Fourier Transforms

Ramses van Zon

PHY1610, Winter 2023

(Discrete) Fourier Transform

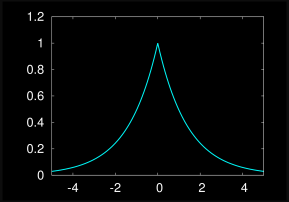
Fourier Transform

In this part of the lecture, we will discuss:

- The Fourier transform,
- The discrete Fourier transform
- The fast Fourier transform
- Examples using the FFTW library

Fourier Transform recap

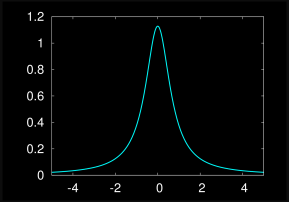
- Let f be a function of some spatial variable x.



$f(x) = e^{-|x|}$

- Transform to a function $\hat{f}$ of the angular wavenumber k:

$$\hat{f}(k) \propto \int f(x)\, e^{\pm i\, k \cdot x}\, dx$$



$f(x) = (1 + k^2)^{-1}$

- Inverse transformation:

$$f(x) \propto \int \hat{f}(k)\, e^{\mp i\, k \cdot x}\, dk$$

Fourier Transform

- Fourier made the claim that any function can be expressed as a harmonic series.
- The FT is a mathematical expression of that.
- Constitutes a linear (basis) transformation in function space.
- Transforms from spatial to wavenumber, or time to frequency, etc.
- Constants and signs are just convention.*

* some restritions apply.

Discrete Fourier Transform

C. F. Gauss

- Given a set of n function values on a regular grid:

$$x_j = j\Delta x; \quad f_j = f(j\Delta x)$$

- Transform to n other values

$$\hat{f}_q = \sum_{j=0}^{n-1} f_j\, e^{\pm 2\pi i\, j\, q/n}$$

- Easily back-transformed:

$$f_j = \frac{1}{n}\sum_{q=0}^{n-1} \hat{f}_q\, e^{\mp 2\pi i\, j\, q/n}$$

- Solution is periodic: $\hat{f}_{-q} = \hat{f}_{n-q}$. You run the risk of aliasing, as q is equivalent to $q + \ell n$. Cannot resolve frequencies higher than $q = n/2$ (Nyquist).

Slow Fourier Transform

$$\hat{f}_q = \sum_{j=0}^{n-1} f_j \, e^{\pm\, 2\pi i\, j\, q/n}$$

- Discrete fourier transform is a linear transformation.
- In particular, it's a matrix-vector multiplication.
- Naively, costs $\mathcal{O}(n^2)$. Slow!

Slow DFT

```
#include <complex>
#include <rarray>
#include <cmath>

typedef std::complex<double> complex;

void fft_slow(const rvector<complex>& f, rvector<complex>& fhat, bool inverse)
{
  int n = fhat.size();
  double v = (inverse?-1:1)*2*M_PI/n;
  for (int q=0; q<n; q++)
  {
    fhat[q] = 0.0;
    for (int m=0; m<n; m++) {
      fhat[q] += complex(cos(v*q*m), sin(v*q*m)) * f[m];
    }
  }
}
```

Even Gauss realized $\mathcal{O}(n^2)$ was too slow and came up with ...

Fast Fourier Transform

- Derived in partial form several times before and even after Gauss, because he'd just written it in his diary in 1805 (published later).
- Rediscovered (in general form) by Cooley and Tukey in 1965.

Basic idea

- Write each n-point FT as a sum of two $\frac{n}{2}$ point FTs.
- Do this recursively ${}^2\log n$ times.
- Each level requires $\sim n$ computations: $\mathcal{O}(n \log n)$ instead of $\mathcal{O}(n^2)$.
- Could as easily divide into 3, 5, 7, ... parts.

Fast Fourier Transform: How is it done?

- Define $\omega_n = e^{2\pi i/n}$.
- Note that $\omega_n^2 = \omega_{n/2}$.
- DFT takes form of matrix-vector multiplication:

$$\hat{f}_q = \sum_{j=0}^{n-1} \omega_n^{qj}\, f_j$$

- With a bit of rewriting (assuming n is even):

$$\hat{f}_q = \underbrace{\sum_{j=0}^{n/2-1} \omega_{n/2}^{qj}\, f_{2j}}_{\text{FT of even samples}} + \omega_n^q \underbrace{\sum_{j=0}^{n/2-1} \omega_{n/2}^{qj}\, f_{2j+1}}_{\text{FT of odd samples}}$$

- Repeat, until the lowest level (for $n = 1$, $\hat{f} = f$).
- Note that a fair amount of shuffling is involved.

Fast Fourier Transform: Already done!

We've said it before and we'll say it again: Do not write your own: use existing libraries!

Why?

- Because getting all the pieces right is tricky;
- Getting it to compute fast requires intimate knowledge of how processors work and access memory;
- Because there are libraries available.

  Examples:

  - FFTW3 (Faster Fourier Transform in the West, version 3)
  - Intel MKL
  - IBM ESSL
- Because you have better things to do.

Example of using a library: FFTW

Rewrite of previous (slow) ft to a fast one using fftw

```
#include <complex>
#include <rarray>
#include <fftw3.h>

typedef std::complex<double> complex;

void fft_fast(const rvector<complex>& f, rvector<complex>& fhat, bool inverse)
{
  int n = f.size();
  fftw_plan p = fftw_plan_dft_1d(n,
                  (fftw_complex*)f.data(), (fftw_complex*)fhat.data(),
                  inverse?FFTW_BACKWARD:FFTW_FORWARD,
                  FFTW_ESTIMATE);
  fftw_execute(p);
  fftw_destroy_plan(p);
}
```

Notes

- Always create a plan first.
- An fftw_plan contains all information necessary to compute the transform, including the pointers to the input and output arrays.
- Plans can be reused in the program, and even saved on disk!
- When creating a plan, you can have FFTW measure the fastest way of computing dft's of that size (FFTW_MEASURE), instead of guessing (FFTW_ESTIMATE).
- FFTW works with doubles by default, but you can install single precision too.

Inverse DFT

- Inverse DFT is similar to forward DFT, up to a normalization: almost just as fast.

$$f_j = \frac{1}{n} \sum_{q=0}^{n-1} \hat{f}_q \, e^{\mp 2\pi i \, j \, q/n}$$

  Many implementations (almost all in fact) leave out the $1/n$ normalization.

- FFT allows quick back-and-forth between space and wavenumber domain, or time and frequency domain.
- Allows parts of the computation and/or analysis to be done in the most convenient or efficient domain.

Consider an example

- Create a 1d input signal: a discretized $sinc(x) = \sin(x)/x$ with 16384 points on the interval [-30:30].
- Perform forward transform
- Write to standard out
- Compile, and linking to fftw3 library.
- Continous FT of $\mathrm{sinc}(x)$ is the rectangle function:

$$\mathrm{rect}(f) = \begin{cases} 0.5 & \text{if } ||k|| \leq 1 \\ 0 & \text{if } ||k|| > 1 \end{cases}$$

  up to a normalization.
- Does it match?

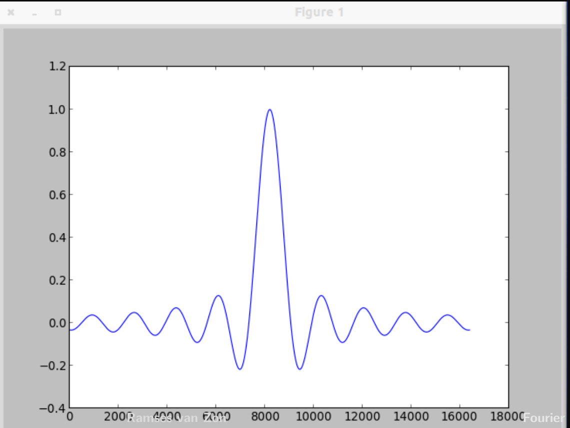
Code for the working example

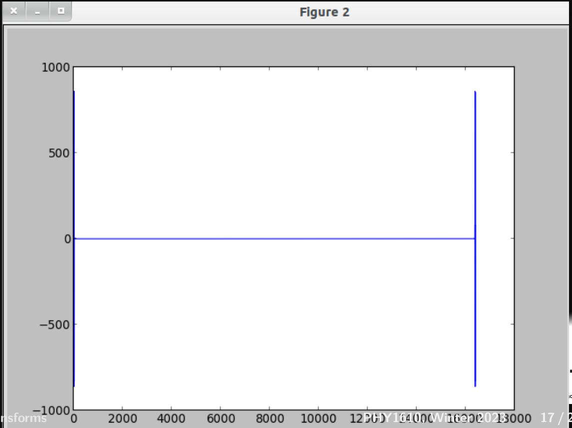
```
//sincfftw.cpp
#include <iostream>
#include <complex>
#include <rarray>
#include <fftw3.h>
typedef std::complex<double> complex;
int main() {
  const int n = 16384;
  rvector<complex> f(n), fhat(n);
  for (int i=0; i<n; i++) {
    double x = 60*(i/double(n)-0.5); // x-range from -30 to 30
    if (x!=0.0) f[i] = sin(x)/x; else f[i] = 1.0;
  }
  fftw_plan p = fftw_plan_dft_1d(n,
                    (fftw_complex*)f.data(), (fftw_complex*)fhat.data(),
                    FFTW_FORWARD, FFTW_ESTIMATE);
  fftw_execute(p);
  fftw_destroy_plan(p);
  for (int i=0; i<n; i++)
    std::cout << f[i] << "," << fhat[i] << std::endl;
  return 0;
}
```

Compile, link, run, plot

```
$ module load gcc/12 fftw/3 python/3
$ g++ -std=c++17 -c -O2 sincfftw.cpp -o sincfftw.o
$ g++ sincfftw.o -o sincfftw -lfftw3
$ ./sincfftw > output.dat
$ ipython --pylab
```

```
>>> data = genfromtxt('output.dat')
>>> plot(data[:,0])
>>> figure()
>>> plot(data[:,2])
```

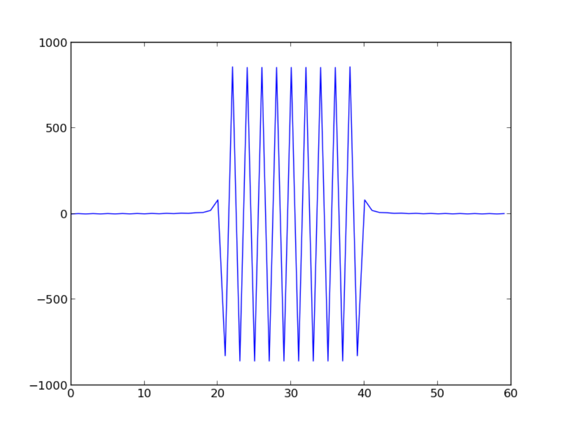




Plots of the output, rewrapped

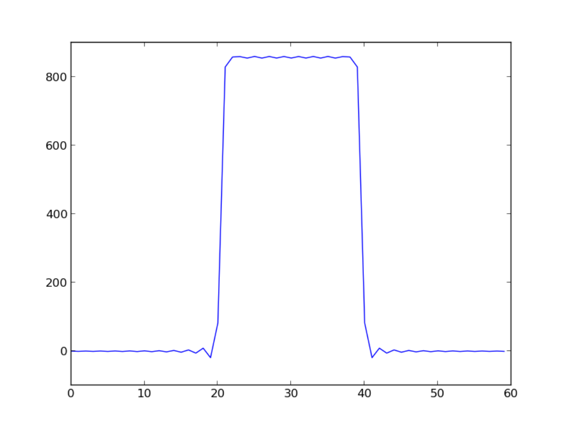
Pick the first and the last 30 points.

```
>>> x1=range(30)
>>> x2=range(len(data)-30,len(data))
>>> y1=data[x1,2]
>>> y2=data[x2,2]
>>> figure()
>>> plot(hstack((y2,y1)))
```



Undo phase factor due to shifting

```
>>> plot(hstack((y2,y1))*array([1,-1]*30)
```



We retrieved our rectangle function!

Precise Relation FT and DFT

- Consider a function on $f(x)$ an interval $[x_1, x_2]$.
- The fourier analysis will express this in terms of periodic functions, so think of f as periodic.
- We will approximate this function with n discrete points on $x_1 + j\Delta x$, where $\Delta x = (x_2 - x_1)/n$, and $j = 0..n-1$, i.e.

$$f(x) = \sum_{j=0}^{n-1} f_j \delta\left(x - (x_1 + j\Delta x)\right) \Delta x$$

- Consider its continuous FT:

$$\hat{f}(k) = \int_{x_1}^{x_2} e^{ikx} f(x)\, dx$$

- e^{ikx} must have period $(x_2 - x_1)$: $k = q \times 2\pi/(x_2 - x_1)$ with q integer.

Precise Relation FT and DFT

Input

$$f(x) = \sum_{j=0}^{n-1} f_j \delta\big(x - (x_1 + j\Delta x)\big)\Delta x$$

$$\Delta x = \frac{x_2 - x_1}{n}$$

$$\hat{f}(k) = \int_{x_1}^{x_2} e^{ikx} f(x)\; dx$$

$$k = \frac{2\pi}{x_2 - x_1} q = \frac{2\pi}{n\Delta x} q$$

Result

$$\hat{f}(k) = e^{ikx_1}\Delta x\; \hat{f}_q$$

$$\hat{f}(k) = \int_{x_1}^{x_2} \sum_{j=0}^{n-1} e^{ikx} f_j \delta\big(x-(x_1+j\Delta x)\big)\Delta x\; dx$$

$$= \sum_{j=0}^{n-1} f_j e^{ik(x_1+j\Delta x)}\Delta x$$

$$= e^{ikx_1}\Delta x \sum_{j=0}^{n-1} f_j e^{ikj\Delta x}$$

$$= e^{ikx_1}\Delta x \sum_{j=0}^{n-1} f_j e^{2\pi i q j/n}$$

Multidimensional transforms

In principle a straighforward generalization:

- Given a set of $n \times m$ function values on a regular grid:

$$f_{ab} = f(a\Delta x, b\Delta y)$$

- Transform these to n other values $\hat{f}_{kl}$

$$\hat{f}_{kl} = \sum_{a=0}^{n-1} \sum_{b=0}^{m-1} f_{ab}\, e^{\pm 2\pi i\,(a\,k+b\;l)/n}$$

- Easily back-transformed:

$$f_{ab} = \frac{1}{nm} \sum_{k=0}^{n-1} \sum_{l=0}^{m-1} \hat{f}_{kl}\, e^{\mp 2\pi i\,(a\,k+b\;l)/n}$$

- Negative frequencies: $f_{-k,-l} = f_{n-k,m-l}$.

Multidimensional FFT

- We could successive apply the FFT to each dimension
- This may require transposes, can be expensive.
- Alternatively, could apply FFT on rectangular patches.
- Mostly should let the libraries deal with this.
- FFT scaling still $n \log n$.

Symmetries for real data

- All arrays were complex so far.
- If input f is real, this can be exploited.

$$f_j^* = f_j \leftrightarrow \hat{f}_k = \hat{f}_{n-k}^*$$

- Each complex number holds two real numbers, but for the input f we only need n real numbers.
- If n is even, the transform $\hat{f}$ has real $\hat{f}_0$ and $\hat{f}_{n/2}$, and the values of $\hat{f}_k > n/2$ can be derived from the complex valued $\hat{f}_{0<k<n/2}$: again n real numbers need to be stored.

Symmetries for real data

- A different way of storing the result is in "half-complex storage' '. First, the $n/2$ real parts of $\hat{f}_{0<k<n/2}$ are stored, then their imaginary parts in reversed order.
- Seems odd, but means that the magnitude of the wave-numbers is like that for a complex-to-complex transform.
- These kind of implementation dependent storage patterns can be tricky, especially in higher dimensions.

Applications?

Application of the Fourier transform

- Signal processing, certainly.
- Many equations become simpler in the fourier basis.
  - Reason: $\exp(ik \cdot x)$ are eigenfunctions of the $\partial/\partial x$ operator.
  - Partial diferential equation become algebraic ones, or ODEs.
  - Thus avoids matrix operations.

Example: Solving a 1D diffusion with FFT

$$\frac{\partial\rho}{\partial t} = \kappa\frac{\partial^2\rho}{\partial x^2}$$

for $\rho(x,t)$ on $x \in [0,L]$, with boundary conditions $\rho(0,t) = \rho(L,t) = 0$, and $\rho(x,0) = f(x)$.

Write

$$\rho(x,t) = \sum_{k=-\infty}^{\infty} \hat\rho_k(t)e^{2\pi ikx/L}$$

then the PDE becomes an ODE:

$$\frac{d\hat\rho_k}{dt} = -\kappa\frac{4\pi^2k^2}{L^2}\hat\rho_k; \qquad \text{with } \hat\rho_k(0) = \hat f_k.$$

Alternatively, one can first discretize the PDE, then take an FFT. This is numerically different.