Fourier transforms and convolution

(without the agonizing pain)

CS/CME/BioE/Biophys/BMI 279

Oct. 26, 2017

Ron Dror

Outline

- Why do we care?
- Fourier transforms
  - Writing functions as sums of sinusoids
  - The Fast Fourier Transform (FFT)
  - Multi-dimensional Fourier transforms
- Convolution
  - Moving averages
  - Mathematical definition
  - Performing convolution using Fourier transforms

Fourier transforms have a massive range of applications. For this reason, FFT is arguably the most important algorithm of the past century!

Why do we care?

Why study Fourier transforms and convolution?

- In the remainder of the course, we'll study several methods that depend on analysis of images or reconstruction of structure from images:
  – Light microscopy (particularly fluorescence microscopy)
  – Electron microscopy (particularly for single-particle reconstruction)
  – X-ray crystallography
- The computational aspects of each of these methods involve Fourier transforms and convolution
- These concepts are also important for:
  – Some approaches to ligand docking (and protein-protein docking)
  – Fast evaluation of electrostatic interactions in molecular dynamics
  – (You're not responsible for these additional applications)

Fourier transforms

Remember - what is a function?
The simplest idea is that you have an input “x” and an output “f(x)” and for each input you have a unique output.
How do functions get stored on a computer? For simple functions like $f(x) = x^2$, you can represent the function by taking x, multiplying it by itself, and then returning that product. But what about for arbitrary functions, such as the temperature of the room as a function of time? One way to encode the function on the computer is to discretize it — store its values at regular intervals — and assume all the other values in between vary smoothly between these discretized landmarks.

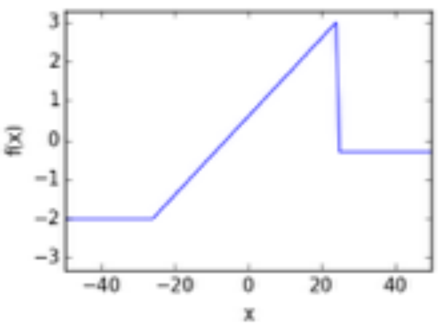
Fourier transforms

Writing functions as sums of sinusoids

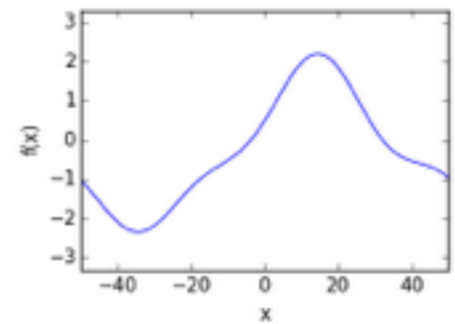
Writing functions as sums of sinusoids

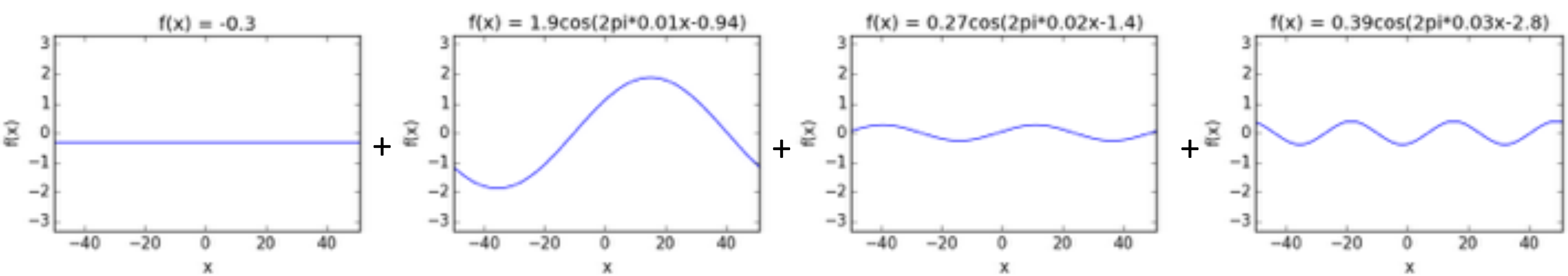
- Given a function defined on an interval of length *L*, we can write it as a sum of sinusoids whose periods are *L, L/2, L/3, L/4*, **…** (plus a constant term)

Original function



Sum of sinusoids below





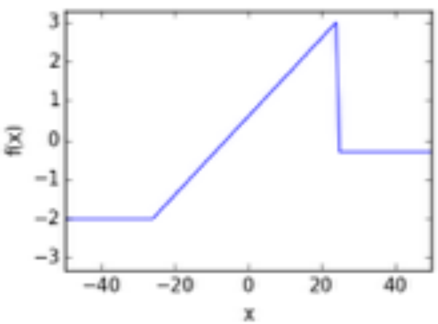
Decreasing period

Increasing frequency

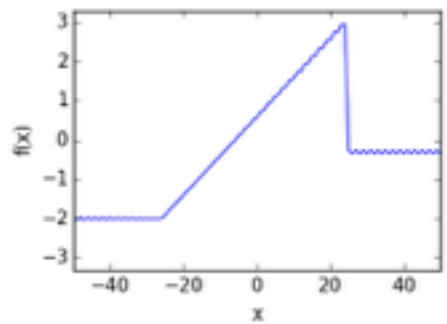
Writing functions as sums of sinusoids

- Given a function defined on an interval of length *L*, we can write it as a sum of sinusoids whose periods are *L, L/2, L/3, L/4*, … (plus a constant term)

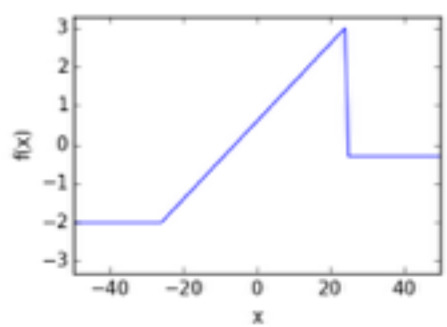
Original function



sum of 49 sinusoids (plus constant term)



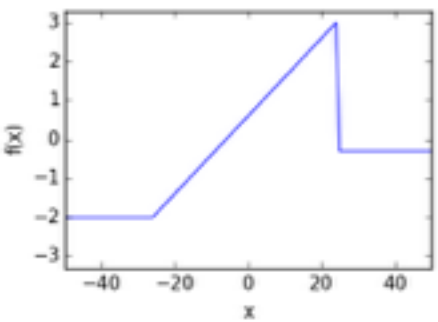
sum of 50 sinusoids (plus constant term)



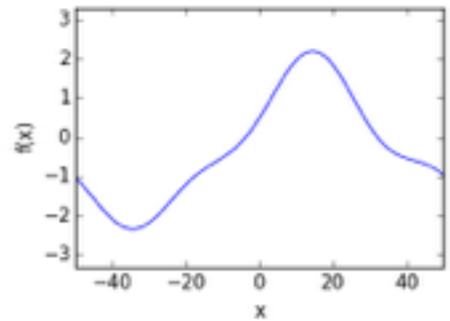
Writing functions as sums of sinusoids

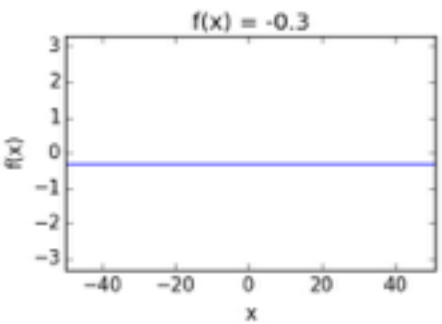
- Each of these sinusoidal terms has a magnitude (scale factor) and a phase (shift).

Original function

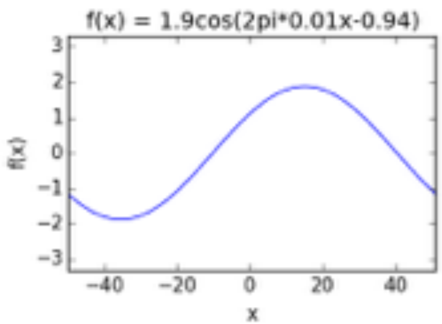


Sum of sinusoids below

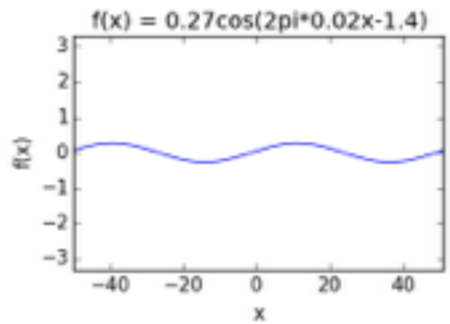




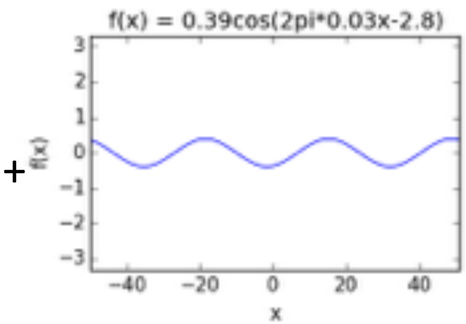
+



+



+



Magnitude: -0.3
Phase: 0

Magnitude: 1.9
Phase: -.94

Magnitude: 0.27
Phase: -1.4

Magnitude: 0.39
Phase: -2.8

Magnitude: height of the wave (scale factor) - it is the coefficient in front of the cos(x) function
Phase: left-right shift, determines where the peaks are - it is the constant term inside the cos(x)
Period: the distance between consecutive peaks of the wave - it is related to the inverse of the coefficient of x inside cos(x)

Expressing a function as a set of sinusoidal term coefficients

- We can thus express the original function as a series of magnitude and phase coefficients
  - If the original function is defined at *N* equally spaced points, we'll need a total of *N* coefficients

    i.e. the fourier transform with N coefficients will pass through the N original points.

  - If the original function is defined on for an infinite set of inputs, we'll need an infinite series of magnitude and phase coefficients—but we can approximate the function with just the first few

Constant term <u>(frequency 0)</u>	Sinusoid 1 <u>(period *L*, frequency 1/*L*)</u>	Sinusoid 2 <u>(period *L*/2, frequency 2/*L*)</u>	Sinusoid 3 <u>(period *L*/3, frequency 3/*L*)</u>
Magnitude: -0.3	Magnitude: 1.9	Magnitude: 0.27	Magnitude: 0.39
Phase: 0 (arbitrary)	Phase: -.94	Phase: -1.4	Phase: -2.8

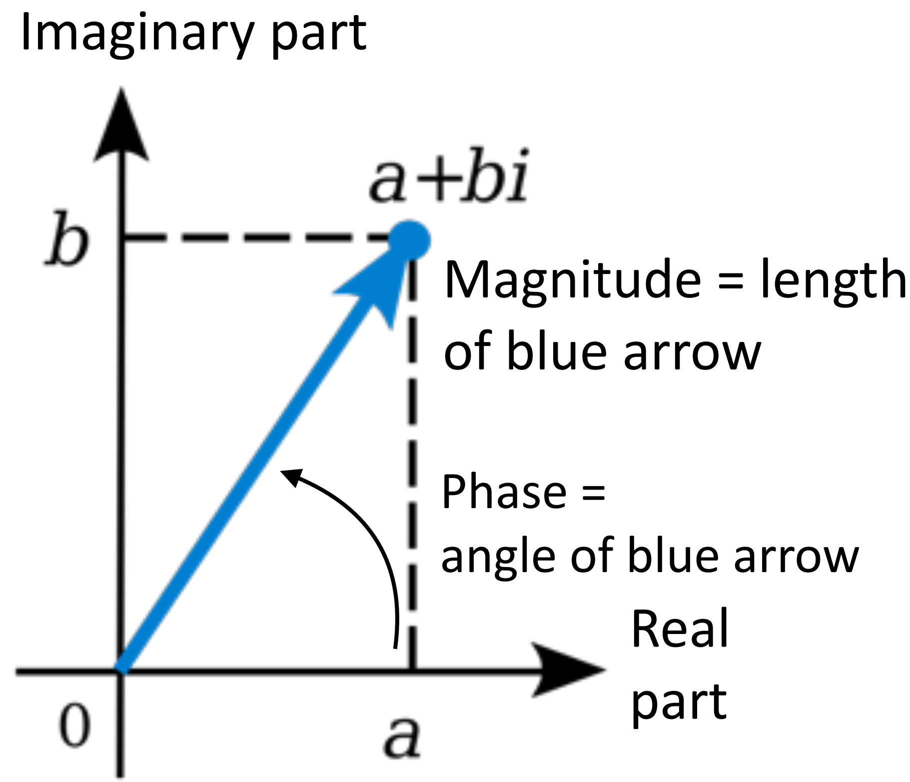
Using complex numbers to represent magnitude plus phase

- We can express the magnitude and phase of each sinusoidal component using a complex number

This is like using polar coordinates, which you may have learned in high school. In that case, polar coordinates (r, theta) and regular cartesian coordinates (x, y) are related by

x = r cos(theta)
y = r sin(theta)

Polar coordinates represent the magnitude (r) and phase (theta) while cartesian coordinates (x, y) are like the complex numbers a + bi.



Using complex numbers to represent magnitude plus phase

- We can express the magnitude and phase of each sinusoidal component using a complex number
- Thus we can express our original function as a series of complex numbers representing the sinusoidal components
  – This turns out to be more convenient (mathematically and computationally) than storing magnitudes and phases

The Fourier transform

- The Fourier transform maps a function to a set of complex numbers representing sinusoidal coefficients
  - We also say it maps the function from “real space” to “Fourier space” (or “frequency space”)
  - Note that in a computer, we can represent a function as an array of numbers giving the values of that function at equally spaced points.
- The inverse Fourier transform maps in the other direction
  - It turns out that the Fourier transform and inverse Fourier transform are almost identical. A program that computes one can easily be used to compute the other.

Why do we want to express our function using *sinusoids*?

- Sinusoids crop up all over the place in nature
  - For example, sound is usually described in terms of different frequencies

e.g. a pure tone can be described by its single frequency; complex sounds like speech can be described by the strength of each frequency across the whole range of human hearing.

- Sinusoids have the unique property that if you sum two sinusoids of the same frequency (of any phase or magnitude), you always get another sinusoid of the same frequency
  - This leads to some very convenient computational properties that we'll come to later

Fourier transforms

The Fast Fourier Transform (FFT)

The Fast Fourier Transform (FFT)

This is a particular algorithm to compute the Fourier transform.

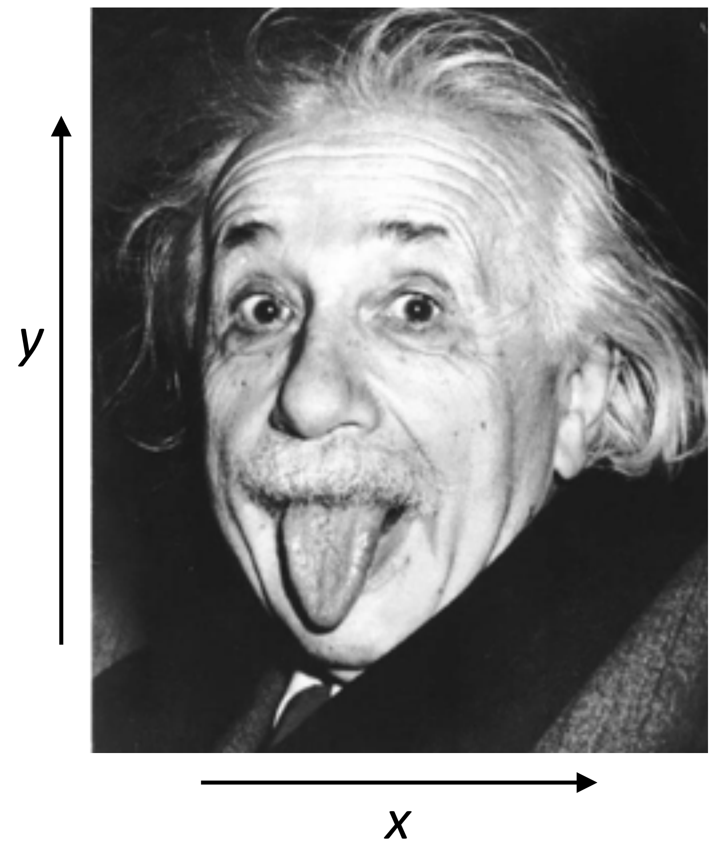
- The number of arithmetic operations required to compute the Fourier transform of N numbers (i.e., of a function defined at N points) in a straightforward manner is proportional to N^2
- Surprisingly, it is possible to reduce this N^2 to $N \log N$ using a clever algorithm
  - This algorithm is the Fast Fourier Transform (FFT)
  - It is arguably the most important algorithm of the past century
  - You do not need to know how it works—only that it exists

Fourier transforms

Multidimensional Fourier Transforms

Images as functions of two variables

- Many of the applications we'll consider involve images
- A grayscale image can be thought of as a function of two variables
  - The position of each pixel corresponds to some value of *x* and *y*
  - The brightness of that pixel is proportional to *f*(*x*,*y*)



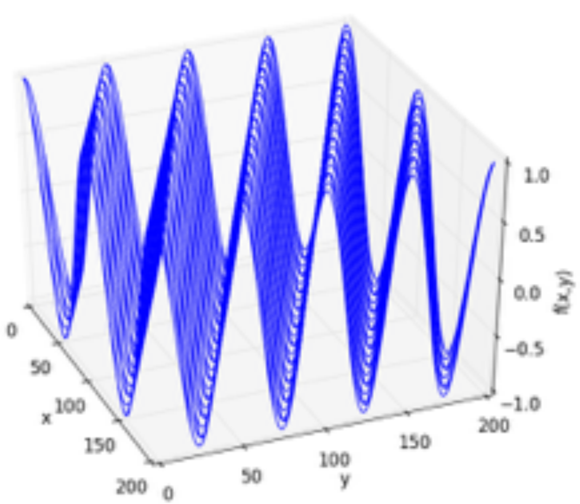
e.g. f(10, 100) = 0 means that the pixel at (10, 100) is perfectly black (brightness of 0)

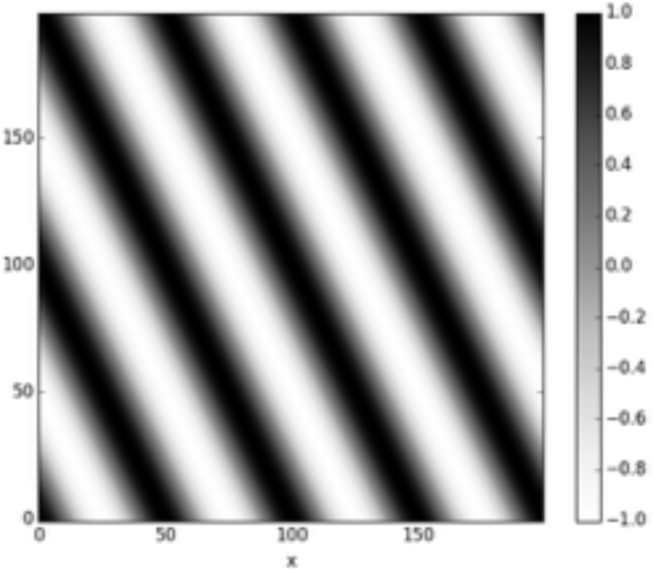
Two-dimensional Fourier transform

- We can express functions of two variables as sums of sinusoids
- Each sinusoid has a frequency in the *x*-direction and a frequency in the *y*-direction
- We need to specify a magnitude and a phase for each sinusoid
- Thus the 2D Fourier transform maps the original function to a complex-valued function of two frequencies

x frequency value is 0.02 and y frequency value is 0.01. Together, the two frequency values give a 2D vector that determines the direction and the speed of fluctuation in 2D.

$$f(x,y) = \sin(2\pi \cdot 0.02x + 2\pi \cdot 0.01y)$$





These two plots are the same function. On the right image, the “z” value is represented by pixel brightness

Three-dimensional Fourier transform

- The 3D Fourier transform maps functions of three variables (i.e., a function defined on a volume) to a complex-valued function of three frequencies
- Multidimensional Fourier transforms can also be computed efficiently using the FFT algorithm

Convolution

Convolution

Moving averages

Convolution generalizes the notion of a (weighted) *moving average*

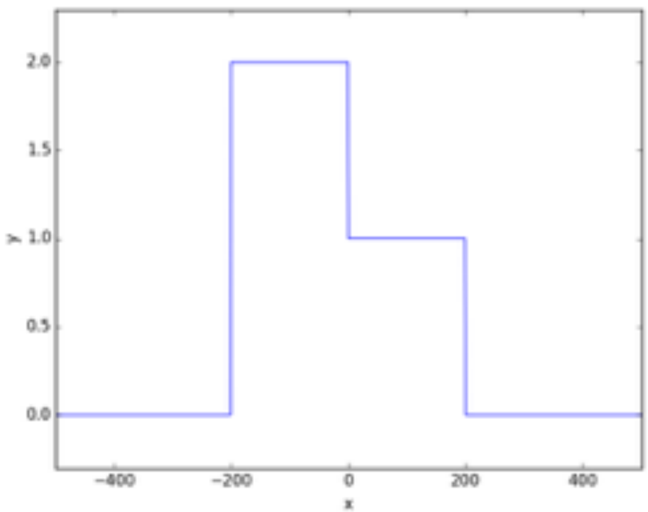
- We're given an array of numerical values
  - We can think of this array as specifying values of a function at regularly spaced intervals
- To compute a moving average, we replace each value in the array with the average of several values that precede and follow it (i.e., the values within a *window*)
- We might choose instead to calculate a *weighted moving average*, where we again replace each value in the array with the average of several surrounding values, but we weight those values differently
- We can express this as a *convolution* of the original function (i.e., array) with another function (array) that specifies the weights on each value in the window

See also: notes on convolution posted online

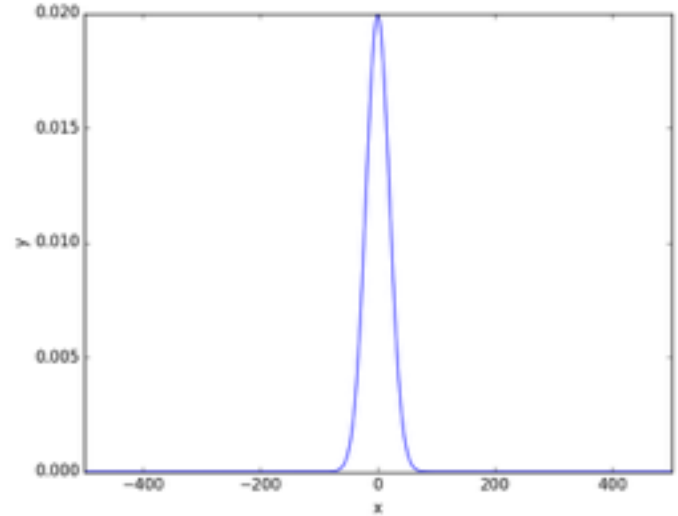
Example

e.g. g(x) specifies the weights for the moving average

f

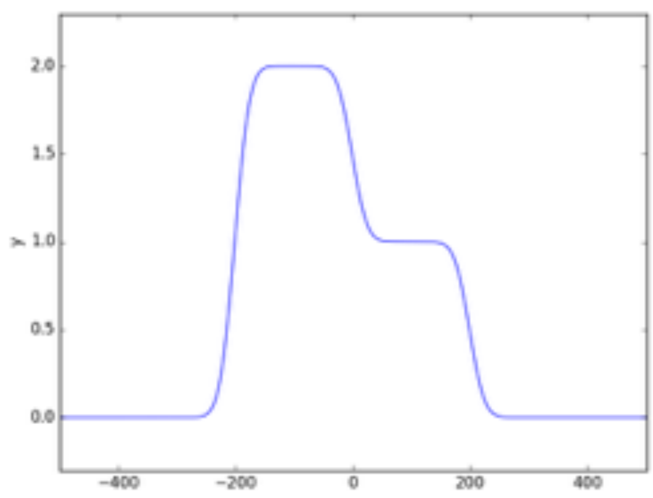


g



f convolved with g (written f∗g)

Note: convolution is commutative!
So f * g = g * f



Convolution

Mathematical definition

Convolution: mathematical definition

- If *f* and *g* are functions defined at evenly spaced points, their convolution is given by:

$$(f*g)[n]=\sum_{m=-\infty}^{\infty} f[m]g[n-m]$$

In practice, usually f(x) or g(x) is 0 for extremely large or small values of x, making this infinite sum easier to compute.

Convolution

Multidimensional convolution

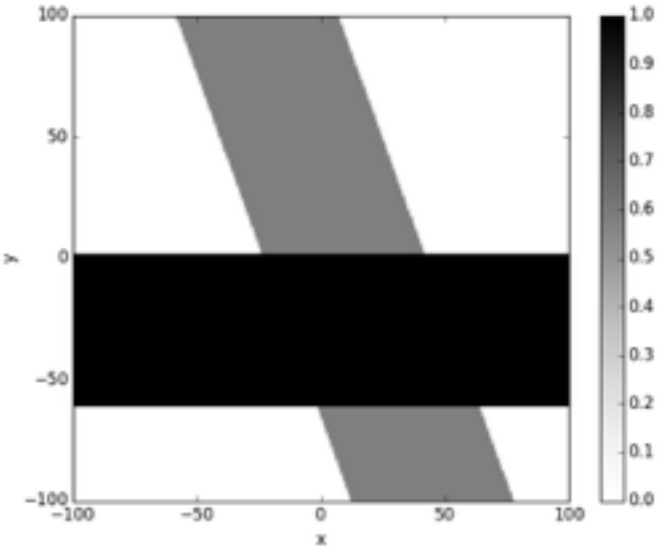
Two-dimensional convolution

- In two-dimensional convolution, we replace each value in a two-dimensional array with a weighted average of the values surrounding it in two dimensions
  - We can represent two-dimensional arrays as functions of two variables, or as matrices, or as images

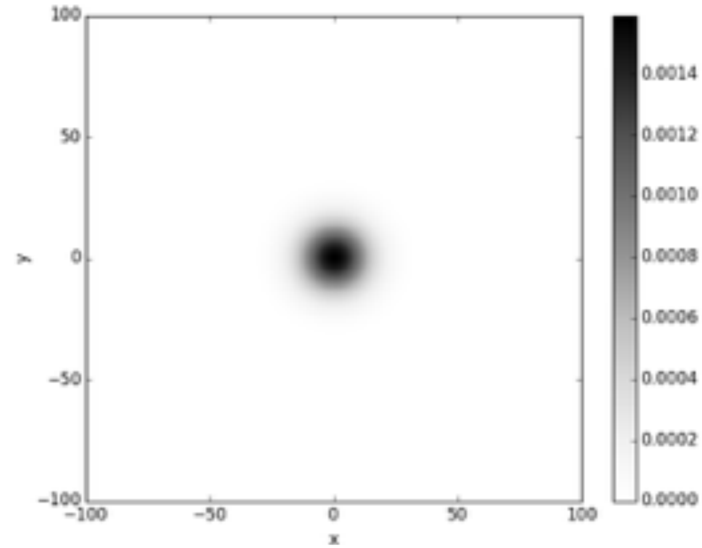
Two-dimensional convolution: example

f and g are functions of two variables, displayed as images, where pixel brightness represents the function value.

f

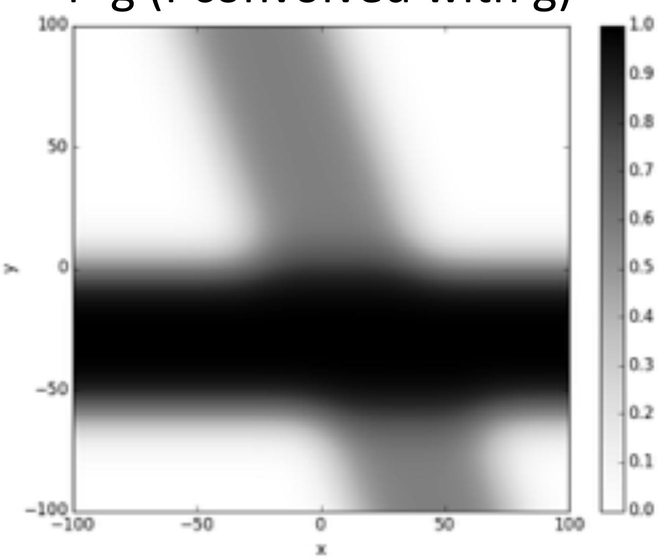


g



g(x,y) is a 2D Gaussian function

f∗g (f convolved with g)



Question: can you invert the convolution, or “deconvolve”? i.e. given g and f*g can you recover f?

Answer: this is a very important question. Sometimes you can recover the original function, but sometimes you can’t. In this example, convolving by g causes blurring and loss of information, so you can’t perfectly reproduce f. Practically, this would be a very useful thing to do, because f * g could represent a microscopy image that has been blurred by g through the process of taking the image, and what you really want is to recover the underlying structure f.

Multidimensional convolution

- The concept generalizes to higher dimensions
- For example, in three-dimensional convolution, we replace each value in a three-dimensional array with a weighted average of the values surrounding it in three dimensions

Convolution

Performing convolution using Fourier transforms

Relationship between convolution and Fourier transforms

- It turns out that convolving two functions is equivalent to *multiplying* them in the frequency domain
  - One multiplies the complex numbers representing coefficients at each frequency
- In other words, we can perform a convolution by taking the Fourier transform of both functions, multiplying the results, and then performing an inverse Fourier transform

The reason why this works is related to the fact that two sinusoids of the same frequency will sum to something with the same frequency again.

Why does this relationship matter?

- First, it allows us to perform convolution faster
  - If two functions are each defined at N points, the number of operations required to convolve them in the straightforward manner is proportional to N^2
  - If we use Fourier transforms and take advantage of the FFT algorithm, the number of operations is proportional to $N \log N$
- Second, it allows us to characterize convolution operations in terms of changes to different frequencies
  - For example, convolution with a Gaussian will preserve low-frequency components while reducing high-frequency components

This is why you can't always deconvolve functions perfectly - these high-frequency components are lost.