Question 1: Fitting natural cubic splines $S_j(x) = a_j + b_j(x - x_j) + c_j(x - x_j)^2 + d_j(x - x_j$

.

(a) Simulate data from $y = cos(x)$ for x in the interval $[-\pi/2, \pi/2]$. Use equally spaced x values.

Lets generate a set of equally spaced x values in the interval $[-\pi/2, \pi/2]$. Then compute the corresponding y values by evaluating $y = \cos(x)$ for each x. To simulate the data, we will generate equally spaced x values in the interval $[-\pi/2, \pi/2]$ and compute the corresponding y values using the cosine function.

In [65]:

```
import numpy as np
import scipy.linalg as linalg
from scipy.interpolate import CubicSpline
import matplotlib.pyplot as plt
import seaborn as sns

# Define the interval and number of data points
x_start, x_end, num_points = -np.pi/2, np.pi/2, 10 # Number of data points


x_start, x_end, num_points = -np.pi/2, np.pi/2, 10
x = np.linspace(x_start, x_end, num_points)  # Generate equally spaced x values
y = np.cos(x)  # Compute corresponding y values using the cosine function
```

(b) Set up the tri-diagonal coefficient matrix A for natural cubic splines.

To determine the number of data points, n. Lets create an $n \times n$ matrix A with all elements initially set to 0. After, lets set the diagonal elements of A to 2. then the sub-diagonal and super-diagonal elements of A to 1.

In [66]:

```
n = len(x)
A = np.zeros((n, n))  # Initialize coefficient matrix A
A += np.diag([2] * n)  # Diagonal elements to 2
A += np.diag([1] * (n - 1), k=-1)  # Sub-diagonal elements to 1
A += np.diag([1] * (n - 1), k=1)  # Super-diagonal elements to 1
```

(c) Set up the y vector and solve the system $Ac = y$ to obtain the c's. Use `scipy linalg` to solve the system.

Creating a vector y of length n, where each element is the corresponding y value from step 1 and utilizing the `scipy linalg.solve` function give us the ability to solve the system of equations $Ac = y$ and obtain the vector c.

In [67]:
```
c = linalg.solve(A, y)  # Solve the system Ac = y to obtain the c's
```

(d) Solve for the other coefficients: a's, b's, and d's.

Creating vectors a, b, and d of length n, initially set to 0 and computing the a, b, and d values for each spline segment using the formulas:

$$a_j = y_j \tag{1}$$

$$b_j = \frac{y_{j+1} - y_j}{x_{j+1} - x_j} - \frac{c_j(x_{j+1} - x_j)^2}{3} - \frac{d_j(x_{j+1} - x_j)^3}{6} \tag{2}$$

$$d_j = \frac{c_{j+1} - c_j}{3(x_{j+1} - x_j)} \tag{3}$$

We solve the system of equations to obtain the c values.

In [68]:
```
a, b, d = y.copy(), np.zeros(n), np.zeros(n)
for j in range(n-1):
    h = x[j+1] - x[j]
    b[j] = (y[j+1] - y[j]) / h - (c[j+1] + 2*c[j]) * h / 3  # Calculate b coefficients
    d[j] = (c[j+1] - c[j]) / (3 * h)  # Calculate d coefficients
# Print the coefficents
for j in range(n):
    print(f"Spline {j+1}: a = {a[j]}, b = {b[j]}, c = {c[j]}, d = {d[j]}")
```

```
Spline 1: a = 6.123233995736766e-17, b = 0.9798155360510165, c = -0.07213376483528113, d
= 0.20664801427253995
Spline 2: a = 0.3420201433256688, b = 0.8134463264833037, c = 0.1442675296705623, d = -
0.017808178437227783
Spline 3: a = 0.6427876096865394, b = 0.5815237262571096, c = 0.12561884881982532, d =
0.11618011655726299
Spline 4: a = 0.8660254037844386, b = 0.25413614998680517, c = 0.24728238237632638, d =
-0.001375664183629711
Spline 5: a = 0.984807753012208, b = -0.0858149735639176, c = 0.24584179021196048, d =
1.0601848938211721e-16
Spline 6: a = 0.984807753012208, b = -0.4262689586435726, c = 0.245841790211960б, d = 0.
001375664183629553
Spline 7: a = 0.8660254037844387, b = -0.7116908116393781, c = 0.24728238237632633, d =
-0.11618011655726272
Spline 8: a = 0.6427876096865395, b = -0.9076544447021211, c = 0.1256188488198255, d =
0.01780817843722751
Spline 9: a = 0.3420201433256688, b = -1.0049949700157157, c = 0.1442675296705622, d = -
0.2066480142725398
Spline 10: a = 6.123233995736766e-17, b = 0.0, c = -0.07213376483528107, d = 0.0
```

(e) Compare the fitted coefficients with values from the scipy builtin cubic splines function for the first three splines.

Using the scipy.interpolate.CubicSpline function to fit a natural cubic spline to the simulated data, we extract the coefficients of the first three splines from the CubicSpline object. Lets compare the coefficients obtained from our spline with the coefficients from the CubicSpline object.

In [69]:

```
spline = CubicSpline(x, y)  # Built-in cubic spline function
coefficients = spline.c[:, :3]  # Extract coefficients for the first three splines

# Print the coefficients from the built-in function.
for j, coeff in enumerate(coefficients.T):
    print(f"Spline {j+1}: a = {coeff[3]}, b = {coeff[2]}, c = {coeff[1]}, d = {coeff[0]}

# 6. Plot the cos(x) curve and interpolate the values using a finer grid and your own sp
```

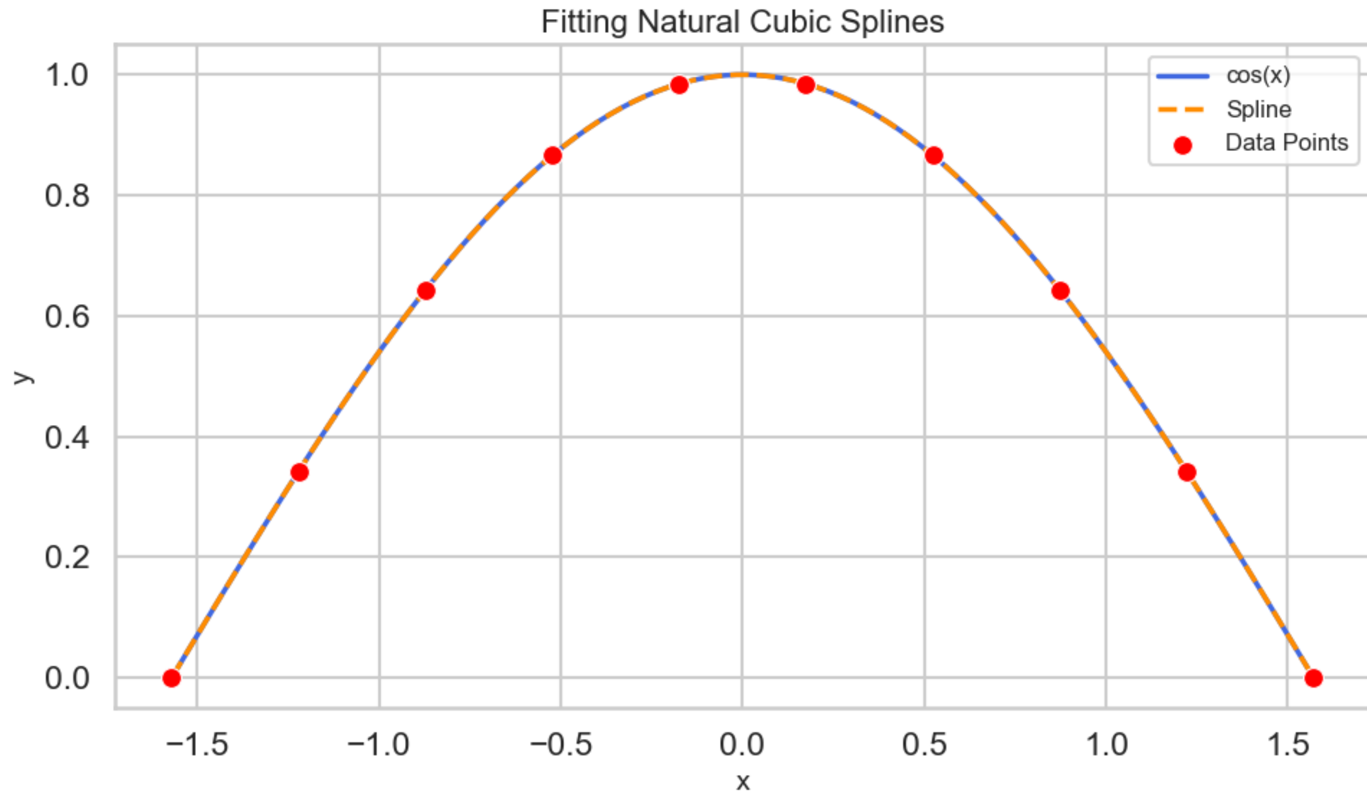
```
Spline 1: a = 6.123233995736766e-17, b = 1.0025285734376974, c = -0.012961744979086351, d = -0.14927359428129927
Spline 2: a = 0.3420201433256688, b = 0.9389139638114324, c = -0.16928068736877716, d = -0.149273594281303
Spline 3: a = 0.6427876096865394, b = 0.7661681450677489, c = -0.3255996297584721, d = -0.10655383681469502
```

(f) Plot the $cos(x)$ curve and the interpolate the values using a finner grid and your own spline code.

Generating a finer grid of x values within the interval $[-\pi/2, \pi/2]$ and then evaluating the $\cos(x)$ function for each x in the finer grid; we use our spline to interpolate the values of y for the finer grid of x values. Lets plot the $\cos(x)$ curve and the interpolated values below.

In [70]:

```
x_finer = np.linspace(x_start, x_end, 100)  # Finer grid of x values
y_cosine, y_spline = np.cos(x_finer), spline(x_finer)  # Evaluate the cosine and spline

import seaborn as sns

# Set Seaborn style
sns.set_style("whitegrid")
sns.set_context("talk")  # Set context to "talk" for better clarity in larger plots

# Create the plot
plt.figure(figsize=(10, 6))

# Plot original cosine curve with a smooth line style
sns.lineplot(x=x_finer, y=y_cosine, label='cos(x)', color='royalblue', linewidth=2.5)

# Plot spline curve
sns.lineplot(x=x_finer, y=y_spline, label='Spline', color='darkorange', linestyle='--',
# Highlight original data points
sns.scatterplot(x=x, y=y, color='red', label='Data Points', s=100, edgecolors='black', z


# Enhance the legend and other visual elements
plt.legend(fontsize=12)
plt.xlabel('x', fontsize=14)
plt.ylabel('y', fontsize=14)
plt.title('Fitting Natural Cubic Splines', fontsize=16)

# Display the enhanced plot
plt.tight_layout()
plt.show()
```

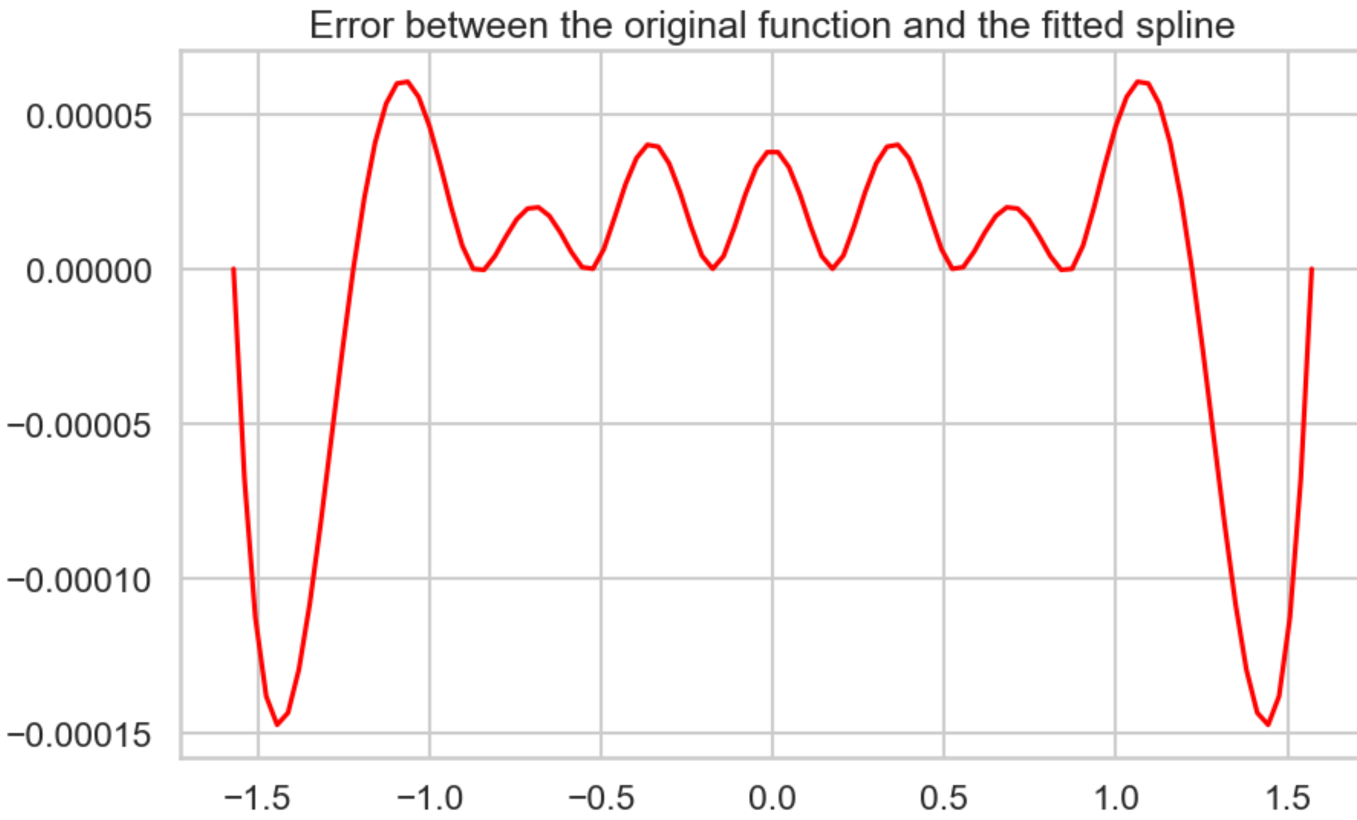


Error Analysis

In [71]:

```
# Calculate the error between the original function and the fitted spline
error = y_cosine - y_spline

# Calculate summary statistics
mean_error = np.mean(error)
std_error = np.std(error)

print(f"Mean error: {mean_error}")
print(f"Standard deviation of error: {std_error}")

# Plot the error
plt.figure(figsize=(10, 6))
sns.lineplot(x=x_finer, y=error, color='red')
plt.title('Error between the original function and the fitted spline')
plt.show()
```

```
Mean error: -3.0149675833190805e-06
Standard deviation of error: 5.4521884780587367e-05
```



Misc

In [72]:
```
# Function to fit cubic spline and other interpolation methods
def fit_interpolations(x, y):
    # Cubic spline
    spline = CubicSpline(x, y)

    # Linear interpolation
    linear_interp = interp1d(x, y)

    # Polynomial interpolation
    poly_interp = interp1d(x, y, kind='quadratic')

    return spline, linear_interp, poly_interp
```

In [73]:
```
# Function to fit cubic spline and other interpolation methods
def fit_interpolations(x, y):
    # Cubic spline
    spline = CubicSpline(x, y)

    # Linear interpolation
    linear_interp = interp1d(x, y)

    # Polynomial interpolation
    poly_interp = interp1d(x, y, kind='quadratic')

    return spline, linear_interp, poly_interp
```

In [ ]:

In [74]:
```
# Function to plot results
def plot_results(x, y, x_finer, y_cosine, y_spline, y_linear, y_poly):
    plt.figure(figsize=(10, 6))
    sns.lineplot(x=x_finer, y=y_cosine, label='cos(x)', color='royalblue')
    sns.lineplot(x=x_finer, y=y_spline, label='Cubic Spline', color='darkorange')
    sns.lineplot(x=x_finer, y=y_linear, label='Linear Interpolation', color='green')
    sns.lineplot(x=x_finer, y=y_poly, label='Polynomial Interpolation', color='purple')
    plt.title('Comparison of Different Interpolation Methods')
    plt.legend()
    plt.show()
```

In [75]:
```
# Function to calculate and plot error
def plot_error(x_finer, y_cosine, y_spline, y_linear, y_poly):
    error_spline = y_cosine - y_spline
    error_linear = y_cosine - y_linear
    error_poly = y_cosine - y_poly

    plt.figure(figsize=(10, 6))
    sns.lineplot(x=x_finer, y=error_spline, label='Error Cubic Spline', color='darkorang
    sns.lineplot(x=x_finer, y=error_linear, label='Error Linear Interpolation', color='g
    sns.lineplot(x=x_finer, y=error_poly, label='Error Polynomial Interpolation', color=
    plt.title('Error of Different Interpolation Methods')
    plt.legend()
    plt.show()
```

In [77]:
```
# Interactive function
def interactive_function(num_points):
    x, y = generate_data(num_points=num_points)
    spline, linear_interp, poly_interp = fit_interpolations(x, y)

    x_finer = np.linspace(-np.pi/2, np.pi/2, 100)
    y_cosine = np.cos(x_finer)
    y_spline = spline(x_finer)
    y_linear = linear_interp(x_finer)
    y_poly = poly_interp(x_finer)

    plot_results(x, y, x_finer, y_cosine, y_spline, y_linear, y_poly)
    plot_error(x_finer, y_cosine, y_spline, y_linear, y_poly)

# Use interactive widget
interact(interactive_function, num_points=IntSlider(min=10, max=100, step=10, value=10))
```

```
interactive(children=(IntSlider(value=10, description='num_points', min=10, step=10), Ou
tput()), _dom_classes=…
```

Out[77]:
```
<function __main__.interactive_function(num_points)>
```

In [ ]:

In [ ]: