Astronomy 8824: Numerical Methods Notes 4
FFTs and Their Applications

Reference: Numerical Recipes, Chapters 12 and 13. Specifically, you should read 12.0-12.2, glance through other sections of 12, and read 13.0-13.2.

The classic (and good) reference on Fourier Transforms is Ronald Bracewell's book, The Fourier Transform and Its Applications.

Googling python fft documentation is also useful.

Definitions

I will mostly use the NR notation, but not always.

If $h(t)$ is a function of time, then its Fourier transform is

$$H(f) = \int_{-\infty}^{\infty} h(t) e^{2\pi i f t} dt.$$

We can recover $h(t)$ from $H(f)$ by applying an inverse Fourier transform,

$$h(t) = \int_{-\infty}^{\infty} H(f) e^{-2\pi i f t} dt,$$

i.e., just change the sign in the exponent.

If t is a time in sec, then f is a frequency in sec^{-1} (Hz).

Another common notation is to write the FT of $f(x)$ as

$$\tilde{f}(k) = \int_{-\infty}^{\infty} f(x) e^{2\pi i k x} dx.$$

This is common when x is a unit of length, in which case the wavenumber k has units of inverse length.

Caution: Other Fourier conventions are frequently used, such as

$$H(\omega) = \int_{-\infty}^{\infty} h(t) e^{i\omega t} dt, \qquad h(t) = \frac{1}{2\pi} \int_{-\infty}^{\infty} H(\omega) e^{-i\omega t} dt.$$

The NR convention (2π in the exponent) is standard for numerical FTs, and it leaves you with less to remember.

However, you always have to check Fourier conventions when comparing to or using papers or books, and figuring out the conversions can be a pain.

If you use Fourier transforms in a paper, make clear what convention you are using.

Why are Fourier transforms so useful?

Physical description is sometimes more natural in the Fourier domain.

Some mathematical operations (e.g., solving some differential equations) are easier in the Fourier domain, or it may be possible to apply an approximate solution to modes of long or short wavelength.

Periodic signals are often interesting. FT can be an efficient way to look for them.

Some numerical operations can be done *much* faster via FFT if the number of data points N is large.

For example, convolution is an operation $O(N^2)$ if done directly for two functions of length N. (Sometimes the response function is much shorter than the function being convolved, in which case the scaling is only $O(N_{\rm data}N_{\rm response})$.)

But the FFT can be computed in $O(N\log_2 N)$ time, which may be orders-of-magnitude less.

Decomposition, identities

Recall that $e^{2\pi ift} = \cos(2\pi ft) + i\sin(2\pi ft)$.

The FT relation expresses a decomposition of the function $h(t)$ into a sum (integral in the continuous case) of sinusoidal functions.

The sinusoidal basis is complete and orthogonal. Any function can be decomposed this way, and different modes are independent, in that, for positive integers m, n:

$$\int_{-\pi}^{\pi} \cos nx \cos mx dx = \pi\delta_{mn},$$
$$\int_{-\pi}^{\pi} \sin nx \sin mx dx = \pi\delta_{mn},$$
$$\int_{-\pi}^{\pi} \cos nx \sin mx dx = 0.$$

In general, a Fourier transform maps a complex function into another complex function.

If $h(t)$ is real, then

$$H(f) = \int_{-\infty}^{\infty} \cos(2\pi ft)\, h(t)dt + i\int_{-\infty}^{\infty} \sin(2\pi ft)\, h(t)dt$$

is Hermitian, $H(f) = [H(-f)]^*$, because cosine is an even function and sine is an odd function.

If $h(t)$ is real and symmetric about zero then $H(f)$ is real (see NR for other symmetries).

Other useful identities:

If $h(t)$ and $H(f)$ are an FT pair then

$$h(at) \text{ is a Fourier transform pair with } \frac{1}{|a|}H(f/a).$$

If a function is narrow in the time domain, then its Fourier transform is broad in the frequency domain, and vice versa.

For example (a useful one), with our Fourier convention, the FT of a time-domain Gaussian $e^{-t^2/2\sigma_t^2}$ is a frequency-domain Gaussian $e^{-2\pi^2\sigma_t^2 f^2} = e^{-f^2/2\sigma_f^2}$ with $\sigma_f = 1/(2\pi\sigma_t)$.

With the $H(\omega)$ convention, $\sigma_\omega = 1/\sigma_t$.

A shift of origin in the time domain is equivalent to shifting the complex phase in the Fourier domain,

$$h(t-t_0) \text{ is a Fourier transform pair with } H(f)e^{2\pi i f t_0}.$$

The FT is a linear operation, so the FT of a linear combination of functions is the linear combination of their FTs.

The Convolution Theorem

The convolution of functions g and h is

$$g * h \equiv \int_{-\infty}^{\infty} g(\tau)h(t-\tau)d\tau.$$

The FT of $g * h$ is $G(f)H(f)$, i.e., the Fourier transform of the convolution is the product of the Fourier transforms.

Nyquist frequency, aliasing

If a function is tabulated at discrete, evenly spaced locations with separation Δ, then the highest frequency Fourier component that can be represented is one with critical frequency

$$f_c = \frac{1}{2\Delta},$$

known as the Nyquist frequency.

A Nyquist frequency sine wave is sampled with two points per cycle.

If the function $h(t)$ is *bandwidth limited*, with the amplitude of all Fourier modes with $f > f_c$ equal to zero, then the sampled values at spacing Δ are enough to compute $h(t)$ exactly at any point using sinc interpolation (NR eq. 12.1.3).

This is a remarkable fact, and a useful one, e.g., for reconstructing a time series or shifting images to register them *IF* the PSF is fully sampled.

However, if it has power at $f > f_c$, then that power will spuriously contaminate Fourier modes with $f < f_c$.

This is known as aliasing.

One-Dimensional Fast Fourier Transform

Suppose that we have a function $h(t)$ tabulated on a grid of N evenly spaced values.

An FFT is an efficient algorithm for computing the *discrete Fourier transform*, the sum

$$H_n \equiv \sum_{k=0}^{N-1} h_k e^{2\pi i\, kn/N},$$

for all values $n = -N/2$ to $N/2$.

In this notation, k is indexing time or position values; n represents frequency or wavenumber.

Naively, it looks like computing this for all n should be an $O(N^2)$ operation.

The FFT uses the fact that a discrete Fourier transform of length N can be written as a sum of two discrete Fourier transforms of length $N/2$.

This rule can be applied recursively to turn computing the Fourier transform into an $O(N \log_2 N)$ operation, vastly more efficient.

In practice, instead of n running from $-N/2$ to $N/2$, it runs from 0 to $N-1$. The 0 value represents zero frequency $n = 0$, the values $1 \leq n \leq N/2 - 1$ correspond to frequencies $0 < f < f_c$, the values $N/2 + 1 \leq n \leq N - 1$ correspond to $-f_c < f < 0$, and the value $n = N/2$ corresponds to both $f = f_c$ and $f = -f_c$.

Here $f_c = N/2$ is the critical or Nyquist frequency, the maximum frequency that can be represented by N evenly spaced values of the function.

From the above equation, one can see that for $n = N/2$

$$H_n = \sum h_k \left(e^{i\pi}\right)^k = \sum h_k (-1)^k = \sum h_k \left(e^{-i\pi}\right)^k = H_{-n}.$$

The discrete inverse Fourier transform is

$$h_k = \frac{1}{N} \sum_{n=0}^{N-1} H_n e^{-2\pi i\, kn/N}.$$

A forward transform followed by an inverse transform returns the original data values h_k.

Some FFTs have a different normalization convention, so that there is no $1/N$ and a forward-then-inverse transform returns the original values multiplied by N.

In general, be careful about normalization issues and about which order the data are stored in. You will typically need to do some experimentation to make sure you have everything right.

One-dimensional Real-Valued FFT

An FFT transforms an array of complex numbers to another array of complex numbers.

If you want to Fourier transform an array of real numbers, you can just set the imaginary parts to zero before applying a regular FFT.

The outputs will be complex numbers, but they will satisfy the *Hermitian* condition $H(f) = H^*(-f)$, so they still contain only N pieces of information.

This strategy is fine if you are not strained for cpu time or memory, but it is clearly wasting some calculational energy, and it requires your array to be twice as long as it really needs to be.

A real-to-complex FFT routine takes a real array and returns the independent values of the Fourier transform.

This is especially valuable when you get to multi-dimensional FFTs, whose storage requirements scale as $N^{n_{\rm dim}}$.

In python, try

```
load numpy as np

x=np.random.random(8)

y=np.fft.fft(x)

y

z=np.fft.ifft(y)

z
```

You'll notice that while you started with a real array x, the FT y is complex, and z is also complex (but with zero imaginary values).

Then change `np.fft.fft` to `np.fft.rfft` and `np.fft.ifft` to `np.fft.irfft`.

Multi-dimensional FFT

A 2-dimensional FFT is

$$H(n_1,n_2) \equiv \sum_{k_2=0}^{N_2-1} \sum_{k_1=0}^{N_1-1} e^{2\pi i\, k_2 n_2/N_2} e^{2\pi i\, k_1 n_1/N_1} h(k_1,k_2).$$

N-dimensional FFTs can be computed by repeated applications of 1-d FFTs, with a fair amount of bookkeeping.

It is generally easiest to use a routine specifically written for multi-dimensional FFTs.

If you are starting with a real array, you should try to use a real-to-complex routine to save memory.

Pay careful attention to how the data are stored — this may be different from one routine to another, and it is the trickiest thing to figure out.

Calculating gravitational accelerations by FFT

Computing gravitational potentials is an interesting special case of convolution.

Suppose we want to compute the gravitational forces in a system of N bodies, where N might be very large ($10^6 - 10^9$, say) to represent a galaxy or a large cosmological volume. A naive calculation of

$$\mathbf{a}_j = \sum_{i=1}^{N} -\frac{Gm_i\mathbf{r}_{ij}}{r_{ij}^3}$$

is $O(N^2)$ for N particles.

Impossible to do large N. Need more efficient method.

One approach:

$$\nabla^2\phi = 4\pi G\rho$$
$$\phi(\mathbf{r}) = \int \widetilde{\phi}(\mathbf{k})e^{-2\pi i\mathbf{k}\cdot\mathbf{r}}d^3k,$$

where

$$\widetilde{\phi}(\mathbf{k}) = \int \phi(\mathbf{r})e^{2\pi i\mathbf{k}\cdot\mathbf{r}}d^3r$$

is the Fourier transform of ϕ.

$$\vec{\nabla}\phi = -2\pi i\mathbf{k}\int \widetilde{\phi}(\mathbf{k})e^{-2\pi i\mathbf{k}\cdot\mathbf{r}}d^3k$$
$$\nabla^2\phi = -4\pi^2k^2\int \widetilde{\phi}(\mathbf{k})e^{-2\pi i\mathbf{k}\cdot\mathbf{r}}d^3k.$$

Hence $4\pi G\rho(r) = \nabla^2\phi(r)$ implies

$$4\pi G \int \widetilde{\rho}(\mathbf{k})e^{-2\pi i\mathbf{k}\cdot\mathbf{r}}d^3k = -4\pi^2k^2\int \widetilde{\phi}(\mathbf{k})e^{-2\pi i\mathbf{k}\cdot\mathbf{r}}d^3k,$$

implying

$$\widetilde{\phi}(\mathbf{k}) = -\frac{G}{\pi k^2}\widetilde{\rho}(\mathbf{k}).$$

This can be a very efficient means of computing the gravitational potential because of numerically efficient Fourier transform routines.

Basic scheme:

Compute $\rho(x, y, z)$ on grid from particle distribution by interpolation.

Compute $\widetilde{\rho}(k_x, k_y, k_z)$ by FFT, then $\widetilde{\phi}$ from above equation.

Inverse FFT to get $\phi(x, y, z)$.

Numerically differentiate $\phi(x, y, z)$ to get accelerations.

There are various subtleties about how to do the differentiation, and about the best choice of "Green's function" (which may not be simply $1/k^2$).

Forces are automatically softened (relative to $1/r^2$) on the scale of grid cell.

If this is adequate force resolution, then the FFT method is very efficient.

The FFT method naturally imposes a periodic boundary condition, which is sometimes desirable (e.g., a cosmological volume) and sometimes not (e.g., a galaxy merger).

FFTW

The best library of routines for high-performance FFTs is probably FFTW, "The Fastest Fourier Transform in the West," freely available from www.fftw.org.

The FFTW routines analyze your machine to decide what algorithm will actually run fastest on your available hardware.

Equally important, the library includes multi-dimensional FFTs, real-to-complex FFTs, and works with any dimension of array, not just powers of 2.

You will probably get the best performance if you use array sizes that are multiples of powers of 2, 3, and 5.

There is a `pyfftw` package that supposedly uses `fftw` implementations with python wrappers so that they are called like the `numpy.fft` routines.

I have not investigated these, and I do not know how the `pyfftw` routines compare to the `numpy.fft` routines for speed.

If you're working in python, my advice is start with `numpy.fft` and investigate `pyfftw` if you need more speed.

If you're working in C or fortran, you may want to use FFTW routines (which are in C) directly.