Image Filtering

Lecture 03

Computer Vision for Geosciences

2021-03-12

1. Introduction
2. Spatial Domain Filtering
    1. linear spatial filter
    2. convolutions
    3. kernels types and applications
3. Frequency domain filtering
    1. 1D Fourier transform
    2. 2D Fourier transform
    3. Butterworth filter

1. Introduction

2. Spatial Domain Filtering
   1. linear spatial filter
   2. convolutions
   3. kernels types and applications

3. Frequency domain filtering
   1. 1D Fourier transform
   2. 2D Fourier transform
   3. Butterworth filter

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.

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

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

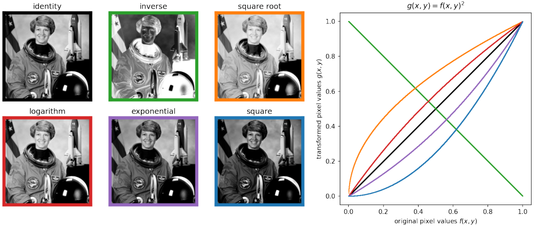
where:

- 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.



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 restitute filtered image

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 restitute filtered image

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 restitute filtered image

1. Introduction

2. Spatial Domain Filtering
    1. linear spatial filter
    2. convolutions
    3. kernels types and applications

3. Frequency domain filtering
    1. 1D Fourier transform
    2. 2D Fourier transform
    3. Butterworth 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

⇒ 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

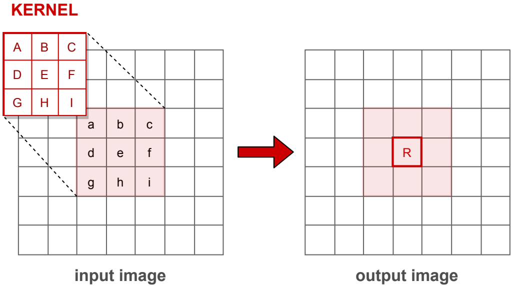
$\Rightarrow$ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- 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

$\Rightarrow$ 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

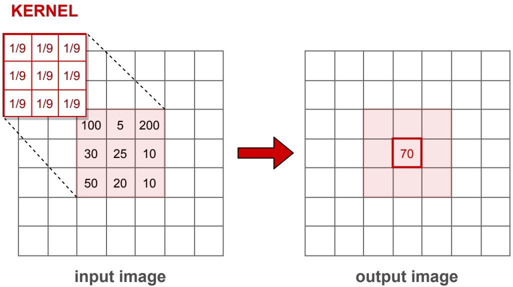


R = A*a + B*b + ... + H*h + I*i

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



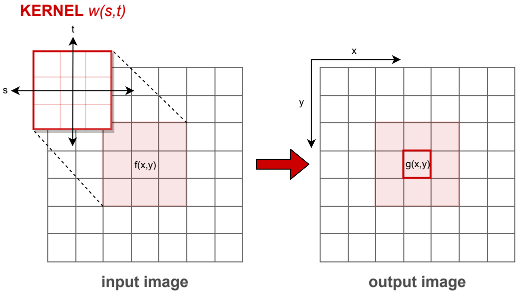
R = 1/9*100 + 1/9*5 + ... + 1/9*20 + 1/9*10

R = 70

linear spatial filter

$\Rightarrow$ 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



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

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

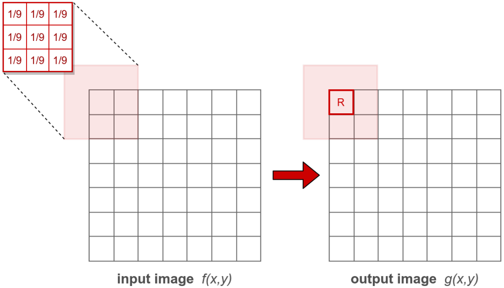
linear spatial filter

⇒ 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)`



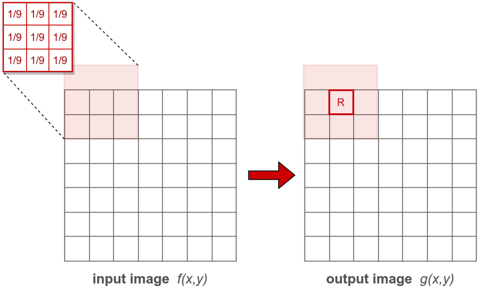
linear spatial filter

⇒ 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)`



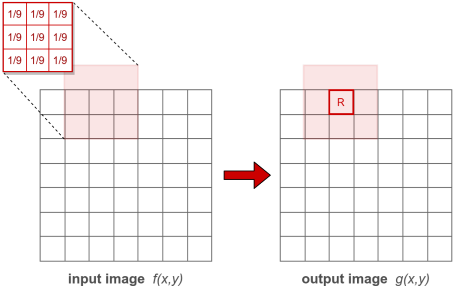
linear spatial filter

⇒ 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)`



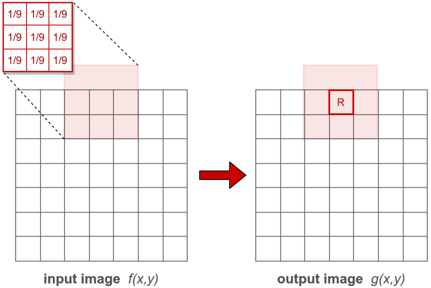
linear spatial filter

⇒ 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)`



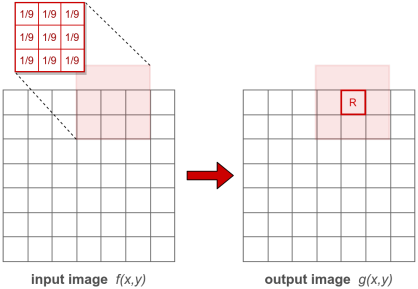
linear spatial filter

⇒ 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)`



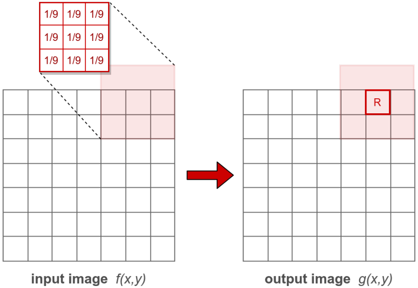
linear spatial filter

⇒ 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)`



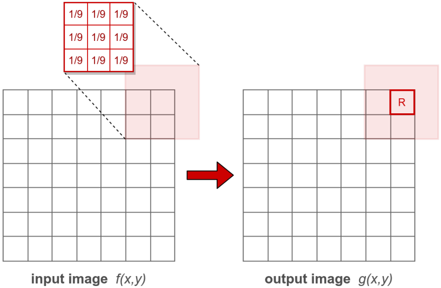
linear spatial filter

⇒ 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)`



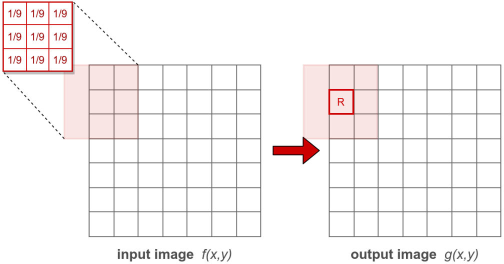
linear spatial filter

⇒ 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)`



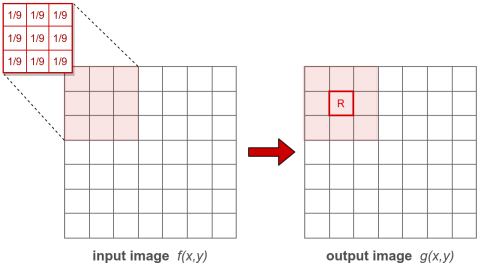
linear spatial filter

⇒ 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)`



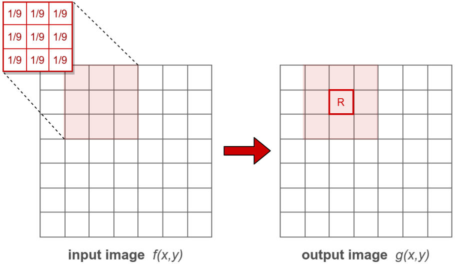
linear spatial filter

⇒ 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)`



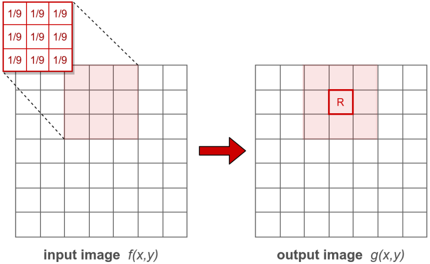
linear spatial filter

⇒ 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)`



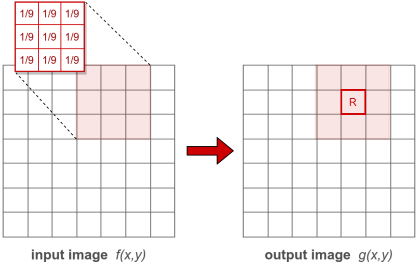
linear spatial filter

⇒ 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)`



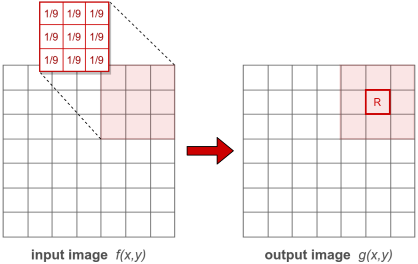
linear spatial filter

⇒ 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)`



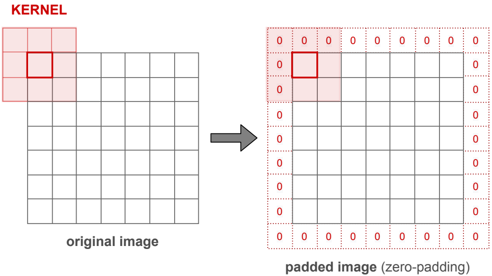
linear spatial filter

⇒ 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)`
- padding = pad the image so the kernel can also operate on the edges `(pad_size=kernel_size//2)`



linear spatial filter

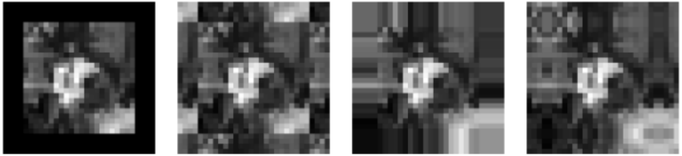
⇒ sum-of-products operation between an **input image f(x,y)** and a **filter kernel w**

- 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

$\Rightarrow$ 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)

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

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

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)

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

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

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

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)

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

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

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

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

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)

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

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

linear spatial filtering $\Longleftrightarrow$ **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° ($\Rightarrow$ when kernel values are symmetric about its center, correlation and convolution yield the same result)

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

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

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

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.

identity

0	0	0
0	1	0
0	0	0

$\Rightarrow$ no change!

LOW PASS FILTER

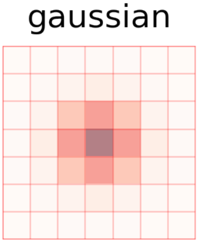
average

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

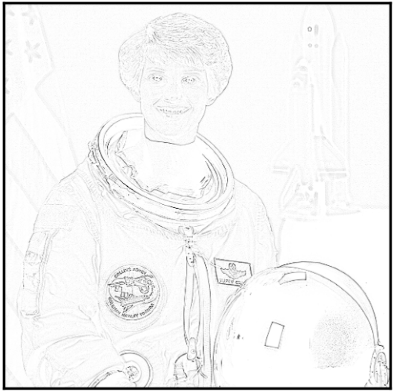


weighted average (low pass)
$\Rightarrow$ blurring effect

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

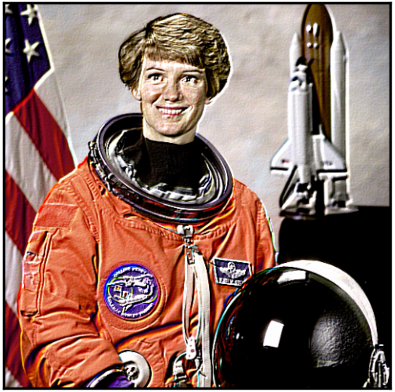
0	-1	0
-1	5	-1
0	-1	0

identity kernel + highpass kernel

$\Rightarrow$ sharpening effect

emboss

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



$\Rightarrow$ styling effect

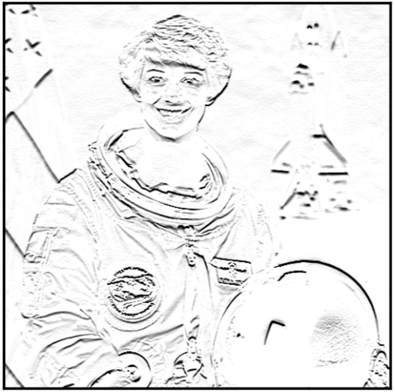
sobel x

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

$\Rightarrow$ edge detection (x-direction)

sobel y

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



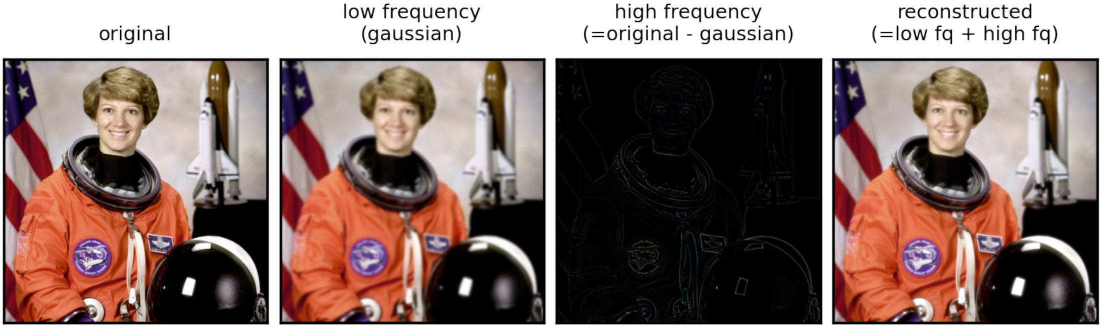
$\Rightarrow$ edge detection (y-direction)

$\Rightarrow$ edges + magnitude

Gaussian filters are a true low-pass filter for the image

$\Rightarrow$ we can retrieve the low-frequency in an image

$\Rightarrow$ we can retrieve the high-frequency in an image by subtracting the low-frequency from the original image



1. Introduction

2. Spatial Domain Filtering
    1. linear spatial filter
    2. convolutions
    3. kernels types and applications

3. Frequency domain filtering
    1. 1D Fourier transform
    2. 2D Fourier transform
    3. Butterworth filter

$\Rightarrow$ convolutions for **spatial domain filtering** is powerful, BUT it has high computational costs

$\Rightarrow$ **frequency domain filtering** offers computational advantages:

*(**convolution** in the time domain $\Longleftrightarrow$ **multiplication** in the frequency domain)*

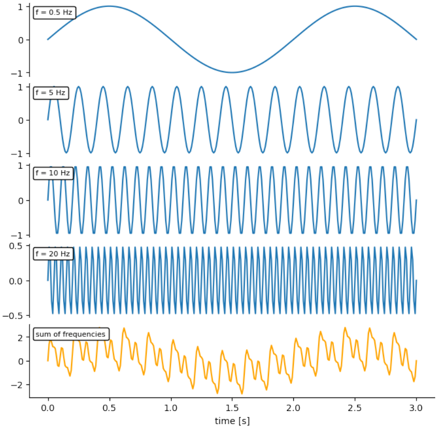
$\Rightarrow$ convolutions for **spatial domain filtering** is powerful, BUT it has high computational costs

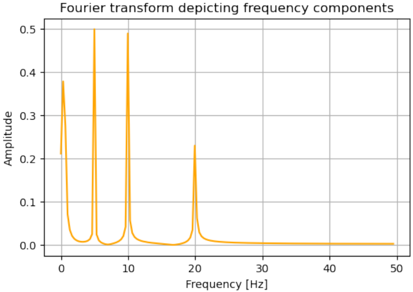
$\Rightarrow$ **frequency domain filtering** offers computational advantages:

*(**convolution** in the time domain $\Longleftrightarrow$ **multiplication** in the frequency domain)*

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



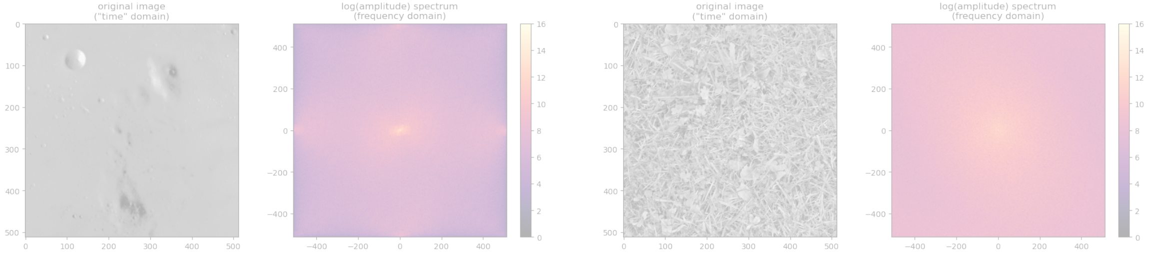


Fourier transform on images ?

⇒ 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 (e.g., the walls in an image of a room)
- **high frequencies** → edges and other sharp intensity transitions

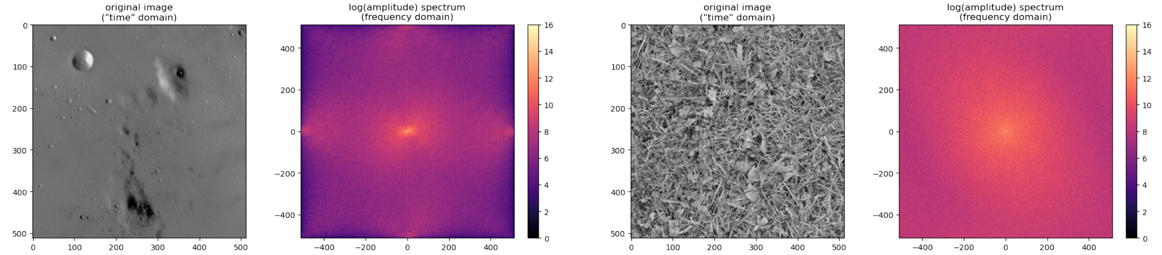


Fourier transform on images ?

⇒ 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)

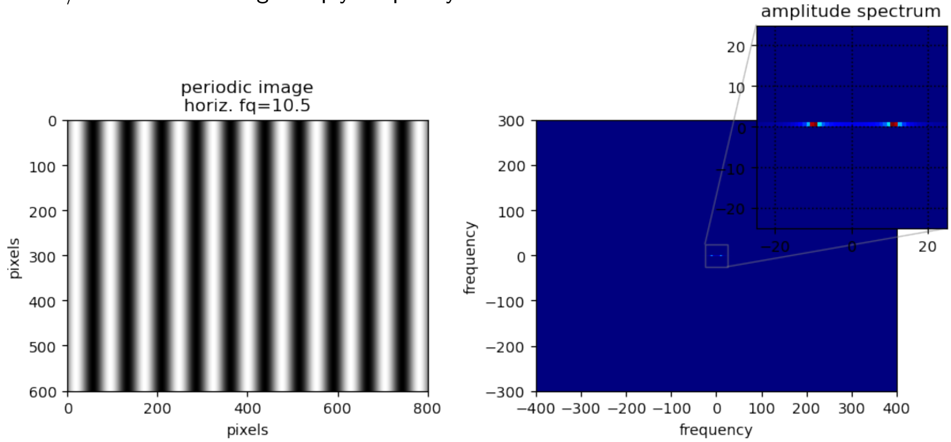
- **low frequencies** → regions with intensities that vary slowly (e.g., the walls in an image of a room)
- **high frequencies** → edges and other sharp intensity transitions



2D Fourier transform on SYNTH images

$\Rightarrow$ "dots" symmetric about origin in amplitude spectrum

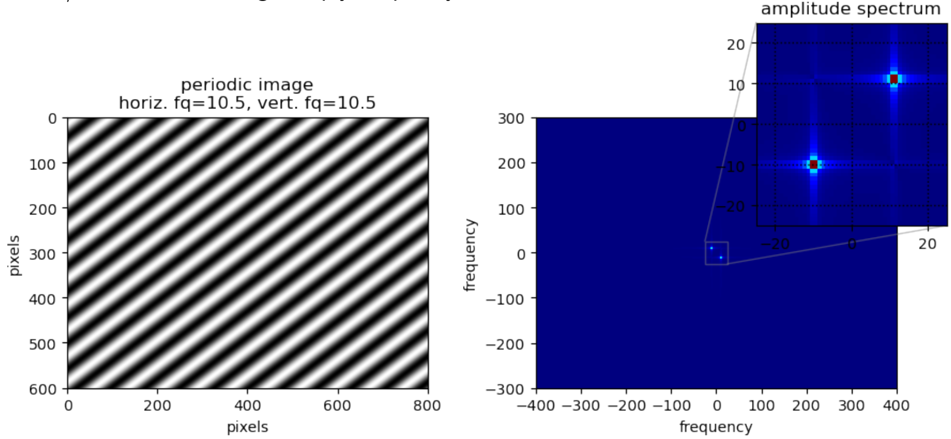
$\Rightarrow$ distance/direction from origin imply frequency in time domain



2D Fourier transform on SYNTH images

⇒ "dots" symmetric about origin in amplitude spectrum

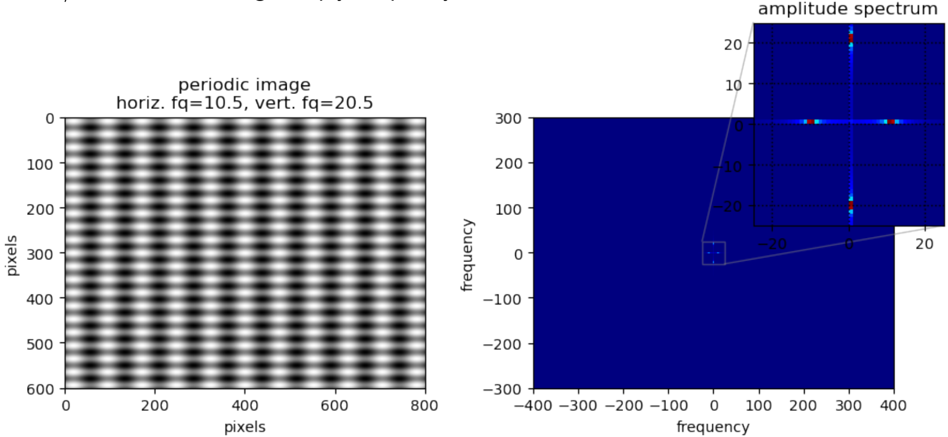
⇒ distance/direction from origin imply frequency in time domain



2D Fourier transform on SYNTH images

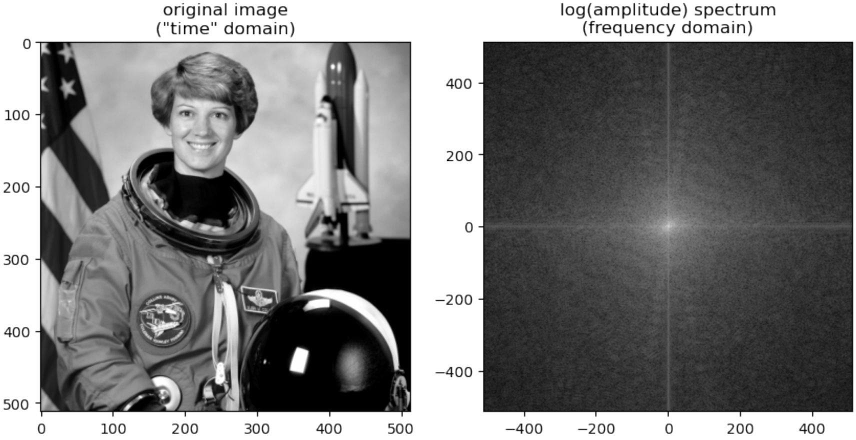
⇒ "dots" symmetric about origin in amplitude spectrum

⇒ distance/direction from origin imply frequency in time domain



2D Fourier transform on REAL images

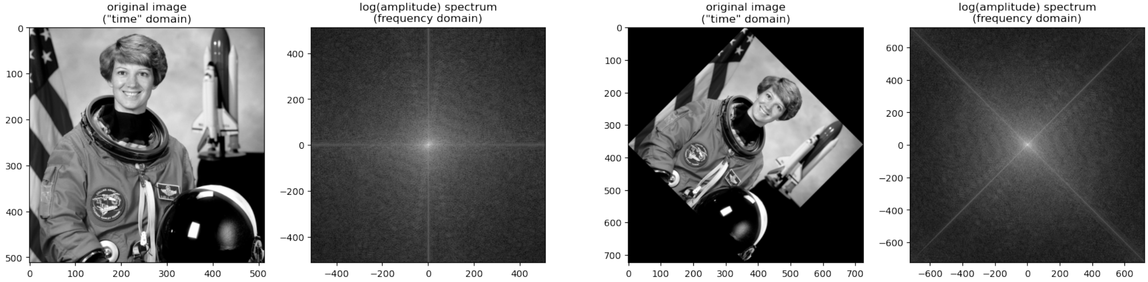
⇒ frequency content concentrated at low frequencies (hence contain more image information than the higher ones)
⇒ amplitude spectrum shows two dominant directions: horizontal & vertical
(dominating directions originate from the regular patterns in the background of the original image)



2D Fourier transform on REAL images

⇒ frequency content concentrated at low frequencies (long wavelengths)

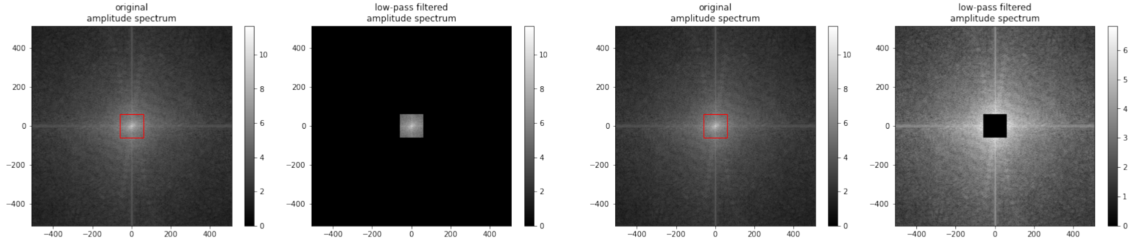
⇒ amplitude spectrum shows two dominant directions: horizontal & vertical



2D Fourier transform on REAL images

$\Rightarrow$ band-pass image frequencies

- **low-pass** filter $\rightarrow$ cut off high-frequencies
- **high-pass** filter $\rightarrow$ cut off low-frequencies



2D Fourier transform on REAL images

$\Rightarrow$ image can be reconstructed using the **inverse Fourier transform**

2D Fourier transform on REAL images

⇒ 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

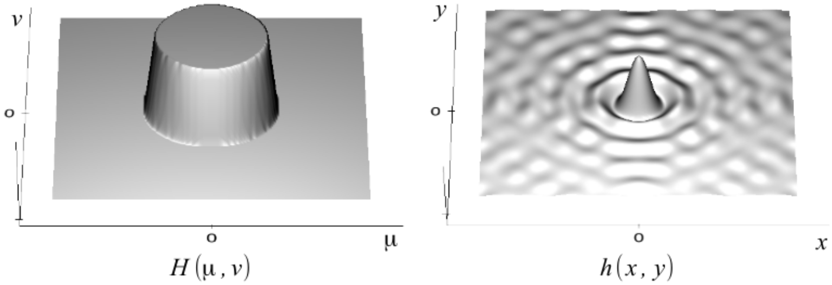
low-pass filtered image

ringing effect

2D Fourier transform on REAL images

⇒ 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

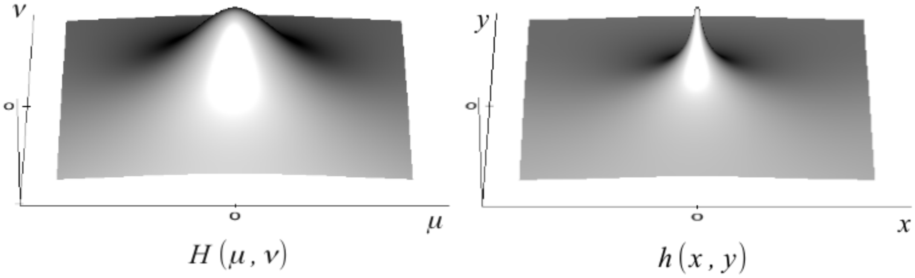


- Ideal LPF has significant 'side-lobes' in the time domain

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