**Sounds and Fourier Transform**

 *Part 3: The Fast Fourier Transform*

So far we have seen that we can write a periodic function as a Fourier series, which exposes the discrete frequencies it is composed of. We also saw that we can compute the Fourier transform of a (not necessarily period) function, which also represents the function in terms of its (continuous) frequencies.

However, both methods are relatively slow, since they involve symbolic integration, which is a comparatively slow process, and even if we replace symbolic (and exact) integration by numerical (and approximate) integration, it is still a relatively slow process. It turns out, though, that there is a super-fast procedure to compute the first N coefficients of a Fourier series approximately, and even better, this computation requires only M points on the original function; we say that we *sample* the function at M points. That process is called Fast Fourier Transform.

**Definition** (**Fast Fourier Transform**)

The *Fast Fourier Transform*, or ***FFT***, is a procedure that takes as input N points representing values of a (periodic) function *f* and produces as output an approximation of the first N/2 coefficients of the Fourier series for . This procedure can be programmed in most programming languages and it is also available as the Mathematica command **Fourier**.

Notes:

* The term FFT usually refers to a particular and very slick algorithm to transform the input values sampled from a function into the Fourier coefficients for that function.
* Many implementations of the FFT require that there are input values. Mathematica’s **Fourier** command does not require that but it *does* works better if you do use input values
* Many mathematicians describe the FFT as "the most important numerical algorithm of our lifetime" and it was included in Top 10 Algorithms of 20th Century by the IEEE journal Computing in Science & Engineering.

We include an appendix showing the source code for this algorithm, written in Java, but for now we want to focus on the *applications* and examples of the FFT:

**Example**: Use Mathematica to create a list of 8 values for , where . Then apply the FFT to compute an approximation of the first four coefficients of the Fourier series for *the function*

Mathematica’s FFT procedure is called **Fourier**. It takes as input an array of length and produces as output an array of complex numbers of length , of which the first terms store the Fourier coefficients. The real parts of these complex numbers are the coefficients of the Fourier series for , the imaginary parts are the coefficients . Here is the complete process:

1. Generate a list of equally spaced points in the interval :

 X = Table[2.0 Pi \* i / 8, {i, 0, 7}]

(2) Sample the function at these points to create an array of 8 values:

 f[x\_] = Sin[x] - 3 Cos[2x]

 Y = f[X]

(3) Apply the Fast Fourier Transform to compute the coefficients of the Fourier series for our function

 coeffs = Fourier[Y] / 2^(1/2)

 Note that the scaling factor for input values would be

(4) The output of the FFT needs to be parsed into the Fourier coefficients , and :

 a0 = Re[coeffs][[1]]/2

a = Re[coeffs][[2;;4]]

 b = Im[coeffs][[2;;4]]

The notation [[m;;n]] extracts the entries at positions n through m from a list

(5) Now we can define the (approximate) Fourier series for our function as follows:

 fft[x\_] = a0 + Total[a\*Table[Cos[i\*x],{i,1,3}]] + b\*Total[Table[Sin[i\*x],{i,1,3}]]

Here is the appropriate output using Mathematica:

which is approximately equal to:

since numbers of the form can be considered equal to zero. Here is the procedure once again, using Mathematica for everything.



So, we see that for our FFT procedure only 8 sample points were sufficient to restore the original function (almost) exactly. Note that in this example the function was of course its own Fourier series, so we already knew the answer. But this technique also works for more or less arbitrary functions:

**Example**: Sample the function at points in the interval to find its approximate Fourier series. Plot the function and its approximate Fourier series to see how good your approximation is. You can use the Mathematica notebook from <http://pirate.shu.edu/~wachsmut/Teaching/MATH4516/2017-01/fft.nb>:



If we plot the Fourier series against the original function, we see that the approximation is pretty good, using only the first 16 (approximate) terms of the Fourier series. For a better match, use more sample points by changing n to, say, 10, to compute the first terms of the Fourier series.

**Exercises**:

1. Compute the approximate Fourier series for for , using sample points. Then compute the exact Fourier series (using the integral definitions of , and adjusted for the interval instead of ). Plot all three functions in one coordinate system. What can you say?
2. How many sample points do I need to provide as input to the FFT to recover the function . What happens if you take fewer samples? More samples?
3. Compute the approximate Fourier series for the step function Note that when you define the function **f[x\_] = Piecewise[{{1,0<=x<2},{2,2<=x<4},{3,4<=x<6.28}}];** you need to also use the command **SetAttributes[f,Listable]** to make sure that the function can be applied over a list. If you don’t add that command right after defining the function, your FFT routine will produce a bunch of errors.
4. The procedures we developed in this segment works for functions defined on . Adjust the Mathematica code so that we can find approximate Fourier series for functions defined on .

**Appendix**: Source Code for FFT Algorithm

Below is the source code for a Java program that can find perform a FFT procedure as described above. You need to save the code in a file named “FFT.java”, then you can compile and execute it. It should not give you any error. If you are a computer science major, you should most definitely try to get this program to work. You could build many cools apps with this as its center code, such as a ‘tuning’ app, or ‘voice recognition’ app, or a ‘sound based password’ program or many more.

/\*\*

 \* Computes the Fourier coefficients of a function via FFT (Fast Fourier Transform).

 \*

 \* **@author** Bert G. Wachsmuth

 \* **@version** Sept. 2017

 \*/

**public** **class** FFT

{

 **private** **final** **static** **double** ***EPSI*** = 10E-5; // approximation for zero

 **private** **final** **static** **int** ***N*** = 32; // sample size

 **private** **final** **static** **double** ***A*** = 0.0; // left endpoint of interval

 **private** **final** **static** **double** ***B*** = 2\*Math.***PI***; // right endpoint of interval

 /\* The function to sample (and whose Fourier coefficients we want find)

 \*/

 **public** **static** **double** f(**double** x)

 {

 **return** Math.*sin*(x) + 2\*Math.*cos*(x) - 4.0\*Math.*cos*(2\*x);

 }

 /\* Generates the sample by evaluating f at N points between A and B.

 \*

 \* On output:

 \* ar: contains the sample

 \* ai: set to zero

 \*/

 **public** **static** **void** generateSample(**double** ar[], **double** ai[])

 {

 **double** delta = (***B*** - ***A***)/***N***;

 **for** (**int** i = 0; i < ***N***; i++)

 {

 **double** x = ***A*** + i\*delta;

 ar[i] = *f*(x);

 ai[i] = 0.0;

 }

 }

 /\* This represents the main FFT algorithm. It's pretty slick and optimized for speed.

 \* It is difficult to understand without help.

 \*

 \* On input:

 \* sign: 1 for forward FFT, or -1 for inverse FFT.

 \* n: the length of the sample (must be a power of 2)

 \* ar: array of length n, contains the function sampled at n points

 \* ai: empty (set to zeros)

 \*

 \* On output:

 \* ar: contains the Fourier coefficients a[n], n=0 to N/2

 \* ai: contains the Fourier coefficients b[n], n=1 to N/2

 \*/

 **public** **static** **void** computeFFT(**int** sign, **int** n, **double** ar[], **double** ai[])

 {

 **double** scale = 2.0 / (**double**) n;

 **int** i, j;

 **for** (i = j = 0; i < n; ++i)

 {

 **if** (j >= i)

 {

 **double** tempr = ar[j] \* scale;

 **double** tempi = ai[j] \* scale;

 ar[j] = ar[i] \* scale;

 ai[j] = ai[i] \* scale;

 ar[i] = tempr;

 ai[i] = tempi;

 }

 **int** m = n / 2;

 **while** ((m >= 1) && (j >= m))

 {

 j -= m;

 m /= 2;

 }

 j += m;

 }

 **int** mmax, istep;

 **for** (mmax = 1, istep = 2 \* mmax; mmax < n; mmax = istep, istep = 2 \* mmax)

 {

 **double** delta = sign \* Math.***PI*** / (**double**) mmax;

 **for** (**int** m = 0; m < mmax; ++m)

 {

 **double** w = m \* delta;

 **double** wr = Math.*cos*(w);

 **double** wi = Math.*sin*(w);

 **for** (i = m; i < n; i += istep)

 {

 j = i + mmax;

 **double** tr = wr \* ar[j] - wi \* ai[j];

 **double** ti = wr \* ai[j] + wi \* ar[j];

 ar[j] = ar[i] - tr;

 ai[j] = ai[i] - ti;

 ar[i] += tr;

 ai[i] += ti;

 }

 }

 mmax = istep;

 }

 }

 // The standard main function where execution starts

 **public** **static** **void** main(String args[])

 {

 **double** ar[] = **new** **double**[***N***];

 **double** ai[] = **new** **double**[***N***];

 *generateSample*(ar, ai);

 *computeFFT*(1, ar.length, ar, ai);

 *showCoefficients*(ar, ai);

 }

 // Convenience function to show the computed non-zero Fourier coefficients

 **public** **static** **void** showCoefficients(**double** ar[], **double** ai[])

 {

 System.***out***.println("a[0] = " + ar[0]/2.0);

 **for** (**int** i = 1; i < ***N***/2; i++)

 {

 **if** (Math.*abs*(ar[i]) >= ***EPSI***)

 System.***out***.println("a[" + i + "] = " + ar[i]);

 }

 **for** (**int** i = 1; i < ***N***/2; i++)

 {

 **if** (Math.*abs*(ai[i]) >= ***EPSI***)

 System.***out***.println("b[" + i + "] = " + ai[i]);

 }

 }

}