Progressive decoding of DNA-stored JPEG data with on-the-fly error correction

Ibrahim Nawaz[1¤*], Parv Agarwal[1], Thomas Heinis[1‡]

1 DNA Storage Research, Imperial College London, United Kingdom

‡These authors also contributed equally to this work.
¤Current Address: Medical Vision Group, Imperial College London, United Kingdom
* ibrahim.nawaz22@imperial.ac.uk

Abstract

DNA storage is a developing field that uses DNA to archive digital data, owing to its superior information density and stability compared to conventional storage mediums. Although DNA storage techniques have been performed on a significant scale, challenges arise from the synthesis and sequencing of data-encoded oligonucleotides. Synthesis of DNA introduces significant noise into the process. Consequently, high-read-quality sequencers are often required, making the process expensive and lack scalability. Recent DNA storage methods use error correction codes to encode data with resilience to noise, at a cost of additional redundancy and decoding complexity. Given such constraints and challenges, the main objective we seek to deliver is a time- and storage-efficient image coding strategy. We introduce a novel DNA-based progressive JPEG decoder with on-the-fly error-correcting & rendering capability. This system can progressively decode an image while also correcting for errors as they occur. It modifies standard JPEG encoding methods to store data in localized chunks and uses adapted Raptor error-correction codes to improve the speed and quality of partial decoding. We optimized and evaluated the method under varying levels of simulated Oxford Nanopore Technologies sequencing errors, as well as show how different parts of the pipeline improve real-time decoding capability. We tested the proposed pipeline with synthesized oligonucleotides and analyzed recovery profiles. Here, we present the first progressive image decoding schema, aimed at realizing real-time rendering of DNA-stored images.

Introduction

Data production across the globe is growing exponentially. The International Data Corporation (IDC) predicted that globally data would increase from 45 to 175 zettabytes between 2019 and 2025 [1]. Moreover, it is expected that the storage supply will not be able to meet such a demand. By 2030, storage shortages will account for two-thirds of demand [2]. Therefore, there is a growing need to find more sustainable and efficient data storage mediums. DNA has been proposed as a suitable candidate, especially for archival purposes. DNA is extremely information dense: a gram can theoretically encode 455 exabytes of data [3]. Solid-state hard drives and magnetic tape are far less optimal. DNA is also very stable, lasting hundreds of years at room temperature or potentially millions of years in permafrost [4]. Data stored on hard drives and tape can only last for decades. DNA is also easily replicable with very low energy operational requirements. Proof-of-concept works have been conducted that

encode digital data in DNA for use in storage [5, 6]. The field has since progressed significantly, from encoding a 23-byte message [7] to 200.2MB of unstructured data [8].

Storing data in DNA involves encoding information into sequences of oligonucleotides (oligos), followed by synthesis and physical storage. Retrieving the information involves sequencing the oligos and decoding them back into digital data. The main challenges to DNA storage arise from synthesis and sequencing of the oligos: alongside increased cost, they both introduce significant noise into the workflow. Noise in DNA storage usually occurs as insertion-deletion-substitution (IDS) errors (where a base is incorrectly added, removed, or replaced, respectively) or erasures (where entire oligos are dropped) [9]. Synthesis, polymerase chain reaction (PCR) amplification, and storage conditions can also cause oligos to decay at different rates [10]. Previous works [11, 12] heuristically defined some key biochemical constraints to minimize such errors when encoding and decoding: 1) Strands below 200 nucleotide (nt) length; 2) No homopolymers (repeats of a base, e.g. "TTT"); 3) GC content between 40-60%; 4) No repeated patterns in an oligo (repeated sequence of bases, e.g. "ACTACTACT"). Synthesized oligos are also stored in a geometrically unstructured fashion, which necessitates the need for extra indexing information within the oligo [12] and increases erasures when sub-sampling from the oligo pool for sequencing & decoding [13].

Prior works have shown that errors can be corrected to recover data if appropriate encoding schemes are used. Error-correcting codes introduce redundant information when encoding, allowing for erroneous data to be recovered [14] at a cost of increased storage size. Simple techniques such as parity checking [12] and overlapping segments [11] resulted in unrecoverable gaps due to error, even after using expensive high-fidelity synthesis and high-accuracy Illumina sequencing. Other studies deployed Reed-Solomon (RS) codes [15–17] which achieved full data recovery upon decoding, though were still far from realizing channel capacity. They also suffered when scaling to larger files due to variations in erasures of different oligos. Such challenges can be solved with more complex encoding schema & higher sequencing coverage. Fountain codes, like Luby Transform (LT) codes [18], are near-optimal rateless erasure codes and are able to approach the DNA storage channel capacity while providing greater robustness against erasures [19]. Modern fountain codes like Raptor [20] & Online codes [21] recently showed significantly better performance when compared with LT coding [22]. While previous literature greatly improved encoding efficiencies, they lacked scalability due to use of expensive high performance synthesis & sequencing techniques.

More scalable methods have been explored: for example using photolithographic synthesis [23] and Oxford Nanopore Technologies (Nanopore) sequencing [24]. Such techniques are more error-prone, resulting in more complex coding strategies and less efficient codes. For example, performing clustering [23] or iterative alignment [25] prior to decoding can leverage high coverage to reduce errors. However, these methods are also slow.

Our work revolves around image encoding. Image-specific encoding methods for DNA storage have also been explored. Previous works have encoded quantized wavelet coefficients of an image into DNA with a controllable compression & oligo length value [26]; implemented a JPEG-style encoding scheme [27], leveraging ternary Huffman tables [12] to replace the Huffman encoding used in JPEG. JPEG-DNA [28], currently in development, implements Raptor encoding [22] of a JPEG image. Due to inherent image compression, such methods require all information to be correctly decoded before rendering is possible. Progressive JPEGs [29] can render an image at reduced resolution before retrieving all image coefficients, and have not been explored in DNA storage before. Although developed primarily for rendering images in poor network conditions, this feature allows us to address the slow nature of decoding. Progressive JPEGs split up an image into separate scans containing different levels of detail: spectral

selection [30] initially encodes lower frequency coefficients from discrete cosine transform (DCT) and iteratively encodes increasing frequency coefficients in subsequent scans. JPEGs can also use restart intervals (RI) to provide error resilience to the internal state of the JPEG decoder, and have not been explored in DNA storage before. In file system fragmentation, RIs have been used to reduce file fragmentation effects when recovering corrupted JPEG images [31].

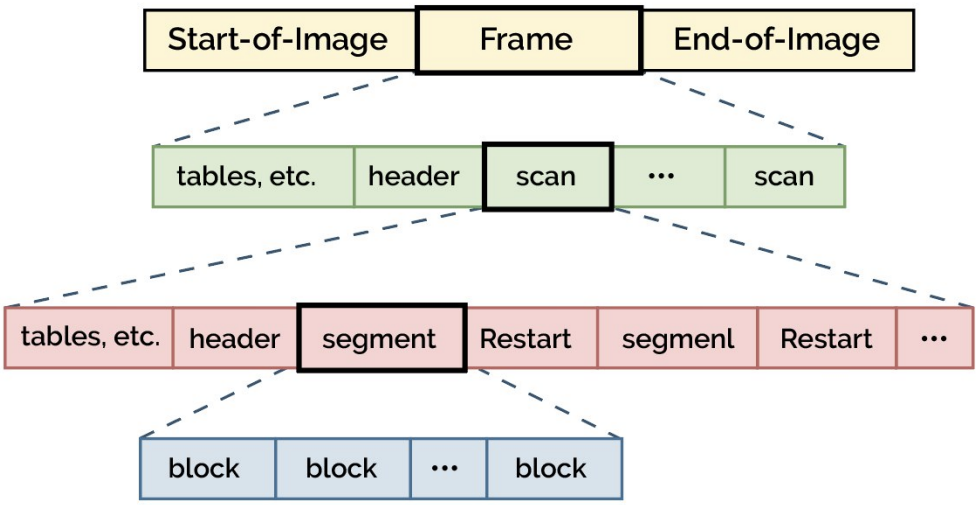
Taken together, these results inspired us to seek a fast & scalable coding strategy with on-the-fly rendering and partial decoding. To reduce the time overhead during sequencing & decoding, we wish to utilize the progressive capabilities of JPEG to iteratively render an image as oligos are being sequenced. Further recovery will add further detail to our rendered image. Previous works have applied JPEG-specific encoding methods to DNA storage [32]. However, to the author's best knowledge, this is the first work to implement progressive JPEG rendering with error correction into a DNA storage workflow.

Materials and methods

Custom JPEG encoding

A 680x453 uncompressed RGB bitmap image, S1 Fig, was encoded into a progressive JPEG using the libjpeg-turbo library (v3.1.0, https://github.com/libjpeg-turbo/). We follow the typical progressive JPEG encoding schema which includes an initial scan for all DC coefficients, followed by three successive AC coefficient scans for the luminance (Y) and chrominance (Cb, Cr) components [33]. The quality parameter was fixed at 75 (default) and chroma subsampling was disabled. During JPEG encoding, coefficients are compressed into variable-length Huffman codes and undergo bit stuffing during encoding into the entropy-coded stream (ECS). In typical JPEGs, RI markers can be inserted into the ECS after a user-defined number of image blocks. This user-defined number is stored in the header of a JPEG file, while the markers themselves are inserted directly into the ECS, as shown in Fig 1.

Fig 1. JPEG bitstream. There are multiple scans - i.e. progressive JPEG. Each scan contains segments (ECS) with 'Restart' markers (RIs) inserted at regular intervals.



These markers have a 2-byte size and range from `0xFFD0` to `0xFFD7`. The markers are inserted in sequential order from `0xD0` to `0xD7`, before restarting back to `0xD0`. Fragmentation during DNA decoding creates gaps in the ECS, necessitating the need to store global location information about RIs. Thus, RI markers were translated into 'global' RI markers to act simultaneously as restart intervals & as image block indices, by storing a global RI count in base-8. For example, the 10th RI marker in a scan would be converted from [`0xFF`, `0xD1`] (2 bytes) to [`0xFF`, `0xD1`, `0xFF`, `0xD1`] (4 bytes). Eq (1) shows how the RI block step size, RI_{step}, was calculated dynamically, based on the required Raptor chunk size:

$$c_n = \lceil x \cdot RI_{\text{step}} \rceil + 2\left(\frac{\sum_0^{n-1} \lfloor \log_8 n \rfloor + 1}{n-1}\right) \tag{1}$$

where $n = \lfloor \frac{y}{RI_{\text{step}}} \rfloor$, x = average block size, y = total number of blocks. RI_{step} was iterated until $c_{\text{n}} < s \cdot c$ (scaling factor · required chunk size) and was returned. s was optimized heuristically. RI_{step} itself was optimized for only the first DC scan, to minimize overhead & maximize progressive decoding when rendering. Therefore, a compressed DNA-adapted JPEG image (.jpg) was generated from the original uncompressed bitmap image (.bmp), with 4 scans and an optimized restart marker step size, RI_{step}.

A full JPEG decoder was also implemented, with extra functionality to translate the global base-8 RI markers into block index information. The header data, accounting for ~5% of information, was stored off-DNA to simplify the workflow. Partial decoding involves recovering random chunks of the bitstream from random oligos. Therefore, the JPEG decoder was also adapted to validate a scan before attempting to decode it. A scan was determined as valid if it contained any span of bits starting from a RI (i.e. `0xFFDX`) and including any coefficients. These spans could be decoded independently from each other due to Huffman's prefix coding.

Adapted Raptor coding

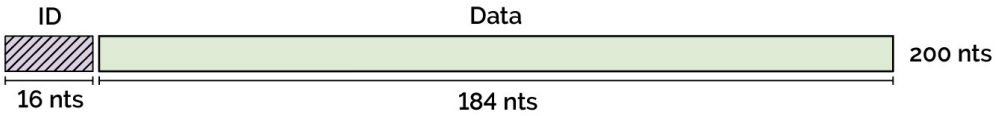
Reading DNA-stored data involves selecting a subset of encoded oligos to sequence from a geometrically unstructured global pool. Therefore, it is highly unlikely that all data will be recoverable initially. This resembles an erasure. Therefore, a modern fountain code was selected. The Raptor encoding schema [34] used in JPEG-DNA (https://github.com/umr-ds/NOREC4DNA, [22]) was adapted to generate oligos from the compressed DNA-adapted JPEG image described above. The bitstream was split up into k equally sized chunks, according to a user-selected chunk size, c; m auxiliary chunks were generated using Gray & low-density parity-check precoding [35, 36]; a Mersenne Twister algorithm [37] was used to randomly select d chunks from the $(k+m)$ original chunks. The algorithm uses the seed of a pseudorandom number generator, which models the Raptor (Robust Soliton) distribution, to sample the degree, d, of the packet. The packet was generated by XORing the d selected chunks together and prepending the seed ID. Fig 1 shows the structure of a candidate oligo packet, after bit-to-DNA translation at the entropy limit of 2-bits-per-nt.

Given a specified chunk size, c the oligo length can be calculated by Eq (2):

$$\text{oligo length} = 4c + 20 \tag{2}$$

The final step enforced previously detailed biochemical constraints. Fountain codes like Raptor are rateless, meaning they can generate an unlimited number of encoded packets. Therefore, if a candidate oligo was invalidated by the constraints, that packet was dropped, and a new one was generated. This process was repeated until $k(1+\theta)$ valid

Fig 2. Raptor-encoded oligo packet. A 200-nt oligo generated from a 50-byte Raptor-coded packet, containing 4 bytes ID & 46 bytes data.



packets were generated, where the user-defined overhead, θ, was set to the default value of 0.4. This value was fine-tuned based on the channel characteristics. Finally, the array of valid encoded packets was saved into a FASTA file.

The Raptor decoder utilized Gaussian elimination with partial pivoting (GEPP), instead of belief propagation [38], owing to its deterministic nature, numerical stability, and predictable performance. We express performance of partial decoding as the normalized area under the recovery curve (AUC_{norm}), given by Eq (3):

$$AUC_{\text{norm}} = \frac{AUC}{\text{total number of packets received}} \tag{3}$$

where the area under the curve (AUC) is calculated using the trapezium rule.

Traditional progressive JPEGs render scans in a sequential nature, where lower frequencies are rendered first. However, Raptor decoding is random – there is no control over which chunks in the ECS are retrieved first. Therefore, we introduce the first application of prioritized LT codes within the context of DNA storage [39]. Low-degree (1 or 2) packets should have a higher likelihood to contain high-priority chunks during encoding. This allows them to be recovered faster during Raptor decoding, with negligible overhead. We assigned chunks containing the first DC scan as high-priority.

Finally, the Mersenne Twister was adjusted in order to evaluate the effect of increasing degree-1 packets on partial decoding performance & overhead. A pseudorandom number was selected, $p \in [0, 1]$. If this value was below the degree-1 threshold (i.e. $p < p_{\text{thr}}$), d was set to 1. Otherwise, the original Raptor distribution was used. We calculate the final overhead, ε, as shown by Eq (4):

$$\varepsilon = \frac{\text{total number of packets received}}{k} - 1 \tag{4}$$

Nanopore sequencing & clustering

Nanopore sequencing was simulated with Badread [40] (v0.4.1, https://github.com/rrwick/Badread) integrated with an agglomerative clustering algorithm developed internally. Similar to previous clustering-based decoders [8, 23], our clustering algorithm loosened the constraints on invalidity & coverage, by aggregating noisy Badreads of each sequence into clusters, using the Levenshtein distance as a similarity metric [9]. The "nanopore2023" error model was used, which used the user-defined read identity distribution to simulate ONT R10.4.1 (2023) sequencing. The read identity distribution is modelled by a beta distribution with parameters identity mean, max, and standard deviation (μ, max, σ). The mean and max read identity values were varied to model the effect of Nanopore performance on overhead. Depth was also varied to model its effect on recovery of strands during clustering. Badread's default settings were used for all other settings: junk reads, random reads, chimeras, adapters.

Given the noisy reads sequenced from our FASTA oligo pool, the first workflow of clustering involved separating strands. Here, repeats of the same oligo were clustered together. A filtering step dropped all strand reads outside of the valid length range, as

shown in Eq (5):

$$\text{oligo length} - 5 \text{ edits} \leq l \leq \text{oligo length} + 2 \cdot \text{adapter length} + 5 \text{ edits} \tag{5}$$

where edits refer to Levenshtein distances. Adapters were removed by iterating through a k-mer search for adapter sub-strands.

Agglomerative clustering was used to iteratively grow clusters by inserting a strand into a cluster if it was within 40 edit distances of the cluster head. If a strand was not within this threshold for any existing clusters, it formed a new cluster. This process was repeated until the number of clusters equaled the number of encoded packets (given as a function of chunk size, c, and user-defined overhead, θ). From the formed cluster, candidates were generated by randomly selecting 15 strands from a cluster, multiple aligning them with MUSCLE [41] and consensus voting bases.

A 4-nt checksum was also appended to Raptor-encoded oligos, acting as a parity check to outputted cluster candidates. IDS errors were averaged out via clustering, and the final checksum ensures validates candidates. Therefore, by integrating this clustering algorithm into our pipeline, the storage channel can be modeled as an erasure-only channel. The erasure rate, ϵ, of the channel is given by Eq 6:

$$\epsilon = \frac{\text{number of packets lost}}{\text{total number of packets communicated}} \tag{6}$$

Finding ϵ can help us define our required overhead, θ, when encoding, as shown by Eq 7:

$$\theta = \frac{k \cdot \epsilon}{1 - \epsilon} + \varepsilon \cdot k \tag{7}$$

We simulate ONT's ReadUntil functionality [42], which allows sequences to be read out in real-time. Thus, we introduce a novel, customizable pipeline capable of progressive JPEG rendering with partial decoding (available at https://github.com/nawazia/RaptorPJPEG). The full pipeline is shown in Fig 3.

Results

We set the scaling factor s to 0.66 heuristically, balancing fragment corruption rates and increasing file size. We also assess the performance of the full pipeline under varied simulated Nanopore sequencing conditions.

Workflow Overview

Figure 4 (left) shows the relative size increase from adding global base-8 RIs during JPEG compression of cat.bmp into cat.jpg, using Equation 1. Figure 4 (right) shows the recovery graph during decoding. A chunk byte size, c=46 (204-nt), degree-1 threshold, p_{thr}=0.4, and 60× depth were used (number of chunks, k=619).

Figure 5 shows the rendering of cat.jpg during partial decoding.

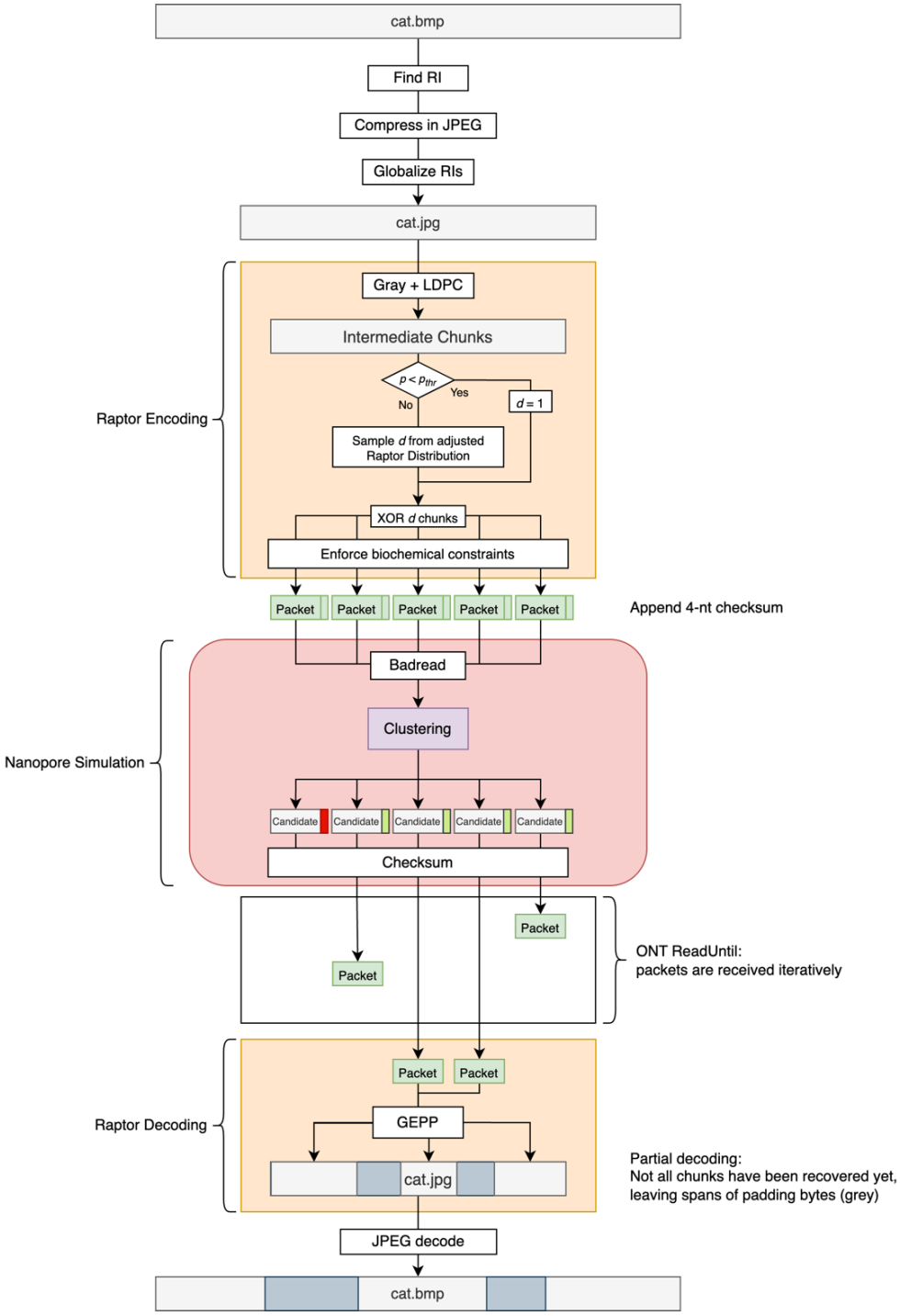
Degree-1 Thresholds

Changing the degree-1 threshold p_{thr} affects the profile of recovery. Figure 6 compares p_{thr}=0.0 (i.e., the original Raptor distribution) with p_{thr}=0.25 & p_{thr}=0.5 in a noiseless scenario (c=46,k=619).

We fit a multivariate linear regression model, modelling AUC_{norm} as a function of $21 \leq c \leq 71$ (step size = 25) and $0 \leq p_{thr} \leq 0.7$ (step size = 0.1). Eq (8) defines the model as:

$$y = X\beta + \varepsilon = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \varepsilon \tag{8}$$

Fig 3. The full pipeline. We simulate real-time sequencing by iteratively reading packets into the decoder.



where $y = AUC_{norm}, x_1 = p_{thr}$, and $x_2 = c$.

We calculated the coefficients as: $\beta = [(61.27, 128.86, -1.01)]$ with a R^2=0.974. The corresponding p-value was 9.710^{-15}. Figure 7 (left) shows the results for c=46 and

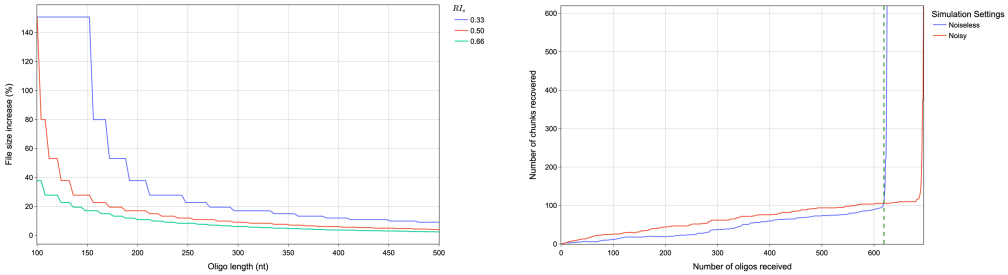


Fig 4. File size increase when for different restart interval scaling factors (left). Number of chunks recovered via GEPP when iteratively receiving more (random) packets, with rendered cat.jpg at different stages of recovery. The green line = k (right).

Fig 5. Iteratively loading more packets to partial decode. Clockwise from top left: 130, 316, 660, 693 oligos received (c = 46, k = 619). We simulate real-time sequencing by iteratively reading packets into the decoder.

c=71. Figure 7 (right) shows the effect of p_{thr} on overhead, ε, for the same c values.

We perform a one-way ANOVA with 4 different c values (c=[34,46,58,71]) to test a null hypothesis: changing c does not affect overhead. We first log-transform our data; running ANOVA returns a F-statistic of 1.5310^{-4} and corresponding p-value of 0.99.

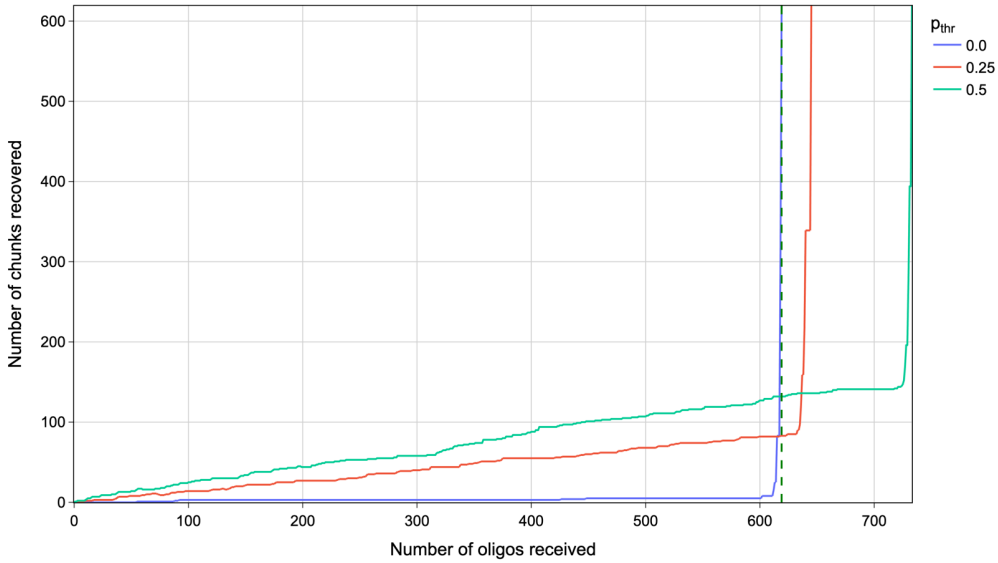
Priority Raptor Codes

Figure 8 compares recovery using naïve Raptor encoding and using prioritized Raptor encoding.

Nanopore Modelling

Setting an oligo length of 208-nt as our target, we generate a FASTA file from cat.jpg containing 909 strands using c=47, k=606,θ=0.5, p_{thr}=0.4. Figure 9 shows the effect of

Fig 6. Different degree-1 thresholds have different recovery profiles. c = 46 bytes = 200-nt oligos; c = 71 bytes = 300-nt oligos



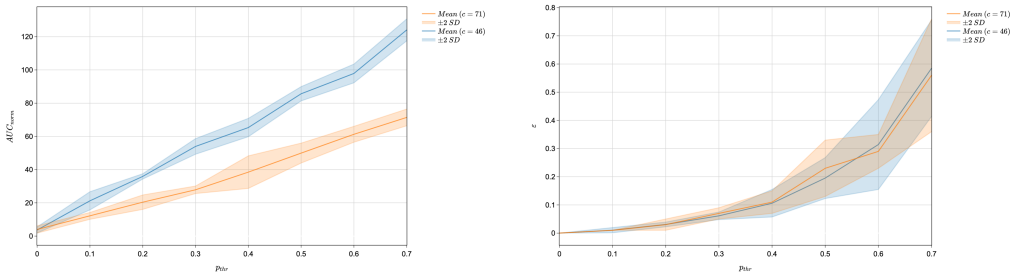


Fig 7. (a) Relationship between Normalised AUC and degree-1 threshold for different chunk sizes. (b) Relationship between overhead and degree-1 threshold for different chunk sizes.

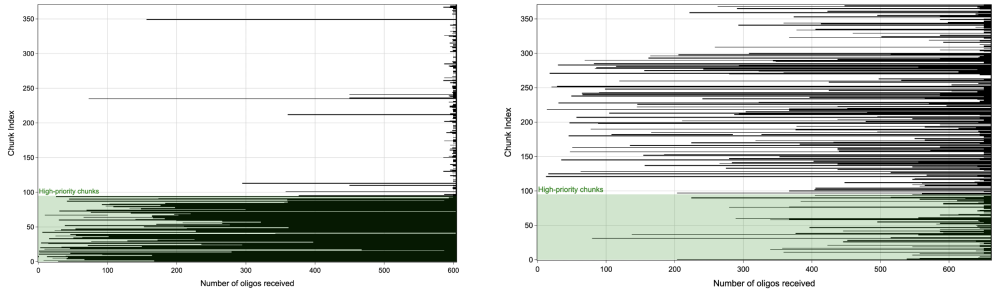


Fig 8. Comparing recovery profiles for individual chunks with high-priority (left) and with naïve encoding (right). Black indicates a chunk is successfully recovered. The green shaded area denotes the chunk high-priority list, comprised of chunks containing DC scan (c = 71, k = 371, p_{thr} = 0.7).

depth and read identity when modelling Nanopore R10.4.1 sequencing and clustering.

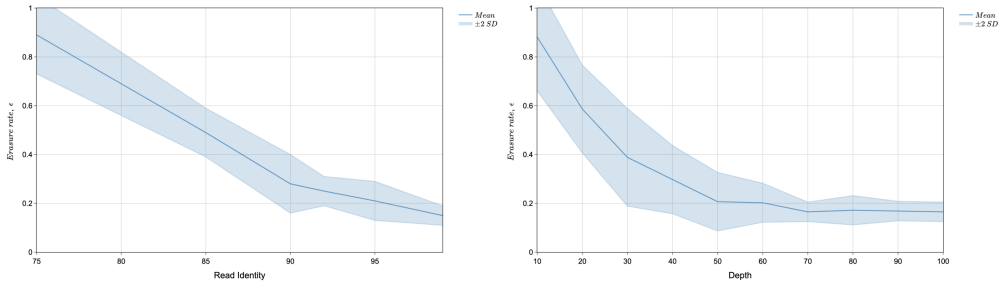


Fig 9. Varying depth affects the erasure rate. Read Identity = (97, 99, 2.5) (left) and varying read identity affects erasure rate (depth = 40x) (right).

Discussion

In this work, we propose a DNA-based progressive JPEG decoder, with on-the-fly error-correcting & rendering capability. We optimize, test and analyze our performance with varying levels of noise. By "stress-testing" our schema *in silico*, we were able to fit our framework to the expected noise characteristics of real Nanopore sequencing conditions, before synthesizing and sequencing *in vitro*.

When developing the pipeline, we encountered problems with partial JPEG rendering. Typically, gaps in a JPEG bitstream will force the decoder to fail, as the MCU index position in the ECS is lost. We adapted our RIs to contain indexing information to circumvent this problem. However, this involved increasing the length of the RI markers (from 2 bytes to 4 or even 6 bytes). When setting $c_n = 1.0$, RI markers were inserted at chunk size, c. We found that this significantly increased the likelihood of RI markers crossing the boundary of two chunks. Therefore, our global index would become fragmented, leading to incorrect pixel rendering positions. Based on these findings, we used $RI_{\text{step}} = 0.66$, which balanced a low file size increase while avoiding fragmentation.

Also shown in 4 (right)is the phase transition between a period of low recovery and an extremely sharp recovery curve. The dotted green line, indicating where the number of packets received equals k, is a good indicator of where that transition occurs. However, adjustments to the Raptor distribution increase the overhead of decoding, shown in Figure 6, and shift this position. Overhead can also be interpreted as the difference in packets received at full recovery (the end of the line) and at the dotted green line. The blue plot in Figure 6 depicts a noiseless naive Raptor decode ($p_{\text{thr}} = 0$) and has an overhead of ˜0 [20]. The phase transition effect is strongest here.

This sharp phase transition is a quality of near-optimal codes, such as fountain codes. Such codes achieve near-optimal (i.e., no overhead) performance by essentially distributing information bits across packets. This has a de-localizing effect on information and packets, making them more resilient to error. Each packet is less critical to the decoding process. This feature is what gives fountain codes such great performance with erasures and allows decoding of a large proportion of data with very low overhead. On the other hand, this property has an effect of diminished partial decoding performance. A certain number of packets need to be retrieved before significant progress can be made during GEPP decoding.

During partial decoding, we attempt to solve an underdetermined system. Some chunks remain unrecoverable until the system reaches full rank, beyond which significant decoding is possible. This accounts for the sharp sudden rise of recovered chunks [43].

Degree-1 packets are constructed of one chunk, making them essentially uncoded data. These packets form the starting point of decoding, as we use these packets to reduce higher-degree packets (from $d = 2$ packets, and so on). Therefore, a higher

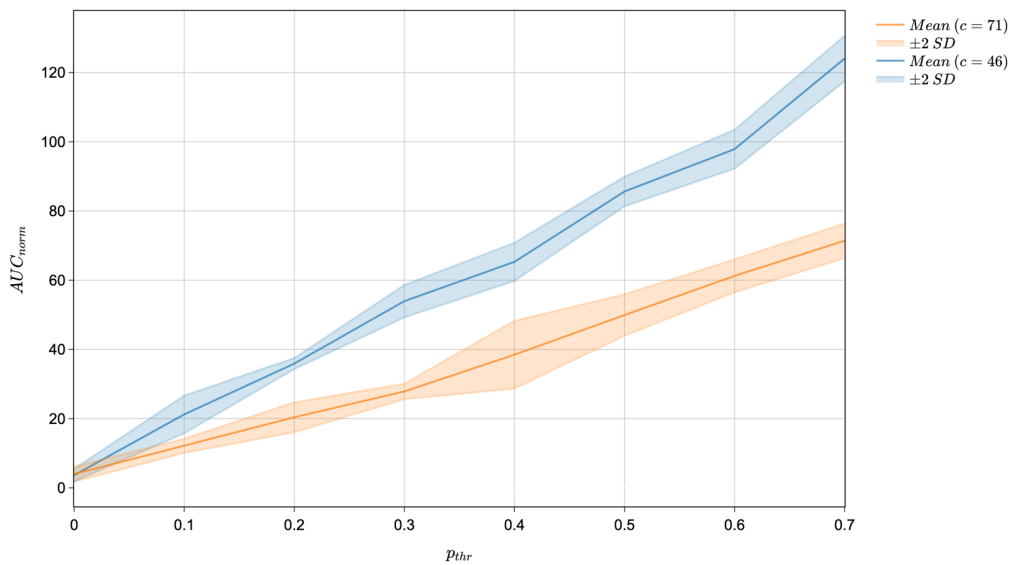
proportion of degree-1 packets can benefit partial decoding and progressive JPEG decoding greatly. A challenge in DNA storage are extremely noisy channels. Usually, this limits the feasibility of increasing degree-1 packets. However, our clustering algorithm is able to reduce the effects of such noise; finding a suitable p_{thr} is key. We can approximate the probability that we form a $d = 1$ packet by Eq 9:

$$p(d = 1) = p_{\text{thr}} + \frac{c \ln\left(\frac{K}{\delta}\right)\sqrt{K}}{K} \tag{9}$$

where c K and δ are parameterized by the Robust Soliton distribution, see **??**.

To characterize the amount by which we can "push" the curve upwards, we calculate AUC. We normalise by the total number of packets received in order to adjust for increasing overhead. We use AUC_{norm} instead of a simple averaging due to its more descriptive nature of the final spike of recovery. We can see in Fig 10 that increasing the degree-1 threshold improves the AUC_{norm}, and that increasing c has a reductive effect on AUC_{norm}, which can be interpreted as a decrease in partial decoding efficiency. The linear regression coefficients also show the effect of c on the partial decoding efficiency.

Fig 10. Normalised AUC and p_{thr} for different chunk sizes. c = 46 bytes = 200-nt oligos; c = 71 bytes = 300-nt oligos



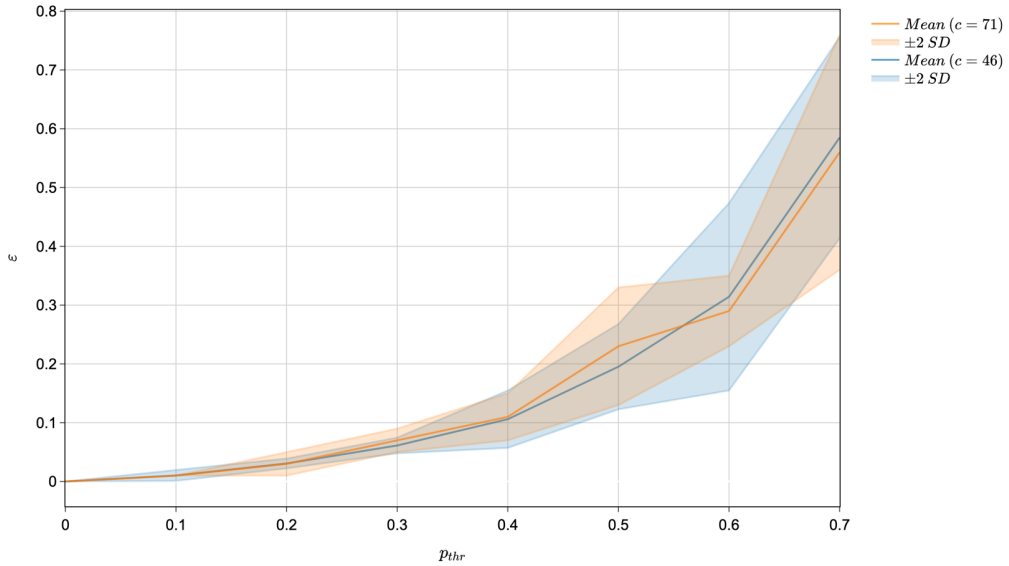
As mentioned, traditional Gaussian elimination (including GEPP) requires full rank systems before solutions can be calculated. There are many alternative methods to enable partial decoding (e.g., generalized inverse, Gauss–Seidel method, belief propagation) but we use partial Gaussian elimination, where we iteratively eliminate as far as we can find a pivot. Once we reach a non-pivot row, we stop and attempt to solve the successfully eliminated rows. This allows us to find solutions even in an underdetermined linear system. The system's binary coefficient matrix rows correspond to a received encoded packet, columns correspond to original source chunks. Element values correspond to whether that chunk was used when generating the packet: 1s denote the presence of a chunk, and 0s denote the absence of a chunk. Therefore, the sparsity of the coefficient matrix is also directly related to the degree distribution.

Fig 10 therefore also shows the effect of increasing sparsity on partial GEPP performance. A sparse system also reduces the effect of fill-in, which decreases sparsity

and efficiency of elimination in a propagative manner. Rank deficient matrices perform much better with increasing sparsity, which increases row independence [44]. These benefits are augmented in a binary matrix.

Figure 11 shows a related result, comparing the degree-1 threshold with overhead. In traditional code analysis, the code rate ($R_c = \frac{k}{n}$) defines the efficiency. Rateless codes like fountain codes can generate an unlimited number of encoded packets. Shannon states that as n tends to infinity (unlimited number of packets), the code rate approaches 0. Practically, the number of packets generated (and therefore n) is fixed by a user-defined parameter, θ. Finding this value involves analysis of the code and channel; fountain codes therefore focus more on overhead rather than code-rates. Given an estimated model of communication, the overhead, ε, can be calculated. This provides a lower bound on θ, the number of packets to be generated. Finding this value allows to estimate packet requirements when synthesizing.

Fig 11. Overhead and $p_{\mathbf{thr}}$ for different chunk sizes. c = 46 bytes = 200-nt oligos; c = 71 bytes = 300-nt oligos

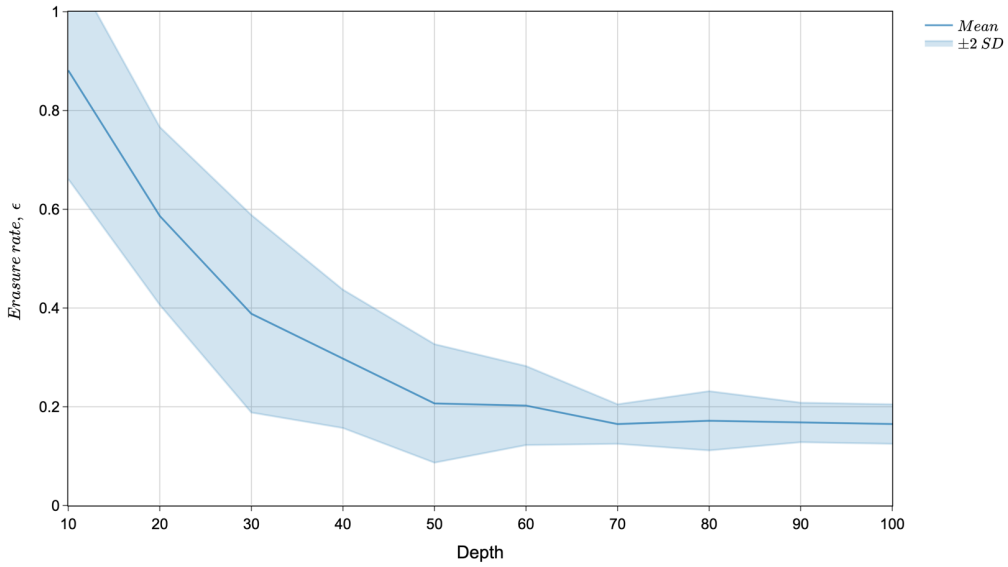


Priority encoding shows great performance even with a non-zero p_{thr}. We are able to recover DC coefficients first. Priority encoding is what enables our pipeline to render progressive JPEGs at low resolution first. Once the coefficient matrix reaches full rank, recovery strongly increases similarly to non-priority encoding. Priority encoding does not seem to affect overhead much. Excluding sequencing & using $p_{\text{thr}} = 0.4$, we introduce only around 10% overhead to do so. A similar rendering effect with uncompressed images would perhaps have a lower overhead, but would require a much larger oligo pool size. For example, encoding the original `cat.bmp` directly would require around 35 times more DNA. Our pipeline provides a novel image-based DNA storage schema which is capable of rendering JPEGs progressively, with very little overhead.

Clustering allows to condense IDS errors into erasures, at a cost of code rate. Our clustering stabilizes partial decoding and thus progressive decoding. We experimented with inner RS codes applied to each oligo but suffered from poor decoding performance. In most cases, errors exceeded the RS code's capability. Furthermore, increasing the number of RS repair symbols increases the overhead greatly. Therefore, our clustering algorithm was developed to mitigate IDS errors by using a Levenshtein distance-based

clustering algorithm. Filtering with the checksum ensured candidates were error-free, and reversed strands were reoriented. Fig 12 shows the effect of depth on the erasure rate, with a fixed read identity Badread error model.

Fig 12. Depth and erasure rate. Read Identity ($\mu = 97, \max = 99, \sigma = 2.5$)



When depth > 60x, the erasure rates stabilize around 0.2. Therefore, using Equation 7 we can calculate the required overhead, θ: around 212 strands. This provides an approximate minimum overhead when encoding at that configuration. A higher depth increases the accuracy and reliability of reads but is not scalable. Lowering depth increases the chance of error, especially insertion & deletion. Sequencing technologies like Nanopore, also have biasing towards certain strand patterns [45]. Badread applies depth to oligos in a non-exact fashion to simulate similar behavior. Increasing depth reduces the effect of these errors & bias. When depth = 60x gives us a good balance of both performance and cost.

Conclusion

We have designed and implemented a novel, DNA-based progressive JPEG decoder, with on-the-fly error correcting and rendering capability. We adapt the JPEG encoding process to store chunk-level data, implement and adjust Raptor encoding to boost progressive and partial decoding performance, and finally we optimise, test and analyse our pipeline in the presence of realistic Nanopore sequencing errors. Our results characterise different parts of our pipeline, and we integrate ideas from different domains to provide better on-the-fly performance.

Supporting information

S1 Fig. **cat.bmp** Uncompressed 680x453 RGB image used for experiments.

References

1. Reinsel D, Gantz J, Rydning J. The Digitization of the World - From Edge to Core. Framingham: International Data Corporation. 2018;.
2. Rydning J, Reinsel D. Worldwide Global StorageSphere Forecast, 2021–2025: To Save or Not to Save Data, That Is the Question. IDC White Paper. 2021;.
3. Garafutdinov RR, Chemeris DA, Sakhabutdinova AR, Kiryanova OY, Mikhaylenko CI, Chemeris AV. Encoding of non-biological information for its long-term storage in DNA; 2022.
4. van der Valk T, Pečnerová P, del Molino DD, Bergström A, Oppenheimer J, Hartmann S, et al. Million-year-old DNA sheds light on the genomic history of mammoths. Nature. 2021;591. doi:10.1038/s41586-021-03224-9.
5. Davis J. Microvenus. Art Journal. 1996;55. doi:10.1080/00043249.1996.10791743.
6. Gibson DG, Glass JI, Lartigue C, Noskov VN, Chuang RY, Algire MA, et al. Creation of a bacterial cell controlled by a chemically synthesized genome. Science. 2010;329. doi:10.1126/science.1190719.
7. Clelland CT, Risca V, Bancroft C. Hiding messages in DNA microdots. Nature. 1999;399. doi:10.1038/21092.
8. Organick L, Ang SD, Chen YJ, Lopez R, Yekhanin S, Makarychev K, et al. Scaling up DNA data storage and random access retrieval. Nature Biotechnology. 2018;36.
9. Levenshtein V. Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady. 1966;10.
10. Heckel R, Mikutis G, Grass RN. A Characterization of the DNA Data Storage Channel. Scientific Reports. 2019;9. doi:10.1038/s41598-019-45832-6.
11. Church GM, Gao Y, Kosuri S. Next-generation digital information storage in DNA; 2012.
12. Goldman N, Bertone P, Chen S, Dessimoz C, Leproust EM, Sipos B, et al. Towards practical, high-capacity, low-maintenance information storage in synthesized DNA. Nature. 2013;494. doi:10.1038/nature11875.
13. Heckel R, Shomorony I, Ramchandran K, Tse DNC. Fundamental limits of DNA storage systems. In: IEEE International Symposium on Information Theory - Proceedings; 2017.
14. Shomorony I, Heckel R. Information-Theoretic Foundations of DNA Data Storage. Foundations and Trends in Communications and Information Theory. 2022;19. doi:10.1561/0100000117.
15. Grass RN, Heckel R, Puddu M, Paunescu D, Stark WJ. Robust chemical preservation of digital information on DNA in silica with error-correcting codes. Angewandte Chemie - International Edition. 2015;54. doi:10.1002/anie.201411378.
16. Blawat M, Gaedke K, Huetter I, Chen XM, Turczyk B, Inverso S, et al. Forward error correction for DNA data storage. Procedia Computer Science. 2016;80:1011–1022.

17. Organick L, Ang SD, Chen YJ, Lopez R, Yekhanin S, Makarychev K, et al. Random access in large-scale DNA data storage. Nature biotechnology. 2018;36(3):242–248.

18. Luby M. LT codes. In: The 43rd Annual IEEE Symposium on Foundations of Computer Science, 2002. Proceedings.; 2002. p. 271–280.

19. Erlich Y, Zielinski D. DNA Fountain enables a robust and efficient storage architecture. Science. 2017;355. doi:10.1126/science.aaj2038.

20. Shokrollahi A. Raptor codes. IEEE Transactions on Information Theory. 2006;52. doi:10.1109/TIT.2006.874390.

21. Maymounkov P. Online codes. Technical report, New York University; 2002.

22. Schwarz PM, Freisleben B. NOREC4DNA: using near-optimal rateless erasure codes for DNA storage. BMC Bioinformatics. 2021;22. doi:10.1186/s12859-021-04318-x.

23. Antkowiak PL, Lietard J, Darestani MZ, Somoza MM, Stark WJ, Heckel R, et al. Low cost DNA data storage using photolithographic synthesis and advanced information reconstruction and error correction. Nature communications. 2020;11(1):5345.

24. Mao W, Diggavi SN, Kannan S. Models and Information-Theoretic Bounds for Nanopore Sequencing. In: IEEE Transactions on Information Theory. vol. 64; 2018.

25. Yazdi SHT, Gabrys R, Milenkovic O. Portable and error-free DNA-based data storage. Scientific reports. 2017;7(1):5011.

26. Dimopoulou M, Antonini M, Barbry P, Appuswamy R. A biologically constrained encoding solution for long-term storage of images onto synthetic DNA. In: European Signal Processing Conference. vol. 2019-September; 2019.

27. Dimopoulou M, Antonio EGS, Antonini M. A JPEG-based image coding solution for data storage on DNA. In: European Signal Processing Conference. vol. 2021-August; 2021.

28. Lazzarotto D, Ramos JE, Testolina M, Ebrahimi T. Technical description of the EPFL submission to the JPEG DNA CfP; 2023.Available from: `https://api.semanticscholar.org/CorpusID:265552094`.

29. Wallace GK. The JPEG Still Picture Compression Standard. Communications of the ACM. 1991;34. doi:10.1145/103085.103089.

30. M RA, M KW, dosuky M A E, Ahmed W. Jpeg Image Compression Using Discrete Cosine Transform - A Survey. International Journal of Computer Science & Engineering Survey. 2014;5. doi:10.5121/ijcses.2014.5204.

31. Sencar HT, Memon N. Identification and recovery of JPEG files with missing fragments. Digital Investigation. 2009;6. doi:10.1016/j.diin.2009.06.007.

32. Li B, Ou L, Du D. IMG-DNA: Approximate DNA storage for images. In: SYSTOR 2021 - Proceedings of the 14th ACM International Conference on Systems and Storage; 2021.

33. Hudson G, Léger A, Niss B, Sebestyén I, Vaaben J. JPEG-1 standard 25 years: past, present, and future reasons for a success. Journal of Electronic Imaging. 2018;27(4):040901–040901.

34. Mladenov T, Kim K, Nooshabadi S. Forward error correction with RaptorQ Code on embedded systems. In: Midwest Symposium on Circuits and Systems; 2011.

35. Gallager RG. Low-Density Parity-Check Codes. IRE Transactions on Information Theory. 1962;8. doi:10.1109/TIT.1962.1057683.

36. Agrell E, Lassing J, Ström EG, Ottosson T. On the optimality of the binary reflected gray code. IEEE Transactions on Information Theory. 2004;50. doi:10.1109/TIT.2004.838367.

37. Matsumoto M, Nishimura T. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. In: ACM Transactions on Modeling and Computer Simulation. vol. 8; 1998.

38. S GW, Golub GH, Loan CFV. Matrix Computations. Mathematics of Computation. 1991;56. doi:10.2307/2008552.

39. Woo SS, Cheng MK. Prioritized LT codes. In: CISS 2008, The 42nd Annual Conference on Information Sciences and Systems; 2008.

40. Wick R. Badread: simulation of error-prone long reads. Journal of Open Source Software. 2019;4. doi:10.21105/joss.01316.

41. Edgar RC. MUSCLE: Multiple sequence alignment with high accuracy and high throughput. Nucleic Acids Research. 2004;32. doi:10.1093/nar/gkh340.

42. Loose M, Malla S, Stout M. Real-time selective sequencing using nanopore technology. Nature Methods. 2016;13. doi:10.1038/nmeth.3930.

43. Bruckstein AM, Donoho DL, Elad M. From sparse solutions of systems of equations to sparse modeling of signals and images; 2009.

44. Pouransari H, Coulier P, Darve E. Fast hierarchical solvers for sparse matrices using extended sparsification and low-rank approximation. SIAM Journal on Scientific Computing. 2017;39. doi:10.1137/15M1046939.

45. Delahaye C, Nicolas J. Sequencing DNA with nanopores: Troubles and biases. PLoS ONE. 2021;16. doi:10.1371/journal.pone.0257521.