Note S1 Methods

The schema of the proposed method is illustrated in Fig. S1. Smash++ takes as inputs a reference and a target file and produces as output a position file, which is then fed to the Smash++ visualizer to produce an SVG image. This process has eight major stages: (1) compression of the original target file, based on the model of original reference file, (2) filtering and segmentation of the compressed file, (3) reference-free compression of the segmented files, obtained by the previous stage, (4) compression of the original reference file, based on the model of segmented files obtained by stage 2, (5) filtering and segmentation of the compressed files, (6) reference-free compression of the segmented files, that are obtained by the stage 5, (7) aggregating positions, generated by stages 3 and 6, and (8) visualizing the positions. The following sections describe the process in detail.

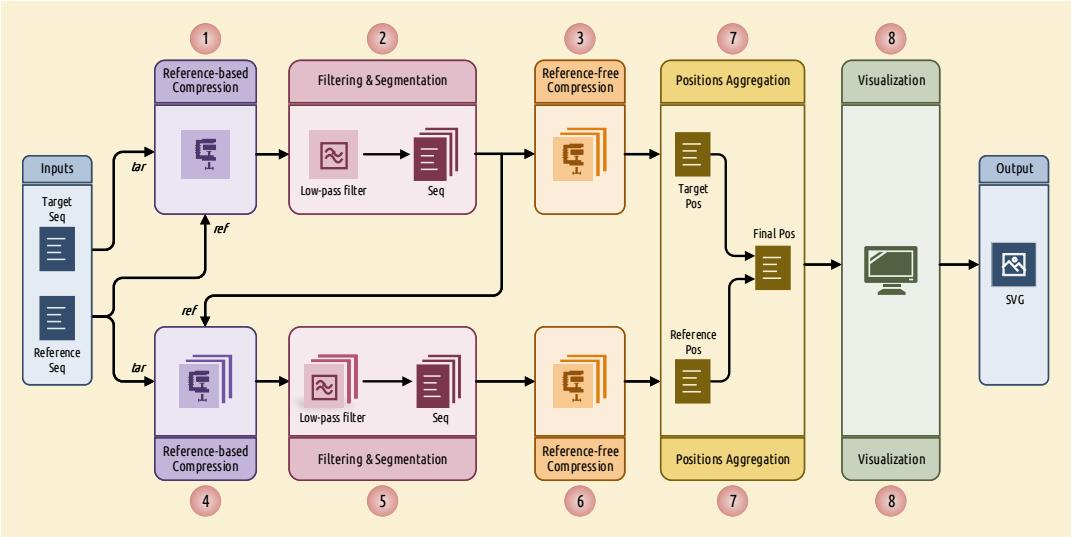

Fig. S1. The schema of Smash++. The process of finding similar regions in reference and target sequences and also, computing redundancy in each region includes eight stages. Finally, Smash++ outputs a *.pos file that includes the positions of the similar regions, and can be then visualized, resulting in an SVG image.

S1.1 Data modeling

Smash++ works on the basis of cooperation between finite-context models (FCMs) and substitutional tolerant Markov models (STMMs). Applying these models on various contexts provides probability and weight values, illustrated in Fig. S2a, which are then mixed (by multiplication and addition, shown in Fig. S2b) to provide the final probability (P) of occurring an input symbol. The following subsections describe FCMs and STMMs in detail.

Finite-context model (FCM) A finite-context model considers Markov property to estimate the probability of the next symbol in an information source, based on the past k symbols (a context of size k) [1–3]. Denoting the context as $c_{k,i} = s_{i-k}s_{i-k+1}\dots s_{i-2}s_{i-1}$, the probability of the next symbol s_i in an information source S, which is posed at i, can be estimated as

$$P_m(s_i|c_{k,i}) = \frac{N(s_i|c_{k,i}) + \alpha}{N(c_{k,i}) + \alpha|\Theta|}, \qquad \textbf{(Eq. S1)}$$

in which m stands for model (FCM in this case), $N(s_i|c_{k,i})$ shows the number of times that the information source has generated symbol s_i in the past, $|\Theta|$ denotes size of the alphabet Θ, $N(c_{k,i}) = \sum_{b\in\Theta} N(b|c_{k,i})$ represents the total number of events occurred for the context $c_{k,i}$ and α allows to keep a balance between the maximum likelihood estimator and the uniform distribution. Eq. S1

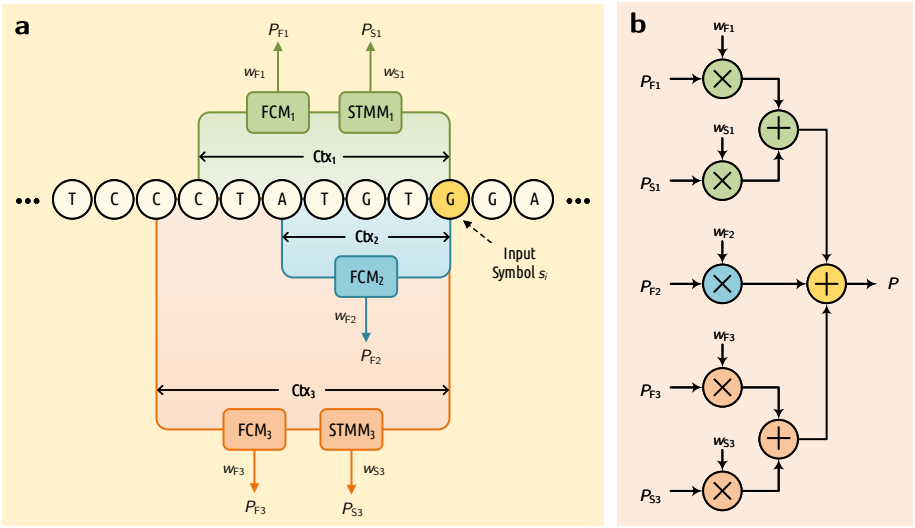

Fig. S2. Data modelling by Smash++. (a) cooperation between finite-context models (FCMs) and substitutional-tolerant Markov models (STMMs). Note that each STMM needs to be associated with an FCM. (b) probability of an input symbol is estimated by employing the probability and weight values that have been obtained from processing previous symbols.

turns to the Laplace estimator, for $\alpha = 1$, and also behaves as a maximum likelihood estimator, for large number of events i [4].

Substitutional tolerant Markov model (STMM) A substitutional tolerant Markov model [5] is a probabilistic-algorithmic model that assumes at each position, the next symbol in the information source is the symbol which has had the highest probability of occurrence in the past. This way, an STMM ignores the real next symbol in the source. Denoting the past k symbols as $c_{k,\,i} = s_{i-k}s_{i-k+1}\ldots s_{i-2}s_{i-1}$, the probability of the next symbol s_i, can be estimated as

$$P_m(s_i|c'_{k,\,i}) = \frac{N(s_i|c'_{k,\,i}) + \alpha}{N(c'_{k,\,i}) + \alpha|\Theta|}, \qquad \textbf{(Eq. S2)}$$

where N represents the number of occurrences of symbols, that is saved in memory, and $c'_{k,\,i}$ is a copy of the context $c_{k,\,i}$ which is modified as

$$c'_{k,\,i} = \underset{\forall b \in \Theta}{\operatorname{argmax}}\, P_m(b|c'_{k,\,i}). \qquad \textbf{(Eq. S3)}$$

STMMs can be used along with FCMs to modify the behavior of Smash++ in confronting with nucleotide substitutions in genomic sequences. These models have the potential to be disabled, to reduce the number of mathematical calculations and consequently, increase the performance of the proposed method. Such operation is automatically performed using an array of size k (the context size), named history, which preserves the past k hits/misses. Seeing a symbol in the information source, the memory is checked for the symbol with the highest number of occurrences. If they are equal, a hit is saved in the history array; otherwise, a miss is inserted into the array. Before getting to store a hit/miss in the array, it is checked for the number of misses and in the case they are more than a predefined threshold t, the STMM will be disabled and also the history array will be reset. This process is performed for each symbol in the sequence.

This example shows the distinction between a finite-context model and a substitutional tolerant Markov model. Assume, the current context at position i is $c_{11,\,i}$ = GGCTAACGTAC, and the number of occurrences of symbols saved in memory is A = 10, C = 12, G = 13 and T = 11. Also, the symbol to appear in the sequence is T. An FCM would consider the next context as $c_{11,\,i+1}$ = GCTAACGTACT, while an STMM would consider it as $c'_{11,\,i+1}$ = GCTAACGTACG, since the base G is the most probable symbol, based on the number of occurrences stored in memory.

Cooperation of FCMs and STMMs When FCMs and STMMs are in cooperation, the probability of the next symbol s_i in an information source S, at position i, can be estimated as

$$P(s_i) = \sum_{m \in M_F} P_m(s_i|c_{k,\,i})\; w_{m,\,i} + \sum_{m \in M_S} P_m(s_i|c'_{k,\,i})\; w'_{m,\,i}, \quad \forall s_i \in S,\ 1 \le i \le |S|,\ 1 \le k \le i-1 \tag{Eq. S4}$$

in which M_F and M_S denote sets of FCMs and STMMs, respectively, $P_m(s_i|c_{k,\,i})$ shows the probability of the next symbol estimated by the FCM, $P_m(s_i|c'_{k,\,i})$ represents this probability estimated by the STMM, and $w_{m\,i}$ and $w'_{m,\,i}$ are weights assigned to each model based on its performance. We have

$$\begin{aligned} &\forall m \in M_F: \quad w_{m,\,i} \propto (w_{m,\,i-1})^{\gamma_m} P_m(s_i|c_{k+1,\,i-1}), \\ &\forall m \in M_S: \quad w'_{m,\,i} \propto (w'_{m,\,i-1})^{\gamma'_m} P_m(s_i|c'_{k+1,\,i-1}), \end{aligned} \tag{Eq. S5}$$

where γ_m and $\gamma'_m \in [0, 1)$ are forgetting factors predefined for each model. Also,

$$\sum_{m \in M_F} w_{m,\,i} + \sum_{m \in M_S} w'_{m,\,i} = 1. \tag{Eq. S6}$$

By experimenting different forgetting factors for models, we have found that higher factors should be assigned to models that have higher context-order sizes (less complexity) and vice versa. As an example, when the context size $k = 6$, γ_m or $\gamma'_m \simeq 0.9$ and when $k = 18$, γ_m or $\gamma'_m \simeq 0.95$ would be appropriate choices. These values show that forgetting factor and complexity of a model are inversely related.

Storing models in memory The FCMs and STMMs include, in fact, count values which need to be saved in memory. For this purpose, four different data structures have been employed considering the context-order size k, as follows:

$$\text{data structure} = \begin{cases} \text{table of 64 bit counters,} & 1 \le k \le 11 \\ \text{table of 32 bit counters,} & k = 12, 13 \\ \text{table of 8 bit approximate counters,} & k = 14 \\ \text{Count-Min-Log sketch of 4 bit counters.} & k \ge 15 \end{cases}$$

The table of 64 bit counters, that is shown in Fig. S3a, simply saves number of events for each context. The table of 32 bit counters saves in each position the number of times that the associated context is observed. When a counter reaches to the maximum value $2^{32} - 1 = 4294967295$, all the counts will be renormalized by dividing by two, as shown in Fig. S3b.

The approximate counting is a method that employs probabilistic techniques to count large number of events, while using small amount of memory [6]. Fig. S4 shows the algorithm for two major functions associated with this method, UPDATE and QUERY. In order to update the counter, a pseudo-random number generator (PRNG) is used the number of times of the counter's current value to simulate flipping a coin. If it comes up 0/Heads each time or 1/Tails each time, the counter will be incremented. Fig. S3c shows the difference between arithmetic and approximate counting, and also the values which are actually stored in memory. Note that since an approximate counter represents the actual count by an order of magnitude estimate, one only needs to save the exponent. For example, if the actual count is 8, we store it in memory as $\log_2 8 = 3$.

The Count-Min-Log Sketch (CMLS) is a probabilistic data structure to save frequency of events in a table by means of a family of independent hash functions [7]. The algorithm for updating and querying the counter is shown in Fig. S5. In order to update the counter, its current value is hashed with d independent hash functions. Then, a coin is flipped the number of times of the counter's current value, employing a pseudo-random number generator. If it comes up 0/Heads each time or 1/Tails each time, the minimum hashed values (out of d values) will be updated, as shown in Fig. S3d.

The CMLS requires a family of pairwise independent hash functions $H = \{h : U \to [m]\}$, in which

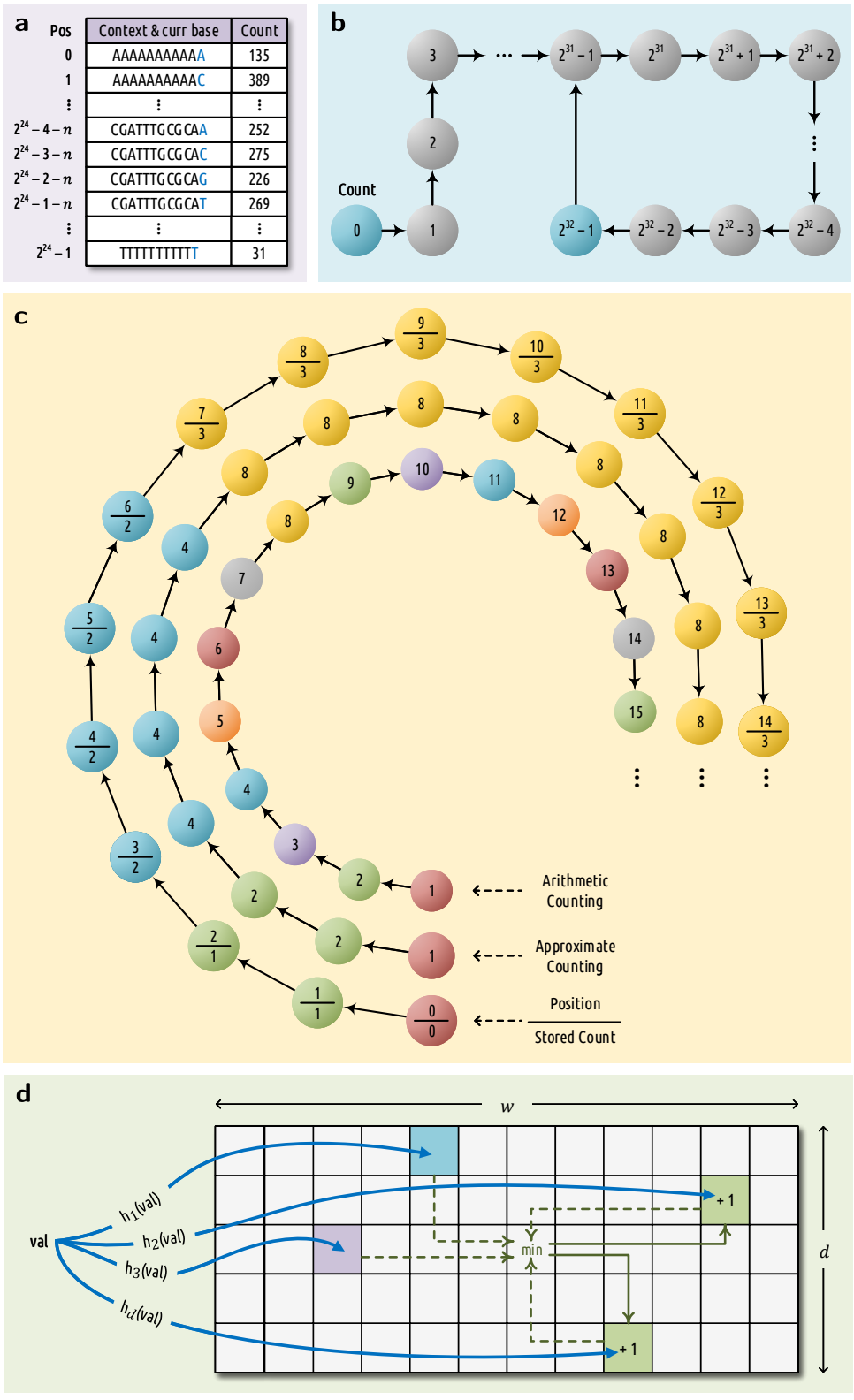

Fig. S3. The data structures used by Smash++ to store the models in memory. (a) table of 64 bit counters that uses up to 128 MB of memory, (b) table of 32 bit counters that consumes at most 960 MB of memory, (c) table of 8 bit approximate counters with memory usage of up to 1 GB and (d) Count-Min-Log sketch of 4 bit counters which consumes up to $\frac{1}{2}w \times d$ B of memory, e.g., if $w = 2^{30}$ and $d = 4$, it uses 2 GB of memory.

```
1: function IncreaseDecision(x)
2:     return True with probability 1/2^x, else False
3: end function

4: function Update(x)
5:     c ← table[x]
6:     if IncreaseDecision(c) = True then
7:         table[x] ← c + 1
8:     end if
9: end function

10: function Query(x)
11:     c ← table[x]
12:     return 2^c − 1
13: end function
```

Fig. S4. Approximate counting update and query.

each function h maps some universe U to m bins. To have this family, we use universal hashing by randomly selecting a hash function from a universal family in which $\forall x, y \in U,\ x \neq y:\ \ P_{h\in H}[h(x) = h(y)] \leq \frac{1}{m}$. The hash function can be obtained by

$$h_{a,b}(x) = ((ax + b) \mod p) \mod m, \qquad \textbf{(Eq. S7)}$$

where $p \geq m$ is a prime number and a and b are randomly chosen integers modulo p with $a \neq 0$.

S1.2 Finding similar regions

To find similar regions in reference and target files, a quantity is required for measuring the similarity. We use "per symbol information content" for this purpose, which can be calculated as

$$I(s_i) = -\log_2 P(s_i) \quad \text{bpb}, \quad \forall s_i \in S,\ 1 \leq i \leq |S| \qquad \textbf{(Eq. S8)}$$

where $P(s_i)$ denotes the probability of the next symbol s_i in the information source S, obtained by Equation S4, and also "bpb" stands for bit per base.

The information content is the amount of information required to represent a symbol in the target sequence, based on the model of the reference sequence. The less the value of this measure is for two regions, the more amount of information is shared between them, and therefore, the more similar are the two regions. Note that a version of this measure has been introduced in [4], which employs a single FCM to calculate the probabilities. In this paper, however, we exploit a cooperation between multiple FCMs and STMMs for highly accurate calculation of such probabilities.

The procedure of finding similar regions in a reference and a target sequence, illustrated in Fig. S6, is as follows: after creating the model of the reference, the target is compressed based on that model and the information content is calculated for each symbol in the target. Then, the content of the whole target sequence is smoothed by Hann window [8], which is a discrete window function given by $w[n] = 0.5 - 0.5\ \cos\left(\frac{2\pi n}{N}\right)$, where $0 \leq n \leq N$ and length of the window is $N + 1$. Next, the smoothed information content is segmented considering a predefined threshold, meaning that the regions with the content greater than the threshold are filtered out. This is carried out for both regular and inverted repeat homologies and at the end, the result would be the regions in the target sequence that are similar to the reference sequence (Fig. S6a). The described phase repeats for all of the target regions found, in the way that after creating the model for each region, the whole reference sequence is compressed to find those regions in the reference that are similar to each of the target regions (Fig. S6b). The final result would have the form of Fig. S6c.

Input: sketch width w, sketch depth d, m bins, prime $p \geq m$, randomly chosen integers $a_{1..d}$ and $b_{1..d}$ modulo p with $a \neq 0$

```
1: function HASH(k, x)                    ▷ Universal hash family
2:     return ((a_k x + b_k) mod p) mod m
3: end function

4: function MINCOUNT(x)
5:     minimum ← 15                       ▷ Biggest 4 bit number
6:     for k ← 1 to d do
7:         h ← HASH(k, x)
8:         if sketch[k][h] < minimum then
9:             minimum ← sketch[k][h]
10:        end if
11:    end for
12:    return minimum
13: end function

14: function INCREASEDECISION(x)
15:    return True with probability 1/2^x, else False
16: end function

17: function UPDATE(x)
18:    c ← MINCOUNT(x)
19:    if INCREASEDECISION(c) = True then
20:        for k ← 1 to d do
21:            h ← HASH(k, x)
22:            if sketch[k][h] = c then
23:                sketch[k][h] ← c + 1
24:            end if
25:        end for
26:    end if
27: end function

28: function QUERY(x)
29:    c ← MINCOUNT(x)
30:    return 2^c − 1
31: end function
```

Fig. S5. Count-Min-Log Sketch update and query.

S1.3 Computing complexities

After finding the similar regions in reference and target sequences, we evaluate redundancy in each region, knowing that it is inversely related to Kolmogorov complexity, i.e., the more complex a sequence is, the less redundant it will be [9]. The Kolmogorov complexity, K, of a binary string s, of finite length, is the length of the smallest binary program p that computes s in a universal Turing machine and halts. In other words, $K(s) = |p|$ is the minimum number of bits required to computationally retrieve the string s [10, 11].

The Kolmogorov complexity is not computable, hence, an alternative is required to compute it approximately. It has been shown in the literature that a compression algorithm can be employed for this purpose [12–14]. In this paper, we employ a reference-free compressor to approximate the complexity and consequently, the redundancy of the founded similar regions in the reference and the target sequences. This compressor works based on cooperation of FCMs and STMMs, which is described in detail in Section S1.1. Note that the difference between reference-based and reference-free version of such compressor is that in the former mode, a model is first created for the reference

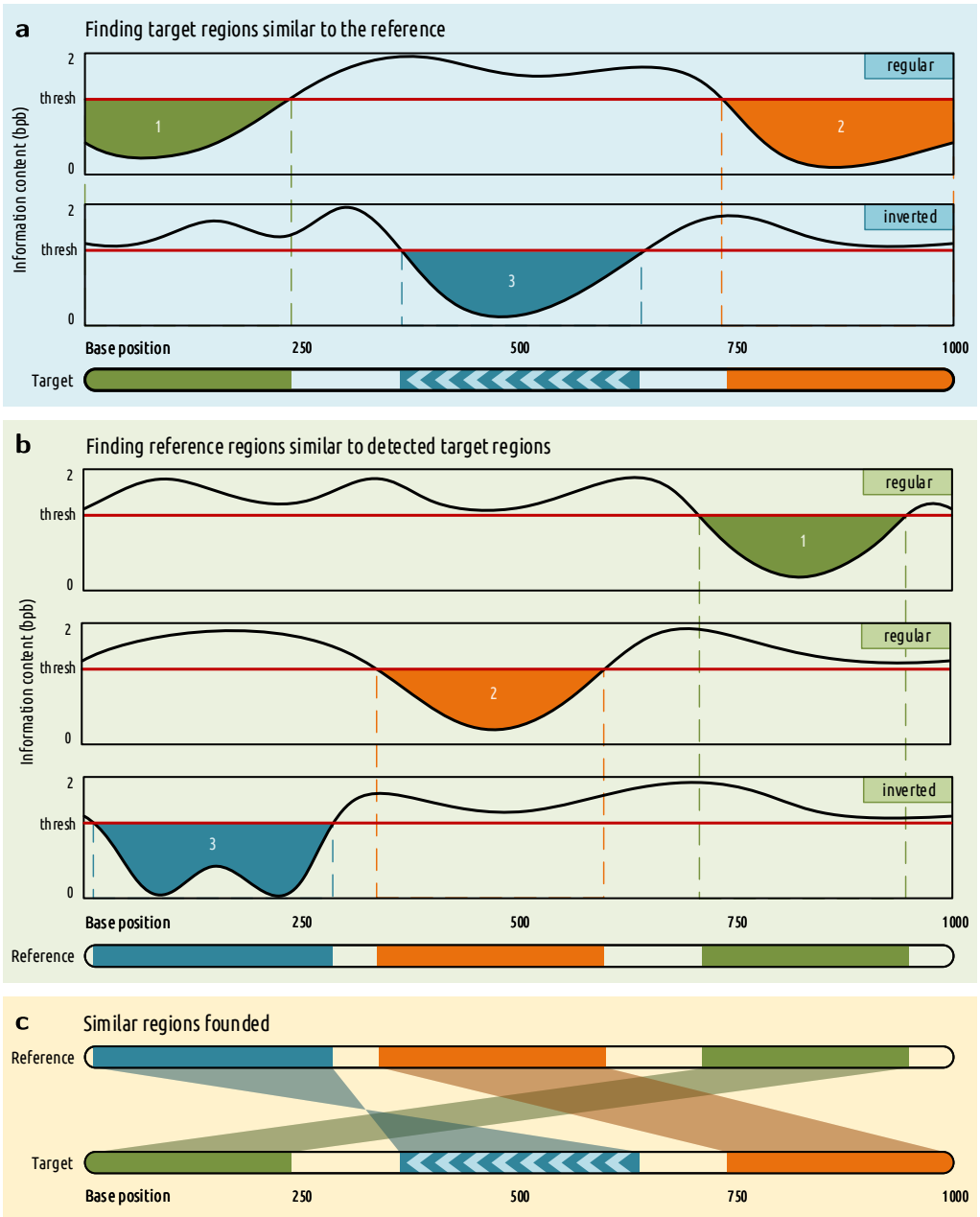

Fig. S6. Finding similar regions in reference and target sequences. Smash++ finds, first, the regions in the target that are similar to the reference, and then, finds the regions in the reference that are similar to the detected target regions. This procedure is performance for both regular and inverted homologies.

sequence and then, the target sequence is compressed based on that model, while in the latter mode, the model is progressively created at the time of compressing the target sequence.

Note S2 Tool availability and implementation

Smash++ is implemented in `C++` language and is available at [15]. This tool is able to find and visualize rearrangements in sequences, FASTA and FASTQ files; although, it is highly recommended to use sequences as input. In the following sections, we describe installing and running the Smash++ tool.

S2.1 Install

To install Smash++ on various operating systems, follow the instructions below. Note that the precompiled executables are available for 64 bit operating systems in the "bin/" directory.

Linux

- Install "git" and "cmake":

```
sudo apt update
sudo apt install git cmake
```

- Clone Smash++ and install it:

```
git clone https://github.com/smortezah/smashpp.git
cd smashpp
./install.sh
```

macOS

- Install "Homebrew", "git" and "cmake":

```
/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
    install/master/install)"
brew install git cmake
```

- Clone Smash++ and install it:

```
git clone https://github.com/smortezah/smashpp.git
cd smashpp
./install.sh
```

Windows

- Download and install "CMake", e.g., from https://github.com/Kitware/CMake/releases/download/v3.14.4/cmake-3.14.4-win64-x64.msi. Make sure to add it to the system PATH. For example, if CMake is installed in "C:\Program Files", add "C:\Program Files\CMake\bin" to the system PATH.

- Download and install "mingw-w64", e.g., from https://sourceforge.net/projects/mingw-w64/files/latest/download. Make sure to add it to the system PATH. For example, if it is installed in "C:\mingw-w64", add "C:\mingw-w64\mingw64\bin" to the system PATH.

- Download and install "git", from https://git-scm.com/download/win.

- Clone Smash++ and install it:

```
git clone https://github.com/smortezah/smashpp.git
cd smashpp
.\install.bat
```

S2.2 Run

A reference file and a target file are clearly mandatory to run Smash++ (without visualization). Running

```
./smashpp
```

provides the following guide:

```
SYNOPSIS
  ./smashpp [OPTIONS]  -r <REF-FILE>  -t <TAR-FILE>

OPTIONS
  Required:
  -r  <FILE>          = reference file (Seq/FASTA/FASTQ)
  -t  <FILE>          = target file    (Seq/FASTA/FASTQ)

  Optional:
  -l  <INT>           = level of compression: [0, 5]. Default -> 0
  -m  <INT>           = min segment size: [1, 4294967295]     -> 50
  -e  <FLOAT>         = entropy of 'N's: [0.0, 100.0]         -> 2.0
  -n  <INT>           = number of threads: [1, 8]             -> 4
  -f  <INT>           = filter size: [1, 4294967295]          -> 256
  -ft <INT/STRING>    = filter type (windowing function):     -> hann
                        {0|rectangular, 1|hamming, 2|hann,
                        3|blackman, 4|triangular, 5|welch,
                        6|sine, 7|nuttall}
  -fs [S][M][L]       = filter scale:                         -> L
                        {S|small, M|medium, L|large}
  -d  <INT>           = sampling steps                        -> 1
  -th <FLOAT>         = threshold: [0.0, 20.0]                -> 1.5
  -rb <INT>           = ref beginning guard: [-32768, 32767]  -> 0
  -re <INT>           = ref ending guard: [-32768, 32767]     -> 0
  -tb <INT>           = tar beginning guard: [-32768, 32767]  -> 0
  -te <INT>           = tar ending guard: [-32768, 32767]     -> 0
  -dp                 = deep compression                      -> no
  -nr                 = do NOT compute self complexity        -> no
  -sb                 = save sequence (input: FASTA/FASTQ)    -> no
  -sp                 = save profile (*.prf)                  -> no
  -sf                 = save filtered file (*.fil)            -> no
  -ss                 = save segmented files (*.s[i])         -> no
  -sa                 = save profile, filetered and           -> no
                        segmented files
  -rm k,[w,d,]ir,a,g/t,ir,a,g:...
  -tm k,[w,d,]ir,a,g/t,ir,a,g:...
                      = parameters of models
                <INT>  k:  context size
                <INT>  w:  width of sketch in log2 form,
                           e.g., set 10 for w=2^10=1024
                <INT>  d:  depth of sketch
                <INT>  ir: inverted repeat: {0, 1, 2}
                           0: regular (not inverted)
                           1: inverted, solely
                           2: both regular and inverted
              <FLOAT>  a:  estimator
              <FLOAT>  g:  forgetting factor: [0.0, 1.0)
                <INT>  t:  threshold (no. substitutions)
  -ll                 = list of compression levels
  -h                  = usage guide
  -v                  = more information
  --version           = show version
```

The arguments "-r" and "-t" are used to specify the reference and the target, respectively, which are highly recommended to have short names. Level of compression, that is an integer between 0 and 5, can be determined with "-l". By setting "-m" to an integer value, only those regions in the reference file that are bigger than that value would be able to be considered for compression.

In implementation of the reference-based compression, we have replaced 'N' bases in the references and the targets with 'A's and 'T's, respectively. On reference-free compression, they are replaced with 'A's, in both references and targets. If a user tends to replace 'N' bases in a sequence with a normal distribution of 'A', 'C', 'G' and 'T's, he/she can employ GOOSE toolkit [16]. Note that we have set by default the entropy of 'N's to 2.0, however, it can be changed to a value of interest using "-e" option.

Creating multiple finite-context models and substitutional-tolerant Markov models can be done in a multi-threaded fashion by setting "-n" to an integer.

In the process of finding similar regions in the reference and the target sequences, the information content that would be obtained by compression needs to be filtered. Size of the window and type of the windowing function can be set by "-f" and "-ft" options, respectively. Besides Hann window that is used by default to smooth the information content (profile), we have implemented several other windowing functions, including Blackman [8], Hamming [17], Nuttall [18], rectangular [19], sine [20], triangular [21] and Welch [22] windows. These functions are given by

$$
\begin{aligned}
&w[n] = 1, && \text{(rectangular)}\\
&w[n] = 1 - \left|\tfrac{n-N/2}{L/2}\right|, \quad L = N, && \text{(triangular/Bartlett)}\\
&w[n] = 1 - \left(\tfrac{n-N/2}{N/2}\right)^2, && \text{(Welch)}\\
&w[n] = \sin\left(\tfrac{\pi n}{N}\right), && \text{(sine)}\\
&w[n] = 0.54348 - 0.45652\ \cos\left(\tfrac{2\pi n}{N}\right), && \text{(Hamming)}\\
&w[n] = 0.42659 - 0.49656\ \cos\left(\tfrac{2\pi n}{N}\right) + 0.07685\ \cos\left(\tfrac{4\pi n}{N}\right), && \text{(Blackman)}\\
&w[n] = 0.35577 - 0.48740\ \cos\left(\tfrac{2\pi n}{N}\right) + 0.14423\ \cos\left(\tfrac{4\pi n}{N}\right) - 0.01260\ \cos\left(\tfrac{6\pi n}{N}\right), && \text{(Nuttall)}
\end{aligned}
$$

(Eq. S9)

and are plotted in Fig. S7. Scale of the filter can be set as S (small), M (medium) or L (large), using "-fs". Also, instead of considering the whole information content, the user is able to make samples of it by steps of which size can be determined by "-d".

For the purpose of segmenting the filtered information content, the average entropy of reference-based compression is used by default as the threshold, but the threshold can be altered by "-th" option.

Smash++ is capable of finding even very small similar regions in two sequences. However, we have found experimentally that when it is running in a very sensitive mode, there might be some cases in which the size of similar regions in the reference and the target are not balanced. These cases can be handled by "-rb", "-re", "-tb" and "-te" options, that can resize the beginning and ending guards of reference and target regions, respectively. For example, if "-tb 10" is used, the first 10 bases in each target region will be ignored, which results in smaller regions. Note that when the guard sizes of target regions are increased, the models built from these regions would be slightly different than the original models; consequently, the sizes of reference regions that are detected as being similar to the ones from the target would be modified, as well. Therefore, changing the guard sizes of target regions will affect the sizes of reference regions. In the case of activating deep compression, by "-dp", changing the guard sizes of reference regions would affect the sizes of target regions, as well.

The "deep compression" means that similar regions in target and reference sequences are found in three phases instead of two: 1) the model of the reference is built and the target is compressed based upon that model, 2) the model of each detected region is built and the whole reference is compressed based on these models and 3) the model of each detected reference region is built and the corresponding target regions will be compressed based on that model.

Triggering "-nr" makes the tool not to perform the reference-free compression (self-complexity computation).

Smash++ accepts FASTA and FASTQ files as input, in addition to sequences. In these cases, the input files are first converted to sequences and then processed further. It is possible to save these

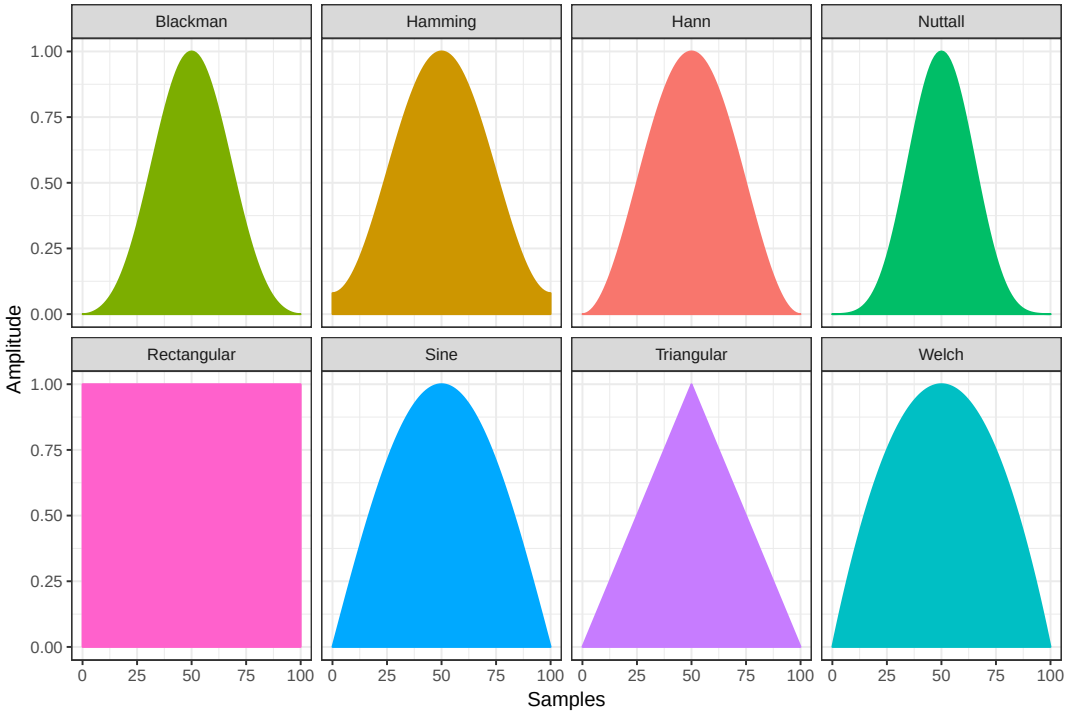

Fig. S7. Various windowing functions implemented and embedded in Smash++.

sequences by "-sb" option. When the information profile is obtained, Smash++ smoothens then removes it by default. However, it can be preserved by "-sp" option. The same thing happens to the smoothed file, i.e., it is segmented then removed. But, the user can use "-sf" to save the filtered file. Also, the segmented files can be saved using "-ss". The user can save all the information profile (content), filtered and segmented files, by triggering "-sa" option.

For the purpose of compression, it is recommended to use "-l" option, since it configures the models automatically. However, using "-rm" and "-tm", the user would be able to manually configure the reference model, for reference-based compression, and the target model, for reference-free compression, respectively. Parameters of the models are described in detail in section S1. Note that using "-ll" option, the list of parameters that would be chosen for each model automatically, will be shown.

Running Smash++ (without visualization), positions of the similar regions in reference and target sequences, and also complexity of the regions is saved in a *.pos file, that can be visualized by

```
./smashpp -viz
```

which gives

```
SYNOPSIS
  ./smashpp -viz [OPTIONS]  -o <SVG-FILE>  <POS-FILE>

OPTIONS
  Required:
  <POS-FILE>          = position file, generated by
                        Smash++ tool (*.pos)

  Optional:
  -o  <SVG-FILE>      = output image name (*.svg)              -> map.svg
  -rn <STRING>        = reference name shown on output. If it
                        has space, use double quotes, e.g.
                        "Seq label". Default: name in header
                        of position file
  -tn <STRING>        = target name shown on output
```

```
  -l  <INT>          = type of the link between maps: [1, 6] -> 1
  -c  <INT>          = color mode: [0, 1]                    -> 0
  -p  <FLOAT>        = opacity: [0.0, 1.0]                   -> 0.9
  -w  <INT>          = width of the sequence: [15, 100]      -> 16
  -s  <INT>          = space between sequences: [15, 200]    -> 62
  -f  <INT>          = multiplication factor for             -> 43
                       color ID: [1, 255]
  -b  <INT>          = beginning of color ID: [0, 255]       -> 0
  -rt <INT>          = reference tick: [1, 4294967295]
  -tt <INT>          = target tick: [1, 4294967295]
  -th [0][1]         = tick human readable: 0=false, 1=true  -> 1
  -m  <INT>          = minimum block size: [1, 4294967295]   -> 1
  -vv                = vertical view                         -> no
  -nn                = do NOT show normalized relative       -> no
                       compression (NRC)
  -nr                = do NOT show self complexity           -> no
  -ni                = do NOT show inverse maps              -> no
  -ng                = do NOT show regular maps              -> no
  -h                 = usage guide
  -v                 = more information
  --version          = show version
```

The output of Smash++ visualizer is an "SVG" file whose name is determined by "-o" option. By default, it is named "map.svg". Names of the reference and the target, which are going to be printed on the output image, can be altered by "-rn" and "-tn", respectively. They are by default the names written in the positions file.

Options "-l", "-c", "-p", "-w", "-s", "-f" and "-b" can be used to change the appearance of the image.

Assigning integers to "-rt" and "-tt" options will change the tick sizes of the reference and the target, respectively. Smash++ prints the sizes on axes in human readable format, e.g., 1K, 2M, etc. However, it can be triggered by "-th" option. Note that, here, 1K is equivalent to 1000 and not 1024, and so on.

By setting "-m" to an integer value, only the regions that are bigger than that value will be illustrated. To have a vertical view of the image, instead of the default horizontal view, one can use "-vv" trigger.

Smash++ performs reference-based and reference-free compressions to calculate the normalized relative compression (NRC) and redundancy (self complexity), respectively. If the user is not interested in showing them, he/she can turn them off by "-nn" and "-nr" triggers. In addition, Smash++ considers by default both regular and reverse complement maps in its calculations. Triggering "-ni" and "-ng" will stop showing inverted and regular maps, respectively.

S2.3 Example

This section guides step-by-step employing Smash++ to find and visualize rearrangements in a sample genomic data. Note that the commands can be run on Linux and macOS, however, they are similar in Windows.

Install Smash++ and provide the required files

First, install Smash++:

```
git clone https://github.com/smortezah/smashpp.git
cd smashpp
./install.sh
```

Then, copy `smashpp` executable file into `example/` directory and go to that directory:

```
cp smashpp example/
cd example/
```

There is a 1000 byte reference sequence, named `refs`, and a 1000 byte target sequence, named `tars`, in this directory. Running

```
./smashpp -r refs -t tars -f 45 -l 3
./smashpp -viz -p 1 -s 50 -w 15 refs.tars.pos
```

results in Fig. S8, which has been saved as “map.svg”.

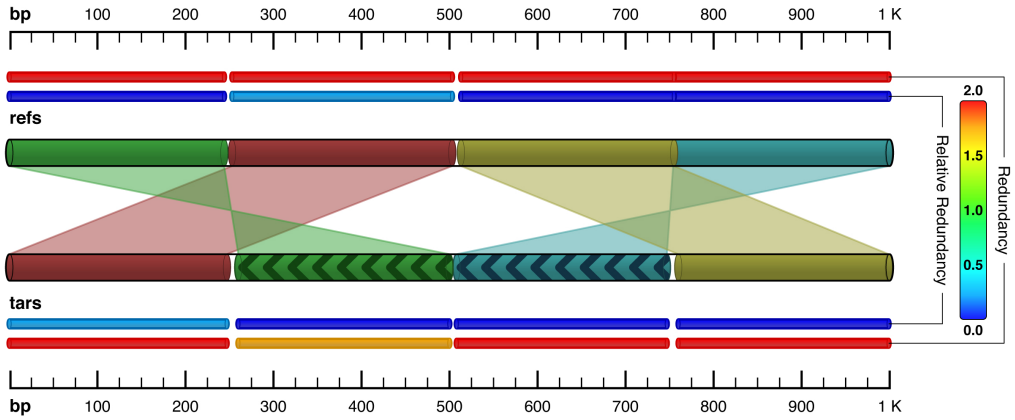

Fig. S8. An example of running Smash++ on two 1000 base sequences. Two similar regions in regular mode and two similar ones in inverted mode are detected.

References

[1] K. Sayood, *Introduction to data compression.* Morgan Kaufmann, 2017.

[2] M. Hosseini, D. Pratas, and A. J. Pinho, "AC: A compression tool for amino acid sequences," *Interdisciplinary Sciences: Computational Life Sciences*, vol. 11, no. 1, pp. 68–76, 2019.

[3] A. J. Pinho and D. Pratas, "MFCompress: a compression tool for FASTA and multi-FASTA data," *Bioinformatics*, vol. 30, no. 1, pp. 117–118, 2013.

[4] D. Pratas, R. M. Silva, A. J. Pinho, and P. J. Ferreira, "An alignment-free method to find and visualise rearrangements between pairs of DNA sequences," *Scientific reports*, vol. 5, p. 10203, 2015.

[5] D. Pratas, M. Hosseini, and A. J. Pinho, "Substitutional tolerant Markov models for relative compression of DNA sequences," in *International Conference on Practical Applications of Computational Biology & Bioinformatics (PACBB)*. Springer, 2017, pp. 265–272.

[6] R. Morris, "Counting large numbers of events in small registers," *Commun. ACM*, vol. 21, no. 10, pp. 840–842, 1978.

[7] G. Pitel and G. Fouquier, "Count-min-log sketch: Approximately counting with approximate counters," in *International Symposium on Web AlGorithms*, Deauville, France, Jun 2015.

[8] R. Blackman and J. Tukey, "Particular pairs of windows," *The measurement of power spectra, from the point of view of communications engineering*, pp. 95–101, 1959.

[9] M. Hosseini, D. Pratas, and A. J. Pinho, "Cryfa: a secure encryption tool for genomic data," *Bioinformatics*, vol. 35, no. 1, pp. 146–148, 2018.

[10] A. Turing, "On computable numbers, with an application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. 42, no. 2, pp. 230–265, 1936.

[11] M. Li and P. Vitányi, *An introduction to Kolmogorov complexity and its applications*, 3rd ed. Springer, 2009.

[12] H. Zenil, F. Soler-Toscano, J.-P. Delahaye, and N. Gauvrit, "Two-dimensional Kolmogorov complexity and an empirical validation of the Coding theorem method by compressibility," *PeerJ Computer Science*, vol. 1, p. e23, Sep. 2015.

[13] R. Antão, A. Mota, and J. A. T. Machado, "Kolmogorov complexity as a data similarity metric: application in mitochondrial DNA," *Nonlinear Dynamics*, vol. 93, no. 3, pp. 1059–1071, Aug 2018.

[14] C. Faloutsos and V. Megalooikonomou, "On data mining, compression, and kolmogorov complexity," *Data Mining and Knowledge Discovery*, vol. 15, no. 1, pp. 3–20, Aug 2007.

[15] M. Hosseini, D. Pratas, and A. J. Pinho. Smash++. [Online]. Available: https://github.com/smortezah/smashpp

[16] D. Pratas. Goose. [Online]. Available: https://github.com/pratas/goose

[17] J. W. Tukey and R. W. Hamming, *Measuring noise color.* Bell Telephone Laboratories, 1949.

[18] A. Nuttall, "Some windows with very good sidelobe behavior," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 29, no. 1, pp. 84–91, 1981.

[19] A. V. Oppenheim, R. W. Schafer, and J. R. Buck, *Discrete-Time Signal Processing.* Upper Saddle River, NJ: Prentice Hall, 1999.

[20] F. J. Harris, "On the use of windows for harmonic analysis with the discrete Fourier transform," *Proceedings of the IEEE*, vol. 66, no. 1, pp. 51–83, 1978.

[21] M. S. Bartlett, "Periodogram analysis and continuous spectra," *Biometrika*, vol. 37, no. 1/2, pp. 1–16, 1950.

[22] P. Welch, "The use of fast Fourier transform for the estimation of power spectra: a method based on time averaging over short, modified periodograms," *IEEE Transactions on audio and electroacoustics*, vol. 15, no. 2, pp. 70–73, 1967.