Filter C Programming

(3A) IIR Filter

Copyright (c) 2018 - 2019 Young W. Lim.
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to youngwlim@hotmail.com.
This document was produced by using LibreOffice.

Based on

Introduction to Signal Processing

S. J. Ofranidis

dot.c

```
/* dot.c - dot product of two length-(M+1) vectors */
// Usage: y = dot(M, h, w);
// h = filter vector, w = state vector
// M = filter order
// compute dot product

double dot(int M, double *h, double *w)
{
    int i;
    double y;

    for (y=0, i=0; i<=M; i++)
        y += h[i] * w[i];

    return y;
}
```

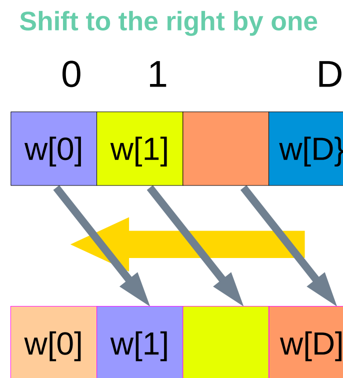
delay.c

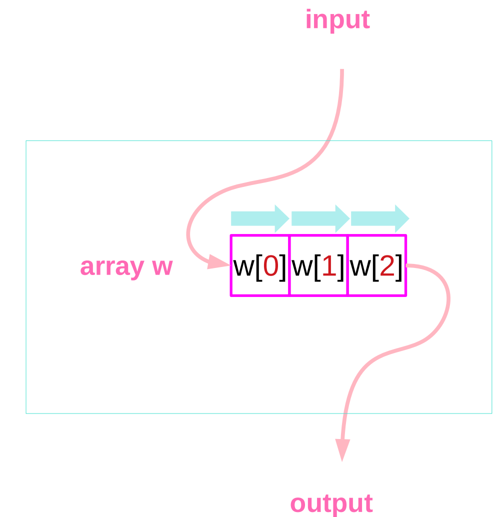
```
/* delay.c - delay by D time samples */
/* w[0] = input, w[D] = output */

void delay(int D, double *w)
{
    int i;

    for (i=D; i>=1; i--)
        w[i] = w[i-1];

    // reverse-order updating
}
```





order of execution

w[D] = w[D-1]

… …

w[2] = w[1]

w[1] = w[0]

wrap.c and wrap2.c

```
/* wrap.c - circular wrap of pointer p, relative to array w */
void wrap(int M, double *w, double *p)
{
        if (*p > w + M)
                *p -= M + 1;          // when ∗p = w + M + 1, it wraps around to ∗p = w
        if (*p < w)
                *p += M + 1;          // when ∗p = w − 1, it wraps around to ∗p = w + M
}
```

```
/* wrap2.c - circular wrap of pointer offset q, relative to array w */
void wrap2(int M, int *q)
{
        if (*q > M)
                *q -= M + 1;          // when ∗q = M + 1, it wraps around to ∗q = 0
        if (*q < 0)
                *q += M + 1;          // when ∗q = − 1, it wraps around to ∗q = M
}
```

Using **cfir2**

$v0(n) = x(n)$
$w0(n) = y(n)$
$v1(n) = x(n-1) = v0(n-1)$
and
$v2(n) = x(n-2) = v1(n-1)$
$w1(n) = y(n-1) = w0(n-1)$
$w2(n) = y(n-2) = w1(n-1)$

$v1(n + 1) = v0(n)$
$v2(n + 1) = v1(n)$
and
$w1(n + 1) = w0(n)$
$w2(n + 1) = w1(n)$

Using **cfir2**

$v0\ (n) = x(n)$

$w0\ (n) = -a1\ w1(n) - a2\ w2(n) + b0\ v0(n) + b1\ v1(n) + b2\ v2(n)$

$y(n) = w\ 0\ (n)$

$v2(n+1) = v1(n),\ w2(n+1) = w1(n)$

$v1(n+1) = v0(n),\ w1(n+1) = w0(n)$

$v0 = x$

$w0 = -a1\ w1 - a2\ w2 + b0\ v0 + b1\ v1 + b2\ v2$

$y = w0$

$v2 = v1\ ,\ w2 = w1$

$v1 = v0\ ,\ w1 = w0$

dir.c

```
/* dir.c - IIR filtering in direct form */
// usage: y = dir(M, a, L, b, w, v, x);
// v, w are internal states
// denominator and numerator orders
double dir(int M, double *a, int L, double *b, double *w, double *v, double x)
{
	int i;
	v[0] = x;			// current input sample
	w[0] = 0;			// current output to be computed
	for (i=0; i<=L; i++)
		w[0] += b[i] * v[i];	// numerator part
	for (i=1; i<=M; i++)
		w[0] -= a[i] * w[i];	// denominator part
	for (i=L; i>=1; i--)
		v[i] = v[i-1];		// reverse-order updating of v
	for (i=M; i>=1; i--)
		w[i] = w[i-1];		// reverse-order updating of w
	return w[0];			// current output sample

}
```

Using **dir**

```
double *a, *b, *w, *v;
a = (double *) calloc(M+1, sizeof(double));        // (M+ 1 ) –dimensional
b = (double *) calloc(L+1, sizeof(double));        // (L+ 1 ) –dimensional

a[0] = 1;                                          // always so
w = (double *) calloc(M+1, sizeof(double));        // (M+ 1 ) –dimensional
v = (double *) calloc(L+1, sizeof(double));        // (L+ 1 ) –dimensional


for (n = 0; n < Ntot; n++)
      y[n] = dir(M, a, L, b, w, v, x[n]);
```

dir2.c

```
/* dir2.c - IIR filtering in direct form */
double dot(int M, double *h, double *w);
void delay(int D, double *w);


double dir2(int M, double *a, int L, double *b, double *w, double *v, double x)
{
      v[0] = x;                                  // current input sample
      w[0] = 0;                                  // needed for dot(M,a,w)

      w[0] = dot(L, b, v) - dot(M, a, w);        // current output

      delay(L, v);                               // update input delay line

      delay(M, w);                               // update output delay line

      return w[0];
}
```

can.c

```
/* can.c - IIR filtering in canonical form */
// usage: y = can(M, a, L, b, w, x);
// w = internal state vector
// M, L denominator and numerator orders

double can(int M, double *a, int L, double *b, double *w, double x)
{
        int K, i;
        double y = 0;

        K = (L <= M) ? M : L;           // K = max (M, L)
        w[0] = x;                       // current input sample
        for (i=1; i<=M; i++)
                w[0] -= a[i] * w[i];    // input adder
        for (i=0; i<=L; i++)
                y += b[i] * w[i];       // output adder
        for (i=K; i>=1; i--)
                w[i] = w[i-1];          // reverse updating of w
        return y;                       // current output sample
}
```

can2.c

```
/* can2.c - IIR filtering in canonical form */
//     usage: y = can2(M, a, L, b, w, x);

double dot(int M, double *h, double *w);
void delay(int D, double *w);

double can2(int M, double *a, int L, double *b, double *w, double x)
{
       int K;
       double y;

       K = (L <= M) ? M : L;               // K = max (M, L)
       w[0] = 0;                           // needed for dot(M,a,w)
       w[0] = x - dot(M, a, w);            // input adder
       y = dot(L, b, w);                   // output adder
       delay(K, w);                        // update delay line
       return y;                           // current output sample
}
```

sos.c

```
/* sos.c - IIR filtering by single second order section */
//      a, b, w are 3-dimensional
//      a[ 0 ]= 1 always

double sos(double *a, double *b, double *w, double x)
{
        double y;

        w[0] = x - a[1] * w[1] - a[2] * w[2];

        y = b[0] * w[0] + b[1] * w[1] + b[2] * w[2];

        w[2] = w[1];
        w[1] = w[0];

        return y;
}
```

Using **sos.c**

a = [1 , a 1 , a 2]
b = [b 0 , b 1 , b 2]
w = [w 0 , w 1 , w 2]

```
a = (double *) calloc(3, sizeof(double));
b = (double *) calloc(3, sizeof(double));
w = (double *) calloc(3, sizeof(double));
```

cas.c

```
/* cas.c - IIR filtering in cascade of second-order sections */
// A, B, W are K× 3 matrices

double sos(double *a, double *b, double *w, double x);
// single second-order section

double cas(int K, double **A, double **B, double **W, double x)
{
    int i;
    double y;

    y = x;                                  // initial input to first SOS
    for (i=0; i<K; i++)
        y = sos(A[i], B[i], W[i], y);       // output of i th section

    return y;                               // final output from last SOS
}
```

Using **cas.c**

```
double **A, **B, **W;
A = (double **) calloc(K, sizeof(double *));
B = (double **) calloc(K, sizeof(double *));
W = (double **) calloc(K, sizeof(double *));
for (i=0; i<K; i++) {
      A[i] = (double *) calloc(3, sizeof(double));
      B[i] = (double *) calloc(3, sizeof(double));
      W[i] = (double *) calloc(3, sizeof(double));
}

double A[K][3], B[K][3], W[K][3];
double A[][3], B[][3], W[][3];

for (n = 0; n < Ntot; n++)
      y[n] = cas(K, A, B, W, x[n]);
```

cas2can.c

```
/* cas2can.c - cascade to canonical */
#include <stdlib.h>
void conv(int M, double *h, int L, double *x, double *y);

void cas2can(int K, double **A, double *a)
{
    nt i,j;
    double *d;

    d = (double *) calloc(2*K+1, sizeof(double));
    a[0] = 1;                                        // initialize
    for(i=0; i<K; i++) {
        conv(2, A[i], 2*i+1, a, d);
        for(j=0; j<2*i+3; j++)
            a[j] = d[j];
        }                                            // d = a[i] ∗ a
                                                     // a = d
}
```

cas2can.c

```
/* cas2can.c - cascade to canonical */
#include <stdlib.h>
void conv(int M, double *h, int L, double *x, double *y);

void cas2can(int K, double **A, double *a)
{
        int i,j;
        double *d;

        d = (double *) calloc(2*K+1, sizeof(double));

        a[0] = 1;                               // initialize
        for(i=0; i<K; i++) {
                conv(2, A[i], 2*i+1, a, d);     // d = a[i] ∗ a
                for(j=0; j<2*i+3; j++)          // a = d
                        a[j] = d[j];
                }

}
```

cas3.c

```
/* can3.c - IIR filtering in canonical form, emulating a DSP chip */
// usage: y = can3(M, a, b, w, x);
// w = internal state vector
// a, b have order M

double can3(int M, double *a, double *b, double *w, double x)
{
        int i;
        double y;
        w[0] = x;                              // read input sample
        for (i=1; i<=M; i++)                   // forward order
                w[0] -= a[i] * w[i];           // MAC instruction
        y = b[M] * w[M];

        for (i=M-1; i>=0; i--) {               // backward order
                w[i+1] = w[i];                 // data shift instruction
                y += b[i] * w[i];              // MAC instruction
        }
        return y;                              // output sample
}
```

cas3.c

```
/* can3.c - IIR filtering in canonical form, emulating a DSP chip */
// usage: y = can3(M, a, b, w, x);
// w = internal state vector
// a, b have order M

double can3(int M, double *a, double *b, double *w, double x)
{
	int i;
	double y;
	w[0] = x;                            // read input sample
	for (i=1; i<=M; i++)                 // forward order
		w[0] -= a[i] * w[i];             // MAC instruction
	y = b[M] * w[M];

	for (i=M-1; i>=0; i--) {             // backward order
		w[i+1] = w[i];                   // data shift instruction
		y += b[i] * w[i];                // MAC instruction
	}
	return y;                            // output sample
}
```

ccan.c (1)

```
/* ccan.c - circular buffer implementation of canonical realization */
// defined in Section 4.2.4
// usage: y = ccan(M, a, b, w, &p, x);
// p = circular pointer to buffer w
// a, b have common order M
void wrap(int M, double *w, double *p)

double ccan(int M, double *a, double *b, double *w, double **p, double x)
{
        int i;
        double y = 0, s0;

        **p = x; read input sample x

        s0 = *(*p)++;                    // s0 = x
        wrap(M, w, p);                   // p now points to s 1
```

ccan.c (2)

```
    for (a++, i=1; i<=M; i++) {           // start with a incremented to a 1
        s0 -= (*a++) * (*(*p)++);
        wrap(M, w, p);
    }

    **p = s0;                              // p has wrapped around once

    for (i=0; i<=M; i++) {                 // numerator part
        y += (*b++) * (*(*p)++);
        wrap(M, w, p);                     // upon exit, p has wrapped
    }                                      // around once again
    (*p)--;                                // update circular delay line
    wrap(M, w, p);

    return y;                              // output sample
}
```

Using **ccan.c**

```
double *a, *b,*w, *p;
a = (double *) calloc(M+1, sizeof(double));
b = (double *) calloc(M+1, sizeof(double));
w = (double *) calloc(M+1, sizeof(double));          // initializes w to zero
a[0] = 1;                                            // not used in the routine

p = w;                                               // initialize p
for (n = 0; n < Ntot; n++)
        y[n] = ccan(M, a, b, w, &p, x[n]);           // p is passed by address
```

csos.c

```
/* csos.c - circular buffer implementation of a single SOS */
// a, b, w are 3-dimensional
// p is circular pointer to w
void wrap(int M, double *w, double *p)
double csos(double *a, double *b, double *w, double **p, double x)
{
    double y, s0;

    *(*p) = x;                                          // read input sample x
    s0 = *(*p)++;             wrap(2, w, p);
    s0 -= a[1] * (*(*p)++);   wrap(2, w, p);
    s0 -= a[2] * (*(*p)++);   wrap(2, w, p);

    *(*p) = s0;                                         // p has wrapped around once
    y = b[0] * (*(*p)++);     wrap(2, w, p);
    y += b[1] * (*(*p)++);    wrap(2, w, p);
    y += b[2] * (*(*p));p                               // now points to s 2

    return y;
}
```

ccas.c

```
/* ccas.c - circular buffer implementation of cascade realization */
// P = array of circular pointers
double csos(double *a, double *b, double *w, double **p, double x);
            // circular-buffer version of single SOS

double ccas(int K, double **A, double **B, double **W, double **P, double x)
{
      int i;
      double y;



      y = x;
      for (i=0; i<K; i++)
            y = csos(A[i], B[i], W[i], P+i, y);             // note, P + i = & P[i]

      return y;
}
```

Using **ccas.c**

```
double **P;

P = (double **) calloc(K, sizeof(double *));            // array of K pointers

for (i=0; i<K; i++)
      P[i] = W[i];                                       // P[i] = i th row of W

for (n = 0; n < Ntot; n++)
      y[n] = ccas(K, A, B, W, P, x[n]);
```

ccan2.c

```
/* ccan2.c - circular buffer implementation of canonical realization */
// q = circular pointer offset index
// a, b have common order M
void wrap2(int M, int *q);

double ccan2(int M, double *a, double *b, double *w, double *q, double x)
{
    int i;
    double y = 0;

    w[*q] = x;                              // read input sample x
    for (i=1; i<=M; i++)
        w[*q] -= a[i] * w[(*q+i)%(M+1)];
    for (i=0; i<=M; i++)
        y += b[i] * w[(*q+i)%(M+1)];
    (*q)--;                                 // update circular delay line
    wrap2(M, q);
    return y;                               // output sample
}
```

Using **ccan2.c**

```
int q;
double *a, *b, *w;

a = (double *) calloc(M+1, sizeof(double));
b = (double *) calloc(M+1, sizeof(double));
w = (double *) calloc(M+1, sizeof(double));          // initializes w to zero

a[0] = 1;                                             // not used in the routine

q = 0;                                                // initialize q

for (n = 0; n < Ntot; n++)
        y[n] = ccan2(M, a, b, w, &q, x[n]);           // p is passed by address
```

csos2.c

```
/* csos2.c - circular buffer implementation of a single SOS */
// a, b, w are 3-dimensional arrays
// q is circular offset relative to w
void wrap2(int M, int *q)

double csos2(double *a, double *b, double *w, int *q, double x)
{
      double y;
      w[*q] = x - a[1] * w[(*q+1)%3] - a[2] * w[(*q+2)%3];

      y = b[0] * w[*q] + b[1] * w[(*q+1)%3] + b[2] * w[(*q+2)%3];

      (*q)--;
      wrap2(2, q);

      return y;
}
```

ccas2.c

```
/* ccas2.c - circular buffer implementation of cascade realization */
// circular-buffer version of single SOS
// Q = array of circular pointer offsets
double csos2(double *a, double *b, double *w, int *q, double x)

double ccas2(int K, double **A, double **B, double **W, int *Q, double x)
{
        int i;
        double y;
        y = x;

        for (i=0; i<K; i++)
                y = csos2(A[i], B[i], W[i], Q+i, y);     // note, Q + i = & Q[i]

        return y;
}
```

ccas2.c

```
int *Q;

Q = (double *) calloc(K, sizeof(double));        // array of K integers

for (i=0; i<K; i++)
     Q[i] = 0;                                    // initialize Q[i]

for (n = 0; n < Ntot; n++)
     y[n] = ccas2(K, A, B, W, Q, x[n]);
```

Direct Form

$v0(n)=x(n)$
$w0(n)=y(n)$
$v1(n)=x(n-1)=v0(n-1)$
and
$v2(n)=x(n-2)=v1(n-1)$
$w1(n)=y(n-1)=w0(n-1)$
$w2(n)=y(n-2)=w1(n-1)$

$v1(n+1)=v0(n)$
$v2(n+1)=v1(n)$
and
$w1(n+1)=w0(n)$
$w2(n+1)=w1(n)$

Direct Form

$v0(n)=x(n)$

$w0(n)=-a1w1(n)-a2w2(n)+b0v0(n)+b1v1(n)+b2v2(n)$

$y(n)=w0(n)$

$v2(n+1)=v1(n), w2(n+1)=w1(n)$

$v1(n+1)=v0(n), w1(n+1)=w0(n)$

For each input sample x do:

$v0=x$

$w0=-a1w1-a2w2+b0v0+b1v1+b2v2$

$y=w0$

$v2=v1, w2=w1$

$v1=v0, w1=w0$

Direct Form

For each input sample x do:
$v_0 = x$
$w_0 = -a_1 w_1 - \cdots - a_M w_M + b_0 v_0 + b_1 v_1 + \cdots + b_L v_L$
$y = w_0$
$v_i = v_{i-1}, \quad i = L, L-1, \ldots, 1$
$w_i = w_{i-1}, i = M, M-1, \ldots, 1$

For each input sample x do:
$v_0 = x$
$w_0 = -0.2 w_1 + 0.3 w_2 - 0.5 w_4 + 2 v_0 - 3 v_1 + 4 v_3$
$y = w_0$
$w_4 = w_3$
$w_3 = w_2, \qquad v_3 = v_2$
$w_2 = w_1, \qquad v_2 = v_1$
$w_1 = w_0, \qquad v_1 = v_0$

Canonical Form

$w(n)=x(n)-a1w(n-1)-a2w(n-2)$
$y(n)=b0w(n)+b1w(n-1)+b2w(n-2)$

$w0(n)=x(n)-a1w1(n)-a2w2(n)$
$y(n)=b0w0(n)+b1w1(n)+b2w2(n)$
$w2(n+1)=w1(n)$
$w1(n+2)=w0(n)$

For each input samplex do:
$w0=x-a1w1-a2w2$
$y=b0w0+b1w1+b2w2$
$w2=w1$
$w1=w0$

Canonical Form

$w(n)=x(n)-a_1w(n-1)-\cdots-a_Mw(n-M)$
$y(n)=b_0w(n)+b_1w(n-1)+\cdots+b_Lw(n-L)$

$w_0(n)=x(n)-a_1w_1(n)-\cdots-a_Mw_M(n)$
$y(n)=b_0w_0(n)+b_1w_1(n)+\cdots+b_Lw_L(n)$
$w_i(n+1)=w_{i-1}(n),$
$i=K,K-1,...,1$

For each input sample x do:
$w_0=x-a_1w_1-a_2w_2-\cdots-a_Mw_M$
$y=b_0w_0+b_1w_1+\cdots+b_Lw_L$
$w_i=w_{i-1}, i=K,K-1,...,1$

Canonical Form

```
for each input sample x do:
w0 = x − 0.2 w1 + 0.3 w2 − 0.5 w4
y = 2 w0 − 3 w1 + 4 w3
w4 = w3
w3 = w2
w2 = w1
w1 = w0
```

```
for each input sample x do:
w0 = 0
w0 = x − dot (M, a , w )
y = dot (L, b , w )
delay (K, w )
```

Cascade Form (1)

$x_0(n) = x(n)$

for $i = 0, 1, \ldots, K-1$ do:

$w_i(n) = x_i(n) - a_{i1} w_i(n-1) - a_{i2} w_i(n-2)$

$y_i(n) = b_{i0} w_i(n) + b_{i1} w_i(n-1) + b_{i2} w_i(n-2)$

$x_{i+1}(n) = y_i(n)$

$y(n) = y_{K-1}(n)$

for each input sample x do:

$x_0 = x$

for $i = 0, 1, \ldots, K-1$ do:

$w_{i0} = x_i - a_{i1} w_{i1} - a_{i2} w_{i2}$

$y_i = b_{i0} w_{i0} + b_{i1} w_{i1} + b_{i2} w_{i2}$

$w_{i2} = w_{i1}$

$w_{i1} = w_{i0}$

$x_{i+1} = y_i$

$y = y_{K-1}$

Cascade Form (2)

```
for each input sample x do:
y=x
for i=0,1,...,K−1 do:
y=sos(ai,bi,wi,y)
```

```
for each input sample x do:
w00=x+0.4w01−0.5w02
x1=3w00−4w01+2w02
w02=w01
w01=w00
w10=x1−0.4w11−0.5w12
y=3w10+4w11+2w12
w12=w11
w11=w10
```

Cascade Form (3)

for each input sample x do:

$w0 = x - 0.84w2 - 0.25w4$

$y = 9w0 - 4w2 + 4w4$

$w4 = w3$

$w3 = w2$

$w2 = w1$

$w1 = w0$

Cascade to Canonical (1)

```
for each input x do:
w0 = x + 0.0625w8
y = w0 + w8
delay(8, w)
```

```
for each input x do:
w0 = x − 0 94w8
x1 = w0 + w8
delay(8 , w)
v0 = x1 + 0.98v8
y = v0 − 0.96v8
delay(8 , v)
```

Cascade to Canonical (2)

$d = \delta$
for $i = 0, 1, \ldots, K-1$ do:
$d = a_i \ast d$
$a = d$

$a = \delta$
for $i = 0, 1, \ldots, K-1$ do:
$d = a_i \ast a$
$a = d$

Hardware (1)

```
for each input sample x do:
w0 : = x − a1 w1
w0 : = w 0 − a2 w2
y : = b2 w2
w2 : = w1 , y : = y + b1 w1
w1 : = w0 , y : = y + b0 w0
```

```
for each input sample x do:
w0 : = x
for i = 1 , 2 , . . . , M do:
w0 : = w0 − ai wi
y : = bM wM
for i = M− 1 , . . . , 1 , 0 do:
wi+ 1 : = wi
y : = y + bi wi
```

Hardware (2)

```
for each input sample x do:
for i = 1 , 2 , . . . , M determine states:
si = tap (M, w , p, i)
s0 = x − a1 s1 − · · · − aM sM
y = b0 s0 + b1 s1 + · · · + bM sM
∗p = s0
cdelay (M, w, &p)
```

```
for each input sample x do:
s1 = tap ( 4, w, p, 1 )
s2 = tap ( 4, w, p, 2 )
s3 = tap ( 4, w, p, 3 )
s4 = tap ( 4, w, p, 4 )
s0 = x − 0.2 s1 + 0.3 s2 − 0.5 s4
y = 2 s0 − 3 s1 + 4 s3
∗p = s0
cdelay ( 4, w, &p)
```

Hardware (3)

```
or each input sample x do:
s8 = tap ( 8, w, p, 8 )
s0 = x + 0.0625 s8
y = s0 + s8
∗p = s0
cdelay ( 8, w, &p)
```

```
for each input sample x do:
s1 = tap ( 3, w, p, 1 )
s2 = tap ( 3, w, p, 2 )
s3 = tap ( 3, w, p, 3 )
s0 = x + s3
y = s0 + s1 + 2 s2
∗p = s0
cdelay ( 3, w, &p)
```

Hardware (4)

```
for each input sample x do:
s8 = tap(8, w, p, 8)
s0 = x + 0.0625 s8
y = s0 + s8
∗p = s0
cdelay(8, w, &p)

for each input sample x do:
s1 = tap( 3, w, p, 1)
s2 = tap( 3, w, p, 2)
s3 = tap( 3, w, p, 3)
s0 = x + s3
y = s0 + s1 + 2 s2
∗p = s0
cdelay(3, w, &p)
```

Hardware (5)

```
for each input sample x do:
s1 = tap( 2, w, p, 1)
s2 = tap( 2, w, p, 2)
s0 = x − a1 s1 − a2 s2
y = b0 s0 + b1 s1 + b2 s2
∗p = s0
cdelay(2, w, &p)
```

```
for each input sample x do:
y = x
for i = 0, 1, . . . , K − 1 do:
y = csos( ai , bi , wi , &pi , y)
```

Hardware (6)

```
for each input sample x do:
s1 = tap( 2, w0, p0, 1)
s2 = tap( 2, w0, p0, 2)
s0 = x + 0.4 s1 − 0.5 s2
x1 = 3 s0 − 4 s1 + 2 s2
∗p0 = s0
cdelay ( 2, w0, &p0)
s1 = tap ( 2 , w1, p1, 1)
s2 = tap ( 2 , w1, p1, 2)
s0 = x1 − 0.4 s1 − 0.5 s2
y = 3 s0 + 4 s1 + 2 s2
∗p1 = s0
cdelay( 2, w1 , &p1)
```

References

[1] S. J. Ofranidis , Introduction to Signal Processing