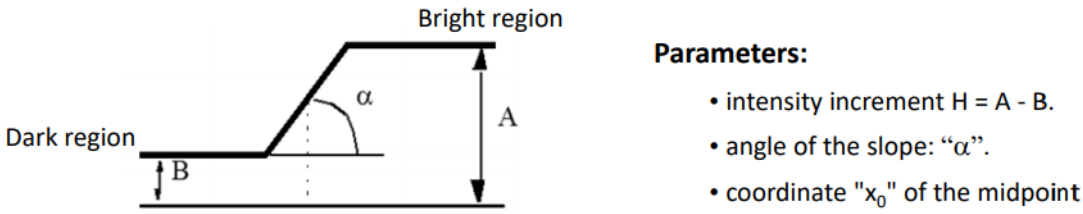
3 Edge detection

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:



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.

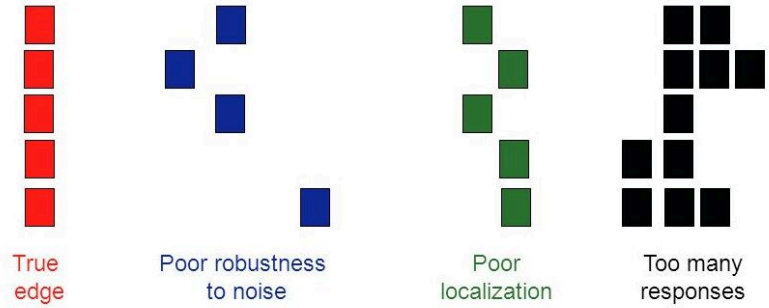
However, the idealized concept of an edge represented as this a continuous model doesn't fully capture the complexity found in discrete digital images, which are subject to noise and resolution limitations.

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.

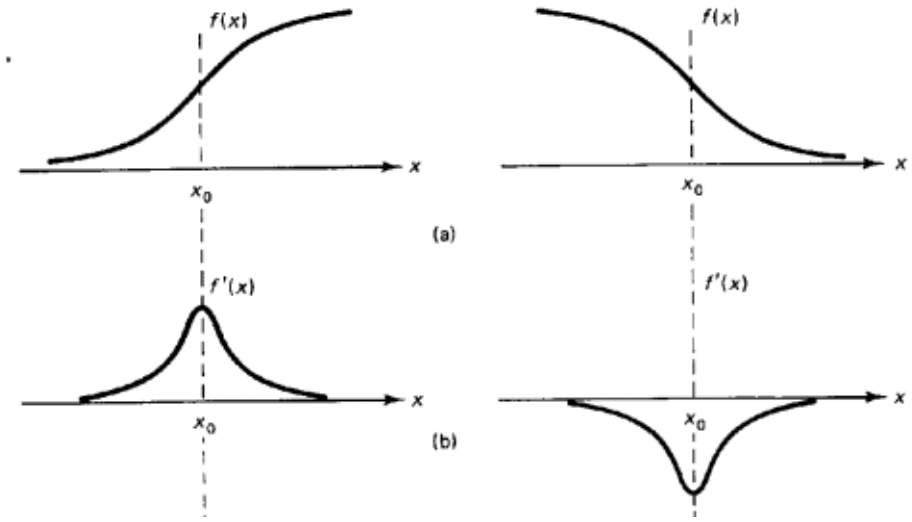


Thereby, when designing a good edge detector, the goal is to achieve low detection and localization errors, as well as to avoid multiple responses.

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:

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

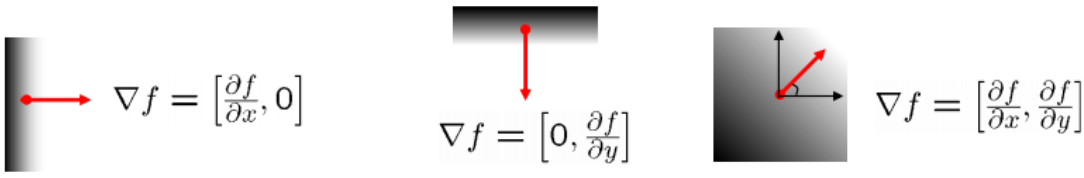
which points at the *direction* of maximum (positive) variation of $f(x, y)$:

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

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:



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_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

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

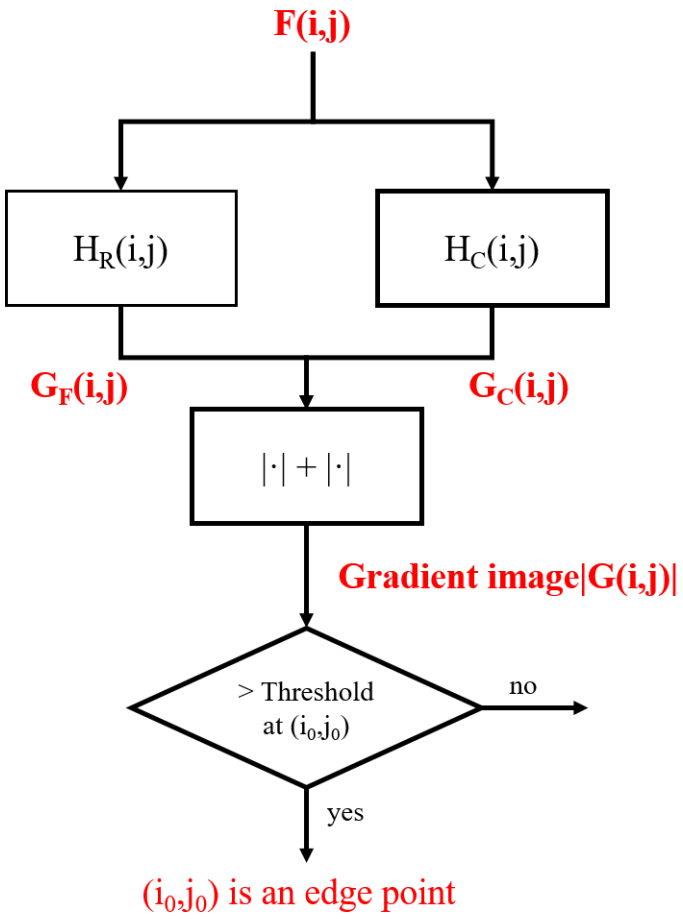
$\frac{1}{2+K}$

-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 ($O(N \times N)$).

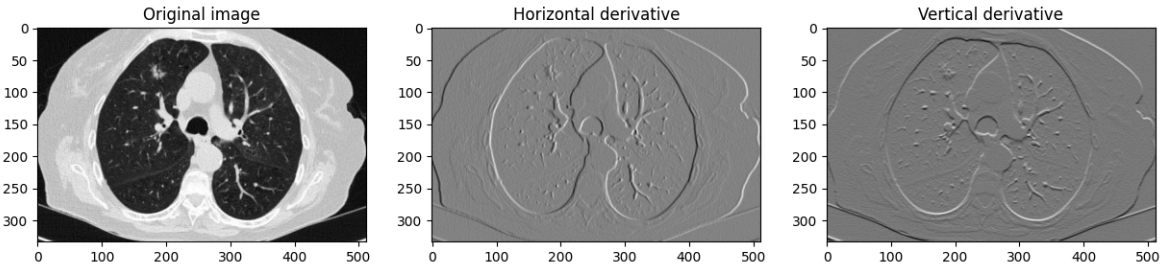
ASSIGNMENT 2: Playing with Sobel derivatives

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 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]:
```python
# 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.abs(d_h) + np.abs(d_v) # Hint: You have to sum both deri

    #Normalize gradient
    norm_gradient = np.copy(image)
    norm_gradient = cv2.normalize(gradient_image, None, 0, 255, 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]]
```

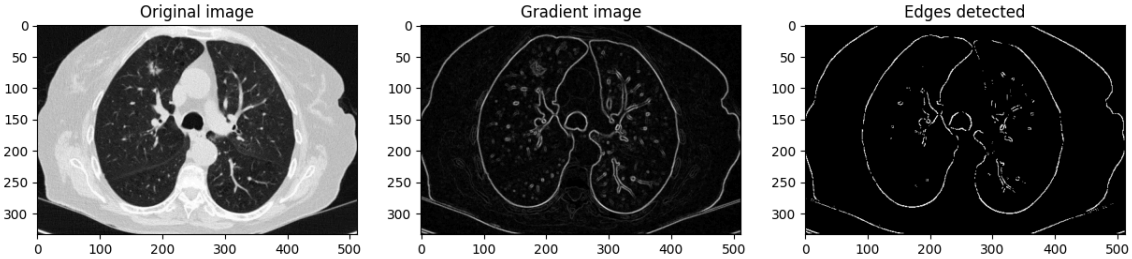
ASSIGNMENT 3b: Testing our detector

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

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 = 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

#Interact with your code!
interactive( edge_detection_chart, image=fixed(image), kernel_h=fixed(kernel_h),
```

Out[6]:

threshold 120



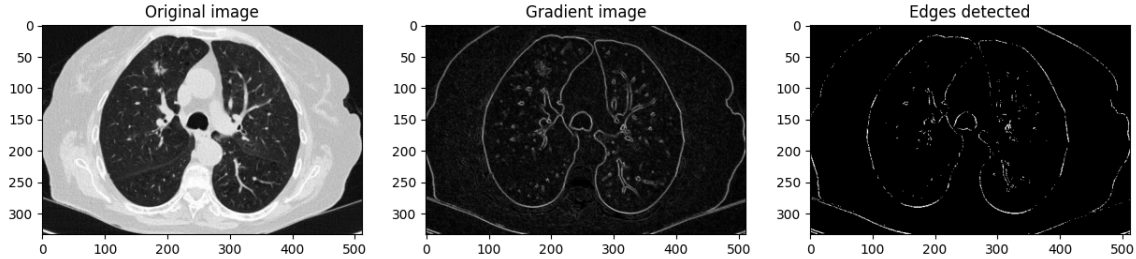
OPTIONAL

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

In [7]:
```
# 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 (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]])

#Interact with your code!
interactive( edge_detection_chart, image=fixed(image), kernel_h=fixed(kernel_h),
```

Out[7]:

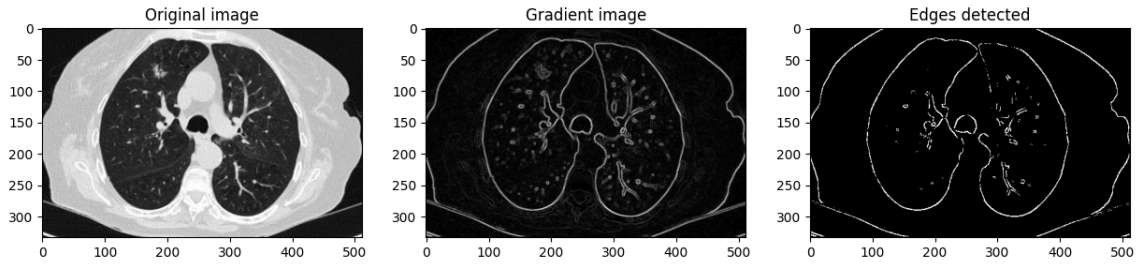
threshold 120



In [8]:
```
# 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 (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

#Interact with your code!
interactive( edge_detection_chart, image=fixed(image), kernel_h=fixed(kernel_h),
```

Out[8]:

threshold 120

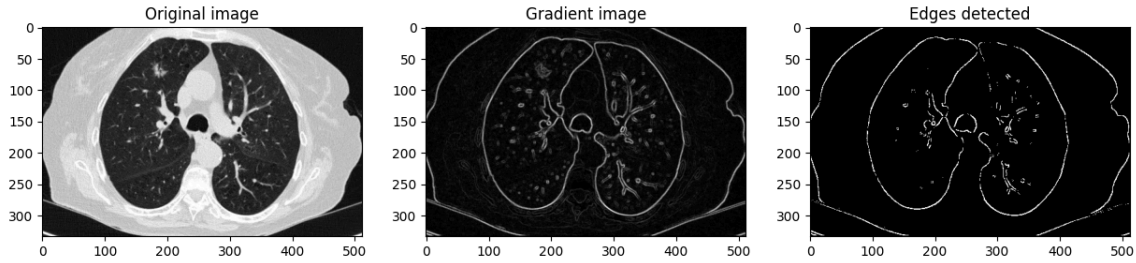


In [9]:
```
# ASSIGNMENT 3b
# Read the image, set you kernels (Sobel, Roberts, Prewitt, etc.) and interact w
# Write your code here!

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

# Define kernel (Frei-chen)
kernel_h = np.array([[1, 0, -1], [np.sqrt(2), 0, -np.sqrt(2)], [1, 0, -1]])*1/(2
```

```
kernel_v = np.array([[-1, -np.sqrt(2), -1], [0, 0, 0], [1, np.sqrt(2), 1]])*1/(2

#Interact with your code!
interactive( edge_detection_chart, image=fixed(image), kernel_h=fixed(kernel_h),
```

Out[9]:

threshold 120



Thinking about it (1)

Now, **answer following questions**:

- What happens if we use a bigger kernel?

  We gain robustness to noise in exchange for worse edge localisation and a higher computational cost.

- There are differences between Sobel and other operators?

  Given the same threshold, I can barely see difference between Sobel, Prewitt and Frei-chen operators. Roberts, however, detects way less edges than the said 3.

- What errors appear using those operators?

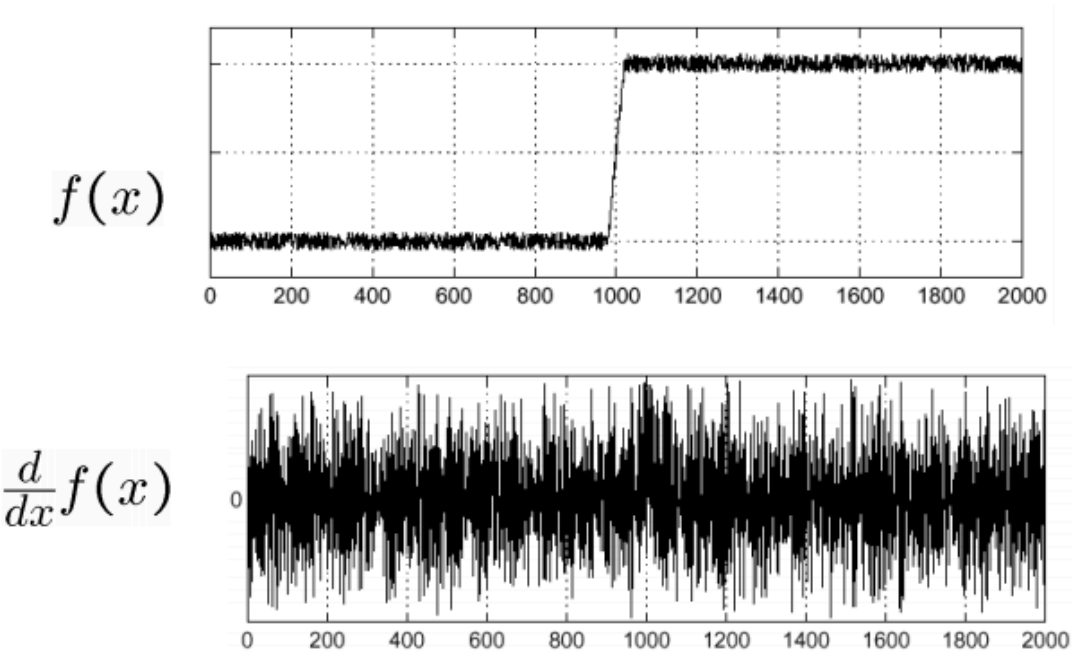
  There might be all type of errors present (detection error, localisation error and multiple responses), but the only visible ones (especially in Roberts) are the detection errors (false positives -> edges that aren't real, and false negatives -> real edges not shown).

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

  As adjacent pixels are given a bigger weight, in order to normalise the resulting image they are divided by the sum of the absolute values of these weights (in case of Sobel 3x3 we divide by 4 (|-2| + |2|)).

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:

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

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

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 [10]:
```
# 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 / (s * np.sqrt(2 * np.pi)) * np.exp(-(np.pc
```

```
    # 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 [11]:
```
# 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)
```

Out[11]:
```
(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?

  The bigger the kernel, the greater the reduction of noise, but the image is also blurred more, meaning a greater information loss. That is, the number of false positives is reduced, the number of false negatives is increased, and multiple responses could be given.

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

  With a lower sigma, false positives appear. With a higher sigma, false negatives and multiple responses appear.

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

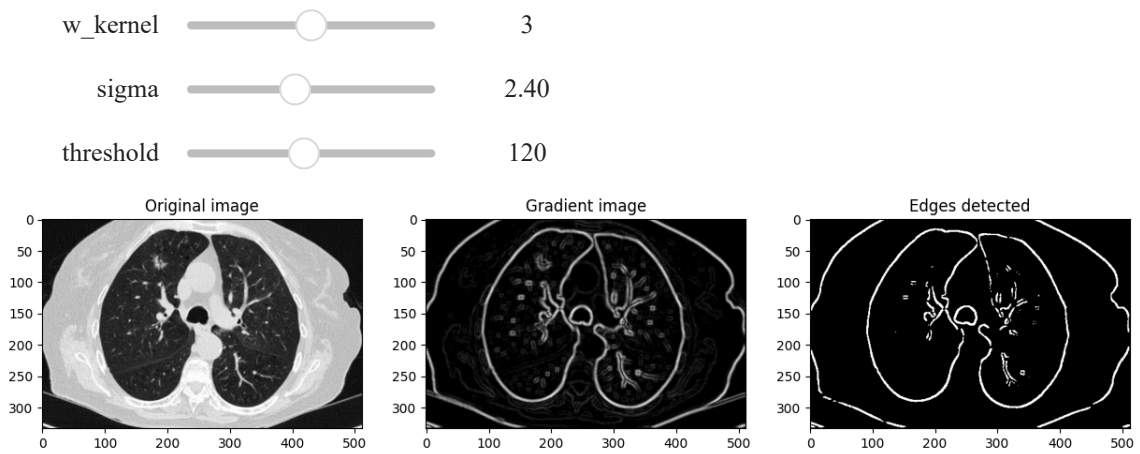
  After derivating, the values of pixels which aren't edges are lower than in the original image so that the binarisation technique is correctly applied. Keep in mind that values are normalised so that they don't get out of the range [0,255].

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

  The DroG, as if used correctly (good combination of the parametres) the result image will be robust to noise without being affected by new detecting errors.

In [12]:
```
# 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[12]:



Conclusion

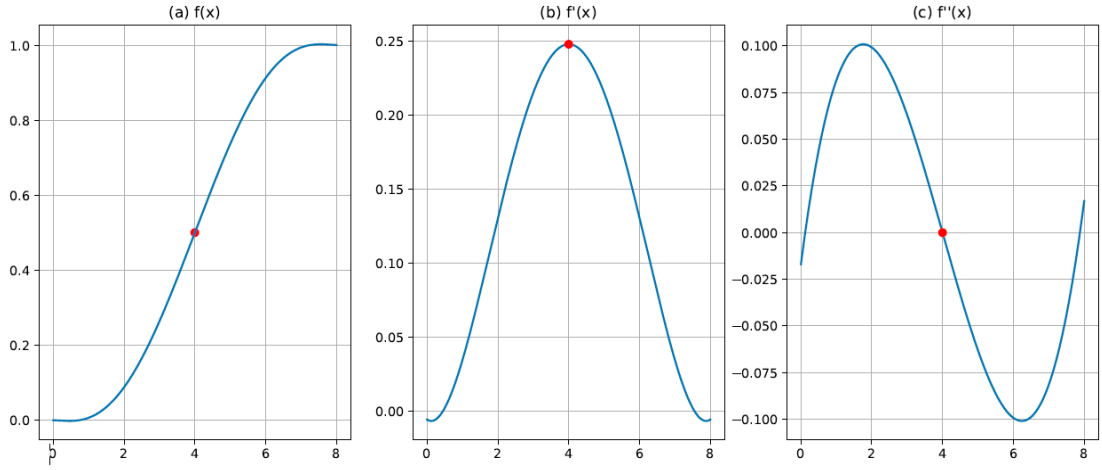
Awesome! Now you have expertise in more applications of the convolution operator. In this notebook you:

- 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.

3.2 Operators based on second derivative

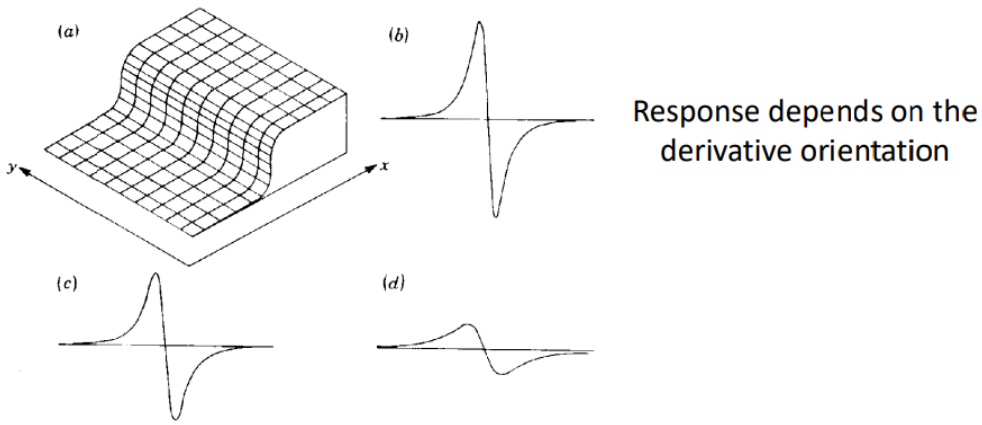
In the previous notebooks we saw how to detect edges by looking at the gradient image (first-derivative), but it is also possible to do that by analyzing the output of operators based on the second-derivative!

As you may remember, first derivative operators try to detect edges by looking for high magnitude values of such derivatives. The figure below shows a one-dimensional continuous function $f(x)$ in (a) and its first derivative in (b), where we can see that the point corresponding to the highest intensity difference reaches a maximum value:



The third figure (c) shows its second derivative, so we can check how such a value corresponds to... **a zero crossing!** That is, a second derivative yields a zero-crossing at points where the gradient presents a maximum, so we could detect edges looking for those crossings.

Unfortunately, things get a little tricky when moving to a 2D space (like images). Why? because depending on the orientation of the edge, this zero-crossing may go almost unnoticed (see, for example, d):



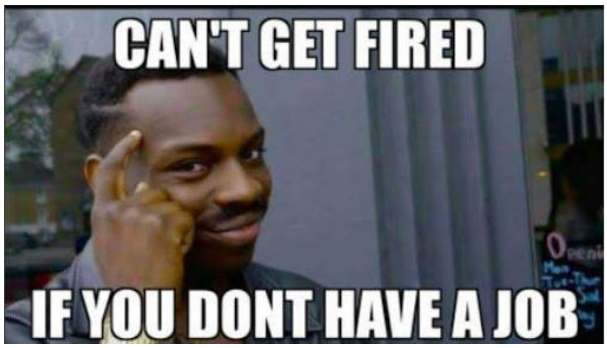
In this notebook we are going two explore two methods that face such issue and detect edges using the second derivative. These are:

- **Laplacian operator** (Section 3.2.1)
- **LoG operator** (Section 3.2.2)

Additionally, we will also take a look at a widely used algorithm that is a combination of different techniques: the **Canny algorithm** (Section 3.2.3).

Problem context - Edge detection for medical images

Unfortunately, you were not accepted (yet!) by the researching team at *Hospital Clínico* because the obtained results in the previous notebook were not as good as expected. Anyway, they have shown you the algorithms that they are currently using so you can learn for future opportunities. Let's have a look!



In [1]:
```
import numpy as np
from scipy import signal
import cv2
import matplotlib.pyplot as plt
import matplotlib
from ipywidgets import interact, fixed, widgets
from mpl_toolkits.mplot3d import Axes3D

matplotlib.rcParams['figure.figsize'] = (15.0, 15.0)

images_path = './images/'
```

3.2.1 Laplacian operator

To face the previously posed issue about unnoticed edges due to the derivate orientation, the idea behind the Laplacian operator is to combine second derivatives in perpendicular directions. Thus, it is defined as:

$$\nabla^2 f(i,j) = \frac{\partial^2}{\partial x^2} f(i,j) + \frac{\partial^2}{\partial y^2} f(i,j)$$

Note that, by definition, **it returns a scalar**, not a vector as in the gradient case. Indeed, the Laplacian is the trace of the *Hessian matrix*, which fully characterizes the second derivative of a function:

$$H(f) = \begin{bmatrix} \frac{\partial f^2}{\partial x^2} & \frac{\partial}{\partial x}\frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial y}\frac{\partial f}{\partial x} & \frac{\partial f^2}{\partial y^2} \end{bmatrix}$$

Compared with the first derivative-based edge detectors such as the Sobel operator, the Laplacian operator have a number of advantages:

- it is a linear operator,
- invariant to image orientation, and
- precise when localizing edges.

Implementation

Now that we know the theory, let's have a look at how the Laplacian operator is implemented:

1. We start by considering first derivatives (OpenCV uses Sobel, but any alternative is valid):

$$\frac{\partial f(x,y)}{\partial x} = f_x(x,y) \approx G_R(i,j) = f(i+1,j) - f(i,j)$$

$$\frac{\partial f(x,y)}{\partial y} = f_x(x,y) \approx G_C(i,j) = f(i,j+1) - f(i,j)$$

2. Then, take second derivatives using the previous definitions:

$$g = \frac{\partial f^2}{\partial x^2} = f_{xx}(x,y) \approx G_R(i,j) - G_R(i-1,j) = f(i+1,j) - 2f(i,j) + f(i-1,j)$$

$$h = \frac{\partial f^2}{\partial y^2} = f_{yy}(x,y) \approx G_C(i,j) - G_C(i-1,j) = f(i,j+1) - 2f(i,j) + f(i,j-1)$$

3. Finally, implement it as a convolution with a certain kernel, so $L[F(i,j)] = F(i,j) \otimes L(i,j)$. This would lead to the operation $L[F(i,j)] = (F(i,j) \otimes g) + (F(i,j) \otimes h)$, but thanks to the distributive property of convolution:

$$\underbrace{f \otimes (g+h)}_{\text{One convolution}} = \underbrace{(f \otimes g) + (f \otimes h)}_{\text{Two convolutions}}$$

We can obtain a kernel that carries out both convolutions at once!:

g + h = g+h=L

0	0	0
1	-2	1
0	0	0

0	1	0
0	-2	0
0	1	0

0	1	0
1	-4	1
0	1	0

Zero-crossing

Note that the result of applying the Laplacian operator is not directly an edges image, but a second-derivative image. Recall that in the case of operators based on the first derivative we had to combine the two images returned by the gradient operator, and then apply a threshold to select edges. In this case, **it is needed an algorithm to detect zero-crossings** in the second-derivative (Laplacian) image in order to return a binary image of edges.

An example of a simple zero-crossing algorithm could be:

1. Select a small positive number th (threshold).
2. A pixel is labelled as an edge if in the Laplacian image:
   - its value is smaller than $-th$ and at least one of its neighbours is bigger that th, or
   - its value is bigger than th and at least one of its neighbours is smaller than $-th$.

Advantages:

- Zero crossing produces a closed (or almost closed) contour, and
- it provides edges of 1-pixel width!

Limitations

- Unfortunately, the Laplacian operator is very sensitive to noise, resulting in a poor edge detection. Solution: If the image is blurred using a Gaussian filter before applying the Laplace operator, we can partially solve the noise problem. If this is done, the resultant o is called **LoG (Laplacian of Gaussian)**.

3.2.2 LoG operator

So, the LoG operator first smoothes the image, and then applies the Laplacian operator (or viceversa, it's conmutative!). Considering the convolution properties:

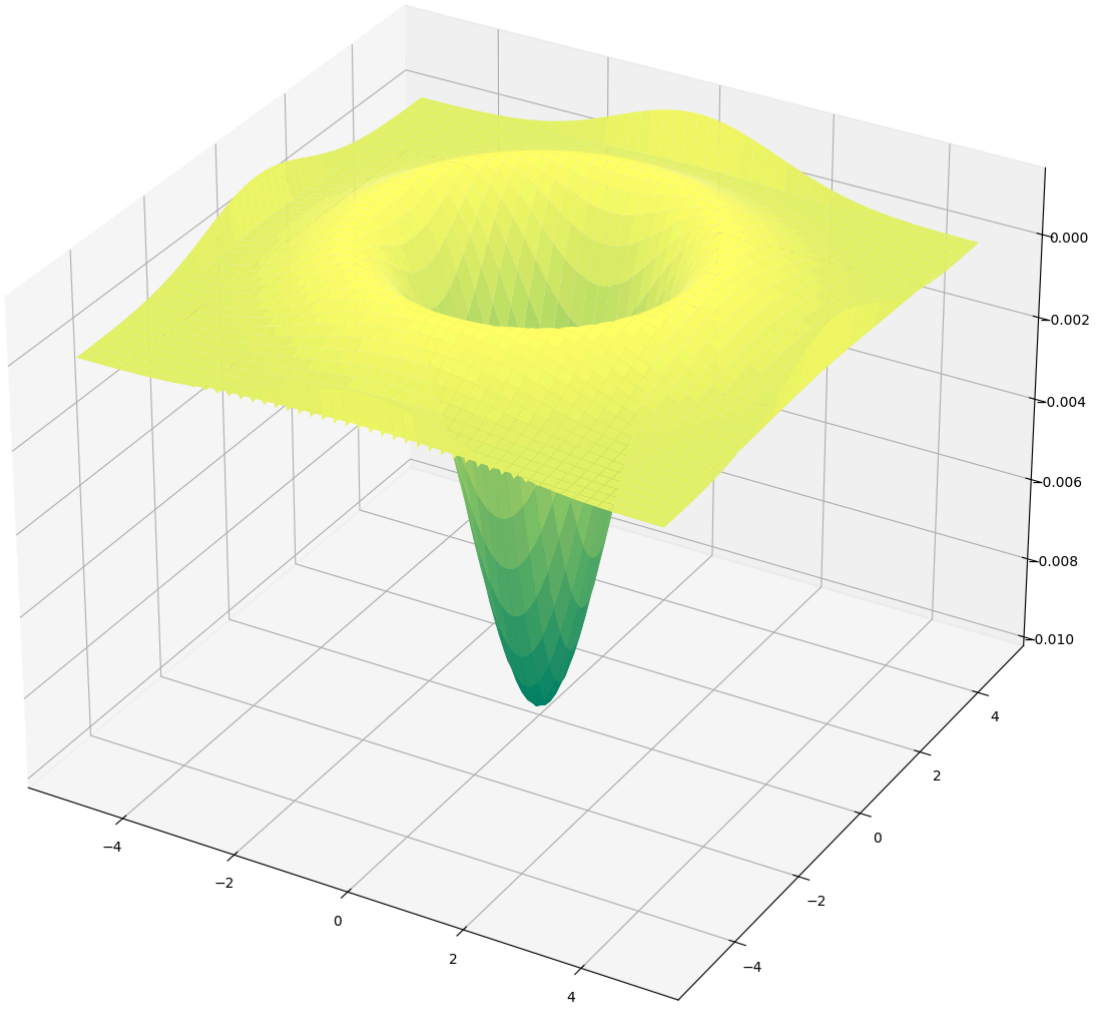
$$\nabla^2[f(x,y) \otimes g_\sigma(x,y)] = f(x,y) \otimes \nabla^2[g_\sigma(x,y)] = f(x,y) \otimes LoG_\sigma(x,y)$$

LoG is an isotropic operator, that is, it keeps radial symmetry. In this way, it is assumed that the covariance in both image dimensions is the same! Mathematically it is expressed as:

$$LoG_\sigma(x,y) = \frac{1}{\pi\sigma^4}\left[\frac{x^2+y^2}{2\sigma^2} - 1\right] \exp^{-\frac{x^2+y^2}{2\sigma^2}} = \frac{1}{\pi\sigma^4}\left[\frac{r^2}{2\sigma^2} - 1\right] \exp^{-\frac{r^2}{2\sigma^2}} = LoG_\sigma(r^2)$$

Let's print the LoG operator!

In [2]:
```
# Gauss filter
v = np.arange(-5,5,0.1)
X, Y = np.meshgrid(v,v)
covar = np.array([[2, 0],[0, 2]]) ## Assuming no correlation between X and Y
gauss_filter = np.exp(-0.5*(X**2/covar[0][0]+Y**2/covar[1][1]))

# Laplace filter
laplace_filter = np.array(([[0,1,0],[1,-4,1],[0,1,0]]), dtype="float")

# LoG operator
LoG = cv2.filter2D(gauss_filter, -1, laplace_filter)

# Plot it!
fig = plt.figure()
ax = plt.axes(projection='3d')
ax.plot_surface(X,Y,LoG,cmap='summer', edgecolor='none');
```



As a side note, the LoG operator is not separable. However, it can be implemented as **DoG (Difference of Gaussians)**, a sum of separable operators, reducing its complexity from $O(N^2)$ to $O(4N)$. The DoG is defined as:

$$DoG_{\sigma_1\sigma_2}(x,y) = g_{\sigma_1}(x,y) - g_{\sigma_2}(x,y) = g_{\sigma_1}(x)g_{\sigma_1}(y) - g_{\sigma_2}(x)g_{\sigma_2}(y)$$

Giving the ratio $\sigma_1/\sigma_2 = 1.6$ the best approximation of LoG. This complexity reduction approach is employed, for example, in the popular SIFT keypoint detector, as we will see in following notebooks.

Limitations

- It is computationally costly,
- it doesn't provide any information about edge orientations,
- the output contains negative and non-integer values, so for display purposes the image should be normalized to the range 0-255,
- it is needed a zero-crossing method, and
- it tends to round object corners (more heavily as σ grows).

Experiencing Laplacian and LoG operators

Now that we are almost experts in the Laplacian and LoG operators, let's play a bit with them!

ASSIGNMENT 1a: Applying Gaussian smoothing

First, complete the function `gaussian_smoothing()` that:

1. blurs an image using a Gaussian filter, then
2. normalizes it to leverage the full range of values $[0, \ldots, 255]$ (this is just a way to process the image in order to increase its contrast), and
3. finally returns the resulting image.

Interesting functions:

- For normalization you can use `cv2.normalize()`.

In [3]:
```python
# ASSIGNMENT 1a
# Implement a function that blurres an input image using a Gaussian filter and t
def gaussian_smoothing(image, sigma, w_kernel):
    """ Blur and normalize input image.

        Args:
            image: Input image to be binarized
            sigma: Standard deviation of the Gaussian distribution
            w_kernel: Kernel aperture size

        Returns:
            smoothed_norm: Blurred image
    """
    # Write your code here!

    # Define 1D kernel
    s=sigma
    w=w_kernel
    kernel_1D = np.array([1 / (s * np.sqrt(2 * np.pi)) * np.exp(-(np.pow(z, 2) /
```

```
    # Apply distributive property of convolution
    vertical_kernel = kernel_1D.reshape(2*w+1,1)
    horizontal_kernel = kernel_1D.reshape(1,2*w+1)
    gaussian_kernel_2D = signal.convolve2d(vertical_kernel, horizontal_kernel)

    # Blur image
    smoothed_img = cv2.filter2D(image,cv2.CV_8U,gaussian_kernel_2D)

    # Normalize to [0 254] values
    smoothed_norm = np.array(image.shape)
    smoothed_norm = cv2.normalize(smoothed_img, None, 0, 255, cv2.NORM_MINMAX) #

    return smoothed_norm
```

ASSIGNMENT 1b: Applying Laplacian and LoG operators

Now, we are going to see the differences between the Laplacian and LoG operators. For that complete the `laplace_testing()` function which:

1. applies the Laplacian operator to the input image and
2. to a blurred version of the input image (use the previously implemented function `gaussian_smoothing()` to smooth it). Notice that applying the Laplacian operator after smoothing the image is equivalent to applying the LoG operator.
3. Finally displays both images along with the original one in a 1x3 plot.

This function uses as inputs:

- an image to be processed,
- the size of the Laplacian filter (should be odd), and
- the parameters of the Gaussian filter.

Note that it would possible to reduce the computation time by precomputing LoG (as commented above). This is convolving the Laplacian and Gaussian filters instead of applying them separately.

Interesting functions:

- OpenCV defines the Laplace operator as cv2.Laplacian().

In [4]:
```
# ASSIGNMENT 1b
# Implement a function that applies the Laplacian operator to the input image an
# Display a 1x3 plot with the original image and the two resulting edge images.
# Inputs: image, size of the Laplacian kernel, sigma and size of the Gaussian ke
def laplace_testing(image, size_Laplacian, sigma, w_gaussian):
    """ Apply Laplacian and Log operators to an image.

        Args:
            image: Input image to be binarized
            size_Laplacian: size of Laplacian kernel (odd)
            sigma: Standard deviation of the Gaussian distribution
            w_gaussian: Gaussian kernel aperture size
    """
    # Write your code here!
```

```
    # Blur image
    blurred_img = gaussian_smoothing(image, sigma, w_gaussian)

    # Apply Laplacian to the original image
    laplacian = cv2.Laplacian(image, cv2.CV_16S, ksize=size_Laplacian)

    # Aplay Laplacian to the blurred image
    laplacian_blurred = cv2.Laplacian(blurred_img, cv2.CV_16S, ksize=size_Laplac

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

    # Show Laplacian
    plt.subplot(132)
    plt.imshow(laplacian, cmap='gray')
    plt.title('Laplacian without blurring')

    # Show LoG
    plt.subplot(133)
    plt.imshow(laplacian_blurred, cmap='gray')
    plt.title('Laplacian blurred (LoG)')
```

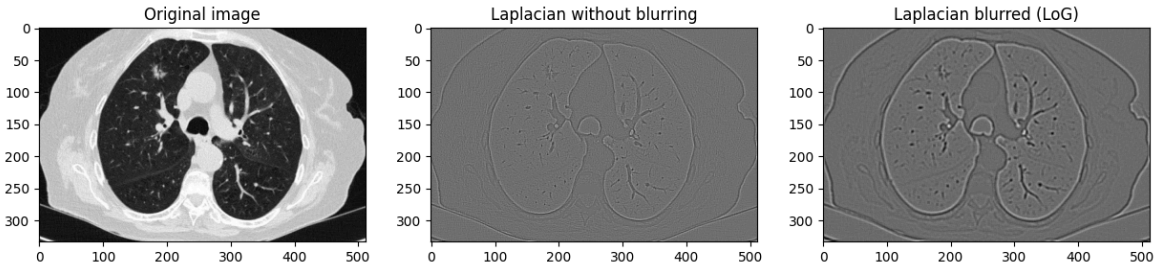
It is time to try this method to our medical images and play with interactive parameters.

In [5]:
```
# Read an image
image = cv2.imread(images_path + 'medical_3.jpg', 0)

# Interact with the parameters!
interact(laplace_testing, image=fixed(image), size_Laplacian=(1,7,2), sigma=(1,3
```

size_Laplacian 3

sigma 2.00

w_gaussian 2



Thinking about it (1)

Now, **answer the following questions**:

- Could be the Laplacian applied without a previous blurring? Does this have any drawback?

  Yes, it could, as shown above. The main drawback of this is the presence of noise.

- Are the images obtained in the previous function *edge images*?

No, they are the second derivative images.

- If not, what would be needed for obtaining the edges from those images?

A zero-crossing detection algorithm is needed.

The next code cell implements zero crossing detection, so it retruns the final edges image.

In [6]:
```
# This function takes an image, applies LoG (in reality, Gaussian and then lapla
# It will show the original image, the image with LoG and the image with Zero-Cr
def zero_crossing(image, size_Laplacian, sigma, w_gaussian, threshold):

    blurred_img = gaussian_smoothing(image, sigma, w_gaussian)

    laplacian = cv2.Laplacian(blurred_img, cv2.CV_16S, ksize=size_Laplacian)

    laplacian_blurred = cv2.normalize(laplacian, None, -255, 255, cv2.NORM_MINMA

    width, height = laplacian_blurred.shape
    edges = np.zeros_like(laplacian_blurred,np.uint8)

    for x in range(1,width-1):
        for y in range(1,height-1):
            neighbors = [
                laplacian_blurred[x-1, y-1], laplacian_blurred[x, y-1], laplacia
                laplacian_blurred[x-1, y],                               laplacia
                laplacian_blurred[x-1, y+1], laplacian_blurred[x, y+1], laplacia
            ]

            if(laplacian_blurred[x,y] > threshold and any(n < -threshold for n i
                edges[x,y] = 255

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

    # Show LoG blurred image
    plt.subplot(132)
    plt.imshow(laplacian_blurred, cmap='gray')
    plt.title('LoG')

    # Show LoG with Zero-Crossing
    plt.subplot(133)
    plt.imshow(edges, cmap='gray')
    plt.title('LoG with Zero-Crossing')

    plt.show()
```

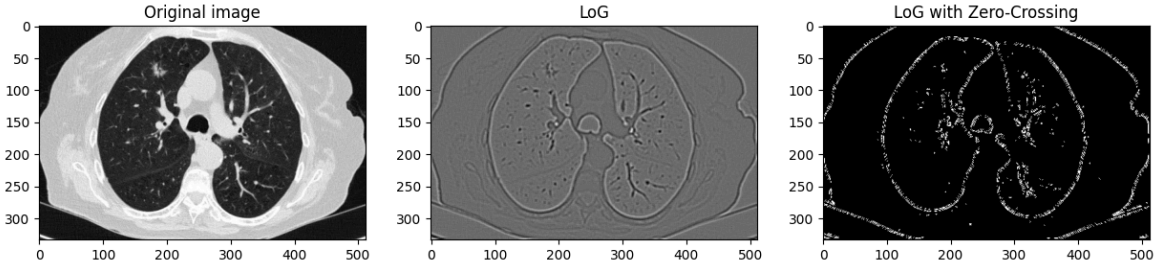
In [7]:
```
image = cv2.imread(images_path + 'medical_3.jpg', 0)

interact(zero_crossing, image=fixed(image), size_Laplacian=(1,7,2), sigma=(1,3,0
```

size_Laplacian 3

sigma 2.00

w_gaussian 2

threshold 20



3.2.3 The Canny algorithm

The Canny edge detector[1] is an algorithm that combines a number of techniques:

- the DroG operator,
- non-maxima suppression, and
- hysteresis.

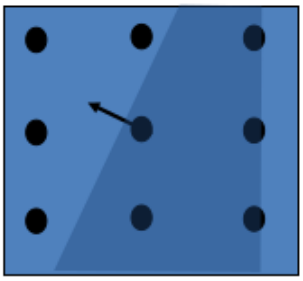
It was designed to be a good detector, yield a good localization, and to provide a single response!

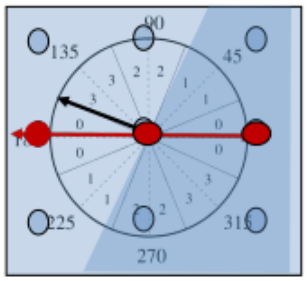
This algorithm consists of the following steps:

1. **Noise filtering and gradient image**. Apply the DroG operator to reduce noise and obtain a gradient image.

2. **Non-maximum suppression**. This removes pixels that are not considered to be part of an edge. Typically, the gradient image obtained after using DroG presents thick edges. The idea is to keep only those pixels that are maximum within their neighborhood in the direction of the gradient, suppressing the rest of them. Hence, only thin lines (candidate edges) will remain. For that:

    - We consider 4 main directions or *angular sectors*: $[0, 45]$, $[45, 90]$, $[90, 135]$, $[135, 180]$. The gradient angle $\theta[i, j]$ is approximated by where it lays.
    - A 3x3 filter is moved over the gradient image $G[i, j]$ at each pixel, and it suppresses the edge strength of the center pixel (for example by setting its value to 0) if its magnitude is not greater than the magnitude of the two neighbors in the gradient direction. This way we have a single response at each edge.

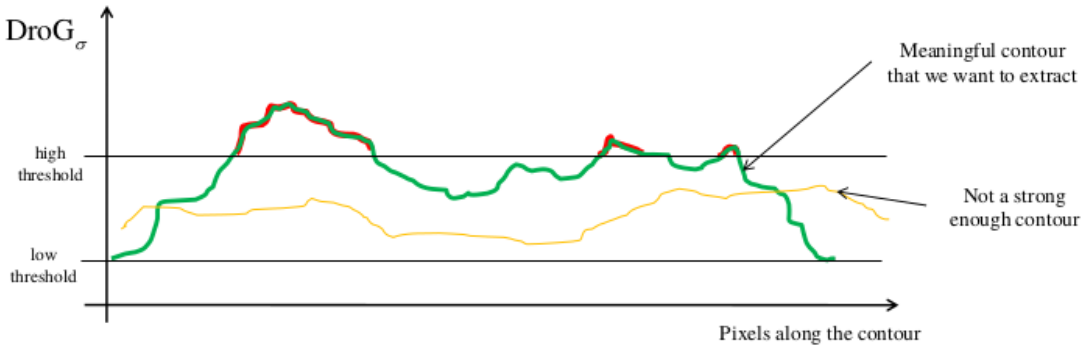




We pick the maximun of these three pixels

3. **Hysteresis**: The final step, for which the Canny algorithm uses two thresholds (upper and lower) to determine edge pixels:

    - If the grey level of a candidate pixel of the gradient image is higher than the upper threshold, the pixel is accepted as an edge.
    - If the grey level of a candidate pixel of the gradient image is below the lower threshold, then it is rejected.
    - If the grey level of a candidate pixel of the gradient image is between the two thresholds, then it will be accepted only if it is connected to a pixel that is above the upper threshold and rejected otherwise.



This algorithm can be executed repeatedly with different levels of smoothing (changing the sigma of the DroG operator). Different sigmas produce edges at different spatial features.

ASSIGNMENT 2: The enormously popular Canny algorithm

Complete `canny_testing()`, which applies the Canny algorithm. Note that OpenCV Canny's implementation does not apply Gaussian smoothing, but directly applies Sobel. This gives to us the opportunity to:

1. check the performance of this technique by considering the initial image and a smoothed version of it. *Note: use our popular `gaussian_smoothing()` function for blurring the image*
2. After this, display both resulting images along the original one.

This function takes as arguments:

- an image,
- both lower and upper Canny thresholds, and

- the parameters of the Gaussian filter.

Interesting functions:

- OpenCV implements the Canny algorithm in cv2.Canny().

In [8]:
```
# ASSIGNMENT 2
# Implement a function that applies the Canny operator to an input image and to
# Display a 1x3 plot with the original image and the two resulting edge images.
# Inputs: image, size of the Laplacian kernel, sigma and size of the Gaussian ke
def canny_testing(image, lower_threshold, upper_threshold, sigma, w_gaussian):
    """ Apply Canny algorithm to an image.

        Args:
            image: Input image to be binarized
            lower_threshold: bottom value for hysteresis
            upper_threshold: top value for hysteresis
            sigma: Standard deviation of the Gaussian distribution
            w_gaussian: Gaussian kernel aperture size
    """

    # Smooth image
    blurred_img = gaussian_smoothing(image,sigma,w_gaussian)

    # Apply Canny to original image
    canny = cv2.Canny(image,lower_threshold,upper_threshold)

    # Apply Canny to blurred image
    canny_blurred = cv2.Canny(blurred_img,lower_threshold,upper_threshold)

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

    # Show Canny without blurring
    plt.subplot(132)
    plt.imshow(canny, cmap='gray')
    plt.title('Canny without smoothing')

    # Show Canny with blurring
    plt.subplot(133)
    plt.imshow(canny_blurred, cmap='gray')
    plt.title('Canny smoothed')
```

Among the multiple parameters of this algorithm, it is interesting to check its performance with different levels of smoothing (changing the sigma of the DroG operator). As commented, different sigma produces edges at different spatial features. **Try the effect of this and other parameters** playing with the interactive parameters in the following code cell. You can also try with your own images.

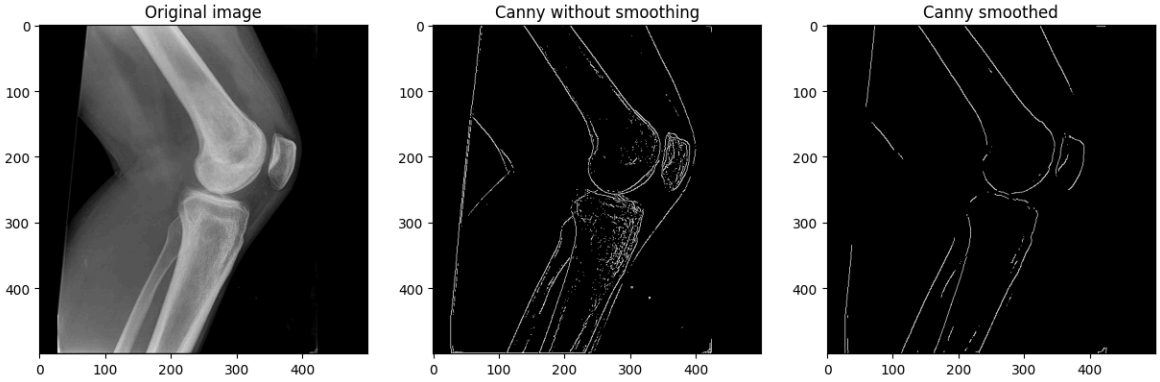
In [9]:
```
# Read an image
image = cv2.imread(images_path + 'medical_2.jpg', 0)

# Interact with the parameters
interact(canny_testing, image=fixed(image), lower_threshold=(0,260,20), upper_th
```

lower_thres… 120

upper_thres… 120

sigma 2.00

w_gaussian 2



Thinking about it (2)

Now, **answer following questions**:

- Could Canny be applied without a previous blurring? Which are the consequences of this?

  Yes, it could. The main consequence is the presence of noise, resulting in many false positives.

- What is a *good* value for both, lower and upper thresholds? Would these values be the same for any input image?

  It really depends on the image and what we want to obtain from it. Since I think what we want in this specific image is the outline of the bones, a lower threshold of 60 and an upper threshold of 140 could work.

- Now that you have tried a good number of edge detection methods, **which one is your favorite, and why?**

  Definitely the Canny algorithm. It (generally) provides the best result by solving problems present in previous techniques (uses DroG to remove noise, non-maximum suppression to avoid multiple responses, and hysteresis for a good selection of edges).

Conclusion

Terrific! You finished this notebook, that includes information about:

- Laplacian and LoG operators and the importance of smoothing, and
- how the Canny algorithm is implemented and how to use it.

Curiosity

The Canny algorithm is a well known algorithm in the computer vision field. It is used in a lot of modern technologies. However, the original paper was published in 1986 by John Canny[1].

References

[1]: CANNY, John. A computational approach to edge detection.. IEEE Transactions on pattern analysis and machine intelligence, 1986, no 6, p. 679-698.