

# **Computational Physics (PHYS6350)**

Lecture 22: Fourier transform

- Discrete Fourier transform (DFT)
- Fast Fourier Transform (FFT)

#### Instructor: Volodymyr Vovchenko (vvovchenko@uh.edu)

**Course materials:** <u>https://github.com/vlvovch/PHYS6350-ComputationalPhysics</u> **Online textbook:** <u>https://vovchenko.net/computational-physics/</u>

#### **Fourier transform**

Periodic functions (e.g. over  $x \in [0,L]$ ) permit Fourier decomposition

$$f(x) = \sum_{k=-\infty}^{\infty} \gamma_k \exp\left(i\frac{2\pi kx}{L}\right)$$

If a function is not periodic, it can always be forced to be periodic over by creating copies of its  $x \in [0,L]$  form

The Fourier coefficients read

$$\gamma_k = \frac{1}{L} \int_0^L f(x) \exp\left(-i\frac{2\pi kx}{L}\right) dx$$

Applications:

- signal processing (e.g. frequencies)
- image compression
- solutions to partial differential equations
- convolutions of functions



If function is even (symmetric) around the midpoint x = L/2, it permits cosine series

$$f(x) = \sum_{k=0}^{\infty} \alpha_k \cos\left(\frac{2\pi kx}{L}\right) \qquad \qquad \alpha_k = \frac{2-\delta_{k0}}{L} \int_0^L f(x) \cos\left(\frac{2\pi kx}{L}\right) dx$$

If function is odd (antisymmetric) around the midpoint x = L/2, it permits sine series

$$f(x) = \sum_{k=1}^{\infty} \beta_k \sin\left(\frac{2\pi kx}{L}\right) \qquad \qquad \beta_k = \frac{2}{L} \int_0^L f(x) \sin\left(\frac{2\pi kx}{L}\right) dx$$

The coefficients are related to the ones of the exponential series

$$\gamma_k = \frac{1 + \delta_{k0}}{2} \left[ \alpha_{|k|} - i \operatorname{sign}(k) \beta_{|k|} \right]$$

#### **Evaluating Fourier coefficients**

$$\gamma_k = \frac{1}{L} \int_0^L f(x) \exp\left(-i\frac{2\pi kx}{L}\right) dx.$$

Apply N-point trapezoidal rule

$$\gamma_k \simeq \frac{1}{L} \frac{L}{N} \left[ \frac{f(0)}{2} + \frac{f(L)}{2} + \sum_{n=1}^{N-1} f(x_n) \exp\left(-i\frac{2\pi k x_n}{L}\right) \right] \qquad h = L/N \qquad x_n = hn$$

The function is periodic, f(0) = f(L), thus

$$\gamma_k = \frac{1}{N} \sum_{n=0}^{N-1} y_n \exp\left(-i\frac{2\pi kn}{N}\right) \qquad \qquad y_n \equiv f(x_n).$$

This is the **discrete Fourier transform (DFT)**.

Typically, one uses the coefficients without the factor 1/N, i.e.

If  $y_n$  are all real,

$$c_k = \sum_{n=0}^{N-1} y_n \exp\left(-i\frac{2\pi kn}{N}\right) \qquad \qquad c_{N-k} = c_k^*$$

### **Evaluating Fourier coefficients**

```
def dft(y):
    N = len(y)
    c = np.zeros(N, complex)
    for k in range(N):
        for n in range(N):
            c[k] += y[n] * np.exp(-2j * np.pi * k * n / N)
    return c
```

Function





Fourier transform

N = 200
y = np.random.rand(N)
#print("y = ", y)
plt.plot(y)
plt.xlim(0,N)
plt.show()

```
c = dft(y)
#print("c = ", c)
plt.plot(np.abs(c))
plt.xlim(0,N)
plt.show()
```

#### **Evaluating Fourier coefficients**

Consider a wavelike form with noise

```
## Data from http://www-personal.umich.edu/~mejn/cp/programs/dft.py
y = np.loadtxt("pitch.txt",float)
plt.plot(y)
plt.xlim(0,len(y))
plt.show()
c = dft(y)
plt.plot(np.abs(c))
plt.xlim(0,len(y))
```

plt.show()





#### **Inverse Fourier transform**

Consider the following geometric progression

$$\sum_{k=0}^{N-1} e^{i2\pi km/N} = \frac{1-e^{i2\pi m}}{1-e^{i2\pi m/N}} = \delta_{m0}.$$

We use this now to evaluate the following sum

$$\begin{split} \sum_{k=0}^{N-1} c_k \exp\left(i\frac{2\pi kn}{N}\right) &= \sum_{k=0}^{N-1} \sum_{k'=0}^{N-1} y_{k'} \exp\left(-i\frac{2\pi k'k}{N}\right) \exp\left(i\frac{2\pi kn}{N}\right) \\ &= \sum_{k=0}^{N-1} \sum_{k'=0}^{N-1} y_{k'} \exp\left(i\frac{2\pi (n-k')k}{N}\right) \\ &= \sum_{k'=0}^{N-1} y_{k'} \delta_{n-k',0} \\ &= N y_n. \end{split}$$

$$y_n = \frac{1}{N} \sum_{k=0}^{N-1} c_k \exp\left(i\frac{2\pi kn}{N}\right),$$

inverse discrete Fourier transform (inverse DFT)

#### **Inverse DFT**

```
def inverse_dft(c):
    N = len(c)
    y = np.zeros(N, complex)
    for k in range(N):
        for n in range(N):
            y[n] += c[k] * np.exp(2j * np.pi * k * n / N)
    return y / N
```

```
N = 20
y = np.random.rand(N)
print("y = ", y)
c = dft(y)
print("c = ", c)
yinv = inverse_dft(c)
print("y = ", yinv)
```

```
y = [0.86046716 0.48175562 0.06576607 0.73962908 0.61970857 0.26800379
0.82606492 0.47751752 0.56001126 0.25493145 0.01241648 0.6918537
0.81113769 0.27735761 0.09617065 0.17598327 0.27550172 0.93691544
0.50237258 0.1342621 ]
```

```
c = [9.06782667+0.0000000e+00] 0.41721559-6.84001863e-01j
 0.0708089 +3.20029273e-01j 0.04707852+1.44889560e+00j
-1.17455143-1.23742486e+00j 1.6240357 +2.81762241e-04j
 0.9057288 -7.21848714e-01j 2.18903831-1.12193177e+00j
-0.06718515-4.86177561e-01j -0.03711474+5.50581297e-01j
 0.19140753+4.00264145e-15j -0.03711474-5.50581297e-01j
-0.06718515+4.86177561e-01j 2.18903831+1.12193177e+00j
 0.9057288 +7.21848714e-01i 1.6240357 -2.81762241e-04i
-1.17455143+1.23742486e+00j 0.04707852-1.44889560e+00j
 0.0708089 -3.20029273e-01j 0.41721559+6.84001863e-01j
y = [0.86046716 - 4.10782519e - 16] 0.48175562 + 6.66133815e - 16]
0.06576607-1.69309011e-16j 0.73962908-2.47649123e-15j
0.61970857-1.02834408e-15j 0.26800379-1.90680804e-15j
0.82606492+3.33066907e-17j 0.47751752-2.77555756e-17j
0.56001126+5.27355937e-16j 0.25493145-2.27595720e-16j
0.01241648+1.91513472e-15j 0.6918537 -7.04991621e-16j
0.81113769+8.63198402e-16j 0.27735761+1.77913240e-15j
0.09617065+2.32869279e-15j 0.17598327+7.63278329e-16j
0.27550172+2.06501483e-15j 0.93691544+1.67643677e-15j
0.50237258-7.99360578e-16j 0.1342621 -1.44884105e-15j]
```

Recover the original function up to round-off error



#### **Discrete cosine and sine transforms**

Mirror the function over  $x \in [0, L/2]$  and make it (anti)symmetric around x = L/2. Then apply the standard Fourier transform



With some optimizations one can avoid using complex numbers

```
Straight DFT complexity: O(N^2)
```

```
def dft(y):
    N = len(y)
    c = np.zeros(N, complex)
    for k in range(N):
        for n in range(N):
            c[k] += y[n] * np.exp(-2j * np.pi * k * n / N)
    return c
```

Impractical for large data sets

Can we do better?

Fast Fourier Transform (FFT) algorithms achieve O(N log N)

#### Fast Fourier Transform: Cooley-Tukey algorithm (1965)

To see how it works, let us consider a case where N is a power of two,  $N = 2^{M}$ , and we want to compute the Fourier transform of  $(y_0, y_1, ..., y_{N-1})$ . By definition we have:

$$c_k = \sum_{n=0}^{N-1} y_n \exp\left(-i\frac{2\pi kn}{N}\right).$$

We can split the sum into even and odd elements:

$$c_{k} = \sum_{n=0}^{N/2-1} y_{2n} \exp\left(-i\frac{2\pi k(2n)}{N}\right) + \sum_{n=0}^{N/2-1} y_{2n+1} \exp\left(-i\frac{2\pi k(2n+1)}{N}\right)$$
$$= \sum_{n=0}^{N/2-1} y_{2n} \exp\left(-i\frac{2\pi kn}{N/2}\right) + \exp\left(-i\frac{2\pi k}{N}\right) \sum_{n=0}^{N/2-1} y_{2n+1} \exp\left(-i\frac{2\pi kn}{N/2}\right)$$
$$= E_{k} + \exp\left(-i\frac{2\pi k}{N}\right)O_{k}.$$

 $c_k$  can be expressed as a sum of two elements:

- one is the kth element from a DFT of all even elements:  $(y_0, y_2, ..., y_{N-2})$ ,
- the other is the kth element from a DFT of all odd elements:  $(y_1, y_3, ..., y_{N-1})$

## Fast Fourier Transform: Cooley-Tukey algorithm (1965)

The interpretation makes sense if k < N/2

Other Fourier components can be expressed as k + N/2 and read:

$$c_{k+N/2} = \sum_{n=0}^{N/2-1} y_{2n} \exp\left(-i\frac{2\pi(k+N/2)n}{N/2}\right) + \exp\left(-i\frac{2\pi(k+N/2)}{N}\right) \sum_{n=0}^{N/2-1} y_{2n+1} \exp\left(-i\frac{2\pi(k+N/2)n}{N/2}\right)$$
$$= E_k - \exp\left(-i\frac{2\pi k}{N}\right) O_k.$$

Combine with

$$c_k = E_k + \exp\left(-i\frac{2\pi k}{N}\right)O_k.$$

To compute DFT of  $(y_0, ..., y_N)$  we only need to compute <u>two</u> N/2 DFTs of even and odd components of y.

Divide and conquer: continue recursively until N = 1, where  $c_k = y_k$ 

At each step N is halved



Complexity: O(N log N)

### **FFT** implementation

```
# Compute DFT of (y st, y st+s, y st+2s, ..., y st+(N-1)s)
def fft recursive(y, st, N, s):
    if (N == 1):
        return np.array([y[st]])
    else:
        c = np.empty(N, complex)
        c1 = fft_recursive(y, st, N//2, 2*s)
        c2 = fft recursive(y, st + s, N//2, 2*s)
        for k in range(N//2):
            p = c1[k]
            q = np.exp(-2j*np.pi*k/N) * c2[k]
            c[k] = p + q
            c[k + N//2] = p - q
        return c
\# N = len(y) must be a power of 2
def fft(y):
    N = len(y)
   return FFT recursive(y, 0, N, 1)
```

#### %%time

# Try naive DFT
cdft = dft(y)

CPU times: user 2.02 s, sys: 26.3 ms, total: 2.05 s Wall time: 1.61 s

%%time

```
# Now compare to FFT
cfft = fft(y)
```

```
print(cdft - cfft)
```

```
[-7.95807864e-13+0.0000000e+00j 4.66293670e-14+6.43929354e-14j
-2.57571742e-14+4.08562073e-14j ... 1.42890144e-11-1.61026747e-12j
2.72741829e-11+5.71453995e-12j 1.94777527e-12-1.27142741e-12j]
CPU times: user 5.85 ms, sys: 86 μs, total: 5.93 ms
Wall time: 5.88 ms
```

### **FFT** for the signal

#### %%time

```
print("N = ",len(y))
y = np.loadtxt("pitch.txt",float)
plt.plot(y)
plt.xlim(0,len(y))
plt.show()
```

```
c = fft(y)
plt.plot(np.abs(c))
plt.xlim(0,len(y))
plt.show()
```

N = 1024

CPU times: user 307 ms, sys: 9.85 ms, total: 317 ms Wall time: 121 ms



### **FFT** for the signal

#### %%time

```
print("N = ",len(y))
y = np.loadtxt("pitch.txt",float)
plt.plot(y)
plt.xlim(0,len(y))
plt.show()
```

```
c = fft(y)
plt.plot(np.abs(c))
plt.xlim(0,len(y))
plt.show()
```

N = 1024

CPU times: user 307 ms, sys: 9.85 ms, total: 317 ms Wall time: 121 ms

```
numpy: np.fft.fft(y)
```

CPU times: user 299 ms, sys: 10.1 ms, total: 309 ms Wall time: 118 ms

