Assignment 1 Report

February 3, 2021

0.1 Assignment 1

CIS 666, Spring 2021 Bradley Dowling 2657649 —

For this project, I utilized standard library functions, Cython, Numpy, and Pillow. The standalone program, images.py, uses a command line interface created with the standard library module ArgParse. Pillow is used only for image decoding and encoding. Numpy is used for multidimensional arrays. Cython is used to improve the execution of the 2D convolution function.

[80]:
```
from itertools import chain
from math import sqrt
import numpy as np
from PIL import Image
```

[3]:
```
def get_image_data(filename, mode="L", dtype=np.uint8):
    """Return nupmy array from image file"""
    return np.asarray(Image.open(filename).convert(mode), dtype=dtype)

def get_new_image(outdata, mode):
    """Return new image from processed image data"""
    return Image.fromarray(outdata, mode)
```

0.2 ### 1. Write a program to perform histogram equalization on a grayscale image.

- Display the original and histogram equalized images.
- Repeat the experiment for three different types: extreme dark, medium, and extreme light

[4]:
```
# First Image
LEVELS = 256
indata = get_image_data("./pictures/butterflies.jpg", mode="L")

print("Original Color Image")
Image.open("./pictures/butterflies.jpg")
```

```
Original Color Image
```

[4]:

[5]:
```
print("Grayscale Image")
Image.open("pictures/butterflies.jpg").convert("L")
```

```
Grayscale Image
```

[5]:

To implement histogram equalization, I take a count of individual luminosity levels using the level as the index of an array, and compute the cumulative distribution function to equalize the original histogram:

[6]:
```
def histogram(imgdata):
    """Returns a histogram of length 255 for a single channel"""
    histogram = [0] * LEVELS
    for pixel in imgdata.flatten():
        histogram[pixel] += 1
    return histogram
```

[7]:
```
hist = histogram(indata)
print(hist[0:9], "...", hist[246:])
```

```
[0, 1, 4, 14, 40, 99, 271, 1252, 6475] … [636, 663, 862, 985, 1206, 1530,
1512, 1049, 1068, 74517]
```

The cumulative distribution function is defined as

$$cdf_x(i) = \sum_{j=0}^{i} p_x(x = j)$$

where i represents the pixels in an image, j is a given luminosity level, x is a given pixel, and p is the probability of an occurance of a particular pixel.

[8]:
```
def cdf(histogram):
    """Returns the cumulative distribution of the image histogram"""
    cdf = [0] * LEVELS
    cdf[0] = histogram[0]
    for i in range(1, LEVELS):
        cdf[i] = cdf[i-1] + histogram[i]
    return cdf
```

[9]:
```
imgcdf = cdf(hist)
print(imgcdf[0:9], "...", imgcdf[250:])
```

```
[0, 1, 5, 19, 59, 158, 429, 1681, 8156] … [2224324, 2225854, 2227366, 2228415,
2229483, 2304000]
```

Verify that this CDF makes sense:

[10]:
```
# Is the cumulative total equal to the dimensions of the image?
imgcdf[255] == indata.shape[0] * indata.shape[1]
```

[10]:
```
True
```

To equalize the image, I use the general histogram equalization formula:

$$h(v) = round(\frac{cdf(v) - cdf_{min}}{(M \times N) - cdf_{min}} \times (L - 1))$$

Where L corresponds to the global constant LEVELS, M, N are the dimensions of the image, and cdf_{min} is the nonzero minimum of the cumulative distribution function.

[11]:
```
def equalize(cdf):
    """Linearize histogram based on CDF and normalize to [0, 255]"""
    nzmin = min(filter(lambda x: x > 0, cdf))
    if cdf[-1] - nzmin == 0:
        raise ValueError("Channel histogram cannot be equalized: Division by␣
↪zero")
    return [round(((cdf[i] - nzmin) / (cdf[-1] - nzmin)) * (LEVELS - 1)) for i␣
↪in range(LEVELS)]
```

[12]:
```
imgeq = equalize(imgcdf)
print(imgeq[0:9], "...", imgeq[90:100], "...", imgeq[250:])
```

[0, 0, 0, 0, 0, 0, 0, 0, 1] … [142, 144, 145, 146, 148, 149, 150, 151, 153, 154] … [246, 246, 247, 247, 247, 255]

Next, transform the data by using the luminosity level of the pixels in the original image as the index into the equalized histogram, and rebuild the output image matrix from those values:

[13]:
```
def transform_grayscale(imgeq, indata):
    """Return matrix of luminosity levels mapped from equalized histogram"""
    outdata = np.zeros_like(indata)
    for i in range(indata.shape[0]):
        for j in range(indata.shape[1]):
            outdata[i, j] = imgeq[indata[i, j]]
    return outdata
```

[14]:
```
outdata = transform_grayscale(imgeq, indata)
```

Finally, reconstruct the image:

[15]:
```
get_new_image(outdata, "L")
```

[15]:

0.3 ### Repeat for three different types: extreme dark, medium dark, extreme light

[16]:
```
def hist_equalize(filename, mode="L"):
    indata = get_image_data(filename, mode)
    return get_new_image(transform_grayscale(equalize(cdf(histogram(indata))),␣
↪indata), mode)
```

0.4 #### Extreme dark

[17]:
```
# Extreme Dark
Image.open("pictures/extreme_dark.jpg")
```

[17]:

[18]:
```
# Extreme Dark
hist_equalize("pictures/extreme_dark.jpg")
```

[18]:

0.5 #### Medium Dark

[19]:
```
Image.open("pictures/colorcity.jpg").convert("L")
```

[19]:

[20]:
```
hist_equalize("pictures/colorcity.jpg")
```

[20]:

0.6 #### Extreme Light

[21]:
```
Image.open("pictures/extreme_light.jpg").convert("L")
```

[21]:

[22]:
```
hist_equalize("pictures/extreme_light.jpg")
```

[22]:

- Display the original and histogram-equalized images
- Repeat for three image types: extreme dark, medium dark, and extreme light

For RGB images, the procedure is similar to grayscale, but we must perform the procedure on each channel. For HSV images, we can either manipulate all three channels to get some strange effects, or only apply the procedure to the Value channel. RGBA could be done in a similar fasion, but is not treated here.

[23]:
```
def rgb_histogram(indata):
    red = [0] * LEVELS
    green = [0] * LEVELS
    blue = [0] * LEVELS
    for color in chain.from_iterable(indata):
        red[color[0]] += 1
        green[color[1]] += 1
        blue[color[2]] += 1
    return red, green, blue
```

[24]:
```
def hsv_histogram(indata):
    value = [0] * LEVELS
    for i in chain.from_iterable(indata):
        value[i[2]] += 1
```

```
    return value
```

[25]:
```
def rgb_cdf(rgb):
    return cdf(rgb[0]), cdf(rgb[1]), cdf(rgb[2])
```

[26]:
```
def rgb_eq(rgb):
    return equalize(rgb[0]), equalize(rgb[1]), equalize(rgb[2])
```

[27]:
```
def transform_rgb(rgbeq, indata):
    outdata = np.zeros_like(indata)
    for i in range(indata.shape[0]):
        for j in range(indata.shape[1]):
            outdata[i, j] = [
                rgbeq[0][indata[i, j, 0]],
                rgbeq[1][indata[i, j, 1]],
                rgbeq[2][indata[i, j, 2]]
            ]
    return outdata
```

[28]:
```
def transform_hsv(hsveq, indata):
    outdata = np.zeros_like(indata)
    for i in range(indata.shape[0]):
        for j in range(indata.shape[1]):
            outdata[i, j] = [
                indata[i, j, 0],
                indata[i, j, 1],
                hsveq[indata[i, j, 2]]
            ]
    return outdata
```

[29]:
```
def hist_eq_rgb(filename, mode="RGB"):
    indata = get_image_data(filename, mode)
    return transform_rgb(rgb_eq(rgb_cdf(rgb_histogram(indata))), indata)

def hist_eq_hsv(filename, mode="HSV"):
    indata = get_image_data(filename, mode)
    return transform_hsv(equalize(cdf(hsv_histogram(indata))), indata)
```

[30]:
```
# Initial Image
f = "pictures/butterflies.jpg"
mode = "RGB"
Image.open(f)
```

[30]:

[31]:
```
get_new_image(hist_eq_rgb(f), mode)
```

[31]:

[32]:
```
# Jupyter will not print this to the report, so this one will launch separately
mode = "HSV"
get_new_image(hist_eq_hsv(f, mode), mode).show()
```

[33]:
```
# What if we apply the RGB process to the channels of an HSV image?
get_new_image(hist_eq_rgb(f, mode), mode).show()
```

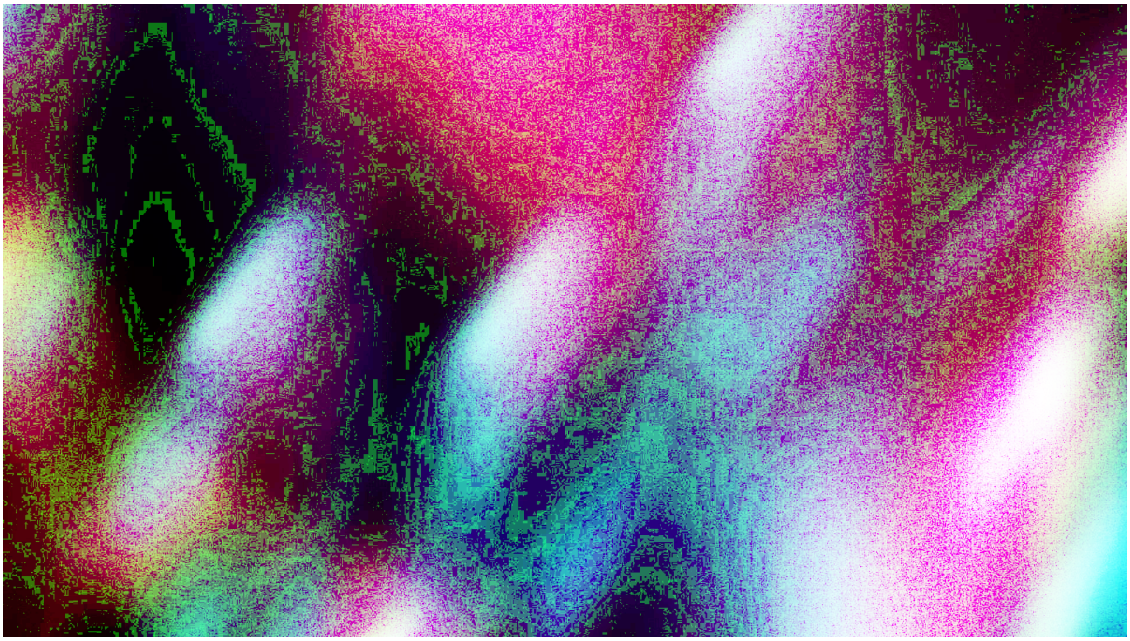
0.7 #### Extreme Dark

[34]:
```
f = "pictures/extreme_dark_color.jpg"
mode = "RGB"
Image.open(f).convert(mode)
```

[34]:

[35]:
```
get_new_image(hist_eq_rgb(f, mode), mode)
```

[35]:



[36]:
```
mode = "HSV"
get_new_image(hist_eq_hsv(f, mode), mode).show()
```

[37]:
```
get_new_image(hist_eq_rgb(f, mode), mode).show()
```

0.8 #### Medium Dark

[38]:
```
f = "pictures/colorcity.jpg"
mode = "RGB"
Image.open(f).convert(mode)
```

[38]:

[39]:
```
get_new_image(hist_eq_rgb(f, mode), mode)
```

[39]:

[40]:
```
mode = "HSV"
get_new_image(hist_eq_hsv(f, mode), mode).show()
```

[41]:
```
get_new_image(hist_eq_rgb(f, mode), mode).show()
```

0.9 #### Extreme Light

[42]:
```
f = "pictures/extreme_light_rgb.jpg"
mode = "RGB"
Image.open(f).convert(mode)
```

[42]:

[43]:
```
get_new_image(hist_eq_rgb(f, mode), mode)
```

[43]:

[44]:
```
mode = "HSV"
get_new_image(hist_eq_hsv(f, mode), mode).show()
```

[45]:
```
get_new_image(hist_eq_rgb(f, mode), mode).show()
```

- Display the original, the gradient in x and y directions, and the gradient magnitude image.
- Apply a simple threshold to the gradient magnitude image and display the gradient magnitude and the thresholding image. You may use 100 as you threshold value or any other appropriate value.

To compute the x, y gradients of the grayscale image, I convolve the images with well known filter operators, like the Sobel and Scharr filters. I implemented a naive convolution function, but in pure Python it was very slow. Using Cython, I added static types, kept types stable, and integrated the high performance arrays from numpy. I tested with OpenMP as well, but observed little difference. Cython is well supported in Jupyter environments.

[47]:
```
%load_ext Cython
```

[104]:
```
%%cython -a -3 --link-args=-fopenmp -lm -c=-O3 -c=-ffast-math -c=-march=native

cimport cython
from cython.parallel cimport prange, parallel
cimport openmp
```

```
from libc.math cimport sqrt, round
import numpy as np
cimport numpy as np

DTYPE = np.int64
ctypedef np.int64_t DTYPE_t
DTYPE8 = np.uint8
ctypedef np.uint8_t DTYPE8_t

@cython.boundscheck(False)
@cython.wraparound(False)
cpdef convolve(np.ndarray[DTYPE8_t, ndim=2] img, np.ndarray[DTYPE_t, ndim=2]␣
↪window, DTYPE8_t thr, bint norm):
    cdef int x, y, s, t, v, w, i, wx_from, wx_to, wy_from, wy_to, lsum, sos
    cdef unsigned char ctr, idx, sd, mean, sqd
    cdef float var
    cdef int imgx = img.shape[0]
    cdef int imgy = img.shape[1]
    cdef int winx = window.shape[0]
    cdef int winy = window.shape[1]
    cdef int wxmid = winx // 2
    cdef int wymid = winy // 2
    cdef int outx = imgx + 2 * wxmid
    cdef int outy = imgy + 2 * wymid

    cdef DTYPE8_t value
    cdef np.ndarray[DTYPE8_t, ndim=1] neighborhood = np.zeros([winx*winy],␣
↪dtype=DTYPE8)
    cdef np.ndarray[DTYPE8_t, ndim=2] out = np.zeros([outx, outy], dtype=DTYPE8)

    with nogil, parallel(num_threads=8):
        for x in prange(outx, schedule='guided'):
            for y in range(outy):
                wx_from = max(wxmid - x, -wxmid)
                wx_to = min((outx - x) - wxmid, wxmid + 1)
                wy_from = max(wymid - y, -wymid)
                wy_to = min((outy - y) - wymid, wymid + 1)
                value = 0
                ctr = 0
                lsum = 0
                for s in range(wx_from, wx_to):
                    for t in range(wy_from, wy_to):
                        v = x - wxmid + s
                        w = y - wymid + t
                        if norm:
                            neighborhood[ctr] = img[v, w]
                            ctr = ctr + 1
```

```
                            lsum = lsum + img[v, w]
                        value = value + window[wxmid - s, wymid - t] * img[v, w]
                        # Abs
                        if value < 0:
                            value = -value
                        # Threshold
                        if thr > 0 and value < thr:
                            value = img[v, w]
                # Local normalization
                if norm:
                    idx = (wx_to - wx_from) * (wy_to - wy_from)
                    mean = lsum // idx
                    sd = 0
                    sos = 0
                    for i in range(idx):
                        if neighborhood[i] == 0:
                            continue
                        sqd = (neighborhood[i] - mean) * (neighborhood[i] -␣
↪mean)
                        sos = sos + sqd
                        neighborhood[i] = 0
                    var = sos / idx
                    sd = <unsigned char>round(sqrt(var))
                    if mean != 0:
                        value = <unsigned char>((value-sd) // mean)
                out[x, y] = value
    return out
```

[104]: <IPython.core.display.HTML object>

I experimented with several different operators: Sobel, Prewitt, Scharr, Roberts, basic [1, -1], [-1, 1], and a so-called `optimal' Scharr operator

[98]:
```
BASIC_X = np.array([[1],[-1]])
BASIC_Y = np.array([[-1], [1]])
ROBERTS = np.array([[1, 0], [0, -1]])
PREWITT_GX = np.array([[1, 0 , -1], [1, 0, -1], [1, 0, -1]])
PREWITT_GY = np.array([[1, 1, 1], [0, 0, 0], [-1, -1, -1]])
SOBEL_GX = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])
SOBEL_GY = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]])
SCHARR_GX = np.array([[3, 10, 3], [0, 0, 0], [-3, -10, -3]])
SCHARR_GY = np.array([[3, 0, -3], [10, 0, -10], [3, 0, -3]])
OPTIMAL_GX = np.array([[47, 162, 47], [0, 0, 0], [-47, -162, -47]])
OPTIMAL_GY = np.array([[47, 0, -47], [162, 0, -162], [47, 0, -47]])
```

I implemented a threshold in the magnitude, but also in the convolution function itself, which produced some interesting results. Additionally, I attempted to locally normalize the value within the window function, which also produced some

interesting results. The command line version of this program has many options that can be combined in different ways. In the following examples, I will only use the Sobel filter. You can try out the others by running images.py at the command line and using the -o option. images.py -h has some description of all the switches and options.

[99]:
```
def gmag(gx, gy, threshold=0):
    row, col = gx.shape
    out = np.zeros_like(gx)
    timg = np.zeros_like(gx)
    for i in range(row):
        for j in range(col):
            val = sqrt(gx[i, j]**2 + gy[i, j]**2)
            if val > threshold:
                out[i, j] = val
                timg[i, j] = 255
            else:
                out[i, j] = 0
                timg[i, j] = 0
    return out, timg
```

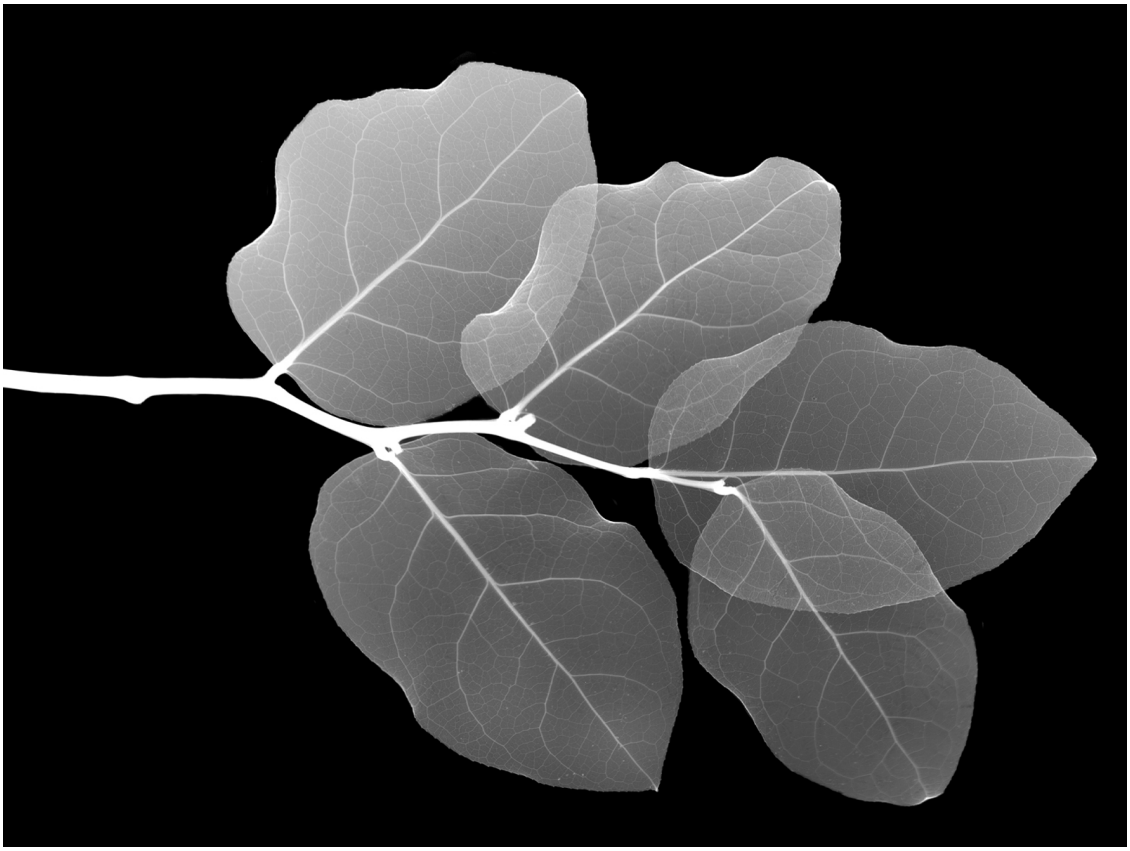
[100]:
```
def sobel(indata, threshold=0, norm=0):
    "Pass as an argument to gradient_images"
    sx = convolve(indata, SOBEL_GX, threshold, norm)
    sy = convolve(indata, SOBEL_GY, threshold, norm)
    return sx, sy

def gradient_images(indata, kernel, threshold=0, norm=0, m=False):
    """Returs x, y, mag gradient images. Accepts image data and an operator␣
 ↪function"""
    if m:
        x, y = kernel(indata, 0, norm)
        mag = gmag(x, y, threshold)
    else:
        x, y = kernel(indata, threshold, norm)
        mag = gmag(x, y, threshold)
    return (get_new_image(x, "L"),
            get_new_image(y, "L"),
            get_new_image(mag[0], "L"),
            get_new_image(mag[1], "L"))
```

0.10 ### Image Gradients

[101]:
```
# leaf.jpg - extreme contrast
mode = "L"
indata = get_image_data("pictures/leaf.jpg", mode)
Image.open("./pictures/leaf.jpg").convert(mode)
```

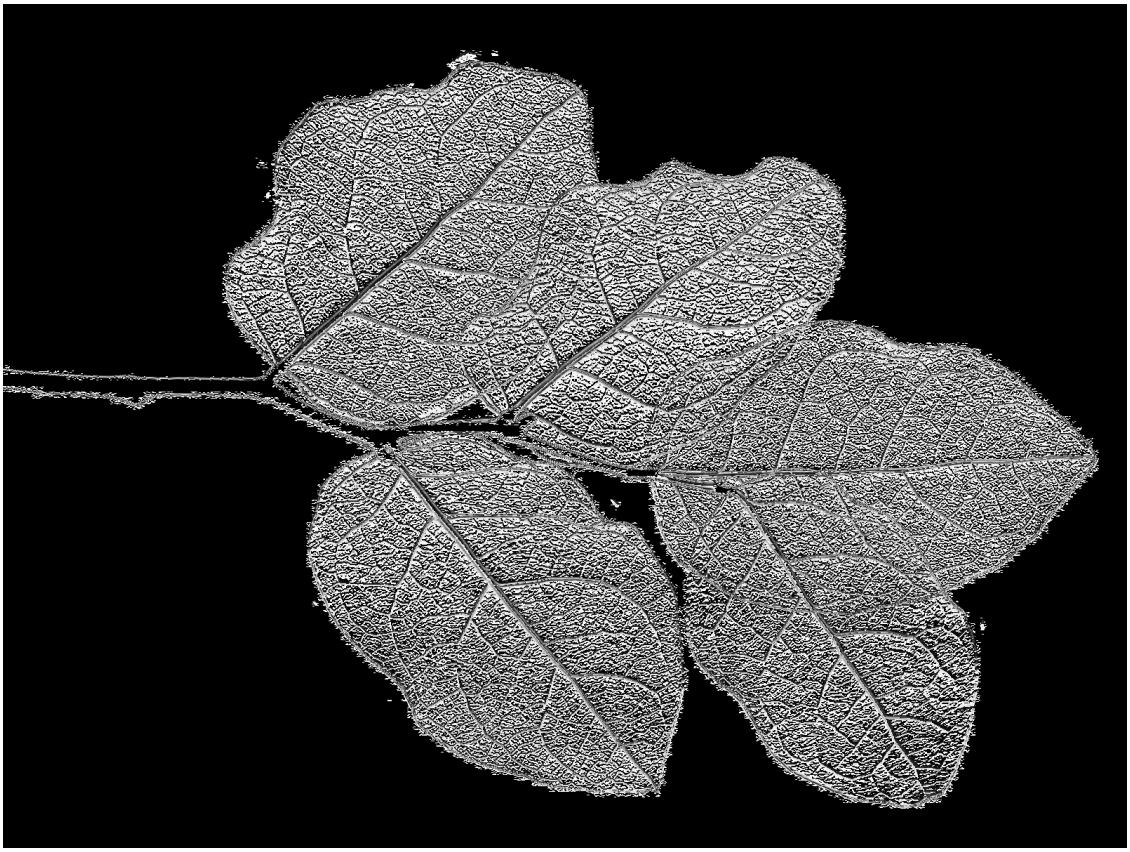
[101]:



[105]:
```
# Simple Gradients, Sobel operator, no threshold, no normalization
x, y, mag, thr = gradient_images(indata, sobel)
```

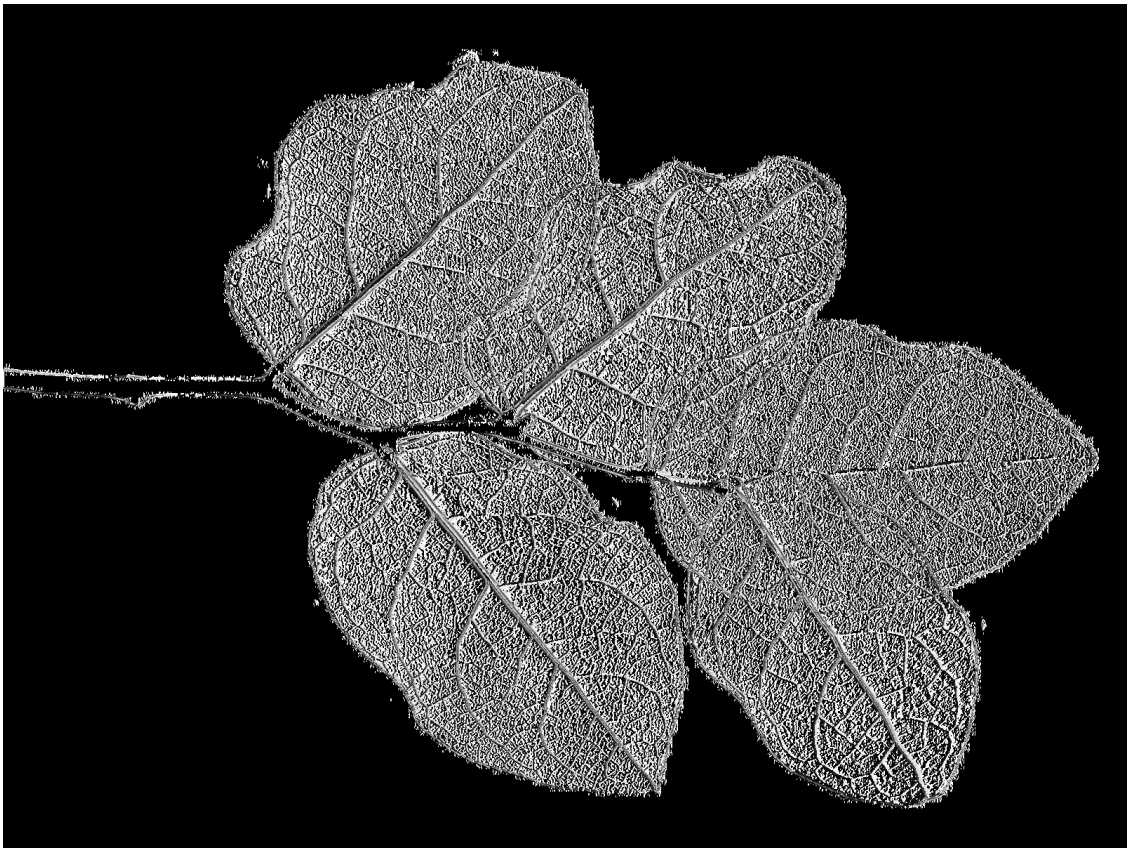
[106]:
```
x
```

[106]:



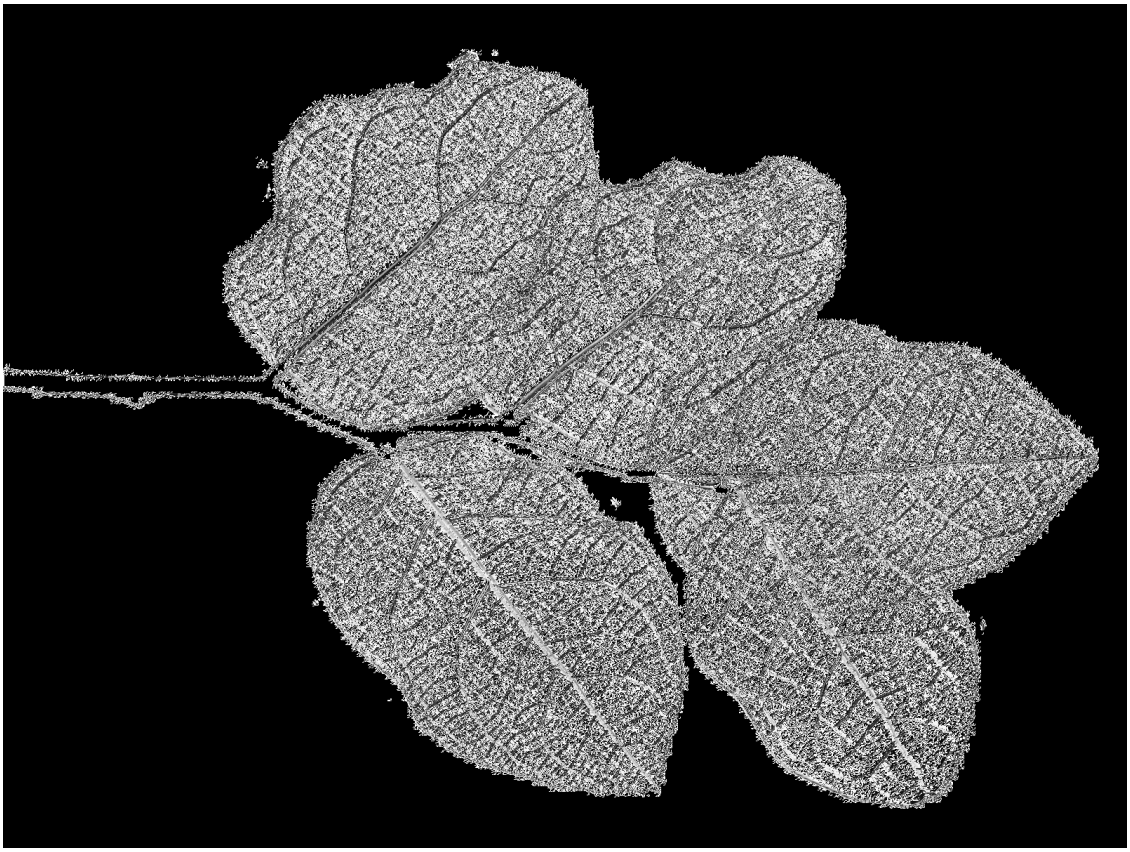
[107]:
```
y
```

[107]:



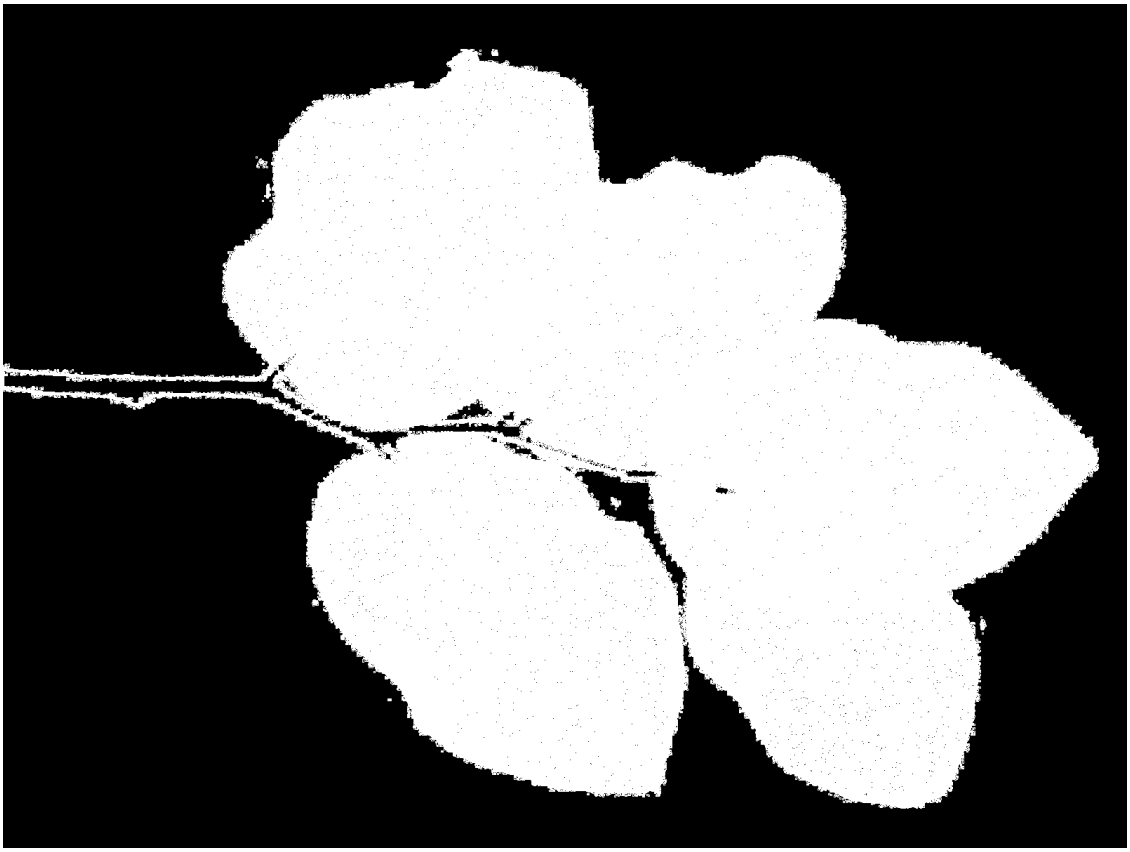
[109]:
```
mag
```

[109]:



[110]:
```
thr
```

[110]:

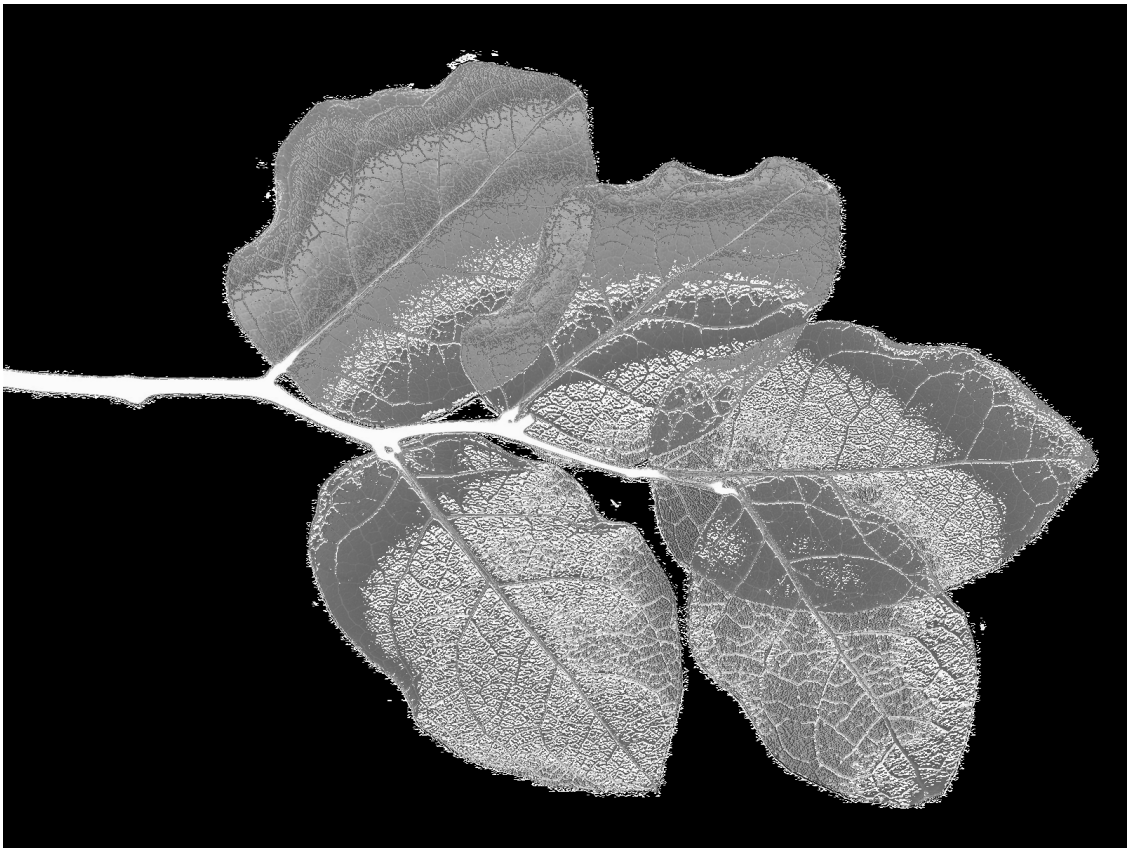


0.11 ### Apply Threshold

[111]:
```
x, y, mag, thr = gradient_images(indata, sobel, threshold=100)
```

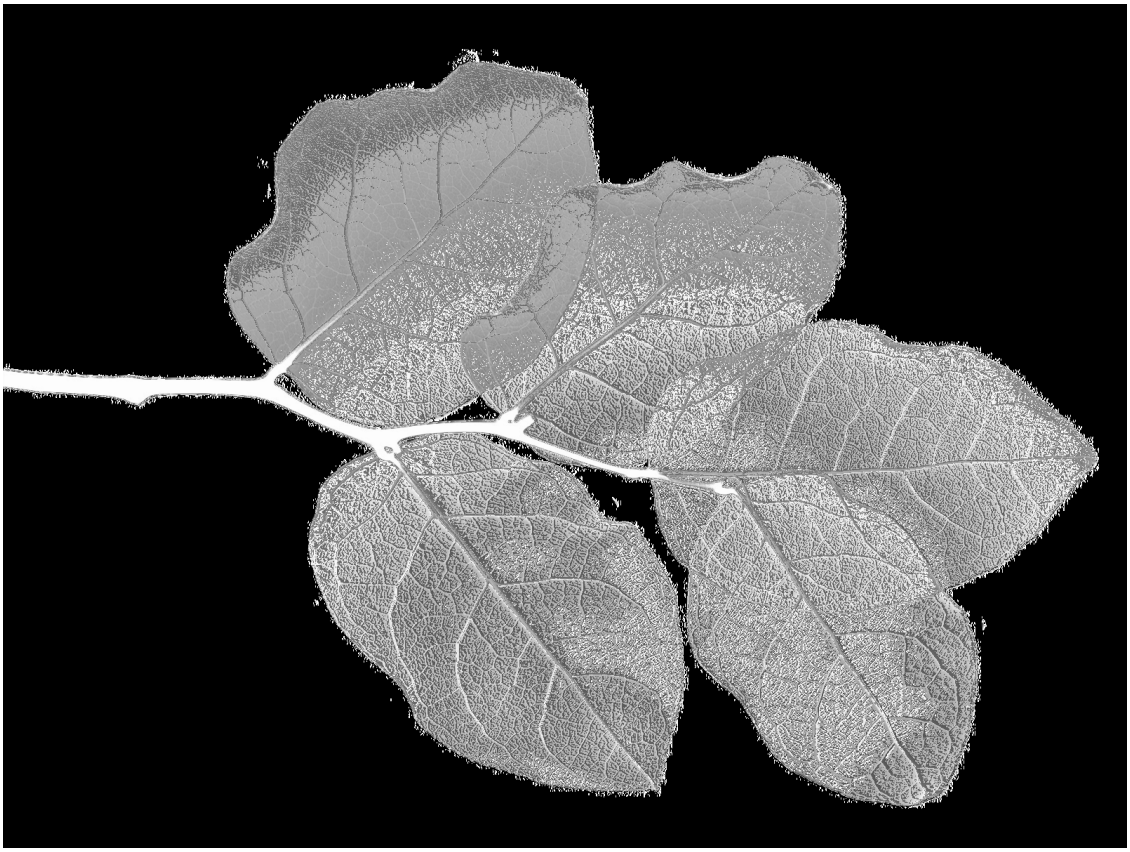
[112]:
```
x
```

[112]:



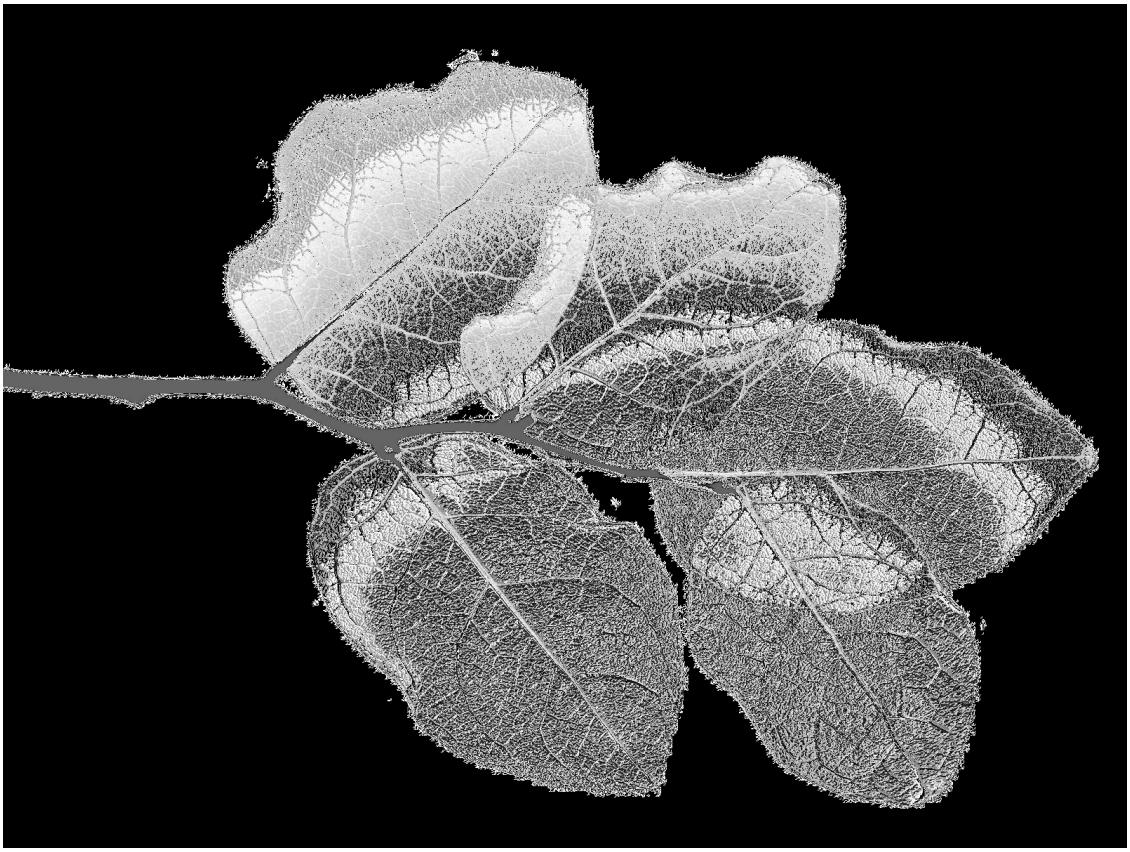
[113]:
```
y
```

[113]:



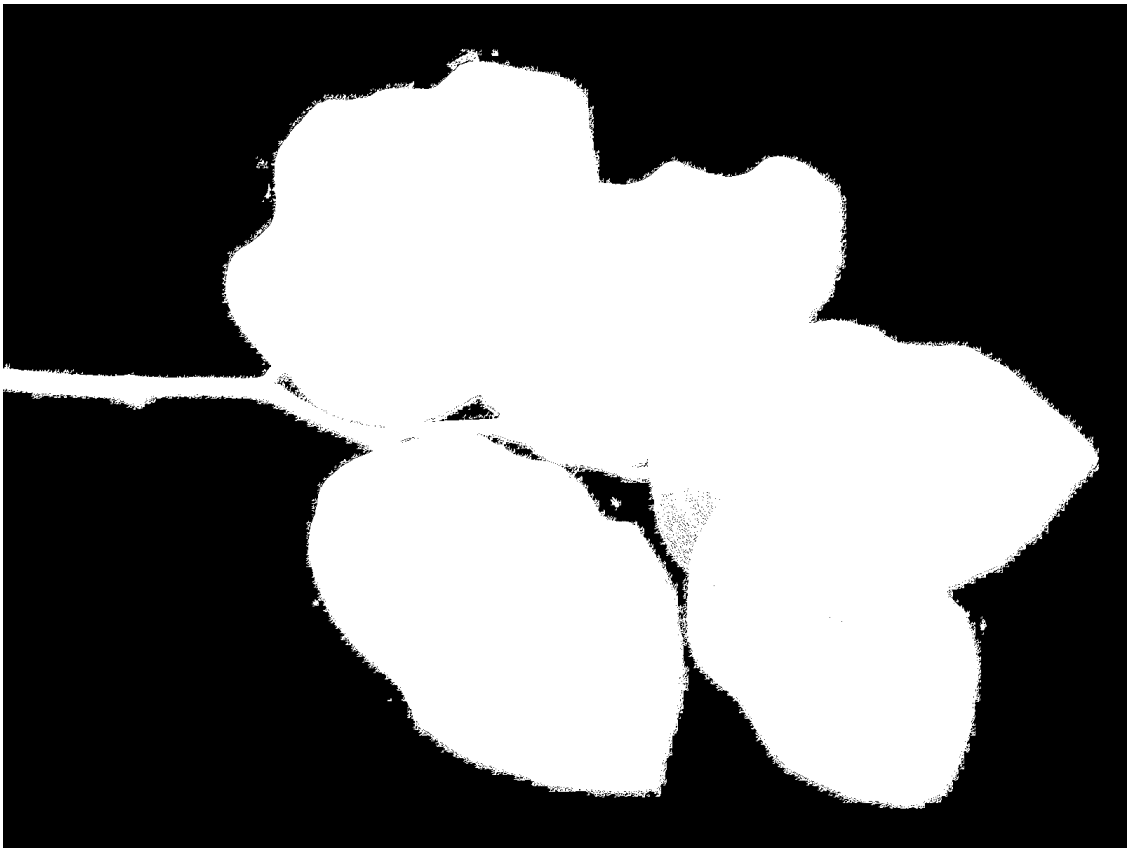
[114]:
```
mag
```

[114]:



[115]:
```
thr
```

[115]:

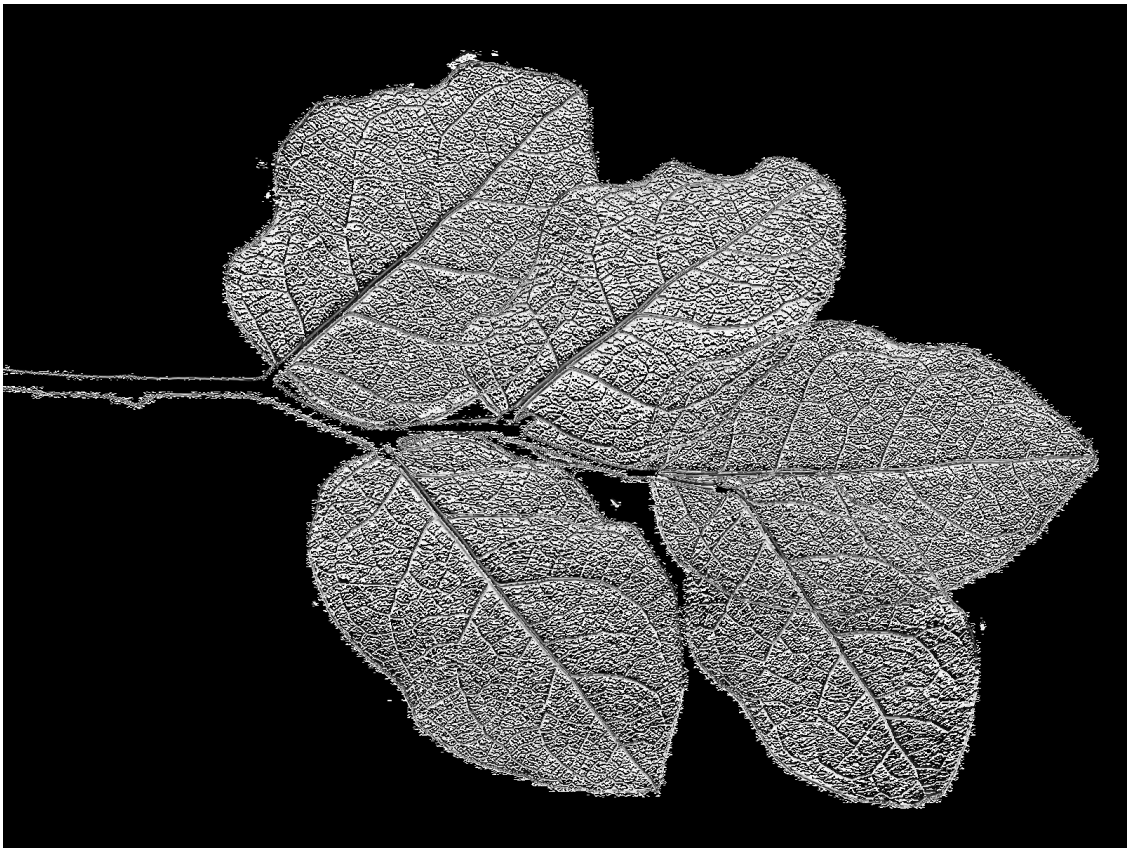


0.12 ### Threshold on Magnitude Only

[116]:
```
x, y, mag, thr = gradient_images(indata, sobel, threshold=100, m=True)
```

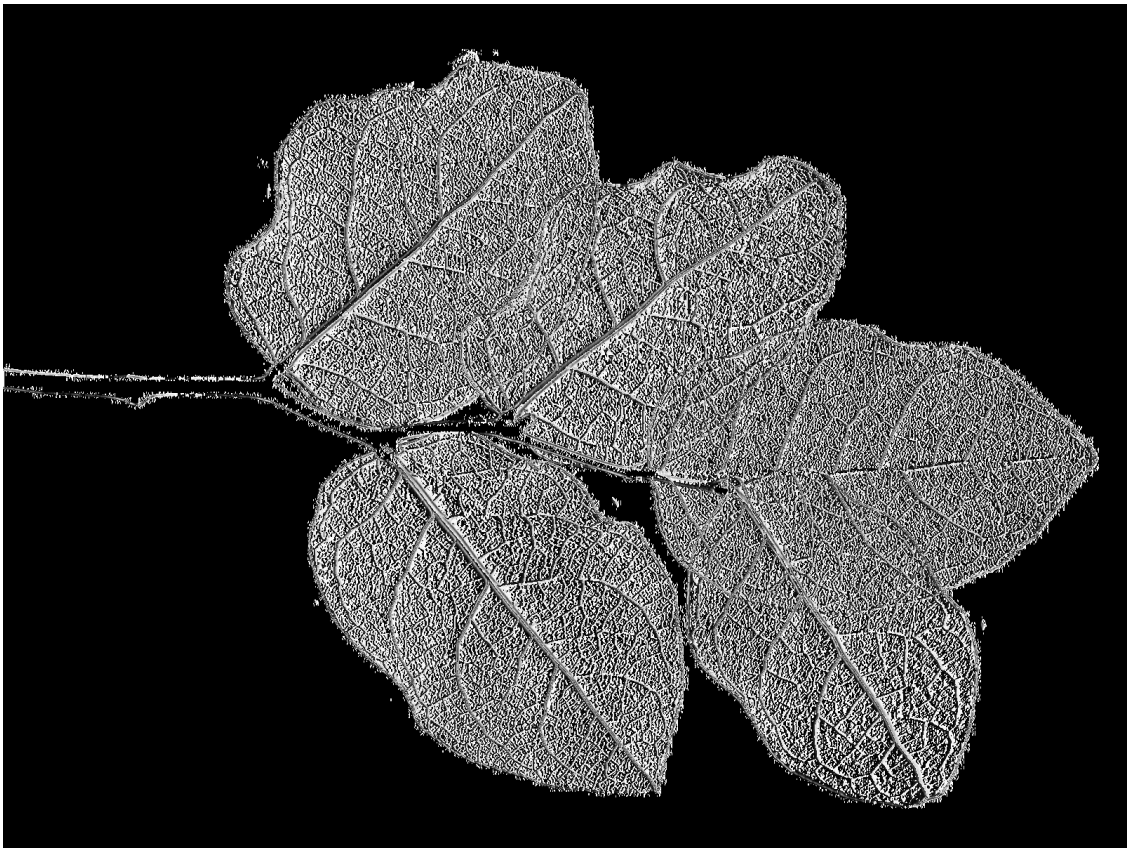
[117]:
```
x
```

[117]:



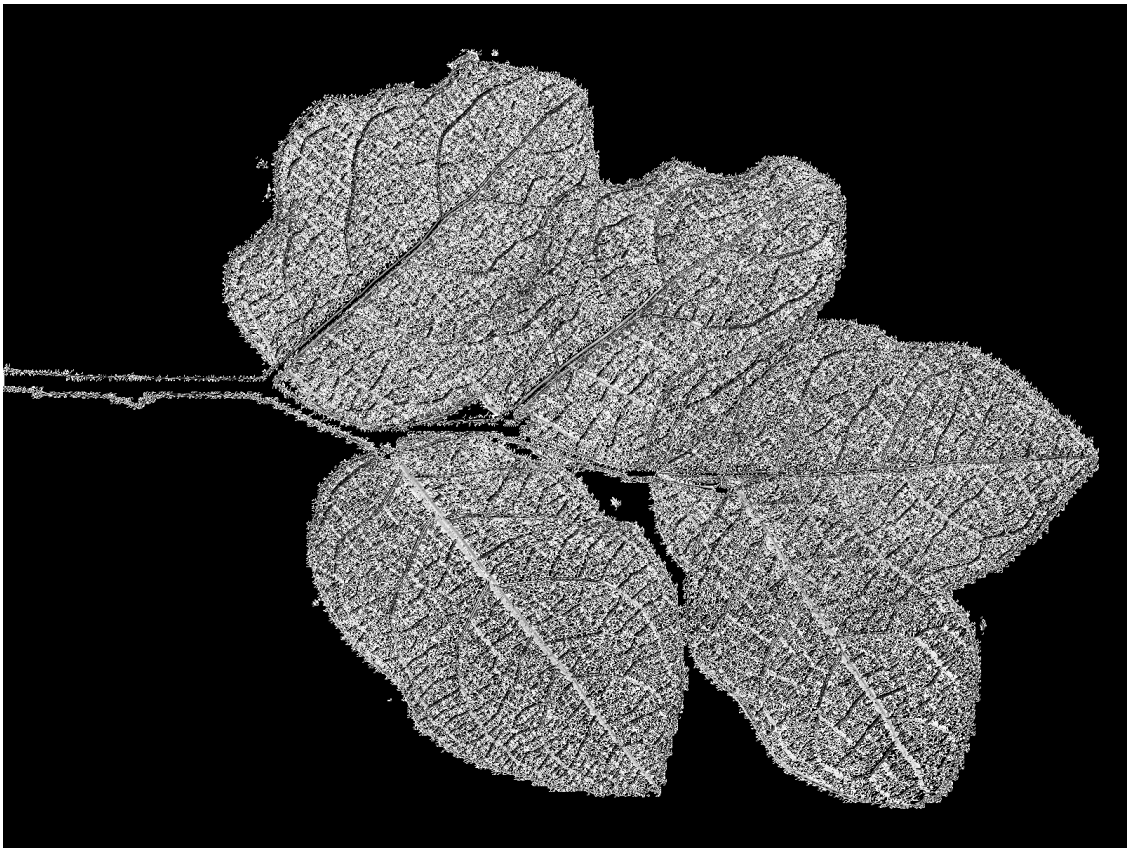
[118]:
```
y
```

[118]:



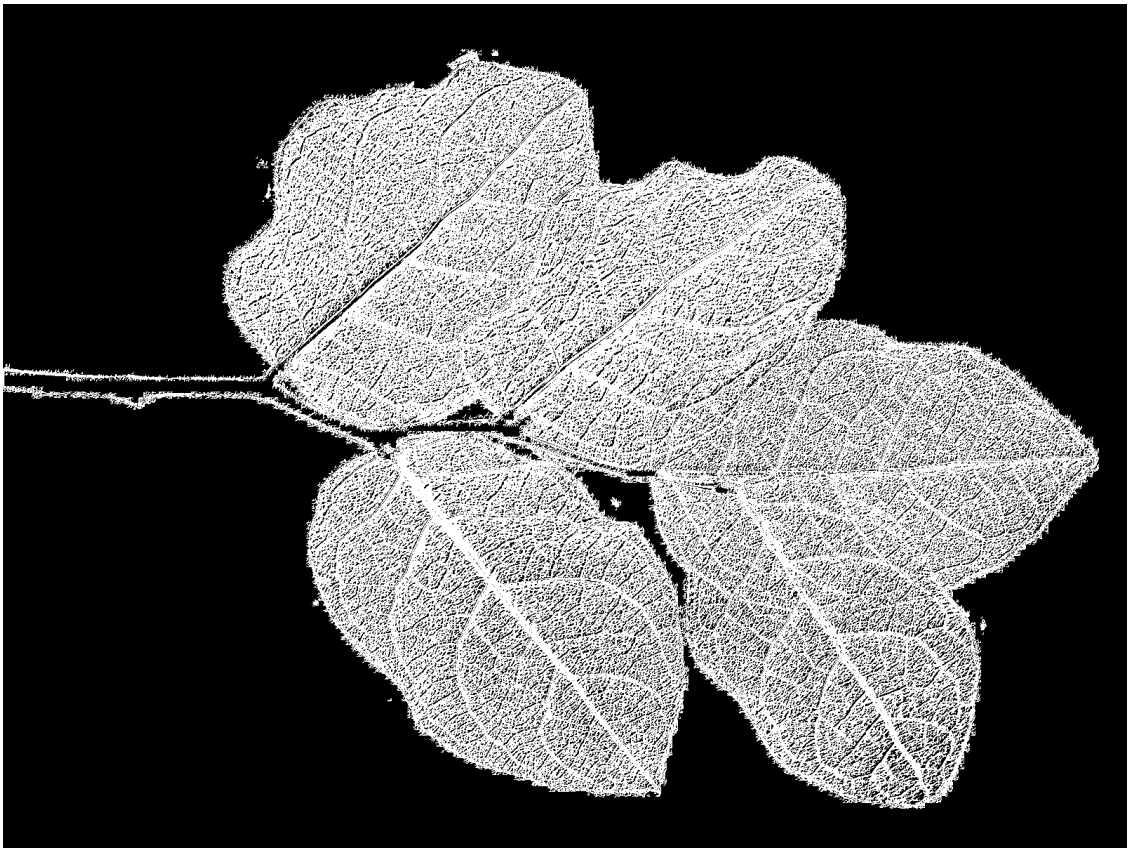
[119]:
```
mag
```

[119]:



[120]:
```
thr
```

[120]:

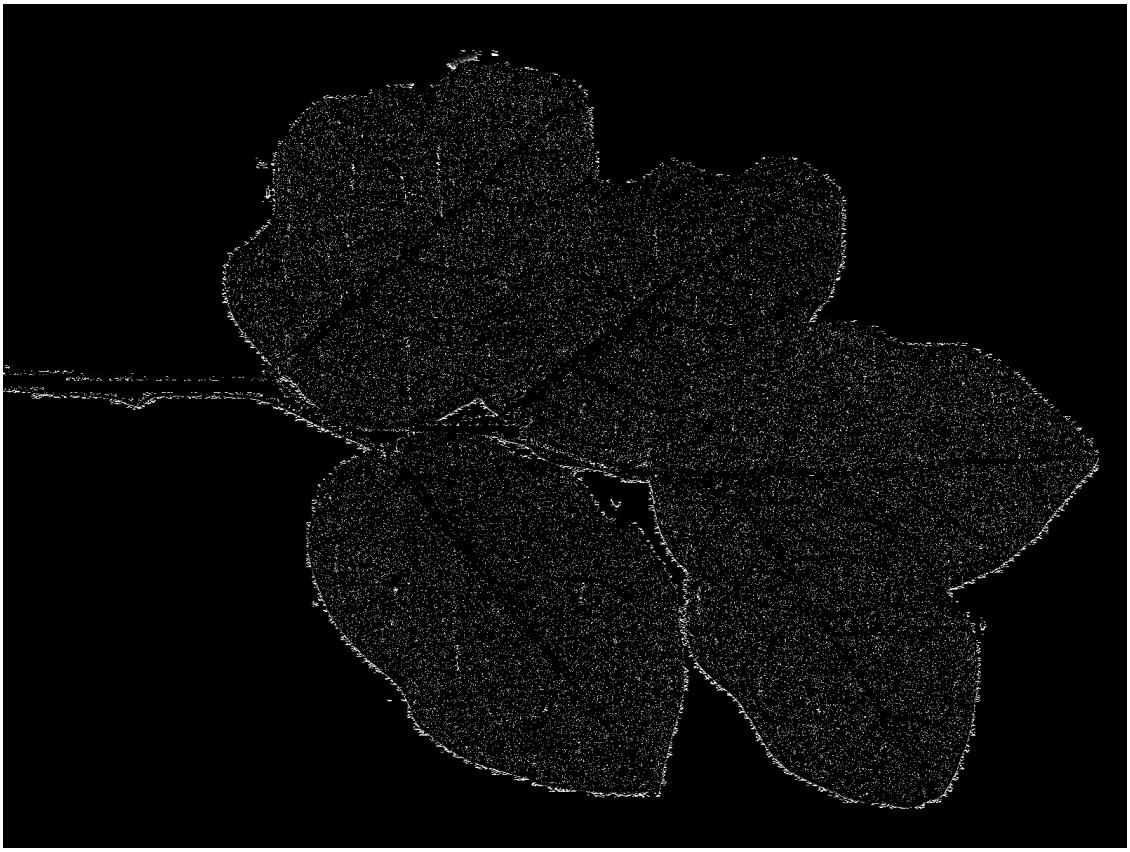


0.13 ### Local Normalization, no threshold

[123]:
```
x, y, mag, thr = gradient_images(indata, sobel, 0, 1)
```

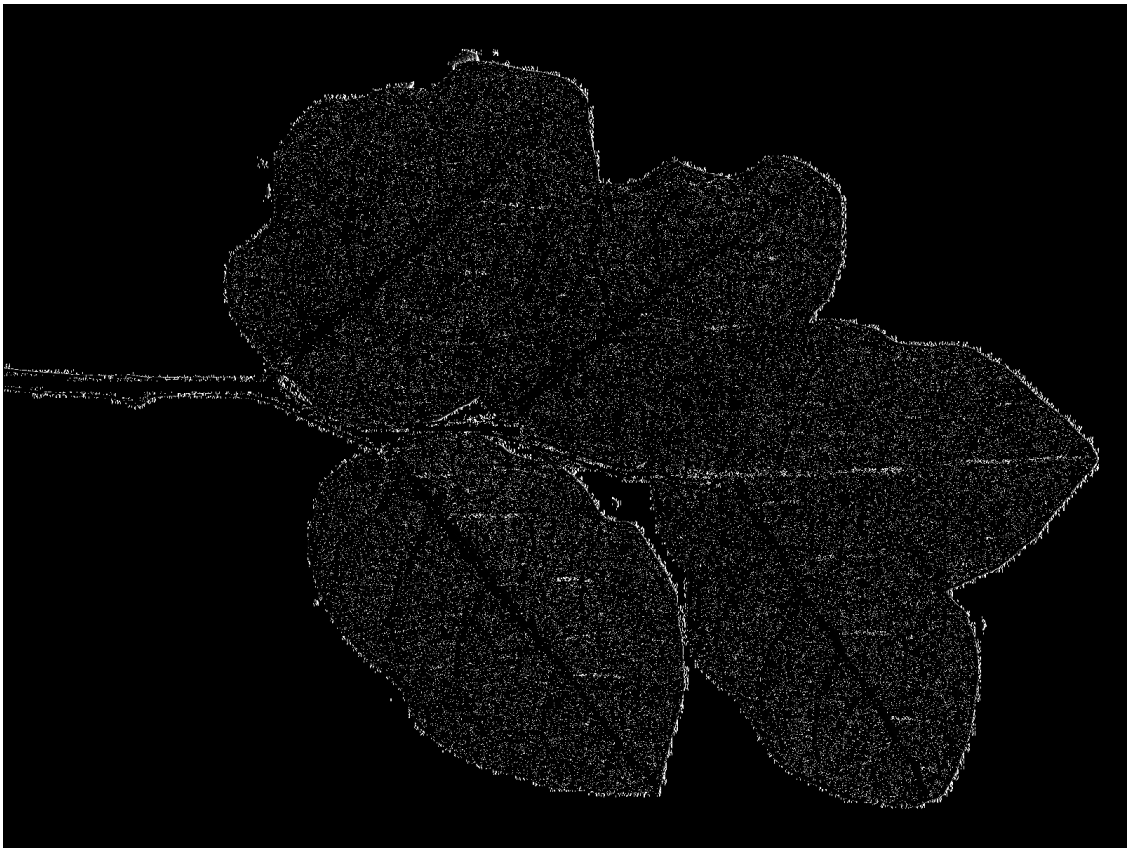
[124]:
```
x
```

[124]:



[125]:
```
y
```

[125]:

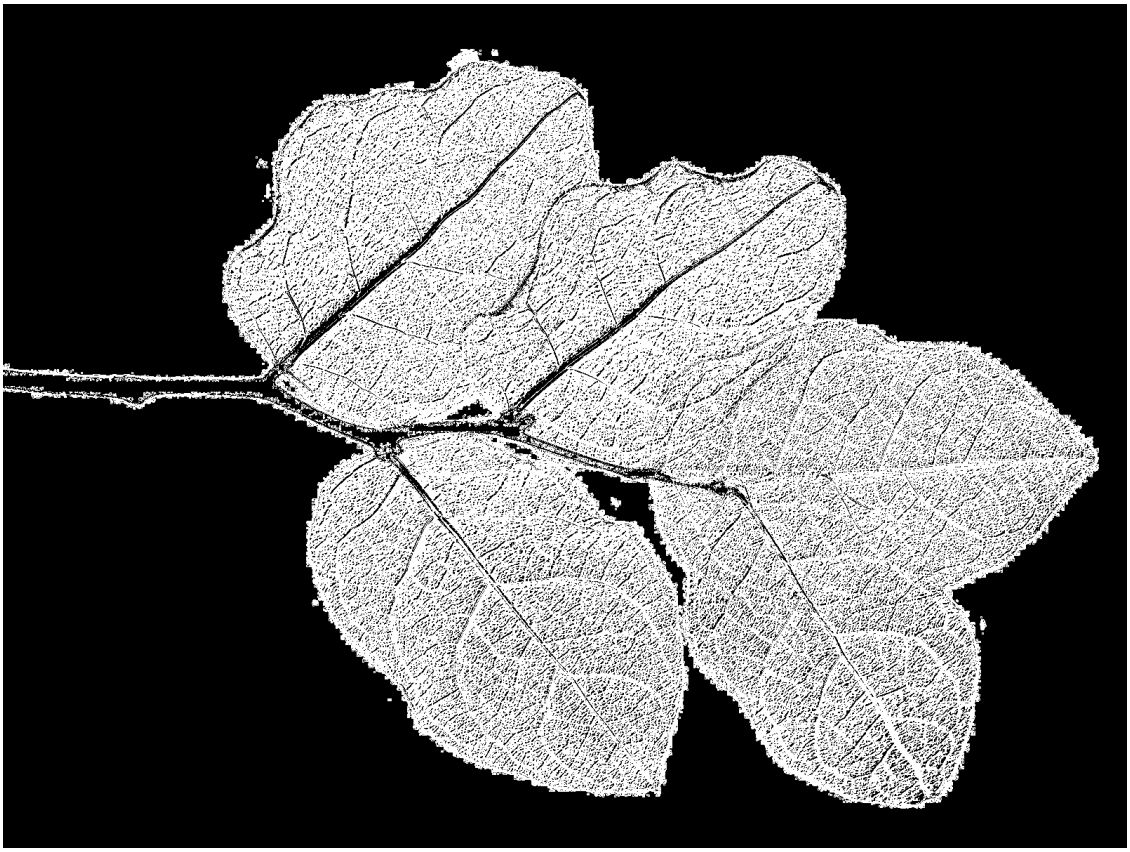


[126]:
```
mag
```

[126]:

[127]:
```
thr
```

[127]:

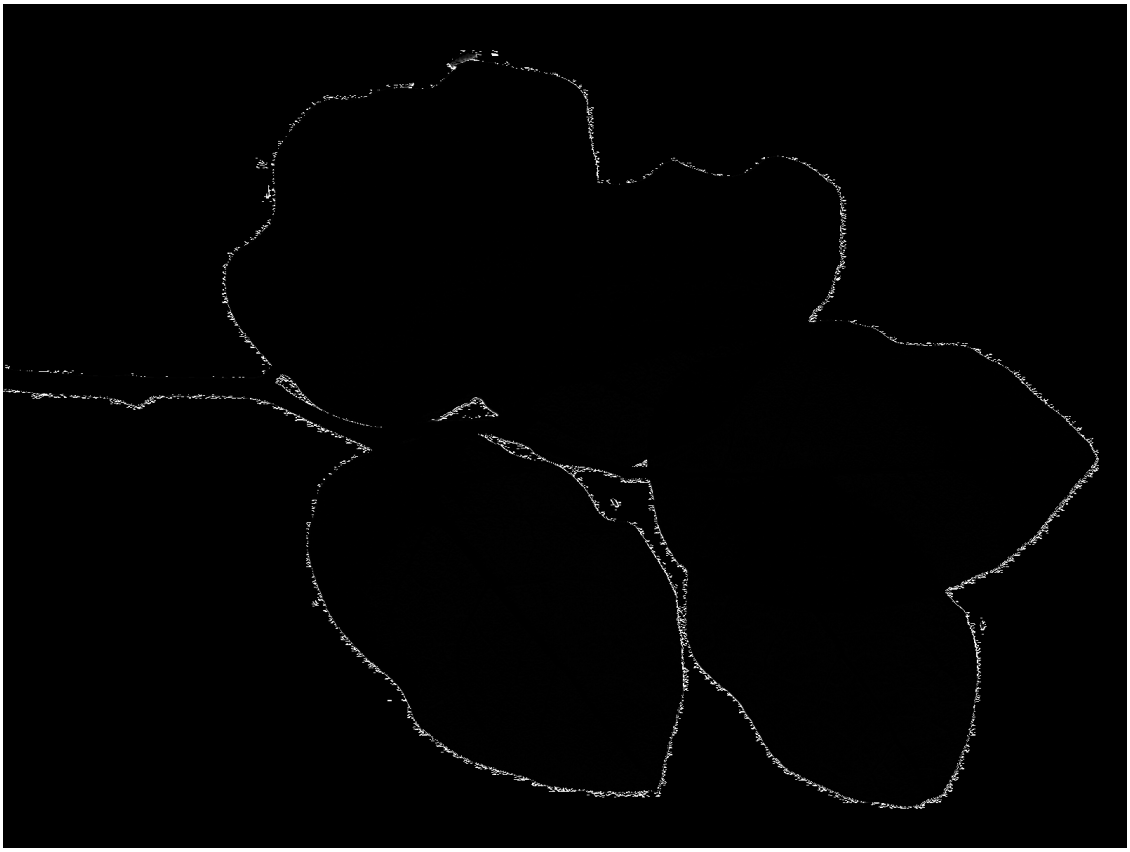


0.14 ### Local normalization, Threshold

[130]:
```
x, y, mag, thr = gradient_images(indata, sobel, 100, 1)
```

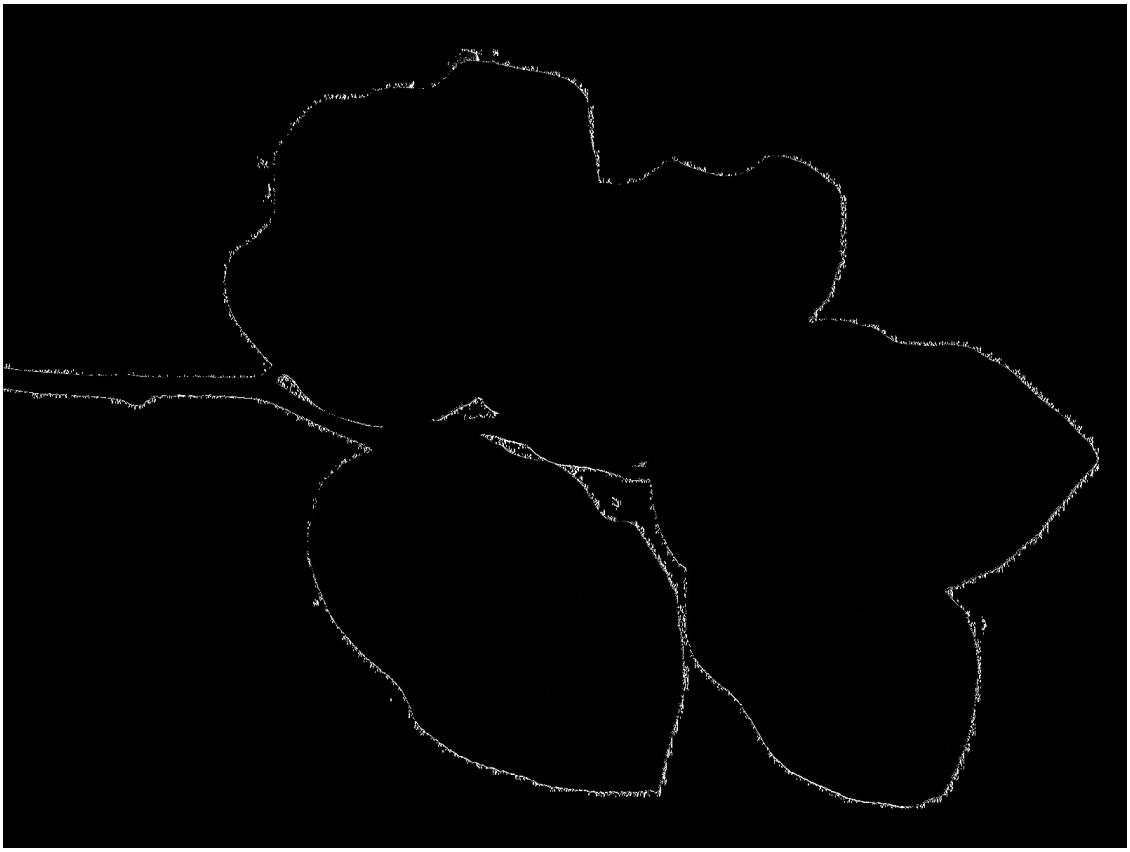
[131]:
```
x
```

[131]:



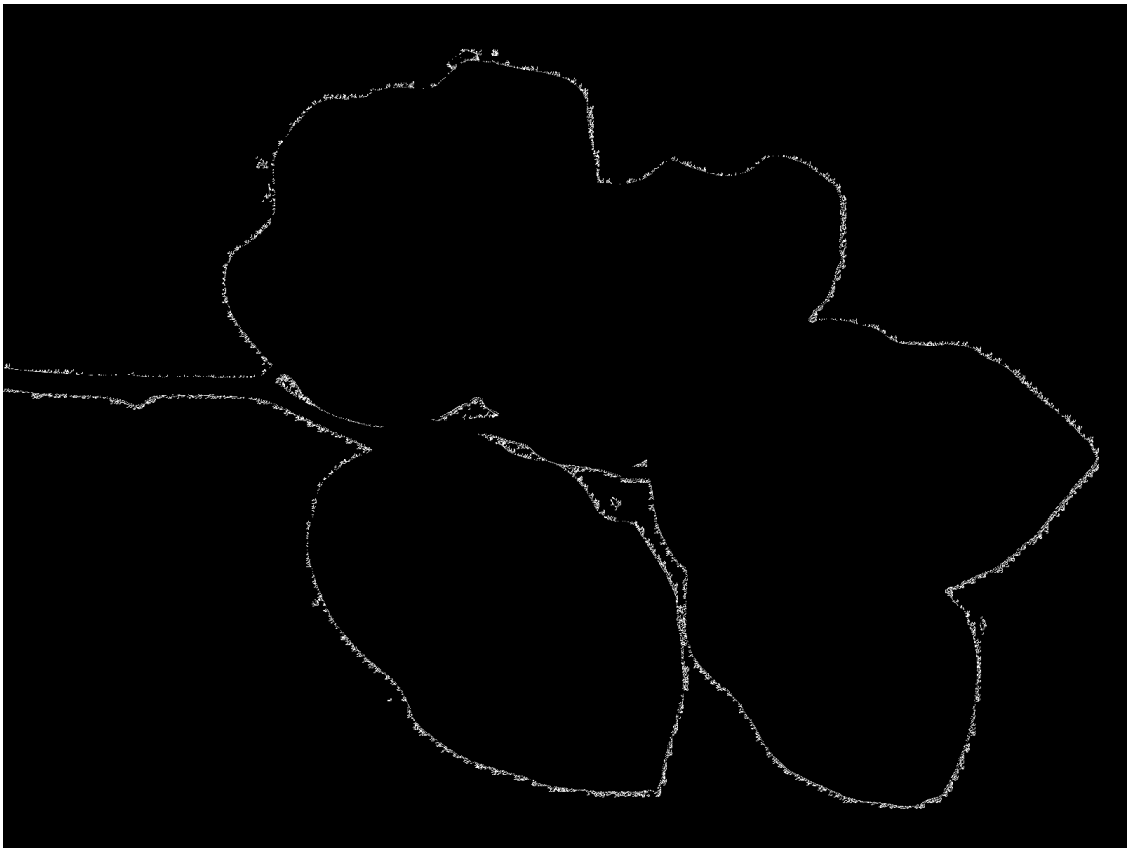
[132]:
```
y
```

[132]:



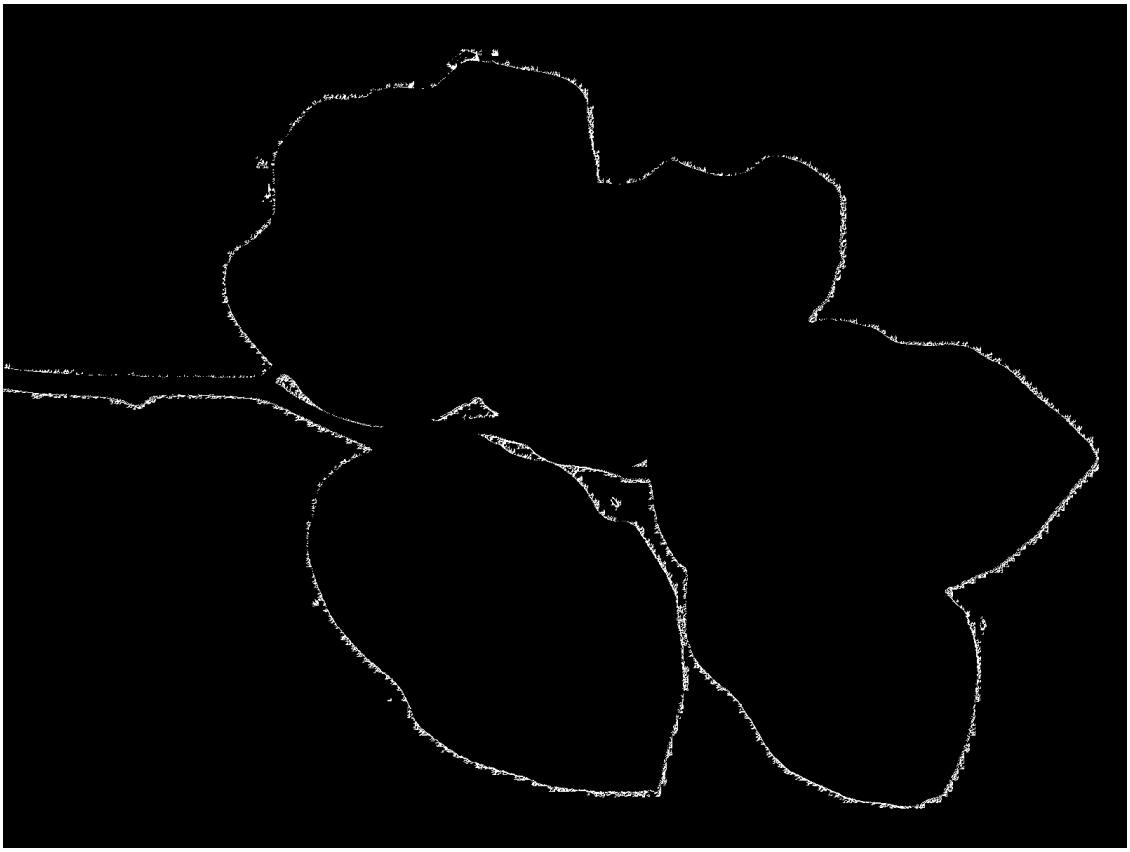
[133]:
```
mag
```

[133]:



[134]:
```
thr
```

[134]:

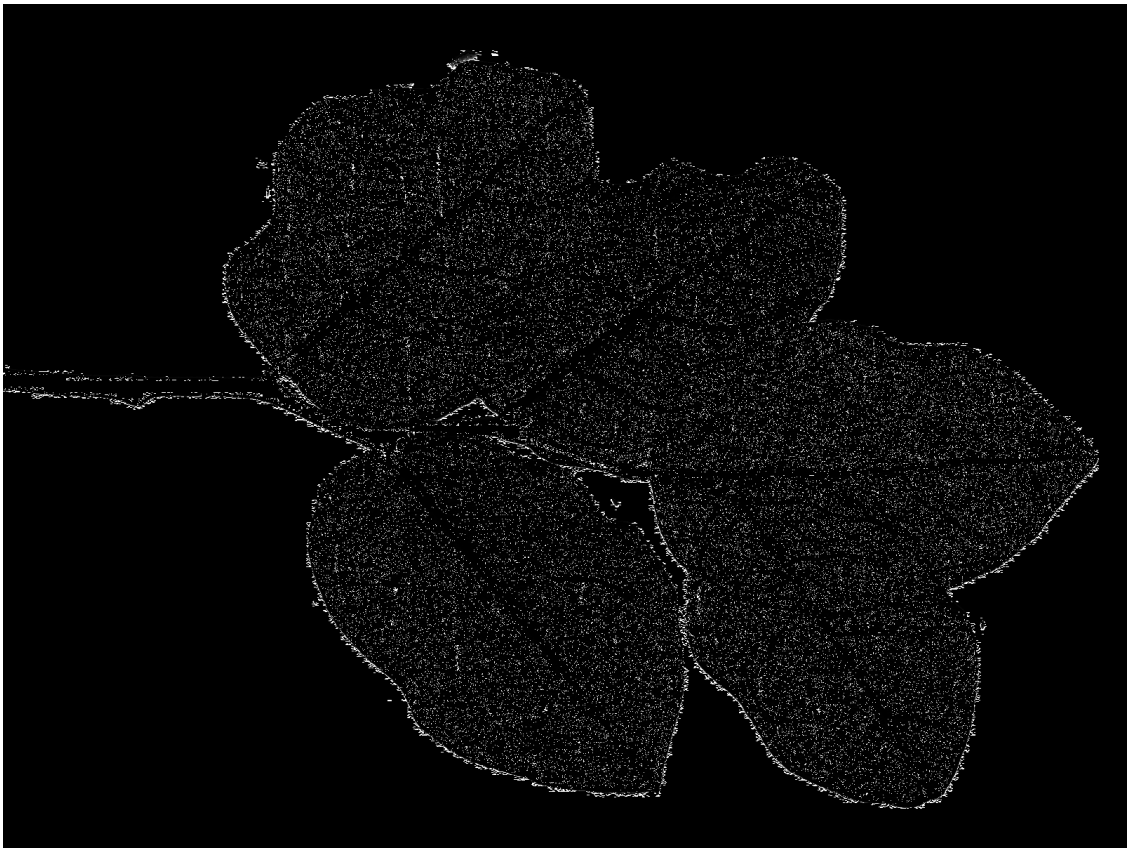


0.15 ### Local Normalization, Threshold on Magnitude only

[135]:
```
x, y, mag, thr = gradient_images(indata, sobel, 100, 1, True)
```

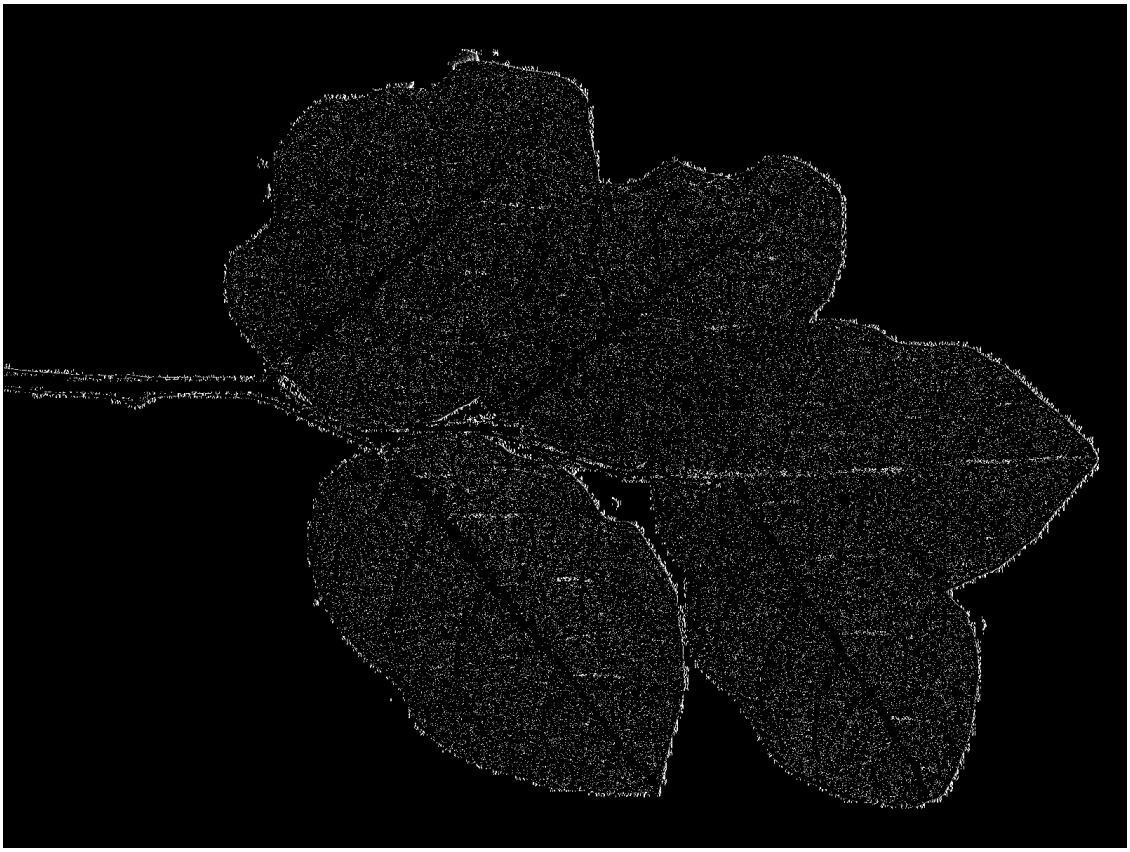
[136]:
```
x
```

[136]:



[137]:
```
y
```

[137]:



[138]:
```
mag
```

[138]:

[139]:
```
thr
```

[139]: