Assignment 4: Discrete Fourier Transform

Guilherme Gomes Haetinger - 00274702

Proposal: *The goal of this assignment is to allow you to explore and verify several of the properties of the Discrete Fourier Transform (DFT) and its inverse that have been presented in class. The following task can be easily implemented in MATLAB, but you can use any library of your prefence to compute the DFT and its inverse.*

For this assignment, I'll be using Julia and its FFTW library!

Real and Imaginary components

```
cameraman =
```

```
cameraman = testimage("cameraman")
```

To complete the first assignment, we must separate the real and imaginary components of the image by using the fourier transform, filtering the real/imaginary components and, then, reconstructing the image with an inverse fourier transform.

We do this in the cell below (`|>` is a pipe operator).

```
(real, imaginary) = let
    reconstructed =
    real = cameraman |> fft |> reals |> ifft |> reals .|> Gray
    imaginary = cameraman |> fft |> imaginaries |> ifft |> imaginaries .|> Gray
    (real, imaginary)
end;
```

As we can see, both images seem to have a mirrored copy of itself layed on top of the originally oriented image. This can be explained by the complex conjugates in this DFT decomposition. Things get mirrored because cor every `sin/cos` pair, there is another pair with the same amplitude but different phase! The dark parts of the immaginary image are due to negative values that work towards compensating values in the real image to result in the `cameraman` image we see at the top of the report.

Image reconstruction

Below, I replicate the small image provided by the teacher using the `imresize` method.

`small_cameraman =`

```
small_cameraman = imresize(cameraman, (100, 100))
```

To manually compute `DFT` and `IDFT`, I need to replicate the following formuli:

- **DFT**: $F[u,v] = \sum_{x=0}^{M} \sum_{y=0}^{N} f(x,y) * e^{-2\pi i(\frac{u*x}{M} * \frac{v*y}{N})}$
- **IDFT**: $f(x,y) = \sum_{u=0}^{M} \sum_{v=0}^{N} F[u,v] * e^{2\pi i(\frac{u*x}{M} * \frac{v*y}{N})}$

This can be quite easy us ing for loops, like on the two functions below:

DFT (generic function with 1 method)

```
function DFT(f)
    f = Float64.(f)
    (N, M) = size(f)
    F = zeros(ComplexF64, (N, M))
    for u ∈ 1:M
        for v ∈ 1:N
            for x ∈ 1:M
                for y ∈ 1:N
                    uv = (((u-1)*(x-1)/M)+((v-1)*(y-1))/N)
                    F[u,v] += f[x,y] * exp(-1im*2*pi*uv)
                end
            end
        end
    end
    return F
end
```

IDFT (generic function with 1 method)

```
function IDFT(F)
    (N, M) = size(F)
    f = zeros(ComplexF64, (N, M))
    for x ∈ 1:M
        for y ∈ 1:N
            for u ∈ 1:M
                for v ∈ 1:N
                uv = (((u-1)*(x-1)/M)+((v-1)*(y-1))/N)
                f[x,y] += F[u,v] * exp(1im*2*pi*uv)
                end
            end
        end
    end
    return 1/(M*N) * f
end
```

Below, we see the requested items (a, b, c) and a picture with manual DFT and manual IDFT applied, respectively. They all look pretty much the same.

(a vector displayed as a row to save space)

```
let
    a = reals(IDFT(fft(small_cameraman)))           .|> Gray
    b = reals(ifft(fft(small_cameraman)))           .|> Gray
    c = reals(ifft(DFT(Float64.(small_cameraman)))) .|> Gray
    manual = reals(IDFT(DFT(small_cameraman)))      .|> Gray
    [a, b, c, manual]
end
```

DC Component

Below, we see the two requested results:

- a DFT-reconstructed image after setting DC (`F[1, 1]` in Julia) to 0
- the image after removing the mean intensity from all pixels.

They reslut in the same dark image as they do the same thing. Since the DC term holds the sum of all pixel values, removing $MN * |I|$, where MN is the number of pixels and $|I|$ is the mean value of the image, results in the removal of the entirety of DC's value.

(a vector displayed as a row to save space)

```
let
    dft_cam = fft(cameraman)
    dft_cam[1, 1] = 0
    zero_dc = reals(ifft(dft_cam)) .|> Gray

    avg_intensity = mean(cameraman)
    averaged_img = cameraman .- avg_intensity

    [zero_dc, averaged_img]
end
```

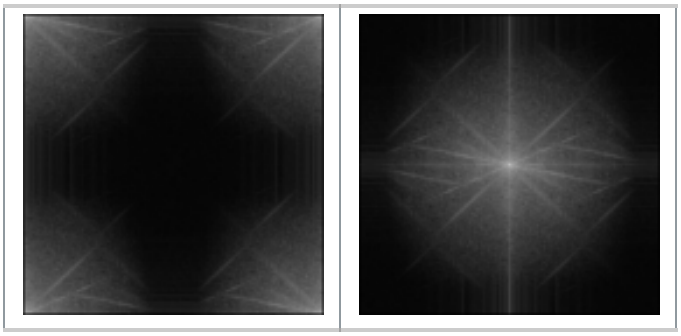
Fourier Spectrum Shift

To depict the fourier spectrum, we need to properly set up the image. This is, the initial DFT representation is somewhat crazy all-around and there isn't much we can see and thus we need to smooth things out as well as remove negative values that wouldn't be rasterized here in Julia. We must do the `abs` value and apply a `log` function. Once that's done, we normalize the values. All is done in the following function:

```
fourier_spectrum (generic function with 1 method)
```

```
function fourier_spectrum(dft)
    a = log.(abs.(dft) .+ 1)
    (l, u) = extrema(a)
    (a .- l) ./ (u - l)
end
```

Below, we see the fourier spectrum of the cameraman image and the shifted spectrum (with the DC component in the middle). This holds all kinds of informations and can even be used to understand if an image is rotated/crooked.



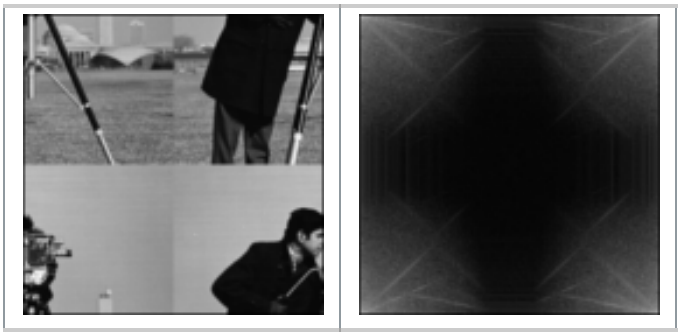
(a vector displayed as a row to save space)

```
let
    dft_cam = fft(cameraman)
    simple_spectrum = fourier_spectrum(dft_cam) .|> Gray
    shifted_spectrum = fourier_spectrum(fftshift(dft_cam)) .|> Gray
    [simple_spectrum, shifted_spectrum]
end
```

We can also see below that applying -1^{x+y} to an image results in the same as reconstructing the image after shifting the fourier spectrum! It produces a darker image because there are negative values spreaded uniformly throughout the image that map to black in the rasterization.

```
(shifted_reconstruction, g) = let
    dft_cam = fft(cameraman)
    shifted_reconstruction = ifft(fftshift(dft_cam)) |> reals .|> Gray
    (N, M) = size(cameraman)
    g = zeros(Float64, N, M)
    for x ∈ 1:M
        for y ∈ 1:N
            g[x, y] = ((-1)^(x-1 + y-1))*(cameraman[x, y])
        end
    end
    g = g .|> Gray
    (shifted_reconstruction, g)
end;
```

Shifting the image and retrieving the spectrum results in the same (or very similar) spectrum of the non-shifted image. that's it because the spectrum is based on magnitudes and phases instead of positional arguments.

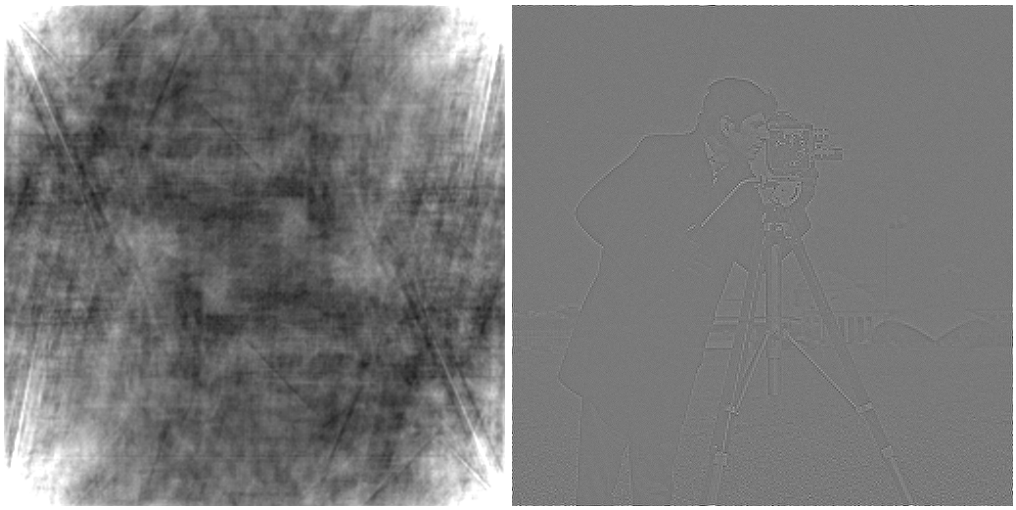


(a vector displayed as a row to save space)

```
let
    shifted_image = fftshift(cameraman)
    reconstructed_shifted =
        fourier_spectrum(fft(Float64.(shifted_image))) .|> Gray
    [shifted_image, reconstructed_shifted]
end
```

Phase + Amplitude

In Julia, retrieving the amplitude and phase spectra is quite easy. The first one, we just need to retrieve the image by the absolute value of the fourier spectrum. The last one, we need to use the `angle` function provided by `FFTW.jl` and perform the exponential operation on the fourier spectrum. We see that the latter provides us with a image with good delimitations to where objects are present.



```
(amplitude, normalized_phases) = let
    dft_cam = fft(cameraman)
    amplitude = ifft(abs.(dft_cam)) |> reals .|> Gray

    phases = ifft(exp.(im .* angle.(dft_cam))) |> reals

    (l, u) = extrema(phases)
    normalized_phases = (phases .- l) ./ (u - l) .|> Gray

    (amplitude, normalized_phases)
end;
```

Conclusions

It's fascinating to see how much information there is to an image. Although very complex, the fourier transform and the concepts it carries along are fundamental to many theories and work that has been done in the field of image processing. This assignment has helped me understand how to manipulate some of the properties seen in class and was essntial to my understanding of the topic.