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

SciNet HPC Consortium, University of Toronto

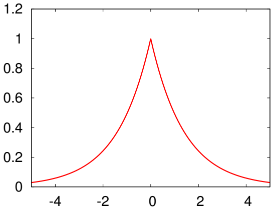
Lecture 15: Fast Fourier Transform

Winter 2013

Fourier Transform (FT)

- Let $\mathbf{f}$ be a function of some variable $\mathbf{x}$.

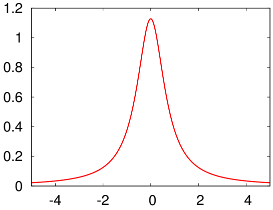
- Transform to a function $\hat{\mathbf{f}}$ of $\mathbf{k}$:



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

J. Fourier

- This transformation can be inverted. If $\mathbf{k}$ is continuous:



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

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: $\mathbf{exp(ik \cdot x)}$ 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

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

$$\Downarrow$$

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

Schrödinger equation:

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

$$\Downarrow$$

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

Discrete Fourier Transform (DFT)

C. F. Gauss

- Given a set of $\mathbf{n}$ function values on a regular grid:

$$\mathbf{f_j = f(j\Delta x)}$$

- Transform these to $\mathbf{n}$ other values $\mathbf{\hat{f}_k}$

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

- Easily back-transformed:

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

- Negative frequencies: $\mathbf{f_{-k} = f_{n-k}}$.
- General aliasing: $\mathbf{k}$ becomes equivalent to $\mathbf{k + \ell n}$. (max frequency $= \mathbf{k = n/2}$: Nyquist)

Slow Fourier Transform

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

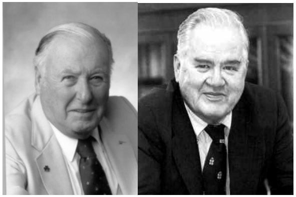
C. F. Gauss

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

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


J. W. Cooley J. Tukey

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×
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×

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)

```
#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 $\mathbf{sinc(x) = sin(x)/x}$ with 16384 points on the interval [-30:30].
- Perform forward transform
- Write to standard out
- Compile, and linking to fftw3 library.
- Continous FT of $\mathbf{sinc(x)}$ is

$$\mathbf{rect(f)} = \begin{cases} \mathbf{a} & \text{if } |\mathbf{f}| \leq \mathbf{b} \\ \mathbf{0} & \text{if } |\mathbf{f}| > \mathbf{b} \end{cases}$$

- Let's see if it matches?

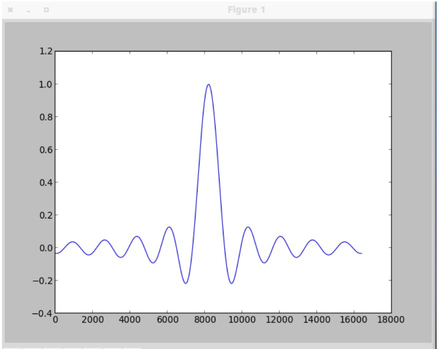
```cpp
#include <iostream>
#include <complex>
#include <cmath>
#include <fftw3.h>
typedef std::complex<double> complex;
int main(int argc, char ** argv) {
   int n = 16384;
   double len = 60, dx = len/n;
   complex* f = new complex[n];
   complex* g = new complex[n];
   for (int i=0; i<n; i++) {
      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;
}
```

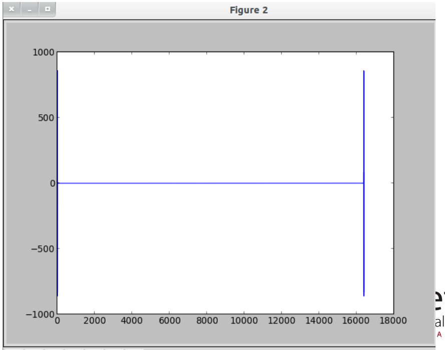
Compile, link, run

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

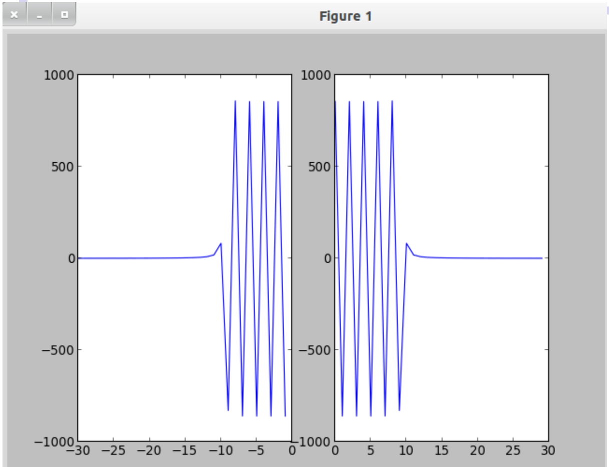
Plots

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

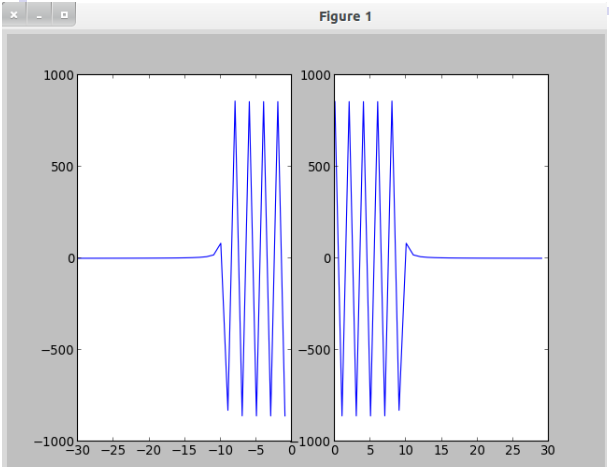




```
>>> x1=range(30)
>>> x2=range(len(data)-30,len(data))
>>> y1=data[x1,2]
>>> y2=data[x2,2]
>>> subplot(1,2,1)
>>> plot(range(-30,0),y2)
>>> subplot(1,2,2)
>>> plot(x1,y1)
```



```
>>> 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

About FFTW

Supposedly the "Fastest Fourier Transform in the West"

version 2 **! =** version 3

Capabilities

- Complex one-dimensional transforms
- Complex multi-dimensional transforms.
- Real-to-half-complex array transforms
- Format real transforms different in 1d and nd.
- Threaded, MPI, SIMD vectorized
- Read the manual!

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.

Symmetries for real data

- All arrays were complex so far.
- If input $\mathbf{f}$ is real, this can be exploited.

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

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

Symmetries for real data

- A different way of storing the result is in "half-complex storage". First, the $\mathbf{n/2}$ real parts of $\hat{\mathbf{f}}_{\mathbf{0<k<n/2}}$ 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.

Multidimensional transforms

In principle a straighforward generalization:

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

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

- Transform these to $\mathbf{n}$ other values $\hat{f}_k$

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

- 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**.
- Real transform even more convoluted.