Day 17 Trigonometric interpolation, cont'd

We have seen that the function

$$p(x) \equiv \sum_{j=0}^{n-1} c_j e^{ijx} \quad \text{with} \quad c_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k e^{-ijx_k} \quad *$$

interpolates the data $\{(x_k, y_k)\}_{k=0,1,\ldots,n-1}$.

That is $y_k = \sum_{j=0}^{n-1} c_j e^{ijx_k}$. Recall $x_k = \frac{2\pi k}{n}$.

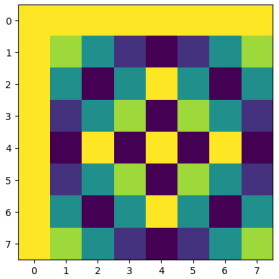
Let's explore the relationship between the data vector y and the coefficient vector c.

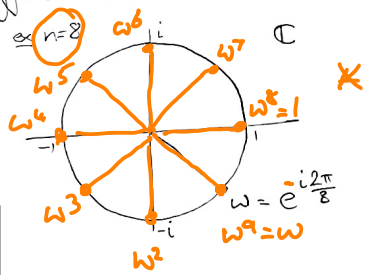
It's a linear mapping, called the Discrete Fourier Transform.

Define $\omega = e^{-i\frac{2\pi}{n}}$. Then $c_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k e^{-ij\frac{2\pi k}{n}} = \frac{1}{n}\sum_{k=0}^{n-1} \omega^{jk} y_k$. *

Thus $c = \frac{1}{n} W y$ where the j,k element of matrix W is $W_{jk} = \omega^{jk}$.

$$W = \begin{matrix} & k=0 & k=1 & k=2 & k=3 & \cdots & k=n-1 \\ j=0 & \omega^0 & \omega^0 & \omega^0 & \omega^0 & \cdots & \omega^0 \\ j=1 & \omega^0 & \omega^1 & \omega^2 & \omega^3 & \cdots & \omega^{n-1} \\ j=2 & \omega^0 & \omega^2 & \omega^4 & \omega^6 & \cdots & \\ j=3 & \omega^0 & \omega^3 & \omega^6 & & \cdots & \\ & \vdots & & & & & \\ & \omega^0 & \omega^{n-1} & \omega^{2(n-1)} & & \cdots & \end{matrix}$$






```
n = 8
w = np.exp(-1j*2*np.pi/n)
A = np.empty((n,n),dtype=complex)
v = [w**j for j in range(n)]
A[:,0] = 1
for j in range(1,n):
    A[:,j] = A[:,j-1]*v
A /= n
plt.imshow(np.real(A));
```

Note W is symmetric.

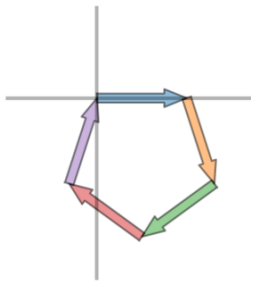
From above, we also have $y_k = \sum_{j=0}^{n-1} e^{ij\frac{2\pi k}{n}} c_j = \sum_{j=0}^{n-1} \overline{W}_{jk} c_j$ (complex conjugate).

So $y = \overline{W}^T c = \overline{W} c$.

That is, $\frac{1}{n} W^{-1} = \overline{W}$. Nice!

The orthogonality of the columns of W can also be seen directly: $\overline{\omega} = e^{+i\frac{2\pi}{n}} = \omega^{-1}$

$$(W\overline{W})_{jk} = \begin{bmatrix} 1 & \omega^j & \omega^{2j} & \omega^{3j} & \cdots \end{bmatrix} \begin{bmatrix} 1 \\ \omega^{-k} \\ \omega^{-2k} \\ \omega^{-3k} \\ \vdots \end{bmatrix} = 1 + \omega^{(j-k)} + \omega^{2(j-k)} + \omega^{3(j-k)} + \cdots$$



$$= \begin{cases} j=k: & 1+1+\cdots+1 = n \\ j\neq k: & 1+\gamma+\gamma^2+\cdots+\gamma^{n-1} = \frac{1-\gamma^n}{1-\gamma} = \frac{0}{\neq 0} \end{cases} \quad \text{since } \gamma^n = e^{-i2\pi(j-k)\frac{n}{n}} = 1.$$

$$= n\,\delta_{jk}.$$

Let us use F to denote the Fourier transform" (Discrete): DFT.

$$c = Fy \quad , \quad y = F^{-1}c$$

Definitions vary in normalization:

- $F_{Ackleh} = \frac{1}{n}W \Rightarrow F_{Ackleh}^{-1} = \overline{W}$
- $F_U = \frac{1}{\sqrt{n}}W \Rightarrow F_U^{-1} = \frac{1}{\sqrt{n}}\overline{W}$

  ↖ for unitary

  $\overline{F_U}^T F_U = I$

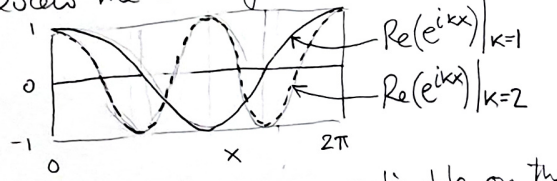
  This version preserves 2-norm:

$$(c,c) = (F_U y, F_U y) = \overline{(F_U y)}^T F_U y$$
$$= \bar{y}^T \overline{F_U}^T F_U y = \bar{y}^T I y = \bar{y}^T y = (y,y)$$

- $F_{Numpy} = W \Rightarrow F_{Numpy}^{-1} = \frac{1}{n}\overline{W}$

The basis functions used in the interpolant p(x) of Thm 4.18

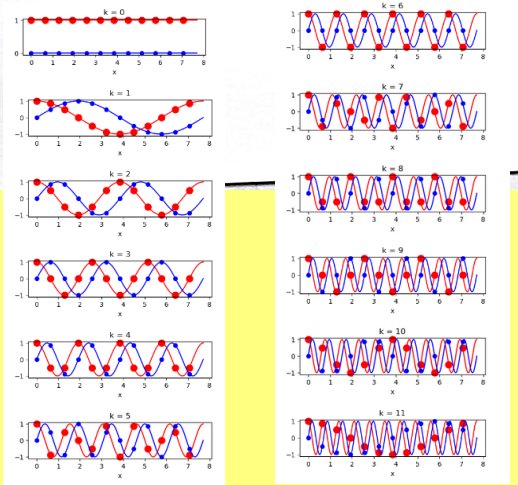
Review the basis functions from which our interpolant is built:



The highest frequency realizable on the grid is $k = \frac{n}{2}$, "Nyquist frequency":

But in p(x) we are using e^{ikx} with $k > \frac{n}{2}$. ☹

See plots.



All the codes in this document are included as text so you can copy and paste them into a Jupyter notebook or plain text .py file.

```
n = 12
L = 7.7  # arbitrary right endpoint of interval
x  = np.linspace(0,L,n,endpoint=False) # grid points
xx = np.linspace(0,L,400)  # quasi-continuum
twopi = 2*np.pi
I = 1j
for k in range(0,n):  # first let k range from 0 to n-1
    plt.figure(figsize=(5,1))
    y  = np.exp(twopi*I*k*x /L)
    yy = np.exp(twopi*I*k*xx/L)
    plt.plot(x ,np.real(y),'ro',markersize=10,clip_on=False)  # real part red
    plt.plot(xx,np.real(yy),'r')
    plt.plot(x ,np.imag(y),'bo',clip_on=False)  # imaginary part blue
    plt.plot(xx,np.imag(yy ),'b')
    plt.xlabel('x')
    plt.title(f'k = {k}')
```

We can do something more sensible by observing

$$e^{i(k-n)x}\Big|_{x=x_j} = e^{i(k-n)2\pi\frac{j}{n}} = \underbrace{e^{i2\pi j}}_{=1} e^{ik2\pi\frac{j}{n}} = e^{ikx}\Big|_{x=x_j}$$

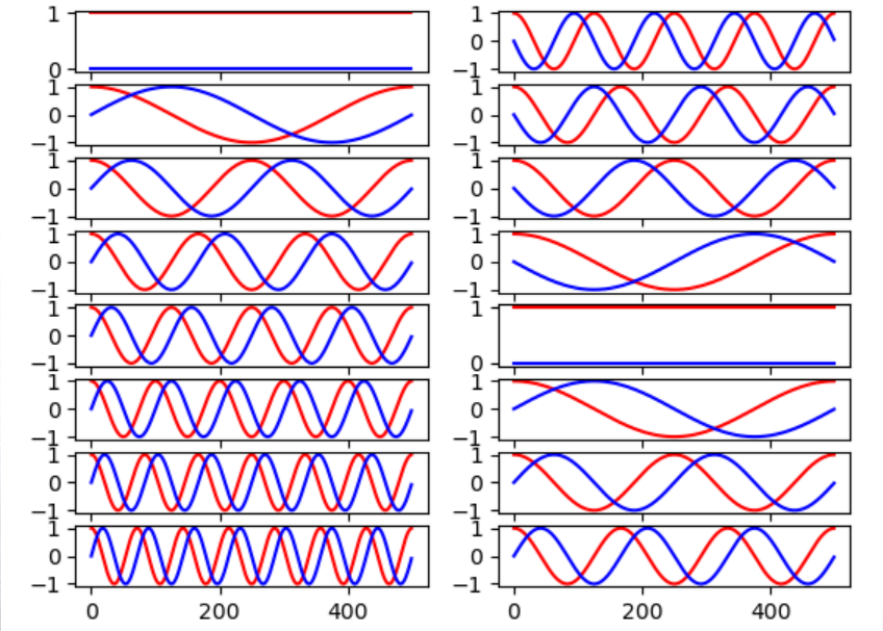
That is, subtracting n from the wavenumber k does not change the values of e^{ikx} at the grid points.

So for $k \geq \frac{n}{2}$, why not use $e^{i(k-n)x}$ instead of e^{ikx} in expression for $p(x)$: it still interpolates.

And with k running from $-\frac{n}{2}$ to $\frac{n}{2}-1$, we've eliminated the spurious high frequencies from $p(x)$.

Lets take a look ...

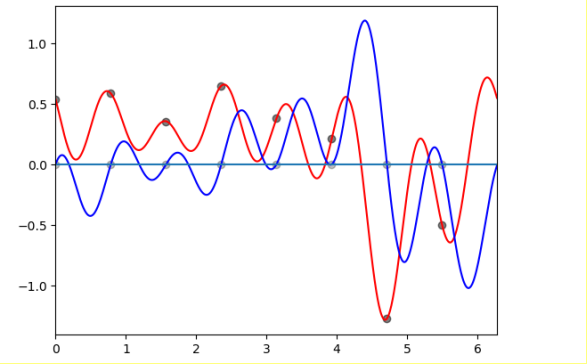
```
# previous basis on left, alternate basis on right
```



```
from nsm import *
n = 8
x = np.linspace(0,2*np.pi,500,endpoint=False)
w = np.exp(1j*x)
for j in range(n):
    plt.subplot(n,2,2*j+1)
    plt.plot(np.real(w**j),'r')
    plt.plot(np.imag(w**j),'b')

for j in range(-n//2,n//2):
    plt.subplot(n,2,2*j+n+2)
    plt.plot(np.real(w**j),'r')
    plt.plot(np.imag(w**j),'b')
```

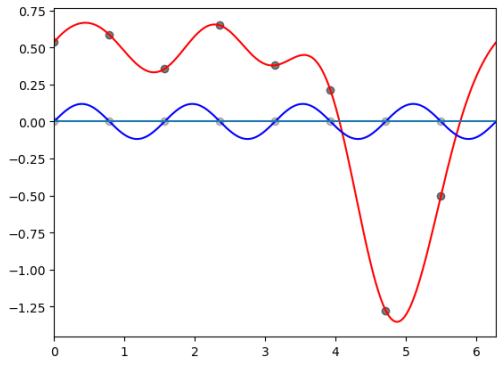
p(x) of Thm 4.18 has unjustifiable oscillations at high frequenc

```
np.random.seed(223)
y = np.random.randn(n)
x = np.linspace(0,2*np.pi,n,endpoint=False)
plt.plot(x,y,'ko',alpha=0.5)
plt.plot(x,y*0,'ko',alpha=0.25)
nn = 1000
xx = np.linspace(0,2*np.pi,nn,endpoint=False)
ww = np.exp(1j*xx)
twopi = 2*np.pi
I = 1j
w = np.exp(-I*2*np.pi/n)
A = np.empty((n,n),dtype=complex)
v = [w**j for j in range(n)]
A[:,0] = 1
for j in range(1,n):
    A[:,j] = A[:,j-1]*v
A /= n
c = A@y
#print(c)
p = np.zeros(nn,dtype=complex)
for j in range(n):
    #print(c[j])
    p += c[j]*ww**j
plt.plot(xx,np.real(p),'r')
plt.plot(xx,np.imag(p),'b')
plt.axhline(0)
plt.xlim(0,2*np.pi);
```



Use k range $-\frac{n}{2}, \ldots, \frac{n}{2} - 1$ instead

```
np.random.seed(223)
y = np.random.randn(n)
x = np.linspace(0,2*np.pi,n,endpoint=False)
plt.plot(x,y,'ko',alpha=0.5)
plt.plot(x,y*0,'ko',alpha=0.25)
nn = 1000
xx = np.linspace(0,2*np.pi,nn,endpoint=False)
ww = np.exp(1j*xx)
twopi = 2*np.pi
I = 1j
w = np.exp(-I*2*np.pi/n)
A = np.empty((n,n),dtype=complex)
v = [w**j for j in range(n)]
A[:,0] = 1
for j in range(1,n):
    A[:,j] = A[:,j-1]*v
A /= n
c = A@y
#print(c)
p = np.zeros(nn,dtype=complex)
for k in range(-n//2,n//2):
    p += c[k%n]*ww**k
plt.plot(xx,np.real(p),'r')
plt.plot(xx,np.imag(p),'b')
plt.axhline(0)
plt.xlim(0,2*np.pi);
plt.savefig('temp2.pdf')
```

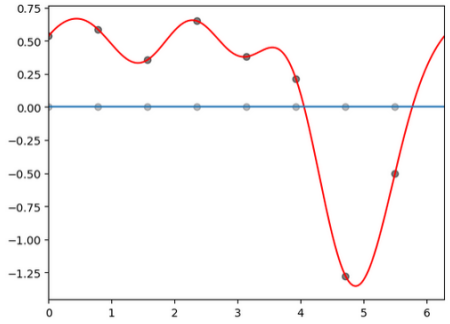


Much improved! But we can do even better ...

To do even better, treat k=-n/2 and k=n/2 symmetrically:
use the average of the k=-n/2 and k=n/2 terms ...

And with special treatment of Nyquist frequency components

```
np.random.seed(223)
y = np.random.randn(n)
x = np.linspace(0,2*np.pi,n,endpoint=False)
plt.plot(x,y,'ko',alpha=0.5)
plt.plot(x,y*0,'ko',alpha=0.25)
nn = 1000
xx = np.linspace(0,2*np.pi,nn,endpoint=False)
ww = np.exp(1j*xx)
twopi = 2*np.pi
I = 1j
w = np.exp(-I*2*np.pi/n)
A = np.empty((n,n),dtype=complex)
v = [w**j for j in range(n)]
A[:,0] = 1
for j in range(1,n):
    A[:,j] = A[:,j-1]*v
A /= n
c = A@y
#print(c)
from numpy.fft import fft
c = fft(y)/n
p = np.zeros(nn,dtype=complex)
for k in range(-n//2+1,n//2):
    p += c[k%n]*ww**k
# now deal with the Nyquist frequency part
for k in [-n//2,n//2]:
    p += c[k%n]*ww**k/2  #half of each of them
plt.plot(xx,np.real(p),'r')
plt.plot(xx,np.imag(p),'b')
plt.axhline(0)
plt.xlim(0,2*np.pi)
plt.savefig('temp.pdf')
```

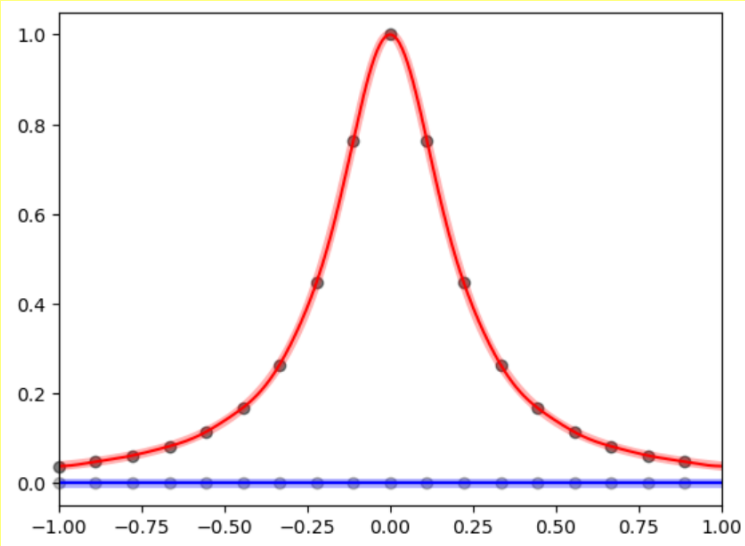


Now this looks sensible!


```
np.random.seed(223)
y = np.random.randn(n)
x = np.linspace(0,2*np.pi,n,endpoint=False)
plt.plot(x,y,'ko',alpha=0.5)
plt.plot(x,y*0,'ko',alpha=0.25)
nn = 1000
xx = np.linspace(0,2*np.pi,nn,endpoint=False)
ww = np.exp(1j*xx)
twopi = 2*np.pi
I = 1j
w = np.exp(-I*2*np.pi/n)
A = np.empty((n,n),dtype=complex)
v = [w**j for j in range(n)]
A[:,0] = 1
for j in range(1,n):
    A[:,j] = A[:,j-1]*v
A /= n
c = A@y
#print(c)
from numpy.fft import fft
c = fft(y)/n
p = np.zeros(nn,dtype=complex)
for k in range(-n//2+1,n//2):
    p += c[k%n]*ww**k
# now deal with the Nyquist frequency part
for k in [-n//2,n//2]:
    p += c[k%n]*ww**k/2  #half of each of them
plt.plot(xx,np.real(p),'r')
plt.plot(xx,np.imag(p),'b')
plt.axhline(0)
plt.xlim(0,2*np.pi)
plt.savefig('temp.pdf')
```

Let's try band-limited trig interpolant on some previously interpolated functions

such as Runge's function

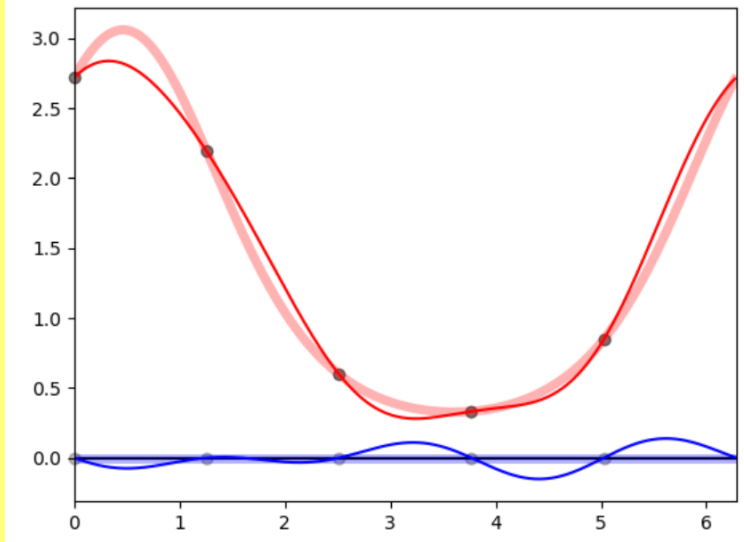


Not bad, especially considering the periodic extension of Runge's function has discontinuous derivative at the odd integers.

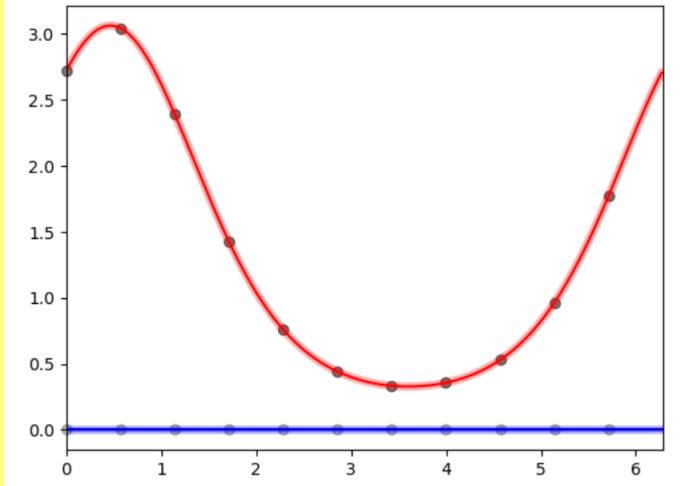
```
from numpy.fft import fft
# trig interpolant of some sampled functions
def trig_interp(F,X):
    L = X[-1]-X[0]
    twopi = 2*np.pi
    x = (twopi*(X-X[0])/L)[:-1]  #map to [0, 2pi] but don't include 2pi
    y = F(X)[:-1]
    n = len(x)
    plt.axhline(0,color='k')
    plt.plot(X[:-1],y,'ko',alpha=0.5,clip_on=False)
    plt.plot(X[:-1],y*0,'ko',alpha=0.25,clip_on=False)
    nn = 500
    xx = np.linspace(0,2*np.pi,nn,endpoint=False)
    XX = X[0] + xx*L/twopi
    ww = np.exp(1j*xx)
    twopi = 2*np.pi
    I = 1j
    c = fft(y)/n
    p = np.zeros(nn,dtype=complex)
    for k in range(-n//2+1,n//2):
        p += c[k%n]*ww**k
    # now deal with the Nyquist frequency part
    for k in [-n//2,n//2]:
        p += c[k%n]*ww**k/2  #half of each of them
    plt.plot(XX,F(XX),'r',alpha=0.3,lw=5)
    plt.plot(XX,np.real(p),'r') #lw=0.5,markersize=0.1)
    plt.plot(XX,F(XX)*0,'b',alpha=0.3,lw=5)
    plt.plot(XX,np.imag(p),'b')
    plt.xlim(X[0],X[-1])
    plt.savefig('temp.pdf')

def runge(x): return 1/(1+25*x**2)
X = np.linspace(-1,1,19)
trig_interp(runge,X)

def exptrig(x): return np.exp(np.cos(x) + 0.5*np.sin(x))
X = np.linspace(0,2*np.pi,6)
#trig_interp(exptrig,X)
```

Here's the trig interpolant of a function that has a C-infinity perodic extension.
Interpolant converges extremely fast as n in increased.

```
def exptrig(x): return np.exp( np.cos(x) + 0.5*np.sin(x) )
X = np.linspace(0,2*np.pi,6)
trig_interp(exptrig,X)
```




```
X = np.linspace(0,2*np.pi,12)
trig_interp(exptrig,X)
```



Applications

As in solving a PDE like $u_t = u_x$ with $u(x,0) = \text{given } g(x)$ for $u(x,t)$

u_x: need spatial derivative

Application: "spectral" differentiation of a grid function $\{(x_j, y_j)\}_{j=0,\dots,n-1}$

From the band-limited interpolant

$$q(x) = \sum_{k=-\frac{n}{2}}^{\frac{n}{2}}{}' c_{\tilde{k}} e^{i\tilde{k}x} \quad \longrightarrow \quad q'(x) = \sum_{k}{}' ik\, c_{\tilde{k}} e^{i\tilde{k}x}$$

where $\tilde{k} = \begin{cases} k, & k \geq 0 \\ k+n, & k<0 \end{cases}$

and use

$$q'(x_j) = \sum_{k}{}' ik\, c_{\tilde{k}} e^{ikx_j}$$

Here " Sigma' " means the sum with only half of the first and last terms.

Application: interpretation of a grid function via its DFT.

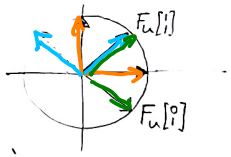
example: a sound recording of a piano note.

The DFT is a unitary linear transformation – essentially a rotation.

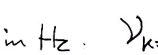
E.g. For $n=2$, $\omega = e^{-i\frac{2\pi}{2}} = e^{-i\pi} = -1$

$$W = \begin{bmatrix} \omega^0 & \omega^0 \\ \omega^0 & \omega^1 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad F_U = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

a 45° rotation followed by a reflection



An audio recording is a sequence of instantaneous measurements of sound pressure, often at 44,100 samples per second.

Relation between wave number k & frequency in Hz. $\nu_{k=1} = \frac{1}{\text{duration}}$  $\Rightarrow \nu_k = \frac{k}{\text{duration}}$

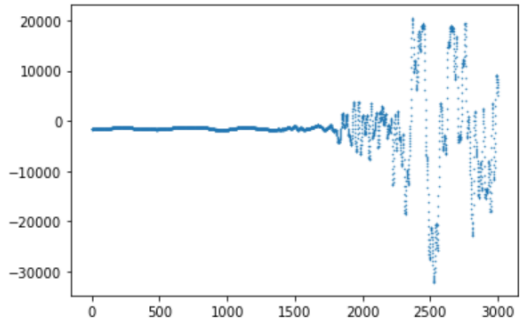
Recording: https://blue.math.buffalo.edu/537_f25/piano_low_f.wav

DFT of an audio recording

```
from scipy.io import wavfile
s,y = wavfile.read('day17/piano_low_f.wav')
s,y
print(len(y),'samples')
duration = len(y)/s
duration
jmax = 3000
plt.plot(np.arange(len(y))[:jmax],  y[:jmax], '.', markersize=1 );
```

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.io import wavfile
s,y = wavfile.read('day17/piano_low_f.wav')
s,y
print(len(y),'samples')
duration = len(y)/s
duration
jmax = 3000
plt.plot(np.arange(len(y))[:jmax],  y[:jmax], '.', markersize=1 );
```

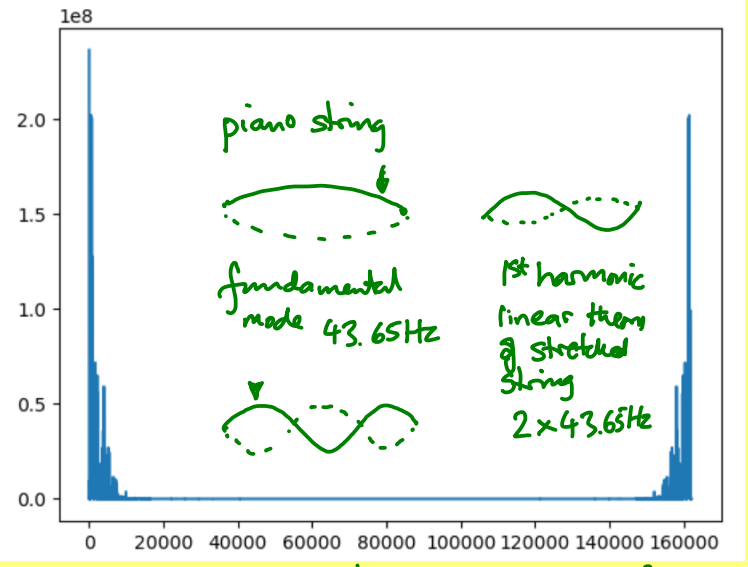
```
162030 samples
```



Now let's look at the data "from a different angle", by applying the DFT:

```
from numpy.fft import fft
c = fft(y)
kvals = np.arange(len(y))
plt.plot( kvals, np.abs(c) );
```

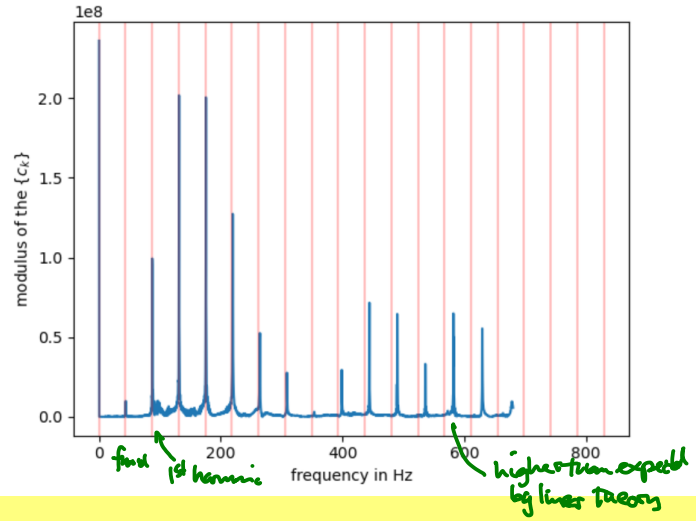
```
from numpy.fft import fft
c = fft(y)
kvals = np.arange(len(y))
plt.plot( kvals, np.abs(c) );
```




```
kmax = 2500
nu = kvals/duration      # frequency in Hz (cycles per second)
plt.plot( nu[:kmax], np.abs(c[:kmax]) )
plt.xlabel('frequency in Hz'); plt.ylabel('modulus of the $\{c_k\}$')
lowf = 43.65  # nominal frequency of the note low F
# draw pink lines at integer multiples of the nominal frequency of low F
[plt.axvline(lowf*m,color='r',alpha=0.25) for m in range(20)];
```

Zooming in on the frequencies below the Nyqist frequency:

```
kmax = 2500
nu = kvals/duration      # frequency in Hz (cycles per second)
plt.plot( nu[:kmax], np.abs(c[:kmax]) )
plt.xlabel('frequency in Hz'); plt.ylabel('modulus of the $\{c_k\}$')
lowf = 43.65  # nominal frequency of the note low F
[plt.axvline(lowf*m,color='r',alpha=0.25) for m in range(20)]; # draw pink lines
```



Brute force : $c_k = \sum_{j=0}^{n-1} w^{jk} y_j$ $(160{,}000)^2$ ops

We can learn a lot about the signal by looking at it from this angle.

1965 FFT

Final thought: labor of computing DFT in this example with n ~ 160,000 ... next the Fast DFT