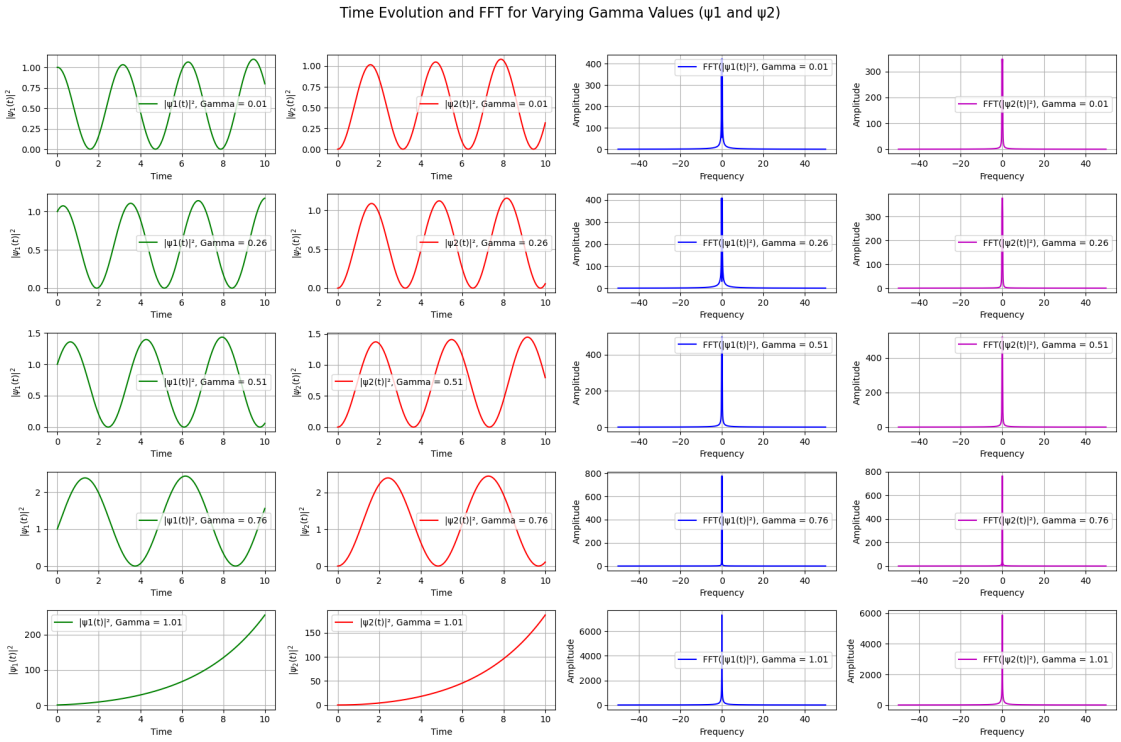
Katha_Computational_Physics_01

September 16, 2024

[1]:
```
import numpy as np
import matplotlib.pyplot as plt

# Constants
hbar = 1.0  # Assume hbar = 1 for simplicity
dt = 0.01  # Time step
t_max = 10  # Maximum time
n_steps = int(t_max / dt)  # Number of time steps
gamma_values = np.linspace(0.01, 1.01, 5)  # Gamma values from 0.01 to 1.01

# Set up subplots
fig, axs = plt.subplots(len(gamma_values), 4, figsize=(18, 12))
fig.suptitle('Time Evolution and FFT for Varying Gamma Values ( 1 and  2)',␣
 ↪fontsize=16)

# Loop over gamma values
for idx, gamma in enumerate(gamma_values):

    # Define the initial wavefunction (start in state |0>)
    psi_0 = np.array([1 + 0j, 0 + 0j])  # Initial wavefunction [psi_0, psi_1]

    # Define the Hamiltonian matrix
    H = np.array([[1j * gamma, -1], [-1, -1j*gamma]])

    # Arrays to store the wavefunction components over time
    psi_0_t = np.zeros(n_steps, dtype=complex)  # For  1(t)
    psi_1_t = np.zeros(n_steps, dtype=complex)  # For  2(t)

    # Set initial condition
    psi_0_t[0] = psi_0[0]
    psi_1_t[0] = psi_0[1]

    # Euler-Cromer method loop
    for i in range(1, n_steps):
        # Compute the derivative of the wavefunction
        dpsi_dt = -1j * np.dot(H, psi_0)
```

```
        # Update the wavefunction using Euler-Cromer method
        psi_0 += dt * dpsi_dt  # Update both components

        # Store the wavefunction components for plotting
        psi_0_t[i] = psi_0[0]  # 1(t)
        psi_1_t[i] = psi_0[1]  # 2(t)

    # Time array
    tlist = np.linspace(0, t_max, n_steps)

    # Perform FFT on both 1 and 2
    fft_psi_0 = np.fft.fft(psi_0_t)
    fft_psi_1 = np.fft.fft(psi_1_t)
    fft_freqs = np.fft.fftfreq(n_steps, dt)

    # Shift the FFT and frequencies for proper negative/positive plotting
    fft_psi_0_shifted = np.fft.fftshift(fft_psi_0)
    fft_psi_1_shifted = np.fft.fftshift(fft_psi_1)
    fft_freqs_shifted = np.fft.fftshift(fft_freqs)

    # Plot time-domain |1(t)|^2
    axs[idx, 0].plot(tlist, np.abs(psi_0_t)**2, label=f'|1(t)|², Gamma =␣
↪{gamma:.2f}', color='g')
    axs[idx, 0].set_xlabel('Time')
    axs[idx, 0].set_ylabel(r'$|\psi_1(t)|^2$')
    axs[idx, 0].legend()
    axs[idx, 0].grid(True)

    # Plot time-domain |2(t)|^2
    axs[idx, 1].plot(tlist, np.abs(psi_1_t)**2, label=f'|2(t)|², Gamma =␣
↪{gamma:.2f}', color='r')
    axs[idx, 1].set_xlabel('Time')
    axs[idx, 1].set_ylabel(r'$|\psi_2(t)|^2$')
    axs[idx, 1].legend()
    axs[idx, 1].grid(True)

    # Plot FFT of |1(t)| (with negative and positive frequencies)
    axs[idx, 2].plot(fft_freqs_shifted, np.abs(fft_psi_0_shifted),␣
↪label=f'FFT(|1(t)|²), Gamma = {gamma:.2f}', color='b')
    axs[idx, 2].set_xlabel('Frequency')
    axs[idx, 2].set_ylabel('Amplitude')
    axs[idx, 2].legend()
    axs[idx, 2].grid(True)

    # Plot FFT of |2(t)| (with negative and positive frequencies)
    axs[idx, 3].plot(fft_freqs_shifted, np.abs(fft_psi_1_shifted),␣
↪label=f'FFT(|2(t)|²), Gamma = {gamma:.2f}', color='m')
```

```
    axs[idx, 3].set_xlabel('Frequency')
    axs[idx, 3].set_ylabel('Amplitude')
    axs[idx, 3].legend()
    axs[idx, 3].grid(True)

plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```



1 Solve ψ_1 & ψ_2, Plot ${\psi_1}^2$ & ${\psi_2}^2$ in Time and Frequency Domain:

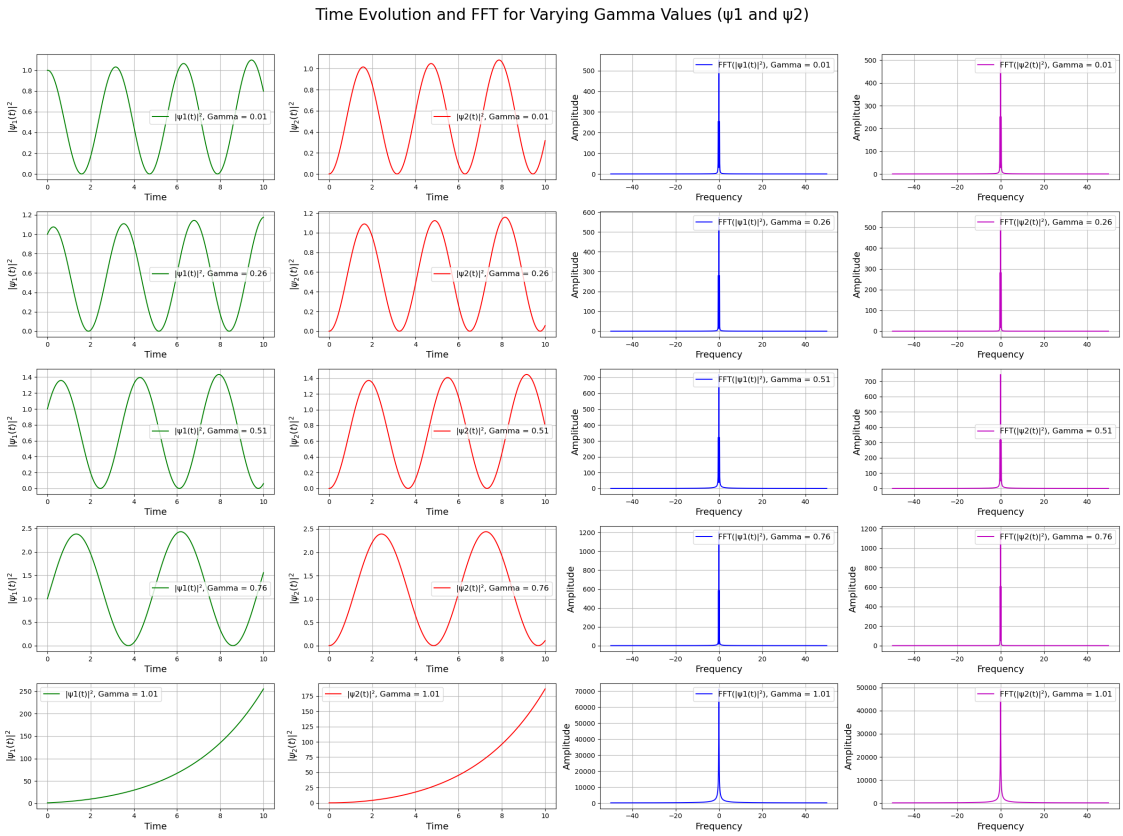
[11]:
```
import numpy as np
import matplotlib.pyplot as plt

# Constants
hbar = 1.0  # Assume hbar = 1 for simplicity
dt = 0.01  # Time step
t_max = 10  # Maximum time
n_steps = int(t_max / dt)  # Number of time steps
gamma_values = np.linspace(0.01, 1.01, 5)  # Gamma values from 0.01 to 1.01

# Set up subplots with enlarged figure size
```

```
fig, axs = plt.subplots(len(gamma_values), 4, figsize=(24, 18))  # Increased␣
 ↪figsize for larger plots
fig.suptitle('Time Evolution and FFT for Varying Gamma Values ( 1 and  2)',␣
 ↪fontsize=24)  # Larger title font size

# Loop over gamma values
for idx, gamma in enumerate(gamma_values):

    # Define the initial wavefunction (start in state |0>)
    psi_0 = np.array([1 + 0j, 0 + 0j])  # Initial wavefunction [psi_0, psi_1]

    # Define the Hamiltonian matrix
    H = np.array([[1j * gamma, -1], [-1, -1j*gamma]])

    # Arrays to store the wavefunction components over time
    psi_0_t = np.zeros(n_steps, dtype=complex)  # For  1(t)
    psi_1_t = np.zeros(n_steps, dtype=complex)  # For  2(t)

    # Set initial condition
    psi_0_t[0] = psi_0[0]
    psi_1_t[0] = psi_0[1]

    # Euler-Cromer method loop
    for i in range(1, n_steps):
        # Compute the derivative of the wavefunction
        dpsi_dt = -1j * np.dot(H, psi_0)

        # Update the wavefunction using Euler-Cromer method
        psi_0 += dt * dpsi_dt  # Update both components

        # Store the wavefunction components for plotting
        psi_0_t[i] = psi_0[0]  #  1(t)
        psi_1_t[i] = psi_0[1]  #  2(t)

    # Time array
    tlist = np.linspace(0, t_max, n_steps)

    # Perform FFT on both  1 and  2
    fft_psi_0 = np.fft.fft(abs(psi_0_t)**2)
    fft_psi_1 = np.fft.fft(abs(psi_1_t)**2)
    fft_freqs = np.fft.fftfreq(n_steps, dt)

    # Shift the FFT and frequencies for proper negative/positive plotting
    fft_psi_0_shifted = np.fft.fftshift(fft_psi_0)
    fft_psi_1_shifted = np.fft.fftshift(fft_psi_1)
    fft_freqs_shifted = np.fft.fftshift(fft_freqs)
```

```
    # Find the frequency with the maximum amplitude for  1 and  2
    peak_freq_psi_0 = fft_freqs_shifted[np.argmax(np.abs(fft_psi_0_shifted))]
    peak_freq_psi_1 = fft_freqs_shifted[np.argmax(np.abs(fft_psi_1_shifted))]

    # Print the peak frequencies for both  1 and  2
    print(f"Gamma = {gamma:.2f}: Peak frequency for  1 = {peak_freq_psi_0:.4f},␣
↪Peak frequency for  2 = {peak_freq_psi_1:.4f}")

    # Plot time-domain | 1(t)|^2
    axs[idx, 0].plot(tlist, np.abs(psi_0_t)**2, label=f'| 1(t)|², Gamma =␣
↪{gamma:.2f}', color='g')
    axs[idx, 0].set_xlabel('Time', fontsize=14)
    axs[idx, 0].set_ylabel(r'$|\psi_1(t)|^2$', fontsize=14)
    axs[idx, 0].legend(fontsize=12)
    axs[idx, 0].grid(True)

    # Plot time-domain | 2(t)|^2
    axs[idx, 1].plot(tlist, np.abs(psi_1_t)**2, label=f'| 2(t)|², Gamma =␣
↪{gamma:.2f}', color='r')
    axs[idx, 1].set_xlabel('Time', fontsize=14)
    axs[idx, 1].set_ylabel(r'$|\psi_2(t)|^2$', fontsize=14)
    axs[idx, 1].legend(fontsize=12)
    axs[idx, 1].grid(True)

    # Plot FFT of | 1(t)| (with negative and positive frequencies)
    axs[idx, 2].plot(fft_freqs_shifted, np.abs(fft_psi_0_shifted),␣
↪label=f'FFT(| 1(t)|²), Gamma = {gamma:.2f}', color='b')
    axs[idx, 2].set_xlabel('Frequency', fontsize=14)
    axs[idx, 2].set_ylabel('Amplitude', fontsize=14)
    axs[idx, 2].legend(fontsize=12)
    axs[idx, 2].grid(True)

    # Plot FFT of | 2(t)| (with negative and positive frequencies)
    axs[idx, 3].plot(fft_freqs_shifted, np.abs(fft_psi_1_shifted),␣
↪label=f'FFT(| 2(t)|²), Gamma = {gamma:.2f}', color='m')
    axs[idx, 3].set_xlabel('Frequency', fontsize=14)
    axs[idx, 3].set_ylabel('Amplitude', fontsize=14)
    axs[idx, 3].legend(fontsize=12)
    axs[idx, 3].grid(True)

plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```

```
Gamma = 0.01: Peak frequency for  1 = 0.0000, Peak frequency for  2 = 0.0000
Gamma = 0.26: Peak frequency for  1 = 0.0000, Peak frequency for  2 = 0.0000
Gamma = 0.51: Peak frequency for  1 = 0.0000, Peak frequency for  2 = 0.0000
Gamma = 0.76: Peak frequency for  1 = 0.0000, Peak frequency for  2 = 0.0000
```

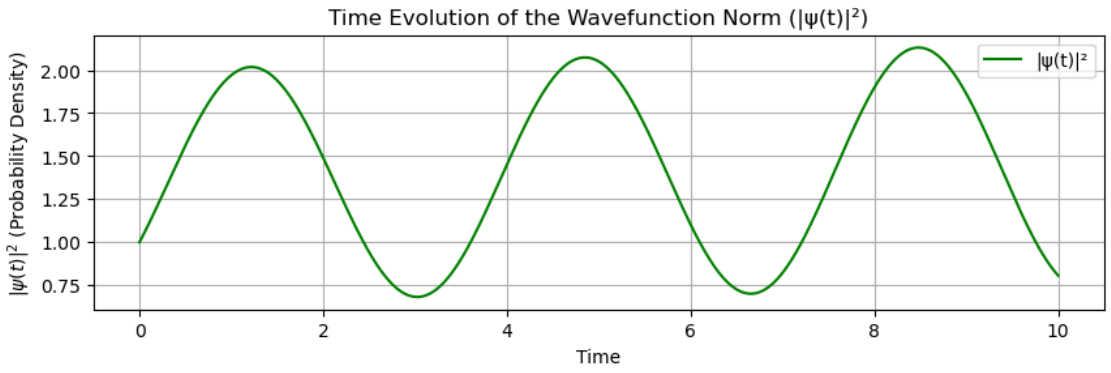
```
Gamma = 1.01: Peak frequency for  1 = 0.0000, Peak frequency for  2 = 0.0000
```

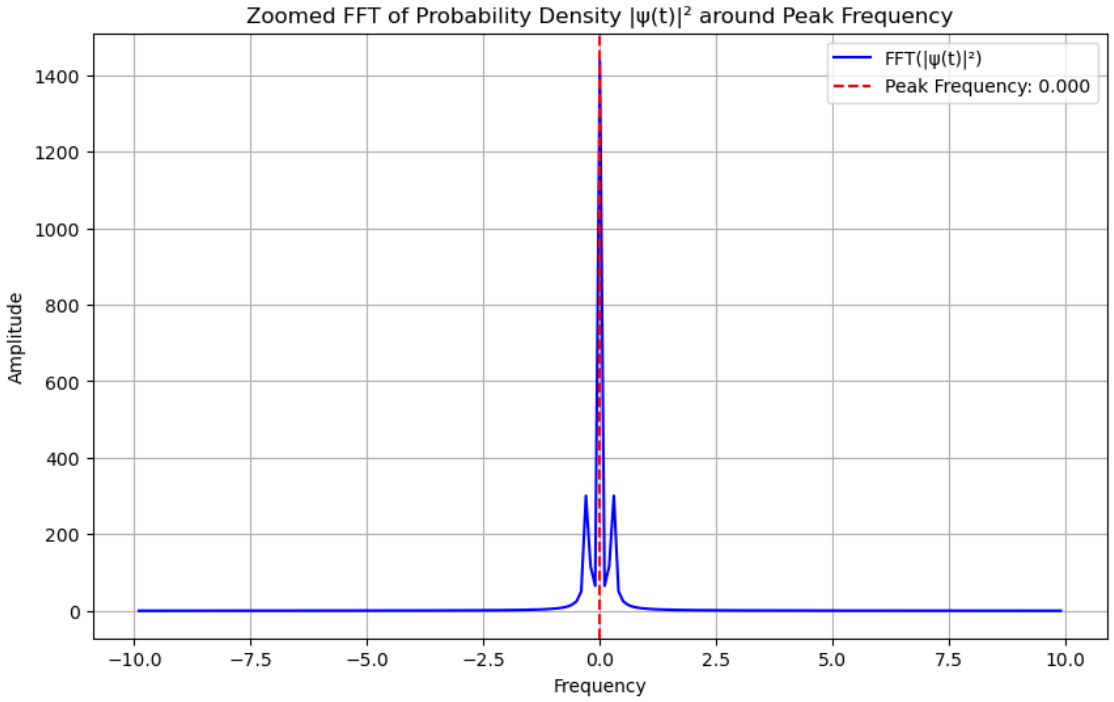


[16]:
```python
import numpy as np
import matplotlib.pyplot as plt

# Constants
gamma = 0.5
hbar = 1.0  # Assume hbar = 1 for simplicity
dt = 0.01  # Time step
t_max = 10  # Maximum time
n_steps = int(t_max / dt)  # Number of time steps

# Define the initial wavefunction (start in state |0>)
psi_0 = np.array([1 + 0j, 0 + 0j])  # Initial wavefunction [psi_0, psi_1]

# Define the Hamiltonian matrix
H = np.array([[1j * gamma, -1], [-1, -1j*gamma]])

# Arrays to store the wavefunction components over time
psi_0_t = np.zeros(n_steps, dtype=complex)
psi_1_t = np.zeros(n_steps, dtype=complex)
```

```
psi_norm_t = np.zeros(n_steps)  # Array to store |psi(t)|^2 (norm)

# Set initial condition
psi_0_t[0] = psi_0[0]
psi_1_t[0] = psi_0[1]
psi_norm_t[0] = np.abs(psi_0[0])**2 + np.abs(psi_0[1])**2  # Initial norm

# Euler-Cromer method loop
for i in range(1, n_steps):
    # Compute the derivative of the wavefunction
    dpsi_dt = -1j * np.dot(H, psi_0)

    # Update the wavefunction using Euler-Cromer method
    psi_0 += dt * dpsi_dt  # Update both components

    # Store the wavefunction components for plotting
    psi_0_t[i] = psi_0[0]
    psi_1_t[i] = psi_0[1]

    # Calculate and store the norm |psi(t)|^2 = |psi_0|^2 + |psi_1|^2
    psi_norm_t[i] = np.abs(psi_0[0])**2 + np.abs(psi_0[1])**2

# Time array
tlist = np.linspace(0, t_max, n_steps)

# Perform FFT on the norm |psi(t)|^2
fft_result = np.fft.fft(psi_norm_t)
fft_freqs = np.fft.fftfreq(n_steps, dt)

# Shift the FFT and frequencies for proper negative/positive plotting
fft_result_shifted = np.fft.fftshift(fft_result)
fft_freqs_shifted = np.fft.fftshift(fft_freqs)

# Plot the norm of the wavefunction |psi(t)|^2
plt.figure(figsize=(10, 6))
plt.subplot(2, 1, 1)
plt.plot(tlist, psi_norm_t, label="|ψ(t)|²", color='g')
plt.xlabel('Time')
plt.ylabel(r'$|\psi(t)|^2$ (Probability Density)')
plt.title('Time Evolution of the Wavefunction Norm (|ψ(t)|²)')
plt.legend()
plt.grid(True)


# Assuming you already have the fft_result_shifted and fft_freqs_shifted arrays
# Define the range to zoom in around the peak
zoom_range = 10  # Adjust this value if you want to zoom more or less
```

```
# Find the index where the peak occurs
zoom_indices = np.where(np.abs(fft_freqs_shifted) < zoom_range)

# Find the frequency of the peak
peak_index = np.argmax(np.abs(fft_result_shifted[zoom_indices]))
peak_frequency = fft_freqs_shifted[zoom_indices][peak_index]

# Plot the zoomed-in FFT around the peak frequency
plt.figure(figsize=(10, 6))
plt.plot(fft_freqs_shifted[zoom_indices], np.
  ↪abs(fft_result_shifted[zoom_indices]), label="FFT(| (t)|²)", color='b')
plt.axvline(peak_frequency, color='r', linestyle='--', label=f'Peak Frequency:␣
  ↪{peak_frequency:.3f}')
plt.xlabel('Frequency')
plt.ylabel('Amplitude')
plt.title(f'Zoomed FFT of Probability Density | (t)|² around Peak Frequency')
plt.legend()
plt.grid(True)
plt.show()

print(f"The peak frequency is located at: {peak_frequency}")
```





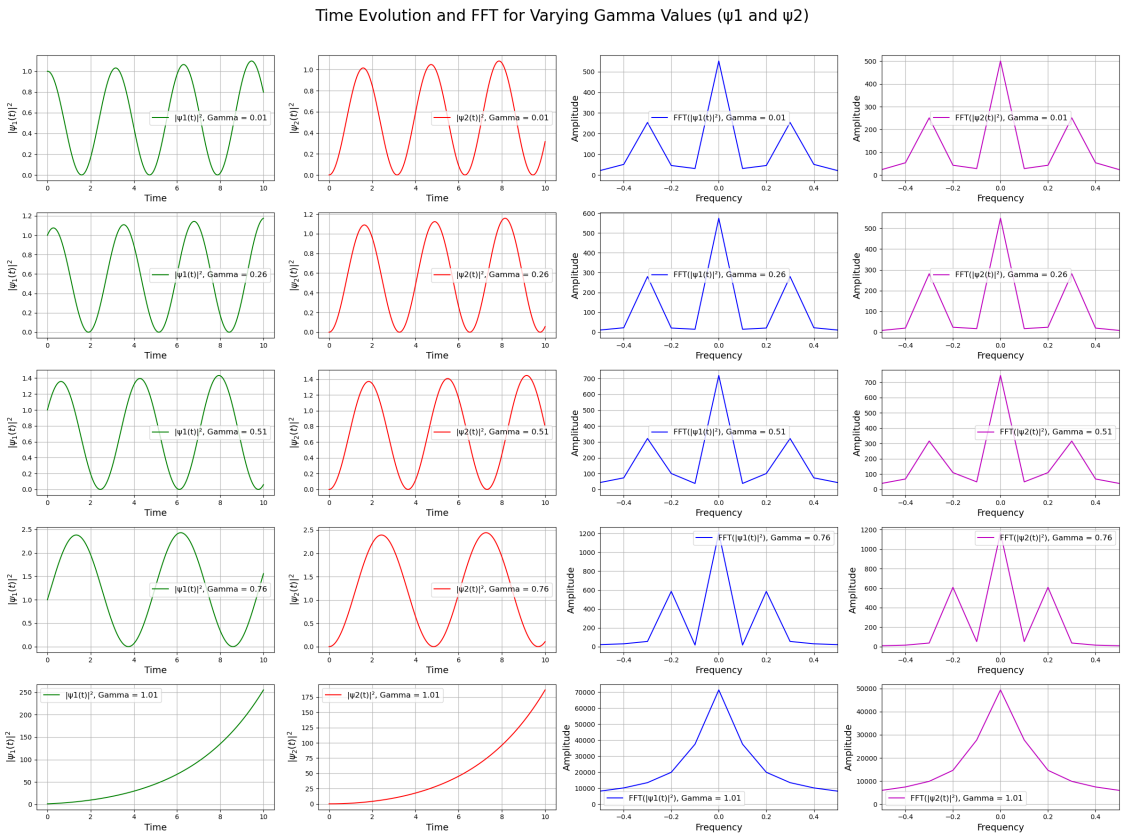
The peak frequency is located at: 0.0

[14]:
```
import numpy as np
import matplotlib.pyplot as plt

# Constants
hbar = 1.0  # Assume hbar = 1 for simplicity
dt = 0.01  # Time step
t_max = 10  # Maximum time
n_steps = int(t_max / dt)  # Number of time steps
gamma_values = np.linspace(0.01, 1.01, 5)  # Gamma values from 0.01 to 1.01

# Set up subplots with enlarged figure size
fig, axs = plt.subplots(len(gamma_values), 4, figsize=(24, 18))  # Increased␣
 ↪figsize for larger plots
fig.suptitle('Time Evolution and FFT for Varying Gamma Values ( 1 and  2)',␣
 ↪fontsize=24)  # Larger title font size

# Loop over gamma values
for idx, gamma in enumerate(gamma_values):

    # Define the initial wavefunction (start in state |0>)
    psi_0 = np.array([1 + 0j, 0 + 0j])  # Initial wavefunction [psi_0, psi_1]
```

```
    # Define the Hamiltonian matrix
    H = np.array([[1j * gamma, -1], [-1, -1j*gamma]])

    # Arrays to store the wavefunction components over time
    psi_0_t = np.zeros(n_steps, dtype=complex)  # For 1(t)
    psi_1_t = np.zeros(n_steps, dtype=complex)  # For 2(t)

    # Set initial condition
    psi_0_t[0] = psi_0[0]
    psi_1_t[0] = psi_0[1]

    # Euler-Cromer method loop
    for i in range(1, n_steps):
        # Compute the derivative of the wavefunction
        dpsi_dt = -1j * np.dot(H, psi_0)

        # Update the wavefunction using Euler-Cromer method
        psi_0 += dt * dpsi_dt  # Update both components

        # Store the wavefunction components for plotting
        psi_0_t[i] = psi_0[0]  # 1(t)
        psi_1_t[i] = psi_0[1]  # 2(t)

    # Time array
    tlist = np.linspace(0, t_max, n_steps)

    # Perform FFT on both 1 and 2
    fft_psi_0 = np.fft.fft(abs(psi_0_t)**2)
    fft_psi_1 = np.fft.fft(abs(psi_1_t)**2)
    fft_freqs = np.fft.fftfreq(n_steps, dt)

    # Shift the FFT and frequencies for proper negative/positive plotting
    fft_psi_0_shifted = np.fft.fftshift(fft_psi_0)
    fft_psi_1_shifted = np.fft.fftshift(fft_psi_1)
    fft_freqs_shifted = np.fft.fftshift(fft_freqs)

    # Find the frequency with the maximum amplitude for 1 and 2
    peak_freq_psi_0 = fft_freqs_shifted[np.argmax(np.abs(fft_psi_0_shifted))]
    peak_freq_psi_1 = fft_freqs_shifted[np.argmax(np.abs(fft_psi_1_shifted))]

    # Print the peak frequencies for both 1 and 2
    print(f"Gamma = {gamma:.2f}: Peak frequency for 1 = {peak_freq_psi_0:.4f},␣
↪Peak frequency for 2 = {peak_freq_psi_1:.4f}")

    # Plot time-domain |1(t)|^2
    axs[idx, 0].plot(tlist, np.abs(psi_0_t)**2, label=f'|1(t)|², Gamma =␣
↪{gamma:.2f}', color='g')
```

```
    axs[idx, 0].set_xlabel('Time', fontsize=14)
    axs[idx, 0].set_ylabel(r'$|\psi_1(t)|^2$', fontsize=14)
    axs[idx, 0].legend(fontsize=12)
    axs[idx, 0].grid(True)

    # Plot time-domain |2(t)|^2
    axs[idx, 1].plot(tlist, np.abs(psi_1_t)**2, label=f'|2(t)|², Gamma =␣
↪{gamma:.2f}', color='r')
    axs[idx, 1].set_xlabel('Time', fontsize=14)
    axs[idx, 1].set_ylabel(r'$|\psi_2(t)|^2$', fontsize=14)
    axs[idx, 1].legend(fontsize=12)
    axs[idx, 1].grid(True)

    # Plot FFT of |1(t)| (with negative and positive frequencies)
    axs[idx, 2].plot(fft_freqs_shifted, np.abs(fft_psi_0_shifted),␣
↪label=f'FFT(|1(t)|²), Gamma = {gamma:.2f}', color='b')
    axs[idx, 2].set_xlabel('Frequency', fontsize=14)
    axs[idx, 2].set_ylabel('Amplitude', fontsize=14)
    axs[idx, 2].legend(fontsize=12)
    axs[idx, 2].grid(True)

    # Zoom in around the peak frequency for 1
    zoom_range_psi_0 = 0.5  # Adjust this value to zoom in as needed
    axs[idx, 2].set_xlim(peak_freq_psi_0 - zoom_range_psi_0, peak_freq_psi_0 +␣
↪zoom_range_psi_0)

    # Plot FFT of |2(t)| (with negative and positive frequencies)
    axs[idx, 3].plot(fft_freqs_shifted, np.abs(fft_psi_1_shifted),␣
↪label=f'FFT(|2(t)|²), Gamma = {gamma:.2f}', color='m')
    axs[idx, 3].set_xlabel('Frequency', fontsize=14)
    axs[idx, 3].set_ylabel('Amplitude', fontsize=14)
    axs[idx, 3].legend(fontsize=12)
    axs[idx, 3].grid(True)

    # Zoom in around the peak frequency for 2
    zoom_range_psi_1 = 0.5  # Adjust this value to zoom in as needed
    axs[idx, 3].set_xlim(peak_freq_psi_1 - zoom_range_psi_1, peak_freq_psi_1 +␣
↪zoom_range_psi_1)

plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```

```
Gamma = 0.01: Peak frequency for  1 = 0.0000, Peak frequency for  2 = 0.0000
Gamma = 0.26: Peak frequency for  1 = 0.0000, Peak frequency for  2 = 0.0000
Gamma = 0.51: Peak frequency for  1 = 0.0000, Peak frequency for  2 = 0.0000
Gamma = 0.76: Peak frequency for  1 = 0.0000, Peak frequency for  2 = 0.0000
Gamma = 1.01: Peak frequency for  1 = 0.0000, Peak frequency for  2 = 0.0000
```



2 Q1. Eigenvalues and Eigenvectors for Different Values of γ:

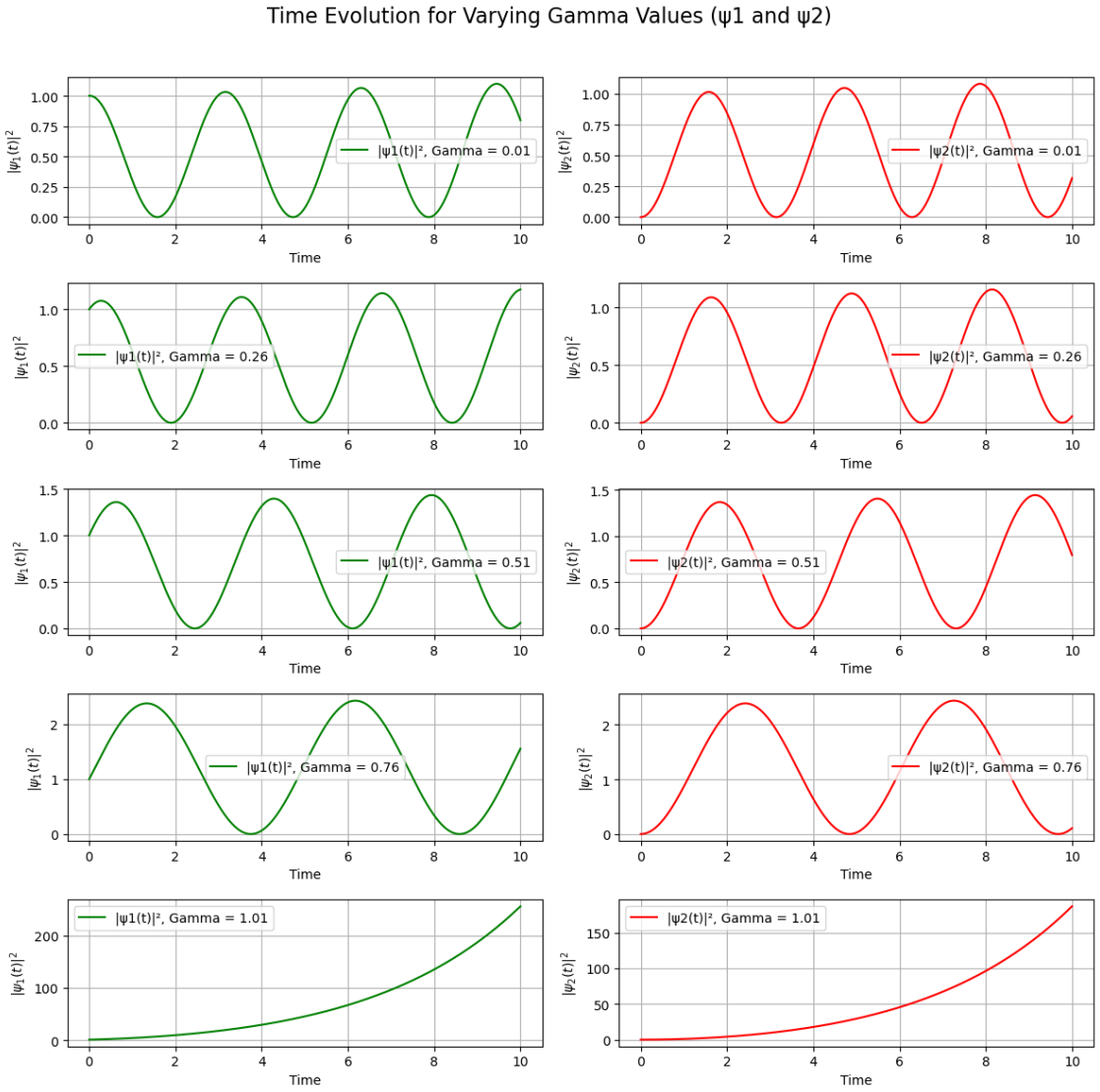
[17]:
```python
import numpy as np
import matplotlib.pyplot as plt

def dot_product_check(H):
    eigenvalues, eigenvectors = np.linalg.eig(H)
    # Dot product
    v1 = eigenvectors[:,0]
    v2 = eigenvectors[:,1]
    dot_product = np.dot(np.conjugate(v1), v2)
    if np.isclose(dot_product, 0):
        result = "Orthogonal"
    else:
        result = "Not orthogonal"
    return f"The eigenvectors are {result}", eigenvalues, eigenvectors

# Constants
dt = 0.01  # Time step
t_max = 10  # Maximum time
```

```
n_steps = int(t_max / dt)  # Number of time steps
gamma_values = np.linspace(0.01, 1.01, 5)  # Gamma values from 0.01 to 1.01

# Set up subplots
fig, axs = plt.subplots(len(gamma_values), 2, figsize=(12, 12))
fig.suptitle('Time Evolution for Varying Gamma Values ( 1 and  2)', fontsize=16)

# Loop over gamma values
for idx, gamma in enumerate(gamma_values):

    # Define the Hamiltonian matrix
    H = np.array([[1j * gamma, -1], [-1, -1j*gamma]])

    # Orthogonality check and get eigenvalues and eigenvectors
    orthogonality_result, eigenvalues, eigenvectors = dot_product_check(H)

    # Print the eigenvalues and eigenvectors for this gamma
    print(f"Gamma = {gamma:.2f}: {orthogonality_result}")
    print(f"Eigenvalues: {eigenvalues}")
    print(f"Eigenvectors:\n{eigenvectors}\n")

    # Define the initial wavefunction (start in state |0>)
    psi_0 = np.array([1 + 0j, 0 + 0j])  # Initial wavefunction [psi_0, psi_1]

    # Arrays to store the wavefunction components over time
    psi_0_t = np.zeros(n_steps, dtype=complex)  # For  1(t)
    psi_1_t = np.zeros(n_steps, dtype=complex)  # For  2(t)

    # Set initial condition
    psi_0_t[0] = psi_0[0]
    psi_1_t[0] = psi_0[1]

    # Euler-Cromer method loop
    for i in range(1, n_steps):
        # Compute the derivative of the wavefunction
        dpsi_dt = -1j * np.dot(H, psi_0)

        # Update the wavefunction using Euler-Cromer method
        psi_0 += dt * dpsi_dt  # Update both components

        # Store the wavefunction components for plotting
        psi_0_t[i] = psi_0[0]  #  1(t)
        psi_1_t[i] = psi_0[1]  #  2(t)

    # Time array
    tlist = np.linspace(0, t_max, n_steps)
```

```
    # Plot time-domain |1(t)|^2
    axs[idx, 0].plot(tlist, np.abs(psi_0_t)**2, label=f'|1(t)|², Gamma =␣
 ↪{gamma:.2f}', color='g')
    axs[idx, 0].set_xlabel('Time')
    axs[idx, 0].set_ylabel(r'$|\psi_1(t)|^2$')
    axs[idx, 0].legend()
    axs[idx, 0].grid(True)

    # Plot time-domain |2(t)|^2
    axs[idx, 1].plot(tlist, np.abs(psi_1_t)**2, label=f'|2(t)|², Gamma =␣
 ↪{gamma:.2f}', color='r')
    axs[idx, 1].set_xlabel('Time')
    axs[idx, 1].set_ylabel(r'$|\psi_2(t)|^2$')
    axs[idx, 1].legend()
    axs[idx, 1].grid(True)

plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```

```
Gamma = 0.01: The eigenvectors are Not orthogonal
Eigenvalues: [ 0.99995-1.73472348e-18j -0.99995+2.16840434e-18j]
Eigenvectors:
[[-0.70707142-0.00707107j  0.70710678+0.j        ]
 [ 0.70710678+0.j          0.70707142+0.00707107j]]

Gamma = 0.26: The eigenvectors are Not orthogonal
Eigenvalues: [ 0.96560862+0.00000000e+00j -0.96560862-3.12250226e-17j]
Eigenvectors:
[[ 0.70710678+0.j          0.6827884 -0.18384776j]
 [-0.6827884 +0.18384776j  0.70710678+0.j        ]]

Gamma = 0.51: The eigenvectors are Not orthogonal
Eigenvalues: [ 0.8601744+2.93160153e-17j -0.8601744-1.81971681e-16j]
Eigenvectors:
[[-0.60823515-0.36062446j  0.60823515-0.36062446j]
 [ 0.70710678+0.j          0.70710678+0.j        ]]

Gamma = 0.76: The eigenvectors are Not orthogonal
Eigenvalues: [ 0.64992307-2.49433278e-16j -0.64992307+2.73886730e-17j]
Eigenvectors:
[[ 0.70710678+0.j          0.70710678+0.j        ]
 [-0.45956501+0.53740115j  0.45956501+0.53740115j]]

Gamma = 1.01: The eigenvectors are Not orthogonal
Eigenvalues: [-8.32667268e-17+0.14177447j -9.92956938e-17-0.14177447j]
Eigenvectors:
[[7.55106205e-01+0.j         5.63724021e-17-0.65560249j]
```

```
[4.38897148e-17+0.65560249j 7.55106205e-01+0.j        ]]
```



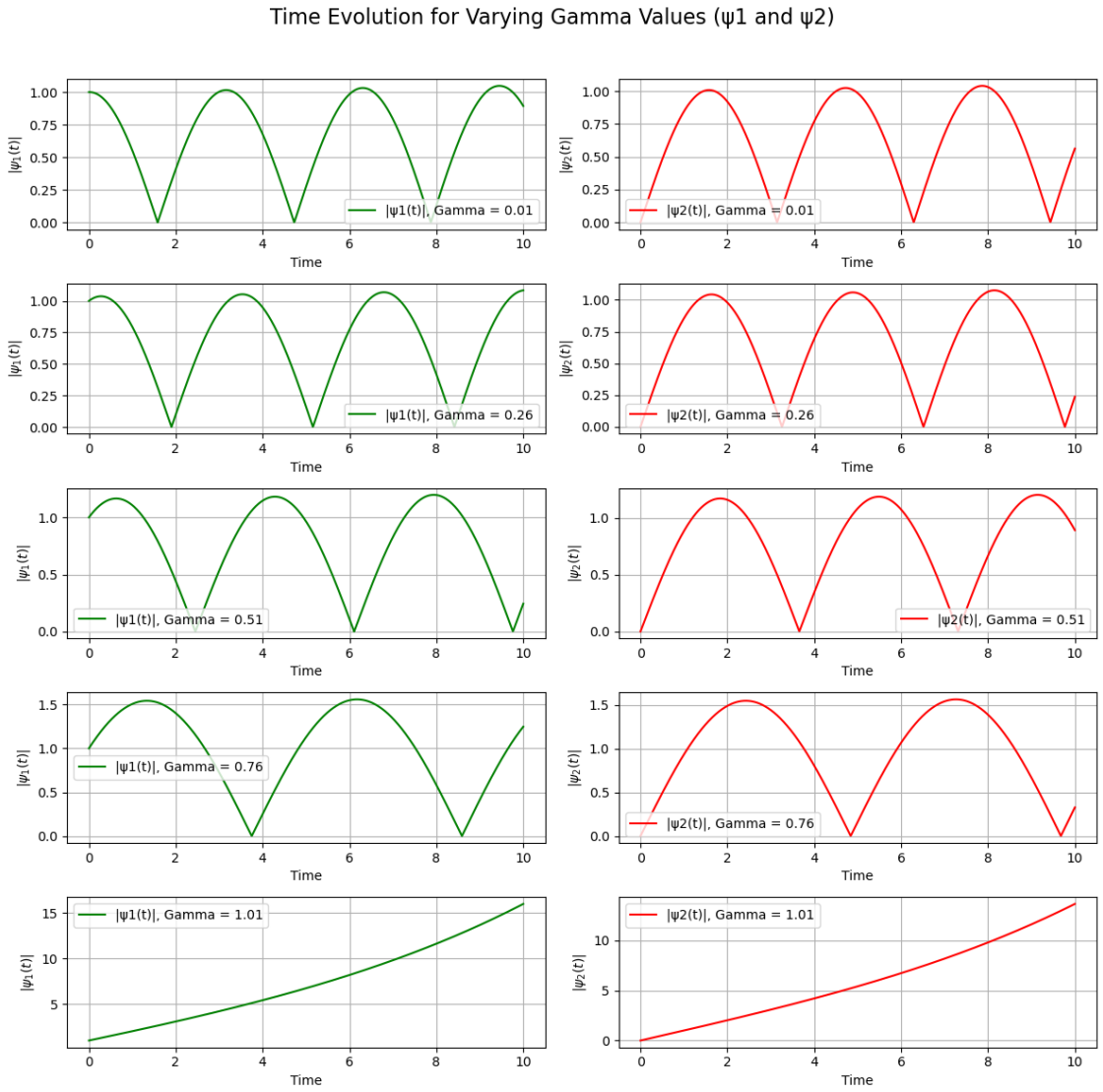
3 Q2. Orthogonality Check of Eigenvectors:

[21]:
```python
import numpy as np
import matplotlib.pyplot as plt

def dot_product_check(H):
    eigenvalues, eigenvectors = np.linalg.eig(H)
    # Dot product
    v1 = eigenvectors[:,0]
```

```
    v2 = eigenvectors[:,1]
    dot_product = np.dot(np.conjugate(v1), v2)
    if np.isclose(dot_product, 0):
        result = "Orthogonal"
    else:
        result = "Not orthogonal"
    return f"The eigenvectors are {result}"

# Constants
dt = 0.01  # Time step
t_max = 10  # Maximum time
n_steps = int(t_max / dt)  # Number of time steps
gamma_values = np.linspace(0.01, 1.01, 5)  # Gamma values from 0.01 to 1.01

# Set up subplots
fig, axs = plt.subplots(len(gamma_values), 2, figsize=(12, 12))
fig.suptitle('Time Evolution for Varying Gamma Values ( 1 and  2)', fontsize=16)

# Loop over gamma values
for idx, gamma in enumerate(gamma_values):

    # Define the Hamiltonian matrix
    H = np.array([[1j * gamma, -1], [-1, -1j*gamma]])

    # Orthogonality check
    orthogonality_result = dot_product_check(H)
    print(f"Gamma = {gamma:.2f}: {orthogonality_result}")

    # Define the initial wavefunction (start in state |0>)
    psi_0 = np.array([1 + 0j, 0 + 0j])  # Initial wavefunction [psi_0, psi_1]

    # Arrays to store the wavefunction components over time
    psi_0_t = np.zeros(n_steps, dtype=complex)  # For  1(t)
    psi_1_t = np.zeros(n_steps, dtype=complex)  # For  2(t)

    # Set initial condition
    psi_0_t[0] = psi_0[0]
    psi_1_t[0] = psi_0[1]

    # Euler-Cromer method loop
    for i in range(1, n_steps):
        # Compute the derivative of the wavefunction
        dpsi_dt = -1j * np.dot(H, psi_0)

        # Update the wavefunction using Euler-Cromer method
        psi_0 += dt * dpsi_dt  # Update both components
```

```
        # Store the wavefunction components for plotting
        psi_0_t[i] = psi_0[0]  # 1(t)
        psi_1_t[i] = psi_0[1]  # 2(t)

    # Time array
    tlist = np.linspace(0, t_max, n_steps)

    # Plot time-domain |1(t)|^2
    axs[idx, 0].plot(tlist, np.abs(psi_0_t), label=f'|1(t)|, Gamma = {gamma:.
↪2f}', color='g')
    axs[idx, 0].set_xlabel('Time')
    axs[idx, 0].set_ylabel(r'$|\psi_1(t)|$')
    axs[idx, 0].legend()
    axs[idx, 0].grid(True)

    # Plot time-domain |2(t)|^2
    axs[idx, 1].plot(tlist, np.abs(psi_1_t), label=f'|2(t)|, Gamma = {gamma:.
↪2f}', color='r')
    axs[idx, 1].set_xlabel('Time')
    axs[idx, 1].set_ylabel(r'$|\psi_2(t)|$')
    axs[idx, 1].legend()
    axs[idx, 1].grid(True)

plt.tight_layout(rect=[0, 0, 1, 0.96])
plt.show()
```

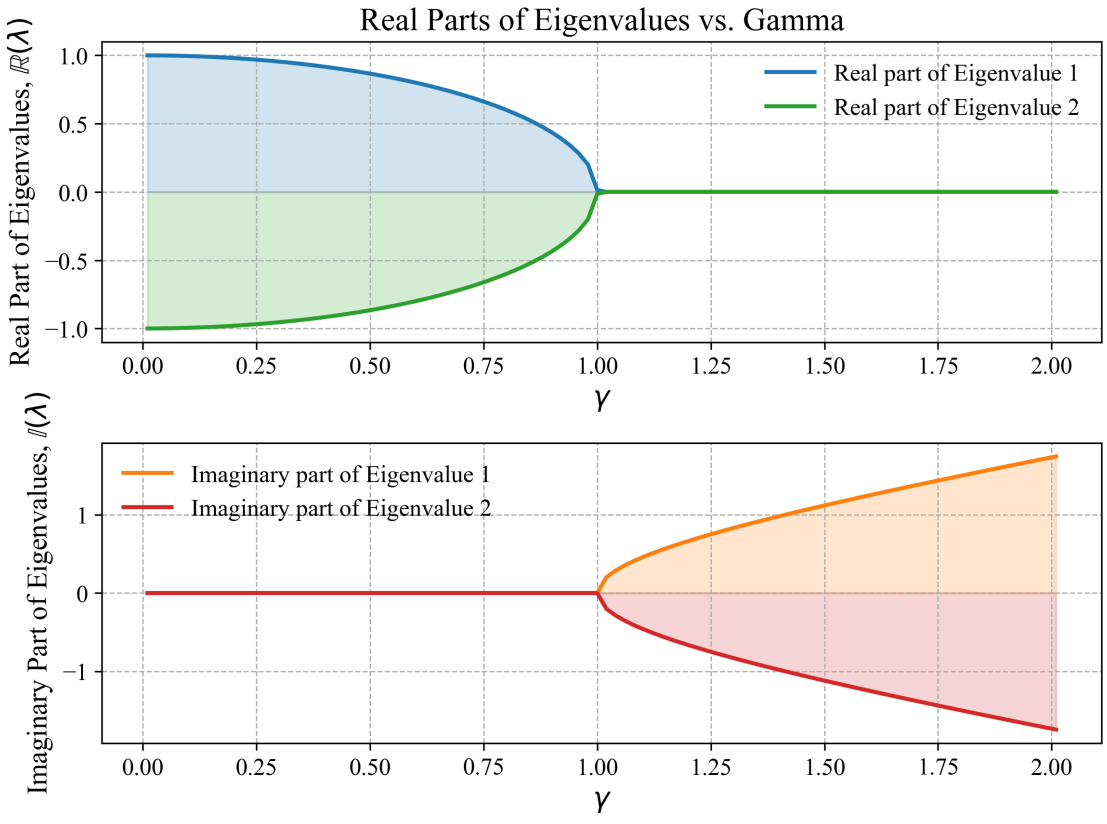
```
Gamma = 0.01: The eigenvectors are Not orthogonal
Gamma = 0.26: The eigenvectors are Not orthogonal
Gamma = 0.51: The eigenvectors are Not orthogonal
Gamma = 0.76: The eigenvectors are Not orthogonal
Gamma = 1.01: The eigenvectors are Not orthogonal
```



[50]:
```
import numpy as np
import matplotlib.pyplot as plt

# Use LaTeX-style fonts and publication-quality settings
plt.rcParams.update({
    "font.family": "serif",
    "font.serif": ["Times New Roman"],
    "font.size": 14,
    "axes.titlesize": 16,
    "axes.labelsize": 14,
    "legend.fontsize": 12,
    "xtick.labelsize": 12,
    "ytick.labelsize": 12,
```

```
    "figure.dpi": 300,  # High DPI for publication
    "lines.linewidth": 2,
    "text.usetex": False,  # Can be switched to True if LaTeX is available
})

# Constants
gamma_values = np.linspace(0.01, 2.01, 100)  # Gamma values from 0.01 to 2.01

# Arrays to store eigenvalues
real_parts = np.zeros((len(gamma_values), 2))  # Real parts of the eigenvalues
imaginary_parts = np.zeros((len(gamma_values), 2))  # Imaginary parts of the␣
 ↪eigenvalues

# Loop over gamma values and calculate eigenvalues
for idx, gamma in enumerate(gamma_values):
    # Define the Hamiltonian matrix for given gamma
    H = np.array([[1j * gamma, -1], [-1, -1j * gamma]])

    # Compute the eigenvalues of the Hamiltonian
    eigvals = np.linalg.eigvals(H)

    # Store the real and imaginary parts of the eigenvalues
    real_parts[idx] = np.real(eigvals)
    imaginary_parts[idx] = np.imag(eigvals)

# Plot the real and imaginary parts of the eigenvalues
fig, ax = plt.subplots(2, 1, figsize=(8, 6))

# Define colors that are visually distinct, colorblind-friendly, and␣
 ↪print-friendly
real_colors = ['#1f77b4', '#2ca02c']  # Blue and Green for real parts
imaginary_colors = ['#ff7f0e', '#d62728']  # Orange and Red for imaginary parts
shade_alpha = 0.2  # Transparency for shading

# Plot real parts of eigenvalues with shading
ax[0].plot(gamma_values, real_parts[:, 0], label='Real part of Eigenvalue 1',␣
 ↪color=real_colors[0], linewidth=2)
ax[0].plot(gamma_values, real_parts[:, 1], label='Real part of Eigenvalue 2',␣
 ↪color=real_colors[1], linewidth=2)
ax[0].fill_between(gamma_values, real_parts[:, 0], color=real_colors[0],␣
 ↪alpha=shade_alpha)
ax[0].fill_between(gamma_values, real_parts[:, 1], color=real_colors[1],␣
 ↪alpha=shade_alpha)
ax[0].set_title('Real Parts of Eigenvalues vs. Gamma', fontsize=16)
ax[0].set_xlabel('$\gamma$', fontsize=14)
ax[0].set_ylabel(r'Real Part of Eigenvalues, $\mathbb{R}(\lambda)$')
```

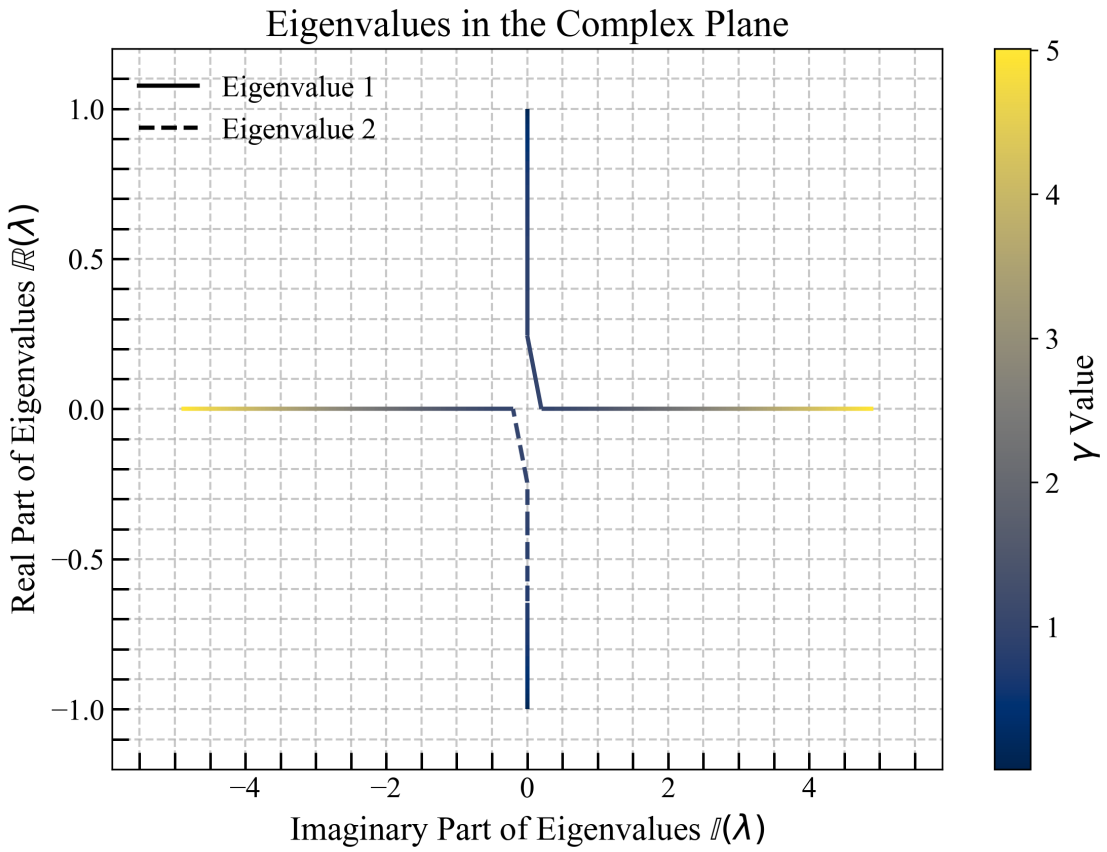
```
ax[0].legend(frameon=False)
ax[0].grid(True, linestyle='--', linewidth=0.7)

# Plot imaginary parts of eigenvalues with shading
ax[1].plot(gamma_values, imaginary_parts[:, 0], label='Imaginary part of␣
 ↪Eigenvalue 1', color=imaginary_colors[0], linewidth=2)
ax[1].plot(gamma_values, imaginary_parts[:, 1], label='Imaginary part of␣
 ↪Eigenvalue 2', color=imaginary_colors[1], linewidth=2)
ax[1].fill_between(gamma_values, imaginary_parts[:, 0],␣
 ↪color=imaginary_colors[0], alpha=shade_alpha)
ax[1].fill_between(gamma_values, imaginary_parts[:, 1],␣
 ↪color=imaginary_colors[1], alpha=shade_alpha)
ax[1].set_ylabel(r'Imaginary Part of Eigenvalues, $\mathbb{I}(\lambda)$')
ax[1].set_xlabel('$\gamma$', fontsize=14)
ax[1].set_ylabel(r'Imaginary Part of Eigenvalues, $\mathbb{I}(\lambda)$')
ax[1].legend(frameon=False)
ax[1].grid(True, linestyle='--', linewidth=0.7)

# Adjust layout for better spacing
plt.tight_layout()

# Show the plot
plt.show()
```



[49]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection
from matplotlib import cm

# Use LaTeX-style fonts for scientific publications
plt.rcParams.update({
    "font.family": "serif",
    "font.serif": ["Times New Roman"],
    "font.size": 16,
    "axes.titlesize": 18,
    "axes.labelsize": 16,
    "legend.fontsize": 14,
    "xtick.labelsize": 14,
    "ytick.labelsize": 14,
    "figure.dpi": 300,  # High DPI for publication
    "text.usetex": False,  # You can switch to True if LaTeX is available
})

# Constants
gamma_values = np.linspace(0.01, 5.01, 100)  # Gamma values from 0.01 to 5.01

# Arrays to store eigenvalues
real_parts = np.zeros((len(gamma_values), 2))  # To store the real parts of the␣
 ↪eigenvalues
imaginary_parts = np.zeros((len(gamma_values), 2))  # To store the imaginary␣
 ↪parts of the eigenvalues

# Loop over gamma values and calculate eigenvalues
for idx, gamma in enumerate(gamma_values):

    # Define the Hamiltonian matrix for given gamma
    H = np.array([[1j * gamma, -1], [-1, -1j * gamma]])

    # Compute the eigenvalues of the Hamiltonian
    eigvals = np.linalg.eigvals(H)

    # Store the real and imaginary parts of the eigenvalues
    real_parts[idx] = np.real(eigvals)
    imaginary_parts[idx] = np.imag(eigvals)

# Helper function to create fading line plots
def plot_fading_lines(x, y, gamma_values, ax, label, line_style):
    points = np.array([x, y]).T.reshape(-1, 1, 2)
```

```
    segments = np.concatenate([points[:-1], points[1:]], axis=1)

    # Create a colormap that varies with gamma values
    norm = plt.Normalize(gamma_values.min(), gamma_values.max())
    lc = LineCollection(segments, cmap='cividis', norm=norm)  # Use a␣
 ↪colorblind-friendly colormap
    lc.set_array(gamma_values)
    lc.set_linewidth(2.0)  # Adjust line width for publication quality
    lc.set_linestyle(line_style)  # Set the line style

    # Add the fading lines to the plot
    ax.add_collection(lc)

# Create the figure and axis
fig, ax = plt.subplots(figsize=(8, 6))  # Larger figure for better clarity

# Plot the eigenvalues with a fading effect for both branches
plot_fading_lines(imaginary_parts[:, 0], real_parts[:, 0], gamma_values, ax,␣
 ↪'Eigenvalue 1', '-')
plot_fading_lines(imaginary_parts[:, 1], real_parts[:, 1], gamma_values, ax,␣
 ↪'Eigenvalue 2', '--')

# Add colorbar to show gamma values
cbar = plt.colorbar(cm.ScalarMappable(cmap='cividis', norm=plt.
 ↪Normalize(gamma_values.min(), gamma_values.max())), ax=ax)
cbar.set_label(r'$\gamma$ Value', fontsize=16)  # Use math text for Greek␣
 ↪symbols
cbar.ax.tick_params(labelsize=14)  # Increase font size of colorbar ticks

# Add labels and title
ax.set_title('Eigenvalues in the Complex Plane', fontsize=18)
ax.set_xlabel(r'Imaginary Part of Eigenvalues $\mathbb{I}(\lambda)$',␣
 ↪fontsize=16)
ax.set_ylabel(r'Real Part of Eigenvalues $\mathbb{R}(\lambda)$', fontsize=16)

# Set axis limits with a buffer
x_buffer = (np.max(imaginary_parts) - np.min(imaginary_parts)) * 0.1
y_buffer = (np.max(real_parts) - np.min(real_parts)) * 0.1
ax.set_xlim(np.min(imaginary_parts) - x_buffer, np.max(imaginary_parts) +␣
 ↪x_buffer)
ax.set_ylim(np.min(real_parts) - y_buffer, np.max(real_parts) + y_buffer)

# Add a fine grid and light ticks
ax.grid(True, which='both', linestyle='--', linewidth=1.0, alpha=0.7)
ax.minorticks_on()  # Minor ticks for precision
```

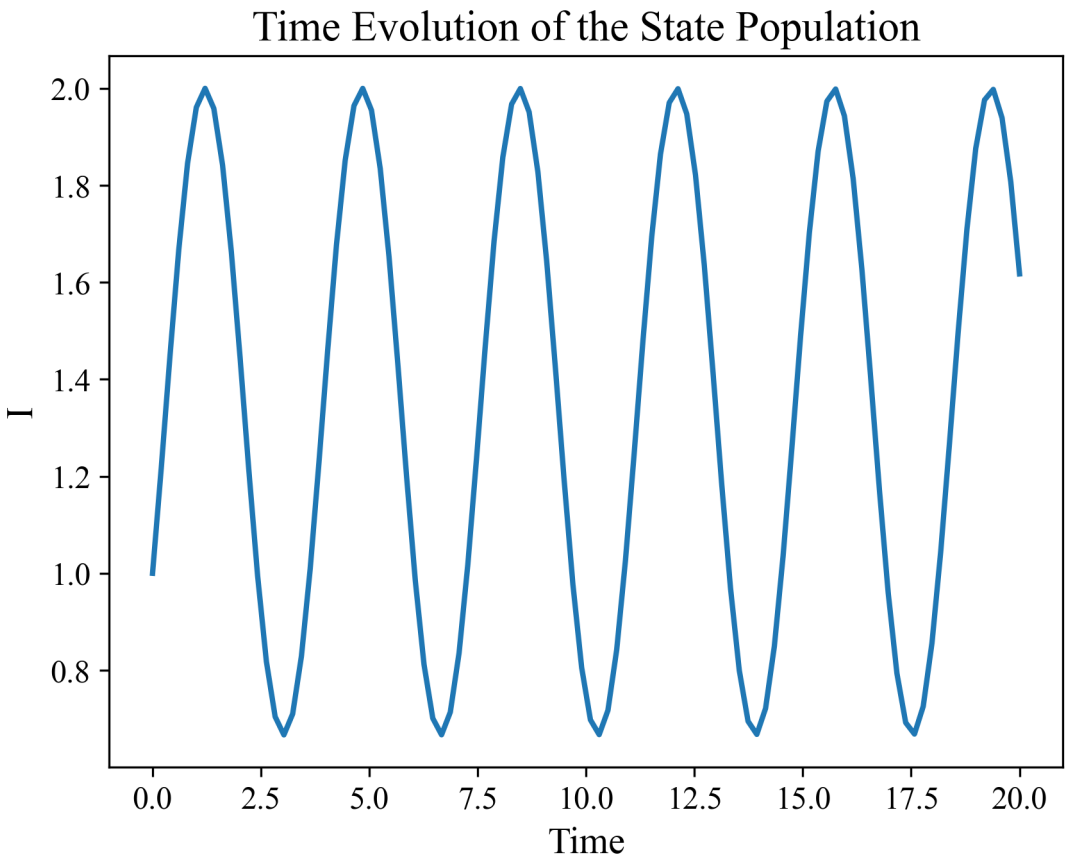
```
ax.tick_params(axis='both', which='both', direction='in', length=8, width=1.2,␣
 ↪colors='black')

# Add legend without markers
lines = [plt.Line2D([0], [0], color='k', linestyle='-', linewidth=2.0),
         plt.Line2D([0], [0], color='k', linestyle='--', linewidth=2.0)]
labels = ['Eigenvalue 1', 'Eigenvalue 2']
ax.legend(lines, labels, loc='upper left', fontsize=14, frameon=False)

# Show the plot
plt.tight_layout()
plt.show()
```



4 Using Evolution Operator:

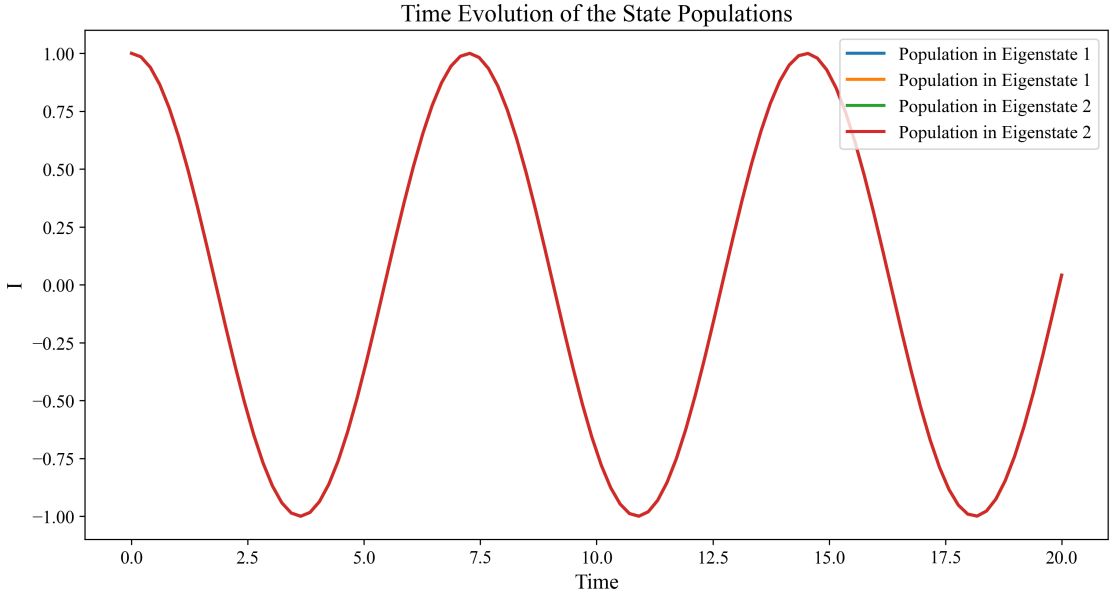
[56]:
```
import numpy as np
import scipy.linalg
import matplotlib.pyplot as plt
from qutip import *
```

```
# Parameters
N = 100
psi0 = basis(2, 0)  # Initial state
t = np.linspace(0, 20, N, endpoint=True)  # Time steps
gamma = 0.5

# Hamiltonian as defined earlier
sigmaaa = tensor(basis(2, 0) * basis(2, 0).dag())
sigmabb = tensor(basis(2, 1) * basis(2, 1).dag())
sigmaab = tensor(basis(2, 0) * basis(2, 1).dag())
sigmaba = sigmaab.dag()

H = 1j * gamma * sigmaaa - 1j *gamma* sigmabb - sigmaab - sigmaba  # Hamiltonian

# Initialize arrays to store results
I = np.zeros(len(t))

# Time evolution loop
for i in range(len(t)):
    U = scipy.linalg.expm(-1j * H.full() * t[i])  # Time evolution operator
    psi_t = Qobj(U @ psi0.full())  # Apply evolution to the initial state
    I[i] = np.real((psi_t.dag() * psi_t).full())  # Compute the population of␣
 ↪the state

# Plotting the results
plt.plot(t, I)
plt.xlabel('Time')
plt.ylabel('I')
plt.title('Time Evolution of the State Population')
plt.show()
```



[63]:
```python
import numpy as np
import scipy.linalg
import matplotlib.pyplot as plt
from qutip import *

# Parameters
N = 100
t = np.linspace(0, 20, N, endpoint=True)  # Time steps
gamma = 0.5

# Basis states
basis_0 = basis(2, 0)
basis_1 = basis(2, 1)

# Hamiltonian
sigmaaa = tensor(basis_0 * basis_0.dag())
sigmabb = tensor(basis_1 * basis_1.dag())
sigmaab = tensor(basis_0 * basis_1.dag())
sigmaba = sigmaab.dag()
```

```
H = 1j * gamma * sigmaaa - 1j * gamma * sigmabb - sigmaab - sigmaba  # Hamiltonian

# Compute eigenstates and eigenvalues
eigenvalues, eigenstates = H.eigenstates()

# Initialize arrays to store results
I = np.zeros((len(t), len(eigenstates)))

# Time evolution loop
for i in range(len(t)):
    U = scipy.linalg.expm(-1j * H.full() * t[i])  # Time evolution operator

    for j, eigenstate in enumerate(eigenstates):
        psi_t = Qobj(U @ eigenstate.full())  # Apply evolution to the eigenstate
        I[i, j] = np.real((psi_t.dag() * eigenstate).full())  # Compute the population in the eigenstate

# Plotting the results
plt.figure(figsize=(12, 6))
for j in range(len(eigenstates)):
    plt.plot(t, I[:, j], label=f'Population in Eigenstate {j+1}')
plt.xlabel('Time')
plt.ylabel('I')
plt.title('Time Evolution of the State Populations')
plt.legend()
plt.show()
```



[ ]: