COMP3204/COMP6223: Computer Vision

Programming for computer vision & other musings related to the coursework

Jonathon Hare
jsh2@ecs.soton.ac.uk

Topics for Discussion

- Writing code to do computer vision
- Convolution
  - Fourier domain convolution & correlation
  - Template convolution
  - Gaussian Filtering
  - “Ideal” filters; constructing a HP filter from a LP one
  - Output of HP filters
- Building Hybrid Images

Writing code for computer vision

Image Storage

- Images usually stored as arrays of integers
  - Typically 8-bits per pixel per channel
    - 12-16 bit increasingly common (e.g. HDR imaging)
    - Uses unsigned pixel values
  - Compressed using a variety of techniques
    - Lossy or lossless

Most vision algorithms are continuous

- E.g. convolution with a continuous function (i.e. Gaussian)
- If we were writing the next Adobe Photoshop, it would be important that we kept out images in a similar format (integer pixels, same number of bits)
  - We would essentially round pixel values to the closest integer and clip those out of range
- For vision applications we don't want to do this as we'll lose precision

Always work with floating point pixels

- Unless they've been specifically optimised for integer math, all vision algorithms should use floating point pixel values
  - Ensure the best possible discretisation from operations involving continuous functions
    - Higher effective bit depth (32/64 bits per pixel per band)
    - Ability to deal with negative values
      - Turns out to be very important for convolution!
    - Ability to deal with numbers outside of the normal range
      - Just because a pixel has a grey level of 1.1 doesn't mean it's invalid, just that it's too bright to be displayed in the normal colour gamut.

Aside: arithmetic in MATLAB

- Guidelines for writing vision code:
  - Convert any images to float types immediately once you’ve read them
  - Don’t convert them back to integer types until you need to (i.e. for display or saving)
    - Be mindful that a meaningful conversion might not just involve rounding if you want to preserve the data.

Convolution

- Convolution is an element-wise multiplication in the Fourier domain (*c.f. Convolution Theorem*)
    - $f * g$ = ifft(fft(f) . fft(g))
    - Whilst S and F might only contain real numbers, the FFTs are complex (*real* + *imag*j)
        - Need to do **complex multiplication**!

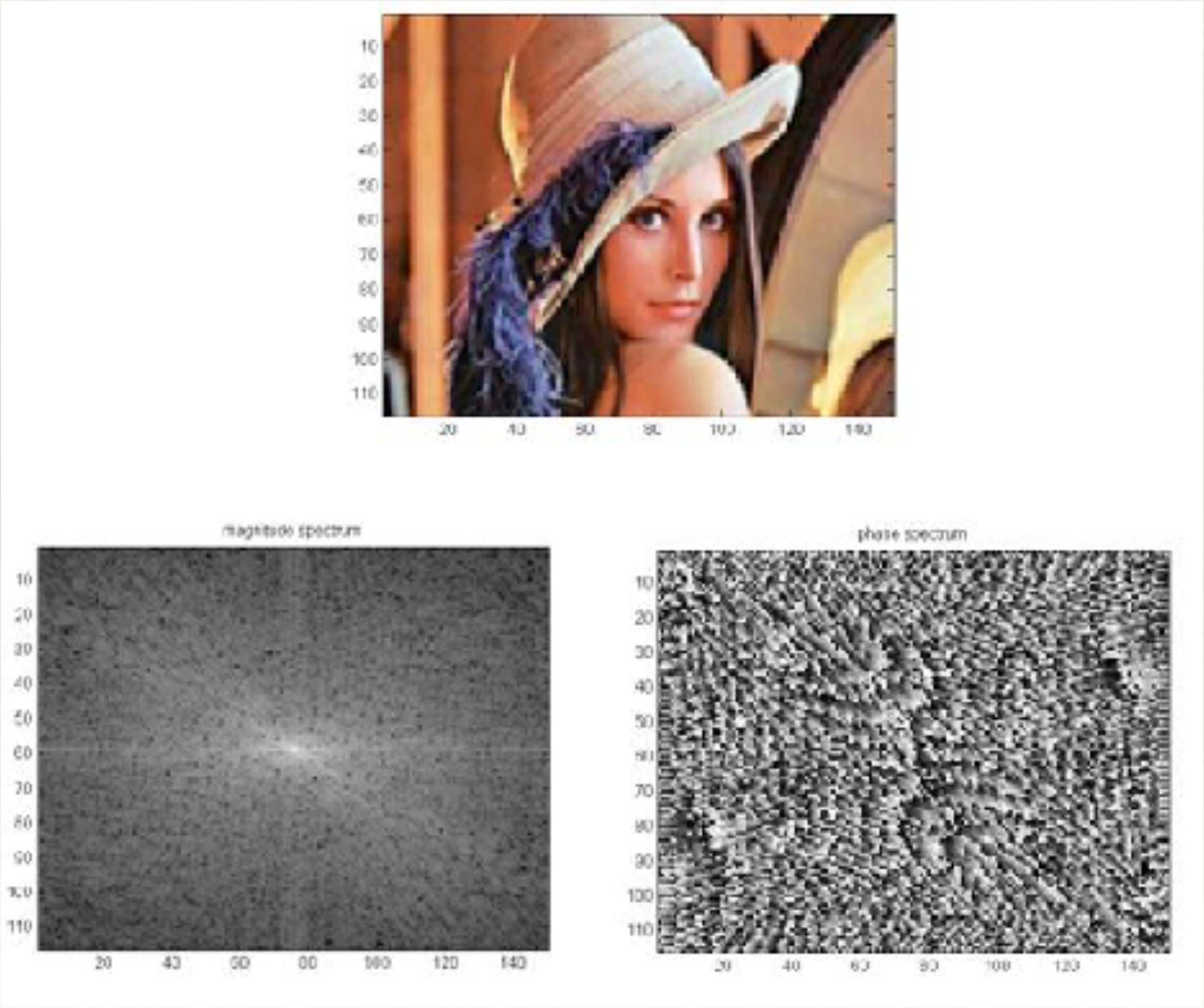
$$\boxed{(x+yi)(u+vi) = (xu-yv) + (xv+yu)i}$$

Aside: phase and magnitude

- Given a complex number ($n = real + imag$j) from an FFT we can compute its **phase** and **magnitude**
  - phase = atan2($imag$, $real$)
  - magnitude = sqrt($real$*$real$ + $imag$*$imag$)
- We might perform this transformation to display the FFT as it conceptually helps us understand what the FFT is doing
- We can't use this representation to perform convolution however (need to transform back to complex form first)

Aside: Displaying FFTs

- FFTs are often re-ordered so that the DC component (0-frequency) component is in the centre:



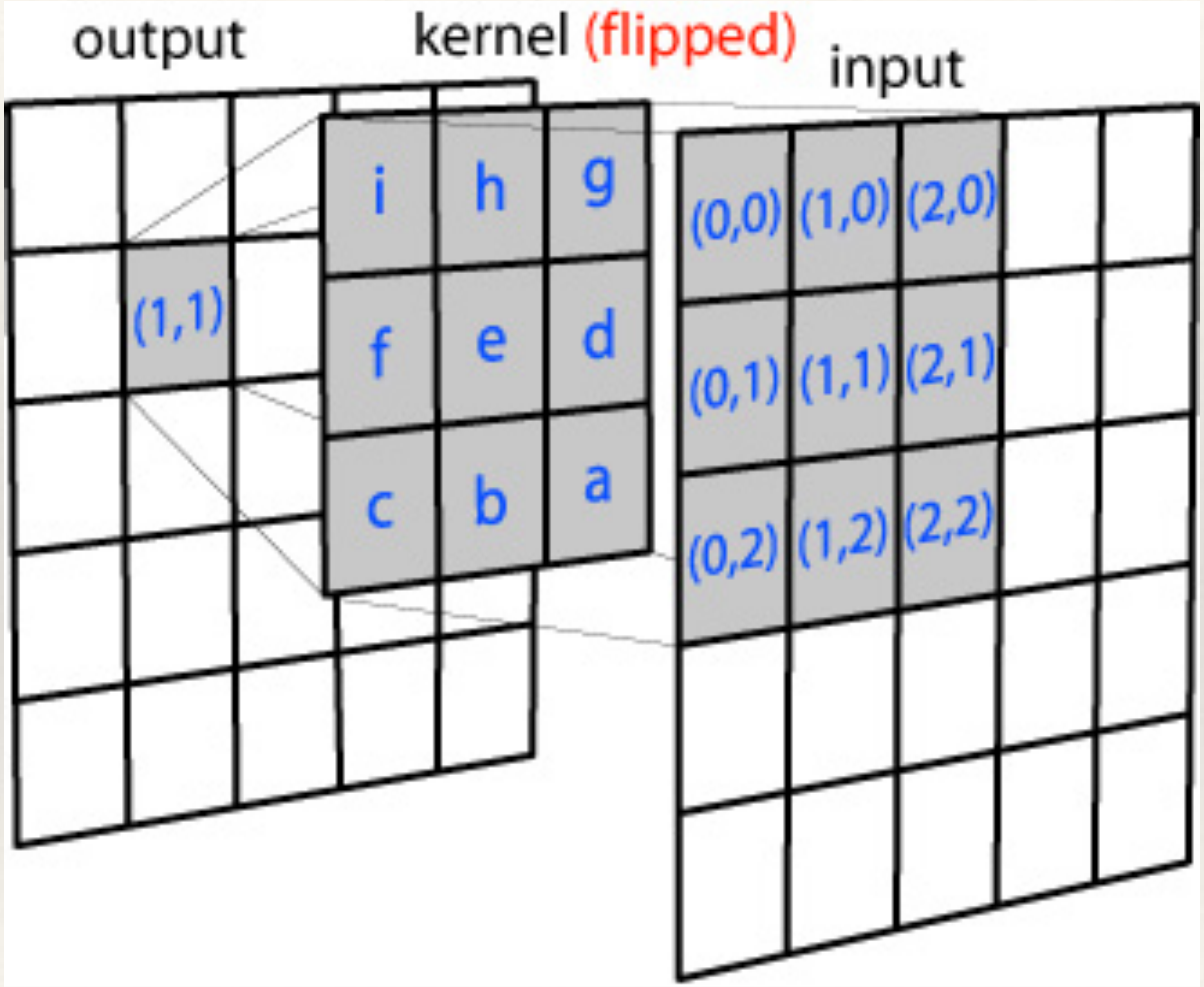
Template Convolution

- In the time domain, convolution is:

$$(f * g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau)\, g(t-\tau)\, d\tau$$
$$= \int_{-\infty}^{\infty} f(t-\tau)\, g(\tau)\, d\tau.$$

- **Notice that the image or kernel is “flipped” in time**
  - Also notice that the is no normalisation or similar

Template Convolution



Template Convolution

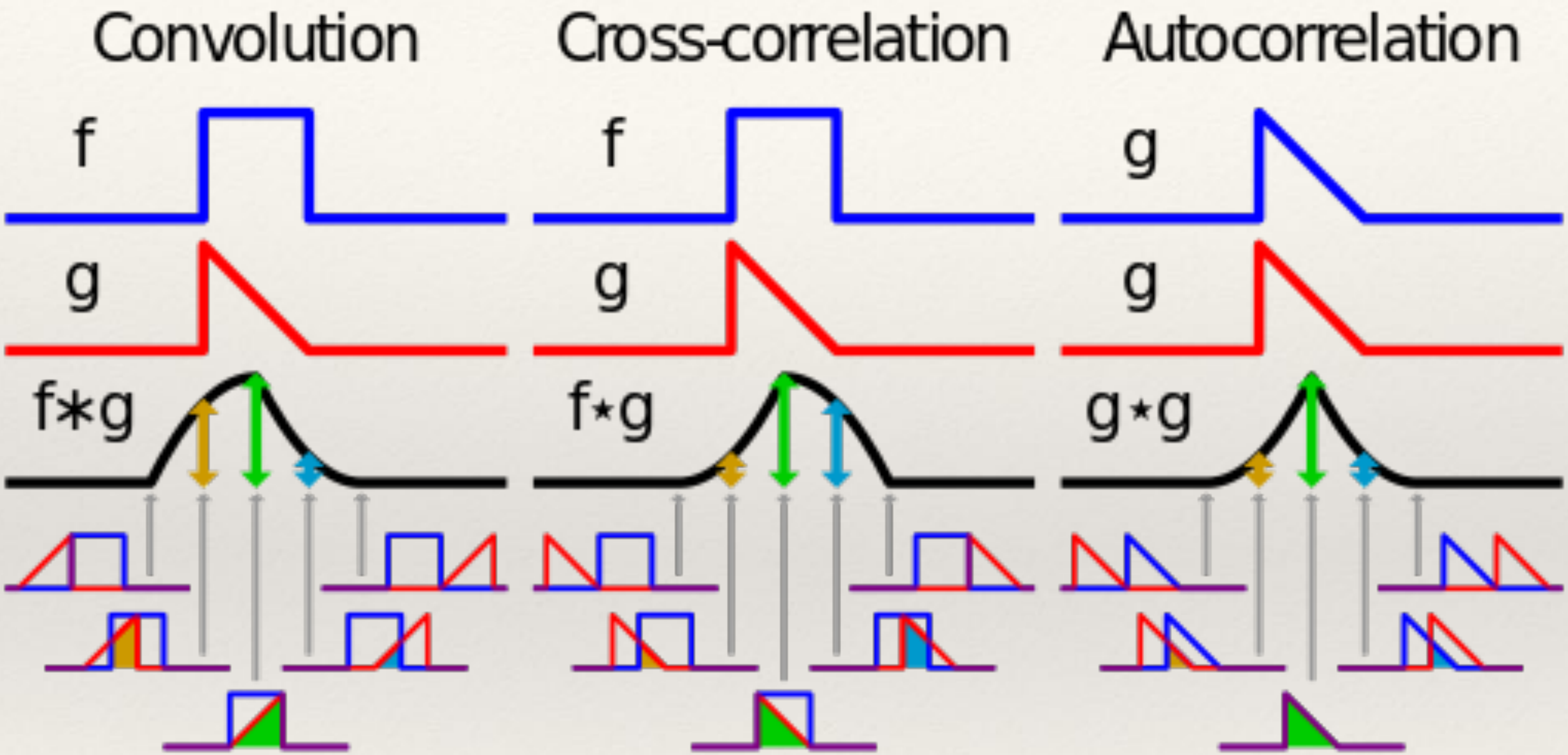
```
int kh = kernel.height;
int kw = kernel.width;
int hh = kh / 2;
int hw = kw / 2;
Image clone = new Image(image.width, image.height);
for (int y = hh; y < image.height - (kh - hh); y++) {
  for (int x = hw; x < image.width - (kw - hw); x++) {
    float sum = 0;
    for (int j = 0, jj = kh - 1; j < kh; j++, jj--) {
      for (int i = 0, ii = kw - 1; i < kw; i++, ii--) {
        int rx = x + i - hw;
        int ry = y + j - hh;

        sum += image.pixels[ry][rx] * kernel.pixels[jj][ii];
      }
    }
    clone.pixels[y][x] = sum;
  }
}
```

What if you don't flip the kernel?

- Obviously if the kernel is symmetric there is no difference
- However, you're actually not computing convolution, but another operation called cross-correlation

$$(f \star g)(\tau) \stackrel{\mathrm{def}}{=} \int_{-\infty}^{\infty} f^*(t)\ g(t+\tau)\, dt,$$

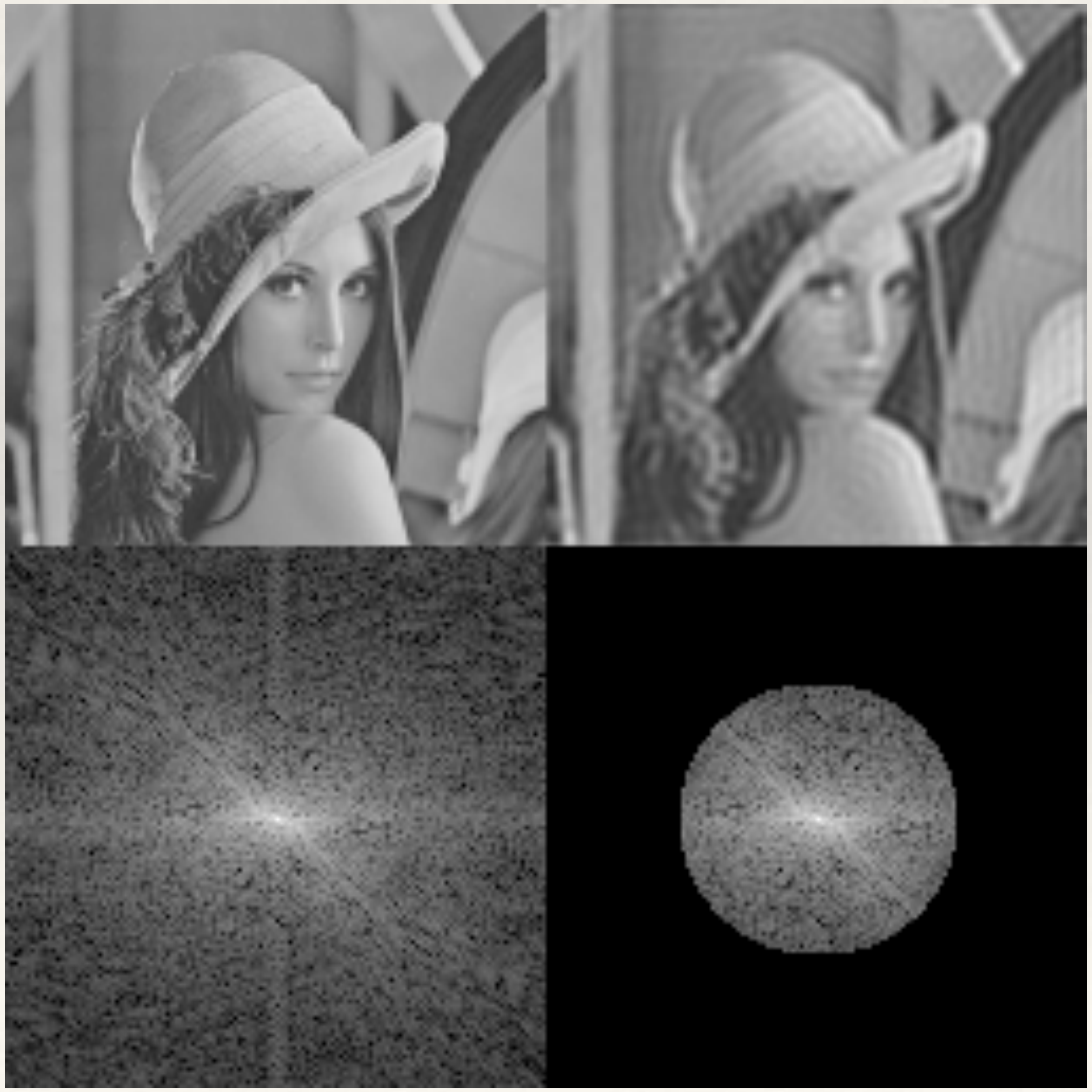
- * represents the complex conjugate
- (you can compute this with the multiplication of the FFTs just like convolution: iFFT(FFT(f)* . FFT(g))



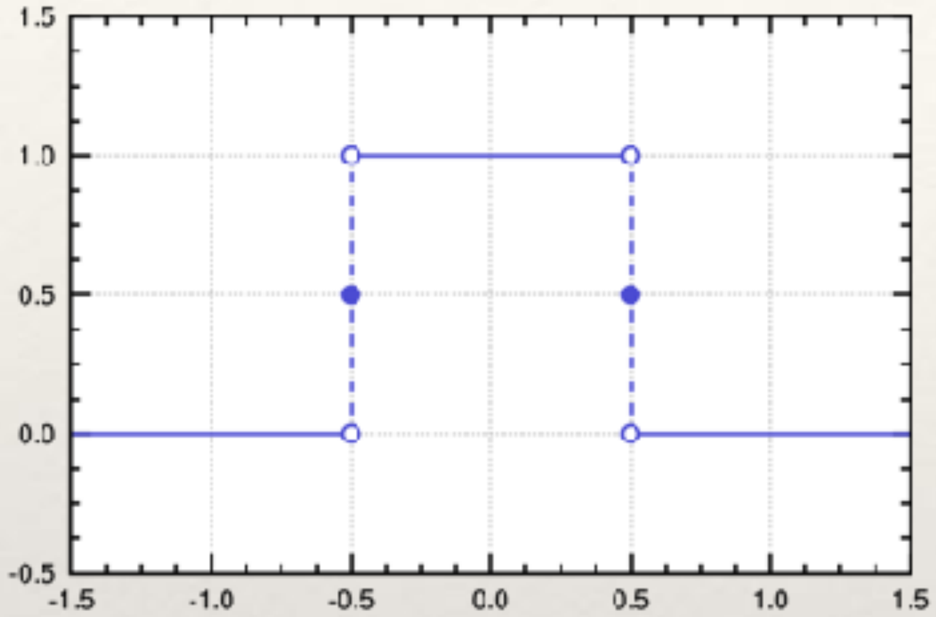
Convolution
Cross-correlation
Autocorrelation
f
f
g
g
g
g
f∗g
f⋆g
g⋆g

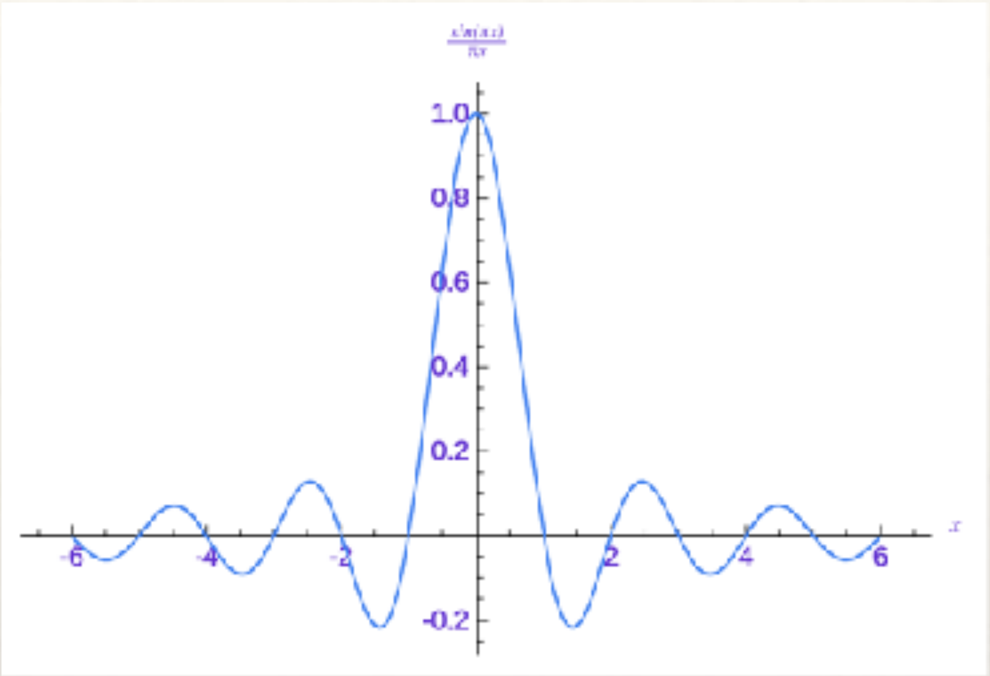
Ideal Low-Pass filter

- “Ideal” low pass filter removes all frequencies above a cutoff

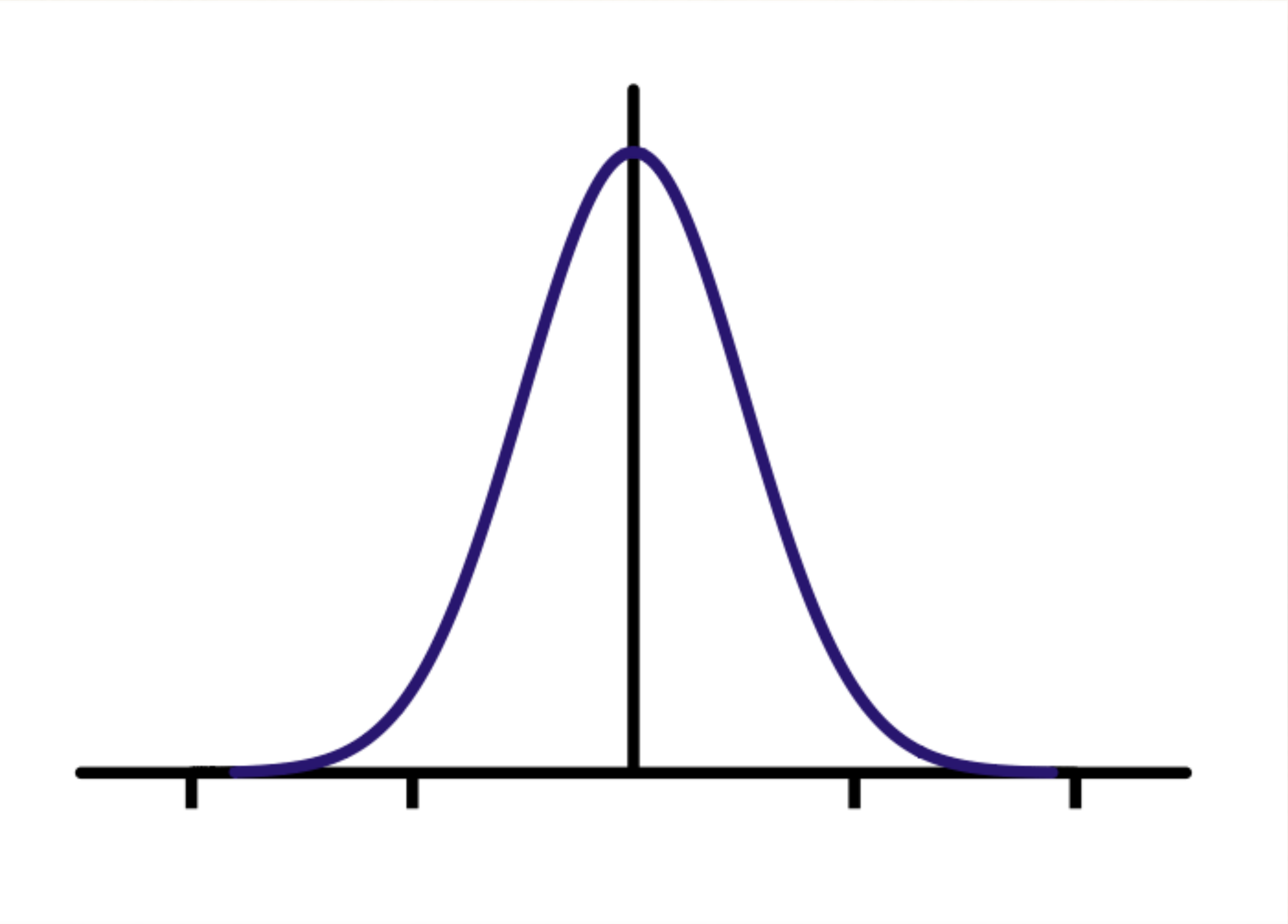


Ideal Low-Pass filter - problems

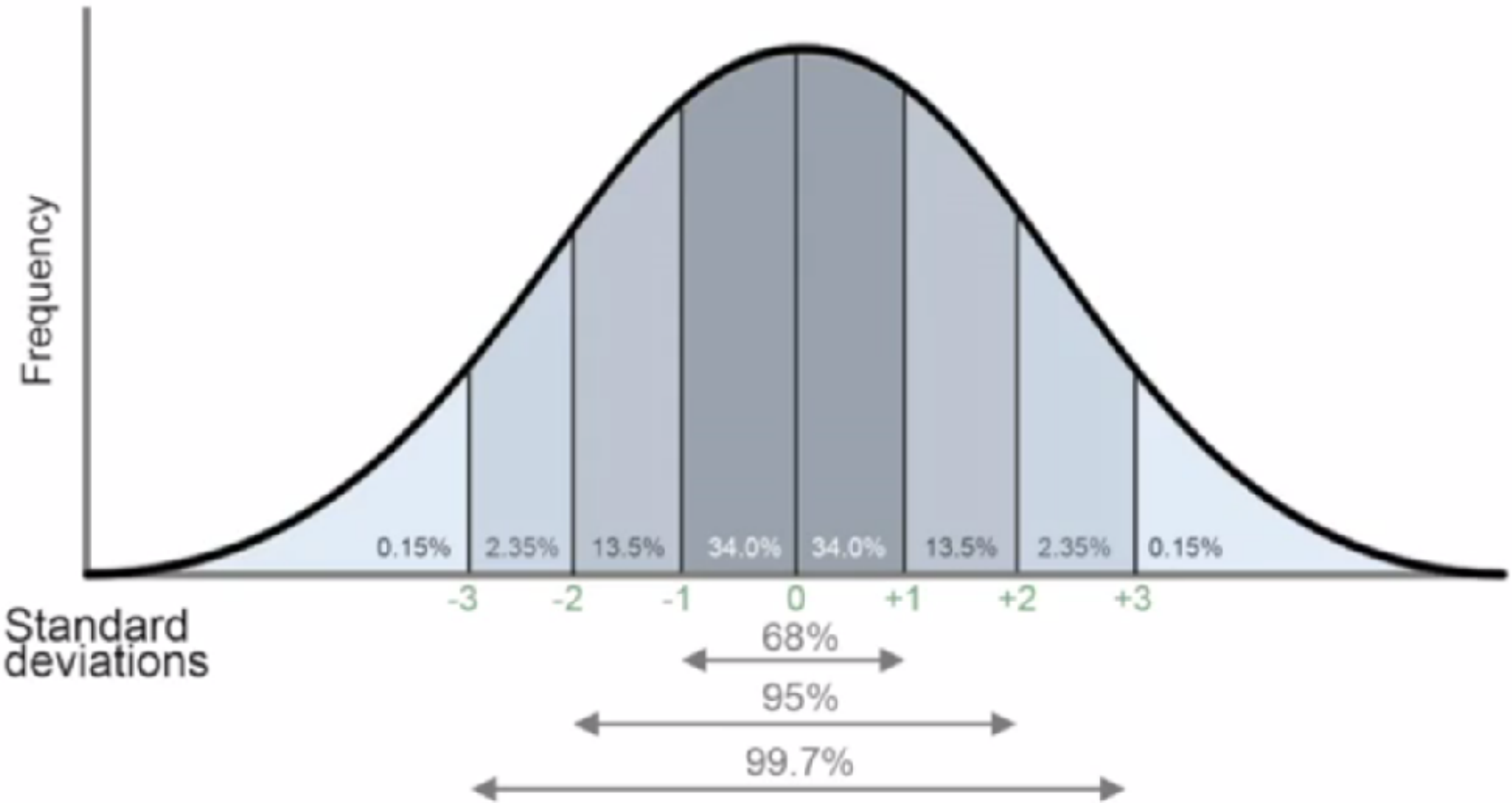




Gaussian filters - why

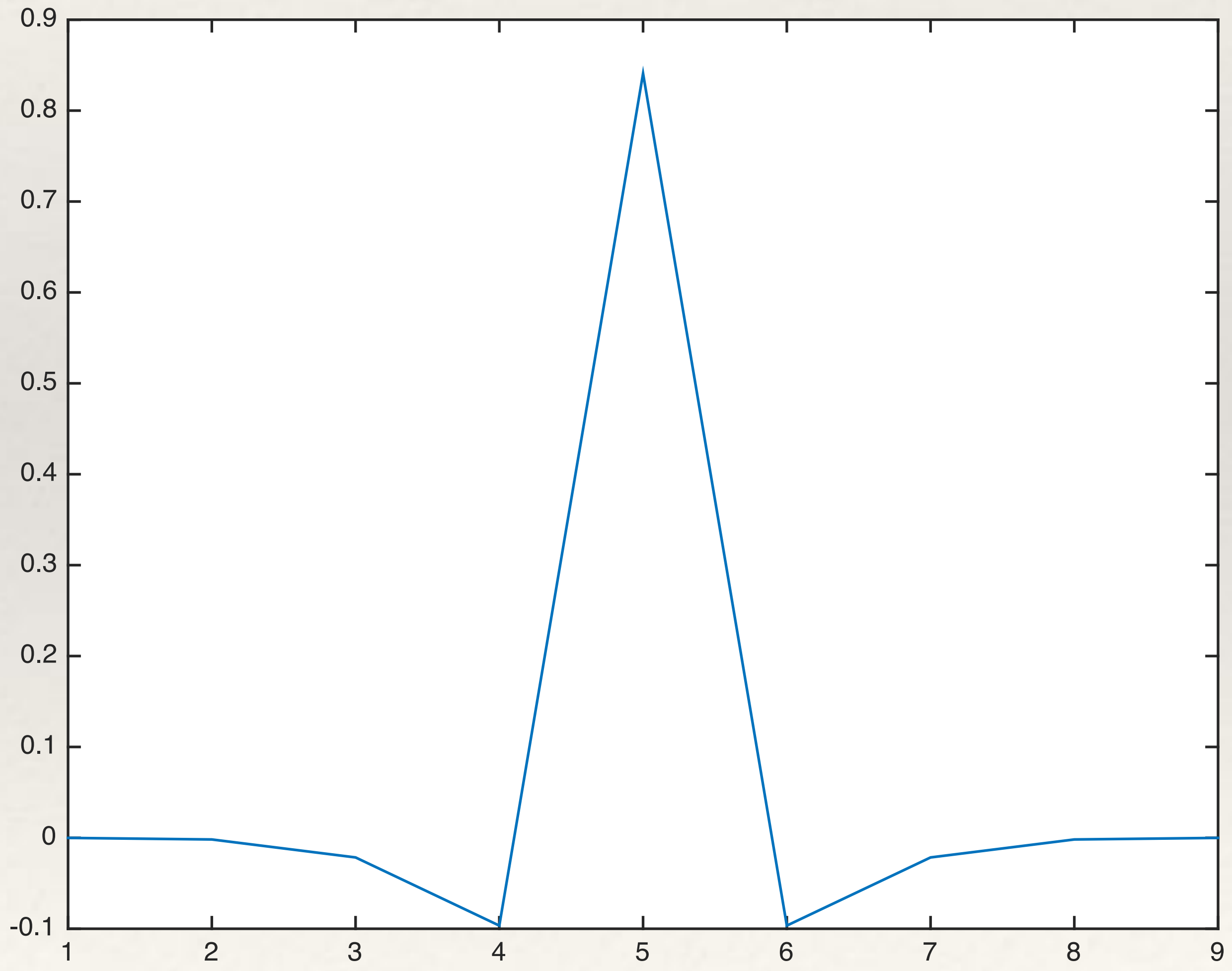


Building Gaussian Filters



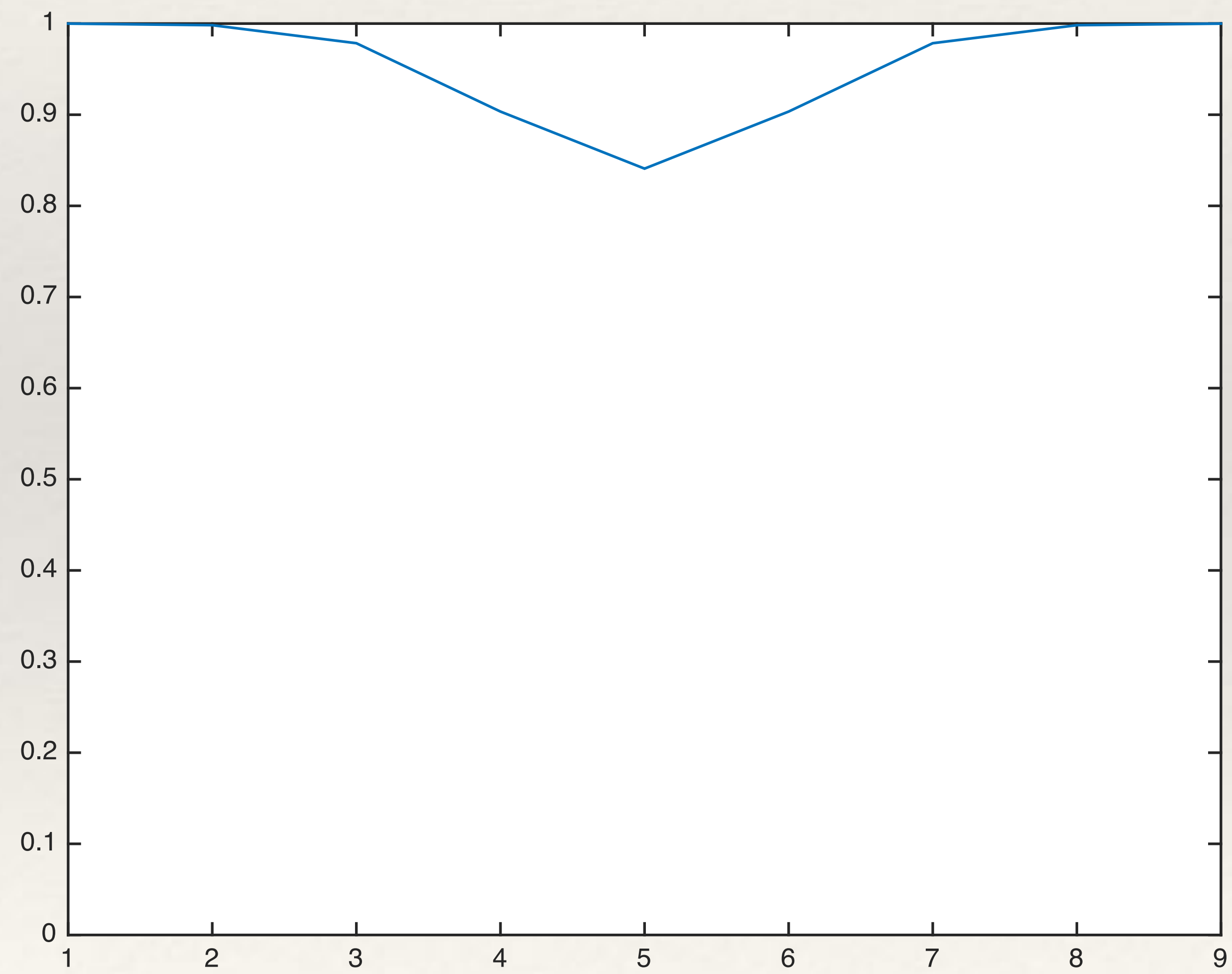
High-pass filters

- “To obtain a high-pass filtered image, subtract a low-pass filtered image from the image itself”
  - $I_{LP} = I * G$
  - $I_{HP} = \mathrm{I}\text{-}I_{LP}$
  - $I_{HP} = I - I * G$
  - $I_{HP} = I * \delta - I * G$
  - $I_{HP} = I * (\delta - G)$



Note - Don't do this!

- $I_{HP} = I * (\delta - G)$ is not the same as $I_{HP} = I * (1 - G)$



This is basically a box filter as sigma increases

(i.e. low pass)

High-pass filters have a mixture of negative and positive coefficients

- …that means the resultant image will also have positive and negative pixels
  - this is important - for example it can tell us about the direction of edges:
    - [-0.5, 0.5] kernel
      - (remember convolution means kernel flipped)
      - +values in the output image mean edge from right to left
      - -values in output image mean edge from left to right
- Convolution implementation MUST NOT:
  - normalise
  - result in unsigned types

Building hybrid images

…is really simple

- Add the low pass and high-pass images together
- Don't:
  - average the two images
  - do a weighted combination of the two images
- just add them (and clip if necessary)

Now it's Time
For The Gallery

Abdullah Hamza
Pawel Ziebl
April Seelog
Daniel Schormans

Questions / Discussion