POLITECNICO DI TORINO

Master's Degree in Communication and Computer Networks Engineering

Master's Degree Thesis

Simulation Framework for Earthquake Early Warning Using Optical Fiber Networks

Supervisors

Prof. Vittorio CURRI

Prof. Emanuele VIRGILLITO

Prof. Roberto PROIETTI

Candidate

Federico NOTARSTEFANO

October 2024

Summary

This thesis presents the development of a simulation framework for optical fiber networks used in earthquake early warning systems (EEWS). The primary objective is to improve the detection and characterization of seismic events to provide timely alerts that can help safeguard cities before the earthquake's impact. The simulation is centered around the propagation of seismic-induced strain through optical fibers and the analysis of changes in the state of polarization (SOP) of light signals within the fiber network.

The proposed work integrates a new model that combines an accurate strain propagation mechanism along the fiber lines with a *waveplate model* to account for the effects of *birefringence* and polarization changes. The network is divided into Nodes and Lines, each line is then divided into segments undergoing their own unique stress evolution due to the seismic waves. The stress evolution is then used to compute the SOP evolution at the end of each Line. The segment abstraction is based on the assumption that the earthquake induces a uniform effect on each segment of the fiber. This means that within each segment, the strain measured from the waveplates is considered to be consistent. This abstraction simplifies the complex interactions within the fiber, allowing for a more manageable analysis of the SOP changes.

A relevant component of the framework is the model calculating of the SOP changes across all segments, which considers the accumulation of polarization changes along the entire fiber length. A core element of the framework is the Simulation Manager, which orchestrates the entire process of seismic event simulation, strain application, and SOP analysis. It also manages the data flow, from preprocessing seismic inputs to the final handling of simulation results, ensuring robust and error-resilient operations.

The time base is unified to synchronize the detection and propagation effects of the earthquake across different parts of the network. This enables the system to measure the impact of the seismic wave as it reaches different segments and to adjust the SOP accordingly. An important metric used in the framework is the SOP Angular Speed (SOPAS), which quantifies the rate of change in polarization due to seismic-induced strain. The analysis of SOPAS offers key insights into the

dynamics of polarization variation, further aiding in early seismic detection.

The developed simulation measures the feasibility of using optical fiber networks for real-time monitoring and early warning of seismic events. The integration of real-time seismic data through the Syngine platform further enhances the simulation's realism, providing detailed strain information based on real-world earthquake parameters. This allows the framework to simulate the behavior of the optical network during actual seismic events with the realistic precision.

By observing the changes in polarization at both the start and end of each fiber line, the system can provide critical seconds of warning that are vital for implementing safety measures. This research contributes to the field of earthquake early warning systems by providing a more detailed understanding of how seismic waves affect optical fibers, paving the way for the deployment of rapid and effective responses to ensure people safety in the event of an earthquake.

Acknowledgements

I would like to express my gratitude to Professor Vittorio Curri for the invaluable opportunity he gave me and for his guidance throughout this project. I would also like to thank Prof. Emanuele Virgillito for his support and his readiness to assist me with this work. Finally, I would like to thank the PLANET team for their collaboration during this experience.

A special thanks goes to my family, Fabrizio, and my friends, whose love and support I always cherish.

Federico

Table of Contents

List of Tables

List of Figures

Acronyms

EEWS
Earthquake Early Warning Systems

SOP
State of Polarization

SOP
State of Polarization Angular Speed

LSTM
Long Short-Term Memory

DAS
Distributed Acoustic Sensing

WP
Wave Plate

Chapter 1

Introduction

1.1 Context

In recent years, the need for more effective earthquake early warning systems has become increasingly important due to the potential devastation caused by seismic events [1]. Traditional seismometers, while accurate, often suffer from limitations in terms of coverage and response time, particularly in densely populated urban areas. To address these challenges, researchers have explored alternative methods for seismic detection that leverage existing infrastructure, one of the most promising being the use of optical fiber networks [2].

Optical fibers, which are already deployed extensively for telecommunications, have shown great potential as seismic sensors due to their sensitivity to strain and environmental changes. By monitoring variations in the *state of polarization* (SOP) of light signals transmitted through these fibers, it is possible to detect the ground motion induced by seismic waves [3]. This approach not only provides a scalable solution but also reduces the cost and complexity of implementing new earthquake early warning systems.

The combination of seismic wave propagation models and optical fiber sensing technology can enhance the precision of early warning systems. By measuring the strain induced in the fiber by seismic waves and calculating the subsequent changes in the SOP, it is possible to predict the arrival of the earthquake in real-time. This thesis explores the development of a simulation framework that models how optical fiber networks respond to seismic events, with the ultimate goal of improving earthquake early warning capabilities.

1.2 Objective

The primary objective of this thesis is to develop a simulation framework capable of modeling the behavior of optical fiber networks in response to seismic events, with the aim of enhancing earthquake early warning systems. This framework will simulate how seismic waves propagate through the ground and induce strain on optical fibers, and it will analyze how this strain affects the state of polarization (SOP) of light signals within the network.

To achieve this, the thesis integrates key components such as:

- A model that accurately simulates seismic wave propagation and the resulting strain along fiber optic lines [4].
- A waveplate model that simulates the effects of birefringence and polarization changes in the fiber due to the induced strain [4].
- An analysis of how these changes can be used to detect and characterize seismic events in real-time.

The overarching goal is to evaluate the feasibility of using existing optical fiber infrastructure for real-time seismic monitoring and early warning. By providing insights into how these networks respond to seismic waves, this research aims to contribute to the development of cost-effective, scalable, and rapid-response systems for earthquake detection. The framework developed will not only simulate different seismic scenarios but also assess how the fiber network's characteristics influence the detection accuracy and warning time.

1.3 Structure

The thesis is organized into eight chapters, each covering different aspects of the research:

- **Chapter 1: Introduction** – Provides an overview of the problem addressed by this research, introduces earthquake early warning systems (EEWS), and presents the objectives of the study.
- **Chapter 2: Background and Literature Review** – Discusses the propagation of seismic waves and the importance of early warning systems. It also reviews related works on Distributed Acoustic Sensing (DAS), State of Polarization (SOP) sensing, and machine learning applications in seismic detection.

- **Chapter 3: Modeling Fiber Optic Network and Generation of Seismic Events** – Introduces the methodology for creating a digital twin of the optical fiber network, simulating seismic events using the Syngine platform, and applying strain to the network.
- **Chapter 4: Strain and Waveplate Model Integration** – Delves into the simulation of strain evolution in fiber optics and explains the integration of the Waveplate Model to capture changes in birefringence and SOP during seismic events.
- **Chapter 5: The Simulation Manager** – Explains the architecture and functionality of the simulation manager, detailing data preprocessing, simulation orchestration, error handling, and robustness.
- **Chapter 6: Visualization Methods for Seismic Data Analysis** – Presents the tools and methods used for visualizing seismic data, including strain evolution and SOP changes, with an emphasis on Mapbox integration.
- **Chapter 7: Results Analysis** – Provides a detailed analysis of the simulation results across three different cases, showing how seismic events impact the optical fiber network and discussing the observed strain, SOP and SOPAS variations.
- **Chapter 8: Conclusion and Future Developments** – Summarizes the key findings, discusses technical challenges, and proposes future improvements to the simulation framework and its potential applications for earthquake detection.

Chapter 2

Background and Literature Review

2.1 Introduction

The growing reliance on optical fiber networks for high-speed communication has opened new avenues for using these networks as distributed sensing platforms. Among the most promising applications is their potential for *seismic detection*, offering a cost-effective solution by leveraging existing fiber infrastructure rather than deploying specialized sensor networks [5].

Seismic waves, caused by tectonic movements or other disturbances, can induce mechanical stress on buried optical fibers. This stress, in turn, alters the *state of polarization (SOP)* of light signals traveling through the fibers. By tracking changes in SOP, it is possible to detect seismic activities and even predict their arrival in urban areas [6]. This approach not only takes advantage of the wide coverage of terrestrial optical networks but also eliminates the need for costly, dedicated hardware.

Two main methods have been explored for seismic detection through optical fibers: *Distributed Acoustic Sensing (DAS)* and polarization-based techniques. While DAS has proven to be effective, it requires specialized, expensive equipment and is limited by its operational range of less than 100 km. In contrast, *polarization sensing* uses existing components of optical networks and offers a longer range at minimal additional cost [7]. This chapter explores these technologies and presents the *Waveplate Model*, a computational method that models birefringence and polarization changes induced by seismic waves in optical fibers.

This chapter will also highlight recent research, including machine learning applications, such as the *LSTM-based neural networks* used to enhance the detection and interpretation of seismic events based on optical data [8].

2.2 Seismic Waves Propagation

Seismic waves are vibrations generated by the release of energy during earthquakes. These waves travel through the Earth, and they can be classified into two main categories: body waves and surface waves. Body waves propagate through the Earth's interior, while surface waves travel along the Earth's surface, causing the most significant damage [8].

Body waves are further divided into Primary (P) waves and Secondary (S) waves. P-waves are longitudinal waves, meaning that the particle displacement is parallel to the wave propagation direction. These waves are the fastest and are the first to be detected by seismic stations. S-waves, on the other hand, are transverse waves, where particle displacement occurs perpendicular to the direction of wave propagation. Although slower than P-waves, S-waves carry more energy and are often associated with greater ground shaking.

Surface waves, which include Love and Rayleigh waves, cause the greatest destruction during an earthquake. These waves propagate along the Earth's surface and have a lower frequency compared to body waves. Love waves move horizontally, causing horizontal shearing of the ground, while Rayleigh waves induce a rolling motion, combining vertical and horizontal movements. Due to their large amplitudes and prolonged shaking, surface waves are typically responsible for the most severe structural damage during an earthquake [9].

The propagation speed and amplitude of seismic waves depend on several factors, such as the type of material they travel through, the depth of the earthquake, and the distance from the epicenter. Detecting and analyzing these waves is crucial for earthquake early warning systems, as the arrival of P-waves can serve as a precursor to more destructive S-waves and surface waves.

2.3 Importance of Early Warning Systems

Early warning systems (EWS) for earthquakes are essential for mitigating the impact of seismic events on infrastructure and human lives. The fundamental principle behind earthquake early warning is to detect the initial, less destructive P-waves and use them as a trigger to alert populations and initiate safety measures before the arrival of more destructive S-waves and surface waves [10].

The time interval between the detection of P-waves and the arrival of the stronger waves, known as lead time, can vary from a few seconds to minutes, depending on the location of the seismic event and the distance of the affected area from the epicenter. Even a warning of a few seconds can make a significant difference, providing valuable time for individuals to take protective actions, for automated systems to stop critical infrastructure (such as power plants or trains), and for

emergency response systems to prepare for the ensuing impact.

Countries like Japan and Mexico have successfully implemented EWS that have proven effective in mitigating earthquake damage. For example, Japan's early warning system, JMA (Japan Meteorological Agency), has been operational since 2007 and has provided timely alerts for numerous seismic events, including the devastating Tohoku earthquake in 2011 [11]. Similarly, Mexico's Seismic Alert System (SASMEX) has been in place for years and has provided valuable warnings for major earthquakes [12].

Implementing a reliable early warning system involves several key components: the detection and analysis of seismic waves in real-time, the ability to transmit alerts quickly, and the integration of automatic response systems to minimize damage. The use of optical fiber networks as a sensor grid for seismic activity, as discussed in this thesis, offers a cost-effective and scalable solution for detecting earthquakes in real-time. By leveraging existing telecommunication infrastructure, optical fiber networks can complement traditional seismic sensors and enhance the overall efficiency of early warning systems [5].

2.4 Related Works

The detection of earthquakes through optical fiber networks has garnered significant interest in recent years. A range of techniques and methodologies have been proposed to enhance the accuracy and efficiency of early warning systems, utilizing both existing infrastructure and advanced sensing technologies. This section provides a review of the key scientific contributions in three main areas: Distributed Acoustic Sensors (DAS), State of Polarization (SOP) monitoring, and machine learning applications for seismic detection.

2.4.1 Distributed Acoustic Sensors (DAS)

DAS systems are widely recognized for their ability to detect mechanical stresses over optical fibers. These systems can measure strain continuously along the fiber by analyzing the backscattered light from a pulsed laser. While DAS technology has proven to be effective for seismic detection [6], it is typically constrained by a maximum operational range of less than 100 km and requires specialized and expensive hardware. In contrast, this thesis explores cost-effective alternatives that utilize polarization sensing within existing optical networks, offering the potential for large-scale deployment without additional hardware costs.

2.4.2 State of Polarization (SOP) Sensing

Monitoring the State of Polarization (SOP) in optical fibers has emerged as a viable alternative to DAS for seismic detection. Changes in the SOP of light signals traveling through optical fibers are induced by mechanical disturbances, such as those caused by seismic waves. In particular, the work of Awad et al. [4] proposes the *Waveplate Model*, which simulates the evolution of SOP along the fiber during seismic events. This model was applied to real earthquake data from a 4.9 magnitude event in Italy, successfully demonstrating the detection of P-waves before the arrival of destructive surface waves. By using SOP sensing, optical networks can function as distributed seismic sensors without requiring dedicated equipment, making them a promising solution for large-scale earthquake monitoring.

2.4.3 Machine Learning Applications

Recent advances in machine learning have also played a critical role in improving the detection capabilities of optical fiber networks. Several studies have applied neural networks to the analysis of SOP variations in optical fibers, enabling more accurate and timely identification of seismic events. For example, a machine learning-driven smart grid approach was proposed in [8], where a neural network with attention mechanisms was used to analyze SOP variations in real-time, predicting the arrival of P-waves and allowing emergency response systems to take action. This approach leverages the computational capabilities of edge devices within the network and offers a scalable solution for enhancing seismic detection.

2.4.4 Final Remarks

In summary, the reviewed works demonstrate the effectiveness of optical fiber networks in detecting seismic activity. While DAS systems provide precise measurements over short distances, SOP-based sensing offers a more scalable and cost-effective alternative for long-range monitoring. Furthermore, the integration of machine learning techniques holds great promise for improving the speed and accuracy of seismic event detection, paving the way for more robust early warning systems.

Chapter 3

Modeling Fiber Optic Network and Generation of Seismic Events

3.1 Introduction

This chapter introduces the methodology used for modeling the optical fiber network and simulating seismic events within the network. The goal is to create a *digital twin* of the physical optical network that can be used to simulate the effects of earthquakes on the State of Polarization (SOP) of light signals propagating through the fiber. A digital twin replicates the network structure and characteristics, enabling the application of seismic strains to observe their impact on the network.

The focus is on defining the network's architecture and integrating it with real seismic event data to simulate ground motions along the optical fibers. This model will serve as the foundation for further analysis in the following chapters, where we integrate the Waveplate Model to study polarization effects and develop algorithms for early detection of seismic activity.

The integration of the Syngine [13] platform for generating synthetic seismic events is also discussed, providing a way to simulate earthquakes with realistic parameters. This chapter lays the groundwork for applying the seismic strains and observing the response of the optical network, contributing to a comprehensive earthquake early warning system based on optical fiber sensing.

3.2 Structure of the Digital Twin

The digital twin in this context refers to a simulated representation of the physical fiber optic network and its response to seismic events. This simulation framework is essential for understanding how seismic-induced strain propagates through optical fiber lines and how it affects the state of polarization (SOP) of the transmitted light signals. The framework models both the seismic wave propagation and the fiber network's physical response to these waves.

3.2.1 Nodes and Lines Abstraction

The network is divided into two key structural components: *nodes* and *lines*. Nodes represent the connection points in the network, while lines represent the optical fiber links that transmit the data. Each line is divided into multiple segments, and each segment receive its own unique strain evolution due to the seismic waves. This segmentation is crucial for simulating the local variations in stress along the fiber, as the seismic wave's impact will not be uniform across the entire network.

3.2.2 Segment Abstraction and Strain Application

To model the effect of seismic waves, each segment of the line is assigned a strain value based on the seismic activity. This strain value is calculated from the ground motion data, which is integrated from real-world seismic events, such as those provided by the Syngine. The strain values are applied to the segments, which represent discrete sections of the optical fiber. By applying strain to each segment, the simulation can track the changes in SOP along the fiber, as the stress disturbs the light's polarization.

3.2.3 Syngine Integration for Seismic Events

The Syngine service provides realistic ground motion time series for different seismic events. For each segment in the network, the Syngine platform generates synthetic seismograms using a moment tensor model. This moment tensor is based on the earthquake's magnitude and fault geometry, which is critical for simulating realistic ground displacement values. Once generated, these seismograms are filtered and converted into strain values that are applied to the network. This process ensures that the ground motion and strain data are accurately represented, making the simulation closely mirror real-world seismic behavior. The simulation incorporates different seismic scenarios by adjusting parameters such as the earthquake magnitude, location, and depth. This approach allows for the analysis of how varying seismic conditions affect the fiber network. Specifically, the query to Syngine is

instantiated for each network segment, enabling the fiber to sense the earthquake as if it were in its actual position.

3.3 Use of Segment Abstraction

Segment abstraction is a key component in modeling the interaction between seismic waves and the optical fiber network. The optical fiber lines are not treated as unique entities but are divided into smaller, manageable segments. Each segment represents a discrete portion of the fiber with its own strain evolution due to the seismic waves. Within each segment, the strain from the earthquake is considered uniform, whereas for the entire fiber line, the seismic wave impacts different points in varying ways. This abstraction simplifies the computational complexity of simulating seismic stress effects on the fiber network and allows for more accurate analysis as the segment length is reduced.

3.3.1 Rationale for Segmenting the Fiber

Seismic waves induce mechanical stresses on the fiber, causing variations in the fiber's internal birefringence, which in turn affects the SOP of the light passing through it. These mechanical stresses, however, are not uniformly distributed along the entire length of the fiber. Different parts of the fiber experience varying levels of strain depending on their distance from the earthquake's epicenter, the local geological conditions, and the characteristics of the seismic wave propagation.

By dividing the fiber into segments, each segment can be assigned a unique strain profile that reflects the local seismic activity. This enables the simulation to model how the SOP changes locally, rather than assuming uniform strain across the entire fiber. The segment abstraction provides a more granular and realistic representation of the seismic effects on the optical network.

3.3.2 Strain Calculation and Application

Each segment's strain is calculated based on the ground motion data from seismic events. The simulation framework, leveraging data taken from Syngine, specifically it generates strain values for each segment as the seismic wave propagates through the ground. These strain values are applied to each segment of the optical fiber, allowing the simulation to track how the fiber's SOP changes over time as the strain evolves.

The strain applied to a segment is influenced by several factors, including:

- The magnitude and depth of the seismic event.
- The distance of the segment from the earthquake epicenter.

- The properties of the ground and the fiber's surrounding environment.

By calculating and applying strain on a per-segment basis, the simulation can model how the polarization of light in the fiber is affected by seismic waves at a highly localized level.

3.3.3 Advantages of Segment Abstraction

The use of segment abstraction offers several advantages:

- **Granularity**: The division into segments allows for detailed analysis of how seismic stress affects different parts of the fiber, providing more accurate predictions of SOP changes. The segment length is variable, and users can decide the level of granularity by choosing the appropriate segment length.
- **Scalability**: By abstracting the fiber into smaller units, the simulation can scale to large networks without losing accuracy or becoming computationally expensive.
- **Localization**: Segment abstraction enables the detection of localized seismic events and can help pinpoint the locations of disturbances within the network. This is critical for early warning systems, which rely on the ability to detect seismic activity in real-time and respond accordingly.

In summary, segment abstraction is crucial for accurately modeling the interaction between seismic events and optical fibers. It allows the simulation to capture localized changes in SOP, providing a more detailed and realistic representation of the fiber network's behavior during an earthquake.

3.4 Syngine Integration

The integration of Syngine, a web-based service provided by IRIS (Incorporated Research Institutions for Seismology), plays a crucial role in generating realistic seismic wave data for the simulation framework. Syngine offers synthetic seismograms based on user-defined parameters, allowing for the simulation of ground motion and strain that would be experienced by the optical fiber network during an earthquake.

3.4.1 Purpose of Syngine Integration

Syngine provides detailed ground motion time series data for specific seismic events, including earthquake magnitude, depth, and location. By integrating this data into the simulation, we can simulate the impact of actual or hypothetical seismic

events on the optical fiber network. This ensures that the seismic strain applied to the fiber segments is both realistic and scientifically accurate.

The Syngine service delivers ground motion data in the form of `.sac` files, which contain time-series data representing the seismic event. This data is used to compute the strain experienced by each segment of the optical fiber, which, in turn, affects the state of polarization (SOP) of light passing through the fiber.

3.4.2 How Syngine is Integrated

To integrate Syngine into the simulation, the following steps are performed:

- The simulation framework sends a request to the Syngine service, specifying the earthquake parameters such as magnitude, source location, depth, and the desired receiver locations (the position of the fiber segments).
- Syngine responds by generating synthetic seismograms in `.sac` format, which are then downloaded and extracted into the simulation environment.
- The downloaded seismic data is processed to extract the relevant ground motion time series for each fiber segment. This data is then converted into strain values using predefined formulas, as seismic ground displacement directly affects the mechanical strain experienced by the optical fibers.

3.4.3 Ground Motion to Strain Conversion

Once the ground motion data is obtained from Syngine, it is necessary to convert this data into strain values that can be applied to the optical fiber segments. This is done using a simplified model of strain, which relates ground displacement to fiber elongation. The strain on each segment is calculated as:

$$\epsilon = \frac{\Delta L}{L}$$

where ϵ is the strain, ΔL is the change in length of the fiber segment due to ground motion, and L is the original length of the segment.

Syngine provides the ground displacement ΔL, which is then applied to the corresponding fiber segment. This allows the simulation to model the physical deformation of the fiber caused by the seismic waves, which is crucial for calculating changes in the state of polarization (SOP).

3.4.4 Advantages of Using Syngine

The integration of Syngine into the simulation framework offers several advantages:

- **Realism**: Syngine provides scientifically accurate, real-time seismic data based on global seismological models, ensuring that the simulated seismic waves closely resemble actual earthquake events.
- **Customization**: Users can define specific earthquake parameters, allowing for tailored simulations of seismic events that could impact specific regions or optical fiber networks.
- **Efficiency**: The Syngine service automates the generation of synthetic seismograms, saving time and reducing the complexity of manually modeling seismic waves. Moreover, using real-time seismogram data generated by Syngine ensures that the simulated strain on the fiber network mirrors actual seismic behavior at the specific sensing location.

3.4.5 Generation of the Moment Tensor and Seismic Data Processing

To accurately simulate seismic events and their impact on the optical fiber network, the Syngine platform is used to generate synthetic seismograms. For each seismic event, the moment tensor is calculated based on the earthquake's magnitude, strike, dip, and rake angles using *Pyrocko's moment tensor library*. The moment tensor represents the source mechanism of the earthquake and is a critical factor in determining how seismic waves propagate through the ground and impact the fiber network.

The moment tensor is converted into synthetic seismograms, which are used to model ground displacement at specific receiver locations along the fiber. Once the seismograms are generated, they undergo preprocessing and filtering:

- **Detrending**: Removes any linear trends in the data, which could distort the analysis of seismic waves.
- **Low-pass and high-pass Filtering**: Filters the data to focus on the frequency range of interest for seismic wave analysis. This is important since the wave can be affected by some kind of noise.

This processed seismic data is then converted into ground displacement values that are used to calculate the strain applied to each fiber segment. The filtering ensures that the strain values used in the simulation are realistic and correspond to the behavior of seismic waves at the appropriate frequency ranges.

Chapter 4

Strain and Waveplate Model Integration

4.1 Introduction

In this chapter, we delve into the integration of seismic strain data with the Waveplate Model to simulate the state of polarization (SOP) changes in optical fiber networks. The interaction between seismic waves and buried optical fibers causes mechanical strain, which, in turn, alters the polarization of light signals traveling through the fibers. By accurately modeling the propagation of strain along the fiber segments, we can predict how the SOP evolves as seismic waves propagate through the network.

The Waveplate Model, which is used to simulate birefringence in the optical fibers, is a key component of this approach. Birefringence, caused by imperfections in the fiber and external perturbations, affects the polarization of light, and the Waveplate Model helps us capture these changes in detail. In this chapter, we will describe how seismic strain is applied to the fiber segments, explain the principles behind the Waveplate Model, and introduce a matrix-based simplification of the model to improve computational efficiency. Finally, we will discuss how the State of Polarization Angular Speed (SOPAS) is calculated, providing insights into how polarization changes can be monitored in real-time for seismic detection.

This chapter forms the core of the simulation framework by connecting the seismic event data with the polarization model, enabling the monitoring of polarization shifts during earthquakes and their application in early warning systems.

4.2 Strain Evolution

The strain evolution in optical fibers is a key factor in determining how seismic events impact the State of Polarization (SOP) of the light traveling through the fiber. In our simulation, strain is applied to each segment of the fiber network to mimic the effect of seismic waves. This strain, induced by seismic activity, alters the birefringence of the fiber, which in turn changes the SOP.

4.2.1 How Strain is Modeled

Strain is modeled as a time-varying phenomenon across different segments of the fiber. Each segment experiences a unique strain evolution based on its location and the characteristics of the seismic wave affecting it. The strain applied to each segment is derived from the ground displacement caused by seismic waves. This displacement is converted into strain using the following formula:

$$\epsilon = \frac{\Delta L}{L_0}$$

where:

- ϵ is the strain (dimensionless),
- ΔL is the change in length (ground displacement),
- L_0 is the original length of the fiber segment.

For each segment of the optical fiber, the ground displacement due to seismic waves is converted into strain using a predefined conversion factor, which represents how much ground displacement results in strain along the fiber.

4.2.2 Strain Evolution Over Time

The strain is not static but evolves over time as the seismic wave propagates. The strain evolution for each segment is represented as a time series, where the strain at each time step is updated based on the seismic wave's impact at that point in time. This is calculated as:

$$\epsilon(t) = \text{convert\_displacement\_to\_strain}(d(t), \text{conversion factor})$$

Where $d(t)$ is the displacement caused by the seismic wave at time t, and the conversion factor ensures that the displacement is appropriately scaled to reflect strain in the optical fiber.

4.2.3 Code Implementation of Strain Evolution

Below is the code representation of how strain evolution is computed for each segment of the fiber:

```
for line in self.lines:
    for segment in line.segments:
        # Generate seismic wave specific to the segment
        segment_wave = earthquake.generate_syngine_wave_for_segment(segment)

        # Initialize the strain evolution array
        segment.initialize_strain_evolution(len(segment_wave))

        for time_step in range(len(segment_wave)):
            # Calculate strain at each time step
            parallel_displacement = segment_wave[time_step]
            parallel_strain = self.convert_displacement_to_strain(parallel_displacement, conversion_factor)
            segment.strain_evolution[time_step] = [parallel_strain, 0]  # Only parallel strain calculated
```

In this implementation, the strain evolution for each segment is calculated by iterating over the time steps of the seismic wave. At each time step, the displacement caused by the seismic wave is converted into strain using the conversion factor.

4.2.4 Impact on the Waveplate Model

Once strain is applied to the segments, it affects the birefringence of the fiber, which is a key input to the *Waveplate Model* discussed in later sections. The strain evolution alters the polarization of the light traveling through the fiber, and this is captured by the State of Polarization Angular Speed (SOPAS) and other metrics.

In summary, strain evolution helps for simulating how seismic waves mechanically affect the fiber, which in turn influences the light's polarization. The accurate modeling of this strain evolution is essential for predicting and detecting seismic events using optical fiber networks.

4.3 Strain Application on Segments

In this section, we will explore the methods used to apply strain to optical fiber segments in response to seismic events. Optical fibers buried underground experience mechanical stress during seismic activity, and this stress can be quantified

as strain. To simulate this effect, strain values are calculated and applied to each segment of the fiber network.

We present two approaches for applying strain, each serving different purposes based on the complexity and precision of the simulation:

- **Strain Application with Receiver Consideration:** This method takes into account the relative position of the earthquake's epicenter and the specific segment of the fiber. The strain is calculated based on the angle and distance from the epicenter, making it more precise for localized seismic events. For each segment, the function makes calls to *Syngine*, a seismic data service, passing the segment's position and the earthquake's epicenter to generate seismic waves specific to that segment. This ensures that the waves applied reflect the actual sensing data that would be observed in each segment during an earthquake event, thus simulating a realistic strain profile across the network.
- **Strain Application without Receiver Consideration:** In this simplified version, a single seismic wave is generated by selecting a reference point near the earthquake's epicenter as the receiver. This wave is then propagated across the entire network, with each segment's strain being adjusted through attenuation calculations based on the segment's distance from the epicenter. The attenuation factor accounts for the energy dissipation of seismic waves over distance, ensuring that segments farther from the epicenter experience lower strain values compared to those closer. This approach is more computationally efficient but sacrifices some precision, as the specific geometric positioning of each segment relative to the earthquake is not considered.

The following subsections will detail these two methods, discussing their implementations and the specific functions involved in each approach.

4.3.1 Strain Application with Receiver Consideration

The `apply_earthquake_strain_with_receiver` function is responsible for applying seismic strain to each segment of the fiber optic network. This function processes seismic data and converts it into strain values based on the earthquake's characteristics and the segment's position relative to the earthquake's epicenter. Below is an explanation of the logic and a breakdown of how the function operates.

- **Line and Segment Processing:**
  The function iterates over all fiber lines and their respective segments. For each segment, a seismic wave is generated through a call to `generate_syngine_wave_for_segment`, which simulates the ground motion experienced by that specific segment during the earthquake:

```
    for line in self.lines:
        for segment in line.segments:
            segment_wave = earthquake.
generate_syngine_wave_for_segment(segment)

```

- **Initialization of Strain Evolution:**
  The strain evolution for each segment is initialized to store the computed strain data for each time step. Additionally, the timestamp of the earthquake is saved:

  ```
  segment.initialize_strain_evolution(len(segment_wave))
  segment.timestamp = earthquake.start_time
  ```

- **Conversion Factor:**
  A conversion factor is used to translate ground displacement (measured in nanometers) into strain (measured in nano-strain). Each nanometer of ground displacement corresponds to a specific amount of strain applied to the waveplate. The conversion factor is defined as follows:

  ```
  conversion_factor = waveplate_interval
  ```

  This factor ensures that the strain calculation is related to the displacement experienced by the optical fiber wave plate.

- **Strain Calculation per Time Step:**
  For each time step in the seismic wave, the function calculates the angle between the earthquake's epicenter and the segment. The angle is used to compute both the parallel and orthogonal displacements that the segment experiences. These displacements are then converted into strain values:

```
    for time_step in range(len(segment_wave)):
        angle = calculate_angle(segment, earthquake.position)
        parallel_displacement, orthogonal_displacement =
calculate_displacement(
            segment_wave[time_step], angle)
        parallel_strain = self.convert_displacement_to_strain(
parallel_displacement, conversion_factor)
        orthogonal_strain = self.convert_displacement_to_strain(
orthogonal_displacement, conversion_factor)
```

```
7             segment.strain_evolution[time_step] = [parallel_strain,
    orthogonal_strain]
8
```

Supporting Functions

Several supporting functions assist in performing precise calculations. These include:

- **generate_syngine_wave_for_segment:** Generates seismic strain waves for each segment based on its position relative to the earthquake.
- **calculate_angle:** Computes the angle between the segment and the earthquake's epicenter, which is important for calculating the strain distribution.
- **calculate_displacement:** Calculates both parallel and orthogonal displacements based on the seismic wave's characteristics and the segment's orientation relative to the earthquake, thus using the angle of the previous function.
- **convert_displacement_to_strain:** Converts displacement values into strain using the pre-defined conversion factor.

In this method, seismic waves are simulated for each segment individually, and the resulting strain values are directly tied to the unique position and angle of the segment relative to the epicenter. This method provides a localized and precise strain calculation for each segment and thus for each part of the network.

4.3.2 Strain Application without Receiver Consideration

In addition to the `apply_earthquake_strain_with_receiver` function, a variant function `apply_earthquake_strain` applies strain to the fiber network without considering specific receiver positions. This variant simplifies the process by calculating the strain on the entire fiber network based solely on the earthquake's characteristics, without detailed geometrical considerations. Below is an explanation of the function logic and its key differences.

Key Differences from the `apply_earthquake_strain_with_receiver` Function

- **Absence of Receiver-Specific Logic:**
  Unlike `apply_earthquake_strain_with_receiver`, this version does not consider the specific positions of fiber segments relative to the earthquake's epicenter. Instead, the earthquake's effect is applied starting from a single waveform and then attenuated, based on distance across the entire network.

- **Simplified Strain Calculation:**
  The function simplifies the process by directly applying the strain from the seismic wave data, computing only the parallel strain for each segment. This eliminates the need for orthogonal strain calculation, making it computationally more efficient.

Functions Involved

This variant introduces fewer computations compared to the receiver-based strain application, using a more generalized set of functions:

- **generate_syngine_wave:**
  This function generates a synthetic seismic wave that applies uniformly to all segments in the fiber network. Unlike the previous approach, where a unique wave was generated for each segment, this function is called only once. The generated wave is then distributed across all segments, assuming that the entire network is equally affected by the earthquake's force. This simplifies the simulation by treating the network as a whole rather than as individual segments with distinct seismic responses.

- **convert_displacement_to_strain:**
  This function remains the same as in the previous method. It converts the displacement measured from the seismic wave into strain values using a predefined conversion factor, ensuring consistency in the strain calculation.

Conclusion on the Differences

While both functions aim to simulate the strain applied to the fiber network, the `apply_earthquake_strain` function simplifies the process by ignoring receiver-specific data and applying the strain uniformly across the network. This makes it computationally more efficient but less precise in scenarios where localized strain calculations are necessary. The absence of segment-specific angle and displacement calculations means this method is faster but sacrifices accuracy when the exact position of each segment relative to the earthquake's epicenter is important.

4.4 Concept of Waveplate Model

The *Waveplate Model* is a computational approach designed to simulate the impact of birefringence on the state of polarization (SOP) of light propagating through an optical fiber. In the context of seismic detection, seismic waves induce strain in the optical fiber, altering the birefringence and, consequently, the SOP of the light signal. The Waveplate Model divides the optical fiber into small pieces (inside each

segment), or "waveplates", to model the accumulated polarization effects caused by these seismic-induced strains.

4.4.1 Internal and External Birefringence

Optical fibers are inherently birefringent due to imperfections during their construction. This internal birefringence affects the polarization of light as it travels through the fiber. Additionally, external mechanical stresses, such as those caused by seismic activity, introduce further birefringence. The Waveplate Model is designed to separate these two effects by simulating the internal birefringence across small uniform segments of fiber and analyzing deviations caused by external forces.

Mathematically, birefringence in the fiber can be described by the *Jones matrix formalism*. Each waveplate can be represented by a Jones matrix M_d, which models the phase retardation caused by the birefringence:

$$M_d = \begin{pmatrix} e^{i\frac{\delta}{2}} & 0 \\ 0 & e^{-i\frac{\delta}{2}} \end{pmatrix}$$

where δ is the phase difference induced by the birefringence, which depends on the strain applied to the segment.

4.4.2 Waveplate Model Representation

The optical fiber is divided into N segments, each representing a waveplate with its own birefringence properties. The cumulative effect of all waveplates on the state of polarization (SOP) is computed by multiplying the Jones matrices for each waveplate in sequence. The orientation of each waveplate is randomized to simulate the natural variations in fiber geometry. The angle θ is generated for each segment to reflect these variations, and the corresponding rotation matrix $R(\theta)$ is applied to the Jones matrix for each waveplate. Let $R(\theta)$ represent the rotation matrix for the waveplate orientation:

$$R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

The Jones matrix for each segment can then be described as:

$$J = R^{-1}(\theta) M_d R(\theta)$$

where θ is the random orientation of the waveplate in the fiber. This randomness in orientation ensures that the simulation captures the unpredictable nature of birefringence effects in real-world fibers.
The final polarization state is the product of all Jones matrices across the fiber:

$$P_{out} = J_N J_{N-1} \cdots J_1 P_{in}$$

where P_{in} is the initial polarization state and P_{out} is the final polarization state at the end of the fiber.

4.4.3 Waveplate Model for Seismic Sensing

In the context of seismic sensing, the Waveplate Model tracks the changes in SOP caused by the strain along the fiber due to seismic activity. The strain alters the birefringence of each segment, modifying the phase retardation δ. This strain-induced birefringence is a direct result of the ground displacement caused by seismic waves.

Model Assumptions

The Waveplate Model makes the following assumptions:

- Each waveplate has a random but uniform internal birefringence.
- The strain induced by seismic waves introduces additional birefringence in each waveplate.
- The seismic-induced birefringence can be modeled as a small perturbation to the internal birefringence of the fiber.

Mathematical Representation of Strain

The strain ε in each segment of the fiber is calculated based on the displacement of the ground during seismic activity. This strain is used to modify the birefringence of the fiber:

$$\delta' = \delta_0 + \varepsilon \cdot \beta$$

where δ_0 is the internal birefringence and β is a constant representing the sensitivity of the fiber to strain.

The total change in the SOP is obtained by applying the modified Jones matrices to the light polarization, yielding the final polarization state after seismic activity.

4.4.4 Waveplate Model Implementation

The Waveplate Model is implemented in Python and simulates the effects of birefringence and strain on the SOP. Each waveplate's random orientation is taken into account, and the cumulative effect on polarization is calculated by multiplying the corresponding Jones matrices. The final output provides the SOP at the end of the fiber, which can be used to detect seismic events.

The output of the model is the state of polarization at the end of the fiber, which changes based on the seismic strain experienced by the fiber.

4.5 Matricial Simplification of Strain Processing

The original `process_strain_in_wp_model` function computes the strain effects along the optical fiber waveplates in an iterative manner, which can be computationally expensive. The optimized version, `process_strain_in_wp_model_matricial`, achieves the same results using matrix operations, leading to a more efficient computation. This section provides an in-depth explanation of the improvements and the mathematical formulations involved.

4.5.1 Original Strain Processing

In the original method [4], the strain evolution is calculated iteratively for each waveplate and time step:

- **Strain Matrix Construction:**
  The strain experienced by each waveplate is extracted from the segment strain evolution, creating a time series of strain for each waveplate:

  $$\text{strain\_matrix}(t, j) = \text{segment.strain\_evolution}(t, 0)$$

  where t represents the time step and j the waveplate index.

- **Rotation Matrix Calculation:**
  For each waveplate, the birefringence and imperfections due to internal fiber characteristics are modeled using a rotation matrix:

  $$R(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$$

  The inverse of the rotation matrix is applied sequentially to account for the effect of random orientation within the fiber segments.

- **Jones Matrix Multiplication:**
  The Jones matrices representing the polarization effects are multiplied iteratively for each waveplate and time step:

  $$P_{\text{out}} = J_n \times \cdots \times J_1 \times P_{\text{in}}$$

  This multiplication is done for each j and t in a loop cycle, making the process computationally expensive.

- **Output Polarization:**
  The final polarization state is computed through repeated matrix multiplications across all waveplates and for all time steps.

4.5.2 Matricial Strain Processing

The optimized version of the strain processing function replaces iterative calculations with matrix operations, significantly enhancing computational efficiency. Below, we break down the key operations involved in this matricial approach and explain their mathematical basis.

Pre-computed Rotation Matrices

In the original iterative approach, rotation matrices were computed individually for each waveplate and time step. In the matricial approach, we precompute these matrices and expand them for all time steps.

Each waveplate introduces a random internal birefringence that can be modeled using a rotation matrix $R(\theta)$, where θ represents the random angle of birefringence due to imperfections:

$$R(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$$

In the matricial version, instead of recalculating this matrix for each step, we precompute the rotation matrices for all time steps and all waveplates:

$$\text{rotation\_matrices} = \text{xp.array}\left([R(\theta)]\right)$$

To handle all waveplates simultaneously, these matrices are expanded across time steps using broadcasting techniques. This pre-computation significantly reduces the redundant calculations required at each iteration, optimizing performance.

Matricial Computation of Strain Effects

The strain effects on the waveplates are determined by the interaction between the seismic strain and the birefringence in the fiber. In the matricial approach, the strain data is processed as a matrix.

The phase shift due to birefringence and external strain is calculated as:

$$d'(t, j) = \text{Birefringence} \times (1 + \text{strain}(t, j))$$

where $\text{strain}(t, j)$ is the strain experienced by waveplate j at time t, and Birefringence is a fixed value that depends on the physical properties of the fiber.

This phase shift $d'(t, j)$ is then used to construct the diagonal Jones matrices for each waveplate:

$$M_d(t, j) = \begin{pmatrix} \exp\left(\frac{1j \cdot d'(t,j)}{2}\right) & 0 \\ 0 & \exp\left(\frac{-1j \cdot d'(t,j)}{2}\right) \end{pmatrix}$$

These diagonal matrices represent the polarization change due to birefringence and strain, and they are applied to all waveplates and time steps in a single matrix operation.

Efficient Jones Matrix Multiplication

In the original iterative approach, Jones matrices were multiplied one by one across all waveplates. In the matricial version, we use numpy's `einsum` function to efficiently compute the Jones matrix multiplication across all waveplates and time steps in one operation.

The Jones matrices for each waveplate and time step are computed as:

$$\begin{aligned} J_{\text{total}}(t) = \text{xp.einsum}('ijkl, ijmn, ijop- > ijkp', \\ \text{inv\_rotation\_matrices}, M_d, \text{rotation\_matrices}) \end{aligned} \tag{4.1}$$

where:

- inv_rotation_matrices is the inverse of the rotation matrix for each waveplate,
- M_d is the diagonal Jones matrix for the birefringence and strain,
- rotation_matrices is the precomputed rotation matrix.

This operation efficiently handles the matrix multiplication across all dimensions (*n_samples*, *n_waveplates*, and the 2x2 matrices), replacing the iterative loop with a single matrix operation.

Final Output Polarization

Once the total Jones matrix has been computed for each time step, the final polarization state of the light can be determined by applying this matrix to the input polarization state.

Let P_{in} be the initial polarization state of the light, which we assume to be linearly polarized at +45°:

$$P_{\text{in}} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

The output polarization state for each time step is then computed as:

$$P_{\text{out}}(t) = J_n(t) \times \cdots \times J_1(t) \times P_{\text{in}}(t)$$

This can be computed for all samples at once using:

$$P_{\text{out}}(t) = \text{xp.einsum}('ijl, l- > ij', J_{\text{total}}, P_{\text{in}})$$

This operation provides the final polarization state for all waveplates and all time steps.

Conversion to Stokes Parameters

After computing the output polarization, we convert it to Stokes parameters to analyze the polarization state.

The Stokes parameters S_1, S_2, and S_3 are computed as:

$$S_1 = |P_{\text{out}}(t)_1|^2 - |P_{\text{out}}(t)_2|^2 \tag{4.2}$$

$$S_2 = 2 \cdot \text{Re}\left(P_{\text{out}}(t)_1 P_{\text{out}}(t)_2^*\right) \tag{4.3}$$

$$S_3 = 2 \cdot \text{Im}\left(P_{\text{out}}(t)_1 P_{\text{out}}(t)_2^*\right) \tag{4.4}$$

These parameters describe the state of polarization on the Poincaré sphere.

Key Benefits of the Matricial Approach

The matricial approach provides several key advantages over the original iterative method:

- **Loop Elimination:** By using matrix operations, the matricial approach eliminates the need for nested loops over waveplates and time steps, significantly improving performance.
- **Vectorized Operations:** Operations such as Jones matrix multiplications and Stokes parameter calculations are vectorized, allowing them to be computed for all samples and waveplates simultaneously.
- **Pre-computation:** Precomputing the rotation matrices reduces the computational overhead by avoiding redundant calculations.

In summary, the matricial approach streamlines the strain processing pipeline by leveraging matrix operations, resulting in a significant reduction in computational time without compromising the accuracy of the polarization evolution.

4.5.3 Key Mathematical Differences

- **Loop Elimination:**
  In the matricial version, loops over waveplates and time steps are replaced by matrix operations. This eliminates the need for nested loops, significantly improving performance.
- **Matrix-based Jones Calculations:**
  In the matricial process, the polarization state is calculated using matrix multiplications:

$$P_{\text{out}}(t) = \text{xp.einsum}('ijl, l- > ij', J(t), P_{\text{in}})$$

This allows the Jones matrices of all waveplates to be applied in one step, rather than iteratively.

- **Pre-computation of Rotation Matrices:**
  The pre-computation and broadcasting of rotation matrices reduce the computational load by avoiding redundant calculations.

4.5.4 Performance Improvement

The matricial process significantly reduces computational complexity by eliminating loops and leveraging efficient matrix operations. This ensures that the same accuracy in polarization evolution is maintained while achieving much faster results, making the method suitable for large-scale simulations.

In conclusion, the matricial simplification of the strain processing pipeline drastically improves the simulation's speed and scalability, without compromising on the physical accuracy of the model.

4.6 SOPAS Calculation

The State of Polarization Angular Speed (SOPAS) is a metric designed to capture the rapidity of change in the state of polarization (SOP) within optical fiber networks. The SOP is typically represented by three Stokes parameters: S_1, S_2, and S_3, which describe the polarization in a 3D space. SOPAS provides a scalar value that encapsulates the speed at which the SOP vector rotates over time, offering insight into changes due to external perturbations such as seismic waves.

4.6.1 Mathematical Definition of SOPAS

SOPAS is calculated by measuring the angular displacement between consecutive SOP vectors, represented by the Stokes parameters, over small time intervals. Formally, it is defined as:

$$\text{SOPAS}(t) = \frac{1}{\Delta t} \arccos\left(\frac{\mathbf{S}(t) \cdot \mathbf{S}(t-\Delta t)}{|\mathbf{S}(t)||\mathbf{S}(t-\Delta t)|}\right)$$

where:

- $\mathbf{S}(t) = [S_1(t), S_2(t), S_3(t)]$ is the Stokes vector at time t,
- $\mathbf{S}(t-\Delta t)$ is the Stokes vector at the previous time step,
- Δt is the time interval between two measurements.

4.6.2 Computation of SOPAS

The SOPAS computation in the code involves the following key steps:

1. The Stokes parameters S_1, S_2, and S_3 are extracted for each time step.
2. For each time step t, the dot product of the Stokes vectors at time t and $t-1$ is computed:
$$\mathbf{S}(t) \cdot \mathbf{S}(t-1)$$
3. The norms of $\mathbf{S}(t)$ and $\mathbf{S}(t-1)$ are calculated:
$$|\mathbf{S}(t)| \quad \text{and} \quad |\mathbf{S}(t-1)|$$
4. SOPAS is then computed using the arc cosine of the normalized dot product divided by the time interval:
$$\text{SOPAS}(t) = \frac{\arccos\left(\frac{\mathbf{S}(t)\cdot\mathbf{S}(t-1)}{|\mathbf{S}(t)||\mathbf{S}(t-1)|}\right)}{\Delta t}$$

4.6.3 Code Implementation

The implementation of SOPAS in the simulation follows these steps:

```
def calculate_sop_angular_speed(self):
    for line in self.lines:
        if line.stokes_evolution is not None:
            S1 = line.stokes_evolution[:, 0]
            S2 = line.stokes_evolution[:, 1]
            S3 = line.stokes_evolution[:, 2]

            # Construct the Stokes vectors
            S_k = np.array([S1, S2, S3]).T
            S_k_minus_1 = np.roll(S_k, 1, axis=0)

            # Compute dot product and norms
            dot_product = np.sum(S_k[1:] * S_k_minus_1[1:], axis=1)
            norms_product = np.linalg.norm(S_k[1:], axis=1) * np.
    linalg.norm(S_k_minus_1[1:], axis=1)

            # Calculate cos(angle) and clip values between -1 and 1
            cos_angle = np.clip(dot_product / norms_product, -1, 1)

            # SOPAS calculation
            fs = 10  # Sampling frequency (10 Hz)
            SOPAS = np.arccos(cos_angle) * fs
            line.sop_angular_speed = SOPAS
```

4.6.4 Interpretation and Significance

SOPAS serves as a valuable tool for detecting external perturbations in optical fibers. High SOPAS values correspond to rapid polarization changes, indicating potential seismic activity. In our simulation, SOPAS is used to monitor the network's response to earthquakes, providing insights into how seismic waves affect light polarization in real-time

Chapter 5

The Simulation Manager

5.1 Introduction

The *Simulation Manager* is the core component responsible for orchestrating the entire seismic simulation workflow. Its primary function is to coordinate the interaction between the network model, the earthquake event, and the results visualization system. By managing all the key processes, the Simulation Manager ensures that the system runs smoothly, efficiently, and can handle complex simulations with varying parameters.

The Simulation Manager operates as a control unit that initializes the system components, processes seismic data, and generates results based on the fiber optic network's response to seismic activity. It is designed to be modular and adaptable, allowing for easy updates and integration of new functionalities without compromising the overall structure of the simulation.

This chapter will delve into the details of how the Simulation Manager orchestrates the various processes in the simulation, with a focus on the initialization of the network and earthquake, data preprocessing, strain application and processing, and finally, the handling and visualization of results. Additionally, it will discuss the system's robustness, including how it handles errors and manages cached data for efficient operation.

5.2 Data Preprocessing

The *Simulation Manager* plays a critical role in preparing the data before starting the simulation. This process, referred to as *Data Preprocessing*, includes cache management, loading or generating `.sac` files, and creating a log file to detect parameter changes. This section explores these aspects in detail.

5.2.1 Cache Management

To optimize the simulation performance and reduce loading times, the *Simulation Manager* utilizes a cache. The cache stores temporary data used during the simulation, including intermediate results or preprocessed data. If the simulation parameters change, the cache must be refreshed and new data generated.

Parameter	Cache Action
Earthquake Magnitude/Epicenter	Re-generate seismic data
Network Node Position	Recalculate strain and SOP
Segment Length or Waveplate Interval	Reset cache files

Table 5.1: Cache management based on simulation parameters.

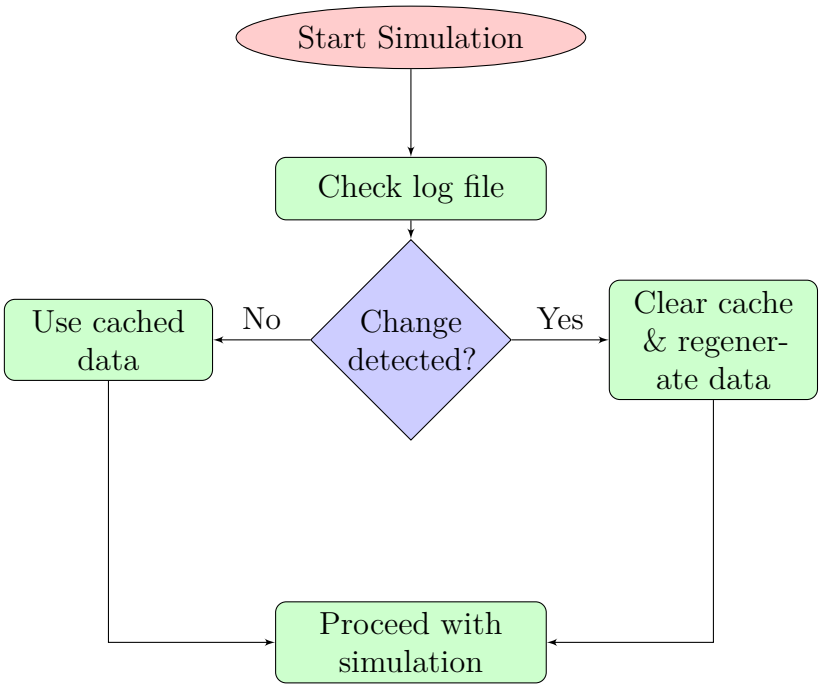

Figure 5.1: Flowchart illustrating the cache management process in the Simulation Manager.

If any of the listed parameters change, the system automatically clears the `cache/` and `resources/` directories, ensuring that the results reflect the new simulation conditions. The *Simulation Manager* checks the log file, comparing current parameters with saved values, to determine whether data needs to be regenerated.

5.2.2 Loading or Generating SAC Files

`.sac` files contain seismic data used in the simulation to apply earthquake effects to the network segments. The *Simulation Manager* performs one of the following operations:

- If the `.sac` files already exist in the `resources/` directory, the system loads them without generating new data.
- If the files are unavailable, the *Syngine* service is used to generate new `.sac` files, using the earthquake parameters specified in the simulation.

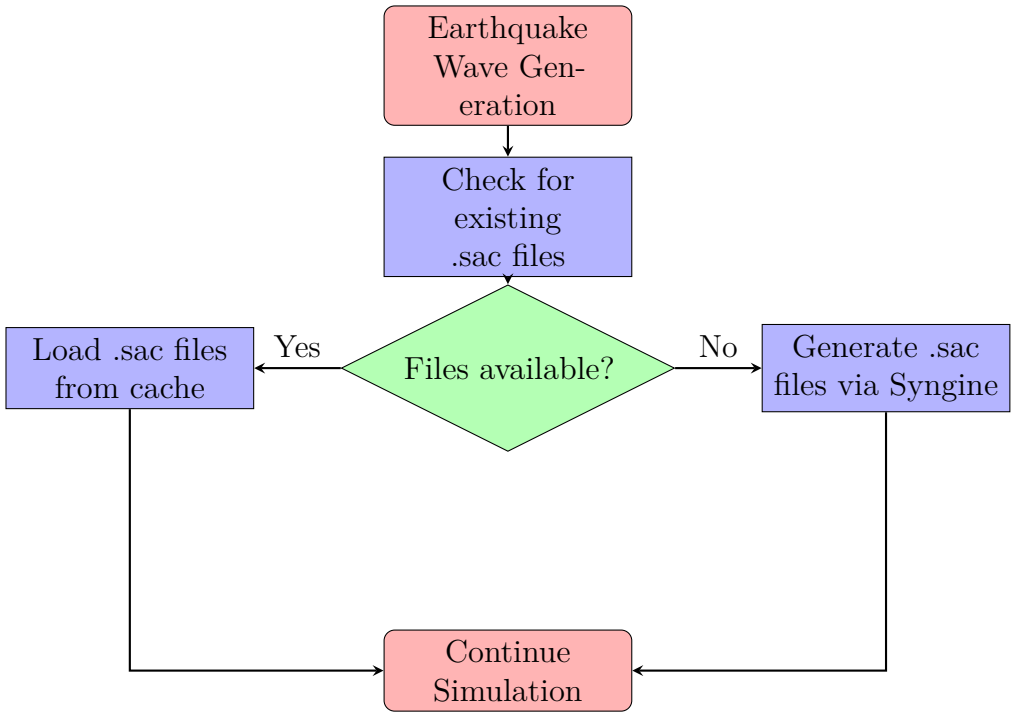

Figure 5.2: Flowchart representing the process of loading or generating `.sac` files.

Using existing `.sac` files reduces simulation time, while regenerating them via *Syngine* ensures that the data accurately reflects updated earthquake parameters.

5.2.3 Log File Generation for Change Detection

To ensure that the data is always up to date, the *Simulation Manager* generates and maintains a log file in the `resources/` directory. This file stores key simulation parameters such as earthquake magnitude, position, segment length, and network

configuration. If a difference is detected between the current parameters and those stored in the log file during a new simulation, the `cache/` and `resources/` directories are cleared, and new data is regenerated.

Parameter	Re-generation Condition
Magnitude	If different from the previous log
Epicenter Location	If different from the previous log
Network Structure	If segment length has changed

Table 5.2: Data re-generation conditions based on the log file.

The log file acts as a "memory" that allows the *Simulation Manager* to avoid unnecessary recalculations when the simulation parameters remain unchanged.

5.3 Simulation Orchestration

The *Simulation Manager* plays an important role in orchestrating the various stages of the simulation, from initializing the network to processing strain and computing polarization effects. This section provides a detailed description of the steps involved in orchestrating the simulation.

5.3.1 General Overview

The *Simulation Manager* acts as the core controller, ensuring that all components of the simulation work together seamlessly. It integrates the network structure, earthquake events, and the necessary computations to model seismic impacts on the optical fiber system.

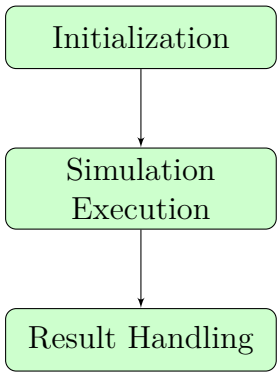

Figure 5.3: Main phases of the simulation orchestration: Initialization, Simulation Execution, and Result Handling.

The Simulation Manager process is divided into three main phases:

- **Initialization:** The network and earthquake data are loaded, and necessary precomputations, such as cache management, are performed.
- **Simulation Execution:** The earthquake strain is applied to the optical network, and the resulting polarization effects are computed using the waveplate model.
- **Result Handling:** The final polarization states are processed, and relevant visualizations are generated.

5.3.2 Earthquake Strain Application

The *Simulation Manager* applies seismic strain to each network segment based on the earthquake parameters. Two variants of strain application are available:

- **Strain Application with Receiver Consideration:** The strain is calculated based on the relative position between the earthquake's epicenter and each segment in the network. This method uses precise seismic data for localized strain calculations.
- **Strain Application without Receiver Consideration:** In this simplified version, the displacement query is performed only once, and the attenuation of the strain across segments is calculated manually using an attenuation factor, this version still considers the earthquake's impact angle and distance from the epicenter.

The strain application process begins by generating seismic waves for each segment, using two methods accordingly to the with/without receiver considerations. The strain is then processed according to the waveplate model.

5.3.3 Strain Processing and Polarization Effects

The strain experienced by each segment leads to birefringence effects in the fiber, altering the state of polarization (SOP) of the light traveling through it. The *Waveplate Model* is used to simulate these polarization effects.

- **Waveplate Model:** Each fiber segment is divided into multiple waveplates, and the strain applied to the segment induces phase shifts and changes in polarization. The model computes the SOP evolution for each waveplate.

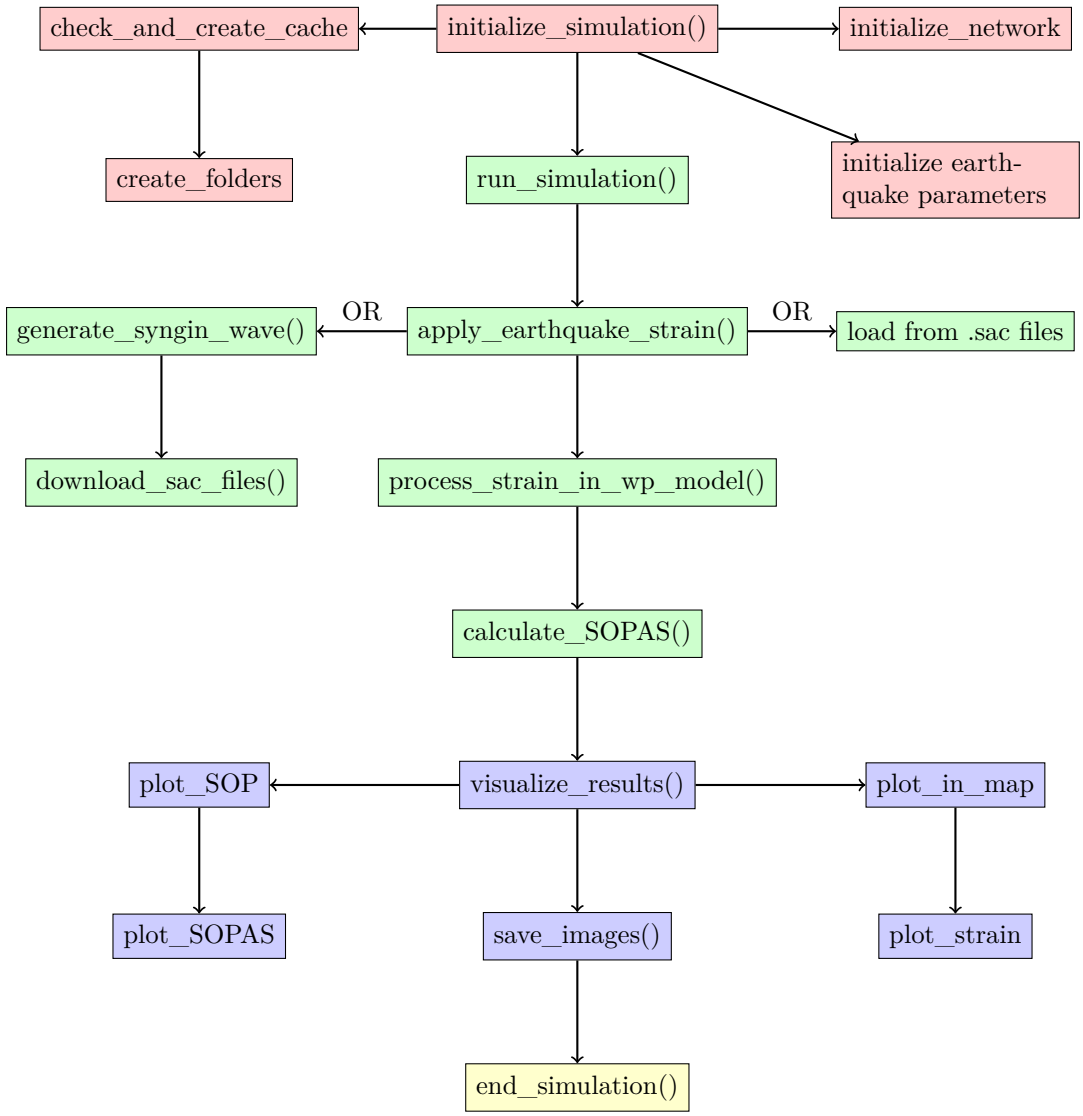

Figure 5.4: Flowchart showing a simplified execution of the program orchestrated by the Simulation Manager.

- **Matrix-based Optimization:** The simulation employs matrix-based operations to efficiently compute the Jones matrices for all waveplates and time steps, significantly reducing computation time compared to the non-matrix-based approach.

- **GPU Acceleration:** The process is significantly accelerated by utilizing GPUs, if available. This allows for faster computation and real-time processing of large datasets, enhancing the overall efficiency of the simulation.

5.3.4 Result Handling

The final stage of the simulation, as shown in 5.4 is the processing and visualization of the results. This phase includes the calculation of the State of Polarization, the SOP Angular Speed (SOPAS) and the visualization of the polarization evolution, which provides insight into how seismic events affect the polarization state in the fiber.

Result Handling

The final stage of the simulation is the processing and visualization of the results. This phase includes the following visualizations:

- **Mapbox Visualization:**
  - **Lines, Nodes, and Earthquake Epicenter:** The optical fiber network, including lines and nodes, is plotted on a geographical map using Mapbox. The earthquake epicenter is also marked to provide a spatial context of the seismic event.
  - **Propagation Circles:** Concentric circles are drawn around the epicenter to indicate the propagation of seismic waves over time. These circles help visualize the spread of the earthquake's impact and the areas affected by the seismic waves.
- **Strain Visualization:**
  - **Strain Evolution Plots:** The strain evolution is plotted for each fiber line, with three sample segments selected within each line. These plots show how the strain levels change dynamically as seismic waves travel through the optical fiber network.
- **State of Polarization (SOP) Visualization:**
  - **2D SOP Plots:** The evolution of the State of Polarization (SOP) over time is visualized in 2D plots for each fiber line. These plots track the Stokes parameters S_1, S_2, and S_3 at each time step.
  - **3D Poincaré Sphere:** The SOP is also visualized on a 3D Poincaré sphere, showing the trajectory of the polarization state over time. This provides a comprehensive view of how seismic events influence the polarization.
- **State of Polarization Angular Speed (SOPAS):**

  - **SOPAS Plots:** The State of Polarization Angular Speed (SOPAS) is plotted over time for each fiber line. These plots help identify periods of rapid polarization changes, indicating potential seismic activity.

These visualizations provide a clear understanding of the polarization evolution and strain distribution over time, offering valuable insights into how seismic events affect the optical fiber network.

5.3.5 Conclusion on Result Handling

The result handling process ensures that both the quantitative data (SOPAS) and visual representations (Poincaré sphere, SOP evolution) are available for analysis. This combination of numerical and visual data provides a comprehensive view of how seismic events affect the polarization state in optical fiber

5.4 Error Handling and Robustness

The robustness of the simulation framework is ensured through careful management of potential errors and interruptions. This section describes the strategies employed to handle missing or corrupt data and to manage critical system resources, ensuring that the simulation can recover from common failure scenarios.

5.4.1 Cache and Resource Folder Management

The `Simulation Manager` relies on cached data and pre-generated resources, such as SAC files, to speed up simulation runs. These resources are stored in specific directories (`cache` and `resources/sac_files`). If these folders are deleted or corrupted, the system must recreate them to prevent failures.

- **Cache Recreation:** During each simulation initialization, the system checks whether the `cache` and `resources` folders exist. If they are missing, the manager recreates them and logs this action in the log file.
- **Automatic Cleanup:** If changes to the simulation parameters (e.g., earthquake magnitude, segment length) are detected, the cache and resource folders are cleared, forcing the system to regenerate data.

The check for the existence of directories can be represented as follows:

if `not exists(cache_folder)` then create_folder(cache_folder)

This guarantees that the necessary directories are always present, avoiding missing folder errors.

5.4.2 Handling Missing or Corrupt Data

One of the most common sources of errors during simulation is missing or corrupt data, especially when dealing with pre-downloaded SAC files or cache data. The `Simulation Manager` includes a robust mechanism to detect these issues.

- **Data Integrity Check:** Before loading any cached data or SAC files, the system verifies the integrity of these files. For SAC files, this involves checking that the file to be reused is exactly the same as the one that the program would download from the servers. If any anomalies are detected, the system automatically triggers a re-download of the affected files.

- **Fallback Mechanism:** If a SAC file is found to be missing or corrupt, the system falls back to querying the IRIS Syngine API to regenerate the seismic wave data. This ensures that even in the event of data corruption, the simulation can continue.

5.4.3 Re-initialization in Case of Data Deletion

In scenarios where data or cache folders are accidentally deleted mid-simulation, the `Simulation Manager` automatically re-initializes the simulation environment. This includes clearing out incomplete data, regenerating files, and restarting the simulation from the last known good state.

The re-initialization process follows these steps:

1. **Detect Missing Data:** During each simulation run, the system checks for the existence of critical data. If a required file is missing, the system logs the error and proceeds to clean up the environment.

2. **Re-create Resources:** The system then recreates the necessary resources, such as SAC files, based on the current simulation parameters. If cache files are missing, the system regenerates them from scratch.

3. **Restart Simulation:** Once the environment is stable, the system restarts the simulation, ensuring that no incomplete or corrupt data is used.

The overall process ensures that the simulation can recover from a wide variety of errors, from missing cache files to data corruption, with minimal user intervention.

This table summarizes the common errors and the corresponding recovery actions taken by the `Simulation Manager` during runtime.

Error Type	Detection Method	Recovery Action
Missing cache folder	Directory existence check	Create missing folder
Corrupt SAC file	File integrity check	Re-download from Syngine
Deleted data during simulation	Periodic file check	Re-initialize and regenerate data

Table 5.3: Error handling and recovery actions based on error type.

5.5 Summary and Potential Future Improvements

The `Simulation Manager` plays a central role in orchestrating the entire simulation process, ensuring the smooth operation of each component. It manages the flow of data, from loading network configurations and earthquake events to the complex task of strain processing and polarization evolution. The implementation includes optimizations, such as the use of matrix-based strain processing and cache management, which significantly enhance performance.

5.5.1 Summary of the Simulation Manager's Contributions

Throughout the simulation, the `Simulation Manager` contributes in several key areas:

- **System Initialization and Configuration Management:** The manager initializes the system by loading network configurations and preparing earthquake parameters. It ensures that all required resources are in place before the simulation begins.
- **Data Preprocessing:** One of the most critical roles is preprocessing, which involves checking for cached data, ensuring SAC files are up-to-date, and loading the necessary data to reduce redundant downloads and calculations.
- **Simulation Orchestration:** The `Simulation Manager` coordinates the application of strain to each fiber segment and orchestrates the complex interactions between strain and polarization models. It applies both receiver-based and simplified strain calculations, depending on the simulation needs.
- **Optimized Strain Processing:** Using matrix-based calculations, the manager efficiently processes strain across all segments and waveplates, reducing the computational overhead compared to traditional methods.
- **Error Handling and Robustness:** Through robust error handling mechanisms, the `Simulation Manager` recovers from common failures, such as

missing data or folder deletion. It ensures that the simulation can proceed smoothly, even in the event of unexpected interruptions.

- **Visualization and Post-processing:** After the strain has been processed, the manager oversees the calculation of polarization changes and visualizes the results in the form of SOP, SOPAS, earthquake waveforms, and the other relevant metrics.

5.5.2 Potential Future Improvements

While the `Simulation Manager` is currently robust and efficient, there are several potential future improvements:

- **Further Parallelization and GPU Optimization:** While the current system uses matrix operations to improve performance - if possible executed on GPU -, further parallelization, particularly with GPUs, could enhance the simulation speed, especially for larger networks.
- **Enhanced Log Management:** Currently, the log system tracks changes in parameters, ensuring consistency across simulations. In the future, more detailed logs, such as performance metrics, memory usage, and detailed error reports, could be included for debugging and optimization.
- **Support for Multiple Earthquake Events:** The current simulation manager handles one earthquake at a time. Extending the system to manage multiple seismic events occurring simultaneously would improve the realism of the simulation.
- **Dynamic Network Adjustments:** The simulation assumes that the network structure remains fixed during the simulation. Future iterations could incorporate dynamic adjustments, allowing the simulation to adapt to real-time changes in the network (e.g., fiber cuts or rerouting).

In summary, the `Simulation Manager` represent a core component that not only coordinates the simulation process but also optimizes performance and manages potential errors. With future improvements, it has the potential to become even more versatile and powerful, accommodating more complex and realistic simulations of seismic events.

Chapter 6

Visualization Methods for Seismic Data Analysis

One of the key components of the developed simulation framework is the visualization of results. Visualization provides an intuitive way to understand how seismic waves impact the optical fiber network, particularly in terms of the evolution of strain and the State of Polarization (SOP) in the network.

This chapter outlines the methods used to visualize seismic events and their effects on the fiber network, focusing on tools such as *Mapbox* for geographic representation and various graphical plots to track the SOP and the State of Polarization Angular Speed (SOPAS). These visualizations allow us to monitor changes in the fiber network and understand the influence of seismic activity on the network's optical properties.

6.1 Mapbox Integration for Geographic Visualization

The integration of *Mapbox* allows the real-time visualization of the optical fiber network on a geographic map, highlighting network nodes, fiber lines, and the position of the earthquake's epicenter. The map dynamically displays concentric seismic waves emanating from the epicenter, showing how different fiber segments are affected by seismic activity.

Key features:

- **Geographical representation:** The fiber network is plotted geographically, helping to visualize the spatial relationship between network segments and the earthquake's epicenter.

- **Dynamic seismic data:** The epicenter and its surrounding wave propagation are visualized in real-time, offering immediate insight into affected fiber lines.

- **Interactive exploration:** Users can zoom in and out to explore specific nodes and fiber lines, provided upon interaction with the map.

6.2 Strain Evolution Visualization

Strain evolution along the fiber lines represents the impact of seismic waves over time. The visualization tracks how strain affects each fiber segment and plots both the parallel and orthogonal strain components using two different colors: blue for the parallel component and orange for the orthogonal component.

Key features:

- **Temporal strain evolution:** The strain in each fiber segment is plotted over time, highlighting the propagation of seismic waves. The distinction between parallel (blue) and orthogonal (orange) strain components provides insight into how different segments react to seismic forces.

This visualization helps identify critical regions in the network where strain levels are highest, providing useful information for seismic monitoring.

6.3 SOP Evolution Visualization

The State of Polarization (SOP) evolves in response to the strain-induced birefringence in the fiber network. This section focuses on visualizing the temporal and spatial evolution of the SOP across different fiber segments.

Key features:

- **Temporal SOP evolution:** The Stokes parameters (S_1, S_2, S_3) are plotted over time, providing detailed insights into how the SOP shifts as the seismic waves propagate through the fiber. Each Stokes parameter is plotted separately to track changes in polarization.

- **Poincaré sphere visualization:** The Stokes parameters are also represented on the Poincaré sphere, providing a 3D view of the polarization states. The trajectory traced by the SOP on the Poincaré sphere shows how seismic waves influence the polarization of light in the fiber.

These visualizations allow us to detect shifts in polarization caused by seismic activity, providing valuable insights into the optical network's response.

6.4 SOPAS Visualization

The State of Polarization Angular Speed (SOPAS) measures the rate of change in the SOP, offering a more concise and interpretable representation of polarization dynamics compared to tracking individual Stokes parameters. This section focuses on visualizing SOPAS across the network during seismic events.

Key features:

- **SOPAS over time:** SOPAS is plotted over time for each fiber segment, showing how quickly polarization changes during seismic activity. Sudden spikes in SOPAS may indicate the arrival of seismic waves, particularly the fast-moving P-waves.
- **Comparison across fiber lines:** SOPAS values are compared across different fiber segments, revealing which parts of the network experience the most rapid polarization changes due to seismic forces.

SOPAS visualization is particularly useful for real-time seismic monitoring and early warning, as it highlights sudden polarization changes that could precede larger seismic disturbances.

Chapter 7

Results Analysis

7.1 Introduction

This chapter presents and discusses the results of the simulations conducted on the optical fiber network. The seismic events tested were generated using the `Syngine` service, and the simulations involved three magnitudes: 3.5, 5.0, and 7.0. Three different epicenter positions relative to the network were considered, each providing distinct strain and polarization responses. For clarity and to avoid overwhelming the reader with too many images, only the results for magnitude 5.0 are presented for all cases.

It is important to note that, while the results presented here are indicative and helpful for development purposes, the numerical values should not be interpreted as definitive or final, as further tuning and validation of the simulation model are ongoing. The results serve as a proof of concept for using optical fiber networks as a seismic early warning tool.

The three main cases analyzed in this chapter are:

- **Case 1:** Epicenter near Node A,
- **Case 2:** Epicenter between Node A and Node B,
- **Case 3:** Epicenter 5 km from Node B towards Node C.

Each case will discuss the seismic waveforms generated by `Syngine`, the corresponding strain evolution along key fiber lines, the changes in the SOP, and the behavior of SOPAS over time.

7.2 Case 1: Epicenter Near Node A

This section presents the results for the simulation where the earthquake's epicenter is located near Node A. The analysis focuses on the strain, SOP, and SOPAS evolution in key lines near and far from the epicenter.

7.2.1 Network and Epicenter Visualization

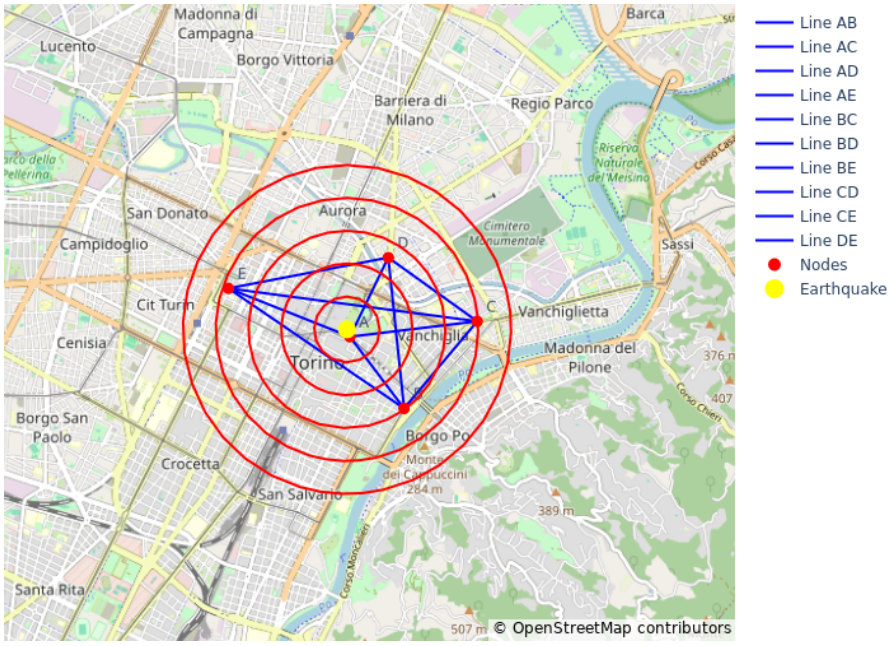

Figure 7.1: Network visualization with the epicenter near Node A.

Figure 7.1 shows the fiber optic network with the epicenter located near Node A. The proximity of the epicenter to line A-B suggests that this segment will experience higher strain values compared to segments farther from the epicenter.

7.2.2 Seismic Waveform (Syngine)

Figure 7.2 presents the seismic waveform generated by Syngine near Node A for a magnitude 5.0 earthquake. This waveform represents the ground displacement due to the earthquake, which is crucial for understanding how strain propagates through the optical fiber network.

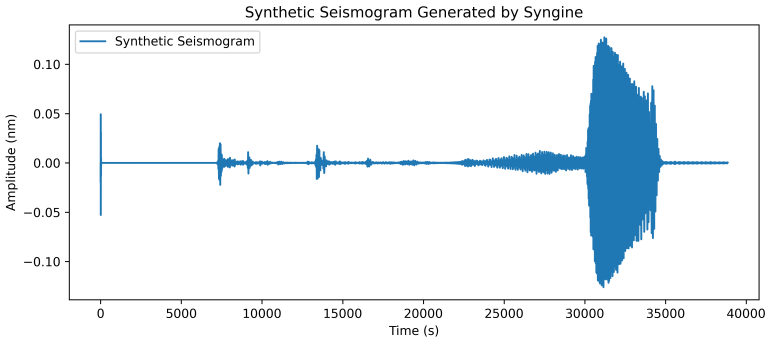

Figure 7.2: Seismic waveform generated by Syngine near Node A.

7.2.3 Strain Evolution in Key Lines

Figure 7.3 shows the strain evolution in line A-B, which is the closest segment to the earthquake epicenter. The blue trace, representing the parallel strain component, is significantly more pronounced than the orthogonal component (orange trace). This indicates that the strain in this segment is predominantly influenced by the parallel component of the seismic wave.

In contrast, Figure 7.4 illustrates the strain evolution in line B-C, a segment located farther from the epicenter. Here, the orthogonal component (orange trace) is more pronounced than the parallel component, suggesting that this line experiences more orthogonal strain due to the orientation of the wave relative to the fiber segment.

Comparison: Interestingly, even though line A-B is closer to the epicenter and experiences more strain overall, the orthogonal component in line B-C is more significant. This difference in wave orientation results in line B-C having a slightly higher SOPAS variation, as discussed in the next section.

7.2.4 State of Polarization (SOP) Changes

Figures 7.5 and 7.6 depicts the State of Polarization (SOP) on the Poincaré Sphere and on the 2d plot for line A-B. The significant shifts in polarization correspond to the strain peaks observed in the strain evolution plot, indicating changes in the polarization state caused by seismic activity.

On the other hand, figures 7.7 and 7.8 shows the State of Polarization for line B-C. The polarization shifts here are much less pronounced compared to line A-B, which reflects the lower strain values in this line.

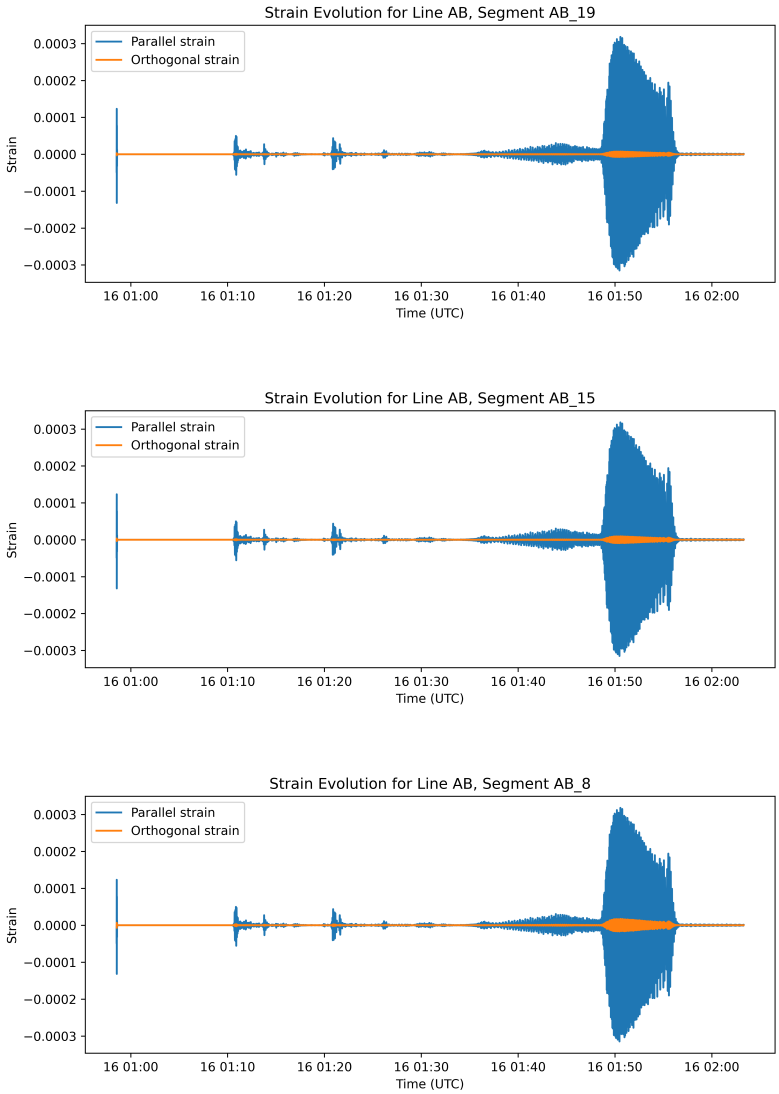

Figure 7.3: Strain evolution in line A-B near Node A.

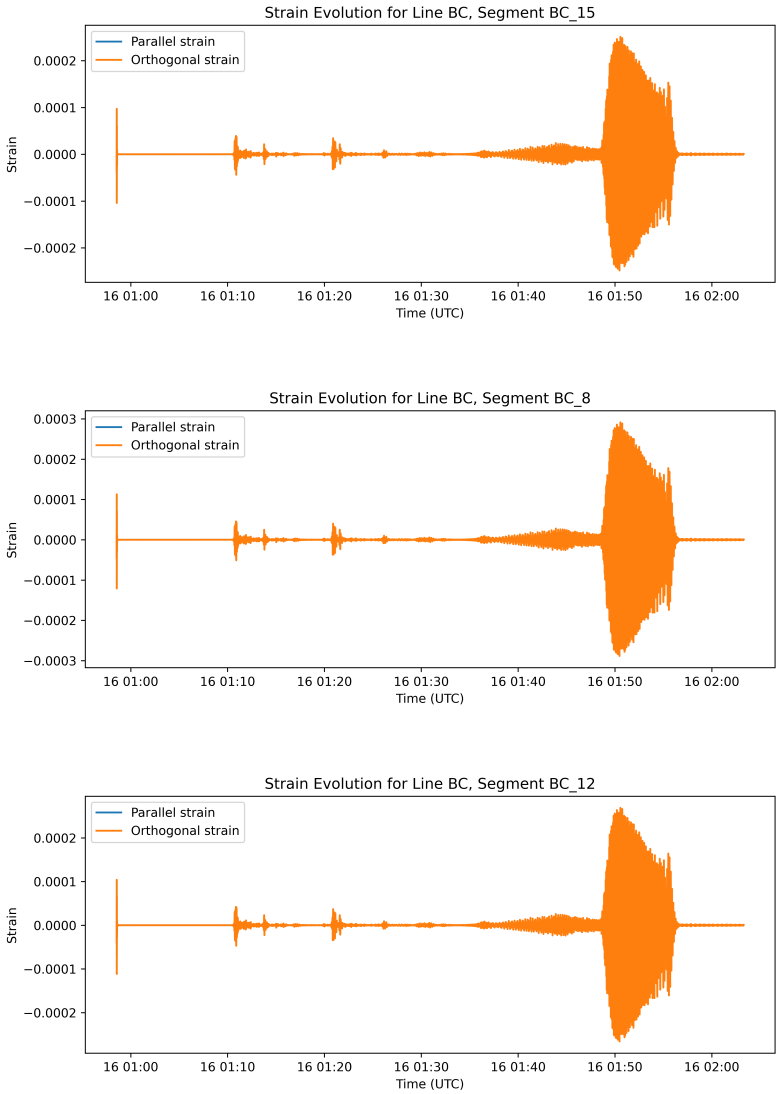

Figure 7.4: Strain evolution in line B-C far from Node A.

7.2.5 SOP Angular Speed (SOPAS)

Figure 7.9 shows the SOP Angular Speed (SOPAS) for line A-B near the epicenter. Although this line experiences high strain, the SOPAS values are lower than

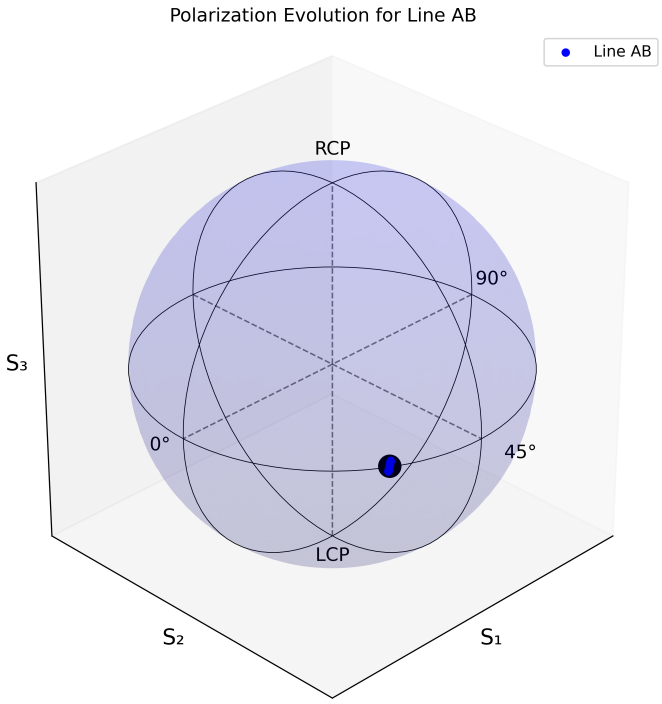

Figure 7.5: State of Polarization (Poincaré Sphere) for line A-B near Node A.

expected because the parallel strain component dominates, which has less effect on the polarization changes compared to orthogonal components.

Interestingly, Figure 7.10 shows that the SOP Angular Speed for line B-C is slightly higher than that of A-B (with a difference on the order of 0.01). This is due to the greater influence of the orthogonal strain component on the polarization changes in line B-C, despite the lower overall strain in this segment.

Comparison: Although line A-B experiences more strain due to its proximity to the epicenter, line B-C shows a greater SOPAS variation. This is a result of the wave orientation, which causes a higher orthogonal strain component on B-C, leading to more significant polarization changes. This contrasts with initial expectations, where we anticipated line A-B to show greater SOPAS due to its higher strain.

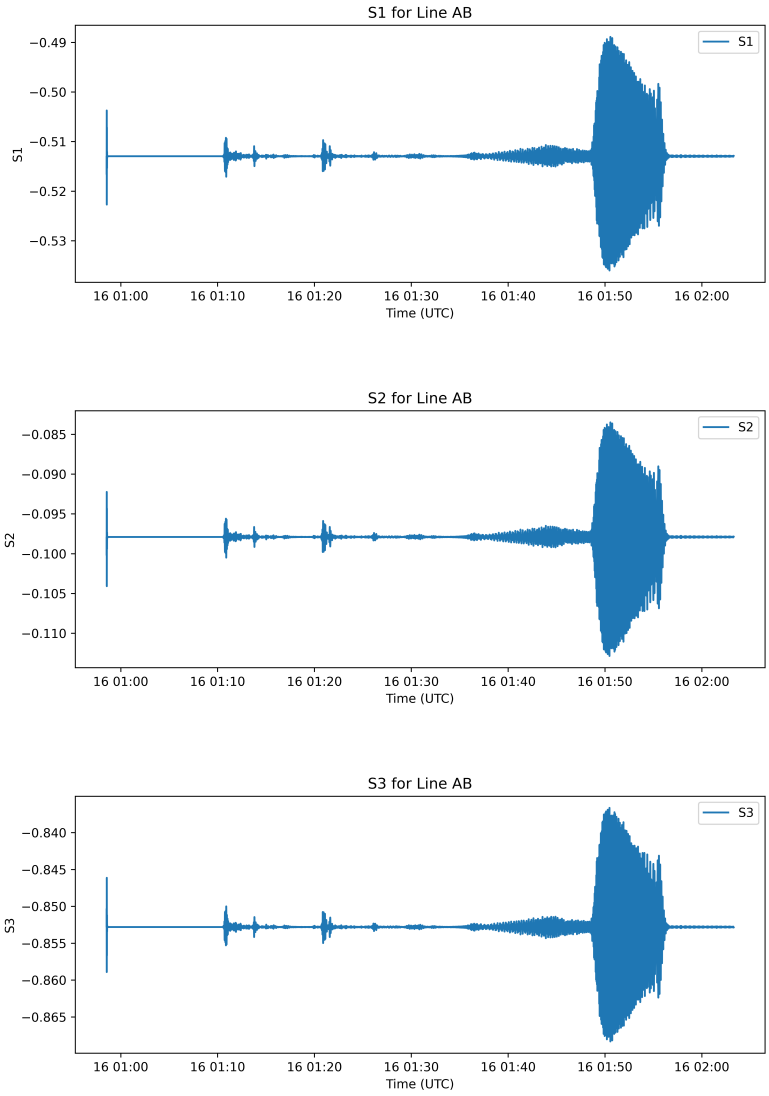

Figure 7.6: SOP (Stokes Parameters) evolution in line A-B near Node A.

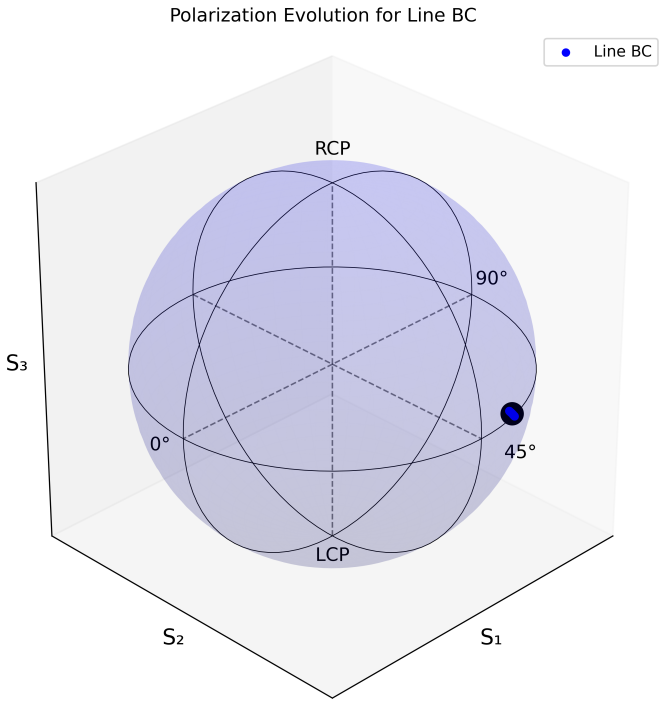

Figure 7.7: State of Polarization (Poincaré Sphere) for line B-C far from Node A.

7.3 Case 2: Epicenter Between Node A and Node B

This section presents the results for the simulation where the earthquake's epicenter is located between Node A and Node B. The analysis focuses on the strain, SOP, and SOPAS evolution in key lines near the epicenter, particularly the line connecting Node A and Node B, and comparisons with lines farther from the epicenter.

7.3.1 Network and Epicenter Visualization

Figure 7.11 shows the fiber optic network with the epicenter located between Node A and Node B. This position suggests that both lines A-B and C-D will experience significant strain, with line A-B being more directly affected due to its proximity to the epicenter.

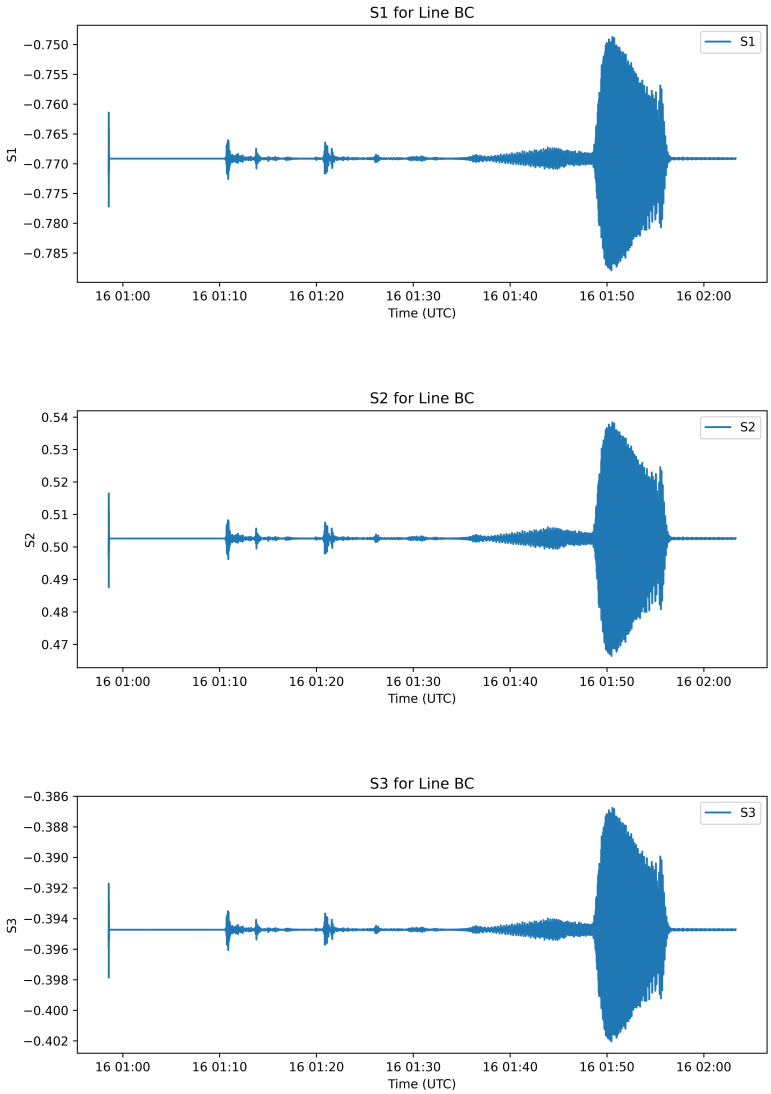

Figure 7.8: SOP (Stokes Parameters) evolution in line B-C far from Node A.

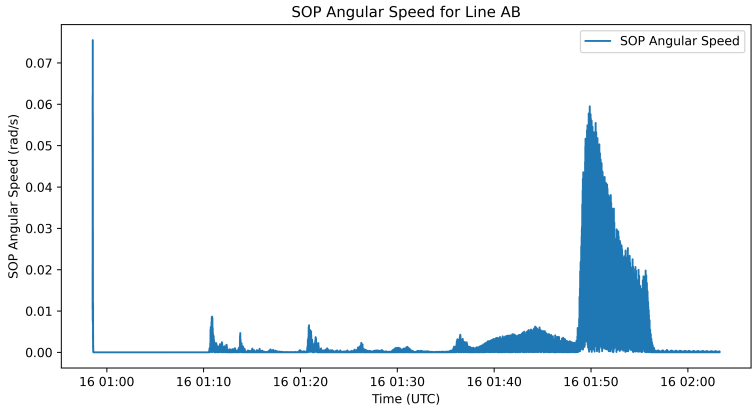

Figure 7.9: SOP Angular Speed (SOPAS) in line A-B near the epicenter.

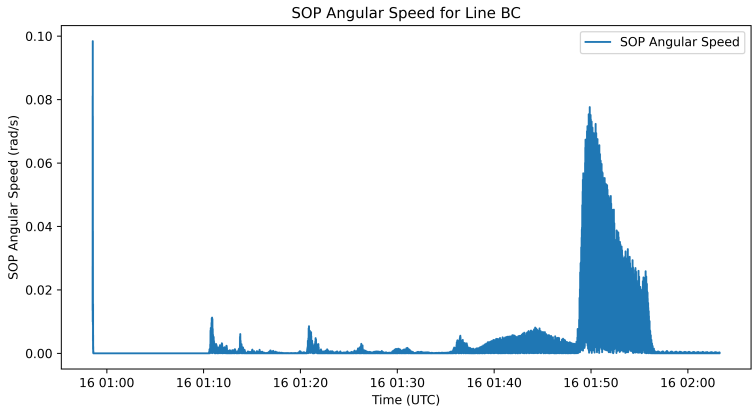

Figure 7.10: SOP Angular Speed (SOPAS) in line B-C far from Node A.

7.3.2 Seismic Waveform (Syngine)

Figure 7.12 presents the seismic waveform generated by Syngine for the earthquake with an epicenter between Node A and Node B. The waveform shows the propagation of seismic energy that impacts the nearby fiber lines.

7.3.3 Strain Evolution in Key Lines

Figure 7.13 shows the strain evolution in line A-B, which is the segment closest to the earthquake epicenter. The strain varies across different segments of the line: at the center, where the epicenter is located, the strain is predominantly orthogonal due to the perpendicular orientation of the seismic wave. Towards the nodes A and

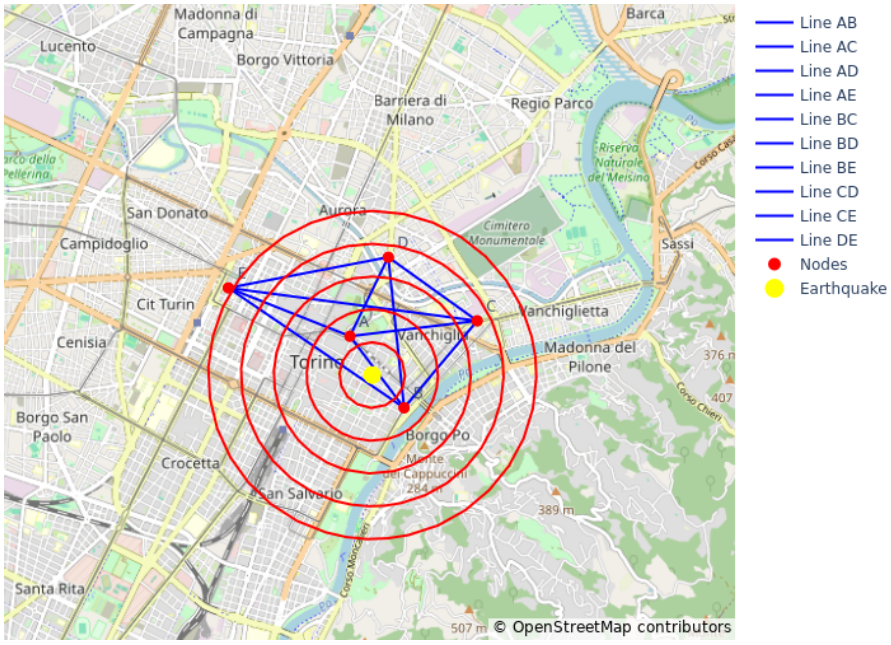

Figure 7.11: Network visualization with epicenter between Node A and Node B.

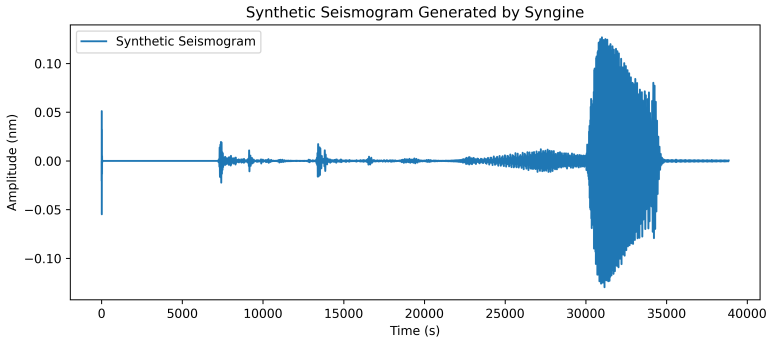

Figure 7.12: Seismic waveform generated by Syngine between Node A and Node B.

B, the strain becomes more parallel as the wave propagates along the line.

In contrast, Figure 7.14 illustrates the strain evolution in line C-D, which is located farther from the epicenter. Here, the strain is predominantly orthogonal across all segments of the line. This is because the distance from the epicenter results in a more uniform wave orientation, causing the strain to be consistently orthogonal to the line.

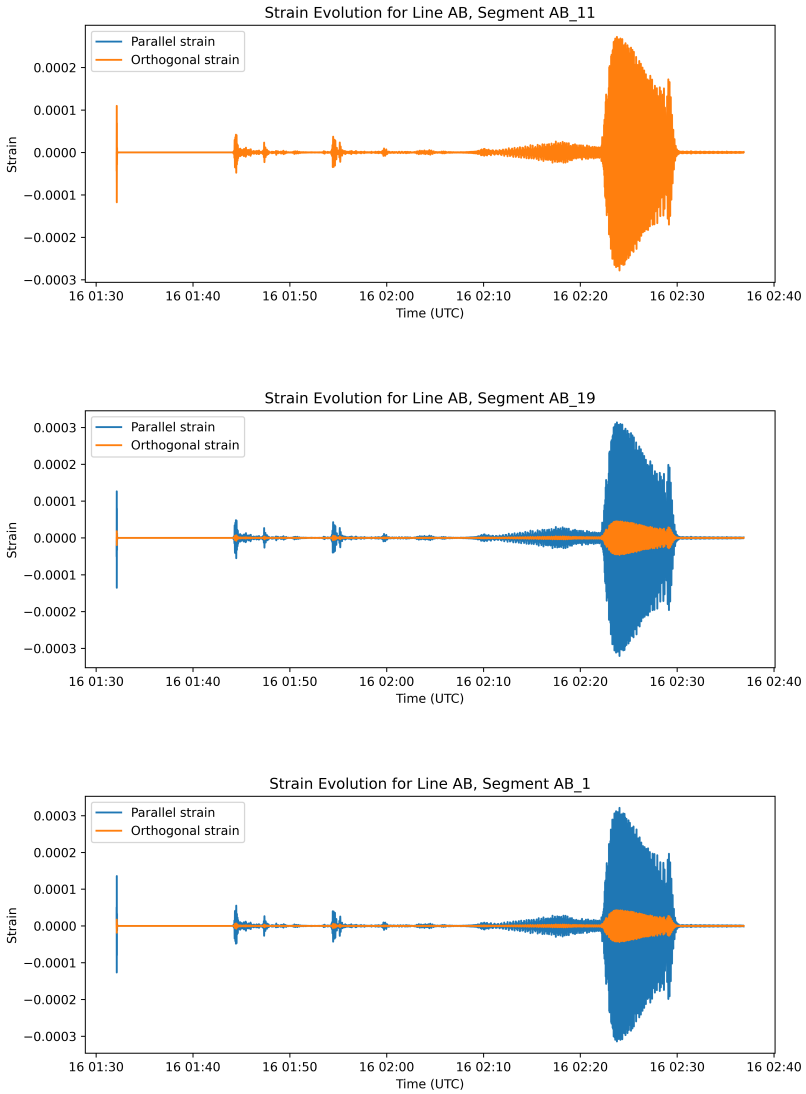

Figure 7.13: Strain evolution in line A-B near the epicenter.

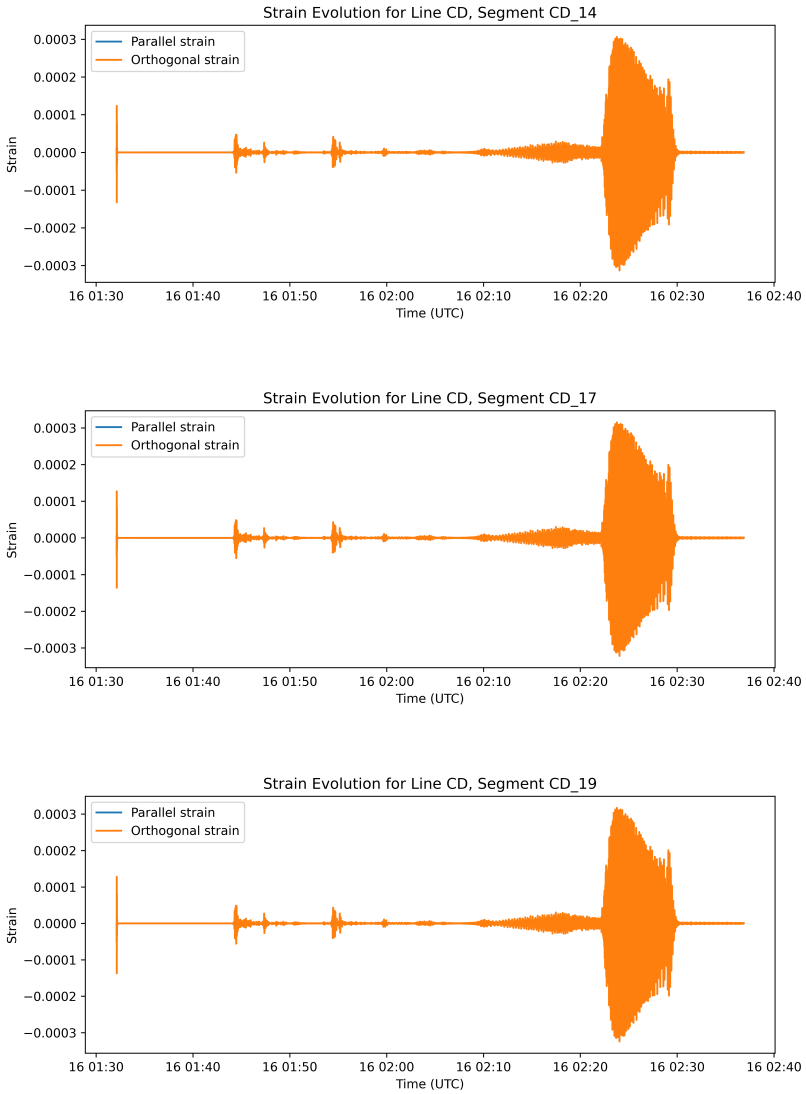

Figure 7.14: Strain evolution in line C-D farther from the epicenter.

7.3.4 State of Polarization (SOP) Changes

In line A-B, the SOP changes displayed on the Poincaré Sphere (Fig. 7.15) show significant shifts, correlating with the peaks in strain near the epicenter. In contrast,

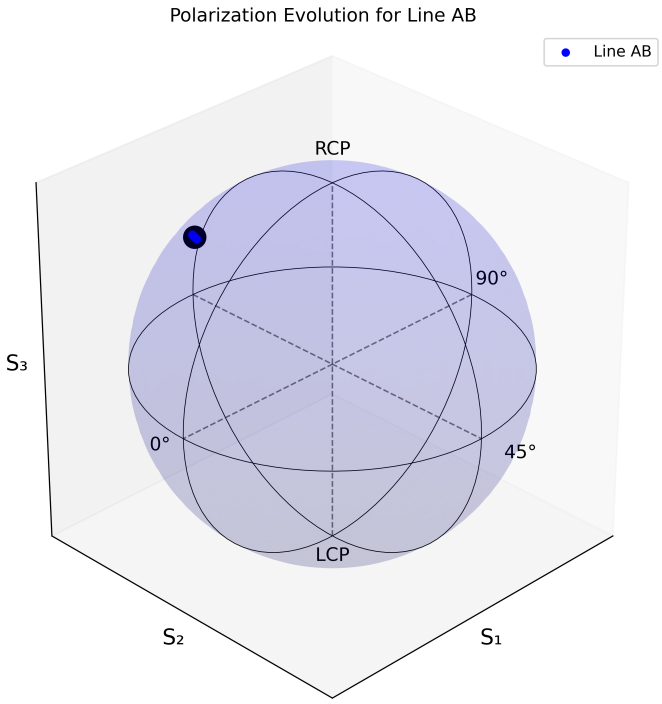

Figure 7.15: State of Polarization (Poincaré Sphere) for line A-B between Node A and Node B.

line C-D (Fig. 7.17), located farther from the epicenter, exhibits more gradual polarization changes due to lower strain levels.

The evolution of Stokes parameters mirrors this behavior: line A-B (Fig. 7.16 and 7.16) shows faster parameter changes, while line C-D (Fig. 7.18 and 7.18) evolves more slowly, reflecting the reduced strain and less significant polarization variations.

7.3.5 SOP Angular Speed (SOPAS)

Figure 7.19 shows the SOP Angular Speed (SOPAS) for line A-B near the epicenter. The contribution of the parallel strain components, particularly towards nodes A and B, has increased the SOPAS. This allows for the distinction of various seismic waves, such as the primary (P) waves, secondary (S) waves, and surface waves.

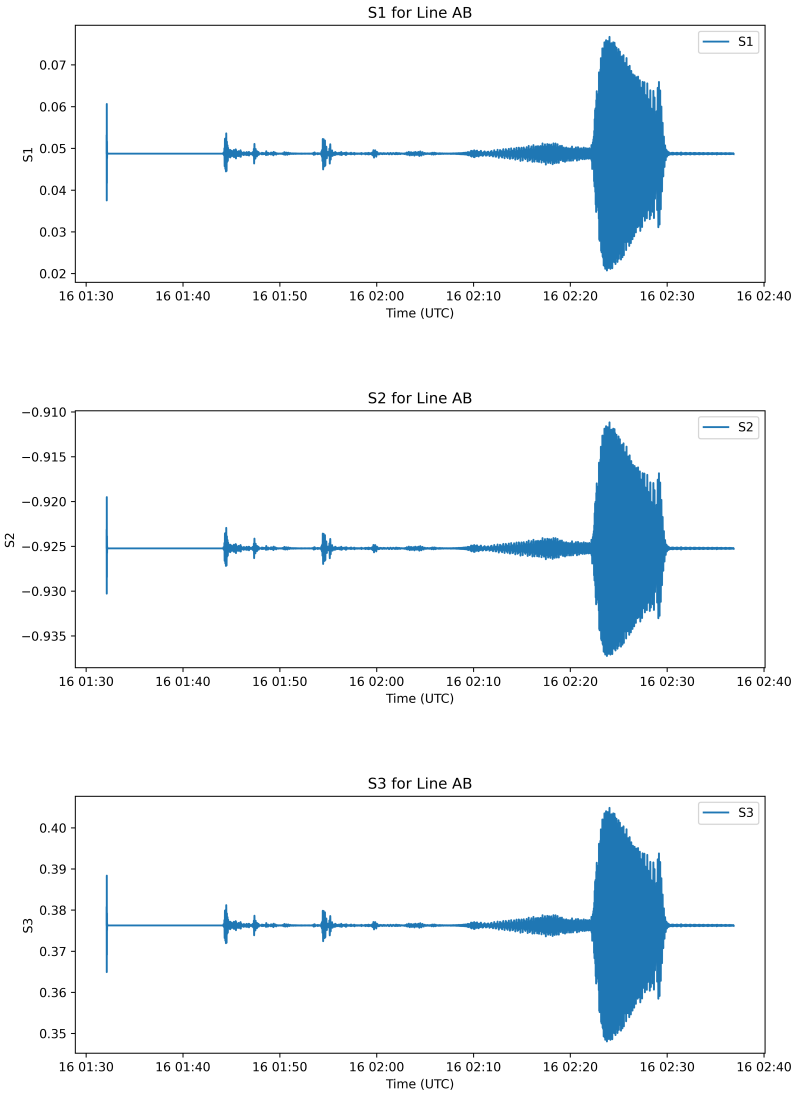

Figure 7.16: SOP (Stokes Parameters) evolution in line A-B between Node A and Node B.

In contrast, Figure 7.20 shows the SOP Angular Speed for line C-D, which is farther from the epicenter. The SOPAS is lower here because the strain is

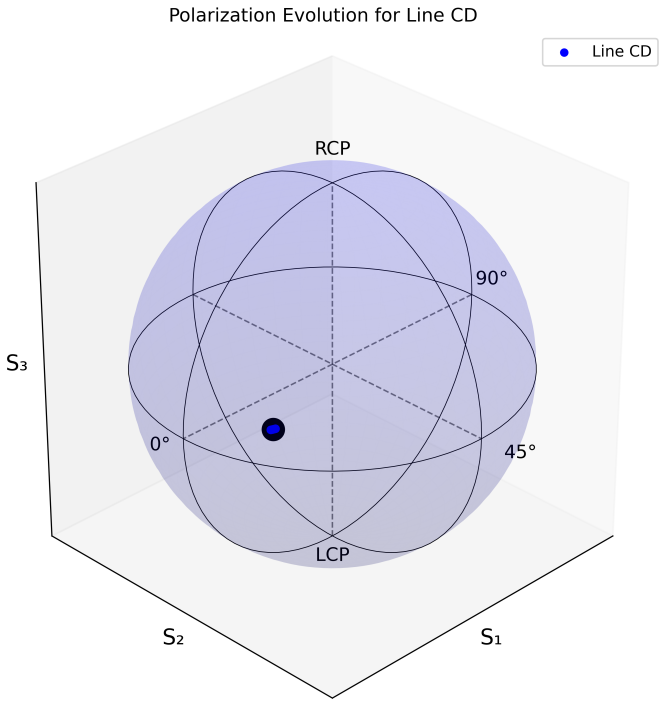

Figure 7.17: State of Polarization (Poincaré Sphere) for line C-D farther from the epicenter.

predominantly orthogonal across the line, resulting in less significant polarization changes.

7.4 Case 3: Epicenter 5 km from Node B Towards Node C

This section presents the results for the simulation where the earthquake's epicenter is located 5 km from Node B, towards Node C. The analysis focuses on the strain, SOP, and SOPAS evolution in key lines near the epicenter, particularly the line B-C, with comparisons to lines farther from the epicenter.

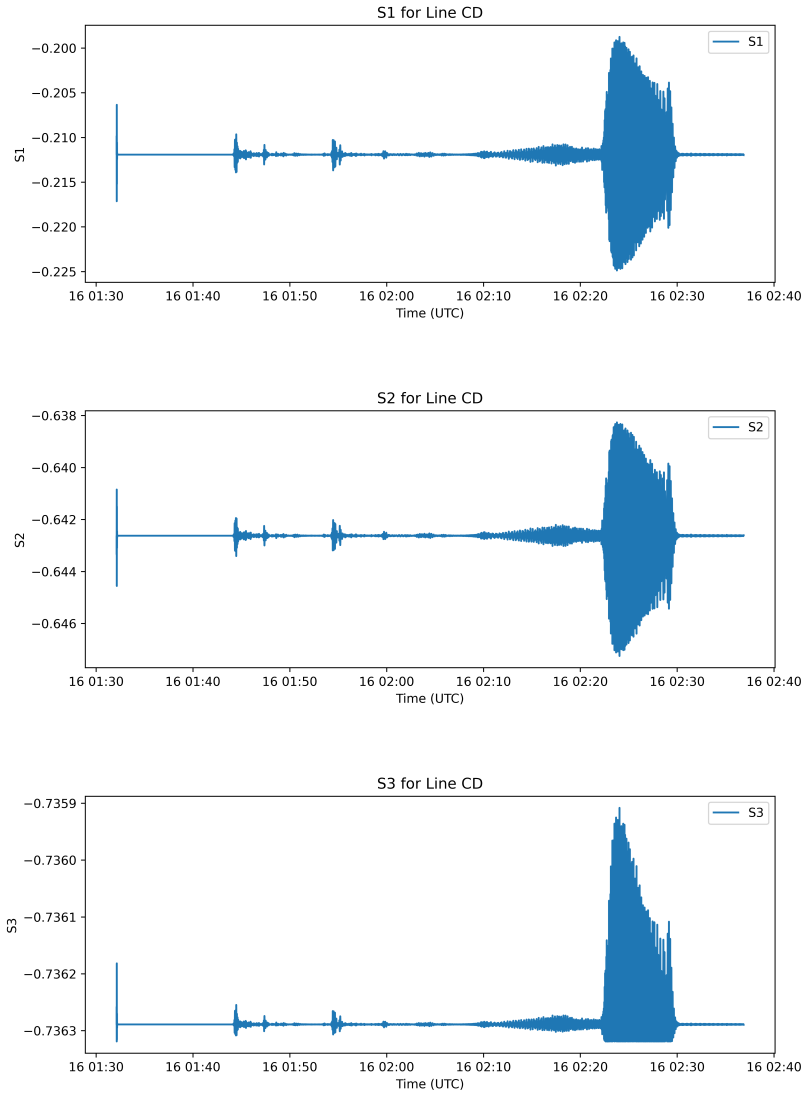

Figure 7.18: SOP (Stokes Parameters) evolution in line C-D farther from the epicenter.

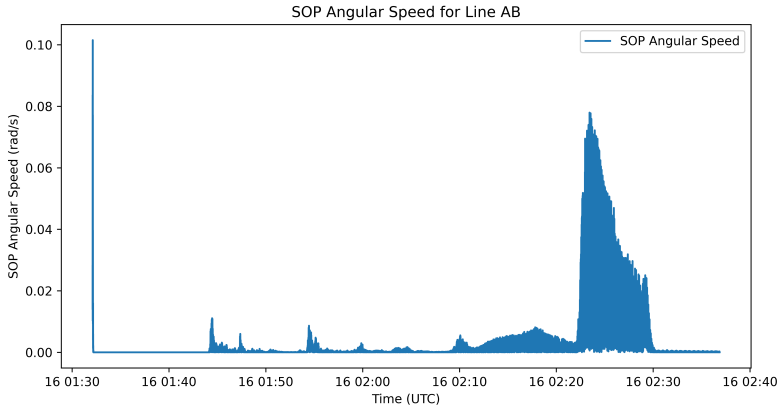

Figure 7.19: SOP Angular Speed (SOPAS) in line A-B between Node A and Node B.

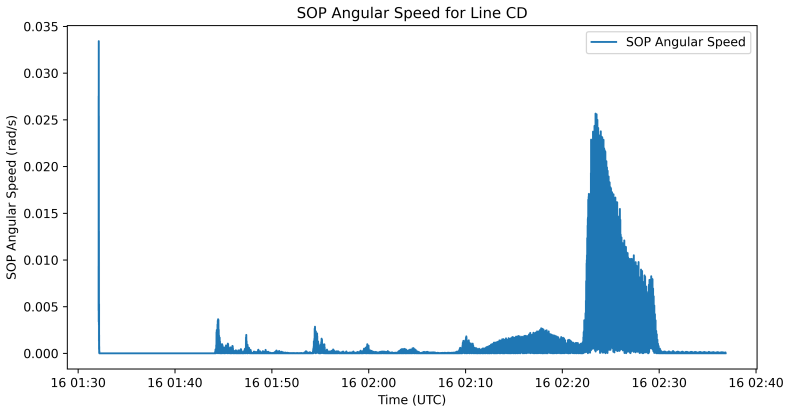

Figure 7.20: SOP Angular Speed (SOPAS) in line C-D farther from the epicenter.

7.4.1 Network and Epicenter Visualization

Figure 7.21 shows the fiber optic network with the epicenter located 5 km from Node B, towards Node C. The proximity of the epicenter to line B-C suggests that this line will experience higher strain values compared to line A-B, which is farther from the epicenter.

7.4.2 Seismic Waveform (Syngine)

Figure 7.22 presents the seismic waveform generated by Syngine near Node B for a magnitude 5.0 earthquake. The waveform shows how the ground displacement

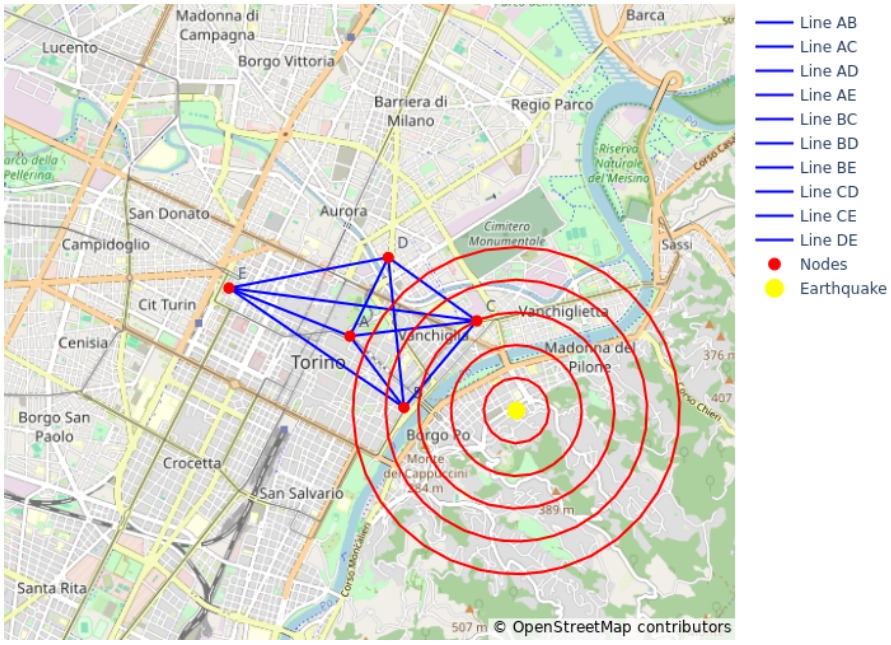

Figure 7.21: Network visualization with epicenter 5 km from Node B towards Node C.

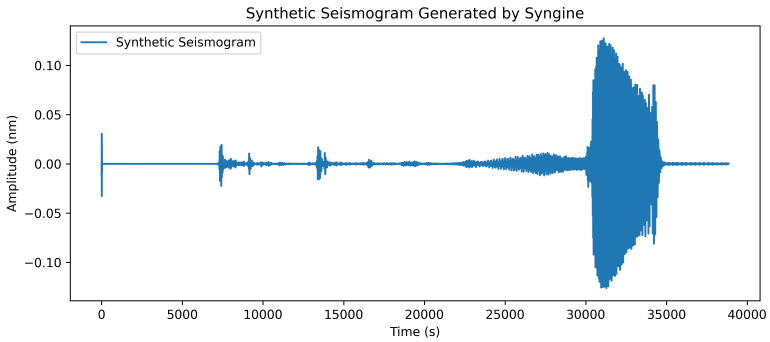

Figure 7.22: Seismic waveform generated by Syngine 5 km from Node B towards Node C.

propagates towards the optical fiber segments in this region.

7.4.3 Strain Evolution in Key Lines

Figure 7.23 shows the strain evolution in line B-C, which is located closest to the epicenter. The plot indicates that both parallel and orthogonal strain components are present, with the orthogonal component being more pronounced. This suggests

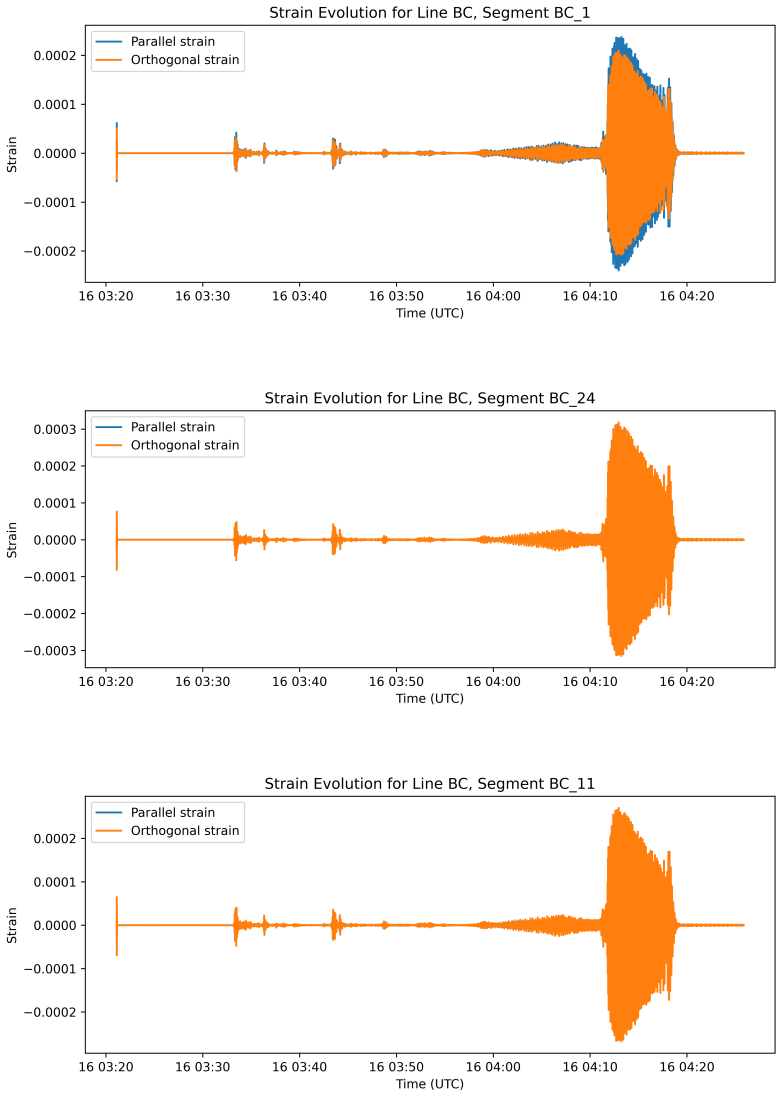

Figure 7.23: Strain evolution in line B-C near the epicenter.

that the orientation of the seismic wave relative to line B-C causes more orthogonal strain.

Figure 7.24 illustrates the strain evolution in line A-B, which is farther from

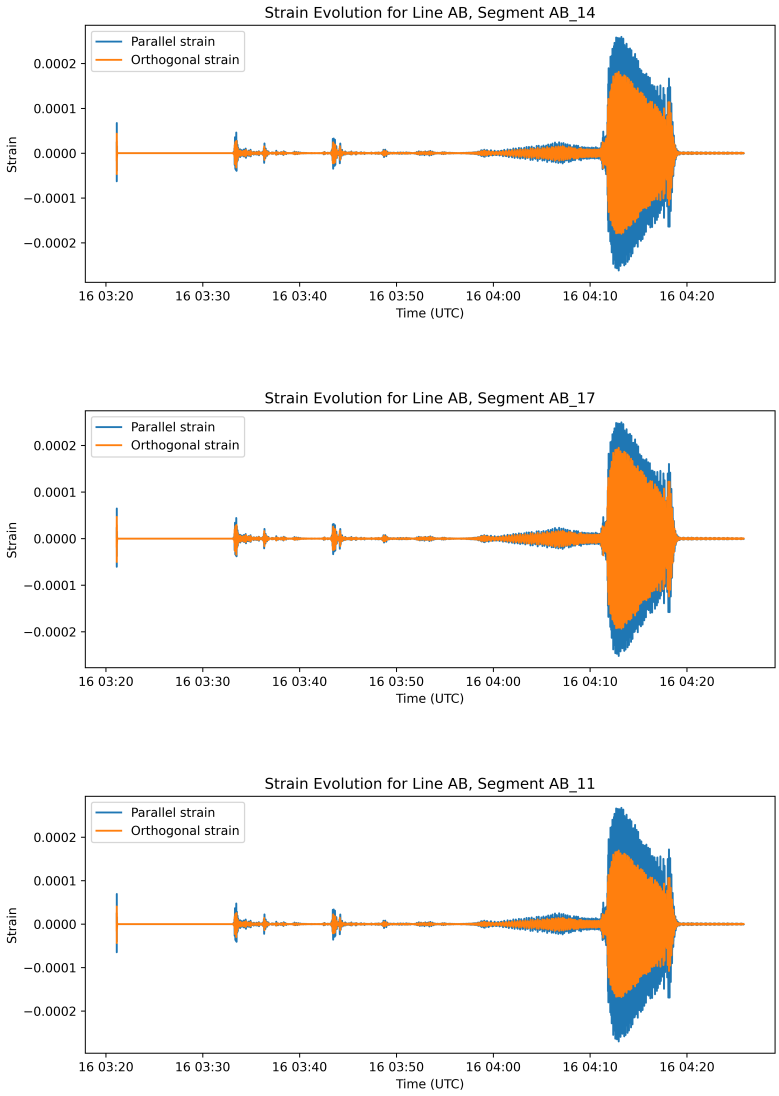

Figure 7.24: Strain evolution in line A-B farther from the epicenter.

the epicenter. As expected, the strain values here are lower, and the parallel strain component dominates due to the relative distance and wave orientation.

Comparison: The strain in line B-C near the epicenter is higher and more

orthogonal compared to line A-B. This is expected given the location of the epicenter and the wave propagation direction. However, the parallel strain component in line A-B still plays a significant role, particularly in segments farther from the epicenter.

7.4.4 State of Polarization (SOP) Changes

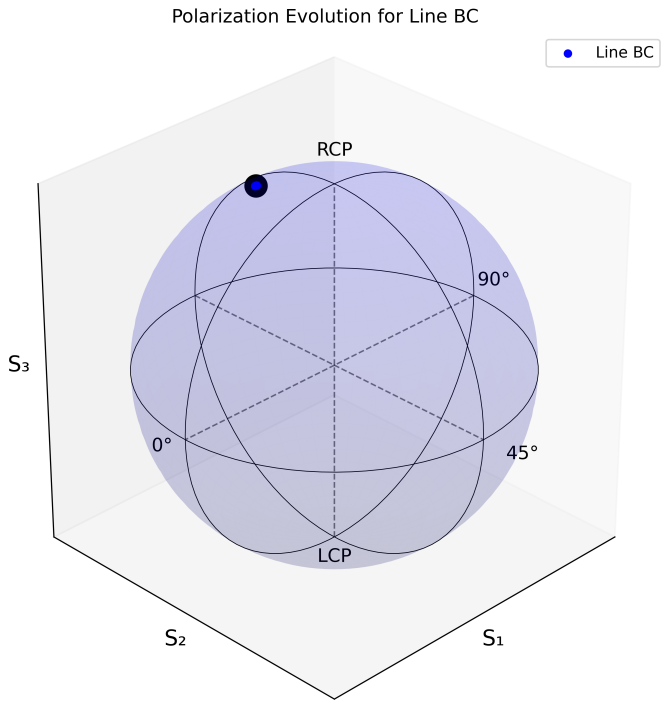

Figure 7.25: State of Polarization (Poincaré Sphere) for line B-C near the epicenter.

Figures 7.25 and 7.26 show the State of Polarization (SOP) changes on the for line B-C. The significant polarization shifts reflect the higher orthogonal strain in this segment, resulting in a slow polarization changes.

Figures 7.27 and 7.28 show the SOP for line A-B, where polarization shifts are less pronounced due to lower strain levels and the dominance of the parallel strain component.

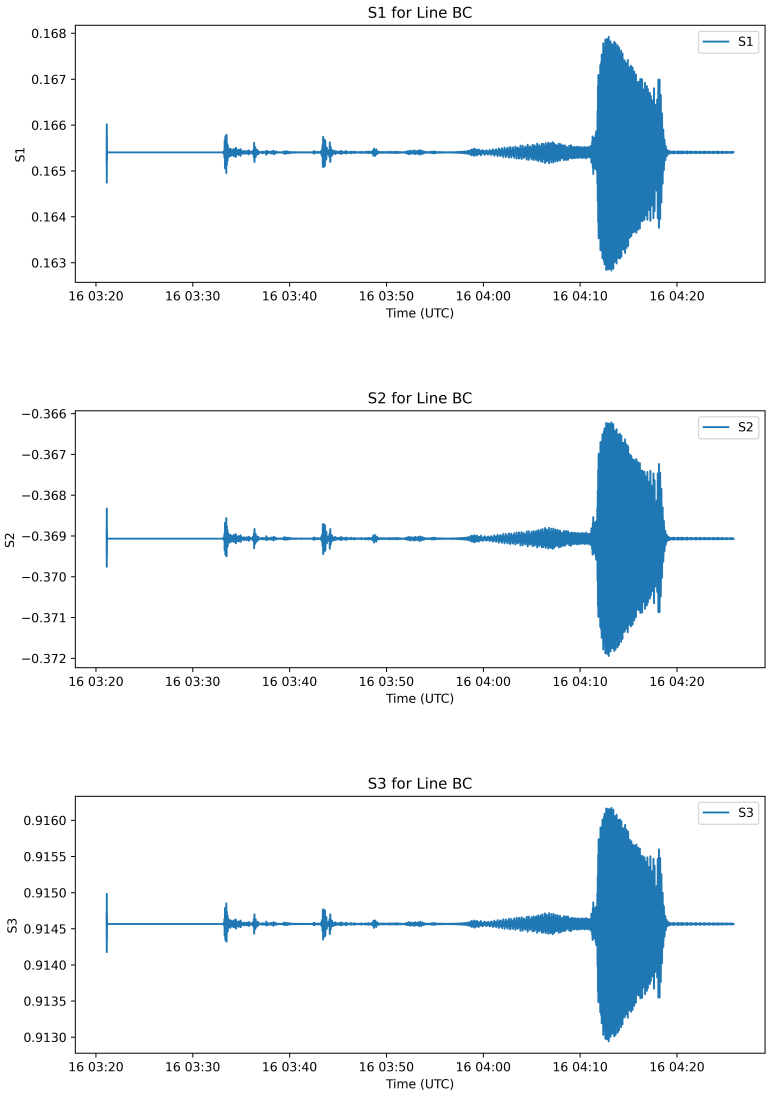

Figure 7.26: SOP (Stokes Parameters) evolution in line B-C near the epicenter.

Comparison: The polarization changes in line B-C are more significant compared to line A-B. This is consistent with the higher orthogonal strain in B-C, which has a greater effect on SOP than the parallel strain in A-B.

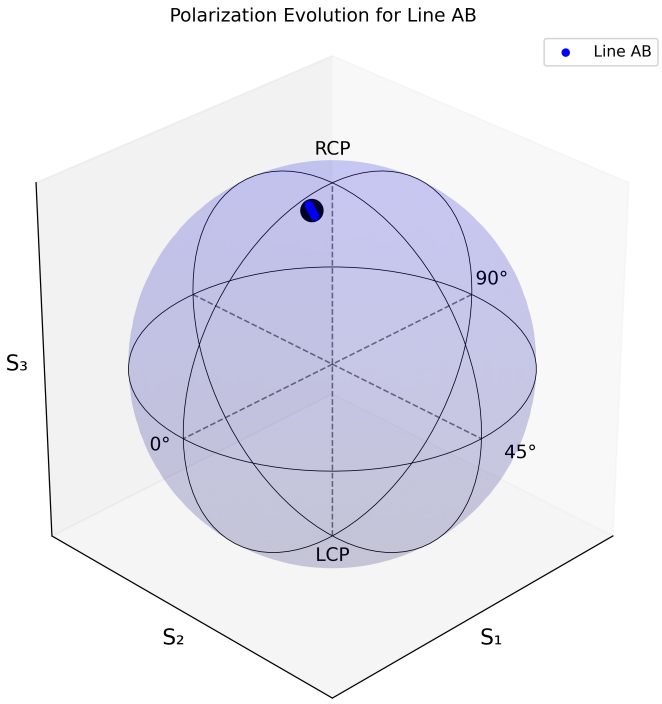

Figure 7.27: State of Polarization (Poincaré Sphere) for line A-B farther from the epicenter.

7.4.5 SOP Angular Speed (SOPAS)

Figure 7.29 shows the SOP Angular Speed (SOPAS) for line B-C near the epicenter. The variation in SOPAS is higher here due to the greater influence of the orthogonal strain component, which accelerates the changes in polarization.

In contrast, Figure 7.30 shows the SOPAS in line A-B, which is farther from the epicenter. The SOPAS variation is lower in this line due to the dominance of the parallel strain component, which has less impact on polarization changes compared to orthogonal strain.

Comparison: The SOP Angular Speed in line B-C is higher than in line A-B, consistent with the higher orthogonal strain component in B-C. The parallel strain in A-B leads to slower changes in SOPAS, as expected.

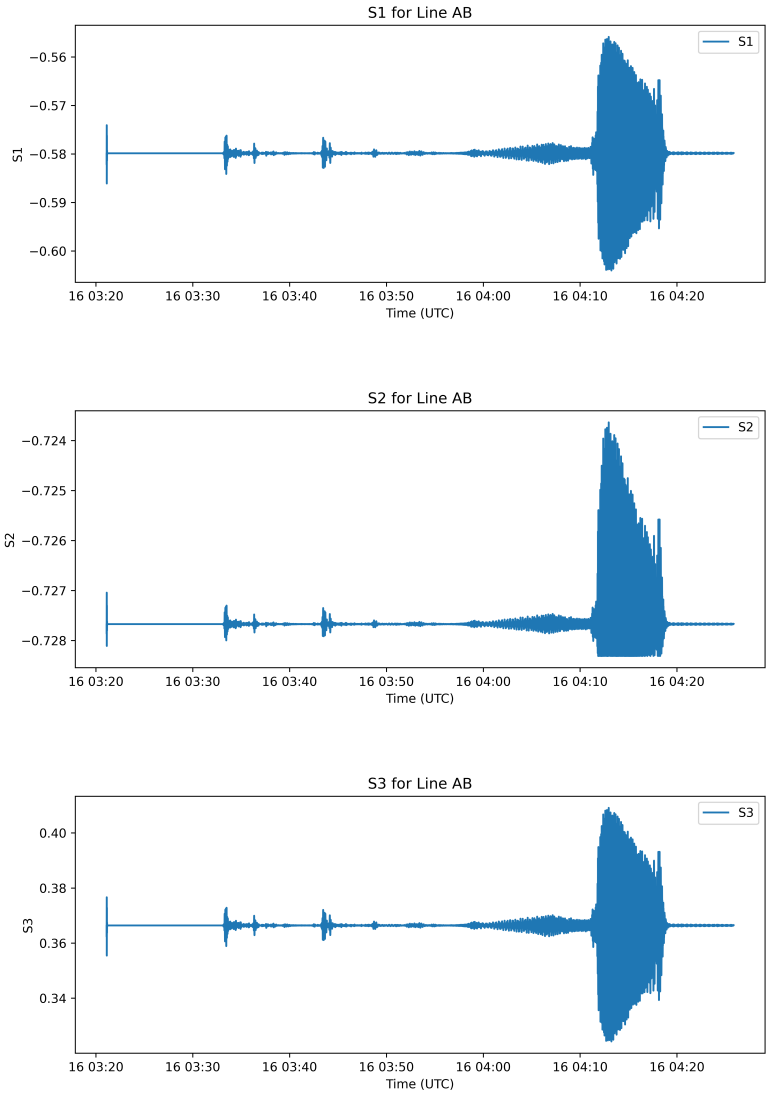

Figure 7.28: SOP (Stokes Parameters) evolution in line A-B farther from the epicenter.

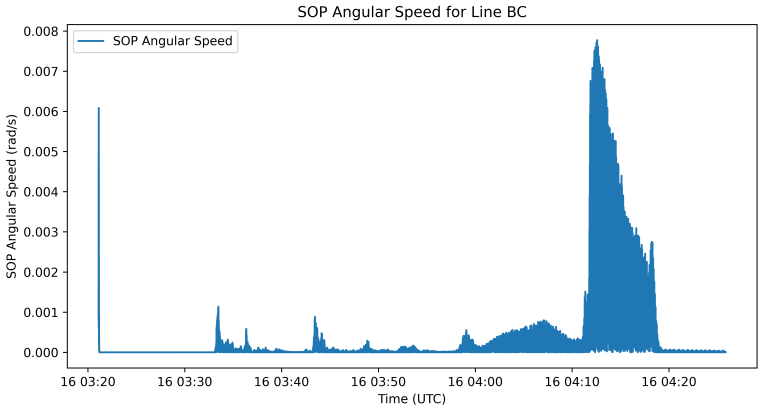

Figure 7.29: SOP Angular Speed (SOPAS) in line B-C near the epicenter.

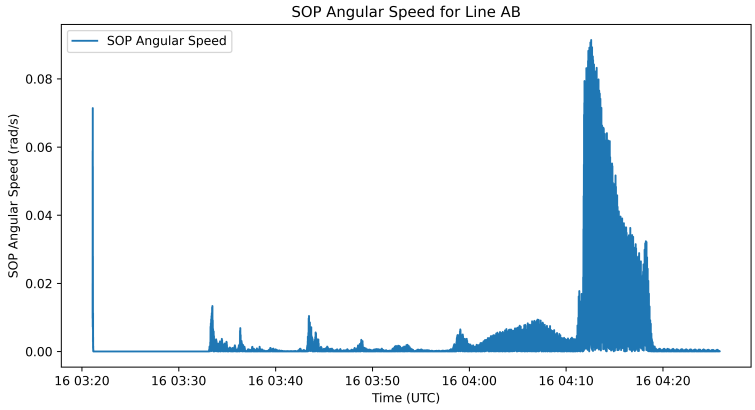

Figure 7.30: SOP Angular Speed (SOPAS) in line A-B farther from the epicenter.

7.5 Comparison of Results Across Cases

This section provides a comparative analysis of the results across the three cases. Differences in strain evolution, SOP changes, and SOPAS are discussed, highlighting how the epicenter position and earthquake magnitude influence the network's response. The orthogonal strain component has been shown to play a more significant role in polarization changes, particularly in lines located near the epicenter, while parallel strain components contribute less to the overall SOP variation.

Chapter 8

Conclusion and future developments

8.1 Summary of Key Outcomes

This thesis has presented a framework for seismic detection using an optical fiber network, leveraging the Waveplate Model to track polarization changes induced by seismic strain. Key advancements achieved include:

- **Seismic Detection Through Optical Networks**: The use of existing optical networks to detect seismic activity without additional infrastructure costs was a significant breakthrough. The method enables the detection of ground displacements through polarization analysis, providing real-time feedback for early warning systems.
- **Waveplate Model Integration**: The application of the Waveplate Model allowed for precise simulation of birefringence changes in fiber optic cables caused by seismic waves. This model effectively translated the mechanical strain induced by seismic events into measurable changes in the State of Polarization (SOP), offering a possible method for continuous seismic monitoring.
- **Computational Optimization**: Significant improvements in simulation performance were achieved through the optimization of strain evolution matrices and the parallelization of key computational tasks. The development of a matrix-based representation of strain evolution reduced computational overhead, allowing for real-time processing of seismic events in large-scale networks.
- **Simulation Flexibility and Scalability**: The framework's flexibility in handling different magnitudes and epicenter locations across various network

configurations demonstrated its potential for wide-scale deployment. The modular structure of the Simulation Manager facilitated the simulation of complex seismic scenarios, ensuring scalability for future extensions and real-world applications.

8.2 Challenges and Technical Limitations

While this research has demonstrated the potential of optical networks for seismic detection, several challenges and technical limitations were encountered during the development of the framework:

- **Computational Constraints**: The real-time processing of seismic events requires significant computational resources, particularly for large-scale networks. Despite optimization efforts, the strain evolution calculations and matrix-based simulations of polarization effects are computationally intensive, especially when simulating multiple seismic events simultaneously. The reliance on GPU-based acceleration mitigates these issues but introduces additional complexity in terms of hardware requirements.

- **Simulation Manager Limitations**: The Simulation Manager, while flexible, faces limitations in handling more advanced parallelization and fault tolerance during simulations. Handling large datasets, particularly for real-time applications, poses challenges in terms of memory usage and scalability. Future work could focus on improving the efficiency of the manager, optimizing data caching, and incorporating more sophisticated error-handling mechanisms.

- **Data Availability and Accuracy**: The accuracy of the seismic wave simulation is heavily dependent on the availability of high-resolution seismic data from external sources, such as Syngine. Limited access to precise real-time seismic data could affect the robustness of the simulations, particularly for smaller magnitude events or remote regions with limited sensor coverage. If Syngine (or an equivalent tool) were available as a local library or software, it would eliminate the need for repeated queries to external servers, thereby improving efficiency and reliability.

- **Fiber Network Infrastructure**: Although the use of existing optical fiber infrastructure reduces costs, it also introduces uncertainties due to the varying quality and age of the cables. Differences in fiber construction, installation, and environmental conditions may introduce unexpected birefringence effects, which are difficult to predict and model accurately.

8.3 Potential Future Improvements

Several future improvements could enhance the framework's performance, scalability, and real-time applicability:

- **Further Parallelization and GPU Optimization**: While the current system already utilizes the CPU for matrix operations, implementing parallelization to execute multiple processes simultaneously could significantly enhance simulation speed. Additionally, leveraging GPU optimization could further improve performance, especially for larger networks.
- **Enhanced Log Management**: Currently, the log system tracks changes in parameters, ensuring consistency across simulations. The cache system operates with a memory of 1, meaning it evaluates whether the current simulation is identical to the previous one. If any differences are detected, the system updates the cache accordingly. While this approach is efficient, it limits the ability to save multiple simulations. Expanding the cache to store more simulations could provide greater flexibility and historical data for analysis, but it would require significantly more memory. In the future, more detailed logs, such as performance metrics, memory usage, and detailed error reports, could also be included for debugging and optimization.
- **Support for Multiple Earthquake Events**: The current simulation manager handles one earthquake at a time. Extending the system to manage multiple seismic events occurring simultaneously would improve the realism of the simulation.
- **Dynamic Network Adjustments**: The simulation assumes that the network structure remains fixed during the simulation. Future iterations could incorporate dynamic adjustments, allowing the simulation to adapt to real-time changes in the network (e.g., fiber cuts or rerouting).
- **Advanced Parallelization**: Future work could focus on implementing more advanced parallelization techniques to improve computational efficiency, especially for large-scale simulations involving multiple seismic events. This would allow the system to handle real-time data streams from extensive networks with reduced latency.
- **Integration with Machine Learning**: Machine learning algorithms could be integrated into the framework to improve the accuracy of seismic wave detection and polarization change prediction. By training models on historical data, the system could potentially enhance its ability to detect subtle seismic events.

- **Real-Time Monitoring and Alerts**: Extending the system to provide real-time monitoring capabilities, with automatic alerts for seismic events, could make the framework more practical for real-world applications. This would require the integration of real-time data feeds and optimization of the simulation process to provide immediate feedback.
- **Adaptation to Different Fiber Networks**: Adapting the framework to function with different types of optical fiber networks, including older or lower-quality infrastructure, could broaden its applicability. This would involve further research into how varying fiber properties affect birefringence and SOP changes during seismic events.

8.4 Final Thoughts

The work presented in this thesis demonstrates the feasibility of using optical fiber networks for seismic detection, offering a novel approach to leveraging existing telecommunication infrastructure for early warning systems. By integrating the Waveplate Model, the framework accurately simulates the strain-induced polarization changes caused by seismic events, providing a powerful tool for real-time monitoring.

While there are challenges, such as computational limits and data availability, the flexibility and scalability of the simulation framework offer vast potential for further development. With continued advancements, this approach could play a crucial role in enhancing global earthquake monitoring and response systems.

This thesis lays the groundwork for future research into fiber-optic-based seismic detection, paving the way for more efficient, scalable, and precise systems capable of providing early warnings in areas vulnerable to earthquakes. As fiber optic networks continue to expand globally, the potential for applying this technology to real-world scenarios grows, making it an exciting frontier for innovation in seismic monitoring.

Appendix A

elements.py

code/elements.py

```
import json
import math
import random
from pyrocko import moment_tensor as pmt
from xmlrpc.client import Error
from pathlib import Path
import hashlib
import shutil
import matplotlib.pyplot as plt
import plotly.graph_objs as go
from math import radians, degrees, sin, cos, atan2, sqrt
from datetime import datetime, timedelta
import requests
import zipfile
import os
from obspy import read
import pandas as pd
from mpl_toolkits.mplot3d import Axes3D
import pypolar.jones as jones
import pypolar.visualization as vis
from py_pol.utils import degrees
from py_pol.jones_matrix import Jones_vector
from pyrocko.trace import project
from scipy.interpolate import interp1d
from sympy.codegen.ast import Raise
import time


try:
    import cupy as xp  # For GPU computation
    use_gpu = True
```

```
    print("CuPy is available. Running on GPU.")
except ImportError:
    import numpy as xp  # For CPU computation
    use_gpu = False
    print("CuPy is not available. Running on CPU.")
import numpy as np

resources_path = Path(__file__).parent / 'resources'
cache_path = Path(__file__).parent / 'cache'
results_path = Path(__file__).parent / 'results'
output_dir = None

class Node:
    def __init__(self, id, position, stokes_params):
        self.id = id
        self.position = position
        self.stokes_params = stokes_params

    @staticmethod
    def load_from_json(file_path):
        with open(file_path, 'r') as f:
            data = json.load(f)
            nodes = [Node(id=node['id'], position=node['position'], stokes_params=node['stokes_params']) for node in data['nodes']]
        return nodes

    def plot(self):
        # Logic to plot nodes on a map
        pass


class Segment:
    def __init__(self, id, start_position, end_position):
        self.id = id
        self.start_position = start_position
        self.end_position = end_position
        self.strain_evolution = None  # Load strain evolution if it exists
        self.timestamp = None  # Timestamp when the seismic wave reaches this segment
        self.segment_wave = None  # Load segment wave if it exists

    def initialize_strain_evolution(self, num_time_steps):
        if self.strain_evolution is None:
            # Initialize strain evolution matrix: [time, (parallel, orthogonal)]
            self.strain_evolution = np.zeros((num_time_steps, 2))

```

```
77
78      def load_segment_wave_cache(self):
79          # Load the wave data from cache if it exists, otherwise
    return None
80          cache_file = f"cache/wave_cache/segment_wave_{self.id}.npy
    "
81          if os.path.exists(cache_file):
82              return np.load(cache_file)
83          return None
84
85      def save_segment_wave_cache(self):
86          # Ensure wave_cache directory exists
87          os.makedirs("cache/wave_cache", exist_ok=True)
88          # Save the wave data to cache
89          cache_file = f"cache/wave_cache/segment_wave_{self.id}.npy
    "
90          np.save(cache_file, self.segment_wave)
91
92      def load_strain_cache(self):
93          # Load strain evolution data from cache if it exists
94          cache_file = f"cache/strain_cache/strain_evolution_{self.
    id}.npy"
95          if os.path.exists(cache_file):
96              return np.load(cache_file)
97          return None
98
99      def save_strain_cache(self):
100         # Ensure strain_cache directory exists
101         os.makedirs("cache/strain_cache", exist_ok=True)
102         # Save the strain evolution to cache
103         cache_file = f"cache/strain_cache/strain_evolution_{self.
    id}.npy"
104         np.save(cache_file, self.strain_evolution)
105
106     def save_all_caches(self):
107         # Save all cache data
108         if self.segment_wave is not None:
109             self.save_segment_wave_cache()
110         if self.strain_evolution is not None:
111             self.save_strain_cache()
112
113
114
115 class Line:
116     def __init__(self, id, start_node, end_node, segment_length
    =500, waveplate_interval=4):
117         self.id = id
118         self.start_node = start_node
119         self.end_node = end_node
```

```
120         self.segment_length = segment_length
121         self.length = haversine_distance(np.array(self.start_node.
    position), np.array(self.end_node.position))
122         self.segments = self.create_segments()
123         self.waveplate_interval = waveplate_interval
124         self.num_waveplates = math.ceil(self.length / self.
    waveplate_interval)
125         self.initial_polarization_state = None  # new attribute to
     store the initial polarization state
126         self.final_polarization_state = None  # new attribute to
    store the final polarization state
127         self.stokes_evolution = None  # new attribute to store the
     stokes evolution
128         self.sop_angular_speed = None  # new attribute to store
    the SOP angular speed
129         self.num_segments=len(self.segments)
130 
131     @staticmethod
132     def load_from_json(file_path, nodes, segment_length,
    waveplate_interval= 4):
133         with open(file_path, 'r') as f:
134             data = json.load(f)
135             lines = [
136                 Line(
137                     id=line['id'],
138                     start_node=nodes[line['start_node']],
139                     end_node=nodes[line['end_node']],
140                     segment_length=segment_length,
141                     waveplate_interval=waveplate_interval
142                 ) for line in data['lines']
143             ]
144         return lines
145 
146     def initialize_stokes_parameters(self, num_time_steps):
147         # Initialize SOP evolution matrix: [time, (S1, S2, S3)]
148         self.stokes_evolution = np.zeros((num_time_steps, 3))
149 
150 
151     def create_segments(self):
152         segments = []
153         start_position = np.array(self.start_node.position)
154         end_position = np.array(self.end_node.position)
155         total_distance = self.length
156         direction = (end_position - start_position) /
    total_distance
157         num_segments = int(total_distance // self.segment_length)
158 
159         for i in range(num_segments):
```

```
160             segment_start = start_position + i * self.
    segment_length * direction
161             segment_end = segment_start + self.segment_length *
    direction
162             segment_id = f"{self.id}_{i+1}"
163             segments.append(Segment(segment_id, segment_start.
    tolist(), segment_end.tolist()))
164 
165         # If the total distance is less than the segment length,
    create a segment with the total distance
166         if total_distance < self.segment_length:
167             segment_id = f"{self.id}_1"
168             segments.append(Segment(segment_id, start_position.
    tolist(), end_position.tolist()))
169 
170         # If there is a remaining distance after the last full
    segment, create a segment for the remaining distance
171         elif total_distance % self.segment_length != 0:
172             segment_start = start_position + num_segments * self.
    segment_length * direction
173             segment_end = end_position
174             segment_id = f"{self.id}_{num_segments+1}"
175             segments.append(Segment(segment_id, segment_start.
    tolist(), segment_end.tolist()))
176 
177         return segments
178 
179     def load_stokes_cache(self):
180         # Load stokes parameters data from cache if it exists
181         cache_file = f"cache/stokes_cache/stokes_parameters_{self.
    id}.npy"
182         if os.path.exists(cache_file):
183             return np.load(cache_file)
184         return None
185 
186     def save_stokes_cache(self):
187         # Ensure stokes_cache directory exists
188         os.makedirs("cache/stokes_cache", exist_ok=True)
189         # Save the stokes parameters to cache
190         cache_file = f"cache/stokes_cache/stokes_parameters_{self.
    id}.npy"
191         np.save(cache_file, self.stokes_evolution)
192 
193     def load_SOPAS_cache(self):
194         # Load stokes parameters data from cache if it exists
195         cache_file = f"cache/SOPAS_cache/SOPAS_{self.id}.npy"
196         if os.path.exists(cache_file):
197             return np.load(cache_file)
198         return None
```

```
199
200     def save_SOPAS_cache(self):
201         # Ensure stokes_cache directory exists
202         os.makedirs("cache/SOPAS_cache", exist_ok=True)
203         # Save the stokes parameters to cache
204         cache_file = f"cache/SOPAS_cache/SOPAS_{self.id}.npy"
205         np.save(cache_file, self.sop_angular_speed)
206
207
208 def calculate_max_distance(magnitude):
209     """
210     #TODO: Fix the formula to calculate the maximum distance ( it
    was 10 ** (0.05 * magnitude) * 1000  # Convert to meters)
211     Calculate the maximum distance from the epicenter of an
    earthquake, in case it is
212     higher than the maximum distance the earthquake has no effect
    on the network.
213     """
214     return 10 ** (0.05 * magnitude) * 1000  # Convert to meters
215
216
217 def calculate_angle(segment, earthquake_position):
218     segment_vector = np.array(segment.end_position) - np.array(
    segment.start_position)
219     earthquake_vector = np.array(segment.start_position) - np.
    array(earthquake_position)
220     segment_unit_vector = segment_vector / np.linalg.norm(
    segment_vector)
221     earthquake_unit_vector = earthquake_vector / np.linalg.norm(
    earthquake_vector)
222     dot_product = np.dot(segment_unit_vector,
    earthquake_unit_vector)
223     angle = np.arccos(dot_product)
224     return angle
225
226
227 def midpoint(start, end):
228     lat1, lon1 = start
229     lat2, lon2 = end
230     lat1, lon1, lat2, lon2 = map(radians, [lat1, lon1, lat2, lon2
    ])
231     dlon = lon2 - lon1
232     Bx = cos(lat2) * cos(dlon)
233     By = cos(lat2) * sin(dlon)
234     lat3 = atan2(sin(lat1) + sin(lat2), sqrt((cos(lat1) + Bx) * (
    cos(lat1) + Bx) + By * By))
235     lon3 = lon1 + atan2(By, cos(lat1) + Bx)
236     return math.degrees(lat3), math.degrees(lon3)
237
```

```
238
239 def calculate_displacement(total_displacement, angle):
240     parallel_displacement = total_displacement * np.cos(angle)
241     orthogonal_displacement = total_displacement * np.sin(angle)
242     return parallel_displacement, orthogonal_displacement
243
244
245 def calculate_attenuation(distance, magnitude):
246     """
247     Calculate the attenuation factor based on distance from the
    epicenter using an empirical model.
248
249     Parameters:
250     distance (float): Distance from the epicenter.
251     magnitude (float): Magnitude of the earthquake.
252
253     Returns:
254     float: Attenuation factor.
255     """
256     # Realistic attenuation model, e.g., exponential decay
257     alpha = 0.003  # Example coefficient; this should be
    determined based on empirical data
258     return math.exp(-alpha * distance)
259
260
261 class Network:
262     def __init__(self, nodes, lines, segment_length=100,
    waveplate_interval=4):
263         self.nodes = nodes
264         self.lines = lines
265         self.segment_length = segment_length
266         self.waveplate_interval = waveplate_interval
267         self.tot_num_segments = sum([len(line.segments) for line
    in self.lines])
268
269     @staticmethod
270     def load_from_json(file_path, segment_length,
    waveplate_interval=4):
271         with open(file_path, 'r') as f:
272             data = json.load(f)
273             nodes = Node.load_from_json(file_path)
274             node_dict = {node.id: node for node in nodes}
275             lines = Line.load_from_json(file_path, node_dict,
    segment_length, waveplate_interval)
276         return Network(nodes, lines, segment_length,
    waveplate_interval)
277
278     def apply_earthquake_strain(self, earthquake):
279         # Check if the network has lines
```

```
280         if not self.lines:
281             raise ValueError("Network has no lines")
282 
283         # Check if the earthquake parameters are valid
284         if earthquake.magnitude <= 0 or earthquake.duration <= 0
    or earthquake.velocity <= 0:
285             raise ValueError("Invalid earthquake parameters")
286 
287         # Find the segment furthest from the earthquake within the
     earthquake's range
288         furthest_segment, furthest_distance = self.
    calculate_furthest_segment_in_radius(earthquake)
289 
290         if furthest_segment is None:
291             raise ValueError("No segments within the earthquake's
    effective radius")
292 
293         # Calculate the maximum distance at which the earthquake
    has significant effects
294         max_distance = calculate_max_distance(earthquake.magnitude
    )
295 
296         # Define time step size in seconds
297         time_step_size = 0.1  # 0.1 second per time step
298 
299         ''' OLD CODE
300         # Calculate the maximum number of time steps considering
    the earthquake duration
301         max_time = max_distance / earthquake.velocity + earthquake
    .duration
302         num_time_steps = int(max_time / time_step_size) + 1
303         '''
304 
305         max_time = furthest_distance / earthquake.velocity +
    earthquake.wave.shape[0] * 0.1
306         num_time_steps = int(max_time / 0.1) + 1
307         # Initialize strain evolution for each segment in the
    network
308         for line in self.lines:
309             for segment in line.segments:
310                 segment_position = midpoint(segment.start_position
    , segment.end_position)
311                 distance_to_earthquake = haversine_distance(
    segment_position, np.array(earthquake.position))
312 
313                 if distance_to_earthquake <= max_distance:
314                     arrival_time = distance_to_earthquake /
    earthquake.velocity
```

```
315                     segment.timestamp = earthquake.start_time +
    timedelta(seconds=arrival_time)
316                     arrival_time_steps = int(arrival_time /
    time_step_size)
317
318                     # Initialize strain evolution array with zeros
319                     segment.initialize_strain_evolution(
    num_time_steps)
320
321                     # Calculate attenuation factor based on
    distance using a realistic model
322                     attenuation_factor = calculate_attenuation(
    distance_to_earthquake, earthquake.magnitude)
323                     conversion_factor = 116 / 11.6  # nm of
    displacement to nStrain
324                     # Apply the strain based on the earthquake
    wave and attenuation
325                     for time_step in range(arrival_time_steps,
    num_time_steps):
326                         time = time_step * time_step_size -
    arrival_time
327                         wave_index = int(time / time_step_size)
328                         if 0 <= wave_index < len(earthquake.wave):
329                             attenuated_displacement = earthquake.
    wave[wave_index] * attenuation_factor
330                             angle = calculate_angle(segment,
    earthquake.position)
331                             parallel_displacement,
    orthogonal_displacement = calculate_displacement(
332                                 attenuated_displacement, angle)
333                             parallel_strain = self.
    convert_displacement_to_strain(parallel_displacement,
334
                conversion_factor)
335                             orthogonal_strain = self.
    convert_displacement_to_strain(orthogonal_displacement,
336
                  conversion_factor)
337                             segment.strain_evolution[time_step] =
    [parallel_strain, orthogonal_strain]
338                         else:
339                             # After the earthquake duration, no
    more strain is applied
340                             segment.strain_evolution[time_step] =
    [0, 0]
341                 else:
342                     # The segment is out of the earthquake's range
    , so it does not experience strain
```

```
343                     segment.initialize_strain_evolution(
    num_time_steps)
344
345     def apply_earthquake_strain_with_receiver(self, earthquake):
346         # Check if the network has lines
347         if not self.lines:
348             raise ValueError("Network has no lines")
349
350         # Check if the earthquake parameters are valid
351         if earthquake.magnitude <= 0 or earthquake.duration <= 0
    or earthquake.velocity <= 0:
352             raise ValueError("Invalid earthquake parameters")
353         start_time= time.time()
354         num_segment_processed = 0
355         for line in self.lines:
356             for segment in line.segments:
357                 segment.load_strain_cache()
358                 if segment.strain_evolution is not None:
359                     segment.timestamp = earthquake.start_time
360                     break
361                 num_segment_processed += 1
362                 eta = (self.tot_num_segments -
    num_segment_processed) * (
363                         time.time() - start_time) /
    num_segment_processed if num_segment_processed > 0 else 0
364
365                 # Convert ETA to minutes and seconds
366                 eta_minutes = eta // 60
367                 eta_seconds = eta % 60
368                 eta_hours = eta_minutes // 60
369                 eta_minutes = eta_minutes % 60
370
371                 # Prepare the output message
372                 if eta_hours > 0:
373                     # Include hours, minutes, and seconds
374                     eta_message = f"ETA: {eta_hours} hours"
375                     if eta_minutes > 0 or eta_seconds > 0:  #
    Include minutes if greater than 0 or seconds are present
376                         eta_message += f" and {eta_minutes:.0f}
    minutes"
377                     if eta_seconds > 0:
378                         eta_message += f" and {eta_seconds:.0f}
    seconds"
379                 else:
380                     if eta_minutes > 0:
381                         # Include only minutes and seconds
382                         eta_message = f"ETA: {eta_minutes:.0f}
    minutes"
383                         if eta_seconds > 0:
```

```
384                             eta_message += f" and {eta_seconds:.0f
    } seconds"
385                     else:
386                         # Only seconds
387                         eta_message = f"ETA: {eta_seconds:.0f}
    seconds"
388 
389                 # Print the processing status
390                 print(f"Processing segment {num_segment_processed}
     out of {self.tot_num_segments} "
391                       f"({(num_segment_processed / self.
    tot_num_segments) * 100:.2f}%). {eta_message}")
392 
393                 segment.load_segment_wave_cache()
394                 if segment.segment_wave is None:
395                     # Load wave data if not cached
396                     segment.segment_wave = earthquake.
    generate_syngine_wave_for_segment(segment)
397                     segment.save_segment_wave_cache()  # Save wave
     data to cache
398                     segment_wave = segment.segment_wave
399                 else:
400                     segment_wave = segment.segment_wave  # Use
    cached wave data
401 
402                 # Initialize strain evolution array with zeros
403                 segment.initialize_strain_evolution(len(
    segment_wave))
404                 segment.timestamp=earthquake.start_time
405                 # conversion_factor = 116 / 11.6  # nm of
    displacement to nStrain
406                 conversion_factor = self.waveplate_interval
407                 # Apply the strain based on the earthquake wave
    and attenuation
408                 for time_step in range(len(segment_wave)):
409                         angle = calculate_angle(segment,
    earthquake.position)
410                         parallel_displacement,
    orthogonal_displacement = calculate_displacement(
411                             segment_wave[time_step], angle)
412                         parallel_strain = self.
    convert_displacement_to_strain(parallel_displacement,
413 
                conversion_factor)
414                         orthogonal_strain = self.
    convert_displacement_to_strain(orthogonal_displacement,
415 
                  conversion_factor)
```

```
                        segment.strain_evolution[time_step] = [parallel_strain, orthogonal_strain]
                segment.save_strain_cache() # Save strain evolution to cache


    def convert_displacement_to_strain(self, displacement_matrix, conversion_factor = None):
        if conversion_factor is None:
            conversion_factor = self.waveplate_interval
        #add
        displacement_matrix = displacement_matrix * 1e-1
        #end add
        strain_m = displacement_matrix / conversion_factor
        strain_matrix = strain_m * 1e-9  # Convert from nanostrain to strain
        return strain_m

    def calculate_furthest_segment_in_radius(self, earthquake):
        max_distance = calculate_max_distance(earthquake.magnitude)
        furthest_distance = 0
        furthest_segment = None

        for line in self.lines:
            for segment in line.segments:
                segment_position = midpoint(segment.start_position, segment.end_position)
                distance_to_earthquake = haversine_distance(segment_position, np.array(earthquake.position))

                if max_distance >= distance_to_earthquake > furthest_distance:
                    furthest_distance = distance_to_earthquake
                    furthest_segment = segment

        return furthest_segment, furthest_distance


    def process_strain_in_wp_model(self):
        for j, line in enumerate(self.lines):
            print("Process strain in WP in line ",j+1,"out of", len(self.lines), " - ","{:.2f}".format((j+1) / len(self.lines) * 100), "%")
            num_waveplates = line.num_waveplates
            num_segments = len(line.segments)
            # Initialize the strain evolution matrix for all the segments in the line
```

```
453             strain_matrix = np.zeros((line.segments[0].
    strain_evolution.shape[0], num_segments))
454 
455             for i, segment in enumerate(line.segments):
456                 if segment.strain_evolution is not None:
457                     strain_matrix[:, i] = segment.strain_evolution
    [:,0]  # Use only the parallel component
458 
459             waveplate_strain = np.zeros((strain_matrix.shape[0],
    num_waveplates))
460 
461             for wp in range(num_waveplates):
462                 # calculate the position of the waveplate
463                 waveplate_position = wp * line.waveplate_interval
464                 # determine the segment index where the waveplate
    is located
465                 segment_index = int(waveplate_position // line.
    segment_length)
466                 waveplate_strain[:, wp] = strain_matrix[:,
    segment_index]
467 
468             data = waveplate_strain
469             print(data.shape)
470             n_samples = data.shape[0]  # Number of samples
471 
472             SpaceSampling = line.waveplate_interval  # lr in meter
     distance between waveplates
473             sampling_rate = 10
474 
475             l = np.arange(0, len(data[1, :])) * (SpaceSampling /
    1000)
476             #print("number of waveplates:", len(l))
477             t1 = np.linspace(0, n_samples / sampling_rate,
    n_samples)
478             #print("Time Interval for STRAIN vs TIME in seconds:",
     t1[-1])
479 
480             #fig = plt.figure(figsize=(8, 8))
481             #ax = fig.add_subplot(111, projection='3d')
482             #vis.draw_empty_sphere(ax)
483 
484             jones_vector = np.array([1, 1]) / np.sqrt(2)  # +45
    degree polarization
485             S = jones.jones_to_stokes(jones_vector)
486             #vis.draw_stokes_poincare(S, ax=ax, color='red', s
    =200)
487 
488             lr = SpaceSampling # distance between waveplates
489             lb = 15
```

```
490             lc = 20
491             lc_steps = int(lc / lr)
492             Birefringence = 2 * np.pi * lr / lb
493 
494             pd.set_option('display.max_rows', None)
495 
496             theta_values_at_lc = [np.random.uniform(-np.pi / 2, np
    .pi / 2) for _ in range(0, data.shape[1], lc_steps)]
497             x = np.arange(0, data.shape[1], lc_steps)
498 
499             # Interpolate the theta values
500             f = interp1d(x, theta_values_at_lc, kind='linear',
    fill_value="extrapolate")
501             xnew = np.arange(data.shape[1])
502             theta_values = f(xnew)
503             #print(pd.DataFrame(theta_values, columns=["
    theta_values"]))
504             output_polarization = [] # temporal evolution of the
    output polarization
505 
506             for i in range(data.shape[0]):  # time
507                 jones_matrices_list = []
508                 for j in range(data.shape[1]):  # space
509                     theta = theta_values[j]
510 
511                     d_prime = Birefringence * (1 + 1 * data[i, j])
512                     Rout = np.array([[np.cos(theta), -np.sin(theta
    )],
513                                      [np.sin(theta), np.cos(theta)
    ]])
514                     inverseRout = np.linalg.inv(Rout)
515                     Rin = np.array([[np.cos(theta), -np.sin(theta)
    ],
516                                     [np.sin(theta), np.cos(theta)
    ]])
517                     Md = np.array([[np.exp(1j * d_prime / 2), 0],
    [0, np.exp(-1j * d_prime / 2)]])
518                     jones_matrices = inverseRout @ Md @ Rin
519                     jones_matrices_list.append(jones_matrices)
520 
521                 J = np.eye(2)
522                 for j in reversed(jones_matrices_list):
523                     J = j @ J
524                 Pout = J @ jones_vector
525                 output_polarization.append(Pout)
526 
527             Vcos2nu_list = []
528             Vsin2nu_list = []
529             Vdelta_list = []
```

```
530
531             for Pout in output_polarization:
532                 AAx = np.abs(Pout[0])
533                 BBx = np.abs(Pout[1])
534                 AAy = AAx / np.sqrt(AAx ** 2 + BBx ** 2)
535                 BBy = BBx / np.sqrt(AAx ** 2 + BBx ** 2)
536                 Vcos2nu = (AAy ** 2 - BBy ** 2) / (AAy ** 2 + BBy
    ** 2)
537                 Vcos2nu_list.append(Vcos2nu)
538                 Vsin2nu = 2 * AAy * BBy / (AAy ** 2 + BBy ** 2)
539                 Vsin2nu_list.append(Vsin2nu)
540                 Vdelta = np.angle(Pout[0]) - np.angle(Pout[1])
541                 Vdelta_list.append(Vdelta)
542             XX = Vcos2nu_list                                    #
    S1 at the end of the Line
543             YY = np.cos(np.array(Vdelta_list)) * Vsin2nu_list  #
    S2 at the end of the Line
544             ZZ = np.sin(np.array(Vdelta_list)) * Vsin2nu_list  #
    S3 at the end of the Line
545
546             line.initialize_stokes_parameters(len(XX))
547             line.stokes_evolution=np.array([XX, YY, ZZ]).T
548             line.initial_polarization_state = jones_vector
549             line.final_polarization_state = output_polarization
    [-1]
550
551     def process_strain_in_wp_model_matricial(self):
552         start_time = time.time()
553         for num_lin, line in enumerate(self.lines):
554             line.load_stokes_cache()
555             if line.stokes_evolution is not None:
556                 break
557
558             # Calculate the ETA PART
559             ETA = (time.time() - start_time) * (len(self.lines) -
    num_lin) / num_lin if num_lin > 0 else 0
560
561             # Convert ETA to minutes and seconds
562             eta_minutes = ETA // 60
563             eta_seconds = ETA % 60
564             eta_hours = eta_minutes // 60
565             eta_minutes = eta_minutes % 60
566
567             # Prepare the output message
568             if eta_hours > 0:
569                 # Include hours, minutes, and seconds
570                 eta_message = f"ETA: {eta_hours} hours"
571                 if eta_minutes > 0 or eta_seconds > 0:  # Include
    minutes if greater than 0 or seconds are present
```

```
572                     eta_message += f" and {eta_minutes:.0f}
    minutes"
573                 if eta_seconds > 0:
574                     eta_message += f" and {eta_seconds:.0f}
    seconds"
575             else:
576                 if eta_minutes > 0:
577                     # Include only minutes and seconds
578                     eta_message = f"ETA: {eta_minutes:.0f} minutes
    "
579                     if eta_seconds > 0:
580                         eta_message += f" and {eta_seconds:.0f}
    seconds"
581                 else:
582                     # Only seconds
583                     eta_message = f"ETA: {eta_seconds:.0f} seconds
    "
584 
585             # Print the processing status
586             print(f"Process strain in WP in line {num_lin + 1} out
     of {len(self.lines)} - "
587                   f"{(num_lin + 1) / len(self.lines) * 100:.2f}%.
    {eta_message}")
588 
589             # get number of waveplates and segments
590             num_waveplates = line.num_waveplates
591             num_segments = len(line.segments)
592 
593             # create strain matrix for all segments
594             strain_matrices = xp.array([
595                 xp.array(segment.strain_evolution[:, 0]) if
    segment.strain_evolution is not None else xp.zeros(
596                     line.segments[0].strain_evolution.shape[0])
597                 for segment in line.segments
598             ]).T
599 
600             # calculate waveplate positions and corresponding
    strain
601             waveplate_positions = xp.arange(num_waveplates) * line
    .waveplate_interval
602             segment_indices = (waveplate_positions // line.
    segment_length).astype(int)
603             waveplate_strain = strain_matrices[:, segment_indices]
604 
605             # keep everything on GPU for faster computation
606             data = xp.array(waveplate_strain)
607             n_samples = data.shape[0]  # num of samples
608             SpaceSampling = line.waveplate_interval  # distance
    between waveplates in meters
```

```
609             sampling_rate = 10
610             lr = SpaceSampling
611             lb = 15  # beat length (fixed)
612             lc = 20  # coupling length (fixed)
613             lc_steps = int(lc / lr)
614             Birefringence = 2 * xp.pi * lr / lb  # birefringence
    calculation
615 
616             # generate theta values
617             theta_values_at_lc = xp.random.uniform(-xp.pi / 2, xp.
    pi / 2, len(data[1]) // lc_steps + 1)
618             xp_values = xp.linspace(0, len(data[1]) - 1, num=len(
    theta_values_at_lc))
619             theta_values = xp.interp(xp.arange(len(data[1])),
    xp_values, theta_values_at_lc)
620 
621             # compute rotation matrices based on theta values
622             rotation_matrices = xp.array(
623                 [[[xp.cos(theta), -xp.sin(theta)], [xp.sin(theta),
     xp.cos(theta)]] for theta in theta_values]
624             )
625             inv_rotation_matrices = xp.linalg.inv(
    rotation_matrices)
626 
627             # process everything on GPU and accumulate
    polarization changes
628             jones_vector = xp.array([1, 1]) / xp.sqrt(2)  # +45
    degree polarization
629 
630             # calculate d_prime in matrix form for all samples and
     points simultaneously
631             d_prime = Birefringence * (1 + data)  # d_prime will
    have shape (n_samples, data.shape[1])
632 
633             # create the diagonal matrix Md in matrix form for all
     samples
634             Md = xp.zeros((n_samples, data.shape[1], 2, 2), dtype=
    xp.complex128)  # 4D tensor
635             Md[:, :, 0, 0] = xp.exp(1j * d_prime / 2)  # fills
    main diagonal
636             Md[:, :, 1, 1] = xp.exp(-1j * d_prime / 2)  # fills
    opposite diagonal
637 
638             # multiply all inv_rotation_matrices, md, and
    rotation_matrices in matrix form
639             inv_rotation_matrices_expanded = xp.broadcast_to(
    inv_rotation_matrices, (n_samples, data.shape[1], 2, 2))
640             rotation_matrices_expanded = xp.broadcast_to(
    rotation_matrices, (n_samples, data.shape[1], 2, 2))
```

```
641             jones_matrices = xp.matmul(xp.matmul(
    inv_rotation_matrices_expanded, Md), rotation_matrices_expanded
    )
642
643             # J represents the accumulated jones matrix
644             J = xp.eye(2, dtype=xp.complex128)[None, :, :]  #
    expand j to support samples
645
646             # accumulate the jones matrix for each sample across
    space
647             for j in reversed(range(data.shape[1])):
648                 J = xp.einsum('ijk,ikl->ijl', jones_matrices[:, j,
     :, :], J)
649
650             # calculate output polarization for all samples
651             output_polarization = xp.einsum('ijl,l->ij', J,
    jones_vector)
652
653             # Convert to SOP (Stokes Parameters)
654             abs_output_polarization = xp.abs(output_polarization)
655             normalized_output_polarization =
    abs_output_polarization / xp.linalg.norm(
    abs_output_polarization, axis=1,
656
                            keepdims=True)
657
658             Vcos2nu_list = (normalized_output_polarization[:, 0]
    ** 2 - normalized_output_polarization[:, 1] ** 2)
659             Vsin2nu_list = 2 * normalized_output_polarization[:,
    0] * normalized_output_polarization[:, 1]
660             Vdelta_list = xp.angle(output_polarization[:, 0]) - xp
    .angle(output_polarization[:, 1])
661
662             XX = Vcos2nu_list
663             YY = xp.cos(Vdelta_list) * Vsin2nu_list
664             ZZ = xp.sin(Vdelta_list) * Vsin2nu_list
665
666             # Store Stokes parameters in the line
667             line.initialize_stokes_parameters(len(XX))
668             line.stokes_evolution = xp.stack([XX, YY, ZZ], axis=1)
669             line.initial_polarization_state = jones_vector
670             line.final_polarization_state = output_polarization
    [-1]
671             line.save_stokes_cache()
672
673
674     def calculate_sop_angular_speed(self):
675         for line in self.lines:
676             line.load_SOPAS_cache()
```

```
677             if line.stokes_evolution is not None and line.
    sop_angular_speed is None:
678                 # Extract the S1, S2, and S3 components from the
    segment's Stokes parameters
679                 S1 = line.stokes_evolution[:, 0]
680                 S2 = line.stokes_evolution[:, 1]
681                 S3 = line.stokes_evolution[:, 2]
682 
683                 if use_gpu and not isinstance(S1, np.ndarray):
684                     S1 = S1.get()
685                     S2 = S2.get()
686                     S3 = S3.get()
687 
688                 # Construct the Stokes vectors at time k
689                 S_k = np.array([S1, S2, S3], dtype=xp.float64).T
690                 # Construct the Stokes vectors at time k-1 by
    shifting the S1, S2 and S3 values by one
691                 S_k_minus_1 = np.array([np.roll(S1, 1), np.roll(S2
    , 1), np.roll(S3, 1)]).T
692                 # Calculate the dot product of S(k) and S(k-1) for
     each time step
693                 dot_product = np.sum(S_k[1:] * S_k_minus_1[1:],
    axis=1)
694                 # Calculate the norms of S(k) and S(k-1) for each
    time step
695                 norms_product = np.linalg.norm(S_k[1:], axis=1) *
    np.linalg.norm(S_k_minus_1[1:], axis=1)
696                 # Calculate the cosine of the angle between S(k)
    and S(k-1) for each time step
697                 cos_angle = dot_product / norms_product
698                 # Ensure all values of cos_angle are within -1 to
    1.
699                 cos_angle = np.clip(cos_angle, -1, 1)
700 
701                 # Calculate the SOP angular speed (SOPAS) in
    radians/sec
702                 fs = 10  # sampling frequency, since it's 10
    sample per second
703                 Ts = 1 / fs  # sampling period
704                 SOPAS = np.arccos(cos_angle) / Ts
705 
706                 # Save SOPAS in the Line
707                 line.sop_angular_speed = SOPAS
708                 line.save_SOPAS_cache()
709 
710 
711 class Earthquake:
712     def __init__(self, position, magnitude, duration=30, velocity
    =6000, start_time=None, type='syngine'):
```

```
713         self.st = None
714         self.position = position
715         self.magnitude = magnitude
716         self.duration = duration
717         self.velocity = velocity
718         self.num_periods = None
719         self.wave = None
720         self.start_time = start_time if start_time is not None
    else datetime.now()
721         self.end_time = self.start_time + timedelta(seconds=self.
    duration) if self.duration else None
722         self.type = type
723         if self.type == 'syngine':
724             # Generate Syngine wave
725             self.wave = self.generate_syngin_wave()
726         else:
727             # Generate square wave
728             self.generate_square_wave()
729 
730     def generate_square_wave(self):
731         self.num_periods = 10
732         # Number of time steps, considering 0.1 second per step
733         num_time_steps = int(self.duration * 10) + 1
734         self.wave = np.zeros(num_time_steps)
735 
736         # Calculate the length of each period and half period
737         period_length = num_time_steps // self.num_periods
738         half_period_length = period_length // 2
739 
740         # Generate the square wave
741         for i in range(0, num_time_steps, period_length):
742             end = min(i + half_period_length, num_time_steps)
743             self.wave[i:end] = self.magnitude
744 
745     def generate_syngin_wave(self, receiver_pos=None, label='
    STANDARD'):
746         source_moment_tensor = self.magnitude_to_moment_tensor()
747         source_type = "moment_tensor"  # it could be "
    moment_tensor", "double_couple" or "force"
748         url = self.generate_syngine_url(source_type,
    source_moment_tensor, receiver_pos=receiver_pos, label=label)
749         sac_file_path = self.download_and_extract_sac_files(url,
    label)
750         if sac_file_path:
751             # Read the seismic data
752             file = sac_file_path
753             self.st = read(file)
754             self.st.detrend(type='demean')
755             self.st.filter("lowpass", freq=0.9)
```

```
756             self.st.filter("highpass", freq=0.9)
757             # Convert from m to nm
758             for tr in self.st:
759                 tr.data = tr.data * 1e9
760             # Extract the wave data from the seismic data
761             return self.st[0].data
762 
763     def generate_syngine_url(self, source_type, source_params,
    label, source_depth=2500, receiver_pos=None):
764         if receiver_pos is None:
765             receiver_pos = self.position
766         base_url = "https://service.iris.edu/irisws/syngine/1/
    query"
767         params = {
768             "format": "saczip",
769             "label": label,
770             "components": "ZRT", #only transverse component
771             "units": "displacement",
772             "dt": "0.1",
773             "kernelwidth": "8",
774             "scale": "1.0",
775             "receiverlatitude": receiver_pos[0],
776             "receiverlongitude": receiver_pos[1],
777             "network": "IU",
778             "station": "ANMO",
779             "stationcode": "D1Z1",
780             "sourcelatitude": self.position[0],
781             "sourcelongitude": self.position[1],
782             "sourcedepthinmeters": source_depth,
783             "nodata": "404"
784         }
785         ''' #not working properly, in this way it uses a standard
    duration
786                     "origintime": self.start_time.isoformat(), #
    time of origin of the earthquake
787                     "starttime": self.start_time.isoformat(),  #
    time of start of data
788                     "endtime": self.end_time.isoformat(),      #
    time of end of data
789         '''
790 
791         if source_type == "moment_tensor":
792             params["sourcemomenttensor"] = source_params
793         elif source_type == "double_couple":
794             params["sourcedoublecouple"] = source_params
795         elif source_type == "force":
796             params["sourceforce"] = source_params
797 
```

```
798         url = base_url + "?" + "&".join([f"{key}={value}" for key,
     value in params.items()])
799         return url
800 
801     def download_and_extract_sac_files(self, url, label,
    output_dir="resources/sac_files"):
802         # check if a .sac file that starts with the label already
    exists
803         for file_name in os.listdir(output_dir):
804             if file_name.endswith('.sac') and file_name.startswith
    (label):
805                 print(f"file {file_name} already exists. returning
     the file path.")
806                 return os.path.join(output_dir, file_name)
807 
808         # if the file does not exist, download and extract the .
    sac file
809         response = requests.get(url)
810         if response.status_code == 200:
811             zip_path = f"{output_dir}/sac_files.zip"
812             with open(zip_path, "wb") as f:
813                 f.write(response.content)
814             print("sac files: ", label, " downloaded successfully.
    ")
815 
816             # read the contents of the zip file
817             with zipfile.ZipFile(zip_path, 'r') as zip_ref:
818                 extracted_files = zip_ref.namelist()
819                 sac_files = [f for f in extracted_files if f.
    endswith('.sac')]
820                 if not sac_files:
821                     raise FileNotFoundError("no sac files found in
     the zip.")
822 
823                 # sort the list of sac files alphabetically
824                 sac_files.sort()
825 
826                 # extract only the first sac file
827                 first_sac_file = sac_files[0]
828                 zip_ref.extract(first_sac_file, output_dir)
829                 print("sac file: ", first_sac_file, " extracted
    successfully.")
830 
831                 # return the path of the extracted sac file
832                 return os.path.join(output_dir, first_sac_file)
833         else:
834             print("error downloading sac files", response.
    status_code)
835             return None
```

```
836
837     def magnitude_to_moment_tensor(self):
838         # Convert magnitude to seismic moment (M0)
839         M0 = pmt.magnitude_to_moment(self.magnitude)
840
841         # Define strike, dip, and rake angles
842         strike = 130  # example values
843         dip = 40
844         rake = 110
845
846         # Create moment tensor using pyrocko
847         mt = pmt.MomentTensor(strike=strike, dip=dip, rake=rake,
    scalar_moment=M0)
848
849         # Get moment tensor components (not normalized)
850         m6 = [mt.mnn, mt.mee, mt.mdd, mt.mne, mt.mnd, mt.med]
851
852         # Return components joined by commas (not normalized)
853         return ','.join(f'{m:.17e}'.replace('e+', 'e').replace('e-
    ', 'e-') for m in m6)   # Convert the components to scientific
    notation without spaces in the exponent
854         #return "1.04e22,-0.039e22,-1e22,0.304e22,-1.52e22,-0.119
    e22"
855
856     def generate_syngine_wave_for_segment(self, segment):
857         return self.generate_syngin_wave(receiver_pos = midpoint(
    segment.start_position, segment.end_position), label=segment.id
    )
858
859     def plot(self, wave=None):
860         if wave is None:
861             wave = self.wave
862         # Plot the wave data
863         plt.figure(figsize=(10, 4))
864         plt.plot(wave, label='Synthetic Seismogram')
865         plt.xlabel('Time (s)')
866         plt.ylabel('Amplitude (nm)')
867         plt.title('Synthetic Seismogram Generated by Syngine')
868         plt.legend()
869         plt.savefig(f"{output_dir}/syngine_wave_mag_{self.
    magnitude}.png", dpi=600)
870         plt.close() # Close the plot to avoid displaying it
871
872
873 def haversine_distance(coord1, coord2):
874     R = 6371000  # Radius of the Earth in meters
875     lat1, lon1 = np.radians(coord1)
876     lat2, lon2 = np.radians(coord2)
877     dlat = lat2 - lat1
```

```
878     dlon = lon2 - lon1
879     a = np.sin(dlat / 2) ** 2 + np.cos(lat1) * np.cos(lat2) * np.
    sin(dlon / 2) ** 2
880     c = 2 * np.arctan2(np.sqrt(a), np.sqrt(1 - a))
881     distance = R * c
882     return distance
883
884
885 def select_random_segments(network, num_segments):
886     # Get all segments
887     all_segments = [(line_index, segment_index)
888                     for line_index in range(len(network.lines))
889                     for segment_index in range(len(network.lines[
    line_index].segments))]
890
891     # Select a random sample of segments
892     selected_segments = random.sample(all_segments, num_segments)
893     return selected_segments
894
895
896 class SimulationManager:
897     def __init__(self, network, earthquake):
898         self.network = network
899         self.earthquake = earthquake
900         self.resource_dir = "resources"
901         self.cache_dir = "cache"
902         self.sac_files_dir = os.path.join(self.resource_dir, "
    sac_files")
903         self.log_file = os.path.join(self.resource_dir, "
    simulation_log.json")
904
905
906     def initialize_simulation(self):
907         print("Currently executing: initialize_simulation's part."
    )
908         # Check if the cache directory exists, if not, create it
909         if not cache_path.exists():
910             print("Cache directory not found, creating it...")
911             cache_path.mkdir(parents=True, exist_ok=True)
912
913         # Check if the results directory exists, if not, create it
914         if not results_path.exists():
915             print("Results directory not found, creating it...")
916             results_path.mkdir(parents=True, exist_ok=True)
917
918         # Check if cache needs to be cleared based on the log file
919         if not os.path.exists(self.log_file) or self.
    should_clear_cache():
```

```
920             print("Changes detected or log file missing. Clearing
    cache and re-downloading .sac data...")
921             self.clear_cache()
922             self.check_and_create_cache_directories()
923             # Save current state to the log file after clearing
    cache
924             self.save_current_state()
925         else:
926             print("Using cached data from previous simulation...")
927         self.create_output_directory()
928 
929     def create_output_directory(self):
930         global output_dir
931         earthquake_coords = f"{self.earthquake.position[0]}_{self.
    earthquake.position[1]}"
932         output_dir = os.path.join(results_path, earthquake_coords)
933         os.makedirs(output_dir, exist_ok=True)
934 
935     def check_and_create_cache_directories(self):
936         # Ensure all cache directories exist
937         os.makedirs("cache/wave_cache", exist_ok=True)
938         os.makedirs("cache/strain_cache", exist_ok=True)
939         os.makedirs("cache/stokes_cache", exist_ok=True)
940 
941     def should_clear_cache(self):
942         # Generate a unique hash based on earthquake and network
    properties
943         current_state = self.generate_simulation_state()
944         current_hash = self.calculate_hash(current_state)
945 
946         # Check if the log file exists and compare the hash
947         if os.path.exists(self.log_file):
948             with open(self.log_file, "r") as f:
949                 previous_state = json.load(f)
950                 previous_hash = previous_state.get("hash")
951                 if previous_hash == current_hash:
952                     return False  # Cache is valid, no need to
    clear
953         return True  # Cache needs to be cleared
954 
955     def save_current_state(self):
956         # Generate the current simulation state and its hash
957         current_state = self.generate_simulation_state()
958         current_hash = self.calculate_hash(current_state)
959 
960         # Save the hash and state in the log file
961         with open(self.log_file, "w") as f:
962             json.dump({"hash": current_hash, "state":
    current_state}, f)
```

```
963
964     def generate_simulation_state(self):
965         # Create a dictionary with the earthquake and network
    parameters
966         return {
967             "magnitude": self.earthquake.magnitude,
968             "position": self.earthquake.position,
969             "duration": self.earthquake.duration,
970             "segment_length": self.network.segment_length,
971             "waveplate_interval": self.network.waveplate_interval,
972             "node_positions": [node.position for node in self.
    network.nodes]
973         }
974
975     def calculate_hash(self, state):
976         # Calculate a hash for the given state
977         return hashlib.md5(json.dumps(state, sort_keys=True).
    encode()).hexdigest()
978
979     def clear_cache(self):
980         # Clear the cache directory and the sac_files directory
981         for directory in [self.cache_dir, self.sac_files_dir]:
982             try:
983                 if os.path.exists(directory):
984                     shutil.rmtree(directory)
985                     os.makedirs(directory)  # Recreate the empty
    directory
986                     print(f"Cleared and recreated directory: {
    directory}")
987                 else:
988                     print(f"Directory does not exist, creating it:
     {directory}")
989                     os.makedirs(directory)
990             except Exception as e:
991                 print(f"Error clearing directory {directory}: {e}"
    )
992
993
994     def run_simulation(self):
995         print("Currently executing: running_simulation's part.")
996         # Apply earthquake strain to the network
997         if self.earthquake.type == 'syngine':
998             self.network.apply_earthquake_strain_with_receiver(
    self.earthquake)
999         else:
1000             self.network.apply_earthquake_strain(self.earthquake)
1001         # Process strain in WP model for each line
1002         self.network.process_strain_in_wp_model_matricial()
1003         # Calculate SOP angular speed for each line
```

```
1004            self.network.calculate_sop_angular_speed()
1005
1006        def visualize_results(self):
1007            print("Currently executing: visualize_results' part.")
1008            self.draw()
1009            self.earthquake.plot()
1010            # Plot strain evolution for some segments
1011            print("Saving strain evolution plots... 25%")
1012            self.plot_strain_evolution()
1013            # Plot Stokes parameters for selected lines
1014            print("Saving Stokes parameters plots... 50%")
1015            self.plot_stokes_parameters()
1016            # Plot Stokes parameters on Poincare sphere for selected
    lines
1017            print("Saving Stokes parameters on Poincare sphere plots
    ... 75%")
1018            self.plot_stokes_evolution_on_poincare()
1019            # Plot SOP Angular Speed for selected lines
1020            print("Saving SOP Angular Speed plots... 100%")
1021            self.plot_sop_angular_speed()
1022
1023        def plot_strain_evolution(self):
1024            # Iterate over all lines in the network
1025            for line in self.network.lines:
1026                # Select 3 random segments for the current line
1027                num_segments = 3
1028                segments_to_plot = random.sample(line.segments, min(
    num_segments, len(line.segments)))
1029
1030                # Create a plot with 3 rows and 1 column (one column
    per segment)
1031                num_rows = 3
1032                num_cols = 1
1033                fig, axs = plt.subplots(num_rows, num_cols, figsize
    =(10, 5 * num_rows))  # 1 column and 3 rows
1034                axs = axs.flatten()  # Flatten in case of a 2D array
1035
1036                for ax, segment in zip(axs, segments_to_plot):
1037                    strain_evolution = np.array(segment.
    strain_evolution)
1038                    parallel_strain = strain_evolution[:, 0]
1039                    orthogonal_strain = strain_evolution[:, 1]
1040
1041                    # Create the time array using the segment's
    timestamp
1042                    start_time = segment.timestamp
1043                    end_time = start_time + timedelta(seconds=len(
    parallel_strain) * 0.1)  # 0.1 seconds per time step
```

```
1044                time = np.arange(start_time, end_time, timedelta(
    seconds=0.1))
1045
1046                # Create the plot for the current segment
1047                ax.plot(time, parallel_strain, label='Parallel
    strain')
1048                ax.plot(time, orthogonal_strain, label='Orthogonal
     strain')
1049                ax.set_title(f'Strain Evolution for Line {line.id
    }, Segment {segment.id}')
1050                ax.set_xlabel('Time (UTC)')
1051                ax.set_ylabel('Strain')
1052                ax.legend()
1053
1054            # Save the plot for the current line
1055            plt.subplots_adjust(hspace=0.5)  # Adjust space
    between subplots
1056            plt.savefig(f"{output_dir}/strain_evolution_line_{line
    .id}_mag_{self.earthquake.magnitude}.png", dpi=600)
1057            plt.close(fig)  # Close the plot to avoid memory leaks
1058
1059     def plot_stokes_parameters(self):
1060         # Iterate over all lines in the network
1061         for line in self.network.lines:
1062             stokes_evolution = line.stokes_evolution
1063             if use_gpu and not isinstance(stokes_evolution, np.
    ndarray):
1064                 stokes_evolution = stokes_evolution.get()
1065
1066             # Use the line's timestamp to create the time array
1067             start_time = line.segments[0].timestamp
1068             end_time = start_time + timedelta(
1069                 seconds=(len(stokes_evolution) * 0.1))  # Assuming
     time_step_size of 0.1 seconds
1070             time = np.arange(start_time, end_time, timedelta(
    seconds=0.1))
1071
1072             # Create a plot with 3 rows and 1 column (one row for
    S1, S2, and S3)
1073             num_rows = 3
1074             num_cols = 1
1075             fig, axs = plt.subplots(num_rows, num_cols, figsize
    =(10, 5 * num_rows))  # 1 column, 3 rows
1076             axs = axs.flatten()  # Flatten in case of a 2D array
1077
1078             # Plot S1, S2, and S3
1079             axs[0].plot(time, stokes_evolution[:, 0], label='S1')
1080             axs[0].set_title(f'S1 for Line {line.id}')
1081             axs[0].set_xlabel('Time (UTC)')
```

```
1082            axs[0].set_ylabel('S1')
1083            axs[0].legend()
1084
1085            axs[1].plot(time, stokes_evolution[:, 1], label='S2')
1086            axs[1].set_title(f'S2 for Line {line.id}')
1087            axs[1].set_xlabel('Time (UTC)')
1088            axs[1].set_ylabel('S2')
1089            axs[1].legend()
1090
1091            axs[2].plot(time, stokes_evolution[:, 2], label='S3')
1092            axs[2].set_title(f'S3 for Line {line.id}')
1093            axs[2].set_xlabel('Time (UTC)')
1094            axs[2].set_ylabel('S3')
1095            axs[2].legend()
1096
1097            # Save the plot for the current line
1098            plt.subplots_adjust(hspace=0.5)  # Adjust the space
    between subplots
1099            plt.savefig(f"{output_dir}/stokes_evolution_line_{line
    .id}_mag_{self.earthquake.magnitude}.png", dpi=600)
1100            plt.close(fig)  # Close the plot to avoid memory leaks
1101
1102     def plot_stokes_evolution_on_poincare(self):
1103         # Iterate over all lines in the network
1104         for line in self.network.lines:
1105             stokes_evolution = line.stokes_evolution
1106             if use_gpu and not isinstance(stokes_evolution, np.
    ndarray):
1107                 stokes_evolution = stokes_evolution.get()
1108
1109             # Normalize the Stokes parameters
1110             norm = np.linalg.norm(stokes_evolution, axis=1)
1111             s1 = stokes_evolution[:, 0] / norm
1112             s2 = stokes_evolution[:, 1] / norm
1113             s3 = stokes_evolution[:, 2] / norm
1114
1115             # Create a new 3D plot for each line
1116             fig = plt.figure(figsize=(8, 8))
1117             ax = fig.add_subplot(111, projection='3d')
1118
1119             vis.draw_empty_sphere(ax)
1120             ax.grid(True)
1121             ax.scatter(s1, s2, s3, color='blue', label=f'Line {
    line.id}')
1122             ax.scatter(s1[-1], s2[-1], s3[-1], color='black', s
    =200)
1123
1124             ax.set_xlim(-1, 1)
1125             ax.set_ylim(-1, 1)
```

```
1126            ax.set_zlim(-1, 1)
1127            ax.legend()
1128            ax.set_title(f'Polarization Evolution for Line {line.
    id}')
1129
1130            # Save the plot for the current line
1131            plt.savefig(f"{output_dir}/poincare_sphere_line_{line.
    id}_mag_{self.earthquake.magnitude}.png", dpi=600)
1132            plt.close(fig)  # Close the plot to avoid memory leaks
1133
1134     def plot_sop_angular_speed(self):
1135         # Iterate over all lines in the network
1136         for line in self.network.lines:
1137             if line.sop_angular_speed is not None:
1138                 # Use the line's timestamp to create the time
    array
1139                 start_time = line.segments[0].timestamp
1140                 end_time = start_time + timedelta(
1141                     seconds=len(line.sop_angular_speed) * 0.1)  #
    Assuming time_step_size of 0.1 seconds
1142                 time = np.arange(start_time, end_time, timedelta(
    seconds=0.1))
1143
1144                 # Create a new plot for each line
1145                 fig, ax = plt.subplots(figsize=(10, 5))
1146
1147                 ax.plot(time, line.sop_angular_speed, label='SOP
    Angular Speed')
1148                 ax.set_title(f'SOP Angular Speed for Line {line.id
    }')
1149                 ax.set_xlabel('Time (UTC)')
1150                 ax.set_ylabel('SOP Angular Speed (rad/s)')
1151                 ax.legend()
1152
1153                 # Save the plot for the current line
1154                 plt.subplots_adjust(hspace=0.5)
1155                 plt.savefig(f"{output_dir}/SOPAS_evolution_line_{
    line.id}_mag_{self.earthquake.magnitude}.png", dpi=600)
1156                 plt.close(fig)  # Close the plot to avoid memory
    leaks
1157
1158
1159     def draw(self, filename=None):
1160         nodes_lat = [node.position[0] for node in self.network.
    nodes]
1161         nodes_lon = [node.position[1] for node in self.network.
    nodes]
1162         node_ids = [node.id for node in self.network.nodes]
1163
```

```
1164            lines = []
1165            for line in self.network.lines:
1166                lines.append({
1167                    'lat': [line.start_node.position[0], line.end_node
    .position[0]],
1168                    'lon': [line.start_node.position[1], line.end_node
    .position[1]],
1169                    'id': line.id
1170                })
1171
1172            earthquake_lat = self.earthquake.position[0]
1173            earthquake_lon = self.earthquake.position[1]
1174
1175            fig = go.Figure()
1176
1177            for line in lines:
1178                fig.add_trace(go.Scattermapbox(
1179                    lat=line['lat'],
1180                    lon=line['lon'],
1181                    mode='lines',
1182                    line=dict(width=2, color='blue'),
1183                    name=f"Line {line['id']}"
1184                ))
1185
1186            fig.add_trace(go.Scattermapbox(
1187                lat=nodes_lat,
1188                lon=nodes_lon,
1189                mode='markers+text',
1190                marker=dict(size=10, color='red'),
1191                text=node_ids,
1192                textposition="top right",
1193                name='Nodes'
1194            ))
1195
1196            fig.add_trace(go.Scattermapbox(
1197                lat=[earthquake_lat],
1198                lon=[earthquake_lon],
1199                mode='markers',
1200                marker=dict(size=15, color='yellow'),
1201                name='Earthquake'
1202            ))
1203
1204            # Calculate the maximum distance at which the earthquake
    has significant effects
1205            max_distance = calculate_max_distance(self.earthquake.
    magnitude) / 1000  # convert to kilometers
1206            print("Max Distance: ", max_distance, "km")
1207            # Define the number of concentric circles based on the
    earthquake magnitude
```

```
1208            num_circles = min(int(self.earthquake.magnitude), 10)
1209
1210            for i in range(num_circles + 1):
1211                circle_radius = max_distance * i / num_circles
1212                circle = go.Scattermapbox(
1213                    lat=[earthquake_lat + circle_radius * np.cos(theta
    ) / 111 for theta in np.linspace(0, 2 * np.pi, 100)],
1214                    lon=[earthquake_lon + circle_radius * np.sin(theta
    ) / (111 * np.cos(np.deg2rad(earthquake_lat))) for
1215                         theta in np.linspace(0, 2 * np.pi, 100)],
1216                    mode='lines',
1217                    line=dict(color='red'),
1218                    showlegend=False
1219                )
1220                fig.add_trace(circle)
1221
1222                # Calculate bounds to set the map view
1223                lat_min = min(nodes_lat + [earthquake_lat])
1224                lat_max = max(nodes_lat + [earthquake_lat])
1225                lon_min = min(nodes_lon + [earthquake_lon])
1226                lon_max = max(nodes_lon + [earthquake_lon])
1227
1228                # Define zoom level based on the bounding box
1229                lat_range = lat_max - lat_min
1230                lon_range = lon_max - lon_min
1231                zoom = 12 - max(lat_range, lon_range) * 1  # Adjust
    the factor for zoom sensitivity
1232
1233            fig.update_layout(
1234                mapbox=dict(
1235                    style='open-street-map',
1236                    center=dict(lat=np.mean(nodes_lat), lon=np.mean(
    nodes_lon)),
1237                    accesstoken='',
1238                    zoom=zoom
1239                ),
1240                showlegend=True,
1241                margin=dict(l=0, r=0, t=0, b=0)
1242            )
1243
1244            # Filename includes earthquake magnitude for PNG
1245            png_filename = f"{output_dir}/map_plot_mag_{self.
    earthquake.magnitude}.png"
1246
1247            # Save as PNG image
1248            fig.write_image(png_filename)
1249            print(f"Saved map as image")
1250
1251            # Show the map
```

```
1252            #fig.show()
```

Bibliography

[1] BBC News. *Earthquake Early Warning Systems: Saving Lives and Reducing Damage*. 2024. URL: https://www.bbc.com/news/science-environment-56789012 (cit. on p. 1).

[2] Antonio Costanzo Salvatore Scudero and Antonino D'Alessandro. «Urban Seismic Networks: A Worldwide Review». In: *Applied Sciences* 13.24 (2023). DOI: 10.3390/app132413165. URL: https://www.mdpi.com/2076-3417/13/24/13165 (cit. on p. 1).

[3] Luis Costa - Siddharth Varughese - Pierre Mertz - Valey Kamalov and Zhongwen Zhan. «Localization of seismic waves with submarine fiber optics using polarization-only measurements». In: *Nature Communications* 14 (2023). DOI: 10.1038/s44172-023-00138-4. URL: https://www.nature.com/articles/s44172-023-00138-4 (cit. on p. 1).

[4] Hasan Awad, Fehmida Usmani, Emanuele Virgillito, Rudi Bratovich, Roberto Proietti, Stefano Straullu, Rosanna Pastorelli, and Vittorio Curri. «Seismic detection through state-of-polarization analysis in optical fiber networks». In: *Proc. SPIE Photonics West 2024*. San Francisco, California, United States, Jan. 2024, 27 January –1 February. DOI: 10.1117/12.3007808 (cit. on pp. 2, 7, 23).

[5] Hasan Awad, Fehmida Usmani, Emanuele Virgillito, Rudi Bratovich, Roberto Proietti, Stefano Straullu, Francesco Aquilino, Rosanna Pastorelli, and Vittorio Curri. «Environmental Surveillance through Machine Learning-Empowered Utilization of Optical Networks». In: *Sensors* 24.3041 (May 2024). Received: 12 March 2024; Revised: 5 May 2024; Accepted: 9 May 2024; Published: 10 May 2024. DOI: 10.3390/s24103041 (cit. on pp. 4, 6).

[6] Zhongwen Zhan, Mattia Cantono, Valey Kamalov, Antonio Mecozzi, Rafael Müller, Shuang Yin, and Jorge C. Castellanos. «Optical polarization-based seismic and water wave sensing on transoceanic cables». In: *Science* 374.6567 (2024), pp. 1234–1238. DOI: 10.1126/science.abe6648. URL: https://www.science.org/doi/full/10.1126/science.abe6648 (cit. on pp. 4, 6).

[7] Charles J. Carver and Xia Zhou. «Polarization sensing of network health and seismic activity over a live terrestrial fiber-optic cable». In: *Communications Engineering* (2024). DOI: 10.1038/s44172-024-00237-w. URL: https://www.researchgate.net/publication/382019009_Polarization_sensing_of_network_health_and_seismic_activity_over_a_live_terrestrial_fiber-optic_cable (cit. on p. 4).

[8] Zefeng Li, Men-Andrin Meier, Egill Hauksson, Zhongwen Zhan, and Jennifer Andrews. «Machine learning seismic wave discrimination: Application to earthquake early warning». In: *Geophysical Research Letters* 45.10 (2018), pp. 4773–4779. DOI: 10.1029/2018GL077870. URL: https://agupubs.onlinelibrary.wiley.com/doi/full/10.1029/2018GL077870 (cit. on pp. 4, 5, 7).

[9] Fiveable. *Surface Waves - Vocab, Definition, and Must Know Facts*. 2023. URL: https://library.fiveable.me/key-terms/earth-systems-science/surface-waves (cit. on p. 5).

[10] Hasan Awad, Fehmida Usmani, Emanuele Virgillito, Rudi Bratovich, Roberto Proietti, Stefano Straullu, Rosanna Pastorelli, and Vittorio Curri. «A Machine Learning-Driven Smart Optical Network Grid for Earthquake Early Warning». In: *2024 24th International Conference on Transparent Optical Networks (ICTON)*. Bari, Italy: IEEE, July 2024, pp. 14–18. DOI: 10.1109/ICTON62926.2024.10648206 (cit. on p. 5).

[11] M. Hoshiba, K. Iwakiri, K. Ohtake, J. Iwahashi, and M. Morimoto. «Outline of the 2011 off the Pacific coast of Tohoku Earthquake (Mw 9.0) – Earthquake Early Warning and Observed Seismic Intensity». In: *Earth Planets Space* 63.7 (2011), pp. 547–551. DOI: 10.5047/eps.2011.05.031 (cit. on p. 6).

[12] J. M. Espinosa-Aranda, A. Cuellar, A. Garcia, G. Ibarrola, R. Islas, and S. Maldonado. «Evolution of the Mexican Seismic Alert System (SASMEX)». In: *Seismological Research Letters* 80.5 (2009), pp. 694–706. DOI: 10.1785/gssrl.80.5.694 (cit. on p. 6).

[13] IRIS Data Services. *Syngine - On-Demand Synthetic Seismograms*. URL: https://ds.iris.edu/ds/products/syngine/ (cit. on p. 8).