

Problem 10 - DeBoor Algorithm

```
In [12]: # Functrions from Exercise shit
def f_x(t): return (t+1)*cos(4*pi*t)
def f_y(t): return (t+1)*sin(4*pi*t)
```

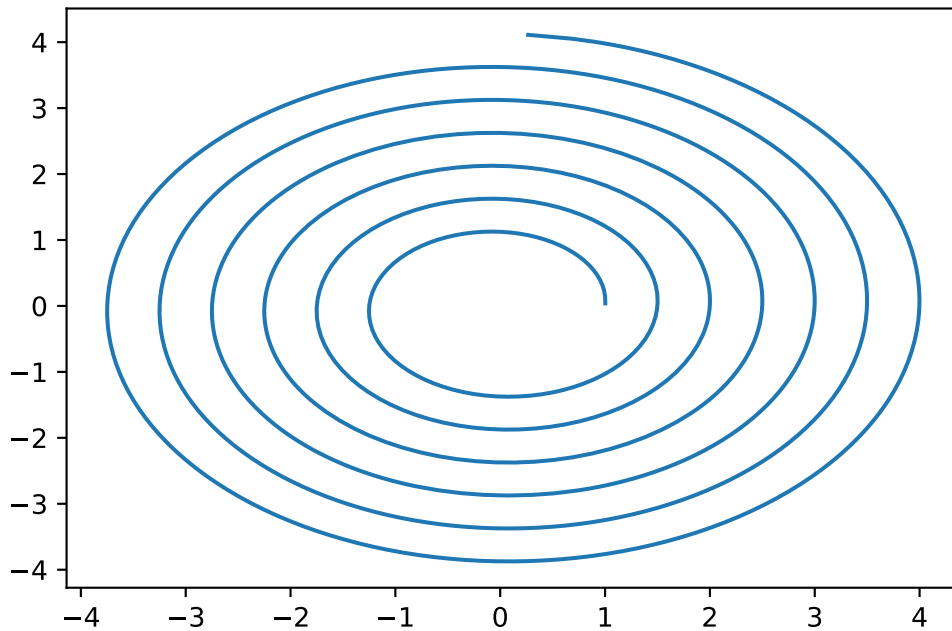
Implementartion of the DeBoor Algorithm

```
In [13]: # C-Coefficients accordinbg to (Dahmen+Reusken Alg 9.12)
def C(j, p, k, x_vals, c_vals, t):
    if p == 0:
        return c_vals[j]
    else:
        return (x_vals - t[j]) / (t[j+k-p] - t[j]) * C(j, p-1, k, x_vals, c_vals, t) +
            (t[j+k-p] - x_vals) / (t[j+k-p] - t[j]) * C(j-1, p-1, k, x_vals, c_vals, t)

# Find m ∈ [t_m, t_{m+1}] while excluding k-multiple points at the ends
# Would work much faster with divide-and-conquer algorithm but this one does the job as
def find_m(x, t, k):
    for m in range(len(t)-1):
        if t[m] <= x < t[m+1]:
            return m
    return len(t)-k-2
```

```
In [14]: def BSpline(f):
    end = pi
    k = 6
    n = 1000
    knots = [0 for i in range(k)] + np.linspace(0, end, n).tolist() + [end for i in range(k)]
    control_points = [f(j) for j in np.linspace(0, end, n+2*k)]
    return [C(find_m(t, knots, k), k-1, k, t, control_points, knots) for t in np.linspace(0, end, n)]

S_x = BSpline(f_x)
S_y = BSpline(f_y)
plt.plot(S_x, S_y)
plt.show()
```



Problem 12 - Discrete Fourier Transform

```
In [3]: import numpy as np
from math import sin, cos, pi, exp, log2
from scipy.fftpack import fft
import time
import matplotlib.pyplot as plt
```

```
In [4]: def f1(x): return sin(40*pi*x)
def f2(x,a=0.25, b=0.75):
    if (a <= x) and (x < b): return 1.0
    else: return 0.0
def f3(x): return min([x, 1 - x])
def f4(x): return exp(-100.0*(x-0.5)**2)
def f5(x): return exp(-4.0*(x-0.5)**2)
def f6(x): return exp(-100.0*(x-0.5)**2) * sin(40*pi*x)

func_list = [f1, f2, f3, f4, f5, f6]
```

a) Evaluation of the runtime and memory requirements

The DFT of a vector of size N is a linear transformation and can therefore be written as:

$$\bar{f} = M \cdot a$$

with $M_{j,k}$ being a $N \times N$ Matrix

$$M_{j,k} = e^{-2\pi i \cdot \frac{j \cdot k}{N}}$$

The complexity of a matrix-vector calculation DFT is $O(n^2)$ We need memory for the input- and output-vector and the matrix

$$\Rightarrow \text{Memory} = (N^2 + 2N) \cdot \text{Bytes/value}$$

Using complex float64 number from numpy (8 Bytes for real and imag. part = 16 bytes / value) and $p = 16$, one DFT would require

$$(2^{16^2} + 2 \cdot 16) \cdot 16 \text{bytes} \approx 69 \text{GB}$$

Using the same approximation but for FFT ($O(N \log(N))$) evaluates to

$$2^{16} \cdot \log(2^{16}) \cdot 16 \text{bytes} \approx 5 \text{MB}$$

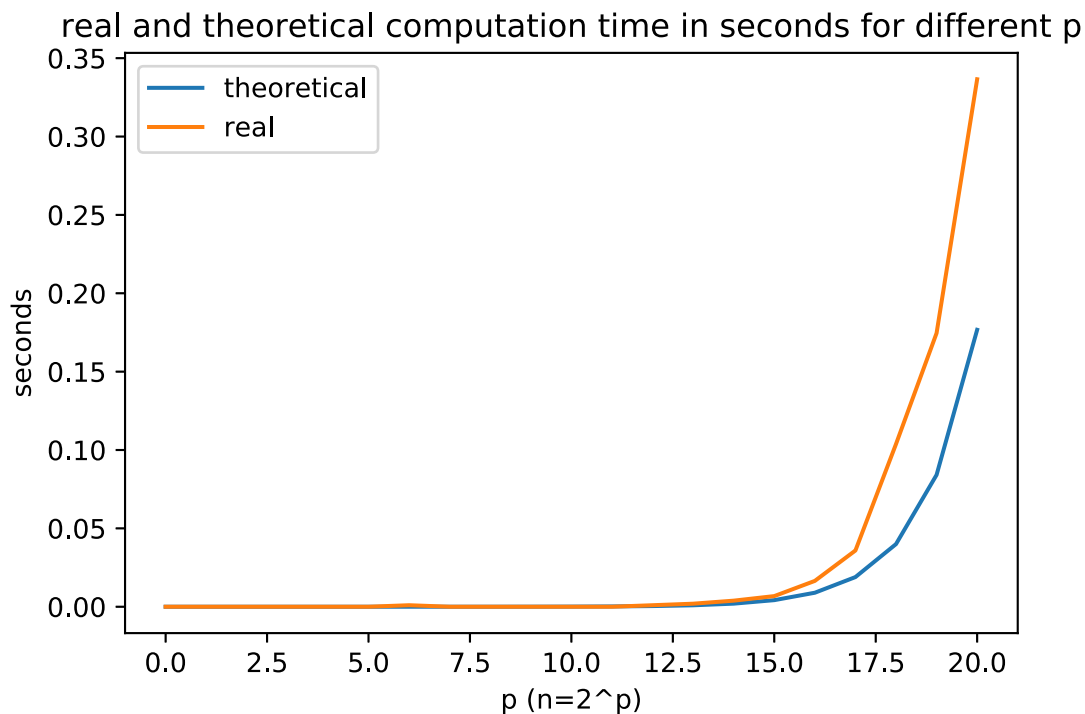
```
In [5]: def normalize(v):
        norm=np.linalg.norm(v, ord=1)
        if norm==0:
            norm=np.finfo(v.dtype).eps
        return v/norm
```

```
In [6]: times = []
        p_max = 22 # evaluated b trial and error
        for p in range(2, p_max+1):
            n = 2**p
            x = [j/n for j in range(n)]
            f1_vals = np.array([f1(x_i) for x_i in x])

            start_time = time.time()
            a_k = fft(f1_vals)
            end_time = time.time()
            times.append(end_time - start_time)

        # calculate the theoretical complexities and then normalize it with the longest time
        # this allows us to roughly compare theoretical vs real
        complexities = []
        for p in range(2, p_max+1):
            n = 2**(p-1)
            complexities.append(n*log2(n))
        complexities = normalize(complexities)*(times[-1])

        plt.plot(complexities, label="theoretical")
        plt.plot(times, label="real")
        plt.legend()
        plt.title("real and theoretical computation time in seconds for different p")
        plt.ylabel("seconds")
        plt.xlabel("p (n=2^p)")
        plt.show()
```



The evaluation for $p = 22$ took on my machine 0.35s

We can estimate the time a DFT of the same size would take by multiplying it with the ration of the complexities of a given N

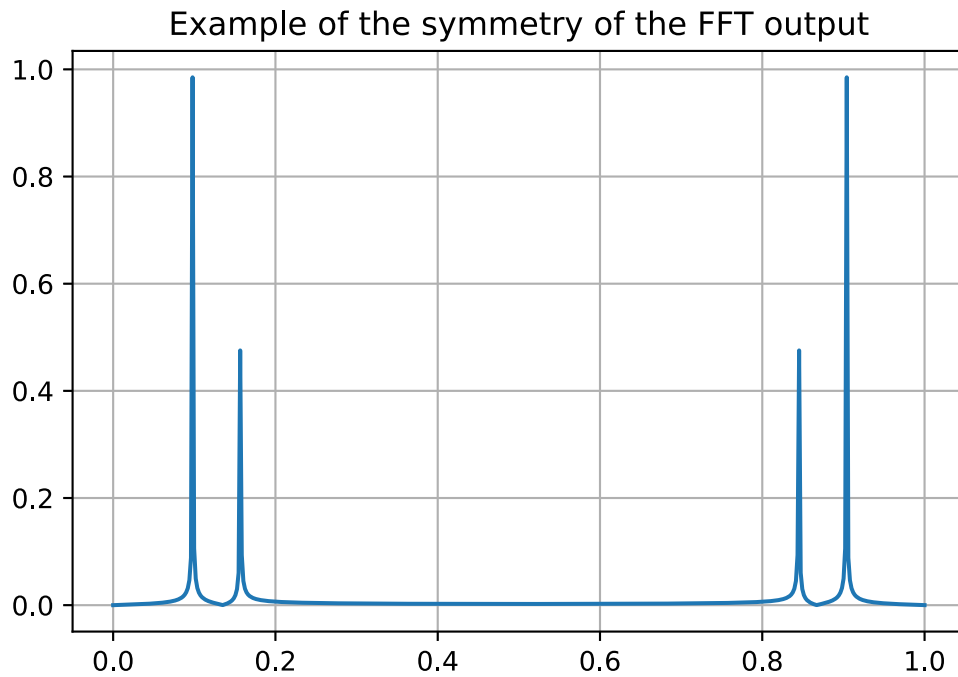
$$t = t_{FFT} \cdot \frac{O(N^2)}{O(N \cdot \log(N))} = 0.35s \cdot \frac{O(2^{16^2})}{O(2^{16} \cdot \log(2^{16}))} \approx 2000 \text{ years}$$

b) Relation between a_k and a_{n-k}

The FFT algorithm returns a series that is mirrored around $a_{n/2}$

$$\implies a_k = a_{n-k}$$

```
In [7]: N = 2**9
x = np.linspace(0.0, N, N)
y = np.sin(50.0 * 2.0*np.pi*x) + 0.5*np.sin(80.0 * 2.0*np.pi*x)
yf = fft(y)
xf = np.linspace(0.0, 1.0, N)
plt.plot(xf, 2.0/N * np.abs(yf))
plt.title("Example of the symmetry of the FFT output")
plt.grid()
plt.show()
```

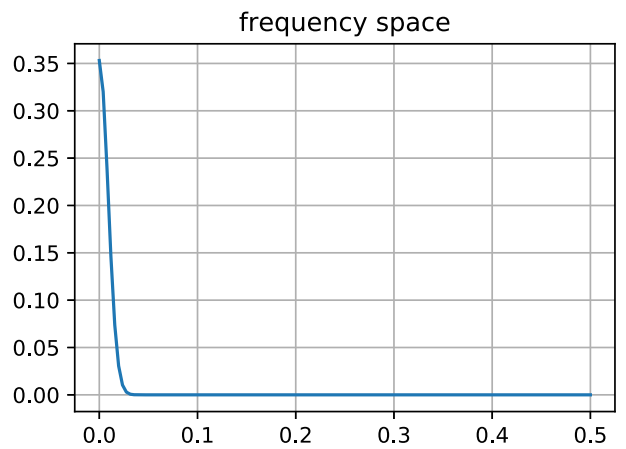
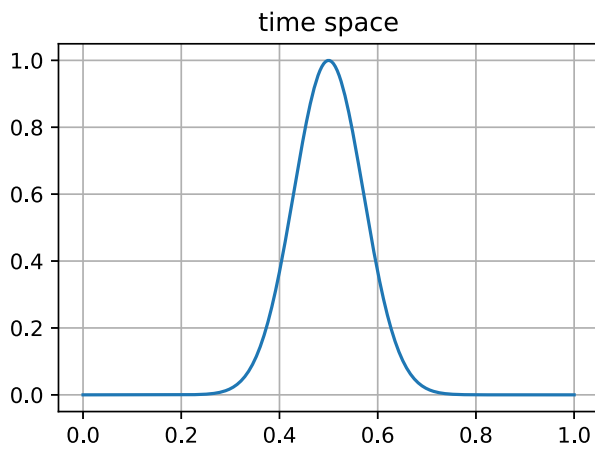
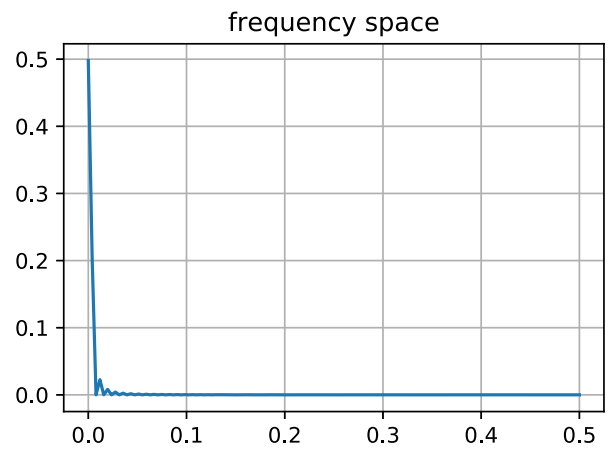
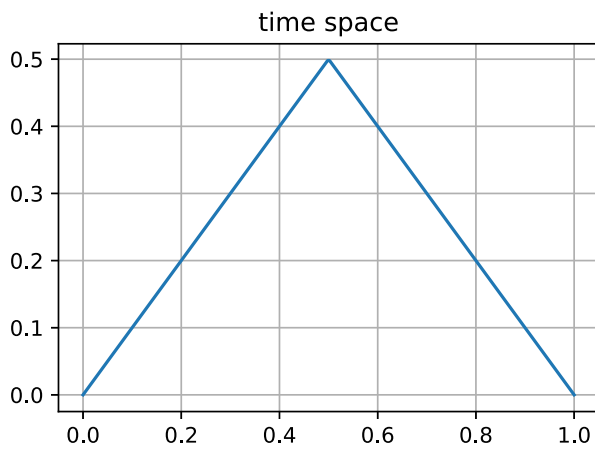
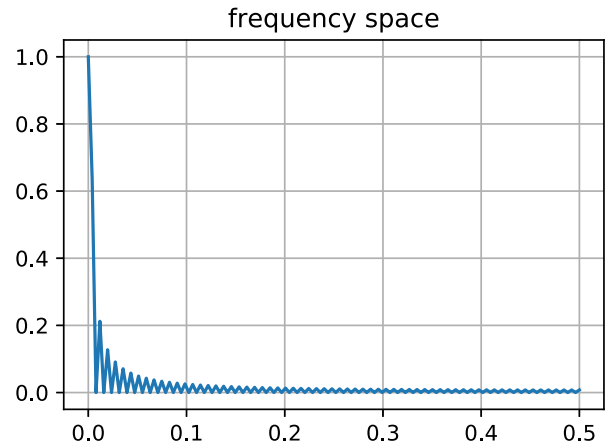
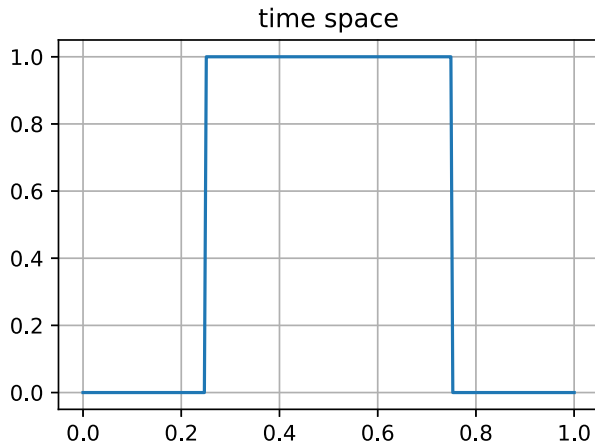
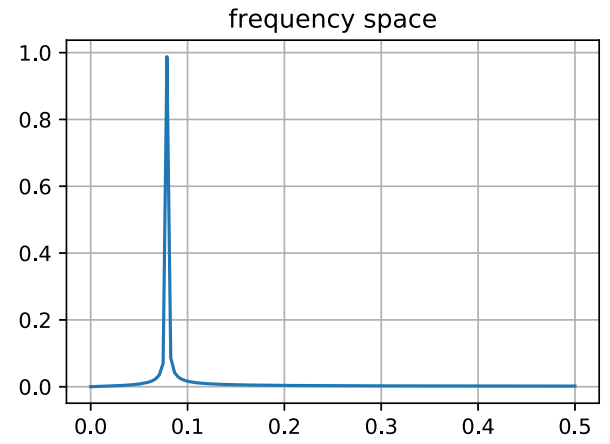
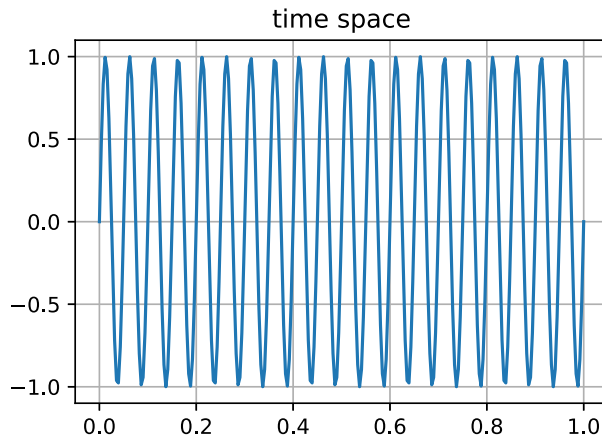


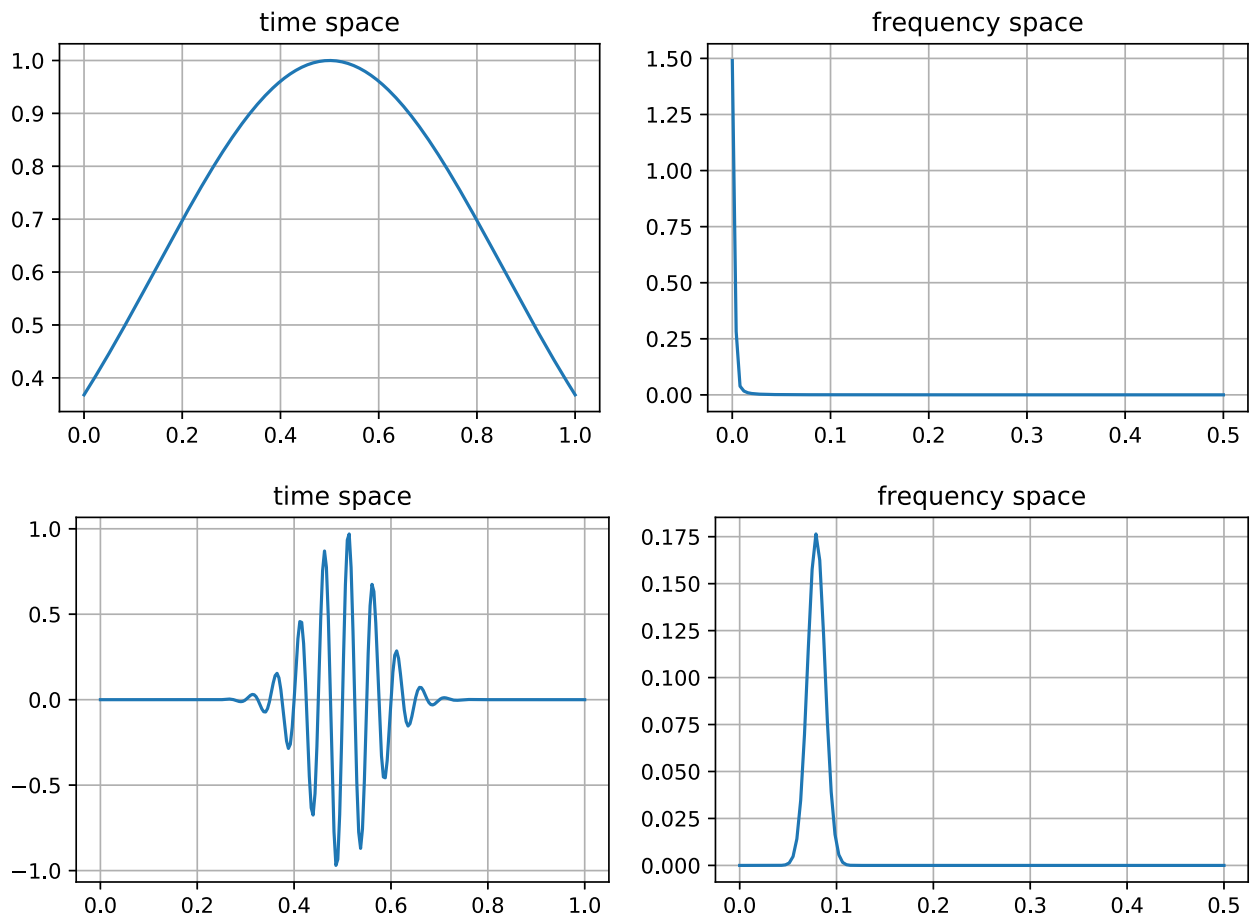
c) Testing different functions

```
In [8]: def plot_fft(f):
    p = 8
    N = 2**p
    x = np.linspace(0.0, 1, N)
    y = [f(x) for x in x]
    yf = fft(y)
    xf = np.linspace(0.0, 1.0/(2.0), N//2)

    #plt.cla()
    #plt.figure(figsize=(10,7))
    plt.subplot(221)
    plt.plot(x,y)
    plt.grid()
    plt.title("time space")
    plt.subplot(222)
    plt.plot(xf, 2.0/N * np.abs(yf[0:N//2]))
    plt.title("frequency space")
    plt.grid()
    plt.show()
```

```
In [11]: for f in func_list:
    plot_fft(f)
```





By messing around I found that

$$C = 1/N$$

$$\beta = 1$$

Very basic Explanation: If we increase N we increase the amount of points in time-space and therefore the highest possible frequency that can be represented below the Shannon Nyquist limit. The discrete frequency space contains "a list" of frequencies that can be represented in time-space. If we increase N we increase the number of possible frequencies in said list. Therefore, any peaks in frequency-space will get sharper with bigger N since those peak frequencies make up a smaller portion of our total frequency space.

Or in other words: If our time-signal contains half of all possible frequencies, $1/2$ of all a_k will be non-zero. If we double the resolution but don't change the signal only $1/4$ of all a_k will be non-zero as more frequencies can no longer be represented, and so on.. $\implies 1/N$ – Relation

```
In [10]: f = f2
p = 8
N = 2**p
x = np.linspace(0.0, 1, N)
y = [f(x) for x in x]
yf = fft(y)
xf = np.linspace(0.0, 1.0/(2.0), N//2)

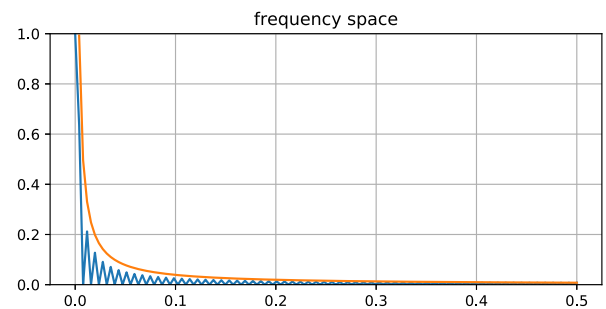
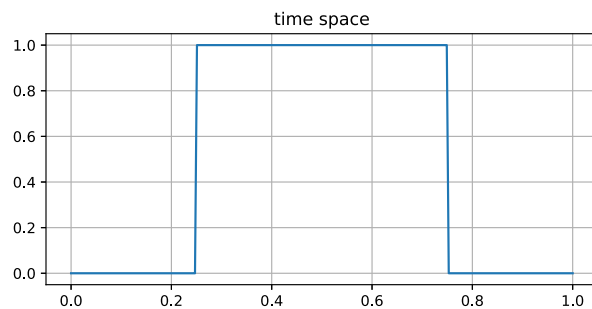
C = 1.0/(N)
b = 1
```

```

envelope = [C*i**-b for i in xf]

plt.figure(figsize=(15,7))
plt.subplot(221)
plt.plot(x,y)
plt.grid()
plt.title("time space")
plt.subplot(222)
plt.plot(xf, 2.0/N * np.abs(yf[0:N//2]))
plt.plot(xf, envelope)
plt.ylim(0,1)
plt.title("frequency space")
plt.grid()
plt.show()

```



In []: