Bias Amplification Algorithms for MMI.

Scott A. Wilber[1]

Bias Amplification is an algorithmic method for compressing a large input sequence of binary bits into a much shorter sequence, while maintaining the information contained in the bias of the input bitstream.

It is easy to be confused by the terms used in this context, so they must be carefully defined.

1) Hit Rate, *HR*, is the number of hits divided by the total number of trials in a series of trials. It is also, $HR = P(Pout > 0.5 \mid Pin, N)$, that is, the probability that $Pout > 0.5$, given *Pin* and input sequence length, *N*.
2) A trial is a measurement, typically initiated by a user or player, from which an outcome is determined.
3) A *hit* describes the result of a trial when an MMI measurement matches the user-intended outcome. For example, a user may intend a binary random number generator to produce a 1 on its next output. If it does, that is a hit. Or the user may intend an output to match, either actually or symbolically, an outcome that has occurred or will occur. When the outcome is revealed or observed, the result is a hit if the MMI measurement matches. Intending an MMI measurement to *not match* a target value is equally valid. When that occurs, it is a hit.

So-called psi-missing, or getting statistically significant results opposite to a user's intention, is a type of noise and is not useful in MMI applications.

4) A *miss* is the result of a trial when an MMI measurement does **not** match the user-intended outcome. It is the opposite of a *hit*.
5) Probability in, *Pin*, is the marginal probability of a 1 occurring in an "input" to a processing algorithm. *Pin* can be estimated as, *Pin* = count of 1s/*N*, where *N* is the number of bits in the sequence in which the count of 1s exists. This is typically a sequence of raw bits from an MMI generator or a sequence prior to a further processing stage.
6) Probability out, *Pout*, is the marginal probability of a 1 occurring in an "output" from a processing algorithm. *Pout* can also be estimated as, *Pout* = count of 1s/*N*, where *N* is the number of bits in the sequence in which the count of 1s exists. This is typically the final output sequence or a sequence after a previous processing stage.
7) Effect Size, $ES = 2\,HR - 1$. Effect size is sometimes also used as, $ESout = 2\,Pout - 1$, though *Pout* and *HR* are not always equivalent. It's clearer to just use $2\,Pout - 1$.
8) Hit Rate in, *HRin*, is undefined and should not be used. The derived term, Effect Size in, *ESin*, should be represented instead as, $2\,Pin - 1$, which is clearly defined.
9) Gain or amplification factor is, $Gain = (2\,Pout - 1)/(2\,Pin - 1)$.

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

There are two configurations of bias amplifiers, each with its own set of equations for predicting results.

1) The first configuration – because I developed it first – uses a variable number of input bits and produces a single output bit. This method can produce the highest *gain* and hit rate, but the potentially large variability of trial generation time can cause issues with user feedback and interaction.
2) The second configuration uses a fixed number of input bits and produces one output. The gain and hit rate are not as high as the first configuration, but the constant time to produce a trial output can be an advantage.

A good bias amplifier will have the following properties.

1) A model or equation exists to accurately predict the performance of the method.
2) The method should be efficient. That is, the measured performance should be very close to the theoretical prediction.
3) The algorithm should be relatively easy to code and use.
4) The computational overhead should be moderate with little added latency to allow near real-time results.

Derivation of the Hit Rate Equation for a Fixed Number of Input Bits.

The mean of a sequence of binary random bits is expected to approach 0.5 for large *N*. The deviation from the mean is just, $Pin - 0.5$, and the z – score is the deviation divided by the standard deviation. The distribution of successes in a binary sequence is Binomial and the standard deviation is, $SD = \sqrt{Pin(1 - Pin)/N}$. When *Pin* is close to 0.5, as it will be in most MMI generators, $SD \cong 0.5/\sqrt{N}$. Therefore,
$z - score \cong (Pin - 0.5)/(0.5/\sqrt{N})$. Simplifying,
$z - score \cong (2Pin - 1)\sqrt{N}$. 1.

The theoretical hit rate, *HR*, is the probability the number of bits of the intended value will be an absolute majority in a sequence of bits of length, *N*. *N* should be an odd number to prevent ties. The probability is the cumulative distribution function of the normal distribution, evaluated at the calculated z-score:

HR = CDF[NormalDistribution [0,1], $(2Pin - 1)\sqrt{N}$] 2.

The z – *score* and hit rate equations use the normal approximation for large *N*. The accuracy of the *HR* approximation is about (25/*N*)%. That is, for *N* as small as 25, the accuracy is 1%; for *N* = 101, 0.25%.

The number of bits required can be normalized to give, $Nbits = N\,(2Pin - 1)^2$. Note, by inspection of equation 2, the z – *score* squared is *Nbits*.

The inverse of the hit rate equation is used to calculate *Nbits* using the inverse cumulative distribution function of the normal distribution evaluated at the hit rate, *HR*.

$Nbits$ = InverseCDF[NormalDistribution[0,1], HR]^2. 3.

To recover the actual number of bits, *N*, divide *Nbits* by $(2Pin - 1)^2$.

Algorithms for approximating both the *cdf* and *invp* used in these two equations are provided in Appendix 1 at the end of this paper.

Example: 1 million sets of 1001 random bits with *Pin* = 0.505 was input into a 5-stage bias amplifier design. The theoretical hit rate was 0.62415 and the averaged hit rates from the million runs was 0.62448. The difference is 0.00033 with a nominal probability, $p = 0.86$ that the difference could have happened by chance in a one million sample data collection.

The Hit Rate Equations for Variable Number of Input Bits.

Most of the relationships for a single output with variable number of input bits was described in an earlier work: Wilber, S., 2013, *Advances in Mind-Matter Interaction Technology: Is 100 Percent Effect Size Possible?* Available at https://coreinvention.com/files/papers/Advances-in-Mind-Machine-Interaction.pdf

For a single output bias amplifier, the hit rate is *Pout*, though the actual hit rate cannot be assessed from a single trial. Equation 4 from *Advances in Mind Matter Interaction…*, gives *N*, the average number of steps taken by a random walk bias amplifier as a function of input and output probability, *Pin* and *Pout* respectively.

$$N = \frac{2Pout - 1}{2Pin - 1} Ln\left[\frac{1 - Pout}{Pout}\right] \Big/ Ln\left[\frac{1 - Pin}{Pin}\right] \quad 4.$$

.

This equation can be simplified by multiplying it by $(2Pin\ -1)^2$. Then by taking the following limit

$$\lim_{Pin \to 0} (2Pin - 1) \Big/ Ln\left[\frac{1 - Pin}{Pin}\right] = -0.5$$

and simplifying, the normalized number of bits, *Nbits*, becomes a function of *Pout* only:

$$Nbits = (0.5 - Pout) Ln\left[\frac{1 - Pout}{Pout}\right] \quad 5.$$

.

The simplified equation works with high accuracy. The error is about 33.3 $(2Pin - 1)^2$ %. That is, the error is 0.00333% when $2Pin - 1 = 0.01$ and about 0.335% when $2Pin - 1 = 0.1$. The error does not depend on *Pout*. To recover the actual number of bits, $N = Nbits/(2Pin - 1)^2$.

There is no analytical solution for *Pout* as a function of *N* and *Pin*. Use a numerical method, such as Newton–Raphson, to approximate that result.

Gain = $(2Pout - 1)/(2Pin - 1)$, *Gain* < square root *N*.

Comparison of Bias Amplification Algorithms.

This figure shows the normalized number of input bits, *Nbits*, required to produce the indicated hit rate. The upper solid curve was generated by the *Nbits* equation for fixed number of input bits, and the red points were produced by simulation. The bottom gray curve was generated by equation 5 for variable number of input bits, and the blue points were produced by simulation. Normalizing the number of input bits gives a compact overview of the two types of amplification algorithms.

The simulated points fall within statically expected error brackets of both curves, even for millions of trials. The simulated data and the theoretical equations are entirely independent, so these results give confidence they are both correct.

These curves provide theoretical boundaries for the “best” design of each type of amplifier. At a glance it is clear using a variable number of input bits requires fewer bits to produce a given hit

rate. However, amplifiers of this type also have a high variability in time to produce an output. The output generation time can be up to **10** times as long as the average, though it is usually no more than a factor of 3. That range may be too large for effective real-time user feedback.

Bias Amplifier Code for Variable and Fixed Numbers of Input Bits

A variety of bias amplifier designs for either fixed or variable input bit configurations are available. The code used for simulation can be directly adapted to real-world bias amplification designs. Extensive testing confirms these implementations produce the best possible results for each configuration. A number of these designs and their pros and cons will be described below.

1) **For variable numbers of input bits**, there are two algorithms: random walk and Bayes' updating. It is possible to implement an algorithm using majority voting, but it is awkward and doesn't provide any advantage over the other two.

   a) Random Walk Bias Amplification is probably the simplest and provides the highest gain possible. In addition, it is easy to implement in hardware designs, such as Field Programmable Gate Arrays (FPGAs), that can operate with extremely high generation rates. This type of amplifier can be operated in parallel generation streams and outputs combined downstream. They may also be operated in series without loss of overall efficiency.

   The following code was implemented in Mathematica, but can easily be translated into other languages:

   Set variables:
   n = 142; This is the count at which an output is produced.

   Input data:
   inputbit is 1 if the next MMI data bit is a 1, or –1 if the next bit is a 0.

   Program:
   countb = 0; This is the running count in the counter, reset to 0 after each output.
   While[Abs[countb] < n, countb = countb + inputbit; If[countb == n, 1, 0]

   Output:
   The output is a 1 if countb reaches n, or 0 if countb reaches negative n.

   This program uses the *While* statement. That means as long as the first statement is true, the operations following the comma will continue. There are two operations separated by a semicolon: countb is added to new input bits, one at a time; and, If countb = n, the output would be a "1" else the output would be a "0". This last result

is the output of the amplifier, and only happens when the conditional, Abs[countb] < n, is no longer true. That is, when countb reaches ± n.

n must be a positive integer. Its value is determined by a simple iterative process.

I) Set the average number of bits, *N*, to be used at the input for each trial. For this example, choose 20,000 bits, or 0.2 seconds of data from the MED100Kx8 MMI generator.

II) Use an initial value of *n* = square root of *N*, or 141. *Pin* depends on the skill of the user, but a value of 0.5005 is assumed for this example. Use these values to calculate the initial *Pout*:

$$Pout = 1/\left(1 + \left(\frac{1 - Pin}{Pin}\right)^{n}\right) \qquad 6.$$

Giving *Pout* = 0.570037.

III) Use equation 5 with this *Pout* to calculate *Nbits* = 0.019751. Convert this to *N* by dividing by $(2\ \mathrm{Pin} - 1)^2$, giving 19,751 bits. This is likely close enough to the target of 20,000 bits, but let's see if there is a better *n* value.

IV) Calculate an adjusted *n*:

$$n(adjusted) = n(initial)\sqrt{Target\ N/Initial\ N} \qquad 7.$$

For this example, that gives the adjusted *n* = 141.89, but *n* must be an integer, so take 142. Using 142 in equation 6 gives *Pout* = 0.570527.

V) Finally use the adjusted *Pout* in equation 5 and divide *Nbits* by $(2Pin - 1)^2$ to give the adjusted *N* = 20,030 bits. This second iteration gives the closest *Pout* and *n* value for the given input variables. It also gives the theoretical *Pout* and finally the effect size at the output, *ES* = 0.141 or 14%. From this the amplifier *gain* = 0.141/0.001 = 141.

The adjustment in the example is quite small, but it can increase significantly if *Pout* is substantially larger. The steps are exactly the same in that case, but it is possible more than 2 iterations will be required.

b) Bayes' Updating can be implemented as a general bias amplification method, but its greatest power is to combine results of trials from a single user or multiple users. The skill levels of various users for particular tasks, represented by historical hit rates, are used as weighting factors in the equations. The target accuracy for the task is also an input variable. These properties make Bayes' Updating the best method to accomplish hard tasks using MMI that cannot be done in a single trial or by a single user.

See the chapter on using Bayes' Updating for a description of the equation and how to use it.

2) **For fixed numbers of input bits** there are two algorithms: majority voting and random walk. They each have different properties so neither is clearly the best.

a) **Majority Voting** is very easy to implement in software designs when only a single output is desired. It is not optimal for high-speed hardware designs and it loses efficiency when operated in series or one after another. As its name implies, count the

number of 1s in a sequence of odd-number length. If the count of 1s > *N*/2, output 1, else output 0. Use equation 2 to calculate the theoretical average hit rate. For *Pin* = 0.5005 and *N* = 19,999 bits, *HR* = 0.55623.

Note for minimizing computational overhead: an odd number just under 20,000 is chosen for *N* because it is one bit less than a full 2500 bytes. The number of 1s in a byte can be precalculated and stored in an 8-bit (256 line) lookup table. During operation the count of 1s is incremented by using the 8-bit MMI data as the address to look up the count of 1s in that byte. The final byte is masked so there are only 7 bits. This is done by performing an AND function of the final byte with 127 binary. That makes the MSB 0. This word-wise approach can be much more efficient than bit-wise operations.

b) **Random Walk** amplification can be used with a fixed number of input bits.

I) Select *N*, the total number of bits to be used for each run of the amplifier. Select the number of outputs to be generated each trial. This will either be 1, or another small odd number, such as 5. The reason to generate more than one output per trial is if intermediate processing is desired. This can be the case for Reveal or Predict MMI modes in order to conceal the target direction from the user to prevent unconscious bias. Example: for 19,995 input bits and 5 outputs, each run of the amplifier will take 3999 bits. Always use an odd number to prevent ties in the output.

II) Run the Random Walk amplifier until all the 19,999 bits have been used for a single output, or 5 times using 3999 bits each (in this example).

III) There are many variations of the algorithm. The following simple program was implemented in Mathematica:

Input data:
inputbit is 1 if the next MMI data bit is a 1, or –1 if the next bit is a 0.

Program:
countb = 0; This is the running count in the counter, reset to 0 after each output.
Do[countb=countb+*inputbit*,{i, *N*}]; Run the loop *N* times.
zout = countb/Sqrt[*N*] This is the output in the form of a z – score.

IV) Process the output(s) to produce the final result. If a single output is desired, threshold the output: If[z – score > 0, output 1, else output 0].

If multiple z – score outputs have been generated, there are many ways of processing them. The 5 outputs (in this example) may be individually multiplied by + 1 or – 1 as chosen by 5 outputs from a true random number generator. This operation will conceal the target direction from the user. Then add together the 5 resulting z – scores. Normalize the combined z – score by dividing by the square

root of 5 (the number of z – scores combined). Note, for normally distributed numbers adding or subtracting gives a normally distributed number. Finally, threshold the resulting z – score above or below 0 to produce a 1 or a 0 output. Please be aware, this inverting of some of the outputs will work for MMI systems, but it will not work in a simulation. This is one of the ways MMI and pure algorithms are different.

Example calculations: a single output using N = 19,999 and Pin = 0.5005, gives a theoretical hit rate (using equation 2), HR = 0.55623. A simulation of 100,000 runs gave an average $Pout$ = 0.5568 (p = 0.64).

A single output using N = 3,999 and Pin = 0.5005, gives a theoretical hit rate (using equation 2), $Pout$ = 0.52521. A simulation of 100,000 runs gave an average $Pout$ = 0.52555. These intermediate results were added together in non-overlapping blocks of 5 outputs. The 20,000 combined results gave, $Pout$ = 0.55701. This two-step $Pout$ result is not significantly different (p = 0.59) from the single result using 19,995 bits in a single random walk, demonstrating good efficiency.

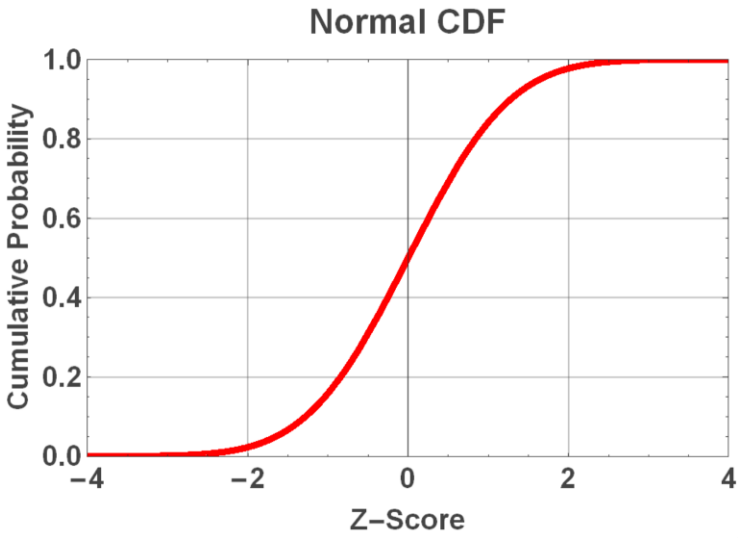
Note for minimizing computational overheard using word-wise processing with the random walk amplifier algorithm. Choose an odd number just under 20,000, N = 19,999, for a single output because it is one bit less than a full 2500 bytes. Or, for 5 outputs (in this example), use N = 3999 for each run. The number of 1s minus the number of 0s in a byte can be precalculated and stored in an 8-bit (256 line) lookup table. During operation *countb* will be added to the value in the table, using the 8-bit MMI word as the address to look up the count of 1s – 0s in that byte. The final byte is masked so the MSB is always 0. This is done by performing an AND function of the final byte with 127 binary. The result of the final lookup must be adjusted by adding 1 prior to adding it to *countb*. The DO loop will be run N/8 times, for bytes instead of bits.

The p values in parentheses are the probabilities the results could have happened by chance given the sample size (number of runs) and the resulting average $Pout$. Probability values of $0.025 \leq p \leq 0.975$ are not statistically significant.

A theoretical calculation using the majority voting equation gives $Pout$ = 0.525213 for Pin = 0.5005 and N = 3,999. Combining 5 of these bits using majority voting again gives, $Pout$ = 0.544888. This result is significantly not as good (p = 0.999697) as the two-step process using random walks described above. Majority voting in such a two-step process is not efficient and should never be used.

APPENDIX 1 – UTILITY PROGRAMS

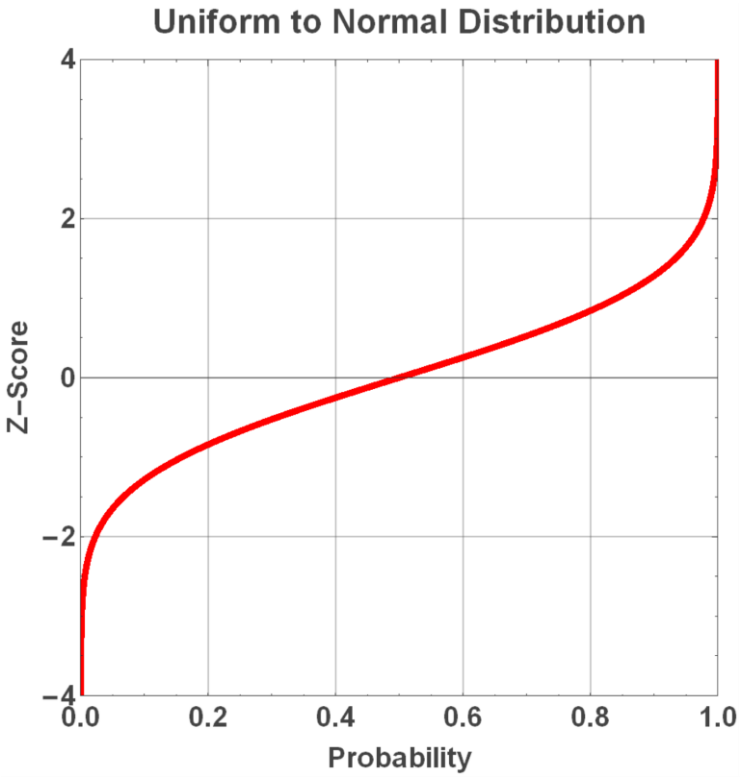
Z-Score to probability conversion program.



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;
0.5 + w (0.5 - (c2 + (c6 + c5 t + c4 t^2 + c3 t^3)/t^4)/(c1 Exp[0.5 z^2] t)) (*return value*)
```

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 (*return value*)
```