Gray code fundamentals

The Gray Code

R W Doran
Department of Computer Science
The University of Auckland

Abstract

This report is a self-contained summary of properties and algorithms concerning the Gray code. Descriptions are given of the Gray code definition, algorithms and circuits for generating the code and for conversion between binary and Gray code, for incrementing, counting, and adding Gray code words. Some interesting applications of the code are also treated.

1. Introduction

What we now call Gray code was invented by Frank Gray. It was described in a patent that was awarded in 1953, however, the work was performed much earlier, the patent being applied for in 1947. Gray was a researcher at Bell Telephone Laboratories; during the 1930s and 1940s he was awarded numerous patents for work related to television. According to Heath [Hea72] the code was first, in fact, used by Baudot for telegraphy in the 1870s, though it is only since the advent computers that the code has become widely known.

The term Gray code is sometimes used to refer to any single-distance code, that is, one in which adjacent code words (perhaps representing integers differing by 1) differ by 1 in one digit position only. Gray introduced what we would now call the canonical binary single-distance code, though he mentioned that other binary single-distance codes could be obtained by permuting the columns and rotating the rows of the code table. The codes of Gray, and natural extensions to bases other than binary, are only a very small subset of all single-distance codes. In this report we will use t term "the Gray code" to refer to the code of Gray and "single-distance" to refer to the more general case; we will be concerned mainly with properties of the Gray code.

The original purpose of this report was to consider algorithms for parallel arithmetic using Gray codes (the Gray representation is particularly suited to serial arithmetic; more ingenuity is requir to operate in parallel). In surveying the literature it became clear that there had been much discovered and written about the Gray code; it is associated with many elegant algorithms and circuits. However, this wealth of technical material had never been gathered together and treated in a consistent form, hence, a self-contained survey of the codes properties, algorithms and circuits, has become the main topic, though parallel operations are included.

2. Definition of the Gray Code

Origin of the code

The Gray code arises naturally in many situations. Grays interest in the code was related to what we would now call analog to digital conversion. The goal was to convert an integer value, represented as a voltage, into a series of pulses representing the same number in digital form. The technique, as described in Grays patent, was to use the voltage being converted to displace vertically an electron beam that is being swept horizontally across the screen of a cathode ray tube The screen has a mask etched on it that only allows the passage of the beam in certain places; a current is generated only when the beam passes through the mask. The passage of the beam will then gives rise to a series of on/off conditions corresponding to the pattern of mask holes that it passes.

The original scheme was to use a mask representing a standard binary encoding. However, this has the problem that, if the beam is close to the boundary between two values, a slight distortion in th beam can give an output that is neither of the two adjacent values but a combination of each (in the example below, in the transition from 011011 (27) to 011100 (28), the device could produce these

The manner in which the primary reflected binary number system is built up will now be explained.

First: write down the first two numbers in the 1-digit orthodox number system, thus:

Zero	0
One	1

```
        Note that the symbols differ in only one digit.

Second: below this array write its reflection in a transverse axis:

                       Zero           0
                       One            1
                       ----------
                                      1
                                      0

        The symbols still differ in not more than one digit. However, the first is
identical with the fourth and the second with the third.

Third: to remove this ambiguity, add a second digit to the left of each symbol, 0 for the
first two symbols and 1 for the last two, thus:
                       Zero           00
                       One            01
                       Two            11
                       Three          10
and identify the last two symbols with the numbers two and three. Each symbol is
now unique and differs from those above and below in not more than one digit. The
array is a representation of the first four numbers in the primary 2-digit  reflected binary
number system.
        The process is next repeated giving -
First:
                       Zero           00
                       One            01
                       Two            11
                       Three          10
Second:
                       Zero           00
                       One            01
                       Two            11
                       Three          10
                       -----------
                                      10
                                      11
                                      01
                                      00
Third:
                       Zero           000
                       One            001
                       Two            011
                       Three          010
                       ------------
                        Four          110
                        Five          111
                        Six           101
                        Seven         100
```

Figure 2. Grays Definition of his Reflected Binary Code

two values but also 011111 (31) or 011000 (24) and others) . To deal with this problem Gray proposed using a mask corresponding to a code in which adjacent code words differed in one bit position only. In that case, a slight beam displacement would give only a small change to the encoding. Figure 1 is an adaptation of the figure in the patent.

Grays definition of the Code

Figure 2 is a word-for-word reproduction of the definition given by Gray in the patent [Gra53] - it has never been explained better.

Grays definition is a procedure for generating, what we now call, the Gray code of width n. As well as discussing the process, he has shown, by construction that:

Property P1: Adjacent words in the Gray code sequence differ in one bit position only.

Direct application of the code

Because, apart from the leading bit, the second half of the code is the same as the first, but reversed, it follows that the first and last words of the code sequence differ in only the leading b
In other words:

Property P2: The Gray code is cyclic.

These first two properties underly the most common practical use found for the code which was for locating the rotational position of a shaft (see, for example, [Fos54]). A pattern representing the Gray code was printed on a shaft, or on a disk, and the pattern sensed by an optical or electric detector (see figure 3). Note that the least significant end of the code has fewer transitions than does normal binary so the Gray code has another apparent advantage that the pattern may be printed to another bit of precision with the same printing resolution [Wal70]. Note that the readout of the shaft's rotational position is a completely parallel operation.

Generation of the code sequence by means related to its definition

Let us say that going through the Gray code sequence normally, is going up, or ascending and the opposite direction down, or descending. Generating a sequence going down is the same as reflecting it, in Grays sense. The sequence of width n comprises, by definition:

    0 preceding each member of the width n-1 sequence
    1 preceding each member of the width n-1 sequence reflected

To generate going down, this is reflected to give:

Figure 3. Gray code as used on a shaft encode for determining angle of rotation

    1 preceding each member of the width n-1 sequence reflected reflected
    0 preceding each member of the width n-1 sequence reflected

But reflecting a sequence twice gives back the original sequence, so the width n sequence reflected is:

    1 preceding each member of the width n-1 sequence
    0 preceding each member of the width n-1 sequence reflected

This gives us the property:

Property P3: A descending Gray code sequence of width n is the same as an ascending sequence except that the leading bit is inverted.

For example, the width 3 sequence:

up	down
000	100
001	101
011	111
010	110
110	010
111	011
101	001
100	000

Lets use the following notation. The Gray code word G, of width n, is a vector of n bits, (Gn-1,Gn-2,......G0) and represents a number G. Likewise, a number B has the standard representation B, as a vector of n bits, (Bn-1,Bn-2,......B0). We will most usually be interested in the situation when B=G.

In expressing algorithms we will use a data type bit that is the union of Boolean and integer, and also word that is an array of bits.

Grays definition of his code sequence of width n is captured by the following algorithm:

```
procedure generate (n:integer, d:bit);
        {generate width-n sequence in d(irection) up = 0, down = 1}
        var G:word;
        procedure generate1 (m:integer; d:bit);
        begin
        if d = 0 then begin
                G[m-1] := 0; generate1(m-1,0);{up}
                G[m-1] := 1; generate1(m-1,1);{down}
                end;
        if d = 1 then begin
                G[m-1] := 1; generate1(m-1,0);{up}
                G[m-1] := 0; generate1(m-1,1);{down}
                end
        end;
begin
generate1(n,0);
end;
```

Dealing with the termination of recursion, and simplifying, we end up with the elegantly simple algorithm:

```
{ALGORITHM A1: GENERATE WIDTH N GRAY CODE SEQUENCE}
procedure generate (n:integer, d:bit);
        {d(irection) up = 0, down = 1}
        var G:word;
        procedure generate1 (m:integer; d:bit);
        begin
        if m = 0 then display(G) else
                begin
                G[m-1] := d;            generate1(m-1,0);{up}
                G[m-1] := not d;                generate1(m-1,1);{down}
                end;
        end;
begin
generate1(n,0);
end;
```

3. Relationship between binary code and Gray code

Generating the Gray code from binary

The above algorithm, with two calls per recursion, has a binary tree of possible procedure calls. We can label the nodes of the tree, and thus give each Gray code word a binary equivalent, by setting a bit prior to each recursive call. Lets concentrate on the ascending sequence:

```
procedure generate (n:integer);
        {direction up}
        var B,G:word;
        procedure generate1 (m:integer; d:bit);
        begin
        if m = 0 then display(B,G) else
                begin
                G[m-1] := d;            B[m-1] := 0; generate1(m-1,0);{up}
                G[m-1] := not d;        B[m-1] := 1; generate1(m-1,1);{down}
                end;
begin
generate1(n,0);
end;
```

The algorithm will now generate the integers B along with the associated Gray codes. The inner compound statement is equivalent to:

```
                begin
                G[m-1] := d;            B[m-1] := 0;    generate1(m-1,B[m-1]);{up}
                G[m-1] := not d;        B[m-1] := 1;    generate1(m-1,B[m-1]);{down}
                end;
```

i.e. (if we set B[n] appropriately):

```
                begin
                G[m-1] := B[m];         B[m-1] := 0;    generate1(m-1);
                G[m-1] := not B[m];     B[m-1] := 1;generate1(m-1);
                end;
```

i.e.

```
                begin
                B[m-1] := 0;G[m-1] := B[m]Å B[m-1]; generate1(m-1);
                B[m-1] := 1;G[m-1] := B[m]Å B[m-1]; generate1(m-1);
                end;
```

i.e.

```
                for B[m-1] := 0 to 1 do begin
                                G[m-1] := B[m]Å B[m-1]; generate1(m-1);
                                end;
```

Now, as G is not used except in "display", the generation of its elements may be done in any order following the generation of the necessary bits of B - it can thus be generated at "display time". Giving:

```
{ALGORITHM A2: GENERATE WIDTH N GRAY CODE SEQUENCE FROM
BINARY SEQUENCE}

procedure generate (n:integer); {generate width-n sequence}
        var G:word; i:integer;
        procedure generate1 (m:integer; d:bit);
                {d(irection) up = 0, down = 1}
        begin
        if m>0 then for B[m-1] := 0 to 1 do generate1(m-1)
                else begin
                        for i := n-1 downto 0 do G[i-1] := B[i]Å B[i-1];
                        display(B,G);
                        end
        end;
begin
B[n] := 0;
generate1(n);
end;
```

Conversion from binary to Gray

The above generation algorithm gives us immediately the property (specified by Gray):

Property P4: (Gi-1 = BiÅ Bi-1), i = n .... 0, where Bn is taken as 0

This gives a parallel algorithm or circuit for generating G from B, because the expressions are independent. Alternatively, if a computer has a bitwise xor between words then we can calculate G using a right shift:

G = BÅ (B/2)

Another way of thinking of this is:

Property P5: Gi-1 = 1 where Bi ¹ Bi-1, i = n .... 0 (where Bn is taken as 0)

i.e. the Gray code word is a record of the transitions within the corresponding binary word.

Example.

Binary word	00111100110011101001101111101101
Gray code word	00100010101010011101011000011011

Conversion of Gray to binary

Conversion of Gray to Binary is not as simple as the other direction. We have from property P4:

"i (BiÅGi-1 = BiÅBiÅ Bi-1) where Bn is taken as 0, i.e.

Property P6: Bi-1 = BiÅ Gi-1, i = n .... 0, where Bn is taken as 0

but unfortunately these are not independent and individual equations. They do give rise naturally to a nice sequential algorithm but the parallel version involves a prefix accumulation of xor:

Property P7: Bi-1 = Gn-1Å Gn-2Å ..... Å Gi-1

This can be generated by a parallel prefix circuit as in Figure 4.

Figure 4. Parallel Gray to Binary Conversion Circuit

Alternatively [Wan66], if a computer has a bitwise xor between words and fast parallel shifts then the binary code may be generated by a succession of xors and shifts that implement the work of figure 4, level by level:

B = GÅ (G/2); B = BÅ (B/4); B = BÅ (B/16); etc

However, there are conversion techniques that are more suited to software. Lets concentrate on the bits of the Gray code word that are 1. Define for each G another vector I of length z. I = (Iz-1,Iz- ... ,I0) which is the set of subscripts for which the Gray code is not zero. Recalling property P5 that the Gray code word is a record of the transitions within the corresponding binary word, Iz-1 is the position of the first 1 in the binary code and Iz-2 is the next 0, etc. Now, we have:

B = Bn-12n-1+Bn-22n-2+... +B020

Listing only the bits of B that are non-zero:

B = [2Iz-1+... +2(Iz-2+1)] + [2Iz-3+... +2(Iz-4+1)] + ...

Applying a Booth recoding:

B = [2(Iz-1+1) - 2(Iz-2+1)] + [2(Iz-3+1) - 2(Iz-4+1)] + ...
(-1 if the number of 1 bits in G is odd)

LetÕs write P(X,a,b) for the parity of (Xa ....Xb), which can be defined as Xa Å....ÅXb , or (Xa+...+Xb)mod 2, or whether the number of bits 1 in (Xa ....Xb) is odd (1) or even (0). Also write P(X,i) for P(X,n-1,i) and P(X) for P(X,n-1,0).

We may write the above:

Property P7: B = (-1)P(G,Iz-1).2(Iz-1+1) + ... + (-1)P(G,I0).2(I0+1) - P(G);

This property my be used to convert from Gray to binary by adding the shifted bits of the Gray code with appropriate sign.

Example:

Binary word	0011100111
Gray code word	0010010100

B = 0100000000 - 0000100000 + 0000001000 - 0000000001

This property also explains the origin of difficulty with doing arithmetic on Gray code words. In a conventional binary word, if bit i is one it means 2i, but bit i in a Gray code word could represent +2i+1 or -2i+1 - the sense can only be resolved if the parity of the leading part of the word up to bit is determined. In a sense, Gray code is a signed-bit ternary representation [Wal70], where each bit can be 1, 0, or -1 (but with the restriction that non-zero bits must alternate in sign).

Although the property P7 could be used to convert from Gray to Binary, it is not a good approach, because the subtractions involve propagation of carry. A better approach, ([Irs87], also noted by Gray himself), is to replace each power 2i in the above by (2i-1)+1. We get:

B = (-1)P(G,Iz-1). [2(Iz-1+1)-1] + .. + (-1)P(G,I0). [2(I0+1)-1]
+ (-1)P(G,Iz-1) +(-1)P(G,Iz-2)+...+(-1)P(G,I0) - P(G)

The second line is zero. So we have:

Property P8: B = (-1)P(G,Iz-1). [2(Iz-1+1)-1] + .. + (-1)P(G,I0). [2(I0+1)-1]

The reason that this is useful is that the successive additions and subtractions can now be performed to construct the binary equivalent with no carry being required at any stage (in fact, addition and subtraction may be replaced by xor).

Example:

Binary word	0011100110
Gray code word	0010010101

B = 0011111111 - 0000011111 + 0000000111 - 0000000001
= 0011100000 + 0000000110

Parity of Gray code word

Property P8 shows that knowledge of the parity of a Gray code word can useful. We will see other examples of its use later.

Recall that in going up from B to B+1 exactly one bit of G changes. It follows that exactly two bits change in going from B to B+2 . Thus the nunber of bits that are 1 remains the same or changes by 2, i.e. the parity remains the same. This gives us:

Property P9: The parity of a Gray code word is 0 iff it represents an even number, i.e. P(G,n-1,0) = B0

One of the drawbacks of the convential binary representation is that the parity of the result of an arithmetic operation is not easy to predict from the parities of its operands. However, the sum or difference of two numbers is even if, and only if, the inputs are both even or both odd, and the product is even if either operand is even. This allows the parity of Gray-code results to be predicted:

Property P10: If the parities of two Gray code operands are Pa and Pb, then the parity of the Gray code result is:

+	Pa Å Pb
*	Pa and Pb

Gray codes arising in binary counters

In [Bur70] it was noted that Gray codes arise naturally if one constructs a binary counter from master-slave (i.e. race-free or edge triggered) toggle flip-flops. In a master-slave flip-flop the ÒsecondÓ flip-flops, represent the value. However, if we concentrate on the ÒfirstÓ flip-flops they are seen to be following a different pattern.

Figure 5. Binary counter with master/slave flip/flops

S4	F3	S3	F2	S2	F1	S1	F0	S0	S	F
0	0	0	0	0	0	0	0	0	00000	0000
0	0	0	0	0	0	0	1	1	00001	0001
0	0	0	0	0	1	1	1	0	00010	0011
0	0	0	0	0	1	1	0	1	00011	0010
0	0	0	1	1	1	0	0	0	00100	0110
0	0	0	1	1	1	0	1	1	00101	0111
0	0	0	1	1	0	1	1	0	00110	0101
0	0	0	1	1	0	1	0	1	00111	0100
0	1	1	1	0	0	0	0	0	01000	1100
0	1	1	1	0	0	0	1	1	01001	1101
0	1	1	1	0	1	1	1	0	01010	1111
0	1	1	1	0	1	1	0	1	01011	1110
0	1	1	0	1	1	0	0	0	01100	1010

....

So, as the input and second flip-flops run through the ordinary binary integers, the first flip-flop run through the Gray code. The behaviour of Fi and Si+1 are entirely governed by the changes that occur in Si. Assuming that the counter is initially cleared, the following sequence of events will

repeat itself:

Si+1	Fi	Si
0	0	0
0	1	1
1	1	0

```
                         1               0               1
                         0               0               0
```

It can be seen that at all times Fi = Si+1 Å Si , so that F indeed is the Gray code. Bec is always set to Fi , but delayed, we see another interesting fact:

Property P11: Column i of a listing of the Gray code is the same as column i+1 of binary, rotated up by 2i.

4. Properties related to the transition bit index

Generation by minimal change

The Algorithms A1 and A2 generate a full Gray Code word at each step. However, because only one bit changes it is possible to generate each word from the previous by changing just that bit. But which bit?

Follow the execution of a certain level of recursion i in Algorithm A1that is called from level i+1 and passes control to level i-1. Successive calls to level i will be with direction:

```
                 up (d=0);  down (d=1); up (d=0);  down (d=1); etc.
```

The action of level i is then:

```
                 G[i-1] := 0,    call level i-1, G[i-1] := 1,    call level i-1; return;
                 G[i-1] := 1,    call level i-1, G[i-1] := 0,    call level i-1;          return; etc
```

Assuming that the Gray code word is initialised to 0, it can be seen that the above sequence is equivalent to:

```
                 call level i-1, G[i-1] := 1,            call level i-1;
                 call level i-1, G[i-1] := 0,            call level i-1; etc
```

That is, level i switches bit i-1 between successive calls to level i-1. So we get [Er85]:

```
{ALGORITHM A3.1: GENERATE WIDTH N GRAY CODE SEQUENCE, BY
SWITCHING}
procedure generate (n: integer);
        var G: word;
                i: integer;
        procedure generate1 (m: integer);
                begin
                if m >= 0 then
                        begin
                        generate1(m - 1);
                        G[m-1] := not G[m-1];
                        display(G);
                        generate1(m - 1);
                        end;
                end;
begin
for i := n-1 downto 0 do G[i] := 0;
display(G);
generate1(n);
end;
```

The sequence of transition indices

The algorithm A3.1 identifies the location of each element as it is switched. It is straightforward modify the algorithm so that it produces the sequence in which indices change:

```
{ALGORITHM A3.2: GENERATE SEQUENCE OF GRAY CODE TRANSITION
INDICES FOR WIDTH N}
procedure generate (n: integer);
        var G: word;
begin
   if n >= 0 then
        begin
        generate(n - 1);
        display(n-1);
        generate(n - 1);
        end;
end;
```

We see that the sequence of transitions has an even simpler structure than the original definition o the Gray code [BER76]:

```
        sequence for width n = sequence for width n-1, n-1, sequence for width n-1
```

5. Gray Code Incrementers

The task of an incrementer is, given a Gray code word, find the next in ascending order (likewise decrementers and descending). Incrementers are related to counters, which may save some auxilliary information to simplify the task, and to generating algorithms based on incrementing.

There are many papers, disclosures, and patents on this topic [Fis57, Maj71, CoSh69]. They all seem to have as a common concept the condition that is satisfied for a count up to occur. Consider algorithm A3.1. When the algorithm switches G[m-1] at level m, then, if m>1, level m-1 has been entered an odd number of times and level m-2, and below, an even number of times. Thus, when G[m-1] is switched, G[m-2]=1 and G[m-3] and below are all zero. Conversely, when this condition occurs then G[m-1] must be the next to be switched.

If m=1 then level 0 does not exist so we need another condition to look at. From the construction of the code we see that every second switch is of G[0]. Every switch changes the parity, thus, when counting up, G[0] will be switched next if P(G) is 0. When counting down, G[0] will be switched next if P(G) is 1.

Property P12: When counting an n-bit Gray Code in direction d (=0 for up, =1 for down), the next bit s to be switched is given by:

```
                P(G) = d                then s = 0
                P(G) = not d    then s is such that Gs-1=1 and Gi=0, i>s-1
```

This converts readily into a circuit if P(G) is known. In making a free-running counter the approach taken seems to be to provide an extra flip/flop that is by driven the clock and is used to select between the two alternatives. So, if flip/flop P is the parity flip-flop then the signals to or switch the counter flip/flops are as in the example in Figure 6.

Figure 6. Gray code up counter

In terms of an algorithm for generating the code, Boothroyd [Boo64] calculates the parity and finds the last set bit by a scan from left to right.

```
{ALGORITHM A4: INCREMENT/DECREMENT A GRAY CODE WORD G OF
WIDTH N}
procedure increment (var G: word; n: integer, d:bit);
        {d is direction, 0 up, 1 down}
        var p:bit;{parity}
                i, last1, switch: integer;
begin
p := 0;
last1 := n;
for i := n-1 downto 0 do
        if G[i] then begin
                        p := not p;
                        last1 := i;
                        end;
if p Å d
        then switch := 0
        else if last1 < n-1 then switch := last1+1
                                                else switch := n-1;
G[switch] := not G[switch];
end;
```

Misra [Mis75] gives a generation algorithm based on the concept of incrementing. However, he keeps track of the parity separately and maintains a stack of indices of bits that are 1, which give an algorithm that is very fast. [Er85] gives a coding of MisraÕs algorithm and incorporates some improvements.

6. Serial Addition

We have seen that the sign of the weight assigned to a bit in a Gray code word depends on the parity of the word at that bit, starting at the high-order end. However, most serial arithmetic operations must commence with the low-order end. If we know the entire parity of the word then it possible to commence serial operation from the low-order bits, because of the following property:

Property P13:

```
        P(G,n-1,k) = P(G,k-1,0) Å P(G)
```

We have already seen one example, the Gray code counter, where knowledge of the parity overall is maintained in an auxilliary flip-flop. In [Luc59], Harold Lucal proposed using a modified Gray Code where the parity is maintained as the least significant bit. Lucal showed how serial arithmetic could then be implemented.

It is clear that addition of Gray codes can be performed serially if we commence at the least significant end and know the parity of the two operands. We can work out the high-order parities at each bit as we go using property P12. From property P10 we can find the parity of the sum and maintain the parity of each bit, and we can propagate a carry. This is straightforward but involves carrying a large amount of information between bits. Lucal, however, showed that addition could be performed by carrying only two bits between each stage as follows:

```
{ALGORITHM A5: SERIAL ADDITION OF GRAY CODE WORDS}
procedure add (n: integer; A,B:word; PA,PB:bit;
                                 var S:word; var PS:bit; var CE, CF:bit);
var i: integer; E, F, T: bit;
{This adds the Gray code words A and B to form the Gray code word
S. The operand parities are PA and PB, the sum parity is PS. This
propagates two carry bits internally, E and F, and produces two
external carry bits CE and CF}
begin
        E := PA; F := PB;
        for i:= 0 to n-1 do begin {in parallel, using previous inputs}
                S[i] := (E and F) Å A[i] Å B[i];
                E         := (E and (not F)) Å A[i];
                F         := ((not E) and F) Å B[i];
                end;
        CE := E; CF := F;
end;
```

Gray Code Fundamentals Contd...: Tutorials Contd...
Back To Main Page: