lab2

November 21, 2024

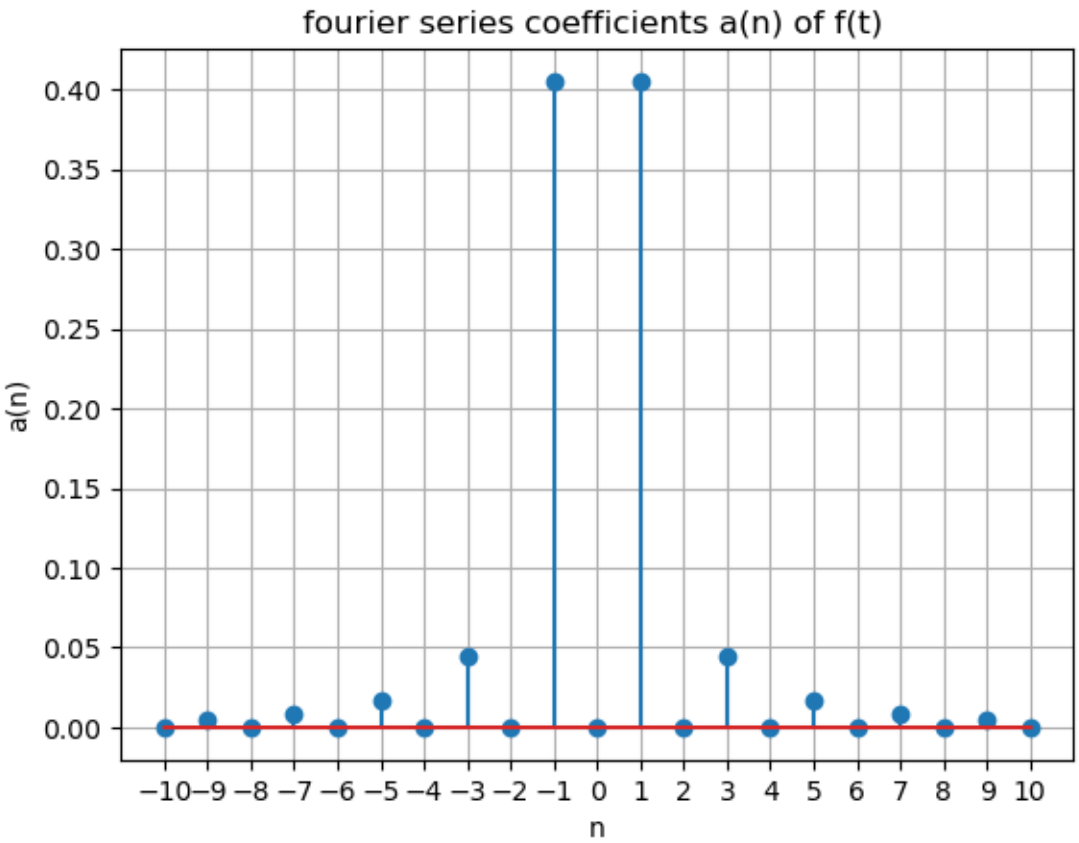
Andy Debrincat, 0135705L, CCE2203, 2024

Q 1.1

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
def a(n) :
    if n%2 == 0 :
        return 0
    return 4/((math.pi*n)**2)
```

Q 1.2

[3]:
```
n = range(-10, 11)
arr = [a(i) for i in n]
plt.stem(n, arr)
plt.title("fourier series coefficients a(n) of f(t) ")
plt.xlabel("n")
plt.ylabel("a(n)")
plt.xticks(range(-10, 11))
plt.grid(True)
plt.show()
```



Q 1.3

$$\hat{f}(t, \omega_0, n_{\max}) = \sum_{n=-n_{\max}}^{n_{\max}} a_n e^{jn\omega_0 t}.$$

$$\hat{f} = \sum_{n=-n_{\max}}^{-1} a_n e^{jn\omega_0 t} + \sum_{n=1}^{n_{\max}} a_n e^{jn\omega_0 t} + a_0.$$

$$\hat{f} = \sum_{n=1}^{n_{\max}} a_{-n} e^{-jn\omega_0 t} + \sum_{n=1}^{n_{\max}} a_n e^{jn\omega_0 t} + 0.$$

For any integer n, if n is even or odd then -n will be even or odd respectively, hence :

$$a_{-n} = a_n \quad \forall n.$$

$$\hat{f} = \sum_{n=1}^{n_{\max}} a_n \left(e^{-jn\omega_0 t} + e^{jn\omega_0 t}\right).$$

using the fact that :

$$e^{-jn\omega_0 t} + e^{jn\omega_0 t} = 2\cos(n\omega_0 t),$$

we can substitute into the summation.

$$\hat{f} = \sum_{n=1}^{n_{\max}} 2a_n \cos(n\omega_0 t).$$

Q 1.4

[4]:
```
def fhat(t, omega0, nmax) :
    fhat = np.zeros_like(t)
    for i in range(1, nmax+1) :
        fhat += 2*a(i)*np.cos(i*omega0*t)
    return fhat
```

Q 1.5

Also, for (T = 4), we have:

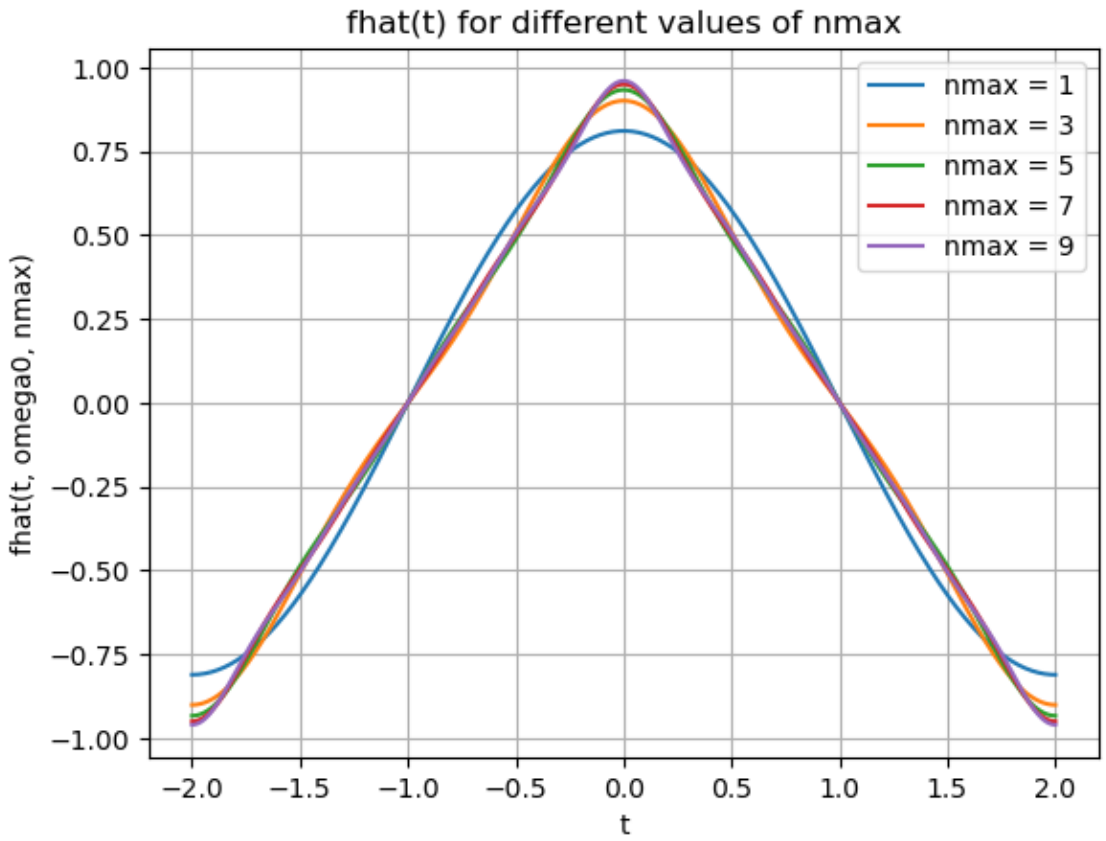
$$\omega_0 = \frac{2\pi}{T} = \frac{2\pi}{4} = 0.5\pi$$

[5]:
```
t = np.arange(-2, 2.01, 0.01)

for i in range(0,5) :

    y = fhat(t,0.5*math.pi, 2*i +1)
    plt.plot(t, y, label=f'nmax = {2*i +1}')

plt.xlabel('t')
plt.ylabel('fhat(t, omega0, nmax)')
plt.title('fhat(t) for different values of nmax')
plt.legend()
plt.grid(True)
plt.show()
```



Q 2.1

Note that in order to make the function work for any array t, the modulo 4 function is used, in order to mimic a periodic function. However %4 maps all values to [0,4). While the function f(t) is defined on the interval [−2,2], not [0,4). The values mapped to the range [0,2] will remain unaffected and may take the value of the function in that range, which is 1−x.

On the other hand, the values mapped to the range [2,4] should actually be in the range [−2,0]. To handle this, the value must first be shifted by -4, which puts us in the desired range. Then x may take the value of the function in [-2,0], which is 1+x. This gives us the expression 1+(−4) = x−3, which corresponds to the value seen in the else statement.

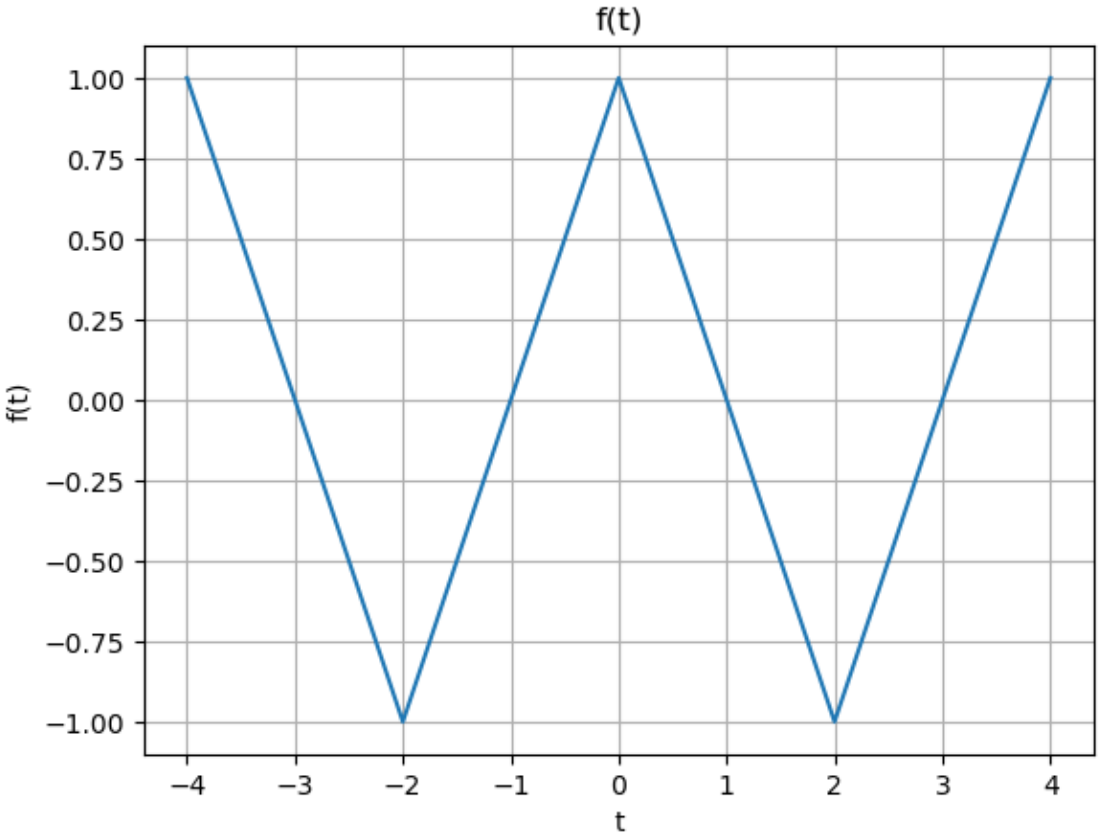
[6]:
```
def f(t) :
    f = np.zeros_like(t)
    for idx, i in enumerate(t):
        x = i % 4
        if 0 <= x < 2:
            f[idx] = 1 - x
        else:
            f[idx] = x - 3
    return f
```

Quick plot to check the validity of f(t)

[7]:
```
t = np.arange(-4, 4.01, 0.01)

plt.plot(t, f(t))

plt.xlabel('t')
plt.ylabel('f(t)')
plt.title('f(t)')
plt.grid(True)
plt.show()
```



Q 2.2

The first error calculated was using the formula:

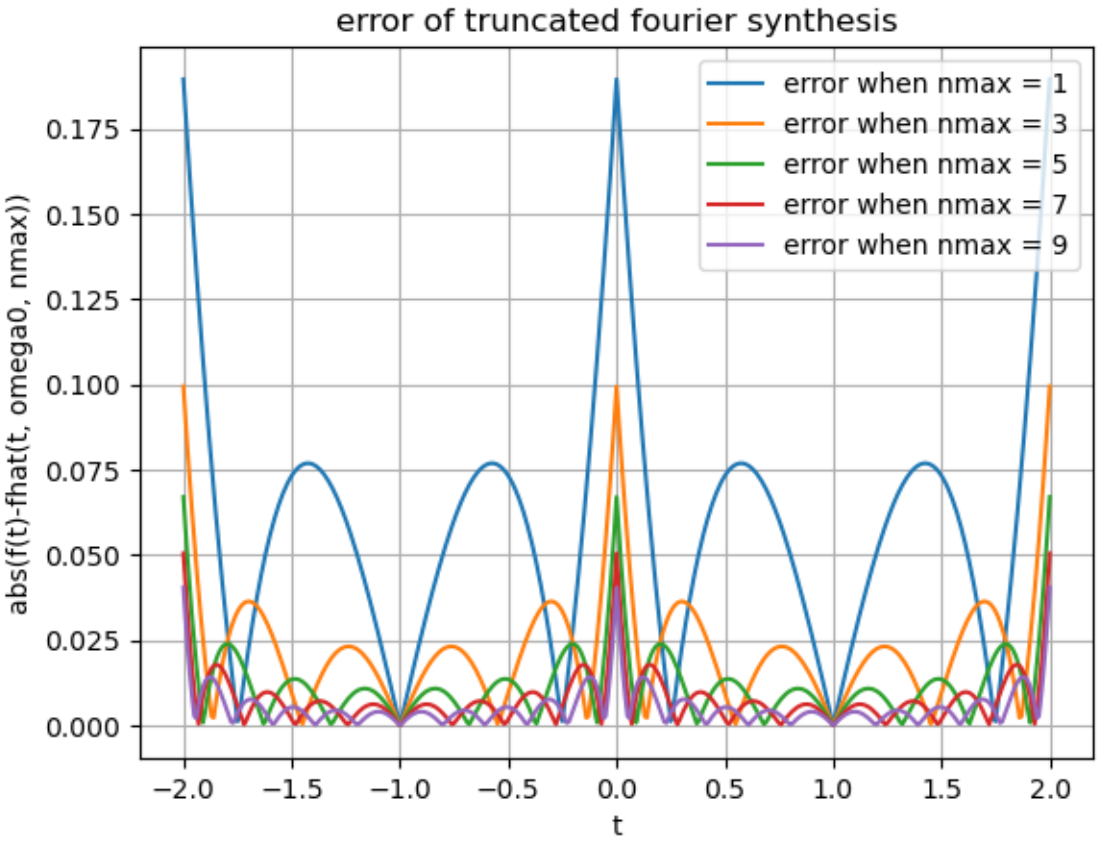
$$\text{error} = \left| f(t) - \hat{f}(t, 0.5\pi, n_{\max}) \right|$$

[8]:
```
t = np.arange(-2, 2.01, 0.01)

for i in range(0,5) :

    y = fhat(t,0.5*math.pi, 2*i +1)
    error = np.abs(f(t)-y)
    plt.plot(t, error, label=f'error when nmax = {2*i +1}')

plt.xlabel('t')
plt.ylabel('abs(f(t)-fhat(t, omega0, nmax))')
plt.title('error of truncated fourier synthesis')
plt.legend()
plt.grid(True)
plt.show()
```



The second error calculated was the percentage error, using the formula:

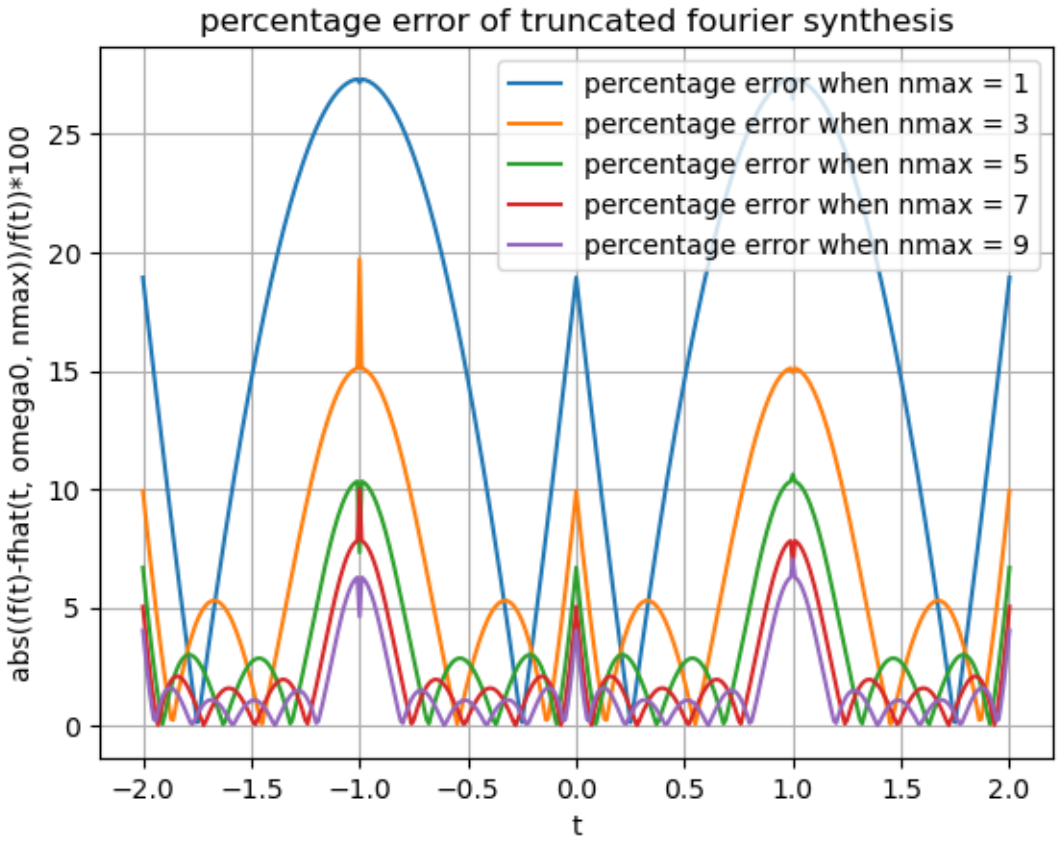
$$\text{error} = \left| \frac{f(t) - \hat{f}(t, 0.5\pi, n_{\max})}{f(t)} \right| \times 100$$

[9]:
```
t = np.arange(-2, 2.01, 0.01)

for i in range(0,5) :

    y = fhat(t,0.5*math.pi, 2*i +1)
    error = abs((f(t)-y)/f(t))*100
    plt.plot(t, error, label=f'percentage error when nmax = {2*i +1}')

plt.xlabel('t')
plt.ylabel('abs((f(t)-fhat(t, omega0, nmax))/f(t))*100')
plt.title('percentage error of truncated fourier synthesis')
plt.legend()
plt.grid(True)
plt.show()
```



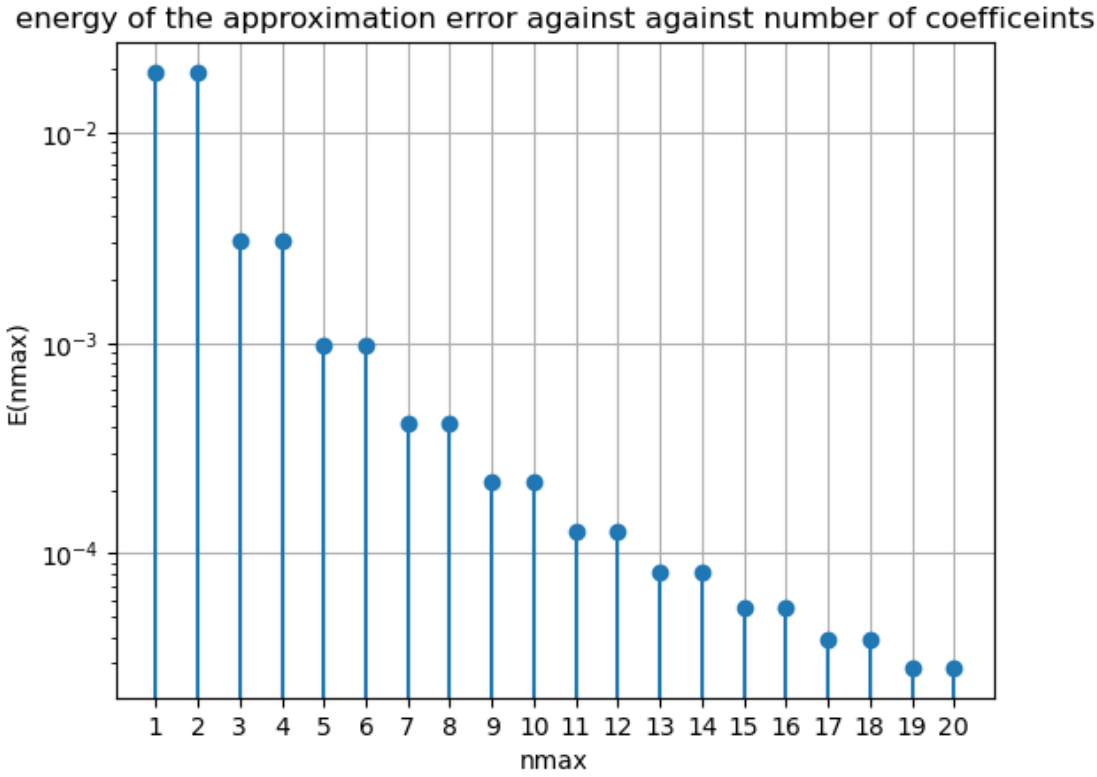
Q 2.3

Note that the single peiod of length 4, was taken to be from -2 to 2

[10]:
```
def E(nmax) :
    t = np.arange(-2, 2.01, 0.01)
    y = abs(f(t)- fhat(t, 0.5*np.pi, nmax))**2
    return np.trapz(y, t)
```

Q 2.4

[11]:
```
x = np.arange(1,21)
y = [E(i) for i in x]
plt.stem(x, y, basefmt=" ")
plt.yscale('log')
plt.title("energy of the approximation error against against number of␣
 ↪coefficeints")
plt.xlabel("nmax")
plt.ylabel("E(nmax)")#
plt.xticks(range(1, 21))
plt.grid(True)
plt.show()
```



From the graph above, it can be observed that as the number of coefficients ($n_{\max}$) increases, the energy approximation error decreases exponentially. This is clear due to straight-line behavior on the logarithmic scale, which indicates exponential decay. Exponential decay quickly converges to 0, which means that :

$$\lim_{n\to\infty} E(n) = 0.$$

Consequently, the fourier series approximation becomes increasingly accurate as the number of terms (coefficients) in the series is increased.

This also shows how the fourier series truncation is able to approximate the graph very well with only a few coeffecients.