Assignment 2
Discrete Fourier Transform

Lucas David, 188972
lucas.david@ic.unicamp.br

1 Introduction

This assignment was submitted to the class of 2021/1 of course Introduction to Image Processing (MO443) at Universidade Estadual de Campinas. Its goal is to apply the Discrete Fourier Transform over images using Python programming language and assess its results.

1.1 Dataset and Setup

Very much like in Assignment 1, I employed TensorFlow and Google Colaboratory to develop this assignment. The notebook produced is available for direct access[1]. Additionally, 10 random images from the TF-Flowers[2] dataset were used to illustrate the results.

Figure 1: Examples in the TF-Flowers dataset.

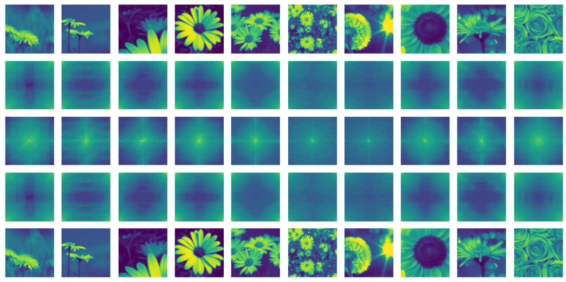


Figure 2: Illustration of the Fourier and Inverse Fourier Transformations over 2D signals (images). From top to bottom: (a) the original images; (b) the (log of) spectrum of the Fourier transform; (c) the spectrum shifted; (d) the un-shifted spectrum; and (e) the (real) result of the Inverse Fourier Transform.

2 Frequency Filtering

2.1 Identity Transform

Proposed activity: apply the Fast Fourier Transform to an image and its inverse, detailing each step.
The application of the Fourier Transformation over images is straight forward: the function *tf.signal.fft2d* can be used to represent a tensor of shape $(b_0, b_1, \dots b_n, h, w)$ in the frequency domain, where the two inner-most axes (namely, h and w) are assumed to contain the information to be transformed and the remaining are assumed to be batch dimensions. Listing 1 describes all steps involved in transforming a signal and reconstructing it its original domain.

```
x = tf.image.rgb_to_grayscale(images)

y = tf.cast(x, tf.complex64)
y = tf.signal.fft2d(y[..., 0])

s = tf.signal.fftshift(y, axes=(1, 2))
z = tf.signal.ifftshift(s, axes=(1, 2))
w = tf.signal.ifft2d(z)
```

Listing 1: FFT and IFFT over 2D signals (images).

2.2 Low-pass Filter

Low-pass filters can be implemented as binary masks that retain low frequencies of a signal while suppressing high frequencies. A vectorized ideal circular filter can be implemented as described in Listing 2. Two vectors are used in this implementation (h and w). They represent the distance between each pixel and the signal center. The h vector is reshaped into a column vector and the *radius* input vector is reshaped into $(r, 1, 1)$, where r is the number of radii passed. Finally, the operation $h^2 + w^2 < \text{radius}^2$ will return a tensor of shape (r, h, w), containing all filters built.

```
def circ_filter2d(
    radius,
    shape,
    dtype=tf.float32
):
  H, W = shape[-2:]

  h = tf.abs(tf.range(H) -H//2)
  h = tf.reshape(h, (-1, 1))
  w = tf.abs(tf.range(W) -W//2)

  radius = as_absolute_length(radius, H, W)
  radius = tf.reshape(radius, (-1, 1, 1))

  return tf.cast(
    h**2 + w**2 < radius**2,
    dtype
  )
```

Listing 2: Ideal circular filter.

A Butterworth low-pass filter can be built in a similar fashion, replacing the circle equation by `1 / (1 + (h**2 + w**2) / radius**(2*n))`.

[1] Iterative report available at colab/mo-443-assignment-2
[2] TF-Flowers dataset is available at tensorflow.org/datasets/catalog/tf_flowers

Fig. 3 illustrates the steps in the low-pass filtering procedure of a signal in the frequency domain, while Fig. 4 and Fig. 5 illustrate the difference of the application of the ideal low-pass and Butterworth low-pass filters. A quality difference of the blur effect is clearly perceptible, where the Butterworth filter generates much better results for small radius arguments.

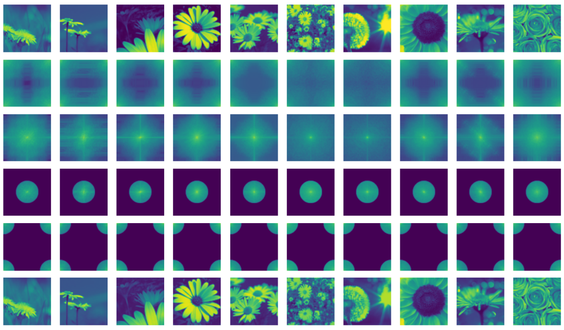


Figure 3: Illustration of steps involved in applying a low-pass filter in the frequency domain using the Fast Fourier Fourier Transformation over 2D signals (images). From top to bottom: (a) the original images; (b) the (log of) spectrum of the Fourier transform; (c) the spectrum shifted; (d) the (log of) spectrum multiplied by the ideal circular low-pass filter; (e) the un-shifted spectrum; and (f) the (real) result of the Inverse Fourier Transform.

2.3 High-pass Filter

A ideal high-pass filter can be trivially obtained from the `circ_filter2d` function described in Lst. 2, by simply subtracting its result from 1. This will effectively switch every zero in the mask by one and vice versa. Fig. 6 illustrates the application of multiple ideal high-pass filters over images. We notice textural information is removed and we are left with edge and boundary information of the original images.

2.4 Band-pass Filter

I once again leverage the `circ_filter2d` function to create the ideal band-pass filter, consisting of the subtraction of two low-pass filters with different radii. Fig. 7 illustrate the application of multiple ideal band-pass filters over images. Results are considerably harder to interpret, compared with the previous ones: filters with small inner radii and small width seem to capture very specific textural patterns; while filters with larger inner radii and large width seem to behave like high-pass filters.

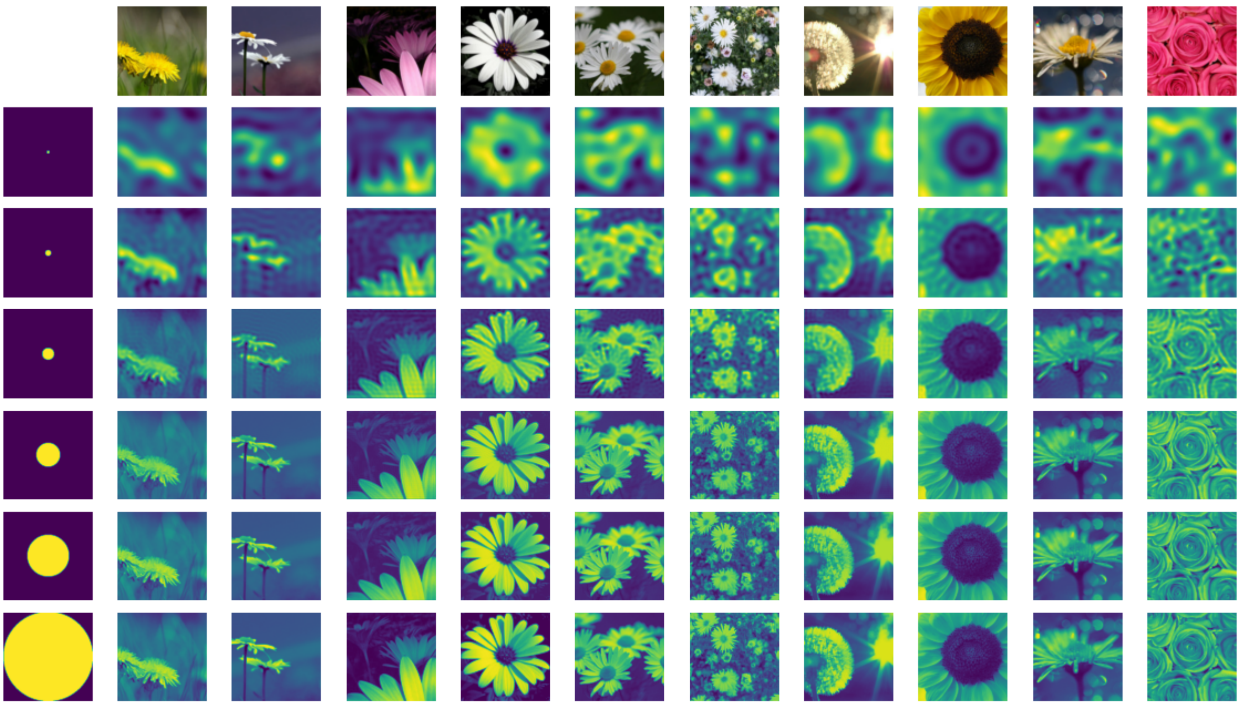


Figure 4: Result of the circular ideal filter in the frequency domain, varying its radius. The following radii values were adopted (top to bottom): 5, 10, 20, 40, 70 and 150.

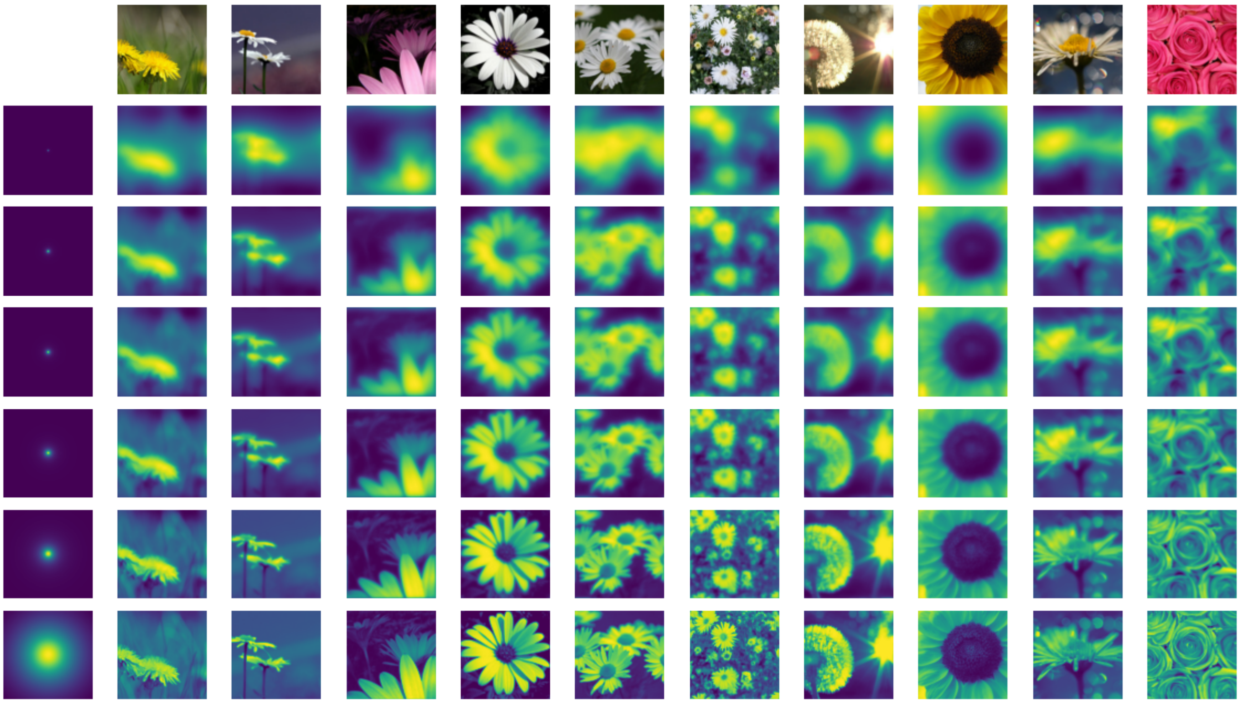


Figure 5: Result of the Butterworth low-pass filter in the frequency domain, varying its radius. The following radii values were adopted (top to bottom): 1 2%, 3%, 5%, 10% 50%.

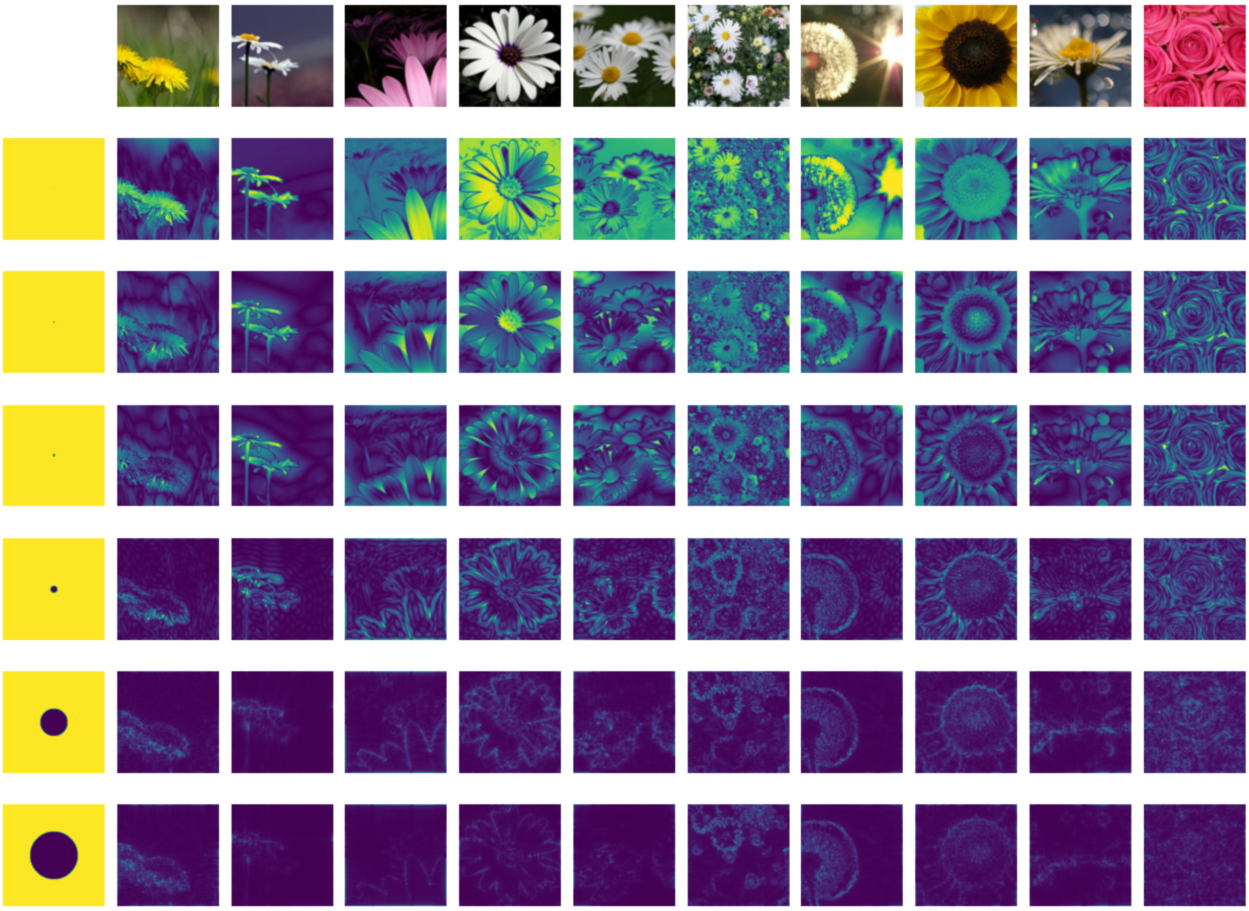


Figure 6: Result of the ideal high-pass filter in the frequency domain, varying its radius. The following radii values were adopted (top to bottom): 1, 2, 3, 10, 40 and 70.

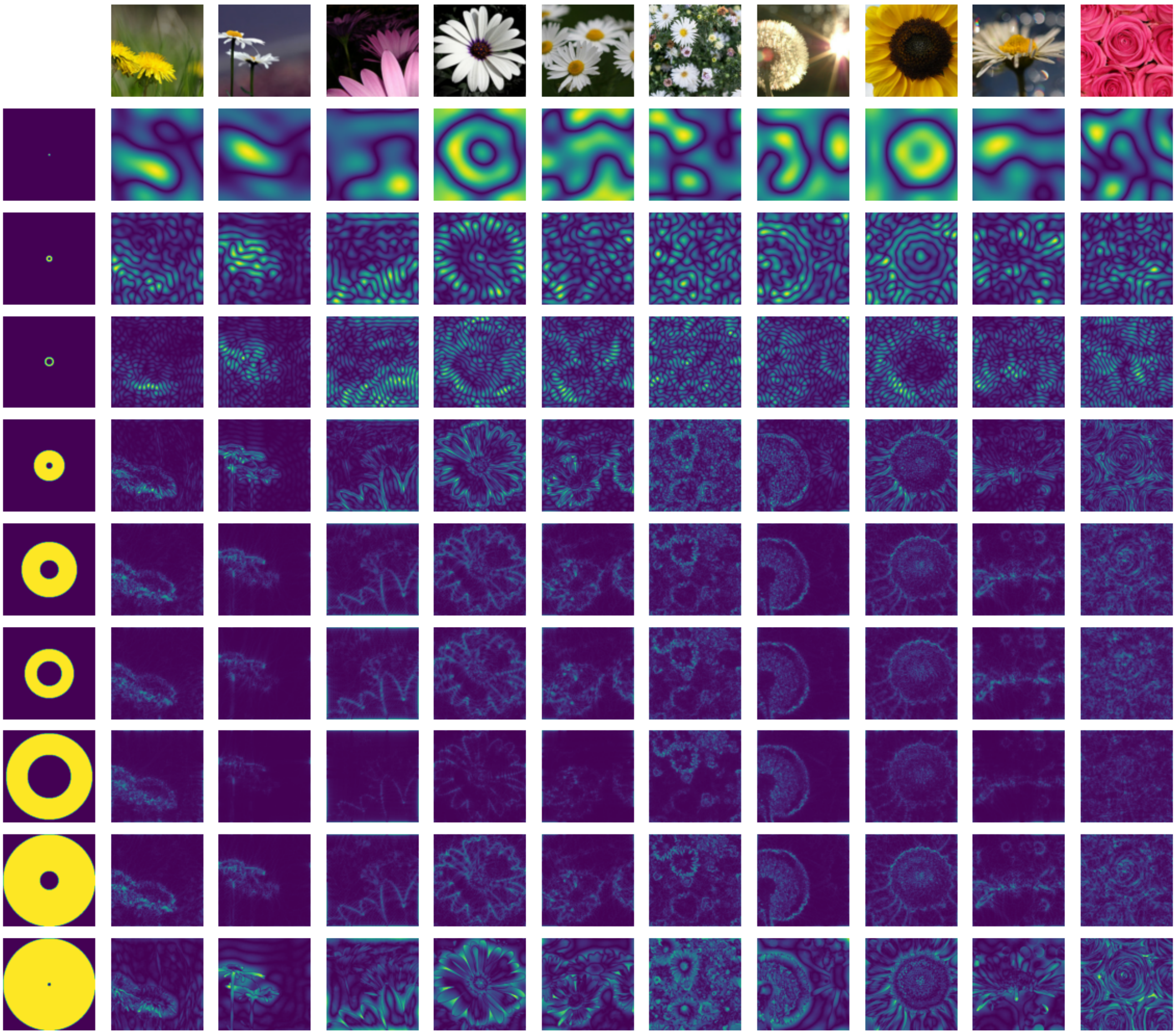


Figure 7: Result of the ideal band-pass filter in the frequency domain, varying its radius. The following (inner, outer) radii values were adopted (top to bottom): (1, 3), (5, 10), (10, 15), (10, 50), (30, 90), (40, 140), (70, 140), (30, 150) and (5, 150).

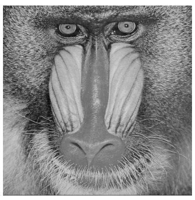
3 Image Compression Using The FFT

Compression can be performed by nullifying coefficients, the magnitude of which does not reach a certain threshold. Lst. 3 describes a compression filter based on a percentile value for the magnitude. As TensorFlow does not provide statistical functions, we are forced to recur to the *tensorflow_probability* (*tfp*) library.

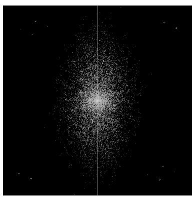
```
def compression_filter2d(rate, s):
    m = tf.abs(s)
    t = tfp.stats.percentile(m, rate, axis
        =(1, 2))

    return tf.cast(
      m > t,
      tf.complex64
    )
```

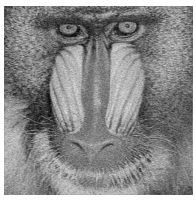
Listing 3: Compression filter based on percentile.



(a) Original



(b) Filter 1



(c) Compressed

Figure 8: From left to right: (a) the original image, (b) the compression filter with rate 95%; and (c) the compression result.

Fig. 9 illustrates the compression of the images using multiple rates. I found the quality of images compressed with a rate below 95% to be very similar of the original copy, with visible quality deterioration being most noticeable at 99.9%.

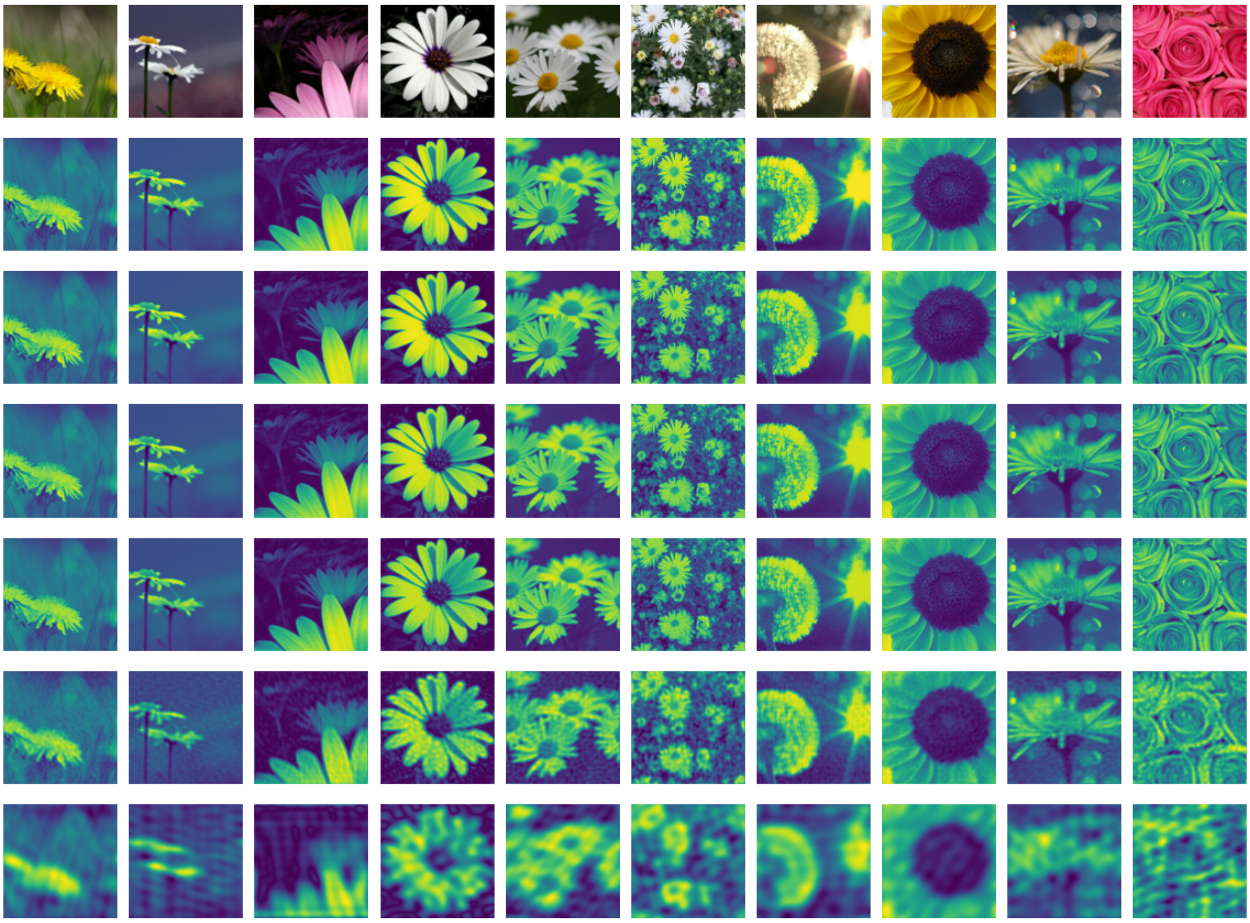


Figure 9: Result of the compression filter in the frequency domain, varying its rate. The following rate values were adopted (top to bottom): 10%, 50%, 90%, 95%, 99%, and 99.9%.

4 Effect of Rotation in the Spectrum

Fig. 10 illustrates the effect observed in the spectrum of the signal generated by the DFT of a rotated input image. The spectrum is also rotated in the same angle as the input image.

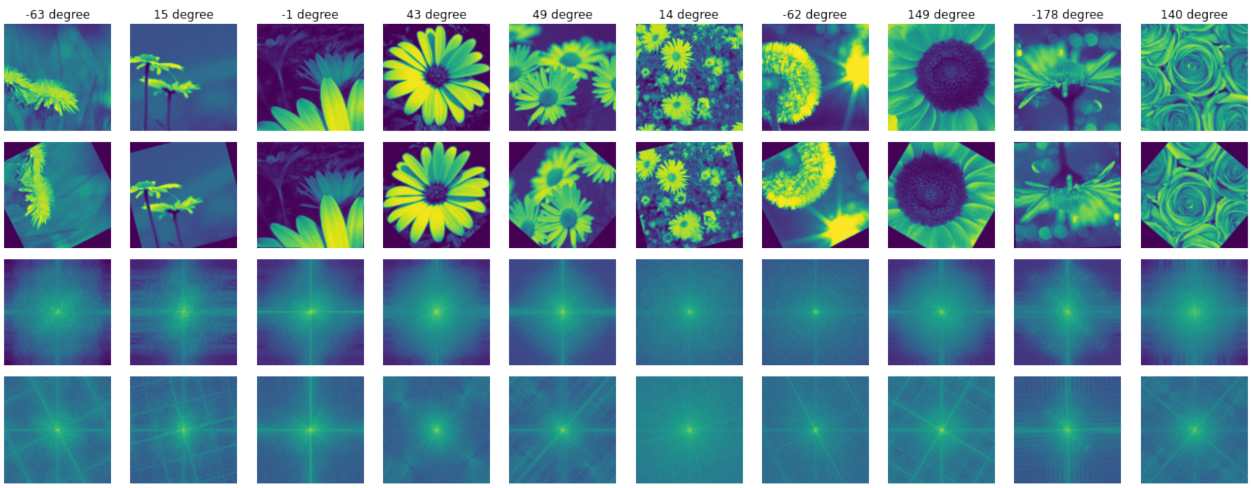


Figure 10: Effect of rotation in the spectrum of the Fourier Transformation of a signal.