Filter C Programming

(1A) Convolution

Young Won Lim
6/19/19

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

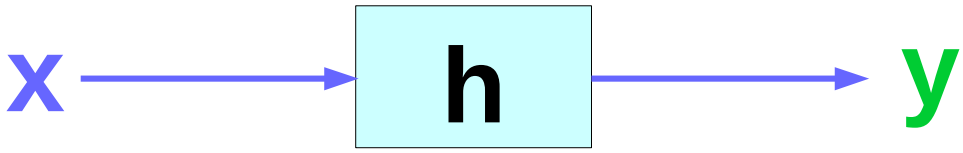
Young Won Lim
6/19/19

Based on

Introduction to Signal Processing

S. J. Ofranidis

Index Variable Constraints



x[0..L-1]
input array
L input length

h[0..M-1]
filter array,
M (filter length)
M-1 (filter order)

y[0..L+M-2]
output array
(M+L-1) output length

Assume
M < L

Case A

$$y[n] \mathrel{+}= h[m] * x[n-m];$$

$$\begin{aligned} n &\in [0, L+M-2] \\ m &\in [0, M-1] \\ n-m &\in [0, L-1] \end{aligned}$$

Case B

$$y[n] \mathrel{+}= x[m] * h[n-m];$$

$$\begin{aligned} n &\in [0, L+M-2] \\ m &\in [0, L-1] \\ n-m &\in [0, M-1] \end{aligned}$$

Flipped and shifted waveforms

h[m] the original waveform

h[-m] The flipped against y-axis

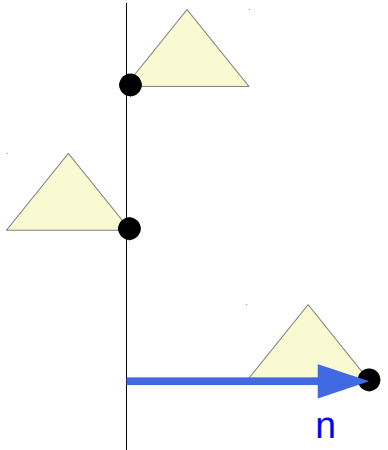
h[-(m-n)] = h[n-m] shift to the right by n

x[m] the original waveform

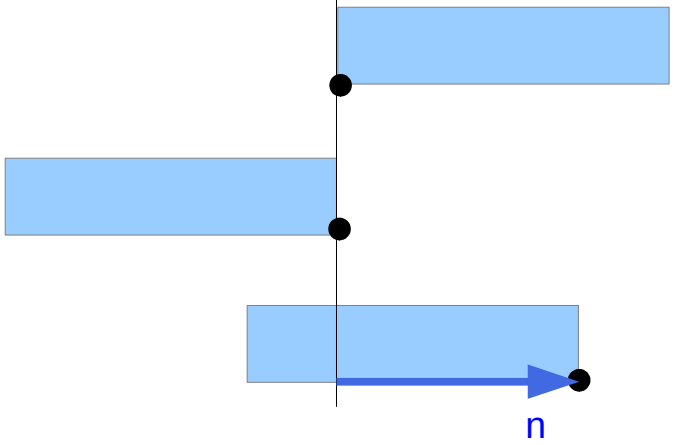
x[-m] The flipped against y-axis

x[-(m-n)] = x[n-m] shift to the right by n

Case A



Case B

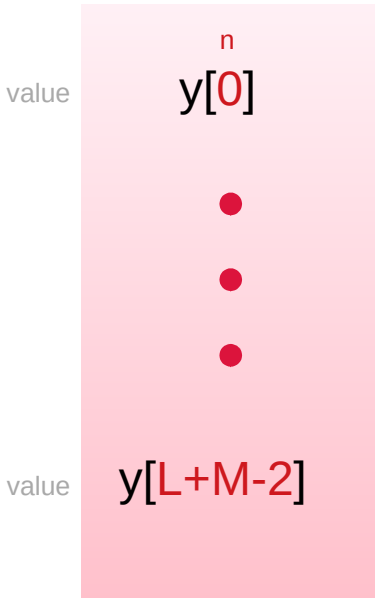


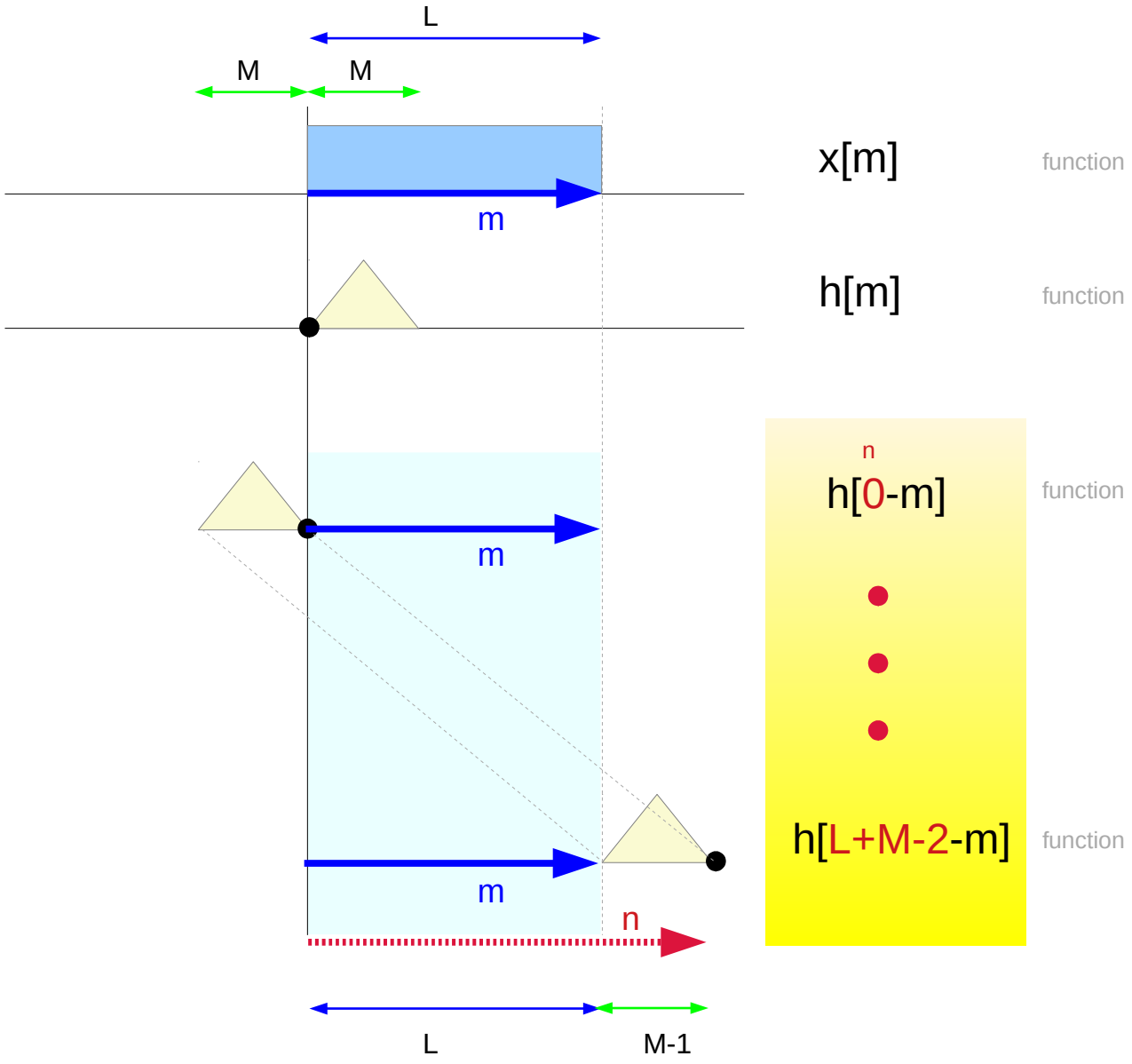
Flipped and shifted function of **h[n-m]**

Case A

```
M+L-1          L          M
y[n] += x[m] * h[n-m];
```

n ∈ [0, L+M-2]
m ∈ [0, L-1]
n-m ∈ [0, M-1]



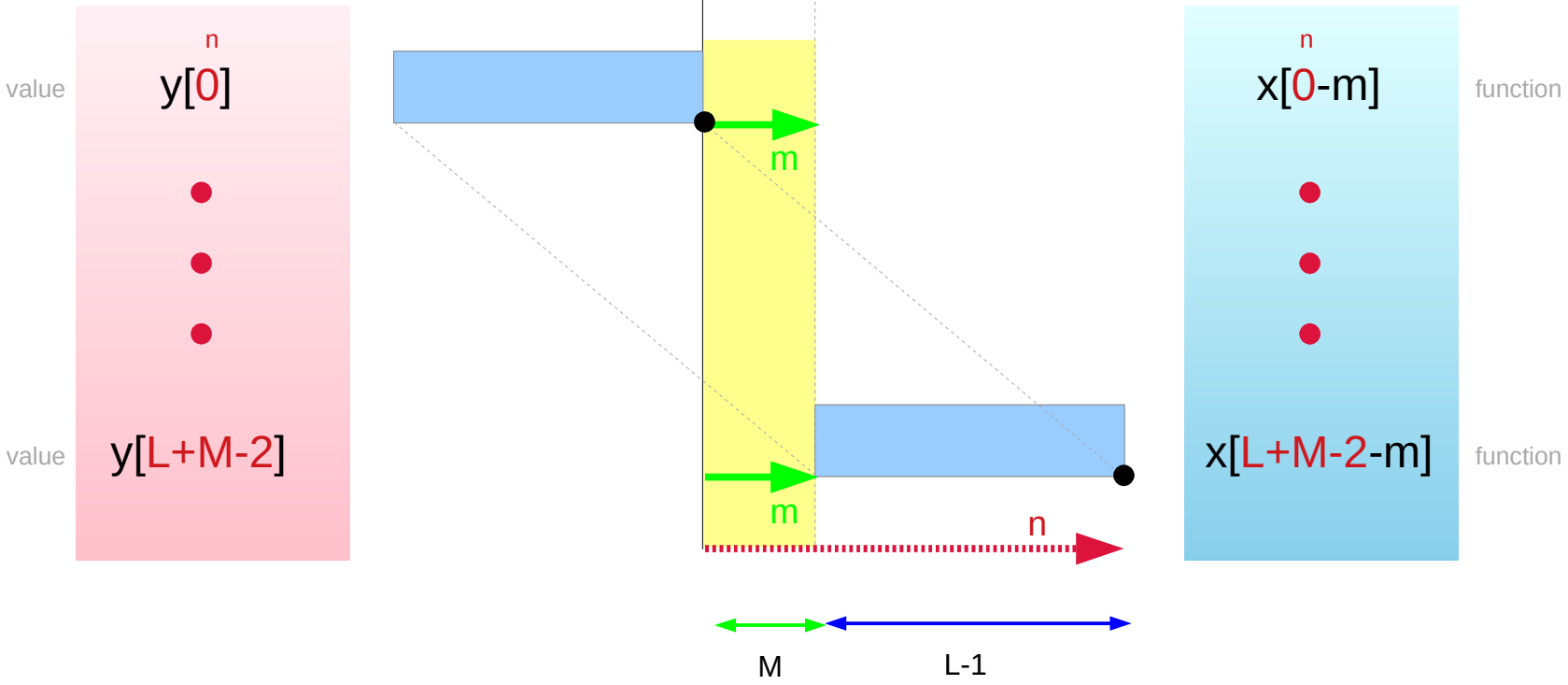


Flipped and shifted function of **x[n-m]**

Case B

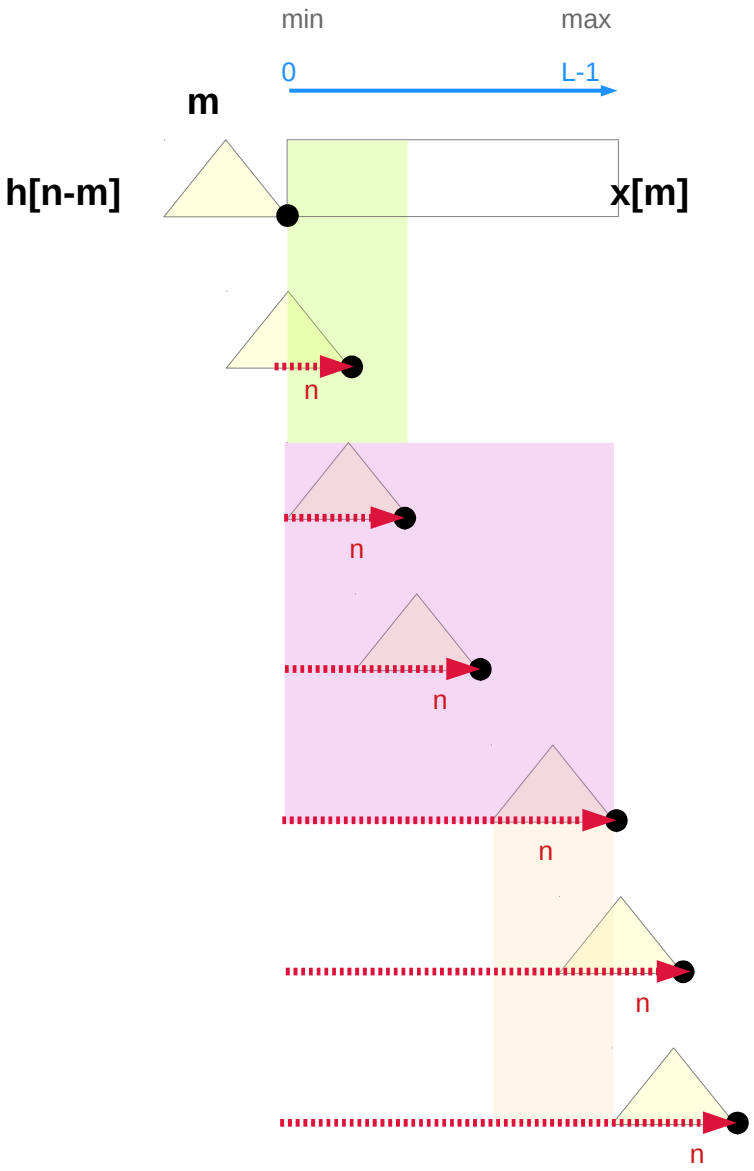
$$y[n] \mathrel{+}= \overset{M}{h[m]} * \overset{L}{x[n-m]};$$

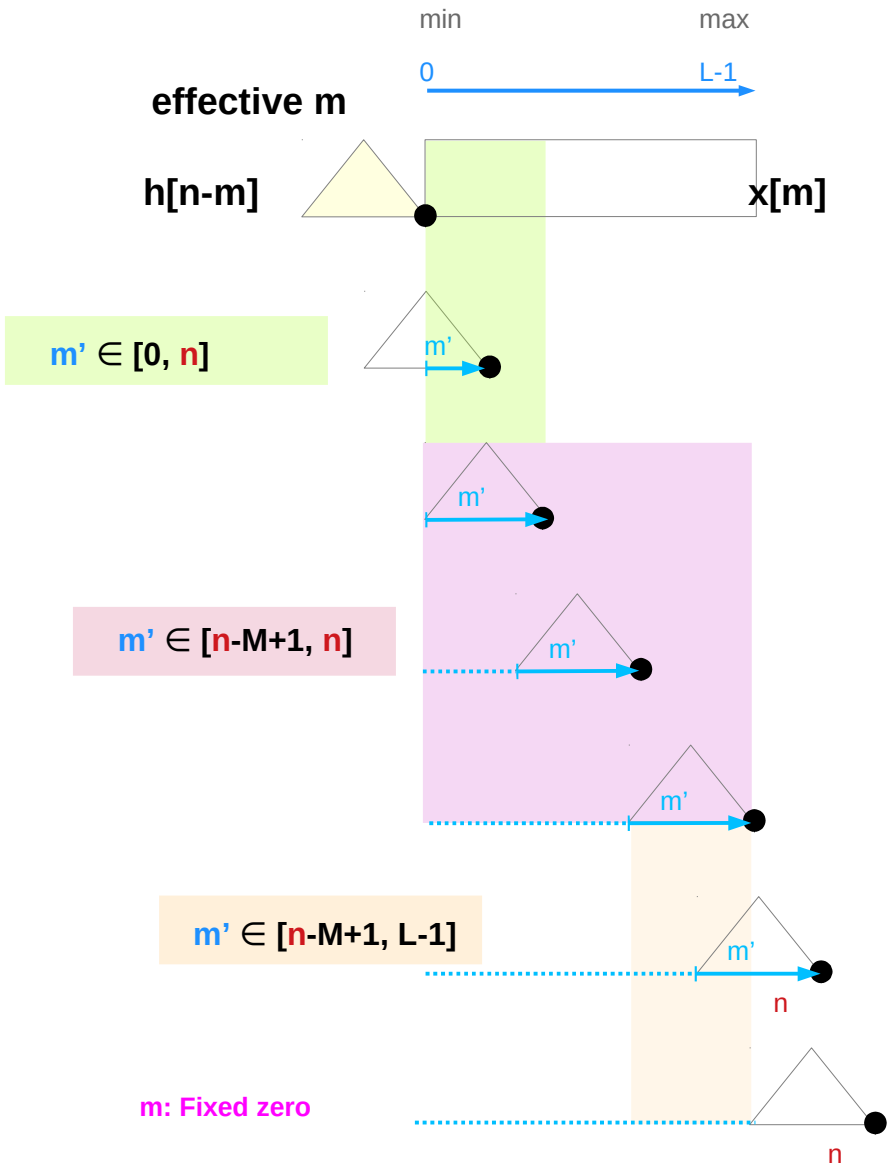
$$\overset{M+L-1}{n} \quad \in [0, L+M-2]$$
$$m \quad \in [0, M-1]$$
$$n-m \quad \in [0, L-1]$$



Index **n** and **m**

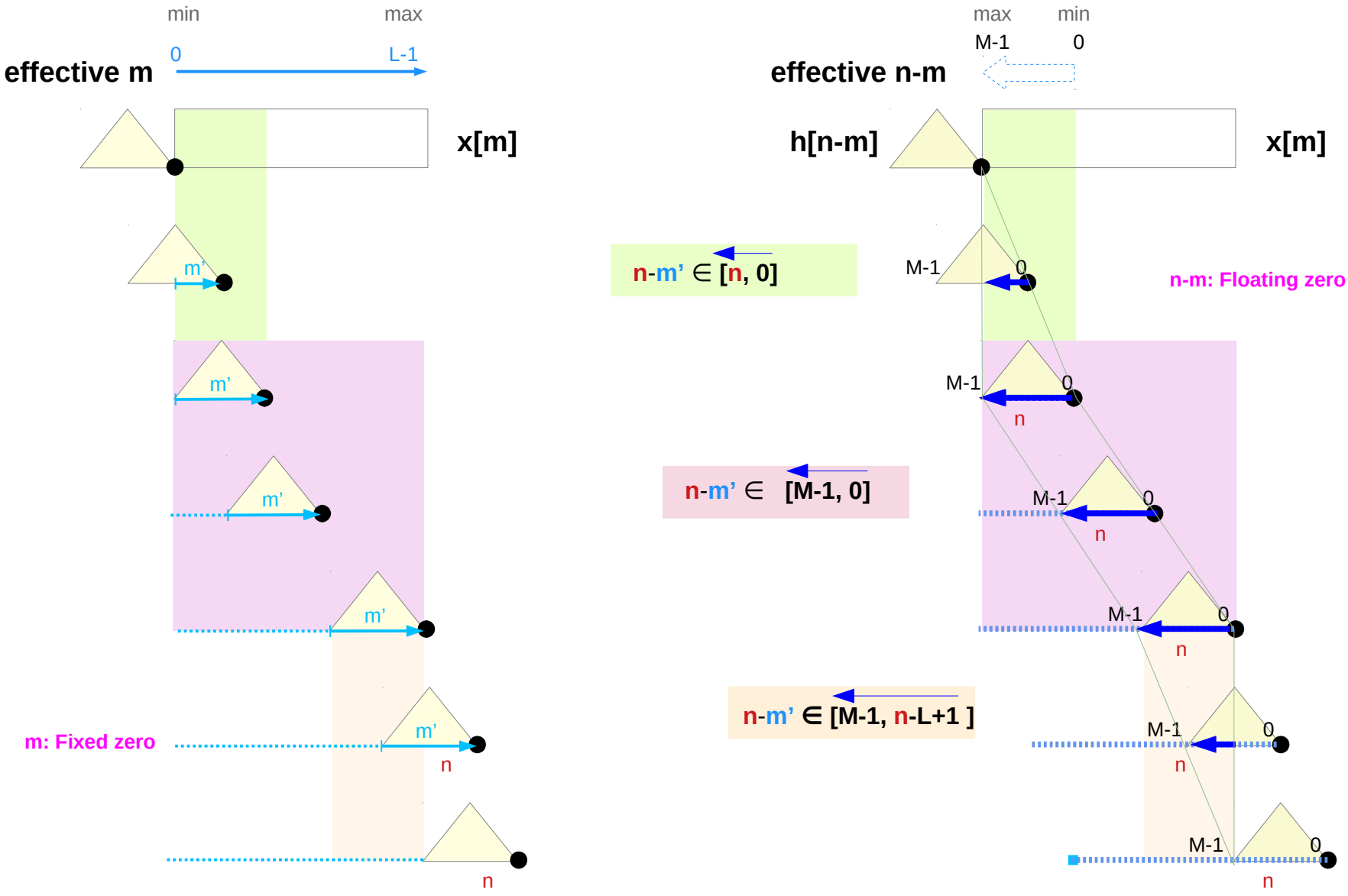
Case A





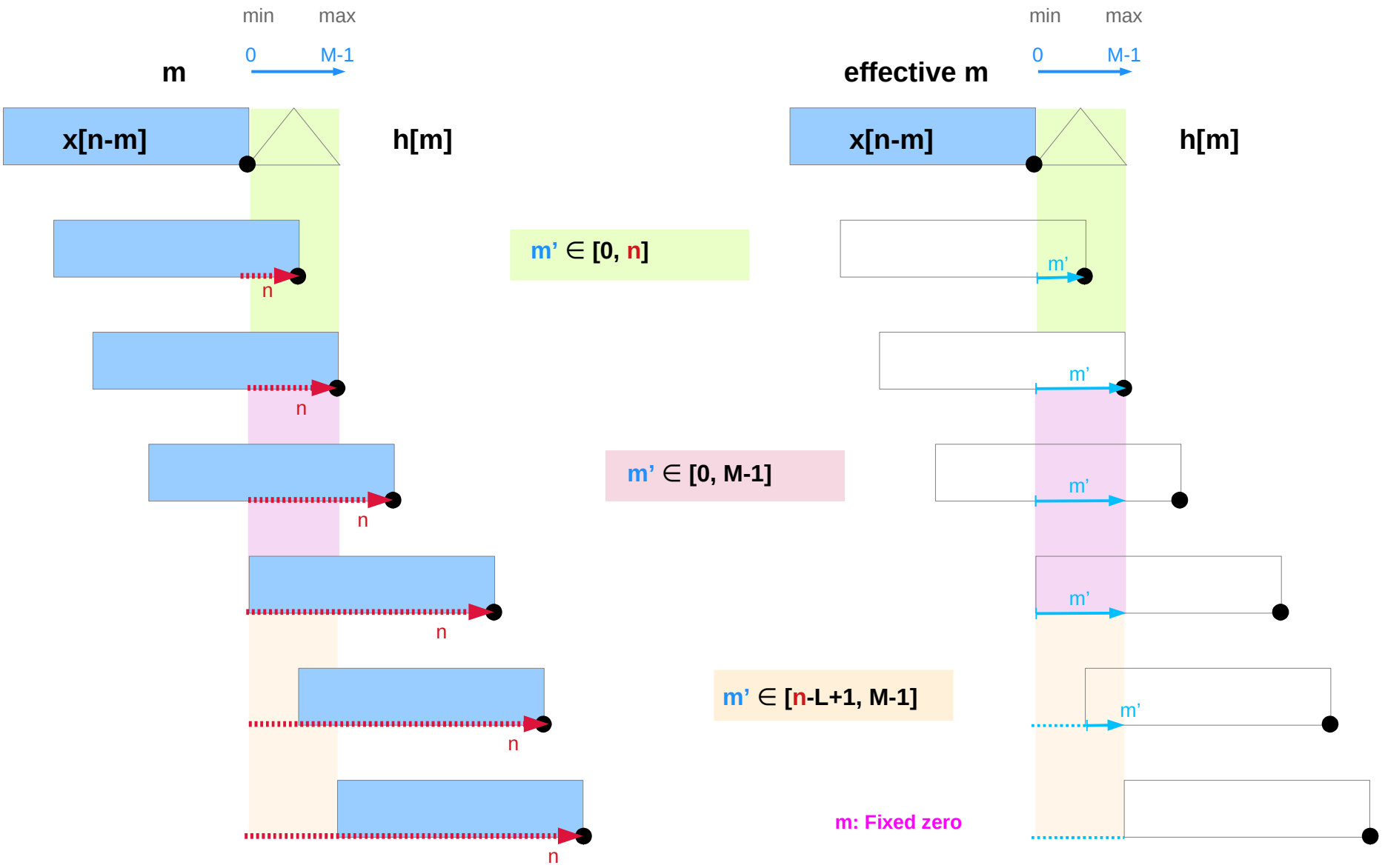
Index **m** and **n-m**

Case A



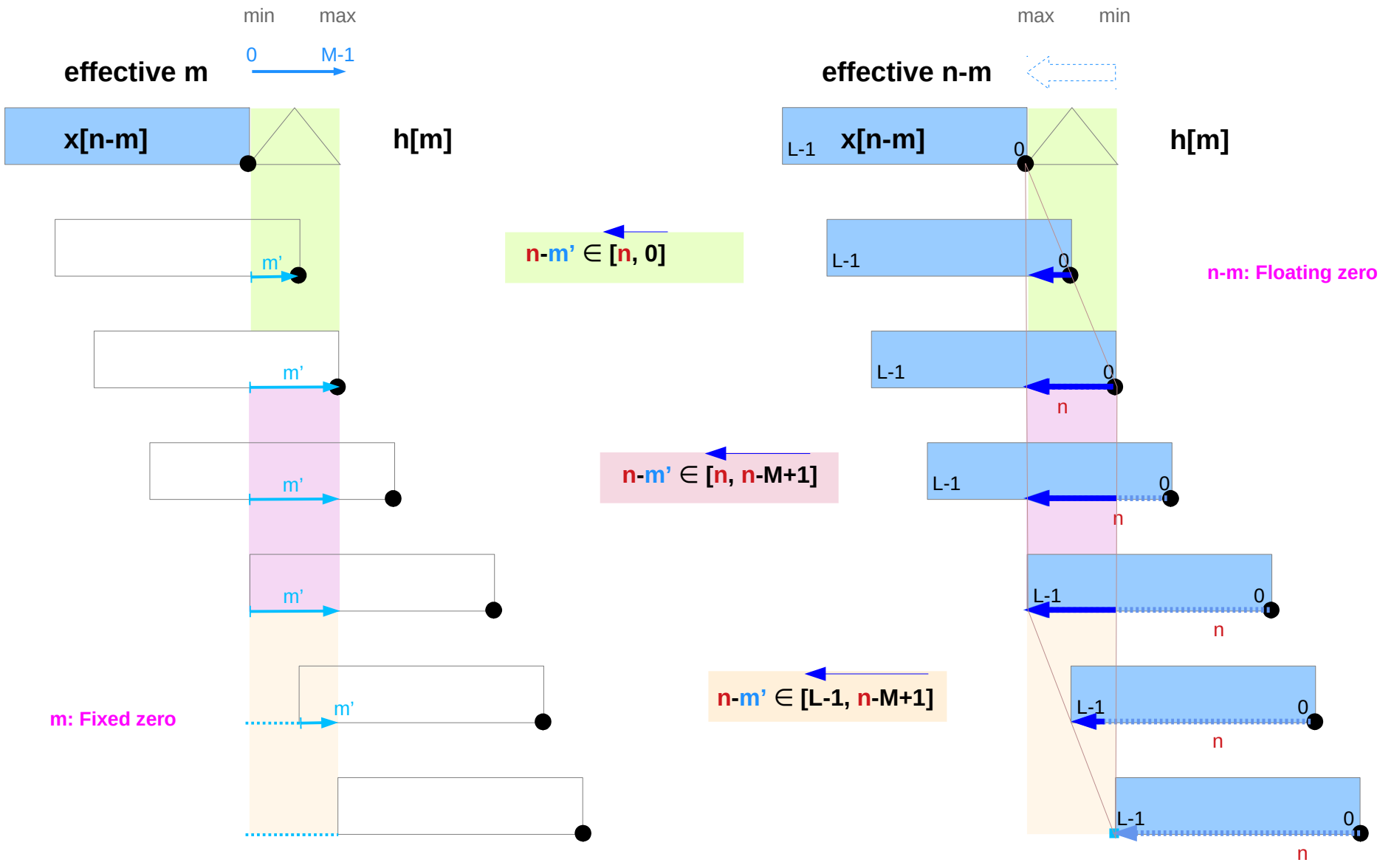
Index **n** and **m**

Case B



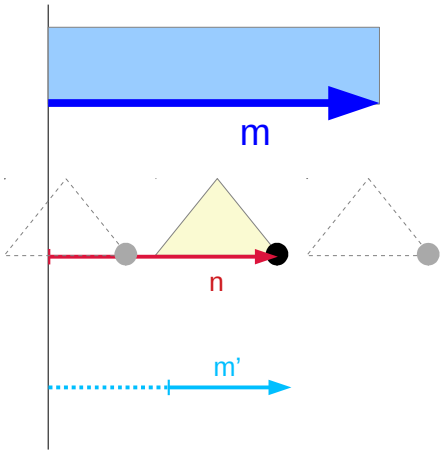
Index **m** and **n-m**

Case B



Summary (1) : effective ranges for **m**

Case A



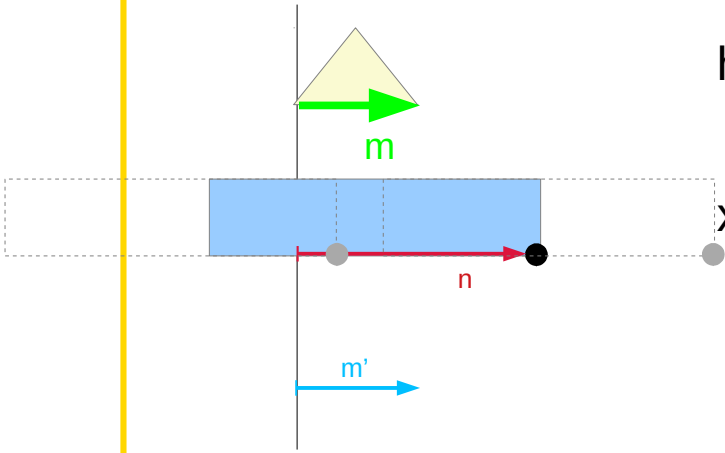
x[m]

h[n-m]

Fixed zero

Part. 1	m' ∈ [0, n]
Part. 2	m' ∈ [n-M+1, n]
Part. 3	m' ∈ [n-M+1, L-1]

Case B



h[m]

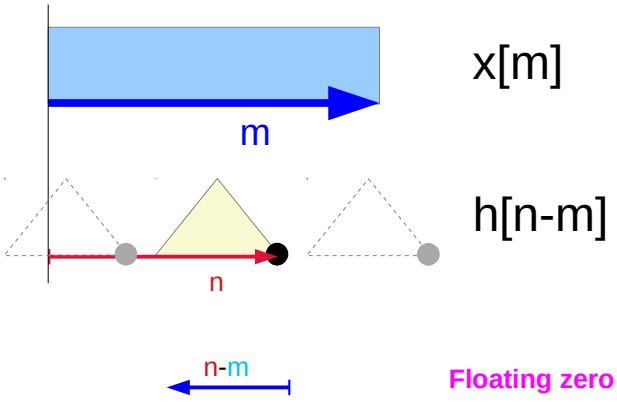
x[n -m]

Fixed zero

Part. 1	m' ∈ [0, n]
Part. 2	m' ∈ [0, M-1]
Part. 3	m' ∈ [n-L+1, M-1]

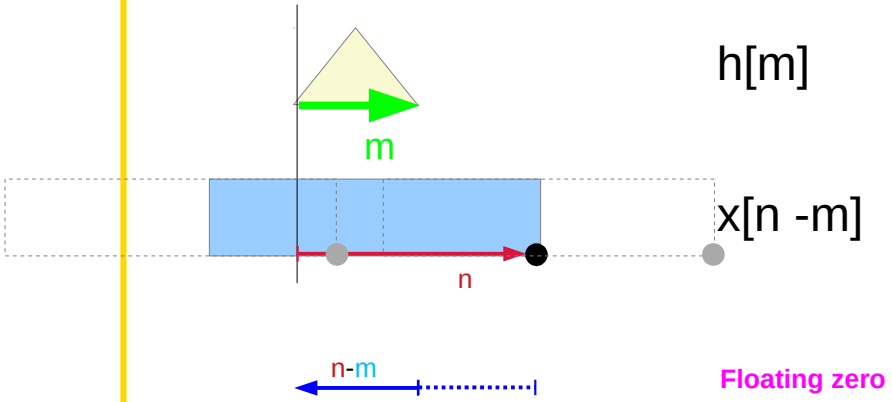
Summary (2) : effective ranges for **n-m**

Case A



Part. 1	n-m' ∈ [n, 0]
Part. 2	n-m' ∈ [M-1, 0]
Part. 3	n-m' ∈ [M-1, n-L+1]

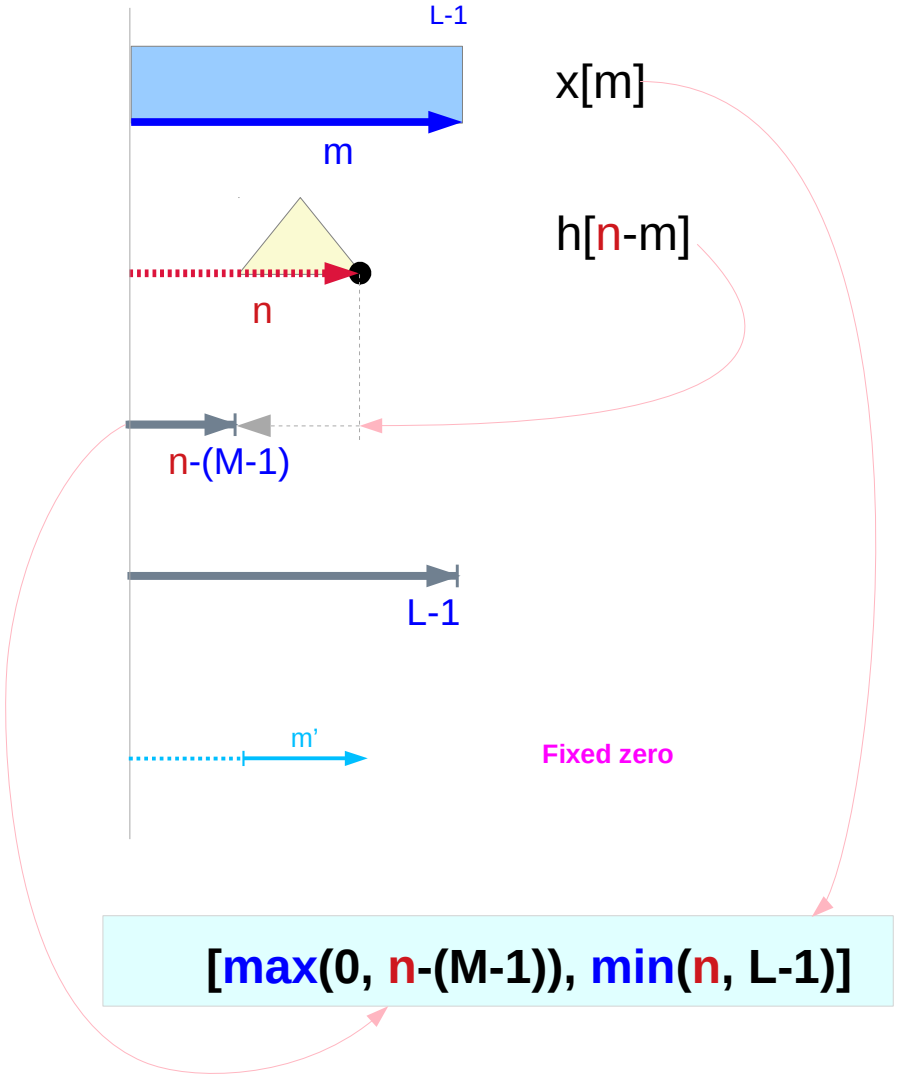
Case B



Part. 1	n-m' ∈ [n, 0]
Part. 2	n-m' ∈ [n, n-M+1]
Part. 3	n-m' ∈ [L-1, n-M+1]

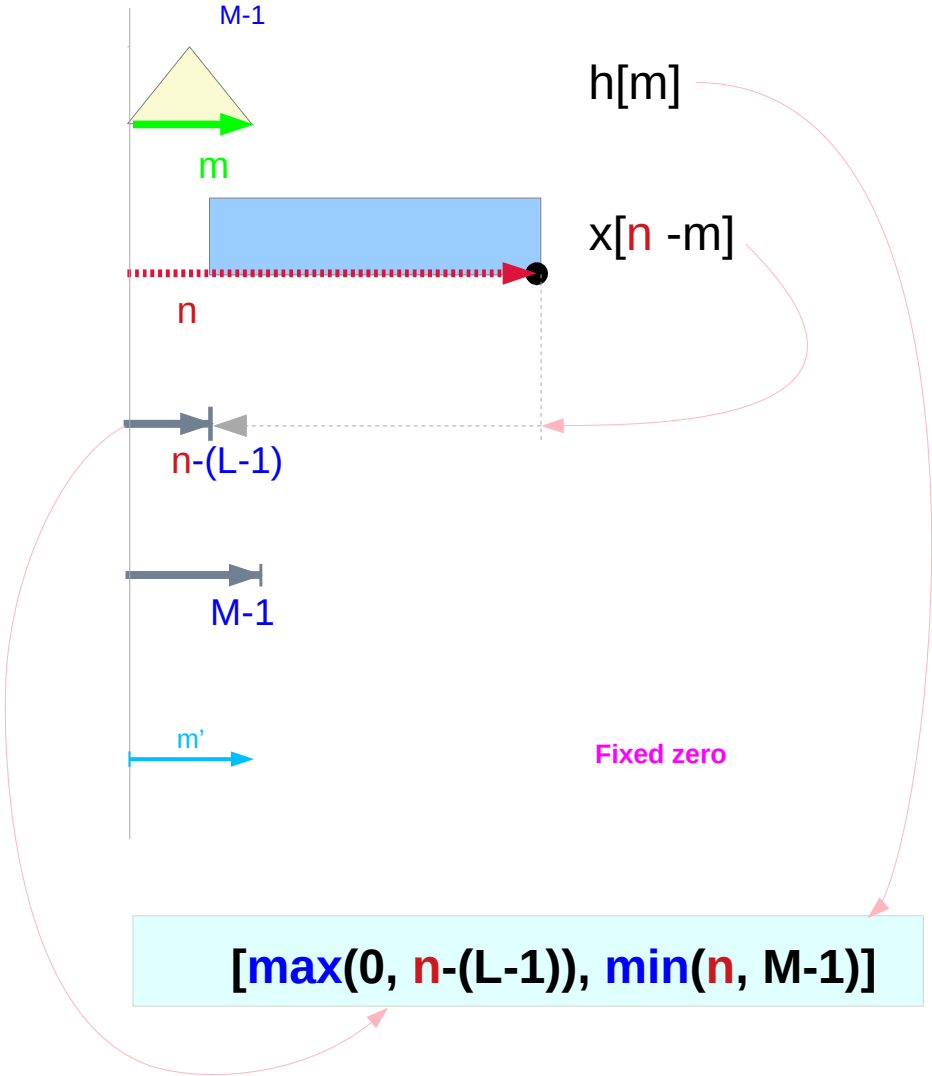
Summary (3) : memorizing effective ranges for **m**

Case A



[max(0, n-(M-1)), min(n, L-1)]

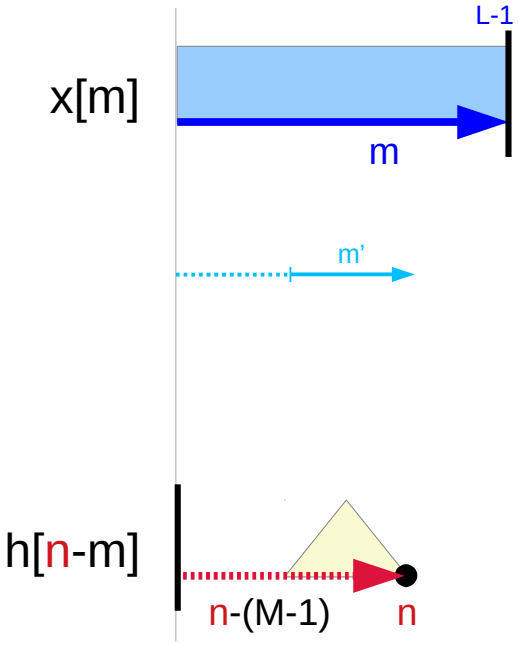
Case B



[max(0, n-(L-1)), min(n, M-1)]

Summary (4) : lower and upper bounds for **m**

Case A



upper bound

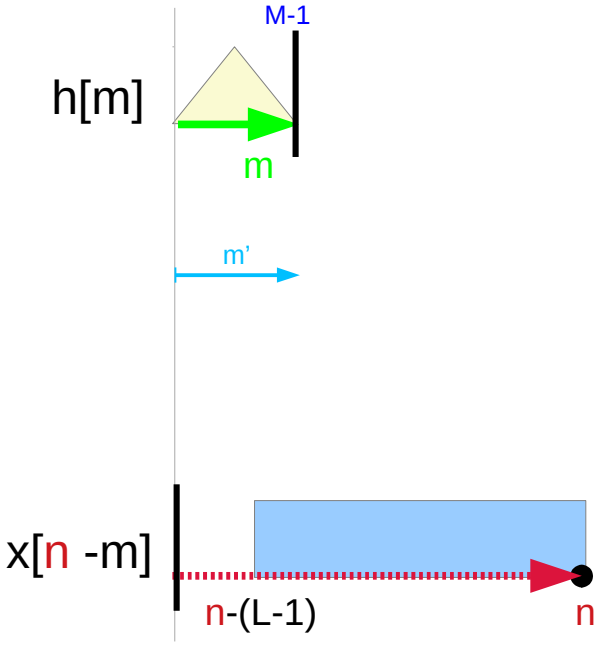
min(n, L-1)

Fixed zero

lower bound

max(0, n-(M-1))

Case B



upper bound

min(n, M-1)

Fixed zero

lower bound

max(0, n-(L-1))

Linear System Theory

Details will be found in

https://en.wikiversity.org/wiki/The_necessities_in_Linear_System_Theory#Time_Domain_System_Analysis_-_Discrete_Time

Convolution (A.pdf, B.pdf)

[max(0, n-(L-1)), min(n, M-1)]

Index n test (1)

```
#include <stdio.h>

#define MIN(a,b) (((a)<(b))?(a):(b))
#define MAX(a,b) (((a)>(b))?(a):(b))

#define L 8
#define M 4


int main(void)
{
  int n, m;


  for (n = 0; n < L+M-1; n++) {
    for (m = MAX(0, n-L+1); m <= MIN(n, M-1); m++)
      printf("n=%d m=%d \n", n, m);
    printf("----------------\n");
  }

}
```

```
n=0 m=0
----------------
n=1 m=0
n=1 m=1
----------------
n=2 m=0
n=2 m=1
n=2 m=2
----------------
n=3 m=0
n=3 m=1
n=3 m=2
n=3 m=3
----------------
n=4 m=0
n=4 m=1
n=4 m=2
n=4 m=3
----------------
n=5 m=0
n=5 m=1
n=5 m=2
n=5 m=3
----------------
n=6 m=0
n=6 m=1
n=6 m=2
n=6 m=3
----------------
n=7 m=0
n=7 m=1
n=7 m=2
n=7 m=3
----------------
n=8 m=1
n=8 m=2
n=8 m=3
----------------
n=9 m=2
n=9 m=3
----------------
n=10 m=3
----------------
```

Index n test (2)

```
#include <stdio.h>

#define MIN(a,b) (((a)<(b))?(a):(b))
#define MAX(a,b) (((a)>(b))?(a):(b))

#define L 8
#define M 4


int main(void)
{
  int n, m;


  for (n = 0; n < L+M; n++) {
    for (m = MAX(0, n-L+1); m <= MIN(n, M); m++)
      printf("n=%d m=%d \n", n, m);
    printf("----------------\n");
  }

}
```

```
n=0 m=0
----------------
n=1 m=0
n=1 m=1
----------------
n=2 m=0
n=2 m=1
n=2 m=2
----------------
n=3 m=0
n=3 m=1
n=3 m=2
n=3 m=3
----------------
n=4 m=0
n=4 m=1
n=4 m=2
n=4 m=3
n=4 m=4
----------------
n=5 m=0
n=5 m=1
n=5 m=2
n=5 m=3
n=5 m=4
----------------
```

```
n=6 m=0
n=6 m=1
n=6 m=2
n=6 m=3
n=6 m=4
----------------
n=7 m=0
n=7 m=1
n=7 m=2
n=7 m=3
n=7 m=4
----------------
n=8 m=1
n=8 m=2
n=8 m=3
n=8 m=4
----------------
n=9 m=2
n=9 m=3
n=9 m=4
----------------
n=10 m=3
n=10 m=4
----------------
n=11 m=4
----------------
```

FIR Filter

A **causal FIR** filter of **order M**

With impulse response $h(n),\ n = 0, 1, \ldots, M$

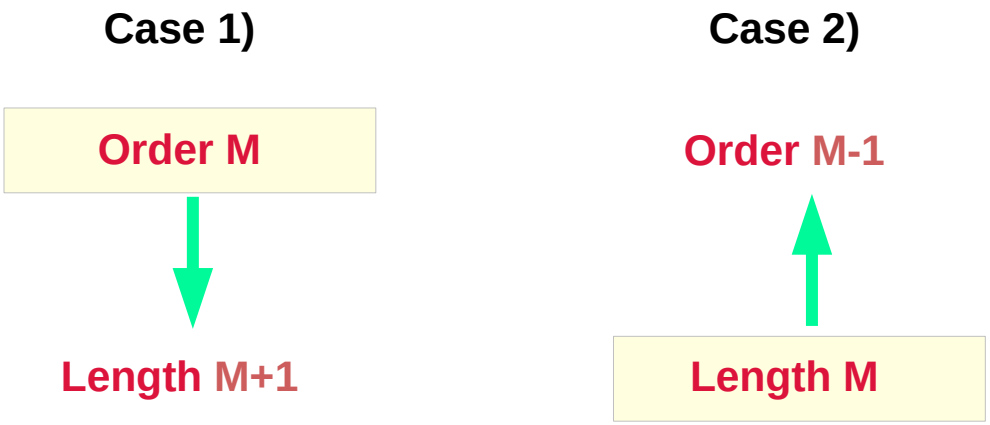
$$\boldsymbol{h} = [h_{0,} h_{1,} \ldots, h_M]$$

filter order M
filter length M+1

Two cases for M

Case 1) **M** represents **filter order** **length = M+1**

Case 2) **M** represents **filter length** **order = M-1**



From **conv_odr** to **conv_len**

Case 1) **M** represents **filter order** $\boldsymbol{h} = [h_0, h_1, \ldots, h_M]$ **length = M+1**

Case 2) **M** represents **filter length** $\boldsymbol{h} = [h_0, h_1, \ldots, h_{M-1}]$ **order = M-1**

Case 1)

```
void conv_odr(int M, double *h, int L, double *x, double *y)
{
      for (n = 0; n < L+M; n++)
      …. M …
}
```

Order M

(int M, doub

M-1 M-1 for **conv_len**

M : **filter order argument**

M+1 : filter length

in **conv_odr**
M : order

in **conv_len**
M : length

M-1 : order

From **conv_len** to **conv_ord**

Case 1) **M** represents **filter order** $\boldsymbol{h} = [h_0, h_1, \ldots, h_M]$ **length = M+1**

Case 2) **M** represents **filter length** $\boldsymbol{h} = [h_0, h_1, \ldots, h_{M-1}]$ **order = M-1**

in **conv_odr**
M : order

in **conv_len**
M : length

M-1 : order

M M for **conv_odr**

```
void conv_len(int M, double *h, int L, double *x, double *y)
{
      for (n = 0; n < L+M-1; n++)
      …. M-1 …
}
```

M : **filter length argument**
M-1 : filter order

Two version of **conv** : **conv_odr** & **conv_len**

Case 1) **M** represents **filter order** $\boldsymbol{h} = [h_0, h_1, \ldots, h_M]$ **length = M+1**

Case 2) **M** represents **filter length** $\boldsymbol{h} = [h_0, h_1, \ldots, h_{M-1}]$ **order = M-1**

Case 1)

```
void conv_odr(int M, double *h, int L, double *x, double *y)
{
      for (n = 0; n < L+M; n++)
      …. M …
}
```

M : **filter order argument**
M+1 : filter length
// L+(M+1)-1 = L+M

Case 2)

```
void conv_len(int M, double *h, int L, double *x, double *y)
{
      for (n = 0; n < L+M-1; n++)
      …. M-1 …
}
```

M : **filter length argument**
M-1 : filter order
// L+M-1

Calling examples of **conv_odr** & **conv_len**

Assume M : filter order

```
double *h, *x, *y;
h = (double *) calloc(M+1,    sizeof(double));    // (M+1)–dimensional
x = (double *) calloc(L,      sizeof(double));    // L–dimensional
y = (double *) calloc(L+M,    sizeof(double));    // (L+M)–dimensional
```

```
conv_odr(M, h, L, x, y);
```

M : filter order argument

```
conv_len(M+1, h, L, x, y);
```

(M+1) : filter length argument

Assume M : filter length

```
double *h, *x, *y;
h = (double *) calloc(M,      sizeof(double));    // M–dimensional
x = (double *) calloc(L,      sizeof(double));    // L–dimensional
y = (double *) calloc(L+M-1,  sizeof(double));    // (L+M-1)–dimensional
```

```
conv_odr(M-1, h, L, x, y);
```

M-1 : filter order argument

```
conv_len(M, h, L, x, y);
```

M : filter length argument

conv_odr

filter order argument

```
#include <stdlib.h>
/* conv_odr.c - convolution of x[n] with h[n], resulting in y[n] */
/* h : filter array, M : filter order (M+1 : filter length) */
/* x : input array, L : input length */
/* y : output array  with length of L+M */

void conv_odr(int M, double *h, int L, double *x, double *y)
{
      int n, m;
      for (n = 0; n< L+M; n++)
            if (1) {
                  for (y[n] = 0, m = max(0, n-M); m <= min(n, L-1); m++)
                        y[n] += x[m] * h[n-m];
            } else {
                  for (y[n] = 0, m = max(0, n-L+1); m <= min(n, M); m++)
                        y[n] += x[n-m] * h[m];
            }
}
```

commutative but different loop index

conv_len

filter length argument

```
#include <stdlib.h>
/* conv.c - convolution of x[n] with h[n], resulting in y[n] */
/* h : filter array, M : filter length (M-1 : filter order) */
/* x : input array, L : input length */
/* y : output array  with length of L+M-1 */

void conv_len(int M, double *h, int L, double *x, double *y)
{
      int n, m;
      for (n = 0; n< L+M-1; n++)
            if (1) {
                  for (y[n] = 0, m = max(0, n-M+1); m <= min(n, L-1); m++)
                        y[n] += x[m] * h[n-m];
            } else {
                  for (y[n] = 0, m = max(0, n-L+1); m <= min(n, M-1); m++)
                        y[n] += x[n-m] * h[m];
            }
}
```

commutative but different loop index

Using gnuplot in C (1)

```
#include <stdlib.h>
#include <stdio.h>
#define NUM_POINTS 5
#define NUM_COMMANDS 2

int main()
{

    double xvals[NUM_POINTS] = {1.0, 2.0, 3.0, 4.0, 5.0};
    double yvals[NUM_POINTS] = {5.0 ,3.0, 1.0, 3.0, 5.0};



    /*    Opens an interface that one can use to send commands as if they were typing into the
     *    gnuplot command line.  "The -persistent" keeps the plot open even after your
     *    C program terminates.
     */
```

https://stackoverflow.com/questions/3521209/making-c-code-plot-a-graph-automatically

Using gnuplot in C (2)

```
    char * gplotCmd[] = { "set title \"TITLEEEEE\" ", "plot 'plot.dat' with impulses lw 2" };
    FILE * data = fopen("plot.dat", "w");
    FILE * gplotPipe = popen ("gnuplot -persistent", "w");
    int i;

    for (i=0; i < NUM_POINTS; i++)     {
        fprintf(data, "%lf %lf \n", xvals[i], yvals[i]);      // Write the data to a temporary file
    }

    for (i=0; i < NUM_COMMANDS; i++)    {
        fprintf(gplotPipe, "%s \n", gplotCmd[i]);              // Send commands to gnuplot one by one.
    }

    return 0;

}
```

https://stackoverflow.com/questions/3521209/making-c-code-plot-a-graph-automatically

Using gnuplot in C (3)

```
fprintf(gplotPipe, "plot '-' \n");
int i;

for (int i = 0; i < NUM_POINTS; i++)
{
        fprintf(gplotPipe, "%lf %lf\n", xvals[i], yvals[i]);
}

fprintf(gplotPipe, "e");
```

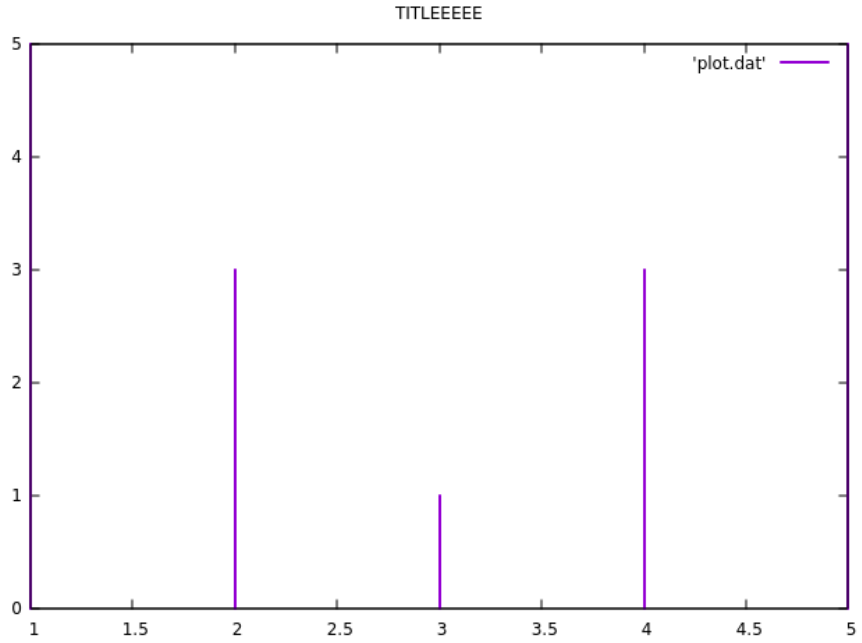
One can avoid having to write to a file by sending gnuplot
the **plot '-'** command followed by data points followed by the letter "**e**".

https://stackoverflow.com/questions/3521209/making-c-code-plot-a-graph-automatically

Using gnuplot in C (4)

gnuplot -persistent | set title \"TITLEEEEE\" plot 'plot.dat' with impulses lw 2

gnuplot -persistent | plot '-'
1.0 5.0
2.0 3.0
3.0 1.0
4.0 3.0
5.0 5.0
e



https://stackoverflow.com/questions/3521209/making-c-code-plot-a-graph-automatically

popen

#include <stdio.h>

FILE ***popen**(const char *command, const char *type);

int pclose(FILE *stream);

The **popen**() function opens a process by creating a pipe, forking, and invoking the shell. Since a pipe is by definition unidirectional, the type argument may specify only reading or writing, not both; the resulting stream is correspondingly read-only or write-only.

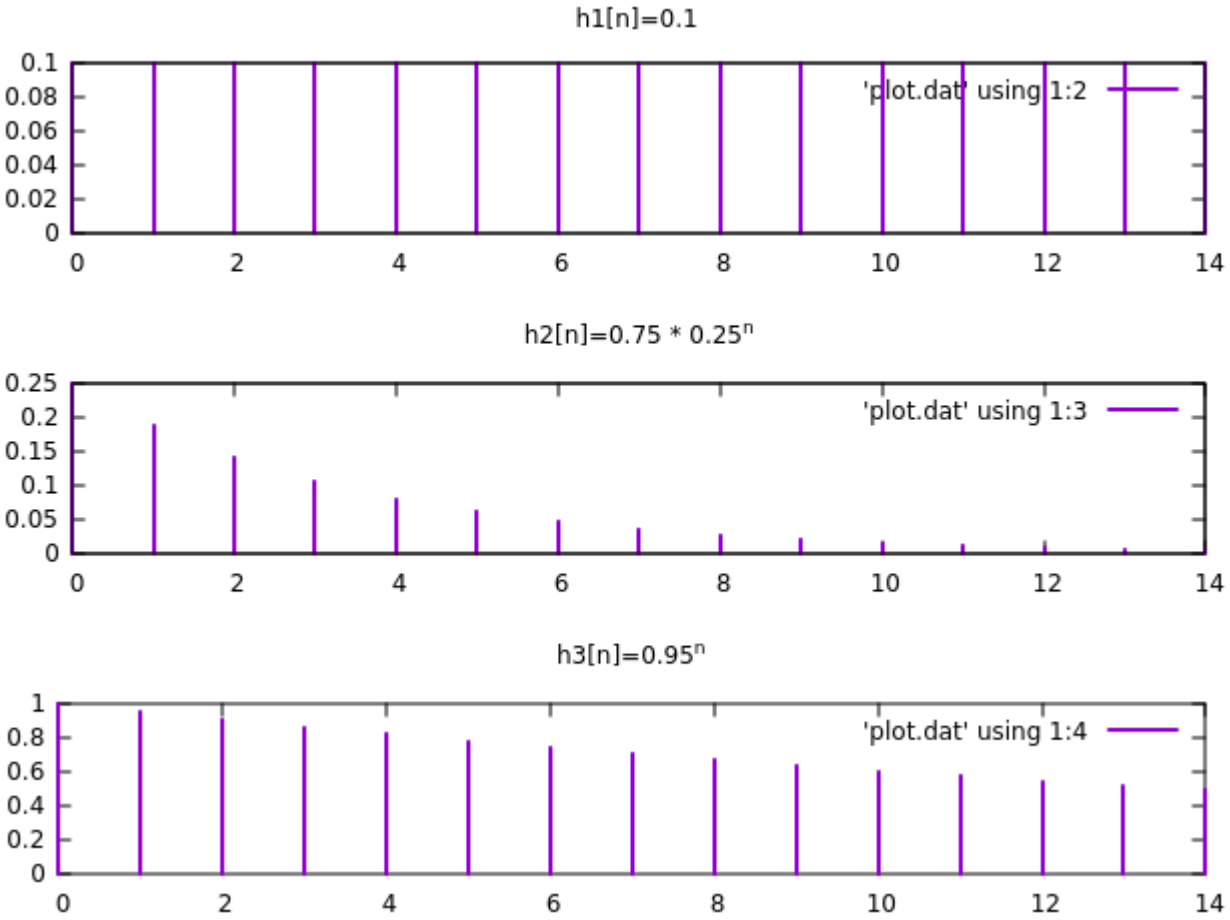
The command argument is a pointer to a null-terminated string containing a shell command line. This command is passed to /bin/sh using the -c flag; interpretation, if any, is performed by the shell.

The type argument is a pointer to a null-terminated string which must contain either the letter 'r' for reading or the letter 'w' for writing.

https://stackoverflow.com/questions/3521209/making-c-code-plot-a-graph-automatically

Impulse Response Examples

```
for (n=0; n<=14; n++)
   h1[n] = G = 0.1;

for (n=0; n<=14; n++)
   h2[n] = (1-.75)*.75^n;

for (n=0; n<=24; n++)
   h2[n] = .95^n;
```



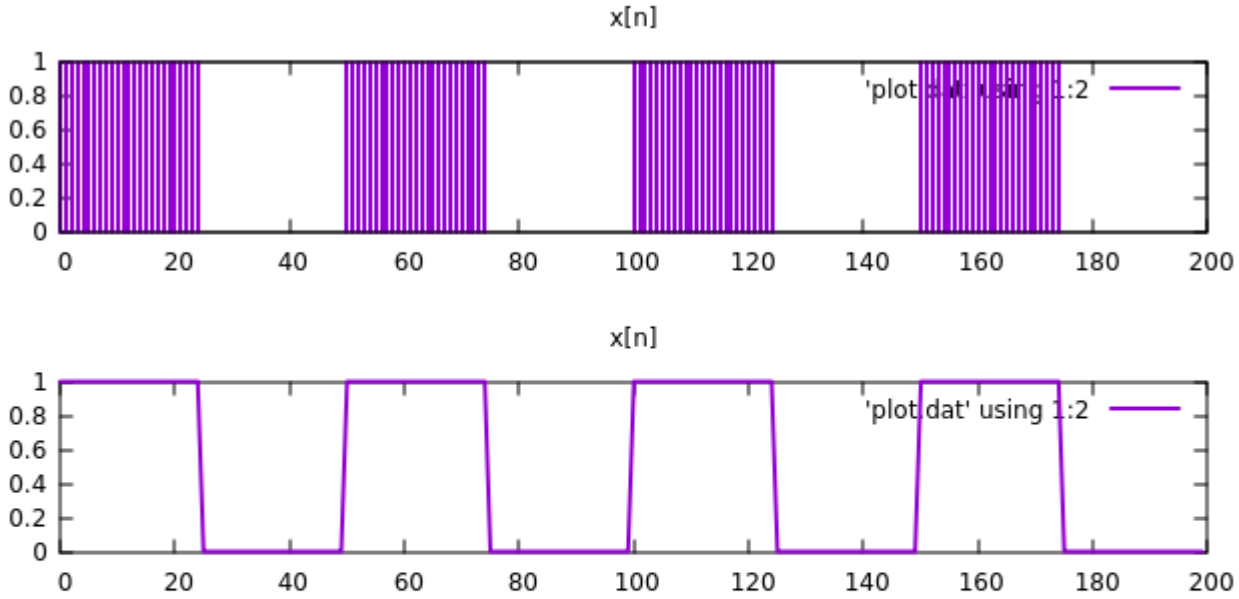
Impulse Response Examples

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#define M 15

int main() {
  FILE * data = fopen("plot.dat", "w");
  FILE * gplotPipe = popen ("gnuplot -persistent", "w");

  int n;
  double h1[M], h2[M], h3[M];

  for (n=0; n<=14; n++) {
    h1[n] = 0.1;
    h2[n] = 0.25 * pow(0.75, n);
    h3[n] = pow(0.95, n);
  }


  for (n=0; n < M; n++)     {
    fprintf(data, "%d %lf %lf %lf\n", n, h1[n], h2[n], h3[n]);
  }
```

```
  fprintf(gplotPipe, "set multiplot layout 3,1\n");

  fprintf(gplotPipe, "set title \"h1[n]=0.1\"\n");
  fprintf(gplotPipe, "plot 'plot.dat' using 1:2 ");
  fprintf(gplotPipe, "with impulses lw 2 \n" );

  fprintf(gplotPipe, "set title \"h2[n]=0.75 * 0.25^n\"\n");
  fprintf(gplotPipe, "plot 'plot.dat' using 1:3 ");
  fprintf(gplotPipe, "with impulses lw 2 \n" );

  fprintf(gplotPipe, "set title \"h3[n]=0.95^n\"\n");
  fprintf(gplotPipe, "plot 'plot.dat' using 1:4 ");
  fprintf(gplotPipe, "with impulses lw 2 \n" );


  return 0;

}
```

Square wave input

```
L = 200;
K = 50;

for (n=0; n<L; n++)
   if (n%K < K/2)
     x[n] = 1;
   else
     x[n] = 0;
```



Square wave

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#define L 200
#define K 50

int main() {
  FILE * data = fopen("plot.dat", "w");
  FILE * gplotPipe = popen ("gnuplot -persistent", "w");

  int n;
  double x[L];

  for (n=0; n<L; n++)
    if (n%K < K/2) x[n] = 1;
    else           x[n] = 0;


  for (n=0; n < L; n++)     {
    fprintf(data, "%d %lf \n", n, x[n]);
  }
```

```
  fprintf(gplotPipe, "set multiplot layout 3,1\n");

  fprintf(gplotPipe, "set title \"x[n]\"\n");
  fprintf(gplotPipe, "plot 'plot.dat' using 1:2 ");
  fprintf(gplotPipe, "with impulses lw 2 \n" );

  fprintf(gplotPipe, "set title \"x[n]\"\n");
  fprintf(gplotPipe, "plot 'plot.dat' using 1:2 ");
  fprintf(gplotPipe, "with lines lw 2 \n" );


  return 0;

}
```

conv.c

```
#define MIN(a,b) (((a)<(b))?(a):(b))
#define MAX(a,b) (((a)>(b))?(a):(b))

// #define M_LEN
// #define FLIP_H

#ifdef M_LEN
//-------------------------------------------------------------
// conv.c - convolution of x[n] with h[n], resulting in y[n]
// h : filter array, M : filter length (M-1 : filter order)
// x : input array, L : input length
// y : output array  with length of L+M-1
//-------------------------------------------------------------
void conv(int M, double *h, int L, double *x, double *y)
{
  int n, m, m1, m2;

  for (n = 0; n< L+M-1; n++) {
#ifdef FLIP_H
      m1 = MAX(0, n-M+1);
      m2 = MIN(n, L-1);
      for (y[n] = 0, m = m1; m <= m2; m++)
        y[n] += x[m] * h[n-m];
#else
      m1 = MAX(0, n-L+1);
      m2 = MIN(n, M-1);
      for (y[n] = 0, m = m1; m <= m2; m++)
        y[n] += x[n-m] * h[m];
#endif
  }
}

#else
//-------------------------------------------------------------
// conv.c - convolution of x[n] with h[n], resulting in y[n]
// h : filter array, M : filter order (M+1 : filter length)
// x : input array, L : input length
// y : output array  with length of L+M
//-------------------------------------------------------------
void conv(int M, double *h, int L, double *x, double *y)
{
  int n, m, m1, m2;

  for (n = 0; n< L+M; n++) {
#ifdef FLIP_H
      m1 = MAX(0, n-M);
      m2 = MIN(n, L-1);
      for (y[n] = 0, m = m1; m <= m2; m++)
        y[n] += x[m] * h[n-m];
#else
      m1 = MAX(0, n-L+1);
      m2 = MIN(n, M);
      for (y[n] = 0, m = m1; m <= m2; m++)
        y[n] += x[n-m] * h[m];
#endif
  }
}

#endif
```

plot_conv.c

```
#include <stdio.h>

int plot_conv(double *h, double *x, double *y, int len) {

  FILE * data = fopen("plot.dat", "w");
  FILE * gplotPipe = popen ("gnuplot -persistent", "w");

  int n;

  fprintf(gplotPipe, "set multiplot layout 3,1\n");

  for (n=0; n < len; n++)
    fprintf(data, "%d %lf %lf %lf \n", n, h[n], x[n], y[n]);
  fclose(data);

  fprintf(gplotPipe, "set title \"h[n]\"\n");
  fprintf(gplotPipe, "plot 'plot.dat' using 1:2");
  fprintf(gplotPipe, "with lines lw 2 \n" );

  fprintf(gplotPipe, "set title \"x[n]\"\n");
  fprintf(gplotPipe, "plot 'plot.dat' using 1:3");
  fprintf(gplotPipe, "with lines lw 2 \n" );

  fprintf(gplotPipe, "set title \"y[n]\"\n");
  fprintf(gplotPipe, "plot 'plot.dat' using 1:4");
  fprintf(gplotPipe, "with lines lw 2 \n" );

  fclose(gplotPipe);

  return 0;
}
```

test_conv.c (1)

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <unistd.h>

#ifdef M_LEN
#define M 15   // filter length
#define TL (L+M-1)
#define STR1 "M: Filter Length"
#else
#define M 14   // filter order
#define TL (L+M)
#define STR1 "M: Filter Order"
#endif

#ifdef FLIP_H
#define STR2 "y[n] = Sum x[n] * h[n-m]"
#else
#define STR2 "y[n] = Sum h[n] * x[n-m]"
#endif

#define L 200  // input  length
#define K 50   // square wave period


void conv(int, double *, int, double *, double *);
void plot_conv(double *, double *, double *, int);
```

```
int main() {

  double *h1 = (double *) calloc(TL, sizeof(double));
  double *h2 = (double *) calloc(TL, sizeof(double));
  double *h3 = (double *) calloc(TL, sizeof(double));
  double *x  = (double *) calloc(TL, sizeof(double));
  double *y  = (double *) calloc(TL, sizeof(double));

  int n;

  printf("%s   %s \n", STR1, STR2);
```

calloc
 memory allocation with zero initialization
 No need for explicit initialization

test_conv.c (2)

```
#ifdef M_LEN
  for (n=0; n<=M-1; n++) {
#else
  for (n=0; n<=M; n++) {
#endif
    h1[n] = 0.1;
    h2[n] = 0.25 * pow(0.75, n);
    h3[n] = pow(0.95, n);
  }

  for (n=0; n<L; n++)
    if (n%K < K/2) x[n] = 1;
    else           x[n] = 0;


  conv(M, h1, L, x, y);
  plot_conv(h1, x, y, TL);

  conv(M, h2, L, x, y);
  plot_conv(h2, x, y, TL);

  conv(M, h3, L, x, y);
  plot_conv(h3, x, y, TL);

  return 0;

}
```

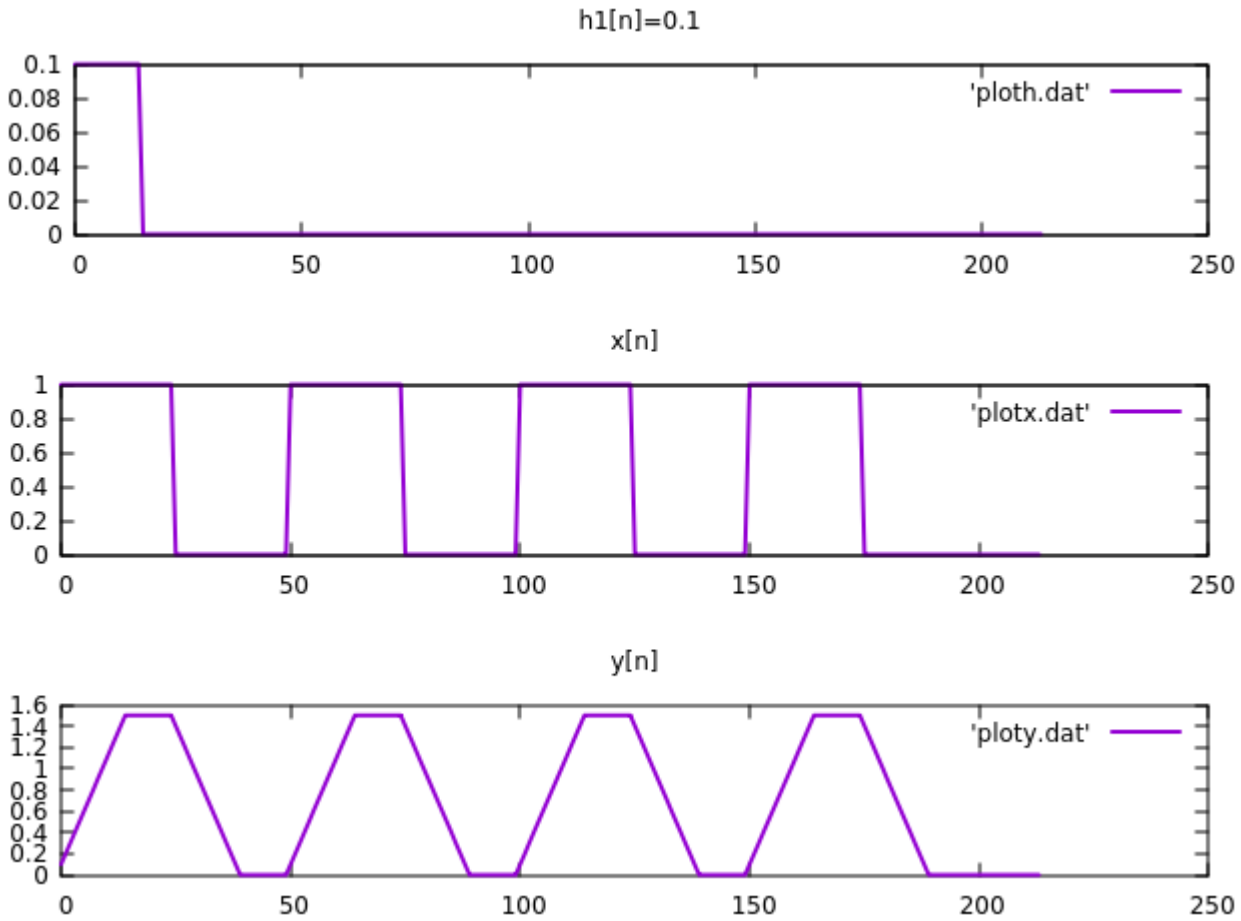
h1, h2, h3, x, y
dynamically allocated array with the length TL
all elements are initialized with **zero** by using calloc

```
#ifdef M_LEN
#define TL (L+M-1)
#else
#define TL (L+M)
#endif
```

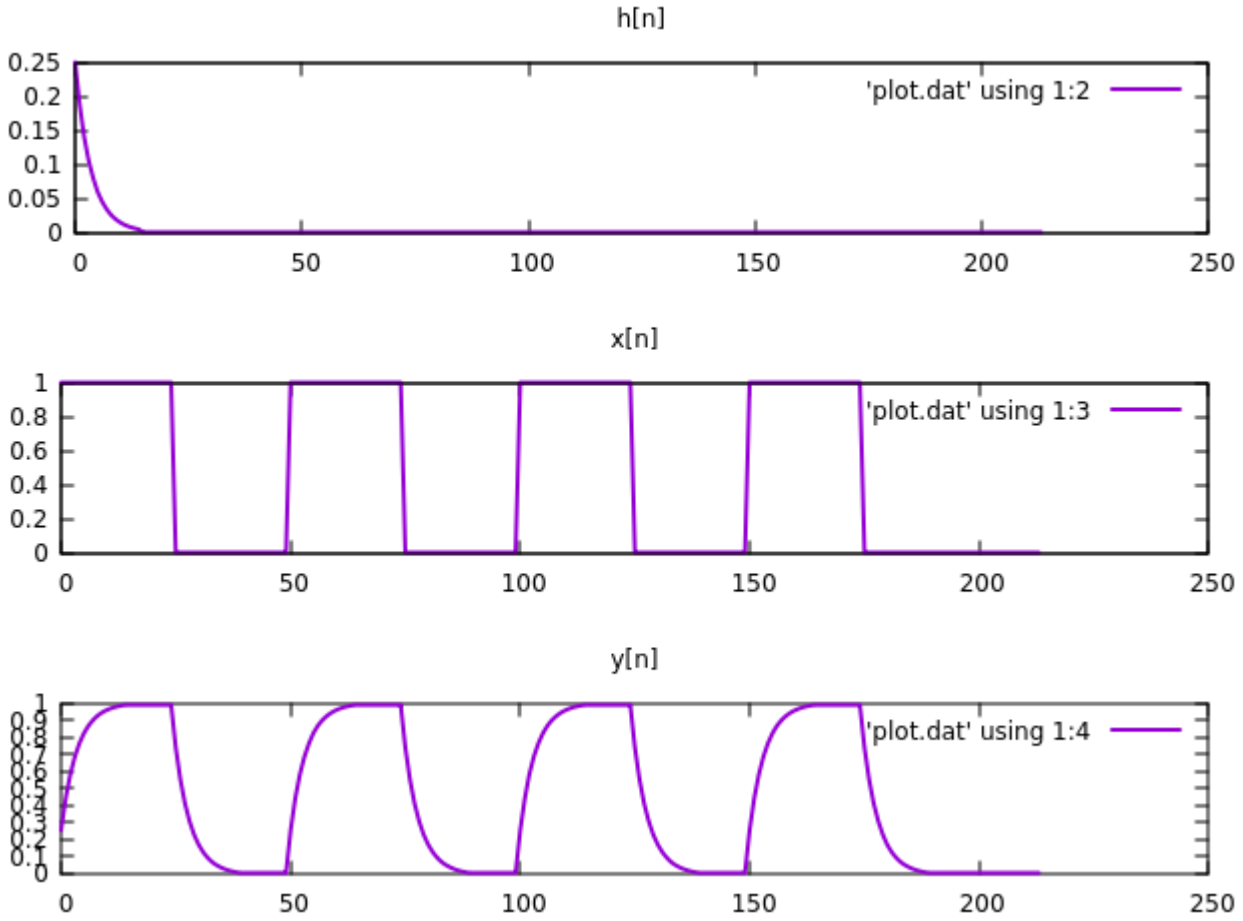
run script

```
#!/bin/bash


set -x

gcc                             conv.c plot_conv.c test_conv.c  -o test1 -lm
gcc                  -DFLIP_H conv.c plot_conv.c test_conv.c  -o test2 -lm
gcc -DM_LEN                     conv.c plot_conv.c test_conv.c  -o test3 -lm
gcc -DM_LEN -DFLIP_H conv.c plot_conv.c test_conv.c  -o test4 -lm

./test1 && cp plot.dat plot1.dat
./test2 && cp plot.dat plot2.dat
./test3 && cp plot.dat plot3.dat
./test4 && cp plot.dat plot4.dat

diff plot1.dat plot2.dat
diff plot1.dat plot3.dat
diff plot1.dat plot4.dat
```

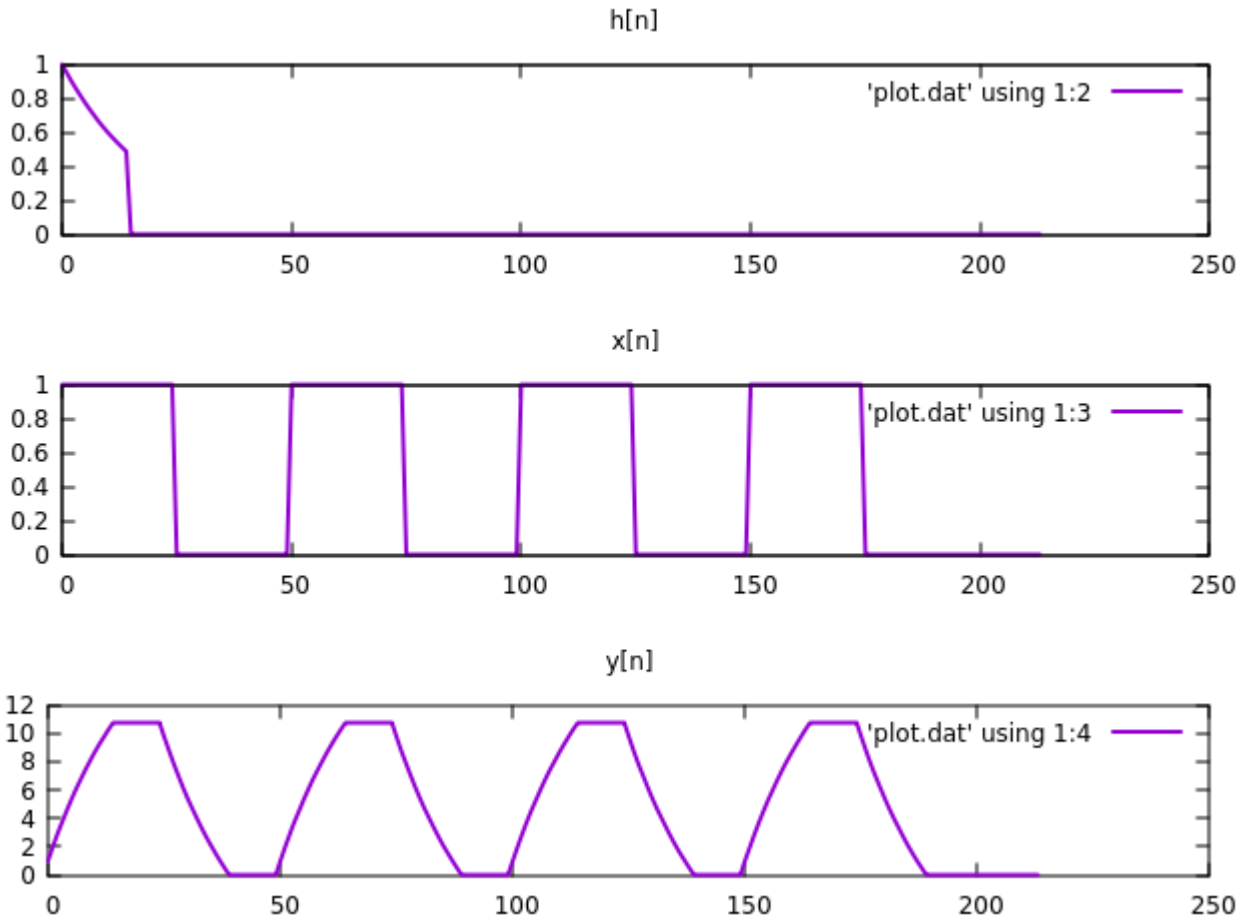
h1[n] = 0.1 (n=0, …, 14)


h1[n]=0.1
'ploth.dat'
0.1
0.08
0.06
0.04
0.02
0
0
50
100
150
200
250
x[n]
'plotx.dat'
1
0.8
0.6
0.4
0.2
0
y[n]
'ploty.dat'
1.6
1.4
1.2

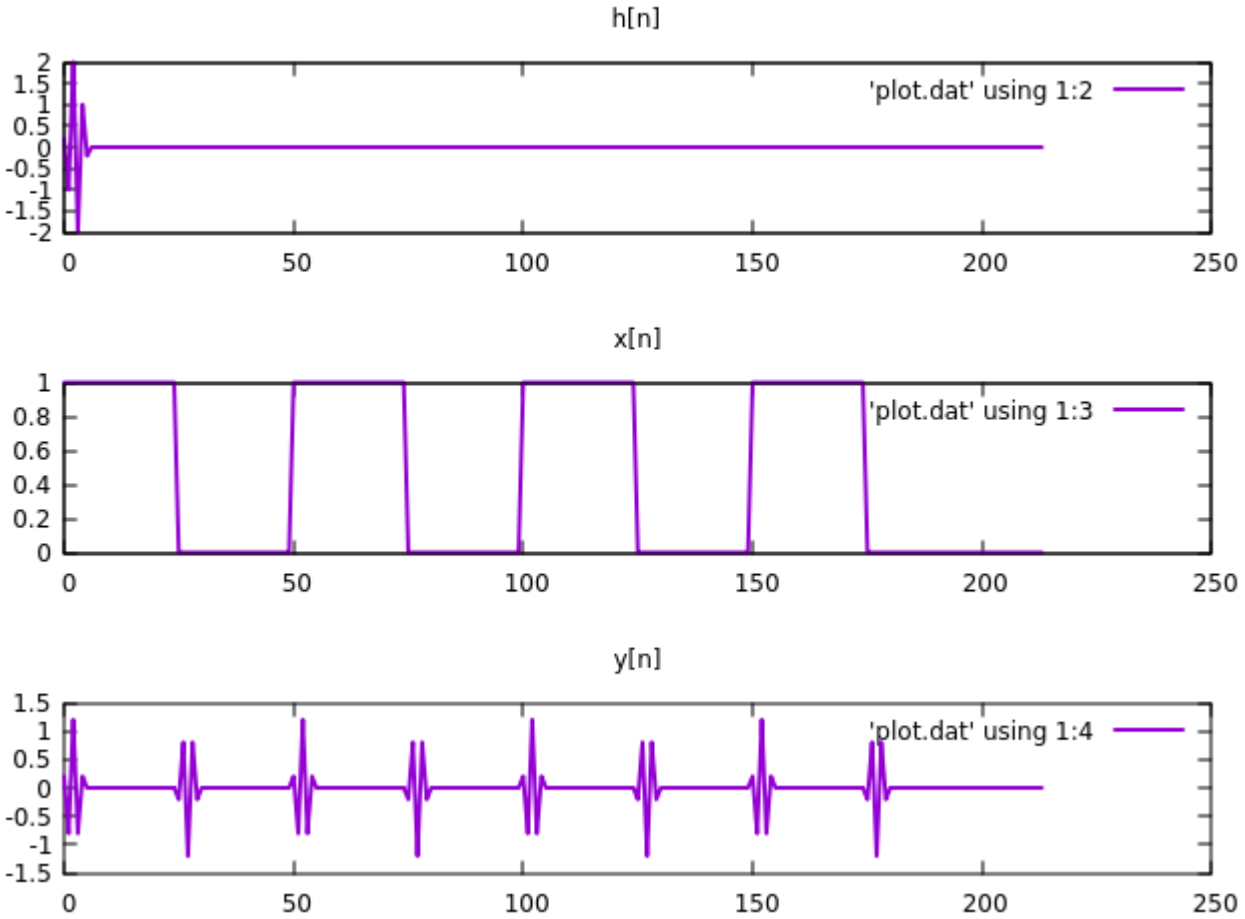
h2[n] = 0.25 * 0.75^n (n=0, …, 14)



h3[n] = 0.95^n (n=0, …, 14)



h4[n] = [0.2, -1, 2, -2, 1, -0.2]

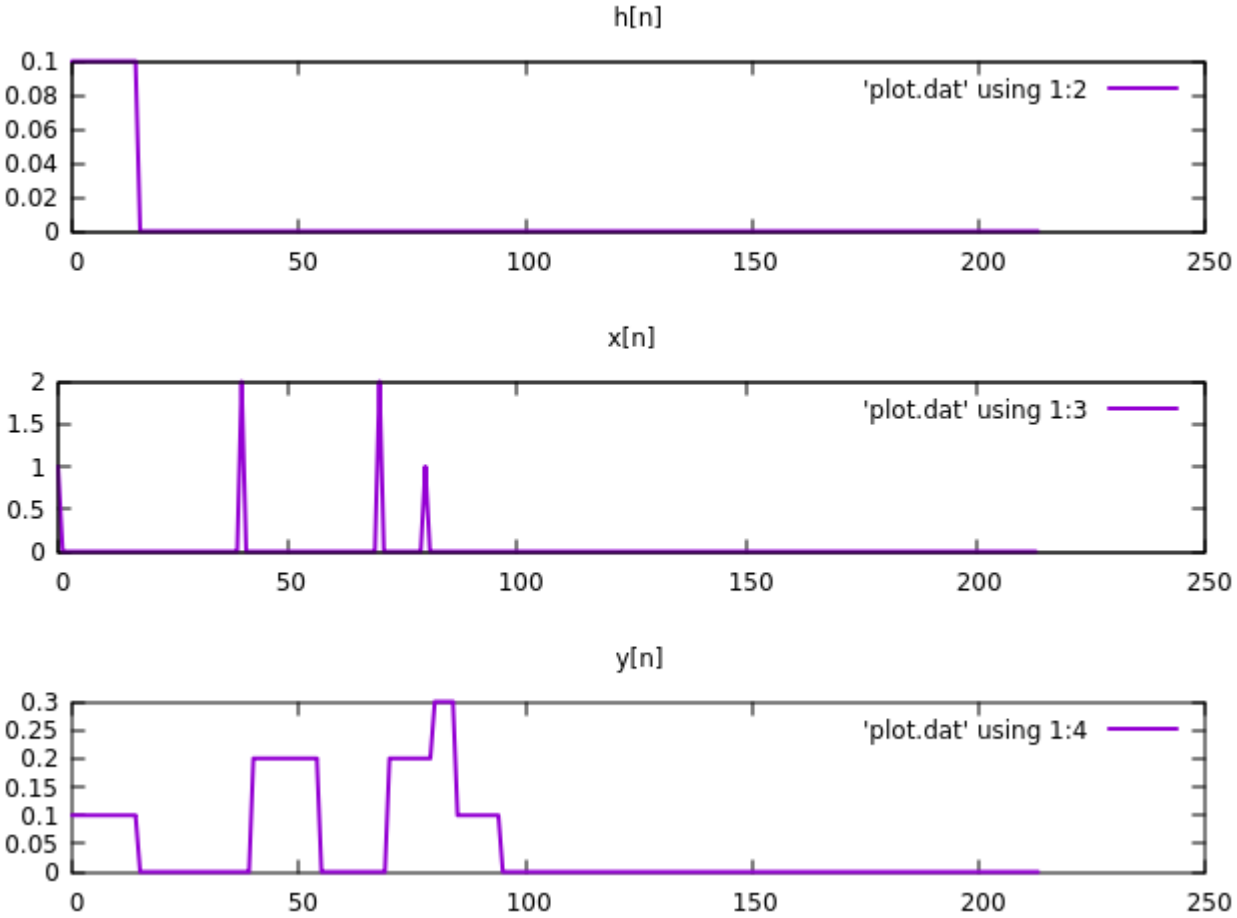


h1[n], x[n] = δ(n) + 2δ(n-40) + 2δ(n-70) + δ(n-80)

```
/* delta.c - delta function */
double delta(int n) {
  if (n == 0)
     return 1;
  else
     return 0;
}
```

```
for (n=0; n<=120; n++)
   x[n] = delta(n)
          + 2*delta(n-40)
          + 2*delta(n-70)
          + delta(n-80);
```

```
conv(24, h, 121, x, y)
```

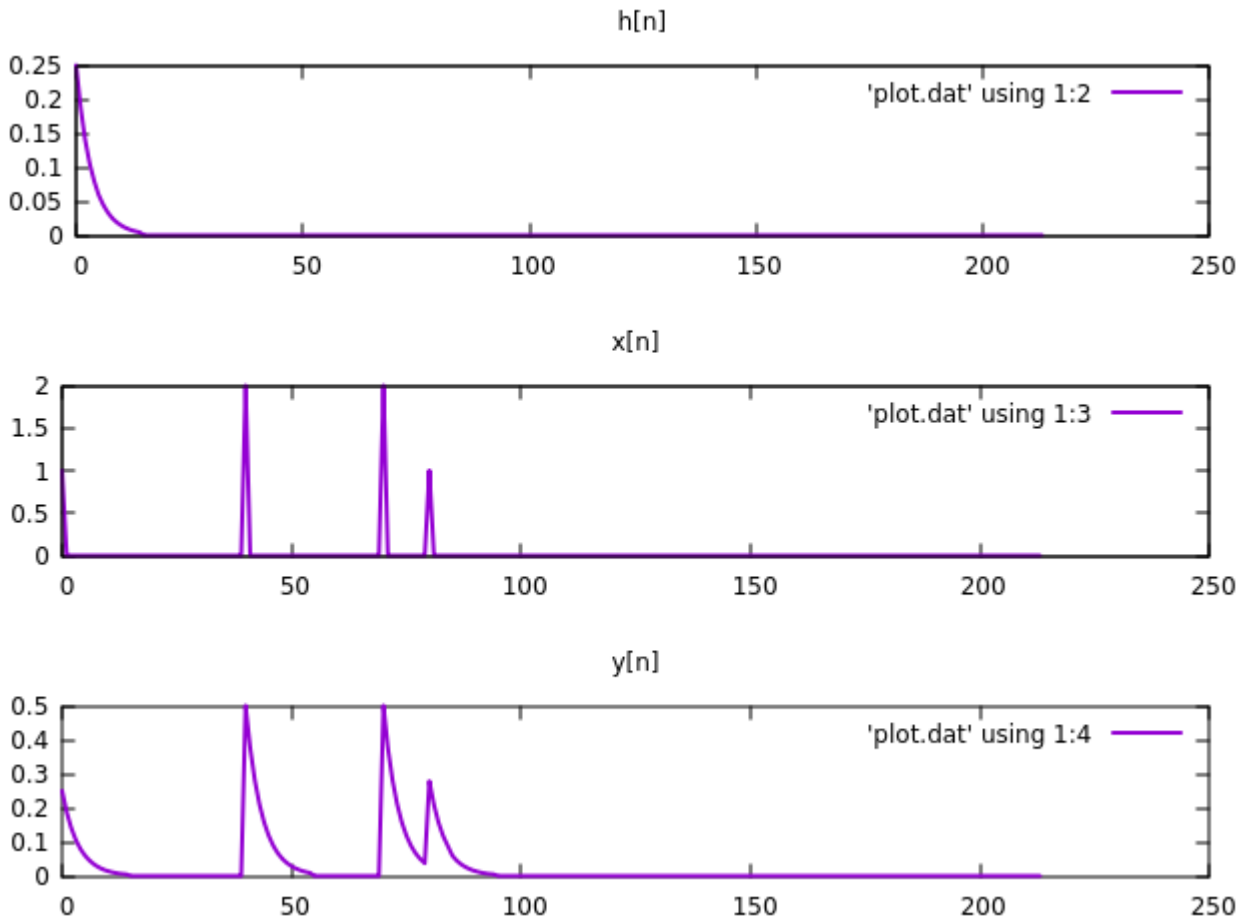


h2[n], x[n] = δ(n) + 2δ(n-40) + 2δ(n-70) + δ(n-80)

```
/* delta.c - delta function */
double delta(int n) {
  if (n == 0)
     return 1;
  else
     return 0;
}
```

```
for (n=0; n<=120; n++)
   x[n] = delta(n)
          + 2*delta(n-40)
          + 2*delta(n-70)
          + delta(n-80);
```

```
conv(24, h, 121, x, y)
```

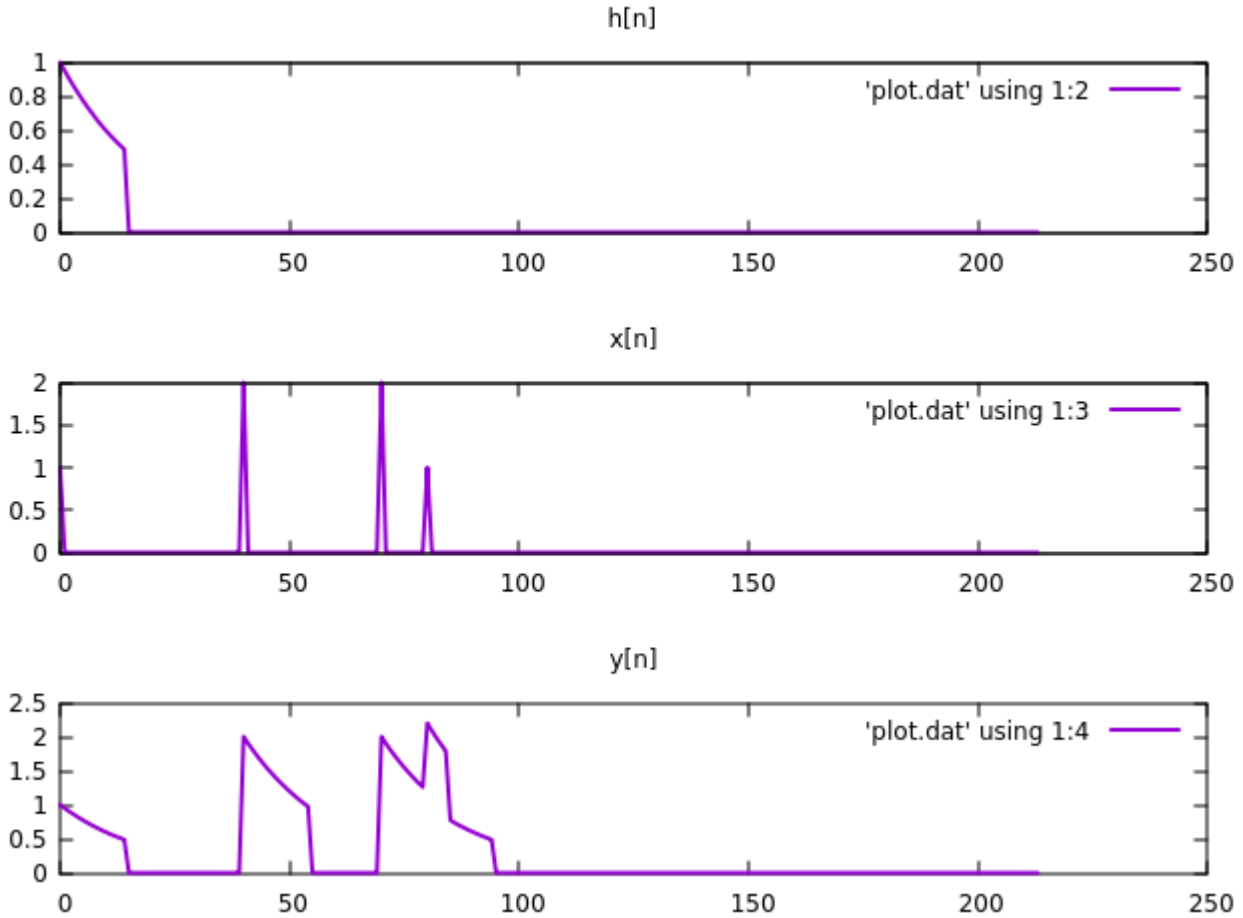


h3[n], x[n] = δ(n) + 2δ(n-40) + 2δ(n-70) + δ(n-80)

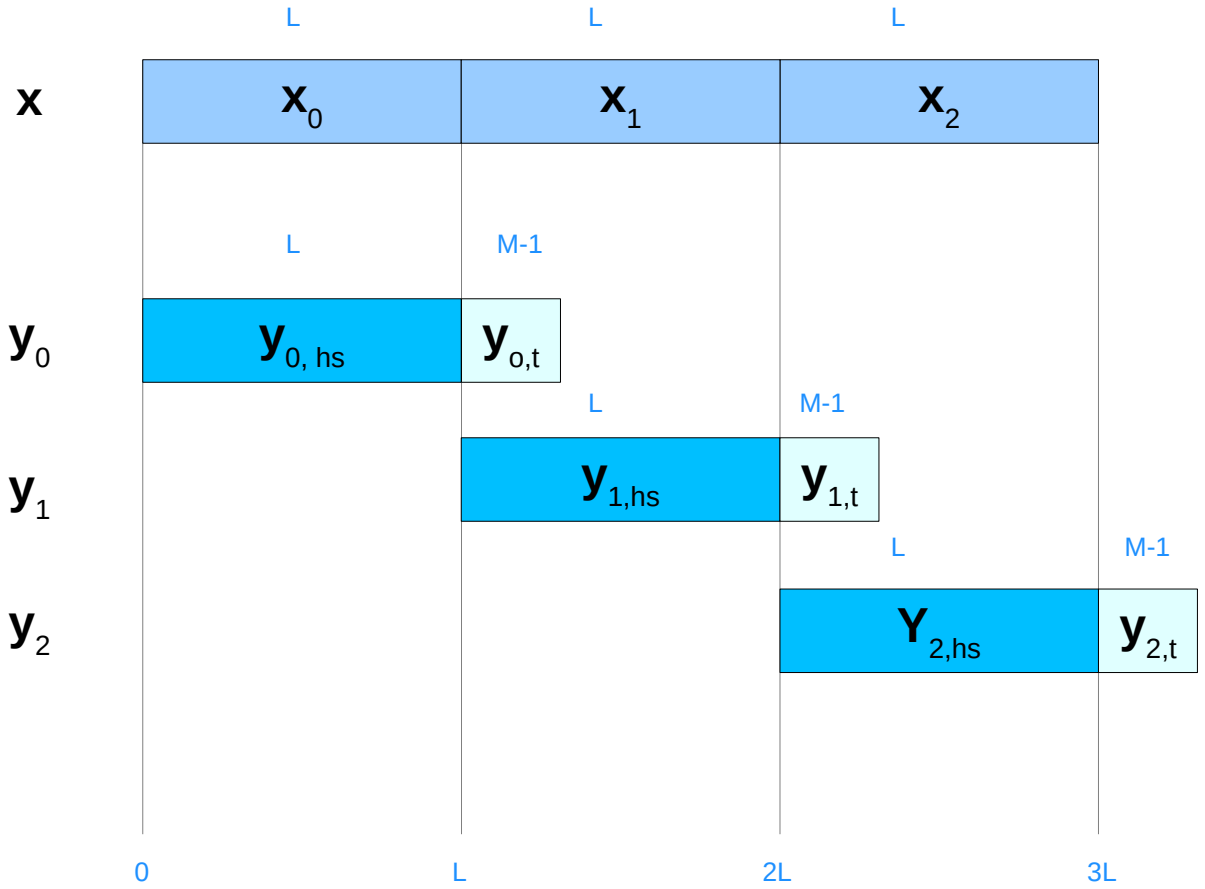
```
/* delta.c - delta function */
double delta(int n) {
  if (n == 0)
     return 1;
  else
     return 0;
}
```

```
for (n=0; n<=120; n++)
   x[n] = delta(n)
          + 2*delta(n-40)
          + 2*delta(n-70)
          + delta(n-80);
```

```
conv(24, h, 121, x, y)
```



Overlap-Add Block Convolution



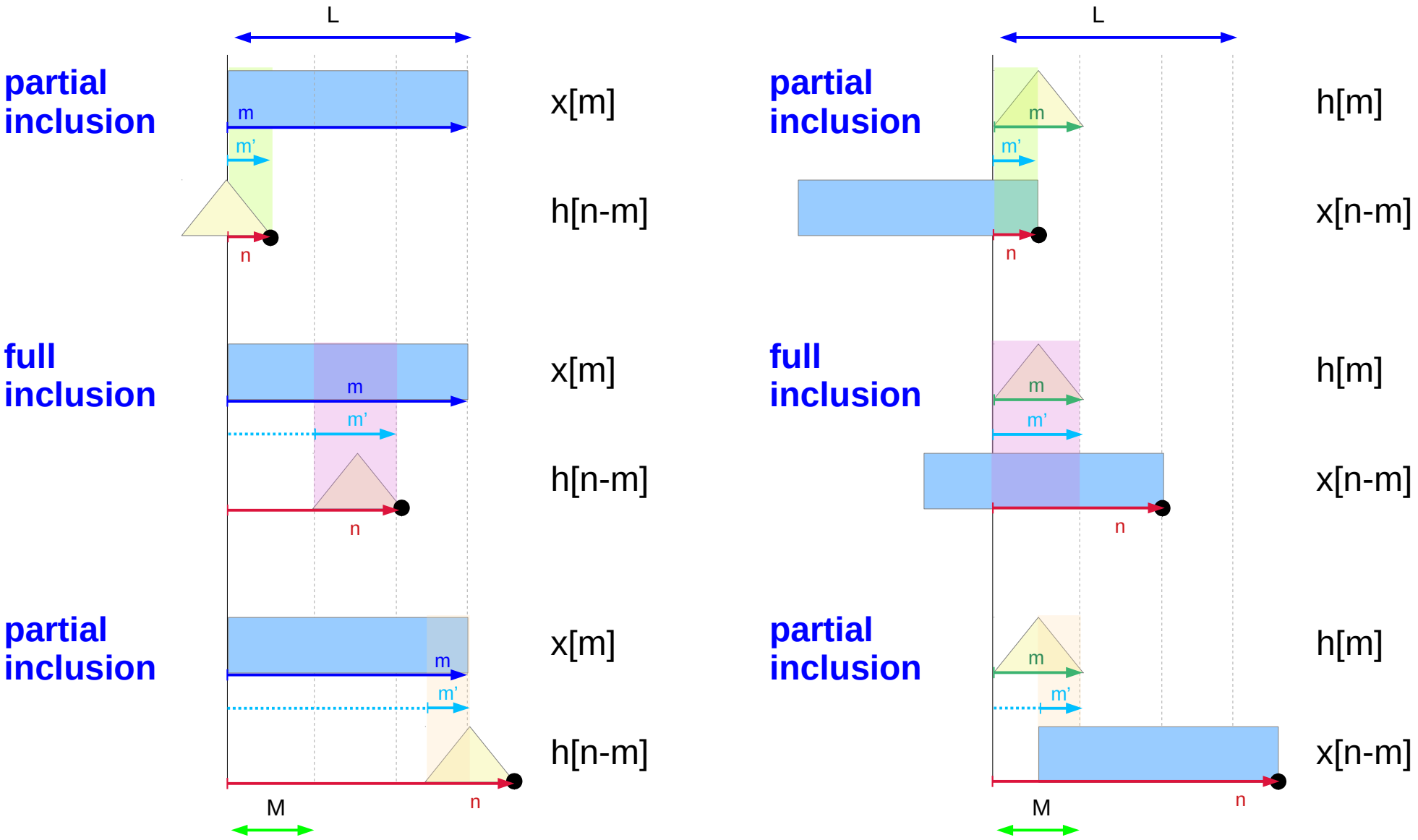
head steady tail

$$y_0 = y_{0,h} + y_{0,s} + y_{0,t}$$

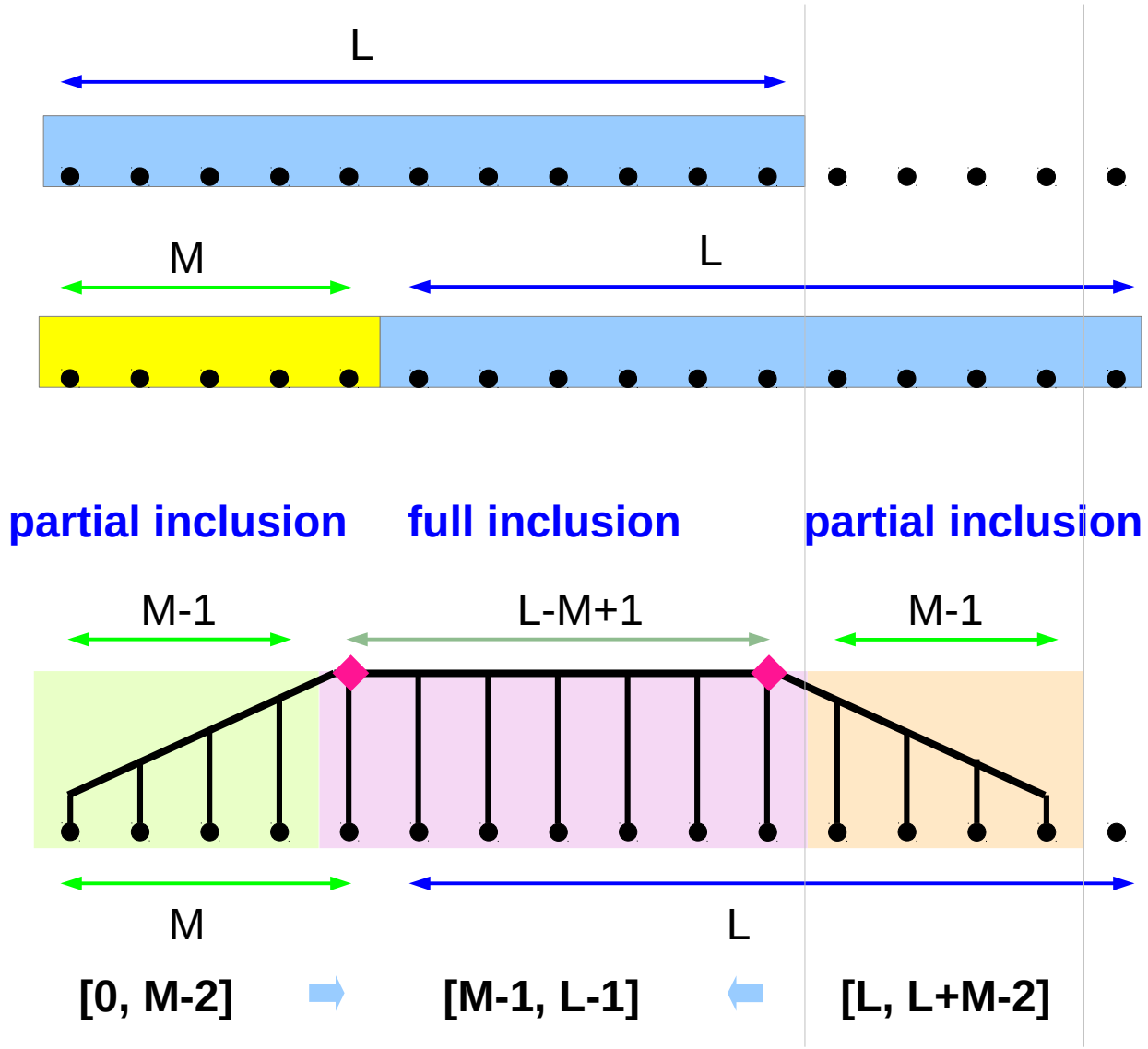
$$y_1 = y_{1,h} + y_{1,s} + y_{1,t}$$

$$y_2 = y_{2,h} + y_{2,s} + y_{2,t}$$

Three including regions in a convolution

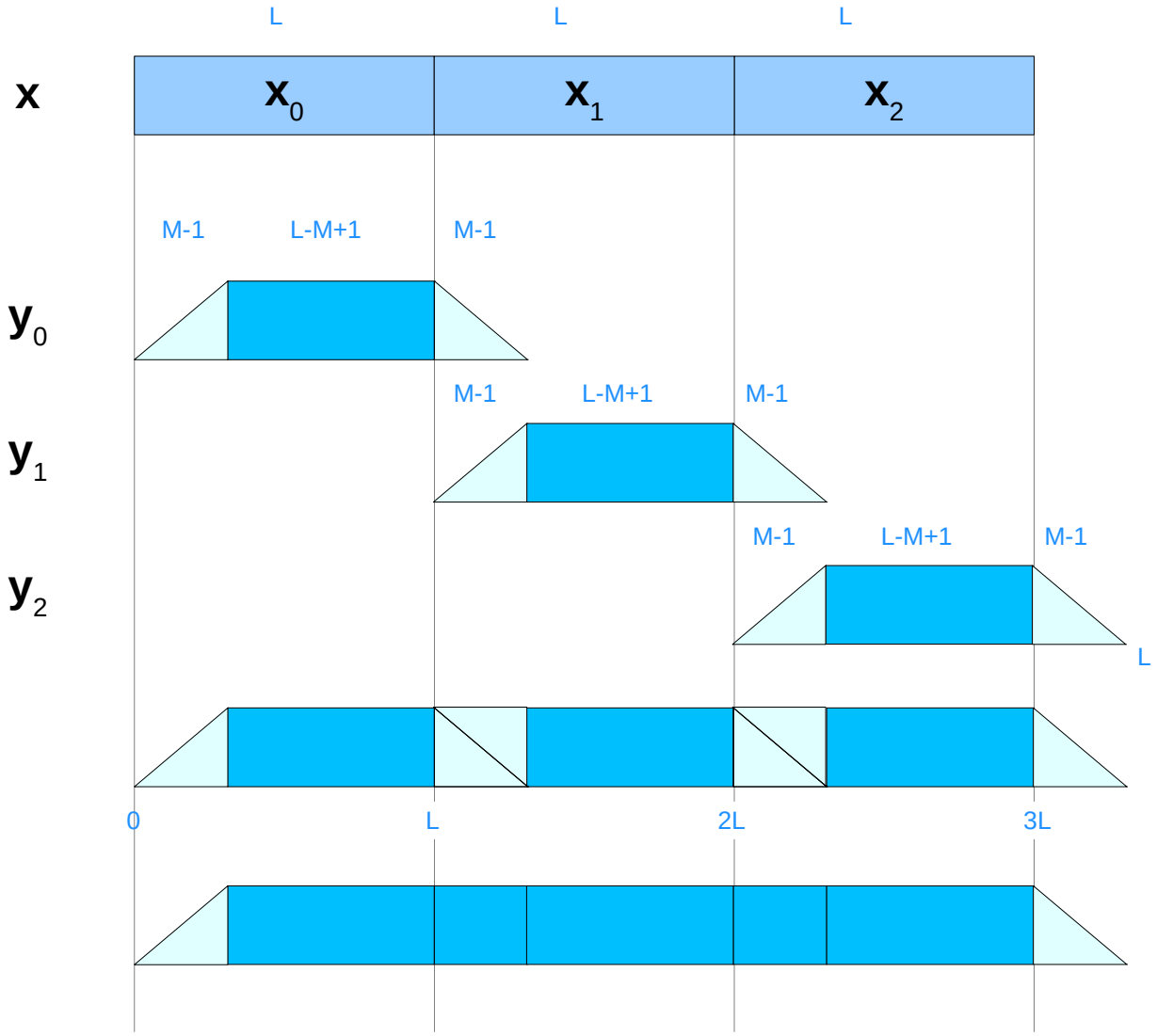


Index sizes of including regions



amount of the including indices

Head and Tail Transition Response



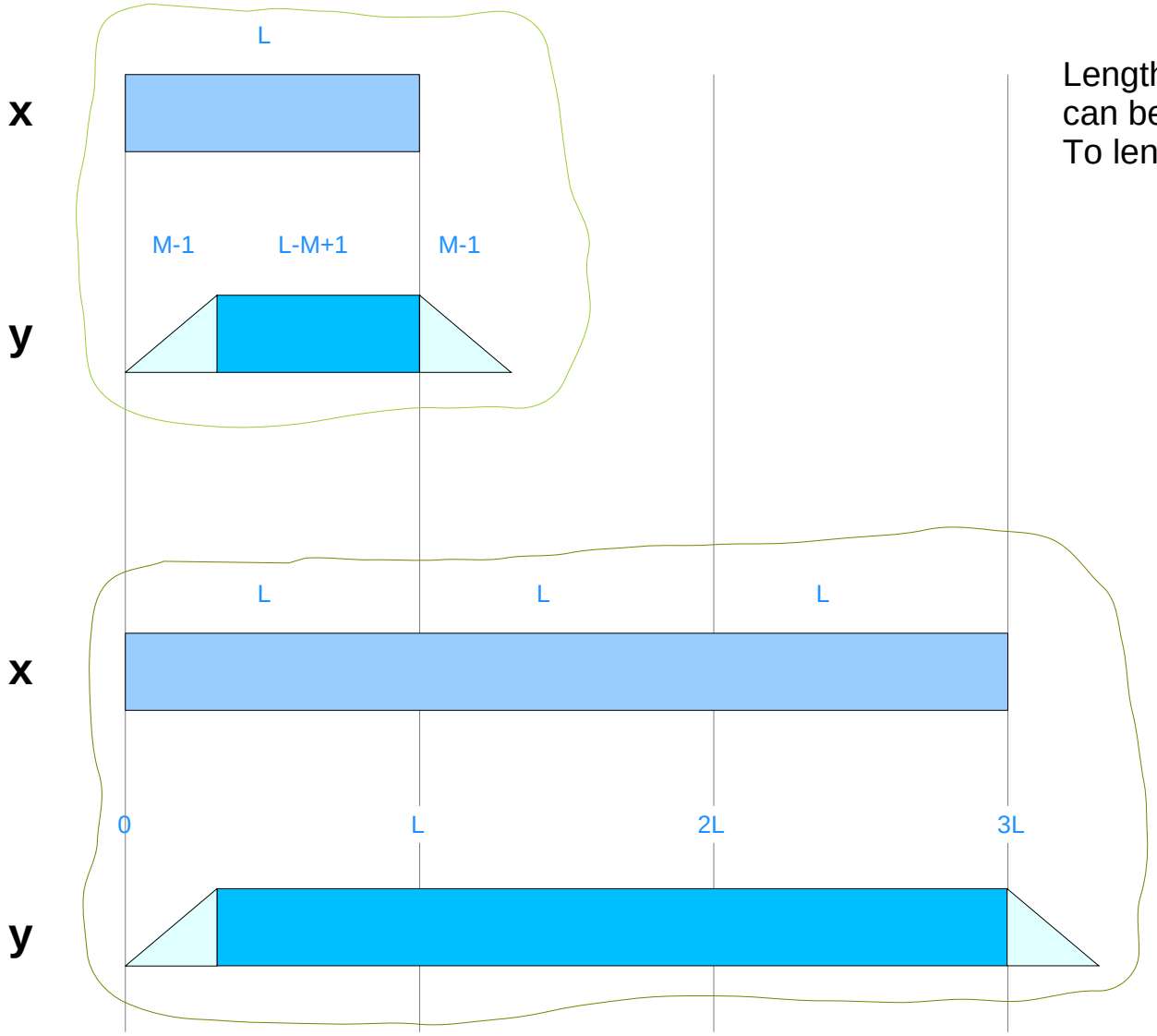
head steady tail

$$y_0 = y_{0,h} + y_{0,s} + y_{0,t}$$

$$y_1 = y_{1,h} + y_{1,s} + y_{1,t}$$

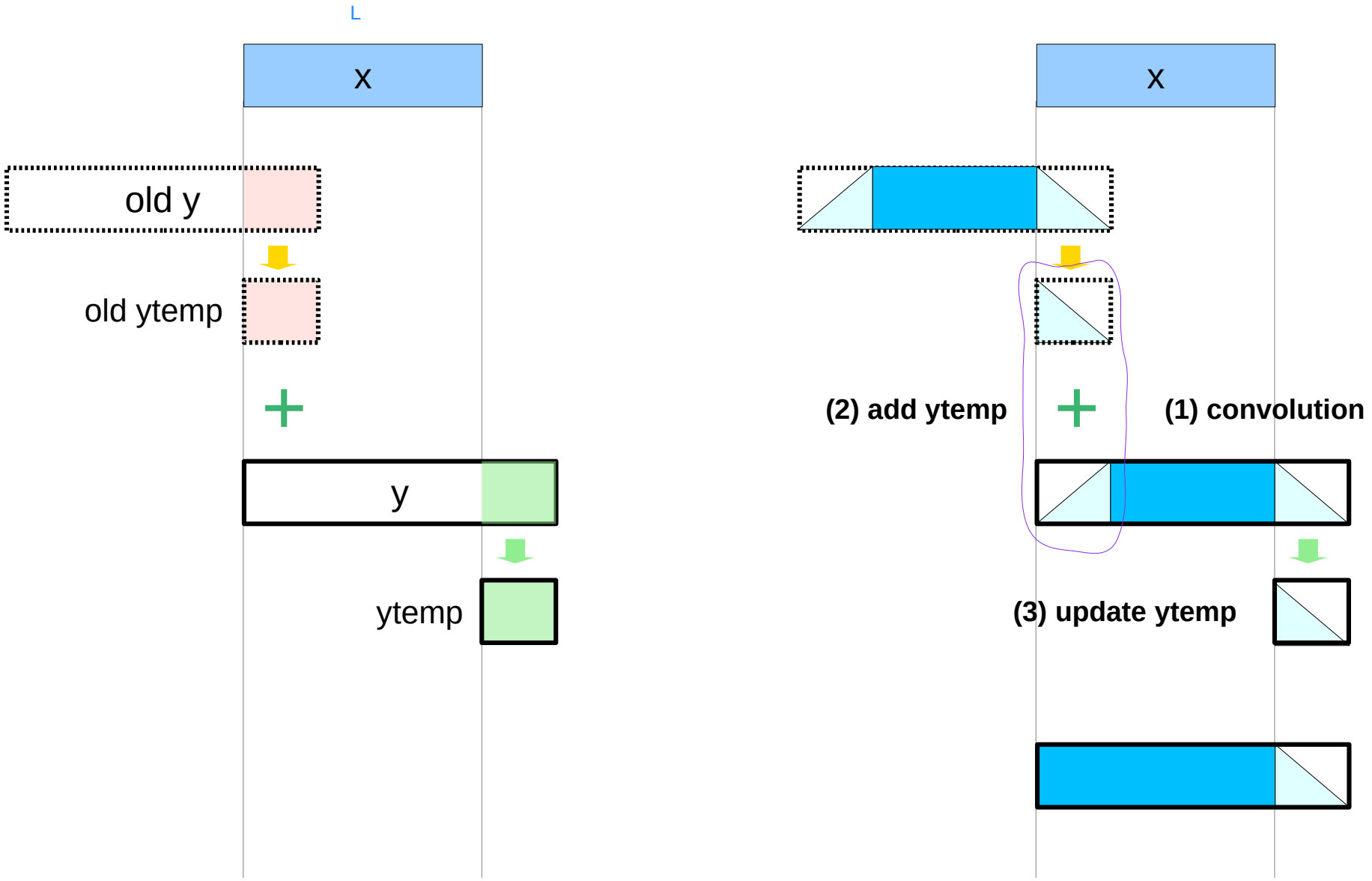
$$y_2 = y_{2,h} + y_{2,s} + y_{2,t}$$

Convolution with **L** and **3L** point inputs x[n]

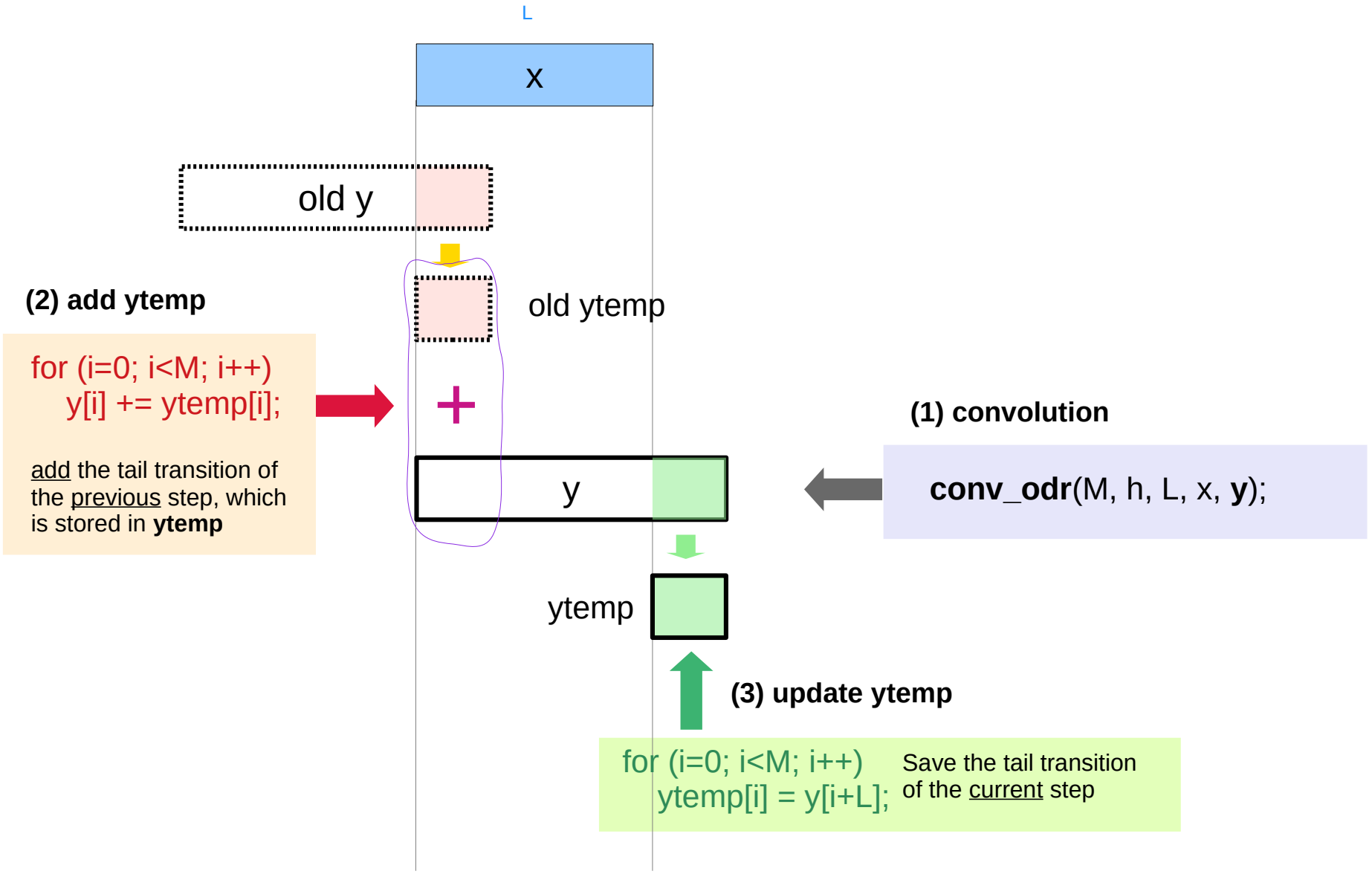


Length L convolution
can be repeatedly applied
To length 3L convolution

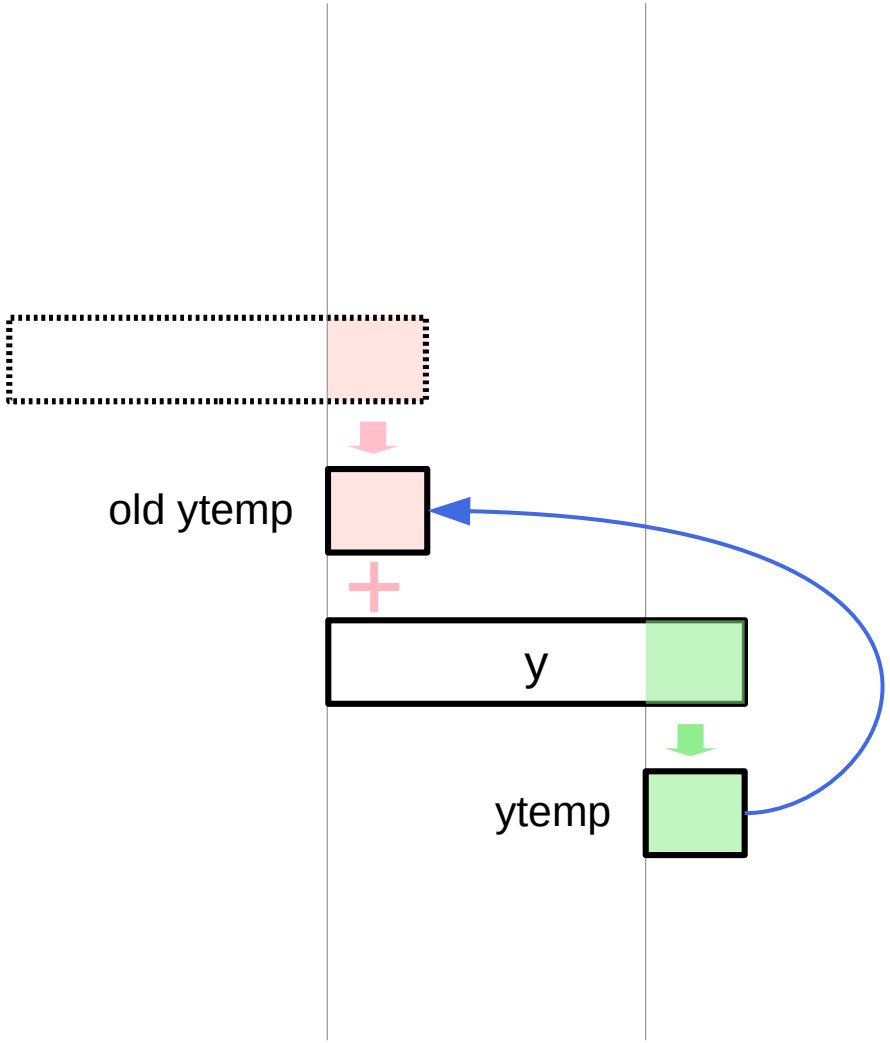
Overlap and add computation (1)



Overlap and add computation (2)



Overlap and add computation (3)



2) for (i=0; i<M; i++)
y[i] += ytemp[i];

1) conv_odr(M, h, L, x, **y**);

3) for (i=0; i<M; i++)
ytemp[i] = y[i+L];

Stack vs Heap memory (1)

```
#include <stdlib.h>
#include <stdio.h>

void func() {
  int *b = (int *) calloc(4, sizeof(int));
  int a[4] = {0,0,0,0};
  int i;

  for (i=0; i<4; ++i) {
    a[i] += 1;
    b[i] += 1;
  }

  for (i=0; i<4; ++i) {
    printf("a[%d]=%d b[%d]=%d \n", i, a[i], i, b[i]);
  }
  printf(" \n");

  free(b);
}

int main() {


  func();
  func();
  func();

}
```

```
a[0]=1 b[0]=1
a[1]=1 b[1]=1
a[2]=1 b[2]=1
a[3]=1 b[3]=1

a[0]=1 b[0]=1
a[1]=1 b[1]=1
a[2]=1 b[2]=1
a[3]=1 b[3]=1

a[0]=1 b[0]=1
a[1]=1 b[1]=1
a[2]=1 b[2]=1
a[3]=1 b[3]=1
```

whenever **func** is called,
the array is allocated in the heap memory
and then deallocated just before returning
to the caller **main**

this has the same effect as the **auto**
variable on the **stack**

Stack vs Heap memory (2)

```
#include <stdlib.h>
#include <stdio.h>

void func(int p[]) {
  int a[4] = {0,0,0,0};
  int i;

  for (i=0; i<4; ++i) {
    a[i] += 1;
    p[i] += 1;
  }

  for (i=0; i<4; ++i) {
    printf("a[%d]=%d p[%d]=%d \n", i, a[i], i, p[i]);
  }
  printf(" \n");
}
```

```
int main() {

  int *b = (int *) calloc(4, sizeof(int));

  func(b);
  func(b);
  func(b);

  free(b);
}
```

```
a[0]=1 p[0]=1
a[1]=1 p[1]=1
a[2]=1 p[2]=1
a[3]=1 p[3]=1

a[0]=1 p[0]=2
a[1]=1 p[1]=2
a[2]=1 p[2]=2
a[3]=1 p[3]=2

a[0]=1 p[0]=3
a[1]=1 p[1]=3
a[2]=1 p[2]=3
a[3]=1 p[3]=3
```

when **main** is called,
the array is allocated in the heap memory

it is pointed by the **local** pointer variable **b**,
which is passed as an **argument** to the **func**

the array retains its value until either new values are assigned to it or the **free** command deallocates it

Stack vs Heap memory (3)

```
#include <stdlib.h>
#include <stdio.h>
int *b;

void func() {

  int a[4] = {0,0,0,0};
  int i;

  for (i=0; i<4; ++i) {
    a[i] += 1;
    b[i] += 1;
  }

  for (i=0; i<4; ++i) {
    printf("a[%d]=%d b[%d]=%d \n", i, a[i], i, b[i]);
  }
  printf(" \n");

}

int main() {
  b = (int *) calloc(4, sizeof(int));

  func();
  func();
  func();

  free(b);
}
```

```
a[0]=1 b[0]=1
a[1]=1 b[1]=1
a[2]=1 b[2]=1
a[3]=1 b[3]=1

a[0]=1 b[0]=2
a[1]=1 b[1]=2
a[2]=1 b[2]=2
a[3]=1 b[3]=2

a[0]=1 b[0]=3
a[1]=1 b[1]=3
a[2]=1 b[2]=3
a[3]=1 b[3]=3
```

the **global** pointer variable **b**

do not have to pass **b** as an **argument** to the **func**

the array retains its value until either new values are assigned to it or the **free** command deallocates it

blockconv_len.c

```
/* blockcon.c - block convolution by overlap-add method */
void conv();

// ytemp is tail of previous block,
// M= filter length,L= block size

void blockcon_odr(int M, double *h, double *x, double *y, double *ytemp, int L) {
{
  int i;
  conv_len(M, h, L, x, y); // compute output block y

  for (i=0; i<M-1; i++) {
    y[i] += ytemp[i];          // add tail of previous blocky
    ytemp[i] = y[i+L];         // update tail for next call
  }
}

double *h, *x, *y, *ytemp;

h = (double *) calloc(M, sizeof(double));          // M–dimensional
x = (double *) calloc(L,   sizeof(double));        // L–dimensional
y = (double *) calloc(L+M-1, sizeof(double));      // (L+M-1)–dimensional
ytemp = (double *) calloc(M-1, sizeof(double));    // M-1–dimensional
```

blockconv_odr.c

```
/* blockcon.c - block convolution by overlap-add method */
void conv();

// ytemp is tail of previous block,
// M= filter order,L= block size

void blockcon_odr(int M, double *h, double *x, double *y, double *ytemp, int L) {
{
  int i;
  conv_odr(M, h, L, x, y);// compute output block y

  for (i=0; i<M; i++) {
    y[i] += ytemp[i];          // add tail of previous blocky
    ytemp[i] = y[i+L];         // update tail for next call
  }
}

double *h, *x, *y, *ytemp;

h = (double *) calloc(M+1, sizeof(double));     // (M+1)–dimensional
x = (double *) calloc(L,   sizeof(double));     // L–dimensional
y = (double *) calloc(L+M, sizeof(double));     // (L+M)–dimensional
ytemp = (double *) calloc(M, sizeof(double)); // M–dimensional
```

blockconv.c

```
/* blockcon.c - block convolution by overlap-add method */
void conv();

void blockcon(int M, double *h, double *x, double *y,
                    double *ytemp, int L) {          // ytemp is tail of previous block
{                                                    // M= filter order,L= block size
  int i;
  conv(M, h, L, x, y);        // compute output block y

  for (i=0; i<M; i++) {
    y[i] += ytemp[i];         // add tail of previous blocky
    ytemp[i] = y[i+L];        // update tail for next call
  }
}

double *h, *x, *y, *ytemp;

h = (double *) calloc(M+1, sizeof(double));     // (M+1)–dimensional
x = (double *) calloc(L,   sizeof(double));     // L–dimensional
y = (double *) calloc(L+M, sizeof(double));     // (L+M)–dimensional
ytemp = (double *) calloc(M, sizeof(double)); // M–dimensional
```

blockconv.c

```
for ( ; ; ) {                          // continuous processing input blocks

    for (n=0; n<L; n++)
      if (fscanf(fpx, "%lf", x+n) == EOF)
        goto last;   // break;

      blockcon(M, h, L, x, y, ytemp);  // process input block

      for (i=0; i<L; i++)
        fprintf(fpy, "%lf\n", y[i]);   // write output block
}

last:
    N = n;
    blockcon(M, h, N, x, y, ytemp);    // last block has N ≤ L

    for (i=0; i<N+M; i++)
      fprintf(fpy, "%lf\n", y[i]);     // last output block including tails
```

len(y) = L+M
len(ytemp) = M
Tail size M
Block Size L

len(y) = N+M
len(ytemp) = M
Tail size M
Block Size N

References

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