Assignment 1

Author

Ari Torkilsson Johannesen - s233913

October 6, 2024

Contents

Introduction

This report presents the solution to the assignment on pulse propagation in optical fibers, implemented using the Python programming language. The problems focus on modeling the impact of dispersion, non-linearity, and their simultaneous effects on chirped and non-chirped Gaussian and soliton pulses in fiber-optic communication systems. The numerical techniques, such as the Fast Fourier Transform (FFT), are implemented in the `numpy` library. The `matplotlib` library is used for generating the plots in the report. The `pandas` and `tabulate` libraries are used for aggregating results from calculations and printing them in tabular form. The entire Python code is included in the appendix.

1 Exercise 1: Impact of Dispersion on Pulse Propagation

1.1 Question 1-1

Answer: In this question, we are asked to compute the sampling period in time, sampling frequency, the frequency sampling period, and the minimum frequency, assuming the total time window $T_W = 2500$ ps and the number of samples $N = 2^{14}$. The calculations follow straightforward formulas given in eq. (1).

$$T_{sa} = \frac{T_W}{N}, \quad F_{sa} = \frac{1}{T_{sa}}, \quad \Delta F = \frac{F_{sa}}{N}, \quad F_{\min} = -\frac{F_{sa}}{2} \tag{1}$$

a) The sampling period T_{sa} is calculated in eq. (2).

$$T_{sa} = \frac{2500 \cdot 10^{-12}\ \text{s}}{2^{14}} = 1.53 \times 10^{-13}\ \text{s} \approx 0.15\ \text{ps}s \tag{2}$$

b) The sampling frequency F_{sa} is calculated in eq. (3).

$$F_{sa} = \frac{1}{1.53 \cdot 10^{-13} s} = 6.55 \cdot 10^{12}\ \text{Hz} = 6553.6\ \text{GHz} \tag{3}$$

c) The sampling period in frequency ΔF is calculated in eq. (4).

$$\Delta F = \frac{6.55 \cdot 10^{12}\ \text{Hz}}{2^{14}} = 400\ \text{MHz} \tag{4}$$

d) The minimum frequency $F_{\min}$ is calculated in eq. (5).

$$F_{\min} = -\frac{6.55 \cdot 10^{12}\ \text{Hz}}{2} = 3.28 \cdot 10^{12} = -3276.8\ \text{GHz} \tag{5}$$

The code is documented in the Appendix section 4.1.

1.2 Question 1-2

We are given the definitions for calculating the pre-chirped input Gaussian field envelope $A(0,t)$ and pulse power $P(0,t)$ as functions of time t in eq. (6):

$$A(0,t) = A_0 \exp\left[-\left(\frac{1+iC}{2}\right)\left(\frac{t}{T_0}\right)^2\right], \quad P(0,t) = A(0,t) \cdot A^*(0,\mathrm{t}) \tag{6}$$

a) We generate a time vector with a start time of $-\mathrm{T_W}/2$, sampling period of $\mathrm{T_{sa}}$ eq. (2) and a total number of samples in the vector corresponding to N (section 1.1). Next we calculate the field envelope $\mathrm{A(0,t)}$ and power $\mathrm{P(0,t)}$ for each of the chirp values $\mathrm{C} = [-10, 0, 5]$, in order to finally plot the power vs. time, as shown in fig. 1.

b) The plotted curves in fig. 1 are identical for all three values of C, which is expected because the chirp parameter affects the phase of the Gaussian pulse, but the power is independent of the phase as the complex values eliminate each other.

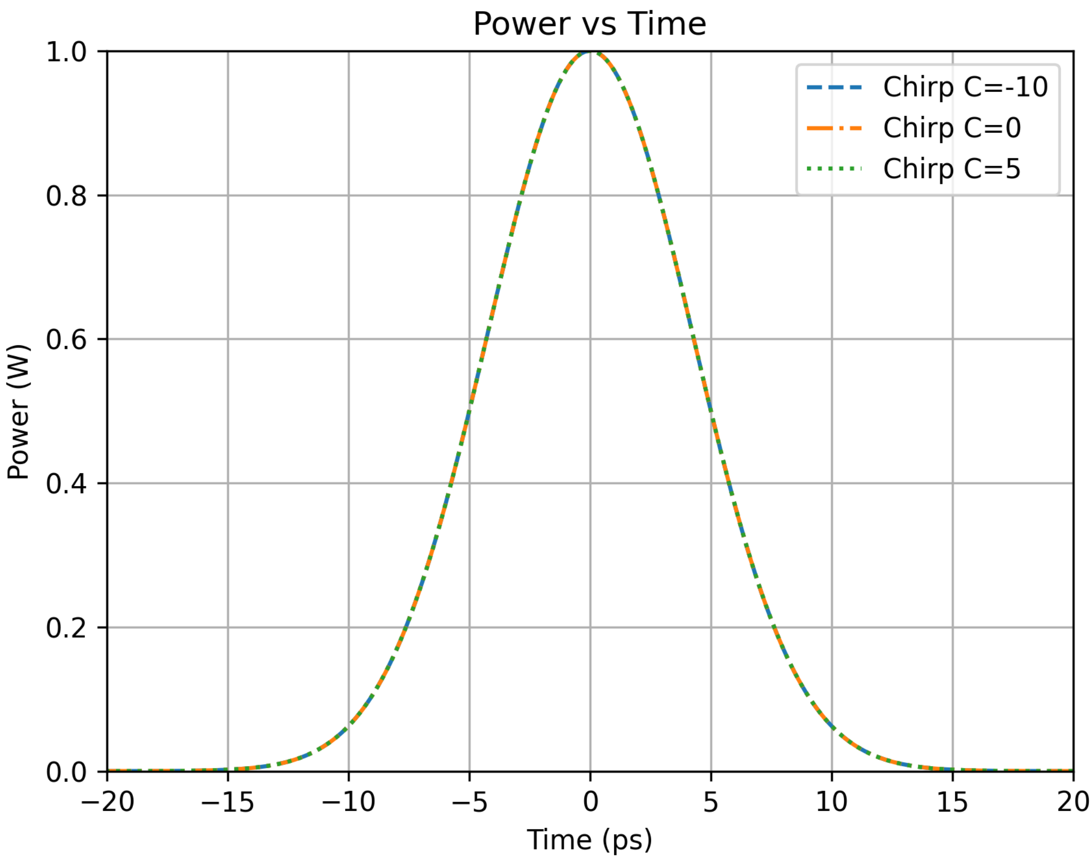

Figure 1: Power vs Time for different chirp values

The code is documented in the Appendix section 4.2.

1.3 Question 1-3

The next step is to compute the frequency spectra of the electric field using the FFT algorithm. We generate a frequency vector with start frequency F_{min} eq. (5), sampling period ΔF eq. (4), a total number of samples in the vector corresponding to N (section 1.1) and apply the FFT to the time-domain signals.

a, b, c) The electrical field envelope and power spectra of the three pulses are calculated by the formula given in section 1.2, normalized and subsequently plotted in fig. 2, showing how the chirp parameter alters the spectrum shape in the frequency domain.

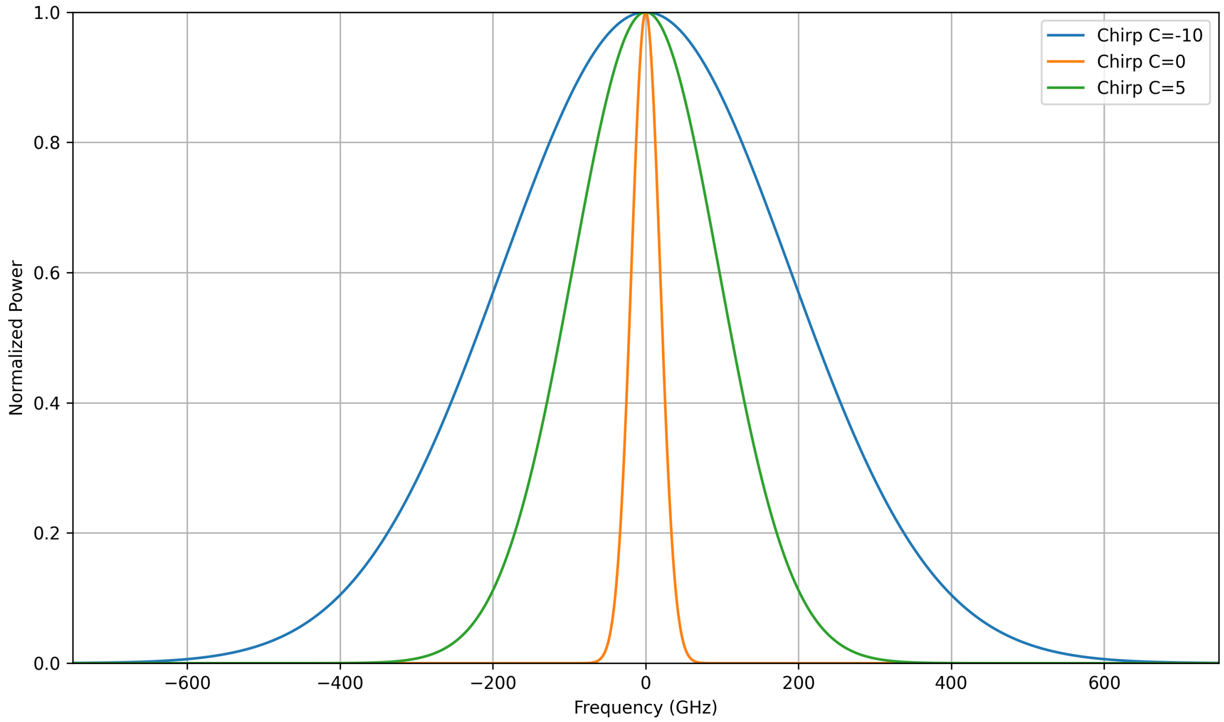

Figure 2: Normalized Power Spectrum vs Frequency for different chirp values

d) The FWHM width of the spectra for the different chirp parameters have been measured as follows:

- **Chirp C = -10**, FWHM = 443.2 GHz
- **Chirp C = 0**, FWHM = 44.0 GHz
- **Chirp C = 5**, FWHM = 224.8 GHz

The code is documented in the Appendix section section 4.3.

1.4 Question 1-4

We are given the following relationship between the spectral FWHM and the chirp parameter in eq. (7).

$$F_{\text{FWHM}} = \frac{\sqrt{\ln 2}}{\pi T_0}\sqrt{1+C^2} \tag{7}$$

a) We calculate the F_{FWHM} spectral widths and compare with the measured FWHM from section 1.3. From table 1 we can see that the values are similar, but not identical. The assumption is that the discrete sample pool in section 1.3 may introduce small discrepancies when measured.

Chirp	Measured FWHM (GHz)	F_{FWHM} (GHz)
-10	443.2	443.472
0	44.0	44.1271
5	224.8	225.005

Table 1: Measured and Theoretical FWHM Values for Different Chirp Parameters

The code is documented in the Appendix section section 4.4.

1.5 Question 1-5

The relationship between the spectrum of the transmitted signal at distance z and the input signal is given by eq. (8).

$$\tilde{\mathrm{A}}(\mathrm{z},\omega) = \tilde{A}(0,\omega)\exp\left[\mathrm{i}\frac{\beta_2}{2}\mathrm{z}\omega^2 + \mathrm{i}\frac{\beta_3}{6}\mathrm{z}\omega^3\right] \tag{8}$$

With the assumption that $\beta_3 = 0$ ps^3 km^{-1}, this simplifies the relationship to eq. (9).

$$\tilde{\mathrm{A}}(\mathrm{z},\omega) = \tilde{A}(0,\omega)\exp\left[\mathrm{i}\frac{\beta_2}{2}\mathrm{z}\omega^2\right] \tag{9}$$

a, b, c) The transfer function for dispersion was applied in the frequency domain, and the inverse FFT was used to return to the time domain. The temporal broadening of the pulses was calculated (section 1.3) and measured (section 1.4) at different propagation distances $\mathrm{z}_1 = 0.3199$ km, $\mathrm{z}_2 = 1.6636$ km and $\mathrm{z}_3 = 3.3272$ km and table 2 shows the results including the case where z = 0 km. We can visually confirm the spectral broadening by looking at fig. 3, which shows that both distance (z) and chirp (C) affect the temporal broadening.

	T_{FWHM} (ps) *z = 0 km*	$T_{FWHM1}(z_1)$ (ps) *z_1 = 0.3199 km*	$T_{FWHM1}(z_2)$ (ps) *z_2 = 1.6636 km*	$T_{FWHM1}(z_3)$ (ps) *z_3 = 3.3272 km*
C = -10	9.92	29.15	110.33	210.74
C = 0	9.92	9.92	13.89	22.13
C = +5	9.92	1.68	41.05	92.02

Table 2: FWHM values at different propagation distances for various chirp values.

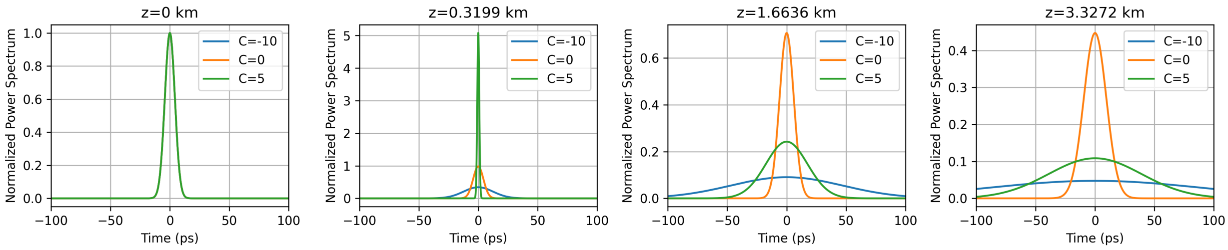

Figure 3: Power vs Time for differen t values of distance (z) and chirp (C)

The code is documented in the Appendix section section 4.5.

1.6 Question 1-6

We are given the analytical ratio between temporal widths for Gaussian transmitted pulses in eq. (10).

$$\frac{\mathrm{T_1(z)}}{\mathrm{T_0}} = \frac{T_{\mathrm{FWHM1}}(z)}{T_{\mathrm{FWHM}}(0)} = \sqrt{\left(1 + \left[\frac{\beta_2 \mathrm{C}}{\mathrm{T_0^2}}\mathrm{z}\right]\right)^2 + \left[\frac{\beta_2}{\mathrm{T_0^2}}\right]^2} \tag{10}$$

a, b, c, d) We calculate and line plot the analytical ratios for different chirp values, comparing with the results obtained in section 1.5 as scatter plots in fig. 4, which shows good agreement.

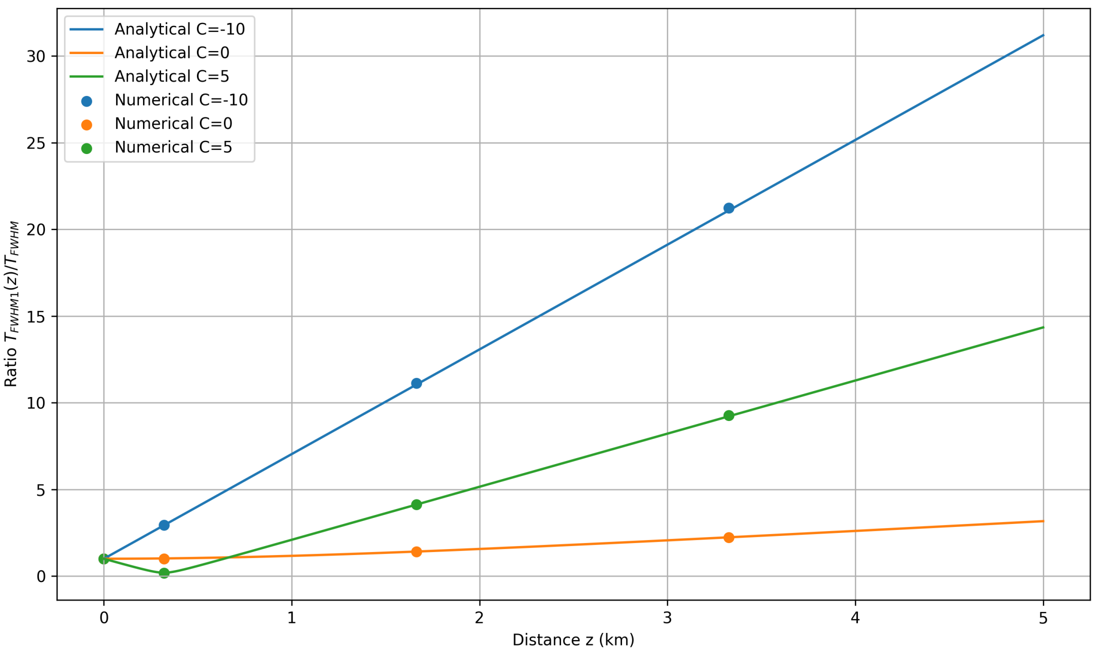

Figure 4: Ratio of Temporal Width vs Distance for Different Chirp Values

The code is documented in the Appendix section section 4.6.

2 Exercise 2: Impact of Non-Linearity

2.1 Question 2-1

We are given the definition for a pre-chirped input Gaussian field envelope $A(0,t)$ as function of time t in eq. (11).

$$\mathrm{A(0,t)} = \mathrm{A_0} \exp\left[-\left(\frac{1+\mathrm{iC}}{2}\right)\left(\frac{\mathrm{t}}{\mathrm{T_0}}\right)^2\right] = \sqrt{\mathrm{P_0}} \exp\left[-\left(\frac{1+\mathrm{iC}}{2}\right)\left(\frac{\mathrm{t}}{\mathrm{T_0}}\right)^2\right] \tag{11}$$

The non-linear phase shift ϕ_{NL} and effective length L_{eff} in an optical fiber are defined in eq. (12).

$$\phi_{\mathrm{NL}} = \gamma P_0 L_{\mathrm{eff}}, \quad L_{\mathrm{eff}} = \frac{1-e^{-\alpha z}}{\alpha} \tag{12}$$

a) The goal is to be able to express transmission distance z as a function of the corresponding non-linear phase ϕ_{NL}. We show the derivation for the transmission distance z in eq. (13).

$$\begin{aligned}
\phi_{\mathrm{NL}} &= \gamma P_0 \frac{1-e^{-\alpha z}}{\alpha} \\
\phi_{\mathrm{NL}} \cdot \alpha &= \gamma P_0 (1-e^{-\alpha z}) \\
\frac{\phi_{\mathrm{NL}} \cdot \alpha}{\gamma P_0} &= 1 - e^{-\alpha z} \\
-e^{-\alpha z} &= 1 - \frac{\phi_{\mathrm{NL}} \cdot \alpha}{\gamma P_0} \\
-\alpha z &= \ln{(1 - \frac{\phi_{\mathrm{NL}}\alpha}{\gamma P_0})} \\
z &= -\frac{\ln{(1 - \frac{\phi_{\mathrm{NL}}\alpha}{\gamma P_0})}}{\alpha}
\end{aligned} \tag{13}$$

b, c) We compute the effective length L_{eff} and transmission distance z for the given values of the non-linear phase shift $\phi_{\mathrm{NL,max}} = [0.5\pi, 1.5\pi, 2.5\pi, 3.5\pi]$. The resulting values are shown in table 3.

$\phi_{NL,max}$ (radians)	Transmission Distance z (km)	Effective Length L_{eff} (km)
1.5708	1.29451	1.25664
4.71239	4.14121	3.76991
7.85398	7.41875	6.28319
10.9956	11.2812	8.79646

Table 3: Calculated non-linear phase shift, transmission distance and effective length values.

The code is documented in the Appendix section 4.7.

2.2 Question 2-2

a) The sampling period T_{sa} is calculated in eq. (2).

$$T_{sa} = \frac{2500 \cdot 10^{-12}\ \mathrm{s}}{2^{14}} = 1.53 \times 10^{-13}\ \mathrm{s} \tag{14}$$

b) The sampling frequency F_{sa} is calculated in eq. (3).

$$F_{sa} = \frac{1}{1.53 \cdot 10^{-13} s} = 6.55 \cdot 10^{12}\ \mathrm{Hz} = 6553.6\ \mathrm{GHz} \tag{15}$$

c) The sampling period in frequency ΔF is calculated in eq. (4).

$$\Delta F = \frac{6.55 \cdot 10^{12}\ \mathrm{Hz}}{2^{14}} = 400\ \mathrm{MHz} \tag{16}$$

d) The minimum frequency $F_{\min}$ is calculated in eq. (5).

$$F_{\min} = -\frac{6.55 \cdot 10^{12}\ \mathrm{Hz}}{2} = 3.28 \cdot 10^{12} = -3276.8\ \mathrm{GHz} \tag{17}$$

The code is documented in the Appendix section 4.8.

2.3 Question 2-3

In this question, both the temporal and spectral pulse at the input of the fibre should be calculated and plotted.

a, b) fig. 5 shows the power of the pulse in time (normalised to the temporal peak power) and power of the pulse in frequency (normalised to the spectral peak power).

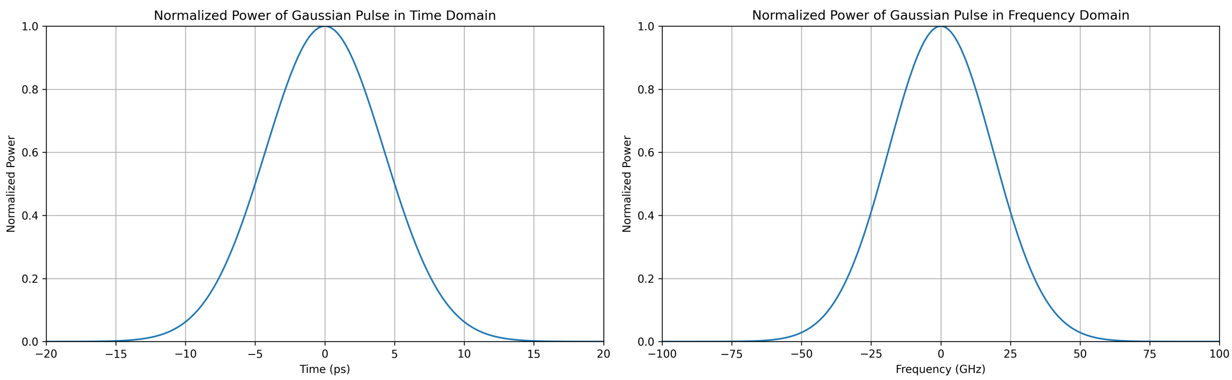

Figure 5: Normalized Power vs Time and Normalized Power vs Frequency for $C = 0$.

c) We measure the FWHM in Time Domain to 9.92 ps and the FWHM in Frequency Domain to 44.00 GHz. The expected FWHM in Time Domain is 10 ps, as per the initial condition.

The code is documented in the Appendix section 4.9.

2.4 Question 2-4

a) Plotting the pulse power over time at the fiber output is interesting, as it reveals details about pulse dispersion and signal distortion. When non-linearity is present, its impact is often more evident in the frequency domain, where phenomena like spectral broadening and new frequency components can be observed.

b, c) We calculate the spectrum of the pulse in frequency for each of the 4 lengths and normalize w.r.t. input peak power, and the resulting plot is shown in fig. 6.

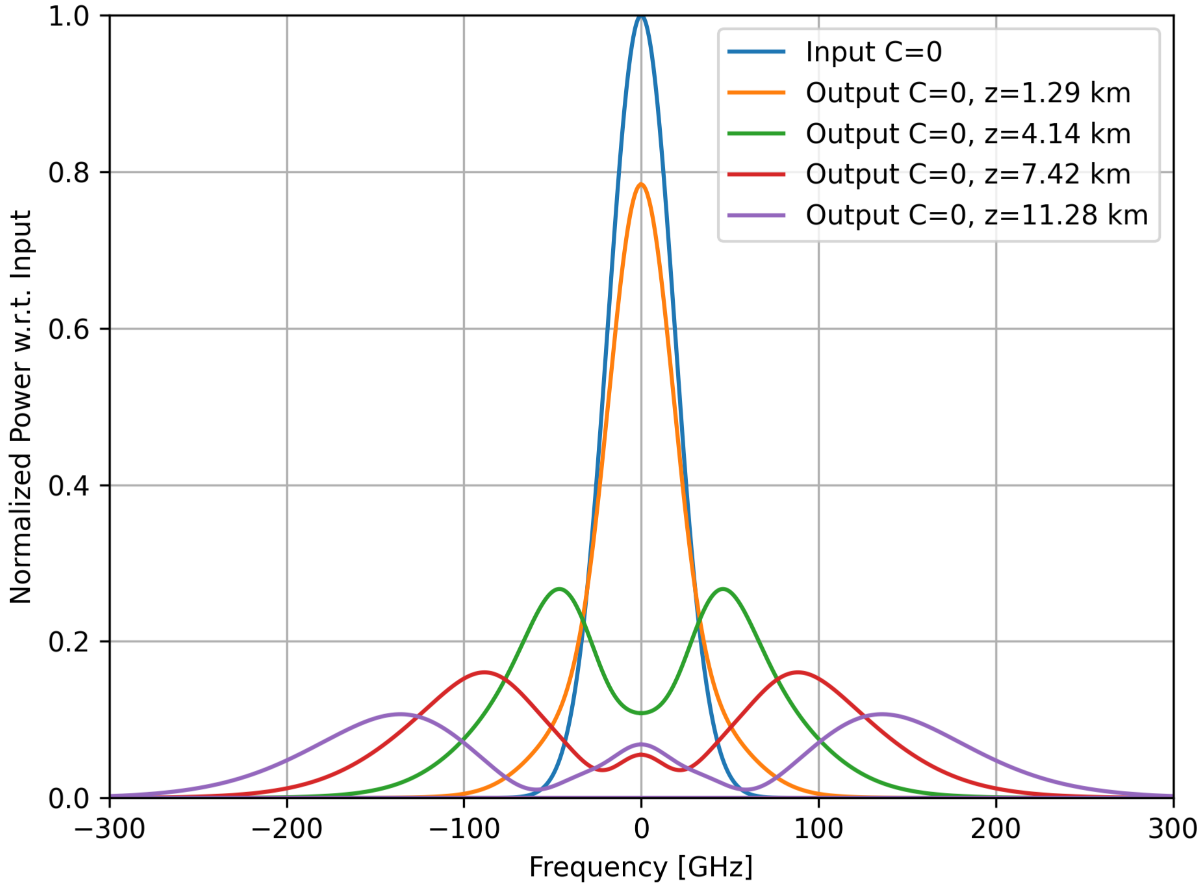

Figure 6: Normalized Power vs Frequency for $C = 0$ and different values of z.

d) From fig. 6 we observe spectral broadening in the frequency domain as the power spreads with increasing propagation distance. Spectral broadening can be problematic because varying propagation speeds of different frequencies in the fiber cause dispersion, resulting in degraded transmission bitrate.

The code is documented in the Appendix section 4.10.

3 Exercise 3: Impact of Non-Linearity and Dispersion Simultaneously

3.1 Question 3-1

In this section, we implement pulse propagation using the split-step technique for a pre-chirped input Gaussian field envelope, i.e. eq. (6)

a, b, c) The full split-step method is implemented and used to propagate the pulse through the fibre for all combinations of z and C and plotted in figs. 7 to 9. The sanity check is plotted in fig. 10, showing that the analytical and numerical simulated results are identical.

d) From the plots we conclude that the split-step implementation works when neglecting non-linearity.

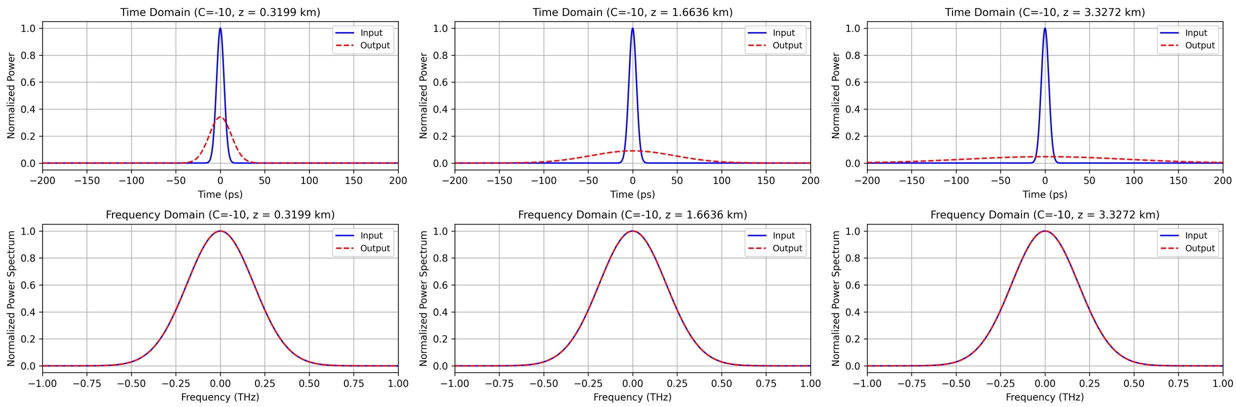

Figure 7: Normalized Power vs Time and Frequency for $C = -10$ and different values of z.

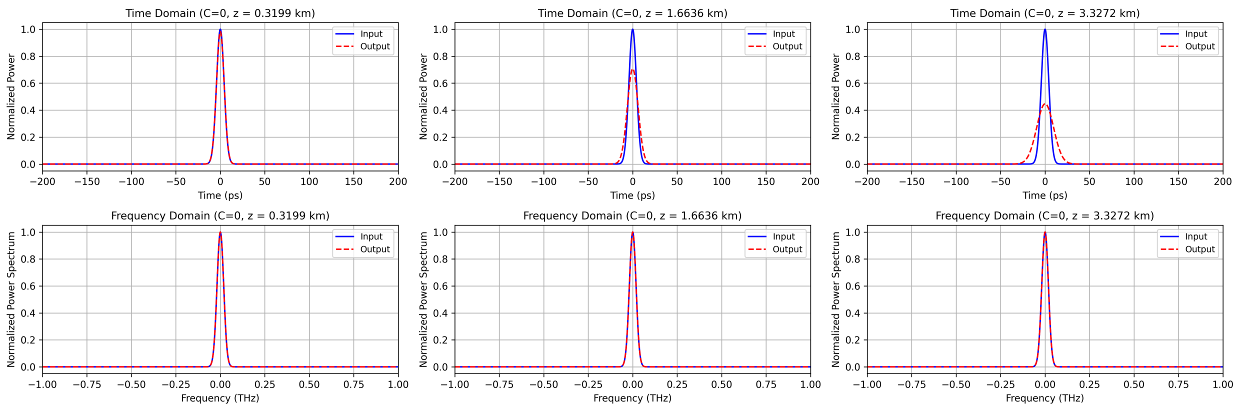

Figure 8: Normalized Power vs Time and Frequency for $C = 0$ and different values of z.

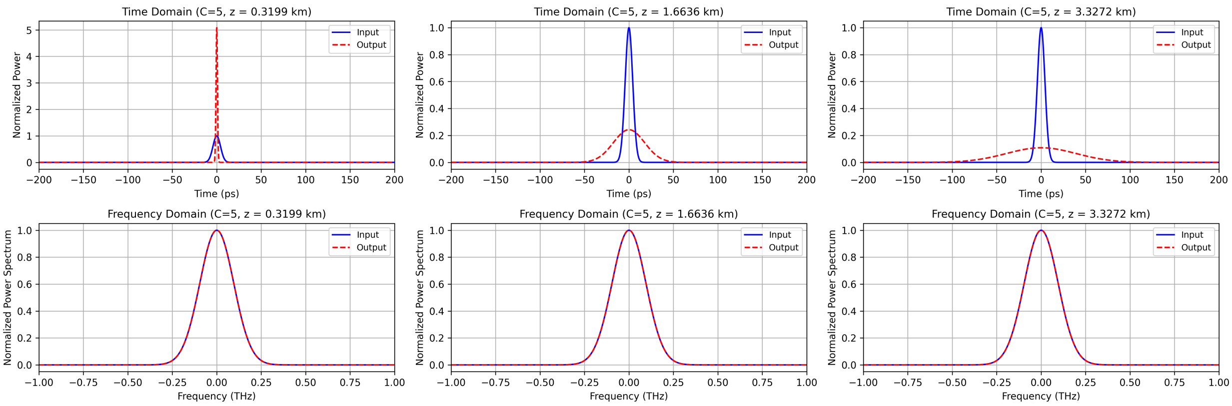

Figure 9: Normalized Power vs Time and Frequency for $C = 5$ and different values of z.

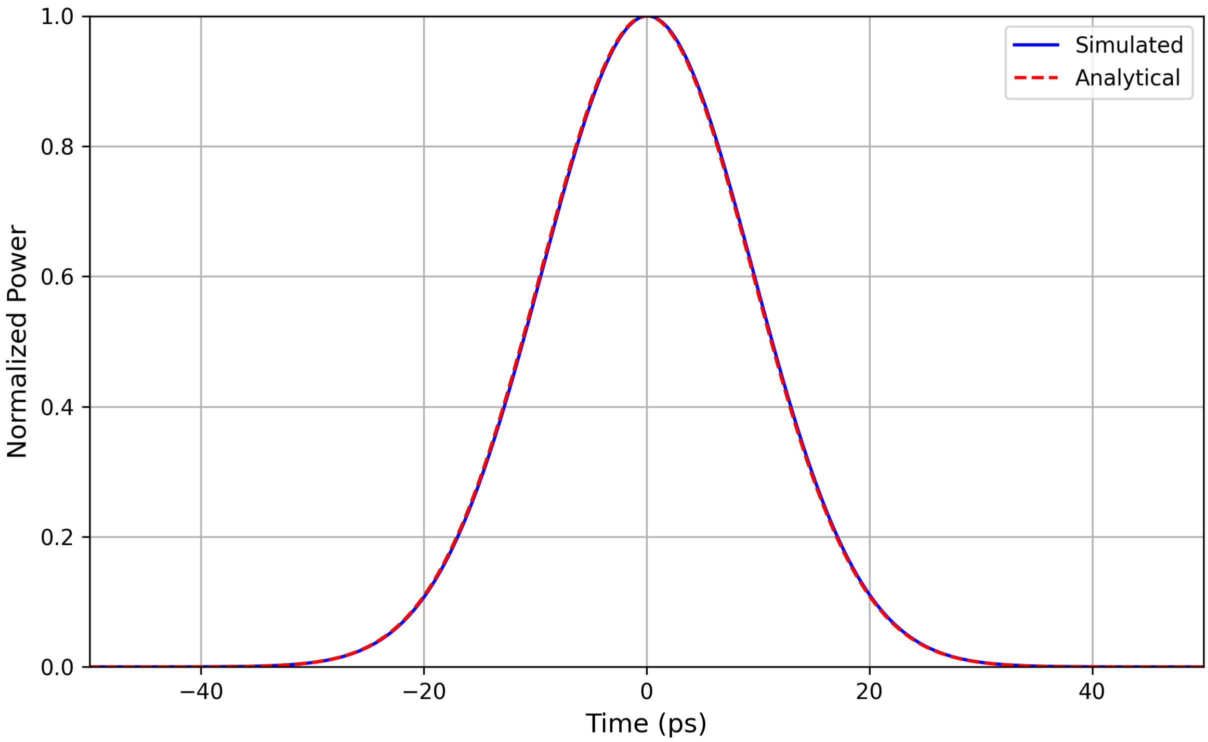

Figure 10: Sanity check for dispersive example.

The code is documented in the Appendix section 4.11.

3.2 Question 3-2

In this section, we implement pulse propagation using the split-step technique for a non-chirped input Gaussian field envelope, i.e. eq. (18).

$$A(0,t) = A_0 \exp\left[-\frac{1}{2}\left(\frac{t}{T_0}\right)^2\right] \tag{18}$$

a, b, c) We propagate the pulse through the fibre for each transmission length z and plot output pulse in time and frequency for all combinations, as shown in fig. 11. The sanity check is plotted in fig. 12, showing that the analytical and numerical simulated results are identical. Based on these plots, and the plots from section 3.1, we have a good indication that the split-step implementation is working as intended.

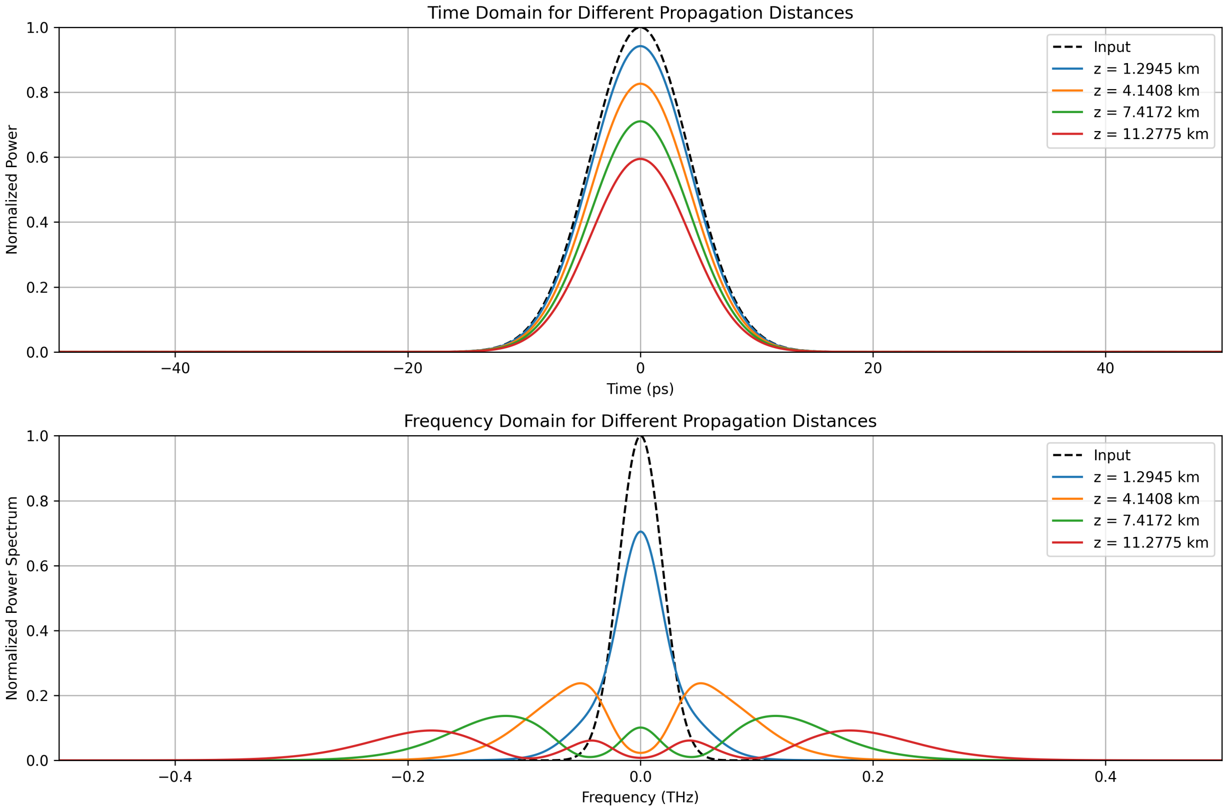

Figure 11: Normalized Power vs Time and Frequency for non-chirped input and different z values.

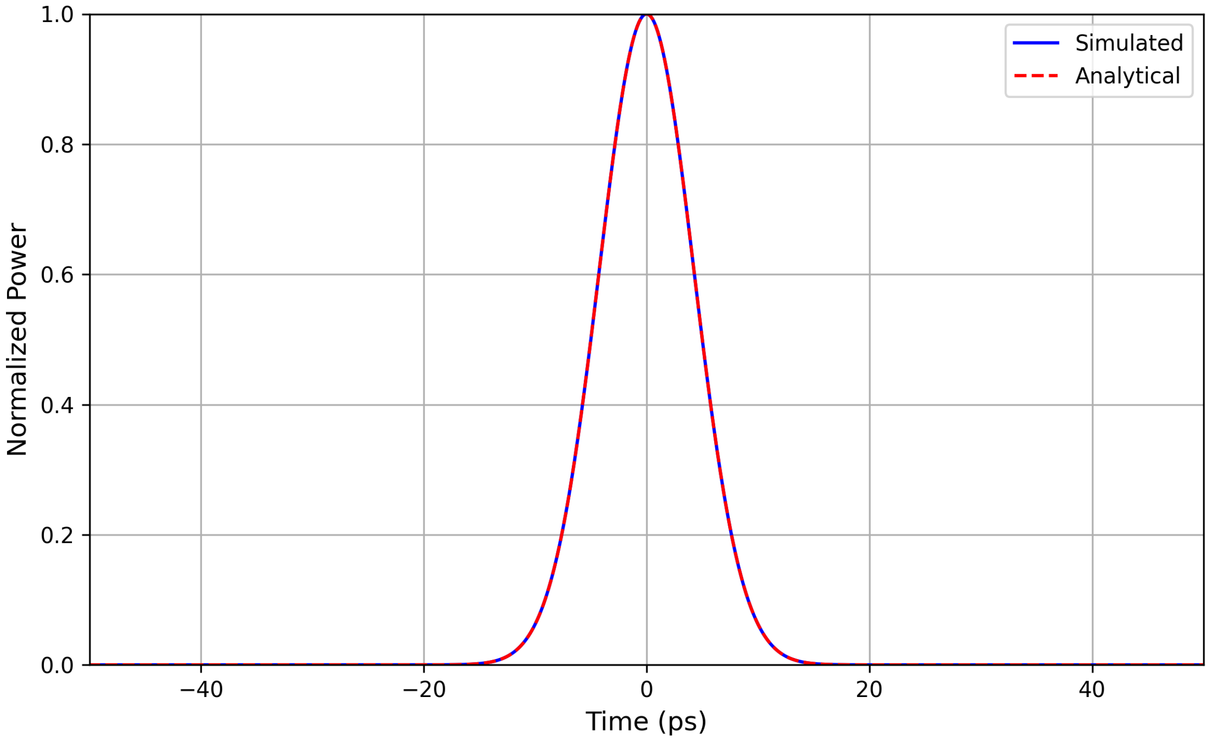

Figure 12: Sanity check for non-linear example.

The code is documented in the Appendix section 4.12.

3.3 Question 3-3

In this section the combined effects of both dispersion and non-linearity will be examined. Typically, the Nonlinear Schrödinger Equation (NLSE) cannot be solved analytically for arbitrary input pulses. However, for a specific pulse shape that meets precise criteria aligned with the fiber's characteristics, an analytical solution can be found. This solution is known as a *soliton*, and the pulse takes the form of a hyperbolic secant function (sech). A notable property of such a sech-shaped soliton is that the pulse maintains its shape and spectrum throughout propagation, meaning the input and output pulses remain identical.

a) We calculate the pulse width $T_0 \approx 4.16$ ps for the sech pulse matched to the fibre.

b, c, d, e) We propagate the sech pulse through the fibre using the split-step technique and compare the input and output pulse in time and in frequency, both for $\alpha = 0\ \text{km}^{-1}$ and $\alpha = 0.0461\ \text{km}^{-1}$, as shown in figs. 13 and 14. From fig. 13 we see, that when $\alpha = 0\ \text{km}^{-1}$ the input and output pulses in time are not completely identical, but rather the output pulse is shifted slightly along the x-axis, but has the same shape as the input pulse. We were not able to determine the cause of this temporal shift, but assume that this is caused by some programmatic error in the code. The input and output signal in frequency are identical. In fig. 14 when $\alpha = 0.0461$ km^{-1}, we see that the pulse changes as it propagates.

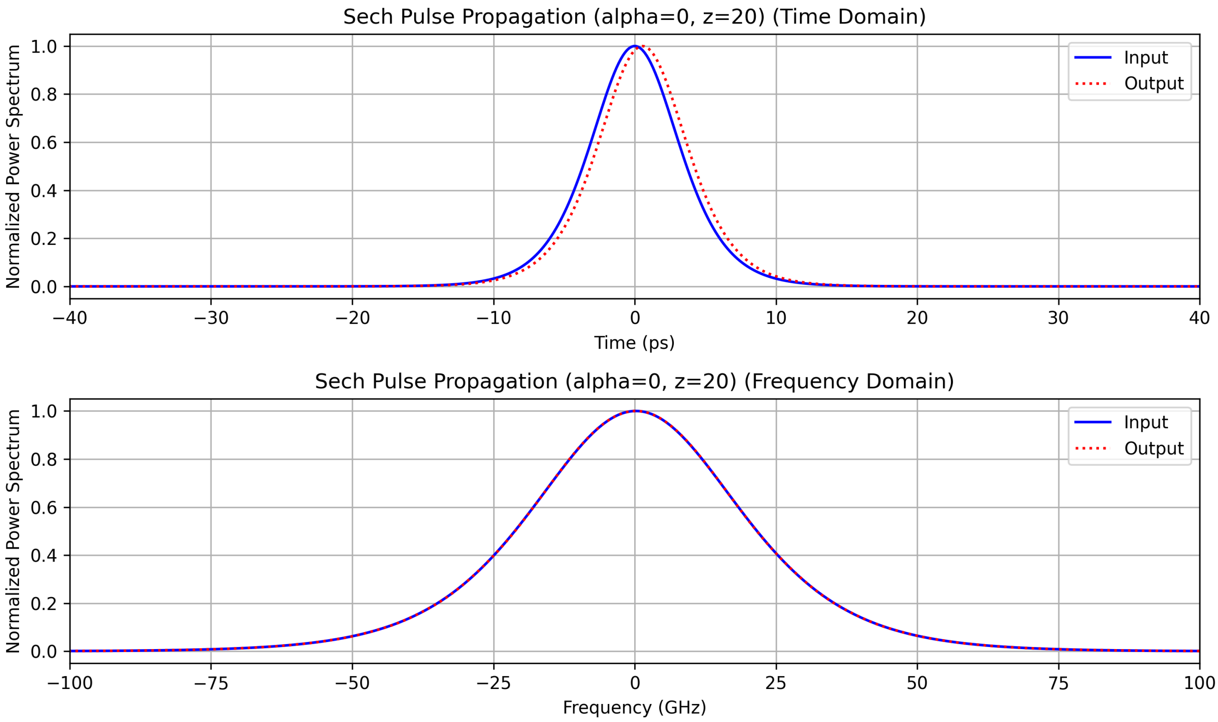

Figure 13: Normalized Power vs Time and Frequency for $\alpha = 0$ km^{-1} and $z = 20$ km.

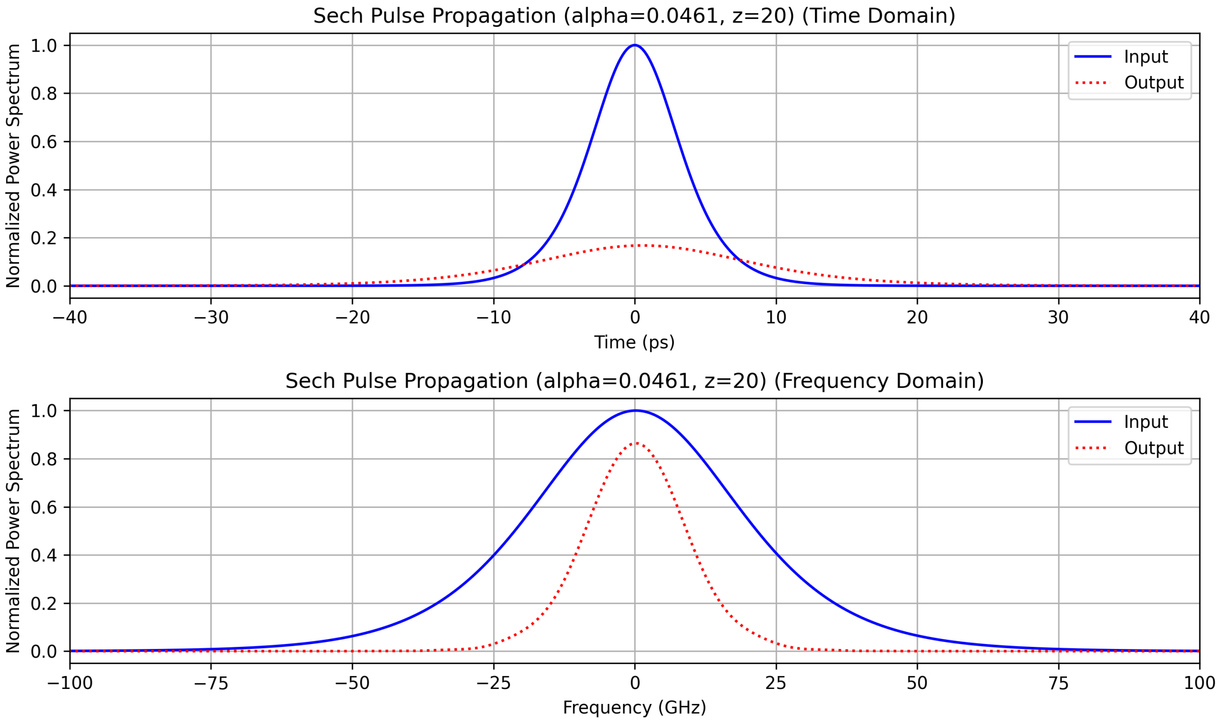

Figure 14: Normalized Power vs Time and Frequency for $\alpha = 0.0461$ km^{-1} and $z = 20$ km.

The code is documented in the Appendix section 4.13.

List of Figures

List of Tables

Nomenclature

FWHM Full Width at Half Maximum

NLSE Nonlinear Schrödinger Equation

w.r.t. With respect to

4 Appendix

4.1 q1_1.py

Python implementation for the calculations and/or visualizations in section 1.1:

```python
# Assumptions
TW: int = 2500  # Time Window in picoseconds
N: int = 2**14  # Number of samples
T_FWHM = 10  # Full Width Half Maximum in ps
C_VALUES = [-10, 0, +5]  # Chirp parameters
A0 = 1  # Peak amplitude (W^1/2)


def sampling_and_frequency_params() -> tuple[float, float, float, float]:
    T_sa: float = TW * 1e-12 / N  # Sampling period in seconds
    F_sa: float = 1 / T_sa  # Sampling frequency in Hz
    Delta_F: float = F_sa / N  # Frequency bin in Hz
    F_min: float = -F_sa / 2  # Minimum frequency based on FFT conventions
    return T_sa, F_sa, Delta_F, F_min


def main() -> None:
    T_sa, F_sa, Delta_F, F_min = sampling_and_frequency_params()
    print(f"a) T_sa : {T_sa:>30} s ({T_sa*1e12} ps)")
    print(f"b) F_sa : {F_sa:>30} Hz ({F_sa*1e-9} GHz)")
    print(f"c) ÎF   : {Delta_F:>30} Hz ({Delta_F*1e-6} MHz)")
    print(f"d) F_min: {F_min:>30} Hz ({F_min*1e-9} GHz)")


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

4.2 q1_2.py

Python implementation for the calculations and/or visualizations in section 1.2:

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

from q1_1 import TW, N, T_FWHM, C_VALUES, A0

# Set figure DPI to 300 (increasing plot resolution)
plt.rcParams["savefig.dpi"] = 300

```

```python
# Calculating T0
T0 = T_FWHM / (2 * np.sqrt(np.log(2)))  # T0 in ps

# Creating the time vector
t = np.linspace(-TW / 2, TW / 2, N)


def electrical_field_envelope(A0: int, T0: float, C: int, t: np.ndarray) ->
↪ np.ndarray:
    return A0 * np.exp(-((1 + 1j * C) / 2) * (t / T0) ** 2)


def power_of_pulse(A_t: np.ndarray) -> np.ndarray:
    return A_t * np.conjugate(A_t)


def main() -> None:
    # Calculate the field envelope A_t and power P_t for every C
    A_t_list = [electrical_field_envelope(A0, T0, C, t) for C in C_VALUES]
    P_t_list = [power_of_pulse(A_t) for A_t in A_t_list]

    # Plot P_t for every C
    plt.figure()

    # Plot for each chirp value
    for i, C in enumerate(C_VALUES):
        plt.plot(t, P_t_list[i], linestyle=["--", "-.", ":"][i],
        ↪  label=f"Chirp C={C}")

    # Plot settings
    plt.xlim(-20, 20)
    plt.ylim(0, 1)
    plt.title("Power vs Time")
    plt.xlabel("Time (ps)")
    plt.ylabel("Power (W)")
    plt.legend()
    plt.grid()
    plt.show()


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

4.3 q1_3.py

Python implementation for the calculations and/or visualizations in section 1.3:

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

from tabulate import tabulate
from q1_1 import N, C_VALUES, A0, sampling_and_frequency_params
from q1_2 import t, T0, electrical_field_envelope, power_of_pulse

# Set figure DPI to 300 (increasing plot resolution)
plt.rcParams["savefig.dpi"] = 300

# Use the sampling period calculated in Q1-1
T_sa, F_sa, Delta_F, F_min = sampling_and_frequency_params()

# Creating frequency vector and shifting zero frequency component to the
↪ center
f = np.fft.fftfreq(N, T_sa)
f = np.fft.fftshift(f)


def normalize(A: np.ndarray, B: np.ndarray = None) -> np.ndarray:
    # Normalize A w.r.t. B (if provided) else normalize A w.r.t. A
    return A / np.max(B) if B is not None and B.size != 0 else A / np.max(A)


def normalized_power_spectrum(A_t: np.ndarray) -> np.ndarray:
    A_f = np.fft.fft(A_t)  # FFT time domain -> frequency domain
    A_f = np.fft.fftshift(A_f)  # Shift FFT
    P_f = power_of_pulse(A_f)  # Calculate power of pulse
    return normalize(P_f)  # Normalize the power spectrum


def measure_FWHM(t: np.ndarray, P: np.ndarray) -> np.float64:
    indices = np.where(P >= np.max(P) / 2)[0]
    return t[indices[-1]] - t[indices[0]]  # Return the difference (FWHM)


def main() -> None:
    # Calculate the field envelope A_t and power P_t for every C
    A_t_list = [electrical_field_envelope(A0, T0, C, t) for C in C_VALUES]
    P_f_list = [normalized_power_spectrum(A_t) for A_t in A_t_list]
```

```python

    # Measure the FWHM width (in GHz) of the spectra
    measured_fwhm_list = [measure_FWHM(f / 1e9, P_f) for P_f in P_f_list]

    # Print the measured values in tabular form
    print(
        tabulate(
            pd.DataFrame({"C": C_VALUES, "Measured FWHM (GHz)": measured_fwhm_list}),
            headers="keys",
            tablefmt="psql",
            showindex=False,
        )
    )

    # Plot normalized power spectrum for each chirp value
    plt.figure(figsize=(10, 6))

    # Plot for each chirp value
    for i, C in enumerate(C_VALUES):
        plt.plot(f / 1e9, P_f_list[i], label=f"Chirp C={C}")

    # Plot settings
    plt.xlim(-750, 750)
    plt.ylim(0, 1)
    plt.xlabel("Frequency (GHz)")
    plt.ylabel("Normalized Power")
    plt.legend()
    plt.grid()
    plt.tight_layout()
    plt.show()


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

4.4 q1_4.py

Python implementation for the calculations and/or visualizations in section 1.4:

```python
import numpy as np
import pandas as pd

from tabulate import tabulate
```

```python
from q1_2 import C_VALUES, A0, T0, t, electrical_field_envelope
from q1_3 import f, normalized_power_spectrum, measure_FWHM


def calculate_F_FWHM(T0: float, C: int) -> np.float64:
    return (np.sqrt(np.log(2)) / (np.pi * T0 * 1e-12)) * np.sqrt(1 + C**2)


def main() -> None:
    # Calculate and verify the spectral widths determined in the previous
    ↪ question
    A_t_list = [electrical_field_envelope(A0, T0, C, t) for C in C_VALUES]
    P_f_list = [normalized_power_spectrum(A_t) for A_t in A_t_list]
    measured_fwhm_list = [measure_FWHM(f, P_f) / 1e9 for P_f in P_f_list]
    theoretical_fwhm_list = [calculate_F_FWHM(T0, C) / 1e9 for C in
    ↪ C_VALUES]

    # Print table with measured and theoretical FWHM values for each C
    ↪ value
    print(
        tabulate(
            pd.DataFrame(
                {
                    "C": C_VALUES,
                    "Measured FHWM (GHz)": measured_fwhm_list,
                    "F_FWHM (GHz)": theoretical_fwhm_list,
                }
            ),
            headers="keys",
            tablefmt="psql",
            showindex=False,
        )
    )


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

4.5 q1_5.py

Python implementation for the calculations and/or visualizations in section 1.5:

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

from tabulate import tabulate
from q1_1 import N, C_VALUES, A0, sampling_and_frequency_params
from q1_2 import T0, t, electrical_field_envelope, power_of_pulse
from q1_3 import measure_FWHM

# Set figure DPI to 300 (increasing plot resolution)
plt.rcParams["savefig.dpi"] = 300

# Get T_sa from Q1-1
T_sa, F_sa, Delta_F, F_min = sampling_and_frequency_params()

# Fiber parameters
beta_2 = -21.68  # Group Velocity Dispersion (ps^2/km)
Z_VALUES = [0, 0.3199, 1.6636, 3.3272]  # Propagation distances in km
f = np.fft.fftfreq(N, T_sa * 1e12)  # Frequency vector in Hz
omega_vector = 2 * np.pi * f  # Angular frequency vector


def spectrum(A_f: np.ndarray, f: np.ndarray, z: float) -> np.ndarray:
    omega_vector = 2 * np.pi * f
    first_term = 1j * (beta_2 / 2) * z * omega_vector**2
    second_term = 0  # assuming Î²_3 = 0
    return A_f * np.exp(first_term + second_term)


def propagate_pulse(A_0, C_values: list[int], z_values: list[float]) ->
↪ dict:
    # Compute the evolution for each chirp value and distance
    results = {}  # Store results for analysis

    # Calculates the pulse propogation for each C and z value
    for C in C_values:
        results[C] = {}
        A_t = electrical_field_envelope(A_0, T0, C, t)
        A_f = np.fft.fft(A_t)  # convert to frequency domain

        # Calculates for each distance
        for z in z_values:
            A_zf = spectrum(A_f, f, z)
            A_zt = np.fft.ifft(A_zf)  # Convert back to time domain
```

```python
                P_zt = power_of_pulse(A_zt)  # b) Calculate power
                results[C][z] = (t, A_zt, P_zt)  # Store results

    return results


def main() -> None:
    # Calculating propogated pulses for each C and z value
    propagated_pulses = propagate_pulse(A0, C_VALUES, Z_VALUES)

    # Measuring FWHM for each C and z value
    table_data = []
    for C in C_VALUES:
        row = {"C": C}
        for i, z in enumerate(Z_VALUES):
            time_vector, A_zt, P_zt = propagated_pulses[C][z]
            header = f"T_FWHM{f'(z_{i})'*bool(z)} (ps)\nz{f'_{i}'*bool(z)} = {z} km"
            row[header] = measure_FWHM(time_vector, P_zt)
        table_data.append(row)

    # Printing the results in a tabular form
    print(
        tabulate(
            pd.DataFrame(table_data), headers="keys", tablefmt="psql", showindex=False
        )
    )

    # Plotting the results, one row of subplots for each z value
    fig, axs = plt.subplots(1, len(Z_VALUES), figsize=(14, 3))

    for j, z in enumerate(Z_VALUES):
        axs[j].set_xlim(-100, 100)

        # Plotting all C values for this z value
        for C in C_VALUES:
            time_vector, A_zt, P_zt = propagated_pulses[C][z]
            axs[j].plot(time_vector, P_zt, label=f"C={C}")

        axs[j].set_xlabel("Time (ps)")
        axs[j].set_ylabel("Normalized Power Spectrum")
        axs[j].legend()
        axs[j].grid()
```

```
        axs[j].set_title(f"z={z} km")

    plt.tight_layout()
    plt.show()


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

4.6 q1_6.py

Python implementation for the calculations and/or visualizations in section 1.6:

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

from q1_2 import A0, T0, C_VALUES
from q1_3 import measure_FWHM
from q1_5 import beta_2, Z_VALUES, propagate_pulse

# Set figure DPI to 300 (increasing plot resolution)
plt.rcParams["savefig.dpi"] = 300

# Define constants and parameters
z_range = np.arange(0, 5.001, 0.001)  # 0-5 km range, 0.001 km interval


def analytical_ratio(beta_2: float, C: int, z: float) -> float:
    return np.sqrt((1 + beta_2 * C * z / T0**2) ** 2 + (beta_2 * z / T0**2) ** 2)


def main() -> None:
    # Calculate the analytical ratio for each C and z
    analytical_ratios = {C: analytical_ratio(beta_2, C, z_range) for C in C_VALUES}

    # Propogate the pulse for each C and z and measure the FWHM
    propagated_pulses = propagate_pulse(A0, C_VALUES, Z_VALUES)
    m_FWHM_values = {C: [] for C in C_VALUES}

    for C in C_VALUES:
        for z in Z_VALUES:
            t, A_zt, P_zt = propagated_pulses[C][z]
```

```python
                measured_FWHM = measure_FWHM(t, P_zt)
                m_FWHM_values[C].append(measured_FWHM)

        # Calculate numerical ratios: TFWHM(z) / TFWHM(0)
        numerical_ratios = {
            C: np.array(m_FWHM_values[C]) / m_FWHM_values[C][0] for C in
            ↪  C_VALUES
        }

        plt.figure(figsize=(10, 6))

        # Plot analytical ratios (line plot)
        for C in C_VALUES:
            plt.plot(z_range, analytical_ratios[C], label=f"Analytical C={C}")

        # Plot numerical ratios (scatter plot)
        plt.scatter(Z_VALUES, numerical_ratios[-10], label="Numerical C=-10",
        ↪  marker="o")
        plt.scatter(Z_VALUES, numerical_ratios[0], label="Numerical C=0",
        ↪  marker="o")
        plt.scatter(Z_VALUES, numerical_ratios[5], label="Numerical C=5",
        ↪  marker="o")

        # Plot settings
        plt.xlabel("Distance z (km)")
        plt.ylabel("Ratio $T_{FWHM1}(z)/T_{FWHM}$")
        plt.legend()
        plt.grid()
        plt.tight_layout()
        plt.show()


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

4.7 q2_1.py

Python implementation for the calculations and/or visualizations in section 2.1:

```python
import numpy as np
import pandas as pd
from tabulate import tabulate

# Given constants
```

```python
T_FWHM = 10 * 1e-12  # s
P0 = 1  # Peak power in W
C = 0  # Chirp parameter
gamma = 1.25  # W^-1 km^-1
alpha = 0.0461  # km^-1
alpha_db = 0.2  # dB km^-1
phi_NL_values = [
    0.5 * np.pi,
    1.5 * np.pi,
    2.5 * np.pi,
    3.5 * np.pi,
]  # Given nonlinear phase shifts


def transmission_distance(
    phi_NL: float, gamma: float, P0: int, alpha: float
) -> tuple[float, float]:
    L_eff = phi_NL / (gamma * P0)
    z = -np.log(1 - alpha * L_eff) / alpha
    return z, L_eff


def transmission_distances() -> pd.DataFrame:
    results = []

    for phi in phi_NL_values:
        z, L_eff = transmission_distance(phi, gamma, P0, alpha)
        results.append(
            {
                "ĎNL_max (radians)": phi,
                "Transmission Distance z (km)": z,
                "Effective Length Leff (km)": L_eff,
            }
        )

    return pd.DataFrame(results)


def main() -> None:
    # Calculate transmission distances and effective lengths
    print("a, b, c)")
    print(tabulate(transmission_distances(), headers="keys",
    ↪ tablefmt="psql"))

```

```python

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

4.8 q2_2.py

Python implementation for the calculations and/or visualizations in section 2.2:

```python
import numpy as np

# Assumptions
TW: float = 2500  # Time Window in picoseconds
N: int = 2**14  # Number of samples


def sampling_and_frequency_params() -> tuple[float, float, float, float]:
    T_sa: float = TW * 1e-12 / N  # Sampling period in seconds
    F_sa: float = 1 / T_sa  # Sampling frequency in Hz
    Delta_F: float = F_sa / N  # Frequency bin in Hz
    F_min: float = -F_sa / 2  # Minimum frequency based on FFT conventions

    return T_sa, F_sa, Delta_F, F_min


def generate_time_and_frequency_vectors(
    T_sa: float, Delta_F: float, F_min: float
) -> tuple[np.ndarray, np.ndarray]:
    # (e) Make a time vector based on the above time choices
    time_vector = np.linspace(-TW / 2 * 1e-12, TW / 2 * 1e-12, N)

    # (f) Make a frequency vector based on the above frequency choices
    frequency_vector = np.linspace(F_min, F_min + (N - 1) * Delta_F, N)

    return time_vector, frequency_vector


def main() -> None:
    # Calculate sampling and frequency parameters
    T_sa, F_sa, Delta_F, F_min = sampling_and_frequency_params()

    # Generate time and frequency vectors
    t, f = generate_time_and_frequency_vectors(T_sa, Delta_F, F_min)

    print(f"a) T_sa : {T_sa:>30} s")
```

```python
    print(f"b) F_sa : {F_sa:>30} Hz ({F_sa/1e9} GHz)")
    print(f"c) ÎF   : {Delta_F:>30} Hz ({Delta_F/1e6} MHz)")
    print(f"d) F_min: {F_min:>30} Hz ({F_min/1e9} GHz)")
    print(f"d) F_max: {F_min:>30} Hz ({F_min/1e9} GHz)")

    print(f"e) Time Vector: {t}")
    print(f"f) Frequency Vector: {f}")


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

4.9 q2_3.py

Python implementation for the calculations and/or visualizations in section 2.3:

```python
import numpy as np
import matplotlib.pyplot as plt
from q2_1 import T_FWHM, C, P0
from q2_2 import sampling_and_frequency_params, generate_time_and_frequency_vectors

# Set figure DPI to 300 (increasing plot resolution)
plt.rcParams["savefig.dpi"] = 300

# Constants
T0 = T_FWHM / (2 * np.sqrt(np.log(2)))
A0 = np.sqrt(P0)  # Peak amplitude

# Calculate parameters from previous steps
T_sa, F_sa, Delta_F, F_min = sampling_and_frequency_params()

# Time and Frequency vectors
t, f = generate_time_and_frequency_vectors(T_sa, Delta_F, F_min)


def electrical_field_envelope(
    A0: int, T0: float, C: list[int], t: np.ndarray
) -> np.ndarray:
    """Calculates pre-chirped Gaussian field envelope"""
    return A0 * np.exp(-((1 + 1j * C) / 2) * (t / T0) ** 2)


def power_of_pulse(A_t: np.ndarray) -> np.ndarray:
```

```python
    """Calculates the power spectrum"""
    return A_t * np.conjugate(A_t)


def measure_FWHM(t: np.ndarray, P: np.ndarray) -> np.float64:
    half_max = np.max(P) / 2
    indices = np.where(P >= half_max)[0]
    return t[indices[-1]] - t[indices[0]]  # Return the difference (FWHM)


def normalize(A: np.ndarray, B: np.ndarray = None) -> np.ndarray:
    # Normalize A w.r.t. B (if provided) else normalize A w.r.t. A
    return A / np.max(B) if B is not None and B.size != 0 else A / np.max(A)


def main() -> None:
    # a, b)Calculate the power of the pulse in time (normalized to temporal peak power)
    A_t = electrical_field_envelope(A0, T0, C, t)
    P_t = power_of_pulse(A_t)

    # Normalize power in time to peak power
    P_t_normalized = normalize(P_t)

    # Calculate the power of the pulse in frequency (normalized to spectral peak power)
    A_f = np.fft.fftshift(np.fft.fft(A_t))  # Frequency-domain representation
    P_f = power_of_pulse(A_f)

    # Normalize power in frequency to peak power
    P_f_normalized = normalize(P_f)

    # Plot the power of the pulse in time and frequency side-by-side
    fig, (ax1, ax2) = plt.subplots(
        1, 2, figsize=(16, 5)
    )  # Create side-by-side subplots

    # Plot the power of the pulse in time domain
    ax1.plot(t * 1e12, P_t_normalized, label=f"Power of Pulse (C = {C})")
    ax1.set_xlabel("Time (ps)")
    ax1.set_ylabel("Normalized Power")
    ax1.set_title("Normalized Power of Gaussian Pulse in Time Domain")
    ax1.grid()
```

```python
    ax1.set_xlim(-20, 20)
    ax1.set_ylim(0, 1)

    # Plot the power of the pulse in frequency domain
    ax2.plot(f * 1e-9, P_f_normalized, label=f"Power of Pulse (C = {C})")
    ax2.set_xlabel("Frequency (GHz)")
    ax2.set_ylabel("Normalized Power")
    ax2.set_title("Normalized Power of Gaussian Pulse in Frequency Domain")
    ax2.grid()
    ax2.set_xlim(-100, 100)
    ax2.set_ylim(0, 1)

    # Show both plots side-by-side
    plt.tight_layout()
    plt.show()

    # State Full Width Half Maximum for the pulse in both time and frequency
    # Calculate the FWHM in time and frequency
    FWHM_time = measure_FWHM(t, P_t_normalized) * 1e12  # in ps
    FWHM_freq = measure_FWHM(f, P_f_normalized) * 1e-9  # in GHz

    print(f"(c) FWHM in Time Domain: {FWHM_time:.2f} ps")
    print(f"(c) FWHM in Frequency Domain: {FWHM_freq:.2f} GHz")


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

4.10 q2_4.py

Python implementation for the calculations and/or visualizations in section 2.4:

```python
import numpy as np
import matplotlib.pyplot as plt
from q2_1 import P0, C, alpha, transmission_distances
from q2_2 import sampling_and_frequency_params, generate_time_and_frequency_vectors
from q2_3 import T0, electrical_field_envelope, power_of_pulse, normalize

plt.rcParams["savefig.dpi"] = 300

# Get transmission lengths from Q2-1
transmission_values = transmission_distances()
```

```python
z_values = transmission_values["Transmission Distance z (km)"]
L_eff_values = transmission_values["Effective Length Leff (km)"]

# Get parameters from Q2-2
T_sa, F_sa, Delta_F, F_min = sampling_and_frequency_params()

# Generate Time and Frequency Vectors
t, freq = generate_time_and_frequency_vectors(T_sa, Delta_F, F_min)
f = np.fft.fftshift(freq)

# Define Gaussian pulse in time and frequency domains
beta_2 = -21.68 * 1e-24  # [s**2 km**-1]
w = 2 * np.pi * f  # Angular frequency


def propogate_pulse(vector: np.ndarray, z: float, L_eff: float) -> np.ndarray:
    power = power_of_pulse(vector)
    return np.sqrt(power) * np.exp(-alpha * z * 0.5) * np.exp(1j * power * L_eff)


def main() -> None:
    # Calculate the electrical field envelope
    A_0_t = electrical_field_envelope(P0, T0, C, t)  # Time Domain
    A_0_w = np.fft.fft(A_0_t)  # Frequency Domain

    A_z_t_values = []
    A_z_w_values = []
    for z, L_eff in zip(z_values, L_eff_values):
        A_z_t = propogate_pulse(A_0_t, z, L_eff)
        A_z_w = np.fft.fft(A_z_t)
        A_z_t_values.append(A_z_t)
        A_z_w_values.append(A_z_w)

    # Normalizing the Power Spectra
    P_0_w = power_of_pulse(A_0_w)
    P_0_w_norm = normalize(P_0_w)

    # Input Power Spectra
    plt.plot(f * 1e-9, P_0_w_norm, label=f"Input C={C}")

    # Output Power Spectra
    for A_z_w, z in zip(A_z_w_values, z_values):
```

```
        P_z_w = power_of_pulse(A_z_w)
        P_z_w = normalize(P_z_w, P_0_w)  # normalizing with the input max
        plt.plot(f * 1e-9, P_z_w, "-", label=f"Output C={C}, z={round(z,2)} km")

    plt.xlabel("Frequency [GHz]")
    plt.ylabel("Normalized Power w.r.t. Input")
    plt.xlim(-300, 300)
    plt.ylim(0, 1)
    plt.legend()
    plt.grid()
    plt.tight_layout()
    plt.show()


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

4.11 q3_1.py

Python implementation for the calculations and/or visualizations in section 3.1:

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

# Set figure DPI to 300 (increasing plot resolution)
plt.rcParams["savefig.dpi"] = 300

# Define constants
TW = 2500e-12  # Total time window (2500 ps)
N = 2**14  # Number of samples

# Time and frequency vectors
t = np.linspace(-TW / 2, TW / 2, N)
fsa = 1 / (t[1] - t[0])  # Sampling frequency
f = np.linspace(-fsa / 2, fsa / 2, N)
w = 2 * np.pi * f  # Angular frequency

# Fiber and pulse parameters
T_FWHM = 10e-12  # 10 ps
A0 = 1  # W^(1/2)
C_values = [-10, 0, 5]
T0 = T_FWHM / (2 * np.sqrt(np.log(2)))

```

```python
# Dispersive case (Question 3-1)
alpha = 0   # km^-1
gamma = 0   # W^-1 km^-1
beta_2 = -21.68e-24  # s^2/km
beta_3 = 0
z_values = [0.3199, 1.6636, 3.3272]  # km
N_seg = 5000


def normalize(A: np.ndarray, B: np.ndarray = None) -> np.ndarray:
    # Normalize A w.r.t. B (if provided) else normalize A w.r.t. A
    return A / np.max(B) if B is not None and B.size != 0 else A / np.max(A)


def power_of_pulse(A: np.ndarray) -> np.ndarray:
    return A * np.conj(A)


def create_pulse(t: np.ndarray, A0: int, T0: float, C: int) -> np.ndarray:
    return A0 * np.exp(-(1 + 1j * C) * (t**2) / (2 * T0**2))


def split_step(
    A: np.ndarray,
    z: float,
    w: np.ndarray,
    beta_2: float,
    beta_3: float,
    alpha: float,
    gamma: float,
    N_seg: int,
) -> np.ndarray:
    dz = z / N_seg

    # Calculate dispersive phase
    beta_2_term = 1j * (beta_2 / 2) * w**2
    beta_3_term = 1j * (beta_3 / 6) * w**3
    dispersive_phase = np.exp((beta_2_term + beta_3_term - alpha / 2) * dz)

    for _ in range(N_seg):
        # Dispersive step (frequency domain)
        A_w = np.fft.fftshift(np.fft.fft(A))  # Use fftshift before fft
        A_w *= dispersive_phase  # Apply dispersive phase

```

```python
        # Non-linear step (time domain)
        A = np.fft.ifft(np.fft.ifftshift(A_w))  # Use ifftshift before ifft
        A *= np.exp(1j * gamma * np.abs(A) ** 2 * dz)

    return A


def analytical_dispersive(
    t: np.ndarray, A0: int, T0: int, C: int, z: float
) -> np.ndarray:
    Q = 1 + ((1j * beta_2 * z) / T0**2)
    return (A0 / np.sqrt(Q)) * np.exp(-(1 + 1j * C) * (t**2) / (2 * T0**2 *
     ↪  Q))


def plot_separate_per_C() -> None:
    for C in C_values:
        fig, axs = plt.subplots(2, len(z_values), figsize=(18, 6))

        # Time domain
        A_in = create_pulse(t, A0, T0, C)
        P_in = power_of_pulse(A_in)  # Calculate power
        P_in_norm = normalize(P_in)  # Normalize Power

        # Frequency domain
        A_in_w = np.fft.fftshift(np.fft.fft(A_in))
        P_in_w = power_of_pulse(A_in_w)
        P_in_w_norm = normalize(P_in_w)

        for j, z in enumerate(z_values):
            # Time Domain calculations
            A_out = split_step(A_in, z, w, beta_2, beta_3, alpha, gamma,
             ↪  N_seg)
            P_out = power_of_pulse(A_out)
            P_out_norm = normalize(P_out, P_in)  # Normalize to input peak

            # Time domain subplot
            axs[0, j].plot(t * 1e12, P_in_norm, "b-", label=f"Input")
            axs[0, j].plot(t * 1e12, P_out_norm, "r--", label=f"Output")
            axs[0, j].set_xlim(-200, 200)
            axs[0, j].set_xlabel("Time (ps)", fontsize=10)
            axs[0, j].set_ylabel("Normalized Power", fontsize=10)
            axs[0, j].set_title(f"Time Domain (C={C}, z = {z:.4f} km)",
             ↪  fontsize=11)
```

```python
            axs[0, j].grid()
            axs[0, j].legend(fontsize=9)

            # Frequency domain calculations
            A_out_w = np.fft.fftshift(np.fft.fft(A_out))
            P_out_w = power_of_pulse(A_out_w)
            P_out_w_norm = normalize(P_out_w, P_in_w)  # Normalize to input peak

            # Frequency domain subplot
            axs[1, j].plot(f * 1e-12, P_in_w_norm, "b-", label="Input")
            axs[1, j].plot(f * 1e-12, P_out_w_norm, "r--", label="Output")
            axs[1, j].set_xlim(-1, 1)
            axs[1, j].set_xlabel("Frequency (THz)", fontsize=10)
            axs[1, j].set_ylabel("Normalized Power Spectrum", fontsize=10)
            axs[1, j].set_title(
                f"Frequency Domain (C={C}, z = {z:.4f} km)", fontsize=11
            )
            axs[1, j].grid()
            axs[1, j].legend(fontsize=9)

        plt.tight_layout()
        plt.show()


def sanity_check() -> None:
    C_check = 0  # Use pulse without chirp for simplification
    z_check = z_values[-1]  # Use the longest propagation distance

    A_in = create_pulse(t, A0, T0, C_check)  # Input signal
    A_out_numerical = split_step(A_in, z_check, w, beta_2, beta_3, alpha, gamma, N_seg)
    A_out_analytical = analytical_dispersive(t, A0, T0, C_check, z_check)
    P_out_numerical = power_of_pulse(A_out_numerical)
    P_out_analytical = power_of_pulse(A_out_analytical)

    # Normalize the outputs to their own peak values
    A_out_numerical_normalized = normalize(P_out_numerical)
    A_out_analytical_normalized = normalize(P_out_analytical)

    # Plotting the sanity check
    plt.figure(figsize=(8, 5))
    plt.plot(t * 1e12, A_out_numerical_normalized, "b-", label="Simulated")
```

```
    plt.plot(t * 1e12, A_out_analytical_normalized, "r--", label="Analytical")
    plt.xlabel("Time (ps)", fontsize=12)
    plt.ylabel("Normalized Power", fontsize=12)
    plt.legend()
    plt.xlim(-50, 50)
    plt.ylim(0, 1)
    plt.grid()
    plt.tight_layout()
    plt.show()


def main() -> None:
    plot_separate_per_C()  # Generate separate plots per C value
    sanity_check()


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

4.12 q3_2.py

Python implementation for the calculations and/or visualizations in section 3.2:

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

from q3_1 import split_step

# Set figure DPI to 300 (increasing plot resolution)
plt.rcParams["savefig.dpi"] = 300

# Define constants
TW = 2500e-12  # Total time window (2500 ps)
N = 2**14  # Number of samples
N_seg = 5000  # Number of segments

# Time and frequency vectors
t = np.linspace(-TW / 2, TW / 2, N)
fsa = 1 / (t[1] - t[0])  # Sampling frequency
f = np.linspace(-fsa / 2, fsa / 2, N)
w = 2 * np.pi * f  # Angular frequency

# Fiber and pulse parameters
```

```python
T_FWHM = 10e-12  # 10 ps
A0 = 1  # W^(1/2)
T0 = T_FWHM / (2 * np.sqrt(np.log(2)))
alpha = 0.0461  # km^-1
gamma = 1.25  # W^-1 km^-1
beta2 = 0  # ps^2/km
beta3 = 0  # ps^3/km
z_values = [1.2945, 4.1408, 7.4172, 11.2775]  # km


# Function to create input pulse
def create_pulse(t, A0, T0):
    return A0 * np.exp(-(t**2) / (2 * T0**2))


def analytical_nonlinear(t, A0, T0, z, alpha, gamma):
    L_eff = (1 - np.exp(-alpha * z)) / alpha
    P0 = np.abs(A0) ** 2
    return np.abs(A0) * np.exp(-alpha * z / 2) * np.exp(1j * gamma * P0 *
     ↪ L_eff)


# Function to plot all z-values in time and frequency domains
def plot_all_z(t, f, A_in, z_values, w, beta2, beta3, alpha, gamma, N_seg):
    plt.figure(figsize=(12, 8))

    # Time domain subplot
    plt.subplot(2, 1, 1)
    plt.plot(
        t * 1e12, np.abs(A_in) ** 2 / np.max(np.abs(A_in) ** 2), "k--",
         ↪ label="Input"
    )
    for z in z_values:
        A_out = split_step(A_in, z, w, beta2, beta3, alpha, gamma, N_seg)
        plt.plot(
            t * 1e12,
            np.abs(A_out) ** 2 / np.max(np.abs(A_in) ** 2),
            label=f"z = {z:.4f} km",
        )
    plt.xlabel("Time (ps)")
    plt.ylabel("Normalized Power")
    plt.title("Time Domain for Different Propagation Distances")
    plt.legend()
    plt.grid(True)
```

```
    plt.xlim(-50, 50)
    plt.ylim(0, 1)

    # Frequency domain subplot
    A_in_w = np.fft.fftshift(np.fft.fft(A_in))
    plt.subplot(2, 1, 2)
    plt.plot(
        f * 1e-12,
        np.abs(A_in_w) ** 2 / np.max(np.abs(A_in_w) ** 2),
        "k--",
        label="Input",
    )
    for z in z_values:
        A_out = split_step(A_in, z, w, beta2, beta3, alpha, gamma, N_seg)
        A_out_w = np.fft.fftshift(np.fft.fft(A_out))
        plt.plot(
            f * 1e-12,
            np.abs(A_out_w) ** 2 / np.max(np.abs(A_in_w) ** 2),
            label=f"z = {z:.4f} km",
        )
    plt.xlabel("Frequency (THz)")
    plt.ylabel("Normalized Power Spectrum")
    plt.title("Frequency Domain for Different Propagation Distances")
    plt.legend()
    plt.grid(True)
    plt.xlim(-0.5, 0.5)
    plt.ylim(0, 1)
    plt.tight_layout()
    plt.show()


def sanity_check(A_in):
    zcheck = z_values[-1]  # Use the longest propagation distance
    A_out_numerical = split_step(A_in, zcheck, w, beta2, beta3, alpha,
    ↪ gamma, N_seg)
    A_out_analytical = A_out_analytical = analytical_nonlinear(
        t, A_in, T0, zcheck, alpha, gamma
    )

    # Normalize the outputs to their own peak values
    A_out_numerical_normalized = np.abs(A_out_numerical) ** 2 / np.max(
        np.abs(A_out_numerical) ** 2
    )
    A_out_analytical_normalized = np.abs(A_out_analytical) ** 2 / np.max(
```

```python
        np.abs(A_out_analytical) ** 2
    )

    # Plotting the sanity check
    plt.figure(figsize=(8, 5))
    plt.plot(t * 1e12, A_out_numerical_normalized, "b-", label="Simulated")
    plt.plot(t * 1e12, A_out_analytical_normalized, "r--", label="Analytical")
    plt.xlabel("Time (ps)", fontsize=12)
    plt.ylabel("Normalized Power", fontsize=12)
    # plt.title('Sanity Check: Simulated vs. Analytical Output Pulse', fontsize=14, weight='bold')
    plt.legend()
    plt.xlim(-50, 50)
    plt.ylim(0, 1)
    plt.grid(True)
    plt.tight_layout()
    plt.show()


# Main function to run the simulation
def main():
    # Create input pulse
    A_in = create_pulse(t, A0, T0)

    # Plot all z-values in both time and frequency domains
    plot_all_z(t, f, A_in, z_values, w, beta2, beta3, alpha, gamma, N_seg)
    sanity_check(A_in)


# Execute the main function if this script is run directly
if __name__ == "__main__":
    main()
```

4.13 q3_3.py

Python implementation for the calculations and/or visualizations in section 3.3:

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

from q3_1 import power_of_pulse, normalize

# Set figure DPI to 300 (increasing plot resolution)
```

```python
plt.rcParams["savefig.dpi"] = 300

TW = 2500e-12  # Total time window (2500 ps)
N = 2**14  # Number of samples
N_seg = 5000  # Number of segments

# Specified time and frequency vectors
t = np.linspace(-TW / 2, TW / 2, N)
fsa = 1 / (t[1] - t[0])  # Sampling frequency
f = np.linspace(-fsa / 2, fsa / 2, N)
w = 2 * np.pi * f  # Angular frequency

# Assumption
A0 = 1  # W^(1/2)
P0 = abs(A0) ** 2
gamma = 1.25  # W^-1 km^-1
beta_2 = -21.68e-24  # s^2/km
beta_3 = 0  # s^3/km
L = 20  # km


def create_sech_pulse(t: np.ndarray, A0: float, T0: float) -> np.ndarray:
    return A0 / np.cosh(t / T0)


def split_step(A: np.ndarray, z: float, alpha: float) -> np.ndarray:
    dz = z / N_seg

    # Precompute dispersive phase term (assuming beta_3=0) for efficiency
    dispersive_phase = np.exp((1j * (beta_2 / 2) * w**2 - alpha / 2) * dz)

    for _ in range(N_seg):
        # Dispersive step (frequency domain)
        A_w = np.fft.fftshift(np.fft.fft(A))  # Use fftshift before fft
        A_w *= dispersive_phase

        # Non-linear step (time domain)
        A = np.fft.ifft(np.fft.ifftshift(A_w))  # Use ifftshift before ifft
        A *= np.exp(1j * gamma * np.abs(A) ** 2 * dz)

    return A


def plot_results(
```

```
    t: np.ndarray, f: np.ndarray, A_in: np.ndarray, A_out: np.ndarray,
     ↪  title: str
):
    # Calculate the power in Time Domain
    P_in = power_of_pulse(A_in)
    P_out = power_of_pulse(A_out)

    # Normalize the power
    P_in_norm = normalize(P_in)  # normalize to self
    P_out_norm = normalize(P_out, P_in)  # normalize to peak input

    # Plot settings
    plt.figure(figsize=(10, 6))
    plt.subplot(2, 1, 1)
    plt.plot(t * 1e12, P_in_norm, "b-", label="Input")
    plt.plot(t * 1e12, P_out_norm, "r:", label="Output")
    plt.xlabel("Time (ps)")
    plt.xlim(-40, 40)
    plt.ylabel("Normalized Power Spectrum")
    plt.title(f"{title} (Time Domain)")
    plt.legend()
    plt.grid()

    # Calculate the power in Frequency Domain
    A_in_w = np.fft.fftshift(np.fft.fft(A_in))
    A_out_w = np.fft.fftshift(np.fft.fft(A_out))

    P_in_w = power_of_pulse(A_in_w)
    P_out_w = power_of_pulse(A_out_w)

    # Normalize the power
    P_in_w_norm = normalize(P_in_w)  # normalize to self
    P_out_w_norm = normalize(P_out_w, P_in_w)  # normalize to peak input

    # Plot settings
    plt.subplot(2, 1, 2)
    plt.plot(f * 1e-9, P_in_w_norm, "b-", label="Input")
    plt.plot(f * 1e-9, P_out_w_norm, "r:", label="Output")
    plt.xlabel("Frequency (GHz)")
    plt.xlim(-100, 100)
    plt.ylabel("Normalized Power Spectrum")
    plt.title(f"{title} (Frequency Domain)")
    plt.legend()
    plt.grid()
```

```python
    plt.tight_layout()
    plt.show()


def main() -> None:
    # Calculate pulse width for the soliton
    T0_sech_pulse = np.sqrt(np.abs(beta_2) / (gamma * P0))  # hyperbolic
    ↪  secant pulse
    print(f"T0 for the sech pulse is {T0_sech_pulse*1e12} ps")
    quit()
    # Create the initial sech pulse
    A_in = create_sech_pulse(t, A0, T0_sech_pulse)

    # Propagation without loss
    alpha = 0  # km^-1
    A_out = split_step(A_in, L, alpha)
    plot_results(t, f, A_in, A_out, f"Sech Pulse Propagation (alpha={alpha},
    ↪  z={L})")

    # Introduce loss
    alpha = 0.0461  # km^-1
    A_out_l = split_step(A_in, L, alpha)
    plot_results(t, f, A_in, A_out_l, f"Sech Pulse Propagation
    ↪  (alpha={alpha}, z={L})")


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