# Numerical Tools for Physical Scientists: Fourier Transform

Erik Spence

SciNet HPC Consortium

4 March 2014

# Today's class

Today we will discuss the following topics:

- The Fourier transform, and introduction.
- The discrete Fourier transform.
- The fast Fourier transform.
- Examples using FFTW.

SciNet compute • calcul CANADA

# The Fourier Transform

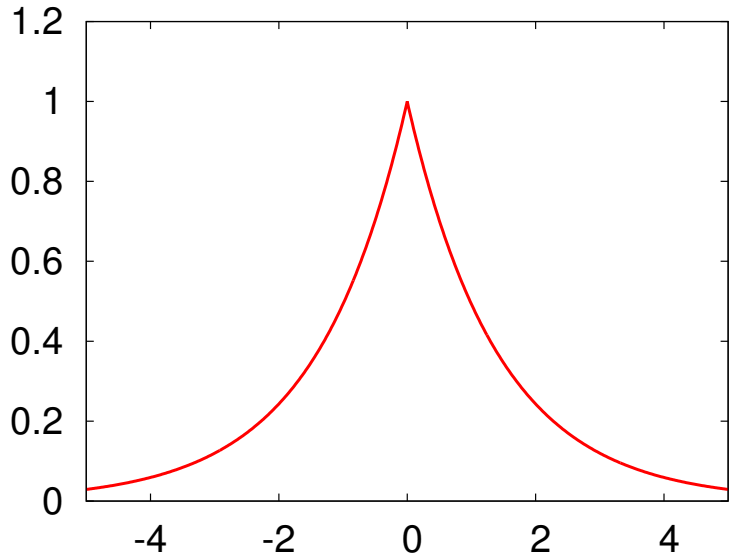
Let $f$ be a function of some variable $x$. Then the Fourier transform can be defined as:

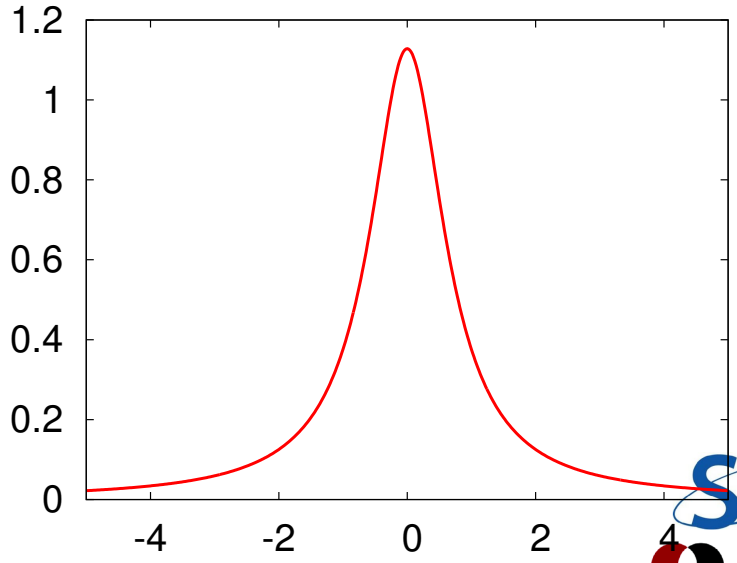
$$\hat{f}(k) = A \int f(x) e^{\pm ikx} dx$$

The transformation can be inverted. If $k$ is continuous:

$$f(x) = \frac{1}{A} \int \hat{f}(k) e^{\mp ikx} dk$$

Where $A$ is a normalization constant.



# The Fourier Transform

- Fourier claimed that any function can be expressed as a harmonic series (series of sines and cosines).
- He was wrong.
- The Fourier transform is the continuous implementation of this claim.
- It constitutes a linear (basis) transformation in function space.
- Transforms from spatial ($x$) to wave-number ($k$), or time ($t$) to frequency ($\omega$), *etc.*
- The constants and signs are just convention (some restrictions apply).

SciNet compute • calcul CANADA

# Application of the Fourier transform

- Many equations become simpler in the Fourier basis.
- Reason:

  $e^{ikx}$ is an eigenfunction of the $\dfrac{\partial}{\partial x}$ operator

- Partial differential equations become algebraic, or ODEs.
- This avoids matrix operations.

Examples:

- Periodic phenomena.
- Spectral analysis.
- Signal processing/filtering.
- PDEs: virtually anything (linear) with a Laplacian.

SciNet compute • calcul CANADA

# Examples

Heat equation:

$$\frac{\partial \mathbf{u}}{\partial t} = \alpha \nabla^2 \mathbf{u} \quad \rightarrow \quad \frac{\partial \hat{\mathbf{u}}}{\partial t} = -\alpha \left|\boldsymbol{k}\right|^2 \hat{\mathbf{u}}$$

Schrödinger equation:

$$i\hbar \frac{\partial \Psi}{\partial t} = -\frac{\hbar^2}{2m} \nabla^2 \Psi \quad \rightarrow \quad i\hbar \frac{\partial \hat{\Psi}}{\partial t} = \frac{\hbar^2 \left|\boldsymbol{k}\right|^2}{2m} \hat{\Psi}$$

# Discrete Fourier transform (DFT)

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

$$f_j = f(j\Delta x),$$

where $\Delta x$ is the grid spacing, transform these to $n$ other values $\hat{f}_k$:

$$\hat{f}_k = \sum_{j=0}^{n-1} f_j e^{\pm 2\pi i j k/n}.$$

This is easily back-transformed:

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

The solution is periodic: $\hat{f}_{-k} = \hat{f}_{n-k}$; you run the risk of aliasing: $k$ is equivalent to $k + \ell n$. Consequently the max frequency you can resolve is $k = n/2$ (the Nyquist frequency).

# Slow Fourier transform

- The discrete Fourier transform is a linear transformation.
- In particular, it is a matrix-vector multiplication.
- Naively, this scales as $\mathcal{O}(n^2)$. Slow!
- Same scaling as many solvers.

# Slow DFT

```
#include <complex>
#include <cmath>
typedef std::complex<double> complex;

void fftn2(int n, complex *f, complex *fhat, int dir) {

  complex *w = new complex[n];
  double v = (dir < 0 ? -2 : 2) * M_PI / n;

  for(int i = 0; i < n; i++)  w[i] = complex(cos(v * i), sin(v * i));

  for(int i = 0; i < n; i++) {
    fhat[i] = 0.0;
    for(int j = 0; j < n; j++)  fhat[i] += w[(i * j) % n] * f[j];
  }
  delete [] w;
};
```

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 $n/2$ point FTs.
- Do this recursively $\log_2 n$ times.
- Each level requires $\sim n$ computations, which means $\mathcal{O}(n \log n)$ instead of $\mathcal{O}(n^2)$.
- Could have as easily split the problem into 3, 5, 7, ..., parts.
- Because it is based on base 2, it is ALWAYS faster to use a number of points that is a power of two (1024 is way way faster than 1023).

SciNet compute • calcul CANADA

# $\mathcal{O}(n \log_2 n)$ vs $\mathcal{O}(n^2)$ doesn't impress you?

Recall that $n$ is the size of the problem.

| $n$ | $n \log_2 n$ | $n^2$ | ratio |
|---|---|---|---|
| 32 | 160 | 1,024 | 6 |
| 128 | 896 | 16,384 | 18 |
| 512 | 4,608 | 262,144 | 57 |
| 2,048 | 22,528 | 4,194,304 | 186 |
| 8,192 | 106,496 | 67,108,864 | 630 |

SciNet compute • calcul CANADA

# How does it work, exactly?

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

$$\hat{f}_k = \sum_{j=0}^{n-1} \omega_n^{kj} f_j$$

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

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

- Repeat for all $k$.
- Note that a fair amount of shuffling is involved.

SciNet compute • calcul CANADA

# Fast Fourier transform: it's already been done!

We've said it before and we'll say it again: don't write your own implementation! Why?

- Because getting all the pieces correct is tricky.
- Beause getting it run fast requires intimate knowledge of how processors work and access memory.
- Because it's already been done for you:
  - FFTW3 (Fastest Fourier Transform in the West, version 3).
  - Intel MKL.
  - IBM ESSL.
- Because you have better things to do.

SciNet compute • calcul CANADA

# Example of using FFTW

```
#include <complex>
#include <fftw3.h>
typedef std::complex<double> complex;

void fftw(int n, complex *f, complex *fhat, int dir) {

  fftw_plan p = fft_plan_dft_1d(n, (fftw_complex*)f, (fftw_complex*)fhat,
    dir < 0 ? FFTW_BACKWARD : FFTW_FORWARD, FFTW_ESTIMATE);

  fftw_execute(p);
  fftw_destroy_plan(p);
};
```

# Inverse DFT

- Inverse DFT is similar to the forward DFT, up to a normalization; almost as fast.

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

- Almost all implementations leave out the $1/n$ normalization.
- FFT allows quick transformations between the $x$ and $k$ domains (time and frequency, for example).
- Allows parts of the computation or analysis to be done in the most convenient of efficient domain.

# Working example

- Create a 1D input signal: a discretized $f(x) = \mathbf{sinc(x)} = \mathbf{sin(x)/x}$, with 16384 points on the interval [-30:30].
- Perform the forward transform.
- Write to standard out.
- Compile and link to the FFTW3 library.
- Continuous FT of $\mathbf{sinc(x)}$ is

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

  up to a normalization.
- Does it match?

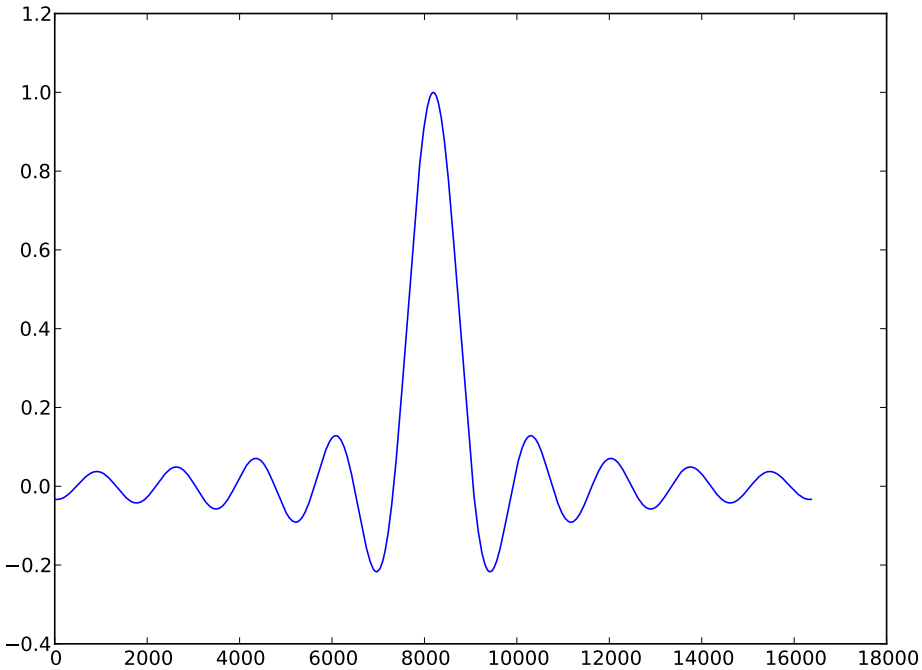
# Working example, continued

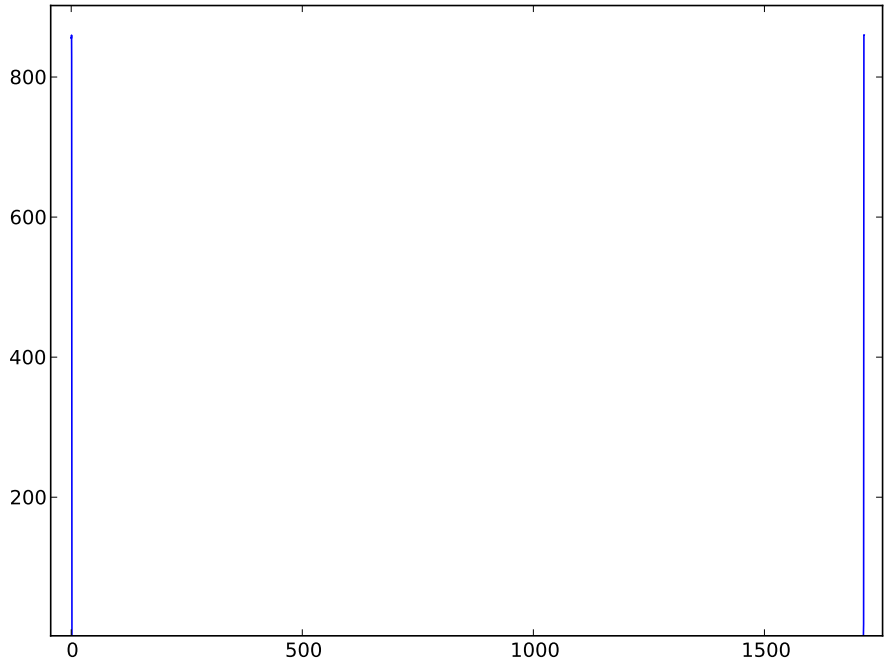
```
// sincfftw.cpp
#include <iostream>
#include <complex>
#include <cmath>
#include <fftw3.h>
typedef std::complex<double> complex;

int main() {
  int n = 16384;
  double len = 60, dx = len / n;
  complex *f = new complex[n];
  complex *g = new complex[n];
  for(int i = 0; i < n; i++) {
    double x = (i - n/2 + 1.0e-5) * dx;
    f[i] = sin(x) / x;
  }
```

```
  fftw_plan p =
    fft_plan_dft_1d(n,
    (fftw_complex*)f, (fftw_complex*)g,
    FFTW_FORWARD, FFTW_ESTIMATE);

  fftw_execute(p);
  fftw_destroy_plan(p);

  for(int i = 0; i < n; i++)
    std::cout << f[i] << ","
      << g[i] << std::endl;

  delete [] f;
  delete [] g;
  return 0;
}
```

```
ejspence@mycomp ~> g++ -O2 -Wall sincfftw.cpp -lfftw3 -o sincfftw
ejspence@mycomp ~> ./sincfftw | tr -d '()' > output.dat
```

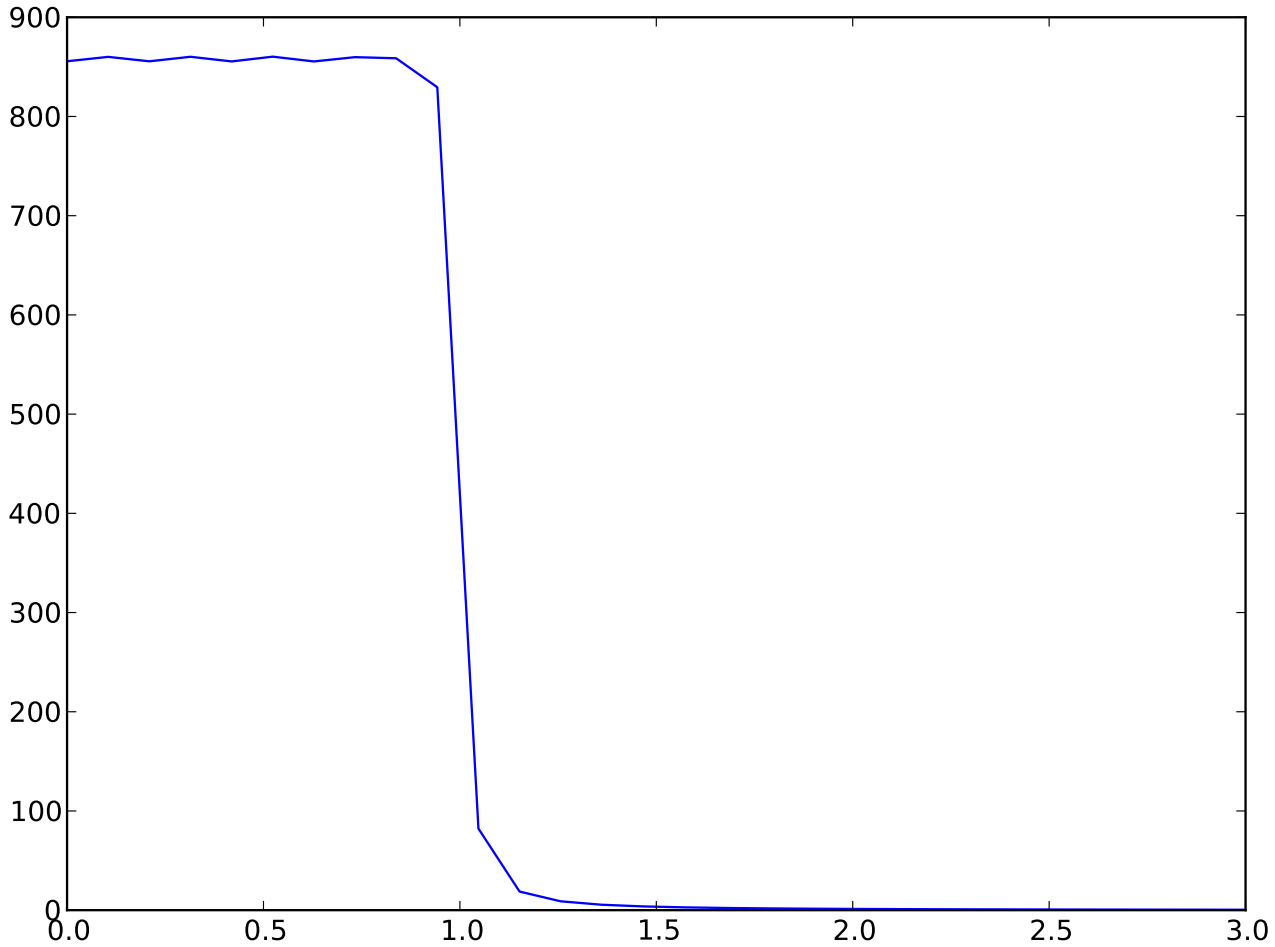
# Plot the output



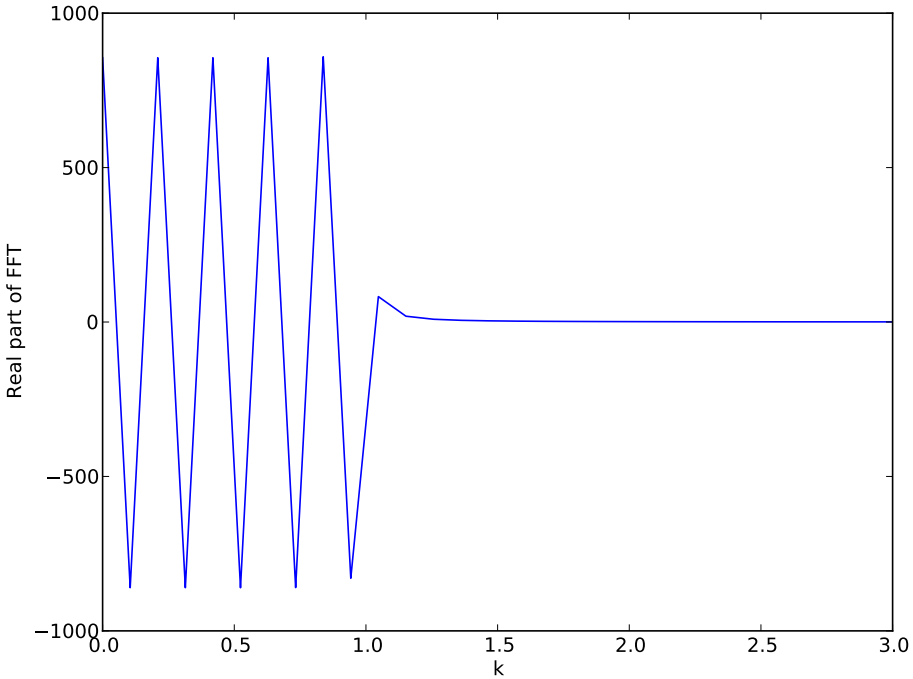

```
ejspence@mycomp ~> ipython --pylab
In [1]: data = genfromtxt('output.dat', delimiter = ',')
In [2]: plot(data[:,0])
In [3]: figure()
In [4]: k = scipy.arange(0, scipy.size(data[:,2])) * 2 * pi / 60.
In [5]: plot(k, abs(data[:,2]))
```

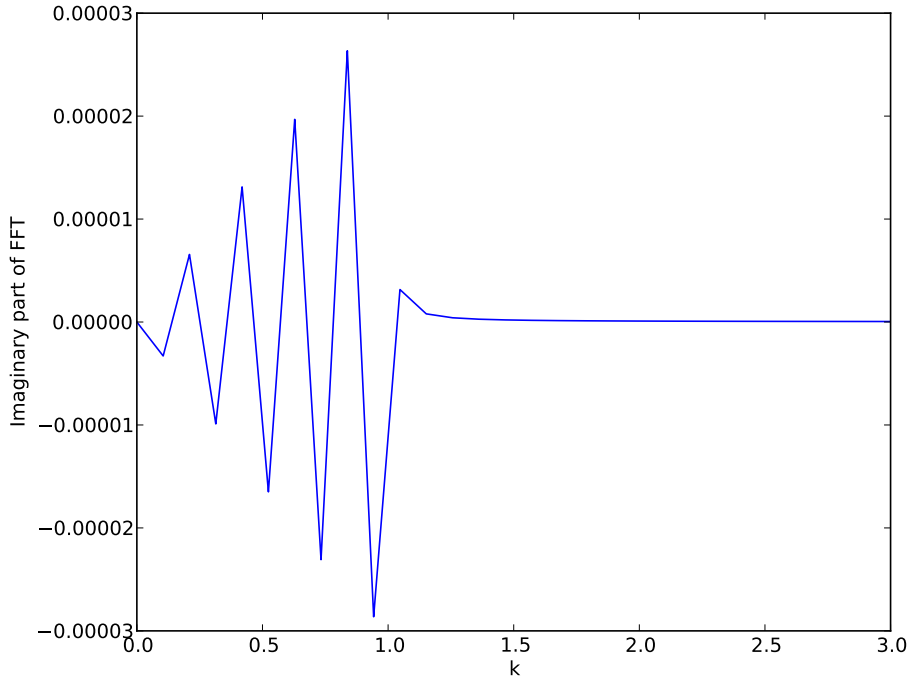
# Plot the output




```
In [6]: xlim(0,3)
```

# Discretization, aliasing, shifting effects




```
In [7]: figure()
In [8]: plot(k, data[:,2])
In [9]: xlim(0,3)
In [10]: figure()
In [11]: plot(k, data[:,3])
In [12]: xlim(0,3)
```