maponti / imageprocessing_course_icmc Public

Code Issues 1 Pull requests Actions Projects Security Insights

master

imageprocessing_course_icmc / 04e_fourier_transform_fft.ipynb

maponti 4 - Fourier update History

1 contributor

472 lines (472 sloc) | 12.6 KB

04 - part 4 - Fast Fourier Transform

In [1]:
```
import numpy as np
import imageio
import matplotlib.pyplot as plt
import time
from cmath import exp, pi
```

The Fast Fourier Transform (FFT) is a *divide and conquer* algorithm that recursively splits the input array into two parts: one for the odd indices, and another for the even indices, until the trivial case is achieved.

It is important to note that complex exponentials (that can be decomposed into a sum of sine and cosine) are periodic and symmetric, and from those properties the FFT is defined.

Im particular, from $e^{-j\frac{2\pi}{N}xu}$, we isolate the constant term, and define it as a variable: $W = e^{-j\frac{2\pi}{N}}$. Note W is constant because it does not depend on the time sampling (controlled by x), nor depends on the frequencies (u).

For example, for a signal with 4 observations, i.e. $N = 4$:

In [4]:
```
N = 4
W = exp(-1j*(2*pi)/N)
print(W)
```

```
(6.123233995736766e-17-1j)
```

This value does not depend on u nor x. The two properties we are going to use to implement the FFT are:

1. periodicity in u,x: $W_N^{ux} = W_N^{u(N+x)} = W_N^{(u+N)x}$
2. symmetry of the complex conjugate: $W_N^{u(N-x)} = W_N^{-ux} = (W_N^{ux})^*$ for example this is easy to see for $x = N$, $W_N^{uN} = e^{-j2\pi u} = 1$

Now we define the **division** step of the algorithm. This is done by decomposing the transform into even and odd indices of x. To avoid a cluttered notation, let us express the transform in terms of the variable W:

$F(u) = \sum_{x=0}^{N-1} f(x) W_N^{ux}$

Now we write the function of evaluating the even indices $2x$ and the odd indices $2x + 1$:

$F(u) = \sum_{x=0}^{N/2-1} f(2x) W_N^{u(2x)} + \sum_{x=0}^{N/2-1} f(2x+1) W_N^{u(2x+1)}$

Note $2x$ forms the sequence 0, 2, 4, 6, while $2x + 1$ the sequence 1, 3, 5, 7 as we

Note $2x$ forms the sequence $0, 2, 4, 6$, while $2x+1$ the sequence $1, 3, 5, 7$ as we wanted, therefore:

$$F(u) = \sum_{x=0}^{N/2-1} f(2x) \cdot (W_N^2)^{ux} + \sum_{x=0}^{N/2-1} f(2x+1) \cdot (W_N^2)^{(2x+1)u}$$

Let us manipulate this sum, isolating the terms that are independent of x, which is the W_N^u:

$$F(u) = \sum_{x=0}^{N/2-1} f(2x) \cdot (W_N^2)^{ux} + W_N^u \cdot \sum_{x=0}^{N/2-1} f(2x+1) \cdot (W_N^2)^{ux}$$

But we not that $W_N^2 = e^{-j\frac{2\pi}{N}2} = e^{-j\frac{2\pi}{N/2}} = W_{N/2}$

and this is defines the 'trick', since it allows to write the tranform as:

$$F(u) = \sum_{x=0}^{N/2-1} f(2x) \cdot W_{N/2}^{ux} + W_N^u \cdot \sum_{x=0}^{N/2-1} f(2x+1) \cdot W_{N/2}^{ux}$$

The first term is the DFT of the $N/2$ elements corresponding to the even indices, the second term is the DFT of the $N/2$ elements related to the odd indices.

This way, we can split the DFT of N elements, in a recursive way, into two $N/2$ DFTs, and later combine the results:

$$F(u) = F_{\text{even}}(u) + W_N^u \cdot F_{\text{odd}}(u)$$

Recall the property of symmetry of the complex conjugate:

$$F(u + N/2) = F_{\text{even}}(u) - W_N^u \cdot F_{\text{odd}}(u)$$

In [5]:

```
# Let us try to grasp this idea for a small example
N = 4
f = [0,100,200,300]
f
```

Out[5]:

```
[0, 100, 200, 300]
```

In [6]:

```
# splitting the array into even and odd indices
f_even= f[0::2]
f_odd = f[1::2]
print(f_even)
print(f_odd)
```

```
[0, 200]
[100, 300]
```

In [7]:

```
# recursively, we split the resulting arrays, first the even, into even and
f_even_even = f_even[0::2]
f_even_odd = f_even[1::2]
print(f_even_even)
print(f_even_odd)
```

```
[0]
[200]
```

In this simple example, we partition the elements until we reach the base case, that is when there is only 1 even and 1 odd element, allowing to compute:

In [8]:
```
reseven0 = f_even_even[0] + exp(-2j*pi*0/N) * f_even_odd[0]
reseven1 = f_even_even[0] - exp(-2j*pi*0/N) * f_even_odd[0]
reseven = [reseven0, reseven1]
print(reseven)
```

```
[(200+0j), (-200+0j)]
```

Now this result is stored and we execute the other 'side' of the recursion, relative to the first odd indices

In [9]:
```
# separate the odd indices, into even and odd indicdes
f_odd_even = f_odd[0::2]
f_odd_odd = f_odd[1::2]
print(f_odd_even)
print(f_odd_odd)
```

```
[100]
[300]
```

In [10]:
```
resodd0 = f_odd_even[0] + exp(-2j*pi*0/N) * f_odd_odd[0]
resodd1 = f_odd_even[0] - exp(-2j*pi*0/N) * f_odd_odd[0]
resodd = [resodd0, resodd1]
print(resodd)
```

```
[(400+0j), (-200+0j)]
```

Now, combining the individual results (reseven and resodd):

In [11]:
```
# from the symmetric property, I can use the result 0 to also
# obtain the value for 0+N/2 = N/4 = 2, changing the signal
res0 = reseven[0] + exp(-2j*pi*0/N) * resodd[0]
res2 = reseven[0] - exp(-2j*pi*0/N) * resodd[0]

# similarly the result 1 is used to obtain the result of 1+N/2 = 1+N/4 = 3
res1 = reseven[1] + exp(-2j*pi*1/N) * resodd[1]
res3 = reseven[1] - exp(-2j*pi*1/N) * resodd[1]

# putting everything together
F_manual =  np.array([res0, res1, res2, res3]).astype(np.complex64)
print(F_manual)
```

```
[ 600.  +0.j -200.+200.j -200.  +0.j -200.-200.j]
```

Let us code a function for this algorithm

In [12]:
```
def FFT(f):
    N = len(f)
    if N <= 1:
        return f

    # division
```

```
    # division
    even= FFT(f[0::2])
    odd = FFT(f[1::2])

    # store combination of results
    temp = np.zeros(N).astype(np.complex64)

    # only required to compute for half the frequencies
    # since u+N/2 can be obtained from the symmetry property
    for u in range(N//2):
        temp[u] = even[u] + exp(-2j*pi*u/N) * odd[u] # conquer
        temp[u+N//2] = even[u] - exp(-2j*pi*u/N)*odd[u]  # conquer

    return temp
```

In [13]:

```
# testing the function to see if it matches the manual computation
F_fft = FFT(f)
print(F_fft)
```

```
[ 600.  +0.j -200.+200.j -200.  +0.j -200.-200.j]
```

In [14]:

```
# let us compare it with the DFT1D
def DFT1D(f):
    # create empty array of complex coefficients
    F = np.zeros(f.shape, dtype=np.complex64)
    n = f.shape[0]

    # creating indices for x, allowing to compute the multiplication using r
    x = np.arange(n)
    # for each frequency 'u', perform vectorial multiplication and sum
    for u in np.arange(n):
        F[u] = np.sum(f*np.exp( (-1j * 2 * np.pi * u*x) / n ))

    return F

F_dft = DFT1D(np.array(f))
print(F_dft)
```

```
[ 600.+0.000000e+00j -200.+2.000000e+02j -200.-7.347881e-14j
 -200.-2.000000e+02j]
```

In [15]:

```
# printing the 3 results
print(F_dft)
print(F_manual)
print(F_fft)
```

```
[ 600.+0.000000e+00j -200.+2.000000e+02j -200.-7.347881e-14j
 -200.-2.000000e+02j]
[ 600.  +0.j -200.+200.j -200.  +0.j -200.-200.j]
[ 600.  +0.j -200.+200.j -200.  +0.j -200.-200.j]
```

Due to the approximation, sometimes a small error is observed between the results.

Let us compare the running time of DFT and FFT:

In [20]:

```
# an array with 10000 elements
t = np.arange(0, 2, 0.0001)
```

```
f = 1*np.sin(t*(2*np.pi) * 2) + 0.6*np.cos(t*(2*np.pi) * 8) + 0.4*np.cos(t*(
print(t.shape)
```

(20000,)

In [21]:

```
start = time.time()
F_dft = DFT1D(np.array(f))
end = time.time()
elapsed = end - start
print("DFT Running time: " + str(elapsed) + " sec.")
```