# Fourier Analysis: A Study of the Computation of Fourier Series and Transforms of Various Functions.

David Lawton

24th Nov. 2023

## Contents

# 1 Abstract

The objective of this computational lab was to create a function to produce the Fourier expansion of functions, and to create a function to produce the discrete Fourier transform of functions, while finding interesting results along the way, and using the code produced. In the creation of these functions, we used numerical integration, employing Simpson's rule to great effect.

# 2 Introduction

Throughout this experiment we used several computational and mathematical concepts, foremost of which was Simpson's rule, Fourier series, discrete Fourier transform and back transform.

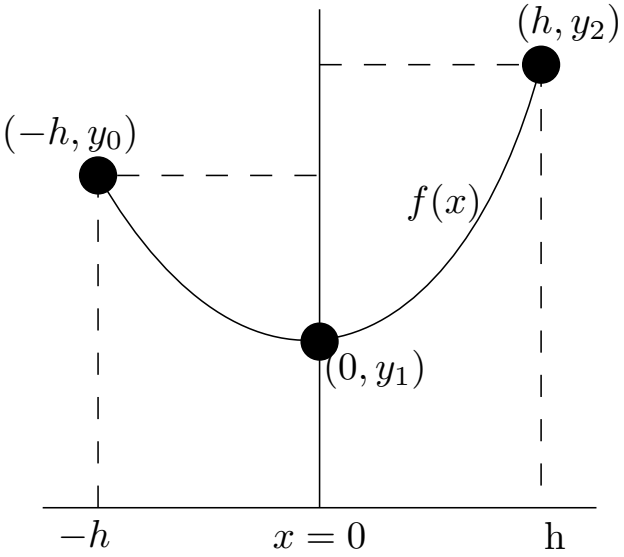


Figure 1: Parabola across interval of $2h$

The experiment began by using Simpson's rule to create a function to integrate over a period. Simpson's rule uses parabolas to approximate the area under a curve, which is a more accurate method than the trapezoidal rule. For integral

$$F = \int_a^b f(x)\mathrm{d}x \tag{1}$$

The method splits the interval of integration $[a, b]$ into $n$ periods of width $h = \frac{(b-a)}{n}$. Due to the nature of the parabola, the area under the curve is

$$\text{Area} = \int_a^b (ax^2 + bx + c)\mathrm{d}x = \frac{h}{3}(2ah^2 + 6c) \tag{2}$$

which is equivalent to Eq. 3, for the area under each curve.

$$A_0 = \frac{h}{3}(y_0 + 4y_1 + y_2) \tag{3}$$

where $y_0$ is the value of $f(x)$ at $a$, $y_2$ is the value at $b$ and $y_1$ is the value at the midpoint, $a+h$. When extended over the whole period it results in the following approximation of a function over a period.

$$\int_a^b f(x)\mathrm{d}x \approx \frac{h}{3}\left[f(x_0) + 2\sum_{j=1}^{\frac{n}{2}-1} f(x_{2j}) + 4\sum_{j=1}^{\frac{n}{2}} f(x_{2j-1}) + f(x_j)\right] \tag{4}$$

where $x_0$ and $x_j$ are the first $a$ and last $b$.

The next step was to make a function to evaluate the fourier coefficients, and consequentually the Fourier series of the function. Any periodic function, $f(t)$ such that $f(t) = F(t+T)$, can be written as a Fourier series.

$$f(t) = a_0 + \sum_{n=1}^{\infty}\left((a_n\cos\left(\frac{2\pi nt}{T}\right) + b_n\sin\left(\frac{2\pi nt}{T}\right)\right) \tag{5}$$

where

$$a_0 = \frac{1}{T}\int_0^T f(t)\mathrm{d}t \tag{6}$$

$$a_m = \frac{1}{T}\int_0^T f(t)\cos\left(\frac{2\pi mt}{T}\right)\mathrm{d}t, \quad b_m = \frac{1}{T}\int_0^T f(t)\sin\left(\frac{2\pi mt}{T}\right)\mathrm{d}t \tag{7}$$

This is an extremely useful way of expanding periodic functions that is used across all areas of maths, physics and engineering. Being able to write complicated functions as series of simple trigonometric functions is one of the handiest mathematical tools.

It followed from the previous steps to use our methods for an interesting function, such as, in this case, the square wave and rectangular wave.

$$f(t) = \begin{cases} 1, & \text{if } 0 \le \theta \le \pi \\ -1, & \text{if } \pi \le \theta \le 2\pi \end{cases} \quad g(t) = \begin{cases} 1, & \text{if } 0 \le \theta \le \omega\tau \\ -1, & \text{if } \omega\tau \le \theta \le 2\pi \end{cases} \tag{8}$$

where $\theta = \omega t$, with $\omega$ being the frequency of the periodic motion, these functions are especially useful fo illustrating how the Fourier expansion approximates a function.

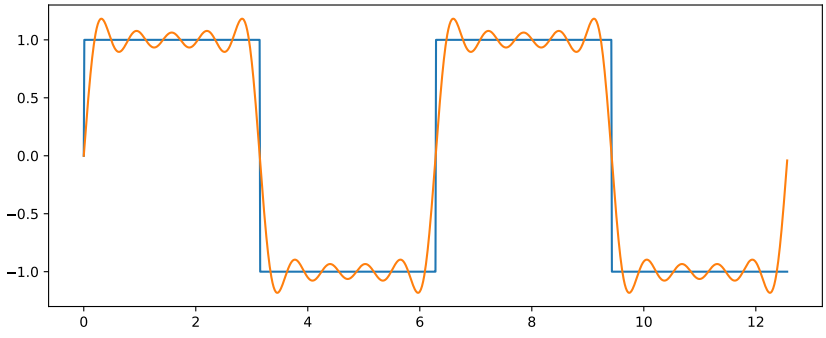


Figure 2: An illustration of the Fourier expansion of a square wave approaching the actual function.

Continuing from the previous sections, we designed a function to get the discrete Fourier transform over an interval of frequencies. Any function can be written, instead of a discrete sum to $\infty$ as in fourier series, an integral over $(-\infty, \infty)$.

$$f(t) = \int_{-\infty}^{\infty}\left(a(\omega)\cos(\omega t) + b(\omega)\sin(\omega t)\right)\mathrm{d}\omega. \tag{9}$$

which is more commonly written as two integrals, shifting to and from the frequency and time domains.

$$f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)\mathrm{e}^{\mathrm{i}\omega t}\mathrm{d}\omega, \quad F(\omega) = \int_{-\infty}^{\infty} f(t)\mathrm{e}^{-\mathrm{i}\omega t}\mathrm{d}t \tag{10}$$

Of course, we cannot integrate across infinity computaionally. We instead use the discrete Fourier transform (DFT) and back transform, which samples our values of $F_n$ and $f_m$ respectively. The discrete variables $n$, $m$ replace the continuous $\omega$, $t$.

$$f_m = \frac{1}{N}\sum_{n=0}^{N-1} F_n \left(\cos\left(\frac{2\pi mn}{N}\right) + \mathrm{i}\,\sin\left(\frac{2\pi mn}{N}\right)\right) \tag{11}$$

$$\Re(F_n) = \sum_{n=0}^{N-1} f_m \cos\left(\frac{2\pi mn}{N}\right), \quad \Im(F_n) = \sum_{n=0}^{N-1} -f_m \sin\left(\frac{2\pi mn}{N}\right) \tag{12}$$

# 3 Methodology

## 3.1 Simpson's Rule

The methodology for Simpson's rule is relatively simple, with it being a numerical approximation itself. The only tricky part being the seperation of odd and even $j$. A function with inputs $x_0$ and $x_j$ and $n$ was created to evaluate integrals. To allow the function to work for any $n$ we arbitrarily multiplied it by two inside the function.

```
def F(a, b, n):
    a_b = np.arange(a, b, (b-a)/(2*n))
    h = (b-a)/(2*n)
    J1 = 0
    J2 = 0
    for j in range(1, len(a_b)):
        if j % 2 == 0:
            J1 += f(a_b[j])
        elif j % 2 != 0:
            J2 += f(a_b[j])

    F_ = (h/3) * (f(a) + 2*J1 + 4*J2 + f(b))\\
    return F_
    print(F_)
```

To test this function we integrated $e^x$ across $[0, 1]$, with our input $n = 4$, so 8 steps, the function reached an answer accurate to six signifiant figures.

## 3.2 Fourier Series

The method for Fourier Series involves utilsing the previous Simpson's rule method, and added function definitions for the functions multiplied by the elementary trigonometric functions, which are used in Fourier series.

```
def cosf(k, t, T):
    v = f(t, k, T) * np.cos(t*k*(2*np.pi/T))
```

```
    return v


def sinf(k, t, T):
    u = f(t, k, T) * np.sin(t*k*(2*np.pi/T))
    return u
```

Then a function is defined to find, save and sum the Fourier coefficients, multiplied by the relative trigonometric function, as in Eqs. 5, 6, 7.

```
def Fourier(T, t):
    Series = 0
    kvals = np.arange(0, 10, 1)
    avals = bvals = []
    for k in kvals:
        if k == 0:
            a0 = (F(0, T, f, k, T))*(1/T)
            Series += a0
        if k != 0:
            a = (2/T) * (F(0, T, cosf, k, T))
            avals.append(a)
            b = (2/T) * (F(0, T, sinf, k, T))
            bvals.append(b)
            Series += a * np.cos(k*(2*np.pi)*t/T) + b * np.sin(k*(2*np.pi)*t/T)

    return Series, avals, bvals, a0
```

This was followed by a function graphing the function, both before and after expansion, as well as the coefficients. This can be found in the appendix.

### 3.3 Square, Rectangular Waves

The code for the Fourier series of a square wave began by defining the square wave.

```
def func(t, x=0, y=0):
    if t % (2*np.pi) == 0 or t % (2*np.pi) == np.pi or t == 0:
        g = 0
    elif t % (2*np.pi) <= np.pi:
        g = 1
    else:
        g = -1
    return g
```

which produces a repeating square wave. The previously outlined function for finding and evaluating Fourier series of functionswas used on this function. We used a series of f-strings to print the coefficient values and their expected values.

```
a0, a, b = fourier_coeffs(T, MAX_K)
for k in range(MAX_K):
    if k == 0:
        print(f"a[{0}]={np.round(a0, decimals=4)} expected 0")
    else:
        print(f"a[{k}]={np.round(a[k-1], decimals=4)} expected 0")
        if k % 2 == 0:
            print(f"b[{k}]={np.round(b[k-1], decimals=4)} expected 0")
```

```
        else:
            print(f"b[{k}]={np.round(b[k-1], decimals=4)} expected {4 / np.pi / k}")
```

Rectangular waves are similar, requiring only a constant $\tau$ to be defined beforehand.

```
def func(t, x=0, y=0):
if 0 <= t <= TAU * 2 * np.pi:
        result = 1
    else:
        result = -1
    return result
```

A similar method was used for the expected values, which can be found in the appendix.

### 3.4 Discrete Fourier Transform

The next section involved creating code to find the discrete fourier transform of a function. The code for this entailed sums over the previously mentioned, discrete time variable $m$, finding $F_n$ for each value of n in a range, as in Eq. 12.

```
def dft(N, func_vals):
    fn_real = []
    fn_imag = []
    fn = []
    for n in range(0, N):
        real_val = imag_val = 0
        for m in range(0, N):
            real_val += func_vals[m] * np.cos(2 * np.pi * m * n / N)
            imag_val += -func_vals[m] * np.sin(2 * np.pi * m * n / N)
        fn_real.append(real_val)
        fn_imag.append(imag_val)
        fn.append(complex(real_val, imag_val))
    return fn, fn_real, fn_imag
```

Also, the opposing back transform (Eq. 11) was constructed in a similar manner.

```
def back_transform(N, fn_real, fn_imag):
    fm_real = []
    fm_imag = []
    for m in range(0, N):
        value = 0
        for n in range(0, N):
            fn = complex(fn_real[n], fn_imag[n])
            theta = 2 * np.pi * m * n / N
            value += fn * complex(np.cos(theta), np.sin(theta))
        fm_real.append(np.real(value) / N)
        fm_imag.append(np.imag(value) / N)
    return fm_real, fm_imag
```

A function was also created to evaluate the real and imaginary components of $F_n$ as a function of $n$, for a given $m$.

```
def DFTFunc_n(N, nlist, m, func_vals):
    Fnn_real = []
    Fnn_imag = []
    for n in nlist:
```

```
        Fnn_real.append(func_vals[m] * np.cos(2 * np.pi * m * n / N))
        Fnn_imag.append(-func_vals[m] * np.sin(2 * np.pi * m * n / N))
    return Fnn_real, Fnn_imag
```

Next, a function was created to analyse a given function using the functions described above, and make plots as part of this analysis. As this is an extremely large function it is better left in the appendix.

# 4 Results

## 4.1 Result 1

The first result concerns the accuracy of Simpson's rule as a method of numerical integration. For even numbers of steps, the method is almost completely accurate. For the simple $e^x$ function, by the second step, the method reaches three significant figures of accuracy, only getting more accurate as the number of even steps increases. Conversely, for odd numbers of steps, the method is significantly less accurate, although it too approaches accurate values.

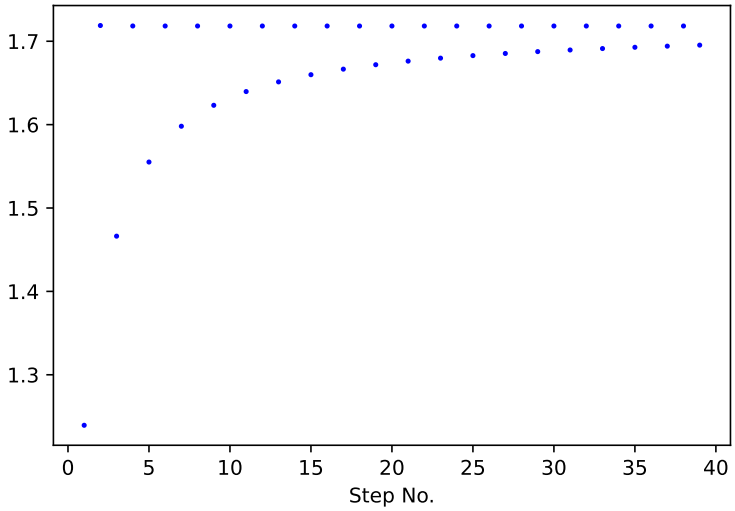


Figure 3: Even vs odd step number error.

As well as this, there was another strange result stemming from the accuracy of the numerical integration. Strangely, as the number of steps increases, some outliers are consistantly produced, and while this could indicate sub-optimal execution of the method, it was taken note of and plotted regardless. The inaccuracy of these outliers also decreases as step number increases.

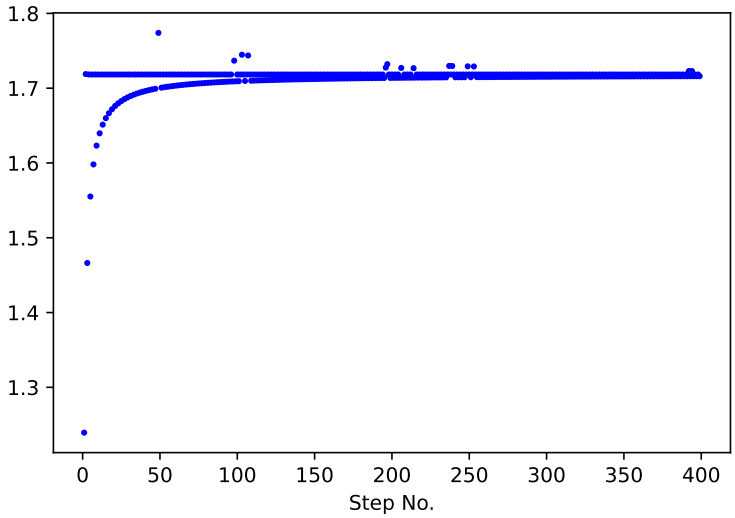


Figure 4: Outlier error.

## 4.2 Result 2

The next result stems from the Fourier series expansion. As can be seen in figure 5, the fourier series of a non-trigonometric function only ever approaches the actual function.

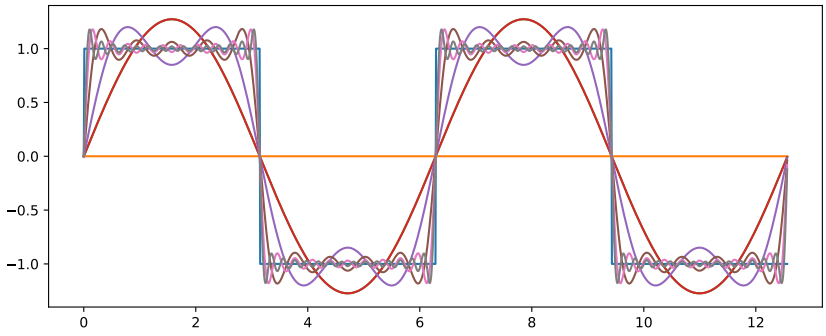


Figure 5: Illustration of fourier expansion approaching the function as $k$ becomes large for a square wave.

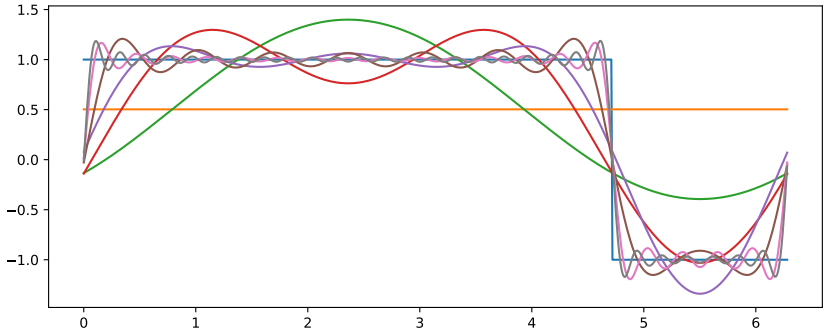

Figure 6: Similar to figure 5, however the rectangular wave stretches out one side of the wave, allows for better illustration of approximation, ringing.

However when more and more terms are taken into account ($k$ becomes large), and the expansion approaches the function, for any finite number there is an amount of 'ringing' around the discontinuities in the actual function.

## 4.3 Result 3

For the analysis of functions using the discrete Fourier transform, we first analysed simple trigonometric function, with large numbers of sampling points.

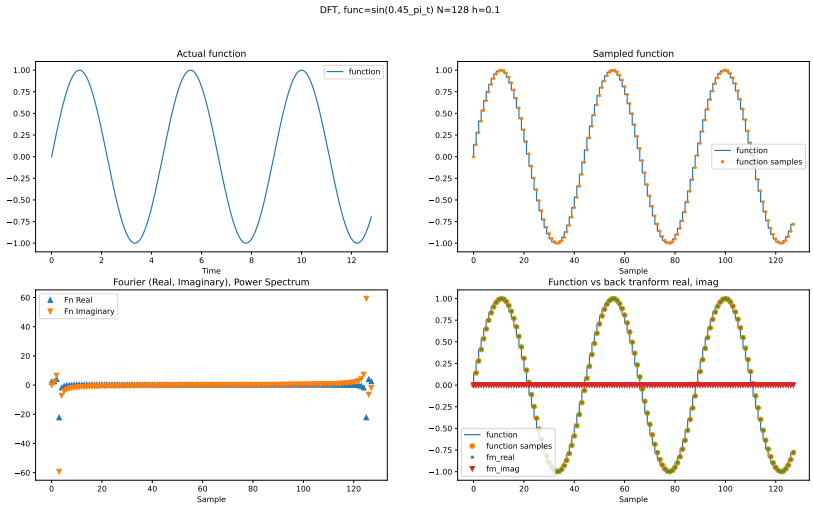


Figure 7: Analysis of the function $f(t) = \sin\left(\frac{9\pi t}{20}\right)$

Clearly the samples decently approximate the function, and the back transform returns the sampled function. The symmetry of the $F_n$ values, $F_{N-n} = F_n^*$, can also be observed in the frequency spectrum. Below are more plots of function analysis for different functions.

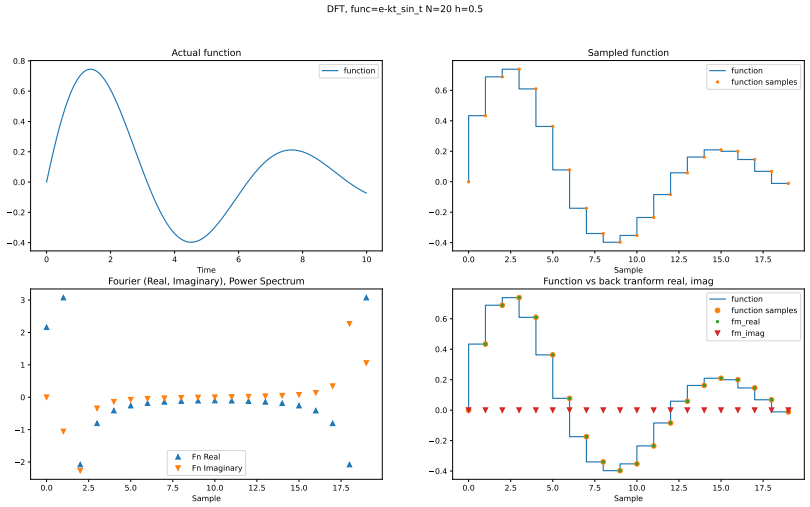


Figure 8: Analysis of $f(t) = e^{-kt}\sin(t)$

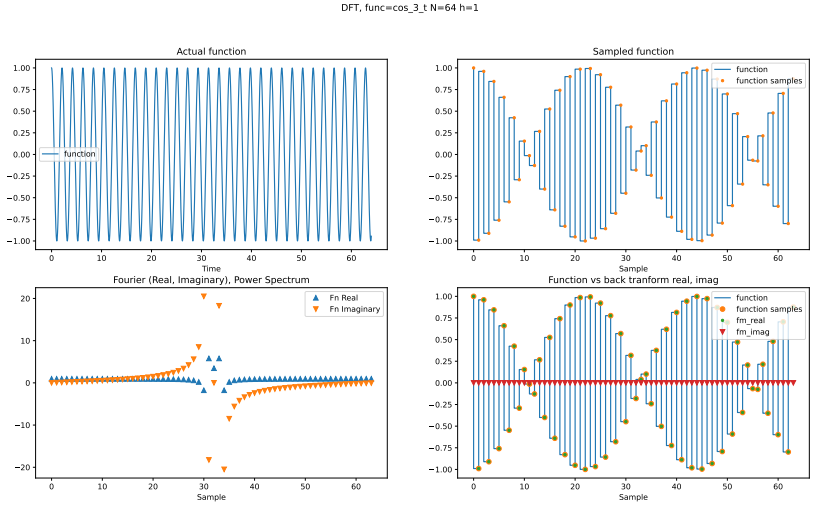


Figure 9: Analysis of $f(t) = \cos(3t)$

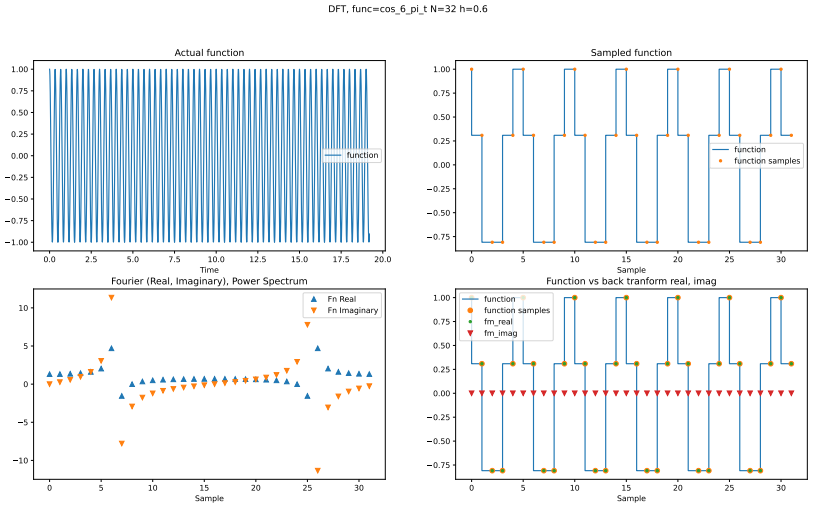


Figure 10: Analysis of $f(t) = \cos(6\pi t)$

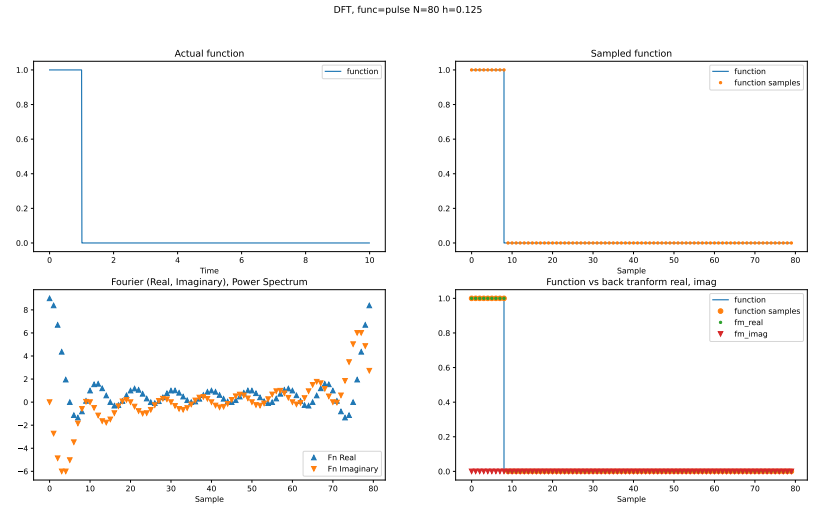


Figure 11: Analysis of a pulse function.

## 4.4 Result 4

An interesting result also emerges from, high frequency functions or low sample number analysis, an effect called aliasing, where the sampled points produce one or more lower frequency waves in the sampled function. This is especially evident in figures 9 10. For the used sample sizes over the range of time, figure 9 produces a wave of periodic amplitude, where as figure 10 just produces a low frequency wave. However, for figure 10 I also noticed significant aliasing, especially at low brightness or when zoomed out, as well as when the plot in motion on the screen.

# 5 Conclusion

To conclude, several interesting reults were found regarding numerical integration, Fourier expansion, Fourier transforms and sampling on periodic functions.

Firstly, numerical integration using Simpson's rule was found to be extremely efficient, although with some strange outliers. I would suppose that the error of these outliers decreased over time as they are compensated for by having a smaller effect on the result as more steps are taken. The Fourier expansion of square and rectangular waves I found to be very interesting, with the most interesting part beingthe ringing around the discontinuities, which are impossible to compensate for with a finite number of coefficients. This presents a problem in electronics, in producing DC current. Finally, the result I found most interesting from the discrete Fourier transform was the aliasing found in sampling of poor quality for a function, which, while a strange effect, makes sense.

# 6 Bibliography

2023. 2nd year Physics Lab Manual. School of Physics, Trinity College Dublin, Dublin 2.

# 7 Appendix

```
import numpy as np

def integrate(func, a, b, k, T, n=10):
    """Integrate function func, between limits (a,b), in n steps, using Simpson's Rule."""
    h = (b - a)/n
    a_to_b = np.arange(a, b, h)
    j1 = j2 = 0
    for j in range(1, n):
        if j % 2 == 0:
            j1 += func(a_to_b[j], k, T)
        else:
            j2 += func(a_to_b[j], k, T)

    result = (h/3) * (func(a, 0, 0) + (2 * j1) + (4 * j2) + func(b, 0, 0))
    return result


# Correct to 5 decimal places in 10 steps (1.71828).

def testfunc(t, x=0, y=0):
    result = np.exp(t)
    return result


n = 10
print(f"{n} steps")
result = integrate(testfunc, 0, 1, 0, 1, n=10)
print(result)

import numpy as np
```

```
import matplotlib.pyplot as plt

INT_STEPS = 500
MAX_K = 6


def func(t, x=0, y=0):
    # result = 1
    # result = t
    # result = np.sin(t)
    # result = np.cos(t) + 3 * np.cos(2 * t) - 4 * np.cos(3 * t)
    # result = np.sin(t) + 3 * np.sin(3 * t) + 5 * np.sin(5 * t)
    result = np.sin(t) + 2 * np.cos(3 * t) + 3 * np.sin(5 * t)
    return result


def cosf(t, k, T):
    result = func(t) * np.cos(t * k * (2 * np.pi / T))
    return result


def sinf(t, k, T):
    result = func(t) * np.sin(t * k * (2 * np.pi / T))
    return result


def integrate(f, a, b, k, T, n=INT_STEPS):
    h = (b - a)/n
    a_to_b = np.arange(a, b, h)
    j1 = j2 = 0
    for j in range(1, n):
        if j % 2 == 0:
            j1 += f(a_to_b[j], k, T)
        else:
            j2 += f(a_to_b[j], k, T)

    result = (h/3) * (f(a, 0, T) + (2 * j1) + (4 * j2) + f(b, 0, T))
    return result


def fourier_coeffs(T, maxk=10):
    a = []
    b = []
    for k in np.arange(0, maxk):
        if k == 0:
            a0 = integrate(func, 0, T, k, T) * (1 / T)
        else:
            a.append(integrate(cosf, 0, T, k, T) * (2 / T))
            b.append(integrate(sinf, 0, T, k, T) * (2 / T))
    return a0, a, b
```

```
def fourier_series(a0, a, b, t, T, maxk=10):
    series = 0
    for k in np.arange(0, maxk):
        if k == 0:
            series += a0
        else:
            series += (a[k-1] * np.cos(k * (2 * np.pi) * t / T)) + b[k-1] * np.sin(k * (2 *
    return series


T = 2 * np.pi

print(f"Fourier series, max-k={MAX_K}, int-steps={INT_STEPS}")

func_vals = []
fourier_vals = []

a0, a, b = fourier_coeffs(T, MAX_K)
for k in range(MAX_K):
    if k == 0:
        print(f"a[{k}]={np.round(a0, decimals=4)}")
    else:
        print(f"a[{k}]={np.round(a[k-1], decimals=4)}")
        print(f"b[{k}]={np.round(b[k-1], decimals=4)}")

timesteps = np.arange(0, T, 0.01)
for t in timesteps:
    func_vals.append(func(t))
    fourier_vals.append(fourier_series(a0, a, b, t, T, MAX_K))

plt.figure()
plt.plot(timesteps, func_vals)
plt.plot(timesteps, fourier_vals)
plt.show()

import numpy as np
import matplotlib.pyplot as plt

INT_STEPS = 500
MAX_K = 50

def func(t, x=0, y=0):
    if 0 <= t <= np.pi:
        result = 1
    else:
        result = -1
    return result


def cosf(t, k, T):
    result = func(t) * np.cos(t * k * (2 * np.pi / T))
```

```
    return result


def sinf(t, k, T):
    result = func(t) * np.sin(t * k * (2 * np.pi / T))
    return result


def integrate(f, a, b, k, T, n=INT_STEPS):
    h = (b - a)/n
    a_to_b = np.arange(a, b, h)
    j1 = j2 = 0
    for j in range(1, n):
        if j % 2 == 0:
            j1 += f(a_to_b[j], k, T)
        else:
            j2 += f(a_to_b[j], k, T)

    result = (h/3) * (f(a, 0, T) + (2 * j1) + (4 * j2) + f(b, 0, T))
    return result


def fourier_coeffs(T, maxk=10):
    a = []
    b = []
    for k in np.arange(0, maxk):
        if k == 0:
            a0 = integrate(func, 0, T, k, T) * (1 / T)
        else:
            a.append(integrate(cosf, 0, T, k, T) * (2 / T))
            b.append(integrate(sinf, 0, T, k, T) * (2 / T))
    return a0, a, b


def fourier_series(a0, a, b, t, T, maxk=10):
    series = 0
    for k in np.arange(0, maxk):
        if k == 0:
            series += a0
        else:
            series += (a[k-1] * np.cos(k * (2 * np.pi) * t / T)) + b[k-1] * np.sin(k * (2 *
    return series


T = 2 * np.pi

print(f"Square wave, max-k={MAX_K}, int-steps={INT_STEPS}")

a0, a, b = fourier_coeffs(T, MAX_K)
for k in range(MAX_K):
    if k == 0:
```

```
        print(f"a[{k}]={np.round(a0, decimals=4)} expected 0")
    else:
        print(f"a[{k}]={np.round(a[k-1], decimals=4)} expected 0")
        if k % 2 == 0:
            print(f"b[{k}]={np.round(b[k-1], decimals=4)} expected 0")
        else:
            print(f"b[{k}]={np.round(b[k-1], decimals=4)} expected {4 / np.pi / k}")

func_vals = []
fourier_vals = {}

timesteps = np.arange(0, T, 0.01)
for t in timesteps:
    func_vals.append(func(t))

for i in (1, 2, 3, 5, 10, 20, 30):
    fourier_vals[i] = []
    for t in timesteps:
        fourier_vals[i].append(fourier_series(a0, a, b, t, T, i))

plt.figure()
plt.plot(timesteps, func_vals)
for i in (1, 2, 3, 5, 10, 20, 30):
    plt.plot(timesteps, fourier_vals[i])
plt.savefig("square-wave.png")
plt.show()

INT_STEPS = 500
MAX_K = 50

TAU = 0.75
ALPHA = 1 / TAU

def func(t, x=0, y=0):
    if 0 <= t <= TAU * 2 * np.pi:
        result = 1
    else:
        result = -1
    return result


def cosf(t, k, T):
    result = func(t) * np.cos(t * k * (2 * np.pi / T))
    return result


def sinf(t, k, T):
    result = func(t) * np.sin(t * k * (2 * np.pi / T))
    return result


def integrate(f, a, b, k, T, n=INT_STEPS):
```

```
    h = (b - a)/n
    a_to_b = np.arange(a, b, h)
    j1 = j2 = 0
    for j in range(1, n):
        if j % 2 == 0:
            j1 += f(a_to_b[j], k, T)
        else:
            j2 += f(a_to_b[j], k, T)

    result = (h/3) * (f(a, 0, T) + (2 * j1) + (4 * j2) + f(b, 0, T))
    return result


def fourier_coeffs(T, maxk=10):
    a = []
    b = []
    for k in np.arange(0, maxk):
        if k == 0:
            a0 = integrate(func, 0, T, k, T) * (1 / T)
        else:
            a.append(integrate(cosf, 0, T, k, T) * (2 / T))
            b.append(integrate(sinf, 0, T, k, T) * (2 / T))
    return a0, a, b


def fourier_series(a0, a, b, t, T, maxk=10):
    series = 0
    for k in np.arange(0, maxk):
        if k == 0:
            series += a0
        else:
            series += (a[k-1] * np.cos(k * (2 * np.pi) * t / T)) + b[k-1] * np.sin(k * (2 *
    return series


T = 2 * np.pi

print(f"Rectangular wave, tau={TAU}, alpha={ALPHA}")

a0, a, b = fourier_coeffs(T, MAX_K)
for k in range(MAX_K):
    if k == 0:
        expected = np.round((2 / ALPHA) - 1, decimals=4)
        print(f"a[{k}]={np.round(a0, decimals=4)} expected {expected}")
    else:
        expected = np.round((2 / (k * np.pi)) * np.sin((2 * k * np.pi / ALPHA)), decimals=
        print(f"a[{k}]={np.round(a[k-1], decimals=4)} expected {expected}")
        expected = np.round((2 / (k * np.pi)) * (1 - np.cos(2 * k * np.pi / ALPHA)), decim
        print(f"b[{k}]={np.round(b[k-1], decimals=4)} expected {expected}")

func_vals = []
```

```
fourier_vals = {}

timesteps = np.arange(0, T, 0.01)
for t in timesteps:
    func_vals.append(func(t))

for i in (1, 2, 3, 5, 10, 20, 30):
    fourier_vals[i] = []
    for t in timesteps:
        fourier_vals[i].append(fourier_series(a0, a, b, t, T, i))

plt.figure()
plt.plot(timesteps, func_vals)
for i in (1, 2, 3, 5, 10, 20, 30):
    plt.plot(timesteps, fourier_vals[i])
plt.savefig("rectangular-wave.png")
plt.show()

import math
import numpy as np
import matplotlib.pyplot as plt


def dft(N, func_vals):
    fn_real = []
    fn_imag = []
    fn = []
    for n in range(0, N):
        real_val = imag_val = 0
        for m in range(0, N):
            real_val += func_vals[m] * np.cos(2 * np.pi * m * n / N)
            imag_val += -func_vals[m] * np.sin(2 * np.pi * m * n / N)
        fn_real.append(real_val)
        fn_imag.append(imag_val)
        fn.append(complex(real_val, imag_val))
    return fn, fn_real, fn_imag


def back_transform(N, fn_real, fn_imag):
    fm_real = []
    fm_imag = []
    for m in range(0, N):
        value = 0
        for n in range(0, N):
            fn = complex(fn_real[n], fn_imag[n])
            theta = 2 * np.pi * m * n / N
            value += fn * complex(np.cos(theta), np.sin(theta))
        fm_real.append(np.real(value) / N)
        fm_imag.append(np.imag(value) / N)
    return fm_real, fm_imag
```

```
def analyse(func, N, h, funcstr):
    tau = N * h

    print(f"\nDFT, N={N} h={h}")
    print(f"Sample frequency: {1/h} Hz")

    w1 = 2 * np.pi / tau
    print(f"Fundamental frequency {w1=}")
    wN = (1 / (2 * h)) - (1 / (N * h))
    print(f"Nyquist frequency {wN=}")

    # Ideal sampling interval
    freq = 6 * np.pi / (2 * np.pi)
    print(f"Actual frequency: {freq} Hz")
    period = 1 / freq
    print(f"Actual period: {period} s")
    # Tau equals period of function = N * h.
    h_ideal = period / N
    print(f"Ideal sampling interval: {h_ideal}")
    print(f"Ideal sampling frequency: {1/h_ideal} Hz")

    actual_func_vals = []
    timesteps = np.arange(0, N * h, 0.01)
    for t in timesteps:
        actual_func_vals.append(func(t))

    func_vals = []
    sample_times = []
    for m in range(0, N):
        sample_times.append(m * h)
    for t in sample_times:
        func_vals.append(func(t))
    fn, fn_real, fn_imag = dft(N, func_vals)

    samples = range(0, N)

    fig, axs = plt.subplots(4)
    fig.set_figwidth(10)
    fig.set_figheight(20)
    fig.suptitle(f"DFT, func={funcstr} N={N} h={h}")

    axs[0].plot(timesteps, actual_func_vals, label='function')
    axs[0].legend()
    axs[0].set_xlabel('Time')
    axs[0].set_title("Actual function")

    axs[1].plot(samples, func_vals, drawstyle='steps-pre', label='function')
    axs[1].plot(samples, func_vals, '.', label='function samples')
    axs[1].legend()
    axs[1].set_xlabel('Sample')
    axs[1].set_title("Sampled function")
```

```
    # (10) Power spectrum
    pn = []
    for n in range(0, N):
        pn.append(fn_real[n] ** 2 + fn_imag[n] ** 2)

    axs[2].plot(samples, fn_real, '.', label='fn_real')
    axs[2].plot(samples, fn_imag, 'v', label='fn_imag')
    axs[2].plot(samples, pn, '^', label='power')
    axs[2].legend()
    axs[2].set_xlabel('Sample')
    axs[2].set_title("Fourier real, imag, power")

    # (10) Back transform
    fm_real, fm_imag = back_transform(N, fn_real, fn_imag)

    axs[3].plot(samples, func_vals, drawstyle='steps-pre', label='function')
    axs[3].plot(samples, func_vals, 'o', label='function samples')
    axs[3].plot(samples, fm_real, '.', label='fm_real')
    axs[3].plot(samples, fm_imag, 'v', label='fm_imag')
    axs[3].legend()
    axs[3].set_xlabel('Sample')
    axs[3].set_title("Function vs back tranform real, imag")

    plt.savefig(f"{funcstr}_{N}_{h}.png")
    plt.show()


# 3.(1)-(8) f(t) = sin(0.45 * pi * t)
def func1(t):
    return np.sin(0.45 * np.pi * t)

N = 128  # Number of samples.
h = 0.1  # Sampling interval (s).
analyse(func1, N, h, "sin_0.45_pi_t")

# (9)(10) f(t) = cos(6 * pi * t)
# (11) f(t) = cos(6 * pi * t)
def func2(t):
    return np.cos(6.0 * np.pi * t)

N = 32
for h in (0.6, 0.2, 0.1, 0.04, 0.01):
    analyse(func2, N, h, "cos_6_pi_t")

# (Supplementary (1) - (4)) f(t) = cos(3 * t)
def func3(t):
    return np.cos(3.0 * t)

h = 1
for N in (8, 16, 32, 64):
```

```
    analyse(func3, N, h, "cos_3_t")


# (Supplementary (5) - (8)) f(t) = e^-kt * sin(t)
def func4(t):
    k = 0.2
    return np.sin(t) * math.exp(-k * t)

N = 20
h = 0.5
analyse(func4, N, h, "e-kt_sin_t")


# (Supplementary (9) - (10)) f(t) = 1, 0<t<tau, 0, t<0, t>tau
def func5(t):
    tau = 1
    if t < 0 or t > tau:
        return 0
    else:
        return 1

N = 20
h = 0.5
for N,h in ((10, 1), (20, 0.5), (40, 0.25), (80, 0.125)):
    analyse(func5, N, h, "pulse")
```