Assignment 5: Fast Fourier Transform and Image Forensics

Guilherme Gomes Haetinger - 00274702

Proposal: *The goal of this assignment is to implement the Fast Fourier Transform algorithm and to explore a simple, but effective strategy for Image Forensics*

Fast Fourier Transform

As introduced in class, the Fourier Transform maps spatial data to the frequency domain, describing signals in many useful ways. However, in the past assignment, we saw how to compute the underlying frequencies using a *slow* and more intuitive version of the Fourier Transform.

This time, we are taking a look at how to compute the Fourier Transform by factoring polynomials together. This is shown in the video listed in the assignment description. This is analogous to factoring different frequencies from a given signal.

The premise of this method is to divide the problem into simpler problems by separating even and odd polynomials. Sampling is done with an *Nth* root method that introduces the complex conjugates. Sampling size is proportional to the polynomial's degree and so it decreases as we divide the problem. **For some reason, the algorithm presented in the video failed for computing the IFFT, which was said it only needed to change ω by setting $\omega = \frac{1}{n} * \omega^{-1}$. In order for it to work, I found out that I could do $\omega = \frac{1}{\omega}$.**

Thus, we can have the following recursive methods and the respective calls for `FFT` and `IFFT`.

recursive_FFT (generic function with 1 method)

```
function recursive_FFT(P, inverse)
    n = length(P)
    if n == 1
        return P
    end

    ω = exp(2 * π * im / n)
    if inverse
        ω = 1 / ω
    end
    Ω = 1

    Pe = P[1:2:end]
    Po = P[2:2:end]

    ye = recursive_FFT(Pe, inverse)
    yo = recursive_FFT(Po, inverse)

    y = zeros(ComplexF64, n)
    for j ∈ 1:Int(n/2)
        even = ye[j]
        odd = Ω*yo[j]
        y[j] = even + odd
        y[j + Int(n/2)] = even - odd
        Ω *= ω
    end

    return y
end
```

FFT (generic function with 1 method)

```
FFT(P) = recursive_FFT(P, false)
```

IFFT (generic function with 1 method)

```
IFFT(p) = recursive_FFT(p, true) / length(p)
```

In order to apply this to images, we need to transform rows followed by columns. To revert the transform, we must apply the inverse to the columns followed by the rows.

row_map (generic function with 1 method)

```
function row_map(foo, M)
    new_M = nothing
    for row ∈ eachrow(M)
        new_row = reshape(foo(row), (1, :))
        new_M = isnothing(new_M) ? new_row : vcat(new_M, new_row)
    end
    return new_M
end
```

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

```
function col_map(foo, M)
    new_M = nothing
    for col ∈ eachcol(M)
        new_col = reshape(foo(col), (:, 1))
        new_M = isnothing(new_M) ? new_col : hcat(new_M, new_col)
    end
    return new_M
end
```

```
FFT2D (generic function with 1 method)
```

```
FFT2D(M) = col_map(FFT, row_map(FFT, M))
```

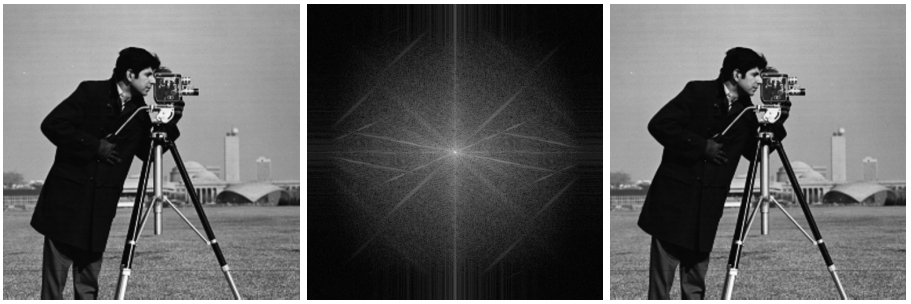
```
IFFT2D (generic function with 1 method)
```

```
IFFT2D(M) = row_map(IFFT, col_map(IFFT, M))
```

Now we can apply these functions to retrieve the real and imaginary components of the image:

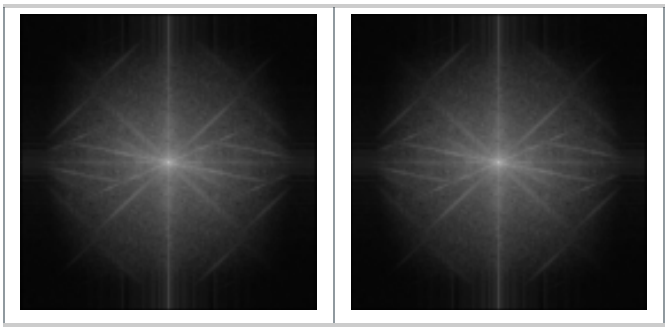
```
real_img, imagine = let
    real_img = FFT2D(cam .|> Float64) |> reals |> IFFT2D |> reals .|> Gray
    imagine = FFT2D(cam .|> Float64) |> imaginaries |> IFFT2D |> imaginaries .|>
    Gray
    real_img, imagine
end;
```

As requested, I display the original image, the DFT spectrum and the reconstructed image using my custom FFT method, respectively. There doesn't seem to be any perceivable difference between the original and recovered image.



```
sp, rec = let
    spectrum = cam .|> Float64 |> FFT2D |> fourier_spectrum |> FFTW.fftshift .|>
    Gray
    reconstr = cam .|> Float64 |> FFT2D |> IFFT2D|> reals .|> Gray
    (spectrum, reconstr)
end;
```

As we can see below, the spectrum generated by my implementation looks quite like the one generated from Julia's `FFTW` library, but not exactly the same. MIne is a bit darker and seems to have higher contrast near the middle!



(a vector displayed as a row to save space)

```
let
my_spectrum = cam .|> Float64 |> FFT2D |> fourier_spectrum |> FFTW.fftshift .|>
Gray
their_spectrum = cam .|> Float64 |> FFTW.fft |> fourier_spectrum |> FFTW.fftshift
.|> Gray
    [my_spectrum, their_spectrum]
end
```

The cell below shows how our result is similar to the ones from Julia's `FFTW` library.

```julia
real_img_std, imagine_std = let
    real_img = FFTW.fft(cam .|> Float64) |> reals |> FFTW.ifft |> reals .|> Gray
    imagine = FFTW.fft(cam .|> Float64) |> imaginaries |> FFTW.ifft |> imaginaries
    .|> Gray
    real_img, imagine
end;
```

I compare the results of the *slow* DFT from the previous assignment applied to the small cameraman image to **FFT's** result. Using the `@benchmark` macro from `BenchmarkTools`, which samples multiple runs of an expression, it's clear why this algorithm is revolutionary. The *slow* DFT takes about 2 seconds to be applied to a $100x100$ image when **FFT** runs on 45 milliseconds when applied to a $512x512$ image. It reduces the time spent to about 2.25% of the *slow* time while running on an image of around 5x the small image size!

```
BenchmarkTools.Trial: 3 samples with 1 evaluation.
 Range (min … max):  2.046 s …   2.084 s  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     2.051 s              ┊ GC (median):    0.00%
 Time  (mean ± σ):   2.060 s ± 20.230 ms  ┊ GC (mean ± σ):  0.00% ± 0.00%

  2.05 s         Histogram: frequency by time        2.08 s <

 Memory estimate: 312.64 KiB, allocs estimate: 6.
```

```julia
@benchmark DFT(small_cameraman .|> Float64)
```

```
BenchmarkTools.Trial: 107 samples with 1 evaluation.
 Range (min … max):  38.611 ms … 49.586 ms  ┊ GC (min … max):  0.00% … 14.69%
 Time  (median):     48.055 ms              ┊ GC (median):    15.23%
 Time  (mean ± σ):   46.976 ms ±  3.064 ms  ┊ GC (mean ± σ):  13.63% ±  5.21%

  38.6 ms         Histogram: frequency by time        49.2 ms <

 Memory estimate: 156.72 MiB, allocs estimate: 786815.
```

```julia
@benchmark FFT(cam .|> Float64)
```

Forensics

For the forensics portion of the assignment, I used a picture of a llama I took when walking around the Berkeley campus.

I used Gimp to resave the image with qualities of `100%` (to test out whether there is any change) `50%` and `25%`.

```
begin
    llama_original = load("./res/llama.jpg")
    llama_100 = load("./res/llama100.jpg")
    llama_50 = load("./res/llama50.jpg")
    llama_25 = load("./res/llama25.jpg")
end;
```

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

```
function composite(A, B)
    canvas = zeros(RGB{N0f8}, size(A))
    h, w = size(canvas)
    middle = floor(Int64, w/2)
    canvas[:, 1:middle] = A[:, 1:middle]
    canvas[:, middle+1:end] = B[:, middle+1:end]
    return canvas
end
```

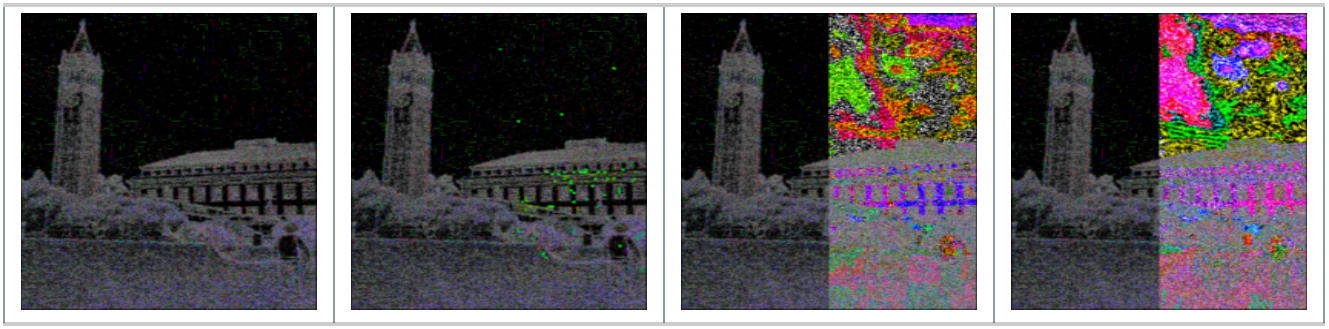
Now, I can create a vertically split composite image using the original and each one of the recompressions I made on Gimp. I'll create the composite image, save it in `jpg` format and reload them. I also save the original image in `jpg` to see whether it differs from the `100%` composition.

```
composites = let
    llama_100_composite = composite(llama_original, llama_100)
    llama_50_composite = composite(llama_original, llama_50)
    llama_25_composite = composite(llama_original, llama_25)

    save("./res/llama_new_original.jpg", llama_original)
    save("./res/llama_100_composite.jpg", llama_100_composite)
    save("./res/llama_50_composite.jpg", llama_50_composite)
    save("./res/llama_25_composite.jpg", llama_25_composite)

    orig = load("./res/llama_new_original.jpg")
    llama_100_composite = load("./res/llama_100_composite.jpg")
    llama_50_composite = load("./res/llama_50_composite.jpg")
    llama_25_composite = load("./res/llama_25_composite.jpg")

    [orig, llama_100_composite, llama_50_composite, llama_25_composite]
end;
```

All the composed images look the same to me 🤦

(a vector displayed as a row to save space)

Below we can see the distance between the original image and the saved composites (+ the resaved original). We can see that obviously the difference is higher with the quality decrease. There is also noticeable discrepancies between the original recompression and the `100%` compression composite, which means loading a `jpg` image in Gimp and reexporting it will have you lose information. The images are in the following order: original recompression, `100%` composite, `50%` composite, `25%` composite.



(a vector displayed as a row to save space)

Conclusions

I was already aware that FFT was really fast, but implementing it after having implemented brute force DFT was really nice as it develops the intuition behind this sleek algorithm. I was very confused as to why my ω variable didn't work the way it should, but I'm pretty happy that I was able to fix it with a different approach.

The image forensics class was amazing and I think out of all the algorithms presented, this one was the simplest one, so I don't think I got anything other than discovering that Gimp alters `jpg` images when reexported in `100%` compression quality.

All things aside, this was really cool to build up.