INTELLIGENT OPTO SENSOR
Number 1
DESIGNER'S NOTEBOOK
TAOS TEXAS ADVANCED OPTOELECTRONIC SOLUTIONS™

Position Detection Using Maximal Length Sequences

Contributed by David J. Mehrl, Dept. of Electr. Engr., Texas Tech University

Design Problem What type of optical encoding technique can be used with linear sensor arrays to perform position sensing with high accuracy, with high tolerance to wobble and misalignment?

Solution Binary encoded disks are often used for optical encoding applications as a means of accurately detecting position or rotation. Popular encoding schemes include the binary code and the Gray code. The Gray code, unlike the binary code, has the characteristic that adjacent code words within the sequence differ only by a single bit. Figure 1 depicts binary and Gray code sequences for n=3 (2^3 = 8 code words). The sequence goes from left to right.

Binary Code: 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111
Gray Code: 000 | 001 | 011 | 010 | 110 | 111 | 101 | 100

Figure 1: Binary and Gray code sequences (n=3).

Typical patterns for translational and angular position detection, based on a 6-bit Gray code, are shown below. Spatial or angular resolution depends on the code length, e.g. using an n-bit binary code gives a spatial resolution of $L/2^n$ or an angular resolution of $360°/2^n$.

(a.) (b.)

Figure 2: Optical encoder patterns for (a) translational and (b) angular position detection (n=6).

Also depicted in Figure 2a is the position of line array detector used to sense the pattern. One shortcoming of this technique is that the detector needs to maintain precise alignment as the pattern undergoes mechanical movement. For example, the

pattern shown in Fig. (2a) is not tolerant to “wobble” in the vertical direction, and the pattern in Fig. (2b) is intolerant to wobble in the radial direction. This weakness can be partially overcome by use of “oversampling” and guard bits, where each bit position within the pattern is imaged onto several elements of the optical detector array.

To circumvent the sensitive alignment problem, we propose a new encoding strategy which makes use of “maximal length sequences”, hereafter abbreviated as MLS. The total length of an n-bit MLS is 2^n. The sequence has the property that any n-bit sub-sequence within the 2^n bit sequence is unique. Figure 3 shows a simple 6-bit maximal length sequence.

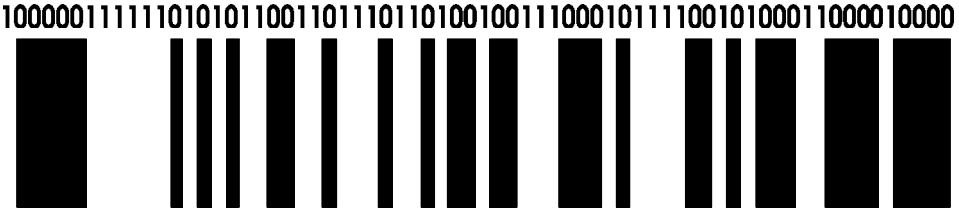

Figure 3: Pattern corresponding to 6-bit maximal length sequence(white=1, black=0).

For example, the leftmost 6 bits are “100000”, and moving 1 position to the right yields a sub-sequence of “000001” etc.. Since any 6-bit sub-sequence is unique, reading the 6-bit sub-sequence (by means of a linear array detector) and referring to a look up table allows this scheme to be used for sensing position.

Generating Maximal Length Sequences:

Maximal length sequences may be generated by a simple shift register as shown in Figure 4.

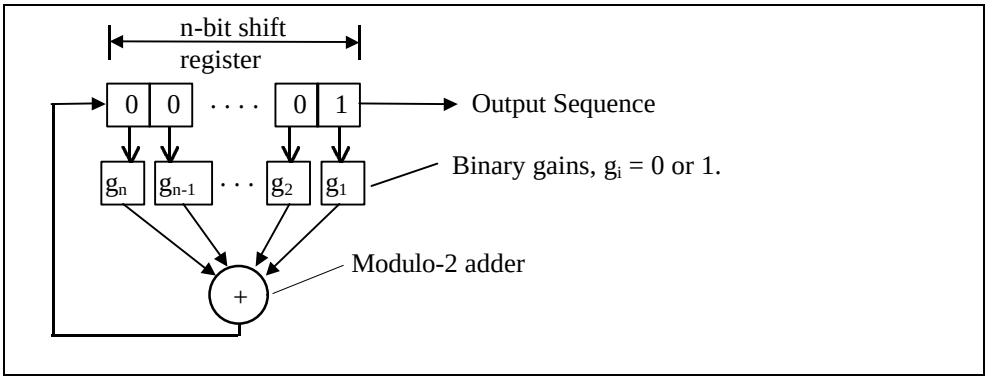

Figure 4: State machine for producing maximal-length sequences.

The shift register can initially be loaded with any binary sequence, with the exception of “all zeros”. Then each bit in the shift register is multiplied by a binary gain factor, i.e. $g_i = 0$ or 1, where $1 \leq i \leq n$, and the outputs are summed by a “modulo 2” adder. Equivalently, the adder generates a binary one if the sum is odd and a binary zero if the sum is even. The gains g_1 through g_n must be chosen carefully, otherwise the generated sequence will not be maximal length, that is it will repeat prematurely. Table 1 shows gains required to generate MLSs for various

bit lengths. The actual length of the sequence gives 2^n - 1 unique n-bit sub-sequences(or codewords) as the “all zeros” codeword is not allowed(note from Fig. 4 that an all zero state would cause the machine to forever remain in that state). Although all zeros is not allowed for the state machine, it can be inserted back into the final generated sequence to obtain the full 2^n codewords.

Table 1: Binary gains for generating maximal length sequences.

n	binary gains $g_n, g_{n-1}, \ldots, g_2, g_1$	n	binary gains $g_n, g_{n-1}, \ldots, g_2, g_1$
6	100001	9	100001000
7	1000001	10	1000000100
8	10001110	11	10000000010

We do not actually construct such a state machine, but simulate the state machine using software, in order to generate MLSs. The listing below shows a “Quick Basic” program for generating various n-bit maximal length sequences. The program should run on either Macintosh or IBM compatible machines, and Quick Basic is usually included free with “Windows”, in the “Applications” directory. The program prints out the sequences as a string of “0’s” and “1’s”, and optionally stores the sequence to a file.

```
REM BEGINNING OF QUICKBASIC PROGRAM ************************************
REM Program to generate maximal length sequences
CLS : DIM g$(11) 'these are gains which will generate max. length sequences
g$(6) = "100001": g$(7) = "1000001": g$(8) = "10001110": g$(9) = "100001000"
g$(10) = "1000000100": g$(11) = "10000000010"
loop1: INPUT "Generate an n-bit sequence where (6 <= n <= 11) - enter n"; n%
IF (n% > 11 OR n% < 6) THEN PRINT "Invalid value-try again": GOTO loop1
DIM st%(n%), g%(n%)
FOR I% = 1 TO n%  'set the gains
st%(I%) = 0  'initialize shift register to all zeros
g%(I%) = VAL(MID$(g$(n%), I%, 1))
NEXT I%
st%(1) = 1   'set the least significant bit to one
REM Now find the modulo-2 sum and shift the register-total sequence length is (2^n -1).
m% = 2 ^ n% - 1: DIM seq%(m%)  'array used to save the full sequence
FOR I% = 1 TO m%: sum% = 0
FOR j% = 1 TO n%: sum% = sum% + g%(j%) * st%(j%)
NEXT j%
mysum% = sum% MOD 2: so% = st%(1): seq%(I%) = so%  'save sequence in an array
FOR k% = 1 TO n% - 1: st%(k%) = st%(k% + 1) 'shift the state register
NEXT k%
st%(n%) = mysum%   'shift in the mod-2 sum to the left side of register
NEXT I%: REM Now print out the sequence
FOR I% = 1 TO m%
test% = I% MOD 64: myout$ = RIGHT$(STR$(seq%(I%)), 1): PRINT myout$;
sq$ = sq$ + myout$ 'also save a string representation of the sequence
IF test% = 0 OR I% = m% THEN PRINT " " 'new line after 64 characters
NEXT I%
REM Examine all n-bit sub-sequences to assure uniqueness-this section optional ========
INPUT "Test sequence (y/n)"; resp$ 'test may take awhile for long sequences
IF resp$ = "y" OR resp$ = "Y" THEN
```

```
flg% = 0: REM Flag for flagging whether repeated pattern found
FOR I% = 1 TO m% - n%: ss$ = MID$(sq$, I%, n%)
FOR j% = I% + 1 TO m% - n%
IF ss$ = MID$(sq$, j%, n%) THEN : PRINT "repeated pattern-Bad!": flg% = 1
NEXT j%: NEXT I%
IF flg% = 0 THEN PRINT "No repeated patterns found:Good!!!"
END IF:REM ====================================================
PRINT "Enter filename to save sequence to:"
INPUT "<default - do not save to file>"; myfn$
IF myfn$ <> "" THEN  'This saves sequence to file for later processing
sq$ = sq$ + LEFT$(sq$, n% - 1) 'the end of full sequence repeats the 1st n-1 bits
OPEN myfn$ FOR OUTPUT AS #1: j% = LEN(sq$)
FOR I% = 1 TO j%: ss$ = MID$(sq$, I%, 1): k% = I% MOD 64: PRINT #1, ss$;
IF k% = 0 OR I% = j% THEN PRINT #1, " "
NEXT I%: CLOSE #1
END IF    'END OF QUICKBASIC PROGRAM *********************************
```

The above program contains an optional routine which verifies that all n-bit sub-sequences are unique. The last part of the program optionally allows the user to save the sequence to a file for later reference. Included below is a program which will read in a previously saved sequence and render a graphics pattern to the screen. If you are using an IBM compatible with VGA graphics, you may want to change “SCREEN 9” to “SCREEN 11” or some other suitable screen mode, to get higher resolution graphics. If using a Macintosh, delete the “SCREEN” statement entirely.

```
REM QuickBasic Program to read sequence file & draw a pattern to the screen ********
CLS : INPUT "Name of sequence data file"; myfn$: OPEN myfn$ FOR INPUT AS #1: b$ = ""
WHILE NOT EOF(1): INPUT #1, a$: b$ = b$ + a$: WEND
m% = LEN(b$) 'requires EGA graphics. Change screen mode as appropriate for your PC.
SCREEN 9 'Comment out or omit this line if using a Macintosh
CLS : FOR I% = 1 TO m%: x% = I% 'horizontal position
c$ = MID$(b$, I%, 1): IF c$ = "0" THEN LINE (x%, 50)-(x%, 100):
NEXT I%: CLOSE #1: REM End of QuickBasic Program ************************************
```

The next program will print all possible n-bit sub-sequences to the screen and (optionally) to a file as well. This is convenient for generating lookup tables. It also allows you to reinsert the “all zeros” sub-sequence to create all possible 2^n sub-sequences.

```
REM QBasic Program to read sequence file and create lookup table **********************
CLS : INPUT "Name of sequence data file "; myfn$: OPEN myfn$ FOR INPUT AS #1: b$ = ""
WHILE NOT EOF(1): INPUT #1, a$: b$ = b$ + a$: WEND: CLOSE #1: k% = LEN(b$)
INPUT "Reinsert all zeros sub-sequence(y/n)-<default = n> "; resp$
IF resp$ = "y" OR resp$ = "Y" THEN a$ = "10" + RIGHT$(b$, k% - 1): b$ = a$: k% = k% + 1
INPUT "Name of file to write lookup table data to:<Default=none>"; myfn$
IF myfn$ <> "" THEN OPEN myfn$ FOR OUTPUT AS #1
n = LOG(k%) / LOG(2): n% = INT(n + .5) 'find substring length :
m% = 2 ^ n% - 1: IF resp$ = "y" OR resp$ = "Y" THEN m% = m% + 1: mycnt% = 0
FOR i% = 1 TO m%: a$ = MID$(b$, i%, n%): PRINT i%; a$
IF myfn$ <> "" THEN PRINT #1, i%; a$:
NEXT i%: IF myfn$ <> "" THEN CLOSE #1
```

REM End of QuickBasic Program**********