Notes on Chapter 32: The Fast Fourier Transform

Chris Lowis

May 15, 2016

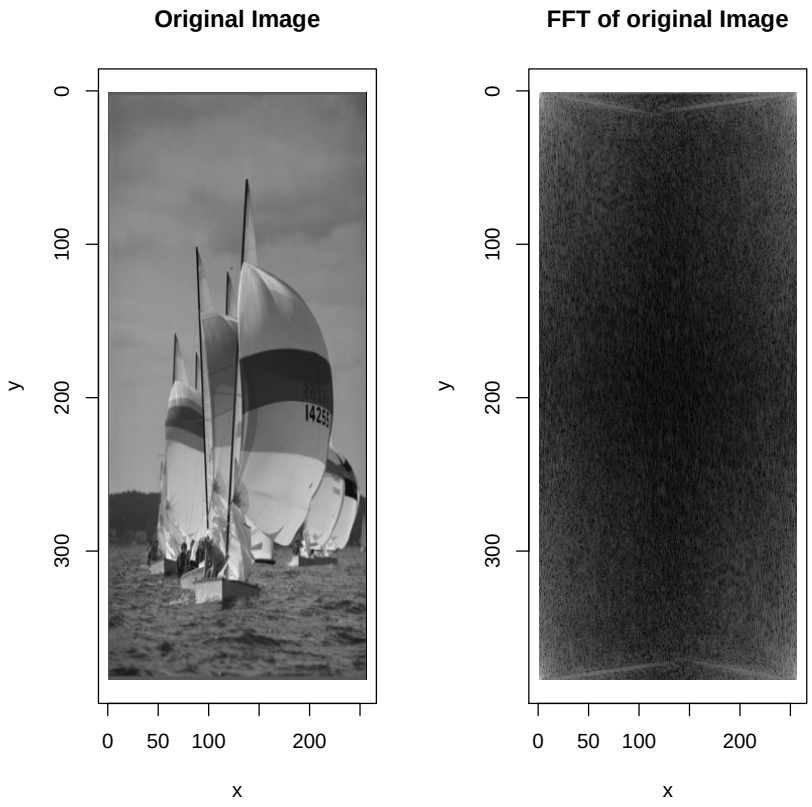
Chapter 32: The Fast Fourier Transform

Motivation

Intuitively I've always understood the Fourier Transform in terms of telling us how much "energy" is in each frequency of an input signal. I like to think of the graphic EQ on a hi-fi amplifier as being like a Fourier Transform. It takes the input music, splits it into frequency bands, and allows us to change the input by manipulating the frequency directly - increasing the bass, or rolling off the treble, for example.

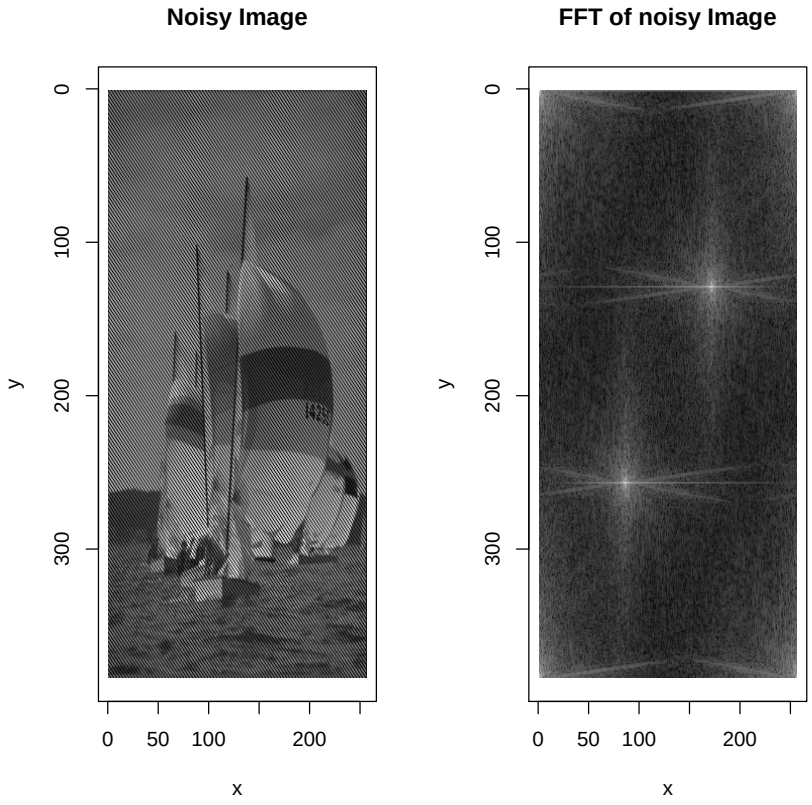
The chapter talks about applications of the FFT for image processing and working with polynomials, which is a bit less intuitive for me, so I've tried to expand on the image example to make it more obvious why we care about the FFT in image processing.

Here is an image and its FFT. I've converted the FFT to a power spectrum (which combines the real and imaginary part and takes the log) because that makes it easier to see what's happening. I think the book did something like this too in the example on page 217.



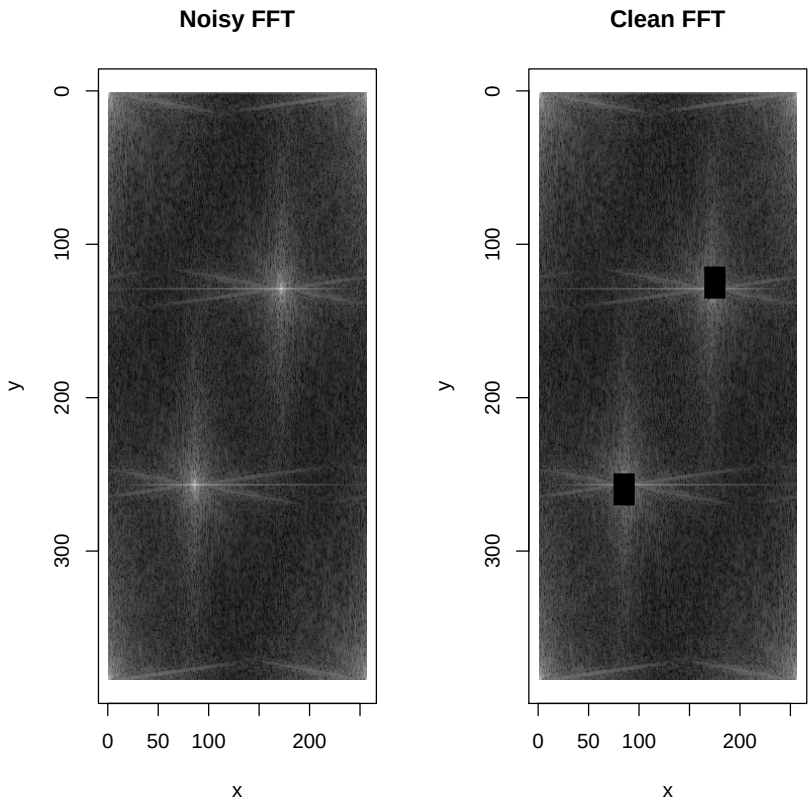
The FFT of the image is a representation of its "spectrum". Higher frequency parts of the image are further away from the centre of the FFT. But it's not clear what we'd expect the spectrum of a picture of some boats to be.

What happens if we add some periodic noise to the original image?

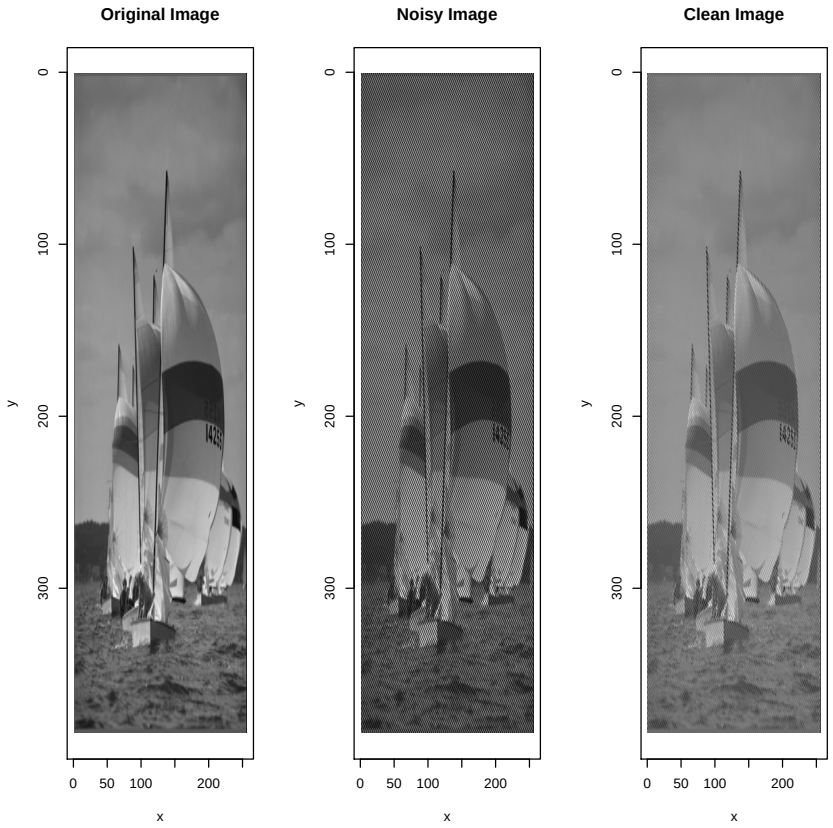


The repeating diagonal lines in the original image appear as bright "stars" when we take the FFT. That's because they have a strong "frequency" that repeats throughout the image. The frequency domain representation that we get from taking the FFT makes this apparent.

Say we wanted to remove this periodic noise from the image. It's much easier to identify these bright spots, and perform some crude surgery on them (by setting their real and imaginary components to 0) when working in the frequency domain.



We can then take the Inverse FFT of the FFT we've operated on, and turn it back into an image.



The DFT

The discrete version of the fourier transform is

$$F_k = \sum_{n=0}^{N-1} f_n e^{-2\pi i k n / N}$$

I've used a slightly different notation here than in the chapter, because I found it easier to read when it was all written on a single line.

The large $\sum$ is notation for summation, and the i is the complex number $\sqrt{-1}$. We can translate this equation into the equivalent ruby code

```
require 'complex'

def dft(f)
  bigN = f.length
```

```
  bigF = Array.new(bigN)

  bigF.each_with_index do |_, k|
    bigF[k] = 0

    0.upto(bigN-1).each do |n|
      bigF[k] = bigF[k] +
        ( f[n] * Math::E ** (-2.0 * Math::PI * Complex(0, 1) * k * n / bigN) )
    end
  end

  bigF
end
```

I've not tried to be idiomatic here, rather to mirror the equation as closely as possible in ruby.

Let's use the code above to calculate the DFT of the input "image" $[0, 1, 0, 1, 0, 1]$ (this is exercise 1 in the problems at the end of the chapter).

```
f = [0, 1, 0, 1, 0, 1]
dft(f)
```

There's a lot of numerical noise here, but the output is $[3, 0, 0, -3, 0, 0]$.

One thing that wasn't very clear in the chapter is that k takes integer values, and by convention these are normally chosen to be $[0 \ldots N-1]$. Note that this also means that F_k is the same length as f, the input to the DFT.

The DFT in Matrix Form

The chapter then goes on to show how the DFT can be written in an alternative, more compact form, using matrices.

You may recall from maths at school that the product of a matrix $\mathbf{A}$

$$\mathbf{A} = \begin{pmatrix} a & b & c \\ p & q & r \\ u & v & w \end{pmatrix}$$

with a vector $\mathbf{x}$

$$\mathbf{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

is defined as follows

$$\mathbf{Ax} = \begin{pmatrix} a & b & c \\ p & q & r \\ u & v & w \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} ax + by + cz \\ px + qy + rz \\ ux + vy + wz \end{pmatrix}$$

Notice that the result of multiplying a matrix by an input vector is a new vector with the same number of elements as the input. The first element in the output is the sums of the products of the the first row of the matrix with each of the input vectors elements. Compare this with the first equation for the DFT.

Consider taking the DFT of a three-element input vector. We can rewrite that first equation in matrix form

$$\mathbf{g} = \mathbf{A}\mathbf{f}^T$$

where

$$\mathbf{A} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & \omega & \omega^2 \\ 1 & \omega^2 & \omega^4 \end{pmatrix}$$

and

$$\omega = e^{\frac{-2\pi i}{3}}$$

In general, the DFT matrix $\mathbf{A}$ for an input image of size N is

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \cdots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \cdots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \cdots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

where

$$\omega = e^{\frac{-2\pi i}{N}}$$

The DFT in Matrix Form in Ruby

We can implement that equation in ruby too. Doing matrix operations in ruby using the standard library is a bit clunky, as accessing individual elements of a matrix by their row and column numbers is not supported. We can get around that by monkey patching the `Matrix` class

```
require 'matrix'

class Matrix
  def []=(row, column, value)
    @rows[row][column] = value
  end
end
```

We then define a matrix version of the DFT as follows

```
def matrix_dft(f)
  bigN = f.size
  bigA = Matrix.identity(bigN)

  omega = Math::E ** (-2.0 * Math::PI * Complex(0, 1) / bigN)

  bigA.each_with_index do |_, row, col|
    bigA[row, col] = omega ** (row * col)
  end

  bigA * Vector.elements(f)
end
```

Evaluating that on the same input as earlier we get (roughly) the same result as before.

```
matrix_dft(f).to_a
```

The Inverse DFT in Matrix Form in Ruby

Converting back from a fourier transformed image to the original image can be achieved by multiplying by the inverse of $\mathbf{A}$

$$\mathbf{f} = \mathbf{A}^{-1}\mathbf{g}^T$$

Which we can write in ruby as

```
def matrix_idft(g)
  bigN = g.size
  bigA = Matrix.identity(bigN)

  omega = Math::E ** (-2.0 * Math::PI * Complex(0, 1) / bigN)

  bigA.each_with_index do |_, row, col|
    bigA[row, col] = omega ** (row * col)
  end

  bigA.inverse * Vector.elements(g)
end
```

We can check that our implementation works by taking the DFT and then the inverse DFT of an input

```
g = matrix_dft(f)
matrix_idft(g).to_a
```