Title

Subtitle

Seminar Datenmanagement
Prof. Dr. Lena Wiese
Semester 7

Institute of Computer Science
Goethe-Universität Frankfurt a. M.

Author:	Nako Nachev
	6464679
	s6327063@stud.uni-frankfurt.de
Degree:	Master Informatik, Semester 7
Module ID:	DM-MS, M-DS-S, DM-BS, B-ATAI-S
Date:	February 24, 2024

Work based on:

- Paper 1: *Mercier, Hugues and Bhargava, Vijay K and Tarokh, Vahid*: A survey of error-correcting codes for channels with symbol synchronization errors
- Paper 2: *Erlich, Yaniv and Zielinski, Dina* DNA Fountain enables a robust and efficient storage architecture

Abstract

This article provides a brief overview of different error-correcting codes and their specifics, such as usage and capability of correcting errors of various kinds. Subsequently, it describes a specific storage strategy called DNA Fountain by examining the different stages, such as encoding and decoding, and provides information about the exact construction and experiments specifics. Furthermore, a python implementation of the storage strategy is suggested, serving as a testing ground to experiment with this storage mechanism. Moreover, two digital logos of Fraunhofer institute are supplied as a benchmark dataset and are used as input data for the python scripts. Lastly, the article concludes with criticism on the topic and lists potential problems or future improvements for the storage strategy.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Summary Overview Paper

The paper in [2] provides a detailed survey about different error correcting codes, their implementation and associated obstacles with each of the codes. The paper begins by examining the historical significance of error synchronization problems in communication systems dating back to Samuel Morse, which remain an important topic even in today's technological systems. Although often treated as different problems, synchronisation and additive noise tend to have the same effect on communication channels by reducing their capacity.According to the article, designing error-correcting codes to tackle these problems proves to be quite challenging, with many techniques of implementation not based on coding or even failing to mention synchronisation-correcting codes.
This article continues with a brief introduction on the error-correcting codes and explaining the different preliminaries associated with the survey. Such preliminaries include the mathematical notations used, the different types of synchronisation errors and their definitions. These range from deletion errors, insertion errors and duplication or repetition errors to bitshift errors. Last but not least, obstacles associated with the error correcting codes are described. Examples of such are huge bursts of substitution errors, splitting long messages of data into blocks with boundaries sometimes unknown to the receiver or codes being unable to keep the same correction capability when encoding several blocks. The authors describe in section III different synchronisation-correcting codes, how they are implemented, the historic research done by various authors and specific limitations for the different types. The section starts with binary algebraic block codes using (0,1) alphabets, nonbinary and perfect codes, codes capable of correcting bursts of synchronisation errors, synchronizable, marker codes, codes for weak synchronisation errors, convolutional codes, expurgated and codes for random synchronisation channels.
The last section of the paper discusses future directions of synchronisation error-correcting codes. Binary block codes are currently unable to detect more than one synchronisation errors per block. Further challenges arise in estimating the capacity of channels corrupted by synchronisation errors as well as situations where random deletion and/or insertion errors can occur. According to the authors, none of the codes described in the paper has all

the properties required to be used in pratical communication systems. One solution could be concatenating different coding schemes in order to use the strengths of the various codes for their specific implementation use cases.
Drawing from the examples in the preciding article, one could divide the error-correcting codes in multiple categories. One separation used could be based on the number of symbols used in the alphabet - either binary alphabet consisting of (0,1)* symbols or non-binary with 2 or more symbols. Another categorization could follow based on other characteristics - such as block codes vs convolutional codes. Block codes break the data down to fixed-sized blocks and append protective or unique bits for error handling. Convolutional codes on the other side process the data continuously as a stream and do not segment the data into blocks. Error correction here is done on the fly.
As far as characteristics of the error-correcting codes are concerned, performance, redundancy, error detection and correction seem to be the most important ones. Redundancy finds its application in appending the extra bits for the receiver to make use of and identify errors that may have happened during the transmition. Performance concerns the rate at which algorithms are able to correct the data, where similar to most systems a good balance between correctness achived and computation should be achieved. Error detection and correction seems to be closely related and are the general characteristics that prove how valuable a given error-correcting code is.
There are various implementations of error-correcting codes. One such example is by using error-correcting output coding for improving text classification and thus reducing text classification errors by 66 percent. [3] Another example is by using error-correcting codes for real time communication in wireless networks. [4] The authors simulate using an adaptive scheme for reducing bandwidth overhead by comparing it to a single code scheme.

1.2 Important Terms and Definitions

In the context of error-correcting codes for channels with synchronisation errors, following important terms and definitions can be listed based on the information provided in [2]:

- **Synchronisation error**: These are either a deletion or an insertion error and excludes substitution errors. For example transmitting 0011 instead of 00011 is a deletion error and transmitting 0011 instead of 011 - insertion error.

- **Duplication or repetition error**: These errors are a special insertion that replaces a bit by two copies of the same bit.
- **Bitshift error**: A special kind of error that might reverse a given string - for example 01 being transmitted as 10.
- **Additive noise**: Disturbances or interference when transmitting signal over a communication network
- **Redundancy**: Additional information introduced by error-correcting codes to correction synchronisation errors
- **Run**: A substring of identical symbols of maximum length
- **Block**: Used to form chunks of words that are used to divide long messages

The concepts and terms provided in [2] are crucial for understanding the challenges in designing error-correcting codes for channels with erasures. Due to the specifics and various mechanisms for correcting errors, one such method is described in 2 to give more clarity and details about the implementation.

2 In-Depth Explanation of Error-correcting Code Chosen

Current storage medias do not have the capacity to meet the demand for the total amount of worldwide data. Moreover, the costs for maintaining and transferring data are increasing, therefore better solutions are needed [5]. One such promising solution is the innovative DNA storage systems. These systems allow us to store digital data into DNA molecules, which are synthesized and far exceed the lifespan of general storage mediums [5]. However, transferring and storing data comes with their complications and certain errors may occur, leading to corrupted or incomplete data. Thus, solutions for correcting these errors are necessary. The storing strategy described in [6] makes use of the so-called DNA fountains. Fountain codes are typically used in channels with erasures, where files are split into packets and each packet is either received without errors or dropped [7]. Standard transfer protocols keep transmitting packets until they are successfully received from the other side, where another channel is also required to keep track of the state of transmitted packets or those that require retransmission. Fountain codes on the other side create packets as a result of functions of the whole file and the receiver only needs to collect any N packets, with $N > K$ (original size), to recover the whole file.In the metaphorical sense, a dna fountain generates droplets representing packets of data and anyone wishing to collect these droplets has to place a bucket under the fountain and wait until the amount collected exceeds the original size K of the data [7]. Due to the nature of the fountain, DNA fountains can be classified as rateless, since the number of packets generated can be limitless. The strategy for data storage using DNA fountains proposed in [6] works in two stages.

2.1 DNA Fountain encoding

The first stage is the encoding phase, which consists of three stages - preprocessing, luby transform and screening.

2.1.1 Preprocessing

The first step of the encoding is the preprocessing stage. It begins by packaging the files we want to encode in a tar file and compressing them using stardard lossless algorithms [1].It partitions the file into non-overlapping segmets of identical length. A simple example would be splitting the code 001001100101 into segments of length 4, resulting in the following segments:

- Segment 1: 0010
- Segment 2: 0110
- Segment 3: 0101

2.1.2 Luby Transform

The next step of the encoding is the so-called **Luby-transform stage**. The first job of the Luby Transform stage is to initialize a pseudorandom number generator (PRNG) with the help of a seed. The seed used by the authors in [1] corresponds to the current state of the PRNG, with each cycle for the seed generated with the help of a Linear Feedback Shift Register (LFSR) [1]. An LFSR is a shift register whose input bit is a linear function of its previous state [8]. A LFSR uses a polynomial to generate the so called ”taps”, which correspond to the coefficients of the polynomial. It works by first shifting the n-bit large string to the right and then XORs together the bits from the previous state corresponding to the ”tap” positions. Given the right polynomial, a LFSR can be considered a ”maximum-length LFSR”, since it is able to produce $2^n - 1$ states before returning to the initial state [8]. In the context of the dna fountain strategy, such LFSR will examine each seed in a given interval without repetition [1]. Below is a quick explanation on how the LFSR works:

- Suppose we generate a 3-bit seed
- Set the initial seed: 100
- Provide a polynomial for the LFSR: $x^2 + x + 1$
- The above polynomial defines taps on positions (2,1)
- Shift the seed to the right: `_10`
- XOR together the bits under the tap positions: $0 \oplus 0 = 0$
- Append the result from the XOR operation to the left, leaving us with **010** as next state

After generating a seed, the next job of the Luby Transform is to decide how many of the segments to combine together. This is achieved with the help of a special robust soliton distribution, defined as follows [1]:

$$\rho(d) = \begin{cases} \frac{1}{K} & \text{if } d = 1, \\ \frac{1}{d(d-1)} & \text{for } d = 2, \ldots, K. \end{cases}$$

$$\beta(d) = \begin{cases} \frac{s}{Kd} & \text{for } d = 1, \ldots, \left(\frac{K}{s}\right) - 1, \\ \frac{s \ln(s/\delta)}{K} & \text{for } d = \frac{K}{s}, \\ 0 & \text{for } d > \frac{K}{s}, \end{cases}$$

$$\mu_{K,c,\delta}(d) = \frac{\rho(d) + \beta(d)}{Z},$$

with

$$s = c\sqrt{\frac{K}{\ln^2\left(\frac{K}{\delta}\right)}}$$

and

$$Z = \sum_d \rho(d) + \beta(d) = 1$$

After the distributions for the robust soliton distribution are calculated, N segments are sampled at randomly without replacement (based on the choice from soliton) and than added bitwise together. The seed generated from the LFSR is attached to the result of the bitwise addition, thus forming the final droplet. One simple example might be:

- Choose two random segments: 0010 and 0110
- Apply bitwise addition: $0010 \oplus 0110 = 0100$
- Generate seed: 11
- Attach seed to the result of the bitwise addition to form the droplet: 110100

Apart from the seed and payload, extra redundancy can be added to the droplet in order for it to be able to correct various errors that may appear during the syntesizing process. This is where Reed-Solomon codes (R-S) come into place. Reed-Solomon codes are nonbinary cyclic codes with symbols up to m-bit, with m being any positive integer having a value greater than 2. R-S (n,k) codes of m-bit symbols comply to the following rules:

$$0<k<n<2^m+2$$

with k being the number of symbols encoded and n the total number of symbols in the encoded block [9]. Furthermore, R-S(n,k) codes also include parity symbols:

$$(n,k) = (2^m-1, 2^m-1-2t)$$

with $2t = n-k$ being the number of parity symbols [9]. For example a R-S(5,3) 3-bit code contains 5 code word bytes, of which 3 are data and 2 parity (n-k).Therefore, this code is capable of correcting up to $2t = 2 => t = 1$ errors.

2.1.3 Screening

The Luby transform stage concludes with a screening stage.This stage first converts the achieved droplet to DNA using the following conversion rules: {00,01,10,11} to {A,C,G,T} [1]. After that, the result dna sequence is screened for invalid sequences, such a homopolymer runs or GC content. Long homopolymer runs and sequences with high GC concentration are undesirable [6] due to being more difficult to syntesize or are more prone to sequencing errors. DNA sequences containing one of the above described properties are rejected and those that pass added to the oligo design file. In [6], the created droplets were 38 bytes (4 bytes for seed, 32 bytes for data payload and 2 bytes for RS code). Therefore, the values for homopolymer runs were set to $<= 3nt$ (nucleotides) and 45-55% GC content, but these values could be changed based on the specific case. Below is the screening stage described with continuation of the example above:

- Convert the droplet to a DNA sequence: 110101 $\rightarrow TCC$
- Check against screening rules: TCC does not contain long homopolymer runs or high % GC content, therefore does not get rejected
- The oligo file now contains the droplet TCC

Since quite a few of the oligos can fail, the authors in [6] recommend aiming for 5-10% more oligos than input segments. The process of luby transform and screening stage is repeated until that number is reached.

2.2 DNA Fountain decoding

The decoding part of the DNA Fountain encoding strategy consists also of 3 stages - preprocessing, droplet recovery and segment interference.

2.2.1 Preprocessing

The preprocessing phase of the decoding process begins by filtering out all sequences that do not equal the predefined length [1]. Subsequently, idential sequences are dropped and the number of occurences is stored, ensuring that more prevalent sequences will appear first when sorted. As the decoder may not always need to scan all oligos, some of the singletons (sequences that have been observed only a small number of times) may be omitted and the decoder would not try to decode them. Below is a simple example illustrating the process:

- Assume these are the generated oligos from the encoding process: GAT, TAG, CAG, GTC, GAT, AAG, GAT, GGAT, CAG
- Step 1: Remove oligos larger than 3nt - GGAT is filtered out
- Step 2: Calculate occurrences - GAT: 3, TAG: 1, CAG: 2, GTC: 1, AAG: 1
- Step 3: Sort sequences based on occurrence - GAT:3, CAG: 2, TAG: 1, GTC: 1, AAG: 1

2.2.2 Droplet Recovery

The second phase of the decoding process is droplet recovery. The generated DNA sequences from the encoding process have to first be translated back into binary using the following scheme, as detailed in [1]: A,C,G,T, to 0; 1; 2; 3. In addition to translating the sequences, the droplet recovery stage must also extract the seed, payload and error-correcting code (if it exists). Unlike other methods, the in [6] described approach does not try to recover sequences with errors. Instead, they are excluded without attempting to correct them. Most of the errors found are mainly due to short insertions and deletions that may result during the oligo synthesis. The Reed Solomon code used in the encoding process is designed to handle only substitution errors, therefore it is not advisable to attempt recovery, as mentioned in [1].

2.2.3 Segment inference

The final step in the decoding process is the segment inference. Once the integrity of the message is verified (meaning that all checks from the previous stages have passed), the decoder initializes the PRNG with the extracted seed obtained from 2.2.2. Using this seed, a list of segment identifiers can be generated as described in [1]. These identifiers represent potential positions of the input segments that were combined to form the droplet, as the same seed was used for the PRNG initialization during the encoding process.
Next, a message-passing algorithm is initiated to try and infer the segments of each droplet. The algorithm first checks whether the droplet contains segments that have been previously inferred. If such are found, they are XOR-ed from the droplet, and their index is removed from the list of indices. When only one segment remains in the droplet, the segment will be assigned to the droplet data payload. The information regarding the inferred segments is propagated to the remaining droplets, and the entire algorithm is recursively repeated until no more updates can be made. This leads to the number of solved segments slowly increasing until the number of observed oligos reaches the number of segments. At this point, the number of solved segments explodes, as illustrated in the picture below:

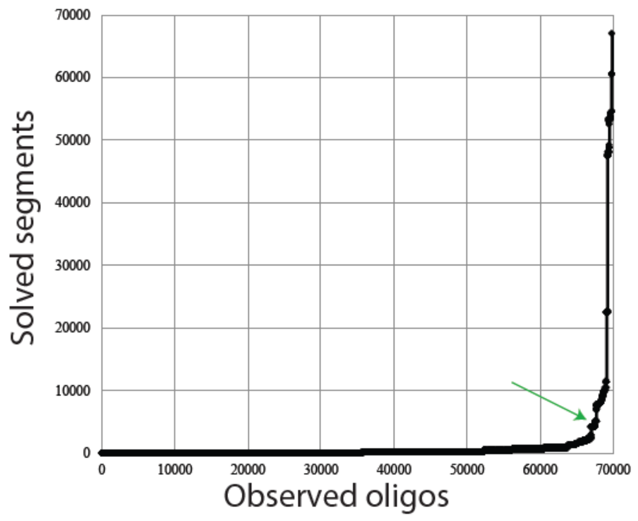

Figure 1: Solved to observed oligos comparison [1]

3 Implementation of Error-correcting Code Chosen

3.1 DNA Fountain implementation

The suggested in [6] dna fountain method was implemented in python by making use of standard libraries and a couple of extra libraries for ready implementations of the LFSR and RS-Codes. The project contains the following files:

- `dna_fountain.py`
- decoder.py
- helper.py
- `image_creator.py`
- models.py
- soliton.py
- input.py

The starting point of the project is the `dna_fountain.py` file. The file begins by initializing the parameters needed for the soliton distribution and the encoding process. The first step involves choosing the input data to use. For the purpose of the seminar, two different inputs were predefined (`picture_short` and `picture_long`), both of which are saved in the input.py file. Both inputs are Python lists with 0s and 1s, corresponding to the different bits saved from the pictures. Based on the chosen input, different globals and defaults are created. Examples of such are `segment_size`, `initial_seed`, polynomial and `segments_multiplier`. The function continues by first splitting the input list into multiple segments based on the initialized parameters. If the input file is not divisible by the number of bits per segment due to the chosen segment size, the last segment is filled with 0s on the left side to match the size of the other segments. After the list of segments is created, it is passed to the `robust_soliton_distribution` function from the soliton.py file, which returns the distribution of the values based on the total number of segments generated. Soliton.py also contains a `ideal_soliton_distribution` function,

which is not used, but remains an option.
After all defaults, parameters and values are initialized, the oligo creation process via the `oligo_creation` functions begins. It first defines the total number of oligos to be generated and starts iterating until the desired number is reached. The `luby_transform` function is responsible for the creation of oligos. It first creates a droplet (with the correspondig seed, data payload and RS-Code (if exists)) and checks it against homopolymer runs or high GC content. If any of the conditions are met, the function with not return anything, meaning the iterator has to try again. Otherwise, the function returns the created oligo, and it is passed to the final list. The core of the `luby_transform` function is the `create_droplet_bin` function. It creates the seed by using the LFSR with the predefined parameters (seed, polynomial) and chooses a number N of segments to combine, with the help of the generated distribution from the robust soliton implementation. The seed is then used to generate N indices for random segments of the list, which are XOR-ed together to form the final payload of the droplet. Along the way, different helper functions and models are used. Typical examples of helper functions include binary arrays to bytes, XOR-ing integers or strings or different utility functions, all of which provided in helper.py file. Models.py contains classes representing different types, which, although not necessary in python due to missing static checking, might improve readability and debugging (if necessary).
Once the oligos are created, they are passed, along with other parameters to the decoder.py file. This file starts by preprocessing the oligos. The first two stages of preprocessing, as suggested by [1], involve retaining only sequences of a certain length and collapsing identical sequences and sorting them so that sequences that occur more often are at the front of the list. Since the created oligos are still in string format (due to nucleotides being the letter A,C,G,T), we need to convert the data back to binary. This is achieved via the `create_oligo` function, which also extracts the seed, data payload and potential error correcting codes that were appended at the end of the oligo. The described in [6] process does not attempt to make error corrections and for that purpose oligos with errors are dropped and do not fall into the decoding process. Last but not least, the `recursively_infer_segments` function is called, which is the final stage of the decoding process. It iterates through the preprocessed oligos and attempts to infer the segments until no more updates can be made. It finally returns the inferred segments and they are passed back to the `dna_fountain.py` file.

The final part of the `dna_fountain.py` file is the image creation. The `image_creator.py` file is called, which contains a function for generating a .PNG picture based on the returned data (list of 0s and 1s). That way we can compare the output picture to the initial picture and compare whether we have successfully decoded the initial file. Since some of the bits may be different, which most of the time is might not be noticable with a human eye, the `calc_similarity` function from the helper.py file can compare both input and output lists and return a value between [0,1] indicating how similar the decoded file is to the initial. Value of 1 means the initial file was decoded without any errors or missing information, anything below it would indicate missing or incorrect data.

3.2 Benchmark Results

In order to test the decoding power and correctness of the in 3.1 described implementation, a benchmark dataset comprised of two different datasets was used. The first dataset is the leftmost logo part of the Fraunhofer Institute logo 2, which is afterwards converted to a python list of 0s and 1s and saved into the input.py file. The second dataset in 3 is the complete Fraunhofer logo, which is then converted to a binary list similarly to the transformation of the first dataset. Both of the generated pictures are black and white and do not take into account the original colours of the logo. For testing purposes a third set was also used, comprised of only 16 bits, which enables easier tracking of the whole encoding/decoding process.
White creating output pictures based on the decoded data a 100% completeness was achieved for both logos. Depending on the chosen logo, different parameters were used for the various stages in the encoding/decoding. Examples of such parameters are the size of the segments in bits, the initial seed, the chosen polynomial for the LFSR and the segments multiplier. Both c and delta values for the soliton distribution remained the same, although they could just as well be made to change accordingly, based on predefined conditions. The table below describes the parameters used for each of the tests:

Several noteworthy aspects consider the chosen parameters for the different inputs. One such aspect is the segments multiplier. When testing with the minimal input (16 bits), a higher amount of oligos needs to be generated in comparison to the input segments. The main reason for that is since we are picking segments at random, there exists the possibility of not picking a

Dataset	Minimal dataset	Short dataset	Long dataset
Segment size	4	32	4
Initial seed	1000	1..0	1..0
Seed size	4 bits	32 bits	32 bits
Polynomial	$x^4 + x^3 + 1$	$x^{32} + x^{30} + x^{26} + x^{25} + 1$	$x^{32} + x^{30} + x^{26} + x^{25} + 1$
Segments multiplier	5	3.5	3
Soliton c value	0.2	0.2	0.2
Soliton delta value	0.001	0.001	0.001

Table 1: Benchmark test parameters

certain segment, which would lead to not being able to fully restore the file. Another important consideration is the seed size. Since we are preprocessing the input into non-overlapping segments, choosing a segment size which does not split the input into equal size can introduce difficulties. For example filling the remaining segments with leading 0 bits may be necessary to match the required size. Furthermore, the initial seed must contain at least one "1" bit, due to how the LFSR works, otherwise it will not be able to cycle through different states. It is important to note that the values provided in the table above may not always achieve 100% completion, as some differences may occur based on how the oligos are created.

4 Conclusion

To summarize, in this work, a DNA storage strategy called "DNA Fountain" was presented. It consists of two main stages - encoding and decoding, both of which work together to enable virtually unlimited data retrieval. According to the authors in [6], the suggested data storage strategy enables "high physical density while approaching the Shannon capacity of DNA storage closer than any previous design".

In the underlying experiments, 2.15 Mbytes of data were encoded and later decoded, while achieving 1.98 bits/nt coding potential, and only using 7% redundancy against dropout. This also led to the full recovery of the encoded data, as demonstrated with the implementation part in 3.1. In contrast to the authors in [6], the implementation in 3.1 required significantly more oligos (up to 3.5 times compared to only 5-10% more) to be able to fully decode the data. This will introduce more overhead and, most likely, higher prices due to the need of synthesizing these oligos. The high cost of DNA synthesis is also mentioned in [6], with data pointing at 3500 dollars / Mbyte. Two general approaches are also mentioned to potentially address the issue of high costs. The first one involves continuous improvements to the DNA synthesis chemistry. The second one aims to achieve cost reduction by exploring "quick-and-dirty" oligo synthesis methods to optimize machine time.

Another aspect worth mentioning is that the above suggested strategy does not attempt to correct encoding errors. The supplied extra bits (for example through Reed Solomon Code) only aim to recognize if there were any errors during oligo synthesis or transferring. If such errors are found, the oligos are dropped and ignored. This is a big difference in comparison to other methods that try to correct and identify various errors by using error-correcting algorithms. Moving forward, with the right optimization and investment, DNA storage methods may prove to be cost-efficient and viable long-term solutions.

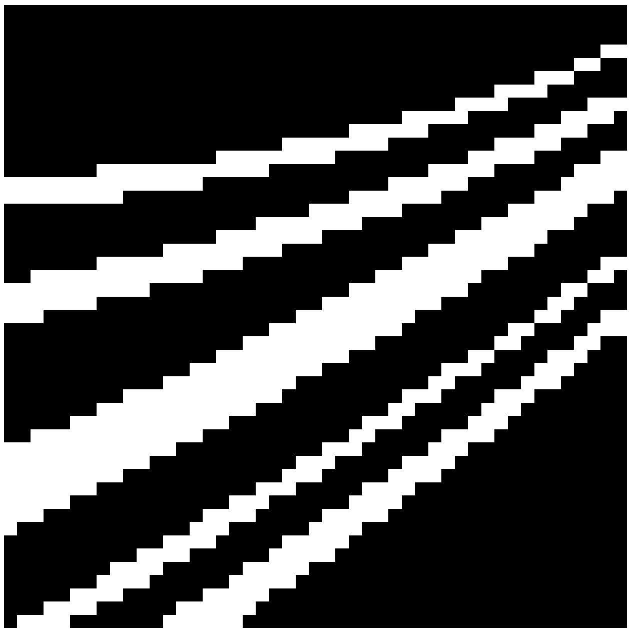

Figure 2: Small Fraunhofer logo

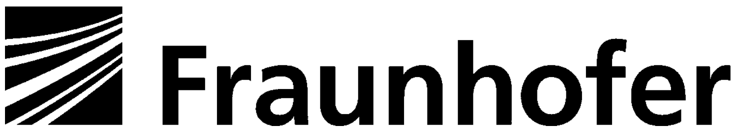

Figure 3: Complete Fraunhofer institute logo

References

[1] Yaniv Erlich and Dina Zielinski. Supplementary materials for dna fountain enables a robust and efficient storage architecturedna fountain enables a robust and efficient storage architecture. *science*, 355(6328):950–954, 2017.

[2] Hugues Mercier, Vijay K Bhargava, and Vahid Tarokh. A survey of error-correcting codes for channels with symbol synchronization errors. *IEEE Communications Surveys & Tutorials*, 12(1):87–96, 2010.

[3] Rayid Ghani. Using error-correcting codes for text classification. In *ICML*, pages 303–310. Citeseer, 2000.

[4] Moncef Elaoud and Parameswaran Ramanathan. Adaptive use of error-correcting codes for real-time communication in wireless networks. In *Proceedings. IEEE INFOCOM'98, the Conference on Computer Communications. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Gateway to the 21st Century (Cat. No. 98*, volume 2, pages 548–555. IEEE, 1998.

[5] Yiming Dong, Fajia Sun, Zhi Ping, Qi Ouyang, and Long Qian. Dna storage: research landscape and future prospects. *National Science Review*, 7(6):1092–1107, 2020.

[6] Yaniv Erlich and Dina Zielinski. Dna fountain enables a robust and efficient storage architecture. *science*, 355(6328):950–954, 2017.

[7] David JC MacKay. Fountain codes. *IEE Proceedings-Communications*, 152(6):1062–1068, 2005.

[8] Shruti Hathwalia and Meenakshi Yadav. Design and analysis of a 32 bit linear feedback shift register using vhdl. *Int. J. Eng. Res. Appl*, 4:99–102, 2014.

[9] Bernard Sklar. Reed-solomon codes. *Downloaded from URL http://www. informit. com/content/images/art. sub.–sklar7. sub.–reed-solomo-n/elementLinks/art. sub.–sklar7. sub.–reed-solomon. pdf*, pages 1–33, 2001.