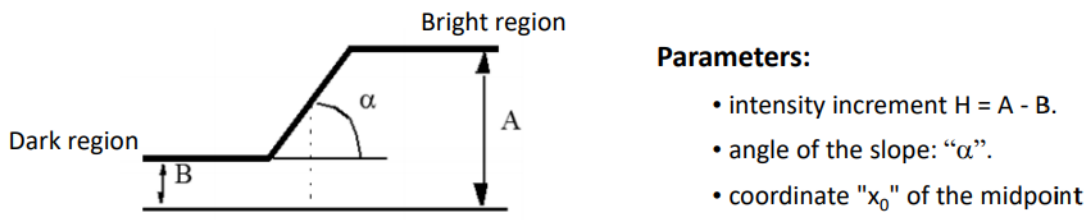
3 Edge detection

At its core, an **edge** in an image represents a boundary or significant change in intensity between adjacent pixels. This change could be in terms of color, brightness, or texture. Essentially, edges help define the shape and structure of objects within an image, making them one of the most fundamental features for understanding and interpreting visual data. The process in which the edges in an image are identified is called **edge detection**. By detecting objects we can, for example:

- **Identify objects:** Edges often correspond to the outlines of objects. For example, detecting the edge of a person against a background is crucial for tasks like object detection or segmentation.
- **Simplify images:** By focusing on the edges, we reduce an image to its most important structures. This simplification is useful in tasks like image compression or recognition.
- **Analyze shapes:** Many shape-based analyses depend on extracting edges to define boundaries. In gesture recognition, for instance, the edges of hands can be used to identify specific movements.

As commented, edges can be defined as transitions between image regions that have different gray levels (intensities). In this way, the unidimensional, continuous model of an ideal edge is:


Bright region
Dark region
α
A
B
Parameters:
• intensity increment H = A - B.
• angle of the slope: "α".
• coordinate "x_0" of the midpoint

That is, and edge is defined by three parameters:

- **Intensity increment** ($H = A - B$): The difference in intensity between the bright and dark regions.
- **Slope angle** (α): The angle of the transition, which represents how quickly the intensity changes from the dark region to the bright region. A steeper slope indicates a sharper edge, while a gradual slope suggests a softer transition (e.g., shadows or blurred boundaries).
- **Midpoint** (x_0): The location of the center of the edge, where the intensity transition is most prominent.

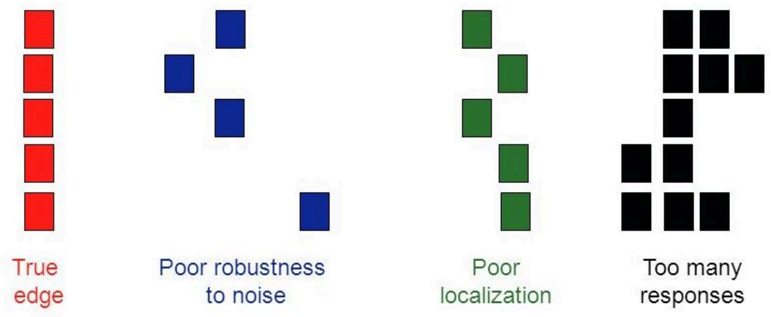
However, the idealized concept of an edge represented as this a continuous model doesn't fully capture the complexity found in discrete digital images, which are subject to noise and resolution limitations.

Error types related to edge detection

Finding edges properly is not a straightforward task, as there exist different errors that can appear when applying edge detection techniques:

- **Detection error.** A good detector exhibits a low ratio of false negative and false positive, that is:
    - False negatives: Existing edges that are not detected.
    - False positives: Detected objects that are not real.
- **Localization error.** Edges are detected, but they are not at the real, exact position.
- **Multiple response.** Multiple detections are raised for the same edge (the edge is thick).

The following figure illustrates such errors.

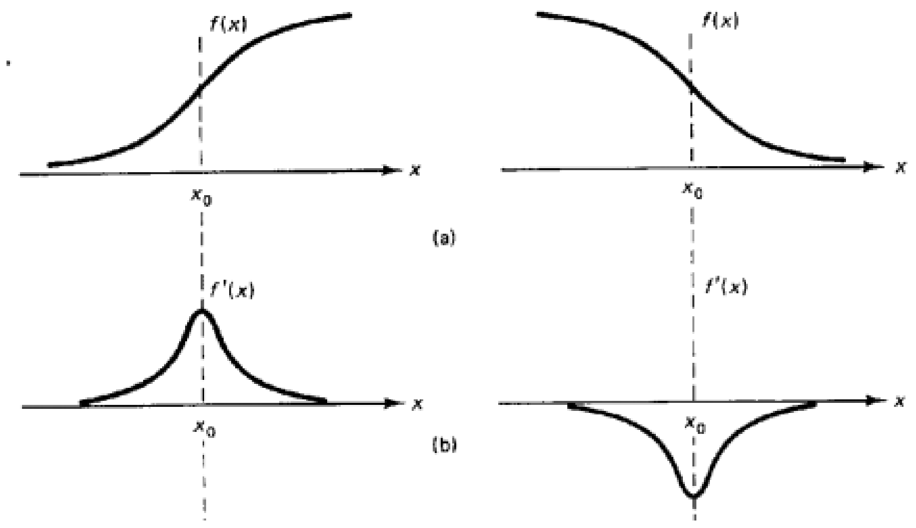


Thereby, when designing a good edge detector, the goal is to achieve low detection and localization errors, as well as to avoid multiple responses.

3.1 Operators based on first derivative (gradient)

In the upcoming chapters, we are going to investigate and implement different edge detection methods. All of them are based on our dear convolution operation, having their own pros and cons.

Concretely, in this notebook we will cover **first-derivative** based operators, which try to detect borders by looking at abrupt intensity differences in neighbor pixels. In the image below we can see two functions $f(x)$ (first row) and how their derivatives (second row) reach their maximum values at the points where the functions' values change more abruptly (around x_o).



If we are dealing with a **two-dimensional** continuous function $f(x,y)$, its derivative is a *vector* (**gradient**) defined as:

$$\nabla f(x,y) = \begin{bmatrix} \frac{\partial}{\partial x} f(x,y) \\ \frac{\partial}{\partial y} f(x,y) \end{bmatrix} = \begin{bmatrix} f_x(x,y) \\ f_y(x,y) \end{bmatrix}$$

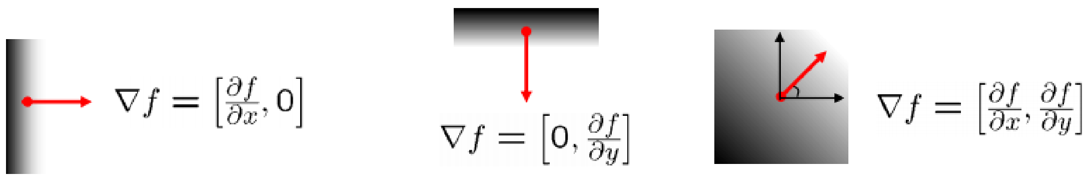
which points at the *direction* of maximum (positive) variation of $f(x,y)$:

$$\alpha(x,y) = \arctan\left(\frac{f_y(x,y)}{f_x(x,y)}\right)$$

and has a *module* proportional to the strength of this variation:

$$|\nabla f(x,y)| = \sqrt{(f_x(x,y))^2 + (f_y(x,y))^2} \approx |f_x(x,y)| + |f_y(x,y)|$$

The image below shows examples of gradient vectors:


$\nabla f = \left[\frac{\partial f}{\partial x}, 0\right]$
$\nabla f = \left[0, \frac{\partial f}{\partial y}\right]$
$\nabla f = \left[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right]$

Concretely, the techniques based on the first derivative explored here are:

- Discrete approximations of a **gradient operator** (Sobel, Prewitt, Roberts, etc., Section 3.1.1).
- The **Derivative of Gaussian** (DroG) operator (Section 3.1.2).

Problem context - Edge detection for medical images

Edge detection in medical images is of capital importance for the diagnosis of different diseases (e.g., the detection of tumor cells) in human organs such as lungs and prostates, becoming an essential pre-processing step in medical image segmentation.

In this context, *Hospital Clínico*, a very busy hospital in Málaga, is asking local engineering students to join their research team. They are looking for a person with knowledge in image processing and, in order to ensure it, they have published 3 medical images: `medical_1.jpg` , `medical_2.jpg` and `medical_3.jpg` . They have asked us to perform accurate edge detection in the three images, as well as to provide an explanation of how it has been made.

In [1]:
```
import numpy as np
from scipy import signal
import cv2
import matplotlib.pyplot as plt
import matplotlib
from ipywidgets import interactive, fixed, widgets
matplotlib.rcParams['figure.figsize'] = (15.0, 15.0)

images_path = './images/'
```

To face this challenge, we are going to use plenty edge detection methods, which will be tested and compared in order to determine the best option.

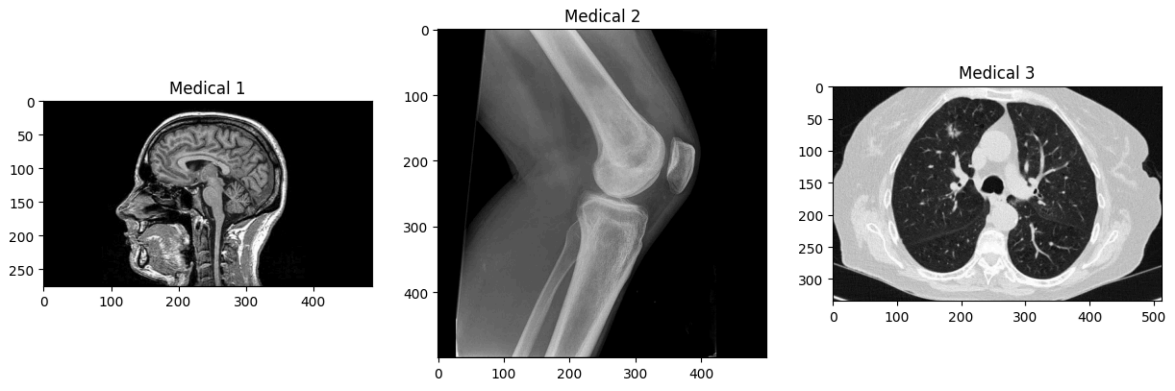
ASSIGNMENT 1: Taking a look at images

First, **display the provided images** to get an idea about what we are dealing with.

Note: As most medical images does not provide color information, we are going to use border detection in grayscale images.

Tip: Different approaches can be followed for edge detection in color images, like converting to YCrCb color space (appendix 2), or detecting edges on each RGB channel.

In [2]:
```python
# ASSIGNMENT 1
# Display the provided images in a 1x3 plot to see what are we dealing with
# Write your code here!

# Read the images
medical_1 = cv2.imread(images_path + 'medical_1.jpg', cv2.IMREAD_GRAYSCALE)
medical_2 = cv2.imread(images_path + 'medical_2.jpg', cv2.IMREAD_GRAYSCALE)
medical_3 = cv2.imread(images_path + 'medical_3.jpg', cv2.IMREAD_GRAYSCALE)

# And show them
plt.subplot(131)
plt.imshow(medical_1, cmap='gray')
plt.title('Medical 1')

plt.subplot(132)
plt.imshow(medical_2, cmap='gray')
plt.title('Medical 2')

plt.subplot(133)
plt.imshow(medical_3, cmap='gray')
plt.title('Medical 3')
plt.show()
```



3.1.1 Discrete approximations of a gradient operator

The first bunch of methods that we are going to explore carry out a **discrete approximation of a gradient operator** based on the differences between gray (intensity) levels. For example, in order to obtain the derivative in the rows' direction, we could apply:

- Backward difference of pixels along a row:
$$f_x(x,y) \approx G_R(i,j) = [F(i,j) - F(i-1,j)]/T$$

0	0	0
0	1	-1
0	0	0

- Symmetric difference of pixels along a row:
$$f_x(x,y) \approx G_R(i,j) = [F(i+1,j) - F(i-1,j)]/2T$$

0	0	0

1	0	-1
0	0	0

These approximations are typically implemented through the convolution of the image with a pair of templates H_R (for rows, computing horizontal derivatives for detecting vertical edges) and H_C (for columns, computing vertical derivatives for detecting horizontal ones), that is:

$$G_R(i,j) = F(i,j) \otimes H_R(i,j)$$
$$G_C(i,j) = F(i,j) \otimes H_C(i,j)$$

Perhaps the most popular operator doing this is such of **Sobel**, although there are many of them that provide acceptable results. These operators use the aforementioned two kernels (typically of size 3×3 or 5×5) which are convolved with the original image to calculate approximations of the derivatives.

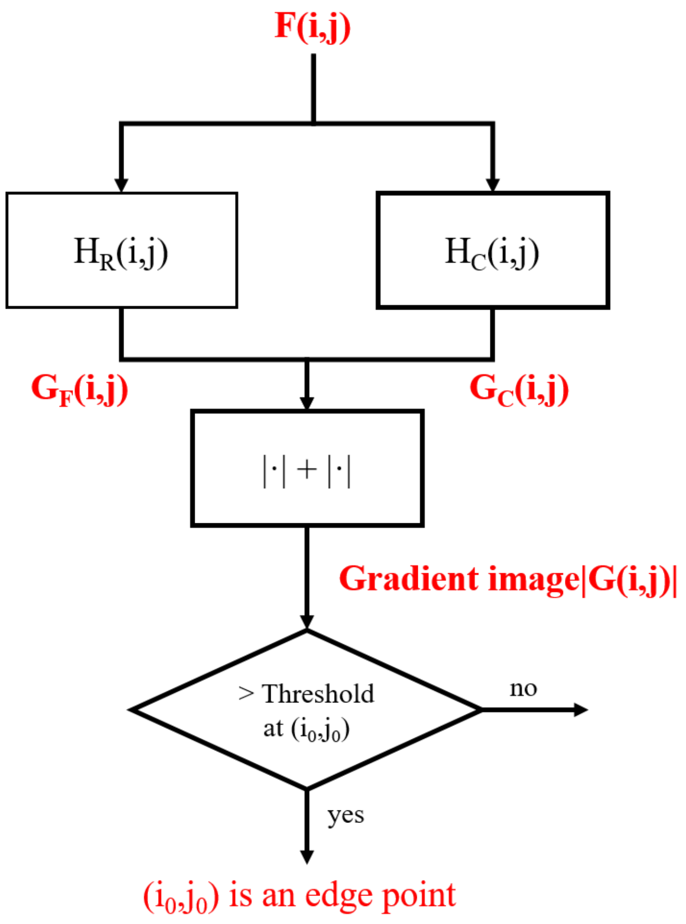
These are some examples (first column: operator name; second one: H_R; third column: H_C):

Roberts	$\begin{array}{\|c\|c\|c\|}\hline 0 & 0 & 0 \\ \hline 0 & 0 & 1 \\ \hline 0 & -1 & 0 \\ \hline \end{array}$	$\begin{array}{\|c\|c\|c\|}\hline -1 & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array}$
Prewitt	$\frac{1}{3}\begin{array}{\|c\|c\|c\|}\hline 1 & 0 & -1 \\ \hline 1 & 0 & -1 \\ \hline 1 & 0 & -1 \\ \hline \end{array}$	$\frac{1}{3}\begin{array}{\|c\|c\|c\|}\hline -1 & -1 & -1 \\ \hline 0 & 0 & 0 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$
Sobel	$\frac{1}{4}\begin{array}{\|c\|c\|c\|}\hline 1 & 0 & -1 \\ \hline 2 & 0 & -2 \\ \hline 1 & 0 & -1 \\ \hline \end{array}$	$\frac{1}{4}\begin{array}{\|c\|c\|c\|}\hline -1 & -2 & -1 \\ \hline 0 & 0 & 0 \\ \hline 1 & 2 & 1 \\ \hline \end{array}$
Frei-Chen	$\frac{1}{2+\sqrt{2}}\begin{array}{\|c\|c\|c\|}\hline 1 & 0 & -1 \\ \hline \sqrt{2} & 0 & -\sqrt{2} \\ \hline 1 & 0 & -1 \\ \hline \end{array}$	$\frac{1}{2+\sqrt{2}}\begin{array}{\|c\|c\|c\|}\hline -1 & -\sqrt{2} & -1 \\ \hline 0 & 0 & 0 \\ \hline 1 & \sqrt{2} & 1 \\ \hline \end{array}$
In general	$\frac{1}{2+K}\begin{array}{\|c\|c\|c\|}\hline 1 & 0 & -1 \\ \hline K & 0 & -K \\ \hline 1 & 0 & -1 \\ \hline \end{array}$	$\frac{1}{2+K}\begin{array}{\|c\|c\|c\|}\hline -1 & -K & -1 \\ \hline 0 & 0 & 0 \\ \hline 1 & K & 1 \\ \hline \end{array}$

At this point we know how to perform a discrete approximation of a gradient operator through the application of a convolution operation with two different kernels, that is:

$$\nabla F(x,y) = \begin{bmatrix} F \otimes H_C \\ F \otimes H_R \end{bmatrix}$$

But, how could we use the output of those computations to detect edges? The following figure clarifies that!



Kernel sizes

As discussed, kernels can be of different size, and that size directly affects the quality of the detection and the localization (e.g. Sobel 3×3 or 5×5):

- Small template:
  - more precise localization (good localization).
  - more affected by noise (likely produces false positives).
- Large template:
  - less precise localization.
  - more robust to noise (good detector).
  - higher computational cost ($O(N \times N)$).

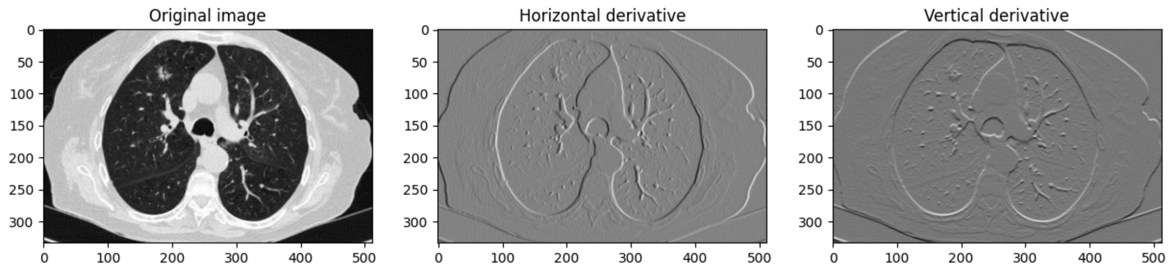
ASSIGNMENT 2: Playing with Sobel derivatives

Now that we have acquired a basic understanding of these methods, let's complete the following code cell to employ the Sobel kernels (S_x, S_y) to compute both derivatives and display them along with the original image (`medical_3.jpg`).

*Notice that the derivative image values can be positive **and negative**, caused by the negative values in the kernel. This implies that the desired depth of the destination image (* `ddepth` *) has to be at least a signed data type when calling to the* `filter2D()` *method.*

In [3]:

```
# ASSIGNMENT 2
# Read one of the images, compute both kernel derivatives, apply them to the ima
# Write your code here!

# Read the image
image = cv2.imread(images_path + 'medical_3.jpg', 0)

# Define horizontal and vertical kernels
kernel_h = np.array([[1,0,-1],[2,0,-2],[1,0,-1]])* (1/4)
kernel_v = np.array([[-1,-2,-1],[0,0,0],[1,2,1]])* (1/4)

# Apply convolution
d_horizontal = cv2.filter2D(image,cv2.CV_16S,kernel_h) # Using ddepth=cv2.CV_16S
d_vertical = cv2.filter2D(image,cv2.CV_16S,kernel_v)

# And show them!
plt.subplot(131)
plt.imshow(image, cmap='gray')
plt.title('Original image')

plt.subplot(132)
plt.imshow(d_horizontal, cmap='gray')
plt.title('Horizontal derivative')

plt.subplot(133)
plt.imshow(d_vertical, cmap='gray')
plt.title('Vertical derivative');
```



Once we have computed both derivative images G_C and G_R, we can determine the *complete* edge image by computing the image gradient magnitude and then binarizing the result. Recall that the image codifying the gradient magnitude can be computed and approximated as:

$$|\nabla F(x,y)| = \sqrt{(F \otimes G_C)^2 + (F \otimes G_R)^2} \approx |F \otimes G_C| + |F \otimes G_R|$$

ASSIGNMENT 3a: Time to detect edges

Complete `edge_detection_chart()` that computes the gradient image of an input one using `kernel_h` and `kernel_v` (kernels for horizontal and vertical derivatives respectively) and **binarize the resultant image** (final edges image) using `threshold` .

Then display in a 1x3 plot `image`, the gradient image, and finally, an image with the detected edges! (Only if `verbose` is True).

Tip: you should normalize gradient image before thresholding.

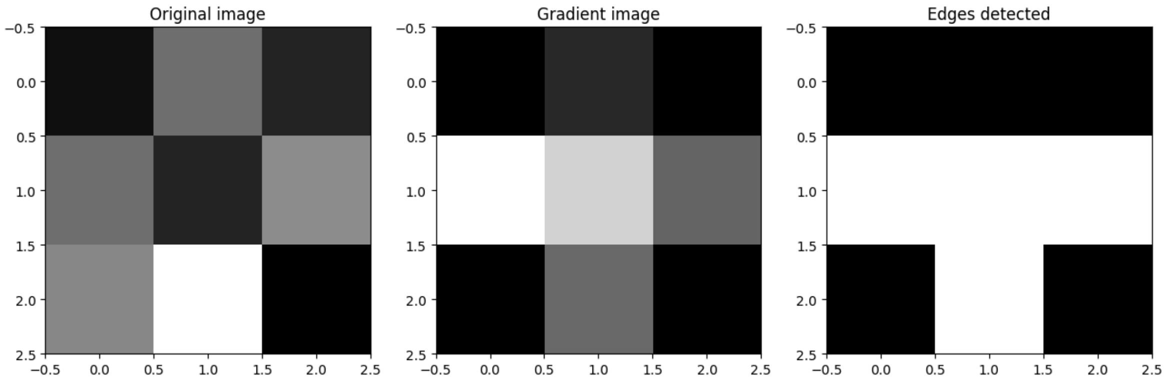
Interesting functions: `np.absolute()`, `np.add()`, `cv2.threshold()`

In [4]:
```
# ASSIGNMENT 3a
# Implement a function that that computes the gradient of an image, taking also
# It must also binarize the resulting image using a threshold
# Show the input image, the gradient image (normalized) and the binarized edge i
def edge_detection_chart(image, kernel_h, kernel_v, threshold, verbose=False):
    """ Computed the gradient of the image, binarizes and display it.

        Args:
            image: Input image
            kernel_h: kernel for horizontal derivative
            kernel_v: kernel for vertical derivative
            threshold: threshold value for binarization
            verbose: Only show images if this is True

        Returns:
            edges: edges binary image
    """
    # Write your code here!

    # Compute derivatives
    d_h = cv2.filter2D(image,cv2.CV_16S,kernel_h) # horizontal
    d_v = cv2.filter2D(image,cv2.CV_16S,kernel_v) # vertical

    # Compute gradient
    gradient_image = np.add(np.absolute(d_h), np.absolute(d_v)) # Hint: You have

    #Normalize gradient
    norm_gradient = np.copy(image)
    norm_gradient = cv2.normalize(gradient_image, norm_gradient, 0, 255, cv2.NOR

    # Threshold to get edges
    ret, edges = cv2.threshold(norm_gradient, threshold, 255, cv2.THRESH_BINARY)

    if verbose:
        # Show the initial image
        plt.subplot(131)
        plt.imshow(image, cmap='gray')
        plt.title('Original image')

        # Show the gradient image
        plt.subplot(132)
        plt.imshow(gradient_image, cmap='gray')
        plt.title('Gradient image')

        # Show edges image
        plt.subplot(133)
        plt.imshow(edges, cmap='gray')
        plt.title('Edges detected')

    return edges
```

You can use next code to **test if your results are correct**:

```
In [5]: image = np.array([[10,60,20],[60,22,74],[72,132,2]], dtype=np.uint8)

        # Sobel derivatives
        kernel_h = np.array([[1,0,-1],[2,0,-2],[1,0,-1]])*1/4
        kernel_v = np.array([[-1,-2,-1],[0,0,0],[1,2,1]])*1/4

        print(edge_detection_chart(image, kernel_h, kernel_v, 100,True))
```

```
[[  0   0   0]
 [255 255 255]
 [  0 255   0]]
```



Expected output:

```
[[  0   0   0]
 [255 255 255]
 [  0 255   0]]
```

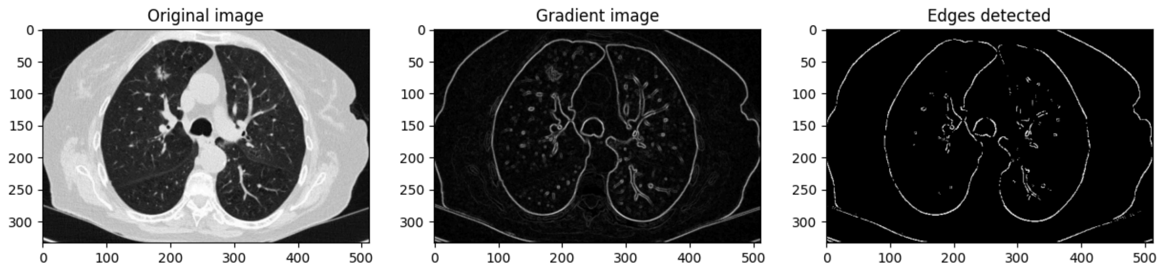
ASSIGNMENT 3b: Testing our detector

Now **try the implemented method** with different size Sobel kernels (3×3, 5×5, ...).

```
In [6]: # ASSIGNMENT 3b
        # Read the image, set you kernels (Sobel, Roberts, Prewitt, etc.) and interact w
        # Write your code here!

        # Read image
        image = cv2.imread(images_path + 'medical_3.jpg', 0)

        # Define kernel (Sobel)
        kernel_h_sobel = np.array([[1,0,-1],[2,0,-2],[1,0,-1]])* (1/4)
        kernel_v_sobel = np.array([[-1,-2,-1],[0,0,0],[1,2,1]])* (1/4)

        kernel_h_sobel_5x5 = np.array([[2,  1,  0, -1, -2],
                                       [3,  2,  0, -2, -3],
                                       [4,  3,  0, -3, -4],
                                       [3,  2,  0, -2, -3],
                                       [2,  1,  0, -1, -2]]) * (1/48)

        kernel_v_sobel_5x5 = np.array([[ 2,  3,  4,  3,  2],
                                       [ 1,  2,  3,  2,  1],
                                       [ 0,  0,  0,  0,  0],
                                       [-1, -2, -3, -2, -1],
                                       [-2, -3, -4, -3, -2]]) * (1/48)


        # And define the rest!
        kernel_h_roberts = np.array([[0,0,0],[0,0,1],[0,-1,0]])
        kernel_v_roberts = np.array([[-1,0,0],[0,1,0],[0,0,0]])

        kernel_h_prewitt = np.array([[1,0,-1],[1,0,-1],[1,0,-1]]) * (1/3)
        kernel_v_prewitt = np.array([[-1,-1,-1],[0,0,0],[1,1,1]]) * (1/3)

        kernel_h_freichen = np.array([[1,0,-1],[np.sqrt(2),0,-np.sqrt(2)],[1,0,-1]]) * (
        kernel_v_freichen = np.array([[-1,-np.sqrt(2),-1],[0,0,0],[1,np.sqrt(2),1]]) * (



        #Interact with your code!
        print("SOBEL 3x3")
        sobel_widget = interactive( edge_detection_chart, image=fixed(image), kernel_h=f
        display(sobel_widget)

        print("SOBEL 5x5")
        sobel_widget_5x5 = interactive( edge_detection_chart, image=fixed(image), kernel
        display(sobel_widget_5x5)

        print("ROBERTS")
        roberts_widget = interactive( edge_detection_chart, image=fixed(image), kernel_h
        display(roberts_widget)

        print("PREWITT")
        prewitt_widget = interactive( edge_detection_chart, image=fixed(image), kernel_h
        display(prewitt_widget)

        print("FREI-CHEN")
        freichen_widget = interactive( edge_detection_chart, image=fixed(image), kernel_
        display(freichen_widget)
```

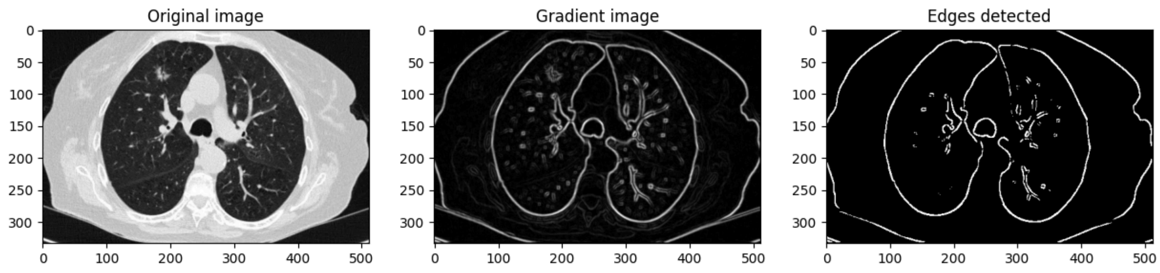
```
SOBEL 3x3
```

threshold 120



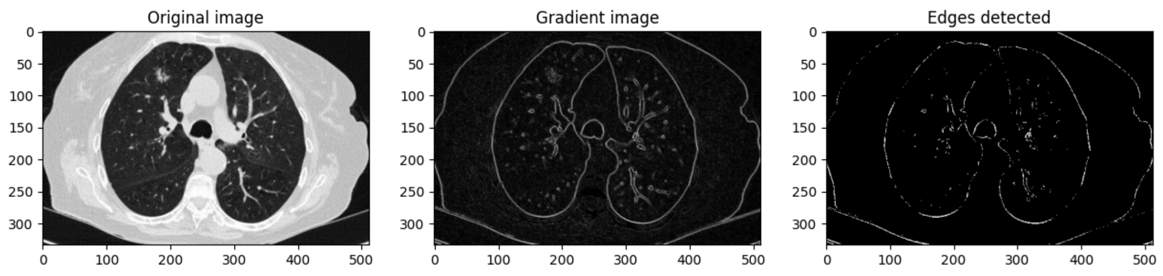
```
SOBEL 5x5
```

threshold 120



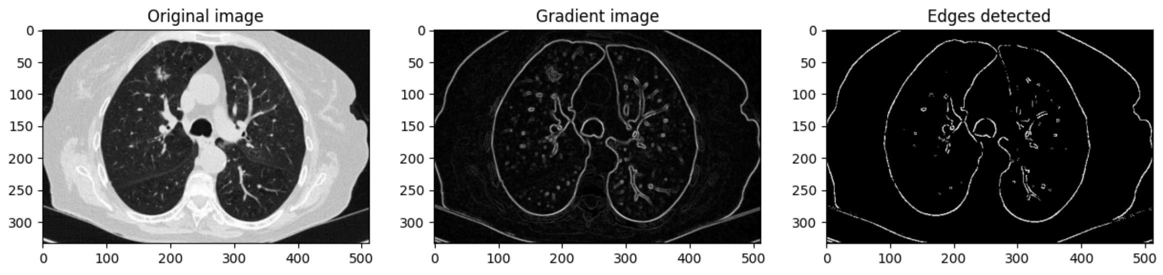
```
ROBERTS
```

threshold 120



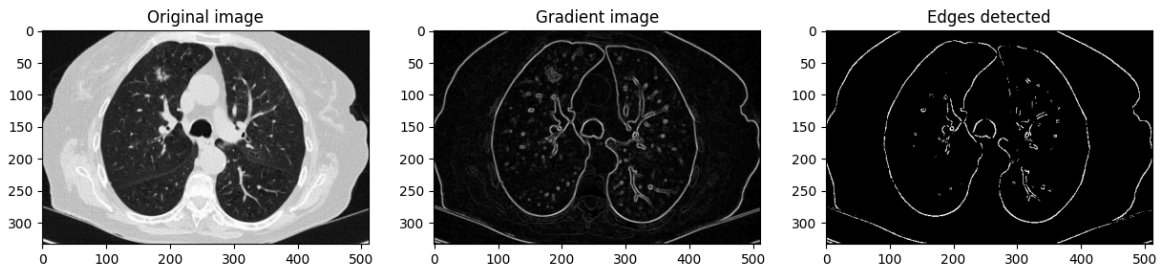
PREWITT

threshold 120



FREI-CHEN

threshold 120



OPTIONAL

Try other edge detection operators based on the first derivative with different kernel sizes (Roberts, Prewitt, etc.).

Thinking about it (1)

Now, **answer following questions**:

- What happens if we use a bigger kernel?

  The bigger the kernel, the less it is affected by noise (better detector) but the worse they are located. As it can be seen above, 5x5 sobel kernel has thickers edges whilst 3x3 sobel kernel has thinner ones (that's what we mean by "localization", the margin of error with which we locate edges). It also increases computational cost.

- There are differences between Sobel and other operators?

  There are, to some extent. Prewitt, Frei-Chen and Sobel are, in this case, at 120 threshold, virtually the same; but Roberts it's far worse than the rest. Edges are weakly detected and there is far more noise at similar threshold levels.

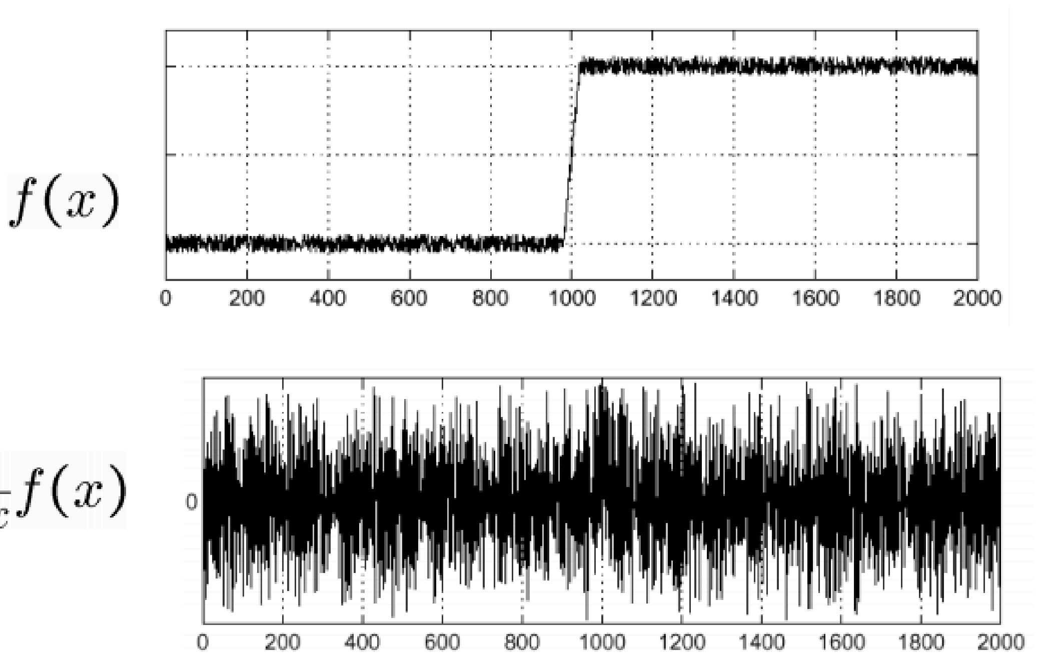
- What errors appear using those operators?

  Their weak point is noise, which leads to poor detection (sometimes). As they are not filtered by a gaussian filter first, noise appears noticeably in every image at some threshold value. Edges are usually thicker, which means that there's multiple responses in some cases.
- Why kernels usually are divided by a number? (e.g. 3×3 Sobel is divided by 4)

  To normalize the values (make them not exceed the [0,255] range), as this helps to scale them to maintain the overall brigthness throughout the image. This normalization also helps to maintein consistency across different kernel sizes, as the value by which they are divided increases with kernel size.

3.1.2 DroG operator

Despite the simplicity of the previous techniques, they have a remarkable drawback: their performance is highly influenced by image noise. Taking a look at the following figure we can see how, having an apparently not so noisy function (first row), where it is easy to visually detect a step (an abrupt change in its values) around 1000, the response of the derivative with that level of noise is as bigger as the step itself!



Source: S. Seitz

But not everything is lost! An already studied image processing technique can be used to mitigate such noise: **image smoothing**, and more concretely, **Gaussian filtering!** The basic idea is to smooth the image and then apply a gradient operator, that is to compute $\frac{\partial}{\partial x}(f \otimes g)$. Not only that, this can be done even more efficiently thanks to the convolution derivative property:

$$\frac{\partial}{\partial x}(f \otimes g) = f \otimes \frac{\partial}{\partial x} g$$

\[5pt]

That is, precomputing the resultant kernels from the convolution of the Gaussisan filtering and the Sobel ones, and then convolving them with the image to be processed. With that we save one operation!

This combination of smoothing and gradient is usually called **Derivative of Gaussian operator (DroG)**. Formally:

$$\nabla[f(x,y) \otimes g_\sigma(x,y)] = f(x,y) \otimes \nabla[g_\sigma(x,y)] = f(x,y) \otimes \text{DroG}(x,y)$$

$$DroG(x,y) = \nabla\left[g_\sigma(x,y)\right] = \underbrace{\begin{bmatrix} \frac{\partial}{\partial x}[g_\sigma(x)g_\sigma(y)] \\ \frac{\partial}{\partial y}[g_\sigma(x)g_\sigma(y)] \end{bmatrix}}_{\text{separability}} = \underbrace{\begin{bmatrix} \frac{-xg_\sigma(x)g_\sigma(y)}{\sigma^2} \\ \frac{-yg_\sigma(x)g_\sigma(y)}{\sigma^2} \end{bmatrix}}_{g(x)'=-xg(x)/\sigma^2} = \begin{bmatrix} \frac{-xg_\sigma(x,y)}{\sigma^2} \\ \frac{-yg_\sigma(x,y)}{\sigma^2} \end{bmatrix}$$

Recall that $g_\sigma(x,y)$ is just the 2D gaussian kernel. We worked with it in Chapter 2!

Also remember from the previous notebooks the expression of the Gaussian distribution with 2 variables centered at the origin of coordinates, where the standard deviation σ controls the degree of smoothness:

Remember from the previous notebooks the expression of the Gaussian distribution with 2 variables centered at the origin of coordinates, where the standard deviation σ controls the degree of smoothness:

$$g_\sigma(x,y) = \frac{1}{2\pi\sigma^2} exp\left(-\frac{x^2+y^2}{2\sigma^2}\right)$$

Take into account that the DroG template or kernel is **created just once**! Then it can applied to as many images as you want.

ASSIGNMENT 4: Applying DroG

We would like to try this robust edge detection technique, so complete the `gaussian_kernel()` method that:

1. constructs a 2D gaussian filter (that is, $g_\sigma(x,y)$ in the previous DroG definition) from a 1D one, and
2. derives it, getting the DroG template (in other words, compute $-xg_\sigma(x,y)/\sigma^2$ and $-yg_\sigma(x,y)/\sigma^2$).
3. Finally, it calls our function `edge_detection_chart()`, but using the DroG template instead of the Sobel one.

Its inputs are:

- an image to be processed,
- the kernel aperture size,

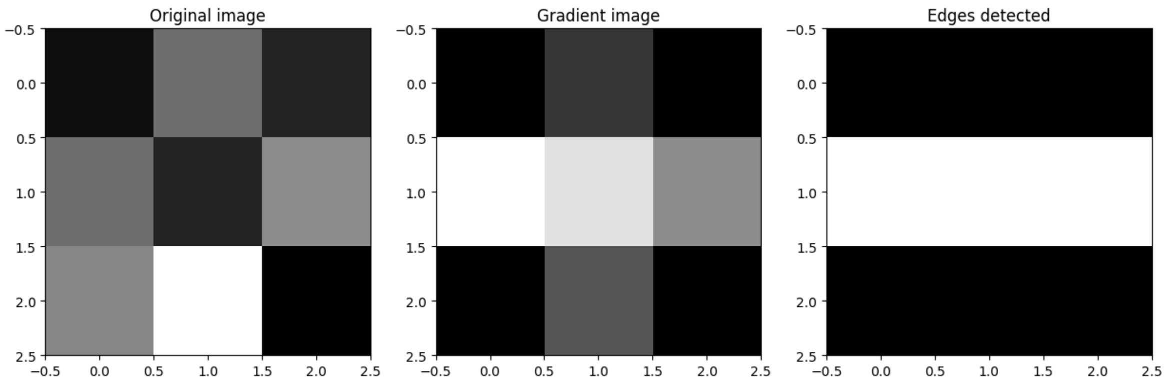
- the standard deviation, and
- the gradient image binaritazion threshold.

In [7]:
```
# ASSIGNMENT 4
# Implement a function that builds the horizontal and vertical DroG templates an
# Inputs: an image, the kernel aperture size, the Gaussian standard deviation an
# It returns the horizontal and vertical kernels
def drog_kernel(image, w_kernel, sigma, threshold, verbose=False):
    """ Construct the DroG operator and call edge_detection_chart.

        Args:
            image: Input image
            w_kernel: Kernel aperture size
            sigma: Standard deviation of the Gaussian distribution
            threshold: Threshold value for binarization
            verbose: Only show images if this is True

        Returns:
            DroG_h, DroG_v: DroG kernerl for computing horizontal and vertical d
    """
    # Write your code here!

    # Create the 1D gaussian filter
    s = sigma
    w = w_kernel
    gaussian_kernel_1D = np.array([(1/(np.sqrt(2*np.pi)*s)) * np.exp(-(z**2)/(2*

    # Get the 2D gaussian filter from the 1D one.
    vertical_kernel = gaussian_kernel_1D.reshape(2*w+1,1)
    horizontal_kernel = gaussian_kernel_1D.reshape(1,2*w+1)
    gaussian_kernel_2D = signal.convolve2d(vertical_kernel, horizontal_kernel)

    # Construct DroG

    # Define x and y axis
    x = np.arange(-w,w+1)
    y = np.vstack(x)

    # Get the kernels for detecting horizontal and vertical edges
    DroG_h = x*(-gaussian_kernel_2D)/s**2 # Horizontal derivative
    DroG_v = y*(-gaussian_kernel_2D)/s**2 # Vertical derivative

    # Call edge detection chart using DroG
    edge_detection_chart(image, DroG_h, DroG_v, threshold, verbose)

    return DroG_h, DroG_v
```

You can use next code to **test if results are correct**:

In [8]:
```
# Create an input image
image = np.array([[10,60,20],[60,22,74],[72,132,2]], dtype=np.uint8)

# Apply the Gaussian kernel
drog_kernel(image, w_kernel=1, sigma=1.2, threshold=100, verbose=True)
```

```
Out[8]: (array([[ 0.03832673, -0.        , -0.03832673],
                [ 0.05423735, -0.        , -0.05423735],
                [ 0.03832673, -0.        , -0.03832673]]),
         array([[ 0.03832673,  0.05423735,  0.03832673],
                [-0.        , -0.        , -0.        ],
                [-0.03832673, -0.05423735, -0.03832673]]))
```



Expected output:

```
(array([[ 0.03832673, -0.        , -0.03832673],
        [ 0.05423735, -0.        , -0.05423735],
        [ 0.03832673, -0.        , -0.03832673]]),
 array([[ 0.03832673,  0.05423735,  0.03832673],
        [-0.        , -0.        , -0.        ],
        [-0.03832673, -0.05423735, -0.03832673]]))
```

Thinking about it (2)

Now **try this method** and play with its interactive parameters in the next code cell. Then **answer the following questions**:

- What happens if a bigger kernel is used?

  Almost the same as before. Thicker and closer edges (better detection, worse localization -> this can lead to missing information), but now this all happens with almost no noise (when the appropiate sigma is selected).

- What kind of errors appear and disappear whenever sigma is modified?

  The bigger the sigma, the lesser the noise, which can lead to missing fine details or multiple response. Consequently, the lesser the sigma, the bigger the noise, which can lead to false detection. A middle ground must be found in order to keep fine details and not have false positives.

- Why the gradient image have lower values than the one from the original image? *Tip:* image normalization

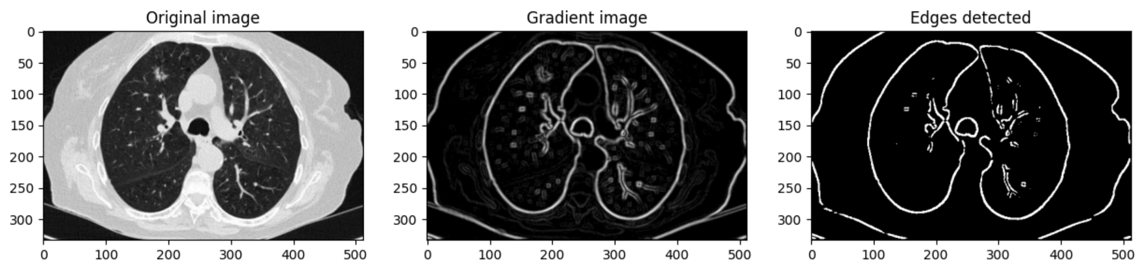
  Because it is computed from the gradient, which is esentially a derivative, which by definition measures the amount of change in the studied function. In this case, that function is the intensity values, and when they are very similar (e.g.: all the whitish intensities in the original image) the value of the gradient is approximately zero. That's why, in the gradient image, the whitish intensities are painted black (intensity *value = 0). Gradient only noticeably changes when abrupt changes in the studied function occur, in this case, when there's a change in colour (an edge ;))*

- Now that you have tried different techniques, in your opinion, which is the best one for this type of images?

  DroG, as it is esentially the same as the rest but with a smoothing (that certainly helps) pre-applied that reduces false positives, in spite of its higher computational cost.

In [9]:
```python
# Read the image
image = cv2.imread(images_path + 'medical_3.jpg', 0)

# Interact with the three input parameters
interactive(drog_kernel, image=fixed(image), w_kernel=(1,5,1), sigma=(0.4,5,0.5)
```

Out[9]:
```
 w_kernel   3
    sigma   2.40
threshold   120
```



Conclusion

Awesome! Now you have expertise in more applications of the convolution operator. In this notebook you:

- Learned basic operators for edge detection that perform a **discrete approximation of a gradient operator**.
- Learned **how to construct a DroG kernel** in an efficient way.
- Played a bit with them in the context of medical images, discovering some real and meaningful utilities.