https://doi.org/10.1093/nar/gkab1209

Fractal construction of constrained code words for DNA storage systems

Hannah F. Löchel, Marius Welzel, Georges Hattab, Anne-Christin Hauschild and Dominik Heider*

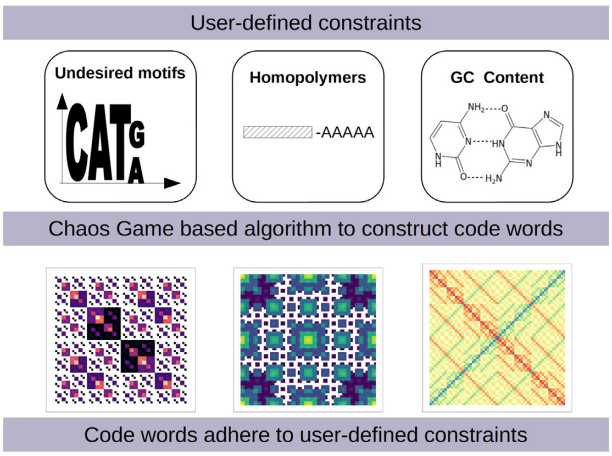
Department of Mathematics and Computer Science, University of Marburg, Germany

Received June 22, 2021; Revised November 16, 2021; Editorial Decision November 17, 2021; Accepted November 24, 2021

ABSTRACT

The use of complex biological molecules to solve computational problems is an emerging field at the interface between biology and computer science. There are two main categories in which biological molecules, especially DNA, are investigated as alternatives to silicon-based computer technologies. One is to use DNA as a storage medium, and the other is to use DNA for computing. Both strategies come with certain constraints. In the current study, we present a novel approach derived from chaos game representation for DNA to generate DNA code words that fulfill user-defined constraints, namely GC content, homopolymers, and undesired motifs, and thus, can be used to build codes for reliable DNA storage systems.

GRAPHICAL ABSTRACT



INTRODUCTION

Due to the increasing speed of global digitization, the amount of digital data produced is growing exponentially (1). Conventional storage mediums such as hard disks have a maximum information density of about 10^3GB/mm^3 (2). However, their life expectancy is rather short, and thus, only magnetic tapes are used for long-term storage. Since these tapes also have a very short life expectancy, they are copied every five years on average to guarantee data safety. An alternative data storage medium exists in nature in the form of deoxyribonucleic acid (DNA), which consists of the four nucleotides adenine (A), thymine (T), guanine (G) and cytosine (C). DNA has an estimated information density of about 4.6×10^8GB/mm^3 and is, under optimal conditions, stable for thousands of years (2). Several groups have developed approaches for DNA data storage (1,3–9) and DNA watermarking (10–15). However, limitations in code word design have only been partly addressed so far.

To store information into DNA, the binary information is encoded into DNA sequences (code words) in the first step. In the next step, these sequences are synthesized and stored. The DNA can be sequenced at any time to retrieve the stored information (16). The development of Next-Generation Sequencing (NGS) technologies allows a quick reading of the information, while the DNA synthesis remains a limitation for DNA data storage, as the synthesis is very time and cost consuming. Thus, DNA storage systems are not competitive for commercial use at the moment. However, it is expected that the costs for synthesis will drop significantly in the near future (17). Nevertheless, DNA storage systems allow easy and low cost copying of media, in contrast to conventional storage systems. A copy of a conventional storage medium efforts the same price as the original medium. For DNA, the first synthesis is relatively expensive, while copies can be achieved at a very low price. The goal of this paper is to describe a novel approach for the construction of codebooks (set of code words), adhering to user-defined constraints, which can be incorporated into any existing error-correction algorithms. To this end, the code word library design can be done separately from the subsequent error-correction codes in some cases, e.g. for the proposed method, which is an advantage regarding computational complexity compared to other codes where unwanted motifs need to be discarded during encoding. In our approach, the code word design

*To whom correspondence should be addressed. Tel: +49 6421 2821579; Email: dominik.heider@uni-marburg.de

is independent of the subsequent error-correcting codes. Thus, we do not address error correction explicitly in our analysis.

DNA synthesis and sequencing methods are error-prone (18), in particular, if the DNA sequence contains specific patterns. Common synthesis approaches first synthesize short fragments (oligos) of the target sequence, which are subsequently assembled. The assembly process requires that the oligos have a similar GC content and, if homopolymers exist, that they are few in number and rather short (19). Sequencing methods are another potential error source (20), with error probabilities increasing for sequences containing long stretches of homopolymers, a strongly deviating GC content, or method-dependent motifs (19).

Thus, due to limitations in the DNA synthesis and sequencing processes, the nucleotide composition of synthetic DNA fragments that can be used for data storage is subject to multiple constraints to reduce errors in, e.g. DNA synthesis and DNA sequencing.

These constraints depend on the chosen methodologies for synthesis, sequencing, and also storage but can be roughly divided into restrictions regarding the GC content of the sequences, homopolymers (hp) and undesired motifs (19).

- **GC content** must either be constant (strongly constrained) or within a certain interval (weakly constrained) (21) to reduce the probability of secondary structure formation and to ensure uniform sequence coverage in the sequencing (22).
- **Homopolymers** are continuous repeats of a certain nucleotide that can lead to increased error rates in sequencing methods (23), as sequencing methods often fail to recognize the correct lengths of homopolymers. Thus, the length of homopolymers should be limited for DNA code words in DNA storage systems.
- **Motifs** are short subsequences in a DNA sequence. Sequencing and synthesis methods depend on short DNA motifs to initiate the amplification steps of the workflows (24,25). Due to these limitations in synthesis, respective motifs have to be excluded (e.g. restriction sites (26)). Moreover, other motifs can introduce errors throughout the sequencing process, for instance, due to secondary structure formation (19).

Due to the fact that DNA-based storage systems are in their early stages of development, further constraints may arise based on the experimental conditions, such as new sets of motifs for novel synthesis or sequencing technologies, as well as for novel concepts such as *in vivo* DNA storages.

To take these limitations and constraints into account, a flexible code word design is required for DNA storage systems. Various deterministic approaches adhering to the homopolymer and GC content constraints exist, for instance in (21,27–30). Other heuristic methods, e.g. in (16,31–34), additionally take into account a large minimal Hamming distance (the number of positions that differ between two strings).

However, one important constraint that is overlooked in the existing literature is that of undesired motifs, which are important for the synthesis, sequencing, and storage of DNA sequences. This is particularly relevant since the amount and composition of these motifs largely depend on the employed DNA manipulation techniques, for instance, primer targets for random access or certain motifs with biological relevance for *in vivo* storage (1,2). Accordingly, solutions for codebook design are required that allow the flexible creation of code words while respecting various motif constraints. This enables the evaluation of different combinations of synthesis, sequencing, and storage techniques.

In our study, we developed a fractal-based method to generate all possible code words that does not only take into account GC content and homopolymer constraints, but also the challenge of excluding user-specified motifs. Our method is based on a modified frequency matrix chaos game representation (FCGR), an extension of the chaos game representation (CGR), which transforms a DNA sequence into a fractal.

The term fractal was introduced by Mandelbrot to describe self-similar geometrical forms (35). Moreover, he described fractals as a set for which the Hausdorff dimension exceeds the topological space. For a self-similar geometrical form, the structural patterns of the form can be found in small sections of the form in a repetitive manner. In ideal fractals (in a mathematical sense), this self-similarity is infinite (36).

The chaos game representation (CGR) was first described by Barnsley (37). It is an iterative function system to construct fractals. The underlying idea of the algorithm is to assign numbers from one to three to the edges of a triangle. The algorithm starts from a randomly chosen vertex. Then the next vertex is drawn by randomly choosing a number from one to three, representing an edge of the triangle. Half of the distance to the direction (so-called scaling factor) to this edge of the triangle is drawn to set the new vertex. After several iteration steps, the Sierpinski triangle appears. Abbreviations of the algorithm, such as a change of angle, the scaling factor, or the number of edges, result in differently shaped fractals (37). The resulting patterns are called chaos game representation (CGR). Jeffrey (38) was the first who used the CGR algorithm for DNA, where the four nucleotides are assigned to the edges of a square (see Figure 1). There are several applications based on CGR (39), e.g. the analysis and comparison of whole-genome sequences (40) or phylogenetic predictions (41). Moreover, the CGR has some interesting properties, as pointed out by Almeida et al. (42). The CGR patterns are unique, the sequence can be reconstructed by the coordinates, and the distributions of the points can be described as a generalized Markov Chain model.

The frequency matrix chaos game representation (FCGR) is an extension (42) of the CGR, in which the CGR is represented as a count matrix, where the number of dots of the CGR in each section of a grid is counted. The FCGR is particularly useful for machine learning approaches as it provides a fixed input dimension for any sequence length and has been successfully applied already in protein classification (43). An FCGR with the order of 2^n represents a matrix of k-mers. Based on that, we can consider it as a representation of all possible DNA words of the length n (see Figure 2). We will refer to this matrix representation in the order of 2^n as matrix chaos

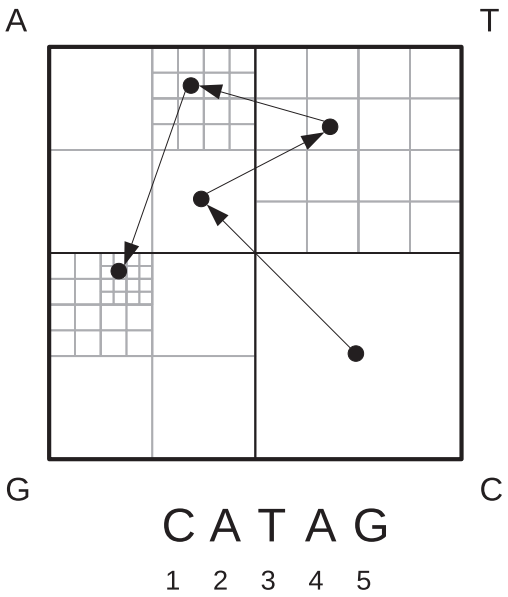


Figure 1. Chaos game representation. Example mapping for creating a CGR based on a given input sequence: *CATAG*.

game representation (mCGR) in the following. As shown in Figure 2, the number of entries in the matrix increases by 2^n with increasing order. The coloring of the nucleotide A shows an emerging fractal pattern. The white elements in the matrix build the Sierpinski triangle. The different gray values represent the amount of A in a sequence with a self-similar pattern. The simple case of a word length $n = 1$ is a representation of the four nucleotides $\begin{matrix} A & T \\ G & C \end{matrix}$. The increasing word length can be represented as Kronecker power (Kronecker Product of the matrix by itself) with the following definition:

$$M^n = M \otimes M \otimes M... \quad (1)$$

In CGR (or mCGR, respectively), the letter is appended to calculate the strings in CGR (or mCGR). For prepending the characters to the string, the representation is also known as genomatrices (44–46). Both notations, CGR/mCGR as well as genomatrices, have been used to demonstrate that fractal patterns emerge from motifs (47,48). Since the positional information of the matrix elements can be used to reconstruct the underlying sequence, the matrix elements can carry additional information, e.g. whether the use of the corresponding sequence is restricted. Anitas (48) assigned probabilities to the squares and applied the Kronecker powers to perform structural analyses of DNA by CGR. While the Kronecker power results in a multiplication of the probabilities, we propose an addition of the substrings instead of a multiplication of probabilities.

Based on this initial idea, we developed and optimized a model to generate a set of all possible code words of a given length which adhere to various constraints, such as limitation of homopolymers, undesired motifs, and variable GC content constraints (both weakly, i.e. in an interval, and strongly, i.e. as a fixed percentage), and thus can be used and integrated into codes for DNA storage systems. To this end, we compared our approach with existing methods for code word design, namely (16,27–31,33,34). Moreover, we integrated our codebooks into a lexicographic encoding (which maps a binary sequence to DNA (29)) as a proof-of-concept, and compared its performance with current state-of-the-art algorithms, namely DNA Fountain codes (3). However, our CGR-based codebook design can be used as a basis for any DNA storage codes, e.g. error-correction codes. We provide a Java implementation and R scripts available at http://mCGR.heiderlab.de and source code at https://github.com/HFLoechel/ConstrainedKaos.

The Hamming distance, which is particularly important for error-correction and DNA-computing, a field that is closely related to DNA storage systems, can also be easily calculated with our model.

MATERIALS AND METHODS

Our approach is based on CGR, and we used the concept of CGR to implement a mathematical model for designing code words that fulfill certain constraints that are important for DNA storage systems, namely homopolymers, undesired motifs, and GC content, as well as additional constraints that are necessary for DNA computing, namely the Hamming distance of code words.

Homopolymers and undesired motifs

Homopolymers are defined as continuous repeats of a certain nucleotide and may lead to increased error rates in sequencing methods (23). Thus, it is important that homopolymers are excluded from code words for DNA data storage systems.

In each quarter of an mCGR, every sequence ends with the same letter, for any order of the matrix. Moreover, mCGRs as fractals are self-repetitive. Consequently, a repetition of a particular pattern for any given subsequence exists. An example demonstrating the construction of an mCGR for the homopolymer *CC* is shown in Figure 3. This approach cannot only be used for the exclusion of homopolymers but also for any undesired motif.

Thus, for any given sequence constraint, we first create an mCGR of the last coordinate in the order of the constrained length, where we denote the element of the excluded sequence with 1 and the remaining elements with 0. This initial matrix serves as an input or generator matrix for the next iteration step. The generator matrix is first used for tiling over a matrix for the next word length. In the second step, the generator matrix is stretched to the order for the next word length and added to the matrix of the first step. This results in the next generator matrix for the next iteration step. This procedure will be repeated until the desired sequence length is reached. For more than one constraint, this procedure can be repeated and the resulting matrices can be added. In Figure 3, an example for homopolymer *CC* is shown, wherein step 1 the last matrix element is denoted with one. A homopolymer with the same length can be added by matrix addition. For instance, for all homopolymers with a sequence length of two, the corre-

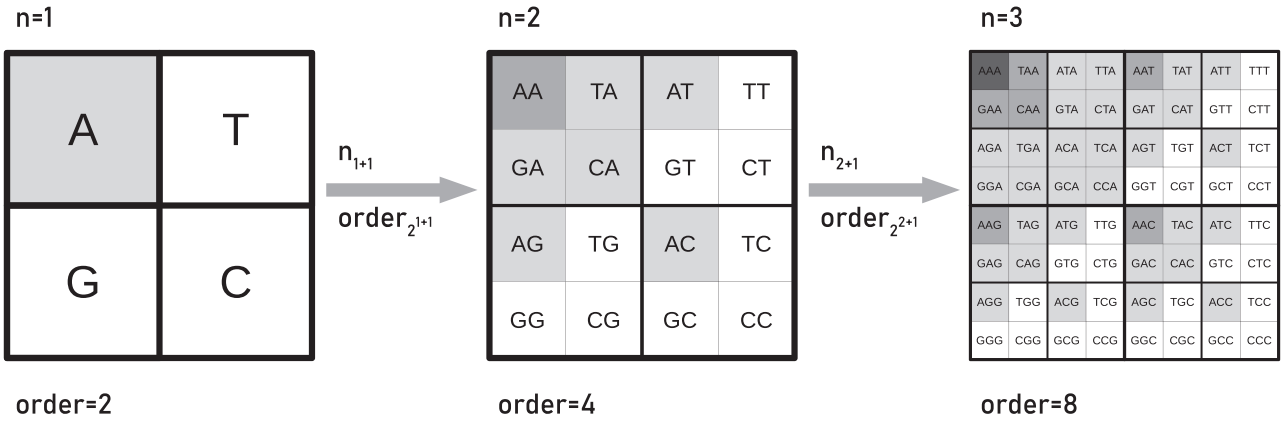


Figure 2. Chaos game representation matrices for increasing word length (n). With the extension of CGR to FCGR in a matrix order of 2^n (mCGR), each element in the matrix represents a word/string of the given length. The increasing order can be achieved by Kronecker powers. The single characters of the strings have a fractal order. The example is shown for the fractal pattern of the count of the nucleotide *A*.

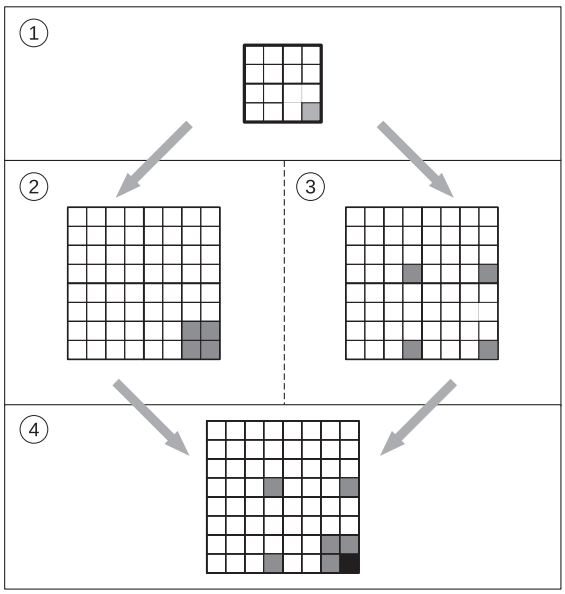


Figure 3. Algorithm for constrained subsequences. (1) mCGR with the constraining motif (here *CC*) is initialized. The matrix size is given by the length of the constraint. (2) The mCGR is scaled to double the size and (3) used for tiling. (4) The result is added and can be used for the next iteration until the order of the matrix for the desired code word length is reached.

sponding generator matrix is $\begin{smallmatrix} 1&0&0&1\\ 0&0&0&0\\ 0&0&0&0\\ 1&0&0&1 \end{smallmatrix}$. For the addition of a constraint with a length of 3, the mCGR of an order 2^3 has to be calculated, and both matrices have to be combined by matrix addition.

Thus, matrices with increasing orders of 2^n are representations of all sequences of a given word length *n*. But instead of using a matrix of characters, we use a representation of ones and zeros for the generator matrix $mCGR^n$ (where *n* indicates the length of the initial word).

In contrast to Equation (13), where we append a string with a single character, we have to take the complete mCGR of the initial iteration step into account (see Figure 3). Thus, we have to exchange the appending part of the equation (Equation (12)). We can achieve this by mirroring the prepending part (Equation (11)):

$$mCGR^n = \mathbb{1}^{2^1} \otimes mCGR^{n-1} + mCGR^{n-1} \otimes \mathbb{1}^{2^1} \qquad (2)$$

where $\mathbb{1}^n$ represents the square matrix of ones with an order of *n*. The matrix elements that are not affected by the constraints remain zero, and the remaining elements are counting the numbers of the events where constraints appear.

GC content

First, we consider the simple case for 50 % GC content. Depending on the order of the nucleotides at the edges of the mCGR, two patterns can evolve, as shown in Figure 4A. A GC content of 50% is shown in gray, for a word length of $n = 2$ and $n = 4$. If *A* and *T* are the opposite of each other, the resulting patterns are bars. While if they are positioned in diagonal directions, we get higher-order, more complex, flower-like patterns. The emerging patterns can be explained by the most simple case, namely the bars. We can reduce the mCGR to the first row and assign $A = T = 0$ and $G = C = 1$. We then convert the characters of a string to their binary representations. By using this simple encoding, new patterns of alternating ones and zeros emerge. For the first character in the sequence, we get an alternating pattern of zeroes and ones, while for the second character, we end up with alternating two zeros and two ones. Similar patterns occur for the next characters (see Figure 4 B. To retrieve the GC content, we calculate the sum of a column and divide it by the sequence length. By using this simple algorithm, we cannot only retrieve sequences with exactly 50% GC content but also defined intervals of interest. For the calculation of these patterns, we use the generator matrix with $A = T = 0$ and $C = G = 1$ to $\begin{smallmatrix} 0&0\\ 1&1 \end{smallmatrix}$. However, we can also make use of our mathematical model in Equation (8c), which is not used for a complete motif but for the occurrence of a single char-

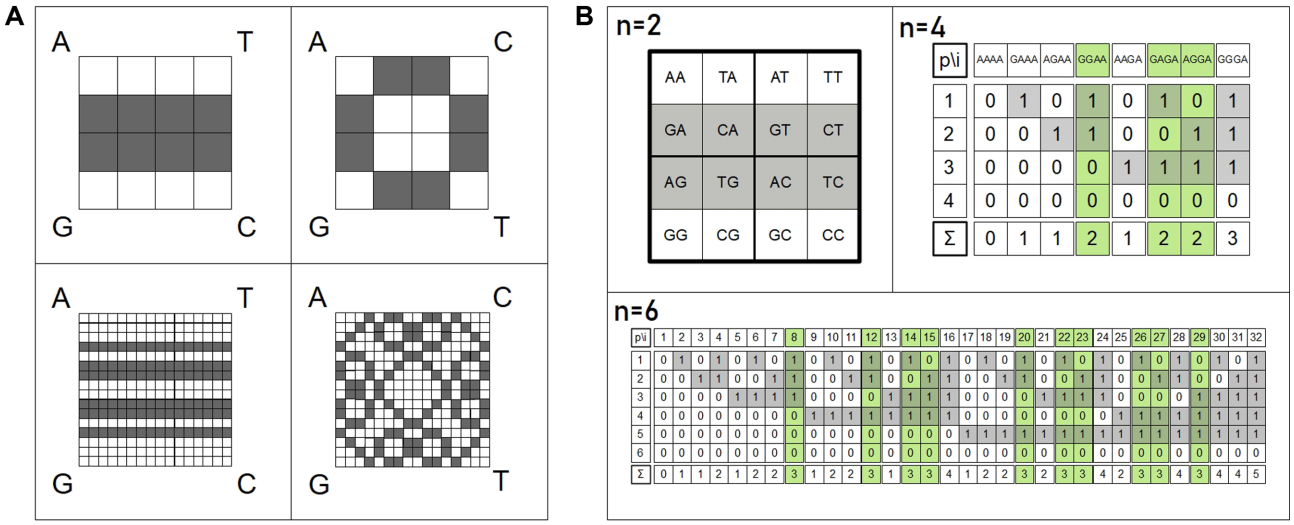


Figure 4. GC content. (**A**) Patterns for 50 % GC content for the word length $n = 2$ (top) and $n = 4$ (bottom). The resulting matrices vary, depending on the order of nucleotides at the edges. (**B**) Calculation of the GC content, shown for word length $n = 2$, $n = 4$, and $n = 6$. For $A = T = 0$, $G = C = 1$, and an order of $\begin{smallmatrix} A & T \\ G & C \end{smallmatrix}$, bars emerge by coloring 50% GC content. The sum of the characters in a string can be used to calculate the amount of $A = T$ and $G = C$, respectively. Therefore, division by the sequence length can serve as an indicator for the GC content. $n = 2$: mCGR with 50 % GC content in gray. For $n = 4$ and $n = 6$, the GC content of 50% is shown in green.

acter. For increasing word length, we have to double the size of the original matrix and either append or prepend the generator matrix (a single character respectively), as shown in Equation (3).

$$\begin{aligned} D^n &= \mathbb{1}^{2^{n-1}} \otimes D^1 + D^{n-1} \otimes \mathbb{1}^{2^1} \\ &= \mathbb{1}^{2^1} \otimes D^{n-1} + D^1 \otimes \mathbb{1}^{2^{n-1}} \\ with\ D^1 &= \begin{cases} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} for \begin{bmatrix} A & C \\ G & T \end{bmatrix} \\ \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} for \begin{bmatrix} A & T \\ G & C \end{bmatrix} \end{cases} \end{aligned} \tag{3}$$

Hamming distance

While the aforementioned constraints are very important for DNA storage systems, another constraint is of the utmost importance when we consider the code words for DNA computing, namely the Hamming distance. The Hamming distance between two sequences of equal length is the number of positions at which the corresponding symbols are different (49). In order to use code words for DNA computing or error correction, the Hamming distance of code words should be maximized. Using mCGR, it is also possible to integrate the Hamming distance as a constraint. To this end, we again make use of the initial matrix with the four nucleotides A, T, G, C. To calculate the Hamming distance for a single character, e.g. A, to all other nucleotides, we can set $A = 0$ and $G = T = C = 1$. To get the Hamming distance for a complete sequence, we have to append the corresponding Hamming distance for each nucleotide on each position (see equation (4)). Based on this, we can create a matrix with the Hamming distance of one single code word to all other code words.

$$H(s)^n = \begin{cases} H(s)^1 = B^1 & n = 1 \\ \mathbb{1}^{2^1} \otimes H(s)^{n-1} + B^1 \otimes \mathbb{1}^{2^{n-1}} & \text{otherwise} \end{cases}$$

$$with\ B^1 = \begin{cases} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} for \quad A \\ \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} for \quad T \\ \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} for \quad G \\ \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} for \quad C \end{cases} . \tag{4}$$

To generate the complete table of the Hamming distance for all code words, we can apply Equation 3) for the GC content. To calculate the GC content, we used a binary representation of the nucleotides, where the GC content correspondents to the number of characters being A or T to the number of characters being G or C, which is equal to the Hamming distance for binary code words. This property enables the exchange of the generator matrix with a generator matrix D_{2^1} for the Hamming distance of all nucleotides:

$$with\ D^1 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} for\ rows = columns \tag{5}$$

Implementation

We implemented all algorithms in R and run-time optimized versions for GC content and homopolymer/motif constraints in Java v. 12.0.1.

Since storing complete matrices would require a large amount of storage space, we decided to exclusively store the positions of the matrix elements affected by the constraints. It is, therefore, a lightweight form of a sparse matrix, where we do not assign values for the matrix elements. The set containing constrained words can be used to deduce the set of allowed words. We split equation (2) into two algorithms for tiling and doubling of the matrix. For the GC content, we implemented the approach shown in Figure 4B. The pseudocode for the algorithms can be found in Supplementary Material S9.

Our implementation allows any combinations of constraints for GC content (as a fixed number or interval), length of homopolymers, and undesired motifs (the latter provided in a FASTA formatted file). We also implemented a back-calculation algorithm for mCGR from elements in the matrix to sequences so that the allowed code words can be selected and stored as a FASTA file.

It is also possible to plot the results using the library JFreeChart v. 1.0.19 with JCommon v. 1.0.23. We implemented the CGR algorithm in Java, as a union square (height and width from −1 to 1). The coordinates of the CGR are stored as big integer fractions to increase the precision, and to avoid floating-point errors, by using the library Apache Commons Math v. 3.6.1.

We further implemented an option to generate codebooks that exclusively contain code words that can be concatenated without building forbidden motifs or homopolymers. Based on the approach proposed by Wang *et al.* (29), we excluded code words, which can build an undesired motif or homopolymer. To do this, we identified the words ending with the start of a motif or homopolymer matching at least half the length of the motif/homopolymer and the sequences beginning with the end of motif/homopolymer at least half the length. For instance, for the homopolymer AAA we removed all words ending with AA and starting with AA. Therefore, the matrix structure of the mCGR allowed us a quick identification of the words that should be excluded. Finally, the generated codebook can be used as a basis for a code. As an example, we implemented a lexicographic encoding/decoding based on our codebook, as proposed by Wang *et al.* (29) in python 3.

To encode an arbitrary input file using the lexicographic approach, the codebook is first sorted lexicographically, followed by partitioning the bits of the input file in blocks according to the information rate of the codebook. The decimal representation of a block is used as the index of the codebook for the mapping of binary sequences to code words. The decoding uses the same principle with a hashmap of code words as keys and the indices of the lexicographically sorted codebook as values.

Comparison

First, we determined the coding rates of the codebooks generated by mCGR and lexicographical mCGR, based on the following equation (with C number of code words and n length of code words):

$$coderate = \frac{log_2(|C|)}{n} \tag{6}$$

We used different constraints and two sets of motifs, named scenario 1 and 2. While the first set contains 10 motifs with a length of 6, in scenario 2, we considered the motifs listed in MESA (19) with a maximum sequence length of 10 (overall 35 motifs).

To test and benchmark the mCGR-lexicographic code, we produced a benchmark dataset consisting of seven images in jpg format with different file sizes ranging from 300kB to 1.7MB. We used mCGR images generated with our R implementation to create the dataset (both the R script and the dataset are available under github.com/HFLoechel/ConstrainedKaos).

Based on this benchmark dataset, we encoded/decoded the benchmark dataset ten times and measured the runtime with the python package timit on a laptop (AMD Ryzen 5 3500 U, 16GB RAM) on a single CPU process. For comparison, we used the same benchmark dataset for encoding/decoding with the DNA Fountain code. For the mCGR approach, we first generated the codebooks in a word length of 10. Then we en- and decoded the seven benchmark files in different sizes, with the lexicographic code. For the DNA Fountain encoding/decoding, we first determined the minimum amount of packages needed for decoding for each file and applied the same restrictions without considering motifs. We measured the runtime 10 times for both codes. We disabled the Reed-Solomon for the DNA Fountain to make a fair comparison. For DNA Fountain, we noticed that with the minimal number of packages, the decoding did not work, and more packages are needed. To address this, we determined the minimal package number for decoding before the actual benchmark. Moreover, we used the encoded files and calculated the coding rates for each benchmark file, by dividing the number of nucleotides in each encoded file by the file size in bits. To make a fair comparison, we removed the header in the DNA Fountain codes. While DNA Fountain codes can not adhere to motif constraints, we analyzed the number of sequences in each encoded file affected by our motif selection in the two different scenarios.

RESULTS

With our approach, it is possible to automatically generate all possible code words in a given length that take into account the GC content, the homopolymers, and the undesired DNA motifs.

Mathematical model

In the following section, we summarize the mathematical background and equations that are used in our model.

For the alphabet $\Sigma = \{A,G,T,C\}$, the possible words of a length of n are the four possible nucleotides. For our mCGR approach, we have to split the concatenated string into substrings. For the simple case of all possible words in the length of two Σ^2 and the initial mCGR for Σ^1 $\begin{matrix} A & T \\ G & C \end{matrix}$, we can get the first letter of all possible two-letter strings with the Kronecker product with the square matrix of ones $\begin{matrix} 1 & 1 \\ 1 & 1 \end{matrix} = \mathbb{1}^2$, which can be considered as an extension with any

possible character:

$$P^2 = \mathbb{1}^{2^1} \otimes mCGR^1 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \otimes \begin{bmatrix} A & T \\ G & C \end{bmatrix} = \begin{bmatrix} A & T & A & T \\ G & C & G & C \\ A & T & A & T \\ G & C & G & C \end{bmatrix} \quad (7)$$

We can proceed with the end of the strings in equal manner by exchanging both matrices:

$$A^2 = mCGR^1 \otimes \mathbb{1}^{2^1} = \begin{bmatrix} A & T \\ G & C \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} A & A & T & T \\ A & A & T & T \\ G & G & C & C \\ G & G & C & C \end{bmatrix} \quad (8)$$

Finally, the addition of both matrices, generates the complete matrix $mCGR^2$ for Σ^2.

$$mCGR^2 = P^2 + A^2 = \begin{bmatrix} AA & TA & AT & TT \\ GA & CA & GT & CT \\ AG & TG & AC & TC \\ GG & CG & GC & CC \end{bmatrix} \quad (9)$$

With this result, we can proceed to calculate the next mCGR:

$$P^3 = \mathbb{1}^{2^1} \otimes mCGR^2$$
$$A^3 = mCGR^1 \otimes \mathbb{1}^{2^3}$$
$$mCGR^3 = P^3 + A^3 \quad (10)$$

We can now formulate the general equation to describe an mCGR by appending a single character as:

$$P^{n+1} = \mathbb{1}^{2^1} \otimes mCGR^n \quad (11)$$

$$A^{n+1} = mCGR^1 \otimes \mathbb{1}^{2^n} \quad (12)$$

$$mCGR^{n+1} = A^{n+1} + P^{n+1} \quad (13)$$

While this approach allows the construction of Σ^n by appending a single letter, the model has to be extended to adhere to the different constraints. To this end, the mCGR model is defined as follows:

The mCGR model is defined as

$$mCGR^n = P^n + A^n \quad (14)$$

Prepending in the mCGR model is defined as

$$P^n = \mathbb{1}^{2^1} \otimes mCGR^{n-1} \quad (15)$$

while appending is defined as

$$A^n = \begin{cases} mCGR^1 \otimes \mathbb{1}^{2^{n-1}} & \text{for Hamming and GC} \\ mCGR^{n-1} \otimes \mathbb{1}^{2^1} & \text{for hp and motifs} \end{cases} \quad (16)$$

The $mCGR^1$ for the Hamming distance is defined as

$$mCGR^1 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix} \quad (17)$$

For the $mCGR^1$ notation $\begin{bmatrix} A & T \\ G & C \end{bmatrix}$ applies:

$$mCGR^1 = \begin{cases} \begin{bmatrix} 0 & 0 \\ 1 & 1 \end{bmatrix} \text{GC} \\ B^1(i) \text{ Hamming (one sequence)} \end{cases} \quad (18)$$

The Hamming distance for the nucleotide at position i is defined as

$$B^1(i) = \begin{cases} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} for\ A \\ \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} for\ T \\ \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} for\ G \\ \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} for\ C \end{cases} . \quad (19)$$

For motifs or homopolymers for $n = i$ and $i =$ length of motif/homopolymer

$$mCGR^i \text{ with elements } e_{r,q} \quad (20)$$

is $e_{r,q} = 1$ for constrained substring, otherwise 0.

Implementation

We implemented all equations in R and Java. Figure 5A–C shows the mCGR of code words implementing different homopolymer constraints, with the usable sequences colored in black. Different fractal patterns emerge from the homopolymer constraints, which is caused by the fractal arrangement of the elements in the matrix (as described in Figure 2). A 50% GC content emerges as stripes in the representation, which is caused by the arrangement of the nucleotides.

Furthermore, we tested our implementation for several constraints concerning undesired motifs (see Supplementary Material S1 and S2). If a single nucleotide, for example, *A*, is forbidden, the Sierpinski triangle emerges. For other constraints, other fractals emerge, e.g. the T-square (a description of this fractal can be found in (50)).

Figure 5D and E shows the resulting representation for the GC content and Hamming distance for a word length of three.

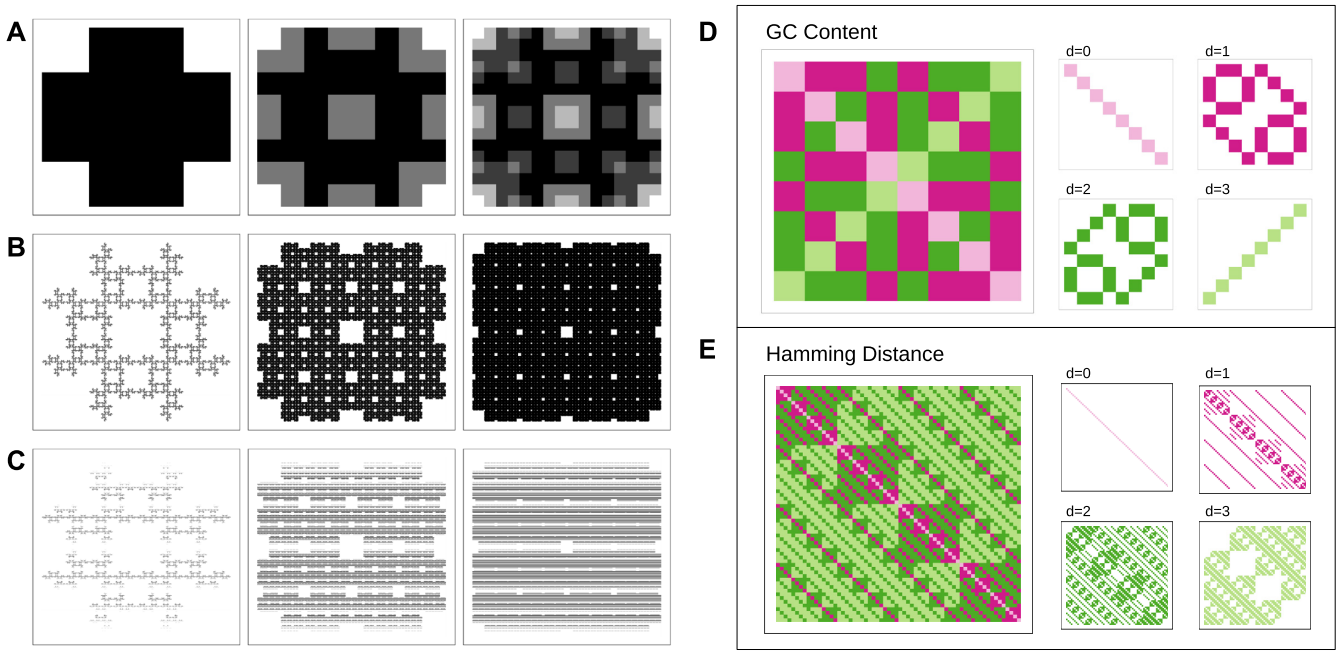


Figure 5. mCGR for different constraints: (A–C) mCGR of code words with different constraints. (**A**) Increasing word length for $hp \geq 2$, 2 to 4 from left to right. A black area represents zero and a white area the highest number in the matrix. Thus, the code words with no homopolymers $hp \geq 2$ are colored in black. (**B**) and (**C**) mCGR of $length = 10$, from left to right: constraints for $hp \geq 2$, $hp \geq 3$, and $hp \geq 4$. White areas contain homopolymers. (**C**) Code words that does not fullfil 50 % GC content are excluded. (D and E) mCGR for the distance d with $length = 3$. (**D**) mCGR of GC content. (**E**) mCGR of Hamming distance. Left: complete representation. Right: different distances are shown.

A detailed view on the GC constraint as mCGR for different lengths and order of edges is given in Supplementary Material S3. The order of the edges has a high impact on the resulting pattern. To calculate the Hamming distance, we can apply the same equation for the GC content with a different generator matrix.

For the GC content, we found a similar pattern as for the Hamming distance of a binary code, as shown in (51). More precisely, when considering the following redefinition of $A = T$ and $G = C$, one can notice that it is a transformation of the Hamming distance in a binary system. Hence, extracting all ones from the mCGR with diagonal order of A and T will result in the adjacency matrix for a hypercube. For other word lengths for the Hamming Distance see Supplementary Material S4. The results for the Hamming distance for a single word can be found in Supplementary Material S5.

To evaluate our approach, we generated a codebook containing 10 nucleotides (nt) long sequences without homopolymers longer than 2 nt, a GC content between 40–60 %, and a set of undesired motifs with relevance to sequence synthesis and sequencing. Due to the recursive definition of our model, a code word length of 10 nt covers all possible words shorter or equal to 10 nt as well as any longer code words. The resulting codebook contained 484 263 DNA sequences. We used the sequence evaluation tool MESA (19) to test whether the code words in the codebook fulfill the desired requirements. MESA is a web application for the assessment of synthetic DNA fragments with respect to homopolymers, GC content, k-mer repetitions, and undesired motifs. Using the MESA API, we evaluated the generated sequences concerning the fulfillment of the aforementioned constraints. MESA could not detect any unfulfilled constraints in our codebook. The input constraints and commands for the Java application as well as the MESA configuration are included in the GitHub repository. The MESA results confirmed that all sequences adhered to the restrictions.

Comparison

In Supplementary Material S6, we compare our approach with other state-of-the-art methods for code word designs. In contrast to other approaches, our method is highly flexible in the number and type of constraints. This allows creating code words that adhere to a multitude of different constraints simultaneously. Furthermore, it is the only method so far that can generate code words adhering to undesired motifs. The GC content can be chosen as an interval or for a fixed percentage.

The Java implementation can be applied either to construct a codebook comprising all possible sequences in a given length under user-defined constraints, or all possible sequences which can freely be concatenated without forming undesired motifs or homopolymers, in order to use them for lexicographic encoding.

Our approach allows the generation of codebooks with user-defined constraints. Therefore, we generated codebooks and calculated the corresponding code rates (Figure 6). For those, we varied the code word length, homopolymer length, and different GC content constraints with and without motif constraints. Additionally, we pre-

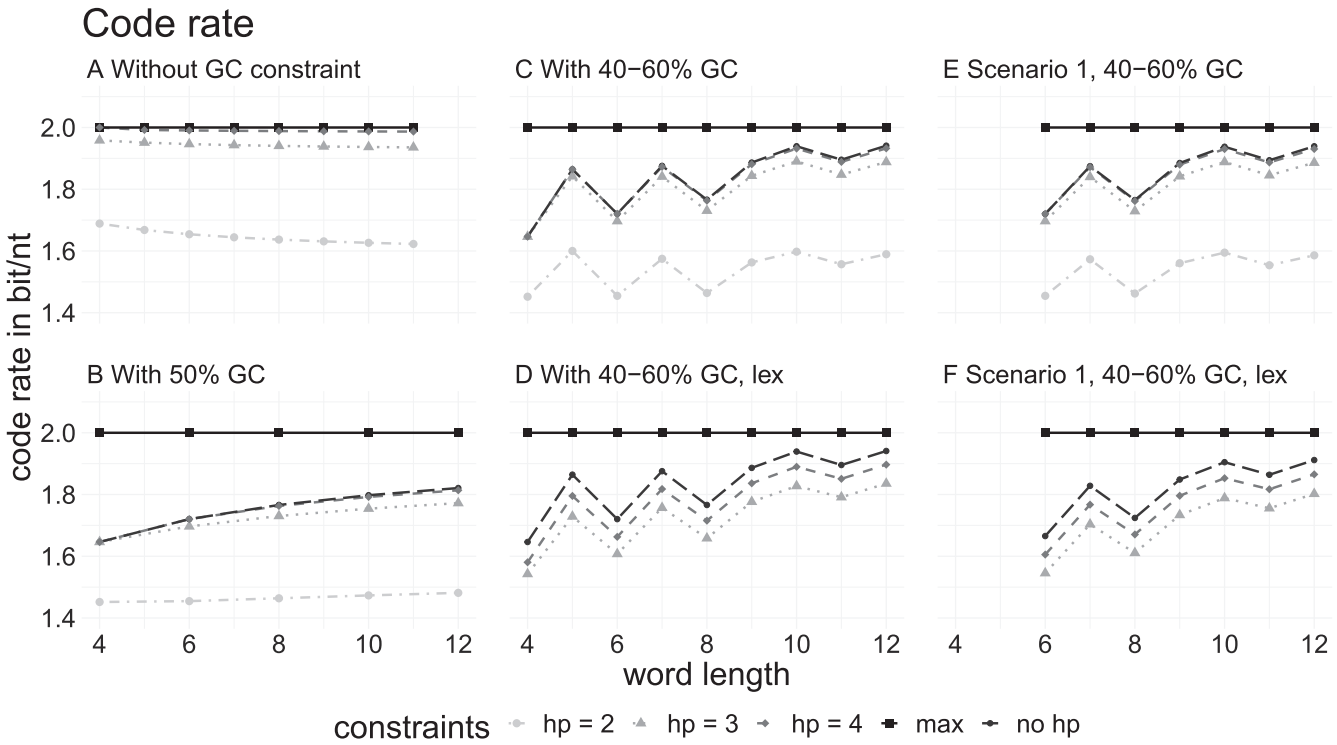


Figure 6. Code rate of mCGR with different constraints for different code word lengths.

pared codebooks for a lexicographic encoding (lex). These codebooks only contain code words that can freely be concatenated while still adhering to the constraints. In scenario 1 (Figure 6E and F), we added 10 undesired motifs with a length of 6. For choosing a GC interval of 40–60%, in different words (Figure 6C–F), a variation between odd and even numbers becomes visible, which is a result of the possible combinations to fit the interval. For example, code words of a length of 6 nucleotides, which have to adhere to an interval of 40–60% GC content, are always at exactly 50% GC to fit the criterion. Three nucleotides are either G or C, and the three remaining are either A or T. In contrast, an uneven code word length, e.g. 7, leaves more options. In this case, three or four nucleotides can be G or C and the remaining A or T to fit in the interval and vice versa. For a 10 nt code word, there are even more options to fit the criterion of a 40–60% GC content. Namely, there are three options: 4–6, 5–5 and 6–4. With an increase of the word length from 10 nt to 12 nt, the increase of the code rate is very small. Thus, regarding the runtime and memory, a code word length of 10 nt is reasonable for preparing a codebook based on the mCGR. The code rates for lexicographic encoding (lex) are shown in Figure 6D and F, which decreases as a result of a smaller codebook. In a second scenario (scenario 2), we created a lexicographic codebook with a set of 35 undesired motifs of a maximum sequence length of 10 nt (hp $\geq$ 4, GC 40–60%, a code word length of 10), leading to a code rate of 1.84. We used the three lexicographic codebooks (all with hp $\geq$ 4, GC 40–60%, code word length of 10), one with no motif constraint, one with the set of motifs of scenario 1, and one with the set of scenario 2, for encoding/decoding a benchmark dataset. The dataset consists of seven image files in different file sizes and compared them with the DNA Fountain code. In Supplementary Material S7, the results for the runtime benchmarks are presented. After preprocessing the codebooks, the lexicographic code has a linear runtime between a few seconds for encoding/decoding. The absolute encoding time is lower than for decoding. The encoding/decoding for DNA Fountain for our benchmark datasets was between minutes and it could be shown that the decoding has a non-linear runtime.

Additionally, we calculated the actual coding rate for each encoded file. For the DNA Fountain algorithm, we removed the 16 nt long headers for each package in advance to make the results comparable to the lexicographic encoding. The maximum code rate for DNA is 2 bits/nt while the lexicographic encoding has a constant code rate of 1.8 bits/nt in all three cases. Wang *et al.* (29) reached a code rate of 1.9 bits/nt with the same constraints. They dismissed all sequences ending with homopolymers of 3 nt and applied a concatenation scheme, avoiding the concatenation of sequences that are building homopolymers. Since we dismissed all code words that could form a motif or homopolymer, we can achieve a code rate of 1.88 for the same setting, while the implementation of our encoding/decoding decreased the actual code rate for our benchmark dataset to 1.8. The DNA Fountain code rate varies for the files with no visible pattern between 1.65 and 1.75 bits/nt. For the DNA Fountain encoded files, we counted the number of sequences affected with motifs for the two scenarios. For scenario 1, about 29% of the packages contained motifs in scenario 2, about 30%, independent of the file size. The re-

lation of affected sequences is constant over the file size but varies with respect to the chosen scenarios. Detailed plots for the code rate comparison and the number of packages including motifs can be found in Supplementary Material S8.

DISCUSSION

In the current study, we present a novel approach for generating codebooks that fulfill certain constraints that are important for DNA storage systems but have not been fully addressed by other approaches. These codebooks can be used as a basis for any DNA storage codes, e.g. error-correcting codes. The constraints include GC content (variable, strong and weak) as well as undesired motifs, which are particularly important for DNA syntheses and DNA sequencing. Our model calculates all possible code words in a given length, excluding those code words that do not adhere to the given constraints. Thus, for a given set of constraints, the maximal information density can be easily calculated.

The removal of undesired patterns is very complex. One can construct all possible code words in advance while discarding those with undesired motifs, homopolymers, or GC content. The remaining code words can further be used for encoding, as we did in our approach. In contrast, Fountain codes search for packages fulfilling all constraints during the encoding process. This step is very time-consuming as every code block needs to be checked against a number of motifs.

Our novel approach further offers the opportunity to calculate the Hamming distance, which is of high relevance for DNA computing (52). It is also possible to apply our approach for the Hamming distance for larger alphabets.

As a proof-of-concept, we used lexicographic encoding (29) to map binary strings to DNA code words. The matrix structure allows quick identification of sequences with a specific prefix or suffix. This enables a dynamic exclusion of matrix areas, depending on the last nucleotide(s) of the previously mapped code word. The mCGR-lexicographic code outperformed the DNA Fountain code, with respect to runtime and code rate. In our approach, we did not incorporate any error-correction due to the fact that the mCGR approach can be combined with any error-correction algorithm as mentioned before. Thus, we also disabled the error-correction in DNA Fountain to make a fair comparison. While Fountain-codes should be able to reach the theoretical maximum of 2 bits/nt, it turned out to be impossible to decode the data under this assumption. Additional packages for decoding need to be added to address this, which affected the code rate of the DNA Fountain code. In our lexicographical encoding, we discarded all sequences that can potentially form motifs/homopolymers, which decreased the code rate compared to the lexicographic approach of Wang *et al.* (29). Moreover, in two possible scenarios, on average, 30 % of the packages in the Fountain codes contain undesired motifs, which could lead to problems in synthesis or sequencing in the DNA storage systems.

To the best of our knowledge, this is the first algorithm that constructs code words that not only adhere to the commonly described constraints in the literature, but also to arbitrary undesired motifs, which play an important role in *in vitro* and *in vivo* studies of DNA codes and DNA storage systems, in particular for DNA synthesis and sequencing. As the structure of the matrix also allows the quick identification of reverse complementary code words, it can further be used in the design of DNA fragments for microarrays. The current JAVA implementation has some limitations regarding the code word and motif length. Without increasing the maximum allowed memory usage of JAVA (4GB for a system with 16GB main memory), a maximal code word length of around 12 nt can be calculated.

The mCGR method offers a new opportunity to construct code words and new perspectives regarding DNA codes. Besides the construction of specific code words, it can also be applied to visualize DNA-based encodings. The mCGR is a matrix representation of all possible code words for a given length Σ^n for DNA. Fractal patterns emerge from different constraints that can be mathematically described, and the back-calculation can be used to retrieve the code words.

DATA AVAILABILITY

The implementation (for Java and R) is available at: http://mCGR.heiderlab.de. The source code is available at: https://github.com/HFLoechel/ConstrainedKaos.

SUPPLEMENTARY DATA

Supplementary Data are available at NAR Online.

ACKNOWLEDGEMENTS

Author contributions: H.F.L. conceived the initial idea, with contributions from M.W. H.F.L. developed and implemented the equations and algorithms, carried out the comparison and analysis, and wrote the initial draft. M.W. contributed to the GC content implementation, implemented the lexicographic encoding/decoding, carried out the validations with MESA, and the literature comparison, and contributed to the writing of the initial draft. G.H. contributed to the figure design. G.H., A.C.H. and D.H. supervised the work, discussed the results and contributed to the writing of the final manuscript.

FUNDING

LOEWE program of the State of Hesse (Germany) in the MOSLA research cluster. Funding for open access charge: LOEWE MOSLA.
Conflict of interest statement. None declared.

REFERENCES

1. Ceze,L., Nivala,J. and Strauss,K. (2019) Molecular digital data storage using DNA. *Nat. Rev. Genet.*, **20**, 456–466.
2. Dong,Y., Sun,F., Ping,Z., Ouyang,Q. and Qian,L. (2020) DNA storage: research landscape and future prospects. *Nati. Sci. Rev.*, **7**, 1092–1107.
3. Erlich,Y. and Zielinski,D. (2017) DNA Fountain enables a robust and efficient storage architecture. *Science*, **355**, 950–954.
4. Bancroft,C., Bowler,T., Bloom,B. and Clelland,C.T. (2001) Long-term storage of information in DNA. *Science*, **293**, 1763–1763.

5. Organick,L., Ang,S.D., Chen,Y.-J., Lopez,R., Yekhanin,S., Makarychev,K., Racz,M.Z., Kamath,G., Gopalan,P., Nguyen,B. *et al.* (2018) Random access in large-scale DNA data storage. *Nat. Biotechnol.*, **36**, 242.
6. Zhirnov,V., Zadegan,R.M., Sandhu,G.S., Church,G.M. and Hughes,W.L. (2016) Nucleic acid memory. *Nat. Mater.*, **15**, 366–370.
7. Church,G.M., Gao,Y. and Kosuri,S. (2012) Next-generation digital information storage in DNA. *Science*, **337**, 1628–1628.
8. Goldman,N., Bertone,P., Chen,S., Dessimoz,C., LeProust,E.M., Sipos,B. and Birney,E. (2013) Towards practical, high-capacity, low-maintenance information storage in synthesized DNA. *Nature*, **494**, 77–80.
9. Yachie,N., Sekiyama,K., Sugahara,J., Ohashi,Y. and Tomita,M. (2007) Alignment-based approach for durable data storage into living organisms. *Biotechnol. Progr.*, **23**, 501–505.
10. Clelland,C.T., Risca,V. and Bancroft,C. (1999) Hiding messages in DNA microdots. *Nature*, **399**, 533–534.
11. Arita,M. and Ohashi,Y. (2004) Secret signatures inside genomic DNA. *Biotechnol. Progr.*, **20**, 1605–1607.
12. Heider,D. and Barnekow,A. (2007) DNA-based watermarks using the DNA-Crypt algorithm. *BMC Bioinformatics*, **8**, 176.
13. Heider,D. and Barnekow,A. (2008) DNA watermarks: a proof of concept. *BMC Mol. Biol.*, **9**, 40.
14. Heider,D., Kessler,D. and Barnekow,A. (2008) Watermarking sexually reproducing diploid organisms. *Bioinformatics*, **24**, 1961–1962.
15. Heider,D., Pyka,M. and Barnekow,A. (2009) DNA watermarks in non-coding regulatory sequences. *BMC Res Notes*, **2**, 125.
16. Limbachiya,D., Gupta,M.K. and Aggarwal,V. (2018) Family of constrained codes for archival DNA data storage. *IEEE Commun. Lett.*, **22**, 1972–1975.
17. DNA data storage alliance (2021) Preserving our digital legacy: an introduction to DNA data storage. https://dnastoragealliance.org/publications/.
18. Heckel,R., Mikutis,G. and Grass,R.N. (2019) A characterization of the DNA data storage channel. *Sci. Rep.-UK*, **9**, 9663.
19. Schwarz,M., Welzel,M., Kabdullayeva,T., Becker,A., Freisleben,B. and Heider,D. (2020) MESA: automated assessment of synthetic DNA fragments and simulation of DNA synthesis, storage, sequencing and PCR errors. *Bioinformatics*, **36**, 3322–3326.
20. Löchel,H.F. and Heider,D. (2020) Comparative analyses of error handling strategies for next-generation sequencing in precision medicine. *Sci. Rep.-UK*, **10**, 5750.
21. Immink,K. A.S. and Cai,K. (2020) Properties and constructions of constrained codes for DNA-based data storage. *IEEE Access*, **8**, 49523–49531.
22. Jensen,M.A., Fukushima,M. and Davis,R.W. (2010) DMSO and betaine greatly improve amplification of GC-rich constructs in de novo synthesis. *PLoS ONE*, **5**, e11024.
23. Minoche,A.E., Dohm,J.C. and Himmelbauer,H. (2011) Evaluation of genomic high-throughput sequencing data generated on Illumina HiSeq and Genome Analyzer systems. *Genome Biol.*, **12**, R112.
24. Ma,S., Tang,N. and Tian,J. (2012) DNA synthesis, assembly and applications in synthetic biology. *Curr. Opin. Chem. Biol.*, **16**, 260–267.
25. Shendure,J., Balasubramanian,S., Church,G.M., Gilbert,W., Rogers,J., Schloss,J.A. and Waterston,R.H. (2017) DNA sequencing at 40: past, present and future. *Nature*, **550**, 345–353.
26. Schindler,D., Milbredt,S., Sperlea,T. and Waldminghaus,T. (2016) Design and assembly of DNA sequence libraries for chromosomal insertion in bacteria based on a set of modified MoClo vectors. *ACS Synth. Biol.*, **5**, 1362–1368.
27. Song,W., Cai,K., Zhang,M. and Yuen,C. (2018) Codes with run-length and GC-content constraints for DNA-based data storage. *IEEE Commun. Lett.*, **22**, 2004–2007.
28. Immink,K. A.S. and Cai,K. (2019) Efficient balanced and maximum homopolymer-run restricted block codes for DNA-based data storage. *IEEE Commun. Lett.*, **23**, 1676–1679.
29. Wang,Y., Noor-A-Rahim,M., Gunawan,E., Guan,Y.L. and Poh,C.L. (2019) Construction of bio-constrained code for DNA data storage. *IEEE Commun. Lett.*, **23**, 963–966.
30. Dubé,D., Song,W. and Cai,K. (2019) DNA codes with run-length limitation and knuth-like balancing of the GC contents. In: *Symposium on Information Theory and its Applications (SITA), Japan*.
31. Wang,Y., Shen,Y., Zhang,X. and Cui,G. (2009) DNA codewords design using the improved NSGA-II algorithms. In: *2009 Fourth International on Conference on Bio-Inspired Computing*. IEEE.
32. Cao,B., Zhao,S., Li,X. and Wang,B. (2020) K-means multi-verse optimizer (KMVO) algorithm to construct DNA storage codes. *IEEE Access*, **8**, 29547–29556.
33. Gaborit,P. and King,O.D. (2005) Linear constructions for DNA codes. *Theor. Comp. Sci.*, **334**, 99–113.
34. Chee,Y.M. and Ling,S. (2008) Improved lower bounds for constant GC-content DNA codes. *IEEE T. Inform. Theory.*, **54**, 391–394.
35. Mandelbrot,B.B. and Mandelbrot,B.B. (1982) In: *The Fractal Geometry of Nature*. WH Freeman, NY, Vol. **1**
36. Peitgen,H.-O., Jürgens,H. and Saupe,D. (2006) *Chaos and Fractals: New Frontiers of Science*. Springer Science & Business Media.
37. Barnsley,M.F. (2012) *Fractals Everywhere: New Edition*. Dover Publications.
38. Jeffrey,H.J. (1990) Chaos game representation of gene structure. *Nucleic Acids Res.*, **18**, 2163–2170.
39. Löchel,H.F. and Heider,D. (2021) Chaos game representation and its applications in bioinformatics. *Comput. Struct. Biotechnol. J.*, **19**, 6263–6271.
40. Joseph,J. and Sasikumar,R. (2006) Chaos game representation for comparison of whole genomes. *BMC Bioinformatics*, **7**, 243.
41. Deschavanne,P.J., Giron,A., Vilain,J., Fagot,G. and Fertil,B. (1999) Genomic signature: characterization and classification of species assessed by chaos game representation of sequences.. *Mol. Biol. Evol.*, **16**, 1391–1399.
42. Almeida,J.S., Carrico,J.A., Maretzek,A., Noble,P.A. and Fletcher,M. (2001) Analysis of genomic sequences by Chaos Game Representation. *Bioinformatics*, **17**, 429–437.
43. Löchel,H.F., Eger,D., Sperlea,T. and Heider,D. (2020) Deep learning on chaos game representation for proteins. *Bioinformatics*, **36**, 272–279.
44. He,M. and Petoukhov,S. (2010) The genetic code, Hadamard matrices and algebraic biology. *J. Biol. Syst.*, **18**, 159–175.
45. He,M. and Petoukhov,S. (2011) In: *Mathematics of Bioinformatics: Theory, Methods and Applications*. John Wiley & Sons, Vol. **19**.
46. Petoukhov,S. and He,M. (2009) In: *Symmetrical Analysis Techniques for Genetic Systems and Bioinformatics: Advanced Patterns and Applications: Advanced Patterns and Applications*. IGI Global.
47. Hao,B.-L., Lee,H.-C. and Zhang,S.-Y. (2000) Fractals related to long DNA sequences and complete genomes. *Chaos Solitons Fractals*, **11**, 825–836.
48. Anitas,E.M. (2020) Small-angle scattering and multifractal analysis of DNA sequences. *Int. J. Mol. Sci.*, **21**, 4651.
49. Hamming,R.W. (1950) Error detecting and error correcting codes. *Bell Syst. Tech. J.*, **29**, 147–160.
50. Ahmed,E.S. (2012) Dual-mode dual-band microstrip bandpass filter based on fourth iteration T-square fractal and shorting pin.. *Radioengineering*, **21**, 617–623.
51. Campbell,J. (2020) On the visualization of large-order graph distance matrices. *J. Math. Arts*, **14**, 297–330.
52. Deaton,R.J., Murphy,R.C., Garzon,M.H., Franceschetti,D.R. and Stevens,S.E. Jr, (1996) Good encodings for DNA-based solutions to combinatorial problems. In: *DNA Based Computers*. pp. 247–258.