Parallel Computation of Fourier Series, with an Introduction to Parallel Imaging

Norman Matloff
Dept. of Computer Science
University of California at Davis

May 23, 2006

Contents

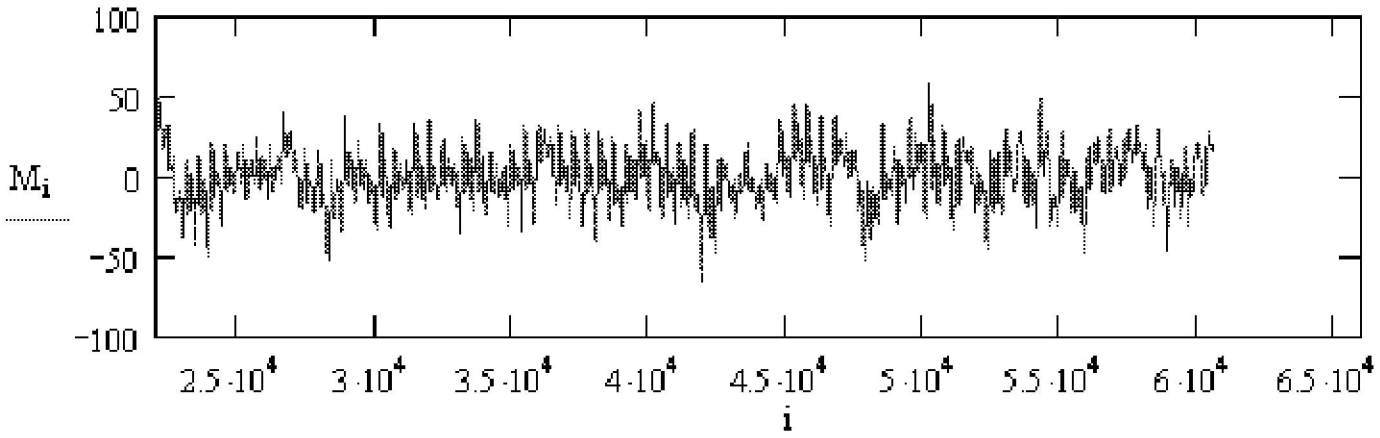
1 Fourier Analysis

Mathematical computations involving sounds and images, for example for voice and pattern recognition are often performed using **Fourier** analysis.

1.1 General Principles

1.1.1 One-Dimensional Fourier Series

A sound **wave form** graphs volume of the sound against time. Here, for instance, is the wave form for a vibrating reed:[1]



Recall that we say a function of time g(t) is **periodic** ("repeating," in our casual wording above) with period T if if g(u+T) = g(u) for all u. The **fundamental frequency** of g() is then defined to be the number of periods per unit time,

$$f_0 = \frac{1}{T} \tag{1}$$

Recall also from calculus that we can write a function g(t) (not necessarily periodic) as a Taylor series, which is an "infinite polynomial":

$$g(t) = \sum_{n=0}^{\infty} c_n t^n \tag{2}$$

For instance, for e^t,

[1] Reproduced here by permission of Prof. Peter Hamburger, Indiana-Purdue University, Fort Wayne. See http://www.ipfw.edu/math/Workshop/PBC.html

$$e^t = \sum_{n=0}^{\infty} \frac{1}{n!} t^n \tag{3}$$

In the case of a repeating function, it is more convenient to use another kind of series representation, an "infinite trig polynomial," called a **Fourier series**. This is just a fancy name for a weighted sum of sines and cosines of different frequencies. More precisely, we can write any repeating function g(t) with period T and fundamental frequency f_0 as

$$g(t) = \sum_{n=0}^{\infty} a_n \cos(2\pi n f_0 t) + \sum_{n=1}^{\infty} b_n \sin(2\pi n f_0 t) \tag{4}$$

for some set of weights a_n and b_n.[2] Here, instead of having a weighted sum of terms

$$1,\ t,\ t^2,\ t^3,\ ... \tag{5}$$

as in a Taylor series, we have a weighted sum of terms

$$1,\ \cos(2\pi f_0 t),\ \cos(4\pi f_0 t),\ \cos(6\pi f_0 t),\ ... \tag{6}$$

and of similar sine terms. Note that the frequences $2\pi n f_0$, in those sines and cosines are integer multiples of the fundamental frequency of x, f_0, called **harmonics**.

The weights a_n and b_n, n = 0, 1, 2, ... are called the **frequency spectrum** of g(). The coefficients are calculated as follows:[3]

$$a_0 = \frac{1}{T} \int_0^T g(t)\, dt \tag{7}$$

$$a_n = \frac{2}{T} \int_0^T g(t)\, cos(2\pi n f_0 t)\, dt \tag{8}$$

$$b_n = \frac{2}{T} \int_0^T g(t)\, sin(2\pi n f_0 t)\, dt \tag{9}$$

By analyzing these weights, we can do things like machine-based voice recognition (distinguishing one person's voice from another) and speech recognition (determining what a person is saying). If for example

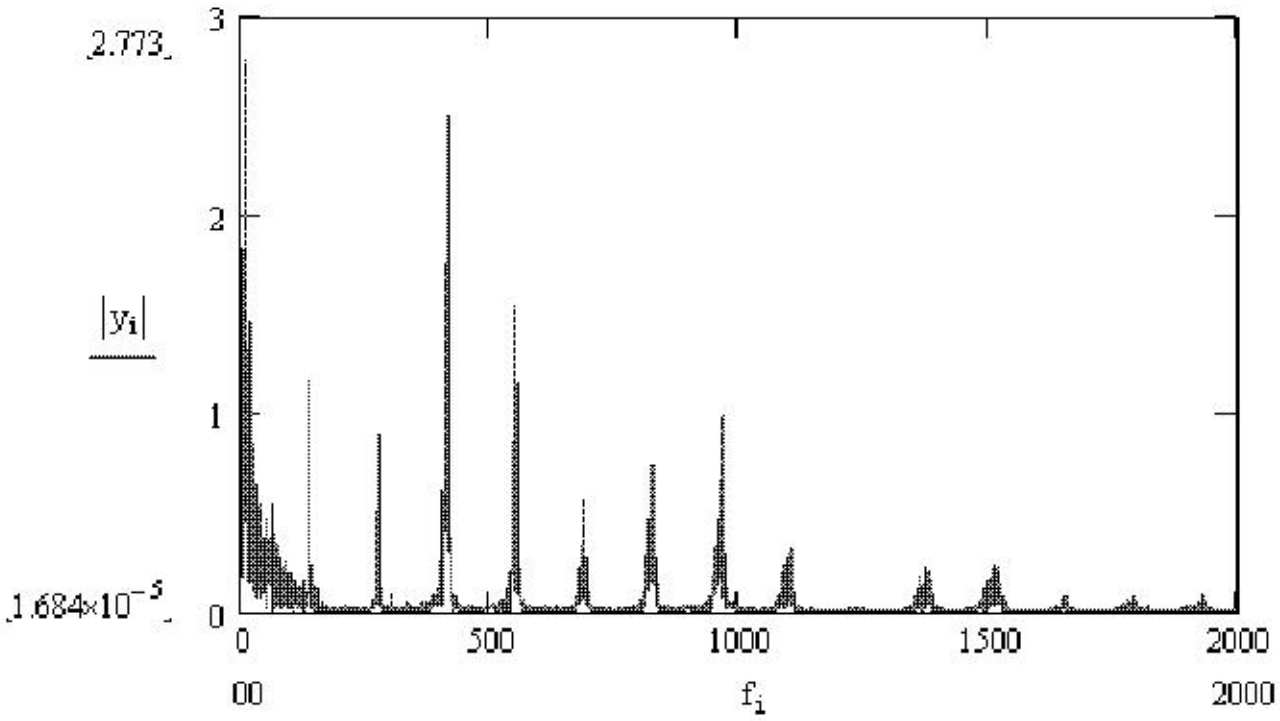
[2] The second summation starts at n = 0 too, but the sine term in that case is zero.

[3] We omit the derivation here. The motivation comes from vector space considerations, where all the possible periodic functions g() (with reasonable conditions placed regarding continuity etc.) forms the vector space, and the sine and cosine functions form an orthonormal basis. The a_n and b_n are then the "coordinates" of g() when the latter is viewed as an element of that space. But if you integrate both sides of 4, you will at least verify that the formulas below do work.

one person's voice is higher-pitched than that of another, the first person's larger weights will be concentrated more on the higher-frequency sines and cosines than will the weights of the second.

Since g(t) is a graph of loudness against time, this representation of the sound is called the **time domain**. When we find the Fourier series of the sound, the set of weights a_n and b_n is said to be a representation of the sound in the **frequency domain**. One can recover the original time-domain representation from that of the frequency domain, and vice versa, as seen in Equations (7), (8), (9) and (4).

Now here is the frequency-domain version of the reed sound:



Note that this graph is very "spiky." In other words, most of the power of the signal is at a few frequencies which arise from the physical properties of a reed.

Fourier series are often expressed in terms of complex numbers, making use of the relation

$$e^{i\theta} = \cos(\theta) + i\ \sin(\theta), \tag{10}$$

where $i = \sqrt{-1}$.

Then (4) becomes

$$g(t) = \sum_{j=-\infty}^{\infty} c_j e^{2\pi i j \frac{t}{T}}. \tag{11}$$

The c_j are now generally complex numbers. They are functions of the a_j and b_j, and thus comprise the frequency spectrum.

1.1.2 Two-Dimensional Fourier Series

It is the same principle with images. Here g() is a function of two variables, g(u,v), where u and v are the horizontal and vertical coordinates of a pixel in the image; g(u,v) is the intensity of the image at that pixel. If it is a gray-scale image, the intensity is brightness of the image at that pixel. If it is a color image, a typical graphics format is to store three intensity values at a point, one for each of red, green and blue. The various color come from combining three colors at various intensities.

Since images are two-dimensional instead of one-dimensional like a sound wave form, the Fourier series for an image is a sum of sines and cosines in two variables, i.e. a double sum $\Sigma_j \Sigma_k ...$ instead of $\Sigma_j ...$.

The terminology changes a bit. Our original data is now referred to as being in the **spatial domain**, rather than the time domain. But the Fourier series coefficients are still said to be in the frequency domain.

1.2 Discrete Fourier Transforms

In sound and image applications, we seldom if ever know the exact form of the repeating function g(). All we have is a **sampling** from g(), i.e. we only have values of g(t) for a set of discrete values of t.

In the sound example above, a typical sampling rate is 8000 samples per second.[4] So, we may have g(0), g(0.000125), g(0.000250), g(0.000375), and so on. In the image case, we sample the image pixel by pixel.

Thus we can't calculate integrals like (7). So, how do we approximate the Fourier transform based on the sample data?

1.2.1 One-Dimensional Data

Let $x_0, ..., x_{n-1}$ denote the sampled values, i.e. the time-domain representation of g() based on our sample data. These are interpreted as data from one period of g().[5] The frequency-domain representation will also consist of n numbers, $c_0, ..., c_{n-1}$, defined as follows:[6]

$$c_k = \frac{1}{n} \sum_{j=0}^{n-1} x_j q^{jk} \tag{12}$$

where

[4] See Appendix A for the reasons behind this.

[5] There may not be any physical interpretation to considering g() to be periodic.

[6] It should be noted that there are many variant definitions of these transforms. One common variation is to include/exclude a scale factor, such as our 1/n in (12) below. Another type of variations involve changing only c_0, in order to make certain matrices have more convenient forms.

$$q = e^{-2\pi i/n} \tag{13}$$

again with $i = \sqrt{-1}$.

Note that (12) is basically a discrete analog of (8) and (9).

As in the continuous case, we can recover each domain from the other. So, while (12) shows how to go to the frequency domain from the time domain, we can go from the frequency domain to the time domain via the inverse transform, whose equation is

$$x_k = \sum_{j=0}^{n-1} c_j q^{-jk} \tag{14}$$

Note that (14) is basically a discrete analog of (4).

Note too that instead of having infinitely many harmonics, we can only have n of them (the last being n-1). This is implied by (14), by reasoning as follows: The x_k are given, q is a constant, and we are solving for the c_j. So, we have n equations in n unknowns. If we had more than n unknowns, the system would be indeterminate.

1.2.2 Two-Dimensional Data

The spectrum numbers are double-subscripted, like the original data. Equation (12) becomes

$$c_{rs} = \frac{1}{n}\frac{1}{m}\sum_{j=0}^{n-1}\sum_{k=0}^{m-1} x_{jk} e^{-2\pi i(\frac{jr}{n}+\frac{ks}{m})} \tag{15}$$

Its inverse is

$$c_{rs} = \sum_{j=0}^{n-1}\sum_{k=0}^{m-1} c_{jk} e^{2\pi i(\frac{jr}{n}+\frac{ks}{m})} \tag{16}$$

1.3 Parallel Computation of Discrete Fourier Transforms

1.3.1 The Fast Fourier Transform

Speedy computation of a discrete Fourier transform (DFT) was developed by Cooley and Tukey in their famous Fast Fourier Transform (FFT), which takes a "divide and conquer" approach:

Equation (12) can be rewritten as

$$c_k = \frac{1}{n}\left[\sum_{j=0}^{\tilde{n}-1} x_{2j} q^{2jk} + \sum_{j=0}^{\tilde{n}-1} x_{2j+1} q^{(2j+1)k}.\right] \tag{17}$$

After some algebraic manipulation, this becomes

$$c_k = \frac{1}{2}\left[\frac{1}{m}\sum_{j=0}^{m-1} x_{2j} z^{jk} + q^k \frac{1}{m}\sum_{j=0}^{m-1} x_{2j+1} z^{jk}\right] \tag{18}$$

where $m = n/2$ and $z = e^{-2\pi i/m}$.

A look at Equation (18) shows that the two sums within the brackets have the same form as Equation (12). In other words, Equation (18) shows how we can compute an n-point FFT from two $\frac{n}{2}$-point FFTs. That means that a DFT can be computed recursively, cutting the sample size in half at each recursive step.

In a shared-memory setting such as OpenMP, we could implement this recursive algorithm in the manners of Quicksort in our unit on parallel sorting, at `http://heather.cs.ucdavis.edu/~matloff/158/PLN/Sorting.pdf`.

In a message-passing setting, one can use the butterfly algorithm, explained for implementation of barriers in our introductory unit on parallel processing, at `http://heather.cs.ucdavis.edu/~matloff/158/PLN/Sorting.pdf`. Some digital signal processing chips implement this in hardware, with a special interconnection network to implement this algorithm.

1.3.2 A Matrix Approach

The matrix form of (12) is

$$C = \frac{1}{n} AX \tag{19}$$

where A is n x n and X is n-1. Element (j,k) of A is w^{jk}, while element j of X is x_j. This formulation of the problem then naturally leads one to use parallel methods for matrix multiplication; see `http://heather.cs.ucdavis.edu/~matloff/158/PLN/Matrix.pdf`.

1.3.3 Parallelizing Computation of the Inverse Transform

The form of the DFT (12) and its inverse (14) are very similar. For example, the inverse transform is again of a matrix form as in (19); even the new matrix looks a lot like the old one.

Thus the methods mentioned above, e.g. FFT and the matrix approach, apply to calculation of the inverse transforms too.

1.3.4 Parallelizing Computation of the Two-Dimensional Transform

Regroup (15) as:

$$c_{rs} = \frac{1}{n}\sum_{j=0}^{n-1}\left(\frac{1}{m}\sum_{k=0}^{m-1} x_{jk}e^{-2\pi i(\frac{ks}{m})}\right)e^{-2\pi i(\frac{jr}{n})} \quad (20)$$

$$= \frac{1}{n}\sum_{j=0}^{n-1} y_r e^{-2\pi i(\frac{jr}{n})} \quad (21)$$

Note that y_r, i.e. the expression between the large parentheses, is the DFT of the jth row of our data. And hey, the last expression above is in the same form as (12)! Of course, this means we are taking the DFT of the spectral coefficients rather than observed data, but numbers are numbers.

In other words: To get the two-dimensional DFT of our data, we first get the one-dimensional DFTs of each row of the data, place these in rows, and then find the DFTs of each column.

This certainly opens possibilities for parallelization. Each thread (shared memory case) or node (message passing case) could handle groups of rows of the original data, and in the second stage each thread could handle columns.

Or, we could interchange rows and columns in this process, i.e. put the j sum inside and k sum outside in the above derivation.

2 Applications to Image Processing

In image processing, there are a number of different operations which we wish to perform. We will consider two of them here.

2.1 Smoothing

An image may be too "rough." There may be some pixels which are noise, accidental values that don't fit smoothly with the neighboring points in the image.

One way to smooth things out would be to replace each pixel intensity value[7] by the mean or median among the pixels neighbors. These could be the four immediate neighbors if just a little smoothing is needed, or we could go further out for a higher amount of smoothing. There are many variants of this.

But another way would be to apply a **low-pass filter** to the DFT of our image. This means that after we compute the DFT, we simply delete the higher harmonics, i.e. set c_{rs} to 0 for the larger values of r and s. We then take the inverse transform back to the spatial domain. Remember, the sine and cosine functions of

[7] Remember, there may be three intensity values per pixel, for red, green and blue.

higher harmonics are "wigglier," so you can see that all this will have the effect of removing some of the wiggliness in our image—exactly what we wanted.

And we can control the amount of smoothing by the number of harmonics we remove.

The term *low-pass filter* obviously alludes to the fact that the low frequencies "pass" through the filter but the high frequencies are blocked. Since we've removed the high-oscillatory components, the effect is a smoother image.[8]

2.2 Edge Detection

In computer vision applications, we need to have a machine-automated way to deduce which pixels in an image form an edge of an object.

Again, edge-detection can be done in primitive ways. Since an edge is a place in the image in which there is a sharp change in the intensities at the pixels, we can calculate slops of the intensities, in the horizontal and vertical directions. (This is really calculating the approximate values of the partial derivatives in those directions.)

But the Fourier approach would be to apply a high-pass filter. Since an edge is a set of pixels which are abruptly different from their neighbors, we want to keep the high-frequency components and block out the low ones.

Below we have "before and after" pictures, first of original data and then the picture after an edge-detection process has been applied.[9]

[8] Note that we may do more smoothing in some parts of the image than in others.

[9] These pictures are courtesy of Bill Green of the Robotics Laboratory at Drexel University. In this case he is using a Sobel process instead of Fourier analysis, but the result would have been similar for the latter. See his Web tutorial at `www.pages.drexel.edu/~weg22/edge.html`.

The second picture looks like a charcoal sketch! But it was derived mathematically from the original picture, using edge-detection methods.

Note that edge-detection methods also may be used to determine where sounds ("ah," "ee") begin and end in speech-processing applications.

2.3 The Cosine Transform

It's inconvenient, to say the least, to work with all those complex numbers. But an alternative exists in the form of the **cosine transform**, which is a linear combination of cosines in the one-dimensional case, and products of cosines in the two-dimensional case.

2.4 Does the Function g() Really Have to Be Repeating?

It is clear that in the case of a vibrating reed, our loudness function g(t) really is periodic. What about other cases?

A graph of your voice would look "locally periodic." One difference would be that the graph would exhibit more change through time as you make various sounds in speaking, compared to the one repeating sound for the reed. Even in this case, though, your voice *is* repeating within short time intervals, each interval corresponding to a different sound. If you say the word *eye*, for instance, you make an "ah" sound and then an "ee" sound. The graph of your voice would show one repeating pattern during the time you are saying "ah," and another repeating pattern during the time you are saying "ee." So, even for voices, we do have repeating patterns over short time intervals.

On the other hand, in the image case, the function may be nearly constant for long distances (horizontally or vertically), so a local periodicity argument doesn't seem to work there.

The fact is, though, that it really doesn't matter in the applications we are considering here. Even though mathematically our work here has tacitly assumed that our image is duplicated infinitely times (horizontally and vertically),[10] we don't care about this. We just want to get a measure of "wiggliness," and fitting linear combinations of trig functions does this for us.

[10] And in the case of the cosine transform, implicitly we are assuming that the image flips itself on every adjacent copy of the image, first right-side up, then upside-own, then right-side up again, etc.

A Bandwidth: How to Read the *San Francisco Chronicle* Business Section

The popular press, especially business or technical sections, often uses the term **bandwidth**. What does this mean?

Any transmission medium has a natural range [f_{min},f_{max}] of frequencies that it can handle well. For example, an ordinary voice-grade telephone line can do a good job of transmitting signals of frequencies in the range 0 Hz to 4000 Hz, where "Hz" means cycles per second. Signals of frequencies outside this range suffer fade in strength, i.e are **attenuated**, as they pass through the phone line.[11]

We call the frequency interval [0,4000] the **effective bandwidth** (or just the **bandwidth**) of the phone line.

In addition to the bandwidth of a **medium**, we also speak of the bandwidth of a **signal**. For instance, although your voice is a mixture of many different frequencies, represented in the Fourier series for your voice's waveform, the really low and really high frequency components, outside the range [340,3400], have very low power, i.e. their a_n and b_n coefficients are small. Most of the power of your voice signal is in that range of frequencies, which we would call the effective bandwidth of your voice waveform. This is also the reason why digitized speech is sampled at the rate of 8,000 samples per second. A famous theorem, due to Nyquist, shows that the sampling rate should be double the maximum frequency. Here the number 3,400 is "rounded up" to 4,000, and after doubling we get 8,000.

Obviously, in order for your voice to be heard well on the other end of your phone connection, the bandwidth of the phone line must be at least as broad as that of your voice signal, and that is the case.

However, the phone line's bandwidth is not much broader than that of your voice signal. So, some of the frequencies in your voice will fade out before they reach the other person, and thus some degree of distortion will occur. It is common, for example, for the letter 'f' spoken on one end to be mis-heard as 's'on the other end. This also explains why your voice sounds a little different on the phone than in person. Still, most frequencies are reproduced well and phone conversations work well.

We often use the term "bandwidth" to literally refer to width, i.e. the width of the interval [f_{min},f_{max}], $f_{max} - f_{min}$.

There is huge variation in bandwidth among transmission media. As we have seen, phone lines have bandwidth intervals covering values on the order of 10^3. For optical fibers, these numbers are more on the order of 10^{15}.

Bandwidth determines how fast we can set digital bits. Think of sending the sequence 10101010... If we graph this over time, we get a "squarewave" shape. Since it is repeating, it has a Fourier series. What happends if we double the bit rate? We get the same graph, only horizontally compressed by a factor of two. The effect of this on this graph's Fourier series is that, for example, our former a_3 will now be our new a_6, i.e. the $2\pi \cdot 3f_0$ frequency cosine wave component of the graph now has the double the old frequency, i.e. is now $2\pi \cdot 6f_0$. That in turn means that the effective bandwidth of our 10101010... signal has doubled too.

In other words: To send high bit rates, we need media with large bandwidths.

[11] And in fact will probably be deliberately filtered out.