improc - Fourier Transform

141 days ago by admin

```
from numpy import *
from pylab import
zeros,ones,diag,imshow,gray,imread,savefig,inv,arange
from scipy.linalg import norm
def normalize(v): return v/norm(v)
def max1(v): return v*1.0/amax(v)
def cshift(l, offset):
   offset %= len(l)
   return concatenate((l[-offset:], l[:-offset]))
```

Definition of the Discrete Fourier Transform

Here is the actual definition of the discrete Fourier transform (DFT):

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

Here we have used the identity:

$$e^{xj} = cos(x) + sin(x)j$$

That is, the output value X_k is a complex number whose real part contains $c_k \cdot x$ and whose complex part contains $s_k \cdot x$.

The way to think about this is that output X_k is the dot product of the input signal x and a basis vector b_k:

$$X_k = x \cdot b_k$$

Here, the basis functions b_k are orthogonal, that is $b_i \cdot b_j = \delta_{ij}$.

Let's implement that.

```
def b(k,n,N):
    return exp(-2*pi*1j*k*n/N)
N = 420
n = arange(0,420)
```

OK, so these b_k are, in fact, orthogonal:

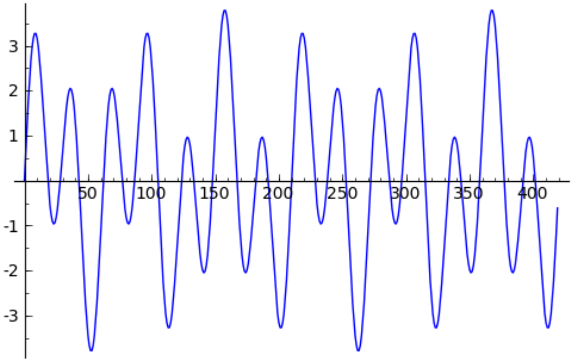
$$b_k \cdot b_{k'} = \delta_{k,k'}$$

```
dot(b(6,n,N),b(11,n,N))
```

```
(-9.2037488741425477e-14+2.1871393585115584e-14j)
```

Let's now construct a mixture of two signals of different frequencies. We will be using the Fourier transform to separate theses.

```
a1 = 1.5
a2 = 2.3
omega1 = 6*2*pi/N
omega2 = 14*2*pi/N
y = a1*sin(n*omega1) + a2*sin(n*omega2)
list_plot(y,1)
```



Let's just take the dot product of each of the basis functions with the mixed signal. In order to keep the output simple, we just look at the absolute value of the complex valued output, and we truncate the numbers.

```
[int(abs(dot(b(i,n,N),y))) for i in range(N)]
```

```
[0, 0, 0, 0, 0, 0, 315, 0, 0, 0, 0, 0, 0, 0, 483, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 482, 0, 0, 0, 0,
0, 0, 0, 314, 0, 0, 0, 0, 0]
```

This has entries in the right places (except for the values at the end; we'll return

to that).

As we did with other filters, let's now just put all these vectors together into a matrix. Note that this is a matrix with complex entries.

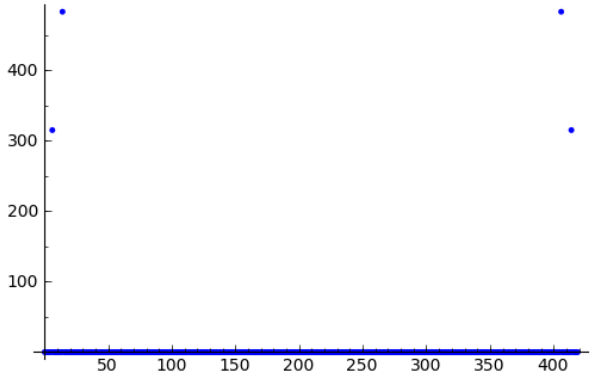
```
F = array([b(k,n,N) for k in range(0,N)])
```

Now we can analyze our signal y with this matrix.

```
Y = dot(F,y)
```

Let's look at the frequencies.

```
list_plot(abs(Y))
```




```
[i for i in range(N) if abs(Y[i])>100]
```

```
[6, 14, 406, 414]
```

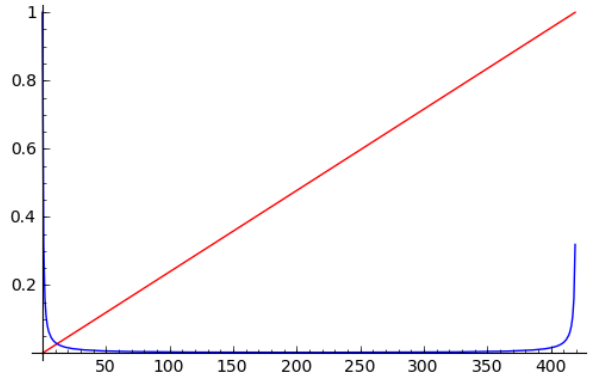
So, we recovered the original frequencies, but we also got two extra entries at 406 and 414; these actually correspond to "negative frequencies".

The reason we get them is because the signal is real. That is, we put in N values and get out N real coefficients plus N imaginary coefficients. The symmetry reduces these $2N$ numbers back down to N numbers.

Fourier Transforms of Common Signals

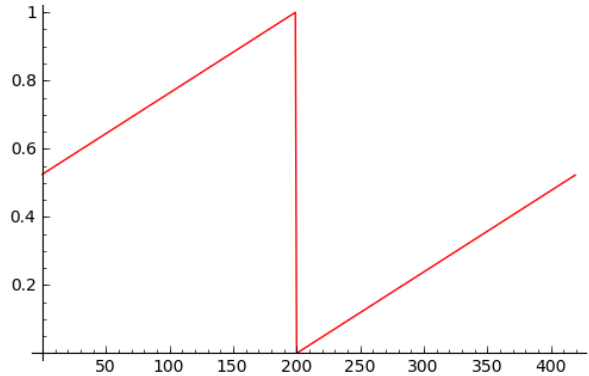
Let's look at another few signals.

```
signal = n
transform = dot(F,signal)
list_plot(max1(signal),1,rgbcolor="red")+list_plot(max1(abs(transform)),1
```



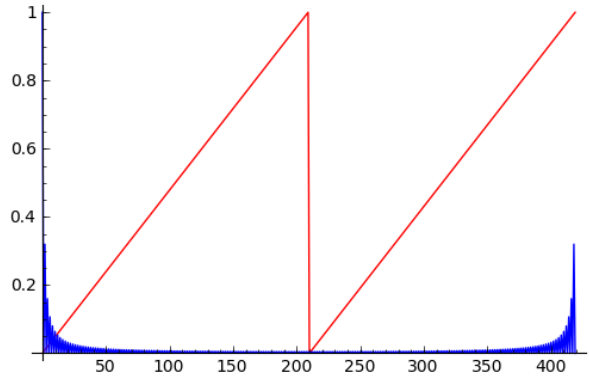
Remember that the Fourier transform implements convolutions with circular boundary conditions. Therefore, the signal that the Fourier transform sees isn't the nice-looking smooth signal above, but one with a sharp discontinuity. We can see this by shifting the signal circularly.

```
list_plot(roll(max1(signal),200),1,rgbcolor="red")
```



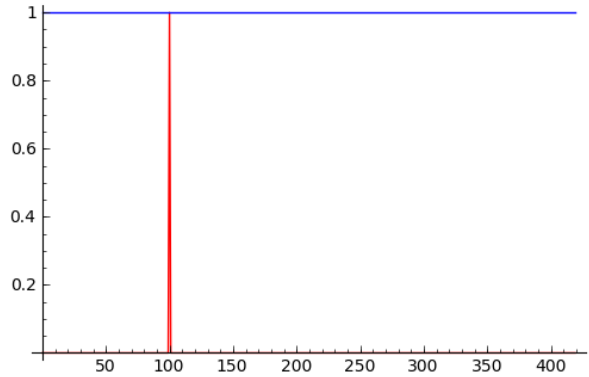
A double sawtooth illustrates this as well and is similar.

```
signal = concatenate([n,n])[2*n]
transform = dot(F,signal)
list_plot(max1(signal),1,rgbcolor="red")+list_plot(max1(abs(transform)),1
```



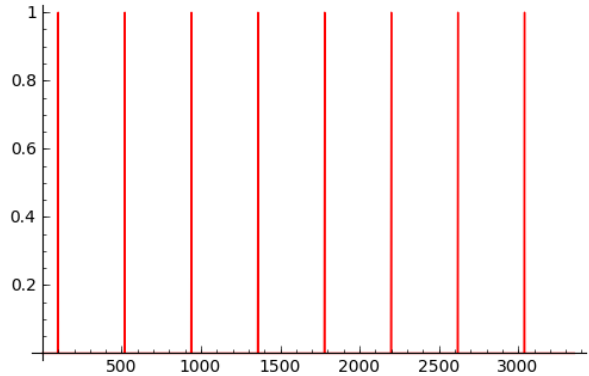
The Fourier transform of an impulse is a flat spectrum.

```
signal = (n==100)
transform = dot(F,signal)
list_plot(max1(signal),1,rgbcolor="red")+list_plot(max1(abs(transform)),1
```



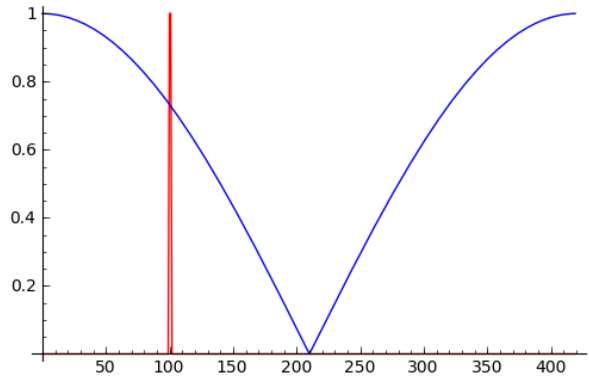
Again, we're actually looking at a circular convolution, so what the Fourier transform "sees" is this.

```
list_plot(max1(concatenate([signal for i in
range(8)])),1,rgbcolor="red")
```

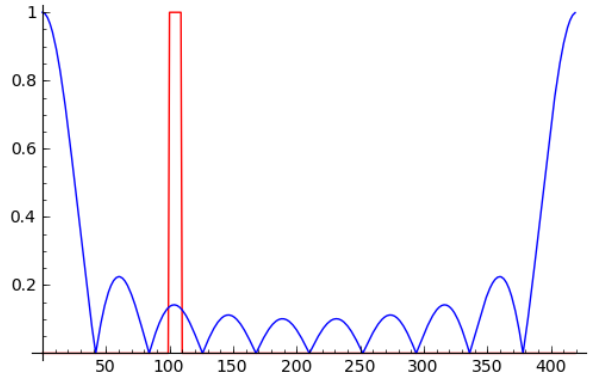


If the signal isn't a perfect impulse but a little broader, the transform changes significantly.

```
signal = (n>=100) * (n<102)
transform = dot(F,signal)
list_plot(max1(signal),1,rgbcolor="red")+list_plot(max1(abs(transform)),1
```

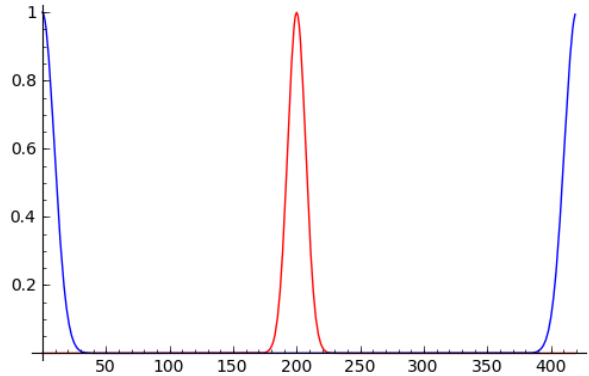



```
signal = (n>=100) * (n<110)
transform = dot(F,signal)
list_plot(max1(signal),1,rgbcolor="red")+list_plot(max1(abs(transform)),1
```



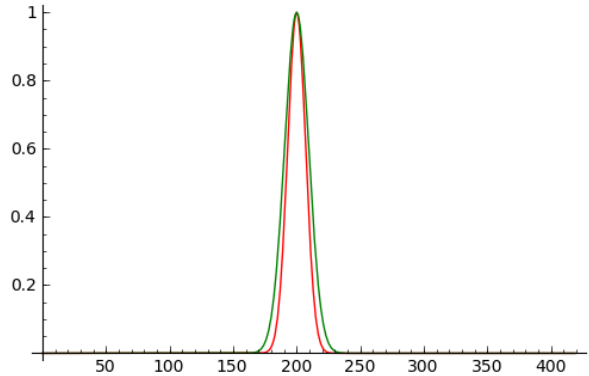
The Fourier transform of the Gaussian is very important in practice.

```
signal = exp(-(n-200)**2/100.0)
signal /= amax(signal)
transform = dot(F,signal)
list_plot(max1(signal),1,rgbcolor="red")+list_plot(max1(abs(transform)),1
```



It's kind of difficult to see what's going on here. It becomes easier if we shift the output of the transform to overlap with the original signal. Then it becomes pretty clear that the Fourier transform of a Gaussian is again a Gaussian. Some experimentation shows that the wider the original Gaussian, the narrower its transform and vice versa.

```
list_plot(max1(signal),1,rgbcolor="red")+list_plot(cshift(max1(abs(transf
```



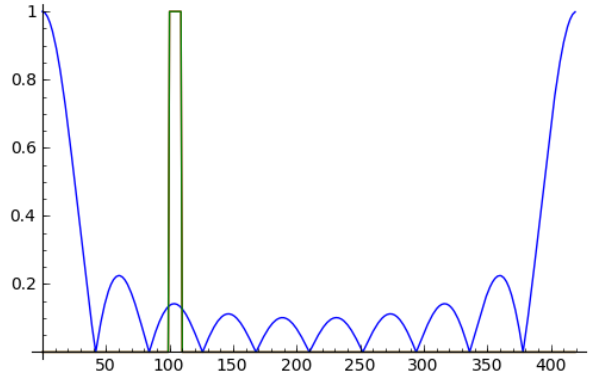
Inverse Fourier Transform

Of course, we can transform back as well.

F is a matrix, so we just take its inverse.

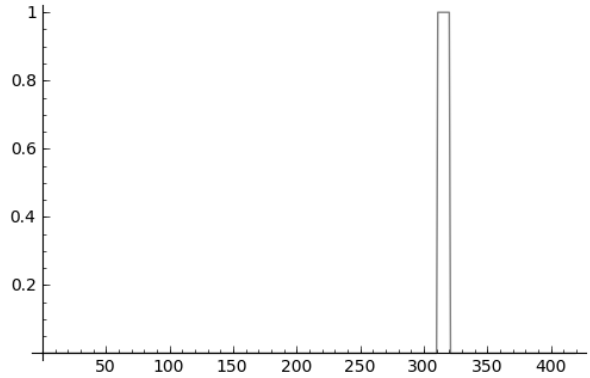
```
signal = (n>=100) * (n<110)
transform = dot(F,signal)
reconstructed = dot(inv(F),transform)

list_plot(max1(signal),1,rgbcolor="red")+list_plot(max1(abs(transform)),1
```



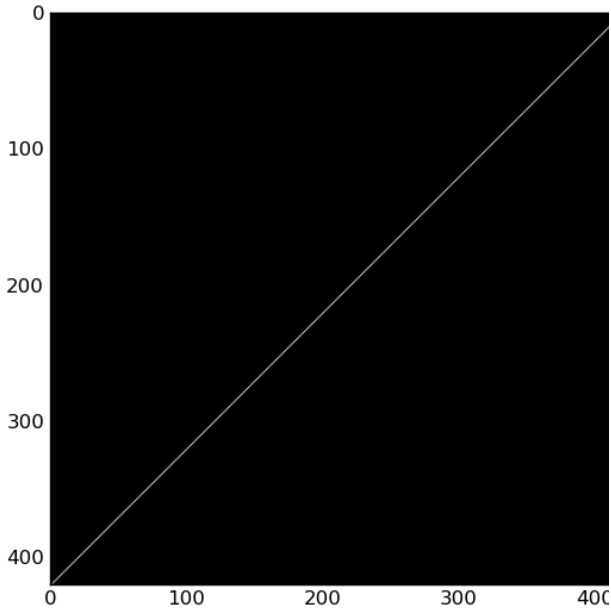
What happens if we apply the Fourier transform matrix F twice?

```
double = dot(F,transform)
list_plot(max1(double),1,rgbcolor="gray")
```




```
imshow(abs(dot(F,F))); gray(); savefig("temp")
```

```
<matplotlib.image.AxesImage object at 0xbde968c>
```



So, from $F \cdot F$, we see that the Fourier transform is almost its own inverse, except for a reflection.

The actual formula for the inverse discrete Fourier transform (IDFT) is

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

Note the sign change and the normalization factor. Other than that, the formula is the same as for the regular disrete Fourier transform.

Properties of the Fourier Transform

The Fourier transform has a number of important properties; you should know these by heart.

The Fourier transform is linear:

$$\mathsf{F}[af + bg] = a\mathsf{F}[a] + b\mathsf{F}[b]$$

The Fourier transform turns translations into multiplications with a complex number (and the other way around):

$$\mathsf{F}[f(x - x_0)](\omega) = e^{-2\pi i x_0 \omega}\mathsf{F}[f(x)](\omega)$$

The Fourier transform computes inverse scaling:

$$\mathsf{F}[f(ax)](\omega) = \frac{1}{|a|}\mathsf{F}[f(x)](\frac{\omega}{a})$$

The Fourier transform turns convolutions into multiplication:

$$\mathsf{F}[f * g] = \mathsf{F}[f] \cdot \mathsf{F}[g]$$