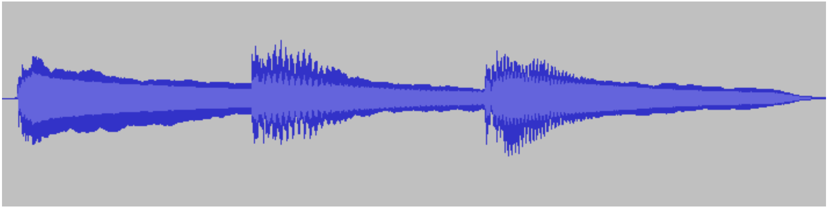
Math 260: Python programming in math

Fall 2020

Signal processing: The Fourier transform

An example...

A motivating example: Here's a sound wave plotted over time



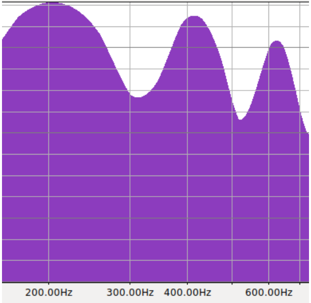
It's a piano playing notes of a minor chord (G$^{\#}$, B, D$^{\#}$).

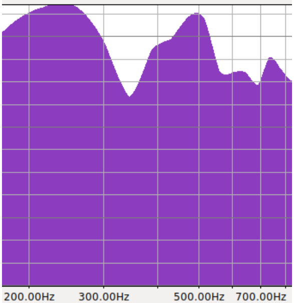
We can't tell the notes being played from this picture (sound vs. time)...

The wave is comprised of sounds of various frequencies - we instead want to plot the **spectrum** (showing the components of each frequency) for the notes.

An example...

For the first two notes:





Peaks: 206Hz ($G^{\#}3$), 412Hz ($G^{\#}4$), 618Hz ($D^{\#}4$)

Peaks: 245Hz (B3), 490Hz (B4), 735Hz ($D^{\#}4$)

- Higher harmonics: multiples of the bases frequency present!
- Frequency plot gives valuable information about the signal...
- Our goal: what is the underlying theory?

image from https://www.musical-u.com/learn/learn-to-recognise-chords-free-mp3-pack/, plots in audacity

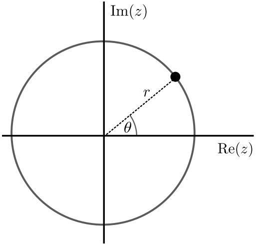
Complex numbers: review

- $z = x + iy$ is at a point (x, y) in the **complex plane** $\mathbb{C}$
  - real/imaginary parts $\mathrm{Re}(z) = x$ and $\mathrm{Im}(z) = y$
- Multiplication: $(a + ib)(c + id) = ac - bd + i(bc + ad)$ (from $i^2 = -1$)
- Conjugate: $z = x + iy \implies \bar{z} = x - iy$

$$\text{Euler's formula: } e^{i\theta} = \cos\theta + i\sin\theta$$

- Polar form(use Euler's formula):

$$x + iy = re^{i\theta}, \qquad \begin{cases} r^2 = x^2 + y^2, \\ \theta = \tan^{-1}(y/x) \end{cases}$$



- Note that $\overline{re^{i\theta}} = re^{-i\theta}$
- The **magnitude** of $z = x + iy$ is

$$|z| = \sqrt{x^2 + y^2} = r.$$

- θ: the **argument** or **phase** (physics)

Complex numbers in python

Python has a **built-in** complex type!

- Multiplication, etc. are all defined (`z*w` etc.)
- The imaginary unit is j (not i!); the number i is `1j`.

```
z = 1 + 2j
b = 4
w = 1 + b*(1j)
```

- numpy arrays are **float** by default. To make complex arrays:

```
z_values = np.array(0, dtype=complex)
```

- `numpy` functions like `exp` are defined for complex numbers:

```
z = np.exp(pi*1j)
print(z)  # z is -1 + 1.22e-16j
x = float(z)  # now a real number
```

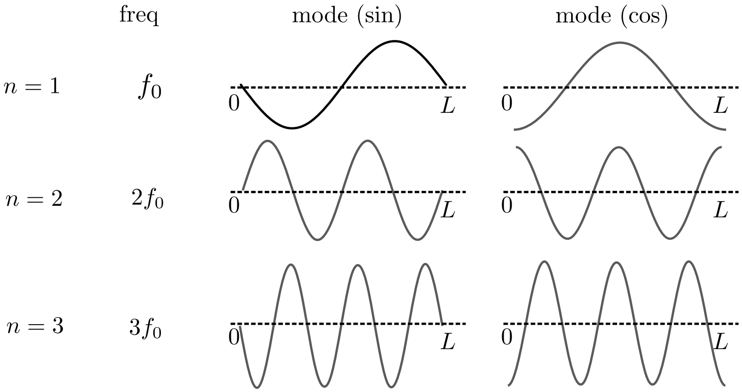
Best practices (complex numbers have two parts):

Results that 'should be' real may be **real up to rounding error**

- Sometimes, you need to convert to an *actual* real number with `float(z)`
- This may hide errors... check real/imaginary parts before converting!

A physical interpretation... standing waves $h(x)$ for a string

- Case one: fixed at both ends ($h(0) = h(L) = 0$)
- Case two: free to slide up and down ($h'(0) = h'(L) = 0$)



Any vibration is a **superposition** (linear combination) of these modes.

Fourier series: context

A fundamental theorem from math: Fourier series representation

- Non-trivial to prove! Take as true here...
- Deep insights into physical systems and more (waves, ...)

Suppose $f(x)$ is a (not terrible) function defined on the interval $[-\pi, \pi]$.

Then it has a (complex) **Fourier series** representation

$$f(x) = \sum_{n=-\infty}^{\infty} c_n e^{inx}$$

where the coefficients are given by

$$c_n = \frac{1}{2\pi} \int_0^{2\pi} f(x) e^{-inx}\, dx.$$

Note that if f is a real function, then coefficients come in 'pairs':

$$c_{-n} = \frac{1}{2\pi} \int_0^{2\pi} f(x) e^{inx}\, dx = \frac{1}{2\pi} \int_0^{2\pi} \overline{f(x) e^{-inx}}\, dx = \overline{c_n}$$

since $\overline{z_1 z_2} = \overline{z_1}\,\overline{z_2}$ and f is real (so $f = \overline{f}$).

Fourier series: complex to real

Assuming that f is real, we get $c_{-n} = \overline{c_n}$. Recall that

$$\cos x = \frac{1}{2}(e^{ix} + e^{-ix}), \quad \sin x = -\frac{i}{2}(e^{ix} - e^{-ix}).$$

Now pair up $+n$ and $-n$ terms and write c_n as

$$c_n = \frac{1}{2}(a_n - ib_n).$$

Then

$$f(x) = c_0 + \sum_{n=1}^{\infty} \left(c_n e^{inx} + \overline{c_n} e^{-inx} \right)$$

$$\implies f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left(a_n \cos nx + b_n \sin nx \right).$$

This is the **real** form of the Fourier series.

We use this idea often to convert between...

- ...the complex form (more elegant, convenient)
- ...the real form (often more meaningful for results)

From (continuous) Fourier series to the discrete...

Fourier series: orthogonality

$$f(x) = \sum_{n=-\infty}^{\infty} c_n e^{inx}, \qquad c_n = \frac{1}{2\pi} \int_0^{2\pi} f(x) e^{-inx}\, dx.$$

Why is this formula true? The key is that the functions e^{inx} are an **orthogonal basis** (in the linear algebra sense). It is true that

$$\int_0^{2\pi} e^{imx} e^{-inx}\, dx = \begin{cases} 2\pi & m = n \\ 0 & m \neq n \end{cases}.$$

Define the 'inner product' (by analogy to the dot product)

$$\langle f, g \rangle = \int_0^{2\pi} f(x) \overline{g(x)}\, dx \qquad (\text{like } \vec{x} \cdot \vec{y} = \sum_{k=1}^{n} x_k \overline{y_k})$$

Then the functions e^{inx} are 'orthogonal':

$$\langle e^{imx}, e^{inx} \rangle = 0 \text{for} m \neq n.$$

Fourier series: orthogonality

Now we can see how the coefficient formula works:

$$f(x) = \sum_{n=-\infty}^{\infty} c_n e^{inx}, \qquad c_n = \frac{1}{2\pi}\int_0^{2\pi} f(x)e^{-inx}\,dx.$$

Take the inner product of the Fourier series with one of the exponentials:

$$\begin{aligned}\langle f, e^{imx}\rangle &= \sum_{n=-\infty}^{\infty} c_n \int_0^{2\pi} e^{inx}e^{-imx}\,dx \\ &= \sum_{n=-\infty}^{\infty} c_n \left(\begin{cases} 2\pi & m = n \\ 0 & m \neq n \end{cases}\right) \\ &= 2\pi c_m.\end{aligned}$$

This means that the coefficient c_m depends only on the e^{imx} part (the 'components' of the series do not overlap, like perpendicular vectors)

But enough theory; here we are looking to *compute* transforms...

Discrete Fourier transform: context

$$f(x) = \sum_{n=-\infty}^{\infty} c_n e^{inx}, \qquad c_n = \frac{1}{2\pi}\int_0^{2\pi} f(x)e^{-inx}\,dx.$$

How do we get a computational version of the Fourier series?

- We can approximate the integrals for c_n
- We need a 'discrete' version of orthogonality

To get there, pick N and consider using grid points ('samples')

$$x_j = 2\pi j/N, \quad j = 0, 1, \cdots, N-1.$$

Now use the trapezoidal rule to approximate c_n:

$$c_n = \frac{1}{2\pi}\int_0^{2\pi} f(x)e^{-inx}\,dx \approx \frac{1}{2\pi}\cdot\frac{2\pi}{N}\left(\frac{1}{2}f(0) + \sum_{k=1}^{N-1} f(x_k)e^{-inx_k} + \frac{1}{2}f(2\pi)\right)$$

The endpoints are a problem - but if f has period 2π then $f(0) = f(2\pi)$ and

$$c_n \approx \frac{1}{N}\sum_{k=0}^{N} f(x_k)e^{-inx_k}.$$

This is the basis of the **discrete Fourier transform**.

Discrete Fourier transform: definition

Now let's define the discrete transform. As an example, consider the 'signal'

$$f(x) = 2\cos 2x + 6\sin x, \qquad x \in [0, 2\pi]$$

and suppose we have **samples** of f at the (standard) **sample points**:

$$f_j = f(x_j) \text{ is known }, \quad x_j = 2\pi j/N, \quad j = 0, 1, \cdots, N-1.$$

We want to identify the frequencies (2 and 1) and amplitudes (2 and 6).

Key observation: orthogonality

For functions sampled at the x_j's, define the 'dot product'

$$\langle f, g\rangle_d = \sum_{j=0}^{N-1} f(x_j)\overline{g(x_j)}$$

i.e. the dot product of f and g at the sample points. Then the e^{inx}'s are **orthogonal** in this dot product, i.e.

$$\langle e^{imx}, e^{inx}\rangle_d = \sum_{j=0}^{N-1} e^{imx_j} e^{-inx_j} = \begin{cases} 0 & m \neq n \\ N & m = n \end{cases}.$$

Discrete Fourier transform: definition

Definition:

The **Discrete Fourier transform** (DFT) of a vector $\vec{f} = (f_0, \cdots, f_{N-1})$ is

$$F_k = \frac{1}{N} \sum_{j=0}^{N-1} f_j e^{-2\pi ikj/N} = \frac{1}{N} \langle f, e^{ikx} \rangle_d$$

which is also a vector $\vec{F}$ of length N.
The **inverse transform** (IDFT) is given by

$$f_j = \sum_{k=0}^{N-1} F_k e^{2\pi ikj/N}$$

Caution:

There are several slightly different ways to write this pair of formulas.

- The 'plus' and 'minus' exponentials (e^{ikx} vs. e^{-ikx}) can be switched
- The product of the factors in front of the sum must be $1/N$. You may see $1/N, 1$ or $1, 1/N$ or $1/\sqrt{N}, 1/\sqrt{N}$ for the DFT/IDFT.

Always check documentation before using a DFT routine!

The discrete Fourier transform

Discrete Fourier transform: definition

Definition:

The **Discrete Fourier transform** (DFT) of a vector $\vec{f} = (f_0, \cdots, f_{N-1})$ is

$$F_k = \frac{1}{N}\sum_{j=0}^{N-1} f_j e^{-2\pi ikj/N} = \frac{1}{N}\langle f, e^{ikx}\rangle_d$$

which is also a vector $\vec{F}$ of length N.
The **inverse transform** (IDFT) is given by

$$f_j = \sum_{k=0}^{N-1} F_k e^{2\pi ikj/N}$$

- We think of $\vec{f}$ as coming from sampling data in $[0, 2\pi]$ at the sample points.
- Letting $\omega = e^{-2\pi i/N}$, we can write the DFT/IDFT nicely as

$$F_k = \frac{1}{N}\sum_{j=0}^{N-1} f_j \omega^{jk}, \quad f_j = \sum_{k=0}^{N-1} F_k \omega^{-jk}.$$

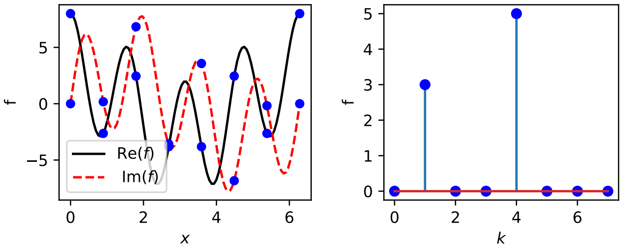
A first example

The DFT $\{F_k\}$ gives amplitudes of **frequencies** in the signal. Example: consider

$$f(x) = 3e^{ix} + 5e^{4ix}.$$

Sample f with $N = 8$ to get samples $\vec{f} = (f_0, f_1 \cdots, f_7)$... then take the DFT:

$$F_k = \frac{1}{N}\sum_{j=0}^{N-1} f_j e^{-2\pi i k j/N}$$



By the discrete orthogonality rule, $F_k \neq 0$ only for $k = 1, k = 4$ so

$$\vec{F} = (0, 3, 0, 0, 5, 0, 0, 0)$$

You can think of the DFT formula for the k-th component,

$$\text{signal} \to \frac{1}{N}\langle \text{signal}, e^{ikx}\rangle_d$$

as 'selecting' that frequency and returning its amplitude
(or from linear algebra: projection...)

The DFT is then selecting each frequency to get the amplitudes at each and returning them as a vector. For instance, consider

$$\begin{aligned} f(x) &= 2\cos 2x \quad + \; 6\sin x \\ &= e^{2ix} + e^{-2ix} - 3ie^{ix} + 3ie^{-ix} \end{aligned}$$

with $N = 6$. Letting $\phi_k(x) = e^{ikx}$, we get

$$\begin{aligned} \langle f, \phi_0\rangle/N &\to 0 \text{ (not present!)} \\ \langle f, \phi_1\rangle/N &\to -3i \\ \langle f, \phi_2\rangle/N &\to 1 \\ \langle f, \phi_3\rangle/N &\to 0 \text{ (not present!)} \\ \langle f, \phi_4\rangle/N &\to 1 \\ \langle f, \phi_5\rangle/N &\to 3i \end{aligned}$$

The complication: aliasing

It is important to know which frequencies are present in the DFT.
This is subtle because the **DFT only knows about the sample points**.

- The exponentials in the DFT are

$$e^{0x}, e^{ix}, \cdots, e^{i(N-1)x}$$

- Notice that at **any sample point**, by Euler's formula

$$e^{iNx_j} = e^{iN(2\pi j/N)} = (e^{2\pi i})^j = 1$$

  We can use this to 'shift' exponentials in the DFT for free:

$$e^{-ikx_j} = {\color{red}e^{iNx_j}} e^{-ikx_j} = e^{i(N-k)x_j}.$$

  Thus, the DFT 'sees' frequency $-k$ as $N-k$.
- Similarly, $k+N, k+2N, \cdots$, are **all** seen as k
- This effect is called **aliasing**

Aliasing

Consider sampling

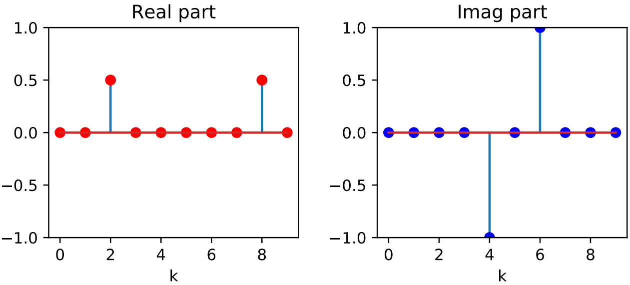
$$f(x) = \cos 2x + 2\sin 4x.$$

From Euler's formula,

$$\cos 2x = \frac{1}{2}e^{2ix} + \frac{1}{2}e^{-2ix}, \quad \sin x = -\frac{i}{2}(e^{4ix} - e^{-4ix}).$$

Thus the frequencies present are ± 2 and ± 1.

The DFT with $N = 10$ does what we want:



Real part: $1/2$ and 2 and $-2 + 10 = 8$

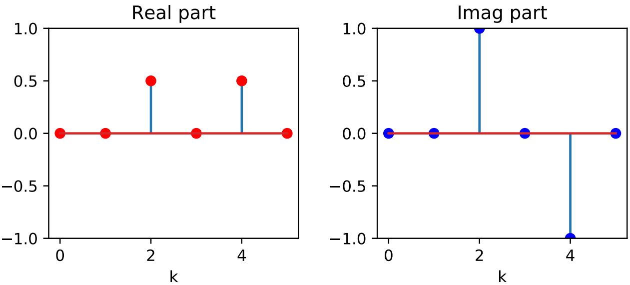
Imag part: $-1/2$ and 4 and $1/2$ at $-4 + 10 = 6$

Aliasing

Consider sampling

$$f(x) = \cos 2x + 2\sin 4x.$$

However, $N = 6$ is not enough samples!

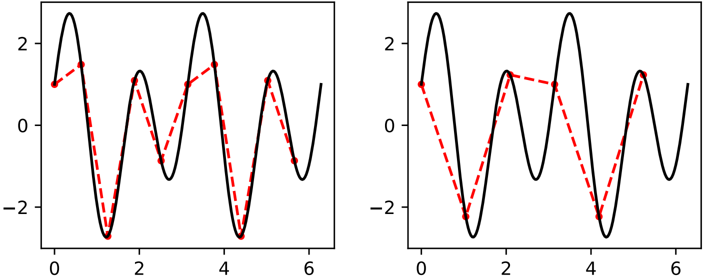


Real part: $1/2$ and 2 and $-2+6=4$ (okay)
Imag part: $-1/2$ and 4 and $1/2$ at $-4+6=2$ (bad!)

This data does not distinguish $f(x)$ from

$$\tilde{f}(x) = \cos 2x - 2\sin 2x.$$

Sampling



To fit all the frequencies, we need to take

$$N > 2\max(\text{frequencies in signal})$$

which is called the **Nyquist rate**.
Otherwise, the sampling is too slow to 'see' the higher frequency parts.

(Example: filming a spinning wheel... the 'wagon-wheel effect')

Shifting frequencies

The fact that the $-k$ frequencies go to $N-k$ suggests that

$$k = 0, 1, \cdots, N-1$$

are wrong for real signals: we need matched $+$ and $-$ parts. Instead:

$$\text{freqs} = -\frac{N}{2}, \frac{N}{2}+1, \cdots, -1, 0 \cdots, \frac{N}{2}-1$$

are the right frequencies.

We can also **shift** the k's by $N/2$ to get

$$\text{freqs}_{\text{shifted}} = {\color{red}-N/2,\ -N/2+1, \cdots, -1,}\, {\color{blue}0, 1 \cdots N/2-1}$$

The transform $\vec{F}$ must then be shifted the same way to line up with the k's...

$$\vec{F} = {\color{blue}F_0, F_1, \cdots, F_{N/2-1},}\ \ {\color{red}F_{N/2}, F_{N/2+1} \cdots F_{N-1}}$$

$$\implies \vec{F}_{\text{shifted}} = {\color{red}F_{N/2}, F_{N/2+1} \cdots, F_{N-1},}\ \ {\color{blue}F_0, F_1, \cdots F_{N/2-1}}$$

The shifted `freq` vector will have `freq[N//2] = 0` (the 'center' point).

Shifting frequencies

An example - take $N = 6$...

The DFT yields

$$\begin{cases} F_0, F_1, F_2, F_3, F_4, F_5 \\ 0,\ \ 1,\ \ 2,\ \ 3,\ \ 4,\ \ 5 \end{cases}$$

The true frequencies (assuming a real signal) are:

$$\begin{cases} F_0, F_1, F_2, F_3, F_4, F_5 \\ 0,\ \ 1,\ \ 2, -3, -2, -1 \end{cases}$$

After shifting, the result is

$$\begin{cases} F_3,\ \ F_4,\ \ F_5,\ \ F_0, F_1, F_2 \\ -3, -2, -1, 0,\ \ 1,\ \ 2 \end{cases}$$

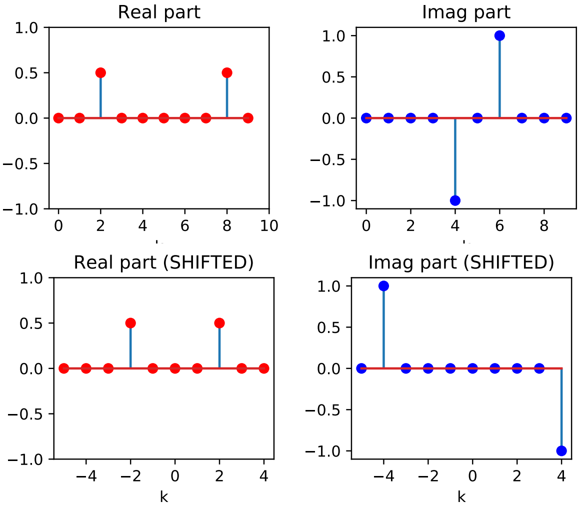
- In python, this is done using two functions `fftfreq` and `fftshift`
- You could place $N/2$ on either end (e.g. $+3$ or -3)
- (check the convention in your code carefully; python uses $-N/2$)

Shifting frequencies: example

For $N = 10$ $f(x) = \cos 2x + 2\sin 4x$, the DFT uses the frequencies

$$k = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$$

$$k_{\text{shifted}} = -5, -4, -3, -2, -1, 0, 1, 2, 3, 4$$



Computation: the Fast fourier transform

Implementation...

Now let's view the formulas just as 'sums to compute':

$$\text{DFT: } F_k = \frac{1}{N}\sum_{j=0}^{N-1} f_j e^{-2\pi ikj/N}, \quad \text{IDFT: } f_j = \sum_{k=0}^{N-1} F_k e^{2\pi ikj/N}$$

First note that the IDFT is actually a DFT. Taking the conjugate:

$$\overline{\text{IDFT}(\vec{F})_j} = \sum_{j=0}^{N-1} \overline{F_k} e^{-2\pi ikj/N}$$

which is just the DFT of $\overline{F_k}$ (up to the $1/N$ factor).

Thus we only need a way to compute the DFT.

The slow method: Brute force - just compute the sum directly.
How many operations ($O(\cdots)$) are required?

Answer: $O(N^2)$.
This seems okay, but we can do **much** better...

Implementation...

$$\text{DFT:} \quad F_k = \sum_{j=0}^{N-1} f_j \omega^{jk}, \quad \omega = e^{-2\pi i/N}.$$

The trick: divide and conquer! Example: let $N = 8$ and $\omega = e^{-2\pi i/8}$. Then

$$\begin{aligned}
F_k &= \mathrm{DFT}(f_0, \cdots, f_7)_k \\
&= \frac{1}{N} \sum_{j=0}^{8} f_j \omega^{jk}, \qquad k = 0, \cdots, 7, \\
&= \frac{1}{N}\left(f_0 + f_2\omega^{2k} + f_4\omega^{4k} + f_6\omega^{6k}\right) + \frac{1}{N}\omega^k\left(f_1 + f_3\omega^{2k} + f_5\omega^{4k} + f_7\omega^{6k}\right) \\
&= \frac{1}{2}\frac{2}{N}\left(f_0 + f_2\xi^{k} + f_4\xi^{2k} + f_6\xi^{3k}\right) + \frac{1}{2}\omega^k\frac{2}{N}\left(f_1 + f_3\xi^{k} + f_5\xi^{2k} + f_7\xi^{3k}\right)
\end{aligned}$$

where $\xi = \omega^2$. But $\xi = e^{-2\pi i/4}$. which is the 'omega' for $N = 4$....
Thus **both** sums in parentheses are DFT's for $N = 4$! We then have

$$F_k = \frac{1}{2}\mathrm{DFT}(f_0, f_2, f_4, f_6)_k + \frac{1}{2}\omega^k \mathrm{DFT}(f_1, f_3, f_5, f_7)_k. \qquad k = 0, \cdots, 7.$$

Note: on the right side, k is taken 'mod 4' in the subscripts ($4, 5$ are $0, 1$, etc.).

More generally, suppose N is a power of 2. Then

$$\mathrm{DFT}(f_0, f_1, \cdots, f_{N-1}) = \frac{1}{2}\mathrm{DFT}(f_0, f_2, \cdots, f_{N-2}) + \frac{1}{2}\omega^k \mathrm{DFT}(f_1, f_3, \cdots, f_{N-1}) \tag{F}$$

Thus the DFT can be computed as follows:

- Split the vector into two half-sized vectors: odds and evens
- Take the DFT of the odd and even parts (recursively)
- Combine them according to the formula (F).

This is the basis of the **fast Fourier transform** (FFT).

Why 'fast'? Suppose $N = 2^p$ and let $C(N)$ be the cost of the transform. Then - just as we saw for mergesort,

$$C(N) = aN + 2C(N/2)$$

$$\implies C(N) = aN + 2a(N/2) + 4a(N/4) + \cdots = paN.$$

It follows that the FFT requires $O(N \log N)$ operations!

The fast Fourier transform

- This is **much faster** than the slow $O(N^2)$ method.

  - For $N = 10^4$, the slow method is $\approx 10^4 / \log_2(10^4) \approx 750$ times slower!

- The FFT is one of the most important algorithms of the twentieth century - essential for signal processing, data analysis...
- First version usually credited to Cooley & Tukey (1965)

  (Side note: was known to Gauss in the 1800s...)

There are more details to making the FFT work (the messy part):

- What if N is not a power of 2?
- What is the right base case? (Answer: 'small' cases like $N = 3$...)
- How do we un-recursion it? (Answer: some clever encoding, plus a stack...)

Scientific computing packages will have an fft available -
In numpy: `numpy.fft` has all the FFT features.

The fast Fourier transform: python

A quick tour of `numpy.fft`....

- FFT/IFFT: `fft(x)` and `ifft(x)` (convention: $1/N$ on **IFFT**)

```
n = 6
t = np.linspace(0, 2*np.pi, n, endpoint=False) # 0, pi/6, ... 5pi/6
d = 2*np.pi/n # sample spacing
samples = some_function(t)
c = fft.fft(samples)
freq = fft.fftfreq(n, d) #[0,1,2,-3,-2,-1]/(2*pi)
freq = fft.fftshift(freq) # now [-3,-2,-1,0,1,2]/(2*pi)
c = fft.fftshift(c) # now c is shifted the same way
```

- `ifft(fft(x)` $\approx$ `x` [up to rounding]
- `fftfreq(n, d)` gets the 'frequencies' for the length N FFT.
  - `d` is the sample spacing L/N (for samples in $[0, L]$)
  - If `d` is in seconds, then the freqs. are in cycles/second (Hz).
  - This in the **original** order (not shifted). If N is even:

$$\texttt{fftfreq(N)} = [0, 1, 2, \cdots N/2 - 1, -N/2, -N/2 + 1, \cdots - 1]/L$$

- `fftshift(v)` centers the data (as discussed)

A practical example: filtering

FFT: physical units

It's worth clarifying the issue of units for frequency...

If we sample N values from $t = 0$ to $t = L$ **seconds**,
the values $k = 0, \pm 1, \cdots$ correspond to k/L **cycles per second** (Hz).

To see this, look at the DFT with N samples on an interval $[0, L]$...

$$\begin{aligned} F_k &= \frac{1}{N} \sum_{j=0}^{N-1} f(x_j) e^{-2\pi ikj/N} \\ &= \frac{1}{N} \sum_{j=0}^{N-1} f(x_j) e^{-(2\pi ik/L)(Lj/N)} \\ &= \frac{1}{N} \langle f, e^{2\pi ikx/L} \rangle_d, \qquad \langle f, g \rangle = \sum_{j=0}^{N-1} f(x_j) g(x_j). \end{aligned}$$

Conclusion: the frequencies for this DFT are $2\pi k/L$ (rad/s) or k/L (cycles/s).

FFT: physical units - interpretation

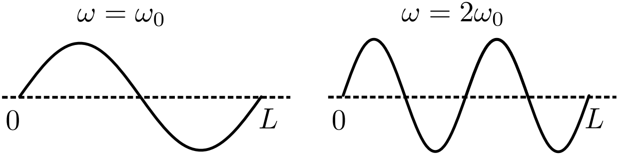
Again, suppose we sample N values from $t = 0$ to $t = L$ time.

The **fundamental frequency**

$$\omega_0 = 1/L \text{ cycles/time} = 2\pi/L \text{ rad./time}$$

is the lowest frequency ω such that $e^{i\omega t}$ is periodic.

The other frequencies are **multiples of the fundamental one**.



- For a Fourier series

$$f = \sum_n c_n e^{i\omega_n x},$$

  frequencies are an infinite sequence (enough to represent f).
- For the DFT, frequencies go up to $N/2L$, and higher ones are aliased

FFT: physical units - interpretation

Yet another intuition for the scaling with L...

Consider a sound wave sampled in $[[0, L]$ with N samples; the frequencies are

$$\frac{1}{L}, \quad \cdots \frac{N}{2L}.$$

Now play the sound in double speed, taking N samples again (interval: $[0, L/2]$). The frequencies are:

$$2 \cdot \frac{1}{L}, \quad \cdots 2 \cdot \frac{N}{2L}.$$

This matches what you know of sound!

FFT: physical units

Example: Consider the ‘pure’ middle-C tone

$$f(t) = \sin(524\pi t), \qquad \text{frequency} = 262\text{Hz}.$$

Sample $N = 1024$ points in the interval $[0, 1]$ and take the DFT. Then

$$F_k = \frac{1}{N} \sum_{j=0}^{N-1} f(x_j) e^{-2\pi i kj} = \frac{1}{N} \langle f, e^{2\pi i kx} \rangle_d.$$

The set $\{e^{2\pi i kx}\}$ is orthogonal in the inner product and

$$f(t) = -\frac{i}{2} e^{262 \cdot 2\pi i t} + \frac{i}{2} e^{-262 \cdot 2\pi i t}$$

Thus, the DFT selects the right frequencies, and we get

$$F_{262} = -\frac{i}{2}, \quad F_{-262+N} = \frac{i}{2}, \qquad F_k = 0 \ \text{ otherwise}$$

corresponding to the frequency $262/L = 262\text{Hz}$.

An example: low pass filter

Now let's look at a real example. Let's construct a **low pass filter**, which removes all frequencies above a cutoff value f_c (Hz) in the signal.

Info on the signal:

- Over a time $[0, L]$ with a sample rate $r = 44100$Hz (the `wav` standard)
- The real frequencies are related to k by `freq[k] = k/L`
- Spacing between samples: $1/r$ seconds

The algorithm:

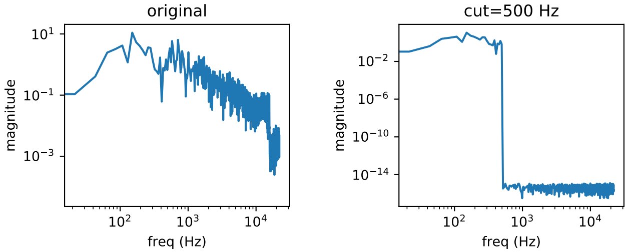
1) Load an audio file (`data`) (here a `.wav`, using `scipy.io.wavfile`)
2) Compute the FFT, `df`, and associated dimensional frequencies `freq` (Hz)
3) et `df[k]` to zero for all k's with $|\text{freq}[k]| > f_c$.
4) Inverse transform with the IFFT, save the result as a `wav`!

In short: **transform**, then **filter**, then **inverse transform** back.

An example: low pass filter

The algorithm:

- Load an audio file (`data`) (here a `.wav`, using `scipy.io.wavfile`)
- Compute the FFT, `df`, and associated dimensional frequencies `freq` (Hz)
- Set `df[k]` to zero for all k's with $|\text{freq}[k]| > f_c$.
- Inverse transform with the IFFT, save the result as a `wav`!



Note: We plot the **power spectrum**: $|F_k|$ vs. freq. for the 'positive' k's.

(Why not also plot F_k for the 'negative' k's?)

Convolutions

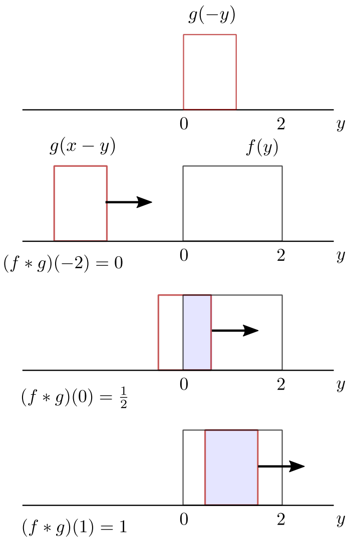
An important operation in mathematics is the **convolution**

$$(f * g)(x) = \int_{-\infty}^{\infty} f(y)g(x-y)\,dy.$$

for functions f, g defined for real numbers.

Often, g is a 'window' that slides by the graph of f, picking out a part of f weighted by some shape.

Example: f and g are boxes, $H = 1$...



Convolutions

For **periodic** functions with period 2π, we instead use

$$(f * g)(x) = \int_0^{2\pi} f(y)g(x-y)\,dy.$$

We can think of the argument $x - y$ as periodic, so it 'wraps around' the interval $[0, 2\pi]$ (e.g. $-\pi/2$ is $3\pi/2$).

The discrete analogue is the **circular convolution**

$$(\vec{f} * \vec{g})_j = \sum_{m=0}^{N-1} f_m g_{j-m}$$

where subscripts are taken 'with period N' (so -1 is $N-1$ and so on).

Theorem (convolution and DFT)

The DFT of a convolution is the **element-wise product** of the DFTs...

$$\mathrm{DFT}(\vec{f} * \vec{g})_k = \mathrm{DFT}(F)_k \mathrm{DFT}(G)_k$$

(Proof: a direct computation...)

Convolutions

Theorem

The DFT of a convolution is the **element-wise product** of the DFTs...

$$\mathrm{DFT}(\vec{f} * \vec{g})_k = \mathrm{DFT}(F)_k \mathrm{DFT}(G)_k$$

- Thus 'pointwise' multiplication of frequencies is convolution in real space
- Convolutions, like the DFT, are $O(N \log N)$ (not $O(N^2)$)!
- 'Filters' (like low pass, etc.) are convolutions in real space

Example: consider the DFT in $[0, 2\pi]$ with spacing $h = 2\pi/N$ and

$$\vec{g} = [-\frac{1}{h}, \frac{1}{h}, 0, 0, \cdots].$$

$$F_1 = f_0 g_1 + f_1 g_0 + \cdots = \frac{f(x_1) - f(x_1 - h)}{h} \approx f'(x_1)$$

$$F_2 = f_0 g_2 + f_1 g_1 + f_2 g_0 + \cdots = \frac{f(x_2) - f(x_2 - h)}{h} \approx f'(x_2)$$

$$F_j = \text{zeros} + f_{j-1} g_1 + f_j g_0 + \text{zeros} = \frac{f(x_j) - f(x_j - h)}{h} \approx f'(x_j)$$

so the convolution gives the backward difference approximation to f'(x)!

A few more notes on the Fourier transform

More on the Fourier transform

A **power spectrum plot** shows $|F_k|$ vs. frequency (or k). For this,

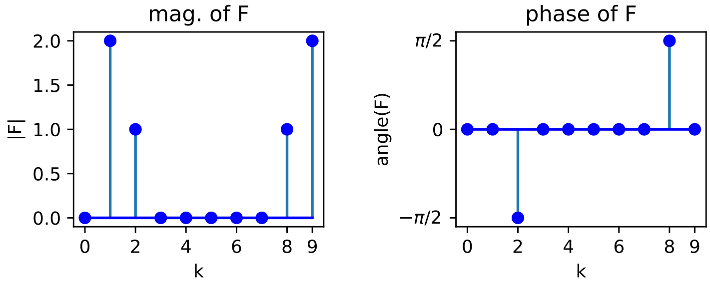
$$f(t) = \sin t,\ g(t) = \cos t \quad \text{have the same } F_k.$$

We can also plot the '**angle**' or '**phase**' of F, i.e.

$$F_k = r_k e^{i\theta_k},\ \Longrightarrow \quad \text{plot } r_k \text{ vs. } k \text{ , plot } \theta_k \text{ vs. } k.$$

Observe that $\theta = \pm 1$ corresponds to pure cosines, and $\theta = \pm\frac{\pi}{2}$ to pure sines.

$$f(t) = 4\cos t + 2\sin 2t, \quad F = (2, -i, 0, \cdots, 0, i, 2)$$



cosine: $2e^{it} \Longrightarrow \arg = 0,$ sine: $-ie^{2it} = e^{-i\pi/2}e^{2it} \Longrightarrow \arg = -\pi/2$

More on the Fourier transform

More generally, suppose we translate ('phase shift') the signal...

$$\sin 2t \implies \sin 2(t - \phi) = \cos 2\phi \sin 2t - \sin 2\phi \cos 2t.$$

The magnitude is $1/2$ - which shows up in the plot of $|F|$. Since

$$\sin 2(t - \phi) = -\frac{i}{2}\left(e^{-2i\phi} e^{2it} - e^{2i\phi} e^{-2it}\right)$$

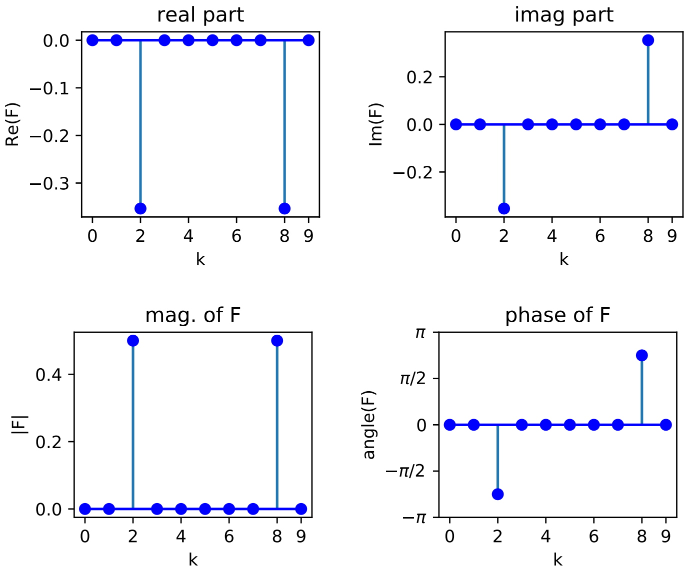
ϕ shows up in the **phase** plot ($\mp 2\phi$ at $\pm$ frequencies).

This means that the frequency k may show up as a mix of sines and cosines.

sin vs. cos depends on the starting point of the sampling (shift in t).

More on the Fourier transform

Example: $f(t) = \sin(2t)$ shifted by $\pi/8$ (so $f(t - \pi/8) = \sin(2(t - \pi/8))$):



Bonus application: multiplying numbers

Recall that we also define the **circular convolution**

$$(\vec{f} *_c \vec{g})_k = \sum_{j=0}^{N-1} f_j g_{k-j}$$

Observe that the periodic nature is avoided if the data is 'padded'.
Claim: If

$$f_j = g_j = 0 \text{ for } j > N/2$$

(or a similar padding scheme) then

$$(\vec{f} *_c \vec{g})_k = \sum_{j=0}^{N-1} f_j g_{k-j} = \sum_{j=0}^{k} f_j g_{k-j}.$$

That is, if only half the vectors are filled, they don't interact when 'looped'.

Details to check:

- If $j > N/2$ then $f_j = 0 \implies$ zero term.
- If $0 < j < N/2$ and $k - j < 0$ then $k - j$ becomes $> N/2 \implies g_{k-j} = 0$.

Thus the sum 'mod N' is just a regular sum (no negative indices!).

Bonus application: multiplying numbers

Thus, with enough padding, we can use the circular convolution to compute

$$(\vec{f} * \vec{g})_k = \sum_{j=-\infty}^{\infty} f_j g_{k-j}$$

if $\vec{f}, \vec{g}$ have finitely many non-zero entries, or the variant

$$(\vec{f} * \vec{g})_k = \sum_{j=0}^{k} f_j g_{k-j}$$

for length N vectors (all the same thing, but using slight variations)...

That is: with enough padding, the 'boundary' interactions don't matter.

Bonus application: multiplying numbers

Now why does this matter? We saw that for the FFT,

$$\texttt{fft}(\vec{f} *_c \vec{g})_k = F_k G_k, \qquad F = \texttt{fft}(\vec{f}),\ G = \texttt{fft}(\vec{g}).$$

The FFT of the convolution is the (element-wise) product of each FFT.

Thus, to compute the circular or other convolution, we can

- Setup:

  -Identify vectors $\vec{f}$, $\vec{g}$ with all the non-zero data

  -Pad the vectors with extra zeros as needed
- FFT part:

  -Take the FFT of the padded vectors to get F and G

  -Compute the (trivial) product $F_k G_k$

  -Take the IFFT to get the convolution

(For the circular convolution, just skip to the FFT part) ('Fourier's law')

Bonus application: multiplying numbers

Suppose I want to compute the product of two n digit numbers with digits

$$c_{n-1}, \cdots c_0, \qquad d_{n-1}, \cdots d_0.$$

Then the numbers are

$$c = \sum_{j=0}^{n-1} c_j 10^j, \quad d = \sum_{j=0}^{n-1} d_j 10^j.$$

Taking the product cd, we see that

$$(c_j 10^j) \cdot d_{k-j} 10^{k-j} \text{ goes to the } 10^k \text{ term of } cd$$

which accounts for all the terms (once multiplied out),and so

$$cd = \sum_{k=0}^{2n} a_k 10^k, \qquad a_k = \sum_{j=0}^{k} c_j d_{k-j}.$$

although the a_k's here are not digits. For instance, for $123 \times 45 = 5535$,

$$c = 1 \times 10^2 + 2 \times 10^1 + 3, \qquad d = 4 \times 10^1 + 5,$$

$$a_0 = 3 \cdot 5, \quad a_1 = 3 \cdot 4 + 2 \cdot 5, \quad a_2 = 1 \cdot 5 + 2 \cdot 4, \quad a_3 = 1 \cdot 4$$

$$15 + 22 \cdot 10 + 13 \cdot 100 + 4 \cdot 1000 = 5535.$$

Bonus application: multiplying numbers

So, for numbers,

$$c = \sum_{j=0}^{n-1} c_j 10^j, \quad d = \sum_{j=0}^{n-1} d_j 10^j.$$

we have that

$$cd = \sum_{k=0}^{2n} a_k 10^k, \qquad a_k = \sum_{j=0}^{k} c_j d_{k-j}$$

which means cd can be computed from the *circular* convolution

$$\vec{a} = \vec{c} *_c \vec{d}$$

where $\vec{c}$ and $\vec{d}$ are padded enough.
Example:

$$c = 1 \times 10^2 + 2 \times 10^1 + 3, \qquad d = 4 \times 10^1 + 5,$$

$$\vec{c} = (3, 2, 1, 0, 0), \quad \vec{d} = (5, 4, 0, 0, 0)$$

$$a_2 = (\vec{c} * \vec{d})_2 = c_0 d_2 + c_1 d_1 + c_2 d_0 + {\color{red} c_3 d_4 + c_4 d_3}$$

'periodic' terms vanish (red), and $a_2 = 3 \cdot 0 + 2 \cdot 4 + 1 \cdot 5 = 13$.

Bonus application: multiplying numbers

$$cd = \sum_{k=0}^{2n} a_k 10^k, \qquad a_k = \sum_{j=0}^{k} c_j d_{k-j}$$

This gives a strange way to find cd:

Multiplication by FFT

Let c, d be n-digit integers. To compute cd, we can...

- Construct vectors $\vec{c}, \vec{d}$ of their digits, padded with zeros ($N = 2n$)
- Take the FFT of $\vec{c}$ nad $\vec{d}$ to get C, D
- Compute CD and then IFFT
  - the non-zero entries are then the coefficients a_k
- Finally, compute $\sum_{k=0}^{2n} a_k 10^k$, **round to an integer**

- The 'by hand' way: $O(n^2)$ operations
- This way: $O(n \log n)$ operations (sort of)!
- Unfortunate fact: this isn't really worth it except for very large n, plus rounding issues (and there are other ways to deal with large numbers...)