UNITED STATES PATENT APPLICATION

For

FREQUENCY DOMAIN EQUALIZATION USING OVERLAPPED FOURIER TRANSFORMS

INVENTOR:

THOMAS HOLTZMAN WILLIAMS

Frequency Domain Equalization Using Overlapped Fourier Transforms

This disclosure should be a parent disclosure to D4430 and D4413, both of which use this novel type of equalizer. It is also related to CP elimination I4058, which de-ghosts a signal as a step towards demodulation.

Abstract

A linearly distorted signal can be equalized in the frequency domain using a single complex multiply for each component subcarrier frequency. This method is now used for deghosting received multicarrier signals, such as OFDM (orthogonal frequency division multiplex) and OFDMA (orthogonal frequency division multiple access) with cyclic prefixes (CP), and on single carrier signals such as SC-FDMA (single carrier frequency division multiple access), also employing a cyclic prefix. However, by using an overlapped Fourier transform, the requirement for an embedded cyclic prefix can be eliminated, and a pseudo-prefix can be created in an overlapped portion of a Fourier transform. The overlapped time portion is later discarded. This improved equalization method can be used for any type of modulated or baseband signal. The advantages of FD (frequency domain) equalization are lower transmission time overhead from the cyclic prefix removal, and improved computational efficiency in the frequency domain. The computational efficiency is improved relative to a FIR (finite impulse response) filter when the signal's bandwidth is wide and the echo is relatively long, because FIR filter structures can become very long with many taps. If employed for a burst-type signal, quiet time can be used for the pseudo-CP.

Demonstration/Simulation

A simulation was made using 4096 TD symbols with no CP. The intention was to extract 2048 clean TD symbols from the middle of the transform, and discard the rest as CP. The C++ code used for simulation is in the appendix. An echo-contaminated signal is equalized by transforming it into the FD, transforming the echo's impulse response into the FD, and performing a complex division of the data symbols by the channel response at each frequency. The quotients are converted back into the TD and the overlapped portions are removed, giving 2048 clean TD symbols.

Figs 1-3 are a heuristic explanation of the FD equalization process. Figs 4-7 are simulations. Fig. 8 is a timing diagram.

In Fig. 1 4096 linearly distorted QPSK symbols are illustrated as a time-frequency block, with time on the vertical axis, and frequency on the horizontal axes. An echo is a linear distortion which spreads its energy in time, but not in frequency.

Fig. 2 is a rotation of the block by 90 degrees after using a FFT. Equalization takes place in the FD using a complex multiply for each component frequency, or subcarrier. Mathematically, a FD complex equalization process is equivalent to a circular convolution with an equalizing impulse response. However, with no cyclic prefix, non-cyclic signal energy is employed distortion is created near the temporal block edges. But the duration of this distortion is limited by the duration of the echoes. Thus, if the length of FFT's transform is longer than the echoes, (1) there will be a period of time the signal is both free from echoes and distortion caused by the transform operating on a linear block, not a circular block.

In Fig. 3 the equalized block is IFFT'ed, and the equalized 2048 symbols are outputted. The CP containing 1024 preceding symbols and 1024 post symbols are discarded.

Fig. 4 is a simulation spectral plot of 4096 complex time domain (TD) symbols which take on values of +/-1 and are contaminated with a real echo with an amplitude of 0.5 and a delay of 100 symbols.

Fig. 5 is a FD channel response of the echo with a delay of 100 symbols and an amplitude of 0.1.

Fig. 6 is a full TD plot of equalized data.

Fig. 7 is TD plot of the first 20% of the TD data illustrated in Fig. 3, showing the first 500 symbols, which should be discarded.

Fig. 8 is a timing diagram illustrating how processing is done on each of two chains, one for deghosting even numbered blocks (B, D, F) and one for deghosting odd numbered blocks (A, C, E, G). Block 2 is comprised of all of block B, part of block A and part of block C. At T1 the start of capture of Block 2 (B) begins and at T2 the end of Block 2 capture occurs. Between T2 and T3 the processing (FFT, FD equalization, IFFT) of Block 2 occurs. Between T3 and T4 equalized symbols in block 2 is clocked out. This timing is delayed and used on Blocks 3, 4, and 5. Observe that if Processing Time is less than Output Time, only one FFT/IFFT engine is needed for both even and odd numbered blocks. Clocking should be continuous for seamless weaving of odd and even blocks. This processing may be continuous for a continuous reception, or batch processing for burst data. Note that when processing burst data, a previous transmission and its longest echo need to die out before starting a capture. Thus dead air time can be used for a pseudo-CP.

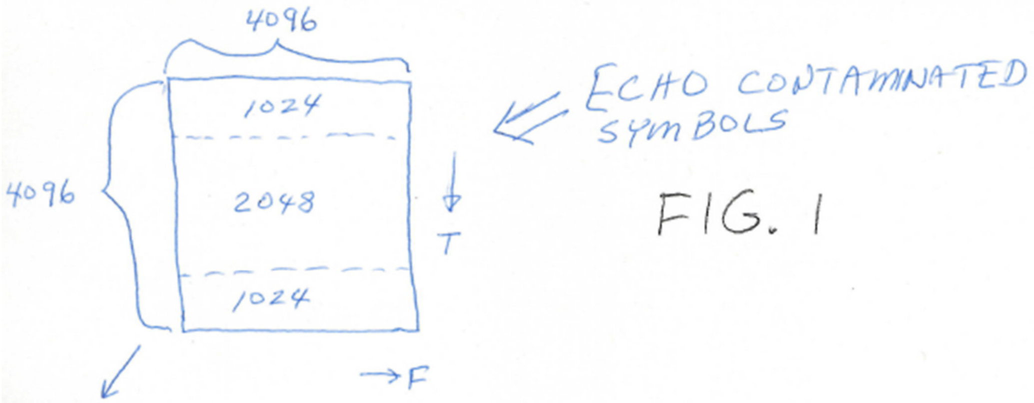

FIG. 1

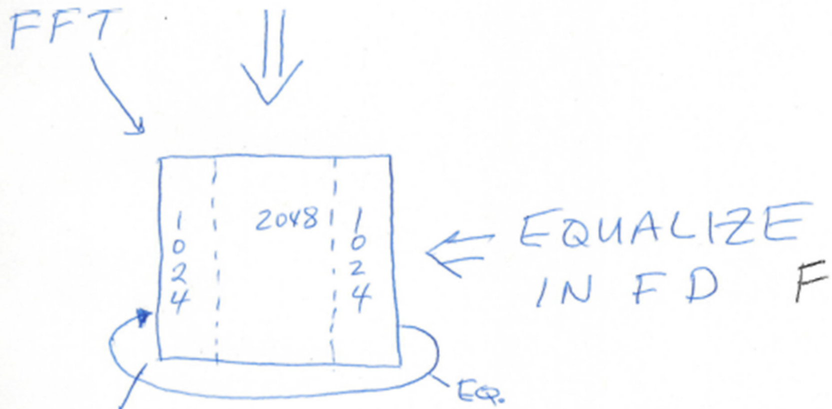

FIG. 2

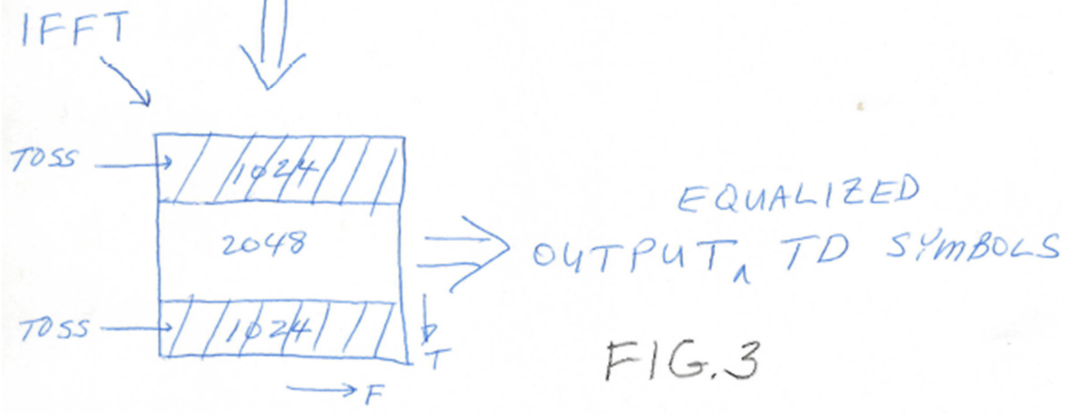

FIG. 3

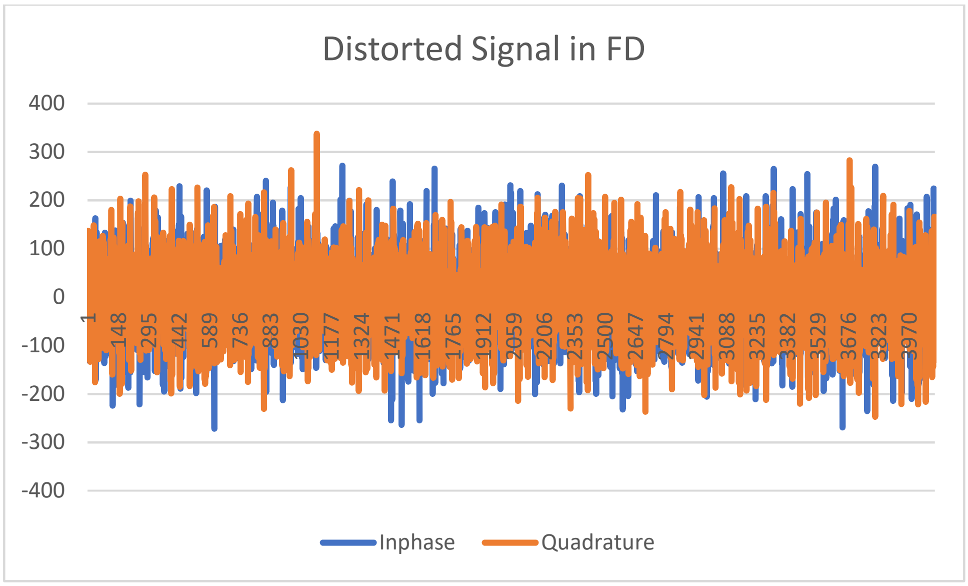
Distorted Signal in FD
400
300
200
100
0
-100
-200
-300
-400
1
148
295
442
589
736
883
1030
1177
1324
1471
1618
1765
1912
2059
2206
2353
2500
2647
2794
2941
3088
3235
3382
3529
3676
3823
3970
Inphase
Quadrature

Fig. 4

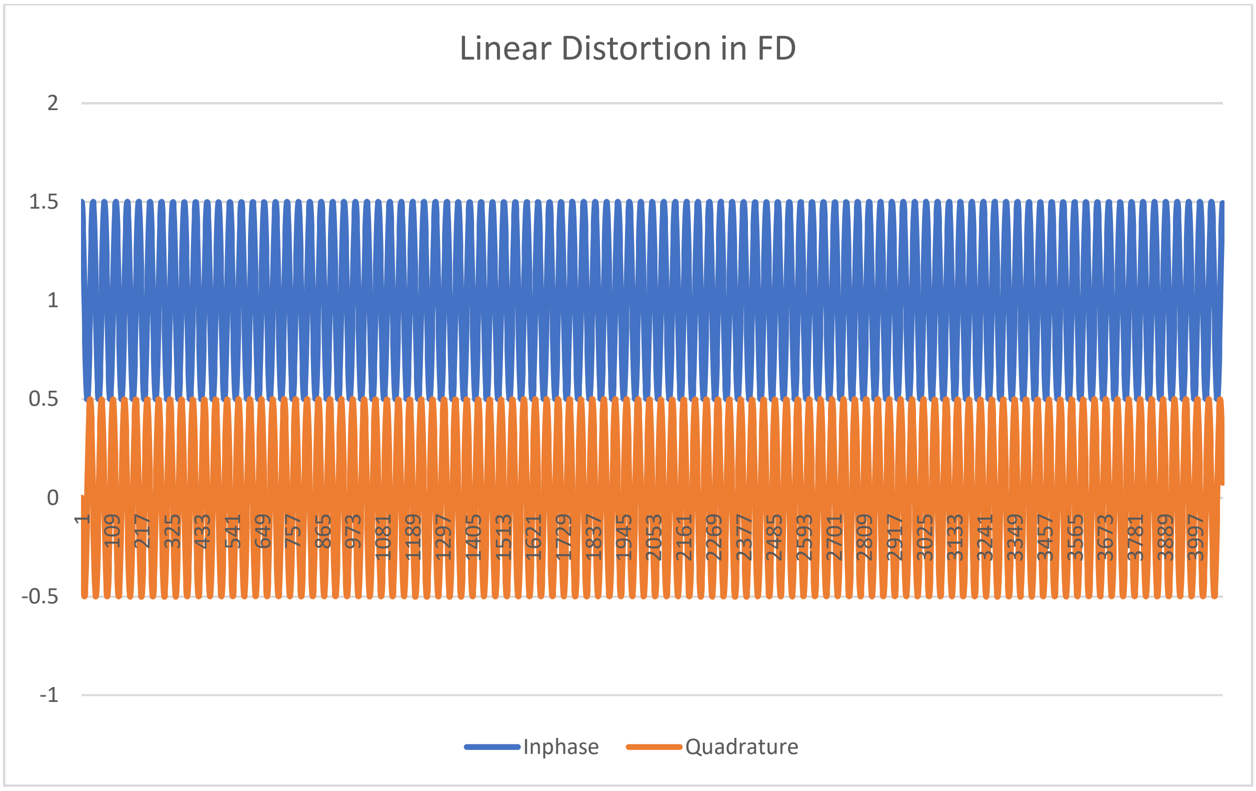
Linear Distortion in FD
2
1.5
1
0.5
0
-0.5
-1
1
109
217
325
433
541
649
757
865
973
1081
1189
1297
1405
1513
1621
1729
1837
1945
2053
2161
2269
2377
2485
2593
2701
2809
2917
3025
3133
3241
3349
3457
3565
3673
3781
3889
3997
Inphase
Quadrature

Fig. 5

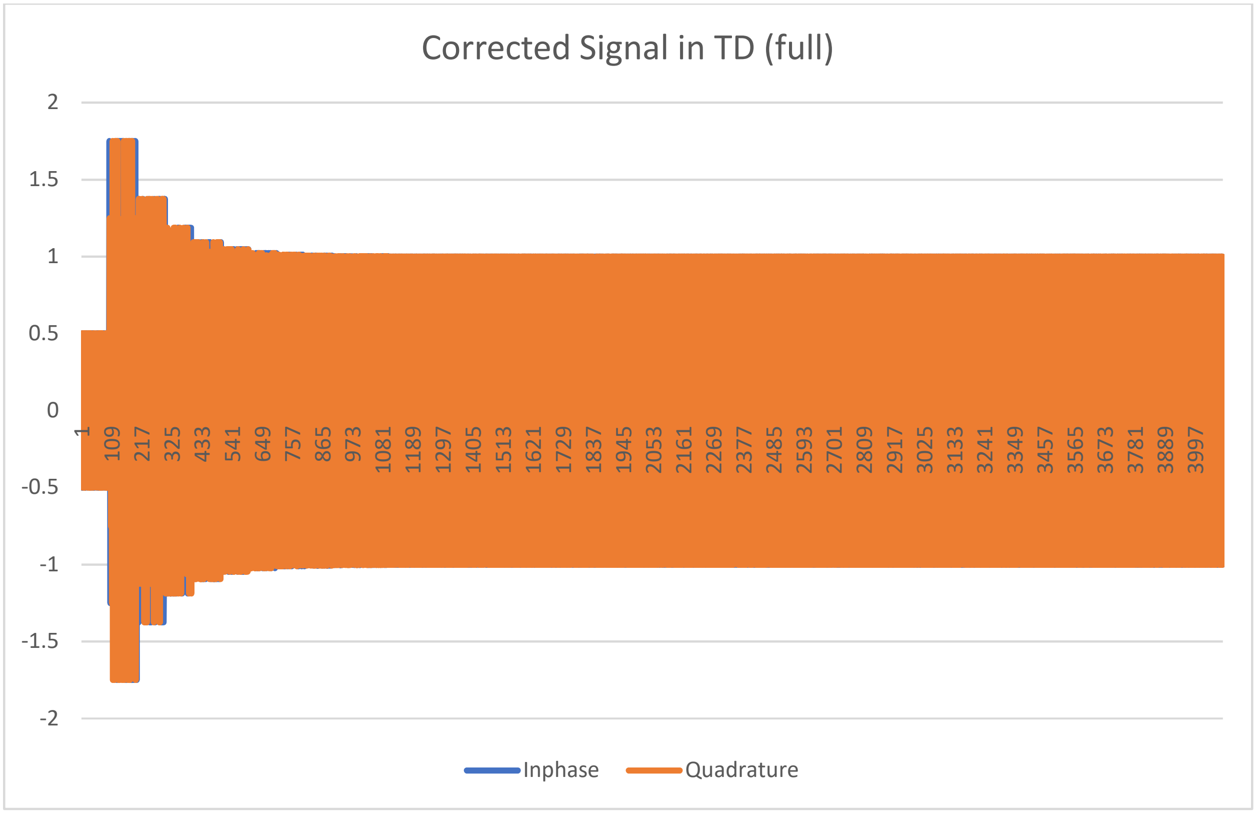
Corrected Signal in TD (full)
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
1
109
217
325
433
541
649
757
865
973
1081
1189
1297
1405
1513
1621
1729
1837
1945
2053
2161
2269
2377
2485
2593
2701
2809
2917
3025
3133
3241
3349
3457
3565
3673
3781
3889
3997
Inphase
Quadrature

Fig. 6

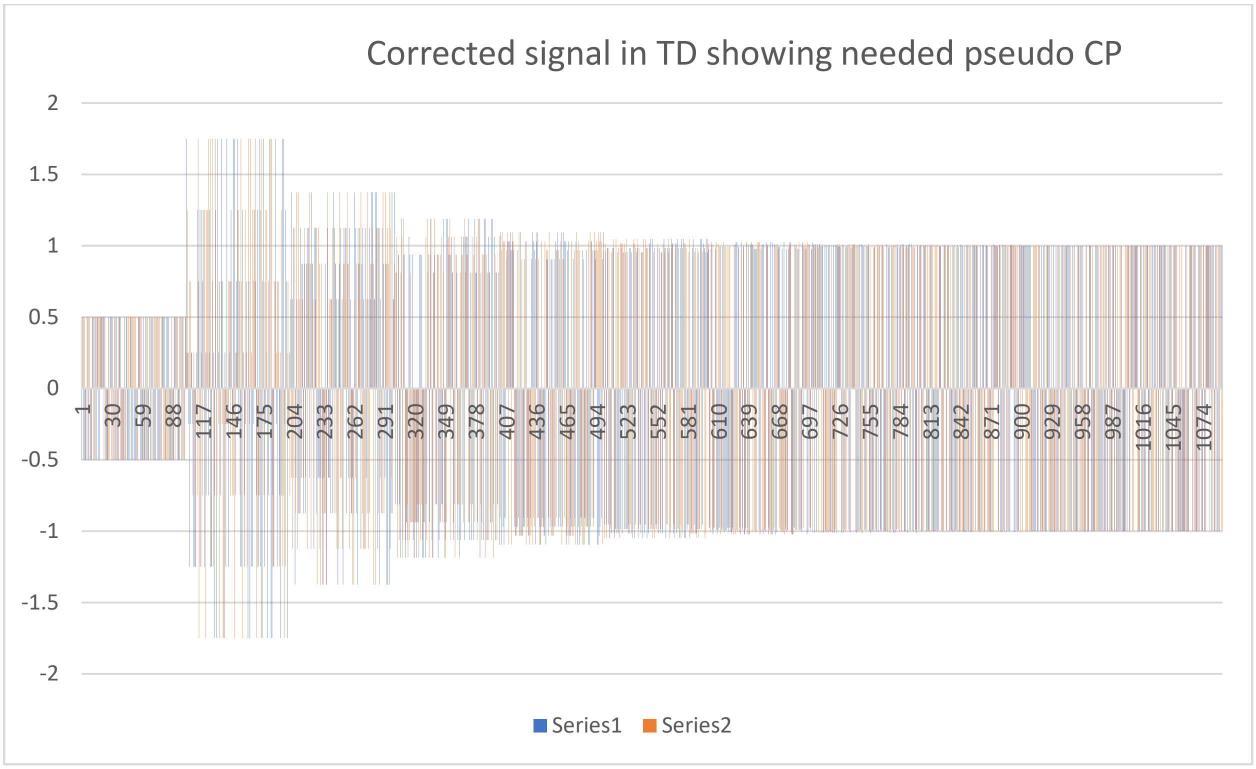
Corrected signal in TD showing needed pseudo CP
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
1
30
59
88
117
146
175
204
233
262
291
320
349
378
407
436
465
494
523
552
581
610
639
668
697
726
755
784
813
842
871
900
929
958
987
1016
1045
1074
Series1
Series2

Fig. 7

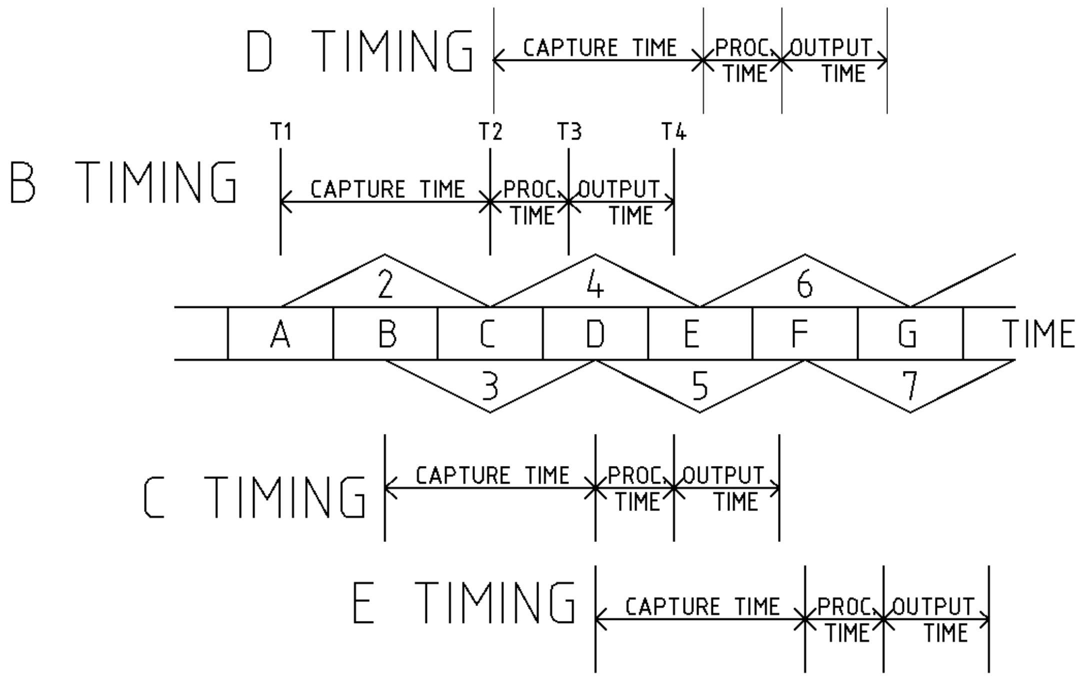

Fig. 8

Comments:

1. So, anything Bob Lucky's adaptive equalizer can do, this method can also do. It has a processing relative advantage when the echo duration is long, and the symbol period is short. The processing advantage translates both to a smaller chip size and lower power usage due to less compute cycles. Both methods need an accurate channel characterization, which can be blind, or assisted by a training (reference or de-ghosting) signal. Also, any use of an equalized signal processed by Lucky's adaptive equalizer is also anticipated by this invention, including, for example, hearing aids, CAT scanners, MRI machines, echo location, recordings, tissue analysis, or communications applications, such as signal demodulation.
2. Works for modulated or baseband signals.
3. Works to remove any type of linear distortion, including echoes, group delay, amplitude tilt, discrete reflections, or continuous (distributed) reflections.
4. Works for signals in any transmission medium, including wired cable, optic cable, wireless, air, water, solids (earth), RF or millimeter wave.
5. Works for any type of signal, including audio, sonar, electromagnetic energy.
6. Equalized signal can be demodulated or used for other purposes, such as analysis, amplification, recording, listening, etc.
7. Multiple equalizations can occur, such as could be used in MIMO (multiple input, multiple output), SISO (single input, single output), MISO (multiple input, single output), SISO (single input, single output).

8. Equalization can also be done with an overlapped discrete convolution with an inverse channel response.
9. An overlapped Zadoff-Chu sequence can be used for any (or all) of the functions of timing estimation, channel response estimation, and receive frequency error measurement.

Footnotes

(1) Ref. "Upstream Cable Echoes Come in Two Flavors" 2010 Feb. CED Magazine, by Williams, Campos, Currivan, and Moore. If an echo has a single recursion, the equalizing solution will be an infinite recursion sequence.

Previous Abstract:

When a condition is encountered in communications of a short symbol period and a long duration echo, a classical approach is to use a FIR (finite impulse response) filter to perform the necessary equalization. The short symbol periods are encountered in wideband communications, such as millimeter-wave, where a large RF bandwidth is available, or fiber optics where an enormous RF bandwidth is available. The long echoes are encountered in both fiber optic paths and wireless links. The problem is that the FIR filter must use many taps to cancel the echo, and the taps are clocked at a high rate. This results in large die area and high-power dissipation. An improved equalization approach requiring less compute cycles is to perform frequency domain (FD) equalization. However, to use traditional FD equalization the signal being equalized needs a cyclic prefix (CP), such as OFDM, SC-FDMA and OFDMA use. This requirement can be avoided if an overlapped Fourier transform is employed, creating a pseudo-cyclic prefix in the overlapped time domain symbols. This results in greatly reduced numerical computation, saving power.

Appendix A, C++ Simulation Code

```
// Generic_FD_Equalization.cpp : main project file.
#include "stdafx.h"
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <conio.h>
#include <string.h>
#include <time.h>       /* time for random number generator seed */

typedef struct {double real, imag;} COMPLEX;
extern void fft(COMPLEX *,int);
extern void ifft(COMPLEX *,int);
extern void rdata(int,int);
float re[4096],im[4096];

using namespace System;

int main(array<System::String ^> ^args)
{
       rdata(1,2);
       int i,j=100;
       FILE *output;
       float w,PI=3.1415926,a,b,c,d,denom,v=.52;

       COMPLEX *x;
       x=(COMPLEX*) calloc(4096, sizeof(COMPLEX));
       if(!x){printf("\n Unable to allocate input memory.\n");printf("\x7");exit(1);};

       COMPLEX *e;
       e=(COMPLEX*) calloc(4096, sizeof(COMPLEX));
       if(!e){printf("\n Unable to allocate input memory.\n");printf("\x7");exit(1);};

       COMPLEX *y;
       y=(COMPLEX*) calloc(4096, sizeof(COMPLEX));
       if(!y){printf("\n Unable to allocate input memory.\n");printf("\x7");exit(1);};


       COMPLEX *z;
       z=(COMPLEX*) calloc(4096, sizeof(COMPLEX));
       if(!z){printf("\n Unable to allocate input memory.\n");printf("\x7");exit(1);};


       if( (output = fopen("output.txt", "w") ) == NULL){     //write results to a file
              printf("could not open file.\n");
              system("pause");
              exit(0);
       }


       for(i=0;i<4096;i++){
              e[i].real = re[i];
```

```
			e[i].imag = im[i];
		}

		//
		//add an echo

		for(i=j;i<4096;i++){
			x[i].real = e[i].real + v*e[i-j].real;
			x[i].imag = e[i].imag + v*e[i-j].imag;
		}

			y[0].real = 1.00;
			y[j].real = v;

		fft(x,12);//take distorted sequence into FD
		fft(y,12);

		for(i=0;i<4096;i++){
			a= x[i].real;
			b=x[i].imag;
			c=y[i].real;
			d=y[i].imag;
			denom = c*c + d*d;
			z[i].real = (a*c + b*d)/denom;
			z[i].imag = (b*c - a*d)/denom;

		fprintf(output,"%d\t%f\t%f\t%f\t%f\t%f\t%f\n",i,a,b,c,d,z[i].real,z[i].imag);
		}

		ifft(z,12);

		for(i=0;i<4096;i++){
			fprintf(output,"%d\t%f\t%f\n",i,z[i].real,z[i].imag);
		}
		//see 1024 to 3072

    Console::WriteLine(L"Hello World");
    return 0;
}


void rdata(int q,int p){
		int i,j,k=0;
		FILE *outputdata;
		float random[4096],Tre[4096],Tim[4096] ;

		COMPLEX *x;
		x=(COMPLEX*) calloc(64, sizeof(COMPLEX));
		if(!x){printf("\n Unable to allocate input memory.\n");printf("\x7");exit(1);};

		COMPLEX *z;
		z=(COMPLEX*) calloc(2048, sizeof(COMPLEX));
		if(!z){printf("\n Unable to allocate input memory.\n");printf("\x7");exit(1);};

		if( (outputdata = fopen("outputD.txt ", "w") ) == NULL){	//write results to a
file for plots
			printf("could not open file.\n");
			system("pause");
```

```
              exit(0);
       }
       if(k!=0)srand (time(NULL));
       if(k==0)srand (5280+q);//mile-hi : always same random sequence
       for (i=0; i<4096; i++){//
              random[i]=rand();
              if(random[i]-16384 > 0){
                     Tre[i]=1;
                     random[i]-=16384;
              }
              else Tre[i] = -1;

              if(random[i]-8192 > 0){
                     Tim[i]=1;
                     random[i]-=8192;
              }
              else Tim[i] = -1;
       }
       for(i=0;i<4096;i++){
              re[i]=Tre[i];
              im[i]=Tim[i];
       }


}//---------------------------------------------END OF RDATA----------------------------
-----------------------------


void fft(COMPLEX *x,int m)

{
    static COMPLEX *w;           /* used to store the w complex array */
    static int mstore = 0;       /* stores m for future reference */
    static int n = 1;            /* length of fft stored for future */

    COMPLEX u,temp,tm;
    COMPLEX *xi,*xip,*xj,*wptr;

    int i,j,k,l,le,windex;

    double arg,w_real,w_imag,wrecur_real,wrecur_imag,wtemp_real;

    if(m != mstore) {

/* free previously allocated storage and set new m */

        if(mstore != 0) free(w);
        mstore = m;
        if(m == 0) exit(1);       /* if m=0 then done */

/* n = 2**m = fft length */

        n = 1 << m;
        le = n/2;

/* allocate the storage for w */

        w = (COMPLEX *) calloc(le-1,sizeof(COMPLEX));
        if(!w) {
            printf("\nUnable to allocate complex W array\n");
            exit(1);
        }

/* calculate the w values recursively */

        arg = 4.0*atan(1.0)/le;         /* PI/le calculation */
        wrecur_real = w_real = cos(arg);
        wrecur_imag = w_imag = -sin(arg);
        xj = w;
        for (j = 1 ; j < le ; j++) {
            xj->real = (double)wrecur_real;
            xj->imag = (double)wrecur_imag;
            xj++;
            wtemp_real = wrecur_real*w_real - wrecur_imag*w_imag;
            wrecur_imag = wrecur_real*w_imag + wrecur_imag*w_real;
            wrecur_real = wtemp_real;
        }
    }

/* start fft */

    le = n;
    windex = 1;
    for (l = 0 ; l < m ; l++) {
        le = le/2;

/* first iteration with no multiplies */

        for(i = 0 ; i < n ; i = i + 2*le) {
            xi = x + i;
            xip = xi + le;
            temp.real = xi->real + xip->real;
            temp.imag = xi->imag + xip->imag;
            xip->real = xi->real - xip->real;
            xip->imag = xi->imag - xip->imag;
            *xi = temp;
        }

/* remaining iterations use stored w */

        wptr = w + windex - 1;
        for (j = 1 ; j < le ; j++) {
            u = *wptr;
            for (i = j ; i < n ; i = i + 2*le) {
                xi = x + i;
                xip = xi + le;
                temp.real = xi->real + xip->real;
                temp.imag = xi->imag + xip->imag;
                tm.real = xi->real - xip->real;
                tm.imag = xi->imag - xip->imag;
                xip->real = tm.real*u.real - tm.imag*u.imag;
                xip->imag = tm.real*u.imag + tm.imag*u.real;
                *xi = temp;
            }
            wptr = wptr + windex;
        }
        windex = 2*windex;
    }

/* rearrange data by bit reversing */

    j = 0;
    for (i = 1 ; i < (n-1) ; i++) {
        k = n/2;
                        while(k <= j) {
            j = j - k;
            k = k/2;
        }
        j = j + k;
        if (i < j) {
            xi = x + i;
            xj = x + j;
            temp = *xj;
            *xj = *xi;
            *xi = temp;
        }
    }
}

void ifft(COMPLEX *x,int m)
{
    static COMPLEX *w;           /* used to store the w complex array */
    static int mstore = 0;       /* stores m for future reference */
    static int n = 1;            /* length of ifft stored for future */

    COMPLEX u,temp,tm;
    COMPLEX *xi,*xip,*xj,*wptr;

    int i,j,k,l,le,windex;

    double arg,w_real,w_imag,wrecur_real,wrecur_imag,wtemp_real;
    double scale;

    if(m != mstore) {

/* free previously allocated storage and set new m */

        if(mstore != 0) free(w);
        mstore = m;
        if(m == 0) exit(1);       /* if m=0 then done */

/* n = 2**m = inverse fft length */

        n = 1 << m;
        le = n/2;

/* allocate the storage for w */

        w = (COMPLEX *) calloc(le-1,sizeof(COMPLEX));
        if(!w) {
            printf("\nUnable to allocate complex W array\n");
            exit(1);
        }
```

```
/* calculate the w values recursively */

        arg = 4.0*atan(1.0)/le;         /* PI/le calculation */
        wrecur_real = w_real = cos(arg);
        wrecur_imag = w_imag = sin(arg);  /* opposite sign from fft */
        xj = w;
        for (j = 1 ; j < le ; j++) {
            xj->real = (double)wrecur_real;
            xj->imag = (double)wrecur_imag;
            xj++;
            wtemp_real = wrecur_real*w_real - wrecur_imag*w_imag;
            wrecur_imag = wrecur_real*w_imag + wrecur_imag*w_real;
            wrecur_real = wtemp_real;
        }
    }

/* start inverse fft */

    le = n;
    windex = 1;
    for (l = 0 ; l < m ; l++) {
        le = le/2;

/* first iteration with no multiplies */

        for(i = 0 ; i < n ; i = i + 2*le) {
            xi = x + i;
            xip = xi + le;
            temp.real = xi->real + xip->real;
            temp.imag = xi->imag + xip->imag;
            xip->real = xi->real - xip->real;
            xip->imag = xi->imag - xip->imag;
            *xi = temp;
        }

/* remaining iterations use stored w */

        wptr = w + windex - 1;
        for (j = 1 ; j < le ; j++) {
            u = *wptr;
            for (i = j ; i < n ; i = i + 2*le) {
                xi = x + i;
                xip = xi + le;
                temp.real = xi->real + xip->real;
                temp.imag = xi->imag + xip->imag;
                tm.real = xi->real - xip->real;
                tm.imag = xi->imag - xip->imag;
                xip->real = tm.real*u.real - tm.imag*u.imag;
                xip->imag = tm.real*u.imag + tm.imag*u.real;
                *xi = temp;
            }
            wptr = wptr + windex;
        }
        windex = 2*windex;
    }

/* rearrange data by bit reversing */
```

```
    j = 0;
    for (i = 1 ; i < (n-1) ; i++) {
        k = n/2;
        while(k <= j) {
            j = j - k;
            k = k/2;
        }
        j = j + k;
        if (i < j) {
            xi = x + i;
            xj = x + j;
            temp = *xj;
            *xj = *xi;
            *xi = temp;
        }
    }

/* scale all results by 1/n */
    scale = (double)(1.0/n);
    for(i = 0 ; i < n ; i++) {
        x->real = scale*x->real;
        x->imag = scale*x->imag;
        x++;
    }
}
```