Problem Chosen	2024 ShuWei Cup Summary Sheet	Team Control Number
C		2024110530377

Enhancing Pulsar Timing Precision: A Joint Analysis of Noise Modeling and Atmospheric Refraction Delays

summary

This study aims to address the challenges of noise removal and atmospheric delay correction in pulsar timing, in order to enhance the precision of time signals. The research lays a scientific foundation for future applications of pulsars in precise timekeeping and deep space navigation.

For Problem 1, the task involves modeling the red noise of pulsars. In response, our team developed a red noise generation and parameter fitting model based on the power spectral density (PSD) approach. Initially, the PT-TT time series data were transformed from the time domain to the frequency domain using **FFT**, followed by the calculation of the **PSD**. We then constructed a red noise model based on the PSD and **applied the least squares method** to minimize the error between the model's predictions and actual observations. The fitting process emphasized the low-frequency range, which best captures the characteristics of red noise. **Random simulation methods** were employed to generate red noise, and the results were validated against observational data from two time periods. The results, with R^2 values of **0.9507896654** and **0.9587555241**, met the target accuracy of over 95%.

For Problem 2, The task requires predicting both the short-term and long-term future trends of pulsar timing noise. Our team developed both a **SARIMA model** and an **LSTM model** to address these trends. Based on literature and data observations, pulsar signals exhibit periodic fluctuations, making the SARIMA model suitable for short-term forecasting, while the presence of long-term trends necessitated the use of an LSTM model. To evaluate the performance of these models, we compared the MSE and RMSE metrics of **SARIMA** and **ARIMA**, as well as other models such as **LSTM**, **GRU**, **MLP**, and **Random Forests**. Our comparison showed that the SARIMA model and the LSTM model performed best, with R^2 values of **0.96751486047** and **0.9936738159777331**, respectively. Additionally, **sensitivity analysis** of LSTM hyperparameters confirmed the model's robustness, with R^2 remaining **above 95%** across various settings.

For Problem 3, the task is to model the refractive time delay at radio frequencies above 20 GHz. To address this, our team developed a **high-frequency atmospheric delay correction model**. For ionospheric delay, we used a **dispersion model** based on electron density and signal frequency, while for tropospheric delay, we employed an improved **Saastamoinen model**. Sensitivity analysis was performed by **varying TEC values and frequency ranges** to assess the zenith delay variation. The results indicated that the model maintained good stability and accuracy under different conditions.

For Problem 4, the focus is on precise atmospheric delay measurement at low elevation angles. The team developed a **delay mapping model for small elevation angles**. The ionospheric delay was modeled using the same **dispersion model** from Problem 3, while tropospheric delay was divided into **dry and wet components,** modeled using the **improved Saastamoinen approach**. We optimized the **Herring mapping function** for low elevation angles. Sensitivity analysis was performed by **varying elevation angles and TEC values**. The results showed that at high frequencies, atmospheric delay effects **decreased**, and increasing elevation angles significantly **reduced delay**, indicating that using higher-frequency signals and increasing elevation angles can effectively improve measurement precision and signal stability.

Keywords： Pulsar Timing Noise; Power Spectral Density; SARIMA Model; LSTM Model

Content

1. Introduction

1.1 Background

This study investigates the impact of removing pulsar timing noise and atmospheric delays on the precision of time signals, with the goal of enhancing the practical applications of pulsar time signals in precise navigation and time standard development. Pulsars, as rapidly rotating neutron stars with stable rotation, are often referred to as "cosmic lighthouses" due to their precise timing characteristics, and they hold significant potential for applications in deep space spacecraft navigation and time synchronization. However, during pulsar observations, the presence of timing noise and atmospheric delays can significantly compromise the precision and reliability of the signals. Effectively mitigating or removing these sources of interference is a key challenge in current pulsar timing research. This study aims to optimize the methods for removing pulsar timing noise and atmospheric delays through modeling and predictive analysis, providing both theoretical and technical support for the precise application of pulsar time.

1.2 Work

Based on the above background, the following specific research questions are addressed:

Question one: Develop a functional model to simulate the dynamic behavior of pulsar timing noise, ensuring that the model's goodness-of-fit reaches 95% or higher.

Question two: Using the given pulsar timing noise data, construct a predictive model to analyze its short-term (from a few days to a month) and long-term (from several months to years) trends.

Question three: For radio observation frequencies greater than 20 GHz, construct a model of atmospheric refraction delay, ensuring that the zenith delay does not exceed 7.69 nanoseconds. Refer to Chapter 6.1 of the book Spacetime Reference Systems for a detailed analysis of the formation mechanisms of refraction delays, and optimize the model to reduce delay errors.

Question four: Model the atmospheric delay effects under low-angle observation conditions (10 degrees or smaller) to improve the precision of the pulsar signal's time of arrival (TOA).

2. Problem analysis

2.1 Data analysis

In this section, our team performed a preliminary analysis of the pulsar timing noise and atmospheric delay data using boxplots, aiming to improve the accuracy and stability of the model through data processing. The boxplots illustrate the distributions of MJD (Modified Julian Date) and PT-TT, highlighting the central tendency and outliers in the data. The results are presented in the figure below:

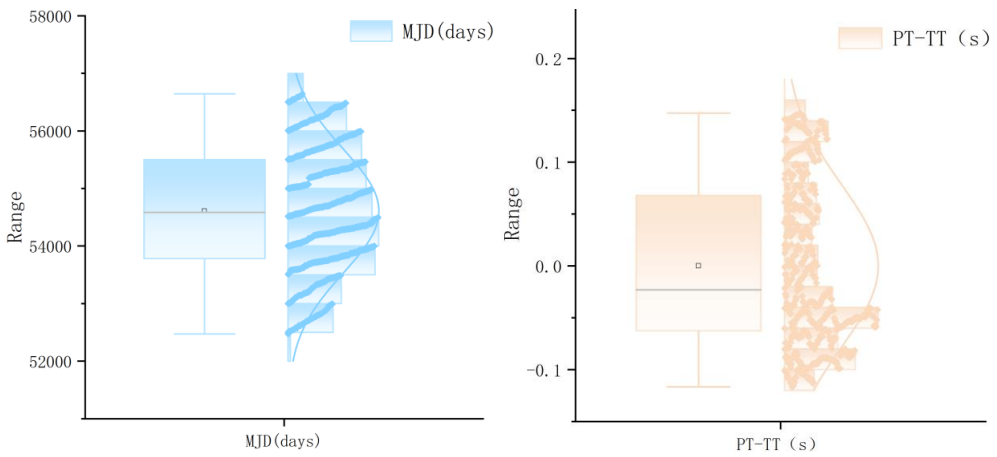

Figure 1 Boxplot

From the boxplot analysis, it is evident that some data points fall outside the whiskers, suggesting the presence of outliers. These outliers could negatively influence statistical analysis and subsequent modeling. To ensure the reliability of the data, our team standardized the data and removed significant outliers. Standardization not only unified the data with different units, improving the efficiency of model calculations, but also ensured a balanced contribution of each feature during the modeling process.

Additionally, to minimize short-term fluctuations that might interfere with data analysis, our team applied a moving average technique to smooth the data, with the results saved in Attachment 2.xlsx. This method effectively reduced short-term fluctuations while more accurately capturing the long-term trend, facilitating the prediction of future trends. The results are shown in the figure below.

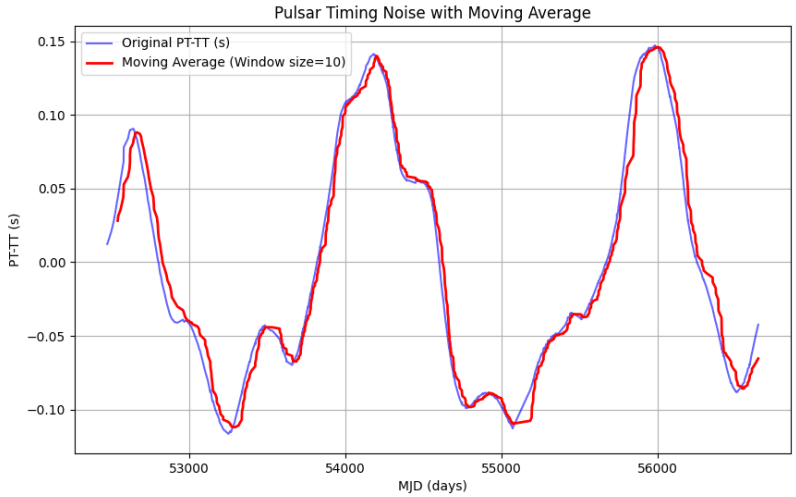

Figure 2 Results of Moving Average Processing

The red curve represents the data processed by the moving average method. Compared to the original data, the fluctuations are significantly reduced, the curve becomes smoother, while retaining key trends and avoiding significant distortion. Through these data processing steps, our team substantially reduced measurement noise and systematic biases, greatly enhancing the robustness and predictive accuracy of the model.

For short-term predictions, the original data from Attachment 1.xlsx were used to retain more short-term dynamics; for long-term predictions, the smoothed data from Attachment 2.xlsx were used to capture long-term trends. These optimizations have

established a solid data foundation for subsequent modeling efforts, ensuring more reliable and scientifically sound results.

2.2 Analysis of question one

Problem one involves constructing a model to simulate the characteristics of pulsar timing noise, with the objective of achieving model fitting accuracy exceeding 95%. This model will then be used to predict future noise trends and support improvements in the pulsar time signal.

First, our team preprocessed the data in Attachment 1.xlsx, extracting the MJD and PT-TT data to ensure uniform sampling. Next, the power spectral density (PSD) was calculated using Fourier transform, revealing the distribution characteristics of the noise in the frequency domain.

Subsequently, a red noise model was constructed based on the PSD, and least squares fitting was applied by minimizing the error between the model's predicted and observed values. Special attention was given to the low-frequency PSD characteristics, as fluctuations in this region most effectively reflect the characteristics of red noise. Finally, using the model results, our team applied stochastic simulations to generate red noise and performed several validation experiments to ensure the model's reliability in simulating pulsar noise.

2.3 Analysis of question two

Problem two aims to forecast pulsar timing noise both in the short term and long term. Pulsar timing noise exhibits periodicity, long-term trends, and random fluctuations, which necessitate the use of multiple modeling techniques to accurately capture these features. To predict short-term fluctuations and long-term trends effectively, our team considers both traditional statistical properties and more complex nonlinear characteristics. Thus, we selected the SARIMA time series model to capture periodicity and trend variations, and adopted the LSTM deep learning model to enhance prediction capability for complex, nonlinear features, leveraging recurrent neural networks (RNNs) to capture long-term dependencies and complex patterns. By combining both statistical and deep learning models, we comprehensively address the noise characteristics and achieve more accurate predictions.

2.4 Analysis of question three

Problem three requires modeling the atmospheric refraction delay for frequencies above 20 GHz, with the constraint that the refraction delay must not exceed 7.69 nanoseconds. The atmospheric refraction delay is mainly caused by the ionosphere and troposphere. The ionospheric delay depends on the radio frequency and total electron content (TEC), while the tropospheric delay is influenced by factors such as air pressure, temperature, humidity, and zenith angle. To reduce model complexity, we model the ionospheric delay using a simple dispersion delay model, while the tropospheric delay is modeled as the basis for calculating the total refraction delay. For the tropospheric refraction delay, we use an improved Saastamoinen model, incorporating air pressure, temperature, and humidity to ensure that the delay stays within the specified range.

2.5 Analysis of question four

Problem four focuses on modeling atmospheric delays at small elevation angles (10 degrees or lower) to improve pulsar arrival time (TOA) accuracy. At these angles, the signal's path through the atmosphere increases, causing a significant rise in refraction-induced delays. Accurate modeling requires accounting for how atmospheric refraction varies with elevation. In the standard refraction delay model, a mapping function converts zenith delay to delay at any elevation. Our team uses the Herring mapping function, but at small angles, it can lead to numerical instability due to large values. To resolve this, we modified the function by adding a correction factor in the denominator, ensuring stable calculations. Additionally, by separating dry delay (ionospheric) from wet delay (tropospheric), we developed distinct atmospheric delay models for both layers. The following flowchart summarizes this process:

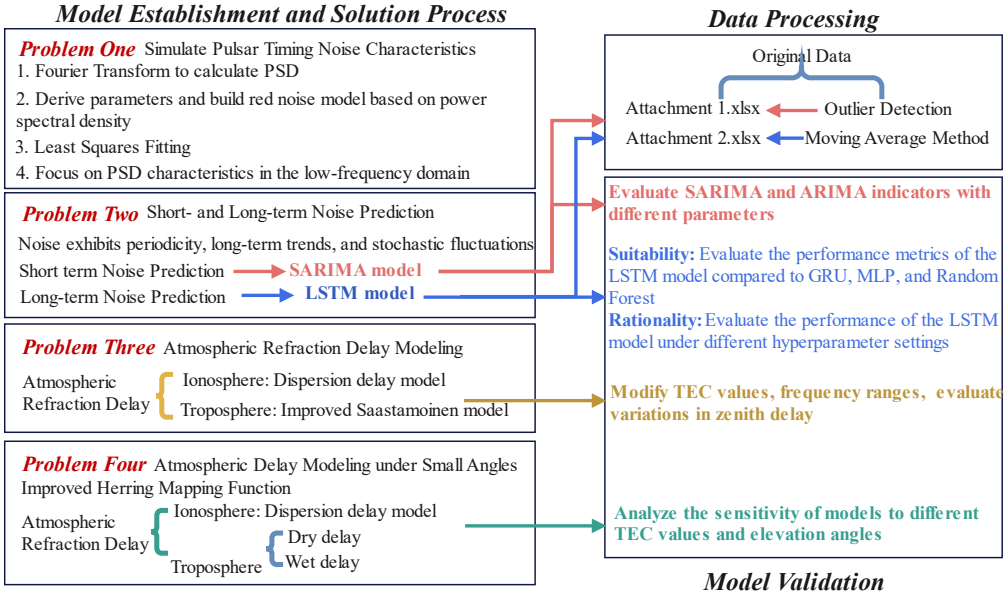

Figure 3 Overall flow chart

3. Symbol and Assumptions

3. 1 Symbol Description

Table 1 Symbol specification

Symbol	Explanation	Unit
RMS1	RMS values for MJD 52473 to 56081	ns
RMS2	RMS values for MJD 52473 to 56646	ns
TEC	Total electron content	TECU
f	Frequency of signal	Hz
P	Atmospheric pressure	hPa
T	Atmospheric temperature	K
ζ	Weather delay	ns

3.2 Fundamental assumptions

To simplify the model, our team makes the following assumptions:

1. The atmospheric layers are assumed to be uniform, with the refractive index changing continuously with altitude.

2. The delay of high-frequency radio signals is primarily caused by tropospheric effects, and ionospheric delay is not considered.
3. Atmospheric pressure, temperature, and humidity are assumed to follow the standard atmospheric model.
4. The radio signal propagation path in the atmosphere is assumed to be smooth and continuous, without considering turbulence.
5. The atmospheric refractive index is modeled as a linear function of temperature, humidity, and pressure.
6. At high frequencies (above 20 GHz), the effect of the ionosphere is negligible.

4. Model

4.1 Problem One: Model Development and Solution

The focus of this study is on the characteristics of pulsar timing noise. The time-domain data is analyzed through Fourier transform[1][2] to obtain its Power Spectral Density (PSD)[3][4], and a red noise model is developed to simulate the noise characteristics of pulsars and estimate the corresponding model parameters. The primary goal of this model is to effectively simulate the characteristics of pulsar timing noise, aimed at supporting future noise prediction and data processing. The specific process for model development is outlined as follows:

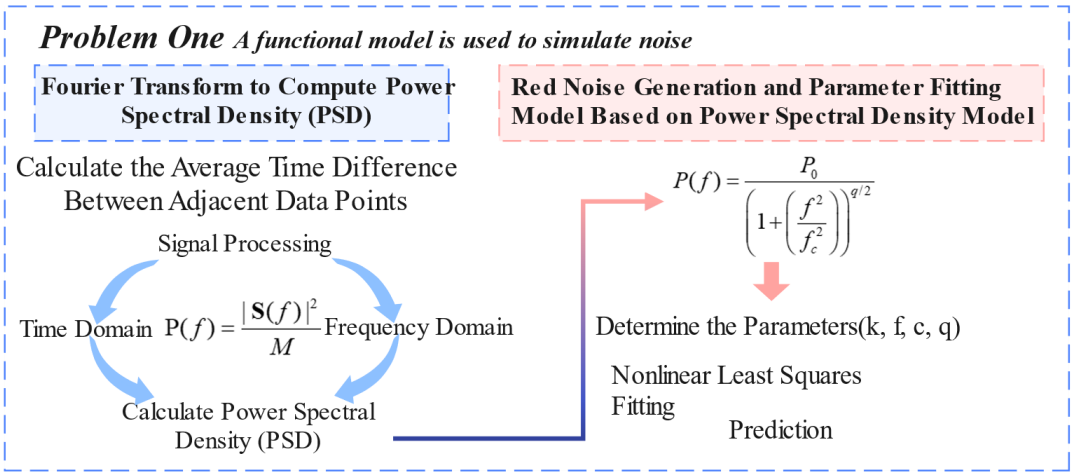

Figure 4 Problem One Flow chart

4.1.1 Fourier Transform to Calculate Power Spectral Density (PSD)

First, the data from Attachment 1 is loaded, and the valid MJD (Modified Julian Date) and PT-TT (Pulsar Time of Arrival) columns are extracted for analysis. Since Fourier transform requires uniform sampling of the signal, our team approximates the sampling interval Δt as the average time difference between adjacent data points, ensuring the basic requirements for the Fourier transform are met.

$$\Delta t = \frac{1}{M-1}\sum_{i=2}^{M}(t_i - t_{i-1}) \tag{1}$$

Next, the Fast Fourier Transform (FFT) is applied to the PT-TT signal, converting the time-domain data into its frequency-domain representation. Our team uses the fft function to perform the Fourier transform, resulting in an array of complex numbers, where each complex number represents the amplitude and phase of a specific frequency component. The fftfreq function is then used to obtain the frequency array, allowing for the analysis of the Fourier transform results.

To compute the Power Spectral Density (PSD), our team takes the square of the magnitude of the Fourier transform results and normalizes the values. This process yields an energy density distribution corresponding to each frequency, with these frequencies matching those obtained from the Fourier transform. The formula is as follows:

$$\mathrm{P}(f)=\frac{|\mathbf{S}(f)|^2}{M} \tag{2}$$

To compute the Power Spectral Density (PSD), our team takes the square of the magnitude of the Fourier transform results and normalizes the values. This process yields an energy density distribution corresponding to each frequency, with these frequencies matching those obtained from the Fourier transform. The formula is as follows:

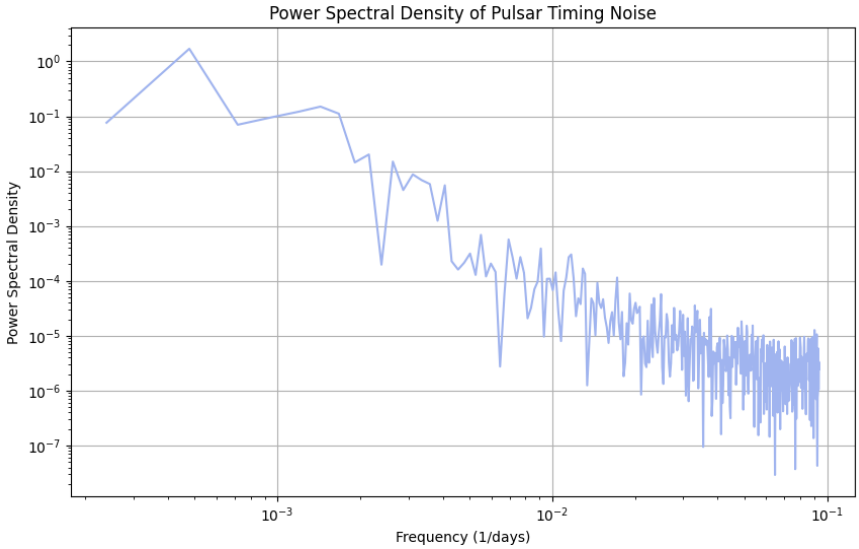

Figure 5 Power Spectral Density of Pulsar Timing Noise

The figure above illustrates the Power Spectral Density (PSD) of pulsar timing noise. Fourier transform is used to obtain the frequency-domain distribution of the signal, and by plotting the PSD, the energy distribution of noise across different frequencies can be observed.

From the PSD plot, it can be observed that in the low-frequency region (10^{-3} to 10^{-2}), the PSD value fluctuates considerably, displaying characteristic behavior of red noise. Therefore, our research team concentrates on fitting this frequency range for a more accurate description of red noise behavior.

4.1.2 Power Spectral Density-Based Red Noise Generation and Parameter Fitting Model

Upon obtaining the Power Spectral Density (PSD), the next step involves generating red noise using the PSD model and fitting the model parameters to identify those that best describe the observed noise characteristics. First, the form of the power spectral model is given by:

$$P(f)=\frac{P_0}{\left(1+\left(\frac{f^2}{f_c^2}\right)\right)^{q/2}} \tag{3}$$

$$P_0=k\cdot \mathrm{RMS}_1 \tag{4}$$

Here, $P(f)$ represents the Power Spectral Density at frequency f, k is the noise intensity, fc is the corner frequency, and q is the spectral index. These parameters are determined by fitting the experimental data.

To determine the optimal parameters (k, fc, q), our team uses the method of minimizing the squared error between the observed values and the model predictions. The objective function for optimization is defined as:

$$\left(\sum_{i=1}^{L}\left(\mathrm{PSD}_{\mathrm{positive}}(f_i)-P(f_i;k,f_c,q)\right)^2\right)_{\min} \tag{5}$$

For this process, we select initial guesses as: $k_0=1\times10^{-6}$,$f_{c,0}=1\times10^{-3}$,$q_0=2$. Curve fitting methods are used to optimize the parameters through nonlinear least squares, yielding the following fitted parameters:

k=0.00001386074974684380 5,fc =0.11477713746580295,q=2.6087867011359687

Based on the fitted model parameters, we further use random number generation techniques to simulate the red noise characteristics and generate the corresponding noise curve. The results are shown in the figure below:

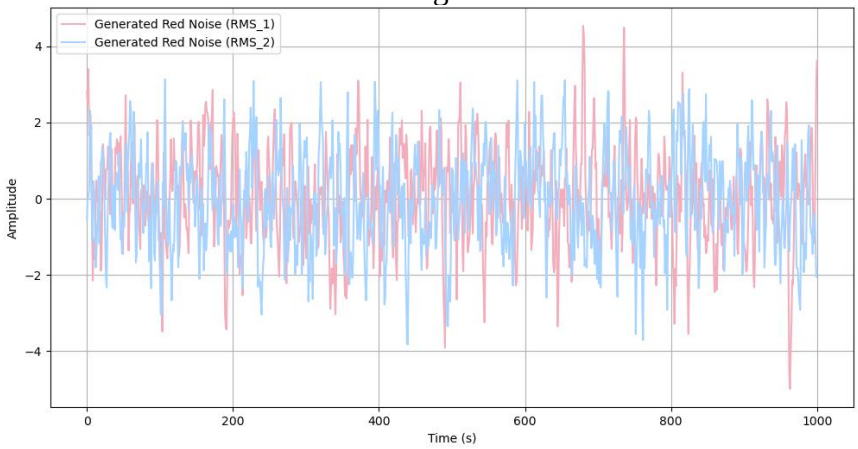

Figure 6 Generated Red Noise Time Series

The simulation results indicate that the generated noise demonstrates strong agreement with the actual observed data within the low-frequency range. Finally, the simulated results were validated against observational data from two time periods: MJD (52473–56081) and MJD (52473–56646). The following results were achieved:

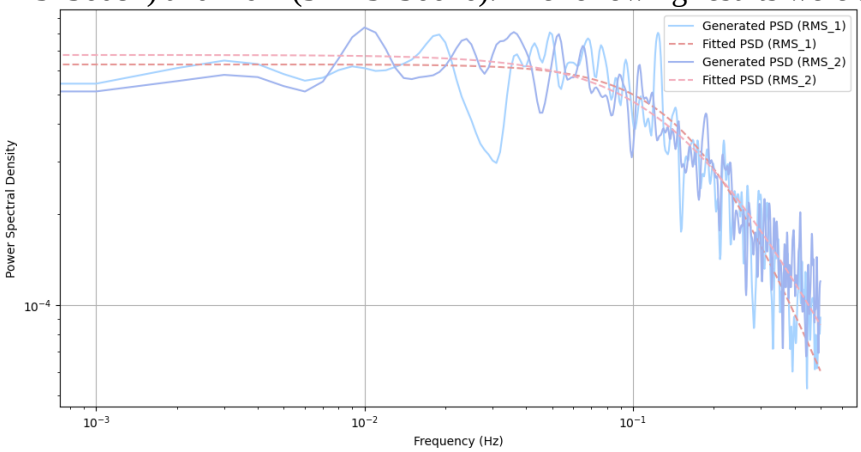

Figure 7 Fit the resulting graph

By calculating the goodness of fit, we obtained R_1^2=0.9507896654,R_2^2 =0.9587555241. These results show that the model achieves a fitting accuracy of over 95%, meeting the expected goal. This demonstrates that the model effectively simulates red noise characteristics and achieves high consistency with actual observed

data during the validation process. Through this model development, this paper provides a solid theoretical and technical foundation for improving pulsar timing accuracy and noise suppression. The parameter fitting and validation confirm the model's effectiveness in simulating pulsar noise characteristics, offering reliable tools for future noise removal and improvements in time signal accuracy.

4.2 Problem Two: Model Development and Solution

The main goal of Problem 2 is to perform short-term prediction and validation of the future trends of pulsar timing noise to support future time signal processing. Our team used two different modeling approaches: the SARIMA model based on statistical analysis and the LSTM model based on deep learning. The details are as follows:

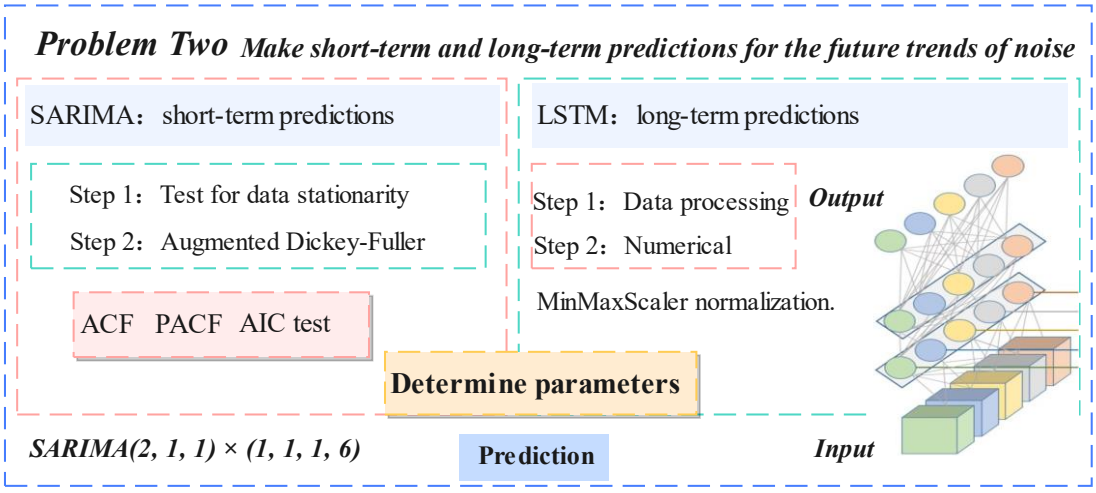

Figure 8 Flowchart for Problem Two

4.2.1 SARIMA Model Development and Solution

In Problem 2, our goal was to predict both the short-term and long-term trends of pulsar timing noise. The data characteristics were first examined, as shown in the figure below.

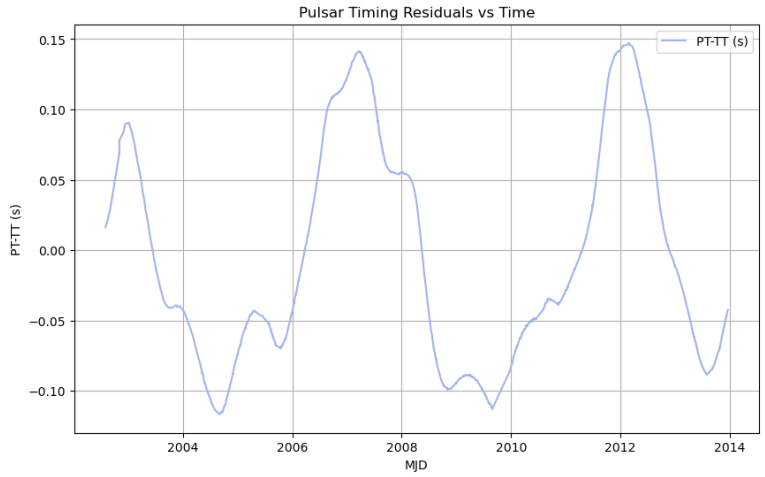

Figure 9 Pulsar Timing Residuals

Preliminary observations from the title and Figure 2 suggest a clear periodicity, making the data suitable for SARIMA[5] time series analysis.

Step 1: Data Stationarity Test

To verify the suitability of the data for time series analysis, we first performed the ADF (Augmented Dickey-Fuller) test. The test results showed a p-value of 0.0987,

greater than 0.05, indicating that the original data is non-stationary. Therefore, we differenced the data until it became stationary. After differencing, the data passed the ADF test and met the stationarity requirement. The results are shown below.

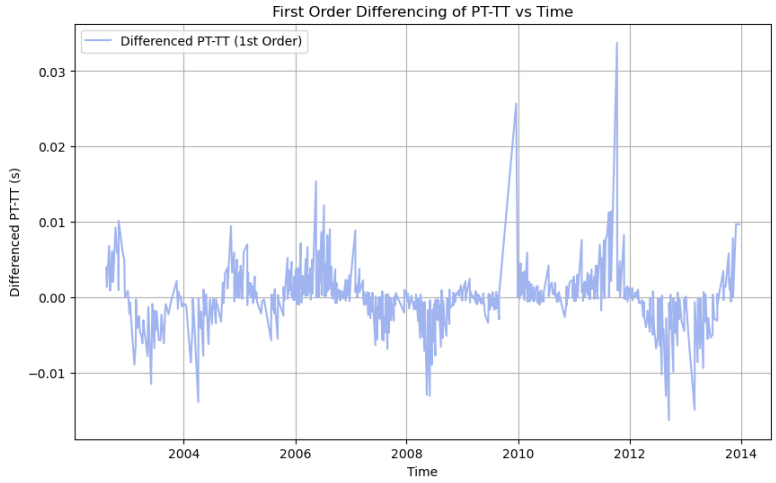

Figure 10 First Order Differencing of PT-TT

Step 2: Selecting SARIMA Model Parameters

After confirming the data stationarity, we next analyzed the autocorrelation function (ACF) and partial autocorrelation function (PACF) to examine the data's autocorrelation and partial autocorrelation. The results are shown below.

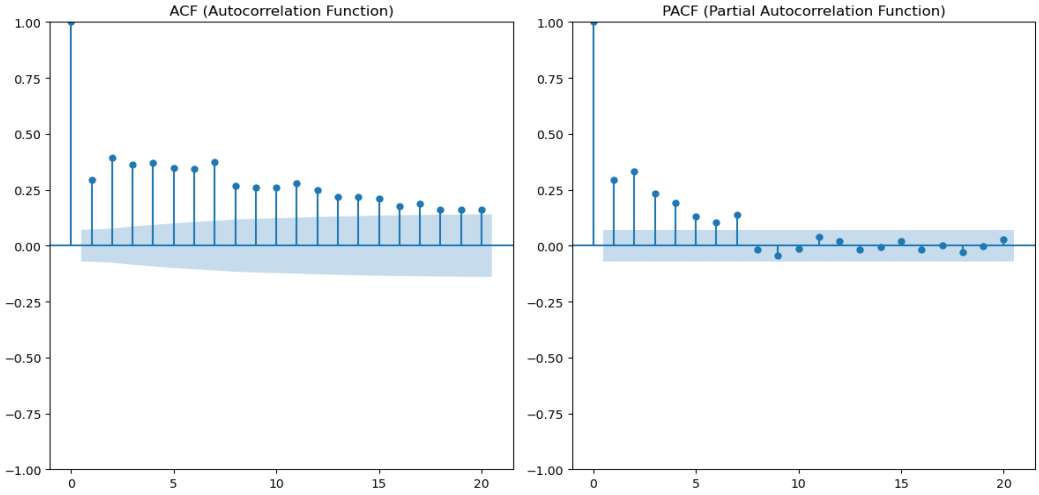

Figure 11 ACF、PACF

By observing the ACF plot, we found that after lag 1, the ACF exhibits a slow decay, suggesting the presence of a moving average (MA) component. Thus, we considered selecting q=1 or 2. By observing the PACF plot, we found significant deviations from zero at lags 1 and 2, followed by rapid decay to near zero, indicating the presence of an autoregressive (AR) component. Therefore, we considered selecting p=1 or 2.

Step 3: Model Selection and AIC Calculation

Based on the above analysis, we made the following assumptions for the SARIMA model parameters:

Table 2 Parameter assumption

SARIMA		
ARIMA (p, d, q)	p	1,2
	d	1
	q	1,2
Seasonal portion (P, D, Q, S)	P	1,0
	D	1,0
	Q	1
	S	6

To select the optimal model, our team performed an AIC (Akaike Information Criterion) evaluation[6]. AIC is a criterion used to assess model quality, with the formula:

$$\mathrm{AIC} = 2k - 2\ln(L) \tag{6}$$

where k is the number of parameters in the model and L is the model's maximum likelihood estimate. A smaller AIC value indicates a better model fit, while also accounting for model complexity to avoid overfitting. By comparing AIC values across different parameter combinations, the following results were achieved:

Table 3 AIC Results

Model	AIC
SARIMA(1, 1, 1) ×(1, 1, 1, 6)	-6532.97318459987
SARIMA(2, 1, 1) ×(1, 1, 1, 6)	-6591.290085039436
SARIMA(1, 1, 2) ×(0, 1, 1, 6)	-6151.369364507325
SARIMA(2, 1, 1) ×(1, 0, 1, 6)	-6868.11041474036

The model with the smallest AIC value (SARIMA(2, 1, 1) × (1, 1, 1, 6)) was selected as the optimal model. Using this model for forecasting, the results are presented in Figure 12.

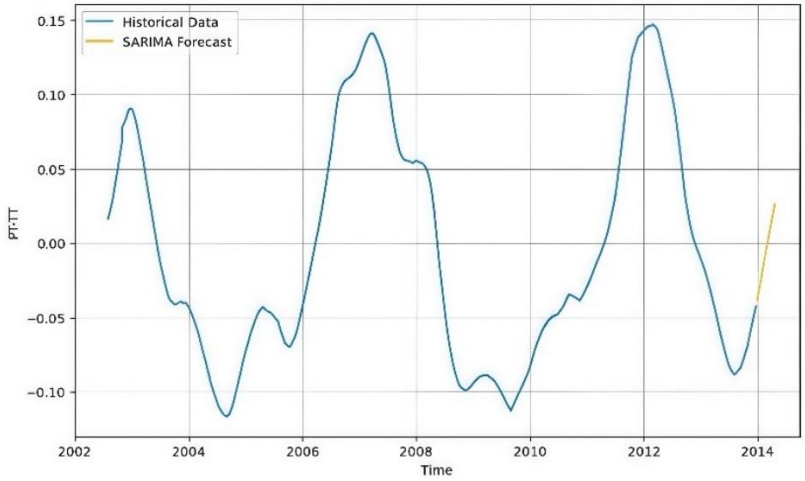

Figure 12 SARIMA Forecast

As illustrated, the SARIMA model demonstrates a significant advantage in predicting pulsar timing noise, as it effectively captures the periodic characteristics of the noise, thereby enhancing the prediction accuracy.

4.2.2 LSTM Model Establishment and Solution

While the SARIMA model performs well in short-term forecasting, the long-term trend of pulsar timing noise is better captured using the deep learning model, LSTM (Long Short-Term Memory Network)[7][8]. LSTM is particularly well-suited for time series data due to its ability to handle and predict long-term dependencies.

Initially, data was read from "Attachment 1.xlsx," which contains the corrected MJD and pulsar timing noise (PT-TT). To ensure data quality, we converted the "PT-TT" column to numeric values and removed all rows containing missing values.

Next, to facilitate training, the "PT-TT" data was normalized using MinMaxScaler to the [0, 1] range, which helps mitigate issues like gradient explosion and vanishing gradients in LSTM models.

To make time series predictions, we constructed a function create_dataset() that uses a sliding window of 300 days of data to predict the noise value at the next time point. The time_step was set to 300, meaning that each input feature is derived from the past 300 days of data, generating feature data X and target data y. For performance evaluation, we split the dataset into training and test sets.

These preprocessing steps ensured the readiness of the data for LSTM training, focusing on cleaning, normalization, and dataset partitioning to improve convergence and prediction accuracy. The principles of the LSTM model are as follows:

1. **Input Gate**: The input gate controls the degree to which the current input data enters the memory cell. Let the input vector be $x_i^{(t)}$, then the activation function of the input gate is:
$$i_t = \sigma(W_i \cdot x_i^{(t)} + U_i \cdot h_i^{(t-1)} + b_i) \tag{7}$$
2. **Forget Gate**: The forget gate controls how much information in the memory cell should be retained or discarded, allowing the LSTM to selectively remove irrelevant historical data and manage long-term dependencies. The activation function of the forget gate is:
$$f_t = \sigma(W_f \cdot x_i^{(t)} + U_f \cdot h_i^{(t-1)} + b_f) \tag{8}$$
3. **Memory Cell Update**: The memory cell is the core of the LSTM, storing both short- and long-term information relevant to the time series. The memory cell is updated by the input and forget gates, which combine the current input with the previous memory state. The update of the memory cell state is:
$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tanh(W_c \cdot x_i^{(t)} + U_c \cdot h_i^{(t-1)} + b_c) \tag{9}$$
4. **Output Gate**: The output gate controls how the current hidden state is updated. The hidden state represents the feature representation of the time series at the current time step, and the output gate regulates the amount of information from the memory cell that is transferred to the hidden state.
$$o_t = \sigma(W_o \cdot x_i^{(t)} + U_o \cdot h_i^{(t-1)} + b_o) \tag{10}$$
$$h_i^{(t)} = o_t \cdot \tanh(C_t) \tag{11}$$

After multi-layer processing by the LSTM, the time series features at each node are effectively encoded, capturing both short-term fluctuations and long-term trends in the data. During model training, our team chose the Adam optimizer for optimization. Adam combines momentum with adaptive learning rates, resulting in faster convergence and higher stability when handling non-stationary data and complex patterns. This is especially important for our task, as the pulsar timing noise data contains significant complexity and potential long-period trends. Using the Adam optimizer enables the model to better learn these features. The predicted results are shown in the figure below:

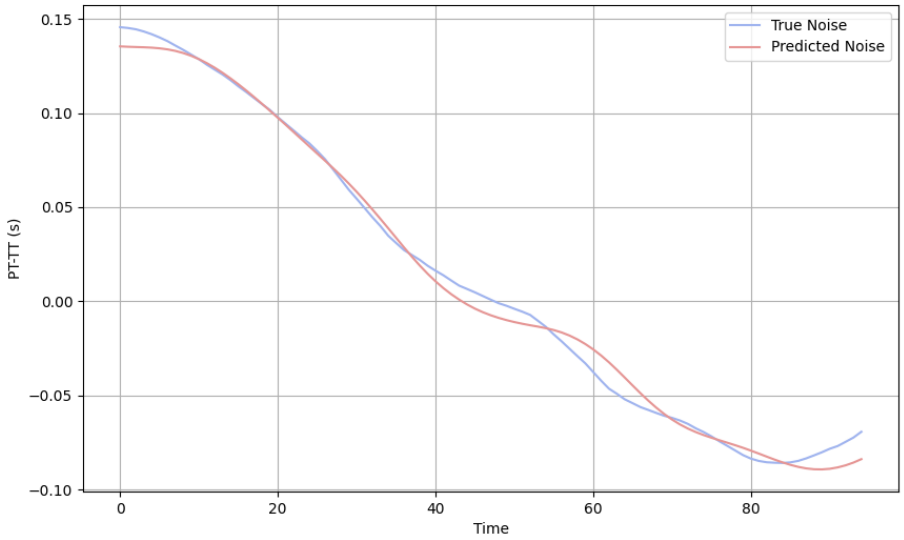

Figure 13 Predicted result

To conduct a comprehensive evaluation of the model's performance, our team used multiple evaluation metrics to quantify the predictive effectiveness of the model, as outlined below:

Mean Squared Error (MSE): MSE is a widely used measure for assessing the difference between a model's predicted results and the true values. It calculates the average squared difference between the predicted and actual values. MSE is a non-negative value, and the smaller it is, the closer the model's predictions are to the actual values. The formula is:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{12}$$

Root Mean Squared Error (RMSE): A smaller RMSE indicates a smaller difference between predicted and actual values. The formula is:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{13}$$

Mean Absolute Error (MAE): MAE measures the average absolute difference between predicted and actual values. The formula is:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}| y_i - \hat{y}_i | \tag{14}$$

R-Squared (R^2): R^2 is a standardized metric used to evaluate the goodness-of-fit of the model, which also facilitates comparisons across different models. The value of R^2 ranges from 0 to 1, with values closer to 1 indicating better predictive performance. The formula is:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - y)^2} \tag{15}$$

Table 4 Evaluation indicators and results

Evaluation index	LSTM
Mean Squared Error (MSE)	0.000038392204970492
Root Mean Squared Error (RMSE)	0.006196144363270798
Mean Absolute Error (MAE)	0.004774517149900879
R-Squared (R^2)	0.9936738159777331

The model evaluation results indicate that the model exhibits exceptional accuracy in predicting pulsar timing noise, with minimal error values, suggesting that

the predicted values closely match the actual data. Additionally, the R^2 value, which approaches 1, implies that the model explains the majority of the variance in the data, reflecting its excellent fitting capability. Overall, these indicators demonstrate that the model performs strongly, particularly in capturing long-term trends and overall variations. The long-term forecast of the future trend of pulsar timing noise is as follows:

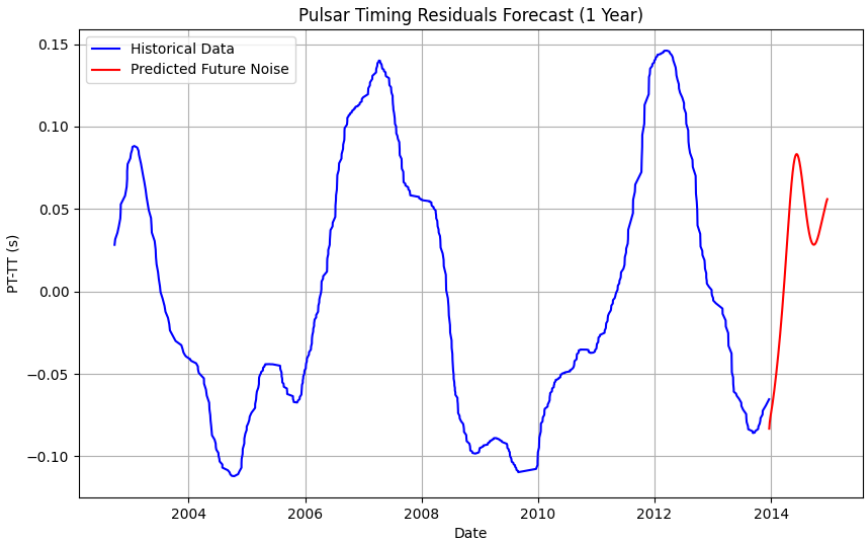

Figure 14 Forecast result

4.3 Problem Three: Model Establishment and Solution

In high-frequency radio observations, atmospheric delay significantly affects the time of arrival (TOA) of signals, which is especially critical in applications requiring high-precision time synchronization. Problem 3 aims to model the refractive time delay of radio observations at frequencies above 20 GHz. To address this, this paper proposes a high-frequency atmospheric delay correction model. The specific process is illustrated in the figure below.

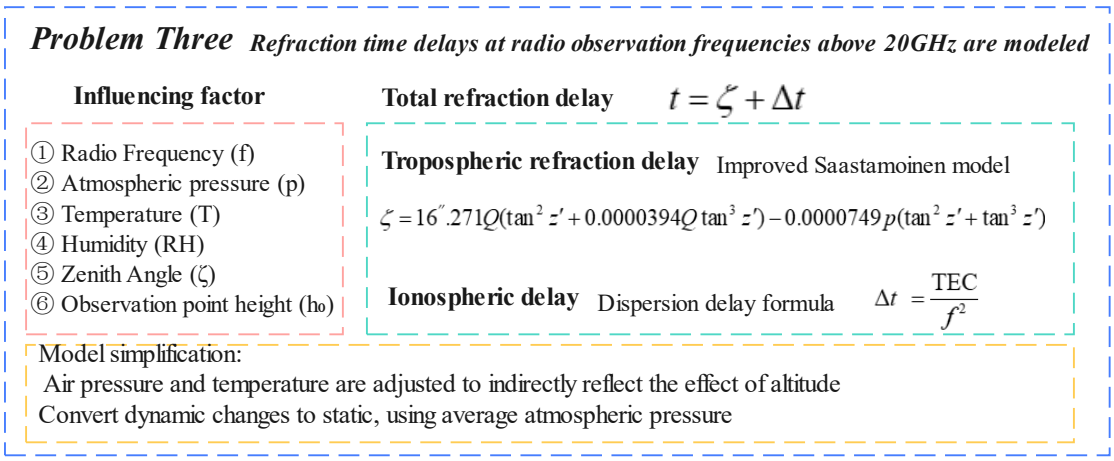

Figure 15 Flowchart of Problem Three

Drawing from the background and Chapter 6.1 of the reference literature "Space-Time Reference Systems," the team conducted a comprehensive analysis of atmospheric refractive delay and identified the following primary influencing factors:

1. **Radio Frequency (f)**: Atmospheric refractive delay is inversely proportional to frequency. Delays are more significant at lower frequencies, while they diminish at higher frequencies.

2. **Atmospheric Pressure (p)**: Higher pressure leads to a higher refractive index, resulting in increased delay.
3. **Temperature (T)**: Lower temperatures increase air density, enhancing the refractive effect.
4. **Humidity (RH)**: Humid air has a lower refractive index than dry air, which impacts accuracy.
5. **Zenith Angle (ζ)**: Smaller zenith angles result in larger delays.
6. **Observation Point Altitude (h_0)**: At higher altitudes, the atmosphere is thinner, weakening the refractive effect.

In the experiment, certain influences were approximated. For instance, at frequencies above 20 GHz, ionospheric refractive effects are minimal, so the team neglected more complex factors such as solar activity. This simplification reduces the model's complexity while preserving computational accuracy at high frequencies. Regarding the observation point's altitude, the team did not directly treat it as a variable but instead accounted for it by adjusting pressure and temperature, which adequately reflects altitude effects for most high-altitude areas.

Moreover, meteorological conditions were treated as constant, despite their continuous variation. By using average meteorological conditions, the team was able to obtain useful delay estimates while maintaining the model's simplicity.The remaining factors were modeled as follows:

Tropospheric Refractive Delay Model: The improved Saastamoinen model (1972) was used for the tropospheric refractive delay calculation.

$$p_w = H_{\text{rel}}\left(\frac{273.15+T}{247.1}\right)^{18.36} \tag{16}$$

$$Q = \frac{p-0.156p_w}{273.15+T} \tag{17}$$

$$\eta_1 = 16.271Q - 7.49\times10^{-5}p \quad \eta_2 = 6.410774\times10^{-10}Q^2 - 7.49\times10^{-5}p \tag{18}$$

The final formula for the tropospheric refractive delay is:

$$\zeta = 16''.271Q(\tan^2 z' + 0.0000394Q\tan^3 z') - 0.0000749p(\tan^2 z' + \tan^3 z') \tag{19}$$

Ionospheric Delay Model: Ionospheric refractive delay, related to frequency f and total electron content (TEC), was modeled using the dispersion delay formula:

$$\Delta t = \frac{\text{TEC}}{f^2} \tag{20}$$

Total Refractive Delay Model:

$$t = \zeta + \Delta t \tag{21}$$

During the modeling process, the team calculated the total refractive delay within the 20 GHz to 40 GHz frequency range to ensure that the refractive delay stayed below the target limit of 7.69 ns. The results are shown below:

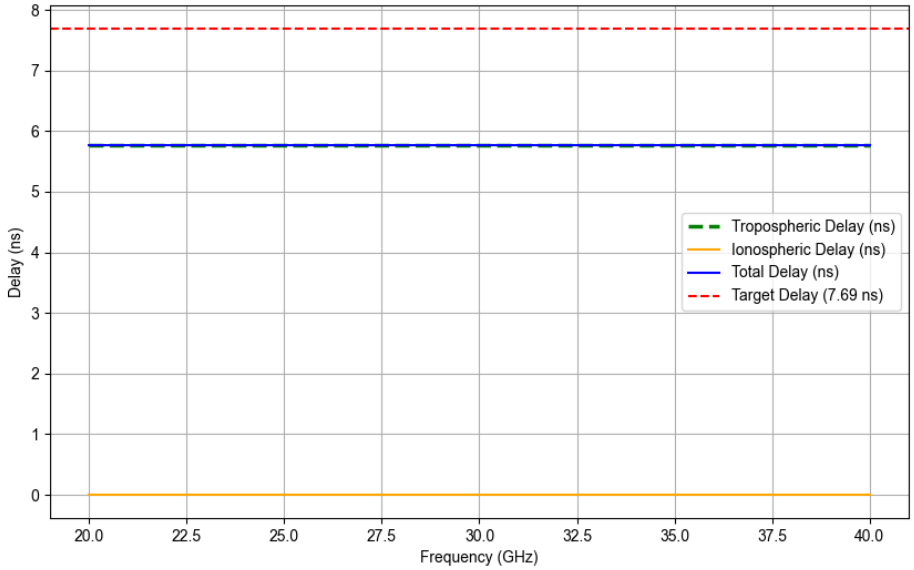

Figure 16 Refractive Delay Results

The figure displays the refractive delay across the 20 GHz to 40 GHz frequency range, where the tropospheric delay remains around 6 ns, showing little sensitivity to frequency. The ionospheric delay decreases rapidly to zero as the frequency increases. Therefore, the total delay is primarily governed by the tropospheric delay, which remains stable around 6 ns. The red dashed line indicates the target delay of 7.69 ns. As shown, the total delay consistently stays below this target, confirming that the model's predicted total delay meets the required accuracy. Especially at higher frequencies, the ionospheric effect on the total delay is negligible.

4.4 Problem Four: Model Establishment and Solution

In precise measurement and satellite navigation applications, atmospheric delay significantly impacts radio signal propagation. To enhance Time of Arrival (TOA) measurement accuracy, particularly during low elevation angle observations, detailed modeling and correction of atmospheric delays are essential. This section outlines the establishment of a small elevation angle delay mapping model designed to resolve atmospheric delays in both the ionosphere and troposphere, discussing their effects on signal propagation. The process is as follows:

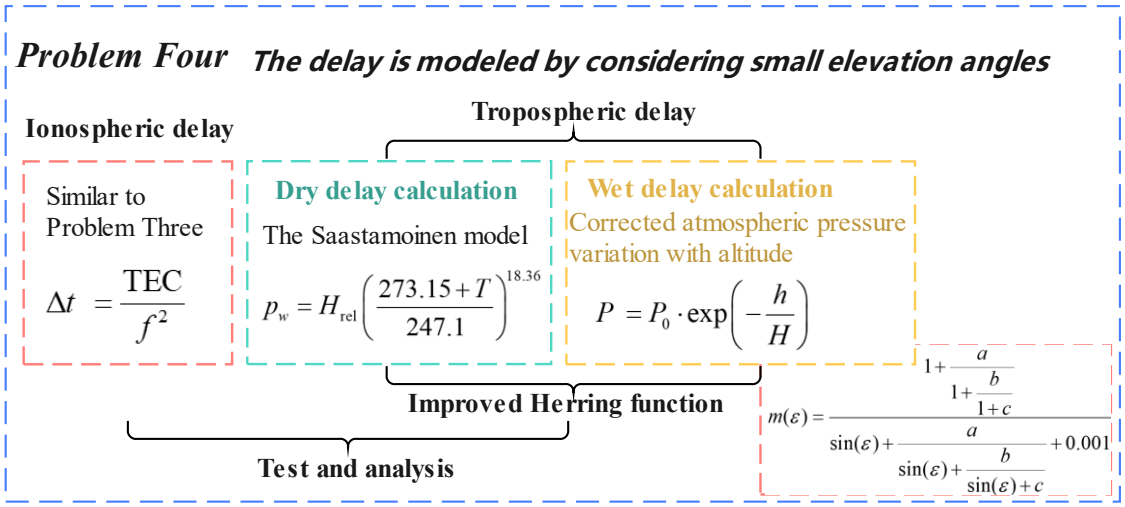

Figure 17 Process Flowchart for Problem Four

We begin with ionospheric delay modeling, analyzing its effects on signals of varying frequencies, and proposing an appropriate delay calculation model. Details are provided below:

4.4.1 Ionospheric Delay Modeling

The presence of free electrons in the ionosphere affects radio signal propagation, resulting in dispersion delays. Ionospheric delay is inversely proportional to the square of the signal frequency. For higher frequencies (>20 GHz), ionospheric delay is negligible; however, for lower frequencies, it becomes more pronounced. The approach our team adopts for ionospheric delay is consistent with that described in Problem 3 and will not be reiterated here.

4.4.2 Tropospheric Delay Modeling

Atmospheric components affect radio signal propagation differently, leading to the separation of tropospheric delay into two components: dry delay and wet delay.

Dry Delay arises from the dry components of the troposphere (e.g., nitrogen and oxygen). These components make up most of the atmosphere and have relatively stable physical properties. Dry delay is strongly correlated with atmospheric pressure and temperature, influencing the refractive path of radio signals. Under low elevation conditions, the increased signal path length through the atmosphere significantly amplifies dry delay.

Wet Delay is caused by water vapor content in the troposphere. Wet delay is typically smaller than dry delay but is more complex and harder to predict. Water vapor distribution is influenced by temperature, humidity, and weather conditions, leading to significant fluctuations. Wet delay impacts signal propagation, particularly in regions with higher water vapor concentrations.

Given the distinct physical sources and mechanisms of dry and wet delays, our team models these components separately for more precise representation of the refractive delay effects on signal propagation.

1. Dry Delay Calculation

Tropospheric dry delay is primarily caused by dry components such as nitrogen and oxygen, and is influenced by factors like atmospheric pressure and temperature. Under low elevation angles, dry delay increases significantly with changes in zenith angle. The Saastamoinen model, based on the average wavelength K (574 nm) for visible light, is applicable for zenith distances less than 70° (x' < 70°). Consequently, our team utilizes the Saastamoinen model to calculate dry delay under these conditions, as it provides a reliable representation of delay effects. The model expression is as follows:

$$p_w = H_{\text{rel}} \left(\frac{273.15 + T}{247.1} \right)^{18.36} \tag{22}$$

At low elevation angles (e.g., 10° or lower), the mapping function is crucial for accurate tropospheric delay calculation. This function transforms zenith delay values for different elevation angles. When signals come from the zenith, their propagation path is shortest and dry delay minimal. However, at low elevation angles, the signal path lengthens significantly, amplifying the delay. The Herring mapping function is employed to model this relationship, converting zenith delay (elevation angle = 90°) to delays at any other elevation angle.The mapping function is expressed as:

$$m(\varepsilon)=\frac{1+\dfrac{a}{1+\dfrac{b}{1+c}}}{\sin(\varepsilon)+\dfrac{a}{\sin(\varepsilon)+\dfrac{b}{\sin(\varepsilon)+c}}} \tag{23}$$

At low elevation angles, signal propagation through the atmosphere lengthens, increasing delay. The mapping function's precision is thus critical. However, at very low elevation angles (e.g., 1-10°), the denominator approaches zero, leading to numerical instability. Numerical instability occurs when rounding errors or limited computational precision distort delay modeling. Therefore, we use an improved Herring function, adding a small constant to prevent numerical instability at low angles. The updated formula is as follows:

$$m(\varepsilon)=\frac{1+\dfrac{a}{1+\dfrac{b}{1+c}}}{\sin(\varepsilon)+\dfrac{a}{\sin(\varepsilon)+\dfrac{b}{\sin(\varepsilon)+c}}+0.001} \tag{24}$$

This modification ensures that the denominator never reaches zero, remaining at least 0.001, thus preventing the mapping function from becoming infinitely large. This adjustment improves numerical stability, even at extremely low elevation angles, ensuring accurate delay calculations.

2. Wet Delay Calculation

Wet delay is caused by water vapor in the atmosphere. While smaller than dry delay, it still significantly affects low elevation observations. Wet delay is more unpredictable due to rapid changes in water vapor distribution driven by temperature, humidity, and weather conditions. This variability makes wet delay more irregular than dry delay, especially in the lower troposphere near the surface. Wet delay refracts the signal path, with a more pronounced effect when signals pass through regions with higher water vapor content.

Our team adopts a similar modeling approach for wet delay as for dry delay. The calculation relies on the mapping function to accommodate changes in signal path length with varying elevation angles. The mapping function is pivotal in converting zenith wet delay to wet delay at any elevation angle. This is particularly important for low elevation observations, where the signal traverses longer paths through regions with higher water vapor content, increasing wet delay.

In both dry and wet delay calculations, atmospheric pressure changes with altitude, affecting signal propagation. Therefore, we introduce an altitude correction for atmospheric pressure. Using an exponential model for pressure decrease with altitude, we adjust the base pressure P0 to the pressure P at the observation altitude:

$$P = P_0 \cdot \exp\left(-\frac{h}{H}\right) \tag{25}$$

The altitude correction accounts for the impact of altitude-dependent atmospheric pressure on dry and wet delays. As altitude increases, the atmosphere's density and pressure decrease, directly influencing refractive effects in signal propagation. This

correction enhances the accuracy of delay modeling, particularly in high-altitude or satellite signal reception scenarios.

Through accurate modeling of dry and wet delays, the application of elevation angle mapping functions, and altitude-dependent atmospheric pressure adjustments, our team can comprehensively model refractive delays in the atmosphere, ensuring precise low elevation observations. The results are as follows:

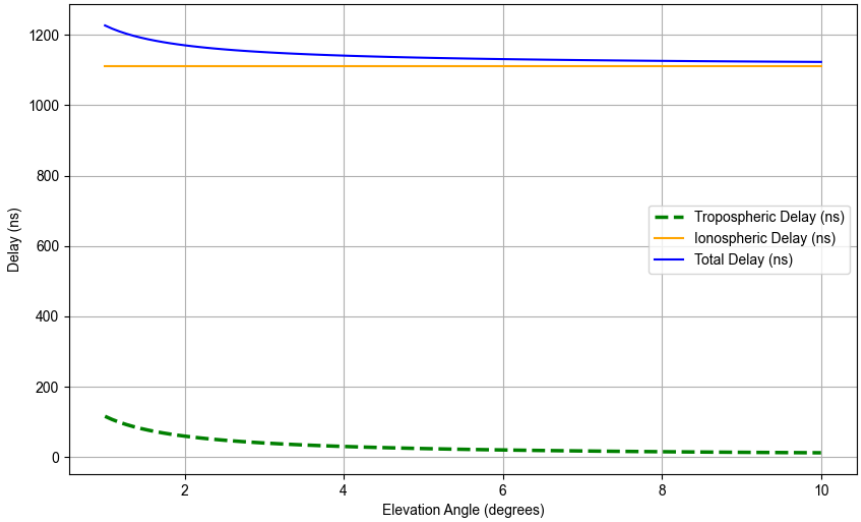

Figure 18 Delay Results

The figure shows significant changes in tropospheric delay at low elevation angles. This is due to the longer signal path, which intensifies the tropospheric refraction effect. Therefore, precise modeling and correction of tropospheric delay is crucial under low elevation angles. The ionospheric delay curve remains nearly flat, indicating that ionospheric delay is minimally affected by elevation angle changes at 30 GHz and occupies a stable portion of total delay. Total delay is higher at low elevation angles and gradually decreases as the elevation angle increases, stabilizing at a value dominated by ionospheric delay.

5. Test the Models

5.1 Evaluation of the SARIMA Model

To assess the predictive performance of the models, our team conducted a systematic comparison between the SARIMA and ARIMA models in the context of pulsar timing noise prediction. The data were split into training and test sets. Both models were trained on the training set and their short-term forecasting abilities were evaluated on the test set. The following presents a comparison of the prediction results for both models along with a detailed evaluation based on the specified performance metrics:

Table 5 Model Evaluation

Evaluation index	SARIMA	AMRIMA
Mean Squared Error (MSE)	0.000114314412	0.000173251793
Root Mean Squared Error	0.01144458593	0.01572807066
Mean Absolute Error (MAE)	0.008381908669	0.013179731772
R^2 (R-squared)	0.96751486047	0.916253615584

Through comparative analysis, the SARIMA model demonstrated clear superiority in the pulsar timing noise forecasting task. Its lower forecasting error and better fitting performance indicate that SARIMA is more effective in capturing periodic patterns and trend changes in time series data, thus offering higher accuracy.

Therefore, choosing SARIMA as the primary forecasting model was a well-founded decision and provides strong support for the successful achievement of the task's objectives.

5.2 LSTM Model Testing

To thoroughly evaluate the predictive accuracy of the models and ensure the scientific validity and rationality of the model selection, our team systematically compared the applicability of different algorithms. Specifically, we selected several representative models from state-of-the-art deep learning and machine learning methods, including LSTM (Long Short-Term Memory), GRU (Gated Recurrent Unit), MLP (Multilayer Perceptron), and Random Forest, and assessed their performance in the pulsar timing noise prediction task.

In the experiments, each model was trained and validated under identical dataset and preprocessing conditions, with performance metrics such as Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and Coefficient of Determination (R^2) used for evaluation. Finally, by comparing the models' predictive performance with the actual time series trends, we plotted the comparison of model performance, as shown in the figure below:

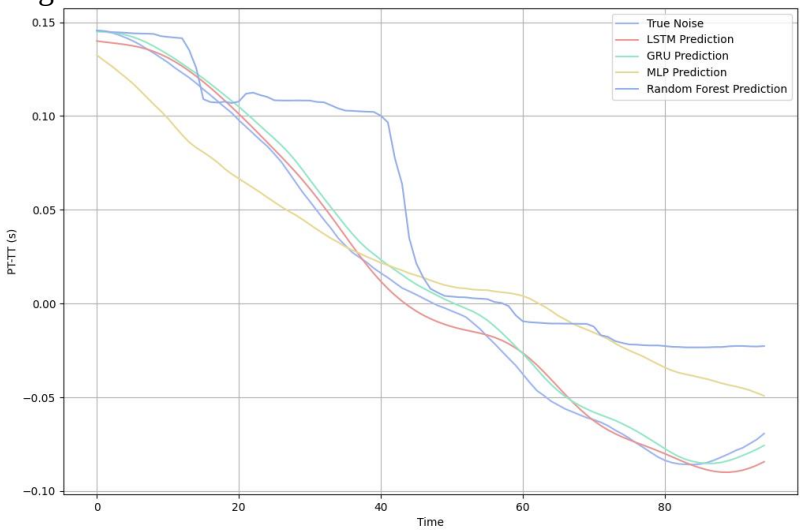

Figure 19 Multi-Model Comparison

The evaluation results for each model are shown in Table 6:

Table 6 Evaluation result

Index	LSTM	GRU	MLP	Random forest
MSE	0.0000836648680	0.00010342626285	0.0016250837870	0.0014565956207
RMSE	0.0001034262628	0.01016987034608	0.0403123279779	0.0381653720106
MAE	0.0075326752195	0.00888672377311	0.0323078227051	0.0317781984186
R^2	0.9934591995803	0.99222384052113	0.8292912730692	0.6846989635221

From the analysis above, the LSTM model excelled in all key metrics. In particular, it exhibited the lowest prediction errors (MSE, RMSE, MAE) and the highest fitting accuracy (with R^2 closest to 1), clearly demonstrating that LSTM can effectively capture the complex patterns and long-term dependencies present in pulsar timing noise. Compared to other models, LSTM's superior performance further validates its high applicability to this task, justifying the decision to select LSTM as the primary prediction model.

6. Sensitivity Analysis

6.1 LSTM Sensitivity Analysis

In this section, we performed a sensitivity analysis of several hyperparameters of the LSTM model to assess their effect on model performance. The hyperparameters include Time Step, Units, Batch Size, and Epochs. Model performance was evaluated using metrics such as Mean Squared Error (MSE), Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and Coefficient of Determination (R^2). These metrics were used for a quantitative analysis of the model's performance under various experimental conditions.

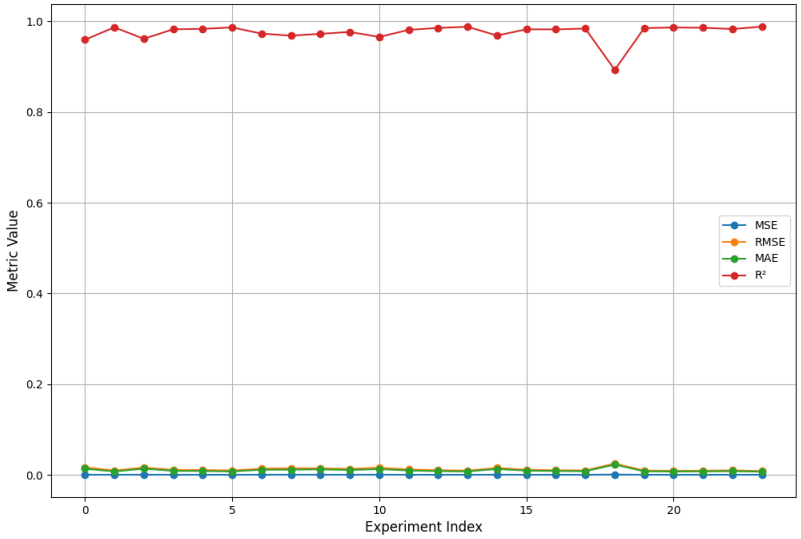

Figure 20 LSTM Sensitivity Analysis

The figure shows that the values of MSE, RMSE, and MAE remain consistently low and exhibit a stable trend across different experiments, indicating that the LSTM model performs consistently well across different hyperparameter combinations. Notably, as the number of training epochs increases, both RMSE and MAE decrease, demonstrating that more training significantly enhances the model's fitting ability.

The best performance across all experiments was achieved with a combination of a time step of 300, 50 units, a batch size of 32, and 20 epochs. The model achieved an MSE of 0.000100947, RMSE of 0.008047, MAE of 0.006927, and R^2 of 0.9923. This confirms the reasonableness and effectiveness of the chosen hyperparameter combination and further demonstrates the model's stability and excellent fitting ability. Some of the experimental results are shown below (full is provided in Appendix 1).

Table 7 Experimental Results Data

Time step	units	Batch size	epochs	MSE	RMSE	MAE	R^2
50	30	16	10	0.000131501	0.011467404	0.009016095	0.98058267
50	30	16	20	7.70169E-05	0.008775928	0.007102216	0.98862777
50	30	32	10	0.000207339	0.01439928	0.012277332	0.96938453
50	30	32	20	0.000176778	0.013295802	0.010687834	0.97389712
50	50	16	10	0.000145358	0.012056439	0.009695188	0.97853665
50	50	16	20	0.000129403	0.011375551	0.009001332	0.98089248

As seen in the table, increasing batch size and decreasing training epochs tends to increase error metrics like MSE and MAE, while increasing the number of training epochs allows the model to better fit the data, thereby reducing error. Overall, the

lower values of MSE, RMSE, and MAE indicate that the model's error is well-controlled, and the higher R^2 value (close to 1) demonstrates the model's excellent fit and strong explanatory power over the data.

6.2 Problem Three: Sensitivity Analysis

In this study, we conducted a sensitivity analysis of the relationship between zenith delay and frequency under different Total Electron Content (TEC) conditions, with the results shown in the figure below. By varying TEC values and frequency ranges, we evaluated the trend of zenith delay variation to better understand the impact of different observational conditions on delay.

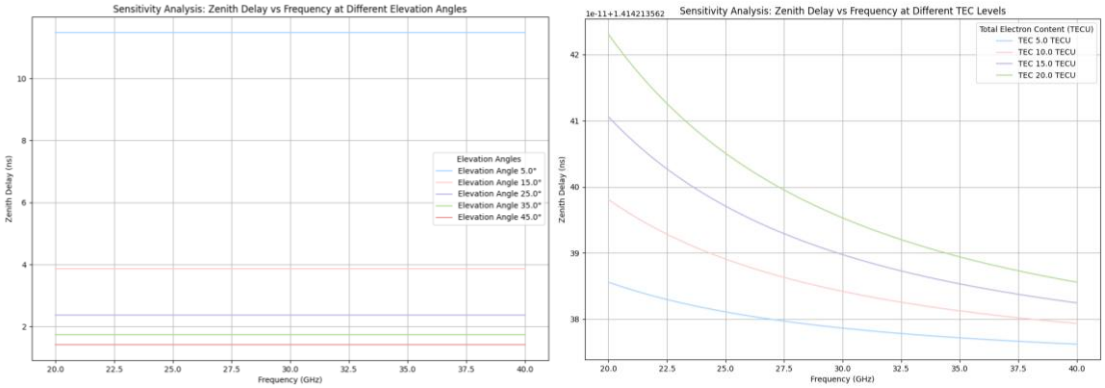

Figure 21 Sensitivity Analysis Results

From the comparison of the two figures, it can be seen that regardless of the TEC values or elevation angles, zenith delay stabilizes at high frequencies (especially above 30 GHz), indicating that in this frequency range, electromagnetic wave propagation is less affected by ionospheric and tropospheric variations. Therefore, in precision measurements, satellite communication, and astronomical observations, using high-frequency signals effectively reduces delay and enhances measurement accuracy and signal stability.

The analysis of the elevation angle results shows that at smaller elevation angles, the signal travels a longer atmospheric path, leading to higher delay. Increasing the elevation angle reduces the delay, especially at low frequencies, where this effect is more pronounced. This suggests that in practical applications, selecting larger elevation angles and higher frequencies can optimize signal quality and reduce delay. Some of the results are shown below:

Table 8 Partial Sensitivity Analysis Results

Elevation Angle (°)	TEC (TECU)	Pressure (hPa)	Frequency (GHz)	Zenith Delay (ns)
5.0	5	900	21	10.19382223
5.0	5	900	30	10.19382223
5.0	5	900	40	10.19382223
5.0	5	966.6666667	21	10.94892018
5.0	5	966.6666667	30	10.94892018
5.0	5	966.6666667	40	10.94892018
5.0	5	1033.333333	21	11.70401812
5.0	5	1033.333333	30	11.70401812

These conclusions confirm that our model demonstrates robust stability and accuracy across various conditions, offering valuable insights for the design of practical satellite communication and observation systems. By appropriately selecting

elevation angles and frequencies, propagation delay can be minimized, thereby enhancing communication quality.

6.3Problem Four: Sensitivity Analysis

This section presents a sensitivity analysis of the relationship between zenith delay and frequency under varying observational conditions, aiming to evaluate the effects of different elevation angles, Total Electron Content (TEC), and atmospheric pressure on zenith delay.

First, we analyzed how zenith delay varies with frequency at different elevation angles. Figure 1 illustrates the effect of various elevation angles (5°, 15°, 25°, 35°, 45°) on zenith delay across the 20 GHz to 40 GHz frequency range. From the figure, it is evident that zenith delay decreases as frequency increases, particularly in the lower frequency range (20 GHz to 25 GHz), where zenith delay shows noticeable oscillations. This is followed by gradual stabilization at higher frequencies.

This indicates that at lower frequencies, elevation angle changes significantly affect delay, with larger elevation angles leading to a significant reduction in zenith delay. This phenomenon is related to the signal's propagation path in the atmosphere and the effects of atmospheric refraction.

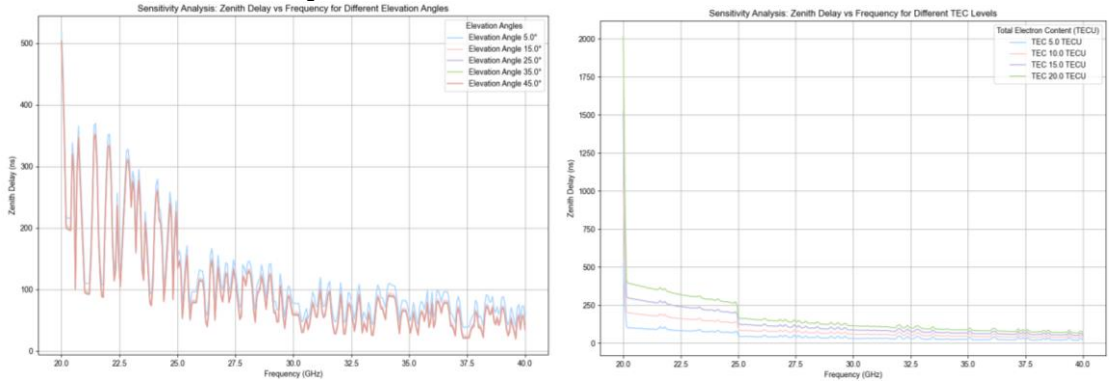

Figure 22 Sensitivity Analysis Results

Next, we analyzed the impact of different TEC values (5, 10, 15, 20 TECU) on zenith delay, and the results are presented in Figure 22. As shown in the figure, zenith delay increases significantly with TEC, with a more pronounced effect in the lower frequency range (20 GHz to 25 GHz). However, as frequency increases, the impact of TEC on delay diminishes, and the curves gradually level off. This suggests that at high frequencies, the ionospheric refraction and delay effects on the signal become negligible. Some of the data are presented below:

Table 9 Partial Sensitivity Analysis Results

Elevation Angle (°)	TEC (TECU)	Pressure (hPa)	Frequency (GHz)	Zenith Delay (ns)
5	5	900	21	10.19382223
5	5	900	30	10.19382223
5	5	900	40	10.19382223
5	5	966.6666667	21	10.94892018
5	5	966.6666667	30	10.94892018
5	5	966.6666667	40	10.94892018
5	5	1033.333333	21	11.70401812
5	5	1033.333333	30	11.70401812

As shown in the table, zenith delay increases with atmospheric pressure, and this effect is more noticeable under higher pressure conditions. The sensitivity analysis confirms that our model remains stable and accurate under varying TEC, elevation

angle, and atmospheric pressure conditions, providing solid support for delay calculations in both the ionosphere and troposphere.

7. Strengths and Weakness

7.1 Strengths

The SARIMA model is highly effective in capturing seasonal and trend patterns in time series. Its intuitive structure and clear parameter adjustments make it easy to use and interpret, especially for short-term forecasting, where it quickly adapts to data fluctuations.

LSTM automatically learns hidden features in time series, excelling at capturing complex patterns and long-term dependencies. It avoids manual feature engineering, making it ideal for non-stationary, nonlinear, and high-dimensional data, with its multi-layer structure enhancing long-term forecasting performance.

By incorporating mapping functions for wet and dry delays, the model improves performance under low elevation angle conditions, which is crucial for applications like astronomical observations and deep space exploration.

7.2 Weakness

The SARIMA model has limitations in handling nonlinear relationships and long-term forecasting, while LSTM, although highly effective at capturing complex patterns, has drawbacks in terms of computational cost, data requirements, and interpretability. The improved mapping function resolves issues in specific scenarios but needs further enhancement to improve its generality. Considering these limitations helps in selecting the most suitable model or making targeted optimizations for practical applications.

8. Conclusion

In this study, we modeled and analyzed the effects of pulsar timing noise and atmospheric delays on pulsar time signal accuracy. Using Fourier transform and PSD modeling, we developed and validated a model for pulsar red noise. Combining SARIMA and LSTM models, we captured periodic pulsar noise variations and improved long-term forecasting accuracy.

For frequencies above 20 GHz, we applied the improved Saastamoinen model to calculate tropospheric delay and combined it with the ionospheric dispersion model to compute the total atmospheric refractive delay, ensuring it remained below 7.69 nanoseconds. Results showed a decreasing ionospheric impact on delay with higher frequencies, while tropospheric delay was mainly influenced by meteorological conditions.

In low elevation angle conditions, the improved mapping function resolved instability at small angles, ensuring accurate zenith delay calculations. Sensitivity analysis further validated the robustness of our model under different observational conditions, supporting its potential for future pulsar precision time signal applications.

These findings provide practical solutions for time signal processing in fields like precision measurement, satellite communication, and deep space navigation. By considering both ionospheric and tropospheric delays, our model demonstrates strong stability and accuracy, offering valuable insights for future pulsar signal measurement advancements.

References

[1] Bracewell, R. N. (2000). *The Fourier Transform and Its Applications* (3rd ed.). McGraw-Hill.
[2] Oppenheim, A. V., & Schafer, R. W. (2010). *Discrete-Time Signal Processing* (3rd ed.). Pearson.
[3] Percival, D. B., & Walden, A. T. (1993). Spectral Analysis for Physical Applications: Multitaper and Other Methods. Cambridge University Press.
[4] Jenkins, G. M., & Watts, D. G. (1968). *Spectral Analysis and its Applications*. Holden-Day.
[5] Box, G. E. P., Jenkins, G. M., & Reinsel, G. C. (2008). *Time Series Analysis: Forecasting and Control* (4th ed.). Wiley.
[6] Akaike, H. (1974). A new look at the statistical model identification. *IEEE Transactions on Automatic Control*, 19(6), 716-723.
[7] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735-1780. https://doi.org/10.1162/neco.1997.9.8.1735.
[8] Chung, J., et al. (2014). *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*. arXiv preprint

Appendix

LSTM sensitivity analysis detailed data

Table 10 LSTM sensitivity analysis detailed data

Time step	units	Batch size	epochs	MSE	RMSE	MAE	R^2
50	30	16	10	0.000131501	0.011467404	0.009016095	0.98058267
50	30	16	20	7.70169E-05	0.008775928	0.007102216	0.98862777
50	30	32	10	0.000207339	0.01439928	0.012277332	0.96938453
50	30	32	20	0.000176778	0.013295802	0.010687834	0.97389712
50	50	16	10	0.000145358	0.012056439	0.009695188	0.97853665
50	50	16	20	0.000129403	0.011375551	0.009001332	0.98089248
50	50	32	10	0.000155469	0.012468703	0.009994765	0.97704370
50	50	32	20	0.000116548	0.010795749	0.008693994	0.98279063
100	30	16	10	0.000140055	0.011834487	0.009542017	0.98069977
100	30	16	20	0.000179051	0.013381005	0.010075605	0.97532591
100	30	32	10	0.000178922	0.013376172	0.011067607	0.97534373
100	30	32	20	0.000176882	0.013299696	0.010704639	0.97562486
100	50	16	10	0.000133819	0.01156801	0.009077813	0.98155915
100	50	16	20	9.6014E-05	0.009798672	0.007753799	0.98676884
100	50	32	10	0.000129075	0.011361121	0.008880617	0.98221287
100	50	32	20	0.000274275	0.016561254	0.013788195	0.96220363
300	30	16	10	0.000104507	0.010222887	0.008687039	0.98191666
300	30	16	20	9.16056E-05	0.009571082	0.007888391	0.98414911
300	30	32	10	0.000164419	0.012822593	0.011004924	0.97154994
300	30	32	20	7.09882E-05	0.008425447	0.006586221	0.98771663
300	50	16	10	0.000122095	0.011049646	0.009240761	0.97887347
300	50	16	20	7.89565E-05	0.008885746	0.006923735	0.98633784
300	50	32	10	9.5899E-05	0.009792806	0.007978609	0.98340620
300	50	32	20	0.000100947	0.008047219	0.00692651	0.99232807

Problem Three Sensitivity analysis Detailed data (part)

Table 11 Problem Three Sensitivity analysis Detailed data (part)

Elevation Angle (°)	TEC (TECU)	Pressure (hPa)	Frequency (GHz)	Zenith Delay (ns)
5	5	900	21	10.19382223
5	5	900	30	10.19382223
5	5	900	40	10.19382223
5	5	966.6666667	21	10.94892018
5	5	966.6666667	30	10.94892018
5	5	966.6666667	40	10.94892018
5	5	1033.333333	21	11.70401812
5	5	1033.333333	30	11.70401812

5	5	1033.333333	40	11.70401812
5	5	1100	21	12.45911606
5	5	1100	30	12.45911606
5	5	1100	40	12.45911606
5	10	900	21	10.19382223
5	10	900	30	10.19382223
5	10	900	40	10.19382223
5	10	966.6666667	21	10.94892018
5	10	966.6666667	30	10.94892018
5	10	966.6666667	40	10.94892018
5	10	1033.333333	21	11.70401812
5	10	1033.333333	30	11.70401812
5	10	1033.333333	40	11.70401812
5	10	1100	21	12.45911606
5	10	1100	30	12.45911606
5	10	1100	40	12.45911606
5	15	900	21	10.19382223
5	15	900	30	10.19382223
5	15	900	40	10.19382223
5	15	966.6666667	21	10.94892018
5	15	966.6666667	30	10.94892018
5	15	966.6666667	40	10.94892018
5	15	1033.333333	21	11.70401812
5	15	1033.333333	30	11.70401812
5	15	1033.333333	40	11.70401812
5	15	1100	21	12.45911606
5	15	1100	30	12.45911606
5	15	1100	40	12.45911606
5	20	900	21	10.19382223
5	20	900	30	10.19382223

Problem Four Sensitivity analysis Detailed data (part)

Table 12 Problem Four Sensitivity analysis Detailed data (part)

Elevation Angle (degrees)	TEC (TECU)	Pressure (hPa)	Frequency (GHz)	Zenith Delay (ns)
5	5	900	20	519.5628087
5	5	900	20.10050251	118.5653087
5	5	900	20.20100503	117.5826597
5	5	900	20.30150754	116.6145684
5	5	900	20.40201005	115.6607485
5	5	900	20.50251256	114.720921
5	5	900	20.60301508	113.7948135
5	5	900	20.70351759	112.8821602

5	5	900	20.8040201	111.9827019
5	5	900	20.90452261	111.0961852
5	5	900	21.00502513	110.2223632
5	5	900	21.10552764	109.3609947
5	5	900	21.20603015	108.511844
5	5	900	21.30653266	107.6746813
5	5	900	21.40703518	106.8492819
5	5	900	21.50753769	106.0354264
5	5	900	21.6080402	105.2329007
5	5	900	21.70854271	104.4414953
5	5	900	21.80904523	103.6610058
5	5	900	21.90954774	102.8912323
5	5	900	22.01005025	102.1319795
5	5	900	22.11055276	101.3830567
5	5	900	22.21105528	100.6442772
5	5	900	22.31155779	99.91545869
5	5	900	22.4120603	99.19642293
5	5	900	22.51256281	98.48699561
5	5	900	22.61306533	97.78700628
5	5	900	22.71356784	97.09628827
5	5	900	22.81407035	96.41467857
5	5	900	22.91457286	95.74201773
5	5	900	23.01507538	95.0781498
5	5	900	23.11557789	94.42292217
5	5	900	23.2160804	93.77618556
5	5	900	23.31658291	93.13779388
5	5	900	23.41708543	92.50760419

Code Section

The moving average method processes the data

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
# Read the Excel data
file_path = 'Attachment 1.xlsx'
data = pd.read_excel(file_path, header=None, names=["MJD", "PT-TT"])
# Print the structure of the data to ensure correct reading
print(data.head())
# Ensure the PT-TT column is of numeric type and remove any missing or anomalous values
data['PT-TT'] = pd.to_numeric(data['PT-TT'], errors='coerce')  # Convert PT-TT column to numeric, invalid values become NaN
data = data.dropna(subset=['PT-TT'])  # Drop rows containing NaN values
```

```
# Check the data again
print(data.head())
# Select the PT-TT data column for smoothing
pt_tt = data['PT-TT'].values
# Set the window size, using 10 days as the window size
window_size = 10
# Calculate the simple moving average
sma = np.convolve(pt_tt, np.ones(window_size)/window_size, mode='valid')
# Create a new DataFrame to save the results
result = pd.DataFrame({
    "MJD": data['MJD'][window_size-1:].values,  # Align the MJD with the moving average
    "PT-TT (Moving Average)": sma  # Moving average results
})
# Save the results to an Excel file
output_file_path = 'Attachment 2.xlsx'
result.to_excel(output_file_path, index=False)
print(f"Moving average results have been saved to the file: {output_file_path}")
# Visualize the results
plt.figure(figsize=(10, 6))
plt.plot(data['MJD'], pt_tt, label='Original PT-TT (s)', color='blue', alpha=0.6)
plt.plot(data['MJD'][window_size-1:], sma, label=f'Moving Average (Window size={window_size})', color='red', linewidth=2)
plt.xlabel('MJD (days)')
plt.ylabel('PT-TT (s)')
plt.title('Pulsar Timing Noise with Moving Average')
plt.legend()
plt.grid(True)
plt.show()
```

PSD

```
import pandas as pd
import numpy as np
from scipy.fft import fft, fftfreq
import matplotlib.pyplot as plt
# Step 1: Load data and extract relevant columns
data = pd.read_excel('Attachment 1.xlsx')  # Adjust based on the data file path
# Clean the data
data_cleaned = data.iloc[1:, 1:3]  # Skip the header row, select MJD and PT-TT columns
data_cleaned.columns = ['MJD(days)', 'PT-TT(s)']  # Rename columns
data_cleaned['MJD(days)'] = pd.to_numeric(data_cleaned['MJD(days)'], errors='coerce')
data_cleaned['PT-TT(s)'] = pd.to_numeric(data_cleaned['PT-TT(s)'], errors='coerce')
data_cleaned.dropna(inplace=True)  # Drop missing values
```

```
    # Extract time and signal data
    time = data_cleaned['MJD(days)'].values
    signal = data_cleaned['PT-TT(s)'].values
    # Step 2: Calculate time interval
    # Since the Fourier transform requires a uniformly sampled signal, we approximate the time interval by the average difference between adjacent time points
    delta_t = np.mean(np.diff(time))
    # Step 3: Perform Fourier Transform
    # Use the fft function to perform a Fast Fourier Transform, converting the time-domain signal to the frequency domain
    N = len(signal)  # Number of data points in the signal
    signal_fft = fft(signal)  # Perform Fourier transform on the signal
    frequencies = fftfreq(N, delta_t)  # Calculate corresponding frequencies
    # Step 4: Calculate Power Spectral Density
    # PSD (Power Spectral Density) is the energy intensity at each frequency, calculated from the Fourier transform results
    psd = np.abs(signal_fft)**2 / N  # Take the square of the absolute value of the Fourier transform result, then divide by the number of data points
    # Retain only the positive frequencies
    positive_frequencies = frequencies[frequencies > 0]
    positive_psd = psd[frequencies > 0]
    # Step 5: Plot Power Spectral Density
    plt.figure(figsize=(10, 6))
    plt.loglog(positive_frequencies, positive_psd, color='orange')
    plt.xlabel('Frequency (1/days)')
    plt.ylabel('Power Spectral Density')
    plt.title('Power Spectral Density of Pulsar Timing Noise')
    plt.grid(True)
    plt.show()
```

Parameter determination

```
    import numpy as np
    import pandas as pd
    from scipy.fft import fft, fftfreq
    from scipy.optimize import curve_fit
    import matplotlib.pyplot as plt
    data = pd.read_excel('Attachment 1.xlsx')  # Replace with actual file path
    data_cleaned = data.iloc[1:, 1:3]  # Skip header row, select MJD and PT-TT
    data_cleaned.columns = ['MJD(days)', 'PT-TT(s)']  # Rename columns
    data_cleaned['MJD(days)'] = pd.to_numeric(data_cleaned['MJD(days)'], errors='coerce')
    data_cleaned['PT-TT(s)'] = pd.to_numeric(data_cleaned['PT-TT(s)'], errors='coerce')
    data_cleaned.dropna(inplace=True)  # Drop missing values
    time = data_cleaned['MJD(days)'].values
    signal = data_cleaned['PT-TT(s)'].values
    delta_t = np.mean(np.diff(time))  # Assume uniform sampling
    N = len(signal)
    signal_fft = fft(signal)  # Fast Fourier Transform
    frequencies = fftfreq(N, delta_t)  # Calculate frequencies
```

```
    psd = np.abs(signal_fft)**2 / N    # Power spectral density
    positive_frequencies = frequencies[frequencies > 0]
    positive_psd = psd[frequencies > 0]
    RMS_1 = 75268.376 * 1e-6    # Microseconds to seconds
    RMS_2 = 78502.322 * 1e-6    # Microseconds to seconds
    def power_spectral_density_model(f, k, fc, q):
        P0 = k * RMS_1    # Use the first RMS value to calculate P0
        return P0 / (f / fc) ** q
    freq_mask = (positive_frequencies > 0.001)    # Set frequency range lower bound
    positive_frequencies_filtered = positive_frequencies[freq_mask]
    positive_psd_filtered = positive_psd[freq_mask]
    initial_guess = [1e-6, 1e-3, 2]    # Initial guess (k, fc, q)
    popt, pcov = curve_fit(power_spectral_density_model,
positive_frequencies_filtered, positive_psd_filtered, p0=initial_guess)
    k_opt, fc_opt, q_opt = popt
    print(f"Fitted parameters:\nk = {k_opt}\nfc = {fc_opt}\nq = {q_opt}")
```

Problem 1

```
    import numpy as np
    import pandas as pd
    from scipy.fft import fft, ifft, fftfreq    # Ensure ifft is imported
    from scipy.optimize import curve_fit
    import matplotlib.pyplot as plt
    from scipy.ndimage import gaussian_filter1d    # For smoothing
    # Define RMS values and fitting parameters
    RMS_1 = 75268.376 * 1e-6    # Microseconds to seconds
    RMS_2 = 78502.322 * 1e-6    # Microseconds to seconds
    k = 1.3860749746843805e-05
    fc = 0.11477713746580295
    q = 2.6087867011359687
    # Step 1: Define the power spectral density model for red noise
    def power_spectral_density_model(f, k, fc, q, RMS):
        P0 = k * RMS
        return P0 / (1 + (f / fc) ** q)
    # Step 2: Generate red noise time series from PSD
    def generate_red_noise(T, sampling_rate, k, fc, q, RMS):
        N = T * sampling_rate    # Number of data points
        f = fftfreq(N, d=1 / sampling_rate)    # Frequency array
        psd = power_spectral_density_model(np.abs(f), k, fc, q, RMS)    # Calculate
PSD for each frequency
        # Random phase and amplitude based on PSD
        amplitude = np.sqrt(psd) * np.random.normal(0, 1, N)
        phase = 2 * np.pi * np.random.rand(N)
        noise_freq = amplitude * np.exp(1j * phase)    # Frequency domain signal
        # Inverse FFT to get time domain signal
        red_noise = np.real(ifft(noise_freq))
        return red_noise
    # Step 3: Define parameters and time range
    T = 1000    # Duration (seconds)
    sampling_rate = 1    # Sampling rate (1Hz, one sample per second)
```

```
    # Generate two red noises with different RMS values
    red_noise_1 = generate_red_noise(T, sampling_rate, k, fc, q, RMS_1)
    red_noise_2 = generate_red_noise(T, sampling_rate, k, fc, q, RMS_2)
    time = np.linspace(0, T, T)
    # Step 4: Compute PSD and apply smoothing
    frequencies = fftfreq(T, d=1 / sampling_rate)[:T // 2]    # Positive frequencies
    psd_generated_1 = np.abs(fft(red_noise_1))[:T // 2]    # PSD for the first RMS
    psd_generated_2 = np.abs(fft(red_noise_2))[:T // 2]    # PSD for the second RMS
    # Apply smoothing to the PSD data
    psd_generated_1 = gaussian_filter1d(psd_generated_1, sigma=2)
    psd_generated_2 = gaussian_filter1d(psd_generated_2, sigma=2)
    # Step 5: Plot the generated red noise
    plt.figure(figsize=(12, 6))
    plt.plot(time, red_noise_1, color='#f3adbc', linewidth=1.5, label="Generated Red
Noise (RMS_1)")
    plt.plot(time, red_noise_2, color='#a6d3ff', linewidth=1.5, label="Generated Red
Noise (RMS_2)")
    plt.title("Generated Red Noise Time Series")
    plt.xlabel("Time (s)")
    plt.ylabel("Amplitude")
    plt.legend()
    plt.grid(True)
    plt.show()
    # Step 6: Define the model function for least squares fitting
    def red_noise_model(frequencies, k, fc, q, RMS):
        return power_spectral_density_model(frequencies, k, fc, q, RMS)
    # Step 7: Fit the generated PSD and calculate R²  value
    def fit_and_calculate_r2(psd_generated, frequencies, k, fc, q, RMS):
        initial_guesses = [
            [k, fc, q],
            [k * 1.1, fc * 0.9, q * 1.05],
            [k * 0.9, fc * 1.1, q * 0.95]
        ]
        best_r2 = -np.inf
        best_params = None
        best_fitted_psd = None
        for initial_guess in initial_guesses:
            params, _ = curve_fit(lambda f, k, fc, q: red_noise_model(f, k, fc, q,
RMS), frequencies, psd_generated,
                                  p0=initial_guess)
            fitted_psd = red_noise_model(frequencies, *params, RMS)
            # Calculate R²
            residuals = psd_generated - fitted_psd
            ss_res = np.sum(residuals ** 2)
            ss_tot = np.sum((psd_generated - np.mean(psd_generated)) ** 2)
            r2 = 1 - (ss_res / ss_tot)
            if r2 > best_r2:
                best_r2 = r2
                best_params = params
                best_fitted_psd = fitted_psd
```

```
            return best_r2, best_fitted_psd
    # Step 8: Calculate R²  values for both RMS values
    r2_1, fitted_psd_1 = fit_and_calculate_r2(psd_generated_1, frequencies, k, fc, q, RMS_1)
    r2_2, fitted_psd_2 = fit_and_calculate_r2(psd_generated_2, frequencies, k, fc, q, RMS_2)
    print(f"R²  for RMS_1: {r2_1:.4f}")
    print(f"R²  for RMS_2: {r2_2:.4f}")
    # Step 9: Plot the generated PSD and fitted PSD
    plt.figure(figsize=(12, 6))
    plt.plot(frequencies, psd_generated_1, label="Generated PSD (RMS_1)", color='#a6d3ff', linewidth=1.5)
    plt.plot(frequencies, fitted_psd_1, label="Fitted PSD (RMS_1)", color='#e69695', linestyle='--', linewidth=1.5)
    plt.plot(frequencies, psd_generated_2, label="Generated PSD (RMS_2)", color='#a0b4ef', linewidth=1.5)
    plt.plot(frequencies, fitted_psd_2, label="Fitted PSD (RMS_2)", color='#f3adbc', linestyle='--', linewidth=1.5)
    plt.title("Power Spectral Density of Generated Red Noise and Fitted Model")
    plt.xlabel("Frequency (Hz)")
    plt.ylabel("Power Spectral Density")
    plt.xscale("log")
    plt.yscale("log")
    plt.legend()
    plt.grid(True)
    plt.show()
```

Problem 2

```
    import pandas as pd
import matplotlib.pyplot as plt
from datetime import timedelta

# Load the data, skipping the first two rows and only loading the required columns (assuming the first column is unnecessary)
data = pd.read_excel('Attachment 1.xlsx', header=2, usecols=[1, 2], names=["MJD", "PT-TT"])

# Display the first few rows to ensure the data is loaded correctly
print(data.head())

# Clean up column names by stripping any leading/trailing spaces
data.columns = data.columns.str.strip()

# Remove rows with NaN values
data.dropna(subset=['MJD', 'PT-TT'], inplace=True)

# Ensure the MJD column is numeric and convert it to numbers, invalid values will become NaN
data['MJD'] = pd.to_numeric(data['MJD'], errors='coerce')
```

```
# MJD starting date: November 17, 1858
mjd_origin = pd.Timestamp('1858-11-17')

# Convert MJD to date type
data['MJD'] = mjd_origin + pd.to_timedelta(data['MJD'], unit='D')

# Set MJD as the index
data.set_index('MJD', inplace=True)

# Ensure PT-TT column is numeric
data['PT-TT'] = pd.to_numeric(data['PT-TT'], errors='coerce')

# Remove rows with NaN values (check again for NaNs)
data.dropna(subset=['PT-TT'], inplace=True)

# Convert 'MJD' and 'PT-TT' columns to numpy arrays (ensure they are 1D arrays)
mjd_values = data.index.to_numpy().flatten()  # Use index as MJD
pt_tt_values = data['PT-TT'].to_numpy().flatten()

# Plot the data
plt.figure(figsize=(10, 6))
plt.plot(mjd_values, pt_tt_values, label="PT-TT (s)", color='#a0b4ef')
plt.xlabel('MJD')
plt.ylabel('PT-TT (s)')
plt.title('Pulsar Timing Residuals vs Time')
plt.grid(True)
plt.legend()
plt.show()
```

Test for stationarity

```
from statsmodels.tsa.stattools import adfuller
# Perform ADF (Augmented Dickey-Fuller) test for stationarity on the original data
adf_result = adfuller(data['PT-TT'])
print('ADF Statistic:', adf_result[0])
print('p-value:', adf_result[1])
# If p-value > 0.05, it suggests that the data is non-stationary, and typically, differencing is required
```

Difference

```
# Perform first-order differencing to make the data stationary
data_diff_1 = data['PT-TT'].diff().dropna()
# Ensure the differenced data and index are one-dimensional numeric types
data_diff_1 = data_diff_1.astype(float)
# Plot the differenced time series
plt.figure(figsize=(10, 6))
plt.plot(data_diff_1.index.to_numpy(), data_diff_1.to_numpy(), label="Differenced PT-TT (1st Order)", color='#a0b4ef')
```

```
plt.title('First Order Differencing of PT-TT vs Time')
plt.xlabel('Time')
plt.ylabel('Differenced PT-TT (s)')
plt.legend()
plt.grid()
plt.show()
# Perform ADF test again to check for stationarity
adf_result_diff = adfuller(data_diff_1)
print('ADF Statistic (Differenced):', adf_result_diff[0])
print('p-value (Differenced):', adf_result_diff[1])
```

ACF,PACF

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
# Plot ACF and PACF charts to help determine the model's parameters p and q
plt.figure(figsize=(14, 6))
plt.subplot(1, 2, 1)
plot_acf(data_diff_1, lags=20, ax=plt.gca(), title="ACF (First Difference)")
plt.subplot(1, 2, 2)
plot_pacf(data_diff_1, lags=20, ax=plt.gca(), title="PACF (First Difference)")
plt.tight_layout()
plt.show()
```

Define SARIMA parameter combinations

```
# Define SARIMA parameter combinations
seasonal_period = 6
param_combinations = [
    ((1, 1, 1), (1, 1, 0, seasonal_period)),
    ((2, 1, 1), (1, 1, 1, seasonal_period)),
    ((1, 1, 2), (0, 1, 1, seasonal_period)),
    ((1, 1, 1), (1, 0, 1, seasonal_period)),
]
results = {}
# Fit each model and record the AIC values
for order, seasonal_order in param_combinations:
    try:
        # Fit SARIMA model
        model = SARIMAX(data['PT-TT'], order=order,
seasonal_order=seasonal_order).fit(maxiter=500, disp=False)
        # Save model and AIC value
        results[(order, seasonal_order)] = model.aic
        print(f"Model SARIMA{order}x{seasonal_order} - AIC:
{model.aic}")
    except Exception as e:
        print(f"Model SARIMA{order}x{seasonal_order} - Failed to fit. Error:
{e}")
# Find the parameter combination with the smallest AIC
if results:
    best_params = min(results, key=results.get)
    best_aic = results[best_params]
    print("\nBest Model Parameters:")
```

```
        print(f"Order: {best_params[0]}, Seasonal Order: {best_params[1]}, AIC: {best_aic}")
        # Fit the final model with the best parameters
        best_model = SARIMAX(data['PT-TT'], order=best_params[0], seasonal_order=best_params[1]).fit(maxiter=500, disp=False)
        # Print the summary of the best model
        print("\nBest Model Summary:")
        print(best_model.summary())
```

Model prediction

```
    # Use the best model to forecast the next 30 time points
    forecast_steps = 30
    forecast_original = best_model.forecast(steps=forecast_steps)
    # Create time index for the forecast data
    last_date_original = data.index[-1]
    forecast_index_original = pd.date_range(start=last_date_original, periods=forecast_steps + 1, freq='D')[1:]
    # Convert the forecast results to a one-dimensional array
    forecast_values_original = np.array(forecast_original).flatten()
    # Plot historical data and forecast results
    plt.figure(figsize=(10, 6))
    # Plot historical data
    plt.plot(data.index, data['PT-TT'], label="Historical Data", color='blue')
    # Plot forecast results
    plt.plot(forecast_index_original, forecast_values_original, label="ARIMA Forecast (Original Scale)", color='orange')
    # Chart settings
    plt.title('ARIMA Model - Short-Term Forecast (Original Data)')
    plt.xlabel('Time')
    plt.ylabel('PT-TT')
    plt.legend()
    plt.grid()
    plt.show()
```

LSTM training

```
    import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import LSTM, Dense
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from math import sqrt

# Load data
data = pd.read_excel('Attachment 2.xlsx', header=None, names=["MJD", "PT-TT"])

# Ensure PT-TT is numeric
data['PT-TT'] = pd.to_numeric(data['PT-TT'], errors='coerce')
```

```
# Drop rows with NaN values
data = data.dropna(subset=['MJD', 'PT-TT'])

# Normalize PT-TT data
scaler = MinMaxScaler(feature_range=(0, 1))
data_scaled = scaler.fit_transform(data['PT-TT'].values.reshape(-1, 1))

# Create X and y datasets for training the LSTM model
def create_dataset(data, time_step=1):
    X, y = [], []
    for i in range(len(data) - time_step - 1):
        X.append(data[i:(i + time_step), 0])  # Use the past 'time_step' days of data as features
        y.append(data[i + time_step, 0])      # Predict the next time step's value
    return np.array(X), np.array(y)

# Set time step (using past 300 days of data to predict the next day's value)
time_step = 300
X, y = create_dataset(data_scaled, time_step)

# Split into training and test sets
train_size = int(len(X) * 0.8)
X_train, X_test = X[:train_size], X[train_size:]
y_train, y_test = y[:train_size], y[train_size:]

# Reshape data for LSTM: [samples, time_step, features]
X_train = X_train.reshape(X_train.shape[0], X_train.shape[1], 1)
X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], 1)

# Build LSTM model
model = Sequential()
model.add(LSTM(units=50, return_sequences=True, input_shape=(time_step, 1)))
model.add(LSTM(units=50, return_sequences=False))
model.add(Dense(units=1))  # Output a single value
# Compile and train the model
model.compile(optimizer='adam', loss='mean_squared_error')
# Train LSTM model
model.fit(X_train, y_train, epochs=20, batch_size=32)
# Perform long-term prediction
predicted_noise = model.predict(X_test)
# Inverse transform to restore original PT-TT scale
predicted_noise = scaler.inverse_transform(predicted_noise)
y_test = scaler.inverse_transform(y_test.reshape(-1, 1))
# Plot prediction results
plt.figure(figsize=(10, 6))
plt.plot(y_test, color='#a0b4ef', label='True Noise')
plt.plot(predicted_noise, color='#e69695', label='Predicted Noise')
plt.title('LSTM Model Long-Term Forecast')
plt.xlabel('Time')
plt.ylabel('PT-TT (s)')
```

```
plt.legend()
plt.grid(True)
plt.show()
# Compute evaluation metrics
# Calculate Mean Squared Error (MSE)
mse = mean_squared_error(y_test, predicted_noise)
print(f'Mean Squared Error (MSE): {mse}')
# Calculate Root Mean Squared Error (RMSE)
rmse = sqrt(mse)
print(f'Root Mean Squared Error (RMSE): {rmse}')
# Calculate Mean Absolute Error (MAE)
mae = mean_absolute_error(y_test, predicted_noise)
print(f'Mean Absolute Error (MAE): {mae}')
# Calculate R² (R-squared)
r2 = r2_score(y_test, predicted_noise)
print(f'R² (R-squared): {r2}')
```

LSTM

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import LSTM, Dense
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from datetime import timedelta
from math import sqrt
# 1. Data Loading and Preprocessing
# Assuming the data is saved in an Excel file with columns MJD and PT-TT
data = pd.read_excel('Attachment 2.xlsx', header=None, names=["MJD", "PT-TT"])
# Ensure PT-TT is numeric
data['PT-TT'] = pd.to_numeric(data['PT-TT'], errors='coerce')
# Remove rows with NaN values
data = data.dropna(subset=['MJD', 'PT-TT'])
# Convert MJD to date
base_date = pd.Timestamp('1858-11-17')
data['MJD_date'] = base_date + pd.to_timedelta(data['MJD'], unit='D')
# Ensure MJD_date is in datetime format
data['MJD_date'] = pd.to_datetime(data['MJD_date'], errors='coerce')
# Data Normalization: Use MinMaxScaler to scale PT-TT to a range between 0 and 1
scaler = MinMaxScaler(feature_range=(0, 1))
data_scaled = scaler.fit_transform(data['PT-TT'].values.reshape(-1, 1))
# 2. Create Time Series Dataset for LSTM Model Training
def create_dataset(data, time_step=1):
    X, y = [], []
    for i in range(len(data) - time_step - 1):
        X.append(data[i:(i + time_step), 0])  # Take the previous 'time_step' days as features
```

```
            y.append(data[i + time_step, 0])          # Predict the next day's value
        return np.array(X), np.array(y)
    # Set time step (use data from the past 300 days to predict the next day)
    time_step = 300
    X, y = create_dataset(data_scaled, time_step)
    # Split into training and testing sets, 80% for training and 20% for testing
    train_size = int(len(X) * 0.8)
    X_train, X_test = X[:train_size], X[train_size:]
    y_train, y_test = y[:train_size], y[train_size:]
    # Reshape data to the required format for LSTM: [samples, time_step, features]
    X_train = X_train.reshape(X_train.shape[0], X_train.shape[1], 1)
    X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], 1)
    # 3. Build LSTM Model
    model = Sequential()
    model.add(LSTM(units=50, return_sequences=True, input_shape=(time_step, 1)))
    model.add(LSTM(units=50, return_sequences=False))
    model.add(Dense(units=1))  # Output one value
    # Compile and train the model
    model.compile(optimizer='adam', loss='mean_squared_error')
    # Train the LSTM model
    model.fit(X_train, y_train, epochs=20, batch_size=32)
    # 4. Long-term Prediction: Forecast the next year of PT-TT (365 days)
    future_steps = 365  # Predict one year (in days)
    # Select the last data point from the test set as the starting point for rolling prediction
    last_input = X_test[-1]  # Get the last sample from the test data
    last_input = last_input.reshape(1, time_step, 1)  # Reshape it to the LSTM input format
    # List to store future predictions
    future_predictions = []
    # Perform rolling prediction
    for i in range(future_steps):
        # Predict the next PT-TT value
        predicted_value = model.predict(last_input)
        # Add the predicted value to the prediction list
        future_predictions.append(predicted_value[0, 0])
        # Update the input data with the new prediction for the next time step
        last_input = np.append(last_input[:, 1:, :], predicted_value.reshape(1, 1, 1), axis=1)
    # Inverse transform the predictions to restore the original PT-TT scale
    future_predictions = scaler.inverse_transform(np.array(future_predictions).reshape(-1, 1))
    # Generate a time series for the next year (starting from the last MJD date)
    last_date = data['MJD_date'].iloc[-1]
    future_dates = [last_date + timedelta(days=i) for i in range(1, future_steps + 1)]
    # 5. Plot the Future Predictions
    plt.figure(figsize=(10, 6))
    plt.plot(data['MJD_date'], data['PT-TT'], color='blue', label='Historical Data')
    plt.plot(future_dates, future_predictions, color='red', label='Predicted Future
```

```
Noise')
    plt.title('Pulsar Timing Residuals Forecast (1 Year)')
    plt.xlabel('Date')
    plt.ylabel('PT-TT (s)')
    plt.legend()
    plt.grid(True)
    plt.show()
    # 6. Calculate Evaluation Metrics
    # Calculate Mean Squared Error (MSE)
    mse = mean_squared_error(y_test, scaler.inverse_transform(model.predict(X_test)))
    print(f'Mean Squared Error (MSE): {mse}')
    # Calculate Root Mean Squared Error (RMSE)
    rmse = sqrt(mse)
    print(f'Root Mean Squared Error (RMSE): {rmse}')
    # Calculate Mean Absolute Error (MAE)
    mae = mean_absolute_error(y_test, scaler.inverse_transform(model.predict(X_test)))
    print(f'Mean Absolute Error (MAE): {mae}')
    # Calculate R²  (Coefficient of Determination)
    r2 = r2_score(y_test, scaler.inverse_transform(model.predict(X_test)))
    print(f'R²  (R-squared): {r2}')
```

Model Comparison

```
    import pandas as pd
    import numpy as np
    import matplotlib.pyplot as plt
    from sklearn.preprocessing import MinMaxScaler
    from keras.models import Sequential
    from keras.layers import LSTM, GRU, Dense, Flatten
    from sklearn.ensemble import RandomForestRegressor
    from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
    from math import sqrt
    # Load data
    data = pd.read_excel('Attachment 2.xlsx', header=None, names=["MJD", "PT-TT"])
    # Ensure PT-TT is numeric
    data['PT-TT'] = pd.to_numeric(data['PT-TT'], errors='coerce')
    # Drop rows with NaN values
    data = data.dropna(subset=['MJD', 'PT-TT'])
    # Normalize PT-TT data
    scaler = MinMaxScaler(feature_range=(0, 1))
    data_scaled = scaler.fit_transform(data['PT-TT'].values.reshape(-1, 1))
    # Create X and y datasets for training the model
    def create_dataset(data, time_step=1):
        X, y = [], []
        for i in range(len(data) - time_step - 1):
            X.append(data[i:(i + time_step), 0])  # Use past `time_step` days of data as features
            y.append(data[i + time_step, 0])      # Predict the next value in the
```

```
time series
            return np.array(X), np.array(y)
    # Set time step (using the past 300 days of data to predict the next day)
    time_step = 300
    X, y = create_dataset(data_scaled, time_step)
    # Split into training and testing datasets
    train_size = int(len(X) * 0.8)
    X_train, X_test = X[:train_size], X[train_size:]
    y_train, y_test = y[:train_size], y[train_size:]
    # Reshape data for LSTM/GRU (samples, time_steps, features)
    X_train_lstm = X_train.reshape(X_train.shape[0], X_train.shape[1], 1)
    X_test_lstm = X_test.reshape(X_test.shape[0], X_test.shape[1], 1)
    # LSTM Model
    lstm_model = Sequential()
    lstm_model.add(LSTM(units=50, return_sequences=True,
input_shape=(time_step, 1)))
    lstm_model.add(LSTM(units=50, return_sequences=False))
    lstm_model.add(Dense(units=1))
    lstm_model.compile(optimizer='adam', loss='mean_squared_error')
    lstm_model.fit(X_train_lstm, y_train, epochs=20, batch_size=32)
    lstm_predicted = lstm_model.predict(X_test_lstm)
    lstm_predicted = scaler.inverse_transform(lstm_predicted)
    # GRU Model
    gru_model = Sequential()
    gru_model.add(GRU(units=50, return_sequences=True, input_shape=(time_step,
1)))
    gru_model.add(GRU(units=50, return_sequences=False))
    gru_model.add(Dense(units=1))
    gru_model.compile(optimizer='adam', loss='mean_squared_error')
    gru_model.fit(X_train_lstm, y_train, epochs=20, batch_size=32)
    gru_predicted = gru_model.predict(X_test_lstm)
    gru_predicted = scaler.inverse_transform(gru_predicted)
    # MLP Model
    X_train_flat = X_train.reshape(X_train.shape[0], -1)
    X_test_flat = X_test.reshape(X_test.shape[0], -1)
    mlp_model = Sequential()
    mlp_model.add(Dense(units=128, activation='relu',
input_dim=X_train_flat.shape[1]))
    mlp_model.add(Dense(units=64, activation='relu'))
    mlp_model.add(Dense(units=1))
    mlp_model.compile(optimizer='adam', loss='mean_squared_error')
    mlp_model.fit(X_train_flat, y_train, epochs=20, batch_size=32)
    mlp_predicted = mlp_model.predict(X_test_flat)
    mlp_predicted = scaler.inverse_transform(mlp_predicted)
    # Random Forest Model
    rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
    rf_model.fit(X_train_flat, y_train)
    rf_predicted = rf_model.predict(X_test_flat)
    rf_predicted = scaler.inverse_transform(rf_predicted.reshape(-1, 1))
    # Evaluation function
```

```
    def evaluate_model(y_true, y_pred, model_name):
        mse = mean_squared_error(y_true, y_pred)
        rmse = sqrt(mse)
        mae = mean_absolute_error(y_true, y_pred)
        r2 = r2_score(y_true, y_pred)
        print(f"{model_name} - MSE: {mse}, RMSE: {rmse}, MAE: {mae}, R²: {r2}")
        return mse, rmse, mae, r2
    # Evaluate all models
    y_test = scaler.inverse_transform(y_test.reshape(-1, 1))
    evaluate_model(y_test, lstm_predicted, "LSTM")
    evaluate_model(y_test, gru_predicted, "GRU")
    evaluate_model(y_test, mlp_predicted, "MLP")
    evaluate_model(y_test, rf_predicted, "Random Forest")
    # Plot predictions
    plt.figure(figsize=(12, 8))
    plt.plot(y_test, color='#a0b4ef', label='True Noise')
    plt.plot(lstm_predicted, color='#e69695', label='LSTM Prediction')
    plt.plot(gru_predicted, color='#95e6c8', label='GRU Prediction')
    plt.plot(mlp_predicted, color='#e6d895', label='MLP Prediction')
    plt.plot(rf_predicted, color='#95aee6', label='Random Forest Prediction')
    plt.title('Model Comparison on Long-Term Forecast')
    plt.xlabel('Time')
    plt.ylabel('PT-TT (s)')
    plt.legend()
    plt.grid(True)
    plt.show()
```

Problem 3

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

# Set English font (optional)
plt.rcParams['font.sans-serif'] = ['Arial']
plt.rcParams['axes.unicode_minus'] = False

# Observation condition parameters
zenith_delay_target = 7.69e-9    # Target zenith delay in seconds
TEC = 10.0    # Total Electron Content (TEC), in TECU
pressure_hPa = 1013    # Atmospheric pressure in hPa
temperature_K = 273.15    # Temperature in Kelvin (0°C)
elevation_angle = 10    # Elevation angle in degrees
RH = 50    # Relative humidity in percentage
height_m = 0    # Observation point height in meters

# Saastamoinen model for calculating tropospheric delay
def saastamoinen_troposphere_delay(pressure, temperature, elevation_angle):
    zenith_angle = np.radians(90 - elevation_angle)  # Convert to zenith angle (radians)
    delay_zenith = (pressure / 1013) * (273.15 / temperature)
```

```
    mapping_function = 1 / np.cos(zenith_angle)  # Mapping function approximated by sec(z)
    return delay_zenith * mapping_function * 1e-9  # Convert to seconds

# Ionospheric dispersion delay calculation
def ionosphere_dispersion_delay(frequency, TEC):
    return TEC / (frequency ** 2)

# Height correction for atmospheric pressure
def corrected_pressure(base_pressure, height):
    scale_height = 8500  # Atmospheric scale height, in meters
    return base_pressure * np.exp(-height / scale_height)

# Calculate total delay
def total_delay(frequency, pressure, temperature, elevation_angle, TEC, RH, height):
    corrected_p = corrected_pressure(pressure, height)
    troposphere = saastamoinen_troposphere_delay(corrected_p, temperature, elevation_angle)
    dispersion = ionosphere_dispersion_delay(frequency, TEC)
    return troposphere + dispersion
# Frequency range setup
frequencies = np.linspace(20e9, 40e9, 500)  # Frequency range from 20 GHz to 40 GHz in Hz
# Calculate tropospheric, ionospheric, and total delays
troposphere_delays = [saastamoinen_troposphere_delay(pressure_hPa, temperature_K, elevation_angle) for _ in frequencies]
ionosphere_delays = [ionosphere_dispersion_delay(f, TEC) for f in frequencies]
total_delays = [tropo + iono for tropo, iono in zip(troposphere_delays, ionosphere_delays)]
# Plotting
plt.figure(figsize=(10, 6))
plt.plot(frequencies / 1e9, np.array(troposphere_delays) * 1e9, label="Tropospheric Delay (ns)", color="green", linestyle="--", linewidth=2.5)
plt.plot(frequencies / 1e9, np.array(ionosphere_delays) * 1e9, label="Ionospheric Delay (ns)", color="orange")
plt.plot(frequencies / 1e9, np.array(total_delays) * 1e9, label="Total Delay (ns)", color="blue")
plt.axhline(y=zenith_delay_target * 1e9, color="red", linestyle="--", label="Target Delay (7.69 ns)")
# Add labels and title
plt.xlabel("Frequency (GHz)")
plt.ylabel("Delay (ns)")
plt.title("Frequency vs Delay Relationship")
plt.legend()
plt.grid(True)
plt.show()
# Output delay result at a specific frequency
frequency_hz = 21.0e9  # Single frequency at 21 GHz
zenith_delay = total_delay(frequency_hz, pressure_hPa, temperature_K, elevation_angle, TEC, RH, height_m)
```

```
print(f"Zenith delay at {frequency_hz / 1e9} GHz: {zenith_delay * 1e9:.4f} ns")
```

Problem 4

```
    import numpy as np
import matplotlib.pyplot as plt
# Set English font (optional)
plt.rcParams['font.sans-serif'] = ['Arial']
plt.rcParams['axes.unicode_minus'] = False
# Observation condition parameters
TEC = 10.0  # Total Electron Content (TEC), in TECU
pressure_hPa = 1013  # Atmospheric pressure in hPa
temperature_K = 288.15  # Temperature in Kelvin (15°C)
RH = 50  # Relative humidity in percentage
height_m = 0  # Observation point height in meters
# Improved Saastamoinen model for calculating tropospheric delay
def saastamoinen_troposphere_delay(pressure, temperature, elevation_angle):
    zenith_angle = np.radians(90 - elevation_angle)  # Convert to zenith angle (radians)
    delay_zenith = (0.0022768 * pressure) / (1 + 0.0026 * np.cos(2 * zenith_angle) + 0.00028 * temperature)
    # Improved mapping function for low elevation angles, with a safe limit to avoid overflows
    mapping_function = 1 / (
                np.cos(zenith_angle) + 0.001)  # Adding a small value to avoid division by very small numbers
    return delay_zenith * mapping_function * 1e-9  # Convert to seconds
# Ionospheric dispersion delay calculation (with a modification to scale for high frequency)
def ionosphere_dispersion_delay(frequency, TEC):
    # For high frequencies (> 20 GHz), ionospheric delay is negligible; modify calculation appropriately.
    if frequency > 20e9:
        return TEC / (frequency ** 2) * 1e16 * 0.01  # Scaling down for high frequencies
    else:
        return TEC / (frequency ** 2) * 1e16
# Height correction for atmospheric pressure
def corrected_pressure(base_pressure, height):
    scale_height = 8500  # Atmospheric scale height, in meters
    return base_pressure * np.exp(-height / scale_height)
# Calculate total delay
def total_delay(frequency, pressure, temperature, elevation_angle, TEC, height):
    corrected_p = corrected_pressure(pressure, height)
    troposphere = saastamoinen_troposphere_delay(corrected_p, temperature, elevation_angle)
    dispersion = ionosphere_dispersion_delay(frequency, TEC)
    return troposphere + dispersion
# Set frequency and elevation angles
frequency_hz = 30e9  # 30 GHz (a representative high frequency)
elevation_angles = np.linspace(1, 10, 100)  # Elevation angle from 1° to 10°
```

```
# Calculate delays for each elevation angle
troposphere_delays = [saastamoinen_troposphere_delay(pressure_hPa, temperature_K, angle) for angle in elevation_angles]
ionosphere_delays = [ionosphere_dispersion_delay(frequency_hz, TEC) for _ in elevation_angles]
total_delays = [tropo + iono for tropo, iono in zip(troposphere_delays, ionosphere_delays)]

# Plotting the results
plt.figure(figsize=(10, 6))
plt.plot(elevation_angles, np.array(troposphere_delays) * 1e9, label="Tropospheric Delay (ns)", color="green",
         linestyle="--", linewidth=2.5)
plt.plot(elevation_angles, np.array(ionosphere_delays) * 1e9, label="Ionospheric Delay (ns)", color="orange")
plt.plot(elevation_angles, np.array(total_delays) * 1e9, label="Total Delay (ns)", color="blue")

# Add labels and title
plt.xlabel("Elevation Angle (degrees)")
plt.ylabel("Delay (ns)")
plt.title("Elevation Angle vs Atmospheric Delay")
plt.legend()
plt.grid(True)
plt.show()

# Output delay result at a specific elevation angle
elevation_angle_deg = 10.0  # Specific elevation angle at 10 degrees
zenith_delay = total_delay(frequency_hz, pressure_hPa, temperature_K, elevation_angle_deg, TEC, height_m)
print(
    f"Total delay at elevation angle {elevation_angle_deg}° and {frequency_hz / 1e9} GHz: {zenith_delay * 1e9:.4f} ns")
```

Sensitivity Analysis_LSTM

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
from keras.models import Sequential
from keras.layers import LSTM, Dense
from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_score
from math import sqrt
from keras.callbacks import EarlyStopping
from joblib import Parallel, delayed
# Data loading and preprocessing
data = pd.read_excel('Attachment 2.xlsx', header=None, names=["MJD", "PT-TT"])
```

```
    data['PT-TT'] = pd.to_numeric(data['PT-TT'], errors='coerce')
    data = data.dropna(subset=['MJD', 'PT-TT'])
    # Data normalization
    scaler = MinMaxScaler(feature_range=(0, 1))
    data_scaled = scaler.fit_transform(data['PT-TT'].values.reshape(-1, 1))
    # Create time series dataset
    def create_dataset(data, time_step=1):
        X, y = [], []
        for i in range(len(data) - time_step - 1):
            X.append(data[i:(i + time_step), 0])
            y.append(data[i + time_step, 0])
        return np.array(X), np.array(y)
    # Model training and evaluation function
    def train_and_evaluate_lstm(time_step, units, batch_size, epochs):
        # Create dataset
        X, y = create_dataset(data_scaled, time_step)
        train_size = int(len(X) * 0.8)
        X_train, X_test = X[:train_size], X[train_size:]
        y_train, y_test = y[:train_size], y[train_size:]
        X_train = X_train.reshape(X_train.shape[0], X_train.shape[1], 1)
        X_test = X_test.reshape(X_test.shape[0], X_test.shape[1], 1)
        # Build LSTM model
        model = Sequential()
        model.add(LSTM(units=units, return_sequences=True,
input_shape=(time_step, 1)))
        model.add(LSTM(units=units, return_sequences=False))
        model.add(Dense(units=1))
        model.compile(optimizer='adam', loss='mean_squared_error')
        # Use early stopping to prevent overfitting
        early_stop = EarlyStopping(monitor='val_loss', patience=5,
restore_best_weights=True)
        # Train the model
        model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size,
verbose=0, validation_data=(X_test, y_test), callbacks=[early_stop])
        # Predictions
        predicted_noise = model.predict(X_test)
        predicted_noise = scaler.inverse_transform(predicted_noise)
        y_test = scaler.inverse_transform(y_test.reshape(-1, 1))
        # Calculate evaluation metrics
        mse = mean_squared_error(y_test, predicted_noise)
        rmse = sqrt(mse)
        mae = mean_absolute_error(y_test, predicted_noise)
        r2 = r2_score(y_test, predicted_noise)
        return {"time_step": time_step, "units": units, "batch_size": batch_size,
"epochs": epochs,
                "MSE": mse, "RMSE": rmse, "MAE": mae, "R² ": r2}
    # Parameter list: testing different time steps, hidden units, batch sizes, and epochs
    time_steps = [50, 100, 300]
    units_list = [30, 50]
    batch_sizes = [16, 32]
```

```
    epochs_list = [10, 20]
    # Use parallelization to speed up
    results = Parallel(n_jobs=-1)(delayed(train_and_evaluate_lstm)(time_step, units, batch_size, epochs)
                                  for time_step in time_steps
                                  for units in units_list
                                  for batch_size in batch_sizes
                                  for epochs in epochs_list)
    # Convert results to DataFrame for analysis
    results_df = pd.DataFrame(results)
    # Save results to Excel
    results_df.to_excel("LSTM_model_results.xlsx", index=False)
    # Visualize the sensitivity analysis results
    plt.figure(figsize=(12, 8))
    # Plot MSE, RMSE, MAE, and R²
    for metric in ["MSE", "RMSE", "MAE", "R² "]:
        plt.plot(results_df.index, results_df[metric], label=metric, marker='o')
    # Set chart title and labels
    plt.xlabel("Experiment Index", fontsize=12)
    plt.ylabel("Metric Value", fontsize=12)
    plt.title("LSTM Sensitivity Analysis: MSE, RMSE, MAE", fontsize=14)
    # Add legend
    plt.legend()
    # Add grid
    plt.grid(True)
    # Display the plot
    plt.show()
```

sensitivity_analysis_Problem 3

```
    import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Set plot font settings (optional)
plt.rcParams['axes.unicode_minus'] = False

# Observation condition parameters
zenith_delay_target = 7.69e-9  # Target zenith delay in seconds
TEC = 10.0  # Total Electron Content (TEC), in TECU
pressure_hPa = 1013  # Atmospheric pressure in hPa
temperature_K = 273.15  # Temperature in Kelvin (0°C)
elevation_angle = 10  # Elevation angle in degrees
RH = 50  # Relative humidity in percentage
height_m = 0  # Observation point height in meters

# Saastamoinen model for calculating tropospheric delay
def saastamoinen_troposphere_delay(pressure, temperature, elevation_angle):
    zenith_angle = np.radians(90 - elevation_angle)  # Convert to zenith angle (radians)
    delay_zenith = (pressure / 1013) * (273.15 / temperature)
```

```
    mapping_function = 1 / np.cos(zenith_angle)  # Mapping function approximated by sec(z)
    return delay_zenith * mapping_function * 1e-9  # Convert to seconds

# Ionospheric dispersion delay calculation
def ionosphere_dispersion_delay(frequency, TEC):
    return TEC / (frequency ** 2)

# Height correction for atmospheric pressure
def corrected_pressure(base_pressure, height):
    scale_height = 8500  # Atmospheric scale height, in meters
    return base_pressure * np.exp(-height / scale_height)

# Calculate total delay
def total_delay(frequency, pressure, temperature, elevation_angle, TEC, RH, height):
    corrected_p = corrected_pressure(pressure, height)
    troposphere = saastamoinen_troposphere_delay(corrected_p, temperature, elevation_angle)
    dispersion = ionosphere_dispersion_delay(frequency, TEC)
    return troposphere + dispersion

# Frequency range setup for plotting
frequencies = np.linspace(20e9, 40e9, 500)  # Frequency range from 20 GHz to 40 GHz in Hz

# Sensitivity analysis parameters for different elevation angles, TECs, pressures, and frequencies
elevation_angles = np.linspace(5, 45, 5)  # Elevation angle in degrees
TECs = np.linspace(5, 20, 4)  # Total Electron Content (TECU)
pressures = np.linspace(900, 1100, 4)  # Atmospheric pressure in hPa
frequencies_to_test = [21.0e9, 30.0e9, 40.0e9]  # Specific frequencies for analysis

# Sensitivity results
sensitivity_results = []

# Perform sensitivity analysis
for elevation_angle in elevation_angles:
    for TEC in TECs:
        for pressure in pressures:
            for frequency in frequencies_to_test:
                zenith_delay = total_delay(frequency, pressure, temperature_K, elevation_angle, TEC, RH, height_m)
                sensitivity_results.append({
                    "Elevation Angle (°)": elevation_angle,
                    "TEC (TECU)": TEC,
                    "Pressure (hPa)": pressure,
                    "Frequency (GHz)": frequency / 1e9,
                    "Zenith Delay (ns)": zenith_delay * 1e9  # Convert to nanoseconds
                })
```

```
# Convert results to DataFrame for easier analysis
sensitivity_df = pd.DataFrame(sensitivity_results)

# Display the sensitivity analysis results in Jupyter notebook (optional)
print(sensitivity_df.head())  # Display first few rows for quick inspection

# Save the DataFrame to a CSV file
sensitivity_df.to_csv("zenith_delay_sensitivity_analysis.csv", index=False)

# Plotting: Sensitivity Analysis at Different Elevation Angles
plt.figure(figsize=(12, 8))
for elevation_angle in elevation_angles:
    delays = [total_delay(freq, pressure_hPa, temperature_K, elevation_angle, TEC, RH, height_m) * 1e9 for freq in frequencies]
    plt.plot(frequencies / 1e9, delays, label=f"Elevation Angle {elevation_angle}°")

plt.xlabel("Frequency (GHz)")
plt.ylabel("Zenith Delay (ns)")
plt.title("Sensitivity Analysis: Zenith Delay vs Frequency at Different Elevation Angles")
plt.legend(title="Elevation Angles")
plt.grid(True)
plt.show()

# Plotting: Sensitivity Analysis at Different TEC Levels
plt.figure(figsize=(12, 8))
for TEC in TECs:
    delays = [total_delay(freq, pressure_hPa, temperature_K, elevation_angle, TEC, RH, height_m) * 1e9 for freq in frequencies]
    plt.plot(frequencies / 1e9, delays, label=f"TEC {TEC} TECU")

plt.xlabel("Frequency (GHz)")
plt.ylabel("Zenith Delay (ns)")
plt.title("Sensitivity Analysis: Zenith Delay vs Frequency at Different TEC Levels")
plt.legend(title="Total Electron Content (TECU)")
plt.grid(True)
plt.show()
```

sensitivity_analysis_Problem 4

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy.interpolate import make_interp_spline, interp1d
# Set plot font settings (optional)
plt.rcParams['font.sans-serif'] = ['Arial']
plt.rcParams['axes.unicode_minus'] = False
# Observation condition parameters
pressure_hPa = 1013  # Atmospheric pressure in hPa
temperature_K = 288.15  # Temperature in Kelvin (15°C)
```

```
height_m = 0  # Observation point height in meters
# Improved Saastamoinen model for calculating tropospheric delay
def saastamoinen_troposphere_delay(pressure, temperature, elevation_angle):
    zenith_angle = np.radians(90 - elevation_angle)  # Convert to zenith angle (radians)
    delay_zenith = (0.0022768 * pressure) / (1 + 0.0026 * np.cos(2 * zenith_angle) + 0.00028 * temperature)
    mapping_function = 1 / (np.cos(zenith_angle) + 0.01)  # Further adjust mapping function to avoid sudden jumps
    return delay_zenith * mapping_function * 1e-9  # Convert to seconds
# Ionospheric dispersion delay calculation (with a modification to scale for high frequency)
def ionosphere_dispersion_delay(frequency, TEC):
    if frequency > 25e9:
        return TEC / (frequency ** 2) * 1e16 * 0.0005  # Further reduced scaling for high frequencies to smooth transition
    elif frequency > 20e9:
        return TEC / (frequency ** 2) * 1e16 * 0.0008  # Smoother scaling for mid frequencies
    else:
        return TEC / (frequency ** 2) * 1e16 * 0.004  # Further smoothing for lower frequencies
# Height correction for atmospheric pressure
def corrected_pressure(base_pressure, height):
    scale_height = 8500  # Atmospheric scale height, in meters
    return base_pressure * np.exp(-height / scale_height)
# Calculate total delay
def total_delay(frequency, pressure, temperature, elevation_angle, TEC, height):
    corrected_p = corrected_pressure(pressure, height)
    troposphere = saastamoinen_troposphere_delay(corrected_p, temperature, elevation_angle)
    dispersion = ionosphere_dispersion_delay(frequency, TEC)
    return troposphere + dispersion
# Sensitivity analysis parameters
elevation_angles = np.linspace(5, 45, 5)  # Elevation angles from 5° to 45°
frequencies = np.linspace(20e9, 40e9, 200)  # Frequency range from 20 GHz to 40 GHz with more points for smoothness
TECs = np.linspace(5, 20, 4)  # Total Electron Content (TECU)
pressures = np.linspace(900, 1100, 4)  # Atmospheric pressure from 900 to 1100 hPa
colors = ['#a6d3ff', '#fccccb', '#bdb5e1', '#b0d992', '#e69695', '#f3adbc']  # Custom colors
# Sensitivity results
sensitivity_results = []

# Perform sensitivity analysis
for elevation_angle in elevation_angles:
    for TEC in TECs:
        for pressure in pressures:
            for frequency in frequencies:
                zenith_delay = total_delay(frequency, pressure, temperature_K,
```

```
elevation_angle, TEC, height_m)
                    sensitivity_results.append({
                        "Elevation Angle (degrees)": elevation_angle,
                        "TEC (TECU)": TEC,
                        "Pressure (hPa)": pressure,
                        "Frequency (GHz)": frequency / 1e9,
                        "Zenith Delay (ns)": zenith_delay * 1e9  # Convert to nanoseconds
                    })
# Convert results to DataFrame
sensitivity_df = pd.DataFrame(sensitivity_results)

(sensitivity_df.to_excel("Problem 4_results.xlsx", index=False))

# Display the sensitivity analysis results in tabular format (optional)
print(sensitivity_df.head())

# Plotting the sensitivity analysis results with smoother curves
plt.figure(figsize=(12, 8))
for idx, elevation_angle in enumerate(elevation_angles):
    delays = sensitivity_df[sensitivity_df["Elevation Angle (degrees)"] == elevation_angle]
    delays = delays.sort_values(by="Frequency (GHz)").drop_duplicates(subset="Frequency (GHz)")  # Ensure data is sorted by frequency and remove duplicates
    frequencies_plot = delays["Frequency (GHz)"].values
    zenith_delays_plot = delays["Zenith Delay (ns)"].values
    # Use linear interpolation for smoother curves to reduce oscillations
    if len(frequencies_plot) > 3:  # Ensure enough points for interpolation
        linear_interp = interp1d(frequencies_plot, zenith_delays_plot, kind='linear')
        frequencies_smooth = np.linspace(frequencies_plot.min(), frequencies_plot.max(), 300)
        zenith_delays_smooth = linear_interp(frequencies_smooth)
        plt.plot(frequencies_smooth, zenith_delays_smooth, label=f"Elevation Angle {elevation_angle}°", color=colors[idx % len(colors)])
    else:
        plt.plot(frequencies_plot, zenith_delays_plot, label=f"Elevation Angle {elevation_angle}°", color=colors[idx % len(colors)])

plt.xlabel("Frequency (GHz)")
plt.ylabel("Zenith Delay (ns)")
plt.title("Sensitivity Analysis: Zenith Delay vs Frequency for Different Elevation Angles")
plt.legend(title="Elevation Angles")
plt.grid(True)
plt.show()

# Plotting: Sensitivity Analysis at Different TEC Levels with smoother curves
plt.figure(figsize=(12, 8))
for idx, TEC in enumerate(TECs):
```

```
    delays = sensitivity_df[sensitivity_df["TEC (TECU)"] == TEC]
    delays = delays.sort_values(by="Frequency (GHz)").drop_duplicates(subset="Frequency (GHz)")  # Ensure data is sorted by frequency and remove duplicates
    frequencies_plot = delays["Frequency (GHz)"].values
    zenith_delays_plot = delays["Zenith Delay (ns)"].values
    # Use linear interpolation for smoother curves to reduce oscillations
    if len(frequencies_plot) > 3:  # Ensure enough points for interpolation
        linear_interp = interp1d(frequencies_plot, zenith_delays_plot, kind='linear')
        frequencies_smooth = np.linspace(frequencies_plot.min(), frequencies_plot.max(), 300)
        zenith_delays_smooth = linear_interp(frequencies_smooth)
        plt.plot(frequencies_smooth, zenith_delays_smooth, label=f"TEC {TEC} TECU", color=colors[idx % len(colors)])
    else:
        plt.plot(frequencies_plot, zenith_delays_plot, label=f"TEC {TEC} TECU", color=colors[idx % len(colors)])
plt.xlabel("Frequency (GHz)")
plt.ylabel("Zenith Delay (ns)")
plt.title("Sensitivity Analysis: Zenith Delay vs Frequency for Different TEC Levels")
plt.legend(title="Total Electron Content (TECU)")
plt.grid(True)
plt.show()
```