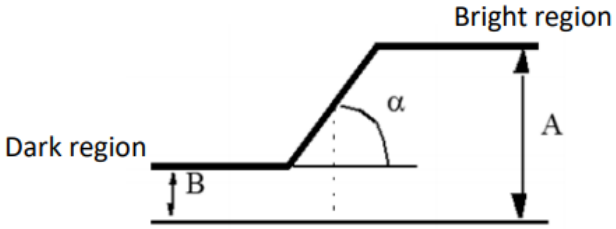
3.1 Operators based on first derivative

Let's suppose an image processing technique that pursues the finding of those pixels in images that exhibit brightness variations, that is, their intensity differ from that of their neighbor pixels. Such pixels are often called **edges**, so this technique receives the name of **edge detection**. Edge detection has many applications in computer vision tasks like image segmentation or data extraction/compression. We will explore some of them later in this book.

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:



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: its change in intensity, the angle of its slope, and its midpoint. However, in real images that model is not exactly followed by edges since images are discrete, and they are corrupted by noise.

Moreover, **the nature of edges may be diverse**:

- occlusion borders,
- different orientation of surfaces,
- different reflectance properties,
- different textures,
- illumination effects: shadows, reflections, etc.

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).

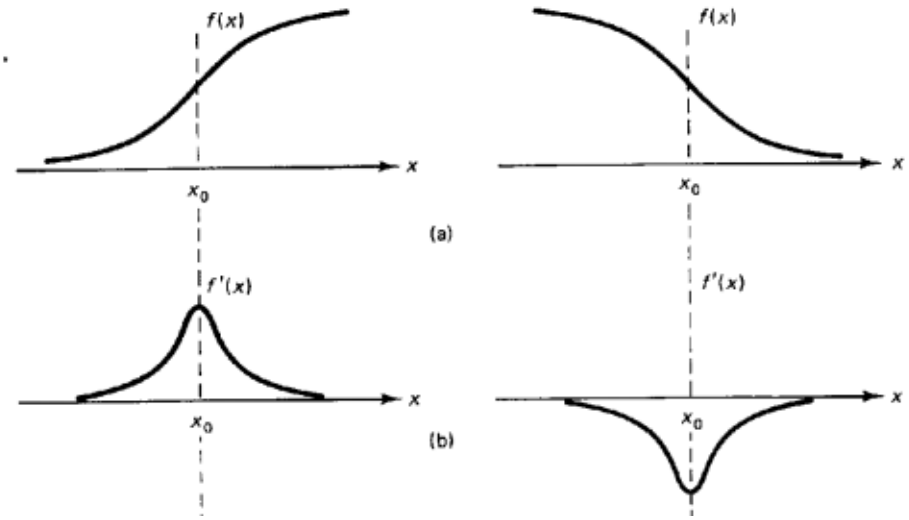
The following figure illustrates such errors.

Thereby, when designing a good edge detector, the goal is to achieve low detection and localization errors, as

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:

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

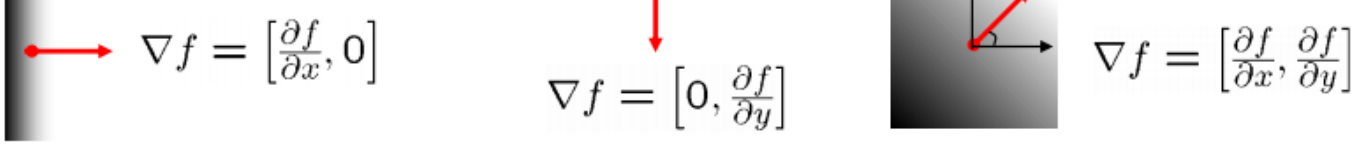
which points at the *direction* of maximum (positive) variation of $f(x, y)$:

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

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:



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]:
```
# 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', 0)
medical_2 = cv2.imread(images_path + 'medical_2.jpg', 0)
medical_3 = cv2.imread(images_path + 'medical_3.jpg', 0)

# 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_C (for columns, detecting vertical edges) and H_R (for rows, 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

0	0	0
0	0	1
0	-1	0

-1	0	0
0	1	0
0	0	0

Prewitt

$\frac{1}{3}$

1	0	-1
1	0	-1
1	0	-1

$\frac{1}{3}$

-1	-1	-1
0	0	0
1	1	1

Sobel

$\frac{1}{4}$

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

$\frac{1}{4}$

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

Frei-Chen

$\frac{1}{2+\sqrt{2}}$

1	0	-1
√2	0	-√2
1	0	-1

$\frac{1}{2+\sqrt{2}}$

-1	-√2	-1
0	0	0
1	√2	1

In general

$\frac{1}{2+K}$

1	0	-1
K	0	-K
1	0	-1

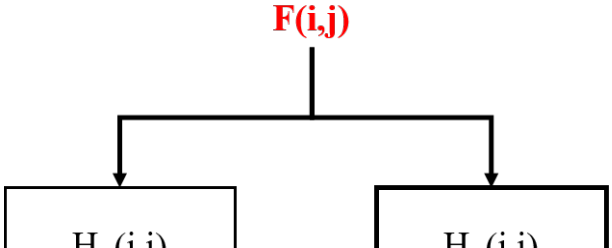
$\frac{1}{2+K}$

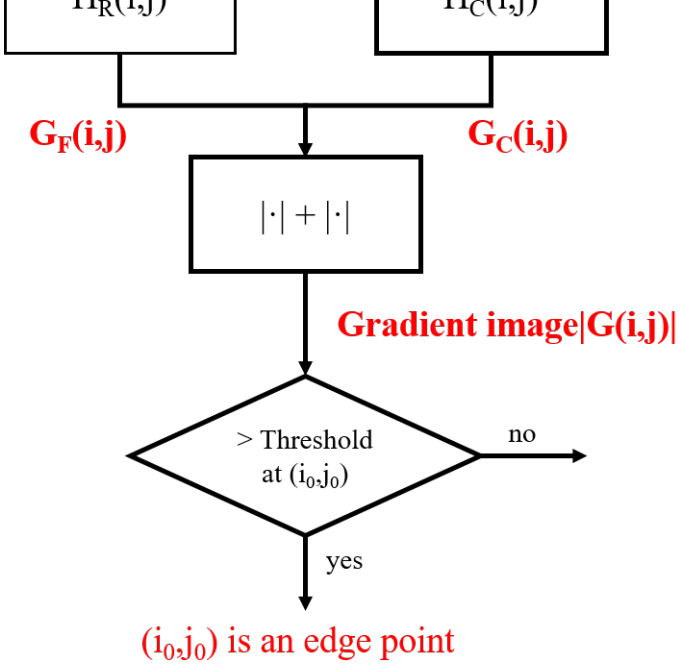
-1	-K	-1
0	0	0
1	K	1

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):

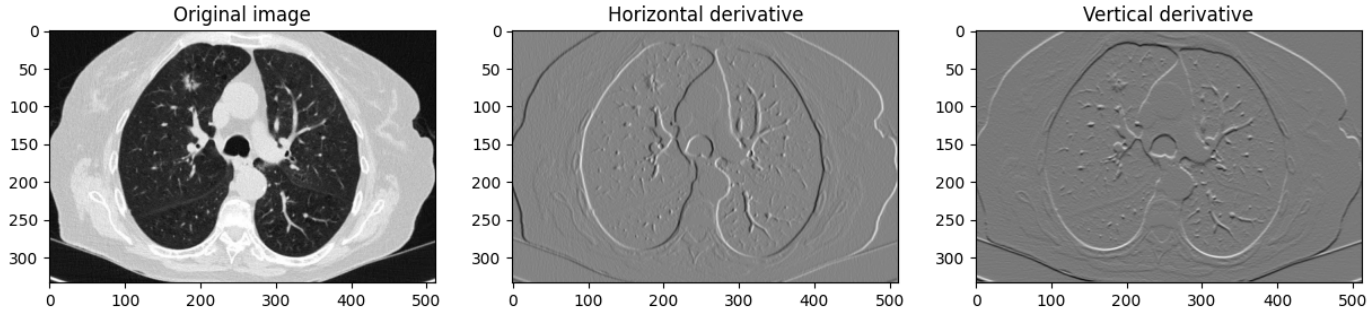
- Small template:
  - more precise localization (good localization).
  - more affected by noise (likely produces false positives).
- Large template:
  - less precise localization.
  - more robust to noise (good detector).
  - higher computational cost ($O(N \times N)$).

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 image (separat
# 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 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 (https://docs.opencv.org/2.4/modules/core/doc/operations_on_arrays.html#normalize) gradient image before thresholding.

Interesting functions: `np.absolute()` (https://numpy.org/doc/stable/reference/generated/numpy.absolute.html), `np.add()` (https://numpy.org/doc/stable/reference/generated/numpy.add.html),

cv2.threshold() (https://docs.opencv.org/master/d7/d1b

In [4]:
```
# ASSIGNMENT 3a
# Implement a function that that computes the gradient of an image, taking also as input th
# It must also binarize the resulting image using a threshold
# Show the input image, the gradient image (normalized) and the binarized edge image in a 1
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 to sum both
    #gradient_image = np.absolute(np.add(d_h,d_v))
    #Normalize gradient
    norm_gradient = np.copy(image)
    norm_gradient = cv2.normalize(gradient_image, None, 0, 256, cv2.NORM_MINMAX)

    # 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))
```

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

Expected output:

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

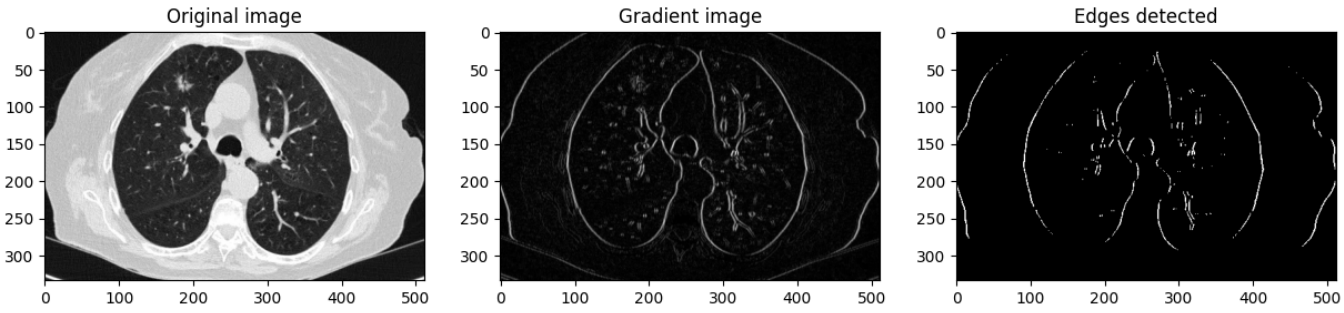
ASSIGNMENT 3b: Testing our detector

Now **try the implemented method** with different size Sobel kernels (3×3, 5×5, ...) and with other operators (Roberts, Prewitt, ...).

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

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

# Define kernel (Sobel, Roberts, Prewitt, ...)
#sobel
kernel_h = np.array([[1,0,-1],[2,0,-2],[1,0,-1]])*1/4
kernel_v = np.array([[1,0,-1],[2,0,-2],[1,0,-1]])*1/4
"""#roberts
kernel_h = np.array([[0,0,0],[0,0,1],[0,-1,0]])
kernel_v = np.array([[-1,0,0],[0,1,0],[0,0,0]])
#prewitt
kernel_h = np.array([[1,0,-1],[1,0,-1],[1,0,-1]])*1/3
kernel_v = np.array([[-1,-1,-1],[0,0,0],[1,1,1]])*1/3
#frie-chen
kernel_h = np.array([[1,0,-1],[np.sqrt(2),0,-np.sqrt(2)],[1,0,-1]])/(2+np.sqrt(2))
kernel_v = np.array([[-1,-np.sqrt(2),-1],[0,0,0],[1,np.sqrt(2),1]])/(2+np.sqrt(2))
#Interact with your code!"""
interactive( edge_detection_chart, image=fixed(image), kernel_h=fixed(kernel_h), kernel_v=f
```

Out[6]:

threshold 120



Thinking about it (1)

Now, **answer following questions**:

- What happens if we use a bigger kernel?

  larger kernel represents that more pixel are now a part of convolution that means it gives us less precise localization . so if we increase kernel size the image will become more blurry!

- There are differences between Sobel and other operators?

  Sobel is slightly superior in noise-suppression which is importany to deal with derivatives.!

- What errors appear using those operators?

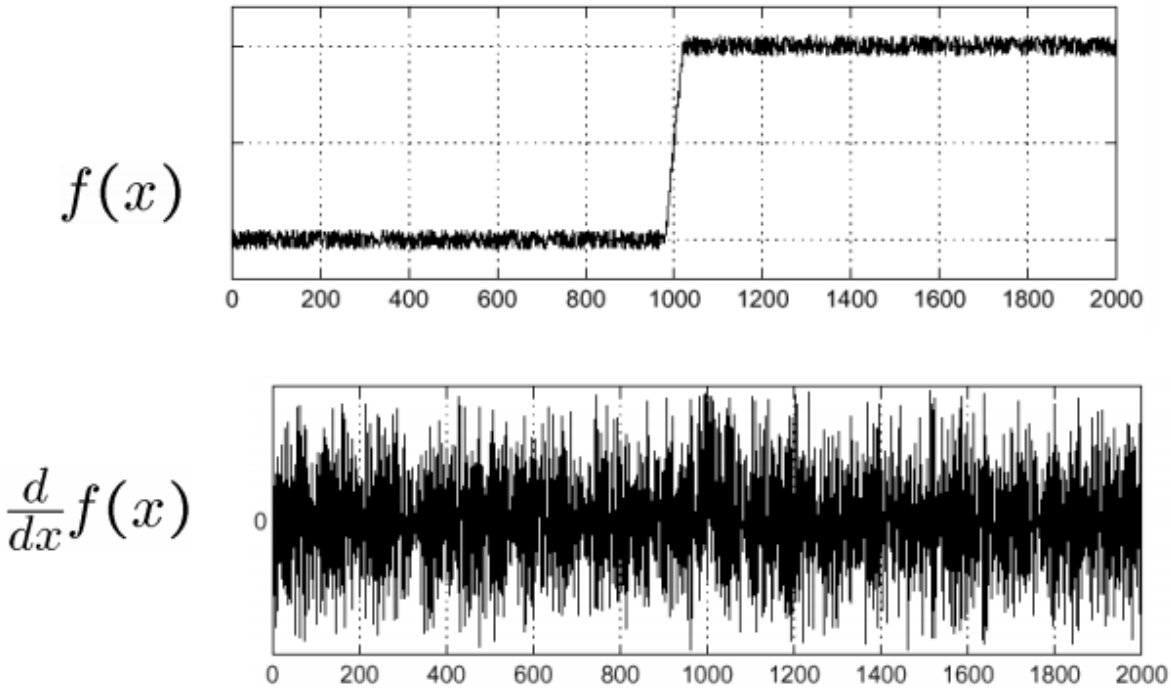
  the disadvantage to those opeartor is their senstivity to noise

- Why kernels usually are divided by a number? (e.g. 3×3 Sobel is divided by 4)

  The scale value 4 in sobel is use to ensure that intermediate results cannot go outside the range [−1, 1].similarly in each kernel the scale value is the weight of vertical values of the matrix .for example in prewitt the values are 1,1,1 which has a sum of 3 , so prewitt filter is divided by 3 to make the result in range [-1,1] !

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:

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

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:

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

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

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:

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

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 and calls to
# Inputs: an image, the kernel aperture size, the Gaussian standard deviation and the thres
# It returns the horizontal and vertical kernels
def gaussian_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 derivatives
    """
    # Write your code here!

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

    # 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 = gaussian_kernel_2D*(-x)/sigma**2 # Horizontal derivative
    DroG_v = gaussian_kernel_2D*(-y)/sigma**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
gaussian_kernel(image, w_kernel=1, sigma=1.2, threshold=100)
```

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?

  bigger kernel is better in reducing noise but makes the image more blurry and we miss some edges

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

  bigger sigma means we take more pixels into account which makes the image blurry while reducing more noise .bigger sigma is good detecter , more robust to noise

- Why the gradient image have lower values than the one from the original image? *Tip:* *image normalization (https://stackoverflow.com/questions/40645985/opencv-python-normalize-image/42164670)*

  The values of the gradient image correspond to the values of the first derivative, which are not high unless there are very sudden changes in intensity (and even in such cases, the values are greater than 255 only when the change from 0 to 255 are or are the same). To adjust the values to the intensity scale, we normalize the gradient image, to make the image appear correctly.

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

  in my opinion the derivative of gaussian is more robust than other techniques because it is less sensible to noise and a a better edge detecter!

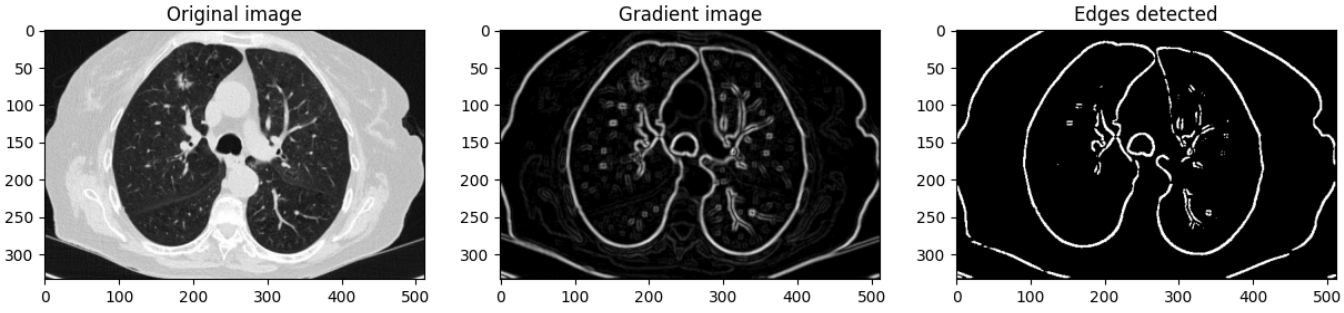
In [9]:
```
# Read the image
image = cv2.imread(images_path + 'medical_3.jpg', 0)

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

Out[9]:

w_kernel 3

sigma 2.40

threshold 120



ASSIGNMENT 5: Measuring efficiency

Finally, **you are asked to** compare the execution time of creating a DroG template using your previous code (combining two 1-D gaussian filters), with creating a DroG template by directly building the 2-D gaussian filter. Play with different (big) aperture sizes (up to 500).

Interesting resource: how to measure execution time in Python (https://stackoverflow.com/questions/14452145/how-to-measure-time-taken-between-lines-of-code-in-python)

In [10]:
```
# ASSIGNMENT 5
# Write your code here!

import time

print("Measuring the execution time needed for ...")

s = 1
w = 100

start = time.process_time() # Start timer

# FIRST WAY: Building the 2D gaussian filter from the 1D one

# Create 1D Gaussian filter
gaussian_kernel_1D = np.array([np.exp(-(pow(z,2)/(2*pow(s,2))))/(s*np.sqrt(2*np.pi)) for z

# 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 = gaussian_kernel_2D*(-x)/s**2 # Horizontal derivative
DroG_v = gaussian_kernel_2D*(-y)/s**2 # Vertical derivative
print("DroG building time using 1D Gaussian:", round(time.process_time() - start,5), "secor

start = time.process_time() # Start timer

# SECOND WAY: Directly building the 2D gaussian filter

# Create 2D Gaussian filter
gaussian_kernel_2D = np.array([[np.exp(-((pow(z,2)+pow(k,2))/(2*pow(s,2))))/(s*np.sqrt(2*np

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

# Construct DroG
DroG_h = gaussian_kernel_2D*(-x)/s**2 # Horizontal derivative
DroG_v = gaussian_kernel_2D*(-y)/s**2 # Vertical derivative

print("DroG building time using 2D Gaussian:", round(time.process_time() - start,5), "secor
```

```
Measuring the execution time needed for ...
DroG building time using 1D Gaussian: 0.01562 seconds
DroG building time using 2D Gaussian: 0.09375 seconds
```

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.

In [ ]:

In [ ]: