Scientific Computing (Phys 2109/Ast 3100H)
II. Numerical Tools for Physical Scientists

SciNet HPC Consortium, University of Toronto
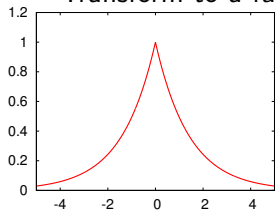
Lecture 15: Fast Fourier Transform

Winter 2013

Fourier Transform (FT)

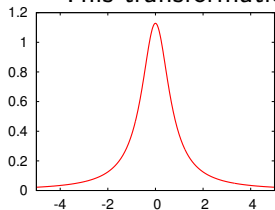▶ Let **f** be a function of some variable **x**.

▶ Transform to a function $\hat{\mathbf{f}}$ of **k**:

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

J. Fourier

▶ This transformation can be inverted. If **k** is continuous:

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

Fourier Transform (FT)

- 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.

Application of the Fourier transform

- Many equations become simpler in the fourier basis.

- Reason: $\exp(i\mathbf{k} \cdot \mathbf{x})$ are eigenfunctions of the $\partial/\partial\mathbf{x}$ operator.

- Partial diferential equation become algebraic ones, or ODEs.

- Thus avoids matrix operations.

Examples

- Periodic phenomena

- Spectral analysis

- Signal processing/filtering

- PDEs: virtually anything with a Laplacian

Application of the Fourier transform: examples

Heat equation

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u$$

$$\Downarrow$$

$$\frac{d\hat{u}}{dt} = -\alpha \|k\|^2 \hat{u}$$

Schrödinger equation:

$$i\hbar \frac{\partial \Psi}{\partial t} = -\frac{\hbar^2}{2m} \nabla^2 \Psi$$

$$\Downarrow$$

$$i\hbar \frac{d\hat{\Psi}}{dt} = \frac{\hbar^2 \|k\|^2}{2m} \hat{\Psi}$$

Discrete Fourier Transform (DFT)

▶ Given a set of **n** function values on a regular grid:

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

▶ Transform these to **n** other values $\hat{f}_k$

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

C. F. Gauss

▶ Easily back-transformed:

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

▶ Negative frequencies: $f_{-k} = f_{n-k}$.

▶ General aliasing: **k** becomes equivalent to $k + \ell n$. (max frequency = $k = n/2$: Nyquist)

Slow Fourier Transform

- Discrete fourier transform is a linear transformation.

- In particular, it's a matrix-vector multiplication.

- Naively, costs $\mathcal{O}(n^2)$. Slow!

- Same scaling as many solvers.

Slow DFT

```cpp
#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 j=0; j<n; j++)
        w[j] = complex(cos(v*j), sin(v*j));
    for (int k=0; k<n; k++) {
        fhat[k] = 0.0;
        for (int l=0; l<n;l++)
            fhat[k] += w[(k*l)%n]*f[l];
    }
    delete[] w;
}
```

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

Fast Fourier Transform (FFT)

C. F. Gauss

Derived in partial form several times before and even after Gauss, because he'd just written it in his diary in 1805 (published later).

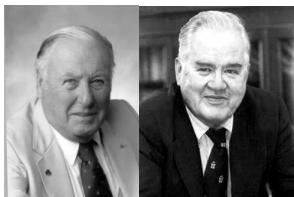Rediscovered (in general form) by Cooley and Tukey in 1965.

Basic idea

J. W. Cooley   J. Tukey

- ▶ 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: $\mathcal{O}(\mathbf{n \log n})$ instead of $\mathcal{O}(\mathbf{n^2})$.

- ▶ Could as easily into 3, 5, 7, ... parts.

If $\mathcal{O}(n \log n)$ versus $\mathcal{O}(n^2)$ does not impress you...

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

Fast Fourier Transform: How can you do that?

- Define $\omega_n = e^{2\pi i/n}$.
  Note that $\omega_n^2 = \omega_{n/2}$.

- DFT takes form of matrix-vector multiplication:

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

- 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, until the lowest level (for $n = 1$, $\hat{f} = f$).

- Note that a fair amount of shuffling is involved.

Fast Fourier Transform: How do you really do that?

Do not write your own: use existing libraries.

- 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: fftw, Intel mkl, IBM essl

- Because you have better things to do.

Example of using a library: FFTW

Example (rewrite of slow ft)

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


typedef std::complex<double> complex;


void fftw(int n, complex* f, complex* fhat, int dir)
{
    fftw_plan p = fftw_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 forward DFT, up to a normalization: almost just as fast.

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

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

- FFT allows quick back-and-forth between **x** and **k** domain (or e.g. time and frequency domain).

- Allows parts of the computation and/or analysis to be done in the most convenient or efficient domain.

Working 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 **sinc(x)** is

$$\mathbf{rect(f)} = \left\{ \begin{array}{ll} \mathbf{a} & \text{if } |\mathbf{f}| \leq \mathbf{b} \\ \mathbf{0} & \text{if } |\mathbf{f}| > \mathbf{b} \end{array} \right.$$

- Let's see if it matches?

```cpp
#include <iostream>
#include <complex>
#include <cmath>
#include <fftw3.h>
typedef std::complex<double> complex;
int main(int argc, char ** argv) {
    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 = fftw_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;
}
```
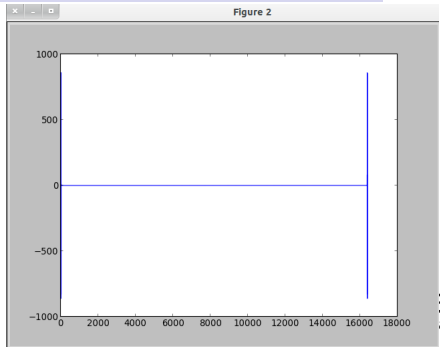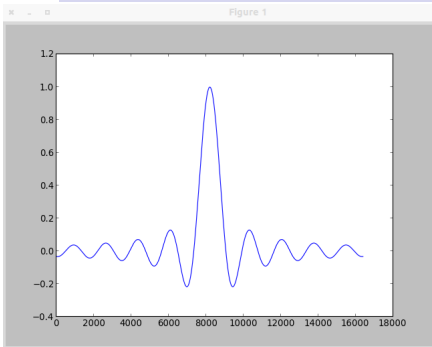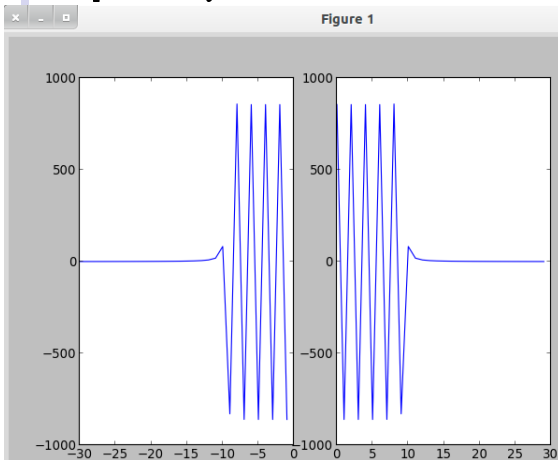
Compile, link, run

```
$ g++ -c -O2 -Wall sincfftw.cc -o sincfftw.o
$ g++ sincfftw.o -o sincfftw -lfftw3
$ ./sincfftw | tr -d '()' > output.dat
```
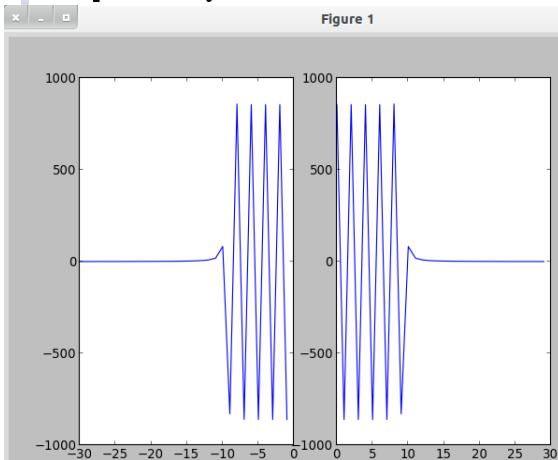
Plots

```
$ ipython --pylab
>>> data=genfromtxt('output.dat',delimiter=',')
>>> plot(data[:,0])
>>> figure()
>>> plot(data[:,2])
```

```
>>> x1=range(30)
>>> x2=range(len(data)-30,len(data))
>>> y1=data[x1,2]
>>> y2=data[x2,2]
>>> subplot(1,2,1)
>>> plot(range(-30,0),y2)
>>> subplot(1,2,2)
>>> plot(x1,y1)
```

```
>>> x1=range(30)
>>> x2=range(len(data)-30,len(data))
>>> y1=data[x1,2]
>>> y2=data[x2,2]
>>> subplot(1,2,1)
>>> plot(range(-30,0),y2)
>>> subplot(1,2,2)
>>> plot(x1,y1)
```


Aliasing

Phase factor