TWO N POINT REAL FFTS USING AN N POINT COMPLEX FFT

The following technique was briefly mentioned in a 1967 paper [BINGH67] by C. Bingham, M. D. Godfrey and J. W. Tukey (of Cooley-Tukey fame, the inventors of FFT). Part of the paper briefly describes how to transform two N point real valued sequences using a single N point complex input FFT. Although the technique has been around for quite some time, there are still many who don't really understand it, so let's spend a little time figuring it out, because it's a very useful thing to know.

Suppose we have two real valued N point time sequences xr[n] and yr[n] (n = 0 to N-1) with zero filled imaginary arrays xi[n] and yi[n], respectively. We could transform (xr[n], xi[n]) and (yr[n], yi[n]) separately using two N point FFTs for complex valued inputs. If we wanted to do things a bit more efficiently, we could transform them separately using two N point FFTs for real valued inputs. Note that, either way, we get 2N complex outputs - N complex points for the transform of x, and N complex points for the transform of y. But there's another way to do it. Suppose we use a single N point complex input FFT to transform xr[n] and yr[n] simultaneously. Let's do it that way by forming a new complex valued sequence: z[n] = xr[n] + jyr[n]. We do this by using the real sequence xr[n] as the real input of the complex FFT, and the real sequence yr[n] as the imaginary input. Note that we're not actually multiplying anything by j. It's just that when we put the real values of yr[n] into the imaginary locations of the complex input FFT, the computer will deal with them as jyr[n].

After the transform, we get Z[f] = N complex numbers. Since an FFT is a linear function, these complex numbers represent Z[f] = X[f] + jY[f], where X[f] and Y[f] are the <u>transforms</u> of x and y. But don't be misled here. Getting X[f] and Y[f] from the FFT's outputs isn't as simple as it seems, because <u>both X[f] and Y[f] are transforms</u>, <u>so they each have real and imaginary parts</u>, <u>and one of them is multiplied by j</u>, <u>and they're added together in the output</u>. <u>As a result</u>, <u>they're all jumbled together</u>.

Let's think about that for a moment. We use the real sequence xr[n] as the real inputs to the FFT, and the real sequence yr[n] as the imaginary inputs. We compute the FFT and get N complex results. They are represented by the xr[n] real and yr[n] imaginary outputs of the FFT. They are the sum of X[f] + jY[f]. What we really want to know is: what are the transforms X[f] and Y[f] of our original two real valued sequences? The problem is that X[f] has N real/imaginary parts, as does Y[f]. <u>So we need to obtain a total of 2N complex results</u> - N complex points for X[f] and N complex points for Y[f]. <u>But we've only got N complex outputs from the FFT</u> (xr[n] and yr[n]). They are the sum of X[f] + jY[f]. It would seem that we're stuck. Or perhaps not. It turns out that we can impose an additional constraint that will allow us to reduce the 2N complex unknowns versus the N complex knowns.

We make use of the following. Z[f] is represented by the N complex FFT results xr[n] and yr[n]. They in turn represent a <u>sum of the transforms of 2 real valued sequences</u>. Each of those 2 transforms have upper parts that are conjugates of their lower parts. So we can write:

Z[f] = X[f] + jY[f]	positive frequency outputs of the FFT (points 0 to N/2)
Z[-f] = X[-f] + jY[-f]	negative frequency outputs of the FFT (points N/2 + 1 to N - 1)

We <u>know</u> the values of Z[f] and Z[-f]. They are the outputs of our FFT. What we want to know is X[f], X[-f], Y[f] and Y[-f]. But X[-f] and Y[-f] are just conjugates of X[f] and Y[f], respectively (0 and N/2 points excepted - they don't have conjugates). So we have 2 equations, 2 knowns (Z[f] and Z[-f]) and 2 unknowns (X[f] and Y[f]), and we can solve for the unknowns in terms of the knowns:

real part of X[f] = real part of (Z[f] + Z[-f]) / 2
imaginary part of X[f] = imaginary part of (Z[f] - Z[-f]) / 2

real part of Y[f] = imaginary part of (Z[f] + Z[-f]) / 2
imaginary part of Y[f] = - real part of (Z[f] - Z[-f]) / 2

Now let's figure out the values of Z[f] and Z[-f] to use in the above. Z[f] are the complex valued positive frequencies of the FFT: real parts xr[n] and imaginary parts yr[n], n = 1 to N/2 - 1, plus the 0 and N/2 points, neither of which has a conjugate. Z[-f] are the complex valued negative frequencies (upper part) of the FFT: real parts xr[N - n] and imaginary parts yr[N - n], n = 1 to N/2 - 1. So the real and imaginary parts of Z[f] and Z[-f] are:

real part of Z[f] = xr[n]	imaginary part of Z[f] = yr[n]	positive frequencies
real part of Z[-f] = xr[N - n]	imaginary part of Z[-f] = yr[N - n]	negative frequencies

Now we can plug the above into the equations on the bottom of the previous page to get the real and imaginary parts of X[f] and Y[f]. Once we have them, we simply conjugate them to get X[-f] and Y[-f]. The code shown below is how we do it. The left side gets X[f] and Y[f], and the right side gets conjugates X[-f] and Y[-f].

```
fft_recur (N, xr, yr)    ;  // fft uses recursive twiddle and its outputs are scaled by N internally

for (n = 1; n < N/2; n++) {             // sequence is important below - do the imaginary parts first

        xi[n] =   ( yr[n]  - yr[N-n] ) / 2.   ;   xi[N-n] = - xi[n]  ;         // imaginary parts of X[f] and X[-f]
        yi[n] = - ( xr[n]  - xr[N-n] ) / 2.   ;   yi[N-n] = - yi[n]  ;         // imaginary parts of Y[f] and Y[-f]
        xr[n] =   ( xr[n] + xr[N-n] ) / 2.    ;   xr[N-n] =   xr[n]  ;         // real parts of X[f] and X[-f]
        yr[n] =   ( yr[n] + yr[N-n] ) / 2.    ;   yr[N-n] =   yr[n]  ;         // real parts of Y[f] and Y[-f]

} // end for
```

First, we use the real valued sequences xr and yr in the FFT calculation. After the FFT, the arrays xr and yr are our knowns. They are the real/imaginary parts of Z[f] (positive frequencies) and Z[-f] (negative frequencies). We compute everything from them. Second, when we compute X[f], X[-f], Y[f] and Y[-f] from the FFT's output, we calculate the imaginary parts xi and yi and their conjugates before the real parts. When we calculate the real parts, we will overwrite the old values, and we couldn't compute the imaginary parts correctly if we did them later. So the sequence of the above is very important. Third, the above routine presumes that N is even. If N were odd, we'd have to change the routine (N/2 above would be (N+1)/2). Fourth, if we only wanted the (double amplitude) one-sided graphs from 0 to N/2 (e.g.: p. 6-4), we'd delete the divides by 2 and the right side conjugates.

The above may be a new way of coding two N point real valued sequences with an N point complex FFT. I've always seen it done using temporary arrays, and they compute the real parts before the imaginary parts. Then they compute the 0 and N/2 points. Using the above, we don't have to. After the FFT, xr[0], xr[N/2], yr[0] and yr[N/2] will be correct. As long as the 0 and N/2 points of the imaginary arrays xi[ ] and yi[ ] have 0's in them before you start, then your results should be correct. It's done in the CD using some saw tooth test data so you can verify that it works right (chap9_w_testdata.cpp).

Obviously, doing things this way adds some overhead compared to doing each sequence separately using a transform that is specifically designed for real valued inputs. But bear in mind that this overhead is for doing 1 complex FFT, in contrast to having to do 2 FFTs for real valued inputs. So your 2 real valued input FFTs have to run in less time than the 1 complex FFT plus overhead. For the [BINGH67] technique, the overhead as a percent of total run time is roughly $1/[1+\log_2 N]$. So for N = 1024, $\log_2$ of 1024 is 10, and our overhead is 1 part in 11, or 9%, and it decreases for larger N. This should be considered an upper bound, since we're only doing adds, subtracts and divisions by 2 (and we could do the divides while scaling the FFT outputs). The actual overhead is probably lower. Another way to view it is to consider that it's equivalent to adding an extra column of add/subtract butterflies to an N point complex FFT.

The math behind the [BINGH67] technique above sometimes confuses people, but it's actually not that difficult if you spend a little time figuring it out. And it's really a rather elegant method.

There's a second technique in [BINGH67] that allows us to transform a 2N point real valued sequence using an N point complex FFT. Once again, it's been around for quite a while, but it's very often done wrong. We'll do it correctly on the next few pages with the help of a graphical interpretation and a negative exponent forward FFT.

2N POINT REAL FFT USING AN N POINT COMPLEX FFT

I once had a problem trying to use this algorithm in an unusual way and came to the realization that I didn't really understand it well enough. So I spent a little time figuring it out. It's really quite simple. First, let's look at the 16 point bit reversed in/sequential out FFT of DIT type T1 seen before in Fig. 4. Inputs are real.

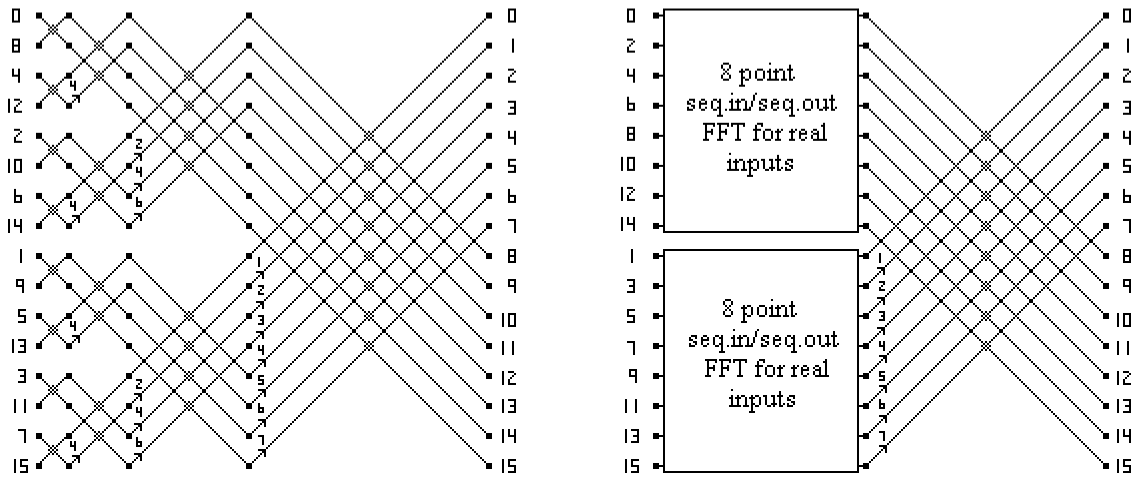
8 point seq.in/seq.out FFT for real inputs
8 point seq.in/seq.out FFT for real inputs

Fig. 6. (left) 16 point bit reversed in/sequential out FFT of DIT type T1. (right) Modified graph.

As noted before (p. 9-4), the upper 3 stages on the left side are an 8 point, bit reversed in/sequential out FFT of DIT type T1. So are the lower left 3 stages. The 16 inputs are in radix 2 bit reversed order. Note that the bit reversal also results in a splitting of the data into even and odd parts (even numbered inputs on the top, and odd numbered inputs on the bottom). Now suppose we put the upper (even) data back into sequence and the lower (odd) data back into sequence. This would seriously mess up the processing because we'd be putting sequential data into a bit reverse in/sequential out type of graph. So we'll have to change part of the graph. Let's replace the upper and lower left side 8 point bit reverse in/sequential out graphs with two 8 point sequential in/sequential out FFTs for real inputs. Then we'll combine the results using the last stage of twiddle/add/subtracts. The right side above shows the result. Note the reordering of the input data. The graph on the right is in fact equivalent to the one on the left, as long as your data is ordered correctly and you use the proper 8 point algorithm. To do the 8 point FFTs, perhaps we'd use a sequential in/bit reversed out type and bit reverse their outputs.

So if we start with 16 point sequential data, split it into sequential even and odd parts, then perform two 8 point sequential in/sequential out FFTs on real inputs, then combine them in a final stage, we'd get a 16 point FFT for real inputs. But, as was seen before, we can transform two N point real valued sequences using one N point complex input FFT.

So let's do it that way. Call the original 2N point real sequence r[n], where n = 0 to 2N-1. We should also have a 2N point i[n] to store our imaginary outputs, and we'll initialize it to zero. Now form two N point sequences Er[ ] and Or[ ] from r[n] as follows:

Er[ ] = N point sequence that contains the even numbered points in r[n],
i.e.: r[0], r[2], r[4], ….. r[2N-2]
Or[ ] = N point sequence that contains the odd numbered points in r[n],
i.e.: r[1], r[3], r[5], ….. r[2N-1]

Now we use Er[ ] as our N real inputs, and Or[ ] as our N imaginary inputs to the N point complex FFT. We'll end up with an N point complex output. This result contains the sum of two transforms, one for the even sequence, and one for the odd sequence. But, as before, their real and imaginary parts will be jumbled. <u>The transform of each sequence must be separated by using the same processing shown before</u>.

Once this is done, we'll have two N point complex valued transforms. One for the even sequence: Er[n] and Ei[n] real/imaginary parts, and one for the odd sequence: Or[n] and Oi[n] real/imaginary parts. Then we combine them in a final twiddle/add/subtract stage. Using 2N for the large graph and N for the FFT, the twiddles in the last stage are n*twopi/(2*N), which reduces to n*pi/N. After we do the twiddles on the outputs of the odd transform, we only have to add them to the outputs of the even transform. The result will give us the lower part (from n = 1 to N-1) of the transform of our 2N point original sequence. Since the original 2N point sequence was real, the upper part of its transform (from N+1 to 2N-1) are conjugates of the lower part. Then we'll compute the 0 and N outputs (for a 2N transform, neither has a conjugate).

The following code segment from the CD (chap9_2N_w_N.cpp) uses a 64 point, 18 component saw tooth as the input (not shown below) and a 32 point complex FFT):

```
for (n = 0; n < 2*N; n = n + 2) {                // split 2N input data into even and odd N point arrays
        Er[n/2] = r[n] ;    Or[n/2] = r[n+1] ;
}  // end for

fft_recur(N, Er, Or);        // fft uses recursive twiddle - outputs are scaled by 2*N

for (n = 1; n < N/2; n++) {      // sequence is important below - do the imaginary parts first
        Ei[n] =   ( Or[n]  - Or[N-n] ) / 2.     ;    Ei[N-n] =  - Ei[n]    ;   // separate E
        Oi[n] = - ( Er[n]  - Er[N-n] ) / 2.     ;    Oi[N-n] = - Oi[n]     ;   // and O
        Er[n] =   (  Er[n] + Er[N-n] ) / 2.     ;    Er[N-n] =    Er[n]    ;   // transforms
        Or[n] =   ( Or[n] + Or[N-n] ) / 2.      ;    Or[N-n] =   Or[n]     ;   //
} // end for

for (n = 1; n < N; n++) {
        C = cos(n*pi/N)  ;  S = sin(n*pi/N)  ;
        r[n] = Er[n] + Or[n]*C + Oi[n]*S   ;            // combine E and O transforms and get
        i[n] = Ei[n] +  Oi[n]*C  - Or[n]*S   ;          // conjugates - the +/- signs used are correct
        r[2*N-n] = r[n] ;    i[2*N-n] = -i[n] ;         // for a negative exponent FFT
} // end for
r[0] = Er[0] + Or[0] ;    r[N] = Er[0] - Or[0]  ;   // imaginary parts i[0] and i[N] were initialized to 0
```

We started with a 2N point real sequence, split it up into even and odd N point sequences, and then did an N point complex FFT. We get two transforms in the result, and we have to separate them as shown previously. Then we combine them in a final stage, and we get the lower part of the 2N transform. Once we have it, we simply conjugate the lower part to get the upper part. Then we compute the 0 and N points (for a 2N real transform, neither has a conjugate). Our final result is in r[2N] and i[2N].

It's very important to point out that the +/- signs in the combination part above are correct for a negative exponent FFT (C - jS twiddles). Today, almost everyone defines their forward FFT that way. But it didn't use to be that way. For instance, [BINGH67] used a positive exponent (C + jS twiddles) for the forward FFT. It's a source of much confusion, because it affects the signs used in the last stage.

And note that our overhead is worse than before, because in addition to separating E[n] and O[n], we have to perform complex multiplies. But once again, we're doing one complex N point FFT plus overhead, as opposed to doing a 2N point FFT for real valued inputs. And you could also use look-up tables or a recursive twiddle for C and S in the last stage to eliminate the math library calls. Or you could use the 3 add/3 multiply identity.

The beginning and end of this Chapter are not included in this tutorial, so you're missing quite a bit: the graph categorizations, coding and optimization techniques from Chapter 8, plus the code for the smaller sized real FFTs at the beginning of Chapter 9. But the two [BINGH67] techniques shown above are somewhat stand-alone. The remainder of Chapter 9 describes how to do the above without using temporary storage, and it does so with a new coding technique that significantly reduces the number of computations required.