Lecture 03
Image Filtering

2024-02-19

Sébastien Valade

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Introduction

The image transformations discussed so far are based on the expression:

$$g(x, y) = T[f(x, y)]$$

where:

- f(x, y) is an input image
- g(x, y) is the output image
- T is an operator on f defined over a neighborhood of point (x, y)

Previous lecture:

$\Rightarrow$ the operator T was applied to individual pixels ("Point Operations"), i.e. neighborhood = 1x1 pix

$\Rightarrow$ the function is an *intensity transformation function*, to change image contrast, etc.

Introduction

The image transformations discussed so far are based on the expression:

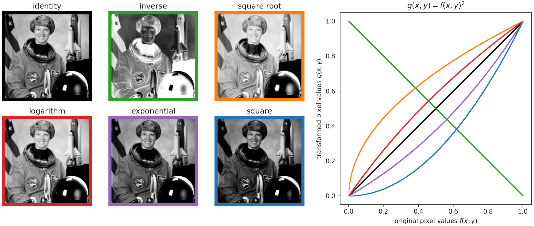
$$g(x, y) = T[f(x, y)]$$

where:

- f(x, y) is an input image
- g(x, y) is the output image
- T is an operator on f defined over a neighborhood of point (x, y)

Previous lecture:

⇒ the operator T was applied to individual pixels (“Point Operations”), i.e. neighborhood = 1x1 pix
⇒ the function is an *intensity transformation function*, to change image contrast, etc.



Introduction

Today: filtering!

⇒ Purpose: blur, sharpen, remove noise, filter frequencies, etc.

⇒ Approaches:

1. **spatial domain filtering**
   - the neighborhood is >1 pixel ("Point Processing" → "Neighborhood Processing")
   - spatial filtering modifies an image by replacing the value of each pixel by a function of the values of the pixel and its neighbor
   - if the operation performed on the image pixels is linear, then the filter is called a linear spatial filter
   - spatial filters are applied by **convolution**
2. **frequency domain filtering**
   - the **2D direct Fourier transform** is applied to extract image frequencies
   - the amplitude spectrum can be band-passed to filter certain frequencies
   - the inverse 2D direct Fourier transform is used to reconstruct the filtered image

Introduction

Today: filtering!

⇒ Purpose: blur, sharpen, remove noise, filter frequencies, etc.

⇒ Approaches:

1. **spatial domain filtering**
   - the neighborhood is >1 pixel ("Point Processing" → "Neighborhood Processing")
   - spatial filtering modifies an image by replacing the value of each pixel by a function of the values of the pixel and its neighbor
   - if the operation performed on the image pixels is linear, then the filter is called a linear spatial filter
   - spatial filters are applied by **convolution**
2. **frequency domain filtering**
   - the **2D direct Fourier transform** is applied to extract image frequencies
   - the amplitude spectrum can be band-passed to filter certain frequencies
   - the inverse 2D direct Fourier transform is used to reconstruct the filtered image

Introduction

Today: filtering!

⇒ Purpose: blur, sharpen, remove noise, filter frequencies, etc.

⇒ Approaches:

1. **spatial domain filtering**
   - the neighborhood is >1 pixel ("Point Processing" $\rightarrow$ "Neighborhood Processing")
   - spatial filtering modifies an image by replacing the value of each pixel by a function of the values of the pixel and its neighbor
   - if the operation performed on the image pixels is linear, then the filter is called a linear spatial filter
   - spatial filters are applied by **convolution**
2. **frequency domain filtering**
   - the **2D direct Fourier transform** is applied to extract image frequencies
   - the amplitude spectrum can be band-passed to filter certain frequencies
   - the inverse 2D direct Fourier transform is used to reconstruct the filtered image

Linear spatial filter

$\Rightarrow$ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- kernel size (m,n) defines the neighborhood of operation on pixel at position (x,y)
- kernel coefficients define the nature of the filter

Linear spatial filter

⇒ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- kernel size (m,n) defines the neighborhood of operation on pixel at position (x,y)
- kernel coefficients define the nature of the filter

Linear spatial filter

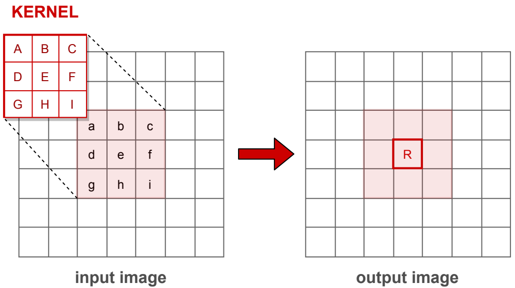
$\Rightarrow$ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- kernel size (m,n) defines the neighborhood of operation on pixel at position (x,y)
- kernel coefficients define the nature of the filter

Linear spatial filter

⇒ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- kernel size (m,n) defines the neighborhood of operation on pixel at position (x,y)
- kernel coefficients define the nature of the filter

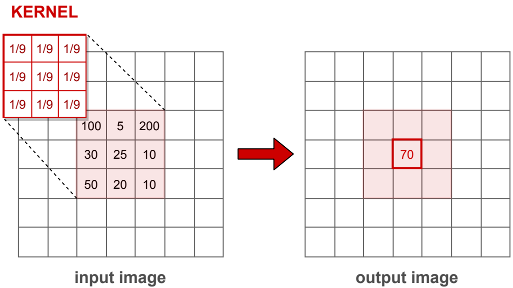


R = A*a + B*b + ... + H*h + I*i

Linear spatial filter

⇒ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- kernel size (m,n) defines the neighborhood of operation on pixel at position (x,y)
- kernel coefficients define the nature of the filter



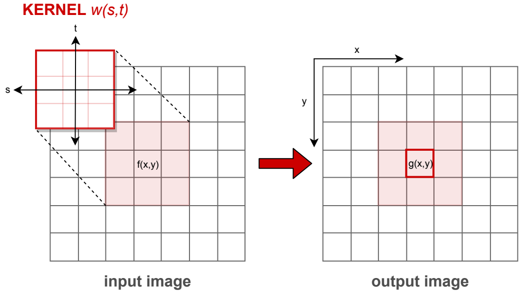
R = 1/9*100 + 1/9*5 + ... + 1/9*20 + 1/9*10

R = 70

Linear spatial filter

⇒ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- kernel size (m,n) defines the neighborhood of operation on pixel at position (x,y)
- kernel coefficients define the nature of the filter



$$g(x,y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s,t) \cdot f(x+s, y+t)$$

where a and b define an odd-shape kernel size (m=2a+1, n=2b+1)

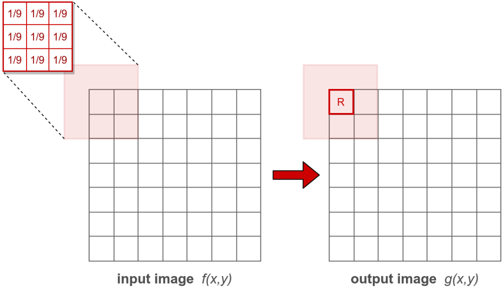
Linear spatial filter

⇒ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- kernel size (m,n) defines the neighborhood of operation on pixel at position (x,y)
- kernel coefficients define the nature of the filter

⇒ kernel slides across the input image to produce a *filtered* **output image g(x,y)**

- stride = sliding step `(stride=1 => kernel will slide by 1 pixel per column/row at a time)`



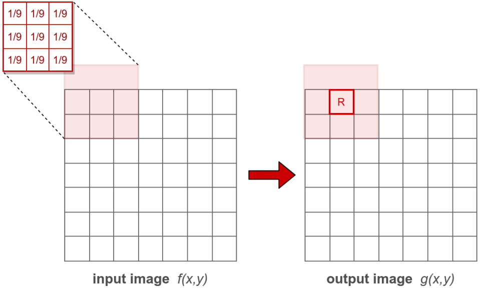
Linear spatial filter

⇒ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- kernel size (m,n) defines the neighborhood of operation on pixel at position (x,y)
- kernel coefficients define the nature of the filter

⇒ kernel slides across the input image to produce a *filtered* **output image g(x,y)**

- stride = sliding step `(stride=1 => kernel will slide by 1 pixel per column/row at a time)`



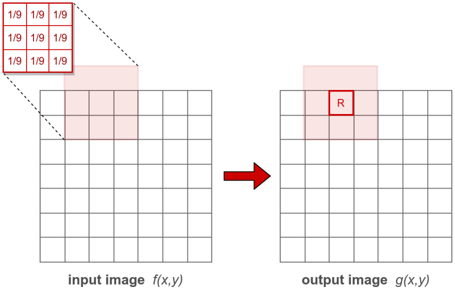
Linear spatial filter

⇒ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- kernel size (m,n) defines the neighborhood of operation on pixel at position (x,y)
- kernel coefficients define the nature of the filter

⇒ kernel slides across the input image to produce a *filtered* **output image g(x,y)**

- stride = sliding step `(stride=1 => kernel will slide by 1 pixel per column/row at a time)`



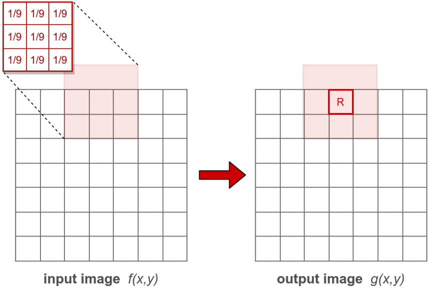
Linear spatial filter

$\Rightarrow$ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- kernel size (m,n) defines the neighborhood of operation on pixel at position (x,y)
- kernel coefficients define the nature of the filter

$\Rightarrow$ kernel slides across the input image to produce a *filtered* **output image g(x,y)**

- stride = sliding step `(stride=1 => kernel will slide by 1 pixel per column/row at a time)`



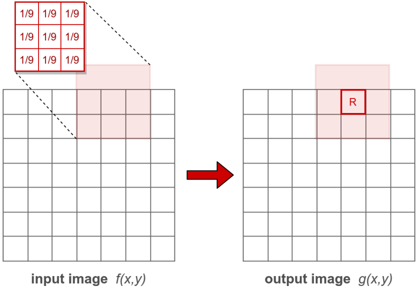
Linear spatial filter

⇒ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- kernel size (m,n) defines the neighborhood of operation on pixel at position (x,y)
- kernel coefficients define the nature of the filter

⇒ kernel slides across the input image to produce a *filtered* **output image g(x,y)**

- stride = sliding step `(stride=1 => kernel will slide by 1 pixel per column/row at a time)`



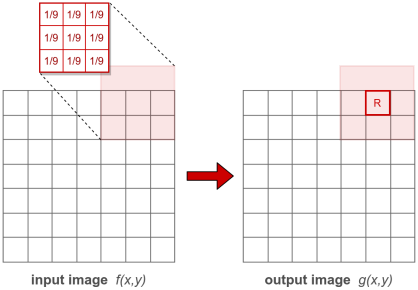
Linear spatial filter

⇒ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- kernel size (m,n) defines the neighborhood of operation on pixel at position (x,y)
- kernel coefficients define the nature of the filter

⇒ kernel slides across the input image to produce a *filtered* **output image g(x,y)**

- stride = sliding step `(stride=1 => kernel will slide by 1 pixel per column/row at a time)`



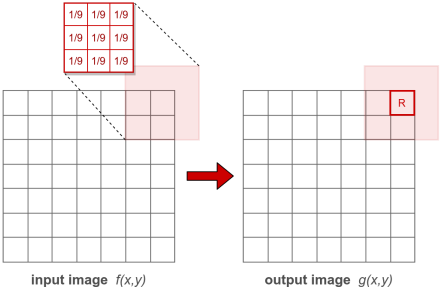
Linear spatial filter

⇒ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- kernel size (m,n) defines the neighborhood of operation on pixel at position (x,y)
- kernel coefficients define the nature of the filter

⇒ kernel slides across the input image to produce a *filtered* **output image g(x,y)**

- stride = sliding step `(stride=1 => kernel will slide by 1 pixel per column/row at a time)`



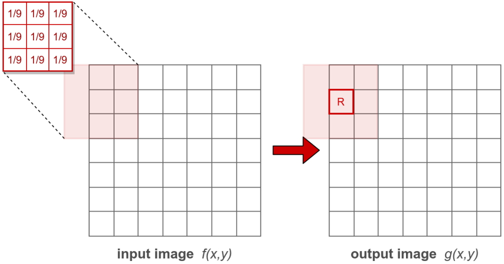
Linear spatial filter

⇒ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- kernel size (m,n) defines the neighborhood of operation on pixel at position (x,y)
- kernel coefficients define the nature of the filter

⇒ kernel slides across the input image to produce a *filtered* **output image g(x,y)**

- stride = sliding step `(stride=1 => kernel will slide by 1 pixel per column/row at a time)`



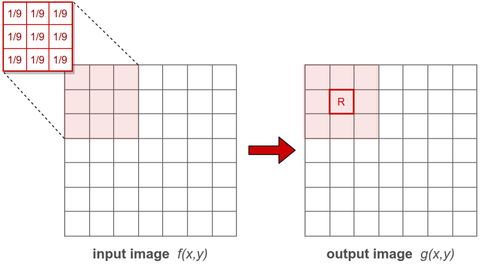
Linear spatial filter

⇒ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- kernel size (m,n) defines the neighborhood of operation on pixel at position (x,y)
- kernel coefficients define the nature of the filter

⇒ kernel slides across the input image to produce a *filtered* **output image g(x,y)**

- stride = sliding step `(stride=1 => kernel will slide by 1 pixel per column/row at a time)`



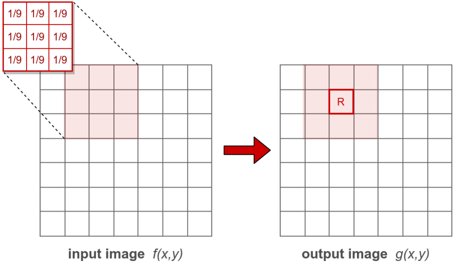
Linear spatial filter

⇒ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- kernel size (m,n) defines the neighborhood of operation on pixel at position (x,y)
- kernel coefficients define the nature of the filter

⇒ kernel slides across the input image to produce a *filtered* **output image g(x,y)**

- stride = sliding step `(stride=1 => kernel will slide by 1 pixel per column/row at a time)`



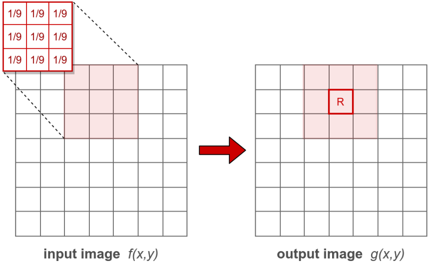
Linear spatial filter

⇒ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- kernel size (m,n) defines the neighborhood of operation on pixel at position (x,y)
- kernel coefficients define the nature of the filter

⇒ kernel slides across the input image to produce a *filtered* **output image g(x,y)**

- stride = sliding step `(stride=1 => kernel will slide by 1 pixel per column/row at a time)`



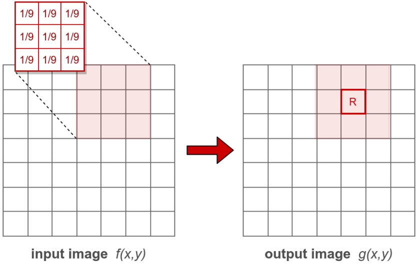
Linear spatial filter

⇒ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- kernel size (m,n) defines the neighborhood of operation on pixel at position (x,y)
- kernel coefficients define the nature of the filter

⇒ kernel slides across the input image to produce a *filtered* **output image g(x,y)**

- stride = sliding step `(stride=1 => kernel will slide by 1 pixel per column/row at a time)`



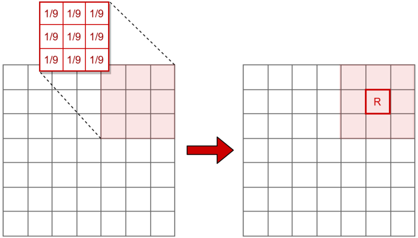
Linear spatial filter

⇒ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- kernel size (m,n) defines the neighborhood of operation on pixel at position (x,y)
- kernel coefficients define the nature of the filter

⇒ kernel slides across the input image to produce a *filtered* **output image g(x,y)**

- stride = sliding step `(stride=1 => kernel will slide by 1 pixel per column/row at a time)`



input image *f(x,y)* output image *g(x,y)*

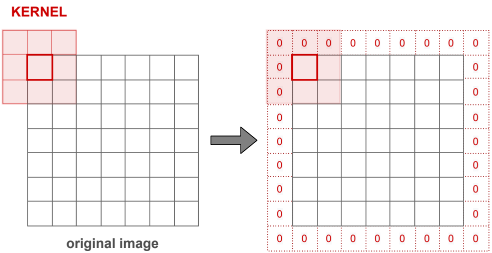
Linear spatial filter

⇒ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- kernel size (m,n) defines the neighborhood of operation on pixel at position (x,y)
- kernel coefficients define the nature of the filter

⇒ kernel slides across the input image to produce a *filtered* **output image g(x,y)**

- stride = sliding step `(stride=1 => kernel will slide by 1 pixel per column/row at a time)`
- padding = pad the image so the kernel can also operate on the edges `(pad_size=kernel_size//2)`



padded image (zero-padding)

`padding_size = kernel_size // 2`

Linear spatial filter

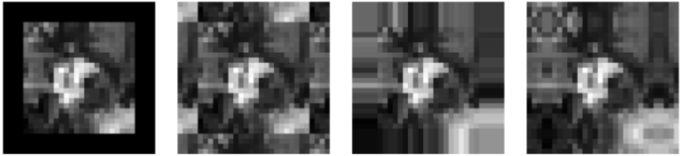
⇒ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- kernel size (m,n) defines the neighborhood of operation on pixel at position (x,y)
- kernel coefficients define the nature of the filter

⇒ kernel slides across the input image to produce a *filtered* **output image g(x,y)**

- stride = sliding step (stride=1 => kernel will slide by 1 pixel per column/row at a time)
- padding = pad the image so the kernel can also operate on the edges (pad_size=kernel_size//2)

various padding types (Richard Szeliski, 2010)



zero wrap clamp mirror

Linear spatial filter

⇒ the sum-of-products operation between the input image $f(x, y)$ and filter kernel w (eq.1) is the implementation of a **spatial convolution** (eq.2):

$$g(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s, t) \cdot f(x - s, y - t) \tag{1}$$

$$g = w * f \tag{2}$$

Linear spatial filter

$\Rightarrow$ the sum-of-products operation between the input image $f(x, y)$ and filter kernel w (eq.1) is the implementation of a **spatial convolution** (eq.2):

$$g(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s, t) \cdot f(x - s, y - t) \quad (1)$$

$$g = w * f \quad (2)$$

linear spatial filtering $\Longleftrightarrow$ **spatial convolution**

Linear spatial filter

⇒ the sum-of-products operation between the input image $f(x, y)$ and filter kernel w (eq.1) is the implementation of a **spatial convolution** (eq.2):

$$g(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s, t) \cdot f(x - s, y - t) \tag{1}$$

$$g = w * f \tag{2}$$

linear spatial filtering ⟺ **spatial convolution**

convolutions are the core operations used by **Convolutional Neural Networks** (CNN)

Linear spatial filter

⇒ the sum-of-products operation between the input image $f(x, y)$ and filter kernel w (eq.1) is the implementation of a **spatial convolution** (eq.2):

$$g(x, y) = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w(s, t) \cdot f(x - s, y - t) \quad (1)$$

$$g = w * f \quad (2)$$

linear spatial filtering ⟺ **spatial convolution**

convolutions are the core operations used by **Convolutional Neural Networks** (CNN)

Nota Bene: spatial **convolution** and spatial **correlation** operate in the same way, except that the correlation kernel is rotated by 180° (⇒ when kernel values are symmetric about its center, correlation and convolution yield the same result)

Kernel coefficients define the nature of the filter
⇒ vary kernels coefficients according to the desired filtering operation:

- smoothing spatial filters (low-pass)
  - box filter
  - gaussian filter
- sharpening spatial filters (high-pass)
  - Sobel filter, Prewitt filter
  - Laplacian filter
- other
  - emboss filter
  - etc.

Kernel coefficients define the nature of the filter
$\Rightarrow$ vary kernels coefficients according to the desired filtering operation:

- smoothing spatial filters (low-pass)
  - box filter
  - gaussian filter

- sharpening spatial filters (high-pass)
  - Sobel filter, Prewitt filter
  - Laplacian filter

- other
  - emboss filter
  - etc.

Kernel coefficients define the nature of the filter
$\Rightarrow$ vary kernels coefficients according to the desired filtering operation:

- smoothing spatial filters (low-pass)
  - box filter
  - gaussian filter
- sharpening spatial filters (high-pass)
  - Sobel filter, Prewitt filter
  - Laplacian filter
- other
  - emboss filter
  - etc.

Kernel coefficients define the nature of the filter
$\Rightarrow$ vary kernels coefficients according to the desired filtering operation:

- smoothing spatial filters (low-pass)
  - box filter
  - gaussian filter
- sharpening spatial filters (high-pass)
  - Sobel filter, Prewitt filter
  - Laplacian filter
- other
  - emboss filter
  - etc.

2.3. kernels types and applications

identity

0	0	0
0	1	0
0	0	0

$\Rightarrow$ no change!

LOW PASS FILTER

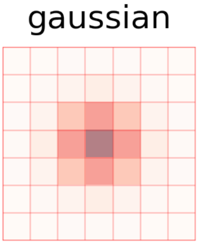
average

0.1	0.1	0.1
0.1	0.1	0.1
0.1	0.1	0.1

unweighted average, a.k.a. **box filter** (low pass)

$\Rightarrow$ blurring effect

LOW PASS FILTER

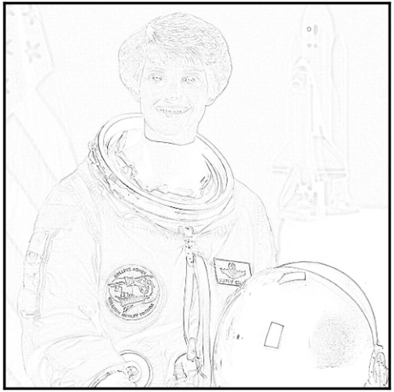


weighted average (low pass)
$\Rightarrow$ blurring effect

HIGH PASS FILTER

highpass

0	-1	0
-1	4	-1
0	-1	0



(extension of the Laplacian kernel)

$\Rightarrow$ edge detection (no orientation)

HIGH PASS FILTER

sharpen

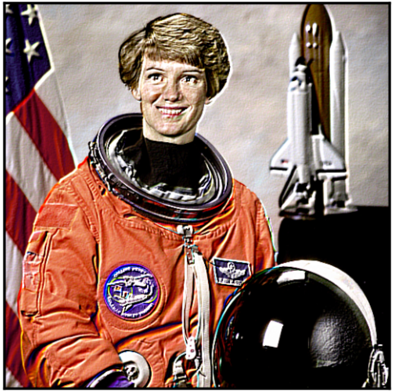
0	-1	0
-1	5	-1
0	-1	0

identity kernel + highpass kernel
$\Rightarrow$ sharpening effect

2.3. kernels types and applications

emboss

-2	-1	0
-1	1	1
0	1	2



$\Rightarrow$ styling effect

2.3. kernels types and applications

sobel x

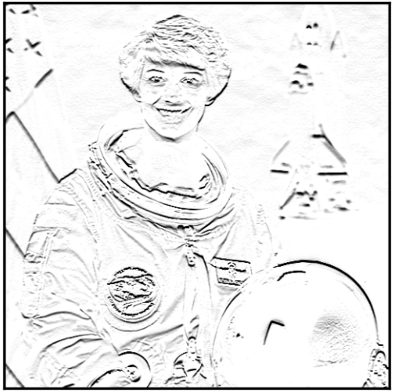
-1	0	1
-2	0	2
-1	0	1

$\Rightarrow$ edge detection (x-direction)

2.3. kernels types and applications

sobel y

-1	-2	-1
0	0	0
1	2	1



$\Rightarrow$ edge detection (y-direction)

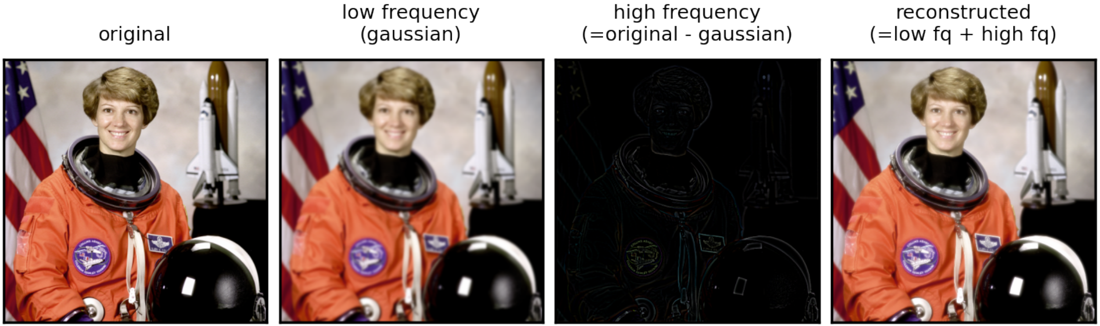
2.3. kernels types and applications

$\Rightarrow$ edges + magnitude

Gaussian filters are a true <u>low-pass filter</u> for the image

⇒ we can retrieve the low-frequency in an image

⇒ we can retrieve the high-frequency in an image by subtracting the low-frequency from the original image



3.1. 1D Fourier transform

⇒ convolutions for **spatial domain filtering** is powerful, BUT it has high computational costs

⇒ **frequency domain filtering** offers computational advantages:

*(**convolution** in the time domain ⟺ **multiplication** in the frequency domain)*

3.1. 1D Fourier transform

⇒ convolutions for **spatial domain filtering** is powerful, BUT it has high computational costs

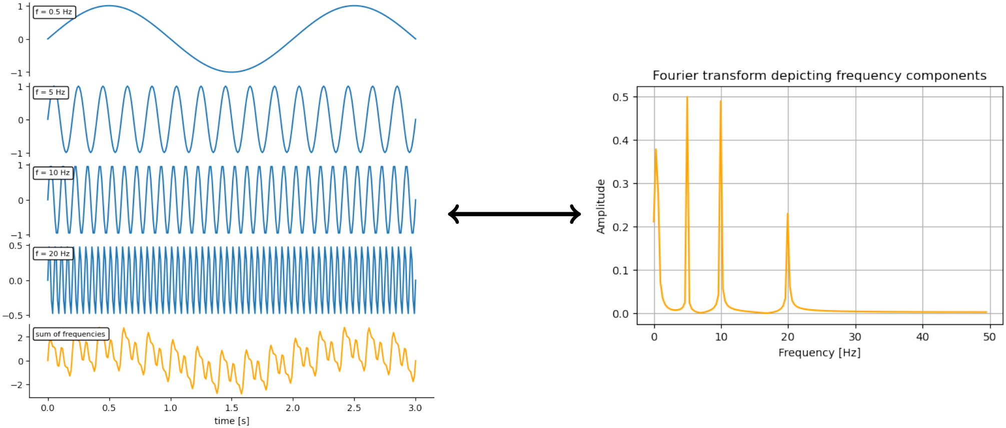
⇒ **frequency domain filtering** offers computational advantages:

*(**convolution** in the time domain ⟺ **multiplication** in the frequency domain)*

3.1. 1D Fourier transform

Fourier theorem: a continuous and periodic function can be approximated as infinite sum of sine- and cosine-functions

- **Forward transform**: Time Domain → Frequency Domain
- **Inverse transform**: Frequency Domain → Time Domain

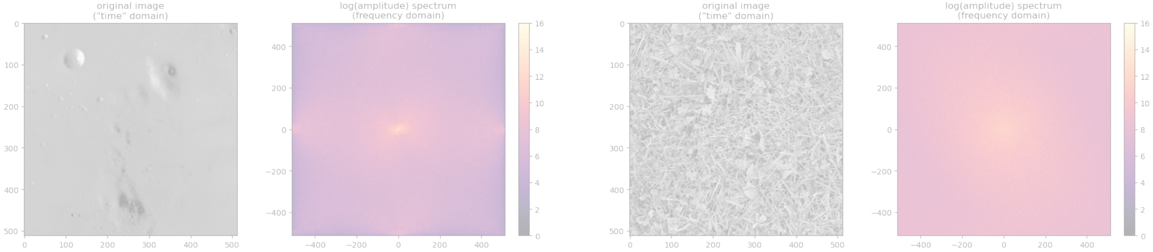


Fourier transform on images ?

⇒ an image can also be expressed as the sum of sinusoids of different frequencies and amplitudes

⇒ the appearance of an image depends on the frequencies of its sinusoidal components:
(NB: Fourier transform of a real function is symmetric about the origin; by convention frequency 0 is set at the center of image)

- **low frequencies** → regions with intensities that vary slowly
- **high frequencies** → edges and other sharp intensity transitions

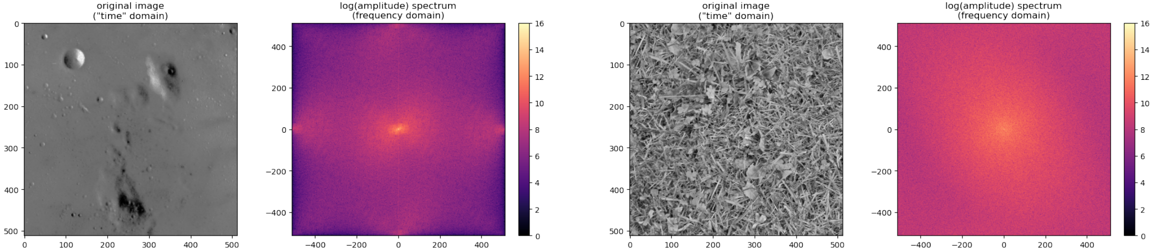


Fourier transform on images ?

⇒ an image can also be expressed as the sum of sinusoids of different frequencies and amplitudes

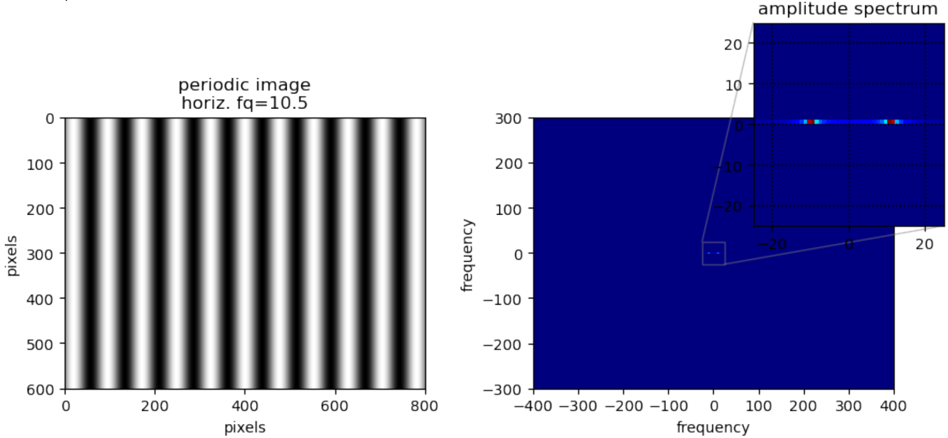
⇒ the appearance of an image depends on the frequencies of its sinusoidal components:
(NB: Fourier transform of a real function is symmetric about the origin; by convention frequency 0 is set at the center of image)

- **low frequencies** → regions with intensities that vary slowly
- **high frequencies** → edges and other sharp intensity transitions



2D Fourier transform on SYNTH images

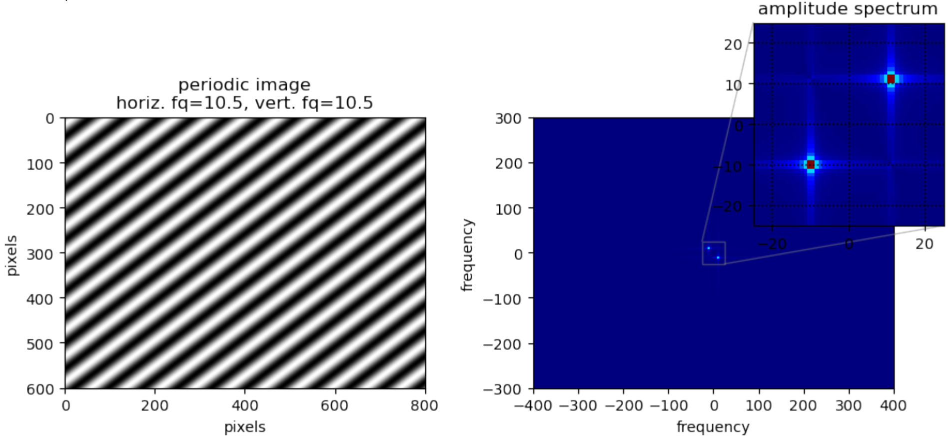
⇒ "dots" symmetric about origin in amplitude spectrum
⇒ distance/direction from origin imply frequency in time domain



2D Fourier transform on SYNTH images

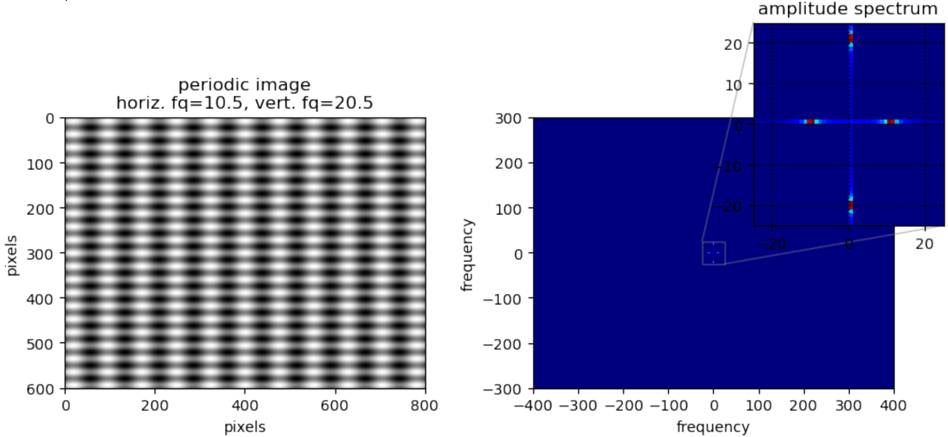
⇒ "dots" symmetric about origin in amplitude spectrum

⇒ distance/direction from origin imply frequency in time domain

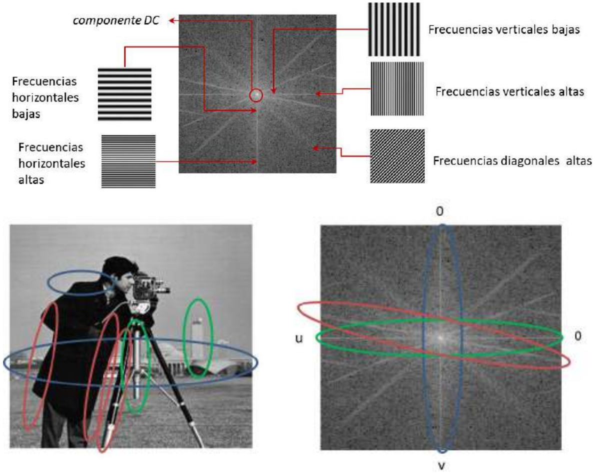


2D Fourier transform on SYNTH images

⇒ "dots" symmetric about origin in amplitude spectrum
⇒ distance/direction from origin imply frequency in time domain



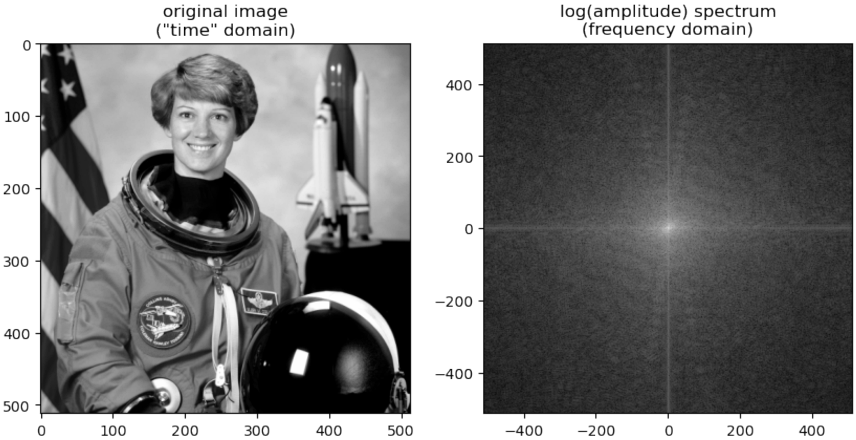
2D Fourier transform on REAL images



Credit: Alegre et al. 2016

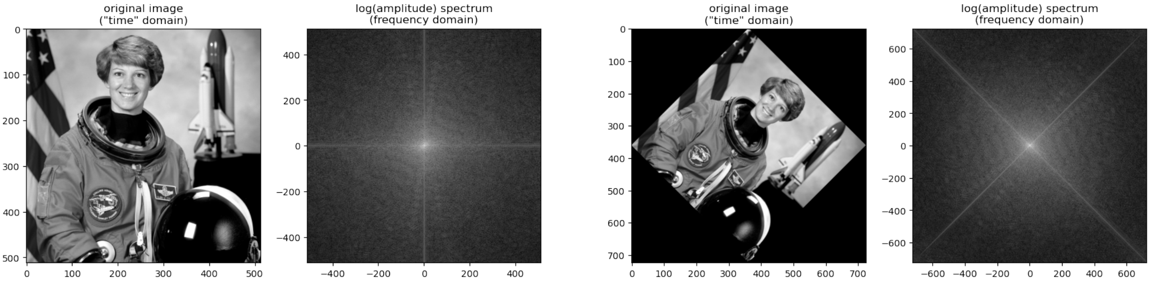
2D Fourier transform on REAL images

⇒ let's try on our astronaut



2D Fourier transform on REAL images

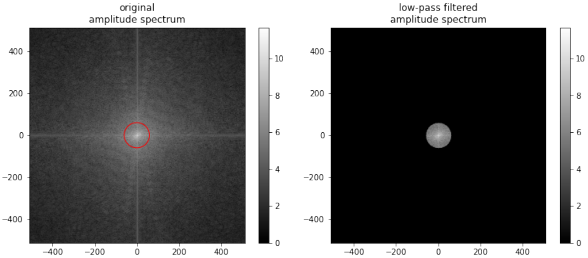
⇒ let's try on our astronaut



2D Fourier transform on REAL images

⇒ band-pass image frequencies?

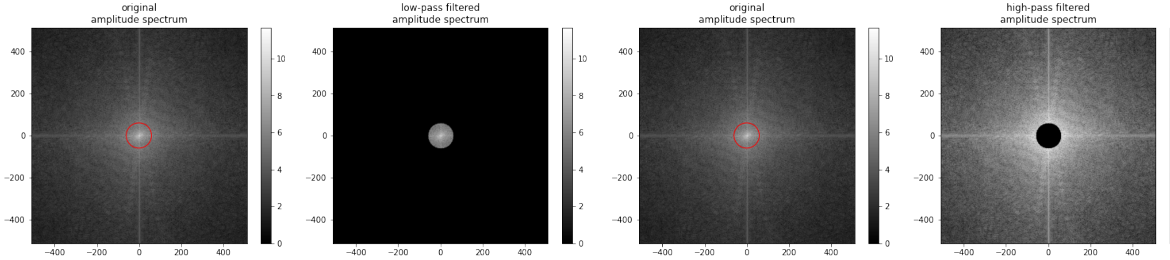
- **low-pass** filter → cut off high-frequencies
- **high-pass** filter → cut off low-frequencies



2D Fourier transform on REAL images

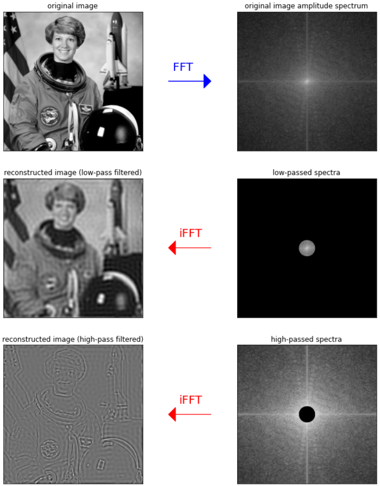
⇒ band-pass image frequencies?

- **low-pass** filter → cut off high-frequencies
- **high-pass** filter → cut off low-frequencies



2D Fourier transform on REAL images

⇒ image can be reconstructed from band-passed spectra using the 2D **inverse Fourier transform** (iFFT2)



2D Fourier transform on REAL images

$\Rightarrow$ the **ideal low-pass filter** (LPF) introduces artefacts:
- "ripples" near strong edges in the original image: **ringing effect**
- related to the sharp cut-off in ideal frequency domain

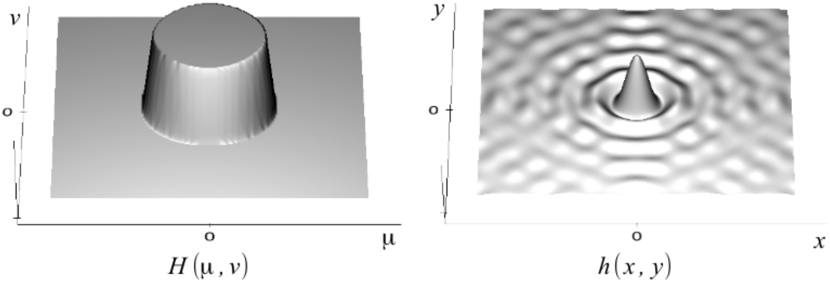
low-pass filtered image

ringing effect

2D Fourier transform on REAL images

⇒ the **ideal low-pass filter** (LPF) introduces artefacts:
- "ripples" near strong edges in the original image: **ringing effect**
- related to the sharp cut-off in ideal frequency domain

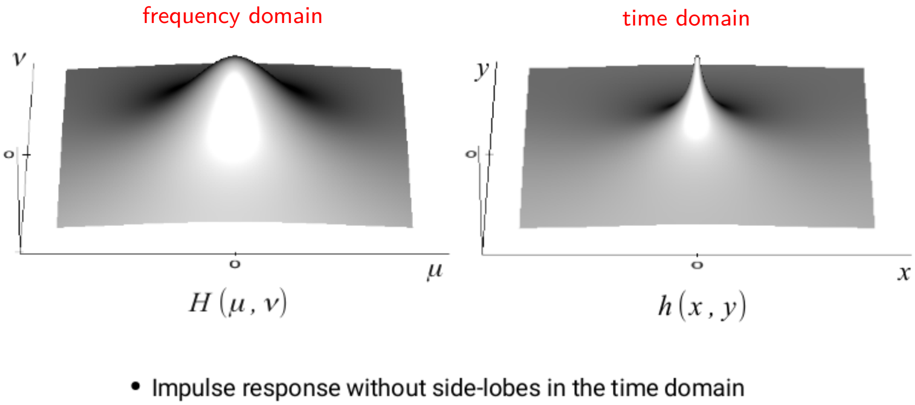
frequency domain time domain



• Ideal LPF has significant 'side-lobes' in the time domain

2D Fourier transform on REAL images

⇒ the **Butterworth** filter offers impulse response without side-lobes in the time domain ideal
→ no "ringing effect", due to the absence of discontinuity in spectrum



2D Fourier transform on REAL images

⇒ the **Butterworth** filter offers impulse response without side-lobes in the time domain ideal
→ no "ringing effect", due to the absence of discontinuity in spectrum