm_outputs[idx])
    print(f"\nSeq {idx+1}: EKF MSE={mse_ekf:.6f}, MAE={mae_ekf:.6f}, SNR={snr_ekf:.2f} dB")
    print(f"\nSeq {idx+1}: LLM MSE={mse_llm:.6f}, MAE={mae_llm:.6f}, SNR={snr_llm:.2f} dB\n")

#overall evaluation
basic_flat = np.concatenate([basic_test[0], basic_test[1], basic_test[2]])
ekf_flat = np.concatenate([ekf_test[0], ekf_test[1], ekf_test[2]])
llm_flat = np.concatenate([llm_outputs[0], llm_outputs[1], llm_outputs[2]])
mse_ekf, mae_ekf, snr_ekf = evaluate(basic_flat, ekf_flat)
mse_llm, mae_llm, snr_llm = evaluate(basic_flat, llm_flat)
print("\n\nOverall:")
print(f"\nEKF MSE={mse_ekf:.6f}, MAE={mae_ekf:.6f}, SNR={snr_ekf:.2f} dB")
print(f"\nLLM MSE={mse_llm:.6f}, MAE={mae_llm:.6f}, SNR={snr_llm:.2f} dB")
plot_error_spectrum(ekf_test[0] - basic_test[0], 'EKF', 0)
plot_error_spectrum(llm_outputs[0] - basic_test[0], 'LLM', 0)

#plot
for idx in range(3):
    plt.figure(figsize=(15,5))
    plt.plot(t_test, basic_test[idx], label='Basic Signal', linewidth=2)
    plt.plot(t_test, ekf_test[idx], label='EKF Output', alpha=0.7)
    plt.plot(t_test, llm_outputs[idx], label='LLM Output', alpha=0.7)
    plt.xlabel('Time Step')
    plt.ylabel('Signal Value')
    plt.title(f'Sequence{idx+1}: Basic Signal vs EKF vs LLM')
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(f'sequence{idx+1}_evaluation_new.png')
    plt.show()
```

Figure 23. The third part codes of Evaluation.

```
    for idx in range(3):
        error_ekf = ekf_test[idx] - basic_test[idx]
        error_llm = llm_outputs[idx] - basic_test[idx]

        #plot error spectrum
        plot_error_spectrum(error_ekf, 'EKF', idx)
        plot_error_spectrum(error_llm, 'LLM', idx)

        #plot error histogram
        plot_error_histogram(error_ekf, 'EKF', idx)
        plot_error_histogram(error_llm, 'LLM', idx)

        #plot error autocorrelation
        plot_error_autocorrelation(error_ekf, 'EKF', idx)
        plot_error_autocorrelation(error_llm, 'LLM', idx)

    #plot error correlation
    ekf_err_1 = ekf_test[0] - basic_test[0]
    ekf_err_2 = ekf_test[1] - basic_test[1]
    ekf_err_3 = ekf_test[2] - basic_test[2]

    llm_err_1 = llm_outputs[0] - basic_test[0]
    llm_err_2 = llm_outputs[1] - basic_test[1]
    llm_err_3 = llm_outputs[2] - basic_test[2]

    ekf_corr12 = pearsonr(ekf_err_1, ekf_err_2)[0]
    ekf_corr13 = pearsonr(ekf_err_1, ekf_err_3)[0]
    ekf_corr23 = pearsonr(ekf_err_2, ekf_err_3)[0]

    llm_corr12 = pearsonr(llm_err_1, llm_err_2)[0]
    llm_corr13 = pearsonr(llm_err_1, llm_err_3)[0]
    llm_corr23 = pearsonr(llm_err_2, llm_err_3)[0]

    print("\nError Correlations Across Sequences")
    print(f"EKF error correlations: seq1-seq2={ekf_corr12:.3f}, seq1-seq3={ekf_corr13:.3f}, seq2-seq3={ekf_corr23:.3f}")
    print(f"LLM error correlations: seq1-seq2={llm_corr12:.3f}, seq1-seq3={llm_corr13:.3f}, seq2-seq3={llm_corr23:.3f}")


if __name__ == "__main__":
    main()
```

Figure 24. The last part codes of Evaluation.

Appendix 2. Other Results in Experiment

```
Epoch 0: Loss=429.1594
Epoch 5: Loss=39.3558
Epoch 10: Loss=33.2480
Epoch 15: Loss=30.7872
Epoch 20: Loss=28.5221
Epoch 25: Loss=30.9919
Epoch 30: Loss=27.5789
Epoch 35: Loss=27.4916
Epoch 40: Loss=33.8707
Epoch 45: Loss=23.8490
Epoch 50: Loss=20.8730
Epoch 55: Loss=19.7754
Epoch 60: Loss=19.8933
Epoch 65: Loss=22.3863
Epoch 70: Loss=21.3155
Epoch 75: Loss=20.3380
Epoch 80: Loss=19.4850
Epoch 85: Loss=19.2597
Epoch 90: Loss=18.9576
Epoch 95: Loss=18.3111
Complete training stage in 72.47 seconds
Inference completed in 0.46 seconds
```

Figure 25. The process of training and inference in LLM.

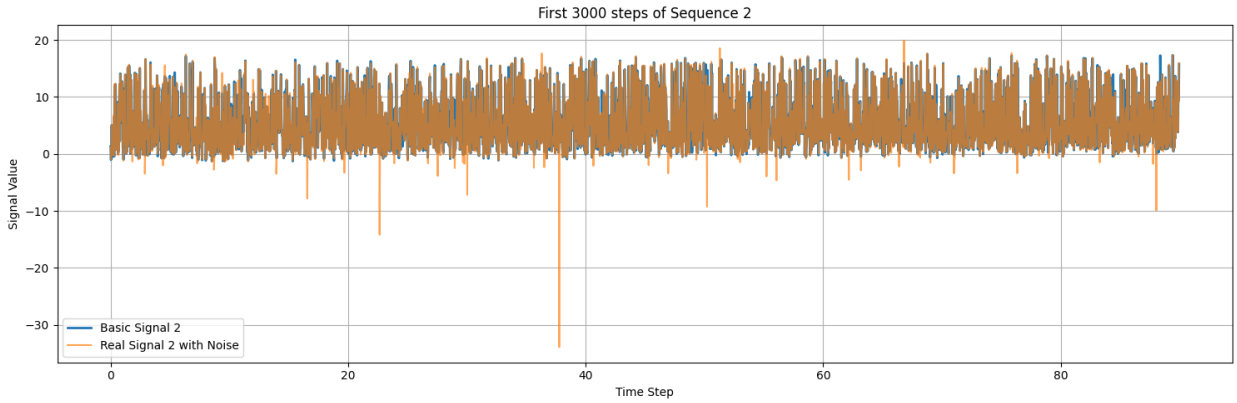

Figure 26. The first 3000 time step signals of the sequence 2.

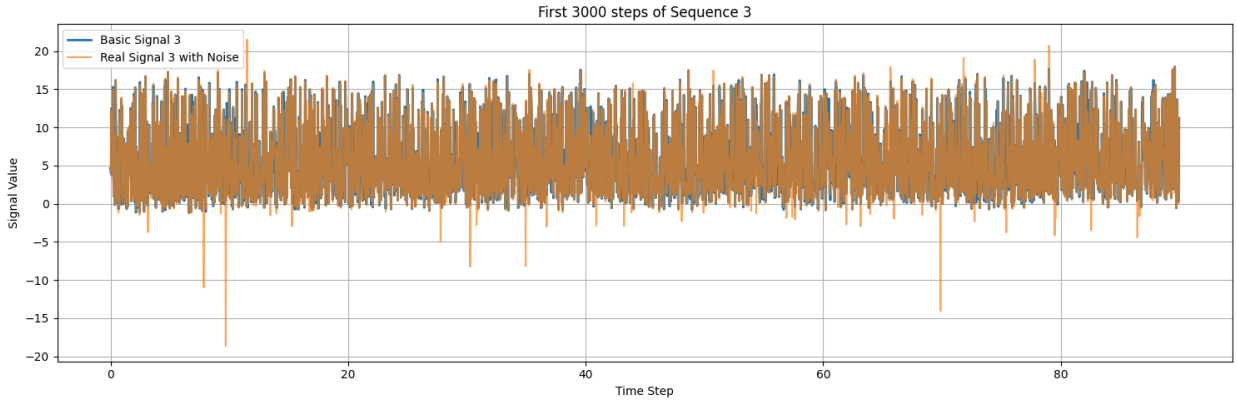

Figure 27. The first 3000 time step signals of the sequence 3.

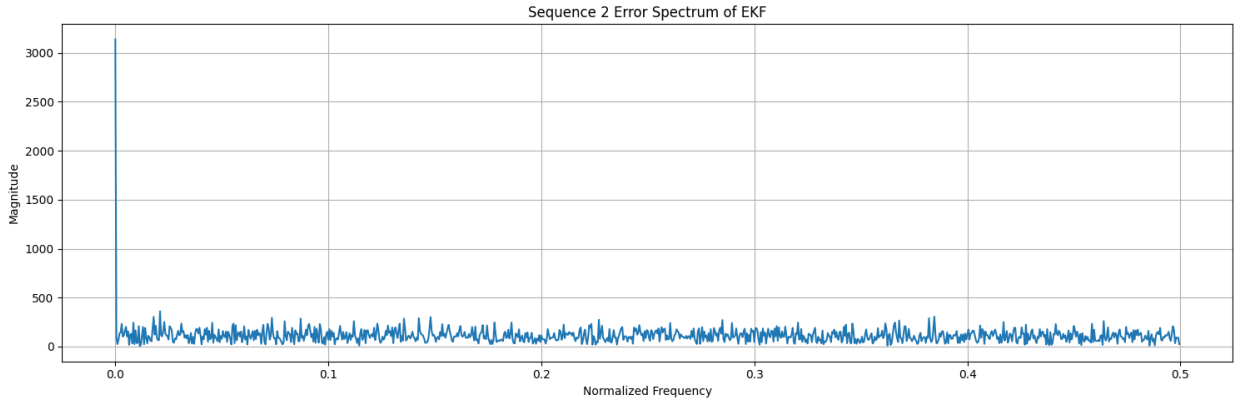

Figure 28. Error spectrum of the sequence 2 generated by EKF.

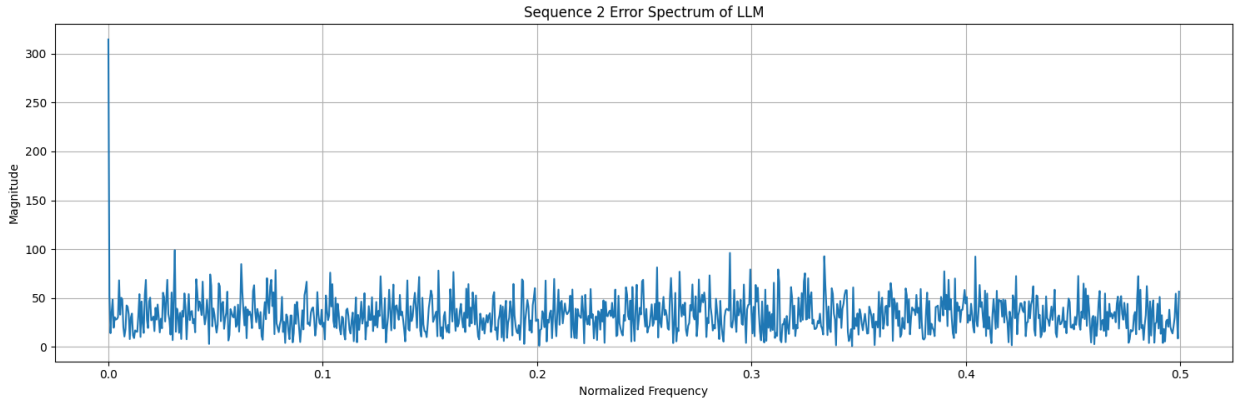

Figure 29. Error spectrum of the sequence 22 generated by LLM.

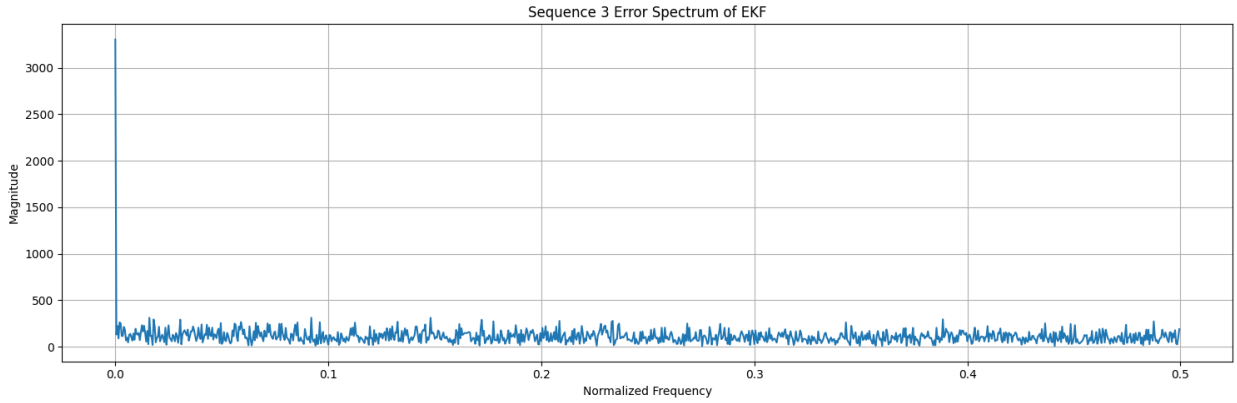

Figure 30. Error spectrum of the sequence 3 generated by EKF.

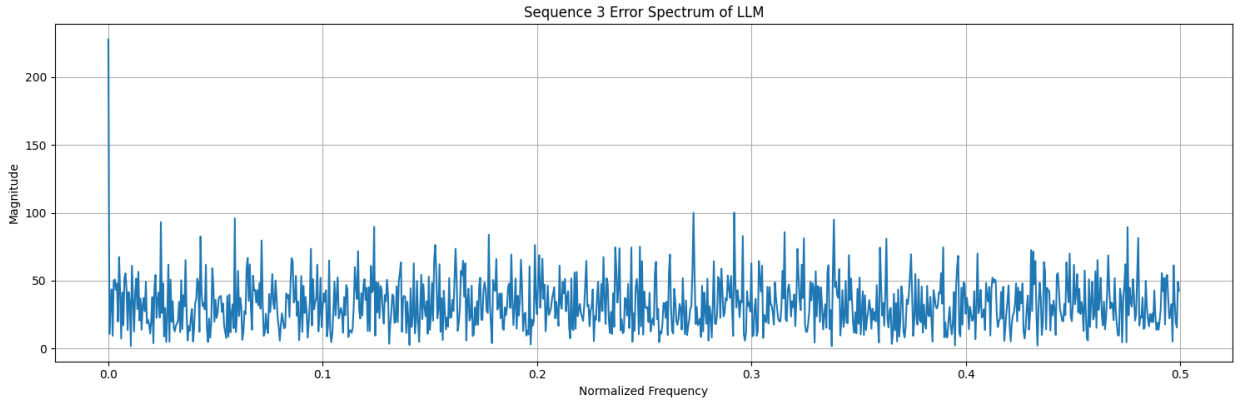

Figure 31. Error spectrum of the sequence 3 generated by LLM.

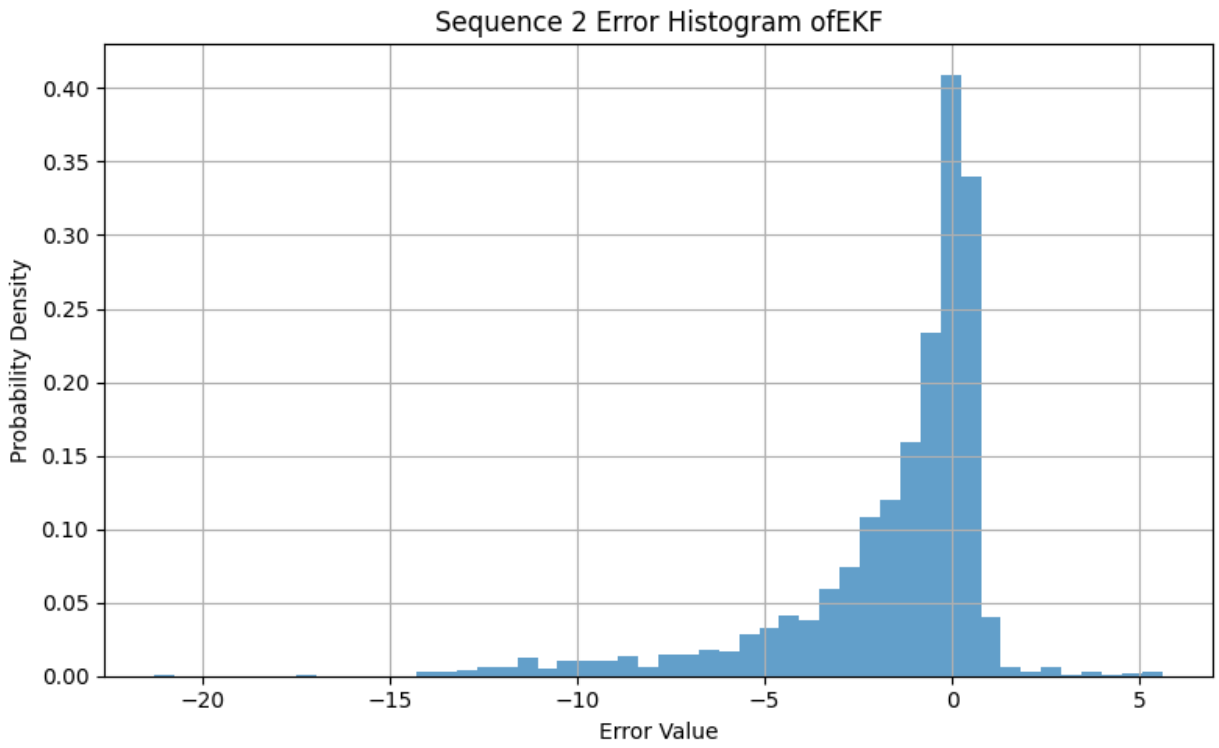

Figure 32. Error histogram of the sequence 2 generated by EKF.

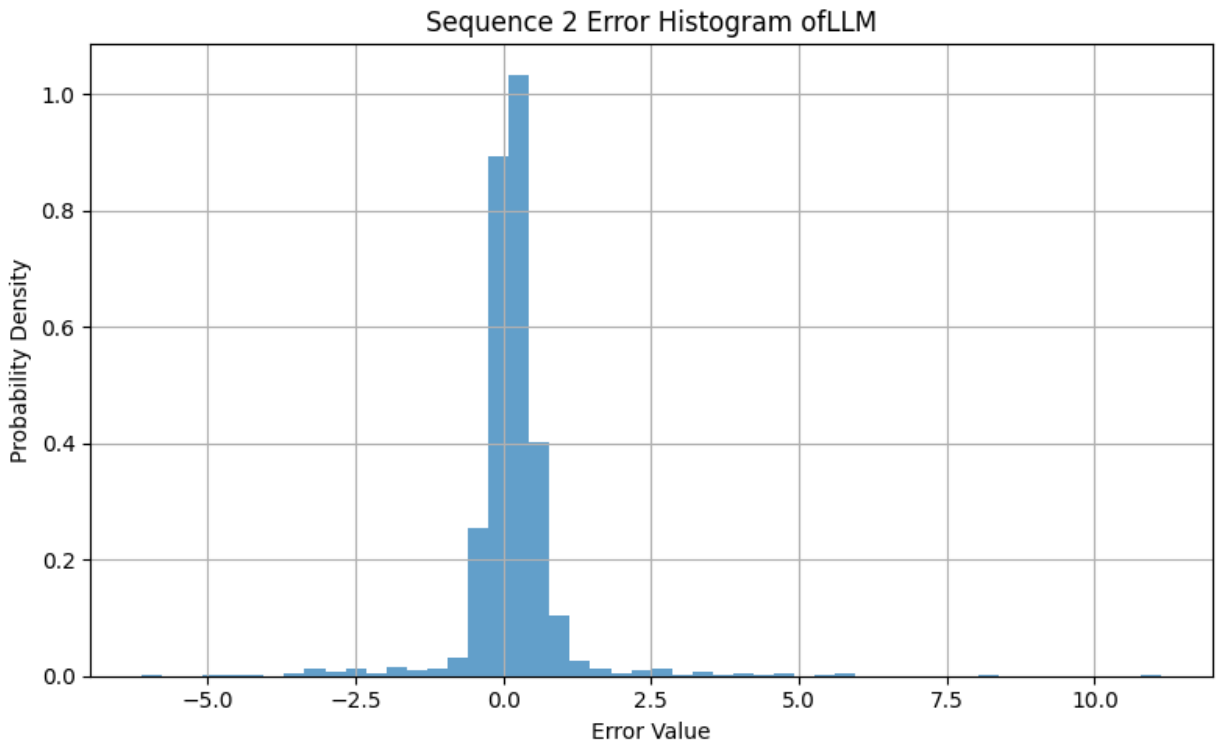

Figure 33. Error histogram of the sequence 2 generated by LLM.

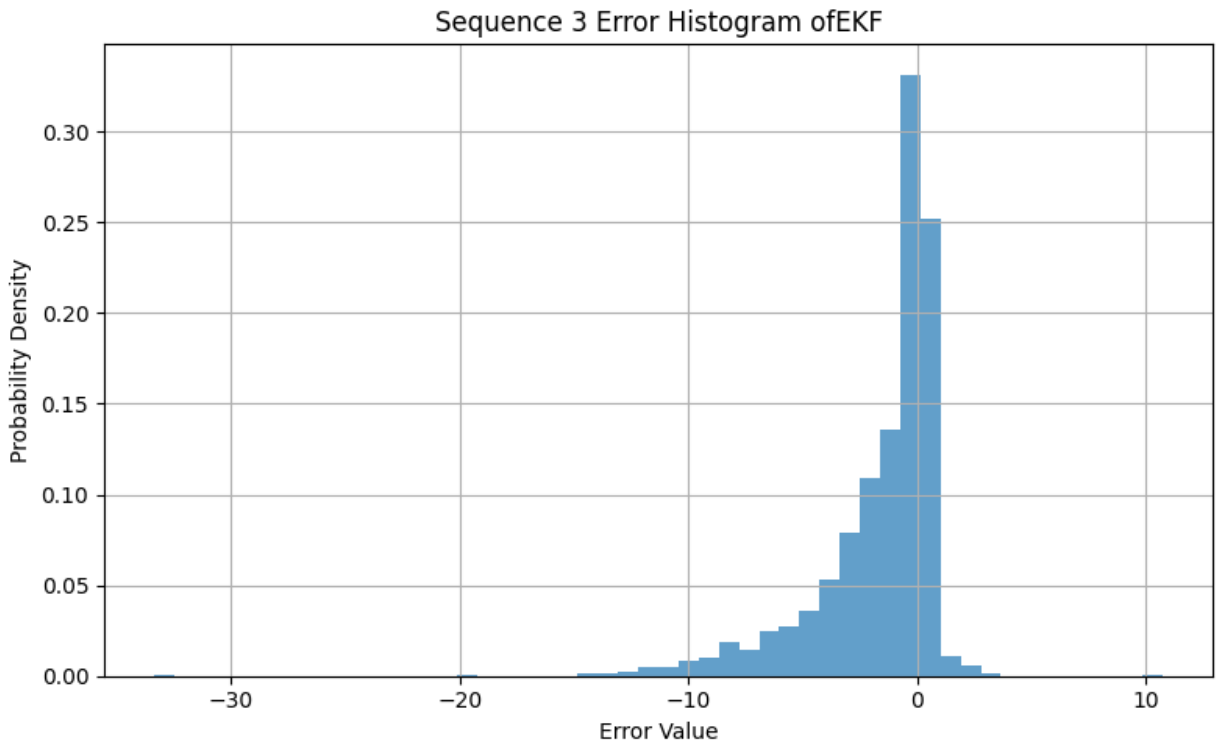

Figure 34. Error histogram of the sequence 3 generated by EKF.

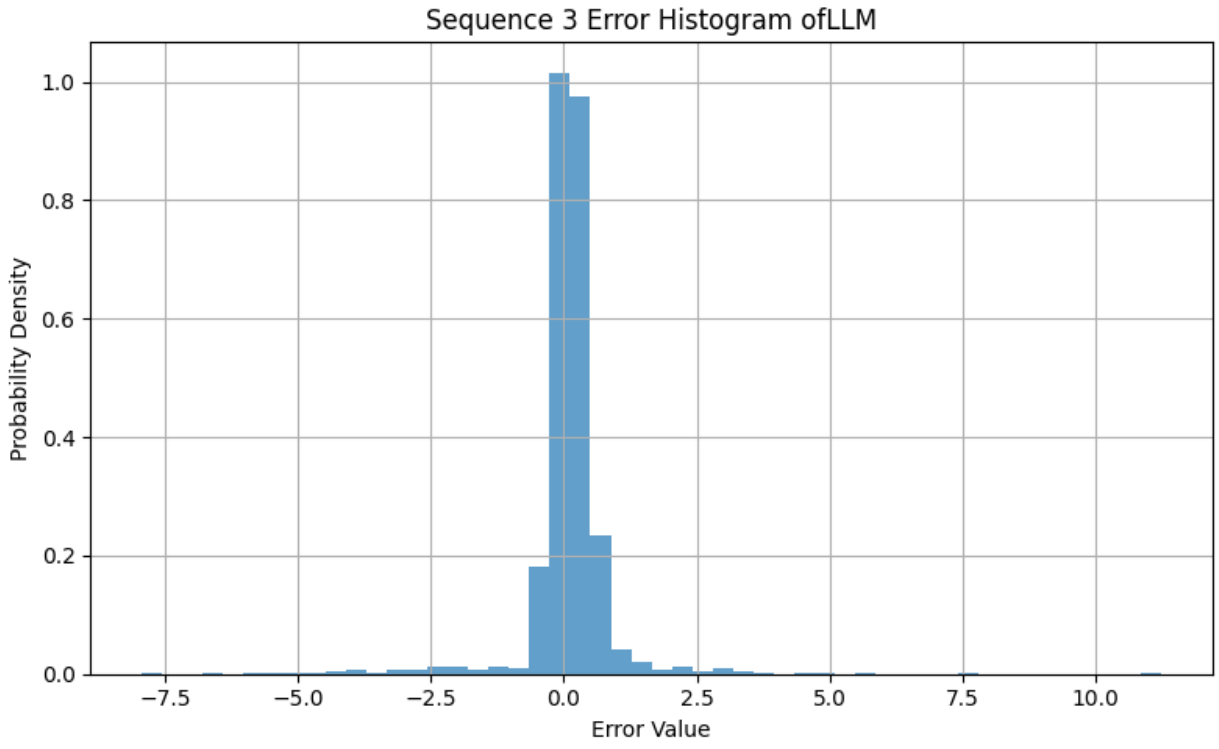

Figure 35. Error histogram of the sequence 3 generated by LLM.

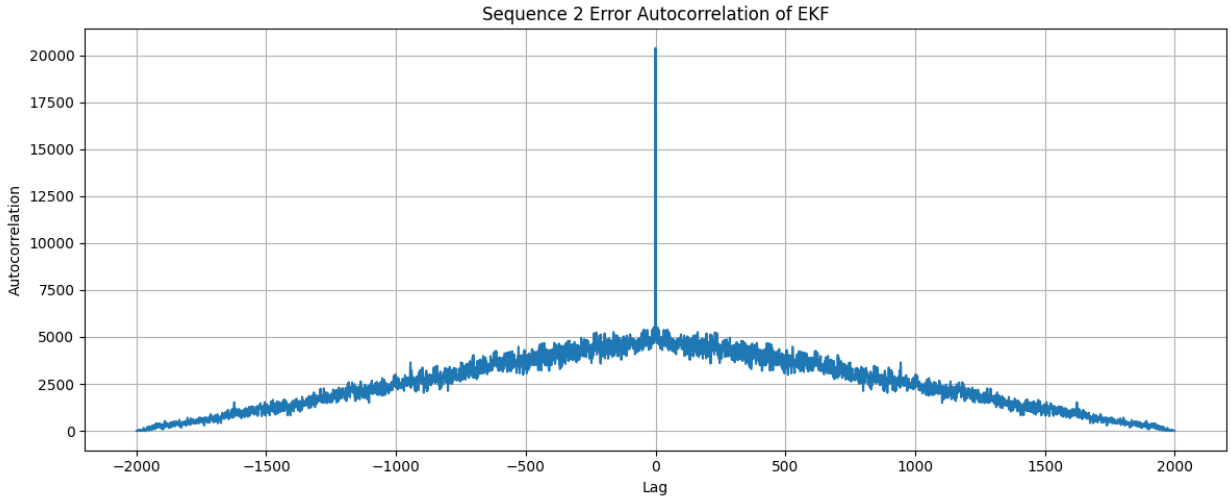

Figure 36. Error autocorrelation of the sequence 2 generated by EKF.

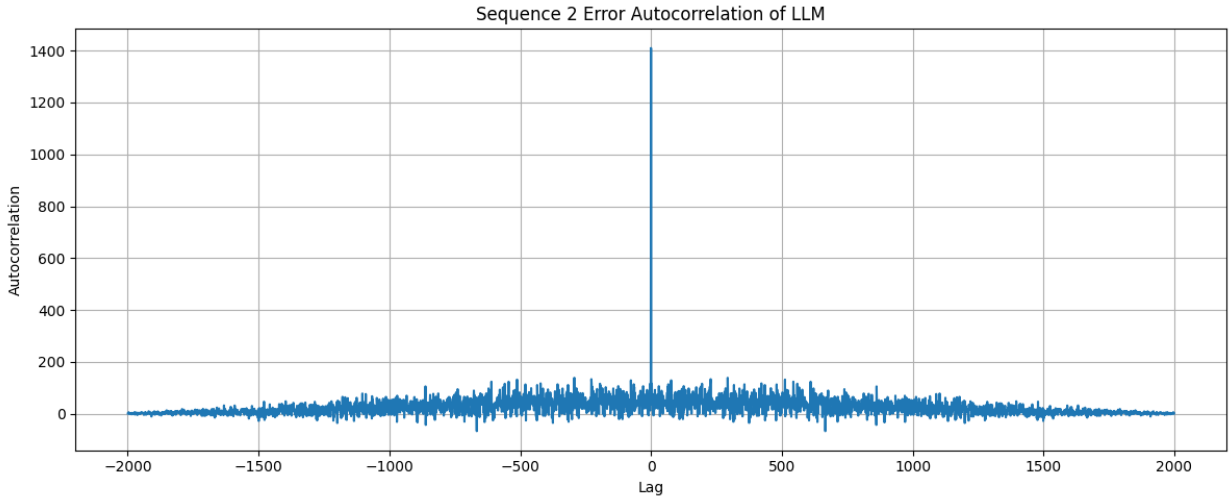

Figure 37. Error autocorrelation of the sequence 2 generated by LLM.

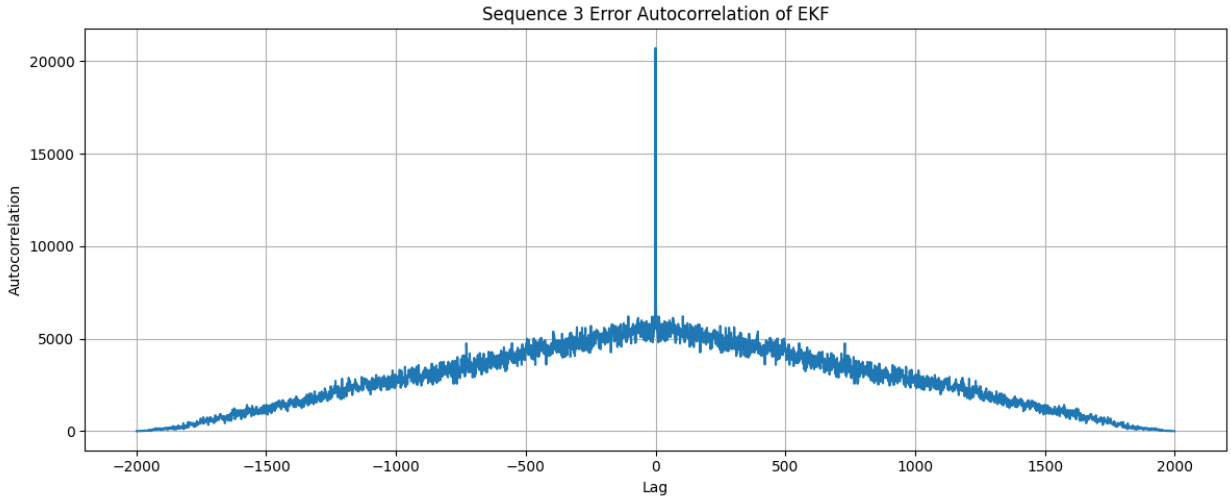

Figure 38. Error autocorrelation of the sequence 3 generated by EKF.

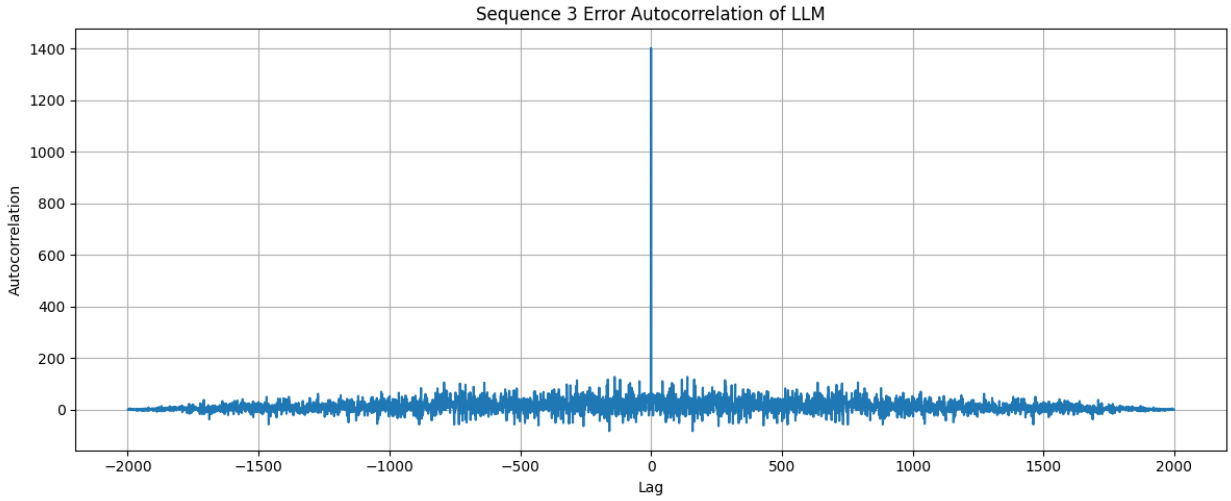

Figure 39. Error autocorrelation of the sequence 3 generated by LLM.