Snake-in-the-Box Codes for Rank Modulation

Yonatan Yehezkeally and Moshe Schwartz, *Senior Member, IEEE*

***Abstract*—Motivated by the rank-modulation scheme with applications to flash memory, we consider Gray codes capable of detecting a single error, also known as snake-in-the-box codes. We study two error metrics: Kendall's τ-metric, which applies to charge-constrained errors, and the ℓ_∞-metric, which is useful in the case of limited magnitude errors. In both cases we construct snake-in-the-box codes with rate asymptotically tending to 1. We also provide efficient successor-calculation functions, as well as ranking and unranking functions. Finally, we also study bounds on the parameters of such codes.**

***Index Terms*—Snake-in-the-box codes, rank modulation, permutations, flash memory**

I. Introduction

FLASH memory is non-volatile storage medium which is electrically programmable and erasable. Its current wide use is motivated by its high storage density and relative low cost. Among the chief disadvantages of flash memories is their inherent asymmetry between cell programming (injecting cells with charge) and cell erasure (removing charge from cells). While single cells can be programmed with relative ease, in the current architecture, the process of erasure can only preformed by completely depleting large blocks of cells of their charge. Moreover, the removal of charge from cells physically damages cells over time.

This issue is exacerbated as a result of the ever-present demand for denser memory: smaller cells are more delicate, and get damaged faster during erasure. They also contain less charge and are therefore more prone to error. In addition, flash memories, at present, use multilevel cells, where charge-levels are quantized to simulate a finite alphabet – the more levels, the less safety margins are left, and data integrity is compromised. Thus, over-programming (increasing a cell's charge-level above the designated mark) is a real problem, requiring a costly and damaging erasure cycle. Hence, in a programming cycle, charge-levels are usually made to gradually approach the desirable amount, making for lengthier programming cycles as well (see [3]).

In an effort to counter these effects, a different modulation scheme has been suggested for flash memories recently – rank modulation [10]. This scheme calls for the representation of the data stored in a group of cells in the permutation suggested by their relative charge-levels. That is, if $c_1, c_2, \ldots, c_n \in \mathbb{R}$ represent the charge-levels of $n \in \mathbb{N}$ cells, then that group of cells is said to encode that permutation $\sigma \in S_n$ such that:

$$c_{\sigma(1)} > c_{\sigma(2)} > \ldots > c_{\sigma(n)}.$$

Yonatan Yehezkeally is with the Department of Electrical and Computer Engineering, Ben-Gurion University of the Negev, Beer Sheva 84105, Israel (e-mail: yonatany@bgu.ac.il).

Moshe Schwartz is with the Department of Electrical and Computer Engineering, Ben-Gurion University of the Negev, Beer Sheva 84105, Israel (e-mail: schwartz@ee.bgu.ac.il).

This work was supported in part by ISF grant 134/10.

This scheme eliminates the need for discretization of charge-levels. Furthermore, restricting ourselves to programming the group of cells only by increasing the charge-level of a given cell above that of any other cell in the group, over-programming is no longer an issue. This operation was named in [10] as a "push-to-the-top" operation.

In addition, storing data using this scheme also improves the memory's robustness against other noise types. Retention, the process of slow charge leakage from cells, tends to affect all cells in a similar direction [3]. Since rank modulation stores information in the differences between charge-levels rather than their absolute values, data stored using it is more resilient to this sort of noise.

Gray codes using "push-to-the-top" operations and spanning the entire space of permutations were also studied in [10]. The Gray code [7] was first introduced as a sequence of distinct binary vectors of fixed length, where every adjacent pair differs in a single coordinate. It has since been generalized to sequences of distinct states $s_1, s_2, \ldots, s_k \in S$ such that for every $i < k$ there exists a function in a predetermined set of transitions $t \in T$ such that $s_{i+1} = t(s_i)$ (see [15] for an excellent survey). When the states one considers are permutations on $n \in \mathbb{N}$ elements and the allowed transitions are "push-to-the-top" operations, [10] referred to such Gray codes as *n-length Rank-Modulation Gray Codes* (n-RMGC's), and it presented such codes traversing the entire set of permutations. In this fashion, a set of n rank-modulation cells could implement a single logical multilevel cell with $n!$ levels, where increasing the logical cell's level by 1 corresponds to a single transition in the n-RMGC. This allows for a natural integration of rank modulation with other multilevel approaches such as rewriting schemes [4], [8], [9], [20].

Other recent works have explored error-correcting codes for rank modulation, where different types of errors are addressed by a careful choice of metric. In [11], Kendall's τ-metric was considered, since a small charge-constrained error translates into a small distance in the metric. In contrast, the ℓ_∞-metric was used in [13], [18], as small distances in the metric correspond to small limited-magnitude errors.

In this paper, we explore Gray codes for rank modulation which detect a single error, under both metrics mentioned above. Such codes are known as *snake-in-the-box codes*, and have been studied extensively for binary vectors with the Hamming metric and with single-bit flips as allowable transitions (see [1] and references therein).

The paper is organized as follows: In Section II we present basic notation and definitions. In Section III we review properties of Kendall's τ-metric, present a recursive construction of snake-in-the-box codes over the alternating groups of odd orders, with asymptotically-optimal rate, then present auxiliary functions needed for the use of codes generated by this

construction, and conclude by presenting upper-bounds on the size of such snake-in-the-box codes. In Section IV we present a direct construction of snake-in-the-box codes of every order in the ℓ_∞-metric based on results from [10] which we show have asymptotically-optimal rate, and also present some required auxiliary functions. We conclude in Section V with some ad-hoc results, as well as some open questions.

II. Preliminaries

We shall denote by $\sigma = [a_1, a_2, \ldots, a_n]$ the permutation over $[n] \triangleq \{1, 2, \ldots, n\}$ such that for all $i \in [n]$ it holds that $\sigma(i) = a_i$ (and, naturally, $\{a_1, a_2, \ldots, a_n\} = [n]$). This form is called the *vector notation* for permutation. We let $S_n = \mathrm{Sym}[n]$ be the symmetric group on $[n]$, and $A_n \leqslant S_n$ be the alternating group of the same order. For $\sigma, \tau \in S_n$, their composition, denoted $\sigma\tau$, is the permutation for which $\sigma\tau(i) = \sigma(\tau(i))$ for all $i \in [n]$.

A cycle, denoted $(a_1, a_2, \ldots, a_k)$, is a permutation mapping $a_i \mapsto a_{i+1}$ for all $i \in [k-1]$, as well as $a_k \mapsto a_1$. We shall occasionally use *cycle notation* in which a permutation is described as a composition of cycles. We also recall that any permutation may be represented as a composition of cycles of size 2, and that the parity of the number of these cycles does not depend on the decomposition. Thus we have *even* and *odd* permutations, with positive and negative *signs*, respectively.

Definition 1. *Given a set S and a subset of transformations $T \subseteq S^S = \{f \mid f : S \to S\}$, a* Gray code *over S, using transitions T, of size $M \in \mathbb{N}$, is a sequence $C = (c_0, c_1, \ldots, c_{M-1})$ of M distinct elements of S, called* codewords*, such that for all $j \in [M-1]$ there exists $t \in T$ such that $c_j = t(c_{j-1})$.*

Alternatively, when the original permutation c_0 is known (or irrelevant), we use a slight abuse of notation in referring to the sequence of transformations $(t_{k_1}, \ldots, t_{k_{M-1}})$ generating the code (i.e., $c_j = t_{k_j}(c_{j-1})$) as the code itself.

In the above definition, when $M = |S|$ the Gray code is called *complete*. If there exists $t \in T$ such that $t(c_{M-1}) = c_0$ the Gray code is called *cyclic*, M is called its *period*, and we shall, when listing the code by its sequence of transformations, include $t_{k_M} \triangleq t$ at the end of the list. The *rate* of C, denoted $R(C)$, is defined as

$$R(C) \triangleq \frac{\log_2 M}{\log_2 |S|}.$$

In the context of rank modulation for flash memories, the set of transformations T comprises of "push-to-the-top" operations, first used in [10], and later also in [6], [16], [19]. We denote by $t_i \in \mathrm{Aut}(S_n)$ the "push-to-the-top" operation on index i, i.e.,

$$t_i[a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n] = \\ = [a_i, a_1, a_2, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n],$$

and throughout the paper set $T = \{t_2, t_3, \ldots, t_n\}$. Restricting the transformations to "push-to-the-top" operations allows fast cell programming, and eliminates overshoots (see [10]).

For ease of presentation only, we also denote by $\underline{t}_i$ the "push-to-the-bottom" operation on index $n+1-i$, i.e.,

$$\underline{t}_i[a_1, a_2, \ldots, a_{n-i}, a_{n+1-i}, a_{n+2-i}, \ldots, a_n] = \\ = [a_1, a_2, \ldots, a_{n-i}, a_{n+2-i}, \ldots, a_n, a_{n+1-i}].$$

Let $d : S \times S \to \mathbb{N} \cup \{0\}$ be a distance function inducing a metric $\mathcal{M}$ over S. Given a transmitted codeword $c \in C$ and its received version $\tilde{c} \in S$, we say a single error occurred if $d(c, \tilde{c}) = 1$. We are interested in Gray codes capable of detecting single errors, which we now define.

Definition 2. *Let $\mathcal{M}$ be a metric over S induced by a distance measure d. A* snake-in-the-box code *over $\mathcal{M}$ and S, using transitions T, is a Gray code C also over S and using T, in which for every pair of distinct elements $c, c' \in C$, $c \neq c'$, one has $d(c, c') \geqslant 2$.*

Since throughout the paper, our ambient space is S_n, and the transformations we use are the "push-to-the-top" operations T, we shall abbreviate our notation and call the snake-in-the-box code of size M an $(n, M, \mathcal{M})$-*snake*, or an $\mathcal{M}$-*snake*. We will be considering two metrics in the next sections: Kendall's τ-metric, $\mathcal{K}$, and the ℓ_∞-metric, with their respective $\mathcal{K}$-snakes and ℓ_∞-snakes.

It is interesting to note that the classical definition of snake-in-the-box codes (see the survey [1]) is slightly weaker in the sense that $d(c, c') \geqslant 2$ is required for distinct $c, c' \in C$, *unless* c and c' are adjacent in C. This, however, is a compromise due to the fact that in the classical codes over binary vectors, the transformations (which flip a single bit) always create adjacent codewords at distance 1 apart. This compromise is unnecessary in our case since, as we shall later see, the "push-to-the-top" operations allow adjacent words at distance 2 or more apart.

III. Kendall's τ-Metric and $\mathcal{K}$-Snakes

Kendall's τ-metric [12], denoted $\mathcal{K}$, is induced by the bubble-sort distance which measures the minimal amount of adjacent transpositions required to transform one permutation into the other. For example, the distance between the permutations $[2, 1, 4, 3]$ and $[2, 4, 3, 1]$ is 2, as

$$[2, 1, 4, 3] \to [2, 4, 1, 3] \to [2, 4, 3, 1]$$

is a shortest sequence of adjacent transpositions between the two. More formally, for $\alpha, \beta \in S_n$, as noted in [11],

$$d_K(\alpha, \beta) = \{(i, j) \mid \alpha(i) < \alpha(j) \wedge \beta(i) > \beta(j)\}.$$

The metric $\mathcal{K}$ was first introduced by Kendall [12] in the study of ranking in statistics. It was observed in [11] that a bounded distance in Kendall's τ-metric models errors caused by bounded changes in charge-levels of cells in the flash memory. Error-correcting codes for this metric were studied in [2], [11].

We let Kendall's τ *adjacency graph* of order $n \in \mathbb{N}$ be the graph $G_n = (V_n, E_n)$ whose vertices are the elements of the symmetric group $V_n = S_n$, and $\{\alpha, \beta\} \in E_n$ if and only if $d_K(\alpha, \beta) = 1$. It is well known that Kendall's τ-metric is *graphic* [5], i.e., for every $\alpha, \beta \in S_n$, $d_K(\alpha, \beta)$ equals the length of the shortest path between the two in the adjacency graph, G_n.

A. Construction

We begin the construction process by restricting ourselves to Gray codes using only "push-to-the-top" operations on odd indices. The following lemma provides the motivation for this restriction.

Lemma 3. *A Gray code over S_n using only "push-to-the-top" operations on odd indices is a $\mathcal{K}$-snake.*

Proof: One can readily verify that a "push-to-the-top" operation on an odd index is an even permutation. Thus, the codewords in a Gray code using only such operation are all with the same sign.

On the other hand, an adjacent transposition is an odd permutation, thus, flipping the sign of the permutation it acts on. It follows that in a list of codewords, all with the same sign, there are no two codewords which are adjacent in G_n, i.e., the Gray code is a $\mathcal{K}$-snake. ∎

Lemma 3 saves us the need to check whether a Gray code is in fact a $\mathcal{K}$-snake, at the cost of restricting the set of allowed transitions. In particular, if n is even, the last element cannot be moved. By starting with an even permutation and using only "push-to-the-top" operations on odd indices we get a sequence of even permutations, i.e., from the alternating group of same order. Thus, throughout this part, the context of the alternating group A_{2n+1} is assumed, where $n \in \mathbb{N}$.

The construction we are about to present is recursive in nature. As a base for the recursion, we note that three consecutive "push-to-the-top" operations on the 3rd index of permutations in A_3 constitute a complete cyclic $(3,3,\mathcal{K})$-snake:

$$C_3 \triangleq ([1,2,3],[3,1,2],[2,3,1]).$$

Now, assume that there exists a cyclic $(2n-1, M_{2n-1}, \mathcal{K})$-snake, C_{2n-1}, and let

$$t_{k_1}, t_{k_2}, \ldots, t_{k_{M_{2n-1}}}$$

be the sequence of transformations generating it, where k_j is odd for all $j \in [M_{2n-1}]$. We also assume that $k_1 = 2n-1$ (this requirement, while perhaps appearing arbitrary, is actually quite easily satisfied. Indeed, every sufficiently large cyclic $\mathcal{K}$-snake over S_{2n-1} must, WLOG, satisfy it. We shall make it a point to demonstrate that this holds for our construction).

We fix arbitrary values for $a_0, a_1, \ldots, a_{2n-2}$ such that

$$\{a_0, a_1, \ldots, a_{2n-2}\} = [2n+1] \setminus \{1,3\}. \tag{1}$$

Throughout the paper we shall take the indices of a to be modulo $2n-1$. For all $i \in [2n-1]$ we define

$$\sigma_0^{(i)} \triangleq [1, a_i, 3, a_{i+1}, \ldots, a_{i+2n-2}],$$

such that we indeed have $\sigma_0^{(i)} \in A_{2n+1}$, i.e., $\sigma_0^{(i)}$ is an even permutation (one simple way of achieving this is to choose them in ascending order).

We now define for all $i \in [2n-1]$ and $j \in [M_{2n-1}]$ the permutation

$$\sigma_{j(2n+1)}^{(i)} \triangleq \underline{t}_{k_j}\left(\sigma_{(j-1)(2n+1)}^{(i)}\right),$$

i.e., we construct cycles corresponding to a mirror view of C_{2n-1} on all but the two uppermost indices of $\sigma_0^{(i)}$ (which, as we recall, are $(1, a_i)$). We now note the following properties of our construction:

Lemma 4. *Let $i,k \in [2n-1]$ and $j,l \in [M_{2n-1}]$. The following are equivalent:*

1) *The permutations $\sigma_{j(2n+1)}^{(i)}$ and $\sigma_{l(2n+1)}^{(k)}$ are cyclic shifts of each other.*
2) $\sigma_{j(2n+1)}^{(i)} = \sigma_{l(2n+1)}^{(k)}$.
3) $i = k$ *and* $j = l$.

Proof: First, if $\sigma_{j(2n+1)}^{(i)}$ is a cyclic shift of $\sigma_{l(2n+1)}^{(k)}$, since

$$\sigma_{j(2n+1)}^{(i)}(1) = 1 = \sigma_{l(2n+1)}^{(k)}(1)$$

then necessarily

$$\sigma_{j(2n+1)}^{(i)} = \sigma_{l(2n+1)}^{(k)}.$$

It then follows that

$$a_i = \sigma_{j(2n+1)}^{(i)}(2) = \sigma_{l(2n+1)}^{(k)}(2) = a_k,$$

hence $i = k$. Moreover, since the two permutations' last $n-1$ elements agree, and $t_{k_1}, t_{k_2}, \ldots, t_{k_{M_{2n-1}}}$ induce a Gray code, then $j = l$.

Finally, that the last statement implies the first is trivial. ∎

Lemma 5. *For all $i \in [2n-1]$ it holds that*

$$\sigma_{M_{2n-1}(2n+1)}^{(i)} = \sigma_0^{(i)}.$$

Proof: The transformations $t_{k_1}, t_{k_2}, \ldots, t_{k_{M_{2n-1}}}$ induce a cyclic code, and the claim follows directly. ∎

Therefore we have constructed $2n-1$ cycles comprised of cyclically non-equivalent permutations (although, at this point they are not generated by "push-to-the-top" operations).

It shall now be noted that

$$\underline{t}_k = t_{2n+1}^{2n} t_{2n+2-k}.$$

Hence, if we define for all $i \in [2n-1]$, $0 \leqslant j < M_{2n-1}$, and $1 < m \leqslant 2n$, the permutations

$$\sigma_{j(2n+1)+1}^{(i)} \triangleq t_{2n+2-k_{j+1}} \sigma_{j(2n+1)}^{(i)}$$
$$\sigma_{j(2n+1)+m}^{(i)} \triangleq t_{2n+1}^{m-1} \sigma_{j(2n+1)+1}^{(i)},$$

then it holds that

$$\sigma_{(j+1)(2n+1)}^{(i)} = t_{2n+1} \sigma_{j(2n+1)+2n}^{(i)}.$$

Our observation from one paragraph above means that at this point we have $2n-1$ disjoint cycles, which we conveniently denote

$$C_{2n+1}^{(i)} \triangleq \left(\sigma_0^{(i)}, \sigma_1^{(i)}, \ldots, \sigma_{M_{2n-1}(2n+1)-1}^{(i)}\right),$$

for all $i \in [2n-1]$ (for ease of notation, we let $C_{2n+1}^{(0)} = C_{2n+1}^{(2n-1)}$). Each of the cycles is of size $(2n+1)M_{2n-1}$, is generated by "push-to-the-top" operations, and contains all cyclic shifts of elements present in our previous version of that cycle.

Theorem 6. *Given a cyclic $(2n-1, M_{2n-1}, \mathcal{K})$-snake using only "push-to-the-top" operations on odd indices, and such*

$$\begin{array}{c|c|c|c|c|c}
[5,3,1,2,4] & \sigma_2^{(0)} & [2,3,1,4,5] & \sigma_2^{(1)} & [4,3,1,5,2] & \sigma_2^{(2)} \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
[1,2,4,5,3] & \sigma_5^{(0)} & [1,4,5,2,3] & \sigma_5^{(1)} & [1,5,2,4,3] & \sigma_5^{(2)} \\
[4,1,2,5,3] & \sigma_6^{(0)} & [5,1,4,2,3] & \sigma_6^{(1)} & [2,1,5,4,3] & \sigma_6^{(2)} \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
[1,2,5,3,4] & \sigma_{10}^{(0)} & [1,4,2,3,5] & \sigma_{10}^{(1)} & [1,5,4,3,2] & \sigma_{10}^{(2)} \\
[5,1,2,3,4] & \sigma_{11}^{(0)} & [2,1,4,3,5] & \sigma_{11}^{(1)} & [4,1,5,3,2] & \sigma_{11}^{(2)} \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
[1,2,3,4,5] & \sigma_0^{(0)} & [1,4,3,5,2] & \sigma_0^{(1)} & [1,5,3,2,4] & \sigma_0^{(2)} \\
[3,1,2,4,5] & \sigma_1^{(0)} & [3,1,4,5,2] & \sigma_1^{(1)} & [3,1,5,2,4] & \sigma_1^{(2)}
\end{array}$$

Figure 1. A $(5, 45, \mathcal{K})$-snake, C_5, from Theorem 6. Down arrows stand for an omitted sequence of t_5 transformations. The transition from column to column uses a single t_3 transformation.

that its first transformation is t_{2n-1}, there exists a cyclic $(2n+1, M_{2n+1}, \mathcal{K})$-snake with the same properties, whose size is $M_{2n+1} = (2n-1)(2n+1)M_{2n-1}$.

Proof: Since $k_1 = 2n-1$, it holds for all $i \in [2n-1]$ that $\sigma_1^{(i)} = t_3\sigma_0^{(i)}$, and we recall $\sigma_2^{(i)} = t_{2n+1}\sigma_1^{(i)}$. More explicitly,

$$\begin{aligned}
\sigma_1^{(i)} &= [3, 1, a_i, a_{i+1}, \ldots, a_{i+2n-2}] \\
\sigma_2^{(i)} &= [a_{i+2n-2}, 3, 1, a_i, a_{i+1}, \ldots, a_{i+2n-3}],
\end{aligned}$$

where, again, the indices are taken modulo $2n-1$. Thus for all $i \in [2n-2]$ we have

$$t_3\sigma_1^{(i)} = [a_i, 3, 1, a_{i+1}, \ldots, a_{i+2n-2}] = \sigma_2^{(i+1)}$$

and $t_3\sigma_1^{(2n-1)} = \sigma_2^{(1)}$.

Let E denote the left-shift operator, and so

$$E^2C_{2n+1}^{(i)} = \left(\sigma_2^{(i)}, \sigma_3^{(i)}, \ldots, \sigma_{M_{2n-1}(2n+1)-1}^{(i)}, \sigma_0^{(i)}, \sigma_1^{(i)}\right).$$

By the above observations we conclude that

$$C_{2n+1} \triangleq E^2C_{2n+1}^{(0)}, E^2C_{2n+1}^{(1)}, \ldots, E^2C_{2n+1}^{(2n-2)}$$

is a cyclic $(2n+1, M_{2n+1}, \mathcal{K})$-snake, consisting of

$$M_{2n+1} = (2n-1)(2n+1)M_{2n-1}$$

permutations. The code C_{2n+1} obviously uses t_{2n+1}, and so some cyclic shift of it has it as its first transition (in fact, for every $i \in [2n-1]$ one has $\sigma_3^{(i)} = t_{2n+1}\sigma_2^{(i)}$, and in particular, $E^2C_{2n+1}^{(0)}$ has t_{2n+1} as its first transition, and so does C_{2n+1}). Finally, it is easily verifiable that all "push-to-the-top" operations are on odd indices. (See an example in Figure 1.) ∎

A property of rank-modulation cell programming is that an erasure of an entire cell block is required only when a specific cell is to exceed its maximal permitted charge level. It is therefore of interest to analyze the rate with which our constructed codes increase the charge level of any given cell.

Repeated "push-to-the-top" operations on a given cell will result in a fast increase in that cell's charge level, and growing gaps between it and the charge levels of other cells. It is therefore most cost-economic, in the sense that it delays the need for a time-consuming erasure and reprogramming cycle, to employ a programming strategy which retains the charge levels of individual cells as balanced as possible. Such balanced Gray codes were constructed in [10].

In this part's context, this goal is achieved if and only if every two subsequent incidents in a cyclic $(2n+1, M, \mathcal{K})$-snake where a "push-to-the-top" operation is applied to a certain cell are separated by at most $2n+1$ operations on other cells. Our family of codes nearly achieves this goal:

Lemma 7. *For every permutation $\sigma \in C_{2n+1}$, in the $\mathcal{K}$-snake constructed in Theorem 6, there exists another $\sigma' \in C_{2n+1}$ such that $\sigma(1) = \sigma'(1)$, following it by no more than $2n+3$ steps.*

Proof: Recall that

$$C_{2n+1} = E^2C_{2n+1}^{(0)}, E^2C_{2n+1}^{(1)}, \ldots, E^2C_{2n+1}^{(2n-2)}.$$

By the nature of our construction, for $n \geqslant 2$, every "push-to-the-top" operation, on all but the last rank in the code, appears either as part of the pattern

$$\ldots, \underbrace{t_{2n+1}, \ldots, t_{2n+1}}_{2n}, t_i, \underbrace{t_{2n+1}, \ldots, t_{2n+1}}_{2n}, \ldots$$

or as

$$\ldots, \underbrace{t_{2n+1}, \ldots, t_{2n+1}}_{2n}, t_3, t_3, \underbrace{t_{2n+1}, \ldots, t_{2n+1}}_{2n}, \ldots$$

It is therefore the case that there exist $0 \leqslant k \leqslant 2n$ and $j \in [n]$ such that the transformations used in C_{2n+1} after σ are of the following two forms:

1) $\underbrace{t_{2n+1}, \ldots, t_{2n+1}}_{k}, t_{2j+1}, \underbrace{t_{2n+1}, \ldots, t_{2n+1}}_{2n}$
2) $\underbrace{t_{2n+1}, \ldots, t_{2n+1}}_{k}, t_3, t_3, \underbrace{t_{2n+1}, \ldots, t_{2n+1}}_{2n}$

In the second case, one notes:

$$\sigma(1) = \begin{cases}
t_{2n+1}^{2n-1}t_3^2\sigma(1) & k = 0 \\
t_3^2t_{2n+1}\sigma(1) & k = 1 \\
t_3t_{2n+1}^2\sigma(1) & k = 2 \\
t_{2n+1}^{2n+1-k}t_3^2t_{2n+1}^k\sigma(1) & k > 2.
\end{cases}$$

Finally, in the first case, we note that

$$\sigma(1) = \begin{cases}
t_{2n+1}^{2n-k}t_{2j+1}t_{2n+1}^k\sigma(1) & k < 2j+1 \\
t_{2j+1}t_{2n+1}^k\sigma(1) & k = 2j+1 \\
t_{2n+1}^{2n+1-k}t_{2j+1}t_{2n+1}^k\sigma(1) & k > 2j+1.
\end{cases}$$

∎

It is of interest to note that, of all cases discussed in the last proof, the second case where $k > 2$ is the only situation in which another instance of programming to the specific cell fails to occur in $2n+2$ steps, i.e., for the large majority of cases (in all but $\frac{2n-1}{M_{2n+1}}$ of them), the construction of Theorem 6 yields optimally-behaving codes.

We now turn to consider the rate of the constructed codes, and show that it is asymptotically optimal.

Theorem 8. *The $\mathcal{K}$-snakes constructed in Theorem 6 have an asymptotically-optimal rate.*

Proof: Starting from our base case of a complete cyclic $(3,3,\mathcal{K})$-snake, we define for all $n \in \mathbb{N}$ the ratio

$$D_{2n+1} \triangleq \frac{M_{2n+1}}{(2n+1)!},$$

which is the size of our constructed code over the total size of S_{2n+1}. We note that

$$\frac{D_{2n+1}}{D_{2n-1}} = \frac{M_{2n+1} \cdot (2n-1)!}{(2n+1)! \cdot M_{2n-1}} = \frac{2n-1}{2n}.$$

Therefore, since $D_3 = \frac{1}{2}$, we have for all $2 \leqslant n \in \mathbb{N}$ that

$$D_{2n+1} = \frac{1}{2} \prod_{m=2}^{n} \frac{2m-1}{2m} = \frac{(2n)!}{n!^2 \cdot 2^{2n}}.$$

Using Stirling's approximation one observes

$$\begin{aligned} \lim_{n\to\infty} D_{2n+1}\sqrt{\pi n} &= \lim_{n\to\infty} \frac{(2n)!\sqrt{\pi n}}{n!^2 \cdot 2^{2n}} \\ &= \lim_{n\to\infty} \frac{\sqrt{4\pi n}\left(\frac{2n}{e}\right)^{2n}\sqrt{\pi n}}{\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^{n}\right)^2 \cdot 2^{2n}} = 1, \end{aligned}$$

and therefore it holds that

$$\lim_{n\to\infty} R(C_{2n+1}) = \lim_{n\to\infty} \frac{\log_2 M_{2n+1}}{\log_2 |S_{2n+1}|} = 1.$$

■

B. Successor Calculation and Ranking Algorithms

We now turn to present algorithms associated with the codes we constructed in the previous section. The algorithms are brought here for completeness of presentation, and are straightforward derivations from the construction. We shall, therefore, only provide an intuitive sketch of correctness for them, as we shall later do in the section corresponding to ℓ_∞-snakes.

In order to use the codes described in Theorem 6 in the implementation of a logic cell (with M_{2n+1} levels), importance is known to the ability of efficiently increasing the cell's level, i.e., one needs to know for every given permutation in the code the appropriate "push-to-the-top" operation required to produce the subsequent permutation.

For the code C_{2n+1} from Theorem 6, the function $\texttt{Successor}_{\mathcal{K}}\left(n,[b_1,\ldots,b_{2n+1}]\right)$ takes as input a permutation in the code, and returns as output the index i of the required transformation t_i. It is assumed throughout this part that the elements $\{a_i\}_{i=0}^{2n-2}$ from (1), used in our construction, are known, and we will denote them with superscript (n) to indicate order when it is not clear from context. Furthermore, we require a function

$$\texttt{Ind}_n(b) : [2n+1] \setminus \{1,3\} \to [0,2n-2]$$

which returns the unique index such that $a_{\texttt{Ind}_n(b)} = b$. We assume $\texttt{Ind}_n$ runes in $O(1)$ time[1]. One possible way, among many, of achieving this is by defining:

$$a_i^{(n)} \triangleq \begin{cases} 2 & i = 0 \\ i+3 & i \geqslant 1 \end{cases} \qquad \texttt{Ind}_n(b) \triangleq \begin{cases} 0 & b = 2, \\ b-3 & b \geqslant 4. \end{cases}$$

Finally, we naturally assume validity of the input in all procedures.

Our strategy will be to identify the vertices in C_{2n+1} which require a transformation other than t_{2n+1}. Those are either permutations with leading 1's (those on which we initially performed "push-to-the-bottom" operations in our construction), or the last permutation in each $E^2C_{2n+1}^{(j)}$. In the latter case we need only apply t_3, where the former requires translation of the $a_i^{(n)}$'s according to their respective positions in the originating permutation of each $C_{2n+1}^{(j)}$, and a recursive run of $\texttt{Successor}_{\mathcal{K}}$ to determine the correct "push-to-the-bottom" operation to be performed.

It shall be noted at this point that a degree of freedom exists in the cyclic shift of C_{2n-1} one applies to construct each $C_{2n+1}^{(j)}$ (one only needs to confirm that the first "push-to-the-top" operation shall be on the last index). This shift shall be denoted by the following bijection for every order $n \in \mathbb{N}$ and index $j \in [2n-1]$:

$${}_j^n\!\downarrow : \{3\} \cup \left\{a_i^{(n)}\right\}_{i\neq j} \longrightarrow [2n-1],$$

defined such that the "push-to-the-bottom" operation applied to

$$\left[1, a_j^{(n)}, b_1, \ldots, b_{2n-1}\right] \in C_{2n+1}^{(j)}$$

matches the "push-to-the-top" operation applied in C_{2n-1} to

$$\left[{}_j^n\!\downarrow b_{2n-1}, {}_j^n\!\downarrow b_{2n-2}, \ldots, {}_j^n\!\downarrow b_1\right].$$

We shall further denote its inverse as ${}_j^n\!\uparrow$. These two bijections can be implemented in $O(1)$ time, for example, by taking as a starting point C_{2n-1}'s $(2n-4)$-ranked permutation

$$\left[a_0^{(n-1)}, \ldots, a_{2n-4}^{(n-1)}, 3, 1\right],$$

and defining accordingly

$${}_j^n\!\downarrow b = \begin{cases} 1 & b = 3 \\ 3 & \texttt{Ind}_n(b) = j+1 \\ a^{(n-1)}_{(j-\texttt{Ind}_n(b)-1) \bmod (2n-1)} & \text{otherwise,} \end{cases} \tag{2}$$

where $\texttt{Ind}_n(b) = j+1$ is checked modulo $2n-1$, as well as

$${}_j^n\!\uparrow b = \begin{cases} 3 & b = 1 \\ a_{j+1}^{(n)} & b = 3 \\ a^{(n)}_{j-\texttt{Ind}_{n-1}(b)-1} & \text{otherwise.} \end{cases} \tag{3}$$

Lemma 9. $\texttt{Successor}_{\mathcal{K}}$ *runs in* $O(1)$ *amortized time.*

Proof: We first note that by the nature of our construction the element 1 appears in the leading index precisely $(2n-1)\cdot M_{2n-1}$ times, which constitutes $\frac{1}{2n+1}$ of the code's size. The pair $(3,1)$ leads no more (and in fact strictly less) permutations.

[1]Though the integers used throughout are of magnitude $O(n)$, and so may require $O(\log n)$ bits to represent, we tacitly assume (as in [10]) all simple integer operations, e.g., assignment, comparison, addition, etc., to take $O(1)$ time.

Function $\texttt{Successor}_{\mathcal{K}}\left(n,[b_1,\ldots,b_{2n+1}]\right)$

input : $n\in\mathbb{N}$, A permutation $[b_1,\ldots,b_{2n+1}]\in C_{2n+1}$
output : An odd $i\in\{3,\ldots,2n+1\}$ that determines the transition t_i to the next permutation in C_{2n+1}

if $n=1$ **then**
  return 3
if $b_1=3$ *and* $b_2=1$ *and* $\forall 3\leqslant i\leqslant 2n$: $(\texttt{Ind}_n(b_{i+1})-\texttt{Ind}_n(b_i))\equiv 1 \pmod{2n-1}$ **then**
  return 3
if $b_1=1$ **then**
  $j\leftarrow \texttt{Ind}_n(b_2)$
  $i\leftarrow \texttt{Successor}_{\mathcal{K}}\left(n-1,\left[{}^{n}_{j}\!\downarrow b_{2n+1},{}^{n}_{j}\!\downarrow b_{2n},\ldots,{}^{n}_{j}\!\downarrow b_3\right]\right)$
  return $2n+2-i$
return $2n+1$

Therefore, if we let E_n denote the expected number of steps performed by $\texttt{Successor}_{\mathcal{K}}$ when called on input of length $2n+1$, then we note the recursive connection

$$\begin{aligned}E_n &\leqslant O(1)+\frac{1}{2n+1}O(n)+\frac{1}{2n+1}\left(O(n)+E_{n-1}\right)\\ &= O(1)+\frac{1}{2n+1}E_{n-1}.\end{aligned}$$

Developing this inequality recursively, there exists $L\in\mathbb{N}$ such that

$$\begin{aligned}E_n \leqslant & L+\frac{1}{2n-1}E_{n-1}\\ &\leqslant \left(1+\frac{1}{2n-1}\right)L+\frac{1}{(2n-1)(2n-3)}E_{n-2}\leqslant\\ &\vdots\\ &\leqslant \left(1+\frac{1}{2n-1}+\frac{n-2}{(2n-1)(2n-3)}\right)L+\frac{n!2^n}{(2n)!}E_1,\end{aligned}$$

and so $E_n=O(1)$. ∎

To use C_{2n+1} in the implementation of a logic cell, one also needs a method of computing a given permutation's rank in the code. We implement the function $\texttt{Rank}_{\mathcal{K}}\left([b_1,\ldots,b_{2n+1}]\right)$ which receives as input a permutation $[b_1,\ldots,b_{2n+1}]\in C_{2n+1}$ and returns its rank in

$$C_{2n+1}=E^2C^{(0)}_{2n+1},E^2C^{(1)}_{2n+1},\ldots,E^2C^{(2n-2)}_{2n+1},$$

in the order indicated by that notation. The assumptions made in the previous part are still in effect. Moreover, we will require knowledge of the cyclic shift of C_{2n-1} used in the construction of each $C^{(j)}_{2n+1}$, which we retain in the form of $r^{(j)}_{2n+1}$, the rank of permutation in C_{2n-1} which was chosen as a starting point. For example, in the method suggested by (2) and (3), we have

$$r^{(j)}_{2n+1}=2n-4$$

for all $j\in[2n-1]$.

We use the following method: first identify the position of 1 in the permutation, and the following element, which gives us both the subcode the permutation belongs to and the cyclic shift in our mock "push-to-the-bottom" operation. Armed with that information we then scan the permutation backwards and translate the $a^{(n)}_j$'s indices according to the subcode in the same way we did in $\texttt{Successor}_{\mathcal{K}}$. After that, a

Function $\texttt{Rank}_{\mathcal{K}}\left([b_1,\ldots,b_{2n+1}]\right)$

input : A permutation $[b_1,\ldots,b_{2n+1}]\in C_{2n+1}$
output : The rank $k\in\{0,\ldots,M_{2n+1}-1\}$ associated with the given permutation in C_{2n+1}

if $n=1$ **then**
  return $3-b_2$
$i\leftarrow \min\{l\in[2n+1]\mid b_l=1\}$
$j\leftarrow \texttt{Ind}_n\left(b_{(i \bmod (2n+1))+1}\right)$
for $l\leftarrow 1$ **to** $2n-1$ **do**
  $c_l\leftarrow {}^{n}_{j}\!\downarrow b_{((i-l-1) \bmod (2n+1))+1}$
$r\leftarrow \left(\texttt{Rank}_{\mathcal{K}}\left([c_1,\ldots,c_{2n-1}]\right)-r^{(j)}_{2n+1}\right) \bmod M_{2n-1}$
$rn\leftarrow ((2n+1)(r-1)-1+((i-2) \bmod (2n+1))) \bmod ((2n+1)M_{2n-1})$
return $(2n+1)M_{2n-1}\cdot j+rn$

Function $\texttt{Unrank}_{\mathcal{K}}\left(n,k\right)$

input : $n\in\mathbb{N}$; rank $k\in[0,M_{2n+1}-1]$
output : The permutation $[b_1,\ldots,b_{2n+1}]$ which is kth in C_{2n+1}

if $n=0$ **then**
  return $[1]$
$j\leftarrow \left\lfloor \frac{k}{(2n+1)\cdot M_{2n-1}}\right\rfloor$
$pos\leftarrow k \bmod ((2n+1)M_{2n-1})$
$perm\leftarrow \left(\left\lfloor\frac{pos+1}{2n+1}\right\rfloor+1+r^{(j)}_{2n+1}\right) \bmod M_{2n-1}$
$shift\leftarrow (pos+2) \bmod (2n+1)$
$[c_1,\ldots,c_{2n-1}]\leftarrow \texttt{Unrank}_{\mathcal{K}}(n-1,perm)$
return $t^{shift}_{2n+1}\left[1,a^{(n)}_j,{}^{n}_{j}\!\uparrow c_{2n-1},{}^{n}_{j}\!\uparrow c_{2n-2},\ldots,{}^{n}_{j}\!\uparrow c_1\right]$

recursive run of $\texttt{Rank}_{\mathcal{K}}$ will give us the permutation's position in its subcode, which we will combine with the cyclic shift to produce the correct rank, taking $r^{(j)}_{2n+1}$ into account and remembering that C_{2n+1} is constructed of the $E^2C^{(j)}_{2n+1}$'s rather than the $C^{(j)}_{2n+1}$'s.

Lemma 10. *The function* $\texttt{Rank}_{\mathcal{K}}$ *operates in* $O(n^2)$ *steps.*

Proof: We note that $\texttt{Rank}_{\mathcal{K}}$ performs $O(n)$ operations before calling upon itself with an order reduced by one. It therefore operates in $O(n^2)$ time. ∎

Unranking permutations, i.e., the process of assigning to a given rank in $[0,M_{2n+1}-1]$ the corresponding permutation in the C_{2n+1}, might also be needed if one requires the logic cell to perform as more than a counter. We implement a function $\texttt{Unrank}_{\mathcal{K}}(n,k)$ which returns as output the k-ranked permutation in C_{2n+1}.

Naturally, all assumptions made above still hold. We will follow the same general method used for $\texttt{Rank}_{\mathcal{K}}$, i.e., we shall compute $j\in[2n-1]$ such that the given rank belongs to $\sigma\in E^2C^{(j)}_{2n+1}$, then adjust the rank to indicate the correct position in $C^{(j)}_{2n+1}$. It will then remain to compute the correct permutation in the "push-to-the-bottom" cycle using a recursive run, and shift it the required number of times.

Lemma 11. *The function* $\texttt{Unrank}_{\mathcal{K}}$ *operates in* $O(n^2)$ *steps as well.*

Proof: Follows exactly the same lines as our proof to Lemma 10. ∎

C. Bounds on $\mathcal{K}$-Snakes

We begin by noting a simple upper bound on the size of $\mathcal{K}$-snakes.

Lemma 12. *If C is an $(n, M, \mathcal{K})$-snake then*

1) $M \leqslant \frac{1}{2}|S_n|$.
2) $M = \frac{1}{2}|S_n|$ *if and only if for all* $\{\alpha, \beta\} \in E_n$ *it holds that* $\alpha \in C$ *or* $\beta \in C$.

Proof: Every $\alpha \in S_n$ has exactly $(n-1)$ neighbors in G_n. When we sum the edges for every vertex in G_n, each edge in E_n is counted precisely twice, hence

$$|E_n| = \frac{n-1}{2} \cdot |S_n| = \frac{n!(n-1)}{2}.$$

On the other hand, for every $\alpha, \beta \in C$ and $e_1, e_2 \in E_n$ such that $\alpha \in e_1$ and $\beta \in e_2$ clearly $e_1 \neq e_2$. It follows that there are no less than $M(n-1)$ distinct edges in E_n. Hence

$$M \leqslant \frac{1}{2}|S_n|.$$

Finally, we note that $M = \frac{1}{2}|S_n|$ iff $M(n-1) = |E_n|$, iff every edge in E_n contains a (unique) element of C. ∎

The codes we constructed in the previous section use only "push-to-the-top" operations on odd indices. We would now like to show that using even a single "push-to-the-top" operation on an even index can never result in a code attaining the bound of Lemma 12 with equality. We first require a simple lemma.

Lemma 13. *Let C be a $\mathcal{K}$-snake over S_n. If $\sigma, \sigma' \in C$ and there exists a path in G_n of odd length between them, then that path contains an edge both of whose endpoints are not in C.*

Proof: Consider such a path of odd length in G_n, connecting σ and σ'. Now color the vertices of C black, and those of $S_n \setminus C$ white. Since C is a $\mathcal{K}$-snake, no edge in E_n has both its ends colored black. In the path above the vertices cannot alternate in color since σ and σ' are colored black and the path has odd length. It follows that there is an edge in the path with both ends colored white, as claimed. ∎

A direct result of this lemma is presented in the following theorem:

Theorem 14. *If an $(n, M, \mathcal{K})$-snake C contains a "push-to-the-top" operation on an even index then $M < \frac{1}{2}|S_n|$.*

Proof: We note that a single adjacent transposition acting on a permutation flips the permutation's sign. Furthermore, a "push-to-the-top" operation $t_i \in T$, is equivalent to a sequence of $i-1$ adjacent transpositions moving the ith element of the permutation to the first coordinate. Thus, "push-to-the-top" operations on even indices flip the permutation's sign, while those on odd indices preserve it.

It readily follows that $\sigma, \sigma' \in S_n$ have different signs iff *every* path connecting them in G_n has odd length. Now, if $\sigma' = t_{2m}(\sigma)$ for some $2m \in [n]$, and both are in C, then they differ in sign and so by Lemma 12(b) and Lemma 13, $M < \frac{1}{2}|S_n|$. ∎

We now aim to show a tighter upper-bound on the size of $\mathcal{K}$-snakes employing a "push-to-the-top" operation on an even index.

Theorem 15. *If an $(n, M, \mathcal{K})$-snake C contains a "push-to-the-top" operation on an even index then*

$$M \leqslant \frac{1}{2}|S_n| - \frac{1}{n-1}\binom{\lfloor n/2 \rfloor - 1}{2}.$$

Proof: Let $C = (\sigma_1, \ldots, \sigma_M)$. We take $i \in [M-1]$ such that $\sigma_{i+1} = t_{2m}(\sigma_i)$, where $2m \in [n]$. For all $k, l \in \left[\lfloor \frac{n}{2} \rfloor - 1\right]$, $k < l$, we define

$$k' \triangleq \begin{cases} k & k < m \\ k+1 & k \geqslant m \end{cases} \qquad l' \triangleq \begin{cases} l & l < m \\ l+1 & l \geqslant m. \end{cases}$$

For each k' and l' we can now define the paths in G_n

$$\sigma_i \to \omega_1^{(k',l')} \to \omega_2^{(k',l')} \to \cdots \to \omega_{2m+2}^{(k',l')} \to \sigma_{i+1}$$

in the following recursive manner:

$$\omega_1^{(k',l')} \triangleq \sigma_i(2k'-1, 2k')$$
$$\omega_2^{(k',l')} \triangleq \omega_1^{(k',l')}(2l'-1, 2l'),$$

for all $j \in [2m-1]$ we define

$$\omega_{j+2}^{(k',l')} \triangleq \omega_{j+1}^{(k',l')}(2m-j, 2m-j+1),$$

and finally

$$\omega_{2m+2}^{(k',l')} \triangleq \omega_{2m+1}^{(k',l')}(2l'-1, 2l')$$
$$\omega_{2m+3}^{(k',l')} \triangleq \omega_{2m+2}^{(k',l')}(2k'-1, 2k') = \sigma_{i+1}.$$

We note that these $\binom{\lfloor n/2 \rfloor - 1}{2}$ paths are all of size $2m+3$, connecting σ_i and σ_{i+1}. Moreover, they only possibly ever intersect in the first or last two vertices. It follows from Lemma 13 that each contains an edge disjoint from C, and since we know each path's first and last edge does intersect C, there therefore exist at least $\binom{\lfloor n/2 \rfloor - 1}{2}$ distinct edges in G_n disjoint from C. We can now improve upon the upper-bound from Lemma 12 in the following way:

$$M(n-1) \leqslant \frac{n!(n-1)}{2} - \binom{\lfloor n/2 \rfloor - 1}{2}$$

and reordering gives us the claim. ∎

IV. The ℓ_∞-Metric and ℓ_∞-Snakes

The ℓ_∞-metric is induced on S_n by the embedding in $\mathbb{Z}^n$ implied by the vector notation. More precisely, for $\alpha, \beta \in S_n$ one defines

$$d_\infty(\alpha, \beta) = \max_{i \in [n]} |\alpha(i) - \beta(i)|.$$

We use the ℓ_∞-metric to model a different kind of noise-mechanism than that modeled by Kendall's τ-metric, namely spike noise. In this model, the rank of each memory cell is assumed to have been changed by a bounded amount (see [18]).

Error-correcting and -detecting codes in S_n for the ℓ_∞-metric are referred to in [18] as *limited-magnitude rank-modulation codes* (LMRM codes). In that paper, constructions of such codes achieving non-vanishing normalized distance and rate are presented. Moreover, bounds on the size of

optimal LMRM codes are proven. In particular, it has been shown [18, Th. 20] that if C is an $(n,M,2)$-LMRM then

$$M \leqslant \frac{n!}{2^{\lfloor n/2 \rfloor}}.$$

Using a simple translation to an extremal problem involving permanents of $(0,1)$-matrices (see [17]), this is also the best possible bound using the set-antiset method. For our needs, it follows that the size of every n-length ℓ_∞-snake is bounded by this term. We shall present a construction of ℓ_∞-snakes achieving this upper-bound by a factor of $\lfloor \frac{n}{2} \rfloor 2^{\lceil n/2 \rceil}$, which we will show achieves an asymptotic rate of 1.

A. Construction

In order to use the code constructions presented in [10], we first prove the following lemma.

Lemma 16. *Both constructions in [10, Th. 4,7], when applied recursively, yield complete cyclic n-RMGC's containing both "push-to-the-top" operations t_2 and t_n.*

Proof: The proposition was, while not fully stated, actually proven in [10, Th. 4].

For [10, Th. 7], we shall assume that the recursive process was applied to a length-$(n-1)$ Gray code satisfying these conditions (as is the case with the base example given in that article). The resulting code uses t_n by definition. Moreover, since the original code used t_{n-1}, the resulting code uses $t_{n-(n-1)+1} = t_2$. ∎

This lemma now allows for the construction of a basic building block which we will later use.

Lemma 17. *Let $\{a_j\}_{j=1}^n$, $n \geqslant 2$, be a set of integers of the same parity. Let*

$$\sigma = [x, a_1, a_2, \ldots, a_n, b_{n+2}, b_{n+3}, \ldots, b_m] \in S_m$$

be a permutation such that the parity of x differs from that of the elements of $\{a_j\}_{j=1}^n$. Then there exists a (non-cyclic) $(m, n+(n-1)!, \ell_\infty)$-snake starting with σ and ending with the permutation

$$t_2 t_{n+1}^n(\sigma) = [a_2, a_1, a_3, a_4, \ldots, a_n, x, b_{n+2}, b_{n+3}, \ldots, b_m].$$

Proof: Let $\sigma_0, \ldots, \sigma_{n+(n-1)!-1}$ denote the codewords of the claimed code, and denote by $t_{k_1}, \ldots, t_{k_{n+(n-1)!-1}}$ the list of transformations generating it.

We set $\sigma_0 = \sigma$. For all $i \in [n]$ we let $\sigma_i \triangleq t_{n+1}^i(\sigma)$, i.e., $t_{k_i} = t_{n+1}$. Quite clearly, any two of these $n+1$ permutations are at ℓ_∞-distance at least 2 apart, since the a_j's share parity.

Now, by Lemma 16 there exists a complete cyclic $(n-1)$-RMGC starting with σ_n, with its last operation being t_2. We therefore let $t_{k_{n+i}}$ for $i \in [(n-1)!]$ represent that code, hence $t_{k_{n+(n-1)!}} = t_2$ and $\sigma_{n+(n-1)!} = \sigma_n$ (we then, obviously, omit the last transformation as well as the repeated codeword $\sigma_{n+(n-1)!}$). These $(n-1)!$ permutations, $\sigma_n, \ldots, \sigma_{n+(n-1)!-1}$, also represent an ℓ_∞-snake, for the same reason.

Finally, take $0 \leqslant k < n$ and $0 \leqslant l < (n-1)!$, and observe σ_k and σ_{n+l}. Suppose $d_\infty(\sigma_k, \sigma_{n+l}) \leqslant 1$. Then in particular $|a_{n-k} - x| = 1$. Moreover, if $k = n-1$ then $|x - a_n| = 1$, but then a_n's position in σ_k correlates to one of $\{a_j\}_{j=1}^{n-1}$ in σ_{n+l}, in contradiction. Therefore $k \leqslant n-2$, but then a_n's position in σ_{n+l} (nth from left) correlates to that of a_{n-k-1} in σ_k, where $1 \leqslant n-k-1 \leqslant n-1$, again in contradiction. This concludes our proof. ∎

Having this building block in hand, we continue to describe a construction of a cyclic ℓ_∞-snake. The construction follows by dividing the ranks in a length-n permutation into even and odd elements, and covering permutations on each half separately.

Theorem 18. *For all $4 \leqslant n \in \mathbb{N}$ there exists an (n, M, ℓ_∞)-snake of size*

$$M = \left\lceil \frac{n}{2} \right\rceil ! \left(\left\lfloor \frac{n}{2} \right\rfloor + \left(\left\lfloor \frac{n}{2} \right\rfloor - 1 \right)! \right).$$

Proof: To simplify notations, we start by noting that $[n]$ has $p \triangleq \lceil \frac{n}{2} \rceil$ odd elements and $q \triangleq \lfloor \frac{n}{2} \rfloor$ even ones. We shall use that notation throughout this proof.

Using [10, Th. 4,7] we take a complete cyclic p-RMGC using the operations

$$t_{\alpha(1)}, t_{\alpha(2)}, \ldots, t_{\alpha(p!)}.$$

Moreover, we use Lemma 17 to come by a (q, M_q, ℓ_∞)-snake of size $M_q = q + (q-1)!$ given by the operations

$$t_{\beta(1)}, t_{\beta(2)}, \ldots, t_{\beta(q+(q-1)!-1)}.$$

As the origin for the code we construct we use

$$\sigma_0 \triangleq [1, 2, 4, \ldots, 2q, 3, \ldots, 2p-1].$$

For all $i \in [p!]$ and $j \in [q + (q-1)! - 1]$ we define sequence of transformations generation the code as

$$t_{k_{(i-1)(q+(q-1)!)+j}} \triangleq t_{\beta(j)}$$
$$t_{k_{i(q+(q-1)!)}} \triangleq t_{\alpha(i)+q+1},$$

and where, naturally, the codewords satisfy $\sigma_i = t_{k_i}(\sigma_{i-1})$.

We start by noting that, for all $i \in [p!]$, the permutation $\sigma_{(i-1)(q+(q-1)!)}$ satisfies the requirements of Lemma 17 as a simple matter of induction. It follows that for all $i \in [p!]$ the permutations

$$\left\{ \sigma_{(i-1)(q+(q-1)!)+1}, \sigma_{(i-1)(q+(q-1)!)+2}, \ldots, \sigma_{i(q+(q-1)!)-1} \right\}$$

are at ℓ_∞-distance of at least 2 apart.

Furthermore, for $i, i' \in [p!]$, $i < i'$, since the code generated by $t_{\alpha(1)}, t_{\alpha(2)}, \ldots, t_{\alpha(p!)}$ is indeed a Gray code, we are assured that for all $0 \leqslant j, j' \leqslant q + (q-1)! - 1$ the last $p-1$ elements of both $\sigma_{(i-1)(q+(q-1)!)+j}$ and $\sigma_{(i'-1)(q+(q-1)!)+j'}$ are all odd and represent two distinct permutations, hence

$$d_\infty \left(\sigma_{(i-1)(q+(q-1)!)+j}, \sigma_{(i'-1)(q+(q-1)!)+j'} \right) \geqslant 2.$$

Finally, we note that

$$t_{\alpha(p!)} \left(\sigma_{p!(q+(q-1)!)-1} \right) = \sigma_0,$$

since the code provided by $t_{\alpha(1)}, t_{\alpha(2)}, \ldots, t_{\alpha(p!)}$ is cyclic and $o(t_2) = 2$ divides $p!$. ∎

We note that by switching the roles of odd and even numbers in Theorem 18 we can construct an (n, M, ℓ_∞)-snake of size

$$M = \left\lfloor \frac{n}{2} \right\rfloor ! \left(\left\lceil \frac{n}{2} \right\rceil + \left(\left\lceil \frac{n}{2} \right\rceil - 1 \right)! \right).$$

However, the resulting code is strictly smaller for odd n.

Theorem 19. *The ℓ_∞-snakes constructed in Theorem 18 have an asymptotically-optimal rate.*

Proof: Let C_n denote the ℓ_∞-snake of length n constructed by Theorem 18. Using the crude

$$\left(\frac{n}{e}\right)^n \leqslant n! \leqslant n^n$$

the proof is a matter of simple calculation:

$$\begin{aligned}\lim_{n\to\infty} R(C_n) &= \lim_{n\to\infty} \frac{\log_2\left(\lceil\frac{n}{2}\rceil!\left(\lfloor\frac{n}{2}\rfloor + \left(\lfloor\frac{n}{2}\rfloor - 1\right)!\right)\right)}{\log_2(n!)} \\ &\geqslant \lim_{n\to\infty} \frac{2\log_2\left(\left(\lfloor\frac{n}{2}\rfloor - 1\right)!\right)}{\log_2(n!)} \\ &\geqslant \lim_{n\to\infty} \frac{(n-4)\log_2\left(\frac{n-4}{2e}\right)}{n\log_2 n} = 1.\end{aligned}$$

■

B. Successor Calculation and Ranking Algorithms

Finding the correct "push-to-the-top" operation to propagate a given permutation to the following one is naturally dependent upon one's ability to do the same with the $\lceil\frac{n}{2}\rceil$- and $\left(\lfloor\frac{n}{2}\rfloor - 1\right)$-RMGC's used in our construction. We therefore assume to have the function $\texttt{Succ}\left([a_1, a_2, \ldots, a_n]\right)$ which accepts as input a permutation $[a_1, a_2, \ldots, a_n] \in S_n$ and returns the correct transformation used in the codes we used. Furthermore, we assume to have the function $\texttt{Rn}\left([a_1, a_2, \ldots, a_n]\right)$ which returns the respective rank of the input permutation in that code, where the identity permutation is assumed to have rank zero. Finally, we shall use an auxiliary function $\texttt{sw} : S_n \to S_n$ defined by $\texttt{sw}(\sigma) \triangleq (1,2) \circ \sigma$ (which naturally operates in $O(n)$ steps).

The function $\texttt{Successor}_\infty\left([a_1, \ldots, a_n]\right)$ then returns as output the index i of the required transformation t_i to produce the subsequent permutation in the code from $[a_1, \ldots, a_n]$. It operates by considering the following cases: in each block of Lemma 17 one computes the proper index by propagating the leading element of odd rank as long as that is needed, then applying Succ to the permutation on the elements of even ranks (where one distinguishes between blocks in which $2, 4$ were switched). Only the last permutation of each block calls for applying Succ to the permutation on the elements of odd ranks.

Lemma 20. *If the functions* Succ, Rn *operate in L_n, M_n steps respectively in the average case, then* $\texttt{Successor}_\infty$ *has an average run-time of $O\left(n + L_{q-1} + M_p\right)$.*

Proof: We partition our proof by return cases. $\texttt{Successor}_\infty$ exits at line 3 in precisely $\frac{q}{q+(q-1)!}$ of cases, in which case it returns within a fixed number of operations.

It exits at lines 6, 9 in $\frac{1}{q+(q-1)!}$ of cases, in which case it operates in at most (depending on the data structures in use) $O(n) + M_p + L_p$ steps in the average case.

Finally, $\texttt{Successor}_\infty$ returns from lines 7, 10 in $\frac{(q-1)!-1}{q+(q-1)!}$ of cases, after performing $O(n) + M_p + L_{q-1}$ steps.

Function $\texttt{Successor}_\infty\left([a_1, \ldots, a_n]\right)$

input : A permutation $[a_1, a_2, \ldots, a_n]$
output : $i \in \{2, 3, \ldots, n\}$ that determines the transition t_i to the next permutation in the ℓ_∞-snake from Theorem 18

1 $q \leftarrow \lfloor\frac{n}{2}\rfloor$; $p \leftarrow \lceil\frac{n}{2}\rceil$
2 **if** $a_{q+1} \equiv 0 \pmod 2$ **then**
3 **return** $q + 1$
4 **if** $\texttt{Rn}\left(\left[\frac{a_{q+1}+1}{2}, \ldots, \frac{a_n+1}{2}\right]\right) \equiv 0 \pmod 2$ **then**
5 **if** $[a_1, \ldots, a_q] = [4, 2, 6, \ldots, 2q]$ **then**
6 **return** $q + \texttt{Succ}\left(\left[\frac{a_{q+1}+1}{2}, \ldots, \frac{a_n+1}{2}\right]\right)$
7 **return** $\texttt{Succ}\left(\left[\frac{a_1}{2}, \ldots, \frac{a_q}{2}\right]\right)$
8 **if** $[a_1, \ldots, a_q] = [2, 4, \ldots, 2q]$ **then**
9 **return** $q + \texttt{Succ}\left(\left[\frac{a_{q+1}+1}{2}, \ldots, \frac{a_n+1}{2}\right]\right)$
10 **return** $\texttt{Succ}\left(\texttt{sw}\left(\left[\frac{a_1}{2}, \ldots, \frac{a_{q-1}}{2}\right]\right)\right)$

Function $\texttt{Rank}_\infty\left([a_1, \ldots, a_n]\right)$

input : A permutation $[a_1, a_2, \ldots, a_n]$ in the ℓ_∞-snake from Theorem 18
output : $k \in \mathbb{N}$ that represents the given permutation's rank in the code

1 $q \leftarrow \lfloor\frac{n}{2}\rfloor$; $p \leftarrow \lceil\frac{n}{2}\rceil$
2 **if** $a_{q+1} \equiv 0 \pmod 2$ **then**
3 $i \leftarrow \min\{j \in [n] \mid a_j \not\equiv 0 \pmod 2\}$
4 **return** $i - 1 + (q + (q-1)!) \cdot \texttt{Rn}\left(\left[\frac{a_i+1}{2}, \frac{a_{q+2}+1}{2}, \ldots, \frac{a_n+1}{2}\right]\right)$
5 $R \leftarrow \texttt{Rn}\left(\left[\frac{a_{q+1}+1}{2}, \ldots, \frac{a_n+1}{2}\right]\right)$
6 **if** $R \equiv 0 \pmod 2$ **then**
7 **return** $q + (q + (q-1)!) \cdot R + \texttt{Rn}\left(\left[\frac{a_1}{2}, \ldots, \frac{a_{q-1}}{2}\right]\right)$
8 **return** $q + (q + (q-1)!) \cdot R + \texttt{Rn}\left(\texttt{sw}\left(\left[\frac{a_1}{2}, \ldots, \frac{a_{q-1}}{2}\right]\right)\right)$

In every sensible implementation of Succ (i.e., where we assume $\frac{L_p - L_{q-1}}{q+(q-1)!} \to 0$) we then have an amortized run-time of $O\left(n + L_{q-1} + M_p\right)$. ■

We now note that by [10, Th. 7,10] we may assume Succ to operate in $O(1)$ steps in the average case, and by [10, Part III-C] (which also relies on [14]) we assume Rn runs in $O(n)$ steps, yielding an average run-time of $O(n)$ for $\texttt{Successor}_\infty$.

We shall also present the function $\texttt{Rank}_\infty(n, [a_1, \ldots, a_n])$ that, given a permutation in the ℓ_∞-snake presented in part IV-A, returns that permutation's rank in the code. This function uses the function Rn discussed above as well, and works by considering the same cases discussed above.

Lemma 21. *If the function* Rn *operates in M_n steps, then* $\texttt{Rank}_\infty$ *has a run-time of $O(n + M_p)$ (in the average or worst case respectively).*

Proof: We partition our proof by return condition once more. If the program exits from 4 then it performed $O(q) + M_p$ steps.

If it exits from 7 or 8 then it performed $O(1) + M_p + M_{q-1}$ steps. ■

Again, by results discussed above, we note that $\texttt{Rank}_\infty$ runs in $O(n)$ steps in the average case.

It may prove important to identify the permutation associated with a specific rank in our code. For that purpose we

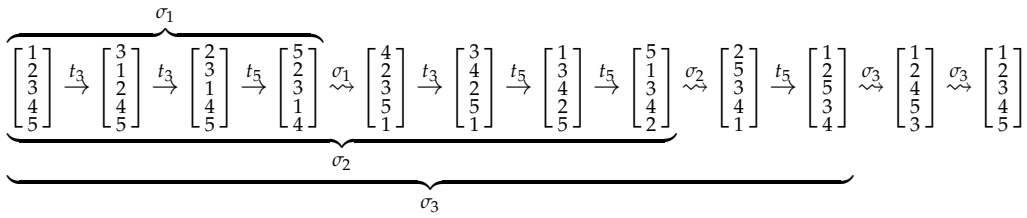

Figure 2. A $(5,57,\mathcal{K})$-snake generated by a computer search. Squiggly arrows stand for a repetition of the transitions defined by the braces.

Function $\texttt{Unrank}_\infty(n,k)$

input : $4 \leqslant n \in \mathbb{N}$; rank $k \in \mathbb{N}$
output : The permutation $[a_1, a_2, \ldots, a_n]$ which is kth in the (n, M, ℓ_∞)-snake from Theorem 18

$q \leftarrow \lfloor \frac{n}{2} \rfloor$; $p \leftarrow \lceil \frac{n}{2} \rceil$
$R \leftarrow \left\lfloor \frac{k}{q+(q-1)!} \right\rfloor$; $r \leftarrow (k \bmod (q+(q-1)!))$
$[b_1, \ldots, b_p] \leftarrow \texttt{UnR}(p, R)$
if $r \geqslant q$ **then**
  $[a_1, \ldots, a_{q-1}] \leftarrow \texttt{UnR}(q-1, r-q)$
  if $R \equiv 1 \pmod 2$ **then**
    $[a_1, \ldots, a_{q-1}] \leftarrow \texttt{sw}([a_1, \ldots, a_{q-1}])$
  return $[2a_1, \ldots, 2a_{q-1}, 2q, 2b_1 - 1, \ldots, 2b_p - 1]$
if $R \equiv 0 \pmod 2$ **then**
  return $[2, 4, 6, \ldots, 2r, 2b_1 - 1, 2(r+1), \ldots, 2q, 2b_2 - 1, \ldots, 2b_p - 1]$
return $[4, 2, 6, \ldots, 2r, 2b_1 - 1, 2(r+1), \ldots, 2q, 2b_2 - 1, \ldots, 2b_p - 1]$

implement the function $\texttt{Unrank}_\infty(n,k)$, accepting as input the length of the code and a specific rank and returning the implied permutation. We will assume the existence of a similar function UnR for the construction used in part IV-A, where again we assume the unit permutation to have rank zero.

Once more, our implementation and estimate of $\texttt{Unrank}_\infty$'s run-time relies heavily on that of its auxiliary functions.

Lemma 22. *If the function* UnR *operates in* N_n *steps, then* $\texttt{Unrank}_\infty$ *runs in* $O(n+N_p)$ *steps.*

Proof: One notes that the only operations in $\texttt{Unrank}_\infty$ that take more than a fixed number of steps are calls for sw (taking $O(n)$), calls for UnR, and, depending on the data structures in use, concatenation of indices (at most $O(n)$ as well). The claim follows. ∎

Again, it shall be noted that, relaying on Lemma 16 and [10, Part III-C], $\texttt{Unrank}_\infty$ can be performed in $O(n^2)$ operations.

V. Conclusion

In this paper we explored rank-modulation snake-in-the-box codes under both Kendall's τ-metric and the ℓ_∞-metric. In both cases we presented a construction yielding codes with asymptotically-optimal rates, and implemented auxiliary functions for the production of the successor permutation, as well as ranking and unranking for permutations in such codes. We also proved upper-bounds on the size of $\mathcal{K}$-snakes.

However, it is not presently known whether the upper-bounds presented and referenced in this paper are achievable. A computer search for *cyclic* codes, performed on S_5, yielded $(5,M,\mathcal{K})$-snakes of maximal size $M = 57$ (for comparison, the construction from Theorem 6 yields a $(5,45,\mathcal{K})$-snake).

n	Defining Transitions
4	55
5	0212206063
6	010204410222042124446130162347

Figure 3. $(4,6,\ell_\infty)$-, $(5,30,\ell_\infty)$- and $(6,90,\ell_\infty)$-snakes generated by a computer search. All codes represented by a sequence of "push-to-the-top" operations, applied in order to the identity permutation, where zeroes stand for t_n's and ones for t_{n-1}'s. The binary strings are given in octal notation and should be read from left to right.

While an abundance of such codes were found (well over 500 nonequivalent codes), they all were in fact codes over A_5. For completeness, we present one of those codes in Fig. 2.

Searches of a higher order appear to be infeasible, but we include one more peculiar result: every maximal code we tested skipped 3 permutations who all agree on $4,5$, i.e., it skipped a coset of S_3. While we have no optimal codes of a higher order to test this phenomenon on, the codes generated by Theorem 6 of lengths 7 and 9 display it as well - several cosets of S_5 and S_7 were absent, respectively.

It shall be noted that a complete (but not cyclic) $(5,60,\mathcal{K})$-snake over A_5 can easily be constructed from each cyclic code we tested by generating the skipped coset of S_3 with two t_3 operations, followed by a t_5 operation and the given code, in order. However, we do not currently know whether $(2n+1, \frac{(2n+1)!}{2}, \mathcal{K})$-snakes over A_{2n+1} exist for every length.

These results, along with the bounds we showed in Lemmas 15 and 12 give rise to the following conjecture: For all $n \in \mathbb{N}$ a $\mathcal{K}$-snake exists over A_n whose size is no less than that of every $\mathcal{K}$-snake over S_n.

In addition, searches done in a computer for ℓ_∞-snakes for lengths $4,5,6$ returned codes of size $6,30,90$ respectively, suggesting that perhaps the upper-bound of [18, Th. 20] is achievable. Moreover, in these cases we were able to find codes generated only by "push-to-the-top" operations on the last two indices. A code for each length is presented in Fig. 3 in binary representation (conveniently written in octal notation), where zeroes stand for t_n's and ones for t_{n-1}'s. Searches for higher lengths again seem infeasible.

References

[1] H. L. Abbot and M. Katchalski, "On the construction of snake in the box codes," *Utilitas Math.*, vol. 40, pp. 97–116, 1991.
[2] A. Barg and A. Mazumdar, "Codes in permutations and error correction for rank modulation," *IEEE Trans. on Inform. Theory*, vol. 56, no. 7, pp. 3158–3165, Jul. 2010.
[3] J. Brewer and M. Gill, *Nonvolatile Memory Technologies with Emphasis on Flash*. Wiley-IEEE Press, 2008.

[4] F. Chierichetti, H. Finucane, Z. Liu, and M. Mitzenmacher, "Designing floating codes for expected performance," *IEEE Trans. on Inform. Theory*, vol. 56, no. 3, pp. 968–978, Mar. 2010.

[5] M. Deza and H. Huang, "Metrics on permutations, a survey," *J. Comb. Inf. Sys. Sci.*, vol. 23, pp. 173–185, 1998.

[6] E. En Gad, M. Langberg, M. Schwartz, and J. Bruck, "On a construction for constant-weight gray codes for local rank modulation," in *Proceedings of the 2010 IEEE 26-th Convention of Electrical and Electronic Engineers in Israel (IEEEI2010), Eilat, Israel*, Nov. 2010, p. 996.

[7] F. Gray, "Pulse code communication," March 1953, U.S. Patent 2632058.

[8] A. Jiang, V. Bohossian, and J. Bruck, "Rewriting codes for joint information storage in flash memories," *IEEE Trans. on Inform. Theory*, vol. 56, no. 10, pp. 5300–5313, Oct. 2010.

[9] A. Jiang, M. Langberg, M. Schwartz, and J. Bruck, "Universal rewriting in constrained memories," in *Proceedings of the 2009 IEEE International Symposium on Information Theory (ISIT2009), Seoul, Korea*, Jun. 2009, pp. 1219–1223.

[10] A. Jiang, R. Mateescu, M. Schwartz, and J. Bruck, "Rank modulation for flash memories," *IEEE Trans. on Inform. Theory*, vol. 55, no. 6, pp. 2659–2673, Jun. 2009.

[11] A. Jiang, M. Schwartz, and J. Bruck, "Correcting charge-constrained errors in the rank-modulation scheme," *IEEE Trans. on Inform. Theory*, vol. 56, no. 5, pp. 2112–2120, May 2010.

[12] M. Kendall and J. D. Gibbons, *Rank Correlation Methods*. Oxford University Press, NY, 1990.

[13] T. Kløve, T.-T. Lin, S.-C. Tsai, and W.-G. Tzeng, "Permutation arrays under the Chebyshev distance," *IEEE Trans. on Inform. Theory*, vol. 56, no. 6, pp. 2611–2617, Jun. 2010.

[14] M. Mares and M. Straka, "Linear-time ranking of permutations," *Algorithms-ESA*, pp. 187–193, 2007.

[15] C. D. Savage, "A survey of combinatorial Gray codes," *SIAM Rev.*, vol. 39, no. 4, pp. 605–629, Dec. 1997.

[16] M. Schwartz, "Constant-weight Gray codes for local rank modulation," in *Proceedings of the 2010 IEEE International Symposium on Information Theory (ISIT2010), Austin, TX, U.S.A.*, Jun. 2010, pp. 869–873.

[17] M. Schwartz and I. Tamo, "Optimal permutation anticodes with the infinity norm via permanents of $(0,1)$-matrices," *J. Combin. Theory Ser. A*, vol. 118, pp. 1761–1774, 2011.

[18] I. Tamo and M. Schwartz, "Correcting limited-magnitude errors in the rank-modulation scheme," *IEEE Trans. on Inform. Theory*, vol. 56, no. 6, pp. 2551–2560, Jun. 2010.

[19] Z. Wang and J. Bruck, "Partial rank modulation for flash memories," in *Proceedings of the 2010 IEEE International Symposium on Information Theory (ISIT2010), Austin, TX, U.S.A.*, Jun. 2010, pp. 864–868.

[20] E. Yaakobi, A. Vardy, P. H. Siegel, and J. K. Wolf, "Multidimensional flash codes," in *Proc. of the Annual Allerton Conference*, 2008.