

Enhanced Resolution, the “ins and outs” M.Miller 14-Nov-95

Enhanced Resolution is the name given by LeCroy to a set of processing functions in the 93xx series oscilloscopes which leverage oversampling to provide additional resolution.

In my experience with speaking of this subject with very (mathematically) educated individuals, it has often been the case that they do not believe it is possible to have more “information” about the digitized waveform than is contained in the original “unprocessed” waveform. There is no magic, they say ... and in fact they are right, there is no magic. In laymen’s terms the fact is, an oversampled waveform has more information than is ordinarily imagined. Oversampled means you measured the phenomenon more often than you needed. In non-laymen’s terms, oversampling means that you sampled *much* more often than you needed (the “needed” rate being defined as twice the nyquist frequency).

Now, if you have the results of more measurements than you really needed, maybe there’s some benefit to the extra work that’s been done ... seems reasonable, right?

I’d like to start with an example of something which is basically a static measurement, and therefore “enough” or often enough could be imagined to be *once*. That is, imagine you had to measure some DC voltage. It’s not changing, so you don’t *need* to measure it more than one time (it’s not going to change, right?). However, if you’ve ever done a real measurement, you might have noticed that you don’t always get the same result (except if you were counting sheep ... and even then...) even with a five and a half digit DVM (that’s about 20 bits) . So what do you do, if you sitting there in front of the 20-bit digitizing system? You maybe take the average of a few of the reading the DVM gives, and you write that down in the notebook.

Now, there are two things to say : 1) if the readings aren’t changing, there’s absolutely no advantage to using any kind of average (because the answer doesn’t change) and 2) usually the variations are bigger than the *supposed* resolution of the instrument (i.e. point 1 is usually not an issue). So in general, or more often than not, there *is* something to be gained by averaging the results of several measurements, instead of only measuring once.

Thus goes the reasoning to justify oversampling and enhanced resolution procedures (“filtering”) on waveforms. The filtering used to implement the enhanced resolution algorithm, is closely related to a local weighted average of sampled data points.

To be more precise, the filters used to perform the calculation are called Finite Impulse Response (FIR) filters. The calculation produces as new series of measurements, e_i , from an original series of measurements, v_i , as follows:

$$e_i = \sum_{k=-N}^N v_i c_{k-i}$$

where the c_k are the coefficients composing the FIR filter (for $k = -N$ to $k = N$). As a qualitative remark, usually the coefficients (for lowpass FIRs, which are noise reducing) are shaped like a bell-curve, symmetric, with maximum values around $N=0$). Please note that there are generally less points in the resulting series of points than in the original (since there are N points “before”, and N points “after” required for the calculation of any “new” point).

So the method behind Enhanced Resolution is to calculate a kind of average of the samples around the region of each final data point, reducing the fluctuations in the individual measurements.

* * * * *

Now, in the rigorous treatment, there’s a lot more to say about the effects and significance of the shape (values) of the FIR filters, and about why the things I’ll state next are “true”. However, it’s not my intention to give mathematical proofs in this document. But rather to give a few tangible rules-of-thumb for how FIRs behave.

For a white noise spectrum (commonly the case for a raw digitized waveform), the noise gain of an FIR filter is given by:

$$noisegain = \sum_{k=-N}^N c_k^2$$

As a practical matter, I will caution you; applying a low-pass FIR twice, doesn’t yield the noise reduction each time. The spectrum of noise has already been cut-off the first time! You will only succeed in further reducing the bandwidth of the resulting signal for little gain in resolution.

As a general rule, the complex frequency response of an FIR filter is given by:

$$H(\omega) = \sum_{k=-N}^N c_k e^{i\omega k}$$

Note this expression yields a continuous function of frequency ($f = 2\pi\omega$) for “any” FIR filter

For symmetric filters ($c_k = c_{-k}$), this results in an easy to calculate series of cosines. Such is the case for each of the FIRs used by LeCroy for the Enhanced resolution feature in LeCroy oscilloscopes. And from this calculation are derived the

estimated -3dB bandwidth figures quoted for each filter. Likewise from the above expression for noise gain, each enhanced resolution filter has a reduction of 6dB (a factor of 2) per bit.

* * * * *

There are just a couple of more things I'd like to say. The filters LeCroy use for enhanced resolution were calculated in a special way. The following criteria were imposed: 1) the impulse response (shape of the filter) was limited to "well behaved step response" forms, and 2) the filters are decomposable into concatenated "rectangular" implementations. The first criterion is in the interest of not producing "sharp cut-off" FIRs which are contrary to the interest if a fundamentally time-domain instrument (e.g. an oscilloscope). The second criterion is aimed at optimal implementation efficiency. To this purpose the program mf.exe was created to estimate the noise gain and coefficients of multiple rectangular FIRs. The source code to this program is provided here (compilable as a console application under WindowsNT).

* * * * *

Note: for this source code, the filter coefficients for 0.5 to 3-bit enhancement are produced from command line invocations:

```
mf -b2 -c -nplus0p5
mf -b2,2,3 -c -nplus_1p0
mf -b3,5,5 -c -nplus_1p5
mf -b7,9,11 -c -nplus_2p0
mf -b15,18,21 -c -nplus_2p5
mf -b28,30,31,32 -c -nplus_3p0
```

The resulting filters consist of concatenations of (normalized) rectangular filters of the following widths:

- +0.5 bits: 1 filter width 2-samples
- +1.0 bits: 3 filters width 2-samples, 2 samples and 3 samples
- +1.5 bits: 3 filters width 3-samples, 5 samples and 5 samples
- +2.0 bits: 3 filters width 7-samples, 9 samples and 11 samples
- +2.5 bits: 3 filters width 15-samples, 18 samples and 21 samples
- +3.0 bits: 4 filters width 28-samples, 30 samples, 31 samples and 32 samples

producing the following output:

```
C:\m2\bits\windebug>mf -b2 -c -nplus0p5
length=2, gain = 1.000000, noise gain = 0.707107, (0.500 bit-enhanced)

float plus0p5[2] =
{
    0.500000f, 0.500000f
};
```

```
C:\m2\bits\windebug>mf -b2,2,3 -c -nplus_1p0
length=5, gain = 0.999939, noise gain = 0.499980, (1.000 bit-enhanced)
```

```
float plus_1p0[5] =
{
    0.083313f, 0.250000f, 0.333313f, 0.250000f,
    0.083313f
};
```

```
C:\m2\bits\windebug>mf -b3,5,5 -c -nplus_1p5
length=11, gain = 0.999878, noise gain = 0.355504, (1.492 bit-enhanced)
```

```
float plus_1p5[11] =
{
    0.013306f, 0.039978f, 0.080017f, 0.119995f,
    0.159973f, 0.173340f, 0.159973f, 0.119995f,
    0.080017f, 0.039978f, 0.013306f
};
```

```
C:\m2\bits\windebug>mf -b7,9,11 -c -nplus_2p0
length=25, gain = 1.000000, noise gain = 0.245381, (2.027 bit-enhanced)
```

```
float plus_2p0[25] =
{
    0.001465f, 0.004333f, 0.008667f, 0.014404f,
    0.021667f, 0.030273f, 0.040405f, 0.050476f,
    0.060608f, 0.069275f, 0.076477f, 0.080811f,
    0.082275f, 0.080811f, 0.076477f, 0.069275f,
    0.060608f, 0.050476f, 0.040405f, 0.030273f,
    0.021667f, 0.014404f, 0.008667f, 0.004333f,
    0.001465f
};
```

```
C:\m2\bits\windebug>mf -b15,18,21 -c -nplus_2p5
length=52, gain = 0.999756, noise gain = 0.173863, (2.524 bit-enhanced)
```

```
float plus_2p5[52] =
{
    0.000183f, 0.000549f, 0.001038f, 0.001770f,
    0.002625f, 0.003723f, 0.004944f, 0.006348f,
    0.007935f, 0.009705f, 0.011658f, 0.013733f,
    0.016052f, 0.018494f, 0.021179f, 0.023804f,
    0.026428f, 0.029114f, 0.031555f, 0.033875f,
    0.035950f, 0.037720f, 0.039124f, 0.040222f,
    0.040894f, 0.041260f, 0.041260f, 0.040894f,
    0.040222f, 0.039124f, 0.037720f, 0.035950f,
    0.033875f, 0.031555f, 0.029114f, 0.026428f,
    0.023804f, 0.021179f, 0.018494f, 0.016052f,
    0.013733f, 0.011658f, 0.009705f, 0.007935f,
    0.006348f, 0.004944f, 0.003723f, 0.002625f,
    0.001770f, 0.001038f, 0.000549f, 0.000183f
};
```

```
C:\m2\bits\windebug>mf -b28,30,31,32 -c -f -nplus_3p0
length=118, gain = 1.000000, noise gain = 0.125828, (2.990 bit-enhanced)
```

```
float plus_3p0[118] =
{
    0.000001f, 0.000005f, 0.000012f, 0.000024f,
```

```

0.000042f, 0.000067f, 0.000101f, 0.000144f,
0.000198f, 0.000264f, 0.000343f, 0.000437f,
0.000546f, 0.000672f, 0.000816f, 0.000979f,
0.001163f, 0.001368f, 0.001596f, 0.001848f,
0.002125f, 0.002429f, 0.002760f, 0.003120f,
0.003510f, 0.003931f, 0.004385f, 0.004872f,
0.005393f, 0.005948f, 0.006534f, 0.007151f,
0.007794f, 0.008461f, 0.009146f, 0.009847f,
0.010559f, 0.011281f, 0.012007f, 0.012734f,
0.013459f, 0.014178f, 0.014887f, 0.015583f,
0.016262f, 0.016921f, 0.017556f, 0.018163f,
0.018739f, 0.019280f, 0.019783f, 0.020244f,
0.020659f, 0.021025f, 0.021339f, 0.021595f,
0.021792f, 0.021925f, 0.021993f, 0.021993f,
0.021925f, 0.021792f, 0.021595f, 0.021339f,
0.021025f, 0.020659f, 0.020244f, 0.019783f,
0.019280f, 0.018739f, 0.018163f, 0.017556f,
0.016921f, 0.016262f, 0.015583f, 0.014887f,
0.014178f, 0.013459f, 0.012734f, 0.012007f,
0.011281f, 0.010559f, 0.009847f, 0.009146f,
0.008461f, 0.007794f, 0.007151f, 0.006534f,
0.005948f, 0.005393f, 0.004872f, 0.004385f,
0.003931f, 0.003510f, 0.003120f, 0.002760f,
0.002429f, 0.002125f, 0.001848f, 0.001596f,
0.001368f, 0.001163f, 0.000979f, 0.000816f,
0.000672f, 0.000546f, 0.000437f, 0.000343f,
0.000264f, 0.000198f, 0.000144f, 0.000101f,
0.000067f, 0.000042f, 0.000024f, 0.000012f,
0.000005f, 0.000001f
};

/* /SOURCE CODE
*-----*
----- (%M%) -----
M.Miller    LeCroy Corporation
            700 Chestnut Ridge Rd.
            Chestnut Ridge, New York 10977

Modified by M.Miller, LeCroy S.A. from original (1987) program in
November 1995 for ANSI-C compatible with Windows NT console
application (MSVC 2.2)
*-----*

Calculate filters for Enhanced Resolution ... for arbitrary
digitizers ... for up to eight (8) concatenated rectangular filters.

This file has been formatted for 4-character tabstops
*-----*

/CODE */

#include <stdio.h>
#include <string.h>
#include <math.h>
#include <ctype.h>

// prototypes for local functions
static void conv(float* dataP, float* coeffP, int ndata, int ncoeff);
static int roundD(double value);

```

```

static void fnorm(float* array, int num);
static void nfill4(float* array, int num);

// NUM is maximum observable size of the impulse response (which
// is the shape and values of the equivalent filter)
#define NUM 1000
float array[NUM];

// NBASE is the maximum size of an intermediate rectangular
// (all coefficients equal) FIR
#define NBASE 50
float base[NBASE];

#define NRECT 8

// default set of rectangular filter sizes
int nbase[8] = {2,0,0,0,0,0,0,0};

// default filter name for 'C' output
char name[30] = "fil";

typedef enum      // output format options
{
    Analysis,
    FloatList,
    FloatListInC,
    FixedPointC,
    RemezExchange,
} SHOWTYPE;

void main( int argc, char** argv)
{
    char *s;
    float c2, amp, value;
    float bits;
    int i, j, k, m;
    int truncate = 1;
    SHOWTYPE ptype = Analysis;

    // scan arguments for command line options
    while (--argc > 0)
    {
        s = *(++argv);
        if ('-' == *s++)
        {
            switch(tolower(*s++))
            {
                case 'b':    // up to 8 values, pre-initialized above
                    sscanf(s, "%d,%d,%d,%d,%d,%d,%d,%d", &nbase[0], &nbase[1],
                        &nbase[2], &nbase[3], &nbase[4], &nbase[5],
                        &nbase[6], &nbase[7]);
                    for (j = 0; j < NRECT; ++j)
                        if (nbase[j] > NBASE)
                            nbase[j] = NBASE;
                    break;
                case 'n':    // filter name for 'C' code output
                    strncpy(name, s, 30);
                    break;
                case 'e':    // simple view of coefficients as list
                    ptype = FloatList;
                    break;
                case 'c':    // simple view of coefficients as list
                    ptype = FloatListInC;
                    break;
                case 'r':    // REMEZ exchange format
                    ptype = RemezExchange;
            }
        }
    }
}

```

```

        break;
    case 'a': // 14-bit fixed-point coeffs
        ptype = FixedPointC;
        break;
    case 'f': // suppress truncation to 14-bit precision
        truncate = 0;
        break;
    }
}

// create an impulse to be convolved with the rectangular FIRs
for ( i = 0; i < NUM; ++i)
    array[i] = 0.0f;
array[NUM/2] = 1.0f;

// there are up to 8 rectangular filters to be applied, which represent
// a single equivalent FIR filter
for ( j = 0; j < NRECT; ++j)
{
    /* terminate if there is no "base" */
    if (nbase[j] == 0)
        break;

    /* normalize the base filter */
    value = 1.0f / (float) nbase[j];
    for ( i = 0; i < nbase[j]; ++i)
        base[i] = value;
    for ( i = nbase[j]; i < NBASE; ++i)
        base[i] = 0.0f;

    /* convolve with the base filter */
    conv(array, base, NUM, nbase[j]);
}

// truncate precision of coefficients to 14-bits
if (truncate)
    nfill14(array, NUM);

/* calculate amplitude (normalization) and noise gain */
for ( i = 0, j = 0, amp = c2 = 0.0f; i < NUM; ++i)
{
    if (array[i] != 0.0f)
        ++j;
    amp += array[i];
    c2 += array[i] * array[i]; // square of coefficient
}
c2 = (float) sqrt(c2);
bits = (float) (-log10(c2) / log10(2.0));

/* print results */
switch(ptype)
{
    default:
    case Analysis: // default behavior
        printf("length=%d, gain = %f, noise gain = %f,", j, amp, c2);
        printf(" (%5.3f bit-enhanced)\n", bits);

        break;
    case FloatList: // list of coefficients (suitable for Excel)
        for ( i = 0; i < NUM; ++i)
        {
            if (array[i] != 0.0)
                printf("%f\n", array[i]);
        }
        break;
    case FloatListInC: // list of coefficients (suitable for)

```

```

printf("length=%d, gain = %f, noise gain = %f,", j, amp, c2);
printf(" (%5.3f bit-enhanced)\n", bits);
// count the non-zero values
for (i = j = 0; i < NUM; ++i)
{
    if (array[i] != 0)
        ++j;
}
printf("\nfloat %s[%d] =\n{\n\t", name, j);

for (i = 0, k = 0; i < NUM; ++i)
{
    if (array[i] != 0)
    {
        printf("%ff", array[i]); // trailing f denotes
                                // single-precision
        if (++k < j)
        {
            printf(", ");
            if ((k % 4) == 0) // limit line length
                printf("\n\t");
        }
    }
}
printf("\n};\n");
break;

case RemezExchange: // for the REMEZ exchange FORTRAN program
for (i = j = 0; i < NUM; ++i)
{
    if (array[i] != 0.0)
        ++j;
}
for (i = m = 0; i < NUM; ++i)
{
    if (array[i] != 0)
    {
        ++m;
        printf("h(%2d) = %e = h(%2d)\n", m, array[i], 1+j-m);
        if (m >= j-m)
            break;
    }
}
break;

case FixedPointC: /* 14-bit word format ... output in 'C' code */

// count the non-zero values
for (i = j = 0; i < NUM; ++i)
{
    m = (int)roundD((double)(16384.0 * array[i]));
    if (m != 0)
        ++j;
}
printf("\nshort %s[] = \n{\n    %d, /* size */\n    ",
name, j);

for (i = 0, k = 0; i < NUM; ++i)
{
    m = roundD((double)(16384.0 * array[i]));
    if (m != 0)
    {
        printf("%d", m);
        if (++k < j)
        {
            printf(", ");
            if ((k % 4) == 0) // limit line length

```

```

        printf("\n  ");
    }
}
printf("\n};\n");
}

// Perform the procedure called "convolution", which is to
// "apply the FIR filter"
void conv(float* dataP, float* coeffP, int ndata, int ncoeff)
{
    int i, j;
    float value;

    // there are (1 + ndata - ncoeff) valid output points
    for ( i = ncoeff; i < ndata; ++i)
    {
        value = 0.0f;
        for (j = 0; j < ncoeff; ++j)
            value += dataP[i-j] * coeffP[j];

        // assign the result to the original data array
        dataP[i-ncoeff] = value;
    }
}

// normalize the coefficients of an FIR filter, such that the
// sum of the coefficients as near as possible, 1.000...
static void fnorm(float* array, int num)
{
    if (num > 0)    // can't renormalize if there aren't any values
    {
        int i;
        double sum = 0.0;

        for (i = 0; i < num; ++i)
            sum += (double)array[i];

        if (sum > 0.0)
            for (i = 0; i < num; ++i)
                array[i] /= (float)sum;
    }
}

// modifies an FIR to have the storage quantization effects
// equivalent to 14-bit fixed-point coefficients
static void nfill14(float* array, int num)
{
    int i, ival;
    double fac2to14 = 16384.0;    // "weight" of 1.0 for 14-bit fixed
    double val;

    for (i = 0; i < num; ++i)
    {
        // scale the value upward as though it were fixed-point
        val = fac2to14 * (double) array[i];

        // round the result
        ival = roundD(val);

        // reinstall the de-scaled coefficient
        val = (double)ival;
        val /= fac2to14;
        array[i] = (float)val;
    }
}

```

```
// round a floating point (double precision) value
int roundD(double value)
{
    if (value > 0.0)      // the usual stuff, so truncation does a "round"
        value += 0.5;
    else
        value -= 0.5;
    return (int) value;   // cast to integer
}
/*----- end of file -----*/
```