Widebandit User Manual

Version 0
June 11, 2021

0. Acknowledgements

Widebandit would never have happened without the assistance or consultations of, in chronological order since 2018: Claudio Maccone, Andrew Siemion, Greg Hellbourg, Chitwan Kaudan, Stelio Montebugnoli, and Roberto Lulli.

1. Purpose

Widebandit (a portmanteau of "wideband" (as in multichannel) and "bandit" (as in thief)) is a wideband signal entropy scanning pipeline. It can be used to find signals of unknown structure spread across many channels (typically but not necessarily frequencies). While a multitude of tools exist for the purpose of finding signals of *known* structure, such as Fourier transforms, the Karhunen-Loeve transfom, and neural networks; Widebandit offers the capability to detect signals that manifest as nothing more than changes in the statistical behavior of various channels over time. What might not reach statistical significance in any one channel might in fact achieve as much when small aberrations are summed across many channels which are not necessarily contiguous. It is the purpose of this software to detect such aggregate aberrations.

Two broad categories of signals are supported, namely, those with Gaussian priors, and those with purely emipirical point-cloud priors.

The purpose of this *document* is to explain the design and function of Widebandit while leaving the syntactical details to the help text which is built into each of its commands.

2. Data Format

Widebandit operates on "spectra" consisting of "channels". Each channel of each spectrum contains a measurement, expressed as a 32-bit IEEE754 floating-point value ("float"), which corresponds to a physical quantity such as power at a certain radio frequency, the signed fractional change in a stock price on a particular day of the week , or the amount of memory currently in use on a particular server. There is no requirement that channel N have any specific relationship to channel (N+1), apart from having the same units and being consistently defined across all spectra. For example, if we have a radio spectrum of power levels measured at 1 KHz, 2 KHz, 3 KHz, etc., they do not necessarily need to appear at the first, second, and third channels.

Note, however, that signals which span smaller groups of nearby channels ("channel neighborhoods") will be more easily detected because they are more likely to share a

scanning window. More on windows later. The bottom line is that if we want to, say, look at signals separated by exactly 10 KHz, then we should arrange the channels accordingly, for example:

1 KHz, 11 KHz, 21 KHz... 2 KHz, 12 KHz, 22 KHz...

...and so on. Thus this new spectrum is just the original one interleaved by steps of size 10. Interleaving can be expensive, and in any event it is usually intractable to try all possible channel permutations. In general, though, there is no motivation for interleaving, and even if there is, a putative signal might be detected anyway on account of its strength.

Back to the basics. A spectrum consisting of 9 floats would have a size of 36 bytes. But what if we have, say, 5 such spectra? Then the size is multiplied accordingly, to 180 bytes. One entire spectrum must be encoded, then another, and another, in a "stack" of spectra. In the foregoing example, we would say that the stack has a "depth" of 5.

There is no requirement that spectra within a given stack have some implied arrangement in time or space. They are just different spectra. Although, as with channel neighborhoods, smaller "spectrum neighborhoods" are more likely to share the same window at scan time, thereby supporting the detection of signals dispersed within them, so similar permutation considerations apply. In any case, it is still required that the units of all channels remain consistent throughout the stack. There is no requirement that channel N of one spectrum have the same meaning as channel N of another spectrum. So for instance, the first channel of the first spectrum might be the temperature on Monday, while the first channel of the second spectrum might be the temperature on Thursday. In practice, though, it would be wise to start by having all channels have the same meaning rather than merely the same units. One would only deviate from this if there were some known rotational or permutative difference between spectra.

Widebandit is capable of dealing with stacks of thousands of spectra, each of which consisting of millions of channels. A laptop can handle as much, often in minutes. Larger memories are required for larger stacks.

3. Pipeline Anatomy

Once could say that Widebandit is an information theoretic pipeline for the operating system commandline, in the sense that it ingests floats and produces entropy values expressed in nats (bits times (log 2)). Those values attempt to express how surprising or mundane a particular subset of those floats appears to be.

3.1. Spectrafy

The first stage in the pipeline is Spectrafy. This utility ingests entire folder trees of stacks of various depths. Thus one folder can contain multiple stacks of differing depths, along with other nested folders containing the same.

Spectrafy aggregates all floats from the same channel across many spectra into a probability density function (PDF) for each channel, which in this case is merely a point cloud -- a list of floats in no particular order. It automatically expands the PDF as necessary, creating free space for additional floats, although it is possible for the user to control how and when this occurs. The product of this process is a "Spectrafy archive", designated as such by its ".sfy" filename extension.

The significance of the PDF is that it represents our cummulative knowledge of some unknown physical PDF called the "generator". All we know about the latter is the set of measurements that it produced, which we have encoded as floats. (If you think you know more about the generator than just those measurements -- for example, how they should evolve over time -- then you should attempt to exploit that information, which is beyond the scope of Widebandit unless it happens to be Gaussian or lognormal. More on that in Section 5.)

Note that, whereas archiving preserves *channel* neighborhoods, it randomly permutes (and thereby destroys) *spectrum* neighborhoods because point cloud PDFs are always notionally, if not actually, sorted (from most negative to most positive) so as to facilitate fast lookup of later measurements. Spectrum neighborhoods are only meaningful at *scan* time -- not *archiving* time.

3.2. Downsample

The next stage is Downsample, a utility which extracts equally sized subsets of each PDF for each channel in an archive. The downsampling is done in a spatially fair manner, such that the resulting (smaller) PDF contains floats which are as close as possible to evenly spaced in the original PDF. For example, the original PDF might contain 10 floats:

{-2.6, -1.3, 0.2, 0.8, 1.5, 1.9, 2.7, 3.0, 3.1, 4.6}

which imply 11 regions of equal probability ("slices"):

0. x <= -2.6 (half-infinite leftmost slice)
1. -2.6 < x <= -1.3
...
9. 3.1 < x <= 4.6
10. 4.6 < x (half-infinite rightmost slice)

We could equally well have created only 10 slices by locating the equilibrium points ("walls") between slices at the midpoints between successive floats, rather than *passing through* them. But doing so would reduce sensitivity. (We want to set things up so that new measurements will land as close to the walls as possible, so that tiny shifts in the mean of the PDF are more likely to show up as significant changes in entropy. This is particularly important in multimodal systems which jump from one small locus to another, such as the wattage measurements of a 4-speed ceiling fan in various operating states.)

Now suppose we wish to downsample these 10 floats to just 2 (which would imply 3 new slices). Then, to be as fair as possible, we should select the floats at (1/3) and (2/3) of the way through the generator. Of course, we do not *know* the generator, so the original point cloud PDF will have to serve as a proxy. But this creates a problem: how do we sample floats from (1/3) and (2/3) of the way through it?

When expressed in terms of the array indexes into the *original* PDF, (10/3) and (20/3) are the respective centers of the zero-based array indexes from which to read the 2 walls for the *new* PDF (wherein we imagine that an array index has unit width). (In this sense, (1/2) would be based at the "center" of index zero which includes half of index one as well.) Of course, we cannot actually use fractional index values. But if we could, then we would have one unit of width centered at each index, hence ((10/3)-(1/2)) through ((10/3)+(1/2)) and ((20/3)-(1/2)) through ((20/3)+(1/2)). This simplifies to (17/6) through (23/6) and (37/6) through (43/6), respectively. This in turn implies that we should weight the float at index 2 by (1/6) and the float at index 3 by (5/6), then weight the float at index 6 by (5/6) and the float at index 7 by (1/6), in order to obtain the fairest estimate of the new wall locations. (Such weighted averaging is not currently implemented because Widebandit uses a form of floating-point emulation in order to enforce cross-platform determinism, which makes things complicated. In this regard, downsampling induces some degree of quantization error which negatively impacts sensitivity. This could be mitigated in the future, but is likely already insignificant with stacks containing statistically significant numbers of spectra.)

In this example, it would end up reading indexes 3 and 6, respectively, resulting in:

{0.8, 2.7}

In its simplest operating mode, Downsample delivers its output as a new Spectrafy archive. But it can also render multiple resolutions to equally many new archives in a single command so as to minimize redundant operations.

3.3. Slice

Slice uses a Spectrafy archive (whether original or downsampled, as there is no distinction) to convert a newly acquired stack into an array of whole numbers of the same dimensions such that each whole number takes only as many bytes as necessary to represent the maximum possible zero-based slice number implied by the number of floats per channel in said archive. So if the archive contains 255 floats (256 slices) per channel, then each whole number would comprise a single byte. But if the former contains 256 per channel (257 slices), than 2 bytes would be required per whole number.

It uses exactly the series of comparisons shown in the example in the previous section, in which each interior slice is open at the left extreme and closed at the right extreme. The leftmost half-infinite slice is closed on its right side. The rightmost half-infinite slice is open on its left side. But these are pedantic technicalities and somewhat arbitrary; we could equally well have inverted the orientation of the closed end of each slice.

Note that, while Spectrafy can output a single archive from many input stacks, and Downsample can output many archives from a single input archive, Slice outputs a single *array* of whole numbers from a single array (spectrum) of floats. (It can, however, inject the output array into any desired row of a stack of such arrays -- or simply append it.) This is because, in an online production situation, many instances of Slice should be processing data in parallel.

Of course, all of this preprocessing is only useful if the training period were sufficiently long to learn the background PDF of each channel well enough to discern signal from noise and interference in subsequently measured spectra. Now, in cases where there is no way to know whether we are observing the noise floor (like an empty patch of sky) or a signal (like a satellite broadcast), it would be useful to order all spectra by time, then remove a random subset of them for use as the background. (The only obvious downside is that this must be done *offline*, after all observations are complete. Widebandit currently lacks a utility for doing so.) In this way, while the background will contain some signal information rather than pure noise, the former would tend to be more temporally concentrated in the residual stack, facilitating detection during scanning. Scanning, for its part, is the subject of the next section.

3.4. Skan

The final stage in the pipeline, and the second stage in online production, is the entropy scanner itself. The input to Skan is a stack of whole numbers produced by Slice. Understanding its outputs requires some discussion of the entropy scanning process.

3.4.1. Spectrum Entropy Scan Topology

As explained above, the objective of the scan is to identify a channel neighborhood and spectrum neighborhood of extremal entropy in a newly observed stack of whole numbers, which implicitly shares no information with its corresponding background archive. That archive represents everything we know about the background. What we now wish to determine is how (un)like that background our new stack happens to be.

To do this, we have the newly acquired stack, which measures X channels wide by Y spectra high. Within these limits, we have a window which measures P channels wide by Q spectra high. We allow the upper left corner of the window to move everywhere within the entire XY area, with the constraint that the entire window must fit within that area. At each such (upper-left-corner) window position, which runs from (0, 0) to and including (X-P, Y-Q), we measure the entropy within the window, ultimately resulting in an array of floats of dimensions (X-P+1, Y-Q+1). Unless the user explicitly requests that it be saved, it will be discarded by Skan, as usually the only output of interest is the minimum or maximum entropy discovered and the corresponding window coordinates. These values are output as text which can then be appended to a CSV file for further sorting or processing.

Any new stack of spectra which are generated from the same generator as the background should result in approximately maximum entropy, given a window large enough to ensure statistical significance. This is because the background PDFs have been downsampled in an

approximately uniform manner, which should in turn ensure that the whole numbers derived from a new observation of the same background should be more or less uniformally distributed. (There is some Poisson jitter polluting the translation process, which results in quantization error, but its significance is merely to dampen sensitivity. In practice, this is unlikely to be the deciding factor in whether or not a given signal gets detected; decibels dominate over quantization error. Ultimately, pseudorandom dithering (wherein one of two neighboring slices is selected in a weighted random manner) could be used during Slice to reduce the quantization error without sacrificing determinism, but this is not currently implemented.)

But what is this "entropy" that we compute within each window?

3.4.2. Sliding Window Topology

It would be reasonable to assume that the entropy of the sliding window depends only upon the frequencies (rates of occurrence) of all whole numbers (symbols) within it. In its purest sense, that is in fact what a 2D entropy scan would imply. But because we are dealing with channels rather than a giant 2D bucket of semantically equivalent whole numbers, we must compute it differently.

Consider a signal which spans several channels. While each channel's point cloud PDF could then differ materially from its corresponding background PDF, there is no reason to assume that that difference -- say, a bias toward observing measurements in the middle 10% of the distribution -- would be consistent from one channel to another. They could all be biased in different ways.

In order to account for this, we must then compute the entropy of each channel within the window *independently*, then sum them up to arrive at the total window entropy.

Thus it helps to think of a window as more of a contiguous side-by-side tiling of columns rather than a simple 2D array.

Given this topology, we can now compute the entropy. We will discuss what "entropy" actually means below. For now, just think of low entropy as suggestive of simpler structure and high entropy as suggestive of more randomness.

3.4.3. Discount Nats

Given a definition of "entropy" (for which Skan supports 3 types), we can approximate, and often precisely derive, its maximum possible value. In any case, entropy is just an estimate of the number of nats required to (losslessly) store all of the whole numbers in a column of a window with no loss of information about their values or the sequence in which they appear. Put another way, if we preserve the PDF of those whole numbers along with some permutative information, then we could reconstruct the original sequence exactly; various entropy definitions presume to estimate the latter, or the sum, of those information costs. (A PDF of wholes is a sum of scaled Dirac delta functions which reach "fractionally infinite"

probability density at whole numbered values less than some *Z* on the real line. In this sense, such PDFs are interchangeable with histograms which map whole numbers to the number of times which they occur in a given list.)

In this context, it is assumed that each whole number in a given channel was chosen independently from a uniform PDF, and transitively that the float which gave rise to that whole number was chosen uniformally from the generator of the corresponding background PDF.

Claude Shannon and others have attempted to quantify this storage requirement. What we call "Shannon entropy" in this context means the cost of encoding only the permutative information because the aforementioned PDF of wholes is assumed to be known at decoding time. And indeed, we expect said PDF to be uniform. Inevitably, though, this as an erroneous assumption, particularly if a signal is present in the new observations. This is why Shannon entropy is empirically the least sensitive. Just the same, Skan supports this conventional and widely documented form of entropy. Later we will explain how to compute it along with the other supported entropy types.

For now, suffice to say that it is possible to accurately compute maximum possible Shannon (or any other) entropy. Then, once we have measured the entropy of a given channel within the window, we could subtract it from said maximum in order to provide a sense of proximity to maximum disorder. This quantity is expressed in units of "discount nats".

Skan outputs total window entropy as the total across all channels. However, it outputs discount nats (when discounting is enabled) as the *average per channel*. This inconsistency exists in order to support 2 very different use cases. For the sake of anomaly ranking, discount nats are more useful because (1) they are less variant across a wide range of Slice and Skan parameters which imply different entropy maxima and (2) on a relative basis, they are much more responsive to entropy changes than total entropy simply because the latter tends to be much closer to the maximum than to zero.

Higher discount nats imply more structure, which is the the opposite of higher total entropy. Therefore if we want to find a signal, then we want to maximize discount nats across various window sizes (which might for instance be selected according to an exponential distribution) and downsampling resolutions.

3.4.4. Optimal Downsampling Resolution

Suppose we want to scan a stack of spectra using a window height of *Q*. How does this intention inform our choice of downsampling resolution (quantizer) to request of Slice?

First of all, in this context, the "downsampling resolution" simply refers to the number of floats in per channel in the downsampled PDF. Recall that *N* floats imply (*N*+1) distinct slices corresponding to whole numbers of zero through *N*. Thus the downsampling resolution is *N* while the "quantizer" *Z* is (N+1). Now how can we choose *Z* in order to maximize sensitivity?

Intuitively, we should attempt to select *Z* and corresponding window height *Q* so that they are

as close as possible to equal, so as to maximize the volatility of the channel entropy within the window. (If (Z>>Q), then every whole number in the channel will be different all the time, so all channels will appear to have equally high entropy. If (Z<<Q), then we will lose too much information about the measurements that gave rise to the whole numbers, so entropy variations will be dominated by quantization error.) Admittedly, we have no proof that this *must* be the case for any given form of entropy, but intuition and experience do suggest as much.

This is not to say that we should choose a single value to use for Q and Z --merely that they should remain in sync. In practice, many different window sizes may be needed, depending upon what we intend to look for. We call an array of window dimensions (P, Q) a "window filter bank", analogous to the filter bank concept in the field of principal component analysis. (Currently, Skan takes only one value of (P, Q) per iteration.)

3.4.5. Computational Precision

Skan performs its calculations using fixed-point interval arithmetic, some of which ultimately manifests as an array of floats representing entropy values or hexadecimal fixed-point outputs representing the same. There are 2 options for precision: (1) 32.32 fixed-point implemented as 64-bit fractional intervals ("fractervals") or (2) 64.64 fixed-point implemented as 128-bit fractervals. The lower limit of a fracterval is thus a fixed-point value with equally many bits before and after the binary point, usually represented in hexadecimal for exactness and consistent text alignment. The upper limit is implicitly understated by a single granularity unit (2^(-32) or 2^(-64), respectively), but this is unlikely to matter in practice.

All outputs are rendered as scalars computed from their (hidden) interval limits. The rounding method by which they are converted is under control of the user via a commandline parameter.

3.4.6. Efficient Scanning

Notionally, Skan calculates the entropy of every possible window across each of its channels in parallel. In reality, it sweeps across the stack of whole numbered spectra in such a manner that very little changes from one window position to the next. This is exploited to massively reduce the computational burden.

In general, Skan's performance is limited by memory bandwidth, so there is unfortunately not much to be gained by making it multithreaded. Some cursory testing suggests that the ensuing performance improvement fraction would be only up to (1/3). On the other hand, there are other potential avenues of optimization, should the need arise. One of these is to add more "delta delta" lookup tables, which compute, effectively, discrete second derivatives of entropy. Another potential enhancement would be to downsample, slice, and scan all in one pipeline, although the performance could be poor if the background archive were large. In any event, if such issues are of concern to you, then visit Widebandit's issue tracker on Github.

4. Entropy Types

As of this writing, Skan supports 3 different entropy types, but all of them have existing mathematical foundations having nothing to do with Widebandit.

4.1 Shannon Entropy

As stated above, in the context of this manual, we define this as the amount of information, expressed in nats, which is required to encode a sequence of whole numbers chosen independently and according to a PDF which exactly equals their *actual* PDF. This formula is not the same as the entropy function, although they share a common basis in Shannon's work in information theory. Within a given channel of a given window, this quantity *S* is given by:

$$S \equiv QlnQ - \sum_{W=0}^{Z-1} F(W)lnF(W)$$

where "ln" denotes the natural logarithm, *Q* and *Z* are as defined above, and F(W) is the number of times that whole number W occurs -- the "frequency" of W. (For all cases in which F(W) is zero, we treat (ln F(W)) as though it were also zero.)

4.1.1. Maximimum Shannon Entropy

In principle, *S* is maximized when *F*(*W*) is equal for all *W*. However, this maybe impossible in practice because *Z* is not a multiple of *Q*. In such cases, we the best we can do is to have 2 values of *F*(*W*):

1. (*Q* / *Z*) for (*Q* - (*Q* % *Z*)) values of *W*.
2. (*Q* / *Z* + 1) for (*Q* % *Z*) values of *W*.

where slash implies an integer quotient and percent implies the remainder.

4.2. Agnostic Entropy (Agnentropy)

Agnentropy is, empirically, of comparable scanning speed as Shannon entropy, but exhibits a modest improvement in sensitivity, most likely because it attempts to roughly account for the encoding cost of the PDF which the latter takes for granted.

Without going into details on its derivation, this quantity *A* is given by:

$$A \equiv ln(Q + Z - 1)! - ln(Z - 1)! - \sum_{W=0}^{Z-1} lnF(W)!$$

where the exclamation point denotes the factorial function. (Note that zero factorial and one factorial are both one.)

4.2.1 Maximum Agnentropy

Under the same conditions which maximize *S*, *A* is also maximized.

4.3. Logfreedom

Empirically, and by design from first principles in combinatorial encoding, logfreedom is several times more sensitive than either of the foregoing entropy measures when it comes to the task of detecting aberrations from uniform distributions. (To be pedantic, logfreedom increases monotonically with the likelihood that a given PDF of wholes was produced by a uniform generator. By the way, a PDF of wholes necessarily involves a superposition of scaled Dirac delta functions, which in this case serve to pick off whole numbers on the real line. Anyway, by way of contrast with the foregoing entropy measures, a perfectly uniform PDF of wholes does *not* have maximum logfreedom except in one trivial case. This is because, from combinatorics, uniform generators are more likely to give rise to slightly *nonuniform* distributions of wholes, even when *Q* is some multiple of *Z*.) As explained above, new stacks of spectra which contain only further background measurements should give rise to distributions of whole numbers up to (*Z*-1), as though sampled fro ma uniform distribution, thereby resulting in approximately maximum logfreedom. But if those spectra were to be sampled from a different distribution, perhaps due to the presence of a signal, then the wholes generated from those samples should result in lesser logfreedom.

Here we need to introduce the concept of "population", which is defined as the number of *unique* values of *W* which share the same *F*(*W*). (In this sense, it's just the "frequency of some given frequency". If there are 5 different values of *W* having frequency 7, then we say that "the population of 7 is 5".) Thus we have the population *H*(*F*) in the following formula for the logfreedom *L*, which attempts to even more rigorously account for the cost of encoding Shannon's missing PDF:

$$L \equiv lnQ! + lnZ! - \sum_{F=0}^{Q} lnH(F)! - \sum_{F=2}^{Q} H(F) lnF!$$

wherein the vast majority of population terms are usually zero. Note that, in the interest of brevity, this formula makes no mention of the whole numbers themselves -- just their respective frequencies and, in turn, their respective populations.

4.3.1 Maximum Logfreedom

Unfortunately, there is no known formula by which to compute the maximum possible value of *L*, given *Q* and *Z*. Skan does this heuristically in trivial time, and fortunately the results appear to be optimal or close enough for practical purposes in all tested cases. Should it underestimate logfreedom in some case, the result would simply be fracterval underflow, which would cause a warning to be printed, assuming that the user has enabled such outputs. It's so unexpected that it's likelier an indication of a bug than actual underflow, so please

report it on Github if it occurs for unknown reasons.

5. Utilities for Gaussian or Lognormal Background PDFs

It is common in observations of physical phenomena to encounter Gaussian (normal) and lognormal distributions. (Taking the logs of all the (positive) samples in a lognormal distribution will yield a Gaussian, so we can consider them as interchangeable.) For this reason, Widebandit features a dedicated Gaussian pipeline. Support for other distribution archetypes (Poisson, Gamma, etc.) could be added in the future.

The pipeline is based on the Gaussian information criterion (GIC), and specifically its ranker and ratio manifestations. This is all explained in a paper available at (https://vixra.org/abs/2106.0036), but the basic idea is to be able to detect noncontiguous wideband signals by comparing a background stack of spectra with an "experimental" stack, as would typically occur when performing "on-target-versus-off-target" radio astronomy searches, for example. Each background PDF is presumed to be the Gaussian (or equivalently, lognormal) implied by its constituent samples, whereas their respective experimental PDFs are not assumed to have any particular shape. By evaluating the GIC ranker (a scale-free information measure) from the former to the latter, we can quantify the degree of "Gaussianity" of each experimental channel. The "least Gaussian" channels are then pushed to the top of a list of potential signal locations. Furthermore, by sorting the GIC rankers in descending order, then taking their successive ratios, we might hope to find a "cliff" which bifurcates the channels into those most likely to be (un)involved in a wideband signal. Finally, by comparing these cliff ratios across multiple stacks of experimental spectra, we might manage to identify those most worthy of more computationally intensive analysis, thereby sparing precious resources for the tiny minority of remarkable cases.

Currently, Widebandit's Gaussian pipeline does not support windowed scanning; it consumes an entire stack of spectra at once. As such, it's more of a proof-of-concept which could be enhanced at a later date, should it prove to be useful.

5.1. Gaussify

This utility ingests Spectrafy archives and produces a list of (mean, variance) pairs describing the Gaussians derived from all samples in corresponding channels. This is done according to the textbook formulae involving sums of samples and their squares, after optionally taking logs in order to support lognormals as well. The mean and variance of each channel, encoded as 64-bit IEEE754 double-precision floating-point values ("doubles"), are then saved to a binary file with a simple format for further analysis. The format is fully disclosed in the help text displayed when Gaussify is run with no parameters. Unlike the foregoing utilities, it does not use interval arithmetic and is therefore subject to the vagaries of floating-point implementation. There is no good reason for this apart from a lack of development time. (Whereas the rest of Widebandit was developed in a piecemeal fashion over the course of a few years, its Gaussian pipeline was implemented from first principles in a few *weeks*.)

5.2. GICRank

This utility can convert a pair of Gaussify binary files -- one background and one experimental -- into a list of GIC rankers sorted in descending order, and paired with their zero-based channel indexes. Optionally, it can also compute successive GIC ratios from this list, then sort them along with their corresponding channel indexes, in order to support cliff searches as explained above. As with Gaussify, the outputs are binary files having a straightforward format described in the help text.

6. Installation

Unpack the source code into any folder. Since Widebandit is written entirely in C, any recent C compiler should suffice. GCC and Linux are recommended but not strictly necessary.

6.1. Compilation

We can build the entire project with simply:

make widebandit

or, in order to enable memory corruption checking along with other debug macros in debug.h, we can do:

make widebandit_debug

instead. (In practice, the corruption checks do not seem to add much to execution time, so it might be a good idea to enable the debugger all the time. Please submit an issue at the Github repo if you observe any messages about memory leaks or segmentation faults.)

Optionally, we can build individual components:

make downsample(_debug)
make gaussify(_debug)
make gicrank(_debug)
make slice(_debug)
make skan(_debug)
make spectrafy(_debug)

All executables end up in the tmp subfolder. (Despite being the name of the project, there is no "widebandit" executable.)

You can run any of the following commands without parameters in order to see a printout of the proper syntax:

tmp/downsample
tmp/gaussify
tmp/gicrank

tmp/slice
tmp/skan
tmp/spectrafy

The syntax is subject to change from time to time, so it will not be reproduced here.

7. Troubleshooting

Post an issue at the Github repo if you think you've found a bug or would like to request a feature. Likewise if you have trouble getting anything to run properly, which might imply the need for better documentation.