Applied Computer Vision

David Vernon
Carnegie Mellon University Africa

vernon@cmu.edu
www.vernon.eu

Lecture 4

Image processing

Point Operations, Neighbourhood Operations,
Image Filtering, Convolution, Fourier Transform

Image Processing

Image processing

- Image to image transformation
- It starts with an image and produces a modified (enhanced) image
- Iconic to iconic transformation

Image analysis

- Image to information transformation
- It starts with an image and produces information representing a description or a decision
- Iconic to symbolic transformation

Image Processing

The Image Processing Phase should :

- facilitate the extraction of information
- compensate for non-uniform illumination
- re-adjust the image to compensate for distortions introduced by the imaging system

Image Processing

There are 3 distinct classes of operations

1. Point Operations

2. Neighbourhood Operations
   - Linear filtering / convolution operations
   - Fourier transform
   - Logical non-linear operations

3. Geometric Operations

Point Operations

Each pixel in the output image is a function of the grey-level of the pixel at the corresponding position in the input image
and only of that pixel

They cannot alter the spatial relationships of the image

Point Operations

Typical uses of Point Operations include:

- Photometric Decalibration
  to remove the effects of spatial variations in the sensitivity of a camera system

- Contrast Stretching
  e.g. if a feature or object occupies a relatively small section of the total grey-scale image, these local operations can manipulate the image so that it occupies the entire range

- Thresholding
  in which all the pixels having grey-levels in specified ranges in the input image are assigned a single specific grey-level in the output image

Point Operations

Background Subtraction

- Pixel values of two images are subtracted on a point by point basis.
- Subtraction of a *known* pattern (or image) of super-imposed noise, e.g. photometric decalibration (compensation for vingnetting)
- Motion Detection: stationary objects cancel each other out while moving objects are highlighted.

Point Operations

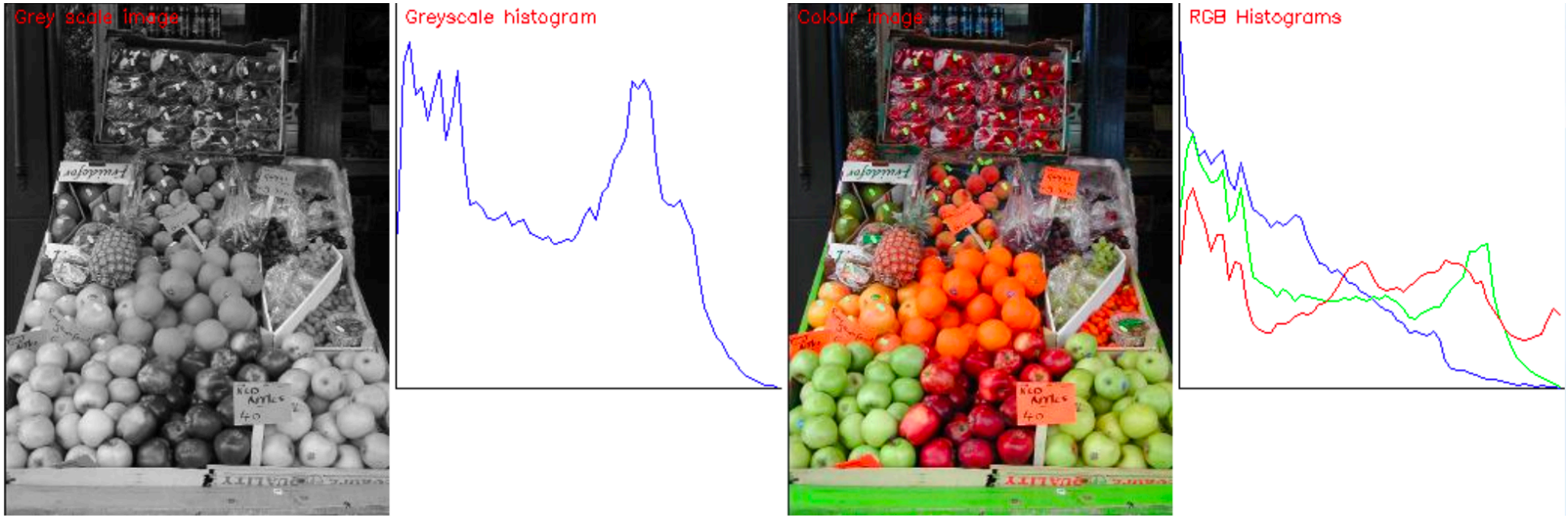
Photometric decalibration for vignetting

- Acquire an image of a uniformly-shaded surface
  - Identify the minimum grey-level of this image
  - Subtract this value from each pixel to generate an image which represents the effective response of the sensor

- Images which are subsequently acquired can then be decalibrated by subtracting this calibration image from them

Point Operations

Image Histogram

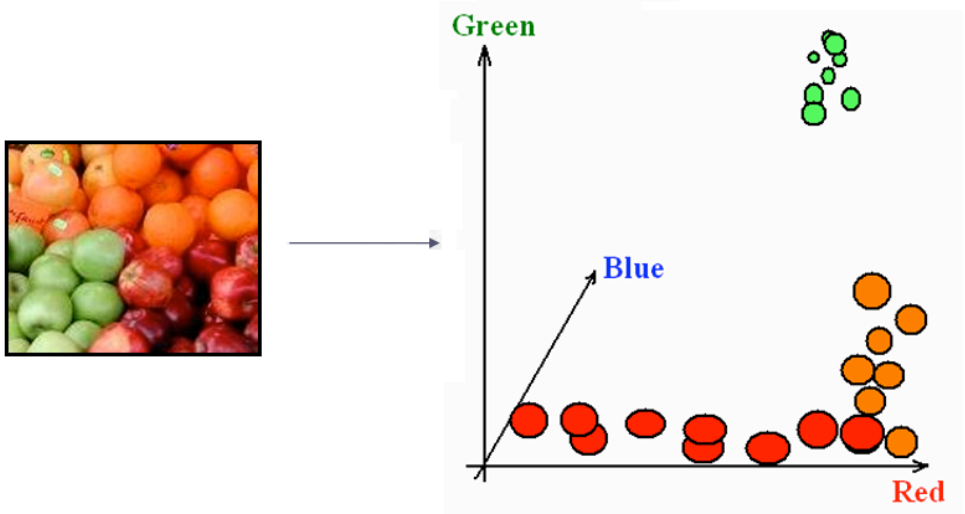


Credit: Kenneth Dawson-Howe, A Practical Introduction to Computer Vision with OpenCV, © Wiley & Sons Inc. 2014

Point Operations

3D Image Histogram

- Channels are not independent
- Better discrimination by considering all channels simultaneously

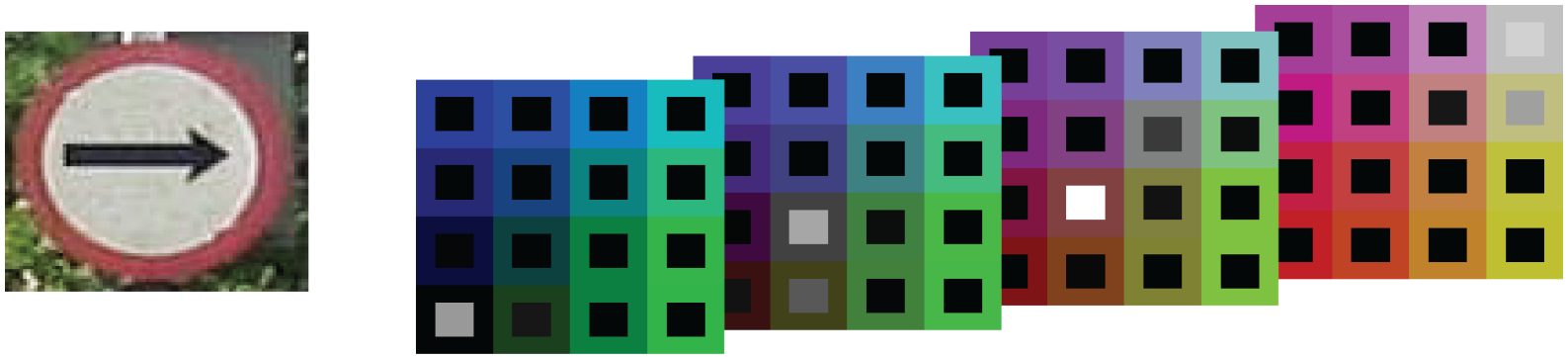


Credit: Kenneth Dawson-Howe, A Practical Introduction to Computer Vision with OpenCV, © Wiley & Sons Inc. 2014

Point Operations

3D Image Histogram

- Number of cells very large
  - 8 bits =16,777,216
- Reduce quantisation
  - 6 bits = 262,144
  - 4 bits = 4,096
  - 2 bits = 64



Credit: Kenneth Dawson-Howe, A Practical Introduction to Computer Vision with OpenCV, © Wiley & Sons Inc. 2014

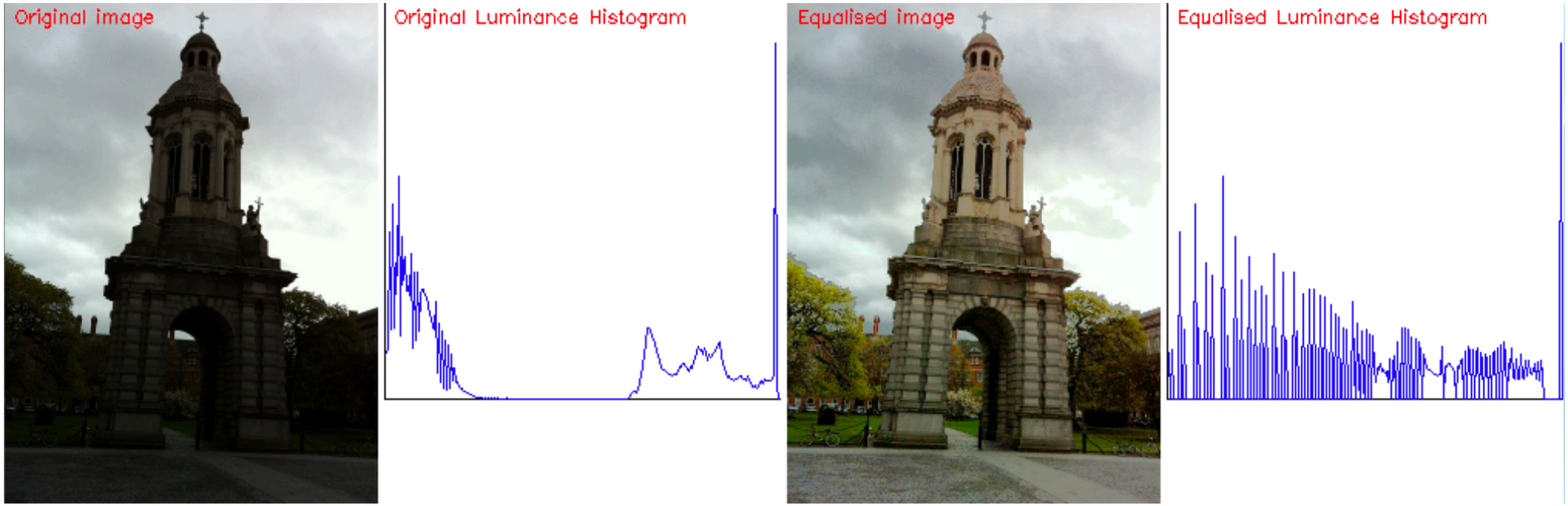
Point Operations

Histogram Equalisation

- Compute the histogram of the grey-level intensity (or hue, or other image feature) distribution in the image

- Re-assign value to pixels in an effort to generate a histogram where there are equally many pixels at every grey-level

Point Operations



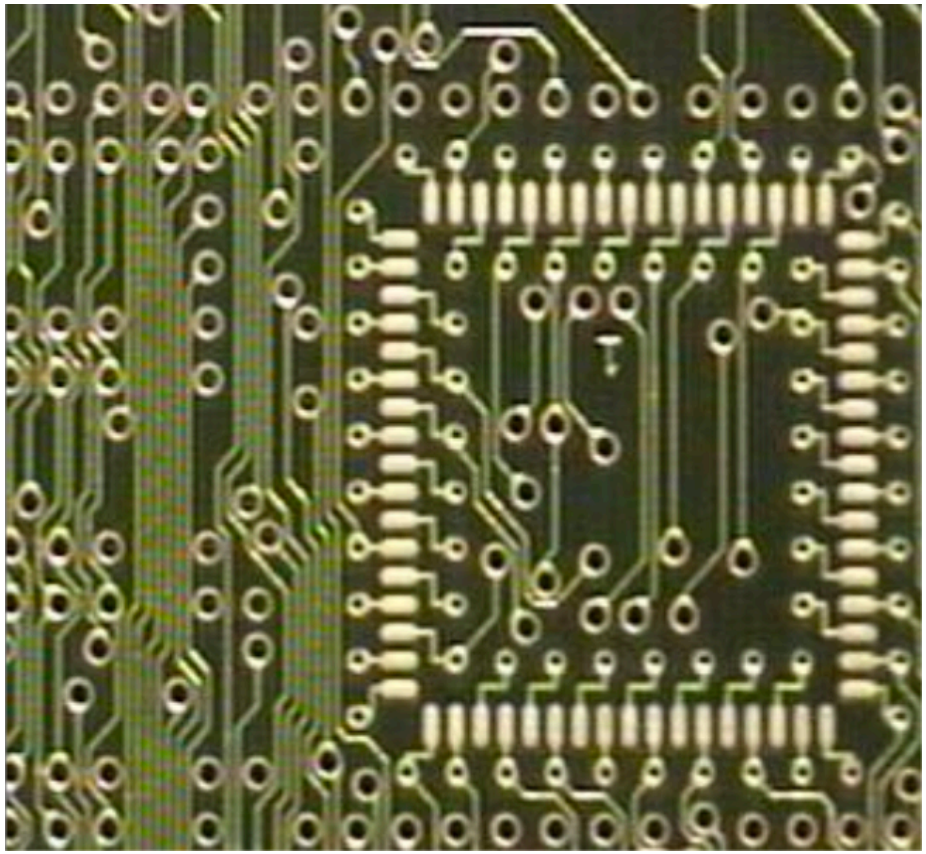
Credit: Kenneth Dawson-Howe, A Practical Introduction to Computer Vision with OpenCV, © Wiley & Sons Inc. 2014

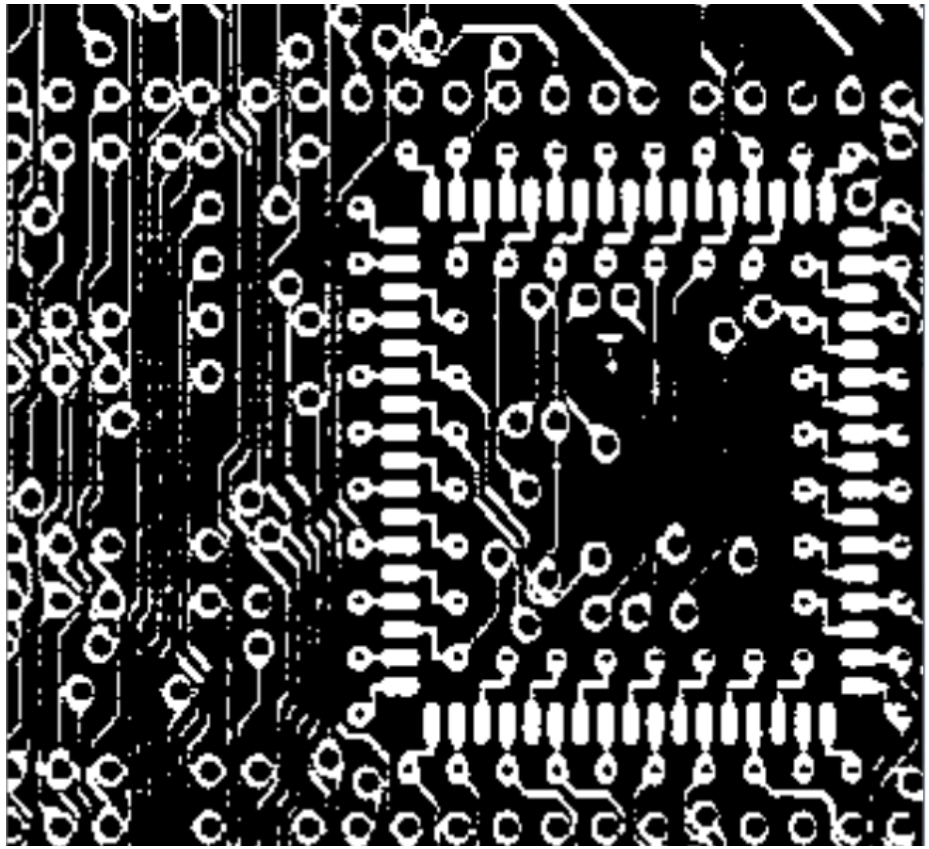
Point Operations

Thresholding

- Elementary segmentation technique (see later) that
  - assigns a value of 0 or 255 to each pixel
  - depending on whether the image value is less than or greater than the threshold
- This is effectively a labelling process
  - Label pixels background
  - Label pixels foreground / object

Point Operations





Credit: Kenneth Dawson-Howe, A Practical Introduction to Computer Vision with OpenCV, © Wiley & Sons Inc. 2014

Neighbourhood Operations

- Generate an output pixel on the basis of the pixel at the corresponding position in the input image and on the basis of its neighbouring pixels

- The size of the neighbourhood may vary :

  3 * 3
  5 * 5
  63 * 63 pixels

- Often referred to as Filtering Operations

Convolution of an image f with a filter kernel or mask h

$$g = f * h$$

Neighbourhood Operations

Other Neighbourhood Operations

- Applying some logical test

  based on, e.g. the presence or absence of object pixels in a local neighbourhood surrounding the pixel in question

- Object Thinning (or Skeletonising)

- Erosion and Dilation (contract/expand an object)

Image Filtering

The 2D Convolution Integral

$$g(i,j) = f * h = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(i-m, j-n)h(m,n)dmdn$$

- output g of a shift-invariant linear system

- is given by the convolution or application of the input signal f with a function h which is characteristic of the system

Image Filtering

- The function h is normally referred to as the filter
  - It dictates what elements of the input image are allowed to pass through to the output image
  - By choosing an appropriate filter, we can enhance certain aspects of the output and attenuate others
  - A particular filter h is often referred to as a filter kernel
- The form of g depends on
  - The input f (obviously)
  - The form of the system h through which it is being passed
  - The relationship is given by the Convolution Integral

Image Filtering

In the discrete domain of digital images the Convolution operation is given by:

$$g(i,j) = f * h = \sum_m \sum_n f(i-m, j-n) h(m,n)$$

The summation is taken only over the area where $(i\text{-}m, j\text{-}n)$ is defined, *i.e.* over the area where f and h overlap

Image Filtering

$h(-1,-1)$	$h(-1,0)$	$h(-1,+1)$
$h(0,-1)$	$h(0,0)$	$h(0,+1)$
$h(+1,-1)$	$h(+1,0)$	$h(+1,+1)$

3 * 3 Convolution Filter h

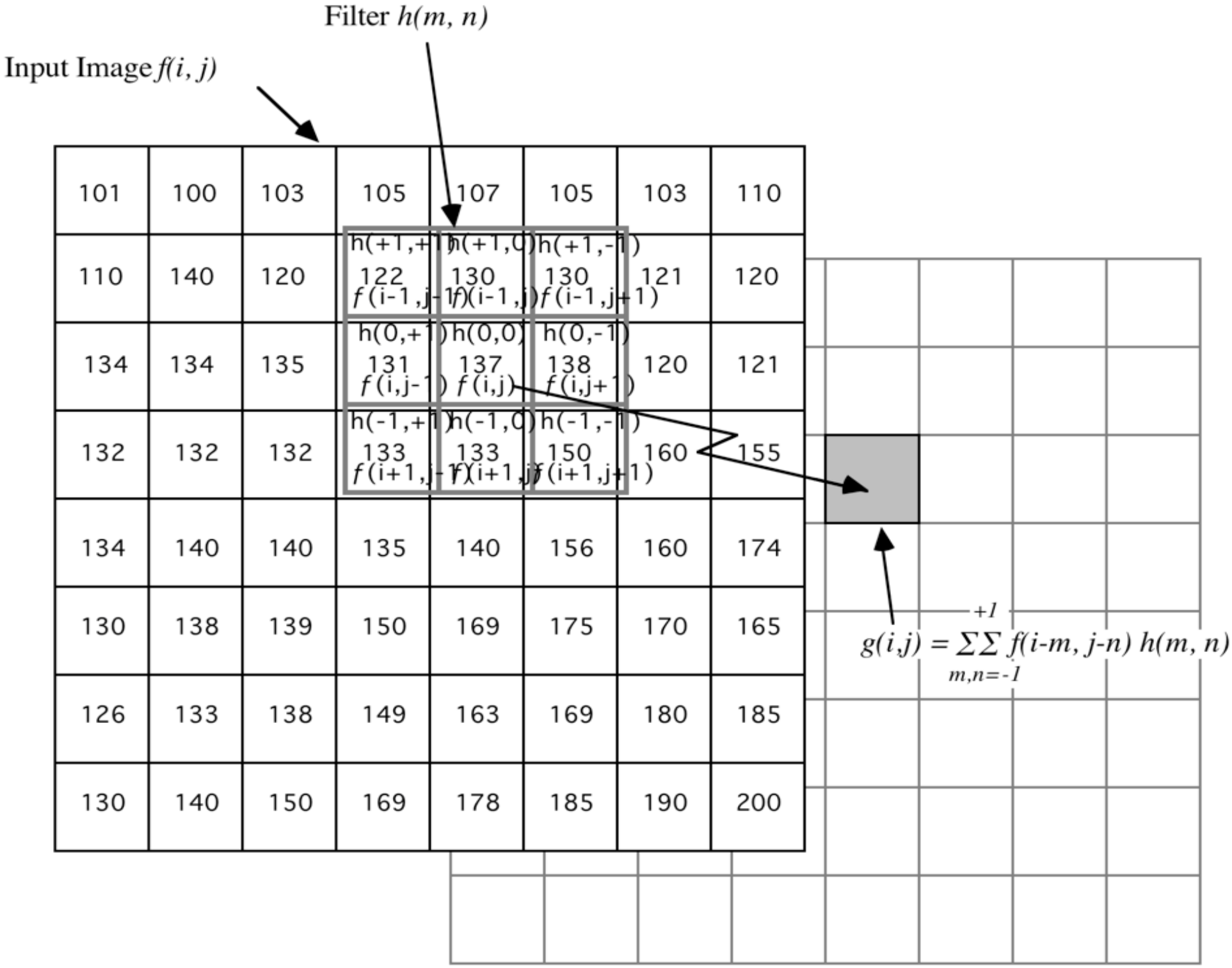

Filter $h(m, n)$
Input Image $f(i, j)$
101 100 103 105 107 105 103 110
110 140 120 122 130 130 121 120
134 134 135 131 137 138 120 121
132 132 132 133 133 150 160 155
134 140 140 135 140 156 160 174
130 138 139 150 169 175 170 165
126 133 138 149 163 169 180 185
130 140 150 169 178 185 190 200
h(+1,+1) h(+1,0) h(+1,-1)
f(i-1,j-1) f(i-1,j) f(i-1,j+1)
h(0,+1) h(0,0) h(0,-1)
f(i,j-1) f(i,j) f(i,j+1)
h(-1,+1) h(-1,0) h(-1,-1)
f(i+1,j-1) f(i+1,j) f(i+1,j+1)
$g(i,j) = \sum_{m,n=-1}^{+1} \sum f(i-m, j-n)\, h(m, n)$

Image Filtering

Note that the mask is first rotated by 180° since

- $f(i\text{-}1, j\text{-}1)$ must be multiplied by $h(1, 1)$
- $f(i\text{-}1, j)$ must be multiplied by $h(1,0)$
- ..... ,
- and $f(i+1, j+1)$ must be multiplied by $h(-1,-1)$

Often the rotation by 180° is omitted if the mask is symmetric

Image Filtering

- All image systems introduce some amount of distortion into an image
- Since optical and electrical systems are (to an approximation) linear, the imaging system may also be assumed to be linear
- This implies that it has some transfer function $H(\omega_x, \omega_y)$
- This is the frequency domain equivalent of the system impulse response $h(x, y)$ in the spatial domain
- $h(x, y)$ captures how the system would respond if a single unit spike were input to it

Image Filtering

Deconvolution

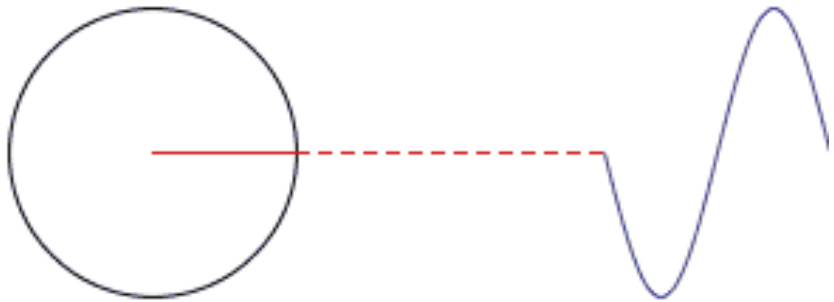
- Model the distortion introduced by a system by identifying its transfer function (equivalently, its impulse response)
- Convolve the image with the inverse of the transfer function
- Rarely stable

Image Filtering

Convolution

- The image may be transformed to the frequency domain using the Digital Fourier Transform
- Multiplied by the inverse of the transfer function
- Transformed back to the spatial domain by use of the Inverse Fourier Transform
- The transfer function describes how some input image is transformed into some output image

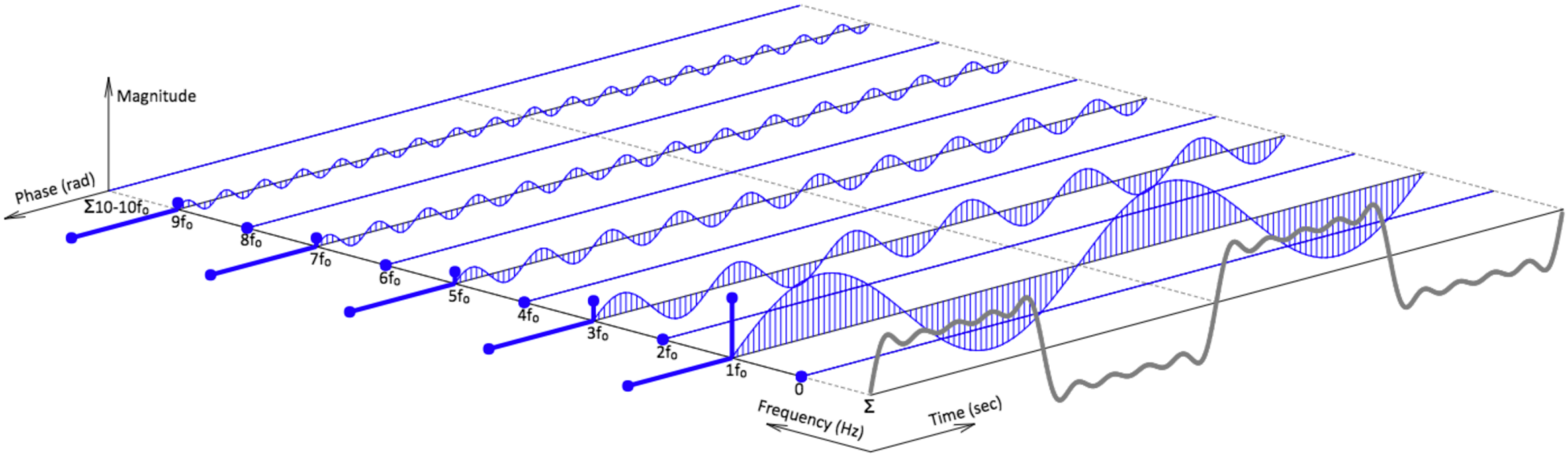
Fourier Transform



Fourier Transform

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

Fourier Transform



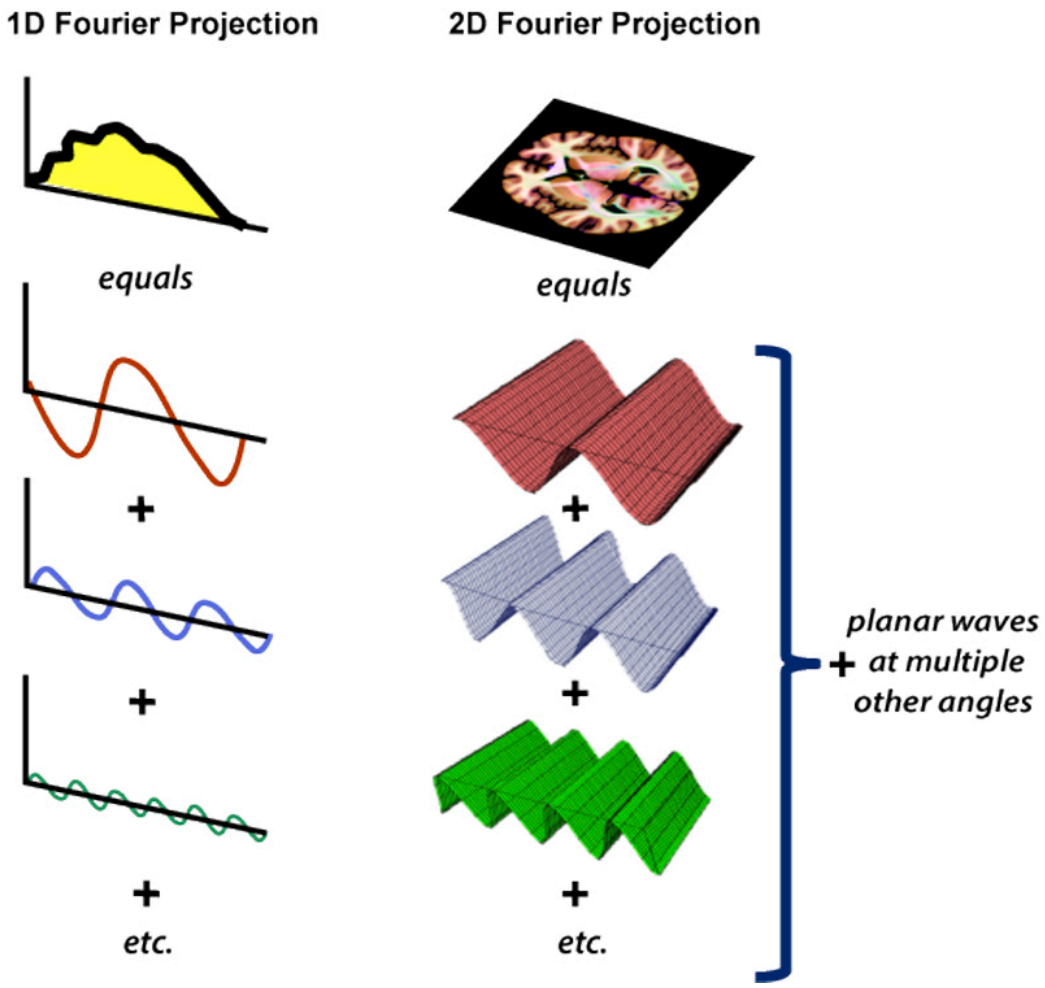
Clear | Cosine | Sine | Square (sin) | Square (cos) | Default | **Harmonics: 10**

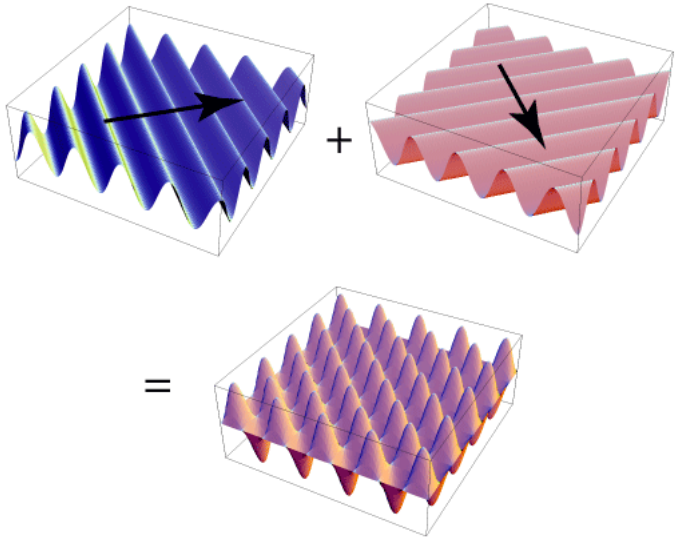
Beats | Sawtooth | Triangle | Impulse | Sawtooth 2 | **Sound** | **Volume** | **f0: 329.6 Hz** | fs: 44100 Hz

(c) Tomáš Bořil, borilt@gmail.com, 2016 Prague, v0.95. JavaScript & Canvas template: Filip Paulů.

http://www.tomasboril.cz/fourierseries3d/en/#

Fourier Transform





http://mriquestions.com/what-are-2d---3d-fts.html

Fourier Transform

Fourier Transform

$$\begin{aligned}\mathcal{F}\left(f(x,y)\right) &= \mathsf{F}(\omega_x,\omega_y) \\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)e^{-i(\omega_x x+\omega_y y)}\mathrm{d}x\mathrm{d}y\end{aligned}$$

Inverse Fourier Transform

$$\begin{aligned}f(x,y) &= \mathcal{F}^{-1}\left(\mathsf{F}(\omega_x,\omega_y)\right) \\ &= \frac{1}{(2\pi)^2}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \mathsf{F}(\omega_x,\omega_y)e^{i(\omega_x x+\omega_y y)}\mathrm{d}\omega_x\mathrm{d}\omega_y\end{aligned}$$

$$i=\sqrt{-1}$$

$$-\infty \le \omega_x,\omega_y \le +\infty$$

Fourier Transform

Fourier Transform

Real-valued continuous two-dimensional spatial function
x and y denote displacement in orthogonal directions

$$\begin{aligned}\mathcal{F}\left(f(x,y)\right) &= \mathsf{F}(\omega_x,\omega_y)\\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)e^{-i(\omega_x x+\omega_y y)}\mathrm{d}x\mathrm{d}y\end{aligned}$$

Inverse Fourier Transform

$$\begin{aligned}f(x,y) &= \mathcal{F}^{-1}\left(\mathsf{F}(\omega_x,\omega_y)\right)\\ &= \frac{1}{(2\pi)^2}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \mathsf{F}(\omega_x,\omega_y)e^{i(\omega_x x+\omega_y y)}\mathrm{d}\omega_x\mathrm{d}\omega_y\end{aligned}$$

$$i=\sqrt{-1}$$

$$-\infty \le \omega_x,\omega_y \le +\infty$$

Fourier Transform

Transformed function

ω_x and ω_x denote **spatial frequencies** in x and y directions

Fourier Transform

$$\begin{aligned}\mathcal{F}\left(f(x,y)\right) &= \mathsf{F}(\omega_x,\omega_y)\\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)e^{-i(\omega_x x+\omega_y y)}\mathrm{d}x\mathrm{d}y\end{aligned}$$

Inverse Fourier Transform

$$\begin{aligned}f(x,y) &= \mathcal{F}^{-1}\left(\mathsf{F}(\omega_x,\omega_y)\right)\\ &= \frac{1}{(2\pi)^2}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \mathsf{F}(\omega_x,\omega_y)e^{i(\omega_x x+\omega_y y)}\mathrm{d}\omega_x\mathrm{d}\omega_y\end{aligned}$$

$$i=\sqrt{-1}$$

$$-\infty \le \omega_x,\omega_y \le +\infty$$

Fourier Transform

$f(x,y)$ can be constructed from a linear combination of elementary functions having the form $e^{i(\omega_x x+\omega_y y)}$

each appropriately weighted in amplitude and phase by a complex factor $\mathrm{F}(\omega_x,\omega_y)$

$$
\begin{aligned}
f(x,y) &= \mathcal{F}^{-1}\left(\mathrm{F}(\omega_x,\omega_y)\right) \\
&= \frac{1}{(2\pi)^2}\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\mathrm{F}(\omega_x,\omega_y)e^{i(\omega_x x+\omega_y y)}\mathrm{d}\omega_x\mathrm{d}\omega_y
\end{aligned}
$$

Fourier Transform

$f(x,y)$ can be constructed from a linear combination of elementary functions having the form $e^{i(\omega_x x+\omega_y y)}$

each appropriately weighted in amplitude and phase by a complex factor $\mathrm{F}(\omega_x,\omega_y)$

... which are given by the Fourier Transform and vary with the content of the image ...

$$\begin{aligned}\mathcal{F}\left(f(x,y)\right) &= \mathrm{F}(\omega_x,\omega_y)\\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)e^{-i(\omega_x x+\omega_y y)}\mathrm{d}x\mathrm{d}y\end{aligned}$$

Fourier Transform

$F(\omega_x, \omega_y)$ is a complex function:

$$F(\omega_x, \omega_y) = A(\omega_x, \omega_y) + iB(\omega_x, \omega_y)$$

Real component

Imaginary component

Fourier Transform

Phasor notation

$$\mathsf{F}(\omega_x, \omega_y) = \mathsf{A}(\omega_x, \omega_y) + i\mathsf{B}(\omega_x, \omega_y)$$

$$\begin{aligned} \mathsf{A}(\omega_x, \omega_y) &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x', y')\cos(\omega_x x' + \omega_y y')\mathrm{d}x'\mathrm{d}y' \\ \mathsf{B}(\omega_x, \omega_y) &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x', y')\sin(\omega_x x' + \omega_y y')\mathrm{d}x'\mathrm{d}y' \end{aligned}$$

$$\mathsf{F}(\omega_x, \omega_y) = |\mathsf{F}(\omega_x, \omega_y)|\, e^{i\phi(\omega_x, \omega_y)}$$

Magnitude

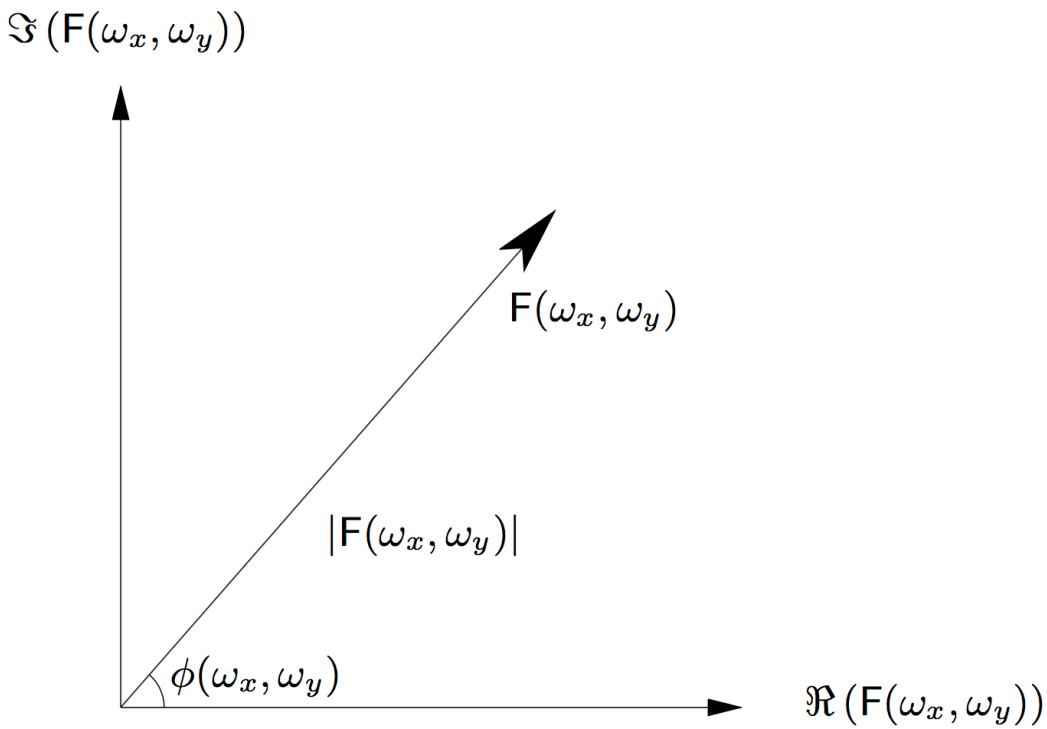
Phase

Fourier Transform

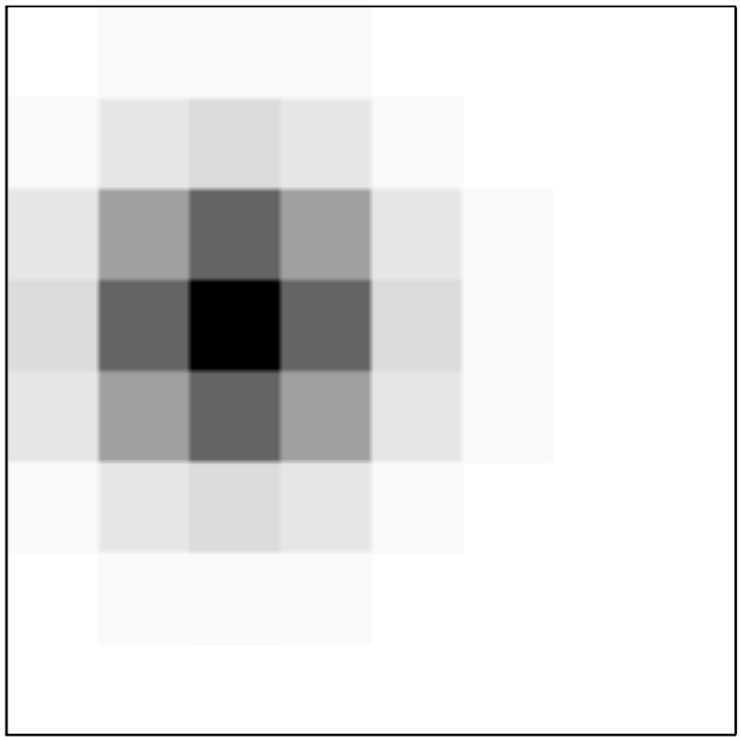
Phasor notation

$$|F(\omega_x, \omega_y)| = \sqrt{A^2(\omega_x, \omega_y) + B^2(\omega_x, \omega_y)}$$

$$\phi(\omega_x, \omega_y) = \arctan \frac{B(\omega_x, \omega_y)}{A(\omega_x, \omega_y)}$$



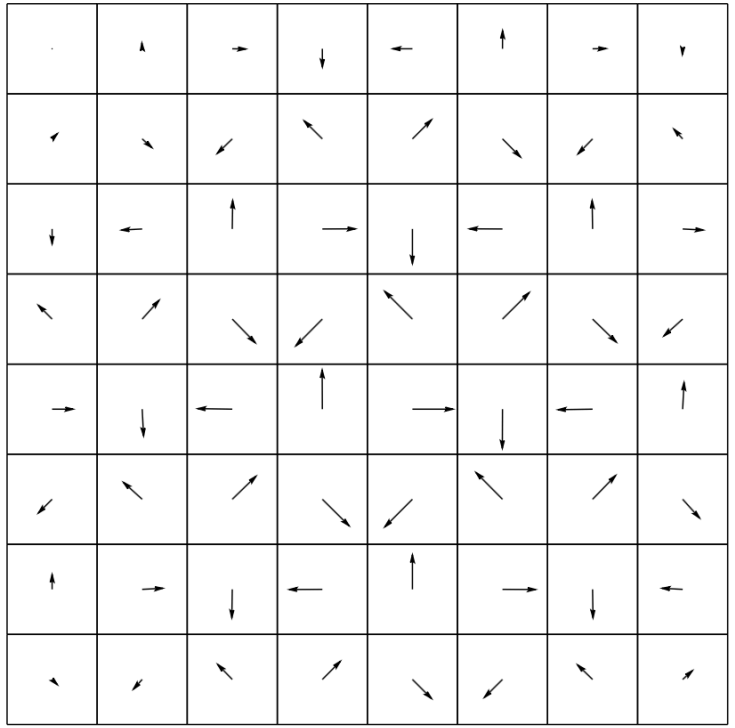
Fourier Transform



$f(x, y)$

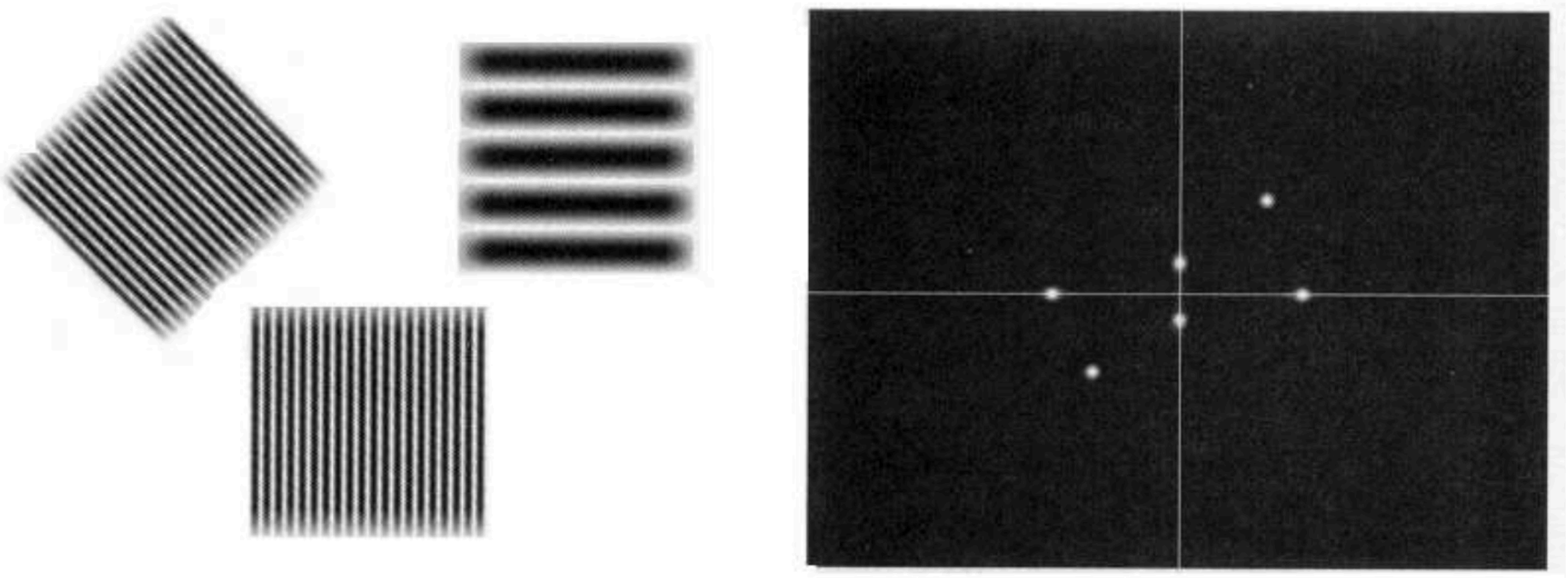
$\mathcal{F}\left(f(x, y)\right)$

$\mathcal{F}^{-1}\left(\mathrm{F}(\omega_x, \omega_y)\right)$



$\mathrm{F}(\omega_x, \omega_y)$

Fourier Transform



Credit: B. Osgood, The Fourier Transform and its Applications, Chapter 8, Stanford University

Fourier Transform

Property	Signal		Transform
superposition	$f_1(x) + f_2(x)$		$F_1(\omega) + F_2(\omega)$
shift	$f(x - x_0)$		$F(\omega)e^{-j\omega x_0}$
reversal	$f(-x)$		$F^*(\omega)$
convolution	$f(x) * h(x)$		$F(\omega)H(\omega)$
correlation	$f(x) \otimes h(x)$		$F(\omega)H^*(\omega)$
multiplication	$f(x)h(x)$		$F(\omega) * H(\omega)$
differentiation	$f'(x)$		$j\omega F(\omega)$
domain scaling	$f(ax)$		$1/aF(\omega/a)$
real images	$f(x) = f^*(x)$	$\Leftrightarrow$	$F(\omega) = F(-\omega)$
Parseval's Theorem	$\sum_x [f(x)]^2$	$=$	$\sum_\omega [F(\omega)]^2$

Convolution Theorem

Image Filtering

Noise

- Any unwanted contamination of an image
- The mechanism by which noise is removed depends on the assumption we make regarding the form of this unwanted contamination

Image Filtering

Noise

- Common (and realistic) assumption made about noise is that it has a high spatial frequency
  - apply a low-pass spatial filter which will
  - attenuate the higher spatial frequencies
  - allow the low spatial frequencies component to pass through to the resultant destination image
- If the image itself exhibits high spatial frequencies then it will be somewhat degraded after filtering

1	1	1
1	1	1
1	1	1

Local Average Mask

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

Local Average Mask

101 * 1/9	100 * 1/9	103 * 1/9	105	107	105	103	110
110 * 1/9	140 * 1/9	120 * 1/9	122	130	130	121	120
134 * 1/9	134 * 1/9	135 * 1/9	131	137	138	120	121
132	132	132	133	133	150	160	155
134	140	140	135	140	156	160	174
130	138	139	150	169	175	170	165
126	133	138	149	163	169	180	185
130	140	150	169	178	185	190	200

Image smoothing using local average mask

The central point becomes 121 instead of 140 and the image will appear much smoother

Image Filtering

Other noise suppression techniques

Median filter

- A pixel is assigned the value of the median of pixel values in some local neighbourhood
- The size of the neighbourhood is arbitrary
- Median filter is superior to the mean filter in that image blurring is minimised

Image Filtering

Other noise suppression techniques

Gaussian filter

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}}$$

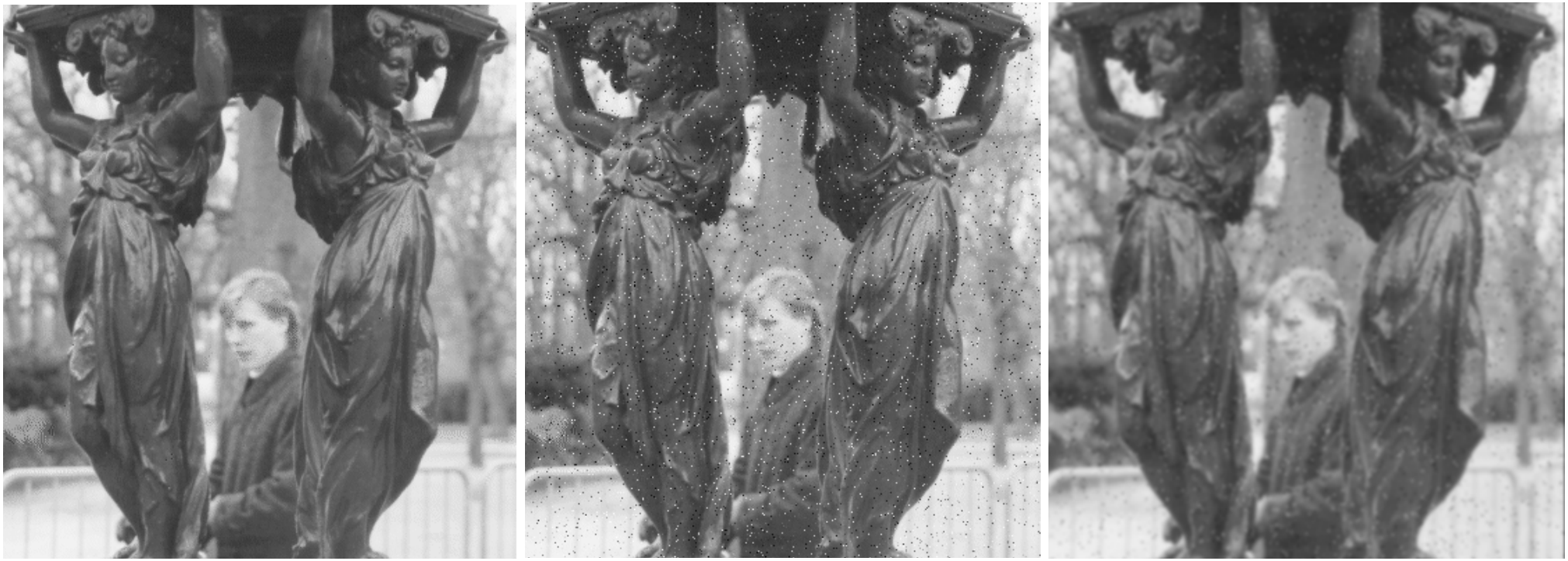
σ defines the effective spread of the function

- Gaussian functions with a small value for σ are narrow
- Those with with a large value for σ are broad
- Note that, since the Gaussian function is defined over an infinite support, i.e. it has non-zero (but very small) values at we must truncate the function

Image Filtering

Salt and pepper noise: 1% of pixels either black or white

Smoothing: outliers are spread but not eliminated



Original image | Salt and Pepper | Gaussian filter σ=8, 5x5

Credit: Markus Vincze, Technische Universität Wien

Image Filtering

Median Filter: Salt and Pepper (5%)

Eliminates outliers (up to 50%)

	123	125	126	130	140
	122	124	126	127	135
	118	120	150	125	134
	119	115	119	123	133
	111	116	110	120	130

Neighbourhood values:

115, 119, 120, 123, 124, 125, 126, 127, 150

Median value: 124

Median filter 3x3

Median filter 7x7

3x Median filter 3x3

Credit: Markus Vincze, Technische Universität Wien

Image Filtering

The Gaussian function for three values of σ

1.5,

3.0

6.0

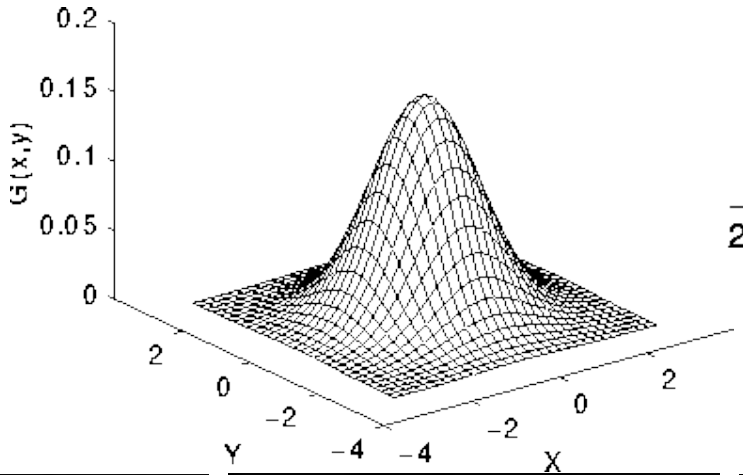
with their corresponding discrete 1-D masks

		1
		2
		3
		6
		11
		18
		28
		43
		65
		95
		135
	1	186
	3	249
	11	324
	28	411
	65	506
	135	606
3	249	706
28	411	800
135	606	882
411	800	945
800	945	986
1000	1000	1000
800	945	986
411	800	945
135	606	882
28	411	800
3	249	706
	135	606
	65	506
	28	411
	11	324
	3	249
	1	186
		135
		95
		65
		43
		28
		18
		11
		6
		3
		2
		1

Note that the result of convolution with these masks should be normalised by dividing by the sum of the mask weights

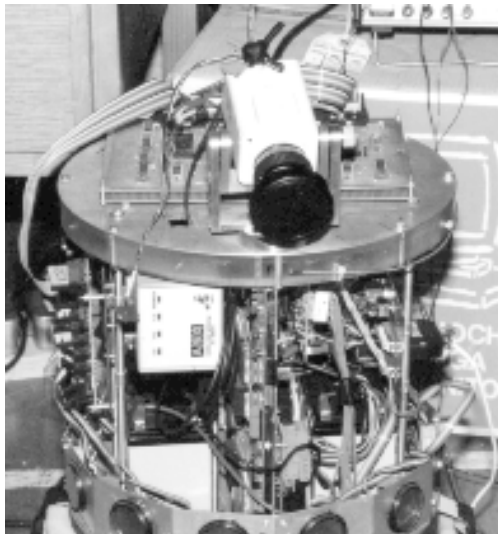
Image Filtering

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{\frac{-x^2+y^2}{\sigma^2}}$$

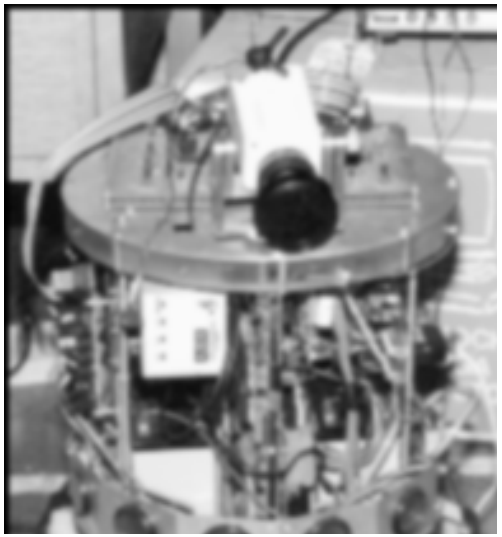


$$\frac{1}{273}$$

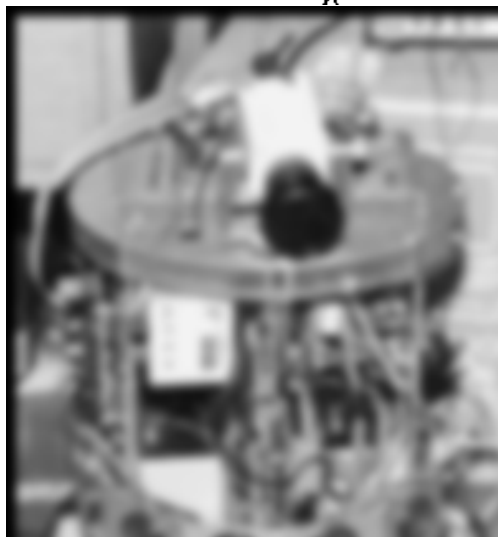
1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1



Original image



σ =1, 5x5



σ =2, 9x9

σ =4, 15x15

Credit: Markus Vincze, Technische Universität Wien

Image Filtering

A filter size of 23*23 pixels would typically be required to represent a 2-D Gaussian with a value of 3.0 pixels for σ

Fortunately, there is no need to use 2-D Gaussian functions since the convolution of a 2-D Gaussian can be effected by two convolutions with 1-D Gaussian functions

$$G(x, y) * I(x, y) = G(x) * \{G(y) * I(x,y)\}$$

- the image is first convolved with a 'vertical' Gaussian
- the resulting image is convolved with a 'horizontal' Gaussian.

Image Filtering

Why is the Gaussian such a popular smoothing function?

It can be adjusted to control the level of detail in the image

- Small σ will retain a significant amount of detail
- Large σ will retain only the gross structure

Image Filtering

The Gaussian function optimizes the trade-off between two conflicting requirements

- Localization in the spatial frequency domain
- Localization of the space domain

- It has the desirable characteristic of being smooth and localized in both the spatial and frequency domains

- It is the unique distribution that is simultaneously optimally localized in both domains

- Thus it is least likely to introduce any changes that were now present in the original image

Demos

The following code is taken from the photometricDecalibration project in the lectures directory of the ACV repository

See:

```
photometricDecalibation.h
photometricDecalibationImplementation.cpp
photometricDecalibationApplication.cpp
```

```
/*
  Example use of openCV to acquire images from a video camera and perform photometric decalibration using image subtraction
  ---------------------------------------------------------------------------------------------------------------------------

  (This is the implementation file: it contains the code for dedicated functions to implement the application.
  These functions are called by client code in the application file. The functions are declared in the interface file.)

  David Vernon
  26 October 2018

*/

#include "photometricDecalibration.h"

/*=============================================================================================================
  display_decalibrated_image_from_camera()
  display images from a camera in an openCV window after photometric decalibration

  Parameters: index of the camera
              mode: 0 generate a new decalibation image; 1 use decalibration image from file
              filename: path and filename where decalibration image is to be read or stored, depending on mode
  ===========================================================================================================*/
void display_decalibrated_image_from_camera(int cameraNum, int mode, char *filename) {

   bool  debug = false;
   VideoCapture camera;                  //  the camera device
   Mat inputImage;                       //  grabbed image
   Mat decalibratedImage;                //  decalibrated image
   Mat hlsImage;                         //  hue, luminance, and saturation image
   std::vector<cv::Mat> input_planes(3); // three images for hue, luminance, and saturation
   Mat greyscaleImage;                   //  greyscale image ... we are only going to decalibrate the intensity
   Mat decalibrationImage;               //  save an image read from a camera
   vector<int> compressionParams;        //  parameters for image write
   int row, col;
   int min, max, value;
   double gaussian_std_dev;
   int filter_size;
   gaussian_std_dev = 6;
   filter_size = (int) (gaussian_std_dev) * 4 + 1;
   char windowName[MAX_STRING_LENGTH];
   char cameraNumber[MAX_STRING_LENGTH];

   if (debug) {
       printf("display_decalibrated_image_from_camera: camera %d, image filename %s\n",cameraNum, filename);
   }

   strcpy(windowName,"Camera");
   sprintf(cameraNumber, " %d", cameraNum);

   namedWindow(windowName,     CV_WINDOW_AUTOSIZE); // create the window
```

```
if (camera.open(cameraNum) == true) {          // open the camera input

   if (mode == 0) {

      /* generate a new decalibration image */
      printf("Place photometric decalibration object in front of the camera; press any key when done.\n\n");

      do {
         camera >> inputImage;                          // read a frame from the camera
         imshow(windowName, inputImage);
         cvWaitKey(30); // this is essential as it allows openCV to handle the display event ...
                        // the argument is the number of milliseconds to wait
      } while (!_kbhit());

      getchar(); // flush the buffer from the keyboard hit

      if (inputImage.type() == CV_8UC1) {     // greyscale image
         printf("photometricDecalibration: error - expecting a colour image\n");
         prompt_and_continue();
      }
      else if (inputImage.type() == CV_8UC3) { // colour image

         /* separate the colour channels */
         cvtColor(inputImage, hlsImage, CV_BGR2HLS);
         split(hlsImage,input_planes);

         /* decalibration is based on the luminance component */
         input_planes[1].copyTo(greyscaleImage);

         GaussianBlur(greyscaleImage, greyscaleImage, Size(filter_size,filter_size), gaussian_std_dev);
```

```
      /* compute the maximum and minimum greyscale values in the calibration image */
      min = 255; max = 0;
      for (row=0; row < greyscaleImage.rows; row++) {
         for (col=0; col < greyscaleImage.cols; col++) {
            if (greyscaleImage.at<uchar>(row,col) < min) {
               min = greyscaleImage.at<uchar>(row,col);
            }
            if (greyscaleImage.at<uchar>(row,col) > max) {
               max = greyscaleImage.at<uchar>(row,col);
            }
         }
      }

      if (debug) {
         printf("display_decalibrated_image_from_camera: min %d, max  %d\n", min, max);
      }

      /* Compute the decalibration image                                                    */
      /*                                                                                    */
      /* Note: the decalibration image is inverted, with large values in regions of low sensitivity  */
      /* and low values in regions of high senesitivity.                                    */
      /* Conseuqently, we ADD it to an input image rather than SUBTRACT it                  */
      /* This allows us to preserve the contrast of the input image,                        */
      /* increasing the lower luminance in the vignetted regions rather than                */
      /* reducing the luminance in the central region that exhibits higher luminance values */

      for (row=0; row < greyscaleImage.rows; row++) {
         for (col=0; col < greyscaleImage.cols; col++) {
            greyscaleImage.at<uchar>(row,col) =  max - greyscaleImage.at<uchar>(row,col);
         }
      }

      imshow(windowName, greyscaleImage);
      cvWaitKey(30); // this is essential as it allows openCV to handle the display event ...

      compressionParams.push_back(CV_IMWRITE_PNG_COMPRESSION);
      compressionParams.push_back(0);                              // 9 implies maximum compression
      imwrite(filename, greyscaleImage, compressionParams);        // write the image to a file
   } // inputImage.type() == CV_8UC3
}
else {

   /* read decalibration image from file */

   greyscaleImage = imread(filename, CV_LOAD_IMAGE_GRAYSCALE);   // Read the file

   if (!greyscaleImage.data) {                              // Check for invalid input
      printf("display_decalibrated_image_from_camera - Error: failed to read decalibration image\n");
      prompt_and_exit(-1);
   }
}
```

```
      /* acquire, decalibrate, and display images from the camera */
      /* -------------------------------------------------------- */

      printf("Press any key to stop image display\n");

      do {
         camera >> inputImage;                        // read a frame from the camera

         if (inputImage.data) {                       // Check for invalid input

            /* separate the colour channels */
            cvtColor(inputImage, hlsImage, CV_BGR2HLS);
            split(hlsImage,input_planes);

            /* decalibration is based on the luminance component */
            for (row=0; row < greyscaleImage.rows; row++) {
               for (col=0; col < greyscaleImage.cols; col++) {
                  value = input_planes[1].at<uchar>(row,col) + greyscaleImage.at<uchar>(row,col);

                  if (value > 255)
                     value = 255; // we clip the value at the maximum

                  input_planes[1].at<uchar>(row,col) = value;
               }
            }

            /* regenerate the colour image from the H, equalized L, and S components */
            merge(input_planes, hlsImage);
            cvtColor(hlsImage, decalibratedImage, CV_HLS2BGR);

            imshow(windowName, decalibratedImage);
            cvWaitKey(30); // this is essential as it allows openCV to handle the display event ...
                           // the argument is the number of milliseconds to wait
         }
      } while (!_kbhit());

      getchar(); // flush the buffer from the keyboard hit
      destroyWindow(windowName);
   }

   else {
      printf("Failed to open camera %d\n",cameraNum);
      prompt_and_continue();
   }
}
```

Demos

The following code is taken from the histogram project in the lectures directory of the ACV repository

See:

```
histogram.h
histogramImplementation.cpp
histogramApplication.cpp
```

```
void generateHistogram(char *filename) {

   char inputWindowName[MAX_STRING_LENGTH]            = "Input Image";
   char histogramWindowName[MAX_STRING_LENGTH]        = "Histogram";

   Mat inputImage;
   Mat histogramImage;

   const int* channel_numbers = { 0 };
   float channel_range[] = { 0.0, 255.0 };
   const float* channel_ranges = channel_range;
   int number_bins = 64;

    ...


   if (inputImage.type() == CV_8UC1) {     // greyscale image
      MatND grey_histogram;

      calcHist( &inputImage, 1, channel_numbers, Mat(), grey_histogram, 1, &number_bins, &channel_ranges);
      OneDHistogram::Draw1DHistogram( &grey_histogram, 1, histogramImage );
   }
   else if (inputImage.type() == CV_8UC3) { // colour image
      MatND* colour_histogram = new MatND[ inputImage.channels() ];

      vector<Mat> colour_channels( inputImage.channels() );
      split(inputImage, colour_channels );

      for (int chan=0; chan < inputImage.channels(); chan++) {
         calcHist( &(colour_channels[chan]), 1, channel_numbers, Mat(), colour_histogram[chan], 1, &number_bins, &channel_ranges);
      }

      OneDHistogram::Draw1DHistogram( colour_histogram, inputImage.channels(), histogramImage );
   }

   imshow(inputWindowName, inputImage );
   imshow(histogramWindowName, histogramImage);

   /* This can also be done using the OneDHistogram class provided above: */

   // Mat histogramImage2;
   // OneDHistogram histogram(inputImage,64);
   // histogram.Draw(histogramImage2);
   // imshow("Histogram 2", histogramImage2);
```

Demos

The following code is taken from the histogramEqualization project in the lectures directory of the ACV repository

See:

```
histogramEqualization.h
histogramEqualizationImplementation.cpp
histogramEqualizationApplication.cpp
```

```
void histogramEqualization(char *filename) {

   char inputWindowName[MAX_STRING_LENGTH]              = "Input Image";
   char outputWindowName[MAX_STRING_LENGTH]             = "Equalized Image";
   char histogramWindowName[MAX_STRING_LENGTH]          = "Histogram";
   char equalizedHistogramWindowName[MAX_STRING_LENGTH] = "Equalized Histogram";

   Mat inputImage;
   Mat histogramImage;
   Mat equalizedHistogramImage;
   Mat processedImage;
   Mat hlsImage;
   std::vector<cv::Mat> input_planes(3);

   const int* channel_numbers = { 0 };
   float channel_range[] = { 0.0, 255.0 };
   const float* channel_ranges = channel_range;
   int numberOfBins = 64;
```

```
if (inputImage.type() == CV_8UC1) {     // greyscale image

   /* generate histogram */
   OneDHistogram histogram(inputImage,numberOfBins);
   histogram.Draw(histogramImage);

   /* equalize the luminance component */
   equalizeHist(inputImage,processedImage);

   /* generate histogram of equalized image */
   OneDHistogram equalizedHistogram(processedImage,numberOfBins);
   equalizedHistogram.Draw(equalizedHistogramImage);
}
else if (inputImage.type() == CV_8UC3) { // colour image

   /* separate the colour channels */
   cvtColor(inputImage, hlsImage, CV_BGR2HLS);
   split(hlsImage,input_planes);

   /* generate histogram of luminance component */
   OneDHistogram input_luminance_histogram(input_planes[1],numberOfBins);
   input_luminance_histogram.Draw(histogramImage);

   /* equalize the luminance component */
   equalizeHist(input_planes[1],input_planes[1]);

   /* generate histogram of equalized luminance component */
   OneDHistogram output_luminance_histogram(input_planes[1],numberOfBins);
   output_luminance_histogram.Draw(equalizedHistogramImage);

   /* regenerate the colour image from the H, equalized L, and S components */
   merge(input_planes, hlsImage);
   cvtColor(hlsImage, processedImage, CV_HLS2BGR);
}
```

Demos

The following code is taken from the binaryThresholding project in the lectures directory of the ACV repository

See:

```
binaryThresholding.h
binaryThresholdingImplementation.cpp
binaryThresholdingApplication.cpp
```

```
void binaryThresholding(int, void*) {

   extern Mat inputImage;
   extern int thresholdValue;
   extern char* thresholded_window_name;
   Mat greyscaleImage;
   Mat thresholdedImage;
   int row, col;

   if (thresholdValue < 1)  // the trackbar has a lower value of 0 which is invalid
      thresholdValue = 1;

   if (inputImage.type() == CV_8UC3) { // colour image
      cvtColor(inputImage, greyscaleImage, CV_BGR2GRAY);
   }
   else {
      greyscaleImage = inputImage.clone();
   }

   thresholdedImage.create(greyscaleImage.size(), CV_8UC1);

   for (row=0; row < greyscaleImage.rows; row++) {
      for (col=0; col < greyscaleImage.cols; col++) {
         if(greyscaleImage.at<uchar>(row,col) < thresholdValue) {
            thresholdedImage.at<uchar>(row,col) = (uchar) 0;
         }
         else {
            thresholdedImage.at<uchar>(row,col) = (uchar) 255;
         }
      }
   }

   /* alternatively, use OpenCV */

   // threshold(greyscaleImage,thresholdedImage,thresholdValue, 255,THRESH_BINARY);
   // threshold(greyscaleImage,thresholdedImage,thresholdValue, 255,THRESH_BINARY  | THRESH_OTSU); // automatic threshold selection

   imshow(thresholded_window_name, thresholdedImage);
}
```

Demos

The following code is taken from the localAveraging project in the lectures directory of the ACV repository

See:

```
localAveraging.h
localAveragingImplementation.cpp
localAveragingApplication.cpp
```

```
/*
 * function processNoiseAndAveraging
 * Trackbar callback - add Gaussian noise with standard deviation input from user
 * Trackbar callback - remove noise with local averaging using filter size input from user
*/

void processNoiseAndAveraging(int, void*) {

   extern Mat src;
   extern int noise_std_dev;
   extern int half_kernel_size;
   extern char* processed_window_name;

   Mat noisy_image;
   Mat filtered_image;

   int filter_size;

   filter_size = half_kernel_size * 2 + 1;
   noisy_image = src.clone();
   addGaussianNoise(noisy_image, 0.0, (double)noise_std_dev);

   blur(noisy_image,filtered_image,Size(filter_size,filter_size));
   //GaussianBlur(noisy_image,filtered_image,Size(filter_size,filter_size),gaussian_std_dev);
   //medianBlur(noisy_image,filtered_image,filter_size);

   imshow(processed_window_name, filtered_image);
 }
```

Demos

The following code is taken from the gaussianFiltering project in the lectures directory of the ACV repository

See:

```
gaussianFiltering.h
gaussianFilteringImplementation.cpp
gaussianFilteringApplication.cpp
```

```
/*
 * function processNoiseAndAveraging
 * Trackbar callback - add Gaussian noise with standard deviation input from user
 * Trackbar callback - remove noise with local averaging using filter size input from user
*/

void processNoiseAndAveraging(int, void*) {

   extern Mat src;
   extern int noise_std_dev;
   extern int gaussian_std_dev;
   extern char* processed_window_name;

   Mat noisy_image;
   Mat filtered_image;

   int filter_size;

   filter_size = gaussian_std_dev * 4 + 1;
   noisy_image = src.clone();

   addGaussianNoise(noisy_image, 0.0, (double)noise_std_dev);

   GaussianBlur(noisy_image,filtered_image,Size(filter_size,filter_size),gaussian_std_dev);

   imshow(processed_window_name, filtered_image);
 }
```

Demos

The following code is taken from the medianFiltering project in the lectures directory of the ACV repository

See:

```
medianFiltering.h
medianFilteringImplementation.cpp
medianFilteringApplication.cpp
```

```
/*
 * function processNoiseAndAveraging
 * Trackbar callback - add Gaussian noise with standard deviation input from user
 * Trackbar callback - remove noise with median filtering using filter size input from user
*/

void processNoiseAndAveraging(int, void*) {

   extern Mat src;
   extern int noise_std_dev;
   extern int half_kernel_size;
   extern char* processed_window_name;

   Mat noisy_image;
   Mat filtered_image;

   int filter_size;

   filter_size = half_kernel_size * 2 + 1;
   noisy_image = src.clone();
   addGaussianNoise(noisy_image, 0.0, (double)noise_std_dev);

   medianBlur(noisy_image,filtered_image,filter_size);

   imshow(processed_window_name, filtered_image);
 }
```

Demos

The following code is taken from the fourierTransform project in the lectures directory of the ACV repository

See:

```
fourierTransform.h
fourierTransformImplementation.cpp
fourierTransformApplication.cpp
```

```
/*
  Example use of openCV to perform the Fourier transform
  ------------------------------------------------------
  Implementation file

  Adapted from the example at
  http://docs.opencv.org/2.4/doc/tutorials/core/discrete_fourier_transform/discrete_fourier_transform.html

  David Vernon
  9 May 2017

*/

#include "fourierTransform.h"

void fourierTransform(char *filename) {

   char inputWindowName[MAX_STRING_LENGTH]           = "Input Image";
   char fourierWindowName[MAX_STRING_LENGTH]         = "Fourier Magnitude Image";
   char inverseFourierWindowName[MAX_STRING_LENGTH]  = "Inverse Fourier Image";
   int m;
   int n;

   Mat image;
   Mat I;
   Mat padded;                            //expand input image to optimal size

   namedWindow(inputWindowName,           CV_WINDOW_AUTOSIZE);
   namedWindow(fourierWindowName,         CV_WINDOW_AUTOSIZE);
```

```
I = imread(filename, CV_LOAD_IMAGE_GRAYSCALE); // Read the file

if (I.empty()) {                            // Check for invalid input
   printf("Error: failed to read image\n");
   prompt_and_exit(-1);
}

printf("Press any key to continue ...\n");

m = getOptimalDFTSize( I.rows );
n = getOptimalDFTSize( I.cols ); // on the border add zero values

//cout << I.rows << " " << I.cols << endl;

//copyMakeBorder(I, padded, 0, m - I.rows, 0, n - I.cols, BORDER_CONSTANT, Scalar::all(0));
copyMakeBorder(I, padded, 0, m - I.rows, 0, n - I.cols, BORDER_REFLECT, Scalar::all(0));

//cout << I.rows << " " << I.cols << endl;

Mat planes[] = {Mat_<float>(padded), Mat::zeros(padded.size(), CV_32F)};
Mat complexI;

merge(planes, 2, complexI);         // Add to the expanded another plane with zeros

dft(complexI, complexI);            // this way the result may fit in the source matrix

// compute the magnitude and switch to logarithmic scale
// => log(1 + sqrt(Re(DFT(I))^2 + Im(DFT(I))^2))
split(complexI, planes);                   // planes[0] = Re(DFT(I), planes[1] = Im(DFT(I))
magnitude(planes[0], planes[1], planes[0]);// planes[0] = magnitude
Mat magI = planes[0];
magI += Scalar::all(1);                    // switch to logarithmic scale
//log(magI, magI);
```

```
// crop the spectrum, if it has an odd number of rows or columns
magI = magI(Rect(0, 0, magI.cols & -2, magI.rows & -2));

// rearrange the quadrants of Fourier image  so that the origin is at the image center
int cx = magI.cols/2;
int cy = magI.rows/2;

Mat q0(magI, Rect(0, 0, cx, cy));   // Top-Left - Create a ROI per quadrant
Mat q1(magI, Rect(cx, 0, cx, cy));  // Top-Right
Mat q2(magI, Rect(0, cy, cx, cy));  // Bottom-Left
Mat q3(magI, Rect(cx, cy, cx, cy)); // Bottom-Right

Mat tmp;                            // swap quadrants (Top-Left with Bottom-Right)
q0.copyTo(tmp);
q3.copyTo(q0);
tmp.copyTo(q3);

q1.copyTo(tmp);                     // swap quadrant (Top-Right with Bottom-Left)
q2.copyTo(q1);
tmp.copyTo(q2);

//normalize(magI, magI, 0, 1, CV_MINMAX); // user this for the log version
normalize(magI, magI, 0, 255, CV_MINMAX);
```

```
    //normalize(magI, magI, 0, 1, CV_MINMAX); // user this for the log version
    normalize(magI, magI, 0, 255, CV_MINMAX);

    imshow(inputWindowName  , I   );    // Show the result
    imshow(fourierWindowName, magI);

    do{
       waitKey(30);                                 // Must call this to allow openCV to display the images
    } while (!_kbhit());                            // We call it repeatedly to allow the user to move the windows
                                                    // (if we don't the window process hangs when you try to click and drag

    getchar(); // flush the buffer from the keyboard hit

    destroyWindow(inputWindowName);
    destroyWindow(fourierWindowName);
}
```

Exercises

1. Read OpenCV documentation for all OpenCV functions in sample code
2. Study utility functions in sample code

OpenCV

version 2.4

For documentation on OpenCV data structures, functions, and methods, see

http://docs.opencv.org/2.4/index.html

Reading

R. Szeliski, *Computer Vision: Algorithms and Applications*, Springer, 2010.

Section 3.1 Point operations

- Section 3.1.1 Pixel transform
- Section 3.1.4 Histogram equalization

Section 3.2 Linear Filtering

Section 3.3 More neighborhood operations

- Section 3.3.1. Non-linear filtering

Section 3.4 Fourier transforms

- Section 3.4.1 Fourier transform pairs
- Section 3.4.2 Two-dimensional Fourier transforms