Minimization of Mean Adjacent Different Values for Two-Dimensional Matrices

Matthew Ardizzone

Syosset High School

1 Abstract

A new area of mathematics has been introduced and studied. This area deals with the relative "closeness" or "smoothness" of matrices, by analyzing what is called the Mean Adjacent Difference value, which will be defined in the paper. It was proved that orientation does not affect this value for 2×2 matrices, and the best placement of 4 values in a matrix the best curve for minimizing MAD was found.

2 Introduction

This paper will attempt to introduce and explore a new and proposed institution of mathematics. This area deals with the relative "closeness" or "smoothness" of matrices, by analyzing what is called the Mean Adjacent Difference value. The Mean Adjacent Difference (MAD) is defined as the average distance in value between every pair of adjacent elements for a given matrix (For this paper, two elements are adjacent if they share at least one side). The MAD value can be found for higher dimensions as well, such as three- and four-dimensional matrices, but the focus of the first section of this paper will be strictly on two-dimensional ones. This paper will attempt two things: to find the placement, or relative positioning of a given set of values that minimizes MAD for 2 by 2 matrices, as well as to find the the curve that minimizes MAD.

Because the area has yet to have been worked with, there are few practical applications. However, one in particular lies in the field of sensory substitution. Sensory substitution is the means by which stimuli one sensory modality is converted into stimuli of another sensory modality. It has been shown to miraculously restore lost senses by using sensory information from a functioning sensory modality. One example of real-world implementation of sensory substitution is the vOICe: a device that consists of a camera, in-ear speakers and an advanced processor. In short, live video feed from the camera is converted into sound by scanning each frame from left to right while associating elevation with pitch and brightness with loudness. In theory this could lead to synthetic vision with truly visual sensations, by exploiting the neural plasticity of the human brain through training. The vOICe also acts as a research vehicle for the cognitive sciences to learn more about the dynamics of large-scale adaptive processes in the human brain. Neuroscience research has already shown that the visual cortex of even adult blind people can become responsive to sound, and sound-induced illusory flashes can be evoked in most adult deaf people. The vOICe technology may now build on this with live

video from unobtrusive camera glasses encoded in sound. The extent to which cortical plasticity allows for functionally relevant rewiring or unmasking of neural pathways in the human brain remains under investigation. Apart from functional relevance, inducing visual sensations through sound (like artificial synesthesia) could also prove of great psychological importance. Patients who have lost their ability to see have responded to visual cues that would otherwise be impossible, due to the efforts of the developers of the vOICe.

There are many different ways that an image can be converted to sound. Recall that the vOICe does so by scanning left to right, pixel-column by pixel-column. This results in a changing sound that must be played for about three seconds so the user can interpret the information. Another way this can be done is the entire image at a once. This would enable much faster and more continuous processing by the brain, which more closely mimics the human eye: faster, needing only to be played a fraction of a second; and more continuous, being only a single, constant sound. Relative "closeness" of matrices becomes important in the conversion step, where images are converted to sounds. Since an image is made of an array of pixel data, a line can be drawn through the array, hitting every pixel. That line can be unravelled, and the pixel data converted to sound frequencies to change the form of the data, image to sound. Finding the best curve to achieve this becomes an important question as it would be critical to the effectiveness and difficulty of use of the device.

One crucial criterion of the curve is the extent to which the sound it produces closely represents its origin picture: how "close" points on the frequency line are to points on the image. At this step that the MAD values play and important role. MAD values can be used to determine the "closeness" of a given curve, thus making it very useful in predicting beforehand the efficacy of the curve of used by a prototype device.

There are reasons why the creators of the vOICe opted not to use this method and instead chose to divide the image. For example, it requires much more processing power to convert entire images than it does to convert just part of it at the same rate. It would likely overwhelm most standard computers or processors of reasonable size. In addition, the difficulty associated with training the brain to comprehend this complex sound would be troubling. The users of the vOICe were able to "see" after just 70 hours of training. The users of the device that converts the image in its entirety would likely need four or five times as much training to achieve equal familiarity with the device.

3 Minimizing MAD with value placement

Recall that MAD value is defined as the average distance in value between every pair of adjacent elements for a given matrix (For this paper, two elements are adjacent if they share at least one side). For example, given any matrix, of size $n \times n$:

$$A = \begin{array}{|c|c|c|c|} \hline a_{1,1} & a_{1,2} & ... & a_{1,n} \\ \hline a_{2,1} & a_{2,2} & ... & a_{2,n} \\ ... & ... & & ... \\ a_{n,1} & a_{n,2} & ... & a_{n,n} \\ \hline \end{array}$$

The MAD value (MAD_A) is equal to:

$$\frac{\sum_{i=1}^{n-1}(\sum_{j=1}^{n-1}|a_{i,j}-a_{i+1,j}|+|a_{i,j}-a_{i,j+1}|)+\sum_{x=1}^{n-1}|a_{x,n}-a_{x+1,n}|+\sum_{y=1}^{n-1}|a_{n,y}-a_{n,y+1}|}{2n^2-2n} \quad \text{(Equation 1)}$$

(Note that the matrix above is not represented in its usual format, and instead is tabulated. This was done because the matrices in this paper are often shown in conjunction with other many other matrices (see figure 1), so when represented in their traditional format, such figures would be unorganized and confusing.)

This formula can be simplified for 2×2 matrices. Given 2×2 matrix A, containing values a, b, c, and d:

$$A = \begin{array}{|c|c|} \hline \text{a} & \text{b} \\ \hline \text{d} & \text{c} \\ \hline \end{array}$$

MAD value is strictly defined as:

$$MAD_A = \frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} \quad \text{(Equation 2)}$$

For example, if we were to find the MAD value for this 2×2 array:

$$A = \begin{array}{|c|c|} \hline 3.1 & 0.5 \\ \hline -6.8 & 1.0 \\ \hline \end{array}$$

MAD value would be as follows:

$$MAD_A = \frac{|a-b|+|b-c|+|c-d|+|d-a|}{4}$$
$$MAD_A = \frac{|3.1-0.5|+|0.5-1.0|+|1.0-(-6.8)|+|-6.8-3.1|}{4}$$
$$MAD_A = \frac{|2.6|+|-0.5|+|7.8|+|-9.9|}{4}$$
$$MAD_A = \frac{2.6+0.5+7.8+9.9}{4}$$
$$MAD_A = \frac{20.8}{4}$$
$$MAD_A = 5.2$$

As previously stated, the first goal of this paper will be to find the placement, or relative positioning of a given set of values that minimizes MAD for 2×2 matrices. Ideally, elements of most similar values should be placed near each other to minimize the MAD value; however because MAD value wraps around the matrix and is not linear, it is not immediately obvious what the best positioning is.

For a matrix of size $n \times n$, there are $(n^2)!$ possible ways to place the n^2 numbers that it would contain. However, it is possible that some permutations constitute the same MAD values as others. This conjecture, if true, will expedite the process of many tasks in the future of this paper, so before all else, it will be proven.

3.1 Proof that Orientation does not affect MAD value

For a 2 by 2 matrix, containing the four values a, b, c and d, there are 4!, or 24 ways to place them, pictured below.

(1a)	(1b)	(1c)	(1d)	(1e)	(1f)	(1g)	(1h)
a b d c	d a c b	c d b a	b c a d	a d b c	d c a b	c b d a	b a c d
(2a)	(2b)	(2c)	(2d)	(2e)	(2f)	(2g)	(2h)
a c b d	c d a b	d b c a	b a d c	a b c d	c a d b	d c b a	b d a c
(3a)	(3b)	(3c)	(3d)	(3e)	(3f)	(3g)	(3h)
a d c b	d b a c	d c b a	c a d b	a c b d	b a d c	d b c a	c d a b

Figure 1: This table shows every possible square arrange of four values. This table is organized in such a way that every matrix can be mapped onto any other matrix in the same row through a

series of at least one 1 rotation or reflection, and as a result, they have the same MAD value. Other than that, they are in no particular order.

Despite the large quantity of arrangements, there are only three unique MAD values among them. As stated in the caption, the matrices in the figure above have been arranged in such a way that each of the eight matrices in the same row can be mapped onto any another through a series of transformations: either a reflection, a rotation, or both. As a result, all eight share the same MAD value. The following is a proof of that statement.

Because each row is interconnected by a series of transformations, once the conjecture is proven for one row it can be applied to the other two. We will consider just the first row.

$$MAD_{1a} = MAD_{1a}$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|a-b|+|b-c|+|c-d|+|d-a|}{4}$$

$$MAD_{1a} = MAD_{1b}$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|d-a|+|a-b|+|b-c|+|c-d|}{4}$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|a-b|+|b-c|+|c-d|+|d-a|}{4}$$

$$MAD_{1a} = MAD_{1c}$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|c-d|+|d-a|+|a-b|+|b-c|}{4}$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|a-b|+|b-c|+|c-d|+|d-a|}{4}$$

$$MAD_{1a} = MAD_{1d}$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|b-c|+|c-d|+|d-a|+|a-b|}{4}$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|a-b|+|b-c|+|c-d|+|d-a|}{4}$$

$$MAD_{1a} = MAD_{1e}$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|a-d|+|d-c|+|c-b|+|b-a|}{4}$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|-(d-a)|+|-(c-d)|+|-(b-c)|+|-(a-b)|}{4}$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|d-a|+|c-d|+|b-c|+|a-b|}{4}$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|a-b|+|b-c|+|c-d|+|d-a|}{4}$$

$$MAD_{1a} = MAD_{1f}$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|d-c|+|c-b|+|b-a|+|a-d|}{4}$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|-(c-d)|+|-(b-c)|+|-(a-b)|+|-(d-a)|}{4}$$

$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|c-d|+|b-c|+|a-b|+|d-a|}{4}$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|a-b|+|b-c|+|c-d|+|d-a|}{4}$$

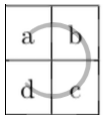
(Proof 1)

$$MAD_{1a} = MAD_{1g}$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|c-b|+|b-a|+|a-d|+|d-c|}{4}$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|-(b-c)|+|-(a-b)|+|-(d-a)|+|-(c-d)|}{4}$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|b-c|+|a-b|+|d-a|+|c-d|}{4}$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|a-b|+|b-c|+|c-d|+|d-a|}{4}$$

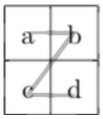
$$MAD_{1a} = MAD_{1h}$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|b-a|+|a-d|+|d-c|+|c-b|}{4}$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|-(a-b)|+|-(d-a)|+|-(c-d)|+|-(b-c)|}{4}$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|a-b|+|d-a|+|c-d|+|b-c|}{4}$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|a-b|+|b-c|+|c-d|+|d-a|}{4}$$

3.2 2x2 Matrices

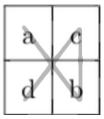
This section will experiment with the placement of values in a 2×2 matrix. Given matrix A, containing the values a, b, c and d, such that $a \neq b \neq c \neq d$, there are only three unique (same MAD value) ways to arrange them, pictured below:



(1) "c" Arrangement



(2) "z" Arrangement



(3) "x" Arrangement

Figure 2: Pictured above are the three unique (same MAD value) ways to arrange the values a, b, c and d. The epithets below each come from the shape that is drawn when a line is drawn connecting the values in increasing or decreasing order (either way will produce the same shape). Note that the "z" arrangement has the special condition where the maximum value, a, is opposite the minimum value, b, whereas the other two have the minimum and maximum values adjacent to each other. Also, since these arrangements can be rotated, it is possible that they could take the shape of other letters. For example, the "c" arrangement could be turned into a "u" arrangement if rotated 90 degrees, and the "z" arrangement could be turned into an "N" arrangement. In cases like that, the arrangements will be referred to with their original name to promote consistency.

That was shown in the previous section. Now, since they are unequal, there must be one of the three that is less than the other two. This section will attempt which of the three that is.

$$MAD_1 = \frac{|a-b|+|b-c|+|c-d|+|d-a|}{4}$$

$$MAD_2 = \frac{|a-b|+|b-d|+|d-c|+|c-a|}{4}$$

$$MAD_3 = \frac{|a-c|+|c-b|+|b-d|+|d-a|}{4}$$

Given that $a > b > c > d$, a few logical deductions about their differences can be made, shown below:

$$|a-c| = |a-b| + |b-c| \quad \text{(Equation 3)}$$

$$|b-d| = |b-c| + |c-d| \quad \text{(Equation 4)}$$

$$|a-d| = |a-b| + |b-c| + |c-d| \quad \text{(Equation 5)}$$

And if we compare the MAD values listed previously we can draw some interesting conclusions about their relative magnitudes.

I. Comparison between MAD_1 and MAD_2:

$$MAD_1 = MAD_2$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|a-b|+|b-d|+|d-c|+|c-a|}{4}$$
$$\frac{|b-c|+|d-a|}{4} = \frac{|b-d|+|c-a|}{4}$$
$$\frac{|b-c|+|a-b|+|b-c|+|c-d|}{4} = \frac{|b-c|+|c-d|+|a-b|+|b-c|}{4}$$
$$0 = 0$$

II. Comparison between MAD_2 and MAD_3:

$$MAD_2 = MAD_3$$
$$\frac{|a-b|+|b-d|+|d-c|+|c-a|}{4} = \frac{|a-c|+|c-b|+|b-d|+|d-a|}{4}$$
$$\frac{|a-b|+|d-c|}{4} = \frac{|c-b|+|d-a|}{4}$$
$$\frac{|a-b|+|d-c|}{4} = \frac{|c-b|+|a-b|+|b-c|+|c-d|}{4} \quad \text{(Proof 2)}$$
$$\frac{0}{4} = \frac{|c-b|+|b-c|}{4}$$
$$\frac{0}{4} = \frac{2|b-c|}{4}$$
$$0 = \frac{|b-c|}{2}$$

III. Comparison between MAD_1 and MAD_3:

$$MAD_1 = MAD_3$$
$$\frac{|a-b|+|b-c|+|c-d|+|d-a|}{4} = \frac{|a-c|+|c-b|+|b-d|+|d-a|}{4}$$
$$\frac{|a-b|+|c-d|}{4} = \frac{|a-c|+|b-d|}{4}$$

$$\frac{|a-b|+|c-d|}{4} = \frac{|a-b|+|b-c|+|b-c|+|c-d|}{4}$$
$$\frac{0}{4} = \frac{|b-c|+|b-c|}{4}$$
$$\frac{0}{4} = \frac{2|b-c|}{4}$$
$$0 = \frac{|b-c|}{2}$$

In conclusion of these three comparisons, we find out out here that the "c" and the "z" arrangements coincidentally have the same MAD value for all values of a, b, c and d within that satisfy the aforementioned relationships. We also find that the MAD value for the "x" arrangement is less than the other two by a difference of $\frac{|b-c|}{2}$. This is likely attributable to the combination of differences found in each arrangement. For the circular arrangement, it is true that this arrangement contains all three of the smallest possible differences ($|a-b|$, $|b-c|$, and $|c-d|$), however it also contains the largest difference ($|a-d|$). This is a consequence of the "circular wrapping" of MAD values. The "z" arrangement takes care of this issue by putting the minimum value on the opposite side of the matrix as the maximum value. While this eliminates the presence of the largest difference, it eliminates the presences of the three smallest as well; they are replaced with the two intermediate differences ($|a-c|$ and $|b-d|$). According to this pattern of consideration, the inferiority of "x" then becomes apparent. It contains the largest difference, two intermediate differences and only one of the three smallest differences ($|b-c|$).

Furthermore, since the "c" and "z" arrangements have the same MAD value, and they can't be mapped onto each other, then it can be deduced that the first proof (Proof 1) is not biconditional.

4 Minimizing MAD with a curve

After experimenting in the placement of single values in 2 by 2 arrays, it becomes important to consider more real-world applications of MAD values in larger matrices. Mentioned in the introduction was a process for converting a two-dimensional images to a one-dimensional line, however no method for achieving this was brought up. One way would be to use space-filling curves. A space-filling curve is type of fractal that maps a 1-dimensional space onto a higher-dimensional space, e.g., the unit interval onto the unit square. In laymen's terms, it is a line (1D) that is bent and crooked to such and extent that it covers every point in a 2D space. This paper will use existing space-filling curves to solve the proposed problem, in particular the Hilbert Curve and the Peano Curve.

Using those two curves and one other that will be mentioned later, this section of the paper will

attempt to find the curve that, when each location is replaced with a number that is equal to the distance along the curve to the starting point, results in the lowest MAD value. In the example below, an arbitrary curve is drawn through a two-dimensional, 3×3 matrix, with ending point X and starting point O, to demonstrate this concept:

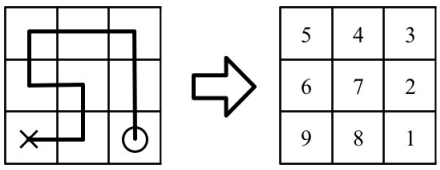


Figure 3: Given an empty 3×3 matrix, a line can be drawn connecting the center of every empty cell by connecting adjacent elements only (left matrix). Thus, a set of integers can be used to fill the same matrix corresponding to a curve, such that each integer represents the distance along the curve to the starting point (right).

Although there may be curves that cross over each other with low MAD values, for this paper, only curves that do not cross over each other will be investigated. This will be checked by verifying the following condition: For every number, k, both k + 1 and k - 1 must be in a location adjacent to k (This creates a continuous line of numbers, if they are connected in order, and prevents the path they create from crossing over itself).

To find the curve that minimizes MAD value, three curves were tested. Two of which are existing space-filling curves up to finite orders, in particular, Hilbert curve and the Peano curve; the other is a simple back-and-forth weaving pattern, given the name "Snake" curve, as it resembles a well-played game of the popular arcade title *Snake*. All three are pictured on the subsequent page.

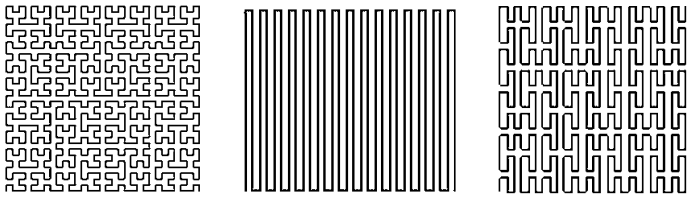


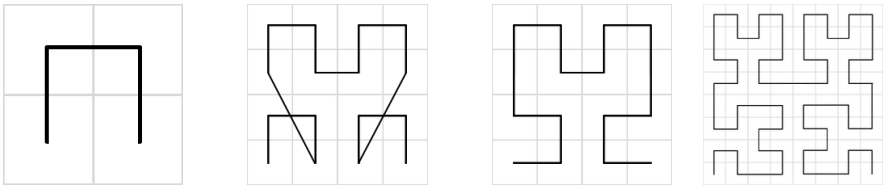
Figure 4: This image shows the three space-filling curves whose MAD values will be investigated. Pictured on the left is the Hilbert curve, in the middle is the snake curve and on the right is the Peano curve.

For each of the follow sections, first will be a brief overview of how each curve is created. Second will be a demonstration of how the MAD value was found first by hand for a few iterations of the

curves and later by computer software. Finally an equation relating the order of each curve to it MAD value will be found using an exponential regression program on a standard graphing calculator. The first to be examined will be the Hilbert Curve.

4.1 The Hilbert Curve

The first curve that will be discussed will be the Hilbert Curve. German mathematician David Hilbert created the curve that bears his name in the early 1900's. It is drawn entirely by connecting the centers of cells of a grid. Like all fractals, it is generated in iterations, shown below:



Iteration 1 Itermediary Iteration Iteration 2 Iteration 3

Figure 5: This image shows the step-by-step creation of Hilbert curve. The initial configuration is simply an inverted U-shape connecting the centers of a 2×2 grid (left). The second iteration is formed by subdividing each grid cell into quarters, duplicating the first iteration into each of the four corners, and connecting them (middle-left). However, to avoid the awkward crossover and messiness, the bottom-left sub-curve is reflected over the line $y = x$, and the bottom-right sub-curve is reflected over the line $y = -x$ (middle-right), assuming the origin at the center of the image. Shown all the way to the right is the third iteration in its completed form.

At each stage, the curve starts in the lower left corner and ends in the lower right corner, never touching or crossing itself. To convert each iteration to a matrix of numbers, we simply establish the lower left as the starting point, the lower right as the ending point and all the values in between.

(Iteration 1)

2	3
1	4

(Iteration 2)

6	7	10	11
5	8	9	12
4	3	14	13
1	2	15	16

(Iteration 3)

22	23	26	27	38	39	42	43
21	24	25	28	37	40	41	44
20	19	30	29	36	35	46	45
17	18	31	32	33	34	47	48
16	13	12	11	54	53	52	49
15	14	9	10	55	56	51	50
2	3	8	7	58	57	62	63
1	4	5	6	59	60	61	64

Figure 6: This table shows the first three iterations of the Hilbert curve in matrix form. Each curve starts with the number 1 and increases along the curve. Note that the size increases exponentially as iteration increases. This has to do with the fact that each iteration is a product of 4 copies of the previous iteration. Thus we can write a formula represent the size of a Hilbert matrix, given order, n: $2^n \times 2^n$; as well as the number of cells given order, n: $(2^n)^2$, which simplifies to 4^n.

In this form, the MAD value can be found. The system devised to find every pair of adjacent numbers and take their difference was to go in order starting from 1 and working the way through the curve, taking the difference with the number below it (if it exists) and the difference with the number to its right (if it exists). Upwards and leftwards differences were excluded because this would cause several differences to be counted for twice, as the upwards differences of one number is referring to the same difference as the downwards difference of the value directly above it. Rightwards and downwards are not the only two directions that will work though, they were chosen arbitrarily because it did not affect the system; differences taken carefully in any two non-opposite directions traced around the curve will eventually refer to all adjacent pairs, provided that missing pairs are accounted for properly.

There are cases where a value would only have one or zero downwards or rightwards adjacent pairs. Each number at the bottom edge only has a rightward pair (no numbers beneath it), every number on the right edge only has a downward pair (no numbers to its right), and the number in the bottom right corner has no adjacent number to its right or beneath it at all. Those spots in the table were represented with a dash and not included in the finding of the MAD value. After all the difference were taken, they were added together to produce the Sum of Adjacent Differences (SAD). Finally, this number was divided by the number of adjacent pairs. In fact, a formula can be written to represent this quantity. Given the edge length x, the number of adjacent pairs is equal to $2x^2 - 2x$, as there

is one downwards and one adjacent adjacent pair for every value (that's where the $2x^2$ comes from), except at the bottom edge ($-x$) and at the right edge ($-x$ again). Where they overlap, the value has no downwards or rightwards adjacent pairs, which conveniently, it doesn't. Below is the application of this system for finding MAD, applied for the first two iterations of the Hilbert curve.

Iteration 1		
Box Number	Downwards Difference	Rightwards Difference
1	-	$\|1-4\|=3$
2	$\|2-1\|=1$	$\|2-3\|=1$
3	$\|3-4\|=1$	-
4	-	-
Total	2	4

Table 1: This table was used to organize all the adjacent differences for an iteration 1 Hilbert curve.

$$\text{Sum of Adjacent Differences} = 2+4=6$$

$$\text{Number of adj. pairs } (x=2) = 2x^2-2x = 2(2)^2-2(2) = 4$$

$$MAD_{i1} = 6/4 = 1.5$$

Iteration 2		
Box Number	Downwards Difference	Rightwards Difference
1	-	$\|1-2\|=1$
2	-	$\|2-15\|=13$
3	$\|3-2\|=1$	$\|3-14\|=11$
4	$\|4-1\|=3$	$\|4-3\|=1$
5	$\|5-4\|=1$	$\|5-8\|=3$
6	$\|6-5\|=1$	$\|6-7\|=1$
7	$\|7-8\|=1$	$\|7-10\|=3$
8	$\|8-3\|=5$	$\|8-9\|=1$

Iteration 2 (contd.)		
Box Number	Downwards Difference	Rightwards Difference
9	$\|9-14\|=5$	$\|9-12\|=3$
10	$\|10-9\|=1$	$\|10-11\|=1$
11	$\|11-12\|=1$	-
12	$\|12-13\|=1$	-
13	$\|13-16\|=3$	-
14	$\|14-15\|=1$	$\|14-13\|=1$
15	-	$\|15-16\|=1$
16	-	-
Total	24	40

Table 2: This table was used to organize all the adjacent differences for an iteration 2 Hilbert curve.

$$\text{Sum of Adjacent Differences} = 24 + 40 = 6$$

$$\text{Number of adj. pairs } (x = 4) = 2x^2 - 2x = 2(4)^2 - 2(4) = 24$$

$$MAD_{i2} = 64/24 = 2.67$$

At orders larger than 2, finding the MAD value by hand becomes very tedious, so to make the process more efficient, MAD values for the Hilbert curve were calculated using computer software (see appendices 1 and 2) up to order 11, shown below:

MAD Values for Hilbert Curves											
Order	1	2	3	4	5	6	7	8	9	10	11
MAD	1.5	2.67	5.07	9.92	19.62	39.05	77.91	155.62	311.05	621.9	1243.62

Table 3: This table was used to organize all the MAD values calculated by the computer for Hilbert curves up to order 11.

Using the values above, an exponential regression with remarkable regression analysis has been computed, where x represents the order of the curve and y represents the MAD value:

$$y = a * b^x$$
$$a = 0.6866610567$$
$$b = 1.970990062 \qquad \text{(Equation 6)}$$
$$r^2 = 0.999629602$$
$$r = 0.9998147838$$

Even though it may be gratifying that the r^2 value is only 0.0002 off from a perfect regression, it still is not a perfect regression. There were only 11 data points, so perhaps the equation may be farther off from actual than the regression values suggest. In future research, finding a formula to find the exact MAD value for a Hilbert curve will be attempted.

4.2 The Snake Curve

This curve originates from the popular video game from the 1970s, Snake. In this game, the player maneuvers a line which grows in length, with the line itself being the primary obstacle. The snake increases in speed as it gets longer, and there's only one life; one mistake means starting from the beginning. In order to deal with the ever-increasing length of the tail, players must wind back and forth to conserve space for the actual game play. To form this curve for the purposes of this paper, it will be started in the bottom left corner, going up until the edge is reached, moving one cell to the right and continuing in the opposite direction. This pattern is continued until there is no more rightwards space, resulting in a snake curve, drawn below:

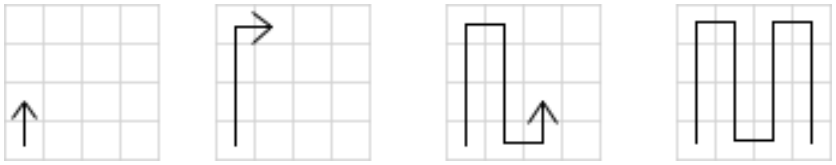


Figure 7: This picture shows the formation of an order 3 Snake Curve. Order 3 was chosen for the demonstration because it had the best balance between size and demonstration of the back-of-forth nature of the curve. In the first image on the left, the start location and direction are indicated, and the path is taken to the top of the image (second image from the left), where, in order to continue sprawling, the "snake" must move over to the right one square. In the third image, this winding pattern is continued, and in the fourth image, the final curve is drawn, completed.

Below are the first five iterations of the snake curve:

(Iteration 1) (Iteration 2) (Iteration 3) (Iteration 4) (Iteration 5)

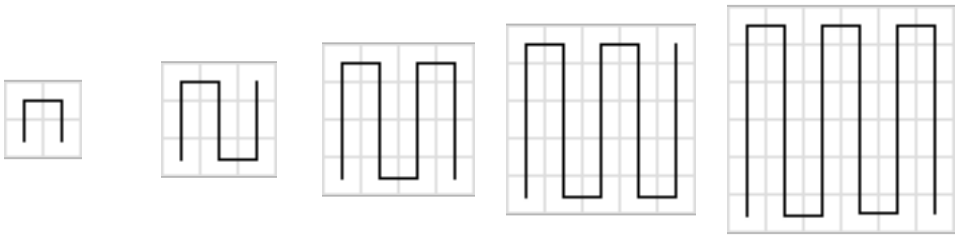


Figure 8: This image shows the first five iterations of the Snake curve. Interestingly, the first two iterations are not unique. The first iteration is identical to the first iteration of the Hilbert curve. The second iteration is identical to the first iteration of the Peano curve. It is not until the third iteration do we find a curve that unlike either of the two already existing space filling curves.

Using the same system as with the Hilbert curve, the resulting numerical matrix form of each curve can be created.

(Iteration 1)

2	3
1	4

(Iteration 2)

3	4	9
2	5	8
1	6	7

(Iteration 3)

4	5	12	13
3	6	11	14
2	7	10	15
1	8	9	16

(Iteration 4)

5	6	15	16	25
4	7	14	17	24
3	8	13	18	23
2	9	12	19	22
1	10	11	20	21

Figure 9: Even though the graphical curves are shown up to iteration five, the matrix forms will only be shown up to iteration four due to space restrictions and to avoid redundancy. Each curve starts with the number 1 and increases along the corresponding curve in figure 8 (not shown above). Note that the size increases linearly as iteration increases, unlike the Hilbert curve, which grew exponentially. We can write a simple formula represent the number of rows/columns of a Snake matrix, given iteration, n: $n + 1$; as well as the number of cells given order, n: $(n + 1)^2$.

As with the Hilbert curve, the MAD values of the first few iterations will be found by hand:

Iteration 1		
Box Number	Downwards Difference	Rightwards Difference
1	-	$\|1-4\|=3$
2	$\|2-1\|=1$	$\|2-3\|=1$
3	$\|3-4\|=1$	-
4	-	-
Total	2	4

Table 4: This table was used to organize all the adjacent differences for an iteration 1 Snake curve.

$$\text{Sum of Adjacent Differences} = 2+4=6$$

$$\text{Number of adj. pairs } (x=2) = 2x^2-2x = 2(2)^2-2(2) = 4$$

$$MAD_{i1} = 6/4 = 1.5$$

Iteration 2		
Box Number	Downwards Difference	Rightwards Difference
1	-	$\|1-6\|=5$
2	$\|2-1\|=1$	$\|2-5\|=3$
3	$\|3-2\|=1$	$\|3-4\|=1$
4	$\|4-5\|=1$	$\|4-9\|=5$
5	$\|5-6\|=1$	$\|5-8\|=3$
6	-	$\|6-7\|=1$
7	-	-
8	$\|8-7\|=1$	-
9	$\|9-8\|=1$	-
Total	6	18

Table 5: This table was used to organize all the adjacent differences for an iteration 2 Snake curve.

$$\text{Sum of Adjacent Differences} = 6+18=24$$

$$\text{Number of adj. pairs } (x=3) = 2x^2-2x = 2(3)^2-2(3) = 12$$

$$MAD_{i2} = 24/12 = 2.0$$

Iteration 3		
Box Number	Downwards Difference	Rightwards Difference
1	-	$\|1-8\|=7$
2	$\|2-1\|=1$	$\|2-7\|=5$
3	$\|3-2\|=1$	$\|3-6\|=3$
4	$\|4-3\|=1$	$\|4-5\|=1$
5	$\|5-6\|=1$	$\|5-12\|=7$
6	$\|6-7\|=1$	$\|6-11\|=5$
7	$\|7-8\|=1$	$\|7-10\|=3$
8	-	$\|8-9\|=1$
9	-	$\|9-16\|=7$
10	$\|10-9\|=1$	$\|10-15\|=5$
11	$\|11-10\|=1$	$\|11-14\|=3$
12	$\|12-11\|=1$	$\|12-13\|=1$
13	$\|13-14\|=1$	-
14	$\|14-15\|=1$	-
15	$\|15-16\|=1$	-
16	-	-
Total	12	48

Table 6: This table was used to organize all the adjacent differences for an iteration 3 Snake curve.

$$\text{Sum of Adjacent Differences} = 12 + 48 = 60$$

$$\text{Number of adj. pairs } (x = 4) = 2x^2 - 2x = 2(4)^2 - 2(4) = 24$$

$$MAD_{i2} = 60/24 = 2.5$$

Also as with the Hilbert curve, working with matrices larger than 4×4 becomes tedious and inefficient, so computer software (see appendices 1 and 2) was used to compute the MAD values for

orders up to 47 (similar size to order 11 of the Hilbert curve):

MAD Values for Snake Curves													
Order	1	3	5	7	9	11	13	15	17	19	21	23	25
MAD	1.5	2.5	3.5	4.5	5.5	6.5	7.5	8.5	9.5	10.5	11.5	12.5	13.5

MAD Values for Snake Curves (contd.)											
Order	27	29	31	33	35	37	39	41	43	45	47
MAD	14.5	15.5	16.5	17.5	18.5	19.5	20.5	21.5	22.5	23.5	24.5

Table 7: This table was used to organize all the MAD values calculated by the computer for Snake curves up to order 47. Even orders were omitted to save space in the paper.

Using the values above, another remarkable regression has been computed. Surprisingly, the MAD value turns out to be perfectly linear, following the simple equation below:

$$\text{MAD, given order n} = n/2 + 1 \qquad \text{(Equation 7)}$$

Like the Hilbert curve, there is no immediately clear reason for this. Finding the reason will be added to the list of research to do in the future.

4.3 The Peano Curve

The Peano curve is the first space-filling to ever have been created, by Giuseppe Peano in 1890. Peano was motivated by a dicovery of Georg Cantor, recent at the time, that the unit interval and the unit square have the same cardinality; Peano's curve is almost a proof of that. This picture shows the steps by which a Peano curve is generated.

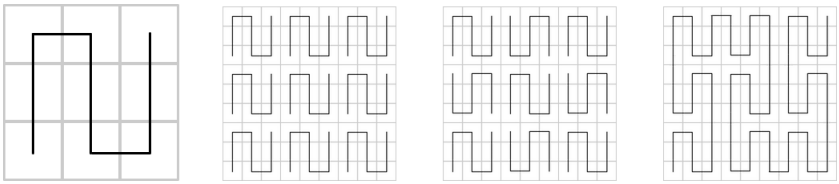


Figure 10: The Peano curve is constructed in a similar way to the Hilbert Curve. It's first iteration is identical to an order 3 snake curve, connecting centers of a 3×3 grid (far left). To get to the next iteration, each cell in the grid is divided into ninths and the current image is copied and pasted to replace into each available 3×3 cell (middle-left). Then, each copy in the center top, middle-left, middle-right and center bottom are flipped so that the connections between the copies do not result in any intersections. Then the final end-to-start connections are made. This pattern is continued indefinitely, resulting in a continuous 2D plane.

Below are the first three iterations of the Peano curve.

(Iteration 1) (Iteration 2) (Iteration 3)

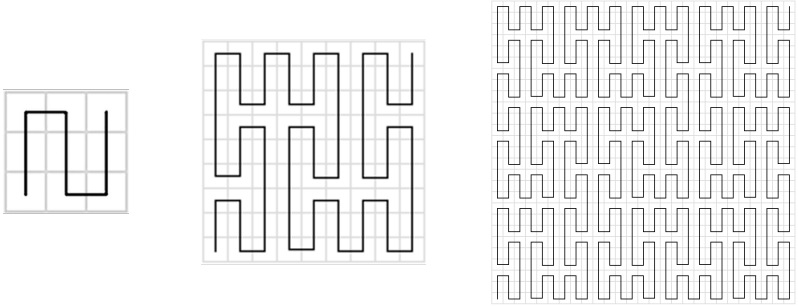


Figure 11: This image shows the first three iterations of the Peano curve, however, they are not to relative scale, as is made clear by the decreasing scales of the gray backdrop grid. In order to fit the

Using the same system as with the Hilbert curve, the resulting numerical matrix form of each curve can be created.

(Iteration 1)

3	4	9
2	5	8
1	6	7

(Iteration 2)

21	22	27	28	33	34	75	76	81
20	23	26	29	32	35	74	77	80
19	24	25	30	31	36	73	78	79
18	13	12	43	42	37	72	67	66
17	14	11	44	41	38	71	68	65
16	15	10	45	40	39	70	69	64
3	4	9	46	51	52	57	58	63
2	5	8	47	50	53	56	59	62
1	6	7	48	49	54	55	60	61

This curve grows in size very quickly, so only the MAD value of the first iteration will be found by hand. Unfortunately, not much information about the curve as a whole will can be determined from just one, so instead this will serve as a check on the computer software.

Iteration 1		
Box Number	Downwards Difference	Rightwards Difference
1	-	$\|1-6\|=5$
2	$\|2-1\|=1$	$\|2-5\|=3$
3	$\|3-2\|=1$	$\|3-4\|=1$
4	$\|4-5\|=1$	$\|4-9\|=5$
5	$\|5-6\|=1$	$\|5-8\|=3$
6	-	$\|6-7\|=1$
7	-	-
8	$\|8-7\|=1$	-
9	$\|9-8\|=1$	-
Total	6	18

Table 8: This table was used to organize all the adjacent differences for an iteration 1 Peano curve.

$$\text{Sum of Adjacent Differences} = 6 + 18 = 24$$

$$\text{Number of adj. pairs } (x = 3) = 2x^2 - 2x = 2(3)^2 - 2(3) = 12$$

$$MAD_{i2} = 24/12 = 2.0$$

As computed by computer software:

MAD Values for Peano Curves							
Order	1	2	3	4	5	6	7
MAD	2.0	5.67	16.74	49.96	149.65	448.73	1345.96

Table 9: This table was used to organize all the MAD values calculated by the computer for Snake curves up to order 7.

Using the values above, a third remarkable regression was found, where x represents the order of the curve and y represents the MAD value:

$$y = a * b^x$$

$$a = 0.6522616866$$

$$b = 2.968767158 \qquad \text{(Equation 8)}$$

$$r^2 = 0.9999371282$$

$$r = 0.9999685636$$

Like the Hilbert curve, there is no immediately clear reason for this. Finding the reason will be added to the list of research to do in the future.

5 Data Analysis

In the previous sections, the formula for MAD values of three space filling curves were found, however each of them mean very little on their own. This section will compare the equations and finally answer the question of which curve minimizes MAD value.

These equations use order for their x-values, but each of the three curves have difference sizes for the same order. To make things worse, there is no order of a Peano curve that has the same size as *any* order of the Hilbert curve. The dimensions of all iterations of the Peano curve are powers of 3, while the dimensions of all iterations of the Hilbert curve are power of 2, and there is no power of 2 that is equal to any power of 3. This has to do with the simple fact that all the powers of 3 are odd and all the powers of 2 are odd (odd $\times$ odd = odd and even $\times$ even = even). To avoid this issue, the equations that were found in previous section will need to be adjusted for the differences in sizes. Below are the three equations to find MAD value given order, n:

$$MAD_{Hilbert} = 0.6866610567 \times 1.970990062^n$$
$$MAD_{Snake} = 0.5n + 1$$
$$MAD_{Peano} = 0.6522616866 \times 2.968767158^n$$

To make these equations comparable, we must first change the input value to be side length instead of order. This can be done by replacing n with the inverse of the side length generating formulas from section 4.

$$length_{Hilbert} = 2^n \qquad length_{Snake} = n + 1 \qquad length_{Peano} = 3^n$$
$$n = log_2(length_{Hilbert}) \quad n = lengthSnake - 1 \quad n = log_3(length_{Peano})$$

After the proper substitutions, the following formulas can be written, where x is the length of the side of the curve:

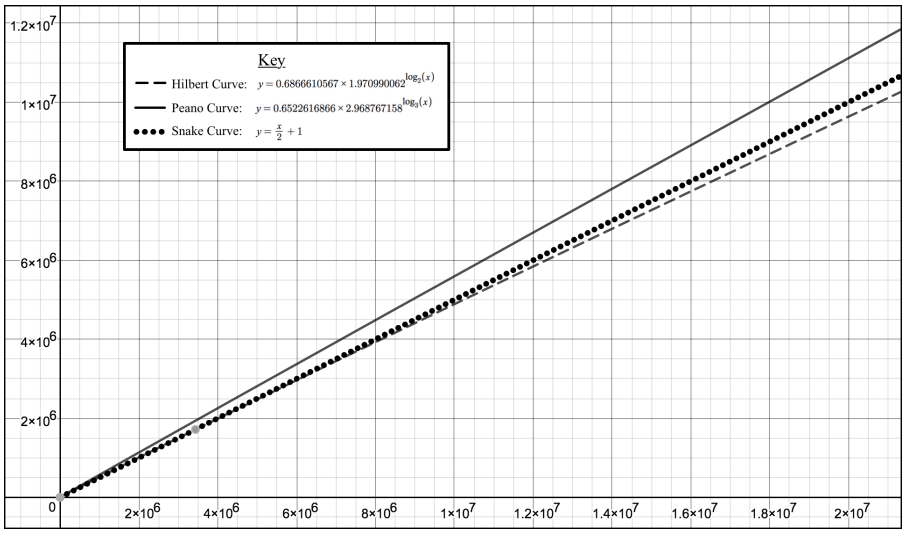
$$MAD_{Hilbert} = 0.6866610567 \times 1.970990062^{log_2(length_{Hilbert})} \qquad \text{(Equation 12)}$$

$$MAD_{Snake} = 0.5length_{Peano} + 0.5 \qquad \text{(Equation 13)}$$

$$MAD_{Peano} = 0.6522616866 \times 2.968767158^{log_3(length_{Peano})} \quad \text{(Equation 14)}$$

In this form, any value can be substituted for x. This implies that side length values can be entered into these equations that are not side lengths of any iteration of the given curve, and would have produce and undefined curve, such as 10 for the Hilbert curve, since an order 2 has side length 4 and an order 3 has side length 16. At values like this, the MAD value is interpolated from the existing iterations.

Below we see the graph of these two equations:



Graph 1: This graph shows the relative MAD values for the Hilbert, Snake and Peano curves. On the x-axis is the length of the side of the matrix and on the y-axis is the MAD value of the curve that fits that side length. The lines of best fit are what are graphed and the exact MAD values for each iteration of every curve was omitted due to the difficulty that results when attempting to distinguish sets of plot points.

From this graph we can conclude that the curve that minimized MAD the most effectively was the Hilbert Curve, the close second being the Snake curve, followed by the Peano curve. There are few reasons to which this can be attributed. Between the exponential equations for Peano and Hilbert curves (in the form $y = a \times b^x$), we see a drastic difference in the b values. The b value for a Peano curve is significantly larger than that of the Peano curve. This means that in the long term, the equation for Peano curves is bound to become significantly larger than the equation for Hilbert curves. Even though there is a difference in a values as well, the difference is small and thus has a negligible effect on the y-values of the equations, especially in the long term. Despite that the Peano curve is

compensated extra for its largest size-to-order ratio, with a base-three logarithm of its exponent it has a larger MAD value in the long term, as shown in the graph, due to the high b value. The Snake curve is of similar effectiveness as the other two curves, regardless of its simplicity, likely because of the way all of its values are strategically arranged such that

6 Conclusion

From the results above we conclude that given four numbers, the way to arrange them such that the MAD value of the matrix they produce is minimal is to order them into increasing order and arrange them into a "c" or "z" shape. The "x" shape has been determined to have a higher MAD value than the aforementioned two. All other arrangements were proven to have identical MAD values and the three stated previously. As for the curve that minimizes MAD, the clear winner would be the Hilbert curve, with the close second being the Snake curve followed finally by the Peano curve.

7 Future Research

Two objectives for future research were mentioned in the body of the paper itself. The first of which was the goal of finding the exact formula for the MAD value of a the two curves that were worked with, Peano and Hilbert. It is also anticipated that the proof that orientation does not affect MAD value will be extended beyond 2×2 matrices to matrices of all sizes. Also, the goal of creating all possible curves that do not intersect themselves and sorting them by MAD value has been set as well.

8 Appendices

```
import java.util.Arrays;
import java.util.ArrayList;

//Created on November 26th, 2016

/* Direction Key:
 * 0 - Up
 * 1 - Right
 * 2 - Down
 * 3 - Left
 */

public class CurveGenerators {
    public static int[][] generateSnake(int order) {
                order++;
                int[][] curve = new int[order][order];
                int x = 0, y = order - 1;
                int val = 1;

                while (x < order) {
                        while (y >= 0 && y < order){
                                curve[y][x] = val;
                                val++;
                                y -= (int)Math.pow(-1, x);
                        }
                        x++;
                        y -= (int)Math.pow(-1, x);
                }

                return curve;
        }

        public static int[][] generateHilbert(int order){
                int size = (int)Math.pow(2, order);
                int[][] curve = new int[size][size];

                int direction = 0; // |  -1 = left turn  |  0 = straight  |  1 = right turn  |

                int[] seq = {1,1};
                int[] oddSeq1 = {1, 0};
                int[] oddSeq2 = {-1, -1};
                int[] oddSeq3 = {0, 1};
                int[] oddSeq4 = {0, 0};

                // Generates the sequence of lefts and rights
                for(int i = 0; i < order - 1; i++){
                        int[] iSeq = inverse(Arrays.copyOf(seq, seq.length));
                        if (i % 2 == 0){
                                seq = addAll(iSeq, addAll(oddSeq1, addAll(seq,
                                addAll(oddSeq2, addAll(seq, addAll(oddSeq3, iSeq))))));
                        }else{
                                seq = addAll(iSeq, addAll(oddSeq3, addAll(seq,
                                addAll(oddSeq4, addAll(seq, addAll(oddSeq1, iSeq))))));
                        }

                        direction = Math.abs(direction - 1);
                }
```

```
        int row = size - 1, col = 0;

        for(int i = 1; i <= size * size; i++){
                curve[row][col] = i;
                //System.out.println("i: " + i + "\t Direction: " + direction
                + "\t row: " + row + "\t col: " + col);
                //System.out.println(seq.length);

                if (direction == 0)row--;
                if (direction == 1)col++;
                if (direction == 2)row++;
                if (direction == 3)col--;

                if (i != size * size && i != size * size - 1) {
                        direction += seq[i - 1];
                        if (direction > 3)direction = 0;
                        if (direction < 0)direction = 3;
                }
        }

        return curve;
}

public static int[][] generatePeano(int order){
        int size = (int)Math.pow(3, order);
        int[][] curve = new int[size][size];

        int direction = 0; // |  -1 = left turn  |  0 = straight  |  1 = right turn  |
        //{0, 1, 1, 0, -1, -1, 0};
        ArrayList <Integer> seq = new ArrayList <Integer>();
        seq.add(0);
        seq.add(1);
        seq.add(1);
        seq.add(0);
        seq.add(-1);
        seq.add(-1);
        seq.add(0);

        for (int i = 1; i < order; i++) {
                for (int pos = 1; pos < seq.size() - 1; pos++) {
                        if (seq.get(pos) == 0 && !(seq.get(pos - 1) == 0 &&
                        seq.get(pos + 1) == 0)) { // Not an S on both sides
                                if (seq.get(pos + 1) == -1 || seq.get(pos - 1) == 1) {
                                        // If next turn is an L
                                        // SLLSRRS  SS  SRRSLLS  SS  SLLSRRS
                                        // Reversed so pos value could stay the same.
                                        // Last S left off to avoid redundancy of deleting
                                        last S and then adding it back in

                                        seq.add(pos + 1, 0);
                                        seq.add(pos + 1, 1);
                                        seq.add(pos + 1, 1);
                                        seq.add(pos + 1, 0);
                                        seq.add(pos + 1, -1);
                                        seq.add(pos + 1, -1);
                                        seq.add(pos + 1, 0);

                                        seq.add(pos + 1, 0);
                                        seq.add(pos + 1, 0);

                                        seq.add(pos + 1, 0);
                                        seq.add(pos + 1, -1);
```

```
                seq.add(pos + 1, -1);
                seq.add(pos + 1, 0);
                seq.add(pos + 1, 1);
                seq.add(pos + 1, 1);
                seq.add(pos + 1, 0);

                seq.add(pos + 1, 0);
                seq.add(pos + 1, 0);

                seq.add(pos + 1, 0);
                seq.add(pos + 1, 1);
                seq.add(pos + 1, 1);
                seq.add(pos + 1, 0);
                seq.add(pos + 1, -1);
                seq.add(pos + 1, -1);

            } else {
                // If next turn is an R
                // SRRSLLS  SS  SLLSRRS  SS  SRRSLLS
                // Reversed so pos value could stay the same.
                // Last S left off to avoid redundancy of deleting
                last S and then adding it back in

                seq.add(pos + 1, 0);
                seq.add(pos + 1, -1);
                seq.add(pos + 1, -1);
                seq.add(pos + 1, 0);
                seq.add(pos + 1, 1);
                seq.add(pos + 1, 1);
                seq.add(pos + 1, 0);

                seq.add(pos + 1, 0);
                seq.add(pos + 1, 0);

                seq.add(pos + 1, 0);
                seq.add(pos + 1, 1);
                seq.add(pos + 1, 1);
                seq.add(pos + 1, 0);
                seq.add(pos + 1, -1);
                seq.add(pos + 1, -1);
                seq.add(pos + 1, 0);

                seq.add(pos + 1, 0);
                seq.add(pos + 1, 0);

                seq.add(pos + 1, 0);
                seq.add(pos + 1, -1);
                seq.add(pos + 1, -1);
                seq.add(pos + 1, 0);
                seq.add(pos + 1, 1);
                seq.add(pos + 1, 1);

            }

            pos += 24;
        }
}

// Take care of first S
// reversed

seq.add(1, 0);
```

```
        seq.add(1, -1);
        seq.add(1, -1);
        seq.add(1, 0);
        seq.add(1, 1);
        seq.add(1, 1);
        seq.add(1, 0);

        seq.add(1, 0);
        seq.add(1, 0);

        seq.add(1, 0);
        seq.add(1, 1);
        seq.add(1, 1);
        seq.add(1, 0);
        seq.add(1, -1);
        seq.add(1, -1);
        seq.add(1, 0);

        seq.add(1, 0);
        seq.add(1, 0);

        seq.add(1, 0);
        seq.add(1, -1);
        seq.add(1, -1);
        seq.add(1, 0);
        seq.add(1, 1);
        seq.add(1, 1);

        // Take care of last S
        // not reversed

        seq.add(1);
        seq.add(1);
        seq.add(0);
        seq.add(-1);
        seq.add(-1);
        seq.add(0);

        seq.add(0);
        seq.add(0);

        seq.add(0);
        seq.add(-1);
        seq.add(-1);
        seq.add(0);
        seq.add(1);
        seq.add(1);
        seq.add(0);

        seq.add(0);
        seq.add(0);

        seq.add(0);
        seq.add(1);
        seq.add(1);
        seq.add(0);
        seq.add(-1);
        seq.add(-1);
        seq.add(0);

} // seq completed
```

```
                int row = size - 1, col = 0;

                for(int i = 1; i <= size * size; i++){ //60 <-- size * size
                        //if (i < seq.size())
                                //System.out.println("i = " + i + "    row = " + row + "
                                col = " + col + "    seq.get(i) = " + seq.get(i));
                        curve[row][col] = i;

                        if (direction == 0)row--;
                        if (direction == 1)col++;
                        if (direction == 2)row++;
                        if (direction == 3)col--;

                        if (i != size * size && i != size * size - 1) direction += seq.get(i - 1);

                        if (direction > 3)direction = 0;
                        if (direction < 0)direction = 3;
                }

                return curve;
        }

        public static int[] inverse(int[] sequenceFragment){
                for (int i = 0; i < sequenceFragment.length; i++) {
                        sequenceFragment[i] = sequenceFragment[i] * -1;
                }
                return sequenceFragment;
        }

    public static int[] addAll(final int[] array1, final int[] array2) {
        if (array1 == null) {
            return clone(array2);
        } else if (array2 == null) {
            return clone(array1);
        }
        final int[] joinedArray = new int[array1.length + array2.length];
        System.arraycopy(array1, 0, joinedArray, 0, array1.length);
        System.arraycopy(array2, 0, joinedArray, array1.length, array2.length);
        return joinedArray;
    }

    public static int[] clone(final int[] array) {
        if (array == null) {
            return null;
        }
        return array.clone();
    }

    public static int indexOf(int needle, int[] haystack, int startPos)
    {
        for (int i = startPos; i < haystack.length; i++)
        {
            if (haystack[i] == (needle)) return i;
        }

        return -1;
    }
}
```

Appendix 1: CurveGenerators class. This code was written and tested in Eclipse Neon version 4.6.1 (IDE for Java Developers). The following is a summary of the functions of each method in the class.

- public static int[][] generateSnake(int order): Generates a 2D array containing values that were converted from a snake curve of order order starting in the bottom left corner and going up. The starting position and direction were chosen arbitrarily because orientation does not not affect MAD value.
- public static int[][] generateHilbert(int order): Generates a Hilbert curve starting in the bottom left corner and going up. The starting position and direction were chosen arbitrarily because orientation does not not affect MAD value.
- public static int[][] generatePeano(int order): Generates a Peano curve starting in the bottom left corner and going up. The starting position and direction were chosen arbitrarily because orientation does not not affect MAD value.
- public static int[] inverse(int[] sequenceFragment): Converts lefts to rights, rights to lefts and straights to straights. Useful for the generation of many space filling curves.
- public static int[] addAll(final int[] array1, final int[] array2): Concatenates two arrays. Useful for the generation of many space filling curves.
- public static int[] clone(final int[] array): Creates a duplicate of a given array.
- public static int indexOf(int needle, int[] haystack, int startPos): This is similar to the indexOf method, but instead of searching for a char among a string, it searches for an integer value among an array.

```
import java.math.BigDecimal;
import java.math.RoundingMode;

public class MeanAdjacentDifference {

	public static void main(String[] args) {

	System.out.println("Order\tMAD"); //Sets up the table

	for (int i = 1; i <= 48; i += 2)
		System.out.println(i + "\t" + getMAD(CurveGenerators.generateSnake(i)));

	}


	public static double getMAD(int[][] array) { // Finds the MAD for a given rectangular array.

		BigDecimal SAD = BigDecimal.valueOf(0); // (Sum of adjacent differences)

		double MAD = 0; // (Mean of adjacent differences)


		/* For each element in the array except the one in the bottom
		 * right corner, it adds the difference between it and the
		 * element to the right and beneath, if either or both are
		 * present, to SAD.
		 */

		for (int i = 0; i < array.length - 1; i++) {

			int j = 0;

			for (j = 0; j < array[0].length - 1; j++) {

				SAD = SAD.add(BigDecimal.valueOf(Math.abs(array[i][j] -
				array[i + 1][j]) + Math.abs(array[i][j] - array[i][j + 1])));

			}

			SAD = SAD.add(BigDecimal.valueOf(Math.abs(array[i][j] - array[i + 1][j])));

		}

		for (int i = 0; i < array.length - 1; i++) {
			SAD = SAD.add(BigDecimal.valueOf(Math.abs(array[array.length -
			1][i] - array[array.length - 1][i + 1])));
		}

		//Divide SAD by the total number of adjacent pairs.
		MAD = (SAD.divide(BigDecimal.valueOf(2 * array.length *
		array[0].length - array.length - array[0].length), 2,
		RoundingMode.HALF_EVEN)).doubleValue();

		return MAD;

	}

}
```

Appendix 2: MeanAdjacentDifference class. This code was written and tested in Eclipse Neon

version 4.6.1 (IDE for Java Developers). This class is capable of calcuating MAD values for any rectangular array of integers. Note that the data type for the SAD variable is BigDecimal. This is because for very large arrays, the SAD value becomes truly massive. For example the SAD value for an order 7 Peano curve is equal to 1.2869497896×10^9. In previous versions of this class, the data type of the SAD was integer, and so overflow became a problem for order 12 and above Hilbert curves and orders 7 and above Peano curves. The method getMAD was generating values that were negative, due to the overflow. For example, for an order 7 peano curve, the MAD was calculated to be -1.6110326720607633, but after the quick switch of data type, it was corrected to 1345.96.

9 Bibliography

1. A.R. Butz (April 1971). "Alternative algorithm for Hilbert's space filling curve.". IEEE Transactions on Computers. 20: 424–42. doi:10.1109/T-C.1971.223258.
2. "Desmos Graphing Calculator". Desmos Graphing Calculator. N.p., 2015. Web. 5 Apr. 2015.
3. Dickau, R. M. "Two-Dimensional L-Systems." http://mathforum.org/advanced/robertd/lsys2d.html.
4. Eavis, T.; Cueva, D. (2007). "A Hilbert space compression architecture for data warehouse environments". Lecture Notes in Computer Science. 4654: 1–12.
5. Hamilton, C. H.; Rau-Chaplin, A. (2007). "Compact Hilbert indices: Space-filling curves for domains with unequal side lengths". Information Processing Letters. 105 (5): 155–163. doi:10.1016/j.ipl.2007.08.034.
6. Hilbert, D. "Über die stetige Abbildung einer Linie auf ein Flachenstück." Math. Ann. 38, 459-460, 1891.
7. Moon, B.; Jagadish, H.V.; Faloutsos, C.; Saltz, J.H. (2001), "Analysis of the clustering properties of the Hilbert space-filling curve", IEEE Transactions on Knowledge and Data Engineering, 13 (1): 124–141, doi:10.1109/69.908985.
8. Peano, G. "Sur une courbe, qui remplit une aire plane." Math. Ann. 36, 157-160, 1890.
9. Voorhies, Douglas: Space-Filling Curves and a Measure of Coherence, p. 26-30, Graphics Gems II.
10. Wagon, S. Mathematica in Action. New York: W. H. Freeman, p. 207, 1991.