Trasformada rápida de Fourier utilizando Python

Michael Spilsbury[1] y Armando Euceda[2]

[1] *Escuela de Física - UNAH , mail: michael.spilsbuy@unah.edu.hn*
[2] *Escuela de Física - UNAH, mail: aeunah@gmail.com*

Recibido: 28 de febrero del 2017 / Aceptado: 30 de abril del 2017

Resumen

A continuación se presenta un programa de computadora para calcular la transformada discreta de Fourier utilizando el algoritmo de la transformada rápida de Fourier (FFT por sus siglas en inglés). Desde 1965[2], cuando James W. Cooley y John W. Tukey publicaron dicho algoritmo, su uso se ha expandido rápidamente y las computadoras personales han impulsado una explosión de aplicaciones adicionales de la FFT. Como lenguaje de programación se usará Python, que es un lenguaje de programación multiparadigma, esto significa que más que forzar a los programadores a adoptar un estilo particular de programación, permite varios estilos: programación orientada a objetos, imperativa y funcional. Los usuarios de Python se refieren a menudo a la filosofía Python que es bastante análoga a la filosofía de Unix. Al mismo tiempo se integrará un módulo de Fortran para mejorar el desempeño.

Palabras clave: Transforma rápida de Fourier, algoritmo, transformada discreta de Fourier, Cooley, Tukey, programació, Python, Fortran.

Next a computer program is presented to calculate the discrete Fourier transform using the fast Fourier transform algorithm. Since 1965[2], when James W. Cooley and John W. Tukey published this algorithm, its use has expanded quickly and personal computers have generated an explosion of further FFT applications. The programming language to use is Python, which is a multi-paradigm programming language, this means that rather than forcing programmers to adopt a particular style of programming allows several styles: object-oriented, functional and imperative programming. Python users often refer to the Python philosophy is quite analogous to the Unix philosophy. At the same time Fortran module will be integrated to improve performance.

Keywords: Fast fourier transform, algorithm, discrete Fourier transform, Cooley, Tukey, programming, Python, Fortran.

I. Programa en Python para cálculos usando FFT

Como se apuntó inicialmente, el lenguaje base para la programación de la FFT en este trabajo es Python, que es un lenguaje interpretado y debido a esto, cuando se tiene un gran número de muestras la velocidad de procesamiento se ve reducida; por lo que para dar velocidad y poder de procesamiento a los cálculos se utilizará un módulo de Fortran (lenguaje compilado) que será utilizado por Python para realizar los cálculos.

Python tiene desarrolladas muchas extensiones que le dan mayor versatilidad, una de ellas es *Numpy* que es el paquete fundamental para el cálculo científico, que contiene entre otras cosas:

- Un poderoso arreglo de objetos N-dimensionales.
- Funciones (transmitidas) sofisticadas.
- Herramientas para la integración de código `C/C++` y Fortran.
- Útiles funciones de álgebra lineal, transformada de Fourier y números aleatorios.

Como se indica, Numpy contiene dentro de sus ventajas varias funciones para transformadas de Fourier, que nos servirán para comparar los tiempos de cálculo y los datos obtenidos. Para conocer sobre Python, ver [3]

II. Programa FFT

Se desarrollará a continuación el programa en código Python, siguiendo el diagrama de flujo expuesto por Brigham[1] (Fig. 1). Cabe mencionar que en el desarrollo se considera que el número de muestras, N, es una potencia de 2 ($N = 2^{\gamma}$). Además recordar que el diagrama surge de la aplicación de la transformada discreta de Fourier y sus propiedades, y en sí el algoritmo no requiere una análisis profundo ya que solamente ayuda a calcular más rápida y eficientemente la transformada discreta de Fourier, por lo tanto no se dará mayor detalle. Se puede consultar la referencia antes citada.

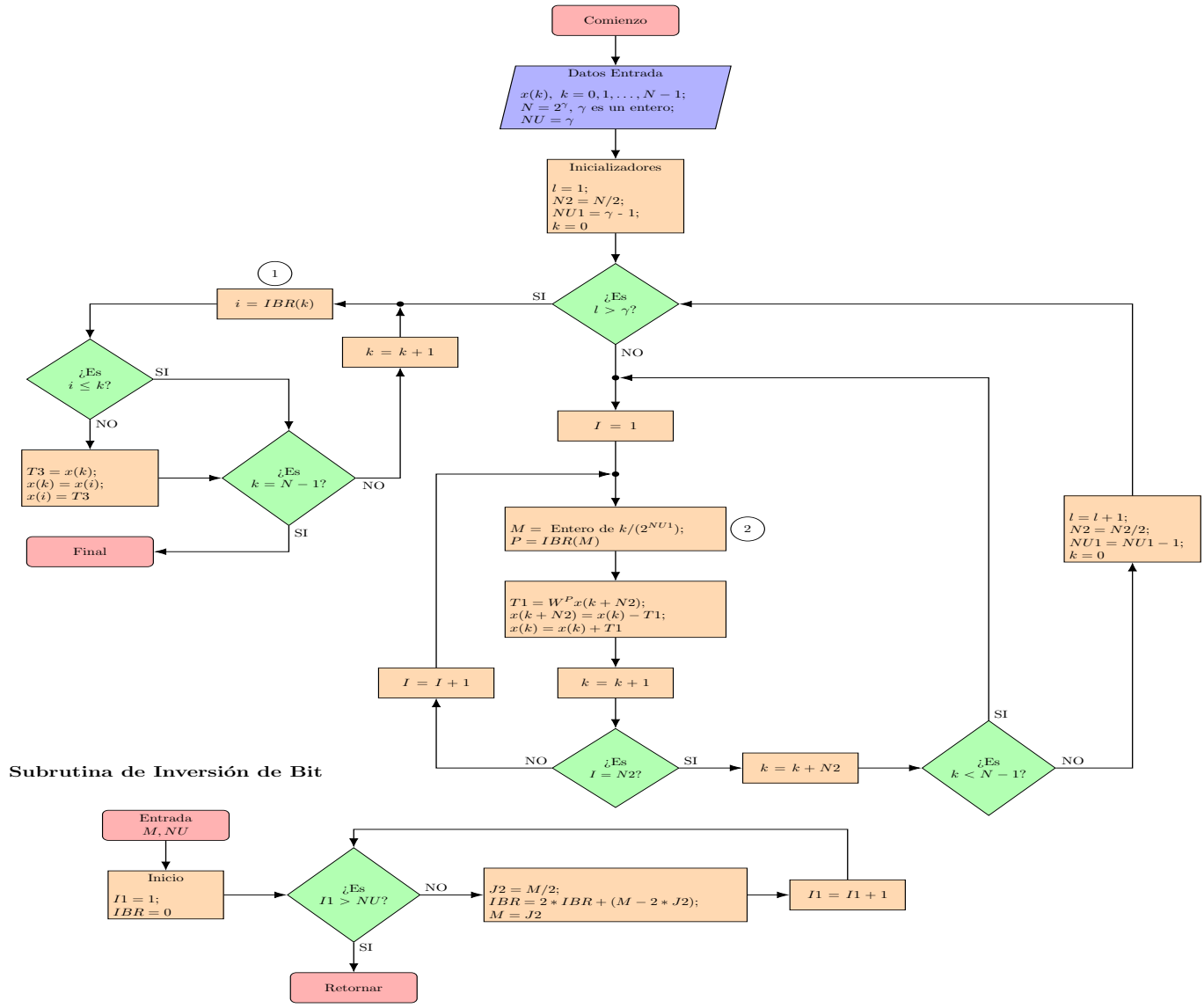

Figura 1: *Diagrama de flujo del programa FFT. Fuente: Brigham[1]*

Para ejemplo se utilizarán muestras de la función $A\,\text{sen}(2\pi f_0 t)$, tomando $A = 1$, $f_0 = 250$ Hz y $N = 2^{10} = 1024$ muestras:

```
##Transformada rápida de Fourier
from __future__ import print_function, division
from pylab import *

##Variables de entrada
##xr = parte real de datos
##xi = parte imaginaria de datos
##W = exp(-1j * 2 * np.pi / N)
##T1 = valor temporal (W^p)*x(k + N2)
xr = []
xi = []
nu = 10
n = np.power(2, nu)

##Datos de pruebas
k_muestras = np.arange(n)

##Frecuencia y periodo función seno
f_0 = 250          ##Frecuencia
T_0 = 1 / f_0      ##Periodo
nT = 20            ##Veces el periodo a graficar
T = nT * T_0
delta_t = 1 / n    ##factor entre TFC y TFD
delta_t1 = T / n   ##Delta_t del muestreo
delta_f = 1 / T    ##Delta_f del muestreo
t = np.array(delta_t1 * k_muestras)
xr = np.sin(2 * np.pi * f_0 * t)
xi = [0] * n
xz = np.array(xr) + 1j * np.array(xi)

##Inicializar variables auxiliares
l = 1
n2 = np.int_(n / 2)
nu1 = nu - 1
k = 0

##Implementación Python-Inicio
xza = np.array(xz)
while l <= nu:
    while k <= n - 1:
        q = 1
        while q <= n2:
            M = np.int_(k / np.power(2, nu1))
            P =
    np.int_('{:0{width}b}'.format(M, width = nu)
                                [::-1],2)
            W = np.exp(- 1j * 2 * np.pi / n)
```

```
47                T1 = xza[k + n2] * W**P
48                xza[k + n2] = xza[k] - T1
49                xza[k] = xza[k] + T1
50                k += 1
51                q +=1
52            k += n2
53        l += 1
54        n2 = np.int_(n2 / 2)
55        nu1 -= 1
56        k = 0
57    while k < n - 1:
58        i = np.int_('{:0{width}b}'.format(k, width
          = nu)
59                    [::-1],2)
60        if i > k:
61            T3 = xza[k]
62            xza[k] = xza[i]
63            xza[i] = T3
64        k +=1
65    xza *= delta_t
66    ##Implementación Python-Final
67
68    ##Ajuste de datos de frecuencia
69    f = []
70    for s in range(len(k_muestras)):
71        if s <= (n / 2 + 1):
72            f.append(s * delta_f)
73        else:
74            f.append((s - n) * delta_f)
```

Se puede apreciar que en el programa en Python no se consideró la subrutina de *inversión de bit* ya que se utilizó una de las muchas ventajas de Python para realizar esto en una sola línea de comando, siendo estas líneas la 44 y la 58 (estas líneas son la caja 1 y la caja 2 del diagrama de flujo, que utilizan la subrutina de inversión de bit). Además la línea 12 corresponde a la potencia de la base 2 que genera el número de muestras tomadas en la función antes mencionada, definiendo parámetros y muestras de la misma entre las líneas 19 a la 29. Las líneas entre la 31 a la 66 corresponden al programa indicado en el diagrama de flujo de la Fig. 1. La línea 65 corresponde al factor de escala resultante del análisis de la transformada discreta de Fourier (Brigham[1]). Las líneas entre la 69 a la 74 corresponden a los ajustes de los datos de frecuencia con los que se representa la transformada contínua de Fourier, siendo los límites de $-f_0$ a f_0.

En la Fig. 2 se muestran los datos obtenidos para el caso mencionado $[A\,\mathrm{sen}(2\pi f_0 t)]$, y al comparar con el par transformado para esta función

$$A\,\mathrm{sen}(2\pi f_0 t) \Leftrightarrow -j\frac{A}{2}\delta(f-f_0) + j\frac{A}{2}\delta(f+f_0) \qquad (1)$$

se observa coincidencia con los resultados arrojados por el programa.

Ademas en la Fig. 3 se observa el caso para $A\cos(2\pi f_0 t)$, tomando $A = 1$, $f_0 = 250$ Hz y $N = 2^{10} = 1024$ muestras y cuyo par transformado de Fourier es

$$A\cos(2\pi f_0 t) \Leftrightarrow \frac{A}{2}\delta(f-f_0) + \frac{A}{2}\delta(f+f_0) \qquad (2)$$

Nótese el color de línea diferente en el gráfico de la transformada de Fourier de la Fig. 3, en contraste con el correspondiente de la Fig. 2.

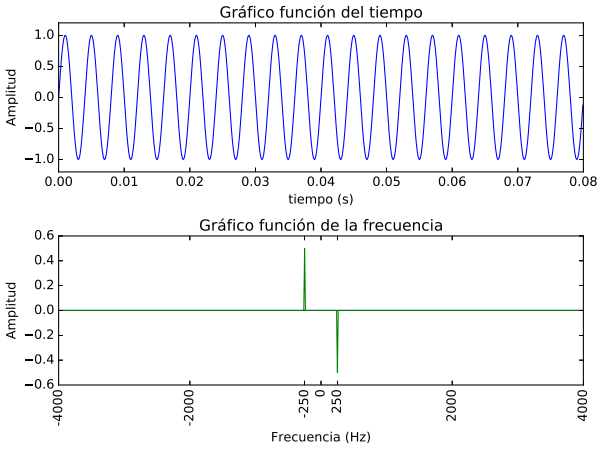

Figura 2: *Gráfico de la función* $A\,\mathrm{sen}(2\pi f_0 t)$ *y su transformada.* $A = 1$ *y* $f_0 = 250$ *Hz*

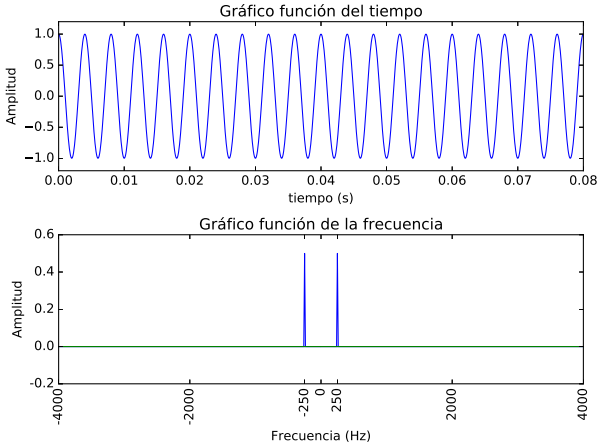

Figura 3: *Gráfico de la función* $A\cos(2\pi f_0 t)$ *y su transformada.* $A = 1$ *y* $f_0 = 250$ *Hz*

Precisamente, ya que la transforma de Fourier en general es una función compleja

$$\begin{aligned} H(f) &= \int_{-\infty}^{\infty} h(t)e^{-j2\pi ft}dt \\ &= R(f) + jI(f) = |H(f)|e^{j\theta(f)} \end{aligned} \qquad (3)$$

en el programa la parte real $[R(f)]$ deberá graficarse con línea de color diferente o tipo de línea a la de la parte imaginaria $[I(f)]$ [en estos gráficos $R(f)$ e $I(f)$ aparecen con color de línea azul y verde respectivamente]. Para ejemplo tómese la función:

$$h(t) = \begin{cases} \beta e^{-\alpha t} & t > 0 \\ 0 & t < 0 \end{cases} \qquad (4)$$

cuya transformada de Fourier es:

$$\begin{aligned} H(f) &= R(f) + jI(f) \\ &= \frac{\beta\alpha}{\alpha^2 + (2\pi f)^2} - \frac{j2\pi f\beta}{\alpha^2 + (2\pi f)^2} \end{aligned} \qquad (5)$$

En la Fig. 4 se muestra el gráfico correspondiente.

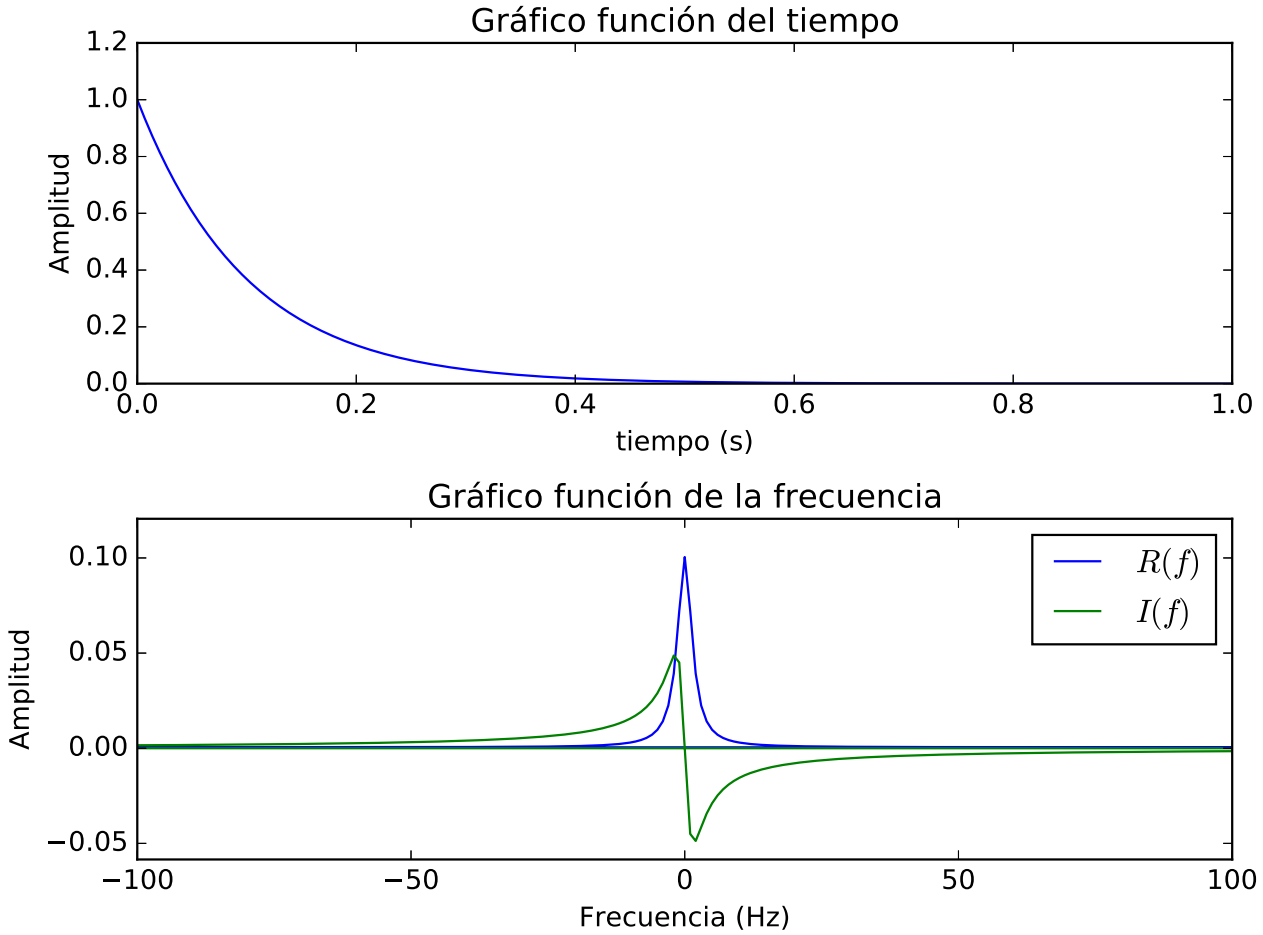

Figura 4: *Gráfico de* $e^{-\alpha t}$ *para* $t > 0$ *y su transformada.* $\alpha = 10$ *Hz*

III. Análisis de resultados del programa

Revisando el tiempo de ejecución del programa utilizando código Python únicamente, le tomaba alrededor de 200 veces el tiempo que le toma a la extensión Numpy de Python. Esta diferencia se debe a que Numpy realiza el cálculo basado en Fortran, por lo que se elaborará un modulo de Fortran que es llamado por Python y de esta forma reducir el tiempo de ejecución de las rutinas de cálculo. Con esto se obtendrá una mejora notable, casi igual al tiempo de la extensión Numpy. Por ejemplo, para la función pulso cuadrado, que tiene el par transformado.

$$h(t) = \begin{cases} A & |t| < T_0 \\ A/2 & |t| = T_0 \\ 0 & |t| > T_0 \end{cases} \Leftrightarrow$$

$$H(f) = 2AT_0 \frac{\text{sen}\,(2\pi T_0 f)}{2\pi T_0 f} \qquad (6)$$

con un número de muestras igual a 1024, se realizó el calculo por los tres métodos obteniéndose los siguientes tiempos de ejecución:

```
Tiempo implementación Python:          0.064370020345 s
Tiempo implementación Python_Fortran:  0.000663668916201 s
Tiempo implementación FFT_Numpy:       0.000339165116232 s
```

En cuando a los errores obtenidos respecto de la función continua, arrojó (el error relativo se calculó de la suma de las diferencias entre el número de muestras multiplicado por 100):

```
Diferencia mayor en los datos =  0.000607593968998
Error relativo =                 0.0178151019639
```

A continuación se muestran las figuras obtenidas los tres métodos

Para que Python llame el módulo de Fortran, en primer lugar se escribio en lenguaje Fortran el programa (siguiendo el diagrama de flujo) y luego con el paquete de Pyhton *F2PY* se genera dicho módulo. Los pasos seguidos fueron:

- Generar un *archivo de firma* (signature file), usando el comando:
  `f2py mod-fpy.f -m mod-fpy -h mod-fpy.pyf`
  El archivo con extensión .pyf es el generado.

- Generar el módulo con el comando:
  `f2py -c mod-fpy.pyf mod-fpy.f95`

lo anterior se hace desde la línea de comandos del sistema operativo. Con la generación del módulo en el programa se debe agregar en la línea 4:

```
4 import mod-fpy
```

y se deben reemplazar las líneas de la 31 a la 66 por:

```
32 ######################################
33 ##Implementacion Python_Fortran-Inicio
34 ######################################
35 mod-fpy.fft_fortran.nu = nu
36 mod-fpy.fft_fortran.n = n
37 mod-fpy.fft_fortran.xzf = list(xz)
38 mod-fpy.fft_fortran.fft_it()
39 spxz = mod-fpy.fft_fortran.xzf
40 spxz *= delta_t
41 ##Implementación Python_Fortran-Final
```

donde *fft_fortran* y *fft_it* es el nombre del módulo y la subrutina dentro del programa en Fortran. A continuación se presenta el *archivo de firma*:

```
1 !    -*- f90 -*-
2 ! Note: the context of this file is case
      sensitive.
3
4 python module mod-fpy ! in
5     interface  ! in :mod-fpy
6         module fft_fortran ! in
      :mod-fpy:mod-fpy.f95
7             complex(kind=8),
      allocatable,dimension(:) :: xzf
8             integer :: nu
9             integer, parameter,optional ::
      dp=selected_real_kind(15, 307)
10            integer :: n
11            subroutine fft_it ! in
      :mod-fpy:mod-fpy.f95:fft_fortran
12            end subroutine fft_it
13            function ibr(val_m,val_nu) ! in
      :mod-fpy:mod-fpy.f95:fft_fortran
14                integer :: val_m
15                integer :: val_nu
16                integer :: ibr
17            end function ibr
18        end module fft_fortran
19    end interface
20 end python module mod-fpy
21
22 ! This file was auto-generated with f2py
      (version:2).
23 ! See http://cens.ioc.ee/projects/f2py2e/
```

IV. Conclusiones

1. El desarrollo del programa por medio del algoritmo, resulta bastante intuitivo, indiferentemente del lenguaje de programación utilizado. Al mismo tiempo no se requiere mayor detalle, como se apunto, de la teoría de la transformada de Fourier.

2. El tiempo del cálculo se vio reducido considerablemente con el uso del lenguaje compilado (Fortran), mejorando el desempeño del programa. Sin embargo, aún se puede reducir el tiempo utilizando lo expuesto por Brigham, en el caso que la función del tiempo sea real [1, Sección 9.3].

Referencias

[1] Brigham, E.O., Ed. (©1988). *The fast Fourier transform and its applications*. Prentice Hall, Englewood Cliffs, N.J. ISBN 0-13-307505-2. URL `http://www.worldcat.org/oclc/17384259`.

[2] Cooley, J.W. y Tukey, J.W., *An algorithm for the machine calculation of complex Fourier series*. En Mathematics of Computation, Volumen 19 (1965) (90): 297. ISSN 0025-5718. doi:10.1090/S0025-5718-1965-0178586-1.

[3] Langtangen, H.P. (©2012). *A primer on scientific programming with Python*, Volumen 6 desde *Texts in computational science and engineering*. Springer, Berlin and New York, 3rd ed enlugar. ISBN 978-3-642-30292-3.