A simplification

The Fourier Series

The Fourier series

Analogy with projections

The component of the vector b along the line spanned by a is $b^{\mathrm{T}}a/a^{\mathrm{T}}a$.

. . .

A Fourier series is projecting $f(x)$ onto $\sin x$. Its component p in this direction is exactly $b_1 \sin x$.

Euler's formula

The complex exponential e^{ix} is a combination of $\cos x$ and $\sin x$:

$$e^{ix} = \cos x + i \sin x$$

. . .

So we can rewrite the Fourier series using complex exponentials:

$$f(x) = c_0 + c_1 e^{ix} + c_2 e^{2ix} + \cdots = \sum_k c_k \cdot e^{ikx}$$

pause

. . .

The formula for finding the coefficients c_k is the same as before, but now we use the complex exponential functions e^{ikx} instead of the sines and cosines:

$$c_k = \int_0^{2\pi} f(x) e^{-ikx} dx$$

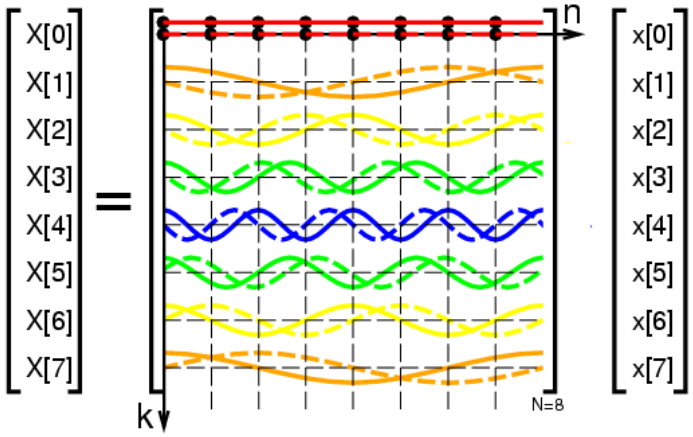
Discrete Fourier Transform

Discrete Fourier Series

DFT matrix

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{N-1} \end{bmatrix} = Fc = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & e^{-i2\pi/N} & e^{-i4\pi/N} & \dots & e^{-i2\pi(N-1)/N} \\ 1 & e^{-i4\pi/N} & e^{-i8\pi/N} & \dots & e^{-i4\pi(N-1)/N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & e^{-i2\pi(N-1)/N} & e^{-i4\pi(N-1)/N} & \dots & e^{-i2\pi(N-1)(N-1)/N} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{N-1} \end{bmatrix}$$

Visualization



Finding the Fourier coefficients

Inverse DFT

We have just shown two things:

1. The DFT can be written as a matrix-vector product $\mathbf{y} = F\mathbf{c}$.
2. The coefficients $\mathbf{c}$ can be found by $\mathbf{c} = F'\mathbf{y}$.
3. But of course it is also true from 1 that if F is invertible, that $\mathbf{c} = F^{-1}\mathbf{y}$. Therefore, $F^{-1} = F' = \frac{1}{N}F^*$.
4. This means that F is **unitary**. Its rows and columns are orthogonal.

A concrete example

Let's take an example with $N = 4$ and $y = (2, 4, 6, 8)$.

Make a table. Note that $i^2 = i^6 = -1$, $i^3 = i^7 = -i$, and $i^4 = 1$.

For n = 0:

k	$i2\pi k \times 0$	$e^{i2\pi k \times 0}$
0	0	1
1	0	1
2	0	1
3	0	1

. . .

For n = 1:

k	$i2\pi k\frac{1}{4}$	$e^{i2\pi k\frac{1}{4}}$
0	0	1

k	$i2\pi k\frac{1}{4}$	$e^{i2\pi k\frac{1}{4}}$
1	$i\pi/2$	i
2	$i\pi$	$-1 = i^2$
3	$3i\pi/2$	$-i = i^3$

For n = 2:

k	$i2\pi k\frac{2}{4}$	$e^{i2\pi k\frac{2}{4}}$
0	0	1
1	$i\pi$	$-1 = i^2$
2	$2i\pi$	$1 = i^4$
3	$3i\pi$	$-1 = i^6$

. . .

For n = 3:

k	$i2\pi k\frac{3}{4}$	$e^{i2\pi k\frac{3}{4}}$
0	0	1
1	$3i\pi/2$	$-i = i^3$
2	$3i\pi$	$-1 = i^6$
3	$9i\pi/2$	$i = i^9$

Solving for the coefficients

$$
\begin{aligned}
c_0 + c_1 + c_2 + c_3 &= 2 \\
c_0 + ic_1 + i^2c_2 + i^3c_3 &= 4 \\
c_0 + i^2c_1 + i^4c_2 + i^6c_3 &= 6 \\
c_0 + i^3c_1 + i^6c_2 + i^9c_3 &= 8
\end{aligned}
$$

. . .

In other words, $Fc = y$ for

$$F = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & i^2 & i^3 \\ 1 & i^2 & i^4 & i^6 \\ 1 & i^3 & i^6 & i^9 \end{bmatrix}$$

Check that $FF' = I$:

$$FF' = \frac{1}{4}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & i^2 & i^3 \\ 1 & i^2 & i^4 & i^6 \\ 1 & i^3 & i^6 & i^9 \end{bmatrix}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & (-i) & (-i)^2 & (-i)^3 \\ 1 & (-i)^2 & (-i)^4 & (-i)^6 \\ 1 & (-i)^3 & (-i)^6 & (-i)^9 \end{bmatrix} = I$$

Check in Sympy:

```
from sympy import *
init_printing()

i = I
F = Matrix([[1, 1, 1, 1], [1, i, i**2, i**3], [1, i**2, i**4, i**6], [1, i**3, i**6, i**9]])

print("F:")
display(F)

Fp = 1/4 * F.conjugate()
print("F prime:")
display(Fp)

print("F F prime:")
display(F * Fp)
```

F:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

`F prime:`

$$\begin{bmatrix} 0.25 & 0.25 & 0.25 & 0.25 \\ 0.25 & -0.25i & -0.25 & 0.25i \\ 0.25 & -0.25 & 0.25 & -0.25 \\ 0.25 & 0.25i & -0.25 & -0.25i \end{bmatrix}$$

`F F prime:`

$$\begin{bmatrix} 1.0 & 0 & 0 & 0 \\ 0 & 1.0 & 0 & 0 \\ 0 & 0 & 1.0 & 0 \\ 0 & 0 & 0 & 1.0 \end{bmatrix}$$

Simplest form of Fourier matrix

Then we can write the Fourier equation as

$$\begin{bmatrix} 1 & 1 & 1 & \cdot & 1 \\ 1 & w & w^2 & \cdot & w^{n-1} \\ 1 & w^2 & w^4 & \cdot & w^{2(n-1)} \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ 1 & w^{n-1} & w^{2(n-1)} & \cdot & w^{(n-1)^2} \end{bmatrix} \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \cdot \\ c_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \cdot \\ y_{n-1} \end{bmatrix}, \quad w = e^{i2\pi/n}$$

Terminology

We call the matrix F the **DFT (Discrete Fourier Transform) matrix**. It is a square unitary matrix of size $N \times N$.

. . .

The matrix F' is the **inverse DFT matrix**. It is the complex conjugate of F divided by N.

Summary

1. The DFT can be written as a matrix-vector product $\mathbf{y} = F\mathbf{c}$.
2. The coefficients $\mathbf{c}$ can be found by $\mathbf{c} = F'\mathbf{y}$.
3. F is easy to compute, with a simple formula for each entry.
4. (There's actually a really really fast way to compute the DFT using the FFT algorithm.)

Applications of the DFT

Filtering

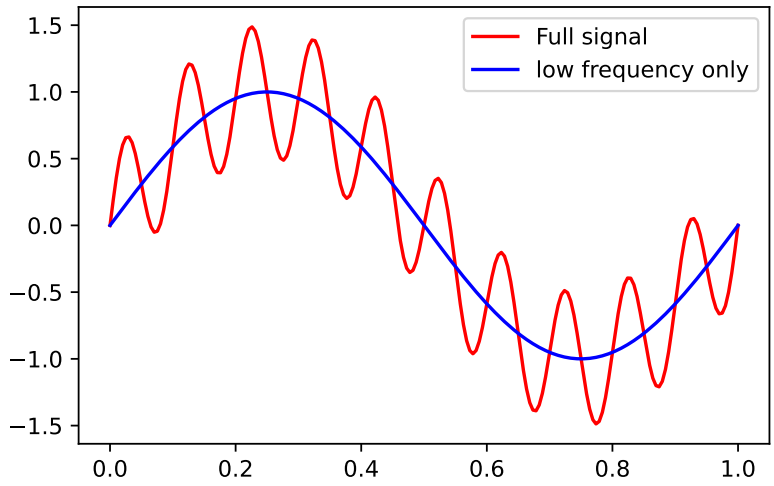
The DFT is used in many applications, but one of the most common is **filtering**.

. . .

For example, suppose we have a signal that is a sum of two sinusoids:

$$f(x) = \sin(2\pi x) + 0.5\sin(20\pi x)$$

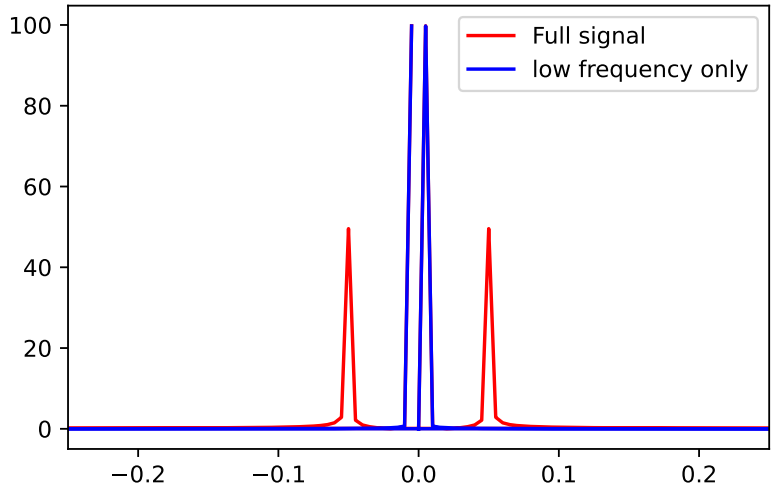
```
import numpy as np
import matplotlib.pyplot as plt

N = 200
x = np.linspace(0, 1, N)
y = np.sin(2*np.pi*x) + 0.5*np.sin(20*np.pi*x)
yalone = np.sin(2*np.pi*x)

plt.plot(x, y, label='full signal', color='red')
plt.plot(x, yalone, label='low frequency only', color='blue')
plt.legend(['Full signal', 'low frequency only'])
plt.show()
```



What's the DFT of this signal?

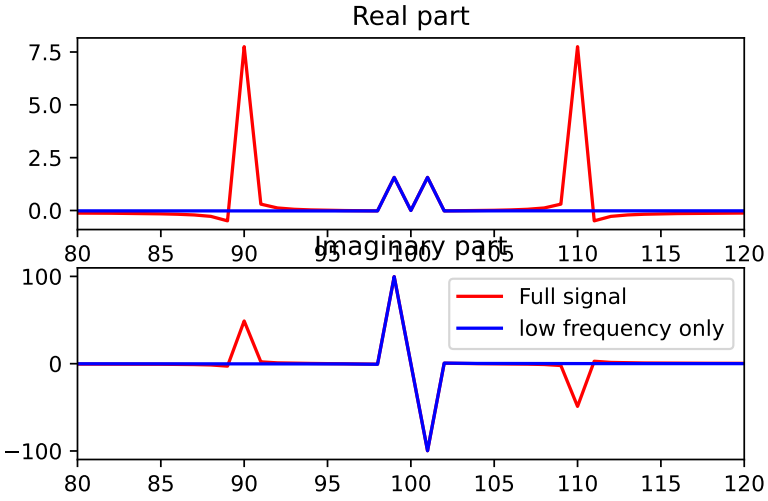
```
from scipy.fft import fft, fftshift
yf = fft(y)
yfalone = fft(yalone)
# make a non connected plot of yf

plt.plot(np.fft.fftfreq(200),np.abs(yf), color='red')
plt.plot(np.fft.fftfreq(200),np.abs(yfalone), color='blue')
```

```
plt.xlim(-0.25, 0.25)
plt.legend(['Full signal', 'low frequency only'])
plt.show()
```



What's with the double peaks?

```
# make two subplots
ax1 = plt.subplot(2,1, 1)
ax2 = plt.subplot(2,1, 2)

ax1.plot(np.real(fftshift(yf)), color='red')
ax1.plot(np.real(fftshift(yfalone)),  color='blue')
ax1.set_xlim(80, 120)
ax1.set_title("Real part")
ax2.plot(np.imag(fftshift(yf)), color='red')
ax2.plot(np.imag(fftshift(yfalone)),  color='blue')
ax2.set_title("Imaginary part")

# set x-axis to be from -100 to 100
plt.xlim(80, 120)
plt.legend(['Full signal', 'low frequency only'])
```

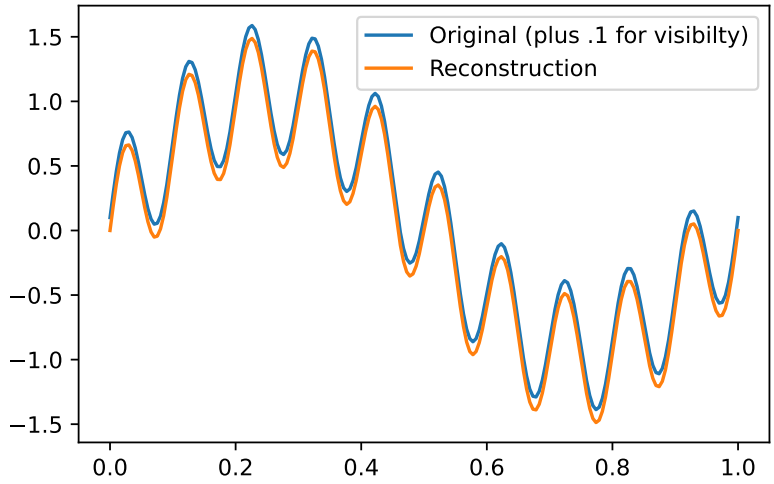


Reconstruct the signal

We can reconstruct the signal by taking the inverse DFT.

```
from scipy.fft import ifft

y_old = ifft(yf)

plt.plot(x, y+.1, label='Original (plus .1 for visibilty)')
plt.plot(x, y_old, label='Reconstruction')
plt.legend()
plt.show()
```

```
/Users/kendra/Library/Python/3.8/lib/python/site-packages/matplotlib/cbook/__init__.py:1345: Co

Casting complex values to real discards the imaginary part
```

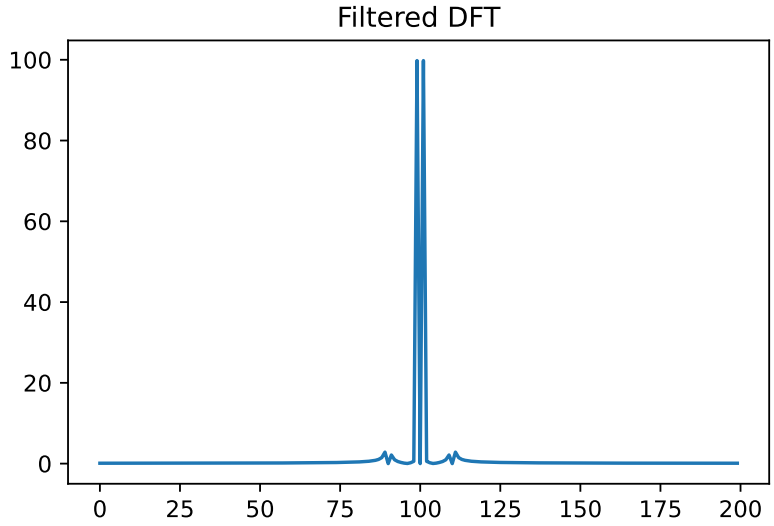


Low-pass filtering

What if we just zero out the coefficient corresponding to the second sinusoid:

. . .

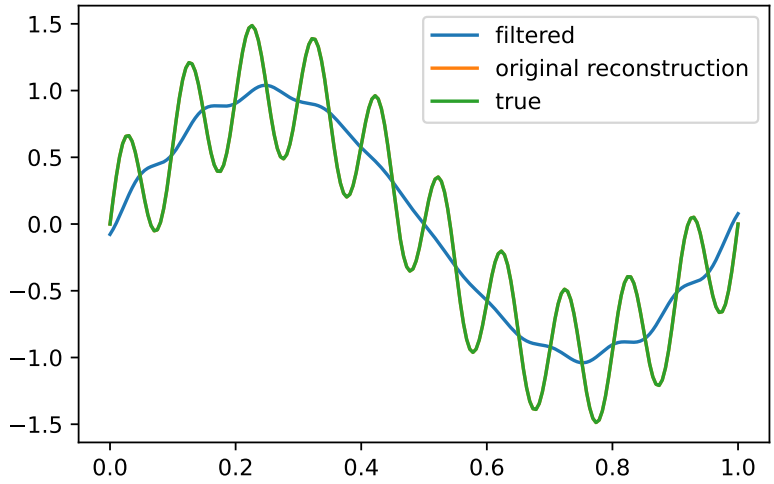
```
yf_new = yf.copy()
yf_new[190] = 0
yf_new[10] = 0
plt.plot(np.abs(fftshift(yf_new)))
plt.title('Filtered DFT')
plt.show()
```



Compute the inverse DFT:

```
from scipy.fft import ifft

y_new = ifft(yf_new)
y_old = ifft(yf)

plt.plot(x, y_new, label='filtered')
plt.plot(x, y_old, label='original reconstruction')
plt.plot(x,y, label='true')
plt.legend()
plt.show()
```

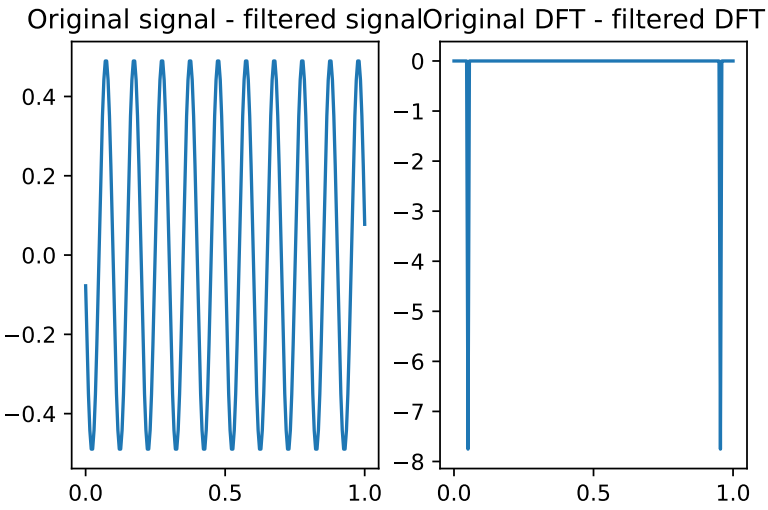
```
/Users/kendra/Library/Python/3.8/lib/python/site-packages/matplotlib/cbook/__init__.py:1345: C

Casting complex values to real discards the imaginary part
```



It's pretty good!

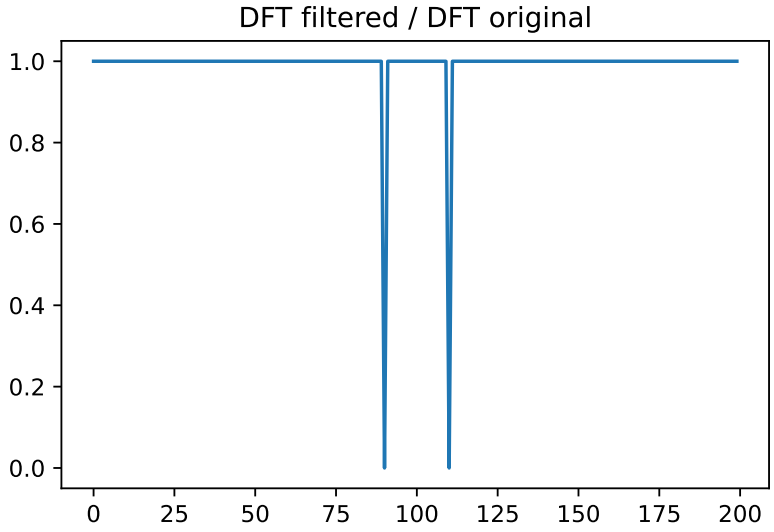
```
# find the ifft of the function with the zeroed out coefficients
change = ifft(yf_new-yf)

# make two subplots
ax1 = plt.subplot(1,2, 1)

ax1.plot(x, change)
ax1.set_title('Original signal - filtered signal')

ax2 = plt.subplot(1,2, 2)
ax2.plot(x,yf_new-yf)
ax2.set_title('Original DFT - filtered DFT')
plt.show()
```

```
/Users/kendra/Library/Python/3.8/lib/python/site-packages/matplotlib/cbook/__init__.py:1345: C

Casting complex values to real discards the imaginary part
```



Try one more thing. Can we describe the change in the DFT as a multiplicative vector? We sure can.

```
change = yf_new/yf
plt.plot(np.abs(fftshift(change)))
plt.title('DFT filtered / DFT original')
plt.show()
```



So we can describe the new DVD as the old DFT multiplied by a vector of 1s and 0s.

The total process went like this:

1. Take the DFT of the signal.
2. Multiply the DFT by a vector of 1s and 0s to filter out the high frequency.
3. Take the inverse DFT to get the filtered signal.

. . .

Filtering as convolution

Suppose we have a signal and we'd like to try to filter out the high frequencies, but we don't know which ones they are.

We could try with a simple filter like $[1, 1, 1, 1, 1]/5$. This is a simple moving average filter.

That means that we replace each point in the signal with the average of the 5 points before it.

Mathematically, this is:

$$f_{\text{filtered}}[n] = \sum_{m=0}^{4} f[n-m]/5$$

OK. We note that this can also be written as a convolution, between the signal and the vector $h = [1, 1, 1, 1, 1, 0, 0, 0, ...]/5.$, where we have padded the vector with zeros so that it is the same length as the signal vector.

That is, $h_0 = 1/5$, $h_1 = 1/5$, $h_2 = 1/5$, $h_3 = 1/5$, $h_4 = 1/5$, and $h_5 = 0$, all the way to $h_n = 0$.

Then we can write this as:

$$f_{\text{filtered}}[n] = \sum_{m=0}^{n} f[n-m]h[m]$$

This operation is called **convolution**.

. . .

It's messy to compute the convolution directly. But we can do it in the Fourier domain…

The convolution theorem

The convolution of two signals f and h is the inverse DFT of the product of the DFTs of f and h.

. . .

In other words, if $f = \text{ifft}(F)$ and $h = \text{ifft}(H)$, then the convolution of f and h is $\text{ifft}(F \cdot H)$.

. . .

That's much easier to calculate – only a dot product!

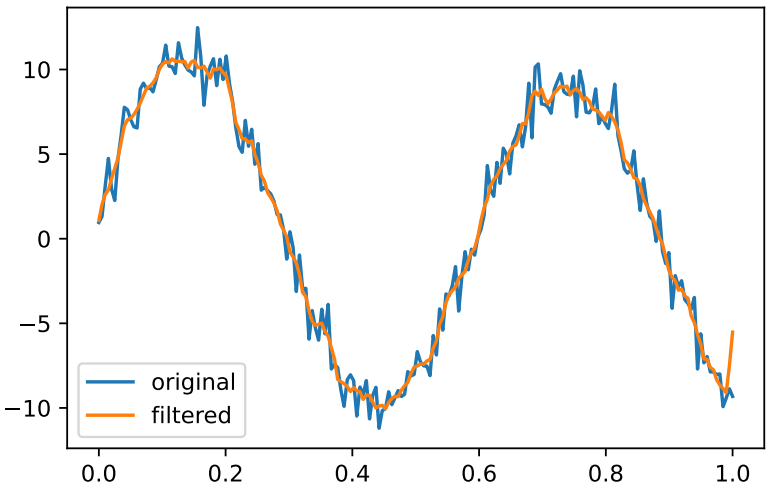
Proof

(fill in if I have time)

Example

Let's take the signal from before, $f(x) = \sin(2\pi x) + 0.5\sin(20\pi x)$, and filter it with the moving average filter $h = [1, 1, 1, 1, 1]/5$.

```
from scipy.signal import convolve
N = 200
x = np.linspace(0, 1, N)
y = np.sin(2*np.pi*x) + 10*np.sin(3.3*np.pi*x+.2)+ 10*np.random.normal(0, 0.1, N)
h = np.array([1, 1,1,1,1])/5
y_filtered = convolve(y, h, mode='same')
plt.plot(x, y, label='original')
plt.plot(x, y_filtered, label='filtered')
plt.legend()
plt.show()
```



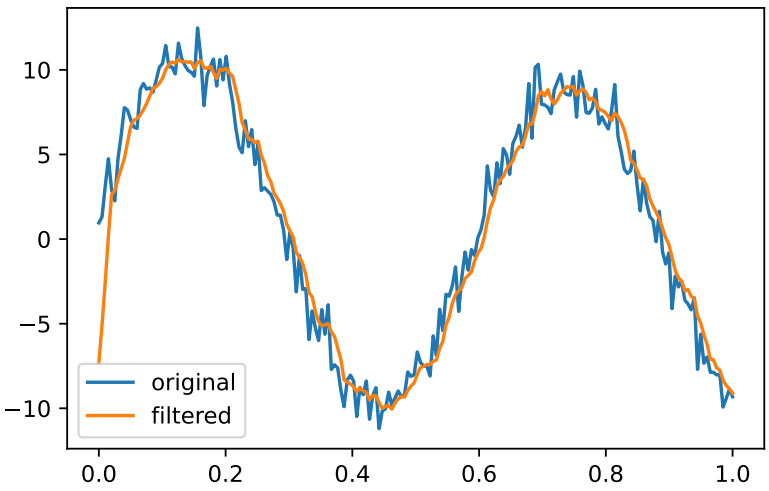
Now try it again using the convolution theorem.

Steps:

1. Take the DFT of the signal.
2. Pad the filter with zeros to make it the same length as the signal.
3. Take the DFT of the filter.
4. Multiply the two DFTs.
5. Take the inverse DFT to get the filtered signal.
6. Done!

```
yf = fft(y)
hf = fft(h, N) # pad h with zeros
y_filtered = ifft(yf*hf)
plt.plot(x, y, label='original')
plt.plot(x, y_filtered, label='filtered')
plt.legend()
plt.show()
```

```
/Users/kendra/Library/Python/3.8/lib/python/site-packages/matplotlib/cbook/__init__.py:1345: C

Casting complex values to real discards the imaginary part
```



Then we can take the inverse DFT to get the filtered signal.