SEE
https://en.wikipedia.org/wiki/Fourier_series

3.5 Lecture 15: Fourier series and transforms

Fourier transforms are useful for signal analysis, and are also an important tool for solving differential equations. First let's recall what Fourier series can do: any periodic function $f(x)$ defined on a finite interval $0 \leq x \leq L$ can be written as a Fourier series.

If $f(x)$ is symmetric about the midpoint at $L/2$, then we can write

$$f(x) = \sum_{k=0}^{\infty} \alpha_k \cos\left(\frac{2\pi k x}{L}\right),$$

where $\{\alpha_k\}$ is the set of coefficients. If $f(x)$ is antisymmetric about the midpoint, then we have

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

So we can write a function with no symmetry as

$$f(x) = \sum_{k=0}^{\infty} \alpha_k \cos\left(\frac{2\pi k x}{L}\right) + \sum_{k=1}^{\infty} \beta_k \sin\left(\frac{2\pi k x}{L}\right)$$

and then making using of $\cos\theta = (e^{i\theta} + e^{-i\theta})/2$ and $\sin\theta = (e^{i\theta} - e^{i\theta})/2i$ to write

$$\begin{aligned} f(x) = \sum_{k=0}^{\infty} \alpha_k \left[\exp\left(i\frac{2\pi k x}{L}\right) + \exp\left(-i\frac{2\pi k x}{L}\right)\right] \\ + \frac{i}{2}\sum_{k=1}^{\infty} \beta_k \left[\exp\left(-i\frac{2\pi k x}{L}\right) - \exp\left(i\frac{2\pi k x}{L}\right)\right] \end{aligned}$$

From which point we can collect terms and write it as

$$f(x) = \sum_{k=-\infty}^{\infty} \gamma_k \exp\left(i\frac{2\pi k x}{L}\right)$$

where

$$\gamma_k = \begin{cases} \frac{1}{2}(\alpha_{-k} + i\beta_{-k}), & k < 0 \\ \alpha_0, & k = 0 \\ \frac{1}{2}(\alpha_{-k} - i\beta_{-k}), & k > 0. \end{cases}$$

see Fig.7.1 from book

Fourier series can only be used for *periodic* functions! To extend to non-periodic ones, just pick out an interval of a function and repeat it infinitely so that it becomes periodic.

How do we calculate the coefficients, γ_k? Just use the fact that $\{e^{-i\frac{2\pi kx}{L}}\}_k$ constitutes an orthonormal basis for the space $[0, L]$. That is, consider that

$$\int_0^L \mathrm{d}x f(x) \exp\left(-i\frac{2\pi kx}{L}\right) = \sum_{k'=-\infty}^{\infty} \gamma_{k'} \int_0^L \exp\left(i\frac{2\pi(k'-k)x}{L}\right) dx.$$

If $k' \neq k$, then

$$\begin{aligned}\int_0^L \exp\left(i\frac{2\pi(k'-k)x}{L}\right) dx &= \frac{L}{i2\pi(k'-k)}\left[\exp\left(i\frac{2\pi(k'-k)x}{L}\right)\right]_0^L \\ &= \frac{L}{i2\pi(k'-k)}\left[\exp(i2\pi(k'-k)) - 1\right] \\ &= 0 \text{ since } e^{2\pi in} = 1 \ \forall n \in \mathbb{Z}\end{aligned}$$

However, if $k' = k$, then the integral is equal to L. In this case,

$$\int_0^L f(x) \exp\left(-i\frac{2\pi kx}{L}\right) dx = L\gamma_k$$

or

$$\gamma_k = \frac{1}{L}\int_0^L f(x) \exp\left(-i\frac{2\pi kx}{L}\right) dx.$$

3.5.1 Discrete Fourier transforms

There are many cases in which it isn't possible to calculate the coefficients γ_k analytically. So we can use numerical methods. It turns out that approximations with the trapezoidal rule is equivalent to the discrete Fourier transform.

Consider N slices of width $h = L/N$. Applying the trapezoidal rule gives

Trapezoidal Eq.(5.3) from book

$$\gamma_k = \frac{1}{L}\left(\frac{L}{N}\right)\left[\frac{1}{2}f(0) + \frac{1}{2}f(L) + \sum_{n=1}^{N-1} f(x_n)\exp\left(-i\frac{2\pi kx_n}{L}\right)\right]$$

when the sample point positions are $x_n = nL/N$. Since $f(x)$ is periodic, we have $f(0) = f(L)$, so then above simplifies to

$$\gamma_k = \frac{1}{N}\sum_{n=0}^{N-1} f(x_n)\exp\left(-i\frac{2\pi kx_n}{L}\right)$$

This is the discrete Fourier transform (DFT)

Note that 1/N was removed, this is the convention

c_k = sum_{n=0}^{N-1} y_n exp(-i 2pi k n/N)

We can use this to evaluate coefficients, at least in cases with evenly sampled data (pretty frequent). It's also worth noting that while these results were derived using the trapezoidal rule, there is a sense in which that are exact. Recall that

$$\sum_{k=0}^{N-1} a^k = \frac{1-a^N}{1-a}, \quad a \neq 1,$$

then

$$\sum_{k=0}^{N-1} \left(e^{i\frac{2\pi m}{N}}\right)^k = \frac{1-e^{i2\pi m}}{1-e^{i2\pi m/N}} = 0,$$

since m is an integer, making the numerator zero. In the case that $m = 0$, or is a multiple of N, then the sum is N. So

$$\sum_{k=0}^{N-1} \exp\left(i\frac{2\pi km}{N}\right) = \begin{cases} N & \text{if } m = 0, N, 2N, \ldots \\ 0 & \text{else} \end{cases}$$

Then consider the sum

$$\begin{aligned}
\sum_{k=0}^{N-1} c_k \exp\left(i\frac{2\pi kn}{N}\right) &= \sum_{k=0}^{N-1}\left[\sum_{n'=0}^{N-1} y_{n'} \exp\left(-i\frac{2\pi kn'}{L}\right)\right] \exp\left(i\frac{2\pi kn}{N}\right) \\
&= \sum_{n'=0}^{N-1} y_{n'} \sum_{k=0}^{N-1} \exp\left(i2\pi k(\frac{n-n'}{N})\right) \\
&= \sum_{n'=0}^{N-1} y_{n'}\delta_{n,n'}N \\
&= Ny_n \\
\implies \boxed{y_n = \frac{1}{N}\sum_{k=0}^{N-1} c_k \exp\left(i\frac{2\pi kn}{N}\right).}
\end{aligned}$$

This is the INVERSE discrete Fourier transform (inverse DFT)

This is the inverse discrete Fourier transform (inv. DFT).

This proves that the matrix with entries

$$U_{kn} = \frac{1}{\sqrt{N}} \exp\left(-i\frac{2\pi kn}{N}\right)$$

is a unitary matrix. So we can recover the original values exactly by performing the inverse DFT. So you can move freely back and force between the original values and the Fourier coefficients.

- We can compute this on a computer because the sum is finite
- This discrete formula only gives sample values $y_n = f(x_n)$. So if the function is oscillating rapidly between samples, the DFT won't capture this, so DFT just gives some idea of the function.

If the function is real, then can use this symmetry to simplify further. Suppose all y_n are real and consider c_k for $N/2 < k \leq N-1$, so $k = N - r$ for $1 \leq r < N/2$. Then

$$\begin{aligned} c_{N-r} &= \sum_{n=0}^{N-1} y_n \exp\left(-i\frac{2\pi(N-r)n}{N}\right) \\ &= \sum_{n=0}^{N-1} y_n \exp\left(-i2\pi n\right) \exp\left(i\frac{2\pi rn}{N}\right) \\ &= \sum_{n=0}^{N-1} y_n \exp\left(i\frac{2\pi rn}{N}\right) = c_r^* \end{aligned}$$

so then $c_{N-1} = c_1^*$, $c_{N-2} = c_2^*$, etc. So when calculating the DFT of a real function, we only have to calculate c_k for $0 \leq k \leq N/2$. However, if the y_n are complex, then we need to calculate all N Fourier coefficients.

See Python lecture 14

Solve an example in class.

We plot ABS[c]!!!

Discuss units.

MAIN frequency + harmonics

SIGNAL ANALYZERS

Bring up `dft.py`. This program uses *exp* from the cmath package, which isn't the quickest way to calculate the DFT. We can instead do FFT. If we shift the positions of the sample points. then not much changes. Suppose that instead of taking samples at $x_n = nL/N$, we take them at $x'_n = x_n + \Delta$. Then

$$\begin{aligned} c_k &= \sum_{n=0}^{N-1} f(x_n + \Delta) \exp\left(-i\frac{2\pi k(x_n+\Delta)}{L}\right) \\ &= \exp\left(-i\frac{2\pi k\Delta}{L}\right) \sum_{n=0}^{N-1} f(x'_n) \exp\left(-i\frac{2\pi k x_n}{L}\right) \\ &= \exp\left(-i\frac{2\pi k\Delta}{L}\right) \sum_{n=0}^{N-1} y'_n \exp\left(-i\frac{2\pi k x_n}{L}\right), \end{aligned}$$

where $y'_n = f(x'_n)$ are the new samples. We can absorb the phase factors into the coefficients as $c'_k = \exp\left(i\frac{2\pi k\Delta}{L}\right) c_k$ so that $c'_k = \sum_{n=0}^{N-1} y'_n \exp\left(-i\frac{2\pi kn}{L}\right)$ so that DFT is independent of where the samples are taken.

We can distinguish between Type-I DFT where we divide interval $[0, L]$ into N slices and take samples at endpoints, and a Type-II where we take samples at the midpoints of slices.

NOTE: from numpy.fft, rfft computes only N/2+1 c_k's.

There is also the invere irfft

3.5.2 2D Fourier transform

It's useful for image processing, for instance in astronomy (classic case: Hubble image correction). Suppose we have $M \times N$ grid of samples y_{mn}. First do a FT on the rows:

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

From numpy.fft, rfft2 computes (N/2+1)*N coefficients.

and then FT the m variable:

$$c_{kl} = \sum_{n=0}^{N-1} c'_{ml} \exp\left(-i\frac{2\pi km}{M}\right).$$

There is also the inverse irfft2

Combined, these read

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

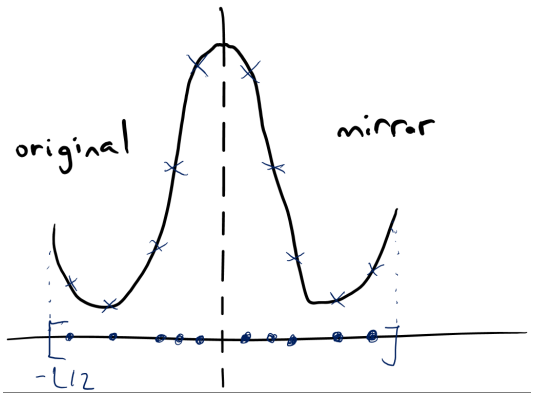
What is the FT doing? Breaking down a signal into its frequency components, like a signal analyzer. Bring up `dft.py`. The first spike is the frequency of the main wave, and the others are harmonics.

Discrete cosine transform

Recall that if a function is symmetric about $x = L/2$ (the midpoint) then we can write

$$f(x) = \sum_{k=0}^{\infty} \alpha_k \cos\left(\frac{2\pi kx}{L}\right)$$

We cannot do this for all functions. However, if we'd like to do so, we can by just sample a function over an interval, and then adding it to its mirror image, i.e.,



So we make the function symmetric, and when the samples are, we have $y_0 = y_N$, $y_1 = y_{N-1}$, $y_2 = y_{N-2}$, etc. We then get for the DFT:

$$\begin{aligned} c_k &= \sum_{n=0}^{N-1} y_n \exp\left(-i\frac{2\pi kn}{N}\right) \\ &= \sum_{n=0}^{N/2} y_n \exp\left(-i\frac{2\pi kn}{N}\right) + \sum_{n=N/2+1}^{N-1} y_n \exp\left(-i\frac{2\pi kn}{N}\right) \\ &= \sum_{n=0}^{N/2} y_n \exp\left(-i\frac{2\pi kn}{N}\right) + \sum_{n=N/2+1}^{N} y_{N-n} \exp\left(i\frac{2\pi k(N-n)}{N}\right) \end{aligned}$$

where in the final line we used $\exp(i2\pi k) = 1$. Make a change of variables $N - n \to n$ to get

$$\begin{aligned} c_k &= \sum_{n=0}^{N/2} y_n \exp\left(-i\frac{2\pi kn}{N}\right) + \sum_{n=1}^{N/2-1} y_n \exp\left(i\frac{2\pi kn}{N}\right) \\ &= y_0 + y_{N/2}\cos\left(\frac{2\pi k(N/2)}{N}\right) + 2\sum_{n=1}^{N/2-1} y_n \cos\left(\frac{2\pi kn}{N}\right). \end{aligned}$$

DCT

Usually though, the discrete cosine transform is applied to real values, which means that the c_k coefficients are real. In this case, we have the $c_{N-r} = c_r' = c_r$, and the inverse transform is

$$\begin{aligned} y_n &= \frac{1}{N}\sum_{k=0}^{N-1} c_k \exp\left(i\frac{2\pi kn}{N}\right) \\ &= \frac{1}{N}\left[\sum_{k=0}^{N/2} c_k \exp\left(i\frac{2\pi kn}{N}\right) + \sum_{k=N/2+1}^{N-1} c_k \exp\left(i\frac{2\pi kn}{N}\right)\right] \\ &= \frac{1}{N}\left[\sum_{k=0}^{N/2} c_k \exp\left(i\frac{2\pi kn}{N}\right) + \sum_{k=N/2+1}^{N-1} c_{N-k} \exp\left(-i\frac{2\pi (N-k)n}{N}\right)\right] \\ &= \frac{1}{N}\left[\sum_{k=0}^{N/2} c_k \exp\left(i\frac{2\pi kn}{N}\right) + \sum_{k=1}^{N/2-1} c_k \exp\left(-i\frac{2\pi kn}{N}\right)\right] \\ &= \frac{1}{N}\left[c_0 + c_{N/2}\cos\left(\frac{2\pi n(N/2)}{N}\right) + 2\sum_{k=1}^{N/2-1} c_k \cos\left(\frac{2\pi kn}{N}\right)\right] \end{aligned}$$

inverse DCT

which is the inverse discrete cosine transform. It has so much symmetry that the DCT is the same as its inverse! If we take the samples at midpoints, then we can show that the coefficients are

$$a_k = 2 \sum_{n=0}^{N/2-1} y_n \cos\left(\frac{2\pi k(n+1/2)}{N}\right),$$

and the inverse transform is

$$y_n = \frac{1}{N}\left[a_0 + 2\sum_{k=1}^{N/2-1} a_k \cos\left(\frac{2\pi k(n+1/2)}{N}\right)\right].$$

A nice feature of the DCT is that it does not assume that the function is periodic. Neither does the DFT, but it does force the first and last values to be the same, which can create a large discontinuity. The DCT does not do this. N.b. we can also calculate the discrete sine transform, but this is rarely used because it forces the endpoints to zero.

The assignment does not have a problem on DCT, but the book has Ex.7.6

DCT have important technological technological uses. They form the mathematical basis for the computer image file called JPEG (Joint Photographic Experts Group), used to store images in the www.

Digital images are represented as regular grids of dots (pixels) of different shade, and the shades are stored on the computer as ordinary numbers.

Because there are too many numbers to store. We can instead store using the DCT.

The JPEG format works by dividing the pixels in an image into blocks and performing DCTs on the blocks, then looking for coefficients a_k that are small and can be discarded.
The remaining coefficients are stored in a file. Since many a_k's are small, the file is smaller.

When you view a picture, your computer reconstitutes the picture using the INVERSE transform of a_k's. The image is not exactly the original, but usually your eyes cannot tell. Sometimes, you can detect small problems in the image, called "compression artifacts", arising from the missing data.

A variant of the same technique is used to compress moving pictures (films, videos) using MPEG.

A similar scheme is used for music, in the file MP3. In this case, the components that are discarded are chosen not only on the grounds of the smallest values, but also with a knowledge of what humans can hear.

Chapter 4

Lectures 16-20

We've gotten pretty far in the lecture notes (3/4 complete), and a little over halfway in the homeworks. In all honestly, I'm feeling a bit burned out on the homeworks; likely because of both having written crap code on the first half of HW3 without internet access, and jamming through the lectures in cars and planes without taking time to focus on the content. But this means a couple things: (1) I'm comfortable enough writing code in Python to do non-trivial things without constantly checking StackExchange, and (2) I've gotten pretty damn far. We're going to keep pushing, with more of a concept focus. I'm still committing to doing the good problems from homeworks 3 and 4, and most of the remaining lectures look pretty interesting. I can put in 3hours/day on this to wrap it up before my semester starts.

I'm at a reasonable level with BaKoMa TEX too. It's obvious that some functionalities – especially matrix operations, or even long lines of math derivations, will be *much* easier handwritten. This is fine, with figure input.

4.1 Lecture 16: Fast Fourier transform, ordinary diffeqs, Euler method, Runge-Kutta

Recall that the discrete Fourier transform is

$$c_k = \sum_{n=0}^{N-1} \gamma_n \exp\left(-i\frac{2\pi kn}{N}\right).$$

Fourier analysis just refers to analyzing the distinct components that contribute to a periodic phenomena. In other words, it's about expression a function as a sum of periodic components, and then recovering those components. A simple example would be to consider a single-frequency whistle.

An audio detector that senses compressions and rarefractions of air would produce a sinusoidal voltage when this whistle is blown. Taking the DFT of that signal would yield a single-frequency peak in the whistle's frequency spectrum. This gets useful when combining many periodic signals together.

The Python program (available to the people taking the class) had a for-loop for each coefficient, and N terms in the sum, implying N^2 operations would be required to get all the coefficients. If we're not willing to wait for more than 1 billion operations, then we can do a DFT for $N^2 = 10^9 \implies N \approx 32000$ samples. This isn't too much - about one second of audio.

The fast Fourier transform (FFT) is a DFT-solving algorithm for cutting the number of computations needed for N points from $2N^2$ to $2N \ln n$. It was discovered by Cooley and Tukey in 1965, although Gauss had pointed at the key step in the algorithm in 1805.

The algorithm is easiest to describe when the number of samples is a power of two. Let $N = 2^m$, for m an integer. Consider the sum in the DFT equation, and divide it into 2 groups of even and odd terms. Consider the even terms first, where $n = 2r$, for $r \in \{0, ..., N/2 - 1\}$. Then

$$\begin{aligned} E_k &= \sum_{r=0}^{N/2-1} y_{2r} \exp\left(-i\frac{2\pi k(2r)}{N}\right) \\ &= \sum_{r=0}^{N/2-1} y_{2r} \exp\left(-i\frac{2\pi kr}{N/2}\right) \equiv E_{k,\mathrm{mod}N/2}. \end{aligned}$$

Note that this is just another DFT, with $N/2$ samples instead of N. Then look at the odd terms:

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

where O_k is another DFT with $N/2$ sample points. Call this $O_{k,\mathrm{mod}N/2}$. So then

$$\begin{aligned} c_k &= E_{k,\mathrm{mod}N/2} + e^{-i2\pi k/N} O_{k,\mathrm{mod}N/2} \\ &= E_{k,\mathrm{mod}N/2} + W_n^k O_{k,\mathrm{mod}N/2}. \end{aligned}$$

We can summarize what we have in terms of a diagram. Let

denote a N-point DFT. The output are the DFT of the input. Then to compute c_0, we do

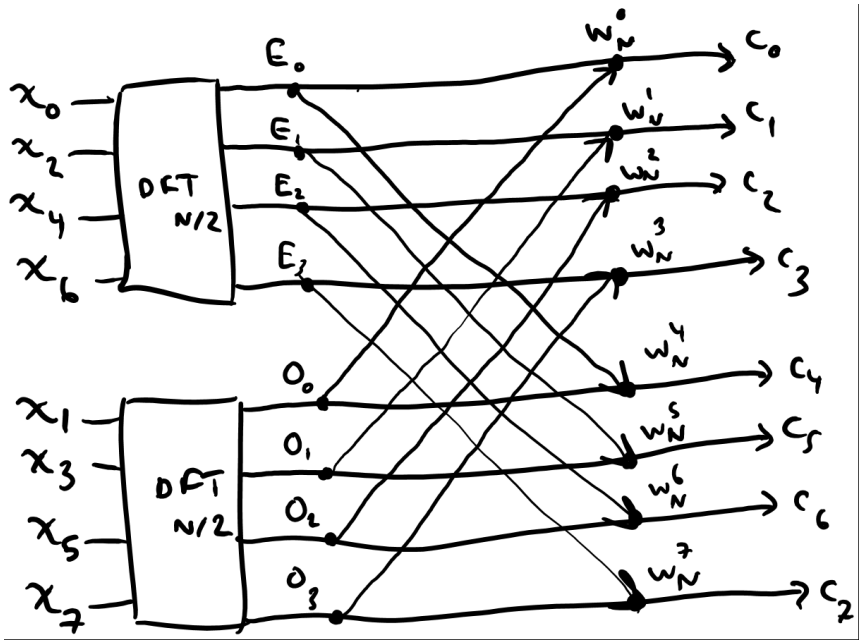


Figure 4.1: "Butterfly" diagram for FFT.

This diagram specifies that recursion. The starting point is DFT_1 which is just the identity transformation. Then at the mth stage that are $\mathcal{O}(N)$ calculations. There are $?\ln N$ stages, meaning that we have $\mathcal{O}(N \ln N)$ operations. In other words, it's nearly linear!

For example, if we have $N = 10^6$ samples to process, then the naive way would require 10^{12} operations. This is highly nontrivial for a typical computer, and not practical. But $N \ln N = 10^7$, since the natural logarithm of anything big is like, 10. This can then be done in under a second! The inverse DFT can be done in the same way.

Numpy provides `numpy.fft` as a FFT package. Skimming the documentation, you'll find there are different types of FFTs within it. *Rfft* computes

the FFT for a set of real numbers, and returns the first half (since the other half are complex conjugates). You can also use *fft* and *ifft* to calculate Foruier transforms of complex data. There are also functions for two dimensional transforms, and well as functions for higher dimensions.

4.1.1 Ch. 8: Solving Ordinary Differential Equations

Consider the first-order equation

$$\frac{dx}{dt} = \frac{2x}{t} + \frac{3x^2}{t^3}.$$

It is not separable, and it's also nonlinear, so we need to use computational methods to solve it.

The general form for a first-order differential equation is

$$\frac{dx}{dt} = f(x,t).$$

To calculate a full solution, we need a boundary condition, e.g., the value of x at one particular value of t (usually $t = 0$).

Euler's method

Suppose we have to solve $\frac{dx}{dt} = f(x,t)$, and we're given an initial condition. Then Taylor expand $x(t+h)$ about t, to get

$$\begin{aligned} x(t+h) &= x(t) + hx'(t) + \frac{h^2}{2}x''(t) + ... \\ &= x(t) + hf(x,t) + \mathcal{O}(h^2), \end{aligned}$$

so if we neglect $\mathcal{O}(h^2)$ terms, we get

$$x(t+h) = x(t) + hf(x,t).$$

So if we know x at time t, we can just use this equation to iterate. If h is small enough, this does pretty well. It's called Euler's method. There's an example in `2-euler.py`, for $dx/dt = -x^3 + \sin t$, but it's not online ☹. So we write our own:

```
#Numerical analysis of dx/dt = -x**3 + sin(t)
import numpy as np
import matplotlib.pyplot as plt

def f(x,t):
```