Programming the Hilbert curve

Cite as: AIP Conference Proceedings **707**, 381 (2004); https://doi.org/10.1063/1.1751381
Published Online: 08 June 2004

John Skilling

Programming the Hilbert curve

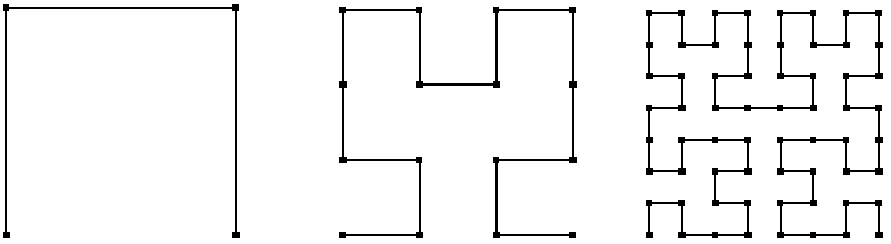
John Skilling

Maximum Entropy Data Consultants Ltd, Killaha East, Kenmare, Kerry, Ireland

Abstract. The Hilbert curve has previously been constructed recursively, using p levels of recursion of n-bit Gray codes to attain a precision of p bits in n dimensions. Implementations have reflected the awkwardness of aligning the recursive steps to preserve geometrical adjacency. We point out that a single global Gray code can instead be applied to all np bits of a Hilbert length. Although this "over-transforms" the length, the excess work can be undone in a single pass over the bits, leading to compact and efficient computer code.

INTRODUCTION

Space-filling curves, sometimes called Peano curves after their first investigator, are intricate lines which pass arbitrarily closely to all points of a multi-dimensional volume. There is no paradox, because distance along the line is stored with whatever extended precision is needed to locate the corresponding point accurately enough in space. We wish to use a specific curve, the "**Hilbert curve**" [1], which covers the interior of the n-dimensional hypercube $[0,2^p)^n$ of side 2^p with unit precision. Explicitly using an integer grid absolves us from formal technicalities of the continuum limit: we cannot store infinite precision in our physical hardware, so do not need to consider what might happen if we did. Distance H along the Hilbert curve is representable by a np-bit integer, decomposable as p **digits** of n bits each. The diagram below shows Hilbert curves in two dimensions for $p = 1, 2, 3$.



The Hilbert curve is as local as possible, in that an arc length along it corresponds, on average, to the least possible distance in space. This leads to a variety of possible uses, such as bandwidth reduction [2] and image storage [3]. More abstractly, altering the dimensionality of a problem at will, while largely preserving locality, is clearly an interesting tool. All this makes the efficient calculation of Hilbert coding a problem of interest.

CP707, *Bayesian Inference and Maximum Entropy Methods in Science and Engineering: 23rd International Workshop,* edited by G. Erickson and Y. Zhai

PREVIOUS PROGRAMS

Successive digits of H encode successive bits of each of the p-bit coordinates $x_0, x_1, x_2, \cdots$. Specifically, the top digit of H defines the top bits $A, B, C, \cdots$ of $x_0, x_1, x_2, \cdots$. Each of the 2^n possibilities can be written as a n-bit binary code, and can be visualised as the corner of a n-dimensional cube. Successive corners are to be adjacent, so the n-cube is traversed along its edges, using a "Gray code" to flip just one bit of the binary code at a time.

H	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Gray-code generator	D		D		D		D		D		D		D		D	
		C					C				C				C	
				B									B			
								A								
$ABCD$	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000

A Gray code is not unique, but this one illustrated for four dimensions is conveniently symmetric, as the typography indicates. After bit A has incremented, the remaining bits BCD for $H = 8, 9, 10, \ldots, 15$ mirror the pattern they took before for $H = 7, 6, 5, \ldots, 0$, so the second half of the pattern can be deduced from the first. Similarly, the second quarter $H = 4, 5, 6, 7$ can be deduced as a mirror of the first quarter $H = 3, 2, 1, 0$, and so on. Reversals are implemented by logical negation, so that writing this Gray-code program is straightforward.

H	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
Gray-code generator	D															
	$\longrightarrow$	C	$\longleftarrow$													
	$\longrightarrow$			B	$\longleftarrow$											
	$\longrightarrow$							A	$\longleftarrow$							
$ABCD$	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000

For example, $H = 13 = 1101_2$ decodes as:

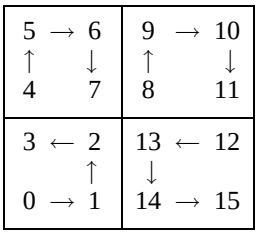
top bit of 1101 is $A = 1$ (ON), leaving lower bits 101 to be inverted as 010;
top bit of 010 is $B = 0$ (OFF), leaving lower bits 10 not needing inversion;
top bit of 10 is $C = 1$ (ON), leaving lower bit 0 to be inverted as 1;
final bit 1 is $D = 1$, completing $(A, B, C, D) = (1, 0, 1, 1)$.

There is also a trick, $ABCD = H \oplus (H/2)$, for doing up to 32 (or other computer word-length) decoding operations at once with a single exclusive-OR instruction $\oplus$.

The **displacement** of the above Gray code increments the first coordinate x_0. Any other coordinate could have been chosen, and we can use the dimensions in any order. Notionally, this gives $n!$ different Gray codes, though in practice we program only the principal one, and keep track of any permutation. Moreover, we can rotate the cube so that the path starts at any desired corner. To do this, exclusive-OR the $ABC\cdots$ digit with the digit of the entry corner. In all, this gives $2^n n!$ routes around the corners.

Lower-order bits of H define correspondingly lower-order bits $a, b, c, \ldots$ of the coordinates $x_0, x_1, x_2, \ldots$, again according to location on what is now a reduced hypercube. But the reduced cubes have to be oriented correctly, to ensure that the ends of succes-

sive segments are adjacent. Even in two dimensions, the problem is apparent, as the four reduced squares need three of the available orientations.



Previous work [4, 5, 6] details the coordinate exchanges and inversions required to accomplish this in n dimensions by top-down recursion.

Generally, before the first flip of a top bit (say C), we need an odd number of lower-order displacements between $000\cdots$ and $111\cdots$ of the corresponding dimension c, so that the lower bits become $111\cdots$, adjacent to the upper cube and ready to be incremented to $1000\cdots$. Then, between the next flips (alternately down and up), we need an even number of displacements to bring the path back next to the opposite cube.

At the beginning, we need the displacement in the first reduced cube to be along the last dimension: in the 4-dimensional illustration this is "*d*", ready for the first top-bit flip "*D*". This can be done by cycling the reduced dimensions once forwards, using *dabc* instead of *abcd*. Then to get ready for a "*C*" flip, we need to bring *c* to the front, which can be done by exchanging *c* and *d* to get *cabd*. According to the Gray code generator, the next couple of steps should be a reversal of these first two. With more than two dimensions, as here, we need to prepare for a "*B*" flip instead of a "*C*", so we exchange *b* and *c* to get *bacd* and *dacb* (instead of *cabd* and *dabc*). This pattern continues, doubling in length for each extra dimension. The next four steps should be a mirror-reversal of the first four, and in three dimensions that would suffice. With four dimensions, as here, we need to prepare for a "*A*" flip, so we exchange *a* and *b* to get permutations *dbca*, *abcd*, *cbad*, *dbac* (instead of *dacb*, *bacd*, *cabd*, *dabc*). The next eight steps should be a mirror-reversal of the first eight, and in four dimensions that does suffice, and no further exchange is needed.

H (top bits)	0 0000	1 0001	2 0010	3 0011	4 0100	5 0101	6 0110	7 0111	8 1000	9 1001	10 1010	11 1011	12 1100	13 1101	14 1110	15 1111
Hilbert generator	cycle $c \leftrightarrow d$ D															
			$b \leftrightarrow c$													
	$\longrightarrow$		C	$\longleftarrow$												
								$a \leftrightarrow b$								
	$\longrightarrow$			B		$\longleftarrow$										
	$\longrightarrow$							A		$\longleftarrow$						
HIGH	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
low	*dabc*	*cabd*	*bacd*	*dacb*	*dbca*	*abcd*	*cbad*	*dbac*	*dbac*	*cbad*	*abcd*	*dbca*	*dacb*	*bacd*	*cabd*	*dabc*

In this table for $n = 4$, the pattern of bit-flips $A, B, C, \ldots$ defines the HIGH bits of the coordinates x, according to the original Gray code. Lower-order bits of the coordinates, if any, are used with dimensions re-ordered acording to the "low" row of

the table, as defined by the generator instructions typed in lower case. That deals with the displacement axes.

One can also track the entry and exit corners of the sub-cubes, which are defined by the parities of the preceding operations, For example, each "*C*" flip and each "*c*" displacement flips the 3rd lower-order entry coordinate between $000\cdots$ and $111\cdots$. Together, the displacements and entry position define which of the $2^n n!$ routes should be taken round the sub-cube. But all this complexity leads to awkward computer code.

NEW PROGRAM

This paper points out that applying a global np-fold Gray code to all the bits of the Hilbert integer at once yields a result that is much closer to the desired spatial coordinates than is the original integer, in that a considerable amount of structure is correct or nearly correct. Let H be the original integer and $G = g_0 g_1 \cdots g_{np-1} = H \oplus (H/2)$ be the preliminary result, expressed as a binary integer. Collect these np bits into n preliminary p-bit integers $x_i = g_i g_{i+n} g_{i+2n} \cdots g_{i+(p-1)n}$ for $i = 0, 1, \cdots, n-1$. For example, the 4-bit Gray code analysed above can be collected into two preliminary 2-bit integers x_0 and x_1. The 16 points along the Hilbert line fall into this square as shown below.

6 ← 5 ↓ ↑ 7 4	9 → 10 ↑ ↓ 8 11
1 → 2 ↑ ↓ 0 3	14 ← 13 ↓ ↑ 15 12

For example, Hilbert point 13 decoded as $(A,B,C,D) = (1,0,1,1)$, which is collected as $x_0 = AC = 11_2 = 3$, $x_1 = BD = 01_2 = 1$, namely the point $(3,1)$ as shown. Clearly this is a reasonable first step. What has gone wrong is that the global decoding has implicitly done too many exchanges and inversions, so some of them need to be undone.

A single backwards pass through the higher-order $n(p-1)$ bits of the integers x does the exchanges and inversions needed to transform them into the desired coordinates.

```
for ( r = p-2, p-3, ..., 1, 0 )
    for ( i = n-1, n-2, ..., 1, 0 )
        if ( bit r of x_i is OFF )
            exchange low bits r+1, r+2, ..., p-1 of x_i and x_0
        else
            invert low bits of x_0
```

We scan the bits in reverse order. The first n bits (which would have had $r = p-1$ in the above pseudo-code) are lowest-order bits. There is no structure of yet lower order beneath them, so nothing can have gone wrong and nothing need be un-done. We already have the result we would want if p were only 1.

Afterwards, there are two possibilities. If a bit of integer x_i is OFF, then an extra exchange has to be reversed. Exchanging the lower-order bits of x_i for the corresponding

bits of x_0 sets the bits of x_i correctly, leaving x_0 ready for the next such exchange. The extreme case of a row of all n bits being OFF corresponds to cycling the dimensions once backwards, thus undoing an incorrect forwards cycle. If the bit of x_i was ON instead, then an extra inversion has to be un-done. The appropriate inversion is to the lower-order bits of x_0, which will then be moved to their correct position in the next exchange. After each row of n bits has been processed, the Hilbert decoding has been completed at that precision level, and when all rows have been processed the decoding is finished.

Consider the example above, with the four bits of $x_0 = AC, x_1 = BD$ to be processed in reverse order D, C, B, A. Bits D and C are lowest-order bits, implying no action. Bit B (the higher bit of x_1) is next. If it is OFF, then its lower bits (here D) need to be exchanged with the corresponding lower bits of x_0 (here C). Otherwise it is ON, and the lower bits of x_0 (here C), need to be inverted instead. Bit A (the higher bit of x_0) is next. If it is OFF, then its lower bits (here C) should be exchanged with the corresponding lower bits of x_0 (here C), though since these are in the same place, no action is needed. Otherwise it is ON, and the lower bits of x_0 (here C again), need to be inverted instead. Processing is complete.

H	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$ABCD$	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
x_0, x_1	0,0	0,1	1,1	1,0	1,2	1,3	0,3	0,2	2,2	2,3	3,3	3,2	3,0	3,1	2,1	2,0
D ——																
C ——																
B	OFF	OFF	OFF	OFF	ON	ON	ON	ON	ON	ON	ON	ON	OFF	OFF	OFF	OFF
Exchange	0,0	0,1	1,1	1,0									3,0	3,1	2,1	2,0
or Invert					1,2	1,3	0,3	0,2	2,2	2,3	3,3	3,2				
A	OFF	OFF	OFF	OFF	OFF	OFF	OFF	OFF	ON	ON	ON	ON	ON	ON	ON	ON
Exchange	0,0	1,0	1,1	0,1	0,2	0,3	1,3	1,2								
or Invert									2,2	2,3	3,3	3,2	3,1	2,1	2,0	3,0
x_0, x_1	0,0	1,0	1,1	0,1	0,2	0,3	1,3	1,2	2,2	2,3	3,3	3,2	3,1	2,1	2,0	3,0

The author's public-domain ANSI C module for p up to 32 and arbitrary n is listed in the Appendix. Passing from Hilbert distance to coordinates, or vice versa through the inverse procedure, costs only about 3 or 4 binary operations for each of the np bits.

ACKNOWLEDGMENTS

I thank Paul Goggans whose interest made me discover how my code worked. The code was in fact generated by polishing a top-down recursive program until it became so short and so fast and so unlike the original that it had to have a simple description. This is it.

REFERENCES

1. Hilbert, D. (1891), "Über die stetige abbildung einer linie auf ein flächenstück", Mathematische Annalen **38**, 459–460.
2. Bially, T. (1969), "Space-filling curves, their generation and their application to bandwidth reduction", IEEE Trans. Information Theory, **IT–15**, 658–664.

3. Song, Z. and Roussopoulos, N. (2002), “Using Hilbert curve in image storing and receiving”, Information Systems, **27(8)**, 523–536.
4. Butz, A.R. (1969), “Convergence with Hilbert’s space-filling curve”, J. Comput, Sys. Sci. **3**, 128–146.
5. Butz, A.R. (1971), “Alternative algorithm for Hilbert’s space-filling curve”, IEEE Trans. Computers, **20**, 424–426.
6. Lawder, A.K. (2000), “Calculation of mappings between one and n-dimensional values using the Hilbert space-filling curve”, Research report JL1/00, School of Computer Science and Information Systems, Birkbeck College, Univ. London.

Appendix: ANSI C code

```
//+++++++++++++++++++++++++++ PUBLIC-DOMAIN SOFTWARE ++++++++++++++++++++++++++
// Functions: TransposetoAxes  AxestoTranspose
// Purpose:   Transform in-place between Hilbert transpose and geometrical axes
// Example:   b=5 bits for each of n=3 coordinates.
//            15-bit Hilbert integer = A B C D E F G H I J K L M N O  is stored
//            as its Transpose
//                   X[0] = A D G J M               X[2]|
//                   X[1] = B E H K N    <------->      | /X[1]
//                   X[2] = C F I L O              axes |/
//                          high  low                  0------ X[0]
//            Axes are stored conventially as b-bit integers.
// Author:    John Skilling  20 Apr 2001 to 11 Oct 2003
//------------------------------------------------------------------------------
typedef unsigned int coord_t; // char,short,int for up to 8,16,32 bits per word
void TransposetoAxes( coord_t* X, int b, int n )  // position, #bits, dimension
{    coord_t  N = 2 << (b-1), P, Q, t;
     int  i;
// Gray decode by H ^ (H/2)
     t = X[n-1] >> 1;
     for( i = n-1; i >= 0; i-- ) X[i] ^= X[i-1];
     X[0] ^= t;
// Undo excess work
     for( Q = 2; Q != N; Q <<= 1 ) {
         P = Q - 1;
         for( i = n-1; i >= 0 ; i-- )
            if( X[i] & Q ) X[0] ^= P;                              // invert
            else{ t = (X[0]^X[i]) & P; X[0] ^= t; X[i] ^= t; }  }  // exchange
}
void AxestoTranspose( coord_t* X, int b, int n )  // position, #bits, dimension
{    coord_t  M = 1 << (b-1), P, Q, t;
     int  i;
// Inverse undo
     for( Q = M; Q > 1; Q >>= 1 ) {
         P = Q - 1;
         for( i = 0; i < n; i++ )
            if( X[i] & Q ) X[0] ^= P;                              // invert
            else{ t = (X[0]^X[i]) & P; X[0] ^= t; X[i] ^= t; }  }  // exchange
// Gray encode
     for( i = 1; i < n; i++ ) X[i] ^= X[i-1];
     t = 0;
     for( Q = M; Q > 1; Q >>= 1 )
         if( X[n-1] & Q ) t ^= Q-1;
     for( i = 0; i < n; i++ ) X[i] ^= t;
}
main()
{    coord_t X[3] = {5,10,20};  // any position in 32x32x32 cube
     AxestoTranspose(X, 5, 3);  // Hilbert transpose for 5 bits and 3 dimensions
     printf("Hilbert integer = %d%d%d%d%d%d%d%d%d%d%d%d%d%d%d = 7865 check\n",
         X[0]>>4 & 1, X[1]>>4 & 1, X[2]>>4 & 1, X[0]>>3 & 1, X[1]>>3 & 1,
         X[2]>>3 & 1, X[0]>>2 & 1, X[1]>>2 & 1, X[2]>>2 & 1, X[0]>>1 & 1,
         X[1]>>1 & 1, X[2]>>1 & 1, X[0]>>0 & 1, X[1]>>0 & 1, X[2]>>0 & 1);
}
```