BRYN MAWR COLLEGE

Department of Physics

Computational Module 9 – Fourier Analysis

Prerequisite modules: Module 4 – Numerical Integration
Estimated completion time: 4-6 hours
Learning objectives: Become familiar with some discrete versions of the Fourier transform, learn of some applications of Fourier transforms, compare the efficiencies of various implementations of the Fourier transform

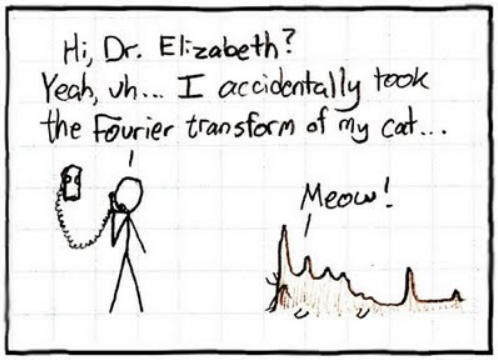


Image credit: xkcd.com

Fourier analysis is an essential tool for detecting patterns in and origins of signals, be they sounds, images, video, or scientific data. As a few specific examples, it can be used to identify the overtones produced by a musical instrument, remove noise from images, reconstruct a crystal's structure from its diffraction pattern, and identify sources in sonar signals. (See the Wikipedia page on Fourier analysis for other examples.) This module explores some of the theory underlying this tool and describes the main numerical method to implement it, the Fast Fourier Transform, regarded as one of the most important numerical algorithms known.[1]

[1] Module adapted from *Computational Physics* by M. Newman.

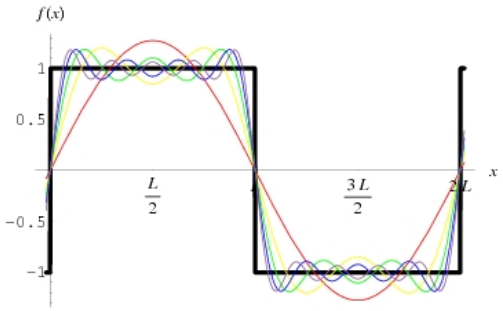


Figure 1: Fourier series of a square wave

9.1 The Fourier Series

As you may know, the Fourier series (FS) is a very powerful tool for representing periodic functions (functions whose form repeats over fixed intervals), for breaking up signals into their component frequencies, and for modeling wave behavior. (Figure 1 shows the results of combining more and more terms in the Fourier series representation of a square wave.[2] An animated version can be found on the Wikipedia page for the Fourier transform.) There are different versions of the Fourier series, depending on the symmetry of the function being modeled. If you are not familiar with the Fourier series, you should consult a text that covers the topic.[3]

To start, consider periodic functions defined on an interval $0 \leq x < L$. [Note that a section of a *non*periodic function may be made into a periodic function simply by repeating copies of the section, as shown in Figure 2.[4] There, the parts of the original function lying outside the region of interest from $x = 0$ to $x = L$ are removed (the solid gray lines), and copies of the section within that region are repeated outside the region (dashed gray lines). One can now use the Fourier method to model the part within the region of interest.] The FS that would represent a periodic function $f(x)$ that's even (symmetric) about the midpoint of the interval at $x = L/2$ is

$$f(x) = \sum_{k=0}^{\infty} a_k \cos\left(\frac{2\pi k x}{L}\right). \tag{1}$$

For an *odd* function, antisymmetric about the midpoint of the interval, the appropriate FS is

$$f(x) = \sum_{k=1}^{\infty} b_k \sin\left(\frac{2\pi k x}{L}\right). \tag{2}$$

The a_k and b_k coefficients depend on the specific functions being represented. Note that an even function is represented by cosine functions, which also would be even relative to the point $x = L/2$, while an odd function would be represented by sine functions, which are odd about the midpoint.

[2] Image credit: Weisstein, Eric W. "Fourier Series." From MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/FourierSeries.html

[3] E.g., *Mathematical Methods in the Physical Sciences*, by Mary Boas.

[4] From *Computational Physics*, by Mark Newman.

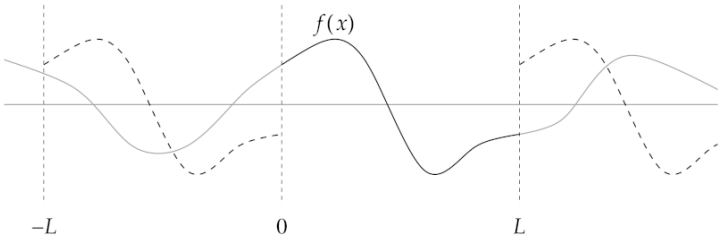


Figure 2: Making a periodic version of a function

Breakpoint 1: Why does the sum over sine functions begin at $k = 1$, while the sum over cosine functions starts at $k = 0$?

An arbitrary function with no particular symmetry would be represented by a combination of the two prior expressions:

$$f(x) = \sum_{k=0}^{\infty} a_k \cos\left(\frac{2\pi kx}{L}\right) + \sum_{k=1}^{\infty} b_k \sin\left(\frac{2\pi kx}{L}\right). \tag{3}$$

A more compact form can be obtained by remembering that cos and sin are related to the complex exponential $e^{i\theta}$; in particular

$$\cos(\theta) = \tfrac{1}{2}\left(e^{i\theta} + e^{-i\theta}\right) \qquad \text{and} \qquad \sin(\theta) = -\tfrac{1}{2}i\left(e^{i\theta} - e^{-i\theta}\right). \tag{4}$$

Substituting these in Eq. (3) and collecting similar terms, get

$$f(x) = \sum_{k=-\infty}^{\infty} c_k \exp\left(i\frac{2\pi kx}{L}\right), \tag{5}$$

where

$$c_k = \begin{cases} \frac{1}{2}\left(a_k - ib_k\right) & \text{if } k > 0, \\ a_0 & \text{if } k = 0, \\ \frac{1}{2}\left(a_{-k} + ib_{-k}\right) & \text{if } k < 0. \end{cases} \tag{6}$$

Note that the sum runs over negative as well as positive values of k [to account for the $\exp(-i\theta)$ terms that appear in Eqs. (4)]. Be aware that, in general, the c_k's are *complex* numbers.

The goal here is to find the c_k values, known as ***Fourier coefficients***, which depend on the function $f(x)$ being modeled, and which fully specify the Fourier expansion. The coefficients are found using the orthogonality of the cosine and sine functions over the interval $0 \le x < L$, by multiplying Eq. (5) by an exponential factor with a different k value, $\exp(i2\pi k'x/L)$, and integrating over the

interval:

$$\int_0^L f(x) \exp\left(-i\frac{2\pi k' x}{L}\right) \mathrm{d}x = \sum_{k'=-\infty}^{\infty} c_{k'} \int_0^L \exp\left[i\frac{2\pi(k'-k)x}{L}\right] \mathrm{d}x. \tag{7}$$

The integral on the right-hand side vanishes if $k' \neq k$; if $k' = k$ then it yields L. Thus, the right-hand side picks out the particular value of $c_{k'}$ for which $k' = k$, namely c_k, and so we have

$$c_k = \frac{1}{L} \int_0^L f(x) \exp\left(-i\frac{2\pi k x}{L}\right) \mathrm{d}x. \tag{8}$$

Breakpoint 2: By direct calculation, prove the claims made about the integral on the right-hand side of Eq. (7) when $k' \neq k$ and when $k' = k$.

A wave with a wavelength λ normally is written in complex form as $\exp(-i\,2\pi x/\lambda)$, so by comparison with the expression above for c_k we see that k/L in that expression is playing the role of $1/\lambda$. Thus, larger k corresponds to smaller λ, and so c_k coefficients with larger values of k are associated with higher spatial frequency (smaller λ) oscillatory terms. Since Eq. (5) shows that $f(x)$ can be constructed as a sum of such oscillatory functions of different (spatial) frequencies, then the magnitude of c_k tells us how much of its corresponding oscillatory function is incorporated in $f(x)$. That is, functions characterized by large values of c_k's with large k's have significant high-frequency components. (We could recast this entire discussion in terms of time t rather than position x, in which case the comments about spatial frequency would apply to the traditional frequency).

9.2 The Discrete Fourier Transform

When the integral in Eq. (8) cannot be computed analytically, it must be done numerically. One can use one of the methods of Module 5; we will explore the trapezoidal method, with N "slices" (i.e., trapezoids) each of width $h = L/N$. From that earlier module, recall the following equation for the integral from $x = a$ to $x = b$ of a function $F(x)$:

$$I\,(F; a, b) = h\left[\tfrac{1}{2}F(a) + \tfrac{1}{2}F(b) + \sum_{k=1}^{N-1} F(a+kh)\right]. \tag{9}$$

In the present case, with $F(x) = \dfrac{1}{L} f(x) \exp\left(-i\dfrac{2\pi k x}{L}\right)$, $a = 0$, and $b = L$, this gives

$$c_k = \frac{1}{L}\frac{L}{N}\left[\tfrac{1}{2}f(0) + \tfrac{1}{2}f(L) + \sum_{n=1}^{N-1} f(x_n) \exp\left(-i\frac{2\pi k x_n}{L}\right)\right], \tag{10}$$

where $x_n = n(L/N)$, with $n = 0, 1, \ldots, N$, are the $N+1$ sample points equally spaced in the interval from 0 to L that mark the boundaries of the trapezoidal slices. Since we assume the function f is periodic, then $f(0) = f(L)$, and this expression simplifies to

$$c_k = \frac{1}{N} \sum_{n=0}^{N-1} f(x_n) \exp\left(-i \frac{2\pi k x_n}{L}\right), \tag{11}$$

where the values of f at the endpoints have been incorporated as a new $n = 0$ term in the sum.

In the common situation that $f(x)$ represents some kind of signal — a sound wave, the output from a digital oscilloscope, etc. — it's natural to define the function values at the sample points as $f(x_n) \equiv y_n$. Using this and the earlier definition of x_n, we can introduce a slight variation of c_k:

$$C_k \equiv N c_k = \sum_{n=0}^{N-1} y_n \exp\left(-i \frac{2\pi k n}{N}\right). \tag{12}$$

Note that this expression, the ***discrete Fourier transform*** (or DFT) of the samples y_n, no longer depends on L or the specific positions of the sample points (but they must be equally spaced).

Amazingly, although the trapezoidal rule gives only approximate results for integrals in general, the DFT is exact in a sense: it turns out (after some algebra) that one can show

$$y_n = \frac{1}{N} \sum_{k=0}^{N-1} C_k \exp\left(i \frac{2\pi k n}{N}\right). \tag{13}$$

This expression is the ***inverse discrete Fourier transform***, or iDFT. It reveals that we can recover the values of the function samples y_n from the C_k *exactly* (apart from rounding errors).

Note that Eq. (13) is similar to the earlier Fourier series expression Eq. (5), apart from a factor of $1/N$ in front and the fact that the sum extends over only a *finite* set of values (from $k = 0$ to $N - 1$). This is a good thing, since we can't do an infinite sum on a computer!

Also notice, though, that Eq. (13) gives the values of $f(x)$ only at the sample points x_n, whereas Eq. (5) in principle gives the values of $f(x)$ everywhere. The price paid for reducing the sum to a finite one is that it tells us the function values at only a finite number of positions. This means that two different functions that happen to have the same y_n values will have the *same DFT*, even though they may behave completely differently between the sample points. This is not a problem in the common situation we encounter in the physical sciences, where we're interested in analyzing a signal consisting of a discrete set of sampled values with nothing in between.

The discussion of the DFT up to this point has been valid for both real and complex functions $f(x)$. In the usual case, in which we want to analyze a real function, the expansion can be further simplified. It can be shown that, for real y_n, the C_k's obey $C_{N-r} = C_r^*$, where $1 \leq r < \frac{1}{2}N$. For

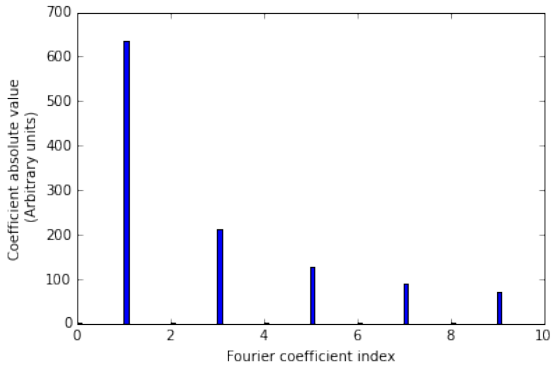


Figure 3: Discrete Fourier transform of a square wave

example, $C_{N-1} = C_1^*$, $C_{N-2} = C_2^*$, and so on. This means that in the case of a real function, only the coefficients C_k with $0 \le k \le \frac{1}{2}N$ need to be computed directly; the other half of the coefficients are then found simply by complex conjugation.

If N is even, there are $\frac{1}{2}N + 1$ coefficients to be determined; if N is odd, there are $\frac{1}{2}(N+1)$ of them. The two mathematical expressions, $\frac{1}{2}N + 1$ and $\frac{1}{2}(N+1)$, actually can be obtained from the single Python expression `N//2 + 1`, where the `//` operator divides one integer by another and gives an integer result, rounding down if necessary to do so.

Breakpoint 3: (a) Compute `N//2 + 1` and $\frac{1}{2}(N+1)$ for $N = 7$. Are they equal? (b) Also compute `N//2 + 1` and $\frac{1}{2}N + 1$ for $N = 8$. Are they the same?

Figure 3 shows an example plot of the absolute value of the Fourier transform coefficients, specifically for a square wave like that shown in Figure 1. (Only the first few coefficients are shown.) Notice how the coefficients decrease with increasing index; i.e., k. (Also notice that only coefficients with odd k are nonzero, because of the asymmetry of the square wave pattern.)

A program to compute the DFT coefficients of Eq. (12) for k values in the range $0 \le k \le \frac{1}{2}N$ is:

```
def dft_half(y):
    from numpy import zeros, exp, absolute

    N = len(y)
    numcoeffs = N//2 + 1
    C = zeros(numcoeffs, complex)

    for k in range(numcoeffs):
        for n in range(N):
            C[k] += y[n] * exp(-2j * pi * k * n/N)   # "j" = sqrt(-1) in Python
    return absolute(C)    # returns absolute value of complex number
```

Here, the input signal is represented by `y`, which should be an array. The main part of the code consists of a double `for` loop: the inner loop over `n` performs the sum indicated in Eq. (12) to compute a particular C_k; the outer loop runs over `k` to do that for all of the coefficients. Note that the code returns the absolute value of the coefficients, for easier plotting. (The `numpy` routines return the full complex coefficients, so one would want to take the absolute values before plotting.)

It's worth noting that Eq. (12) is essentially just a matrix multiplication operation: $C_k = \sum_n M_{kn}\, y_n$ ($\mathbf{C} = \mathbf{My}$ in matrix notation), where we think of C_k and y_n as elements of column vectors, and where matrix $\mathbf{M}$ has the elements $M_{kn} = \exp(-i2\pi kn/N)$. Thus, an alternative coding of the DFT, which makes use of `numpy`'s vectorization capabilities, and which computes the coefficients for *all* of the k values at once, is:

```
def dft_vect(x):
    from numpy import asarray, arange, reshape, exp, dot, pi, absolute

    x = asarray(x, dtype=float)   # force x to be array
    N = x.shape[0]                # length of array "x"
    n = arange(N)                 # list 0,...,N-1
    k = n.reshape((N, 1))         # make "k" a column-vector version of "n"
    M = exp(-2 j * pi * k*n / N)  # k*n computes outer product of k & n vectors
    c = dot(M, x)

    return absolute(c)            # matrix-vector product
```

It's not hard to show that shifting the sample points along the x axis by a fixed amount simply results in multiplying the coefficients C_k by a (k-dependent) complex phase factor of the form $e^{i\phi}$, with ϕ real. However, since we're generally interested in the absolute value squared of the C_k's, which represents the ***power spectrum*** of the data, this phase factor is irrelevant ($|e^{i\phi}|^2 = e^{i\phi}e^{-i\phi} = 1$). This means that the positions of the (equally-spaced) sample points do not affect the Fourier transform, so their choice is not critical.

It's important to mention a fundamental limitation on the usefulness of the DFT.[5] This limitation stems from the common-sense observation that if the signal to be Fourier transformed varies much more rapidly than the rate at which it is sampled, then the high-frequency components of the signal will be missed. This notion can be quantified by defining the interval between sample points to be Δ. (The sample points used above were presented as positions in space, x_n, but more often they are moments in time, t_n, so Δ usually represents a time interval.) In any case, there is a special frequency — the ***Nyquist critical frequency*** (which will not be a frequency in the usual sense if the sample points are not samples in time) — associated with that interval, and defined as

[5] This discussion is adapted from *A Survey of Computational Physics* by Landau, Páez and Bordeianu.

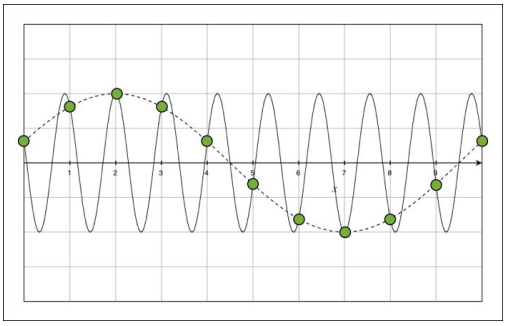


Figure 4: Example of aliasing

$$f_c \equiv \frac{1}{2\Delta}. \tag{14}$$

Breakpoint 4: In terms of the period T of a sine or cosine wave, what should be the sampling interval Δ if the Nyquist frequency is to match the wave's frequency? If the wave is sampled at one peak, where do the other samples occur when the wave is sampled at the Δ you found?

The importance of the Nyquist frequency is that if the Fourier transform of a *continuous* function sampled at intervals Δ vanishes above the Nyquist frequency (i.e., if the c_k associated with frequencies above f_c vanish), then the function is *completely characterized* by the samples y_n, so one can construct a sum involving the sample points that equates *exactly* to the function. This fact is known as the ***sampling theorem***.

This theorem is especially important for experimental physics and engineering, since signals processed by electronic equipment almost always are ***bandwidth limited*** by the equipment to frequencies below some maximum value, call it f_{max}. As long as the signal is sampled at a small enough interval Δ such that the corresponding critical frequency obeys $f_c \geq f_{max}$, then the samples will completely determine the signal.

There is an unfortunate consequence of the sampling theorem as well. It implies that for a continuous function that is *not* bandwidth limited to a frequency lower than f_c, then the frequency components lying outside the range $-f_c < f < f_c$ are spuriously Fourier transformed into that range in a phenomenon known as ***aliasing***. Figure 4 shows what can happen in such a situation: a high-frequency signal (solid sinusoidal wave) is sampled at a rate below its frequency, and the resulting signal appears to be of a lower frequency (dashed wave).[6] Unfortunately, there's not much that can be done to avoid aliasing, other than imposing a bandwidth limit on a signal and then sampling it at appropriately small intervals.

[6]Image credit: http://www.xyobalancer.com/xyobalancerblog/signal_processing_aliasing

9.3 Two-dimensional Fourier Transforms

While one-dimensional Fourier transforms (1dFTs) are suitable for analyzing audio and other signals dependent on a single variable (e.g., time), the analysis of images requires two-dimensional Fourier transforms (2dFTs). These can find periodic structures and other features in images. Fortunately, 2dFTs are just extensions of 1dFTs.

Starting with an $M \times N$ array of data values (i.e. sample points) y_{mn} representing the brightness values of image pixels, one performs a 1dFT as in Eq. (12) on each of the M rows of the image array, giving an array of coefficients:

$$C'_{ml} = \sum_{n=0}^{N-1} y_{mn} \exp\left(-i\frac{2\pi ln}{N}\right). \tag{15}$$

For the m^{th} row, there are N coefficients, one for each value of l. We now transform the set of coefficients with the same value of l across all M rows (i.e., the l^{th} column) to get

$$C_{kl} = \sum_{m=0}^{M-1} C'_{ml} \exp\left(-i\frac{2\pi km}{M}\right). \tag{16}$$

The previous expression for the C'_{ml} can be substituted here to give

$$C_{kl} = \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} y_{mn} \exp\left[-i2\pi\left(\frac{km}{M} + \frac{ln}{N}\right)\right]. \tag{17}$$

The inverse transform corresponding to this result is

$$y_{mn} = \frac{1}{MN}\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} C_{kl} \exp\left[i2\pi\left(\frac{km}{M} + \frac{ln}{N}\right)\right]. \tag{18}$$

9.4 The Discrete Cosine Transform

As noted in section 9.1, a one-dimensional periodic function that's symmetric about the midpoint of the interval $0 \le x < L$ on which it's defined can be represented by the cosine transform of Eq. (1). This symmetry condition seems restrictive, and would appear to mean that the cosine transform is not useful for general functions. However, any function defined in a finite interval can be "duplicated" and extended into a larger region such that the resulting function has definite symmetry. Figure 5 depicts a function lacking symmetry, and originally defined within the shaded region shown, which has been copied and "flipped" appropriately so as to end up with an extended function that's symmetric over the entire region shown in the figure. If a cosine transform is now

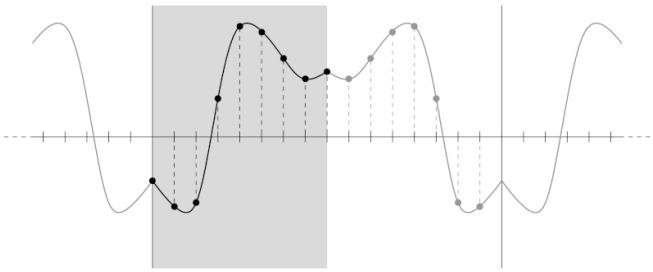


Figure 5: Making a symmetric version of a function

performed on this extended function, that transform will match the original function within the region in which it was originally defined.

The ***discrete cosine transform*** (DCT) of the extended function can be obtained from Eq. (12) in the situation that the samples y_n are symmetric about $\frac{1}{2}L$; i.e., when $y_r = y_{N-r}$, $r = 0, 1, 2, \ldots, \frac{1}{2}N$. If Eq. (12) is separated into two sums — one from $n = 0$ to $n = \frac{1}{2}N$, and the other from $n = \frac{1}{2}N+1$ to $n = N$ — and some algebra is performed, one obtains:

$$C_k = y_0 + y_{N/2} \cos\left[\frac{2\pi k(N/2)}{N}\right] + 2 \sum_{n=1}^{\frac{1}{2}N-1} y_n \cos\left(\frac{2\pi kn}{N}\right). \tag{19}$$

The ***inverse discrete cosine transform*** (iDCT) is

$$y_n = \frac{1}{N}\left\{c_0 + c_{N/2} \cos\left[\frac{2\pi n(N/2)}{N}\right] + 2 \sum_{k=1}^{\frac{1}{2}N-1} c_k \cos\left(\frac{2\pi kn}{N}\right)\right\}. \tag{20}$$

Note: in both of these expressions the equally-spaced sample points are assumed to include the endpoints of the interval ($x = 0$ and $x = L$). Also note that the second term in each expression could be simplified, but they are left as is to show their similarity to the terms in sums. Of course, these are referred to as cosine transforms because they involve cosine functions rather than exponentials.

It's worth mentioning that a common application of Fourier transform methods is in ***data compression***, as implemented by the JPEG, MPEG and MP3 formats (among others). These compression methods involve performing a discrete cosine transform of the data and then disregarding the terms with small coefficients, which contribute little to the overall signal. The remaining smaller set of coefficients are stored, and the original information is recovered (approximately) by performing the inverse transform with the reduced set of coefficients.

9.5 The Fast Fourier Transform

While the first code example in section 9.2 should work fine for determining the Fourier transform of discrete data, it is not especially fast. Note that it loops through N values inside another loop over roughly $N/2$ values; thus, this computation is of order $\mathrm{O}(N^2)$. For computationally-intensive transforms, this approach becomes prohibitively slow, and we need another method. The faster method of choice (discovered in one form by Gauss in 1805!) is the famous ***fast Fourier transform*** (FFT).

This method is simplest when the number of sample points is a power of 2, so assume $N = 2^m$ for integer m, and again consider Eq. (12). (If $N \neq 2^m$, one fix is to "pad" the data with enough zero values to satisfy that condition.) Since the total number of terms in the sum will be even, we can split them into two groups: those with n even and those with n odd. For the even terms, let $n = 2r$ with r an integer in the range $0, \ldots, \frac{1}{2}N - 1$. The sum of these even terms is

$$E_k = \sum_{r=0}^{\frac{1}{2}N-1} y_{2r} \exp\left[-i\frac{2\pi k(2r)}{N}\right] = \sum_{r=0}^{\frac{1}{2}N-1} y_{2r} \exp\left(-i\frac{2\pi kr}{\frac{1}{2}N}\right). \tag{21}$$

Notice that this is another Fourier transform just like Eq. (12), but with $\frac{1}{2}N$ rather than N samples. Similarly, the odd terms, with $n = 2r + 1$, lead to the sum

$$\begin{aligned}\sum_{r=0}^{\frac{1}{2}N-1} y_{2r+1} \exp\left[-i\frac{2\pi k(2r+1)}{N}\right] &= e^{-i2\pi k/N} \sum_{r=0}^{\frac{1}{2}N-1} y_{2r+1} \exp\left(-i\frac{2\pi kr}{\frac{1}{2}N}\right) \\ &\equiv e^{-i2\pi k/N} O_k.\end{aligned}$$

Again, these sums involve only half as many sample points as the original coefficients C_k. We see that each of the original C_k coefficients can be written as $C_k = E_k + e^{-i2\pi k/N} O_k$, where each of these terms requires half as many sample points spaced twice as far apart as in the original transform, and where the exponential factor accompanying O_k is known as the "twiddle factor" (really!)

Now here's the trick: each of these new terms can, in similar fashion, be expressed in terms of two distinct sums, of even- and odd-numbered terms, each involving half of the $\frac{1}{2}N$ sample points. Since $N = 2^m$, this divide-by-two process can continue all the way down until each transform is the transform of just one sample point. By Eq. (12), the Fourier transform of a single point has just one Fourier coefficient, C_0, whose value is

$$C_0 = \sum_{n=0}^{0} y_n \exp\left(-i\frac{2\pi kn}{N}\right) = y_0, \tag{22}$$

showing that the Fourier transform of one sample point is equal to the value of that point itself.

We now have all of the elements of the FFT that we need. In practice, the FFT actually is performed by following the reverse of the process described above: the transforms of N individual points, with one coefficient each, are combined in pairs; the resulting $\frac{1}{2}N$ sets of transforms, with two coefficients each, then are combined into $\frac{1}{4}N$ terms with four coefficients in each, etc. Note that at each step there are N coefficients to calculate, and the process requires a total of m steps, where we recall that there are $N = 2^m$ points in total. Thus, there are $m = \log_2 N$ steps, with N coefficients computed at each step, and so the overall calculation is of order $N \log_2 N$.

To convert a 4 MB MP3 file, which we'll assume contains 4×10^6 data values, the original DFT approach would require $N^2 \sim 10^{13}$ computations, while the new FFT method needs only $N \log_2 N \sim 10^8$: the FFT could be five orders of magnitude faster than the DFT here!

There are several versions of the FFT; perhaps the best known is the *Cooley-Tukey algorithm*, which plays some tricks with the data to achieve even greater efficiency. Python has its own built-in FFT functions based on the Cooley-Tukey algorithm; they are found in the `numpy.fft` package and are called `fft` and `rfft`. (Hence, they have to be imported using `from numpy.fft import fft`, for example). The former function can work with complex data, but the latter is for purely real data. That package also contains the inverse functions, named `ifft` and `irfft`, respectively.

There's a nice discussion of the Cooley-Tukey algorithm here:
http://nbviewer.ipython.org/url/jakevdp.github.io/downloads/notebooks/UnderstandingTheFFT.ipynb

It builds up to the algorithm through a series of increasingly more sophisticated approaches, starting with the simple matrix multiplication method noted earlier. The full Cooley-Tukey algorithm is not enlightening to see, so we will forgo presenting it.

To get a better sense of what the output of FFT routines (including the DFT), such as those in `numpy` often look like, consider the triangular wave pulse shown in Figure 6. The FFT of the triangle function looks as shown in Figure 7. Note that the FFT is symmetric, and that most of the values are very small: only those corresponding to small k, and their complex conjugate "copies" at large k, are significantly different from zero. To show more detail, Figure 8 zooms in on the small-k region. There, we see that the coefficients for smallest k are largest, and that they rapidly decrease in amplitude as k increases: the first five $|C_k|$ values dominate the others. We also can see a kind of wave pattern in the coefficients themselves, which is not atypical. (The data here was plotted as dots, but it could be plotted with vertical lines, as in Figure 3, using `matplotlib`'s `bar` function instead of the `plot` function.)

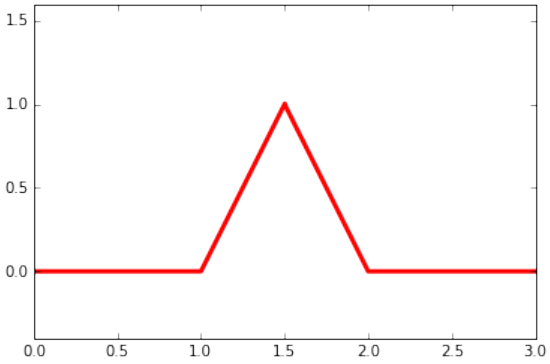


Figure 6: A triangular wave pulse

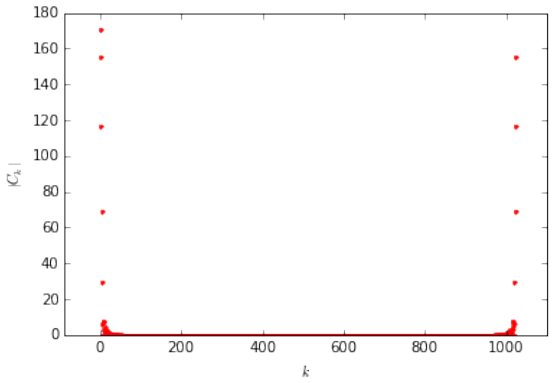


Figure 7: FFT of the triangular pulse

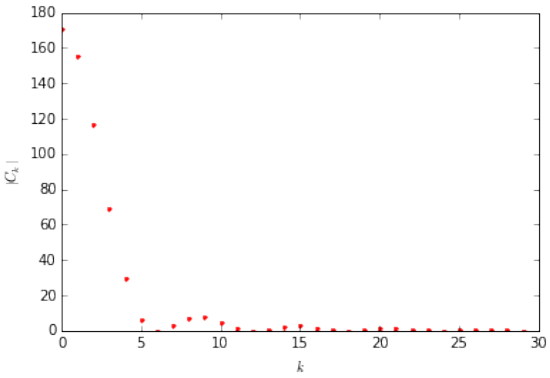


Figure 8: FFT of the pulse at small k

Recap

- A Fourier series can be used to represent a periodic function. The symmetry of the functions used in the series should match that of the function being approximated (if it has symmetry).
- The discrete Fourier transform provides a way to numerically compute the coefficients of a finite series representation of a function, and can be used to model discretely sampled data.
- The frequency at which data samples are acquired determines how accurately the underlying function can be modeled, as explained by the sampling theorem and as seen in the phenomenon of aliasing.
- The Fast Fourier Transform is an elegant and highly efficient algorithm for finding the transform of discrete data.

Exercises

Exercise #1

In order to have a simple function on which to test Fourier transform routines, write a program to return data representing a square wave having its left edge at the origin of the x-axis. Let the amplitude of the square wave be h, and its width be w, and let it be flanked on each side by a region of width w and zero amplitude. Thus, the entire function should extend from $x = -w$ to $x = +2w$, but it should be nonzero only in the range from 0 to w. Use 1024 $(= 2^{10})$ points in your construction of the square wave (including the zero regions). Your program should plot the square wave, to prove that your code works as it should. Test it by creating and plotting a square wave with height 1.0 and width 0.5.

Exercise #2

(a) Apply the first (non-vectorized) DFT function presented in the module to your square wave function using the height and width of Exercise #1, and make two plots of the output: one showing all of the data and another showing just the first 20 values. (It's useful to plot the data as large dots, using the 'o' argument to the `plot` function.) Briefly describe and interpret the results.

(b) Modify the first DFT function so that it computes *all* of the `c[k]` coefficients, including those with $k > \frac{1}{2}N$. (The `numpy` function to compute a complex conjugate comes from the `numpy.ma` library and is called `conjugate`.) Run it and the vectorized version on the square wave data, and plot the two results (on separate graphs; as small dots, using '.'). To determine whether or not they're the same, plot the difference between them (on a third graph), and also use `numpy`'s `allclose` function, which returns `True` if corresponding elements of two arrays are the same to within some small tolerance. (See the documentation for `numpy.allclose` for information on how to use it and on the default tolerances; other values may be provided by the user as arguments passed to the function). Do the results agree with each other?

(c) Using `numpy`'s `linspace` function, create sine wave data in the range (0, 2π) using 1000 data points, run your "full" or vectorized DFT function on it, and plot the first 10 values. Do the same for sine wave data in the range (0, 4π), again with 1000 data points. Explain the two graphs.

(d) Finally, run `numpy`'s `fft` function (from the `numpy.fft` library) on your square wave and on the first sine wave of (c) to check whether its outputs are the same as those from our DFT functions.

Exercise #3

To show the superior speed of `numpy`'s `fft` routine, apply it, the nonvectorized DFT routine you used in Exercise #2, and the vectorized DFT routine, to your square wave. Time how long it takes each routine to perform the Fourier transform by using the `timeit` magic [e.g., `%timeit fft(square)`]. How do the times compare (pay attention to the units)?

Exercise #4

Import the sound data in the file `piano.txt`, which you should find where the modules are posted. You will need to upload it to the directory in which you are running your notebook, and then read it into a variable using the syntax `sound = numpy.loadtxt('piano.txt')`, where "`sound`" is the variable name. (a) Plot the data in this file. (b) Now run `numpy`'s `fft` routine on the data, storing the Fourier transform output and plotting it as well. You should see two large spikes, as well as several small ones (all in a symmetric pattern). (c) Copy the data from the Fourier transform output to a new array, and change that new array so that the data points in the range 2001-98000 have the value 0: this will eliminate most of the smaller spikes in the Fourier transform. Plot the new array to confirm this. (d) Apply the reverse transform `ifft` to this new array, and plot the result. How does it compare to the original `piano` plot? Explain any differences you observe. (e) Run the notebook `Soundfile_Player.ipynb` from the directory where the modules are posted. It will produce audio players that you can run to play the original sound file and the one with the small spikes eliminated. How do they compare? Explain any differences you hear. How does this mesh with what you observed in the plots of those files?

Breakpoint Answers

Breakpoint 1

The cosine sum starts with $k = 0$ because $\cos(0)$ is not zero. However, $\sin(0) = 0$, so there is no point in including the $k = 0$ term in that sum.

Breakpoint 2

From Eq. (7), we're interested in the integral $\int_0^L \exp\left[i2\pi(k' - k)x/L\right] \mathrm{d}x$. If $k' = k$, the integrand becomes $e^0 = 1$, and so the integral gives L immediately. If $k' \neq k$, the integral can be computed as follows:

$$\int_0^L \exp\left[i\frac{2\pi(k' - k)x}{L}\right] \mathrm{d}x = \frac{L}{2i\pi(k' - k)} \exp\left[i\frac{2\pi(k' - k)x}{L}\right]_0^L = \frac{L}{2i\pi(k' - k)} \left[e^{2i\pi(k' - k)} - 1\right].$$

Since $k' \neq k$, then the first term in square brackets is $\exp(2i\pi n)$, where n is an integer, and this just equals one (it corresponds to being at angle 0 on the unit circle in the complex plane). Thus, the combination of terms in square brackets vanishes.

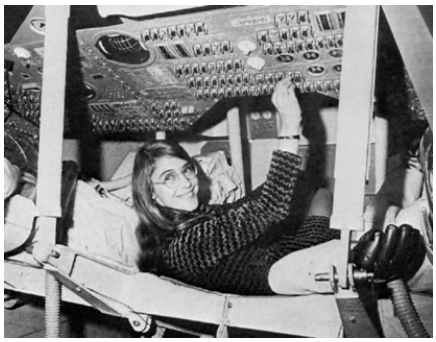
Breakpoint 3

(a) For $N = 7$, $\frac{1}{2}(N + 1) = 4$, while `N//2 + 1 = 7//2 + 1 = 3 + 1 = 4` after rounding down the fraction. (b) For $N = 8$, $\frac{1}{2}N + 1 = 5$, and `N//2 + 1 = 8//2 + 1 = 4 + 1 = 5`. In both cases, the results agree.

Breakpoint 4

A wave with period T has a frequency $f = 1/T$. For the Nyquist frequency to match that frequency, we require $1/T = 1/2\Delta$, or $\Delta = T/2$. Thus, the wave must be sampled at half the period, or twice per cycle — if it's sampled at one peak, it will be sampled at all of the other peaks and troughs. By the way, this is why digital audio usually is sampled at around 44 Mhz or higher: this is a high enough sample rate to accurately detect sound frequencies of 22 Hz, which is at the upper limit of normal human hearing.

Scientist Profile



Margaret Hamilton is one of pioneers in the field of computer programming. She was born in 1936 in Indiana, where she received a B.A. in mathematics from Earlham College, a Quaker college. She subsequently moved to Boston where she intended to pursue a graduate degree in math at Brandeis University. However, in 1960 she took an interim position at MIT developing weather prediction software for Professor Edward Lorenz, the modern discoverer of chaos theory. She later took a job at the Draper Laboratory of MIT, where she eventually became the director of software programming for the Apollo and Skylab programs. Hamilton was especially instrumental in developing technology for landing spacecraft on the moon in the 1960s. Hamilton essentially designed the first-ever software that landed people on the moon, for the Apollo 11 mission. The software prevented disaster on the trip. Three minutes before landing, the spacecrafts computer was overloaded because it was trying to do too many things. But because of the program that Hamilton had written, it was able to prioritize its actions and ignore the ones not associated with the landing. In creating the Apollo space mission software, Hamilton contributed to many subfields of computing. She is credited with coining the term "software engineering," and she helped to develop the techniques of asynchronous software, priority scheduling, and end-to-end testing, to name a few. Hamiltons work paved the way for the growth and development of new ideas and new fields in computational science. And in the process, she took people to the moon! (Some information taken from the Wikipedia page on Margaret Hamilton.)