11 Seconda lezione A

11.1 Trasformate di Fourier

11.1.1 DFT

La trasformata di Fourier è una trasformata integrale che ci permette di cambiare dominio della nostra funzione: ad esempio da tempo a frequenze o da spazio a numero d'onda. Data una funzione:

$$f(t) \quad \text{tale che} \quad \int_{-\infty}^{\infty} |f(t)| dt < \infty \tag{27}$$

Definiamo trasformata e anti trasformata di f come:

$$\tilde{f}(\omega) = \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt = \mathcal{F}\{f\}(\omega) \tag{28}$$

$$f(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} \tilde{f}(\omega) e^{-i\omega t} d\omega = \mathcal{F}^{-1}\{\tilde{f}\}(t) \tag{29}$$

$$\tag{30}$$

e gode di varie proprietà:

$$\mathcal{F}(D^k f) = (-i\omega)^k \mathcal{F}(f) \qquad \mathcal{F}(f(t-a)) = e^{i\omega a}\mathcal{F}(f(t)) \qquad \mathcal{F}(e^{i\omega a} f(t)) = \tilde{f}(\omega - a) \qquad \mathcal{F}((it)^k f) = D^k \mathcal{F}(f) \tag{31}$$

Qui abbiamo in realtà aggiunto altre ipotesi su f (e.g. $f \in C^k$) ma va beh. Vediamo un piccolo grafico che meglio ci consenta di capire, in maniera intuitiva, cosa vuol dire questo cambio di base:

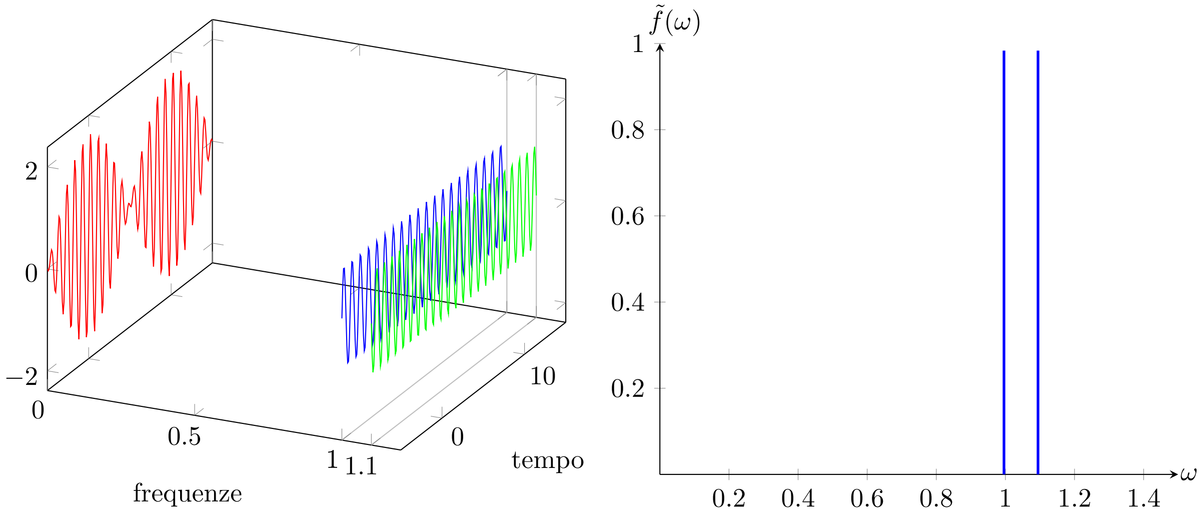


Figura 4: Allora, capisco che i due grafici sopra, in special modo il grafico di sinistra, siano magari non di immediata comprensione ma cerchiamo di descriverli e capirli. Partiamo da quello di sinistra: si tratta di un immagine pittorica per vedere quella che è la serie di Fourier, ovvero una funzione scritta come somma di seni e/o coseni (armoniche). La funzione che vedete plottata sul piano a frequenza zero è: $\sin(2\pi t1) + \sin(2\pi t1.1)$ (ovviamente quella funzione non ha una omega nulla, è solo plottata lì a titolo espositivo). Tale funzione è formata da due seni che sono i due plottati a parte (blu e verde); ognuno giace sul piano alla propria frequenza, rispettivamente $\omega = 1, 1.1$ (purtroppo per avere i battimenti le ω devono essere vicine). Questo plot ci aiuta a capire cosa fa la trasformata di Fourier, perché vediamo in funzione di omega solo due segnali, quindi ci aspettiamo solo due valori di frequenza non nulli. Ciò che la trasformata ci restituisce è il grafico a destra: ovvero un grafico che ci fa capire le quali sono le armoniche che compongono la nostra funzione. Nella fattispecie sono due delta di Dirac (in linea teorica); nella pratica saranno delle gaussiane più o meno strette a seconda dei dati.

Andiamo ora nel discreto; abbiamo:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn}, \qquad k = 0, \dots, N-1 \tag{32}$$

e non è difficile vedere che fondamentalmente la trasformata di Fourier discreta (DFT) è un prodotto matrice per vettore:

$$X_k = W_{kn} x_n \qquad W_{kn} = e^{-\frac{2\pi i}{N}kn} \tag{33}$$

e l'anti trasformata non sarà altro che:

$$X_k = \frac{1}{N} W_{kn}^{-1} x_n \qquad W_{kn}^{-1} = e^{\frac{2\pi i}{N}kn} \tag{34}$$

Dunque è facile notare che la complessità dell'algoritmo è $\mathcal{O}(N^2)$; vediamone una semplice implementazione:

```python
"""
Implementetion of DFT
"""
import time
import numpy as np
import matplotlib.pyplot as plt


def DFT(x, anti=-1):
    '''
    Compute the discrete Fourier Transform of the 1D array x

    Parameters
    ----------
    x : 1darray
        data to transform
    anti : int, optional
        -1 trasform
         1 anti trasform

    Return
    ------
    dft : 1d array
        dft or anti dft of x
    '''

    N = len(x)          # length of array
    n = np.arange(N)    # array from 0 to N
    k = n[:, None]      # transposed of n written as a Nx1 matrix
    # is equivalent to k = np.reshape(n, (N, 1))
    # so k * n will be a N x N matrix

    M = np.exp(anti * 2j * np.pi * k * n / N)
    dft = M @ x

    if anti == 1:
        return dft/N
    else:
        return dft
```

11.1.2 FFT

I signori Cooley e Tukey si inventarono un modo per accelerare un po' il calcolo della DFT e crearono la FFT (trasformata di Fourier veloce) la quale ha ordine $\mathcal{O}(N \ln_2(N))$. Qui vedremo il caso più semplice, quello in cui l'array da trasformare deve essere lungo necessariamente una potenza di 2 (FFT radix 2). Esistono anche altri algoritmi, chiamati a radice mista, in cui possiamo rilassare questo vincolo, ad esempio le funzioni di numpy non hanno questo vincolo.vediamo brevemente l'algoritmo:

$$\begin{align} X_k &= \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N}kn} \tag{35}\\ &= \sum_{n=0}^{N/2-1} x_{2n} e^{-\frac{2\pi i}{N}k(2n)} + \sum_{n=0}^{N/2-1} x_{2n+1} e^{-\frac{2\pi i}{N}k(2n+1)} \tag{36}\\ &= \sum_{n=0}^{N/2-1} x_{2n} e^{-\frac{2\pi i}{N/2}kn} + e^{-\frac{2\pi i}{N}k} \sum_{n=0}^{N/2-1} x_{2n+1} e^{-\frac{2\pi i}{N/2}kn} \tag{37}\\ &= DFT(x_{2n}) + e^{-\frac{2\pi i}{N}k} DFT(x_{2n+1}) \tag{38} \end{align}$$

e quindi ripetiamo ricorsivamente questa divisione, fino ad arrivare ad un N_{min} che blocca la ricorsione e ed esegue una DFT come vista sopra; quindi N/N_{min} DFT:

```python
"""
Implementetion of FFT
"""
import time
import numpy as np
import matplotlib.pyplot as plt


def FFT(x, anti=1):
    '''
    A recursive implementation of the Cooley-Tukey FFT

    Parameters
    ----------
    x : 1darray
        data to transform
    anti : int, optional
        -1 trasform
         1 anti trasform

    Return
    ------
    fft : 1d array
        fft or anti fft of x
    '''

    N = x.shape[0]

    if N % 2 > 0:
        raise ValueError("size of x must be a power of 2")
    elif N <= 32:
        return DFT(x, anti)
    else:
        X_even = FFT(x[0::2])
        X_odd  = FFT(x[1::2])
        factor = np.exp(-anti*2j * np.pi * np.arange(N) / N)
        return np.concatenate([X_even + factor[:N / 2] * X_odd,
                               X_even + factor[N / 2:] * X_odd])
```

Cerchiamo di capire perché questa divisione accelera il calcolo: abbiamo detto che la DFT è $\mathcal{O}(N^2)$. Quindi al primo passo come visto sopra abbiamo 2 DFT e un prodotto di due vettori $\mathcal{O}(N)$ per cui:

- prima divisione: $\frac{N}{2} \longrightarrow 2\underbrace{\left(\frac{N}{2}\right)^2}_{\text{DFT}} + N = \frac{N^2}{2} + N$
- seconda divisione: $\frac{N}{4} \longrightarrow 2\Big(2\underbrace{\left(\frac{N}{4}\right)^2}_{\text{DFT}} + \frac{N}{2}\Big) + N = \frac{N^2}{4} + 2N$

Capendo l'antifona otteniamo che l'ultima divisione è: $\frac{N}{2^p} \longrightarrow \frac{N^2}{2^p} + pN = \frac{N^2}{N} + \ln_2(N)N \longrightarrow \mathcal{O}(N\ln_2(N))$
Dove abbiamo usato che $N = 2^m$ allora al massimo deve essere $p = m = \ln_2(N)$.
Possiamo però costruire un algoritmo iterativo che ottimizzi il codice sopra scritto e la vettorizzazione di Python ci aiuta:

```python
"""
Implementetion of FFT
"""
import time
import numpy as np
import matplotlib.pyplot as plt

def FFT(x, anti=-1):
    '''
    Compute the Fast Fourier Transform of the 1D array x.
    Using non recursive Cooley-Tukey FFT.
    In recursive FFT implementation, at the lowest
    recursion level we must perform  N/N_min DFT.
    The efficiency of the algorithm would benefit by
    computing these matrix-vector products all at once
    as a single matrix-matrix product.
    At each level of recursion, we also perform
    duplicate operations which can be vectorized.

    Parameters
    ----------
    x : 1darray
```

```
        data to transform
    anti : int, optional
        -1 trasform
         1 anti trasform

    Return
    ------
    fft : 1d array
        fft or anti fft of x

    '''
    N = len(x)

    if np.log2(N) % 1 > 0:
        msg_err = "The size of x must be a apower of 2"
        raise ValueError(msg_err)

    # stop criterion
    N_min = min(N, 2**2)

    #  DFT on all length-N_min sub-problems
    n = np.arange(N_min)
    k = n[:, None]
    M = np.exp(anti * 2j * np.pi * n * k / N_min)
    X = np.dot(M, x.reshape((N_min, -1)))

    while X.shape[0] < N:
        # first  part of the matrix, the one on the left
        X_even = X[:,:X.shape[1] // 2 ] # all rows, first  X.shape[1]//2 columns
        # second part of the matrix, the one on the right
        X_odd  = X[:, X.shape[1] // 2:] # all rows, second X.shape[1]//2 columns

        f = np.exp(anti * 1j * np.pi * np.arange(X.shape[0]) / X.shape[0])[:, None]
        X = np.vstack([X_even + f*X_odd, X_even - f*X_odd]) # re-merge the matrix

    fft = X.ravel() # flattens the array
    # from  matrix Nx1 to array with length N

    if anti == 1:
        return fft/N
    else :
        return fft
```

11.1.3 RFFT

Ultima interessante implementazione è il caso in cui l'input sia reale, per cui è possibile definire una variabile complessa fare una FFT lunga la meta e ricostruire lo spettro con le proprietà di simmetria della FFT. Se siamo interessati alle frequenze positive, che è il caso usuale basta fondamentalmente prendere i primi $N/2+1$ elementi della nostra FFT.

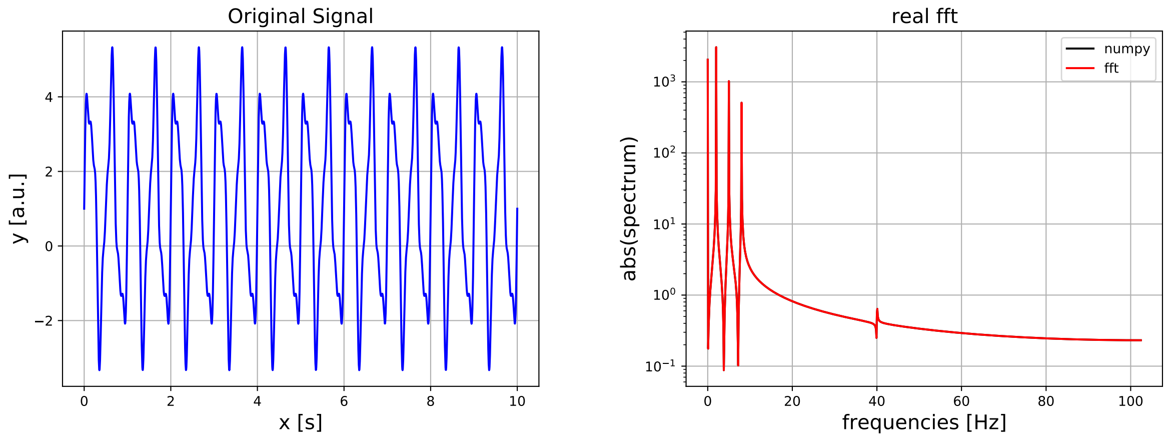
```
def RFFT(x, anti=-1):
    '''
    Compute the fft for real value using FFT
    only values corresponding to positive
    frequencies are returned.

    The transform is implemented by passing to
    a complex variable z = x[2n] + j x[2n+1]
    then an fft of length N/2 is calculated
    For the inverse we adopt an other method
    (the previous method didn't work,
    if you know how to fix it ... I would be grateful)

    Parameters
    ----------
    x : 1darray
        data to transform
    anti : int, optional
        -1 trasform
         1 anti trasform

    Return
    ------
    rfft : 1d array
        rfft or anti rfft of x
```

```python
    '''
    if anti == -1 :
        z  = x[0::2] + 1j * x[1::2]  # Splitting odd and even
        Zf = FFT(z)
        Zc = np.array([Zf[-k] for k in range(len(z))]).conj()
        Zx =  0.5  * (Zf + Zc)
        Zy = -0.5j * (Zf - Zc)

        N = len(x)
        W = np.exp(- 2j * np.pi * np.arange(N//2) / N)
        Z = np.concatenate([Zx + W*Zy, Zx - W*Zy])

        return Z[:N//2+1]

    if anti == 1 :
        # we use the fft symmetries to reconstruct the whole spectrum
        N = 2*(len(x)-1) # length of final array
        x1 = x[:-1]      # cut last value
        S = len(x1)      # length of new array
        xn = np.zeros(N, dtype=complex)
        xn[0:S] = x1
        xx = x[1:]       # cut first element, zero frequency mode
        xx = xx[::-1]    # rewind array
        xn[S:N] = xx.conj()
        z = FFT(xn, anti=1)

        return np.real(z)
```

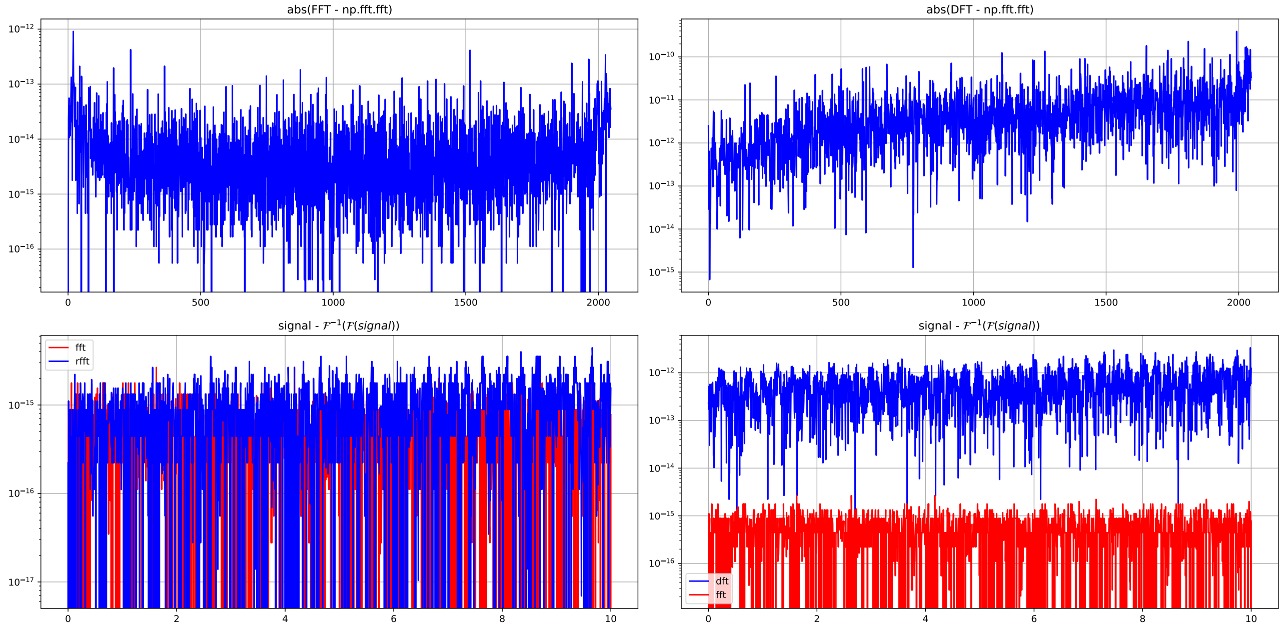
Ultima cosa da vedere è come creare l'array delle frequenze, quello che sarebbe np.fff.fftfreq:

```python
def fft_freq(n, d, real):
    '''
    Return the Discrete Fourier Transform sample frequencies.
    if real = False then:
    f = [0, 1, ...,   n/2-1,     -n/2, ..., -1] / (d*n)   if n is even
    f = [0, 1, ..., (n-1)/2, -(n-1)/2, ..., -1] / (d*n)   if n is odd
    else :
    f = [0, 1, ...,     n/2-1,     n/2] / (d*n)   if n is even
    f = [0, 1, ..., (n-1)/2-1, (n-1)/2] / (d*n)   if n is odd

    Parameters
    ----------
    n : int
        length of array that you transform

    d : float
        Sample spacing (inverse of the sampling rate).
        If the data array is in seconds
        the frequencies will be in hertz
    real : bool
        false for fft
        true for rfft

    Returns
    -------
    f: 1d array
        Array of length n containing the sample frequencies.
    '''
    if not real:
        if n%2 == 0:
            f1 = np.array([i for i in range(0, n//2)])
            f2 = np.array([i for i in range(-n//2,0)])
            return np.concatenate((f1, f2))/(d*n)
        else :
            f1 = np.array([i for i in range((n-1)//2 + 1)])
            f2 = np.array([i for i in range(-(n-1)//2, 0)])
            return np.concatenate((f1, f2))/(d*n)
    if real:
        if n%2 == 0:
            f1 = np.array([i for i in range(0, n//2 +1)])
            return f1 / (d*n)
        else :
            f1 = np.array([i for i in range((n-1)//2 +1)])
            return f1 / (d*n)
```

Fatto ciò possiamo chiamare le nostre funzioni e vedere i risultati. La restante parte del codice non verrà mostrata perché si tratta solo di plot, il codice intero è comunque disponibile. Tutti i codici sopra sono pezzi



di un unico grande codice.



11.2 Applicazioni delle FFT

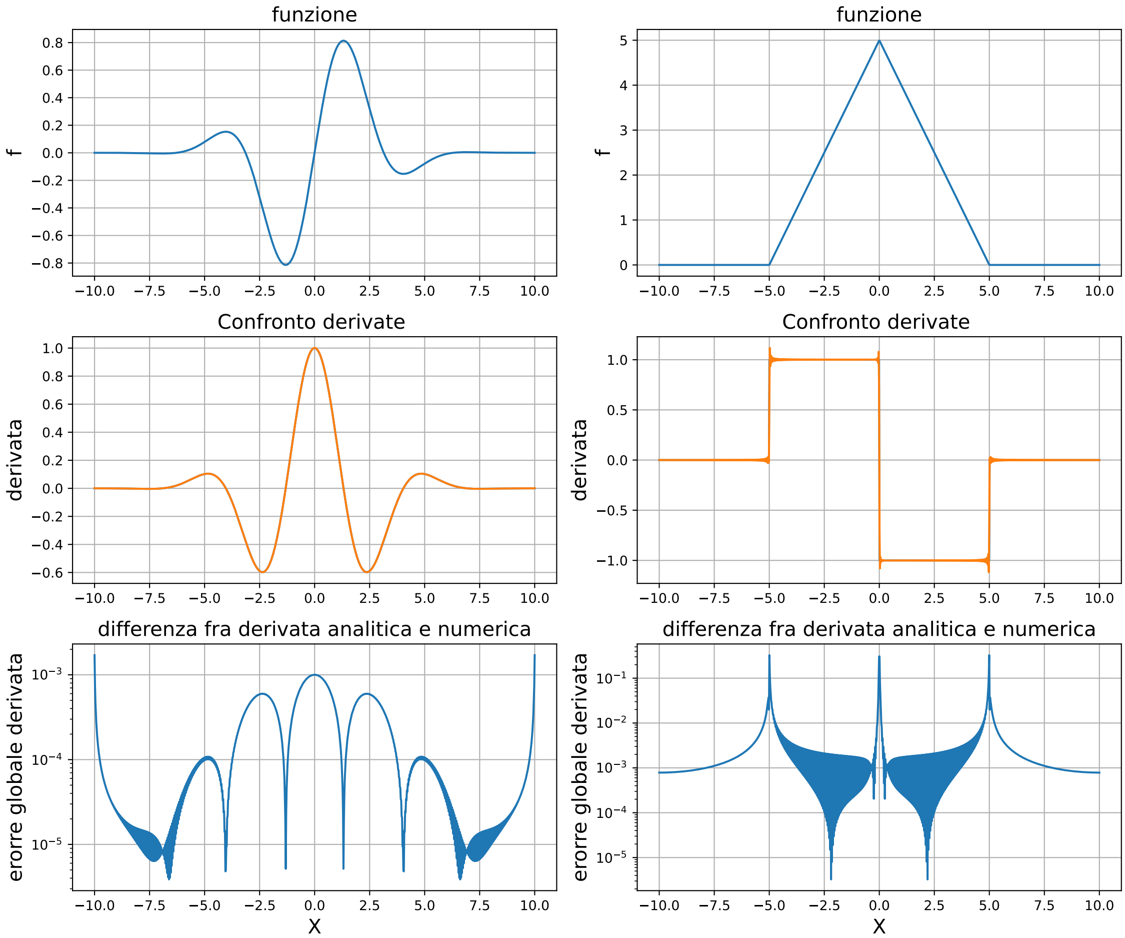
11.2.1 Derivate con FFT

Vediamo ora una comoda applicazione delle fft, ovvero il calcolo delle derivate. In linea di principio quanto stiamo qui per fare si può fare con ogni tipo di polinomio con il quale possiamo sviluppare una certa funzione, ad esempio Legendre o Chebyshev; questo è chiamato metodo dei punti di collocazione. In serie di Fourier la cosa è piuttosto semplice, grazie alla sue proprietà matematiche, una derivata nello spazio (o nel tempo) corrisponde ad una moltiplicazione per ik ($i\omega$) nello spazio degli impulsi (frequenze). Vediamo come:

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

xi = 10          # estemo sinisto
xf = -xi         # estemo destro
N  = 1000        # numero punti
dx = (xf-xi)/N   # spaziatura punti

k = 2*np.pi*np.fft.fftfreq(N, dx) # vettore d'onda

#============================== CASO TRANQUILLO ==============================
x = np.linspace(xi, xf, N)                     # array posizioni
f = np.sin(x)*np.exp(-x**2/10)                 # funzione di cui calcolare la derivata
g = np.cos(x)*np.exp(-x**2/10) - 2/10*x*f      # derivata analitica

f_hat  = np.fft.fft(f)        # trasformo con fourier
```

```
df_hat = 1j*k*f_hat           # moltiplico per l'impulso
df     = np.fft.ifft(df_hat) # derivata nello spazio

plt.figure(1)

plt.subplot(321)
plt.title('funzione', fontsize=15)
plt.ylabel('f', fontsize=15)
plt.plot(x, f)
plt.grid()

plt.subplot(323)
plt.title('Confronto derivate ', fontsize=15)
plt.ylabel('derivata', fontsize=15)
plt.plot(x, g)
plt.plot(x, df)
plt.grid()

plt.subplot(325)
plt.title('differenza fra derivata analitica e numerica', fontsize=15)
plt.ylabel('erorre globale derivata', fontsize=15)
plt.xlabel("X", fontsize=15)
plt.plot(x, abs(g-df))
plt.yscale('log')
plt.grid()
#============================== CASO MENO TRANQUILLO ==============================

def f():
    '''funzione
    '''
    y = []
    for xi in x:
        if xi < -5 :
            y.append(0)
        if xi > -5 and xi < 0:
            y.append(xi + 5)
        if xi > 0 and xi < 5:
            y.append(-xi + 5)
        if xi > 5:
            y.append(0)
    return np.array(y)

def g():
    ''' derivata analitica
    '''
    y = []
    for xi in x:
        if xi < -5 :
            y.append(0)
        if xi > -5 and xi < 0:
            y.append(1)
        if xi > 0 and xi < 5:
            y.append(-1)
        if xi > 5:
            y.append(0)
    return np.array(y)

f = f()
g = g()

f_hat  = np.fft.fft(f)       # trasformo con fourier
df_hat = 1j*k*f_hat           # moltiplico per l'impulso
df     = np.fft.ifft(df_hat) # derivata nello spazio

plt.subplot(322)
plt.title('funzione', fontsize=15)
plt.ylabel('f', fontsize=15)
plt.plot(x, f)
plt.grid()

plt.subplot(324)
plt.title('Confronto derivate ', fontsize=15)
plt.ylabel('derivata', fontsize=15)
plt.plot(x, g)
plt.plot(x, df)
plt.grid()
```

```
plt.subplot(326)
plt.title('differenza fra derivata analitica e numerica', fontsize=15)
plt.ylabel('erorre globale derivata', fontsize=15)
plt.xlabel("X", fontsize=15)
plt.plot(x, abs(g-df))
plt.yscale('log')
plt.grid()

plt.show()
```



Si vede facilmente come nel caso della funzione a tratti la questione sia più delicata, già ad occhio vediamo dei piccoli spike ai bordi dove la funzione cambia e la derivata ha un gradino, inoltre l'errore è maggiore.

11.2.2 Equazione di Burgers

Fare le derivate con le fft inoltre ci dà un'interessante prospettiva sulle PDE, infatti passando in trasformata, una PDE diventa una più tranquilla ODE, vediamo un semplice esempio. consideriamo l'equazione:

$$\frac{\partial u(t,x)}{\partial t} + u(t,x)\frac{\partial u(t,x)}{\partial x} = \nu \frac{\partial^2 u(t,x)}{\partial x^2} \quad , \tag{39}$$

nota come equazione di Burgers, un'equazione abbastanza usale in fluidodinamica che ad esempio modelizza la formazione di un fronte d'onda ad esempio gli shock. Non so se nei corsi di fluidodinamica si veda, vi basti sapere che, come quasi tutto, si parte da:

$$\begin{cases} \partial_t \rho + \partial_x(\rho v) = 0 \\ \partial_t(\rho v) + \partial_x(\rho v^2 + P) = \nu \partial_x^2 v \end{cases} \tag{40}$$

Assumendo come equazione di stato una politropica ed espandendo la densità e la velocità in serie di perturbazioni [e.g. $v = v_0 + v_1 + ...$, $\rho = \rho_0 + \rho_1 + ...$, (in genere v_0=0)] si ottiene che ρ_1 soddisfa l'equazione di Burgers. É facile vedere che passando in trasformata di nello spazio la nostra equazione diventa un'ode nel tempo:

$$\frac{\partial u(t,k)}{\partial t} + u(t,k)(iku(t,k)) = \nu(-k^2 u(t,k)) \quad . \tag{41}$$

Qui usiamo "scipy.integrate.odeint" per risolvere l'equazione, in una delle appendici, anzi due a vol essere precisi, sono trattati degli algoritmi risolutivi. Vediamo le poche righe di codice necessarie:

```python
"""
code to solve burger equation using fourier trasform in space
"""
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
import matplotlib.animation as animation

#==============================================================
# Parameters
#==============================================================

L = 1
T = 0.31

dx = 0.001
dt = 0.001
nu = 0.005

Nx = int(L/dx)
Nt = int(T/dt)

t = np.linspace(0, T, Nt)
x = np.linspace(0, L, Nx)

# wave number
k = 2*np.pi*np.fft.fftfreq(Nx, dx)
# Initial conditions
u0 = np.sin(2*np.pi*x)

#==============================================================
# Solutions
#==============================================================

def eq(u, t, k, nu):
    '''
    equation to be solved in spatial transform
    '''
    u_hat = np.fft.fft(u)
    du_hat = 1j*k*u_hat    # first  derivative
    ddu_hat = -k**2*u_hat  # second derivative

    #antitrasform
    du = np.fft.ifft(du_hat)
    ddu = np.fft.ifft(ddu_hat)

    #pde in time and space -> ode in time
    u_t = -u*du + nu*ddu

    return u_t.real


sol = odeint(eq, u0, t, args=(k, nu,)).T


#==============================================================
# Animations
#==============================================================

fig = plt.figure(2)
plt.xlim(np.min(x), np.max(x))
plt.ylim(np.min(sol), np.max(sol))

line, = plt.plot([], [], 'b')
def animate(i):
    line.set_data(x, sol[:,i])
    return line,

anim = animation.FuncAnimation(fig, animate, frames=Nt, interval=5, blit=True, repeat=True)

plt.grid()
plt.title('burger equation')
plt.xlabel('Distanza')
plt.ylabel('ampiezza')
```

```python

#anim.save('buger.mp4', fps=30, extra_args=['-vcodec', 'libx264'])

plt.show()
```

Eseguendo il codice vedrete l'animazione del fronte d'onda crearsi e grazie al termine diffusivo se ne evita la rottura.

11.2.3 Equazione di Schrödinger

La scorsa lezione abbiamo visto come risolvere l'equazione di Schrödinger nel caso stazionario, ora vogliamo aggiungere il tempo e quindi vedere come una funzione d'onda evolve. L'equazione adesso diventa:

$$i\frac{\partial \psi}{\partial t} = H\psi. \tag{42}$$

Assumiamo intanto che l'hamiltoniana non dipenda dal tempo (cfr. si conserva l'energia). Notiamo subito una cosa: chiaramente questa equazione è una PDE e modulo un'unità immaginaria è praticamente un'equazione diffusiva, tipo quella del calore. Si potrebbe quindi pensare di procedere allo stesso modo (proprio per l'equazione del calore è riportato un esempio in una delle appendici); c'è però un caveat: dato il suo significato di densità di probabilità è necessario che la psi sia normalizzata o per meglio dire e che la norma sia conservata durante l'evoluzione quindi il nostro algoritmo deve essere unitario. Sappiamo però per l'assunzione fatta precedentemente che la soluzione è:

$$\psi(t,x) = e^{-iHt}\psi(0,x). \tag{43}$$

Tutto molto tranquillo e tutto molto unitario, no? No chiaramente, H è una matrice, e il suo esponenziale (in paroloni mi pare si dicesse implementazione unitaria) è definito con la serie di potenze, che non vogliamo star qui a calcolare. Idea: sviluppiamo per piccoli tempi $e^{-iH\delta t} \simeq 1 - iH\delta t$ e applichiamo tutto ciò alla condizione iniziale $\psi(0,x)$. Sorge un nuovo problema: abbiamo perso l'unitarietà, il modulo quadro non si conserva. Quello che si può fare per risolvere è noto come split operator:

$$e^{iH\delta t/2}\psi(t+\delta t, x) = e^{-iH\delta t/2}\psi(t,x), \tag{44}$$

ovvero la funzione d'onda al tempo $t + \delta t$ evoluta indietro di $\delta t/2$ è uguale alla funzione d'onda al tempo t evoluta in avanti di $\delta t/2$. Quindi approssimando sempre l'esponenziale al prim'ordine si ha:

$$\psi(t+\delta t, x) = \frac{1 - iH\delta t/2}{1 + iH\delta t/2}\psi(t,x). \tag{45}$$

Mi perdonerete l'abuso di notazione, ovviamente dividere per $1 + iH\delta t/2$ vuol dire invertire la matrice, ovvero risolvere un sistema; scriviamo così solo per renderci conto del fatto che quel coefficiente è della forma: numero complesso diviso il suo coniugato, e per cui ha modulo uno, garantendo l'unitarietà dell'evoluzione. Si ok molto bello ma la trasformata di Fourier? Questo metodo può essere migliorato proprio grazie ad essa. Presa $H = T + V$, vale:

$$e^{-iH\delta t} = e^{-iV\delta t/2}e^{-iP\delta t}e^{-iV\delta t/2} + \mathcal{O}(\delta t^3). \tag{46}$$

Una cosa simile verrà usata nella prossima lezione quando parleremo brevemente degli integratori simplettici. Perché facciamo questa divisone? Questi tre termini sono più facili da calcolare? Noi vogliamo calcolare l'operatore di evoluzione temporale ma senza dover esponenziare una matrice (In termini di matrici T e V hanno la forma vista nella lezione precedente). Facendo cosi vediamo subito i termini dipendenti da V si calcolano veloci in quanto se la matrice è diagonale l'espansione in serie mi da gli esponenziali delle entrare. Quindi vorremo poter diagonalizzare anche T, quindi l'impulso. Ma per andare dalla base della posizione a quella dei momenti si esegue proprio una trasformata di Fourier. Quindi in quella base T sarà una matrice diagonale contenente i valori (al quadrato) che l'impulso può assumere (sempre nella nostra scatola di lato L in quanto abbiamo discretizzato, come sopra, lo spazio). La regola iterativa è dunque:

$$\psi(t+\delta t, x) = e^{-iV\delta t/2}\mathrm{FFT}^{-1}(e^{-iP\delta t}\mathrm{FFT}(e^{-iV\delta t/2}\psi(t,x))). \tag{47}$$

Vediamo ora quindi il codice:

```python
"""
Code for the solution of Schrodinger's time dependent equation
via split operator method but unlike tunnel_barrier.py using a FFT.
Now the idea is tu use:
exp(1j (T+V) dt) = exp(1j V dt/2) exp(1j T dt) exp(1j V dt/2) + O(dt^3)
and to compute exp(1j T dt) we go in momentum space where T is diagonal
so it easy to compute, so we must use FFT to go from x space to p space
"""
```

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

#==========================================================
# Initial wave function ad potential
#==========================================================

def U(x):
    ''' harmonic potential
    '''
    return 0.5*x**2

def psi_inc(x, x0, a, k):
    ''' Initial wave function
    '''

    A = 1. / np.sqrt( 2 * np.pi * a**2 ) # normalizzation
    K1 = np.exp( - ( x - x0 )**2 / ( 2. * a**2 ) )
    K2 = np.exp( 1j * k * x )
    # let's multiply by five so the animation is prettier
    return A * K1 * K2 * 5

#==========================================================
# Computational parameters
#==========================================================

n  = 1000                       # Number of points
xr = 10                         # Right boundary
xl = -xr                        # Left boundary
L  = xr - xl                    # Size of box
x  = np.linspace(xl, xr, n)     # Grid on x axis
dx = np.diff(x)[0]              # Step size
dt = 1e-3                       # Time step
T  = 10                         # Total time of simulation
ts = int(T/dt)                  # Number of step in time

# Initializzation of gaussian wave packet
psi = psi_inc(x, -1.2, 0.5, 0.3)
PSI = []
PSI.append(abs(psi)**2)

#==========================================================
# Build the propagator in x and k space
#==========================================================

# Every possible value of momentum
k    = 2*np.pi*np.fft.fftfreq(n, dx)
# Propagator
U_r = np.exp(-1j * U(x) * dt/2)   # Half step in space
U_k = np.exp(-1j * k**2/2 * dt)   # Full step in momentum

# Time evolution
for _ in range(ts):
    psi = U_r * psi

    psi_k = np.fft.fft(psi)
    psi_k = U_k * psi_k

    psi = np.fft.ifft(psi_k)
    psi = U_r * psi

    PSI.append(abs(psi)**2)

#==========================================================
# Animation
#==========================================================

fig = plt.figure()
plt.title("Gaussian packet propagation")
plt.plot(x, U(x), label='$V(x)$', color='black')
plt.grid()

plt.ylim(-0.0, np.max(PSI))
plt.xlim(-5, 5)

```

```
line, = plt.plot([], [], 'b', label=r"$|\psi(x, t)|^2$")

def animate(i):
    line.set_data(x, PSI[i])
    return line,

plt.legend(loc='best')

anim = animation.FuncAnimation(fig, animate, frames=np.arange(0, ts, 10),
                               interval=1, blit=True, repeat=True)

#anim.save('ho.mp4', fps=30, extra_args=['-vcodec', 'libx264'])

plt.show()
```

In questo caso abbiamo nuovamente usato un potenziale armonico e possiamo vedere l'oscillazione della funzione d'onda nel nostro potenziale. Adesso ho due domande per voi: se rendete il pacchetto gaussiano iniziale più stretto la simulazione non viene bene; perché? Provate inoltre a fare l'evoluzione nel tempo immaginario $it = \beta$ (capisco sia un tempo ma concedetemi di chiamarlo β); ora perdiamo l'unitarietà quindi ad ogni passo bisogna normalizzare la psi. Cosa succede al nostro sistema? come pensate che evolva?

11.2.4 Altre applicazioni

Anche se non le esponiamo è interessante dire come la fft sia molto utile anche nell'analisi dati. Ad esempio se voglia filtrare un segnale si tratterebbe di fare una convoluzione tra segnale di input e il vostro filtro, ma le convoluzioni nello spazio della trasformata sono semplici prodotti. O magari anche capire se vi è o meno una componente debole ad una data frequenza nel vostro segnale: prendete ad esempio il segnale con cui abbiamo fatto sopra i test; se gli aggiungete del rumore, la componente a 40 Hz nello spetto magari non si vede. Ma se il segnale è a media zero vedrete che vi è un picco in negativo molto importante a frequenza nulla nella trasformata; perciò vi basta moltiplicare tutto il segna per, un seno ad esempio, alla stessa frequenza della componente che cercate; ciò creerà un battimento e una delle due omega sarà nulla. Per cui nella FFT il profondo picco negati sarà notevolmente ridotto, e ciò vi fa quindi capire la presenza di quell'armonica. Non so quanto sono stato chiaro, magari più il là fornirò degli esempi di codice. Intanto magari potete provare a scriverli da voi.