Guidelines and Design Examples for Mind-Enabled Applications Part 1 (June 1, 2020)

Scott A. Wilber[1]

Abstract: Mind-Enabled technology provides an objective measurement of the interaction between a user's focused mental intention and specially designed hardware. Physical entropy sources produce sequences of true random numbers that are processed to reveal certain patterns in the output of the Mind-Enabled system. A number of designs are presented utilizing these measurements for practical, real-world applications.

Key Words: Mind-Matter Interaction, Effect Size, Random Walk, Hands-Free Control, Question Answering System, Information Technology, Pattern Recognition, Artificial Intelligence, Rapid Machine Learning.

INTRODUCTION

For over 50 years a number of researchers have proposed and tested the theory that focused mental intention appears to affect the outcome of experiments involving true random number generators. The effect, otherwise called mind-machine interaction, had previously been too small to be used for any practical application. My hardware and theoretical developments have brought the effect from the realm of statistical curiosity to a level useful for many previously unachievable applications. This paper will focus primarily on examples of Mind-Enabled systems designed to enable selected real-world applications.

Essential developments were made in three categories that comprise a Mind-Enabled System.

1) True random number generators are at the heart of Mind-Enabled Systems. Nondeterministic generators of every sort were tested and the most important factor seemed to be the rate of entropy sampling. It is not clear whether certain entropy sources, such as quantum versus thermal noise sources, are superior in these applications. However, even if one source is twice as responsive to mental influence, the less responsive source providing four times the number of bits will be just as responsive. Generators with bitrates of up to hundreds of gigahertz were eventually developed and extensively tested.
2) A number of signal processing methods were developed for both detecting and amplifying subtle patterns in otherwise random sequences. The two most fundamental properties of random sequences are bias, or an excess of either 1s or 0s, and autocorrelation, a correlation between bits separated by one or more shifts (orders) in the sequence.

[1] President of Core Invention, Inc. sawilber@coreinvention.com

3) Systems are designed to optimize responsivity, or the ability to provide the most accurate and fastest desired results. User feedback is found to be very important, and *real-time* feedback is the best. Real-time here means the user receives feedback typically in a fraction of a second from the time an output is generated. For many applications, feedback latency or delay of about 0.25 seconds or less "feels" real-time or immediate. Between 0.5 and 1.0 seconds is consciously noticeable by the user.

There are two types of Mind-Enabled systems. One is user-initiated, in which the user consciously initiates a measurement or trial by pressing a key or button. The other is continuous, meaning a series of measurements is initiated automatically and continuously without user initiation.

SYSTEN EXAMPLES

Random Walks.

A random walk will most often be either one or two dimensional (1D or 2D), though random walks may exist in any number of dimensions. A common type of random walk functions as follows:

1) The walk usually begins at 0 or at the origin (0, 0,…*n*) in an *n*-dimensional space.
2) Random numbers, binary bits in this example, are taken in *n*-tuples, one bit for each dimension.
3) A 1 in each dimension causes the walker to increment one in that dimension, and a 0 causes the walker to decrement or step backward 1 in that dimension.
4) The random walk may be either bounded, meaning the walk stops when a specific bound is reached, or unbounded, where the number of steps is preset to a constant value or just keeps going as long as numbers are fed to the algorithm.
5) In a bounded random walk and for random bits as the input, the *average* number of steps required to reach a bound, either the positive or the negative bound, is the square of the bound. For example, a walker with a bound of ±100 takes 10,000 steps on average to reach one of those bounds. An unbounded walk with a fixed number of steps will move an average distance of:
   $d_T \cong \sqrt{2N/\pi} = 0.797885\sqrt{N}$ in each dimension. 1.
   Where *D* is the number of dimensions, the walker will move $d_T \sqrt{D}$ from the origin.
6) The walk may be displayed step by step as random bits are provided, or the final position can be calculated without following each step. The final position in each dimension is calculated by counting the number of 1s, *ones*, in the sequence of length *N*, and calculating the terminal count or coordinate, C_T:
   $C_T = (2 \times ones) - N$ 2.
   This calculation is to be repeated in each dimension to obtain the terminal coordinate, (*x*, *y*,…*n*).

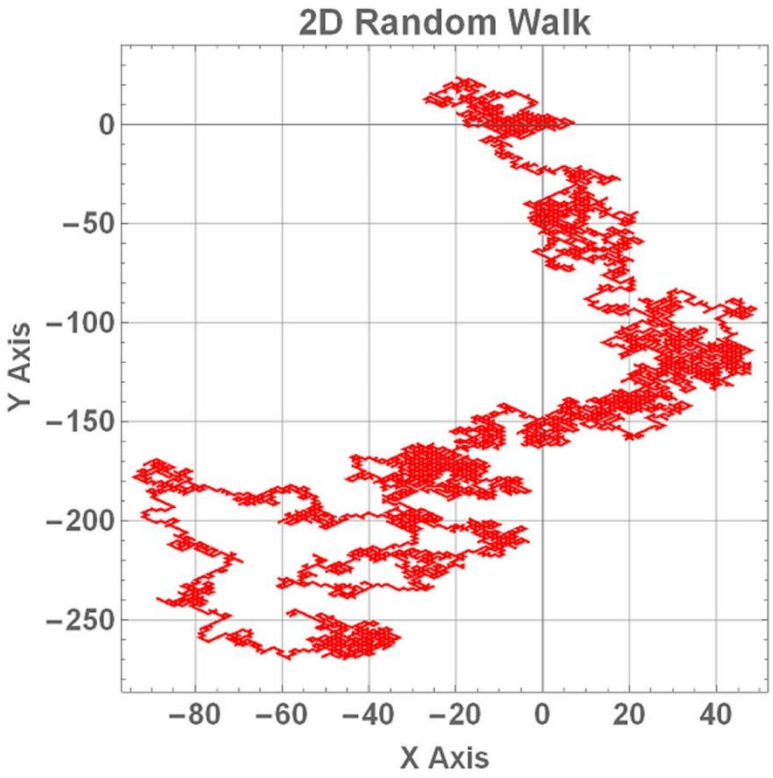


Fig. 1. Example of a two-dimensional bounded random walk. The walk starts at the origin, (0, 0). After 10,000 steps, the walker ends at the coordinates, (-58, -248).

In this example, the number of 1s in the *x* dimension is 4971 and the number of 1s in the *y* dimension is 4876. Using Equation 2, the terminal coordinate is (-58, -248), the same as if the walker is followed step by step.

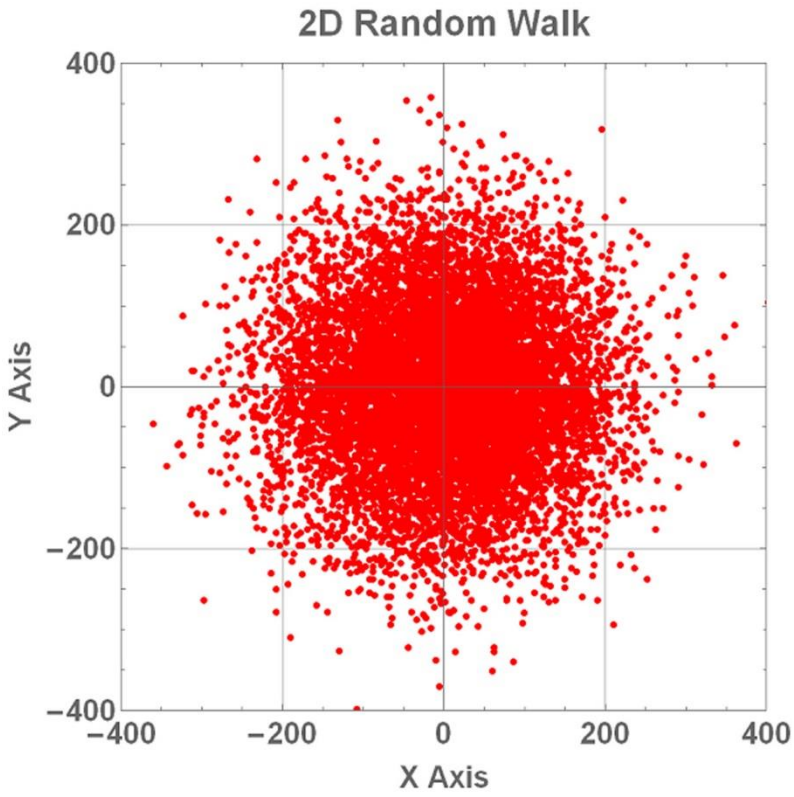


Fig. 2. Example of 2D bounded random walk terminal coordinates for 10,000 separate walks – each of 10,000 steps.

The terminal coordinates are normally distributed in each dimension, with a mean of 0.0 and a standard deviation of $SD = \sqrt{N}$. Since the mean is 0.0, the terminal coordinates can be converted to z-scores by dividing by the standard deviation,

$$z - scores = (x, y)/\sqrt{N} \qquad 3.$$

The z-scores of each coordinate can be converted to uniform variates by a simple inverse approximation.[2]

[2] A Mathematica program for converting z-Scores to uniformly distributed variables is in Appendix 1.

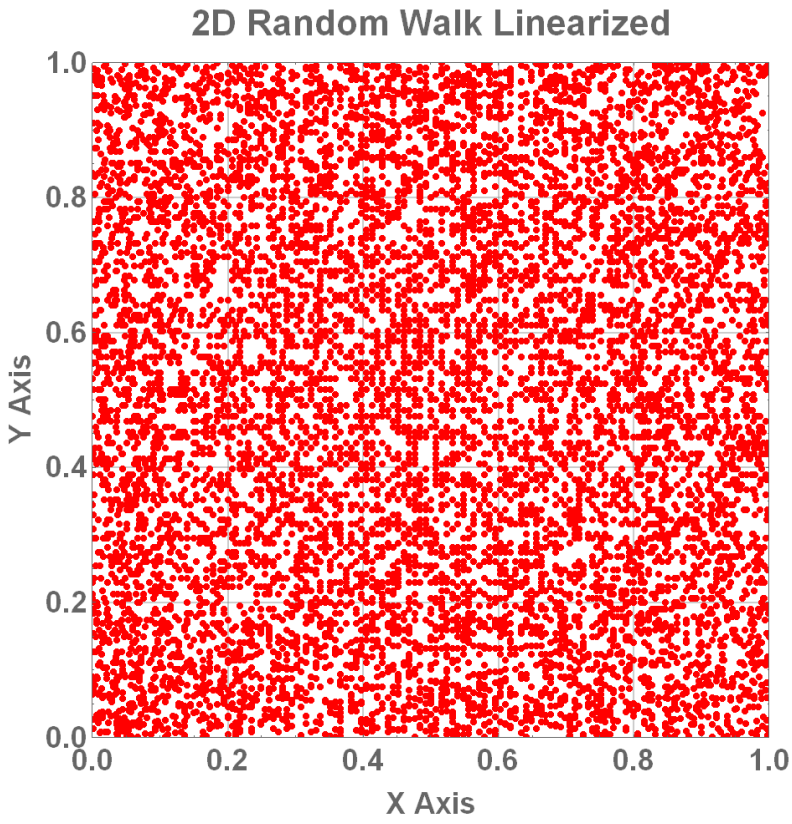


Fig. 3. Example of 10,000 2D 10,000-step random walks with each set of coordinates converted to uniform variates. Note, the coordinates of each terminal point can be easily scaled so they cover a symmetrical range of –1 to +1 in each dimension. That scaling is $(x', y') = (2x - 1, 2y - 1)$ 4. Further scaling may be accomplished by multiplying the coordinates by a constant equal to the maximum range needed by an application. This type of transformed random walk is useful when the terminal location is meant to be equally probable in any part of a square area.

In addition to the linearized 2D walk, a 1D walk is prepared the same way, either in the *x* or *y* dimension. A 1D walk, ranging from 0 to 1 for example, is useful as a simple slider to provide linear control of one or more values.

The walks of Figures 1-3 are examples of bounded walks that end after 10,000 steps (in these examples).

Hands-Free Operation.

Applications meant for hands-free control are required to run continuously without user initiation of a trial or measurement. Random walks can be adapted to run continuously, but must be constrained to remain within prescribed operating ranges. There are a number of ways to accomplish bounding. Three fundamental ways are:

1) A "mirrored" random walk causes the walker to bounce off the bound. When the walker reaches a specified bound, it is moved back to the immediate previous location and allowed to continue walking that way indefinitely. When the walker is moving in more than 1 dimension, the coordinates are processed separately in each dimension. Bounds need not be symmetrical. For example, one bound can be at the origin, preventing the walker from going negative, while the other can be at any selected positive integer value.
2) A random walk can be reset to the origin when it reaches a specified bound, and continue from there on the next step. Reaching a positive bound produces a 1 at the output,

reaching a negative bound outputs –1. Repeated reaching of a bound functions as a bias amplifier whose output can be used as a control function. Since the walker is always moving, the binary outputs of this type must be filtered or weighted to give the most recent output the highest weight. Previous outputs are given lesser weights the older they are. When the weights decrease exponentially with age, the response of the filtered output is a first-order low-pass filter with critical damping (a Butterworth low-pass filter). Weights can be a function of actual time from the latest output, or a function of the number of outputs prior to the latest one. A simple algorithm for making an IIR (infinite impulse response) filter is provided in Appendix 1.

3) A walker can behave as if it is attached by a spring to the origin, making it harder to move farther, the farther it gets from the origin. One way to do this is to move the walker closer to the origin after each step by an amount that is a function of its distance from the origin. The walker will stop moving farther when the drift rate – the average rate of motion in one direction – is equal to the opposite motion due to the "spring" function. Note, the drift rate is equal to the effect size of each automatically generated trial or step.

Fig. 4. 10,000 steps of a 1-dimensional mirrored walk with bounds at $y>50$ or $y<0$. These bounds keep the walker between 0 and 50. At a step rate of 10/second, this walk would take 1000 seconds or just under 17 minutes.

The walk can produce symmetrical output by setting the lower bound to $y<-50$.

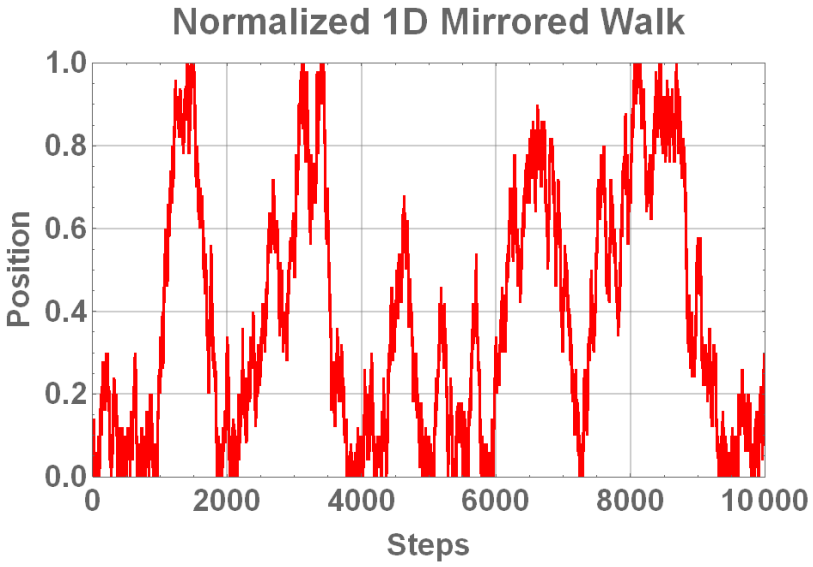


Fig. 5. In a normalized mirrored walk, the position ranges between 0 and 1 by dividing by the maximum value of 50.

The normalized position value can be used as a mind-operated slider. A 3D version requires 3 times the number of random bits, sampled in triplets, to provide 3 simultaneous variable controls.

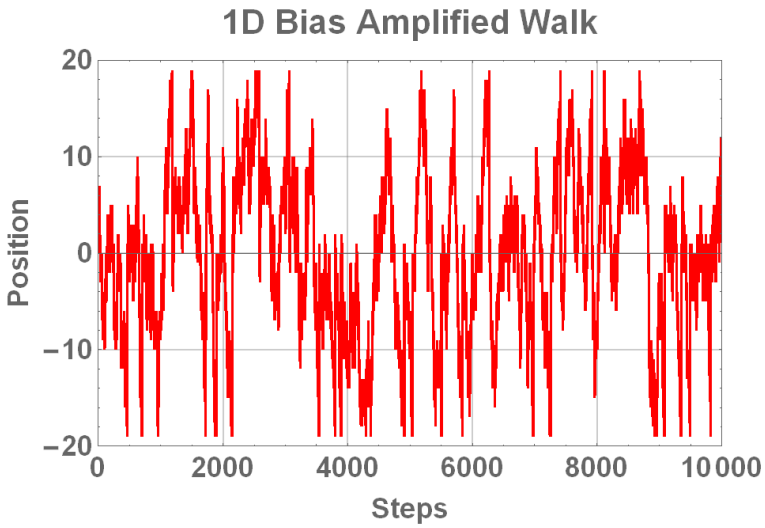


Fig. 6. 10,000 steps in a bias amplified random walk using the same data as figures 4 and 5. The bounds were set at ±20, though the plot only shows the walker at ±19 since it is reset to 0 when it reaches a bound.

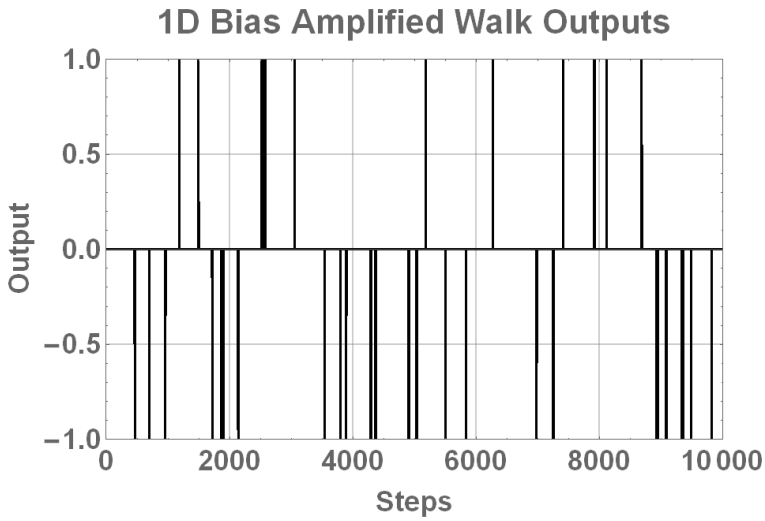


Fig. 7. Outputs of the Bias Amplified Walk of figure 6: +1 when the positive bound is reached and –1 when the negative bound is reached.

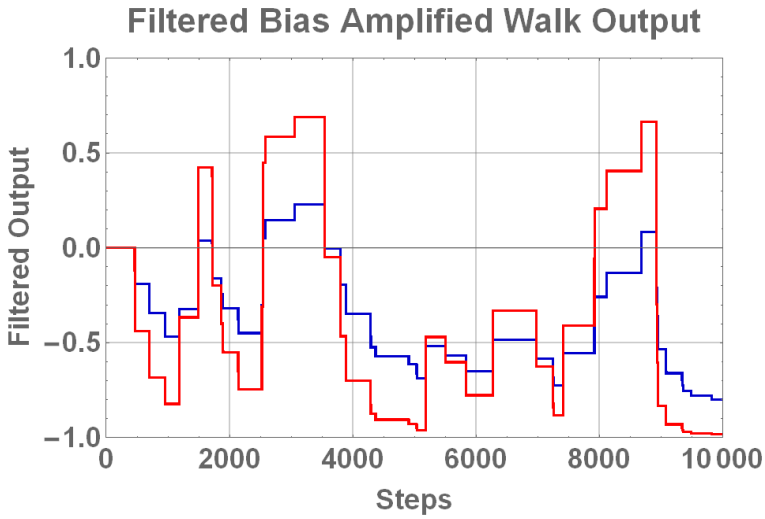


Fig. 8. Filtered output data of figure 7 using the low-pass algorithm in Appendix 1 (updated only when a threshold was reached). The red curve used $w = 0.25$ and the blue curve used $w = 0.10$. Using $w = 0.10$ and setting a threshold at –0.8 would detect the negative movement of the walker. The filtered output can reach more extreme levels when mental influence is present.

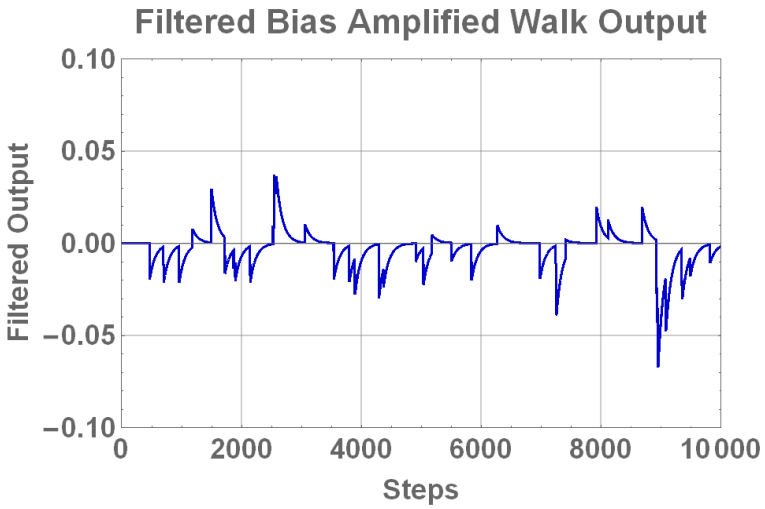


Fig. 9. Filtered output data of figure 7 using the low-pass algorithm in Appendix 1 (updated using every output). The weighting factor, *w*, is set lower, at $w = 0.01$, since the output is updated every trial. The plot suggests for some applications this processing method may be more sensitive for detecting periods of mental influence.

A threshold event occurs, for example in figures 8 and 9, when the filtered data reaches a preset threshold value. Such an event can be used to trigger or control physical devices.

These examples represent hands-free (continuous) operation. Therefore, the system design must include what happens after such an event occurs. An obvious possibility is to reset the *Old* value of the low-pass filter to 0.0 and continue operating as before. With symmetrical thresholds of about ±0.10, reaching the positive threshold can turn on a switch and reaching the negative threshold can turn it off. Clearly, when the switch is already "on," reaching the positive threshold again will not alter the switch state, but will reset the filter output to 0.

An alternative is to make the switch toggle. In that design, reaching the positive threshold only (or the negative threshold only) can be used to toggle the state of the switch. From a user perspective, symmetrical thresholds may be easier to *connect with*, since that is the real-world experience for most switches.

Processing for applications using Model MED100K Mind-Enabled Drive.

The Model MED100K includes a thermal and shot noise entropy source providing 128Mbps without deterministic postprocessing. This is followed within the device by a bias amplifier having an amplification factor of 36. The average output rate is $128{,}000{,}000/36^2 = 98{,}765$ bps.

For an application that takes streaming bits for player-initiated trials, I suggest an output bitrate of 5 to 10 bps. 5bps allows for the maximum overall amplification factor, but may also have over 200ms latency after the player has pressed a button (mouse, keyboard or touchscreen), and the variations in latency may be noticeable. 10bps will have less latency with less variations, but a 40% lower overall amplification factor than 5bps. Both these rates must be tested in a real system to determine which one is better.

Data coming into the server is buffered both in USB interface and the API. To keep the data as fresh and as close to real time as possible, it should be read through the API as fast as it is generated. Additional bias amplification and majority voting will be added at the server side to provide the best data for the player side application.

For 10bps, and starting with 98,765bps from the ME Drive, use a bias amplifier with 36 counts to each bound (taking an average of $36^2 = 1296$ input bits to reach). That brings the bitrate to an average of 76.21bps. To average out the large variations of timing between bias-amplified bits, follow this with a majority voting algorithm using 7 bits (a gain of 2.19). Note, the MV must always use odd numbers of bits to prevent ties that would prevent a valid output. On average, the bitrate after MV would be 10.89bps and the total bias amplification from the entropy source to the output is 36 x 36 x 2.19 = 2,838. For 5bps, use a bias amplifier with 53 counts to each bound. That brings the bitrate to an average of 35.16bps. Follow this with a MV algorithm using 7 bits. On average, the bitrate after MV would be 5.02bps and the total bias amplification from the entropy source to the output is 36 x 53 x 2.19 = 4,179.

The previous calculations are for a single or 1 dimensional output. For 2 simultaneous or 2D outputs, parse the ME drive output into two channels. The simplest way to do this is apply every other 8-bit byte to the two external bias amplifiers. Note, each of the processed outputs are unlikely to finish at the same time. After one channel has produced an output bit, apply all the data to the remaining channel. Taking a starting bitrate of 49,383bps for each channel and a target rate of 10bps: the bias amplifier will have boundaries at ±26 and MV of 7 bits, the output bitrate for each channel will be 10.44bps with a total amplification of 36 x 26 x 2.19 = 2050. For 5bps target rate, the bias amplifier will have boundaries at ±36 and MV of 7 bits, giving an output bitrate of 5.44bps and a total amplification of 36 x 36 x 2.19 = 2838.

A number of additional steps can be taken to ensure a valid output is produced in the allotted time period.

1) MV results of each odd number of bias amplified bits (1, 3, 5 or 7 bits) is stored and updated, and the last update when the allotted time (100 or 200ms) is reached is taken as

the output. The output taken will usually be at the 7th MV bit, but could vary between the 1st and 7th bits due to statistical variations in the bias amplifier output rate. The total amplification for some of the outputs will vary somewhat, but will not noticeably affect the results.

2) As a final backup to ensure an output, if the bias amplifier has not produced a single output in the allotted time, take the sign of bias amplifier counter as the output. (Output 1 if the counter is positive and -1 if the counter is negative.)
3) If the state of the bias amplifier is 0, then take the last input as the output. The last input is used because the only information present is the direction of the last step (from +1→0, negative; or -1→0, positive). Alternatively, take the next bit as the output. That simplifies this final backup method by not having to keep track of previous bits.

To minimize latency between generation and use by the player, and when data is streamed over the Internet, each bit must be sent in a separate packet as soon as it is available. In all cases, a timestamp with millisecond resolution from the system clock should be included in each data packet to indicate when it was prepared. In addition, it is often useful to include the previously sent bit in each data packet. These two details are useful for error checking and latency measurements.

To keep the walk on the screen or playing area, bound the walk by not incrementing it if the next step will move outside the desired area. Bound each dimension separately if more than one dimension is being used.

Comparing Bias Amplifier with Simple Majority Voting.

Majority voting (MV) is a simple, though less efficient, method of increasing the amplitude of a bias at the expense of the number of bits output. MV counts the number of 1s in a sequence of known length. The length is always chosen to be an odd number in order to avoid ties in the count, which would not produce a valid output. When the count of 1s is > $N/2$, output 1, when the count of 1s is < $N/2$, output 0 (or -1), where N is the number of bits used in the MV. An outline for computing MV results are in Appendix 1.

The overall gain/amplification factor using MV is compared to the combination of bias amplification followed by a short MV, described on page 9, paragraph 4:

1) For 10bps, the above example gives an overall gain of 2,838 at an average bitrate of 10.85 bps. Starting with 98,765bps from the ME Drive, a MV using 9875 bits (gain = 78.77) provides an overall gain of 36 x 78.88 = 2,836 at an average bitrate of 10.00bps.
2) For 5bps, the example gives an overall gain of 4,179 at an average bitrate of 5.02bps. A MV using 19,753 bits (gain = 110.68) provides an overall gain of 36 x 110.68 = 3,984 at an average bitrate of 5.00bps.

When the output effect size (ES) is less than about 20%, the two methods produce almost the same overall gain. This is a consequence of having to use MV after the bias amplifier to average

out variations in the bias amplifier output interval (the time interval between output bits). Above 20% ES, the MV only method produces increasingly inferior results. The bias amplifier alone would produce an overall gain of 36 x 160 = 5,760 for the 5bps output bitrate. This is indeed substantially higher than the 3,984 gain for MV only; however, large variations in the output interval make it impractical for most applications.

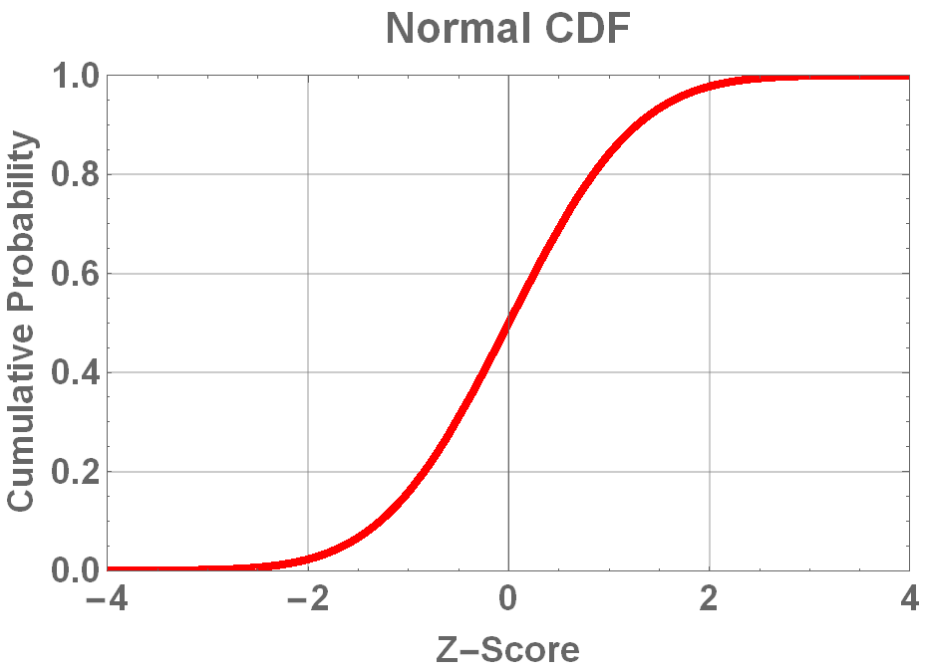
This comparison is useful in that it shows the easier to compute MV result is not significantly inferior to the bias amplifier/MV combination. The requirement for this to be true is that the output ES is usually less than 20%. This is a reasonable assumption for most applications using the MED100K device. Even if the output ES goes as high as 40% – an unlikely prospect with this hardware – the MV alone gain is less than 10% below the combined bias amplifier/MV combination.

APPENDIX 1 – UTILITY PROGRAMS

Z-Score to probability conversion program.

The following Mathematica program employs a curve fit to approximate the cumulative probability of the normal distribution. It can be used to transform normally distributed numbers into uniformly distributed numbers. The accuracy is ±0.05% for z-scores up to ±4, and ±1% for z-scores up to ±7.5.

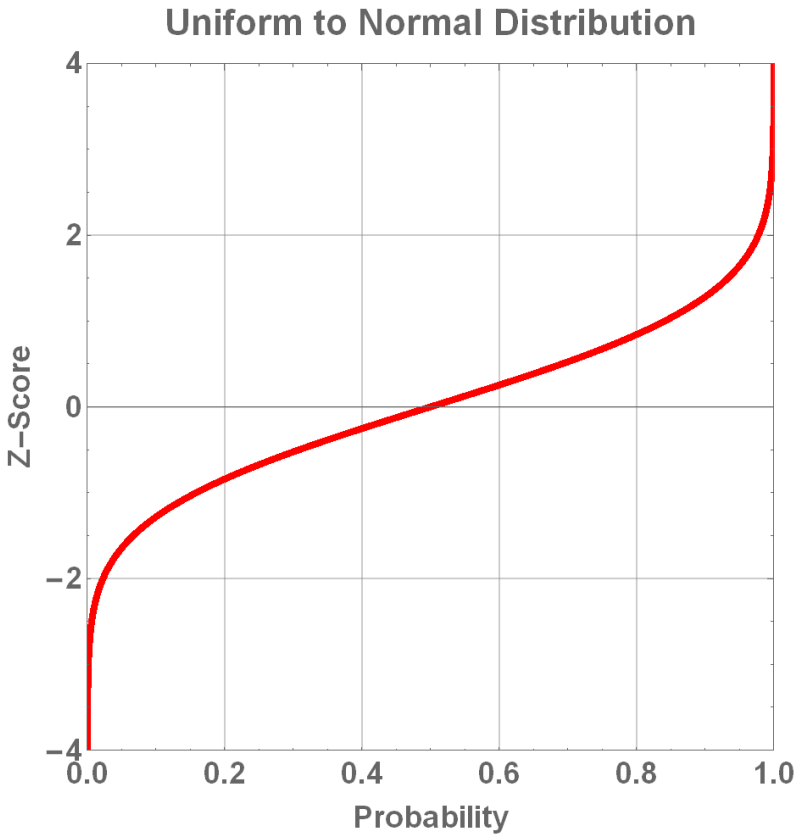
```
cdf[z_]:=
    c1 = 2.506628275; c2= 0.31938153; c3 = -0.356563782; c4 = 1.781477937;
c5 = - 1.821255978; c6 = 1.330274429; c7 = 0.2316419; (*Constants*)
    If [z ≥ 0, w = 1, w = -1];
    t = 1. + c7 w z;
    y = 1./t;
0.5 + w (0.5 - (c2 + (c6 + c5 t + c4 t^2 + c3 t^3)/t^4)/(c1 Exp[0.5 z^2] t))
```



Probability to z-score conversion program.

The following Mathematica program employs a curve fit to approximate the z-score of a normally distributed number from a probability. It can be used to transform uniformly distributed numbers into normally distributed numbers. The accuracy is 6 digits for z-scores up to ±6 and 0.1% for z-scores up to ±7.5.

```
invp[p_]:=
p0 = -.322232431088; p1 = -1.0; p2 = -.342242088547; p3 = -.0204231210245;
    p4 = -.453642210148 10^-4;
q0 = .099348462606; q1 = .588581570495; q2 = .531103462366; q3 = .10353775285;
    q4 = .38560700634 10^-2; (*Constants*)
If[p < .5, pp = p, pp = 1. - p];
y = Sqrt[Log[1/(pp^2)]];
xp = y + ((((y p4 + p3) y + p2) y + p1) y + p0)/((((y q4 + q3) y + q2) y + q1) y + q0);
If[p < .5, xp = -xp];
xp
```



IIR Digital LP Filter.

A sequence of numbers is low-pass filtered as follows:

1) Initialize the previous value, *Old*, to 0.0 when the filter is started.
2) The output of the filter is: *Out* = (1-*w*) *Old* + *w New*, where *New* is the current data input and *w* is a constant that together with the data input rate determines the cutoff frequency of the filter. $w < 1.0$ and typical values are: $0.1 < w < 0.5$, where lower values produce lower cutoff frequencies, giving more weight to older input data.
3) Set *Old* = *Out*.
4) Repeat steps 2 and 3 as long as new data is available.

Bias Amplifier.

A sequence of numbers is bias amplified using a modified bounded random walk. The bias (and effect size) is multiplied by the Amplification Factor and the average number of input bits to reach either bound is Amplification Factor squared (Amplification Factor of 100 takes an average of 10,000 steps to reach either bound):

1) A bidirectional counter (counts both up and down) with thresholds at ±Amplification Factor (with slight adjustments for floating point versus integer counting) is initialized to 0.
2) The counter is incremented for each input bit that is 1, or decremented for each input bit that is 0 (or -1). Note, a -1 is often used instead of a 0 to simplify some of the bitwise computations and random walks.
3) When <u>floating point arithmetic</u> is used for counting: If counter value > (Amplification Factor -0.5), output a 1 and reset the counter to 0; Or, if counter value < (0.5 - Amplification Factor), output a 0 (or -1) and reset the counter to 0, else take another input bit.
   When <u>integer counting</u> is used: If counter value > (Amplification Factor -1), output a 1 and reset the counter to 0; Or, if counter value < (1 - Amplification Factor), output a 0 (or -1) and reset the counter to 0, else take another input bit.
   Note, less than < or greater than > are used in the conditionals rather than equal signs =. This is better programming style. If somehow an error occurs and an equality is missed/passed over, then the program may never get back on track. The greater or less than approach always corrects itself by the next input.

When input numbers are presented as words, a lookup table can be used to speed up processing:

1) Assume 8-bit words, though this method will work with larger tables reasonably up to 16 bits.
   a) A lookup table is constructed with the (relative) address equal to the input word (for 8-bit words, a table of 256 rows is required).
   b) The data at each address is the number of 1s minus the number of 0s in the input word; $-8 \leq$ data ≤ 8. Example, the input word is 10111010, the data is +2. The order

of 1s and 0s does not matter. The first address and data are: 00000000, -8, and the last entry in the table is: 11111111, 8.

2) A bidirectional counter (counts both up and down) with thresholds at ±Amplification Factor is initialized to 0.
3) When an input word arrives, use the lookup table to find the number and direction of steps taken: the data.
4) The data (assuming signed integers or floating-point numbers) is added to the counter value.
5) If counter value > (Amplification Factor -0.5), output a 1 and reset the counter to 0; Or, if counter value < (0.5 -Amplification Factor), output a 0 (or -1) and reset the counter to 0, else take another input word (Step 3).
   Note: strictly speaking, when the counter value is within 8 counts of either bound, the process should shift to bit-by-bit processing (described above) until the bound is reached or the walk steps away from the bound more than 8 counts. That is because the excursion of the counter depends on the sequence of bits in the word. (Note, words are assembled in hardware so the LSB is the oldest bit generated and the MSB is the most recent.) For example, if the word is 11110000 (LSB on the right), the data value is 0, but the walker would have gone -4 steps before going +4 steps. If, however, the word is 00001111, the data value is still 0, but the walker would have first gone +4 steps, then stepped down 4 steps. If the counter is within 4 steps of a bound, bit-by-bit processing would produce an output, while the word-wise processing would not.

Majority Vote.

Majority voting is a simple algorithm that determines whether the number of 1s in a sequence totals more or less than half the total number of bits, i.e., a simple majority of the bits. The number of bits in the sequence, *N*, is always chosen to be an odd number in order to avoid ties in the count, which would not produce a valid output.

Input numbers from the MED100K are formatted as 8-bit words (bytes). They may be processed bit-wise (bit by bit), or as words using a lookup table:

Bit-wise processing:

1) Initialize the bit counter (BC) to 0.
2) Initialize word count (WC) to 0 (keeps track of the total number of words/bits used in the MV). Example, for a target of 9875 bits, divide by 8 (bits per word) to get 1234.375 words, or 1234 words plus the first 3 LSBs of the 1235^{th} word.
3) Count the number of 1s in the data word. There are a number of ways to do this. Here is an example:
   a) Initialize a word bit counter (WBC) to 0. (This counter is only used to count 1s in each word.)
   b) Right shift the word once (divide by 2) and add the LSB to WBC.
   c) Repeat steps 2a – 2b 8 times to find the total number of 1s in the word. $0 \leq WBC \leq 8$.

d) If WC > 1234, count only the first 3 LSBs to hit the exact total of 9875. Set a flag (Flag) indicating this is the last data in the current sequence.

4) Add the bit count from the most recent word: BC = BC + WBC.

If Flag = 1, then (If the count of 1s, BC, is > $N/2$, return 1, else If the count of 1s is < $N/2$, return 0 (or -1).), else (Flag ≠ 1), continue from step 3.

Word-wise processing:

1) Make a Table (just once) with address 0-255 and data = WBC (number of 1s in the address word)
2) Initialize the bit counter (BC) to 0.
3) Initialize word count (WC) to 0 (keeps track of the total number of words/bits used in the MV). Example, for a target of 9875 bits, divide by 8 (bits per word) to get 1234.375 words, or 1235 (1235 total with 5 bits discarded in the last word).
4) Get a data word, use the table to look up the WBC in data word.
   If WC > 1234, mask the upper 5 bits by ANDing with 00000111, use the resulting word to look up the WBC from the first 3 LSBs and set the Flag indicating this is the last data in the current sequence.
5) Add the bit count from the most recent word: BC = BC + WBC.

If Flag = 1, then (If the count of 1s, BC, is > $N/2$, return 1, else If the count of 1s is < $N/2$, return 0 (or -1).), else (Flag ≠ 1), continue from step 4.