Discrete Fourier Transforms

Ramses van Zon

PHY1610, Winter 2025

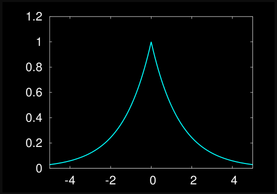
Fourier transform

In this lecture, we will discuss:

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

Fourier transform

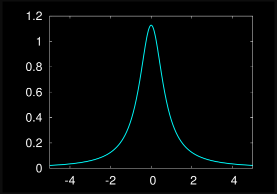
- Let f be a function of a spatial variable x.



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

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

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



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

- Inverse transformation:

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

Fourier transform

- Fourier made the claim that any function can be expressed as a harmonic series.
- The Fourier Transform 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.

Finite interval = periodicity

- Imagine the function is stored numerically as n samples at $x = j\Delta x$.
- Then the function lives on a **finite** x-interval $[0, L)$ where $L = n\Delta x$.
- For a finite interval, only discrete k values are needed, e.g.

$$\hat{f}(k) \propto \int f(x)\, e^{\pm i\,k\,x}\, dx \equiv \hat{f}_q \qquad \text{for } k = \frac{2\pi}{L} q$$

- This is enough to reconstruct the inverse

$$f(x) \propto \sum_{q=-\infty}^{\infty} \hat{f}(k(q))\, e^{\mp i\,k(q)\,x}\, dx$$

- $k(q)$ is such that $e^{\pm ik(q)x}$ has period L,
  so we can view the function f as **periodic** in x with period L.

Discrete case

- We only have the n discrete points $j\Delta x$.
- The Fourier transform is invertible, so it must have the same number of points.
- We'll keep just $q = 0$ up to $q = n - 1$. Then:

$$f_j \equiv f(j\Delta x) \propto \sum_{q=0}^{n-1} \hat{f}(k(q))\, e^{\mp i\, k(q) j \Delta x}\, dx = \sum_{q=0}^{n-1} \hat{f}_q\, e^{\mp 2\pi i\, q\, j}\, dx$$

- Higher q values coincide with lower fourier modes as far as $x = j\Delta x$ is concerned.
- But we have to alter how we compute $\hat{f}$; the integral must become a sum:

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

Discrete Fourier Transform (DFT)

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:

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

- Note: $\hat{f}_{-q} = \hat{f}_{n-q}$.
- Note: Cannot resolve frequencies higher than the Nyquist frequency $q = n/2$ ($k = \pi/\Delta x$).

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>
using complex = std::complex<double>;

void fft_slow(rvector<complex>& f,
              rvector<complex>& fhat,
              bool inverse)
{
  int n = f.size();
  int sign = inverse?-1:1;
  double v = sign*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];
    }
  }
}
```

The inverse left out the $1/n$ normalization; this is common in many implementations.

Fast Fourier Transform (FFT)

- Even Gauss realized $\mathcal{O}(n^2)$ was too slow, and came up with a fast version.
- This fast version was 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. In practice, nobody does.

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.

Inverse DFT

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

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

- 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.

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 not write your own?

- 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)
  - cuFFT
  - Intel MKL
  - IBM ESSL

- Because you have better things to do.

Example of using a library: FFTW

Previous version:

```
#include <complex>
#include <rarray>
#include <cmath>
using complex = std::complex<double>;

void fft_slow(rvector<complex>& f,
              rvector<complex>& fhat,
              bool inverse)
{
  int n = f.size();
  int sign = inverse?-1:1;
  double v = sign*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];
    }
  }
}
```

FFTW version:

```
#include <complex>
#include <rarray>
#include <fftw3.h>
using complex = std::complex<double>;

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

Notes

- Creates a plan first. This is a mandatory step for fftw.
- An fftw_plan contains all information necessary to compute the transform, including the pointers to the input and output arrays.
- FFTW uses its own complex number type, completely compatible with C++'s complex numbers, except C++ does not know that. So, casts.
- 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).
- Link with `-lfftw3` (for double precision).
- For single precision, use `fftwf_` functions and link with `-lfftw3f`. precision too.

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 $\text{sinc}(x)$ is the rectangle function:

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

  up to a normalization.
- Does it match?

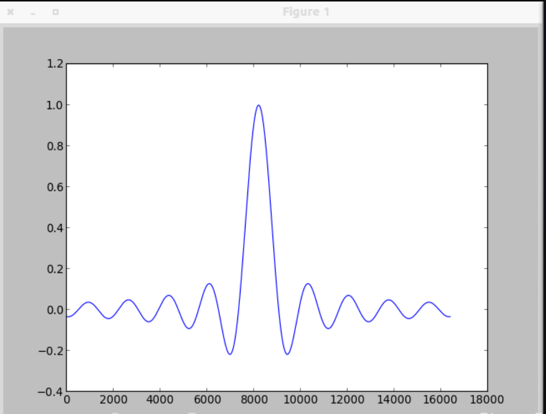
Code for the working example

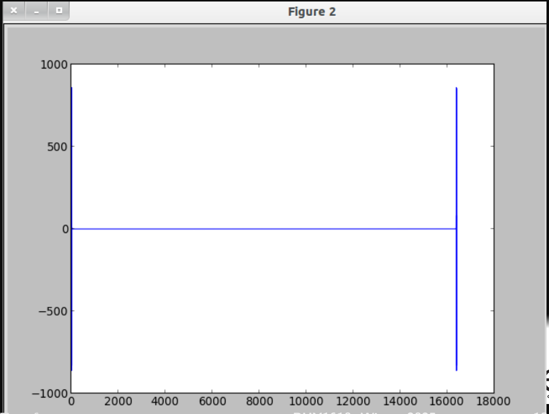
```
//sincfftw.cpp
#include <iostream>
#include <complex>
#include <rarray>
#include <fftw3.h>
using complex = std::complex<double>;
int main() {
    const int n = 16384;
    rvector<complex> f(n), fhat(n);
    for (int i=0; i<n; i++) {
        double x = 60*(i/double(n)-0.5); // x-range from -30 to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

Compile, link, run, plot

```
$ module load gcc/13 rarray fftw/3 python/3
$ g++ -std=c++17 -c -O3 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[:,1])
```

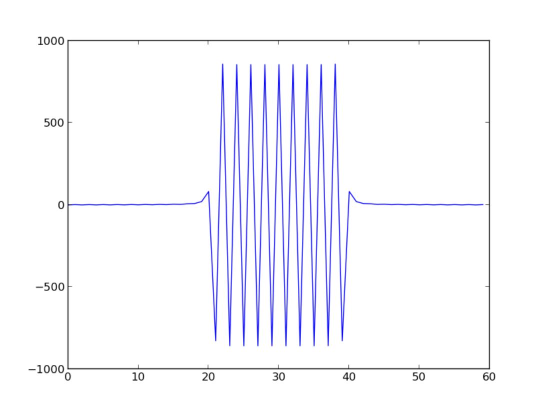




Plots of the output, rewrapped

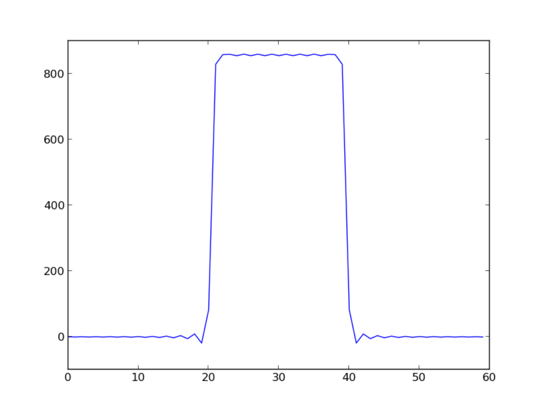
Pick the first and the last 30 points.

```
>>> x1=range(30)
>>> x2=range(len(data)-30,len(data))
>>> y1=data[x1,1]
>>> y2=data[x2,1]
>>> figure()
>>> plot(hstack((y2,y1)))
```



Undo phase factor due to shifting

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



We retrieved our rectangle function!

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

1. Signal processing, certainly.
2. Many equations become simpler in the fourier basis.
   - Reason: $\exp(ik\,x)$ are eigenfunctions of the $\partial/\partial x$ operator.
   - Partial diferential equation become algebraic ones, or ODEs.
   - Thus avoids matrix operations.
3. Optimizing long range particle-particle interactions in N-body simulations and molecular dynamics.

2. Solving diffusion equation with FFT

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

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

Write

$$u(x,t) = \sum_{k=-\infty}^{\infty} \hat{u}_k(t) e^{2\pi i k x/L}$$

then the PDE becomes an ODE:

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

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

3. Long-range particle interactions

- Long-range interactions are those that cannot be cut off without seriously altering the physics.
- Examples of a long range interactions include:
  - Gravity
  - Electrostatics
- In N-body and MD simulations, the force computation is often the bottleneck.
- Without a cut-off (as for short-range) interactions, we are left with a sum over interacting pairs, i.e., an or "Particle-Particle", $\mathcal{O}(N^2)$ method.
- As we saw in the Molecular Dynamics lecture, we can use Particle-Mesh and solve part of the interactions in fourier space; $\mathcal{O}(N \log N)$