

FAST AND RELIABLE VALIDATION SYSTEM FOR PRINTED DOCUMENTS

by
Alexis Denis

An M.Sc. Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering
University of Manitoba
Winnipeg, Manitoba, Canada

Thesis Advisor: Prof. W. Kinsner, Ph.D., P.Eng.

(xxiii + 166 + A124 + B6 =) 319 pp.

© A. Denis; November 2001



National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-76927-5

THE UNIVERSITY OF MANITOBA
FACULTY OF GRADUATE STUDIES

COPYRIGHT PERMISSION PAGE

FAST AND RELIABLE VALIDATION SYSTEM FOR PRINTED DOCUMENTS
BY

Alexis Denis

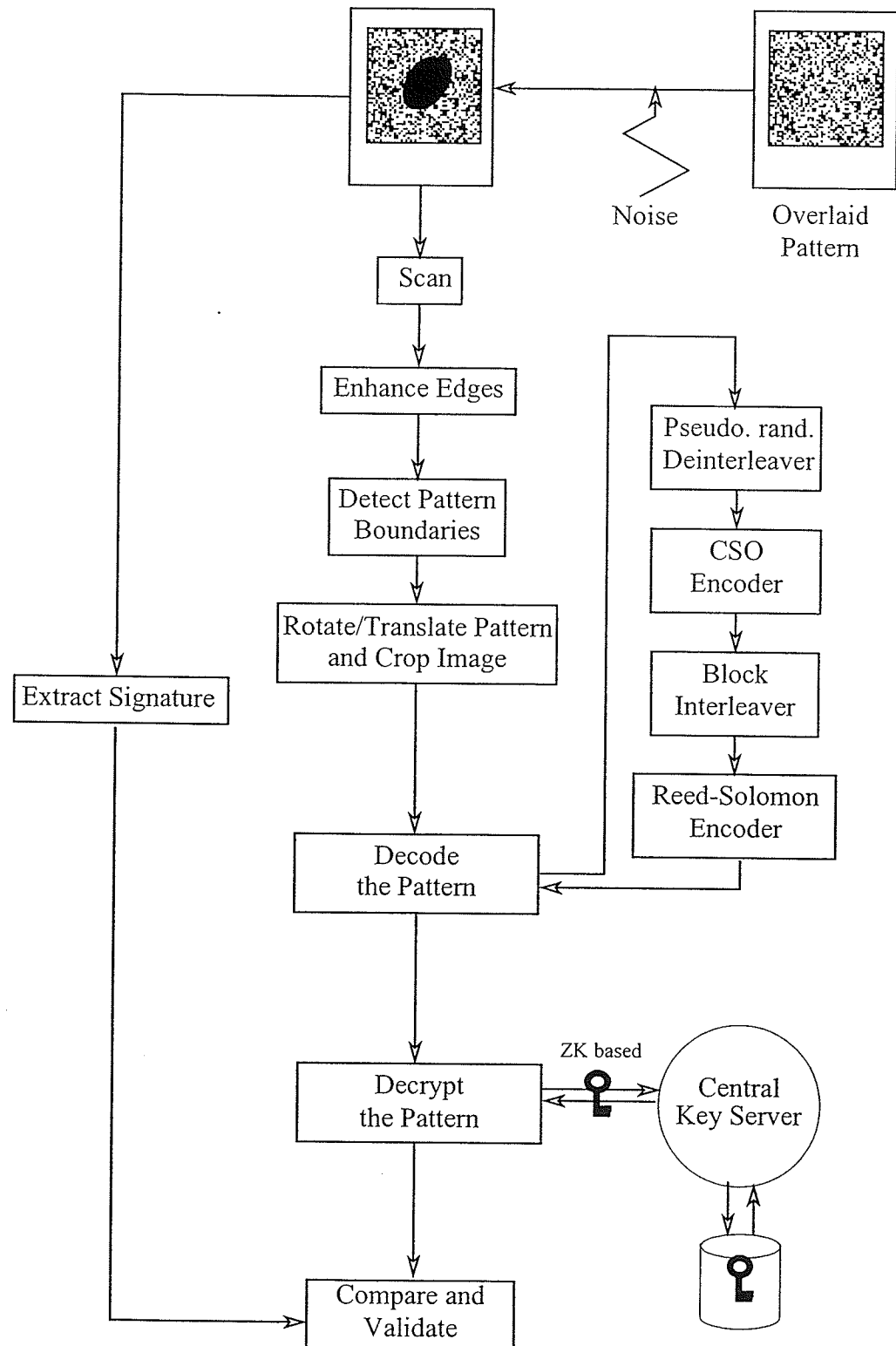
**A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University
of Manitoba in partial fulfillment of the requirements of the degree
of**

MASTER OF SCIENCE

ALEXIS DENIS ©2001

Permission has been granted to the Library of The University of Manitoba to lend or sell copies of this thesis/practicum, to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film, and to University Microfilm Inc. to publish an abstract of this thesis/practicum.

The author reserves other publication rights, and neither this thesis/practicum nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.



ABSTRACT

Numerous printed documents, such as passport, paper money, lottery tickets, carry sensitive and valuable information. This thesis uses techniques from the communications, cryptography, and image processing domains to detect alterations to the printed document and prevent the creation of forged documents.

The security of the document is achieved by overlaying a pattern that contains information about the underlying document using a special ink such as fluorescent ink. The resilience of the pattern against random and compound errors in the paper channel is achieved by using a double interleaved concatenation of a Reed-Solomon (RS) code with a self-orthogonal majority decodable convolutional code. A security system design is provided which is scalable according to the value of the document. The pattern is read using a gray level scanner, and located using an edge filter and a line detection algorithm.

The theoretical performance of the concatenated code is tested under Gaussian noise showing that the higher constraint length convolutional codes, larger block length RS codes, and symbol interleaver perform up to a bit error rate of $10^{-6.5}$ at 2 dB. The demodulation technique locates the pattern with an SNR as low as 2 dB. Under burst noise, the concatenated pattern handled up to 15% erasure and the pattern with RS alone up to 34% for RS on the Galois field of characteristic 6 and with input block length 26.

ACKNOWLEDGMENTS

I dedicate this thesis to my wife Venus and my baby girl Alyssa. Venus' constant support over the years has made many of my achievements possible. Venus I could not have made it through without you. Thank you for coping with all those years of research. Alyssa, you've been an inspiration for the few month you've been in my life and a motivation to finish this thesis.

I would like to thank my advisor, Dr. W. Kinsner, for suggesting the topic of the thesis and guiding me along the way.

I would also like to acknowledge everyone in the Delta Research Group, past and present, including Richard Dansereau, Hongjing Chen, Rasekh Rifaat, Luotao Sun, Tina Ehtiati, Jonathan Greenberg, Steven Miller, Fan Mo, Reza Fazel, Bin Huang, Jin Chen, Hong Zhang. All of these people played important roles in my development as a graduate student, researcher, and friend.

I would also like to thank and acknowledge the support of Pollard Banknote Limited for their contribution that made this thesis possible, the Natural Sciences and Engineering Research Council (NSERC) of Canada, and the University of Manitoba.

TABLE OF CONTENTS

I. INTRODUCTION	1
Motivation.....	1
Printed Documents Security	1
Validation System.....	5
Thesis Organization	6
II. BACKGROUND.....	8
Introduction.....	8
Communication System.....	8
Source Encoding	9
Channel Encoding.....	9
Modulator.....	10
Transmission Channel.....	10
Demodulator	10
Channel Decoder.....	11
Source Decoder.....	11
Summary	11
Cryptography	11
Confidentiality	12
One-Time Pad Cipher.....	14

Additive Stream Ciphers	14
Data Integrity	18
Authentication and Identification	20
Summary	26
Forward Error Correcting Codes	27
Introduction.....	27
Algebra of Finite Fields	34
Reed Solomon Codes.....	37
Encoding.....	39
Decoding.....	40
Convolutional Codes.....	41
Introduction	41
Self-Orthogonal Convolutional Codes	45
Threshold Decoding	47
Summary	53
Barcodes.....	53
1D Barcodes.....	56
2D Barcodes.....	56
Summary	58
Random Number Generators	58
Linear Congruential Generator	59
Encryption Based RNG	59

RNG Test Procedures	60
Chi Square Test	60
Spectral Test	62
Summary	62
Digital Image Processing	62
General Concepts	62
Edge Detection	63
Gradient Operator	65
Roberts Operator	66
Prewitt Operator	67
Sobel Operator	67
Laplacian Operator	68
Canny Operator	69
Deriche Operator	73
Line Tracking Algorithms	79
Radon Transform	79
Hough Transform	82
Summary	83
 III. SECURITY SYSTEM, CODE, AND PATTERN DESIGN.....	84
Introduction.....	84
Scalable Security System.....	84

System Design for Small Value Documents.....	84
System Design for Medium Value Documents	86
System Design for High Value Documents.....	87
Summary	88
Double Interleaved Concatenated RS and Convolutional Self-Orthogonal Code....	88
Square Pattern Modulator with Threshold Demodulator.....	92
Summary	96
IV. DESIGN OF EXPERIMENTS.....	98
Introduction.....	98
Code Performance.....	98
Pattern Functional Testing.....	103
Summary	105
V. EXPERIMENTAL RESULTS AND DISCUSSION.....	106
Random Number Generator Tests	106
Linear Congruential Generator	106
Pseudo DES RNG	112
Code Under AWGN.....	112
Non Coded Transmission.....	115
Convolutional Code Performance.....	116
RS Performance	119

Interleaver Type and Performance.....	123
Code Performance with and without Interleaver.....	123
Random Interleaver vs. Block Interleaver Performance	124
Bit Block Interleaver vs. Symbol Block Interleaver	128
CSOC and RS choice.....	128
RS and Concatenated Code Comparison.....	131
Concatenated Code and RS Code Timing	132
Summary.....	133
Decoding Procedure Testing.....	134
Scanning Resolution of Rotated and Unrotated Pattern	134
Line Detection Tests	137
Demodulation Theory and Choice.....	144
Error Altered Pattern Testing.....	148
Gaussian Noise Testing	148
Burst Noise	151
Summary.....	156
VI. CONCLUSIONS AND RECOMMENDATIONS	158
Conclusions.....	158
Contributions	159
Recommendations for Future Work	160
*A. SOURCE CODE	A1

Debugger.....	A1
Buffer.....	A1
coDec	A9
RS.....	A11
RSEncoder	A14
RSDecoder	A23
interleaver	A40
interleaver_rand	A45
deinterleaver_rand	A49
interleaver8x	A53
convolEncoder	A57
convolDecoder	A61
Interleav_symb.....	A67
random_test.....	A71
gasdev	A75
factor	A76
patternDim	A92
pattern1	A94
depattern1	A101
demodulator	A105
noisify	A107
binToImg	A110

rand_test.....	A113
treatXsqFile.....	A119
scannerSimulator	A121
 B. PATTERNS	 B1

LIST OF FIGURES

Fig. 1.1	Traditional security techniques: (a) ghost printing; (b) embossment; (c) iridescent inks; (d) holograms; (e) UV printing; (f) rainbow printing; and (g) microprinting.	3
Fig. 1.2	Proposed document validation system.....	6
Fig. 2.1	Communication system diagram.	9
Fig. 2.2	Confidential (secure) communication system.....	13
Fig. 2.3	OFB mode for block cipher.	17
Fig. 2.4	Data integrity with hash functions and ciphers.....	19
Fig. 2.6	Geometrical representation of the relationship between correcting power and minimum distance of the code.	29
Fig. 2.5	Three repetition code geometrical interpretation.....	29
Fig. 2.7	ECC hierarchy.....	31
Fig. 2.8	Encoding of (a) block codes and (b) sliding window codes.	31
Fig. 2.9	(2,1) N=2 systematic convolutional encoder.	42
Fig. 2.10	(a) Trellis diagram, (b) FSM for a (2,1) N=2 code.	44
Fig. 2.11	Tree representation of a (2,1) N=2 code.	44
Fig. 2.12	Threshold decoder circuit.	52
Fig. 2.13	1D barcode structure.	54
Fig. 2.14	Sample of 1D barcodes: (a) UPC-A; (b) Code 39; (c) Codabar; (d) Code 11; (e) EAN 13; (f) EAN 8; (g) interleaved 2 of 5; (h) UPC-E; (i) Code 39 and; (j) Code	

128.	55
Fig. 2.15 2D barcodes examples: stacked: (a) 16K; (b) PDF417; (c) SuperCode; matrix: (d) Aztec; (e) Code 1; (f) UltraCode; (g) DataMatrix; and an example of unique 2D barcode: (h) 3Di.	57
Fig. 2.16 Successive edge derivatives.	64
Fig. 2.17 Image matrix for edge detection algorithms.	65
Fig. 2.18 Basic gradient approximation convolution matrices.	66
Fig. 2.19 Roberts cross-gradient convolution matrices.	66
Fig. 2.20 Prewitt convolution matrices.	67
Fig. 2.21 Sobel convolution matrices.	68
Fig. 2.22 Laplacian operator.	69
Fig. 3.1 Cryptosystem schematic for small value documents.	85
Fig. 3.2 Cryptosystem schematic for medium value documents.	86
Fig. 3.3 Cryptosystem schematic for high value documents.	87
Fig. 3.4 Block diagram of the designed code.	91
Fig. 3.5 Decoding process	93
Fig. 4.1 Uniformly spaced 3 bit quantizer.	100
Fig. 4.2 Decoding process with experiments legends.	102
Fig. 5.1 rand48 random bit test, theoretical and experimental results.	107
Fig. 5.2 rand48 mod 3 test, theoretical and experimental results.	108
Fig. 5.3 rand48 3LSB 3-tuples test, theoretical and experimental results.	108
Fig. 5.4 rand48 equidistribution test, theoretical and experimental results.	109

Fig. 5.5	rand48 8LSB pairs test, theoretical and experimental results.....	109
Fig. 5.6	pDES random bit test, theoretical and experimental results.....	113
Fig. 5.7	pDES mod 3 test, theoretical and experimental results.	113
Fig. 5.8	pDES equidistribution test, theoretical and experimental results.....	114
Fig. 5.9	pDES 3LSB 3-tuples test, theoretical and experimental results.	114
Fig. 5.10	pDES 8LSB pairs test, theoretical and experimental results.	115
Fig. 5.11	Non-coded data transmission using bipolar binary modulation theoretical and simulation performance	116
Fig. 5.12	(2,1) CSOC performance with constraint length 2, 7, and 18.	117
Fig. 5.13	RS performance for RS(7,3), RS(15,7), RS(31,13), and RS(63,27).....	118
Fig. 5.14	Theoretical bound and simulated RS(63,27) and RS(7,3) performance.....	121
Fig. 5.15	Concatenated codes simulated performance compared to expected performance with (2,1), N=18 CSOC, and (a) RS(7,3), (b) RS(15,7), (c) RS(31,13), and (d) RS(63,27).....	122
Fig. 5.16	Concatenated codes performance for various block interleavers with (2,1), N=18 CSOC, and (a) RS(7,3), (b) RS(15,7), (c) RS(31,13), and (d) RS(63,27).	123
Fig. 5.17	Random interleaver vs. block interleaver.	125
Fig. 5.18	Concatenated code with a double size random interleaver (512 bits) vs. 16x16 block interleaver.	126
Fig. 5.19	Symbol interleaver vs. block interleaver for RS(15,7), N=18.....	127
Fig. 5.20	Concatenated code performance with RS(15,7), N=7 and N=18.	129

Fig. 5.21	(a) $\text{conc}(m=3, t=2, N=18, I=16, m_i=3)$ with $\text{RS}(m=3, t=3)$, (b) $\text{conc}(m=4, t=4, N=18, I=16, m_i=4)$ with $\text{RS}(m=4, t=6)$, (c) $\text{conc}(m=5, t=9, N=18, I=14, m_i=5)$ with $\text{RS}(m=5, t=11)$, and (d) $\text{conc}(m=6, t=18, N=18, I=8, m_i=6)$ with $\text{RS}(m=6, t=23)$	131
Fig. 5.22	Example of patterns: (a) $\text{RS}(3,1) \text{ Conv}(18)$; (b) $\text{RS}(3,1) \text{ Conv}(7)$; (c) $\text{RS}(6,5)$ $\text{Conv}(2)$; and (d) $\text{RS}(4,3) \text{ Conv}(18)$	135
Fig. 5.23	Scanned patterns at different printing and scanning resolutions: (a) original pattern printed at 37 ppi; (b) scanned pattern in B&W at 37 dpi; (c) gray level at 37 dpi; (d) B&W at 75 dpi; (e) gray level at 75 dpi; (f) gray level at 100 dpi; and (g) gray level at 150dpi.	136
Fig. 5.24	(a) Scanned rotated image and (b) image after transform.	137
Fig. 5.25	Hough transform time (a) against number of pixels in the input file and (b) number of pixels required in the output file.	138
Fig. 5.26	Hough transform of rotated pattern.	141
Fig. 5.27	Hough transform of translated pattern.	142
Fig. 5.28	KUIM line identification process: (a) original image, (b) after Sobel, (c) Hough transform, (d) Hough thresholded, (e) image from (b) after averaging, and (f) identified lines.....	143
Fig. 5.29	Misaligned pixel and scanning grid when scanning at twice the printing resolution.	144
Fig. 5.30	(a) Scanned pattern and (b) simulated scanning pattern.	146
Fig. 5.31	Demodulation grid over the scanned pattern.	147

Fig. 5.32	(a) original image, (b) identified lines, and (c) associated Hough transform.	149
Fig. 5.33	Demodulated patterns after burst noise is applied (a) $m=5$, $t=15$, (b) $m=6$, $t=7$, and $m=6$, $t=20$	151
Fig. 5.34	Difference between the decoded patterns with and without noise, (a) for the pattern, (b) after random deinterleaver, (c) after the convolutional decoder, and (d) after the symbol interleaver.	152
Fig. 5.35	Decoding process of $m=5, t=15$ (see previous figure), (a) after random deinterleaver, (b) after the convolutional decoder, and (c) after the symbol interleaver.	153
Fig. 5.36	RS pattern with symbol interleaver (a) $m=6$, $t=19$, (b) $m=5$, $t=11$, (c) $m=6$, $t=26$, and without interleaver (d) $m=6$, $t=26$	155

LIST OF TABLES

Table 2.1	Three repetition code mapping.	28
Table 2.2	Multiplication in GF(2).....	35
Table 2.3	Addition in GF(2)	35
Table 2.4	GF(8) generated by x^3+x+1	36
Table 4.1	Optimum 8-level quantizer; $E_s/N_0=-6.1$ dB; $I_{8-j}=-I_j$; for $0 < j < 4$	100
Table 5.1	Spectral test results for the BSD LCG	107
Table 5.2	. Timing comparison of RS and concatenated code under AWGN.	133
Table 5.3	Worst case scratch handled by equivalent rate concatenated and RS codes.	155

LIST OF ABBREVIATIONS AND ACRONYMS

1D	One-dimensional
2D	Two-dimensional
ABA	American Bankers Association
AWGN	Average white Gaussian noise
BSD	Berkeley system distribution
B&W	Black and white
BCH	Bose-Chaudhuri-Hocquenghem
BER	Bit error rate
CCD	Charge couple devices
CDF	Cumulative density function
CMOS	Complementary metal oxide semiconductor
CSOC	Convolutional self-orthogonal code
dB	Decibel
DES	Data encryption scheme
dpi	Dot per inch
ECC	Error correcting codes
FEC	Forward error correcting codes
FIR	Finite impulse response
GF	Galois field
GNU	GNU is not UNIX

LCG	L inear c ongruential g enerator
lcm	L owest c ommon m ultiple
LSB	L east s ignificant b it
MAC	M essage a uthentication c ode
MDC	M odification d etection c ode
MDS	M aximum d istance s eparable
MSB	M ost s ignificant b it
OFB	O utput f eedback m ode
pbm	P ortable b it m ap
pdf	P robability d istribution f unction
pgm	P ortable g ray m ap
PIP	P rimitive i rreducible p olynomial
pixel	P icture e lement
PK	P ublic k ey
ppi	P ixel p er i nch
PSK	P hase s hift k eeying
RNG	R andom n umber g enerator
RS	R eed-Solomon
RSA	R ivest, S hamin, A dleman
SNR	S ignal to n oise r atio
TIFF	T agged i mage f ile f ormat
UPC	U niversal p rice c ode

UV	Ultra violet
vel	Volume element
ZK	Zero knowledge

LIST OF SYMBOLS

$\delta(x)$	Dirac function
∇f	Gradient operator
c	Ciphertext
C	Ciphertext space
D_d	Decryption algorithm in public key encryption
d	Decryption key
e	Encryption key
E_e	Encryption algorithm in public key encryption
$f(x, y)$	Pixel (x, y) from an image
$g(x)$	Generator polynomial for error correcting codes
\ln	Natural logarithm, \log_e
\log	Logarithm base 10, \log_{10}
m	Cleartext
mod	Modulo function
N	Convolutional code constraint length
$p(x)$	Primitive irreducible polynomial
p_S	Probability of symbol error for error correcting codes
p_b	Probability of bit error for error correcting codes
$RS(m, t)$	Reed-Solomon code with symbol size m and error correction power of t symbols

x and y Position axes indicators

CHAPTER I

INTRODUCTION

1.1 Motivation

Even in a digital age, some valuable information is still printed on paper. Documents such as paper money, passports, ID, scratch lottery tickets, checks, contracts, are examples of valuable information printed and transmitted on paper. Forgery of printed documents is an increasing concern as people have access to low-cost, high quality reproduction methods (such as color laser printing); over 99.6% of checks can be readily reproduced on laser printers. Check fraud is to the 90s what credit card fraud was to the 80s. In 1998, bounced checks totaled US\$9.9 billion, while in 1999, there were 612 million bounced personal checks written in the U.S., totaling US\$19.9 billion [ACAI00]. Attempted check fraud losses exceeded \$2.2 billion in 1999, actual dollar losses were \$679 million for the banks alone (not including businesses losses), up from the \$512 million in 1997, according to American Bankers Association study [ABAs00]. The increase from one year to another and the losses incurred to companies and banks show that printed document security against fraud is a contemporary and important matter.

1.2 Printed Documents Security

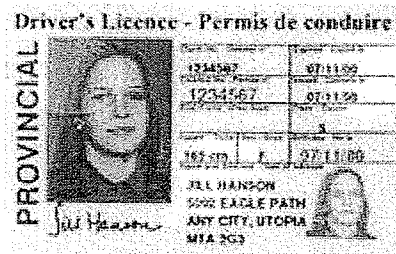
Current paper document security is based on techniques that make the documents more expensive to reproduce than the value of the printed document itself unless the

document is produced in large quantities. The security techniques are aimed to prevent forgery with easily accessible reproduction processes (such as laser printing and photocopying). The techniques involve special paper, inks, patterns or extra material added to the document. Some of the techniques are illustrated in Fig. 1.1., and are described in the following paragraphs [Benb99][Rene98].

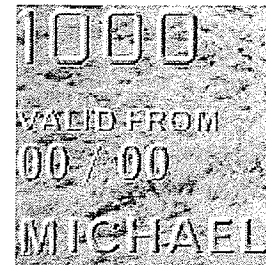
Ghost printing technology is used to create a substantial level of tamper-resistance by adding a lighter reproduction of an image on an identity document, typically in the same area of the document as personalized data. The second image appears as a light background to text data, significantly increasing the difficulty of altering the photo image or the data.

Embossed and indent-printed characters on cards require specialized equipment to make characters either protrude from or recess into the substrate of a plastic card. These characteristics provide a tactile feature.

Iridescent inks consist of either metallic or pearlescent inks that cannot be mimicked by color copiers or reproduced by scanning and reprinting. These inks change appearance when viewed at different angles.



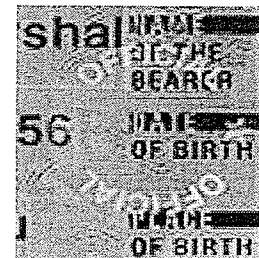
(a)



(b)



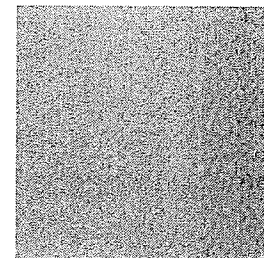
(c)



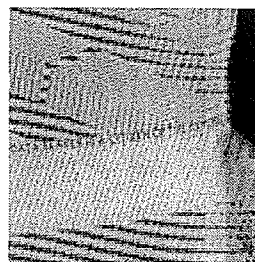
(d)



(e)



(f)



(g)

Fig. 1.1. Traditional security techniques: (a) ghost printing; (b) embossment; (c) iridescent inks; (d) holograms; (e) UV printing; (f) rainbow printing; and (g) microprinting.

Hologram security results from an image that shifts position when viewed from different angles. Holograms are not receptive to photography, photocopying or scanning,

and they require highly specialized equipment to replicate designs. Metallized holograms can be added to a pre-printed document. Transparent holograms can be placed over the photo and date to provide a high level of security.

Ultraviolet (UV) ink is a commonly accepted security feature for identity documents. This invisible printing can be viewed under a long-wave UV light source, and can be produced in a range of colors. UV text or images can be printed on a document.

Rainbow printing enhances document security with an extremely subtle shift in color across an identity document. This color shift cannot be produced accurately by a color copier or scanner.

Microprinting is created by high-resolution printing to create extremely fine, small characters that cannot be replicated with a traditional color copier or scanner. With microprinting in place, counterfeited documents can easily be detected with a standard magnifying glass.

Zero-order grating microstructures (ZOGM) are diffractive structures that change color from red to green as they are rotated about their own axis. Those structures cannot be replicated with a photocopier. The manufacturing capability for these structures is far more expensive than the one used to duplicate hologram.

Kinegrams are optically variable graphics (OVG) that animate as the angle of view to the kinegram changes. This obviously cannot be photographically copied.

The techniques described in this section are all static: once forgers find a way to reproduce the technique for little cost or a way to fake the technique, they can produce new documents or modify existing ones. Furthermore, a new technique has to be used to secure forthcoming documents reliably.

1.3 Validation System

To complement the classical static techniques, a dynamic system must be introduced. The system should identify altered documents and be dynamic so that the compromise of a set of documents does not affect upcoming documents produced with the same system. The system should also be automated and resilient to the noise produced on the paper channel.

The validation system developed in this thesis is based on unique document information encoded and encrypted with the document itself (see Fig. 1.2. for a visual representation of the procedure). This unique information could be the serial number of a banknote or a check, the characteristics of a person's fingerprint, a low resolution version of a person's ID picture, the dollar amount of a contract. The unique information from the document can be extracted by hand (visualized and keyed in) or by scanning portion of the document and extracting a unique feature from the scanned portion (e.g., OCR for serial numbers, edge location for a picture.) The unique information from the document (the signature of the document) is encrypted and encoded and is overlaid on top of the document using an ink that cannot be seen under normal lighting (such as fluorescent or

ultraviolet ink.) The document is validated by scanning the overlay pattern, decrypting it, and comparing it to the document signature.

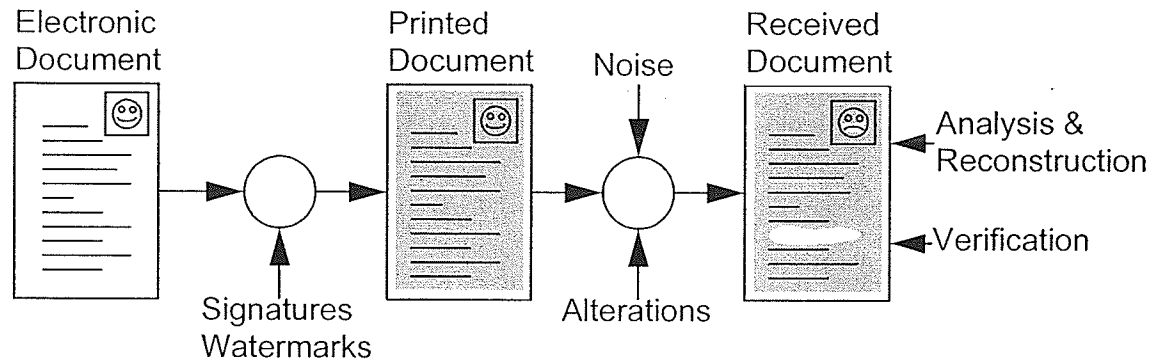


Fig. 1.2. Proposed document validation system.

Questions of speed, resilience to the noise on the printed document, limitation of the compromised documents and choice of encryption technique will be dealt with in the following chapters.

1.4 Thesis Organization

Chapter II provides the necessary background on encryption (confidentiality, data integrity, authentication), image processing (edge enhancement operator and line transforms), forward error correcting codes (convolutional, RS, and concatenated codes), barcodes (1D and 2D), and random number generators (RNG) both linear congruential generators (LCG) and cryptographic.

The system is designed in Ch. III with the choice of different levels of security, the design of code components, and the structure of the pattern.

Chapter IV describes the setup of the experiments performed on the code, the RNG, and the pattern.

Chapter V presents the results of the performance of the code under the average white Gaussian noise (AWGN) with different component sizes and types, the performance of the pattern location algorithm under Gaussian noise, the RNGs randomness test, and the pattern under burst noise with both RS alone and with the concatenated pattern.

Chapter VI concludes the thesis with recommendations for future work.

CHAPTER II

BACKGROUND

2.1 Introduction

The goal of this thesis is to design a secure and reliable communication system for the physical paper channel. The first section of this chapter defines the communication system and its components. Next a background on cryptography is developed as part of the source encoding. The third section describes forward error correcting codes used to protect the information against noise. Since random number generators (RNGs) are used in both simulation of noise and pseudo-random interleaving, a section describes two kinds of generators and the tests to evaluate the quality of random number generators. The final section gives a background on the image processing techniques used during the demodulation of the pattern.

2.2 Communication System

The elements of a one-way communication system are illustrated in Figure 2.1, and are described next.

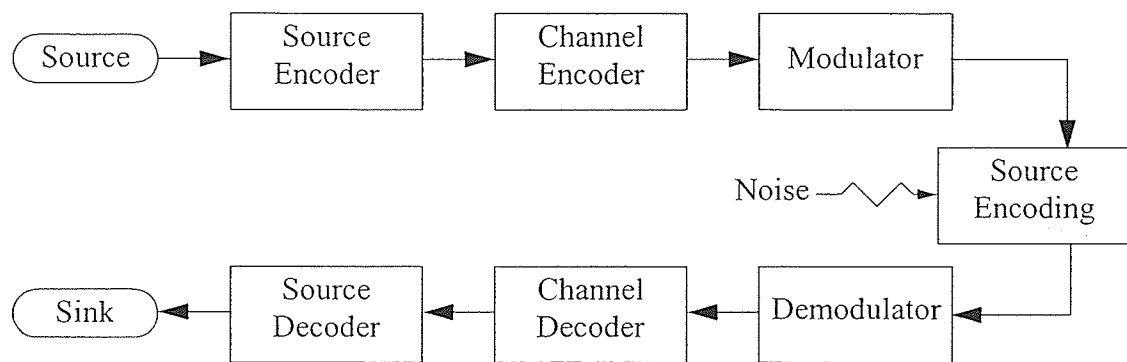


Fig. 2.1. Communication system diagram.

2.2.1 Source Encoding

The source information can be provided in an analog (continuous) or in a discrete form (either infinite or quantized). The source encoding deals with the transformation of the source information into a series of discrete symbols. This stage involves digitization of the source if it is in analog format, removal of source redundancy through compression techniques, and securing of the transmitted source with encryption techniques. This thesis assumes that the digitization and compression are already performed so that the information processed by the encrypter is already in symbolic form.

2.2.2 Channel Encoding

The channel encoder transforms the source stream from the source encoder and adds redundancy through error correcting codes as well as prepares the data in the desired format for the modulator (e.g., the channel encoder makes groups of eight bits for an 8-ary PSK modulator).

2.2.3 Modulator

The modulator involves converting the encoder output to a format suitable for transmission on the channel. A binary modulator matches bits to waveforms of equal duration.

2.2.4 Transmission Channel

The transmission channel includes the introduction of the modulated waveform into the channel, the transmission medium, and the receiving operation required to bring to the point just prior to demodulation.

The characterization of the transmission channel is important as it influences the design of the components of the communication system. The characteristics of the channel are power, bandwidth and noise.

2.2.5 Demodulator

The demodulator matches the received waveforms to a set of values to be fed to the channel decoder. The demodulator can output a definite decision on the received waveform such as a 0 or a 1 for binary data (this case is referred to as a hard-decision demodulator) or a decision with a confidence rating (soft-decision demodulator). The rating associated to the decision can, for example, be output by a matched waveform filter. In the case of symmetric binary modulation, the matched filter outputs the average of the received signal over a signal period.

2.2.6 Channel Decoder

Channel decoder transforms the digital demodulator output and uses the redundancy introduced at the coding stage to remove the noise from the signal. When designing the decoder, the main trade-off is between correcting power and speed. The error correcting power determines the desired number of errors that the code can correct in a certain frame. However, increased correction power means increased bandwidth and increased decoder complexity (thus slower decoding speed).

2.2.7 Source Decoder

The final stage reconstructs the original signal from the symbols received from the channel decoder. The signal might have been altered by digitization errors or noise over the channel that could not be corrected by the channel decoder.

2.2.8 Summary

The communication system has been introduced. The following sections give the necessary theoretical background for each component used in our communication system.

2.3 Cryptography

From the greek crypto (“hidden”) and graphein (“writing”), cryptography is the art and science of transforming information into an intermediate form which secures that information while in storage or in transit.

There are four cryptographic goals [MOVa96]:

- (i) confidentiality;
- (ii) data integrity;
- (iii) authentication and identification; and
- (iv) non-repudiation.

This thesis uses the first three goals of cryptography to secure the paper information. This section on cryptography is based on material from Schneier [Schn96] and Menezes *et al.* [MOVa96].

2.3.1 Confidentiality

The confidentiality of a message is achieved by encrypting a cleartext, m (the message) into a ciphertext, c , using an encryption scheme that only the intended user can invert to decrypt the ciphertext into the cleartext. The pair of encryption, decryption algorithms are called an encryption scheme or cipher. Figure 2.2 provides a simple model of a two-party confidential communication using encryption.

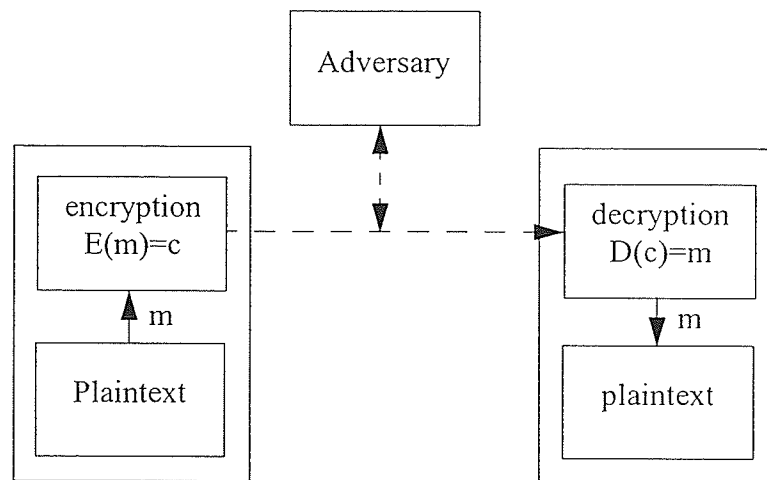


Fig. 2.2. Confidential (secure) communication system.

The class of encryption and decryption transformations can be kept secret, but the security of the entire scheme should not be based on this approach. History has shown that maintaining the secrecy of the transformations is very difficult indeed. A better approach is to base the system on algorithms that are functions of a key (i.e., E_e for encryption e in K , key space, and D_d for decryption d in K). The algorithms are then public but the keys are kept secure. A fundamental premise in cryptography is that the sets M (message space); C (ciphertext space); E_e (encryption algorithm); D_d (decryption algorithm), are public knowledge. When two parties wish to communicate securely using an encryption scheme, the only thing that they keep secret is the particular key pair (e,d) which they are using, and which they must select. Some of the key based algorithms are described in the remainder of this section.

2.3.1.1 One-Time Pad Cipher

The only unconditionally secure encryption scheme is the one-time pad scheme where the message is XORed with a random key string (a string of random characters, or one-time pad), the receiver has a copy of the one-time pad and XORs it with the received ciphertext. Each one-time pad can only be used once. The one-time pad cipher can be shown to be theoretically unbreakable. If a cryptanalyst (person who tries to break the encryption scheme) has a ciphertext string $c_1c_2\dots c_t$ encrypted using a one-time pad which has been used only once, the cryptanalyst can do no better than guess at the plaintext being any binary string of length t (i.e., t -bit binary strings are equally likely as plaintext). It has been proven that to realize an unbreakable system requires a random key of the same length as the message. This reduces the practicality of the system in all but a few specialized situations. Reportedly, until very recently the communication line between Moscow and Washington was secured by a one-time pad. Transport of the key was done by trusted courier.

One-time pads are part of a family of algorithms called additive stream cipher. The next section describes additive stream ciphers with key sizes smaller than the one-time pad cipher.

2.3.1.2 Additive Stream Ciphers

Stream ciphers encrypt individual characters (usually binary digits or bytes) of a plaintext message one at a time, using an encryption transformation which varies with

time. Additive stream ciphers XOR a string of symbols with the message. The variability comes from a different string every time a message is encrypted. By contrast, block ciphers encrypt groups of characters of a plaintext message using a fixed encryption transformation. The additive stream cipher's security depends entirely on the string generator. If the string is constant or periodic with a period less than the message, the security of the system is negligible; e.g., if the string is constant and an attacker has access to a cleartext and a ciphertext, the attacker only needs to XOR the cleartext and the ciphertext to recover the string and decode any other incoming message. If the string generator creates a series of random numbers with infinite period, the stream cipher is then a one-time pad cipher and has perfect security. A realizable string generator that has good security lies in between those two extremes: it outputs a string that looks random over a long period but is actually a deterministic string that can be reproduced at decryption time. The closer the string generator's output is to random, the harder time cryptanalysts will have breaking it.

In situations where transmission errors are highly probable, stream ciphers are advantageous because they have no error propagation. They can also be used when the data must be processed one symbol at a time (e.g., if the equipment has no memory or buffering of data is limited). To make stream ciphers more tractable than the one-time pad, a "pseudo-random" pad is generated using an encryption algorithm and a key. The receiver only needs the key to generate the whole pad and decrypt the message. Of course the stream cipher is only as secure as the encryption algorithm that generates the pseudo-random pad.

2.3.1.2.1 Block Cipher in Output Feedback (OFB) Mode

A block cipher is a function which maps n -bit plaintext blocks to n -bit cipher-text blocks, where n is called the block length. The function is parameterized by a k -bit key K , taking values from a subset κ (the key space) of the set of all k -bit vectors V_k . It is generally assumed that the key is chosen at random. Use of plaintext and ciphertext blocks of equal size avoids data expansion.

If a message is encrypted r bits at a time, the cipher works on n bits ($1 \leq r \leq n$), the output feedback (OFB) mode works in the following way (see Fig. 2.3.) [MOVa96]:

INPUT: k -bit key K ; n -bit IV; r -bit plaintext blocks $x_1 \dots x_u$ $1 \leq r \leq n$.

Encryption: $I_1 \leftarrow IV$. For $1 \leq j \leq u$, given plaintext block x_j :

- (a) $O_j \leftarrow E_k(I_j)$ (Compute the block cipher output.)
- (b) Assign t_j the r leftmost bits of O_j (Assume the left most is identified as bit 1.)
- (c) $c_j \leftarrow x_j \oplus t_j$ (Transmit the r -bit ciphertext block c_j .)
- (d) $I_{j+1} \leftarrow O_j$ (Update the block cipher input for the next block.)

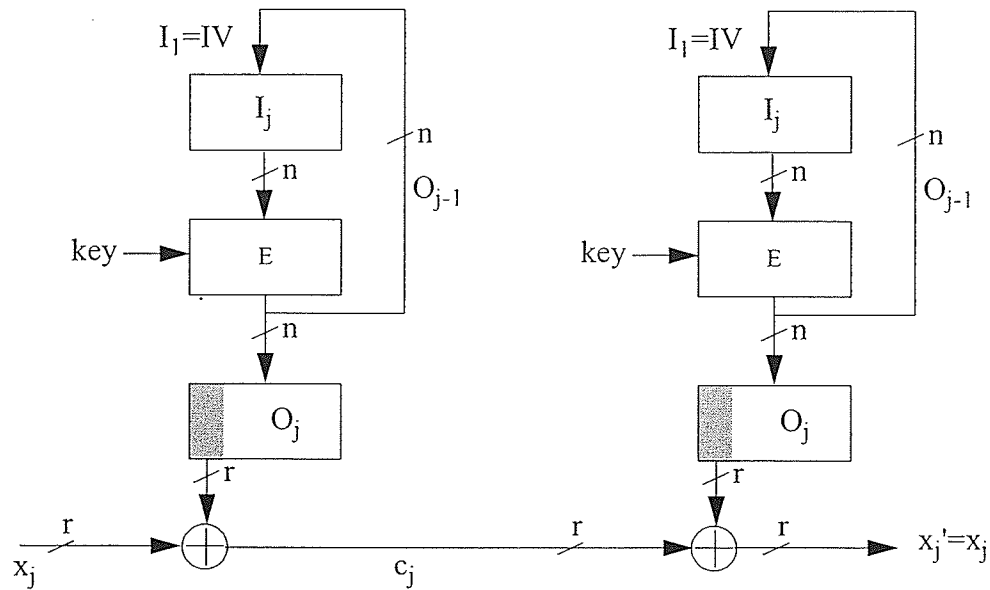


Fig. 2.3. OFB mode for block cipher.

Decryption: $I_1 \leftarrow IV$. For $1 \leq j \leq u$, upon receiving c_j :

$$c_j \leftarrow x_j \oplus t_j, \text{ where } t_j, O_j, \text{ and } I_j \text{ are computed as above.}$$

The top part of Fig. 2.3. is the string generator whose result gets XORed with the message. The string is totally independent from the message and the encryption of the message changes with the IV which is the variability property of stream ciphers. The OFB mode does not self synchronize after the loss of bits, additional framing information has to be provided in this mode. If $r = n$, the throughput of the algorithm is the same as the throughput of the block cipher it is based on.

2.3.2 Data Integrity

Data integrity is the property whereby data has not been altered in an unauthorized manner since the time it was created, transmitted, or stored by an authorized source. Cryptographic techniques for data integrity rely on specific types of hash functions. A hash function (in the unrestricted sense) is a function h which has, as a minimum, the following two properties:

P1. Compression — h maps an input x of arbitrary finite bitlength, to an output $h(x)$ of fixed bitlength n .

P2. Ease of computation — given h and an input x , $h(x)$ is easy to compute.

The hash functions can be used as:

C1. modification detection codes (MDCs)

MDCs are a subclass of unkeyed hash functions, with the following additional properties:

- (i) one-way hash functions (OWHFs): for which, finding an input that hashes to a pre-specified hash-value is difficult; and
- (ii) collision resistant hash functions (CRHFs): for which, finding any two inputs having the same hash-value is difficult.

C2. message authentication codes (MACs)

The purpose of a MAC is (informally) to facilitate, without the use of any additional mechanisms, assurances regarding both the source of a message and

its integrity. MACs are a subclass of keyed hash functions.

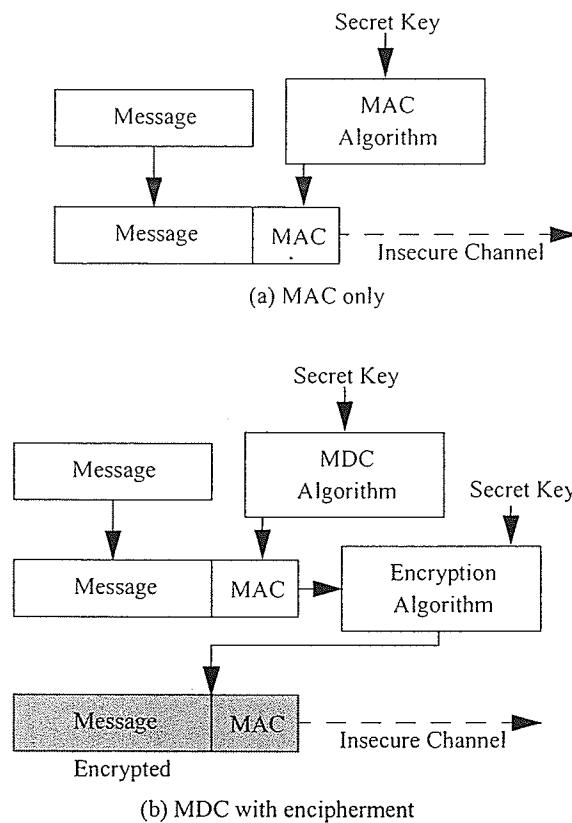


Fig. 2.4. Data integrity with hash functions and ciphers.

It is generally assumed that the algorithmic specification of a hash function is public knowledge. Thus in the case of MDCs, given a message as input, anyone may compute the hash-result; and in the case of MACs, given a message as input, anyone with knowledge of the key may compute the hash result. MDCs and MACs used for encryption have the following properties:

(Pi) MDCs and MACs are used for data authentication in several ways, two of them are described in Fig. 2.4.. The goal is to create additional data based on

the message that is different if recomputed once the message is altered.

(Pii) The reproduction of the additional data should only be possible with the knowledge of a secret key.

The MAC has the required properties. The MDC changes with the message and it is hard to find two messages that compute to the same MDC. However, anybody who knows the MDC algorithm can change the message, recompute the MDC and send the changed message with the new MDC. If the MDC and the message are encrypted using a key cipher, the MDC becomes a MAC.

MACs and MDCs are specially important when using stream ciphers. As the message is XORed with the encryption string, if an attacker flips a bit in the transmitted stream, the same bit will be flipped in the decrypted message as the encryption string is not changed. Without breaking the cipher, the attacker is capable of modifying the received message. If the attacker knows the message format, he is able to alter specific data such as the dollar amount on a check.

2.3.3 Authentication and Identification

Authentication and identification allows two parties to gain assurance of the identity of the other to prevent impersonation. The techniques are based on the owning of a secret by the claimant that allows the verifier to identify it. There are two types of identification schemes: fixed password schemes and challenge-response scheme. Password schemes are referred to as weak authentication scheme and are not reviewed in this thesis.

The idea of cryptographic challenge-response protocols is that one entity (the claimant) “proves” its identity to another entity (the verifier) by demonstrating knowledge of a secret known to be associated with that entity, without revealing the secret itself to the verifier during the protocol.

One technique is based on symmetric key algorithm. Both users share a common secret key. For each message sent from one to another, a timestamp concatenated with the a user identification token (e.g, its name) is sent on the channel with the message. The timestamp prevents replay of the user identifier by a different user. The user token allows for identification of the message sender. This technique uses symmetric key ciphers which provide faster encryption. However, both the sender and the receiver have to share a secret key.

A second technique is based on public-key ciphers. The secret owned by the claimant is its private key. The verifier uses the claimant's public key to send it some challenges that can only be answered by someone with the secret. To prove its identity, the claimant can solve two kind of challenges:

- the claimant decrypts a challenge encrypted under its public key;
- the claimant digitally signs a challenge.

An example of a protocol based on public-key ciphers for identification is a modified Needham-Schroeder PK protocol:

$$1. \quad A \rightarrow B : P_B(r_1; A)$$

$$2. \quad A \leftarrow B : P_A(r_1; r_2)$$

$$3. \quad A \rightarrow B : r_2$$

where P_A and P_B are both public key encryption with A's and B's key respectively and r_1, r_2 are random numbers generated during the protocol. The use of public key is advantageous when a number of users need to identify each other as each user needs only his private and a central server can store the public keys of each user. The database can be publicly readable but has to have write access privileges. In the case of private key algorithm, all users have to have all other user's private keys and can impersonate any user at any time by just using their private key.

The third and final technique is based on zero-knowledge (ZK) protocols. A zero-knowledge protocol allows a proof of the truth of an assertion, while conveying no information (this notion can be quantified in a rigorous sense) about the assertion itself other than its actual truth. In this sense, a zero-knowledge proof is similar to an answer obtained from a (trusted) oracle. The zero-knowledge property implies that a prover executing the protocol does not release any information that cannot be computed in polynomial time from public information alone. Thus, participation does not increase the chances of subsequent impersonation. A recorded ZK interactive proof conveys no guarantees upon playback (which is a flaw of a basic key based identification that do not include timestamps to prevent playback). Interactive proofs convey knowledge only to

(interactive) verifiers able to select their own random challenges. The properties of the ZK protocol makes it interesting compared to other public-key protocols for the following reasons:

1. no degradation with usage: protocols proven to have the ZK property do not suffer degradation of security with repeated use, and resist chosen-text attacks. This is perhaps the most appealing practical feature of ZK techniques. A ZK technique which is not provably secure may or may not be viewed as more desirable than a PK technique which is provably secure (e.g., as difficult as factoring).
2. encryption avoided: many ZK techniques avoid use of explicit encryption algorithms. This may offer political advantages (e.g., with respect to export controls).
3. efficiency: while some ZK-based techniques are extremely efficient (e.g., Fiat-Shamir uses from about 11 to about 30 steps where full exponentiation in unoptimized RSA takes 768 steps), protocols which formally have the zero-knowledge property typically have higher communications and/or computational overheads than PK protocols which do not. The computational efficiency of the more practical ZK-based schemes arises from their nature as interactive proofs, rather than their zero-knowledge aspect.
4. unproven assumptions: many ZK protocols (“proofs of knowledge”) themselves rely on the same unproven assumptions as PK techniques (e.g., the intractability of factoring or quadratic residuosity).

5. ZK-based vs. ZK: although supported by prudent underlying principles, many techniques based on zero-knowledge concepts fall short of formally being zero-knowledge and/or formally sound in practice, due to parameter selection for reasons of efficiency, or for other technical reasons. In fact, many such concepts are asymptotic, and do not apply directly to practical protocols.

Most practical ZK-based protocols are three-move zero-knowledge protocols:

- $A \rightarrow B$: witness
- $A \leftarrow B$: challenge
- $A \rightarrow B$: response

The following Fiat-Shamir protocol is not practically used because of its communication inefficiency, however it illustrates the ZK protocol concepts very clearly:

1. One-time setup.

(a) A trusted center T selects and publishes an RSA-like modulus $n = pq$ but keeps primes p and q secret.

(b) Each claimant A selects a secret s coprime to n , $1 \leq s \leq n-1$, computes $v = s^2 \bmod n$, and registers v with T as its public key.

2. Protocol actions. The following steps are iterated t times (sequentially and independently). B accepts the proof if all t rounds succeed.

(a) Commitment: A chooses a random r , $1 \leq r \leq n-1$, and sends (the witness) $x = r^2 \bmod n$ to B.

(b) Challenge: B randomly selects a bit $e = 0$ or $e = 1$, and sends e to A.

(c) Response: A computes and sends y to B, either $y = r$ (if $e = 0$) or $y = r \cdot s \bmod n$ (if $e = 1$).

(d) B rejects the proof if $y = 0$, and otherwise accepts upon verifying that $y^2 = x$ (if $e = 0$) or $y^2 = xv \bmod n$ (if $e = 1$), since $v = s^2 \bmod n$. Note that checking for $y = 0$ precludes the case $r = 0$.

The claimant proves his knowledge of the secret s without ever revealing the value of the secret. This protocol security is based on the assumption that it is hard to find a square root modulo n : an adversary can send $x = r^2 \bmod n$ with an r of his choice, however, he is not able to answer the challenge when $e = 1$ as it is equivalent to finding the square root of v . Or he could send $x = \frac{r^2}{v}$ and answer the challenge for $e = 1$ but he could not answer $e = 0$ as it is equivalent to finding the square root of x . So every time the adversary receives a challenge, he has a probability of $\frac{1}{2}$ to being able to answer it. In one case r is revealed but no information about the square root of x is disclosed and in the other case the square root of x is sent but the r used in the witness is unknown to the attacker. If t rounds are performed, the probability that the adversary answers all the challenges right without knowing the secret is 2^{-t} which decreases exponentially with t . The data that was exchanged in previous identifications does not give the adversary any

more information to guess subsequent challenges. The adversary cannot replay the protocol as it is based on the random interactive choice by the verifier of a series of bits and the claimant (here the attacker) has no control over the choice which changes every time a new identification is performed.

Zero-knowledge interactive protocols thus combine the ideas of cut-and-choose protocols (this terminology results from the standard method by which two children share a piece of cake: one cuts, the other chooses) and challenge-response protocols. A responds to at most one challenge (question) for a given witness, and should not reuse any witness; in many protocols, security (possibly of long-term keying material) may be compromised if either of these conditions is violated: as there is only two possible answers for a given witness, if a witness is reused and both possible answers are given, the adversary can go through a round with a 100% chance of success.

[MOVa96] describes the Feige-Fiat-Shamir, Guillou-Quisquater, and Schnorr ZK-based protocols.

2.3.4 Summary

Cryptographic techniques are used in a large number of secure transaction and applications. The techniques described in this section allow for:

- confidentiality through additive stream ciphers,
- data integrity using MAC and MDC with encryption,

- and identification with private key, public key, and zero-knowledge protocols.

The use of each technique is described in the design section which gives recommendations on which techniques to use according to the level of security and the lifetime of the printed material.

2.4 Forward Error Correcting Codes

2.4.1 Introduction

Error correcting codes (ECCs) involves the addition of redundancy to the transmitted data to provide the means for detecting and correcting errors. Forward error correcting codes (FEC) are ECC that assume no retransmission of the data. The channel is unidirectional: no feedback can be transmitted from the sink back to the source.

A word is a group of consecutive symbols from a specific alphabet going through the communication system. It can be of finite length (e.g., 1011, a binary word of length 4) or semi-infinite length (e.g., ABAB...., a semi-infinite length hexadecimal word). ECC take a source word and map it to a longer codeword (thus adding redundancy) so that if errors occur in the limit of the error correction capacity of the code, the original source word can be reconstructed using the introduced redundancy. A geometrical interpretation of how an ECC works can be developed using Hamming distance. The Hamming distance between two codewords of same length is the number of positions in which they differ. For example: the Hamming distance between 10011 and 01011 is $d_H(10011, 01011)=2$.

Table 2.1 Three repetition code mapping.

Source word	Codeword
0	000
1	111

The goal of an ECC is to build a set of codewords which have maximum minimum Hamming distance between each other. The minimum distance of a code is the smallest distance that can be found between all pairs of codewords generated by the code. One of the simplest codes is the 3 repetition code. The binary 3 repetition code takes single bits as source words and outputs three bits identical to the source bit as codewords (see Table 2.1).

The geometrical interpretation of the code is shown in Fig. 2.5.. If only one error occurs in the transmission of the channel codeword, the closest codeword (in term of Hamming distance) is the transmitted codeword. However, if two errors occur on distinct bits, the minimum distance decoder will make a decoding error. This code is a one error correcting code which means that all single error can be corrected correctly. Even though the code can correct some double and triple errors (e.g., if two errors occur on the same bit) it cannot correct all double and triple errors. The error correcting power of a code is:

$$t = \left\lfloor \frac{d_{min} - 1}{2} \right\rfloor \quad (2.1)$$

where d_{min} is the minimum distance of the code.

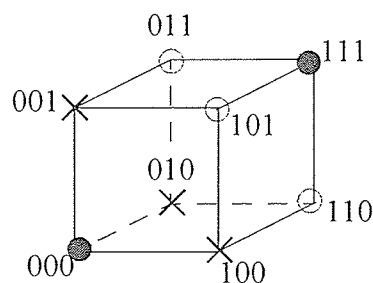


Fig. 2.5. Three repetition code geometrical interpretation.

A geometrical illustration of this rule is in Fig. 2.6.. The central dot represents the codeword. Each circle are words of distance D from the codeword (D is the number under the line), they also correspond to received words with D errors. We see that if a number of errors less or equal to $\left\lfloor \frac{d_{\min} - 1}{2} \right\rfloor$ happen, the code is still able to identify the closest codeword and thus decode the source word correctly. As an example, the three repetition code has a minimum distance of $d_{\min}(000,111)=3$ so its correcting power should be $\left\lfloor \frac{3}{2} \right\rfloor = 1$ which was verified geometrically.

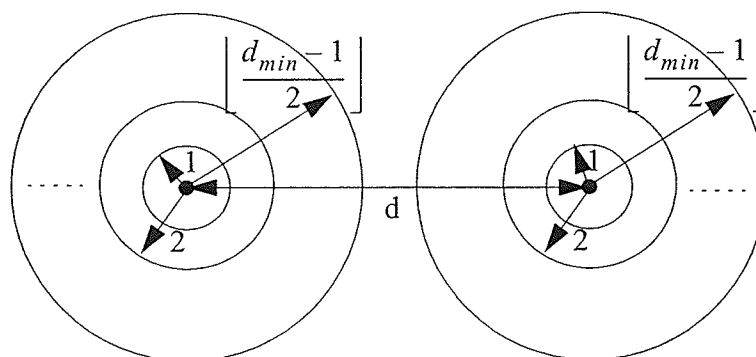


Fig. 2.6. Geometrical representation of the relationship between correcting power and minimum distance of the code.

Coming back to our definition of how ECCs work in the light of our example: ECC take a source word (e.g., a single bit) and map it to a longer codeword (thus adding redundancy, e.g., 0 to 000, 1 to 111) so that if errors occur in the limit of the error correction capacity of the code (e.g., $t=1$), the original source word can be reconstructed using the introduced redundancy.

A graphical illustration of the relationship between various classes of ECC is shown in Fig. 2.7.. The down arrow specifies a subclassing relationship. For example, tree codes are FEC but not all FEC are tree codes. The label on the arrow represent the additional property for the subclass. There are several ways to see ECC hierarchy, this is only one of them.

A tree code is defined by the following property: for any M , if two semi infinite sequences agree in their first Mk components, then their images agree in the first Mn components. Block codes have a memoryless encoder that takes finite k -length word and map them to finite n -length codeword. The ratio k/n is called the rate of the code and determines the bandwidth expansion due to the code. Sliding window codes operate on semi-infinite source words by encoding data contained in a window sliding along the word. The number of additional symbols in the sliding window at each iteration is named k and the number of output symbols is named n . The ratio k/n is the rate of the sliding window code. Even though sliding window code rates and block code rates are not identical, they both represent the channel bandwidth expansion due to the code.

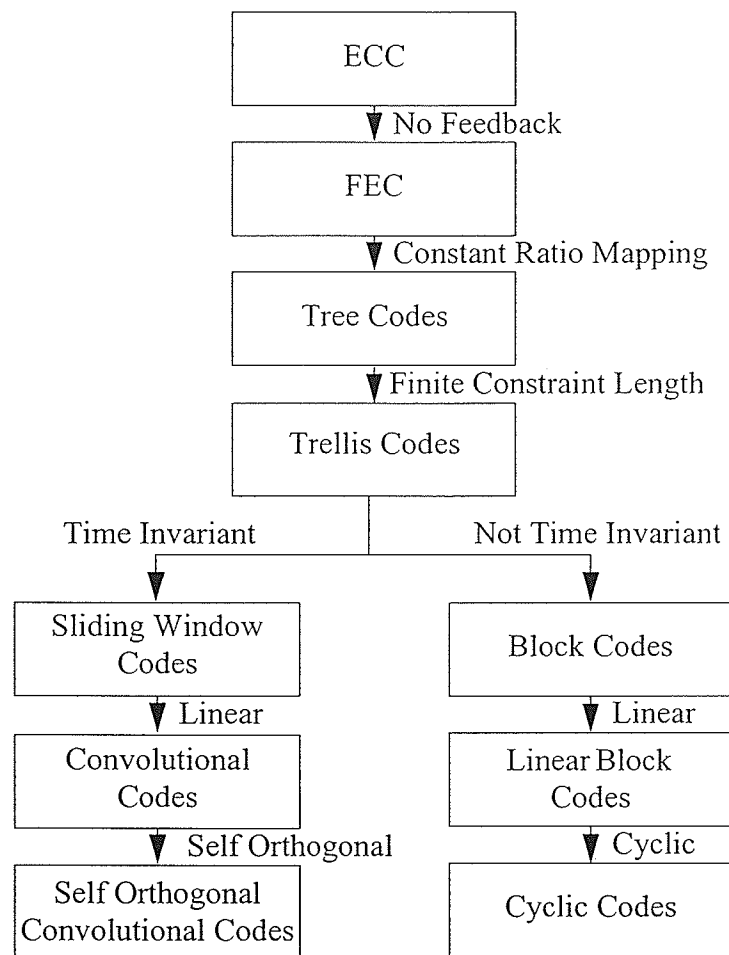


Fig. 2.7. ECC hierarchy.

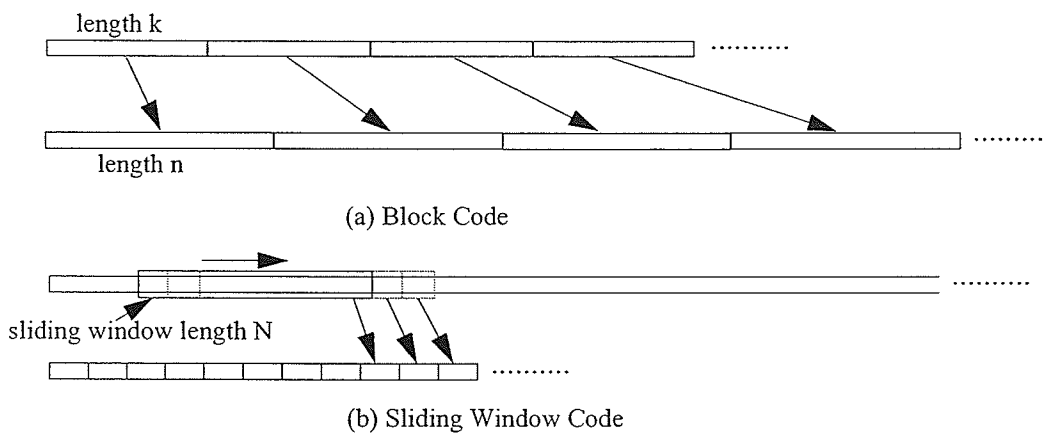


Fig. 2.8. Encoding of (a) block codes and (b) sliding window codes.

Each codeword in a block code depends only on the k input symbols and is independent from any other codeword (memoryless encoder). For a sliding window code, the sliding window generates a continuous dependence of the length of the window between the input symbols and the generated codeword (see Fig. 2.8.).

The constraint length of a code is the number of input symbols on which an output symbol is dependent. In the case of block codes, the constraint length is the size of the input word. For a sliding window code, the constraint length is the size of the sliding window.

A code is linear if the codewords form a linear vector space. This property simplifies the coding scheme as any codeword is a linear combination of a small set of reference codewords called a vector basis. It also simplifies the calculation of the performance by making the distance between two codewords equivalent to the distance between the all zero codeword and the difference of the two codewords. The minimum distance is then equal to the minimum weight of the codewords (the weight of a codeword is the number of non zero elements in the codeword). If the input of the code is also a linear vector space, the code mapping between the two spaces can be expressed in the form of a matrix. The code is then completely defined by the matrix.

A cyclic code is a linear block code C with the following property: if the codeword $(c_0, c_1, \dots, c_{n-2}, c_{n-1})$ is in C then $(c_{n-1}, c_0, \dots, c_{n-2})$ is also in C . As a consequence of this property, as all rows in the code matrix are also codewords (multiplying $(1, 0, \dots, 0)$ by the matrix generates a codeword, the result is the first line of the matrix), they are all cyclic

shifts of each other. Thus only one line is necessary to represent the complete code. This one line can be interpreted as a polynomial called the generator polynomial and the encoding process as the multiplication of the generator polynomial with the source polynomial.

An ECC can be systematic or nonsystematic. A systematic ECC is one that starts each codeword with the information symbols unmodified. The remaining symbols are called parity symbols. The systematic property is not included in the diagram because there are systematic codes at each level of the diagram (systematic convolutional codes, systematic trellis codes). For example every linear code is equivalent to a systematic linear code.

ECCs are characterized by their encoding algorithm, decoding algorithm, and their rate. As was stated in Section 2.2, the choice of an ECC is governed by the complexity of its decoding algorithm (speed, memory requirements), its allowable rate (bandwidth), and the targeted error correcting power (according to the noise characteristics of the transmission channel).

This section gave a broad overview of ECCs, their characteristics and families. The following sections describe the codes used in this thesis: Reed-Solomon codes and self-orthogonal convolutional codes with their theoretical background.

2.4.2 Algebra of Finite Fields

A finite field also called Galois Fields and written $GF(q)$ is a set of q elements for which the following arithmetic rules are defined:

R1. There are two operations defined on the elements of the field addition (+) and multiplication (.);

R2. The two operations are closed which means that the result of addition and multiplication on the elements of the field is an element of the field;

R3. The field has one and only one multiplicative identity (1) and one addition identity (0);

R4. All elements have an additive and a multiplicative inverse (except 0 for the multiplication);

R5. Associativity: $a+(b+c)=(a+b)+c$ and $(a.b).c=a.(b.c)$; commutativity: $a+b=b+a$ and $a.b=b.a$; distributivity of over +: $a.(b+c)=a.b+a.c$.

If q is a prime integer $GF(q)$ is the set of integers from 0 to q , + is the addition modulo q , . is the multiplication modulo q . $GF(q)$ is called a prime field. For example, for $q=2$, $GF(2)$ is the binary field as shown in Table 2.2 and Table 2.3.

Table 2.2 Multiplication in GF(2)

.	0	1
0	0	0
1	0	1

Table 2.3 Addition in GF(2)

+	0	1
0	0	1
1	1	0

If q is a power of a prime number (e.g., $q=p^m$), then the field elements are all possible polynomials of degree $m-1$ where the coefficients are from the prime field $GF(p)$ (polynomials over $GF(p)$). Addition and multiplication are defined modulo $p(x)$ where $p(x)$ is an irreducible polynomial (it cannot be divided by any polynomial with factors in $GF(q)$). $GF(p^m)$ is called the extension field of $GF(p)$ and p is called the characteristic of $GF(p^m)$.

This thesis deals with the $GF(2^m)$ Galois fields as they are represented by polynomials of degree $m-1$ over the binary field. In other words, elements in $GF(2^m)$ are strings of bits of length less than $m-1$.

Another representation for finite fields is through power of primitive elements. It is a property of finite fields that there exists at least one element α , called generator or primitive element, such that every non zero element in the field can be expressed as a power of this element. Polynomial representation is convenient for addition of elements while the power representation is better for multiplication (equivalent to adding the powers).

Theorem 2.1 [Lidl84]: All polynomials over $\text{GF}(p)$ have at least one extension field

$\text{GF}(p^m)$ which contains all the roots of the polynomials (this extension field is called the splitting field of the polynomial).

A primitive irreducible polynomial (PIP) $p(x)$ is an irreducible polynomial over $\text{GF}(p)$ having a primitive element α for $\text{GF}(p^m)$ as one of its roots. PIP of every degree exist over $\text{GF}(p)$ and they have the additional property that in the extension field constructed modulo $p(x)$, the field element represented by x is primitive. All these properties are proven in [Blah83]. We will now describe how to generate both polynomial and power representations for a given Galois field.

Table 2.4 $\text{GF}(8)$ generated by x^3+x+1 .

Zero & powers of x	Polynomial over $\text{GF}(2)$	Vectors over $\text{GF}(2)$
0	0	000
x^0	1	001
x^1	x	010
x^2	x^2	100
x^3	$x+1$	011
x^4	x^2+x	110
x^5	x^2+x+1	111
x^6	x^2+1	101

Starting with the unity polynomial (0...001) and successively multiplying it by x (0...010) modulo $p(x)$, where $p(x)$ is a PIP for $\text{GF}(p^m)$, we generate a translation table between the powers of x and the polynomial representation over $\text{GF}(p)$. The generation

stops when the power of x equals 1. As $p(x)$ is a PIP, x is a primitive element so the powers of x generate all elements of $GF(p^m)$. See Table 2.4 for the generation of $GF(8)$ with the PIP x^3+x+1 .

2.4.3 Reed Solomon Codes

Reed Solomon Codes (RS codes) are a subclass of BCH codes (Bose-Chauhuri-Hocquenghem codes) for which the locator field $GF(q^m)$ is the same as the symbol field $GF(q)$.

A BCH code over $F=GF(q)$ of block length n and designed distance ∂ is a cyclic code generated by a polynomial $g(x) = lcm\{m_i(x); a \leq i \leq a + \partial - 2\} \in F[x]$ whose root set contains $\partial - 1$ distinct elements $\alpha^a, \alpha^{a+1}, \dots, \alpha^{a+\partial-2}$ where α is a primitive n^{th} root of unity and a some integer. $m_i(x)$, minimal polynomials, are the smallest degree irreducible polynomial in F that have α^i as a root in $GF(q^n)$. The lcm is the least common multiple of all the polynomials which explains the statement regarding the roots of $g(x)$. A cyclic code of block length n is formed from any polynomial $g(x)$ that divides x^n-1 [MiLe85]. BCH codes are cyclic as all $m_i(x)$ divide x^n-1 (α^i is n^{th} root of unity and $m_i(x)$ is the smallest degree irreducible polynomial over F , thus $m_i(x)$ divides x^n-1).

For RS codes, because the symbol field and the splitting field are the same, all minimal polynomials are of degree 1. For a t -error correcting code, the generator polynomial for an RS code is:

$$g(x) = (x - \alpha^{m_0})(x - \alpha^{m_0+1}) \dots (x - \alpha^{m_0+2t-1}) \quad (2.2)$$

where m_0 is 0 or 1. This polynomial is always of degree $2t$.

Theorem 2.2: RS codes are maximum distance separable (MDS), i.e. $d=n-k+1$.

Proof: The number of parity symbol is equal to the degree of the generator polynomial, hence $n-k=2t$. The designed distance is defined as $\partial = 2t + 1$ and we know that $\partial \leq d$ (d is the minimum distance of the code) i.e. the code cannot correct more errors than the number assigned by its minimum distance. So $2t + 1 \leq d$, and $n - k + 1 \leq d$. The minimum distance is equal to the smallest weight codeword. The information symbol smallest weight is 1 (the only part that we can minimize) but the parity symbols can have weights as high as $n-k$ so $d \leq n - k + 1$ Hence for RS codes, $d = n - k + 1$ \square

RS codes have been used extensively for many reasons:

- Provided the block length is not excessive, there are good codes in this class;
- Relatively simple and instrumentable encoding and decoding techniques are known;
- They have a well understood distance structure;
- They have a flexible error correcting power capacity.
- They are maximum distance separable (MDS) codes

After the definition of RS codes and their general properties, we now go through their encoding and decoding procedures.

2.4.3.1 Encoding

Any codeword in a cyclic code is the sum of cyclic shifts of the generator polynomial [MiLe85] or equivalently, codewords are generated by multiplying the source words with the generator polynomial mod x^n-1 (multiplying a codeword by x modulo x^n-1 is equivalent to a cyclic shift):

$$c(x) = i(x)g(x) \bmod (x^n - 1) \quad (2.3)$$

where $i(x)$ is the source word and $c(x)$ is the codeword. As $g(x)$ divides both $i(x)g(x)$ and x^n-1 , $g(x)$ divides all the codewords. If $g(x)$ divides $a(x)$ then $a(x)=u(x)g(x) \bmod (x^n-1)$ and $a(x)$ is a codeword associated to $u(x)$. Thus a word $c(x)$ is a codeword if and only if $g(x)$ divides $c(x)$. An alternative way to generate codewords is:

$$c(x) = x^r i(x) + [x^r i(x) \bmod g(x)] \quad (2.4)$$

where r is the degree of $g(x)$. Indeed, the Euclidean division of $x^r i(x)$ by $g(x)$ is:

$$x^r i(x) = d(x)g(x) + [x^r i(x) \bmod g(x)] \quad (2.5)$$

and, in $GF(2)$, $x=-x$, hence

$$d(x)g(x) = x^r i(x) + [x^r i(x) \bmod g(x)] = c(x) \quad (2.6)$$

So, $g(x)$ divides all $c(x)$. Furthermore, all $c(x)$ are distinct (through the $x^r i(x)$ term) so Eq. (2.4) generates a systematic code equivalent to the one generated by Eq. (2.3).

2.4.3.2 Decoding

The general procedure is:

1. Generate the syndromes from the received word $1 \leq k \leq d-1$.
2. Calculate the error locator polynomial $\sigma(x)$ using the syndrome values. Exit if the error cannot be corrected.
3. Solve $\sigma(x) = 0$, the resulting roots are the error locators.
4. Determine the error values using the error locators.
5. Correct the received word.

After the channel, the received word is the codeword plus the error word: $r(x) = c(x) + e(x)$. To generate the syndromes one must evaluate the received word at $x = \alpha^k$ ($1 \leq k \leq d-1$) as α^k are the roots of the generator polynomial and all codewords are multiple of $g(x)$. Hence $S_k = e(\alpha^k)$. The error locator polynomial is calculated using the Berlekamp-Massey algorithm, see [Blah83] for more details. The root of $\sigma(x)$ are found by exhaustive search through all the values of $\text{GF}(2^m)$. There can be at most L error locators where L is the degree of $\sigma(x)$. The error value is found by solving the syndrome equation

$$S_k = \sum_{i=1}^L y_i x_i^k \quad \text{for } 1 \leq k \leq L \quad (2.7)$$

where x_i is the error locator for the i^{th} error and y_i is the erroneous value. Solving the equation is equivalent to inverting a matrix:

$$\begin{bmatrix} y_1 \\ \vdots \\ y_L \end{bmatrix} = \begin{bmatrix} x_1^1 & \dots & x_L^1 \\ \vdots & & \vdots \\ x_1^L & \dots & x_L^L \end{bmatrix}^{-1} \begin{bmatrix} S_1 \\ \vdots \\ S_L \end{bmatrix} \quad (2.8)$$

Once the error values are calculated, they are subtracted from the received word at the determined error locations.

2.4.4 Convolutional Codes

2.4.4.1 Introduction

Convolutional codes are linear, time invariant, finite constraint length trellis codes. Convolutional codes can be interpreted in two ways: as a trellis code or as a linear code with infinitely long generator matrix. The infinite matrix interpretation is the literal interpretation of the encoding process and is used for syndrome decoding of the code. The trellis viewpoint is used in decoding algorithms such as Viterbi decoding, sequential decoding, or turbo decoding.

A convolutional code encodes a stream of data by sliding a window along the stream of symbols and calculates linear combinations of the symbols to generate parity symbols sent on the channel. In the case of systematic convolutional codes, the parity symbols are sent along with the information symbols. A convolutional code is

characterized by its input length, output length, constraint length, and generator polynomial(s). The input length, k , determines how many symbols are processed at each iteration of the encoder. The output length, n , determines the number of symbols output at each iteration. The constraint length, N , is the size plus one of the sliding window (maximum size plus one in case of multiple encoding windows). The generator polynomials are the polynomials used to generate the channel symbols.

An example of a systematic $(2,1)$, $N=2$ convolutional code encoder with $g_{11}=(11)=(g_{11}(0),g_{11}(1))$ is given in Fig. 2.9..

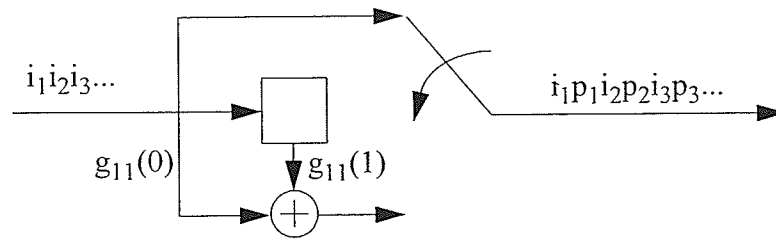


Fig. 2.9. $(2,1)$ $N=2$ systematic convolutional encoder.

The input bits are fed into the shift register (“window”) and a linear combination of the currently input bit and the bits in the register is calculated according to the generator polynomial. The output switches between the input bit and the parity bit at every iteration, outputting a string of information (bit, parity) pair. The associated generator matrix is:

$$G_{\infty} = \begin{bmatrix} 1 & g_{11}(0) & 0 & g_{11}(1) & \dots & 0 & g_{11}(N-1) & 0 & \dots & \dots \\ 0 & 0 & 1 & g_{11}(0) & \dots & 0 & g_{11}(N-2) & 0 & g_{11}(N-1) & \dots \\ \dots & \dots & 0 & 0 & \dots & \dots & 0 & g_{11}(N-2) & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & 1 & g_{11}(0) & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & 0 & 0 & 1 & g_{11}(0) & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \end{bmatrix} \quad (2.9)$$

$$= \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & \dots & 0 \\ \dots & \dots & 0 & 0 & 1 & 1 & 0 & 1 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & \dots & \dots & \dots & \dots & \dots & \dots \end{bmatrix}$$

A codeword is obtained by multiplying G_{∞} and the message: $c = mG_{\infty}$.

Convolutional codes can also be represented using a tree or a trellis. Figure 2.11 represents the code tree for our example. The trellis representation of the code considers the encoder as a finite state machine (FSM). The states of the encoder are contained in the values of the shift register (memory of the circuit). Figure 2.10 is the FSM diagram and the trellis for our code.

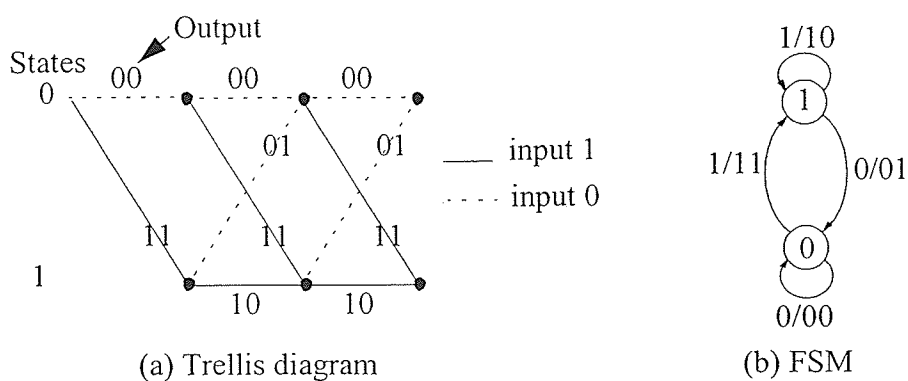


Fig. 2.10. (a) Trellis diagram, (b) FSM for a (2,1) N=2 code.

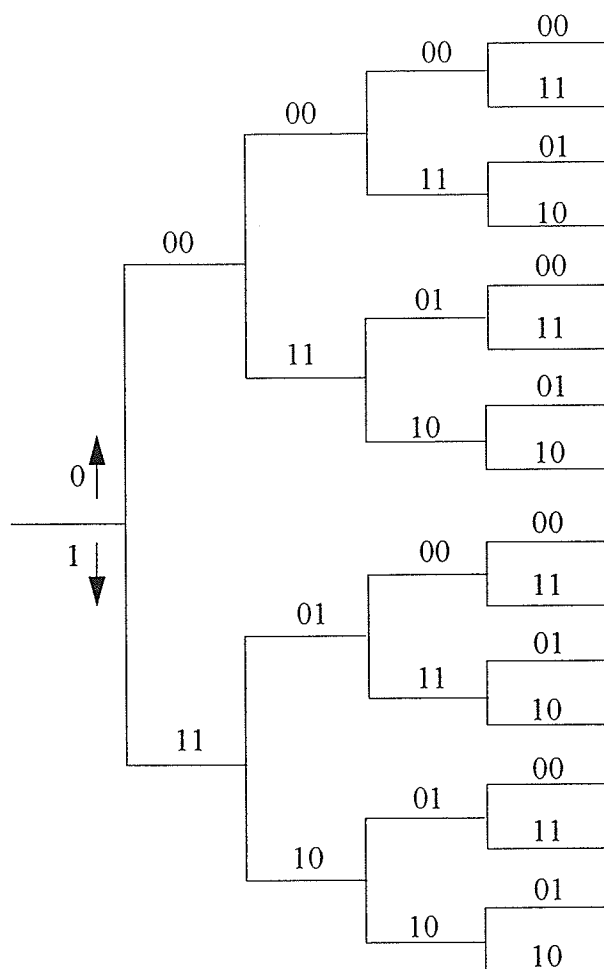


Fig. 2.11. Tree representation of a (2,1) N=2 code.

The arrows in the FSM represent the transition from one state to another. Each arrow is labeled with the input and output associated with the transition. Given a state and an input, the output of the encoder is identical at any point of time. This is the called time invariance property.

2.4.4.2 Self-Orthogonal Convolutional Codes

An (n,k) convolutional code is self-orthogonal if and only if the set of J_i syndrome symbols, which check $e_0(i)$, are orthogonal on $e_0(i)$ for $i=1,2,\dots,k$ [SLin70].

Syndromes are received symbols stripped from their information symbols; the syndromes are a sum of error symbols. The following discussion applies to convolutional codes in general, syndromes are not specific to self-orthogonal codes and there exist other decoding techniques based on syndrome evaluation.

Syndromes are obtained by encoding the received information symbols and XORing the result with the received parity symbols. The XOR operation removes the information symbols and leaves the error symbols only. If the received sequence is of the form:

$$r = r_0(1)r_0(2)\dots r_0(n)r_1(1)r_1(2)\dots r_1(n)\dots \quad (2.10)$$

$$r_i = r_0(1)\dots r_0(k)r_1(1)\dots \quad (2.11)$$

$$r_p = r_0(k+1)\dots r_0(n)r_1(k+1)\dots \quad (2.12)$$

$$r = c + e \quad (2.13)$$

The syndromes are calculated by encoding the received information bits

$$u = r_i G_\infty = c_i G_\infty + e_i G_\infty \quad (2.14)$$

$$r_p = (i G_\infty)_p + e_p \quad (2.15)$$

where i is the source word. For a systematic code $c_0 = i$. The resulting parity bits are XORed with the received parity bits.

$$s = r_p + u_p = (i G_\infty)_p + e_p + (c_0 G_\infty)_p + (e_0 G_\infty)_p = e_p + (e_0 G_\infty)_p \quad (2.16)$$

The result are the syndromes. Note that only errors and linear combination of errors remain and all information bits are gone. Equation (2.16) can also be written literally

$$s_l(j) = e_l(j+k) + \sum_{i=1}^k e_l(i) g_{ij}(0) + \dots + \sum_{i=1}^k e_{l-N+1}(i) g_{ij}(N-1) \quad (2.17)$$

for $1 \leq j \leq k$, where $g_{ij}(s)$ is the s^{th} coefficient of the generator polynomial g_{ij} .

Definition (orthogonal syndromes): Two syndromes are orthogonal on $e_0(i)$ if both contain $e_0(i)$ but they have no other error symbols in common.

For example: $g_{11}=(1010011)$

$$\begin{aligned}
s_1(1) &= e_0(1) + e_0(2) \\
s_2(1) &= e_1(1) + e_1(2) \\
s_3(1) &= e_0(1) + e_2(1) + e_2(2) \\
s_4(1) &= e_1(1) + e_3(1) + e_3(2) \\
s_5(1) &= e_2(1) + e_4(1) + e_4(2) \\
s_6(1) &= e_0(1) + e_3(1) + e_5(1) + e_5(2) \\
s_7(1) &= e_0(1) + e_1(1) + e_4(1) + e_6(1) + e_6(2)
\end{aligned} \tag{2.18}$$

$s_1(1)$, $s_3(1)$, $s_6(1)$, and $s_7(1)$ are orthogonal on $e_0(1)$. The code is self-orthogonal with $J=4$. A list of self-orthogonal convolutional codes is given in [LiWe67].

2.4.4.3 Threshold Decoding

Self-orthogonal codes can be decoded using a technique called threshold decoding developed by Massey (see [Mass63].) From our previous example, we observe that if two bits or less are in error then a majority vote on the syndromes will give the error. The decision rule is given by the following equation:

$$\sum_{l=1}^J s_l(k) > \left\lceil \frac{J}{2} \right\rceil \tag{2.19}$$

if $e_0(1)$ is zero and $\left\lfloor \frac{J}{2} \right\rfloor$ or less error occur then Eq. (2.19) returns a 0; if $e_0(1)$ is one and $\left\lfloor \frac{J}{2} \right\rfloor - 1$ or less error occur then Eq. (2.19) returns a 1. This decoding technique is called majority decoding. It is a special case of threshold decoding. Threshold decoding introduces weights and a threshold calculation:

$$\sum_{i=1}^J s_i(k)w_i > T \quad \text{with } T = \sum_{i=0}^J \frac{w_i}{2} \quad (2.20)$$

where w_i is a weighting term that is proportional to the reliability of the i^{th} syndrome. If all w_i s are equal to one we fall back on the majority decoding rule.

This matter is developed in [Mass63]. In this section a few additional explanations are given to ease the understanding of the proof.

To minimize the probability of symbol error, the decoding rule is:

$$e_m=1 \text{ if } Pr(e_m=1 | \{S_i\}) > Pr(e_m=0 | \{S_i\}) \text{ and } e_m=0 \text{ otherwise.}$$

Using Bayes' rule

$$Pr(e_m=V | \{S_i\}) = \frac{Pr(\{S_i\} | e_m=V)Pr(e_m=V)}{Pr(\{S_i\})} \quad (2.21)$$

The decoding rule becomes: Choose $e_m=1$ if and only if

$$Pr(\{S_i\} | e_m=1)Pr(e_m=1) > Pr(\{S_i\} | e_m=0)Pr(e_m=0) \quad (2.22)$$

Since the syndromes are orthogonal on the m^{th} symbol, the S_i are all independent variables. Hence the decoding rule is:

$$Pr(e_m=1) \prod_{i=1}^J Pr(S_i | e_m=1) > Pr(e_m=0) \prod_{i=1}^J Pr(S_i | e_m=0) \quad (2.23)$$

Finally, taking the logarithm of the rule with equally likely e_m

$$\sum_{i=1}^J \ln \left[\frac{Pr(S_i|e_m=0)}{Pr(S_i|e_m=1)} \right] < 0 \quad (2.24)$$

If $p_i = 1 - q_i$ denotes the probability of an odd number of ones among the noise bits exclusive of e_m that are checked by S_i , the i^{th} syndrome orthogonal to e_m , then it follows that

$$\begin{aligned} Pr(S_i=0|e_m=1) &= Pr(S_i=1|e_m=0) = p_i \\ Pr(S_i=1|e_m=1) &= Pr(S_i=0|e_m=0) = q_i \end{aligned} \quad (2.25)$$

if $S_i=1$

$$\ln \left[\frac{Pr(S_i=1|e_m=0)}{Pr(S_i=1|e_m=1)} \right] = \log \frac{p_i}{q_i} = (2S_i - 1) \log \frac{p_i}{q_i} \quad (2.26)$$

if $S_i=0$

$$\ln \left[\frac{Pr(S_i=0|e_m=0)}{Pr(S_i=0|e_m=1)} \right] = \log \frac{q_i}{p_i} = (2S_i - 1) \log \frac{p_i}{q_i} \quad (2.27)$$

so the decision rule becomes

$$\sum_{i=1}^J (2S_i - 1) \ln \frac{p_i}{q_i} < 0 \quad (2.28)$$

or

$$\sum_{i=1}^J S_i \ln \frac{q_i}{p_i} > \frac{1}{2} \sum_{i=1}^J \ln \frac{q_i}{p_i} \quad (2.29)$$

Using Eq. (2.20) and Eq. (2.29) we deduce

$$w_i = \ln \frac{q_i}{p_i} \quad (2.30)$$

Furthermore

$$p_i = \frac{1}{2} \left[1 - \prod_{j=1}^{n_i} (1 - 2\gamma_{ij}) \right] \quad (2.31)$$

where γ_{ij} is the probability that the j^{th} symbol in the i^{th} syndrome is in error, n_i is the number of symbols in S_i except e_m (refer to [Mass63] for a proof). Using Eq. (2.31) and Eq. (2.30)

$$w_i = \ln \left(\frac{1 - \frac{1}{2} \left[1 - \prod_{j=1}^{n_i} (1 - 2\gamma_{ij}) \right]}{\frac{1}{2} \left[1 - \prod_{j=1}^{n_i} (1 - 2\gamma_{ij}) \right]} \right) \quad (2.32)$$

$$= \ln \left(\frac{1 + \prod_{j=1}^{n_i} (1 - 2\gamma_{ij})}{1 - \prod_{j=1}^{n_i} (1 - 2\gamma_{ij})} \right)$$

$$\text{if } \alpha_{ij} = -\ln(1 - 2\gamma_{ij})$$

$$1 - 2\gamma_{ij} = \exp(-\alpha_{ij}) \quad (2.33)$$

$$w_i = \ln \left(\frac{1 + \exp \left(- \sum_{j=1}^{n_i} \alpha_{ij} \right)}{1 - \exp \left(- \sum_{j=1}^{n_i} \alpha_{ij} \right)} \right) \quad (2.34)$$

$$w_i = \ln \left[\coth \left(\frac{1}{2} \sum_{j=1}^{n_i} \alpha_{ij} \right) \right] \quad (2.35)$$

The decision function

$$f_d = \ln \left[\frac{Pr(e_m=0 | \{S_i\})}{Pr(e_m=1 | \{S_i\})} \right] \quad (2.36)$$

can be fed back into the decoder. The probability of error given f_d is

$$\begin{aligned} Pr(error|f_d) &= Pr(e_m=0 | \{S_i\}) \text{ if } f_d > 0 \\ Pr(error|f_d) &= Pr(e_m=1 | \{S_i\}) \text{ if } f_d \leq 0 \end{aligned} \quad (2.37)$$

if $f_d > 0$

$$e^{f_d} = \frac{Pr(error|f_d)}{1 - Pr(error|f_d)} \quad (2.38)$$

$$Pr(error|f_d) = \frac{e^{f_d}}{1 + e^{f_d}} \quad (2.39)$$

Similarly if $f_d \leq 0$

$$Pr(error|f_d) = \frac{e^{-f_d}}{1 + e^{-f_d}} \quad (2.40)$$

Thus $\forall f_d$

$$Pr(error|f_d) = \frac{e^{|f_d|}}{1 + e^{|f_d|}} \quad (2.41)$$

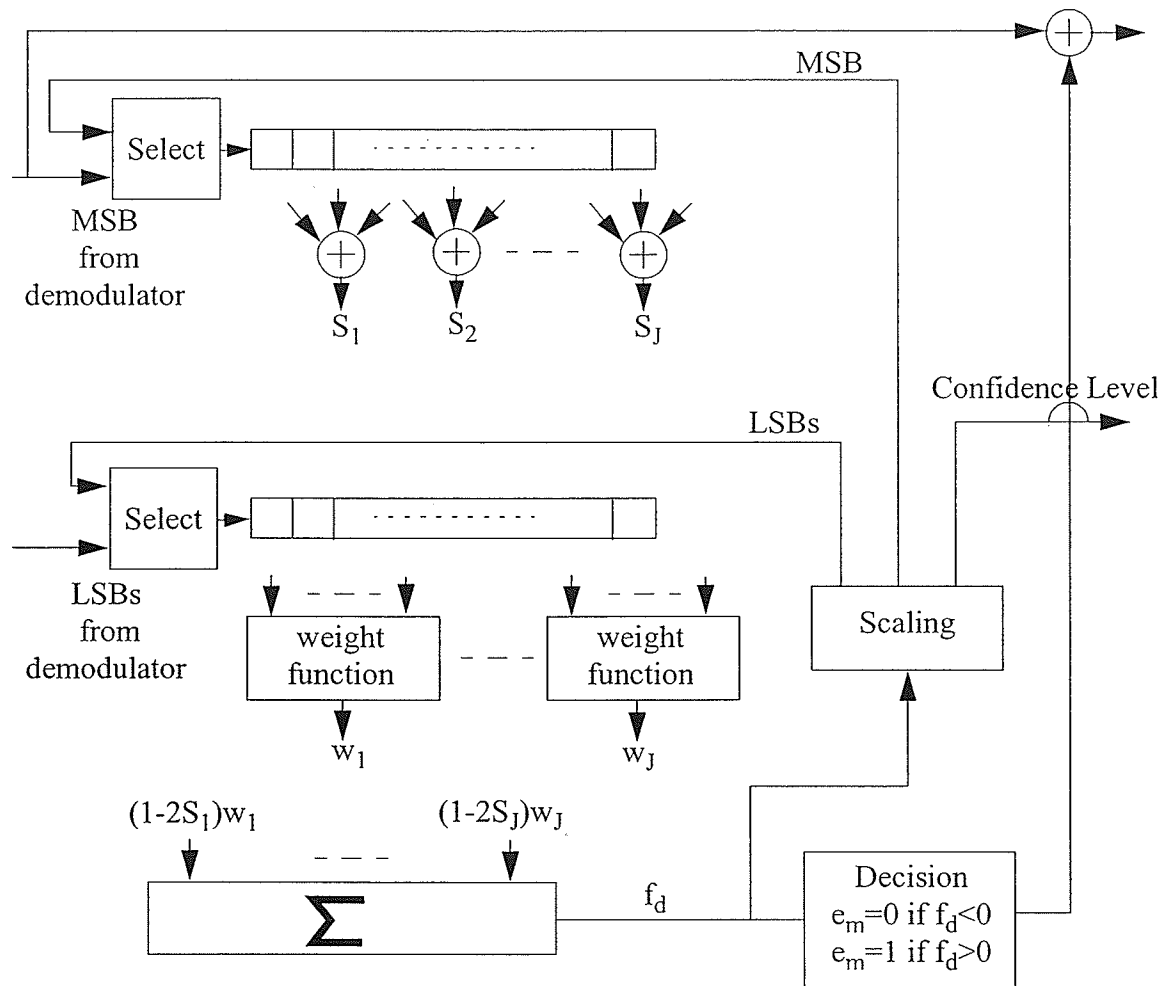


Fig. 2.12. Threshold decoder circuit.

The probability of error can be identified as γ_{ij} and fed back into the decoder. An implementation of the total decoder circuit is in Fig. 2.12..

2.4.5 Summary

This section described forward error correcting codes, both block codes and convolutional codes, with their associated coding and decoding procedures. These codes are the building blocks of the code used to protect the data on the physical paper channel.

2.5 Barcodes

Now that the error correcting code background has been acquired, the state of the art in barcode technology can be reviewed for a better understanding of the uniqueness of the pattern developed in this thesis. Most of the following material comes from Palmer's book with minor editions for content flow [Palm95]. Palmer defines barcodes as “an automatic identification technology that encodes information into an array of adjacent varying width parallel rectangular bar and spaces.” Two parts of the definition are key: automatic and bar and spaces. The automation is one of the goals for this thesis. Bars and spaces are the principal constituents of barcodes.

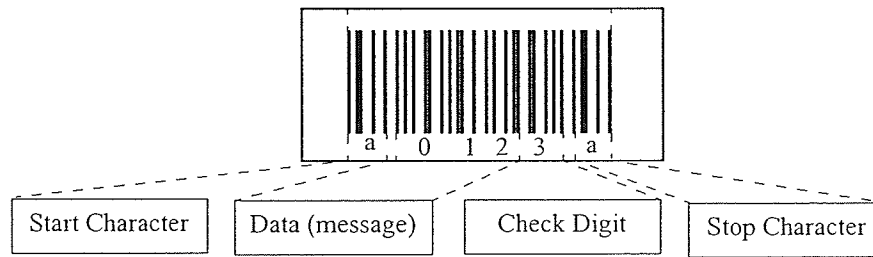


Fig. 2.13. 1D barcode structure.

For a more accurate description, barcodes are a series of printed bars of various colors (usually black and white) and sizes arranged in a certain fashion so as to encode data (see Fig. 2.13.). In general, barcodes bars are parallel and arranged in a rectangular shape. The bars are arranged in groups that are called symbols (the “letters” of the barcode) and the group of all possible symbols is called the symbology of the barcode (the “alphabet” of the barcode). Barcodes contain a synchronizing pattern (such as start and stop characters) and the information symbols. All barcodes use a form of redundancy to protect the information against errors happening during its life on paper. There are two main categories of barcodes: one dimensional (1D) and two dimensional barcodes (2D). 2D barcodes are further divided into stacked and matrix barcodes. Refer to Palmer's book for a complete description and thorough listing of all existing barcodes [Palm95].

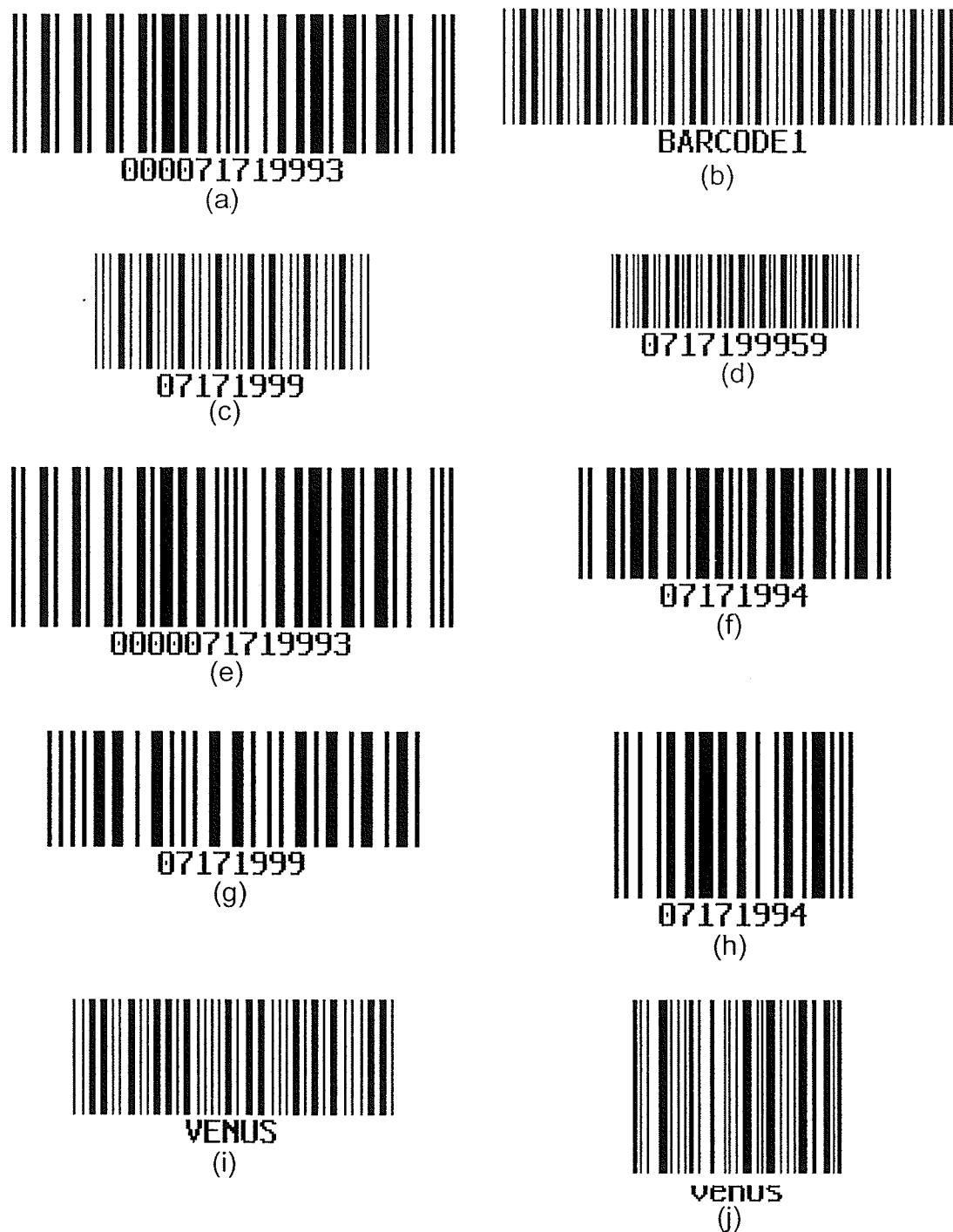


Fig. 2.14. Sample of 1D barcodes: (a) UPC-A; (b) Code 39; (c) Codabar; (d) Code 11; (e) EAN 13; (f) EAN 8; (g) interleaved 2 of 5; (h) UPC-E; (i) Code 39 and; (j) Code 128.

2.5.1 1D Barcodes

1D barcodes usually have a start and stop character at the beginning and the end of the barcode and information symbols in between. They use the height of the symbols as a form of redundancy and sometimes use check characters to guarantee the integrity of the data. Figure 2.14 is an illustration of 1D barcodes.

2.5.2 2D Barcodes

Two dimensional (2D) barcodes, introduced in the mid-80s, use smaller bar height and encode the data with Reed Solomon (RS) codes to protect the code from errors to compensate for the lack of vertical redundancy. The RS codes are used for their resilience to burst errors and for their easy implementation using barcode symbology (each RS field number is associated with a barcode symbol). 2D barcodes were made possible by the advent of better and cheaper imaging technology such as CCD and CMOS arrays and faster, less power hungry processors.

There are two main types of 2D barcodes: 2D stacked and 2D array. 2D stacked are 1D codes piled up on top of one another with shorter bar height. 2D matrix are just a series of dots instead of bars. 2D barcodes can carry more information than regular 1D barcode and can have a much higher resilience to noise. 2D barcodes are only decoded with imaging hardware. Figure 2.15 illustrates a few examples of 2D barcodes.

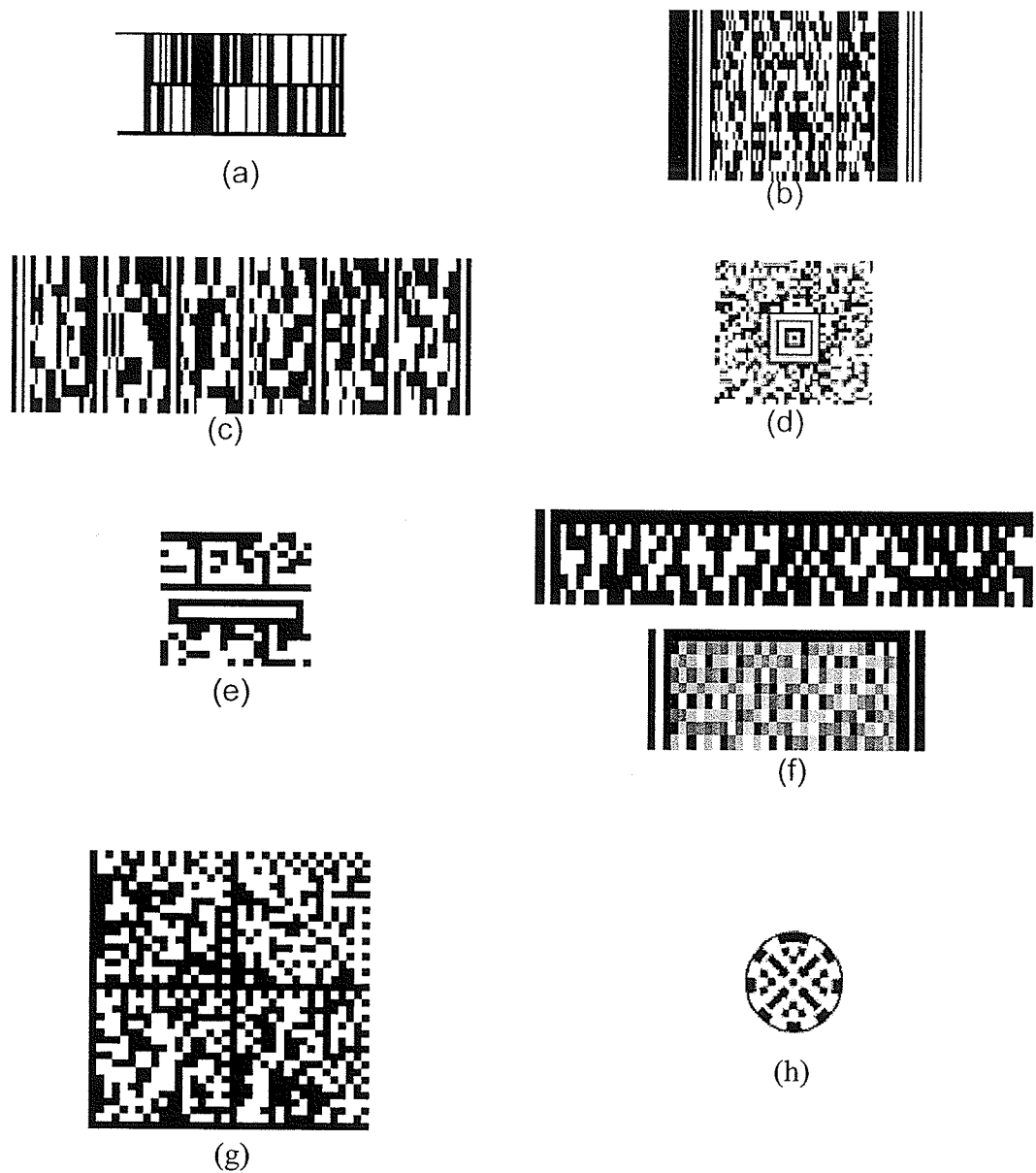


Fig. 2.15. 2D barcodes examples: stacked: (a) 16K; (b) PDF417; (c) SuperCode; matrix: (d) Aztec; (e) Code 1; (f) UltraCode; (g) DataMatrix; and an example of unique 2D barcode: (h) 3Di.

2.5.3 Summary

We have seen different types of barcodes in this section from the original 1D barcodes used for small amount of data exchange such as a key entry into a database for a product. 2D stacked and matrix barcodes used for higher volume of data (for example a macro mode of PDF417 allows to encode up to 99,999 characters) and used for a richer input format than 1D barcodes (some 1D barcodes only handle numbers where PDF417 can encode 76 different kind of symbols). The pattern developed in this thesis will be similar to 2D matrix barcode technology.

2.6 Random Number Generators

Random number generators (RNGs) are algorithms used to generate sequence of numbers that behave like a sequence of independent random numbers with uniform distribution. The meaning of random will not be discussed in this thesis. However, we can define tests that a sequence of independent random number should pass and put the generated sequence through those tests.

Computers are finite state machines, so RNGs implemented on a computer will be periodic. The quality of a RNG on a computer is then measured by the length of its period and by the fact that it passes tests that random numbers should pass. There is however no way to prove that a RNG is good, the generator is good until it fails a test. The quality requirements for a generator also varies with its applications (as shown in Section 5.1.1).

2.6.1 Linear Congruential Generator

The most popular RNG is the linear congruential generator (LCG) which works in the following way:

$$X_{n+1} = (aX_n + c) \bmod m \quad n \geq 0 \quad (2.42)$$

where m is the modulus ($m > 0$), a is the multiplier ($0 \leq a < m$), c is the increment ($0 \leq c < m$), and X_0 is the seed ($0 \leq X_0 < m$). Those generators are fast and have been heavily studied and ways have been found to select moduli and multipliers that give maximum period length [Knut81].

2.6.2 Encryption Based RNG

The goal of data encryption is to generate from a plain text a ciphertext that is as close as possible to white noise so that probabilistic methods performed on the ciphertext cannot give any information about the cleartext (see also Section 2.3 for an introduction on cryptography). This property can be used to generate random numbers by simply encrypting a seed several times and extracting numbers at each iteration. Any encryption algorithm can be used to generate random data. However, encryption algorithms are relatively slow compared to simpler random number generator algorithms such as the LCG. A viable choice is to base the generator on block ciphers algorithms which are some of the fastest strong encryption algorithms.

2.6.3 RNG Test Procedures

In this section, we describe the chi square test which is used, for example, in the equidistribution, pair, triple, and gap test. We also define the spectral test which is used only for LCG.

2.6.3.1 Chi Square Test

The chi square (χ^2) statistic for an experiment with k possible outcomes, performed n times, is defined as [FaDe99]:

$$\chi^2 = \sum_{i=1}^k \frac{(Y_i - np_i)^2}{np_i} \quad (2.43)$$

where Y_i are the number of experiments which resulted in each possible outcome and p_i are the a priori probabilities of each outcome.

χ^2 will be larger to the extent that the observed results diverge from those expected by chance. The probability Q of a χ^2 value for an experiment with d degrees of freedom (df) due to chance is:

$$Q_{\chi^2, d} = \left[2^{\frac{d}{2}} \Gamma\left(\frac{d}{2}\right) \right]^{-1} \chi^2^{\frac{d}{2}-1} e^{-\frac{\chi^2}{2}} dt \quad (2.44)$$

where gamma is the generalization of the factorial function to real and complex arguments:

$$\Gamma(x) = \int_0^{\infty} t^{x-1} e^{-t} dt \quad (2.45)$$

and $d=k-1$ (intuitively, the k^{th} possible outcome value can be deduced from the $k-1$ other possible outcomes, so our experiment has $k-1$ degrees of freedom).

There are two steps to the χ^2 test:

Step 1. If the outcome of an experiment has a probability of happening less than α or more than $100\%-\alpha$ (where α is usually 0.1% or 0.5%) then the generator is rejected and the test stops,

Step 2. If not, the χ^2 cumulative density function (CDF) of a series of experiments with d degrees of freedom is plotted which should be comparable with the theoretical CDF.

If the RNG passes those two steps, it passes the specific test.

The χ^2 test can be performed on various distributions, for example: equidistribution over k categories ($p_i=1/k$), distribution of pairs (k^2 categories, $p_i=1/k^2$), triplets (k^3 categories, $p_i=1/k^3$), distribution of a given bit (2 categories, $p_i=1/2$), mod-3 distribution (3 categories, $p_i=1/3$).

2.6.3.2 Spectral Test

The spectral test is described in detail in [Knut81] and the algorithm to perform the test efficiently is also given. The spectral test applies only to LCG. All known good LCG passed the test and all flawed LCGs failed it.

2.6.4 Summary

RNGs are used to introduce randomness in processes. Two RNGs were detailed (congruential and crypto RNG) and testing procedures to evaluate the quality of the randomness of the number were given. The RNGs are used for noise generation and random interleaving.

2.7 Digital Image Processing

The demodulation of the pattern involves the location of the pattern and transformation to binary information with or without confidence level. The demodulated information is then passed to the decoder. The pattern is acquired through a scanner which transforms it into a digital image. The goal of this section is to give the necessary background in image processing to understand the workings of the demodulator which is designed in the next chapter.

2.7.1 General Concepts

Digital image processing is the action of transforming an image using computers or special purpose digital hardware to enhance it, compress it, extract information from it, or

find patterns in it. The focus of the thesis is on edge enhancement as an image enhancement transformation and line identification algorithms in special transform spaces.

A digital image is represented by a two variable discrete function f :

$$f(x, y), x \in \mathbb{N} \cap [0, N-1], y \in \mathbb{N} \cap [0, M-1] \quad (2.46)$$

where \mathbb{N} is the positive integer set, N and M are the horizontal and vertical dimensions of the image. A single point (x, y) is called a pixel. The value of $f(x, y)$ represents the intensity of the image at pixel (x, y) . For gray scale images, $f(x, y)$ is between 0 and 255 where 0 is black and 255 is white and any number in between is a shade of grey.

2.7.2 Edge Detection

Edges in images are represented by sharp changes in the value of $f(x, y)$. Sharp changes in a function imply a high slope which in turn mean a high positive or negative values for the local derivative. Now consider an ideal step edge. When smoothed the profile of the edge looks like the leftmost sketch of Fig. 2.16.. The next sketches show the first and second derivatives; the presence and location of the edge is marked by a peak and a zero crossing respectively.

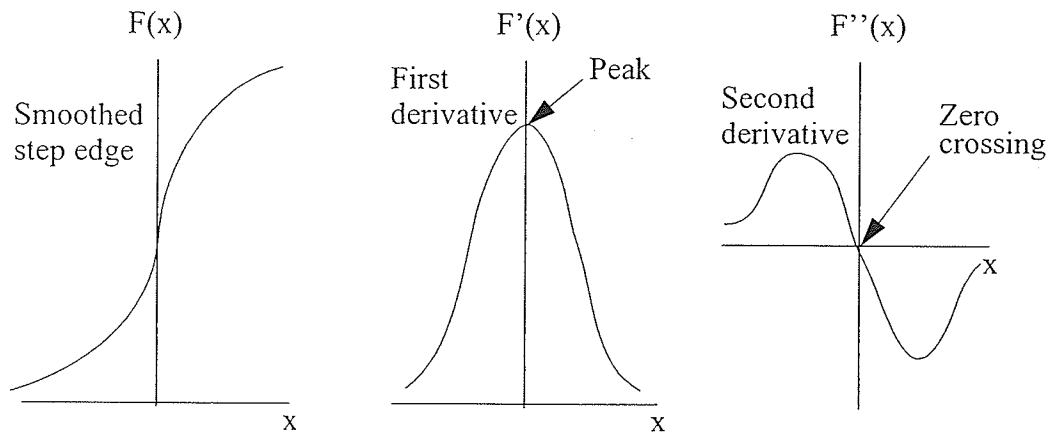


Fig. 2.16. Successive edge derivatives.

The norm of the gradient for a two variable function is expressed by:

$$|\nabla f| = \left[\left(\frac{\partial f}{\partial x} \right)^2 + \left(\frac{\partial f}{\partial y} \right)^2 \right]^{1/2} \quad (2.47)$$

The gradient for a pixel can be approximated as a linear combination of its surrounding pixels. This operation is called convolution and can be represented by a matrix sweeping the image where each pixel is replace by the sum of the surrounding pixel values multiplied by the matrix coefficient. The following operators are all discrete linear approximations of the gradient operator. All operators are then considered in the reference image matrix shown in Fig. 2.17..

$f(x-1,y-1)$	$f(x-1,y)$	$f(x-1,y+1)$
$f(x,y-1)$	$f(x,y)$	$f(x,y+1)$
$f(x+1,y-1)$	$f(x+1,y)$	$f(x+1,y+1)$

Fig. 2.17. Image matrix for edge detection algorithms.

The properties of a good edge detection operator are:

- Isotropic (the edge direction does not matter)
- Good Localization (identify the location of the edge precisely)
- Good Signal-To-Noise Characteristics
- Accurate Determination of Edge Orientation

2.7.2.1 Gradient Operator

The gradient operator can be approximated by the following equation:

$$\nabla f \approx [(f(x,y) - f(x+1,y))^2 + (f(x,y) - f(x,y+1))^2]^{\frac{1}{2}} \quad (2.48)$$

Eq. (2.48) can itself be approximated by:

$$\nabla f \approx |f(x, y) - f(x + 1, y)| + |f(x, y) - f(x, y + 1)| \quad (2.49)$$

Eq. (2.49) is equivalent to perform two convolutions with the masks shown in Figure 2.18 and add the absolute value of the resulting images at each pixel.

1	0	0
-1	0	0
0	0	0

1	-1	0
0	0	0
0	0	0

Fig. 2.18. Basic gradient approximation convolution matrices.

2.7.2.2 Roberts Operator

Another approximation can be obtained by taking the cross terms instead of the terms on the same column or row:

$$\nabla f \approx |f(x, y) - f(x + 1, y + 1)| + |f(x, y + 1) - f(x + 1, y)| \quad (2.50)$$

This is called the Roberts cross-gradient operator. The convolution matrices are shown in Fig. 2.19..

1	0	0
0	-1	0
0	0	0

0	1	0
-1	0	0
0	0	0

Fig. 2.19. Roberts cross-gradient convolution matrices.

2.7.2.3 Prewitt Operator

The Prewitt operator approximates the gradient at (x, y) by taking the difference of three pairs of numbers instead of a single pair of numbers:

$$\begin{aligned} \nabla f \approx & |(f(x-1, y-1) - f(x+1, y-1)) + \\ & (f(x-1, y) - f(x+1, y)) + \\ & (f(x-1, y+1) - f(x+1, y+1))| + \\ & |(f(x-1, y-1) - f(x-1, y+1)) + \\ & (f(x-1, y-1) - f(x-1, y+1)) + \\ & (f(x-1, y-1) - f(x-1, y+1))| \end{aligned} \quad (2.51)$$

The convolution matrices are shown in Fig. 2.20..

-1	-1	-1	-1	0	1
0	0	0	-1	0	1
1	1	1	-1	0	1

Fig. 2.20. Prewitt convolution matrices.

2.7.2.4 Sobel Operator

The Sobel operator is another approximation of the gradient function. However, this operator provides both a smoothing operation with the derivative operation: a simple

1	2	1
---	---	---

operation is performed before the derivative is applied.

-1	2	-1	-1	0	1
0	0	0	-2	0	2
1	2	1	-1	0	1

Fig. 2.21. Sobel convolution matrices.

2.7.2.5 Laplacian Operator

The Laplacian is the second derivative for the two dimensional image function:

$$\begin{aligned}
 \nabla^2 f &= \frac{\partial^2}{\partial x^2} f(x, y) + \frac{\partial^2}{\partial y^2} f(x, y) \\
 &= \frac{\partial}{\partial x} \left(\frac{\partial}{\partial x} f(x, y) \right) + \frac{\partial}{\partial y} \left(\frac{\partial}{\partial y} f(x, y) \right) \\
 &= \frac{\partial}{\partial x} (f(x, y) - f(x-1, y)) + \frac{\partial}{\partial y} (f(x, y) - f(x, y-1)) \\
 &= \{f(x+1, y) - f(x, y)\} - \{f(x, y) - f(x-1, y)\} \\
 &\quad + \{f(x, y+1) - f(x, y)\} - \{f(x, y) - f(x, y-1)\} \\
 &= f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1) - 4f(x, y) \quad (2.52)
 \end{aligned}$$

The equivalent convolution matrix is illustrated in Fig. 2.22.. An edge is represented by a maximum of the absolute value of the derivative which is an increase, a maximum, and then a decrease. The second derivative will then be positive before the

edge and then negative. In other words, the derivative will change sign and the edge location is the zero crossing of the function. Edge location with the Laplacian operator is then performed in two steps: convolution of the image with the Laplacian operator mask, and location of the zero crossing in the image.

0	1	0
1	-4	1
0	1	0

Fig. 2.22. Laplacian operator.

2.7.2.6 Canny Operator

The Canny edge detector arises from the earlier work of Marr and Hildreth, who were concerned with modeling the early stages of human visual perception [MaHi80]. In designing this edge detector, Canny considered an ideal step edge represented as a sign function in one dimension, corrupted by an assumed Gaussian noise process. In practice this is not an exact model but it represents an approximation to the effects of sensor noise, sampling, and quantization. This edge detector is not used in this Thesis but could be implemented in further work. The approach was based strongly on convolution of the image function with Gaussian operators and their derivatives, so we shall consider the 2D Gaussian functions and their derivatives (in polar coordinates):

$$G(r) = \frac{1}{2\pi\sigma^2} e^{-\frac{r^2}{2\sigma^2}} \quad (2.53)$$

and the first derivative with respect to r is,

$$\frac{\partial}{\partial r} G(r) = \frac{-r}{2\pi\sigma^4} e^{-\frac{r^2}{2\sigma^2}} \quad (2.54)$$

and the second derived is,

$$\nabla^2 G = \frac{1}{r} \frac{\partial}{\partial r} \left(r \frac{\partial}{\partial r} (G(r)) \right) = -\frac{r}{\sigma^2} G(r) \left[2 - \frac{r^2}{\sigma^2} \right] \quad (2.55)$$

The first derivative of the image function convolved with a Gaussian,

$$g(x, y) = D[Gauss(x, y) \otimes f(x, y)] \quad (2.56)$$

is equivalent to the image function convolved with the first derivative of a Gaussian,

$$g(x, y) = D[Gauss(x, y)] \otimes f(x, y) \quad (2.57)$$

Therefore, it is possible to combine the smoothing and detection stages into a single convolution in one dimension, either convolving with the first derivative of the Gaussian and looking for peaks, or with the second derivative and looking for zero crossings.

After convolving the image with a derivative of the Gaussian function, edge detection can proceed by the simple thresholding of the data. In practice, the final

determination of the presence or absence of an edge is more complex, and it is common to include post-processing thinning and thresholding stages.

The goals of the Canny Operator were stated explicitly

- Good Detection: the ability to locate and mark all real edges.
- Good Localization: minimal distance between the detected edge and real edge
- Clear Response: only one response per edge.

To fulfil these objectives, the edge detection process included the following stages

Stage 1. Image Smoothing: The image data is smoothed by a two dimensional Gaussian function of width specified by a user parameter.

Stage 2. Differentiation: Assuming two dimensional convolution at stage 1, the smoothed image data is differentiated with respect to the x and y directions. It is possible to compute the gradient of the smooth surface of the convolved image function in any direction from the known gradient in any two directions.

Stage 3. Non-maximum Suppression: Having found the rate of intensity change at each point in the image, edges must now be placed at the points of maxima, or rather non-maxima must be suppressed. A local maximum occurs at a peak in the gradient function, or alternatively where the

derivative of the gradient function is set to zero. However, in this case we wish to suppress non-maxima perpendicular to the edge direction, rather than parallel to (along) the edge direction, since we expect continuity of edge strength along an extended contour (this assumption creates a problem at corners.) Rather than perform an explicit differentiation perpendicular to each edge, another approximation is often used. Each pixel in turn forms the centre of a nine pixel neighborhood. By interpolation of the surrounding discrete grid values, the gradient magnitudes are calculated at the neighborhood boundary in x and y directions perpendicular to the centre pixel. If the pixel under consideration is not greater than these two values (i.e. non-maximum), it is suppressed

Stage 4. Edge Thresholding: The thresholder used in the Canny operator uses a method called "hysteresis". Most thresholders used a single threshold limit, which means if the edge values fluctuate above and below this value the line will appear broken (commonly referred to as "streaking"). Hysteresis counters streaking by setting an upper and lower edge value limit. Considering a line segment, if a value lies above the upper threshold limit it is immediately accepted. If the value lies below the low threshold it is immediately rejected. Points which lie between the two limits are accepted if they are connected to pixels which exhibit strong response. The likelihood of streaking is reduced drastically since the line

segment points must fluctuate above the upper limit and below the lower limit for streaking to occur. Canny recommends the ratio of high to low limit be in the range two to one or three to one, based on predicted signal-to-noise ratios.

Canny proposed a final processing stage, aggregation or feature synthesis. The general principle is to start from the smallest scale, then synthesize the larger scale outputs that would occur if they were the only edges present. Then compare the large scale output to the synthesis. Additional edges are marked if the large scale output is significantly greater than the synthetic prediction. The synthetic data was produced by convolving a Gaussian of large scale with the small scale edge data. The procedure is repeated to mark edges at the second scale which were not present at the first, then third to second, and so on. Canny observed that “in the many cases the majority of edges were picked up by the smaller channel, and that later channels mark mostly shadow and shading edges, or edges between textured regions”.

2.7.2.7 Deriche Operator

The basic idea introduced by Canny behind optimal operators is based on a continuous edge model which consists in a step signal corrupted by a gaussian white noise

$$I(x) = Au_{-1}(x) + n(x) \quad (2.58)$$

where $u_{-1}(x)$ denotes the unit-step function and $n(x)$ is a noise term whose mathematical expectation and variance are respectively 0 and σ^2 .

We then consider the convolution of this signal with an edge detector $f(x)$

$$\theta(x) = \int_{-\infty}^{\infty} I(y)f(x-y)dy \quad (2.59)$$

According to this model Canny has proposed to find an optimal edge detection operator using the following three requirements:

1. Low probability of error (failing to mark or falsely marking real edge points).

This criterion consists in finding an asymmetric operator which maximizes the signal to noise ratio:

$$\Sigma = \frac{A \int_{-\infty}^0 f(x)dx}{\sigma \sqrt{\int_{-\infty}^{\infty} f^2(x)dx}} \quad (2.60)$$

2. Good localization: points marked as edges should be as close as possible to the true edge. The localization is defined as being the inverse of the standard deviation of the position of the true edge:

$$\Lambda = \frac{A|f'(0)|}{\sigma \sqrt{\int_{-\infty}^{\infty} f'^2(x)dx}} \quad (2.61)$$

3. Only one response to a single edge: consists in a constraint on the average distance between two local maxima (x_{max}),

$$x_{max} = \sqrt{\frac{\int_{-\infty}^{\infty} f'^2(x) dx}{\int_{-\infty}^{\infty} f''^2(x) dx}} \quad (2.62)$$

Canny has then proposed a FIR operator (Finite Impulse Response filter: a series of shift registers with no feedback) which maximizes the product $\Sigma\Lambda$ under the constraint that the third criterion is fixed to a constant value k . For the unapproximated Canny operator the performance is $\Sigma\Lambda = 1.12$ with $k=0.58$. In practice, as seen in the previous section, the Canny operator is approximated by a Gaussian function which leads to $\Sigma\Lambda = 0.92$ with $k=0.51$.

Deriche has then proposed an IIR (Infinite Impulse Response filter: a series of shift registers with feedback) operator which optimizes the Canny criteria, this operator has the following form [Deri87]

$$f(x) = \frac{S}{\omega} e^{-\alpha|x|} \sin \omega x \quad (2.63)$$

It is shown that the most efficient operator is the limit of the previous one when ω tends to 0. Therefore,

$$f(x) = Sx e^{-\alpha|x|} \quad (2.64)$$

This filter has a performance $\Sigma\Lambda$ equal to $\sqrt{2}$ (with $k=0.58$) which is approximately 25% better than the unapproximated Canny operator. An interesting property of this operator comes from the α parameter which allows to adapt it to the

content of the images. Roughly, if the image is noisy the α parameter has to be chosen relatively small (0.25 to 0.5) which means that Σ (detection) is favored to the detriment of Λ (localization) ([Deri98] p. 25). For a clean image α is chosen relatively large (≈ 1.0).

The one-dimensional Deriche operator corresponds to two stable second order recursive filters (this proof is based on computing the Z transform of the discrete IIR operator) ([Deri98] p. 17). This results can be extended to the two-dimensional operator. The first step in the computation of the filter consists in processing the rows of the image according to the following scheme (x is the input image)

$$y_{i,j}^{(1)} = a_1 x_{i,j} + a_2 x_{i,j-1} + b_1 y_{i,j-1}^{(1)} + b_1 y_{i,j-2}^{(1)} \quad (2.65)$$

going from left to right, and

$$y_{i,j}^{(2)} = a_3 x_{i,j+1} + a_4 x_{i,j+2} + b_1 y_{i,j+1}^{(2)} + b_2 y_{i,j+2}^{(2)} \quad (2.66)$$

going from right to left. This operation gives a temporary result, say λ , whose expression is

$$\lambda_{i,j} = c_1 (y_{i,j}^{(1)} + y_{i,j}^{(2)}) \quad (2.67)$$

The second step concerns the columns of λ

$$y_{i,j}^{(1)} = a_5 \lambda_{i,j} + a_6 \lambda_{i-1,j} + b_1 y_{i-1,j}^{(1)} + b_1 y_{i-2,j}^{(1)} \quad (2.68)$$

from top to bottom, and

$$y_{i,j}^{(2)} = a_7 \lambda_{i+1,j} + a_8 \lambda_{i+2,j} + b_1 y_{i+1,j}^{(1)} + b_1 y_{i+2,j}^{(1)} \quad (2.69)$$

from bottom to top. The final result is then obtained by

$$i,j = c_2(y_{i,j}^{(1)} + y_{i,j}^{(2)}) \quad (2.70)$$

The following set of parameters is used to compute the horizontal component of the gradient

$$a_1 = 0, a_2 = 1, a_3 = -1, a_4 = 0, c_1 = -(1 - e^\alpha)^2 \quad (2.71)$$

$$a_5 = k, a_6 = ke^{-\alpha}(\alpha - 1), a_7 = ke^{-\alpha}(\alpha - 1), a_8 = -ke^{-2\alpha}, c_2 = 1 \quad (2.72)$$

The vertical component can be obtain by swapping a_i and a_{i+4} , and c_1 and c_2 .

For both components we have

$$k = \frac{(1 - e^{-\alpha})^2}{1 + 2\alpha e^{-\alpha} - e^{-2\alpha}}, b_1 = 2e^{-\alpha}, b_2 = -2e^{-2\alpha} \quad (2.73)$$

Another set of parameters allows to compute the Laplacian of the image but the most efficient way to perform features extraction is the gradient based approach. Since the processing of a row (respectively a column) is independent of the computations required for other rows (respectively columns) it is possible to implement the Deriche operator on a parallel computer [LaLi96].

We now describe contour extraction using the result produced by the Deriche operator. The first step, in order to extract thin contours, i.e. contours of thickness equal to one pixel, is to extract the local maxima of the gradient norm in the gradient direction given by the vector $(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y})$. However, the gradient direction does not coincide (in

general) to integer pixel coordinates. There are two common ways of dealing with this problem:

1. Take the gradient norm at the nearest neighbor of the pair of non-integer coordinates; and
2. Perform a (bi)linear interpolation in order to obtain a value for the gradient norm at the non-integer coordinates given by the gradient direction.

Note that both solutions have no physical meaning. Hysteresis thresholding method can also be used. It consists in eliminating the local maxima which are not significant according to the following rules. Given a high and a low threshold level, we eliminate the following local maxima:

1. Local maxima which gradient norm are less than the low level; or
2. Local maxima which gradient norm are less than the high level unless it is connected (here 8-connected) to another local maximum which gradient norm is greater than the high threshold level.

A local maximum is said to be 8-connected to another local maximum whose gradient norm is greater than the high threshold level if one of its 8 nearest neighbors has a gradient norm greater than the high threshold level or if one of its neighbors is connected to a local maximum whose gradient norm is greater than the high threshold level. This

thresholding technique gives better results than a one level threshold because it takes into account that some contours can be "hidden" in the noise.

2.7.3 Line Tracking Algorithms

2.7.3.1 Radon Transform

The Radon transform extracts line parameters from an image. It is an attractive transform for its resilience to noise. Each line in the image space is transformed into peaks in the Radon space, which position correspond to the parameters of the lines. The location of lines in the transform space is then the identification of local maxima. Once the maxima are located, the associated parameters can be used to reconstruct the lines in the original image.

The Radon transform is defined by integrating a continuous two dimensional function $f(x, y)$ along slanted lines

$$F(a, b) = \int_{-\infty}^{\infty} f(x, ax + b) dx \quad (2.74)$$

Intuitively, if $g(x,y)$ has high values along the line $y = ax + b$ the transform $F(a, b)$ has a high value. The intensity in the (a, b) space indicate the probability of a line in the image. Using the Dirac function, Eq. (2.74) can be rewritten as

$$F(a, b) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y) \delta(y - ax - b) dx dy \quad (2.75)$$

The Radon transform is approximated in the discrete domain by the following equation

$$F(a_k, b_l) = \int_{-\infty}^{\infty} f(x, a_k x + b_l) dx \approx \Delta x \sum_{m=0}^{M-1} f(x_m, a_k x_m + b_l) \quad (2.76)$$

where

$$\begin{aligned} x &= x_m = x_{min} + m\Delta x & m &= 0, \dots, M-1 \\ y &= y_n = y_{min} + n\Delta y & n &= 0, \dots, N-1 \\ a &= a_k = a_{min} + k\Delta a & k &= 0, \dots, K-1 \\ b &= b_l = b_{min} + l\Delta b & l &= 0, \dots, L-1 \end{aligned} \quad (2.77)$$

Both the image coordinates and the Radon transform coefficients are quantized. The quantization of the image coordinates relates to our digital image model. The quantization of the slope coefficients relate to the quantization of the Radon transform space which in turn can be represented as a digital image. The ordinate component $a_k x_m + b_l$ does not necessarily correspond to a y_n , so the image has then to be approximated in those points. Three possible methods to approximate the value of the image are: nearest neighbor, linear, and sinc interpolation. For more details on the algorithms used to perform such interpolation are detailed in [Toft96].

The cartesian representation of lines has a singularity for vertical lines ($a \rightarrow \infty$.) To avoid the singularity, the lines can be represented in polar form

$$\begin{aligned}
F(\rho, \theta) &= \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(x, y) \delta(\rho - x \cos \theta - y \sin \theta) dx dy \\
&= \int_{-\infty}^{\infty} f(\rho \cos \theta - s \sin \theta, \rho \sin \theta + s \cos \theta) ds
\end{aligned} \tag{2.78}$$

where s lies along the line.

Similarly, the Radon transform can be expressed in the discrete domain

$$F(\rho_r, \theta_l) \approx \Delta s \sum_{j=0}^{S-1} f(\rho_r \cos \theta_l - s_j \sin \theta_l, \rho_r \sin \theta_l + s_j \cos \theta_l) \tag{2.79}$$

Similar interpolation methods can be used to approximate the value of $f(x, y)$ where the (x_m, y_n) quantized coordinates do not match the $(\rho_r \cos \theta_l - s_j \sin \theta_l, \rho_r \sin \theta_l + s_j \cos \theta_l)$ coordinates. Again see [Toft96] for details.

The Radon transform of a line in the image domain is a point in the transformed domain. The transformation of a point in the image domain are all the parameters of the lines going through this point. For the cartesian representation, if the point is (x_0, y_0) the lines going through the point have the following form:

$$y_0 = ax_0 + b \tag{2.80}$$

hence

$$b = -x_0 a + y_0 \tag{2.81}$$

The transform of a point using the cartesian Radon transform is a line. Toft shows that the transform of a point in the polar domain is a sinusoid (the same reasoning can be applied: the coefficients of all the lines going through the point) [Toft96].

2.7.3.2 Hough Transform

If only a few points in the image are non zero, the Radon transform calculation adds a lot of zero values with no benefit to the final result. The Hough transform makes calculations only for the points defined in the image: instead of counting the points on a line in the image domain and representing it as a point with the summed intensity in the transform domain, the Hough transform takes each non zero point in the image domain and transforms it into the transform of the point in the transform domain (a line for the cartesian representation and a sinusoid for the polar representation, both represents all possible coefficients of the lines that go through the given point). If the number of zero points in the image is high, the Hough transform has a smaller computational load than the Radon transform. The Radon transform has a complexity of $O(M^3)$ (M number of pixels in a side) and the Hough transform has a complexity of $O(EFM)$ where EF is the number of non-zero pixels in the image (which is hopefully less than M^2 .)

The Hough transform has also both a cartesian and a polar representation and a discrete calculation for digital images.

2.7.4 Summary

We saw in this section different methods for line parameters extraction (Hough and Radon transforms) and methods for edge detection in digital images. This is used in locating the transmitted data on the physical paper channel.

CHAPTER III

SECURITY SYSTEM, CODE, AND PATTERN DESIGN

3.1 Introduction

Each component background was described in Chapter II. It is now time to bring them all together and apply them to the thesis: transmit (pattern design, decoder design) securely (security system) and reliably (error correcting code design) on paper.

3.2 Scalable Security System

Several factors affect the selection of a cryptosystem. The paper documents have two main characteristics:

- (i) Their life expectancy; and
- (ii) Their value.

For this reason, several levels of security are required from the cryptosystem.

3.2.1 System Design for Small Value Documents

If the information on a paper has a small value and a lifetime expectancy of a few months, a simple verification station with a single key for a batch of documents can be implemented. The cryptosystem can be based on a stream based key cipher for

information confidentiality with an MDC for data integrity. If the information carried by the security pattern can be clearly displayed but not altered (like the dollar amount of a banknote), a MAC along with the original message can be used to secure the message content. The keys for the document batch are stored on the verification stations and are expired as new documents are introduced. The updating of the keys is handled by a simple symmetric key communication system. The symmetric communication keys are changed at each new download of document keys. The communication channel bandwidth and the verification station key database are minimal. Figure 3.1 illustrates the cryptosystem.

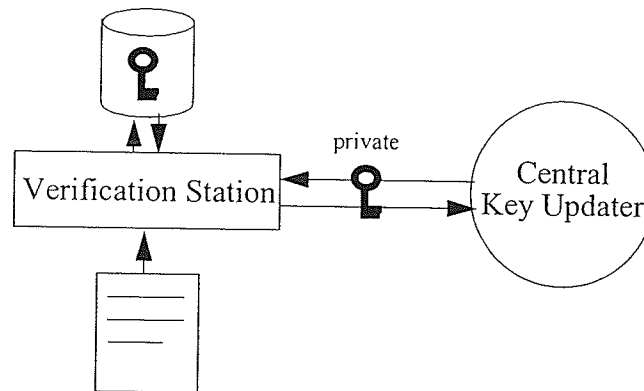


Fig. 3.1. Cryptosystem schematic for small value documents.

A small number of keys are stored on the verification station and the keys are expired after a few months. A key is corrupted only if the verification hardware is broken into. If a key is leaked, the corrupted key only involves a few months' worth of documents and when the fraud is detected, the key can be retired. The key becomes invalid as soon as a new batch of documents is produced.

3.2.2 System Design for Medium Value Documents

The information value is medium and with a lifetime expectancy of a year or two. This system is designed to reduce the impact of a corrupted key. Half of the key or the entire key is stored on the verification station to reduce the storage requirements and the key encrypts only part of a batch (e.g., groups of thousands). A central key server stores the remainder of the key for each part of the batch or is not used interactively if the whole key is stored on the station. The communication is still done using a symmetric key communication system. The communication key is updated at regular intervals of time (e.g. a month). If the communication channel key is broken into, the keys are updated before any new upload of new document key. A corrupted document key only corrupts the small part of the batch it encrypts. The storage requirement is medium and the communication bandwidth is medium to small depending on the quantity of the key that is stored on the verification station. Figure 3.2 illustrates the system.

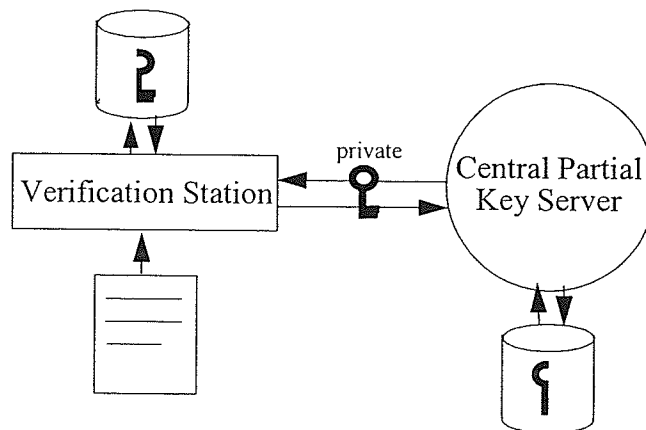


Fig. 3.2. Cryptosystem schematic for medium value documents.

3.2.3 System Design for High Value Documents

This system design requires a high bandwidth communication channel but no storage on the verification station. There is one key per document and all keys are stored on the local server. The key server and the verification stations verify each other's identities using zero-knowledge based protocols. During the identification phase a symmetric session key generated by the key server is exchanged. In this way, the communication key is never stored on any of the stations. In the same way, by using zero-knowledge based protocols, the identification tokens are never stored on the verification stations. It is much easier to secure a central server than all the individual verification stations that go in the field. Once the session key is established, the key server can send the document key or update the station identification token. A corrupted document key only involves one document. Figure 3.3 illustrates the system.

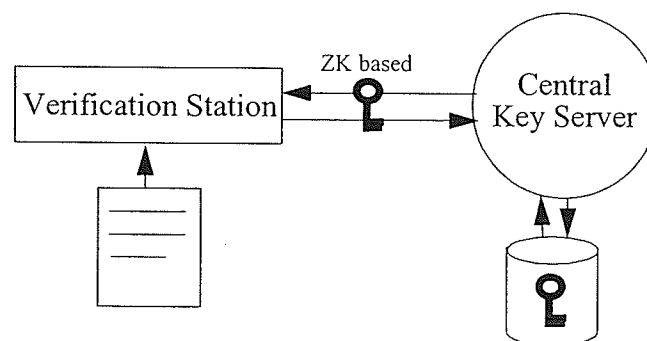


Fig. 3.3. Cryptosystem schematic for high value documents.

3.2.4 Summary

The value of paper documents even of the same kind can vary (paper money of different value). All the previous systems can be mixed for the same type of documents according to the document value. The systems described are recommended systems which can be altered or completely changed depending on the requirements of the documents.

3.3 Double Interleaved Concatenated RS and Convolutional Self-

Orthogonal Code

As Michelson and Levesque said: "the selection of an error control scheme consists in part of matching the features of the technique with the system objectives." [MiLe85]. The system's objectives are to produce a reliable (that detects if and where errors occurred and possibly corrects them), fast communication of data on the paper channel which is a compound error channel. The approach taken in this thesis is to use a concatenated scheme with interleaving.

Concatenated codes, introduced by Forney in 1966, are the consecutive encoding and decoding of a data stream by two or more codes. The first code is called the outer code followed by the inner code. Since decoding is implemented in stages on much simpler codes, decoder complexity is much reduced. Even though concatenated codes are not the best codes in their class, they still have very good performance (as seen in the experimental results) and make decoders for such powerful codes tractable. Concatenated codes match the reliability and speed requirements of our system.

In [CaCl81], the authors show through experimentation that the best concatenated codes are obtained using Reed-Solomon (RS) codes as outer codes and convolutional codes as inner codes. Reed-Solomon are chosen as outer codes because they are MDS which make them the best performing codes for a given rate. Furthermore, different error-correction capabilities can be chosen for RS codes which helps match the requirements of a specific channel by modifying the error correction power of the code according to the noise strength and characteristics on the transmission channel. Finally, fast decoding algorithms based on the Berlekamp-Massey algorithm make them suitable for our speed requirements.

There should be an optimum rate for the outer code for a fixed inner code and a fixed channel. Indeed, the inner code leaves a given BER after decoding the information from the channel, there should thus be an optimal rate for the outer code at which the targeted error correction at the output of the overall code is matched and no additional correcting power is unused. The flexibility of RS codes facilitates finding this optimal rate.

The inner code should have a monotonic performance around the aimed signal to noise ratio (SNR). If the code has a waterfall shaped performance curve (flat before a given SNR and steep afterwards), the outer code will be designed along the flat portion of the curve which is a waste of the inner code decoding power, or along the steep portion, which in turn gives a very unstable code performance when the SNR hits the flat portion of the inner code performance curve. Furthermore, Forney looked for the best operating

point on the inner code performance curve at fixed SNR that gives the optimal outer code rates and found that the inner code should output a stream with a 10^{-2} BER. All these considerations make convolutional codes a good candidate for inner codes. Clark and Cain selected a Viterbi decoding algorithm using demodulator soft decision. However, the Viterbi algorithm complexity grows exponentially with the constraint length of the code, so we decided to use instead a self-orthogonal convolutional code (CSOC) with either majority decoding, or threshold decoding, or even feedback threshold decoding, all of which have fast hardware implementation. We believe that the loss in performance is not big enough to diminish the quality of the overall code. Furthermore, due to the simplicity of the decoding algorithm, codes with longer constraint length can be practically used. The chosen CSOC are (2,1) codes with several constraint lengths.

Due to their continuous decoding, convolutional codes decoding errors tend occur in bursts. However those bursts have finite length characteristics more specifically, for CSOC they are 2 to 3 times the constraint length of the code [MiLe85]. This allows the designer to insert an interleaver between the RS code and the convolutional code to break up those error bursts in several RS blocks. When burst errors have bounded length, the most efficient interleaving in terms of speed and size required to break up the burst noise are periodic interleavers [CaCl81]. There is an optimal size after which no additional decoding performance is gained. For example, in the case of block interleaving, if the interleaver width is longer than the burst, the burst will be spread at longer intervals but there will be no additional gain when the RS code decodes those errors.

We have seen that errors also occur in bursts on the paper channel but convolutional codes are designed to correct random errors best. A way to transform those bursts into random errors is once more by the use of interleavers. However, in this case the burst characteristics are unknown, e.g. it is hard to predict what direction a scratch is going to have, how long and how wide it will be. The only way to cope with unknown burst error noise is whether to estimate worst case scenario and design periodic interleavers accordingly or use pseudorandom interleavers. We have chosen pseudorandom interleavers for this design.

Reed Solomon code concatenated with convolutional codes have been used extensively in many application such as space communication and several thesis treated the specific code used in this thesis [Leon95]. However, they are a new application for transmission of data on paper documents.

Figure 3.4 shows the final block diagram of the code.

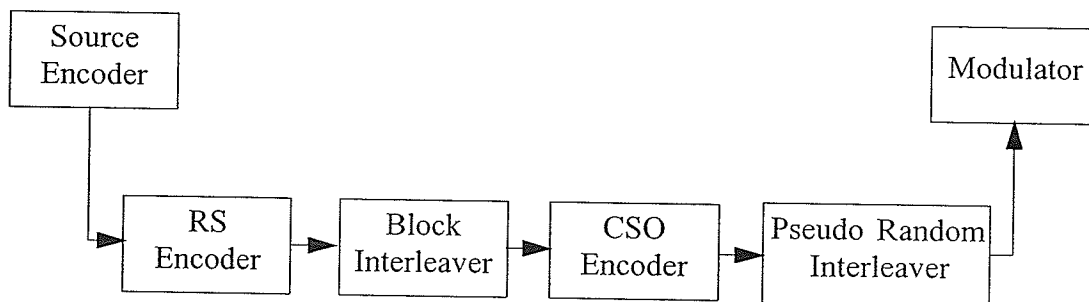


Fig. 3.4. Block diagram of the designed code.

3.4 Square Pattern Modulator with Threshold Demodulator

The following paragraph describes the process used to transmit the encrypted and encoded data on paper. The process is illustrated in Fig. 3.5.. The first step is to print the data overlaid on the document. The choice of the code is a double interleaved concatenated RS and convolutional self-orthogonal code. The choice of the size and constraint length of each component is dependent on the results obtained in the studying of the theoretical code performance under AWGN. This point is developed in the next chapter. The size of the interleavers should be maximized for performance. The size of each interleaver (block and random) should be of the size of the transmitted data to maximize the size of the interleavers without padding with unnecessary zero data to complete the interleaver input.

The pattern is printed and read in a matrix form. Data is printed row by row from left to right. A "1" is represented by a black square and a "0" by a white square. The dimensions of the matrix are calculated so the difference between length and width is minimized (closest dimensions to a square). The smallest difference is calculated by finding the smallest difference between two integers that divide the total number of bits and which product is equal to the total number of bits. Such numbers are built by taking combinations of the prime number decomposition of the total number of bits. The pattern is surrounded by a one pixel border used for location and transformation.

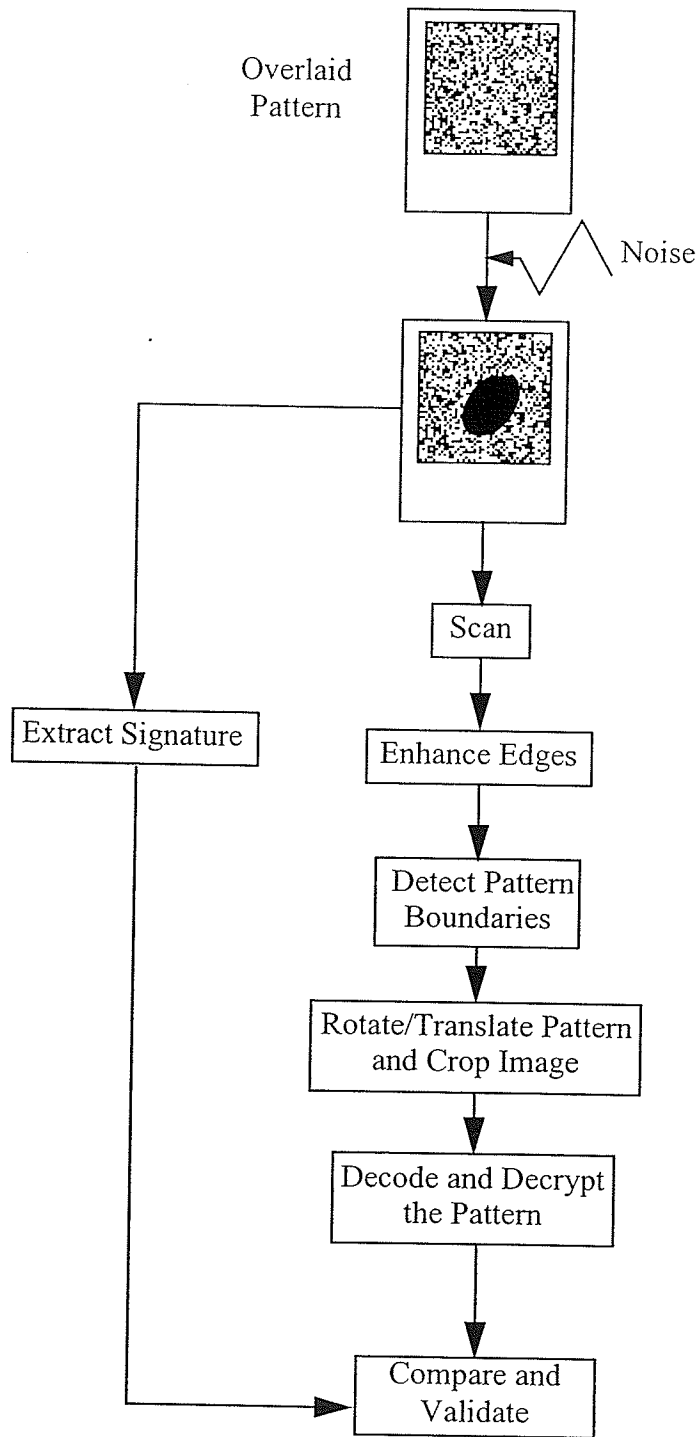


Fig. 3.5. Decoding process

The demodulation consists of: scanning, locating, transforming, and reading. The only transformations corrected is a possible rotation and translation of the pattern, no shearing or perspective. The number of bits in the image read is reduced by applying an edge detector operator and thresholding the resulting image. This allows the number and the range of the pixels to be reduced without losing the important pattern perimeter locator. The selection of the edge detector is based on the detector that will keep the pattern frame best even with surrounding noise.

The image is then transformed using the Radon or the Hough transform. The local peaks in the transformed space are enhanced by applying a local thresholding operator. The pattern perimeter line coefficients are four peaks, grouped in pairs. Each pair of coefficients is located on the same vertical line (each pair of coefficients represent the two parallel lines of the rectangle), the two pairs are separated by 90° (angles in a rectangle). By identifying those coefficients, the pattern is then located and the proper translation and rotation is applied to the image. To avoid the pattern going out of the image bounds, the translation is applied first and the rotation is applied next. If the pattern is located properly, the angles of the lines in the resulting transformed pattern should be respectively 0 and 90° . The opposite of the smallest line angle (in absolute value) (0-angle) is the rotation to apply. If the pattern after rotation is still upside down, the decoding fails and a rotation of 90° is applied to the pattern; if after four rotations the pattern is still not decoded it is a decoding failure (an improvement to the pattern would be to put a unique mark in one corner to know the proper orientation of the pattern). Each parallel line should have an equal and opposite distance from the center. The translation vectors should then be minus

the distance of the center between the two lines and the image center. Let's call θ the rotation angle and $\rho_1, \rho_2, \rho_3, \rho_4$ the distances of the lines from the image center obtained in the transform space. The transformation to apply is the following

$$\begin{bmatrix} x^{(t)} \\ y^{(t)} \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \frac{\rho_1 + \rho_2}{2} \\ \frac{\rho_3 + \rho_4}{2} \end{bmatrix} \quad (3.1)$$

where $\begin{bmatrix} x^{(t)} \\ y^{(t)} \end{bmatrix}$ are the transformed coordinates. We want however to perform the translation first. The transformation then becomes

$$\begin{bmatrix} x^{(t)} \\ y^{(t)} \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \left(\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} \frac{\rho_1 + \rho_2}{2} \\ \frac{\rho_3 + \rho_4}{2} \end{bmatrix} \right) \quad (3.2)$$

The matrix multiplication for the translation coefficient is of course calculated once only and not for each pixel. The image is then cropped to the edges of the pattern. In the following discussion, a pixel is a point in the scanned image and a pattern pixel is a pixel in the printed pattern; a pattern pixel can be formed of several image pixels. The pattern is now centered and straight, using the top and bottom lines thickness, the thickness of a pattern pixel is calculated (average of the lines' thicknesses). The image is then read a square at a time, each square having the size of the calculated frame line thickness. All the pixel values inside the square are added and the pattern pixel is a "1" if its value is more than the value of the sum of a black pattern pixel divided by two, it is a "0" otherwise. However, if a pixel in the center of the pattern pixel is black or white, the

pattern pixel is automatically assigned to the corresponding “1” or “0” because if the pattern is not perfectly aligned with the image pixels, the value of the edges is more likely to be in between the value of the two adjacent pattern pixels). The generated bitstream is then fed to the decoder.

A signature is extracted from the document (such as a bill serial number) and the key database (whether it is stored locally or not) is queried with this document signature and it sends back the encryption key and the random seed for the random interleaver. To limit the data transmitted, the crypto key can be used as the random seed. The decoding and decrypting is as described in Chapter II. The extracted information is then compared to the signature extracted from the document which in turn validates or invalidates the document.

3.5 Summary

The complete design for the secure and reliable transmission of data on paper has been laid out. The important points were: there is different level of security system depending on the lifetime of the document and its value. The chosen code (double interleaved concatenated RS and convolutional self-orthogonal code) is designed to handle the compound error paper channel and the speed requirements for the application. The pattern and demodulation are designed for easy location and handles rotation and translation offsets. The next chapter sets the design of the experiments performed for the

random number generators randomness test, coding evaluation, and pattern demodulation experiments.

CHAPTER IV

DESIGN OF EXPERIMENTS

4.1 Introduction

Several experiments are performed to test the components of the system. The random number generators are put through several tests of randomness, the code's theoretical performance under AWGN (with or without interleavers) is evaluated. Finally, the pattern functioning with or without noise is tested. This chapter describes settings and design of those experiments and validates the decision made for those experiments.

4.2 Code Performance

This experiment tests the code performance in several configurations. The measure used to compare codes is the bit error rate (BER) against the signal to noise ratio (SNR). The BER is plotted on the logarithmic scale, it is calculated by dividing the number of bits in error over the total number of bits. The SNR is measured by dividing the signal power over noise power. The SNR ranges from -5dB to +5dB in steps of +1dB. The source contains 10^6 bits for SNR from -5dB to +2dB and 10^7 bits for +3dB and up because performance is around 10^{-5} when SNR is less or equal than 2dB and at 3dB or more, the BER often passes 10^{-6} requiring an increased number of bits for statistically valid measurements (a rule of thumb is 30 samples for statistically valid data). The signal is

modulated using binary antipodal signaling (+1V for a binary 1 and -1V for a binary 0).

The average power of the binary antipodal signaling is

$$E_S = \frac{1}{T_B} \int_0^{T_B} (\pm 1 V)^2 dt = 1 W$$

where T_B is the period of the signal

Gaussian variables have a power of $N_0 = \sigma^2$ Watts (σ^2 is the variance of the signal). The SNR is then

$$SNR = \log\left(\frac{E_S}{N_0}\right) = 10\log\left(\frac{1}{\sigma^2}\right)$$

$$\sigma^2 = \frac{1}{10^{\left(\frac{SNR}{10}\right)}} W \quad (4.1)$$

A property of a random variable $R(m,v)$ (m mean, v variance) is

$$R(m = 0, v = \sigma^2) = \sigma R(m = 0, v = 1) \quad (4.2)$$

The noise is generated with a zero mean, a variance based on the SNR using Eq. (4.1). The Box Muller algorithm is used to transform the uniform probability variable into Normal distribution as found in [PTVF92]. The chosen RNG to generate the uniform probability variable is the pDES for its very good performance to all the tests we put it through (see Section 5.1.2). Noise is added one bit at a time: a value of +1 or -1 is input, a Gaussian variable is generated and added to the value.

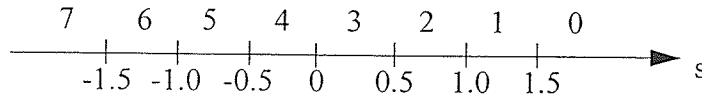


Fig. 4.1. Uniformly spaced 3 bit quantizer.

Table 4.1 Optimum 8-level quantizer; $E_s/N_0 = -6.1$ dB; $I_{8-j} = -I_j$; for $0 < j < 4$.

I_j	Threshold value
I_0	$-\infty$
I_1	-1.760
I_2	-1.056
I_3	-0.503
I_4	0

The demodulator either makes a hard decision on the received signal according to its sign or scales and quantizes the output to 4, 8, or 12 levels. Both [CaCl81] and [MiLe85] deal with the quantization levels and the associated performance for respectively threshold decoding of block code and Viterbi decoding algorithm. Both conclude that 3 bit quantization (8 levels) gives a good decoding performance (0.1 dB from continuous) and higher number of bits do not increase performance significantly considering the added decoder complexity. Michelson and Levesque give the optimal quantization thresholds with regard to maximizing the computational cut-off rate R_{COMP} (for rates $R < R_{\text{COMP}}$ codes with long constraint length or block length can be decoded without suffering an unbounded growth in the number of decoding computations).

The results are very close to the uniform quantizer which we use in this thesis (see Fig. 4.1. and Table 4.1).

Several experiments are performed to test various parts of the code:

- Convolutional code alone;
- Code without inside interleaver;
- Code with pseudorandom inside interleaver;
- Code with block interleaver;
- Code with convolutional code of different constraint length;
- Convolutional codes different decoding techniques.

The outside interleaver is not tested in this experiment as it is designed to break burst noise but only Gaussian noise is simulated.

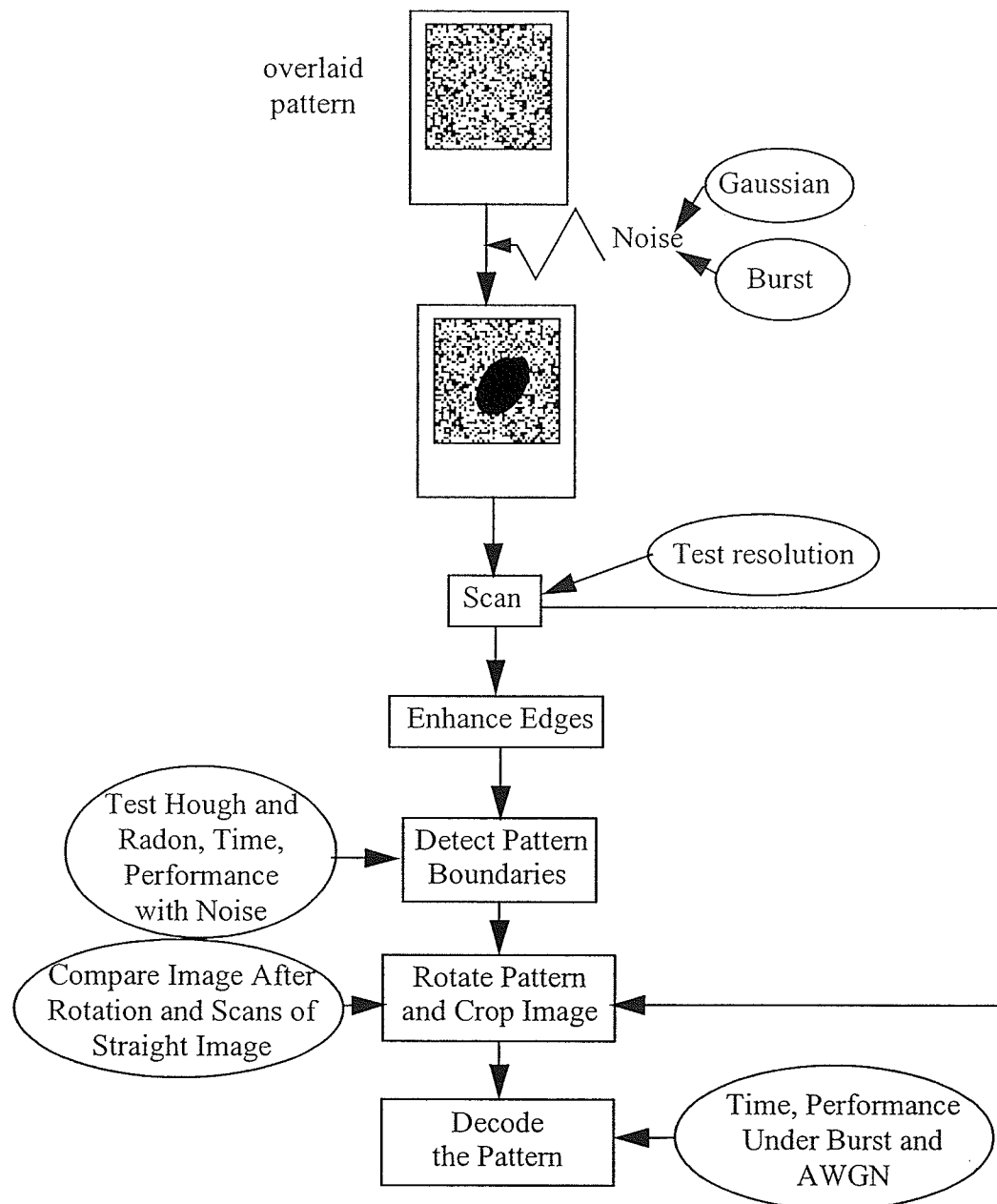


Fig. 4.2. Decoding process with experiments legends.

4.3 Pattern Functional Testing

Several combinations of coding are used to test the pattern. The experiments are aimed at determining the necessary scanning resolution for proper decoding of the pattern, determining the behavior of the pattern under burst and Gaussian noise, selecting the best line detection algorithm for performance and timing issues. Fig. 4.2. shows the flow of the decoding process and the legends indicate the experiments performed at each stage. The patterns are generated into a PBM (portable bitmap) file. The image is scaled up and printed at 37.5 dpi (dot per inch) and 75 dpi (the scaling process is to get the desired resolution on a 300 dpi printer). The data encoded in the pattern is an ascii file with the following quote:

“Be like a postage stamp. Stick to it until you get there.”

Harvey Mackay

The seed for the random interleaver is set to zero for all patterns. The pattern is printed on good quality, matte paper using a laser printer. The printing conditions are ideal, however, it provides an acceptable basis for the testing of the performance of the code under burst noise and validation of the design of the pattern and the demodulator.

The pattern is then scanned using a flat scanner. The paper is scanned straight, rotated, and shifted. The lighting conditions are ideal but the resolutions at which the pattern is scanned is representative of real world performance and is a good evaluation for the pattern demodulation performance. The pattern is scanned in equal (only for 75dpi

pattern), double, and quadruple the printing resolution to test the necessary scanning resolution. The images are scanned in black and white as well as in 256 grey levels. The scanning resolution is sufficient if the pattern without noise can be reproduced without error.

The images are transformed from TIFF format (scanner output option) to PGM format (UNIX image file format supported with some Linux libraries), demodulated and decoded using the algorithms described in the background and system design chapters. The components tested are the different edge detection operators. The Hough and Radon transforms tested under clean and noisy conditions. The demodulator has then a pass or fail test on the perfect signal reconstruction of non noisy images. The noise correction is traced for a sample image illustrating the process of interleaving and decoding.

Random bursts are generated by modifying the image after scanning and by physically altering the printed pattern.

All the software is developed under the Linux operating system using GNU development tools. The image processing tools for edge detection come from a library called ImageMagick which performs a number of operations; the Radon and Hough transforms are implemented according to Peter Toft's thesis [Toft96]; the rest are implemented from scratch and coded in C. For further information on ImageMagick, refer to the project webpage [Imag00].

4.4 Summary

This chapter described the *modus operandi* of the experiments and the motivation behind those. The experiments are designed to test the appropriate randomness of the RNG according to their applications (random interleaving and noise generation); the performance of the code starting with its individual components and ending with an overall performance test; the last experiment deals with the overall system evaluation, verifying the adequate choice of techniques for the entire system under normal and noisy conditions.

CHAPTER V

EXPERIMENTAL RESULTS AND DISCUSSION

5.1 Random Number Generator Tests

This section gives the results and discussion on the RNG used for random interleaving, uniform and, Gaussian random number generation.

5.1.1 Linear Congruential Generator

The chosen LCG is the Berkeley system distribution (BSD) implementation rand48:

$$X_{n+1} = (25214903917 \cdot X_n + 11) \bmod 2^{48} \quad (5.1)$$

$$Out_{n+1} = X_{n+1} \gg 16 \quad (5.2)$$

where \gg is the shift operator and Out_{n+1} is the 32 bit integer output.

From Theorem A p. 16 of [Knut81] and as 11 is relatively prime of 2^{48} , $b=a-1=25214903916$ is a multiple of 2 and b is a multiple of 4, the period of the LCG is $m=2^{48}>10^{14}$. This period is largely sufficient to generate our random permutations.

The spectral test results from [Enta98] are shown in Table 5.1. According to [Knut81], the RNG passes the spectral test for dimensions 2 to 8 as $\mu_s > 0.1$. According to

the author, all bad generators failed the spectral test which gives us a confidence in the good quality of the chosen generator.

Table 5.1 Spectral test results for the BSD LCG

Spectral Dimension	2	3	4	5	6	7	8
Spectral Distance μ_s	0.51	0.80	0.45	0.58	0.66	0.80	0.60

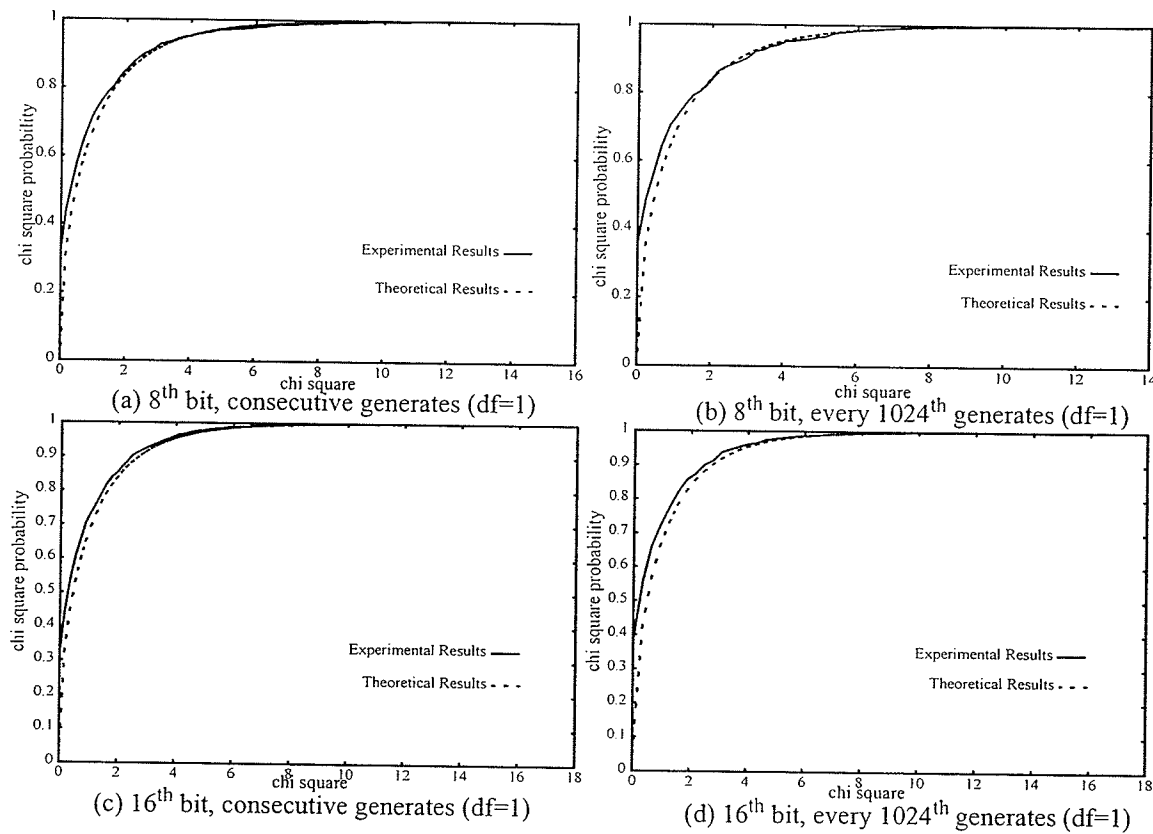


Fig. 5.1. rand48 random bit χ^2 test, theoretical and experimental results.

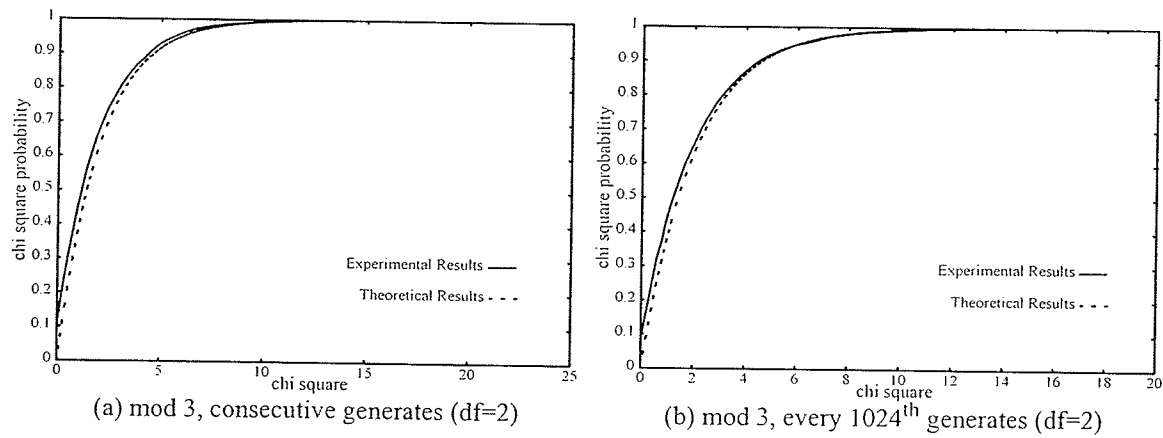


Fig. 5.2. rand48 mod 3 χ^2 test, theoretical and experimental results.

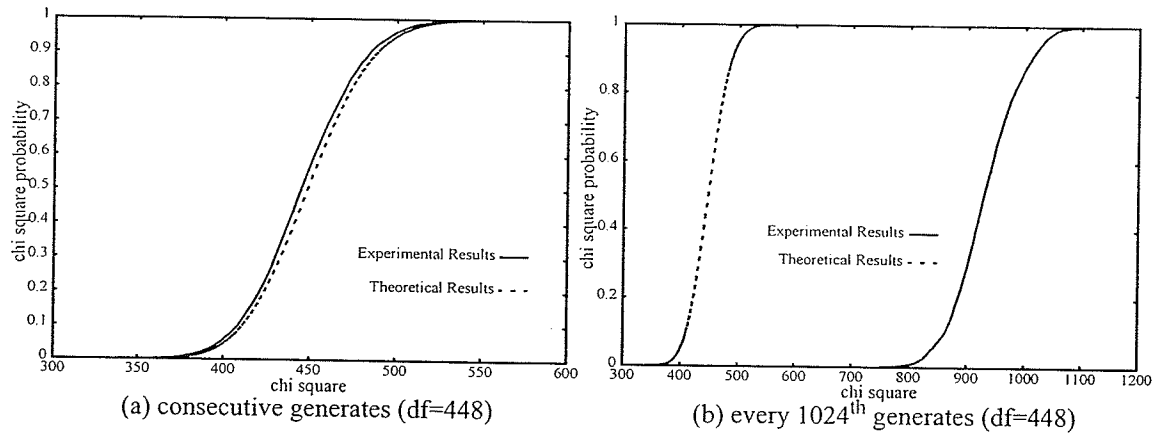


Fig. 5.3. rand48 3LSB 3-tuples χ^2 test, theoretical and experimental results.

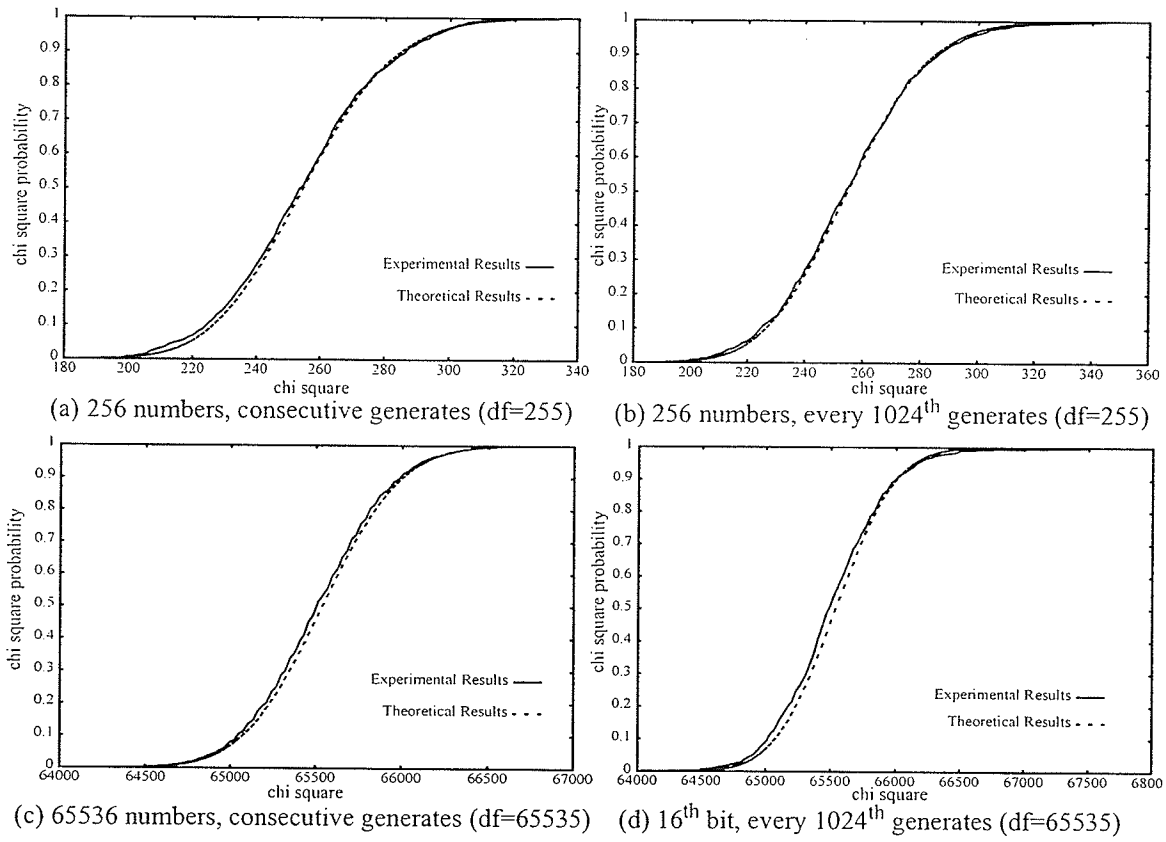


Fig. 5.4. rand48 equidistribution χ^2 test, theoretical and experimental results.

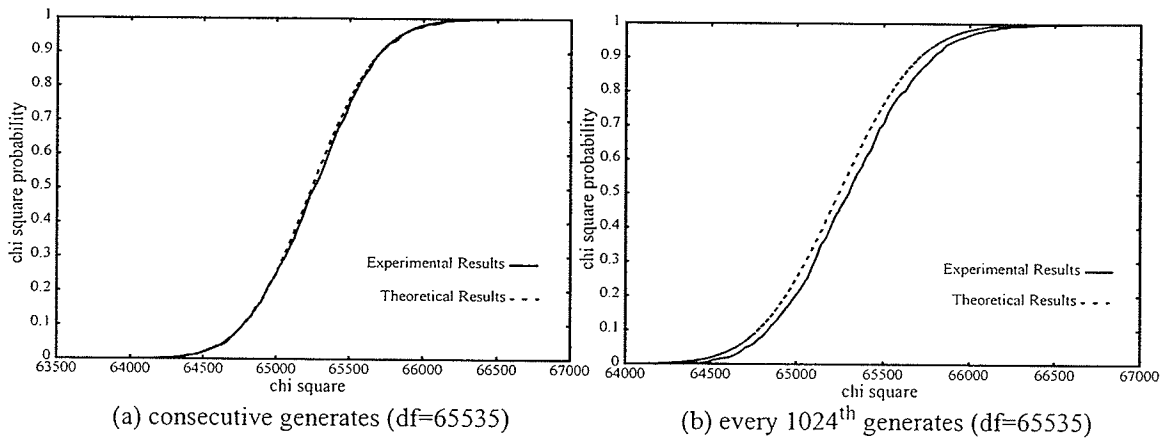


Fig. 5.5. rand48 8LSB pairs χ^2 test, theoretical and experimental results.

The BSD generator also passes all the empirical χ^2 test (as shown in Fig. 5.1. to Fig. 5.5.) except the 3 lower bits 3-tuples test when each sample are separated by 1023 generated numbers. For this test, $\chi^2=961$, $k=449$. This happens with a probability less than 10^{-9} . The generator fails the test for the following reasons:

Corollary 5.1:

if

$$X_{n+1} = (aX_n + b) \bmod(m) \quad (5.3)$$

and $p|m$ and if

$$X_{n+1}^{(p)} = X_n \bmod(p) \quad (5.4)$$

then

$$X_{n+1}^{(p)} = (aX_n^{(p)} + b) \bmod(p) \quad (5.5)$$

Proof:

from Eq. (5.3)

$$X_{n+1} = aX_n + b + k_1m = aX_n + b + k_1up \quad (5.6)$$

from Eq. (5.4)

$$\begin{aligned} X_{n+1}^{(p)} &= X_{n+1} + k_2p \\ &= aX_n + b + k_1up + k_2p \\ &= aX_n^{(p)} + b - k_3p + k_1up + k_2p \end{aligned} \quad (5.7)$$

Hence Eq. (5.5) is true. \square

In the case of the BSD generator:

$$X_{n+1}^{(2^e)} = (aX_n^{(2^e)} + b) \bmod 2^e \quad 1 \leq e \leq 48 \quad (5.8)$$

For example, if $e=1$, the least significant bit has period 2 or less. In the same manner, the first two bits have period 4 or less and the 16^{th} bit has period 2^{16} or less. The test then fails because it takes the 16^{th} , 17^{th} , and 18^{th} bit of numbers that are 2^{10} apart so the least significant bit of the new set of numbers has period $2^{16}/2^{10}=2^6=64$ or less, the second 128 or less and the third 256 or less. Furthermore, the test is performed in a 3 dimensional space which increases the clustering of the points.

To avoid such a behavior, the implementation of the random interleaver picks its numbers from the most significant bits of the generated numbers.

Corollary 5.2: The n^{th} bit of the BSD generator has period exactly 2^n .

Proof:

(X_n) has period 2^{48} , and all bits lower than the 48^{th} have period 2^{47} or less so the 48^{th} has a period of exactly 2^{48} . Furthermore, from Theorem 5.1, we know that $X_{n+1}^{(2^k)} = (aX_n^{(2^k)} + b) \bmod(2^k)$ As well 2 does not divide a and $a-1$ is divisible by 4. Hence, $(X_n^{(2^k)})$ has period 2^k and from the same reasoning as for the 48^{th} bit, the k^{th} bit has period exactly 2^k . \square

The maximum number needed by the interleaver is 2^{16} so only the 16 MSB are needed. The lowest extracted bit has a period of 2^{32} and our application only needs a maximum of 2^{16} numbers. The period is thus large enough.

5.1.2 Pseudo DES RNG

The pseudo DES (pDES) RNG is described and implemented in [PTVF92]. The RNG is tested using the mod3, equidistribution of the 8 MSB, MSB, pairs of 8 MSB, equidistribution of the 16 MSB, triplets of 3 MSB tests, all of which are performed with and without gap of 2^{10} .

The χ^2 CDF is generated using the algorithm from [HiPi85]. 10,000 numbers are generated to calculate each χ^2 value and calculate its probability. 1,000 χ^2 values are calculated. The resulting χ^2 probability distribution is plotted along with the theoretical CDF. Results are shown in Fig. 5.6. to Fig. 5.10.. The pDES passes all the χ^2 tests.

5.2 Code Under AWGN

Several experiments are performed on the code to test for the best RS block size, convolutional code constraint length, and interleaver type and size. The experiments are also aimed at verifying assertions regarding the design choices of the code such as CSOC performance, optimum interleaver size and type, and RS optimum size. The overall concatenated code performance evaluation is broken down by the evaluation of the CSOC

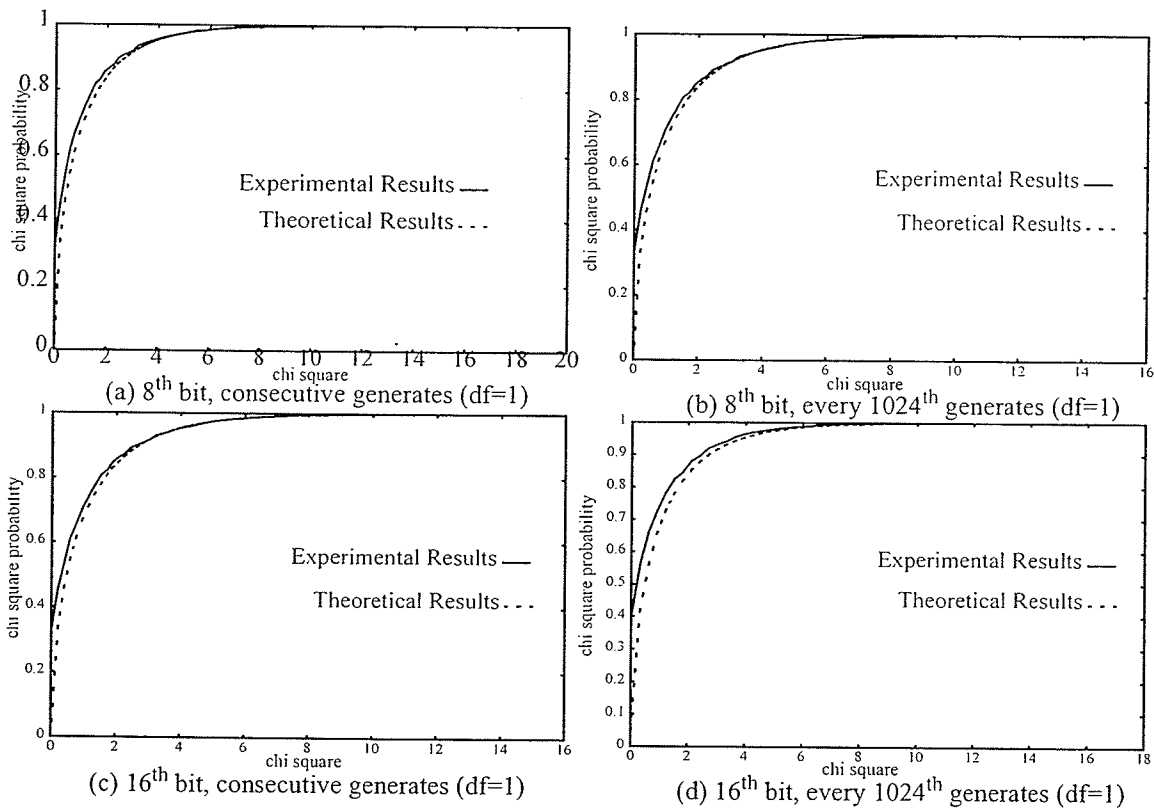


Fig. 5.6. pDES random bit χ^2 test, theoretical and experimental results.

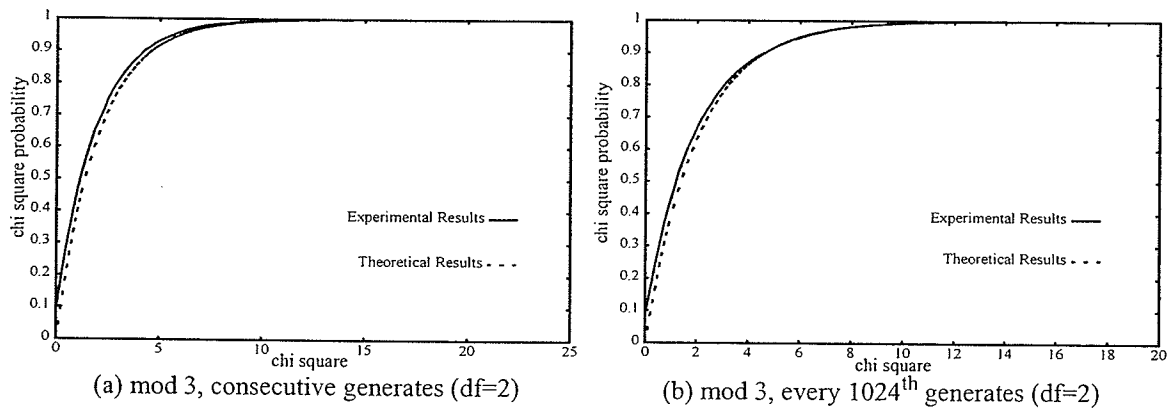


Fig. 5.7. pDES mod 3 χ^2 test, theoretical and experimental results.

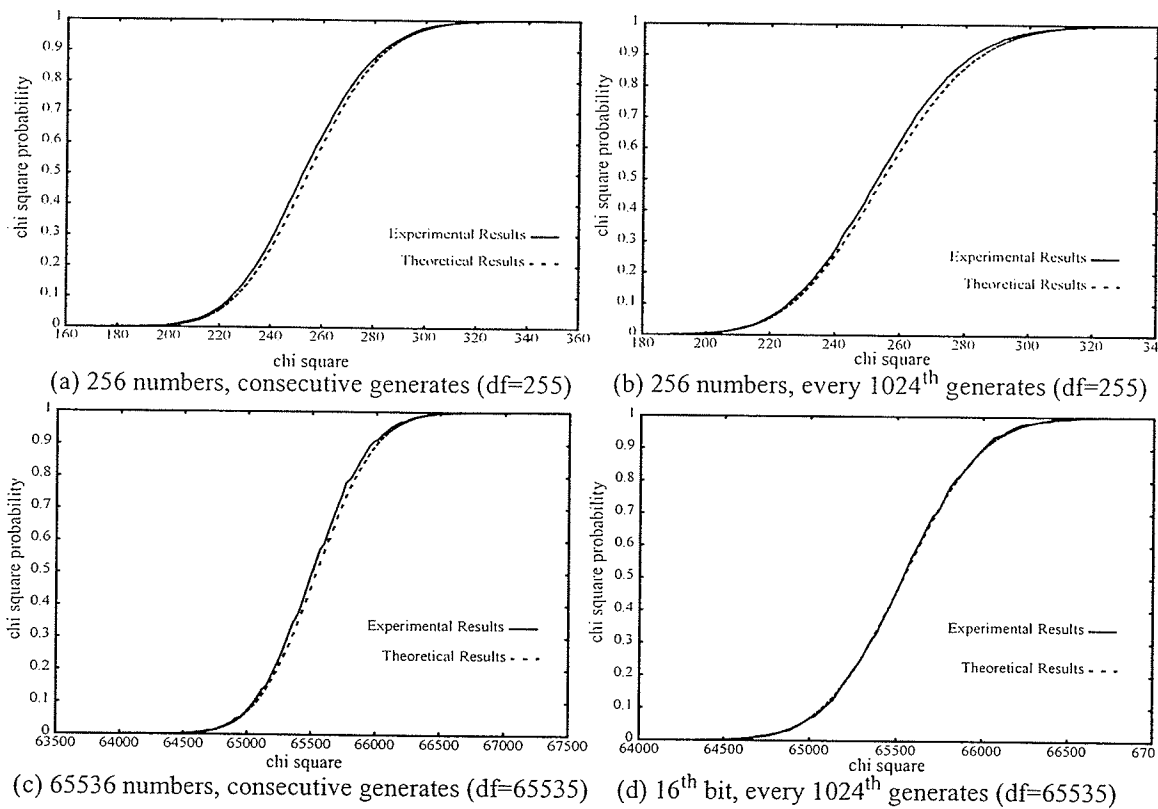


Fig. 5.8. pDES equidistribution χ^2 test, theoretical and experimental results.

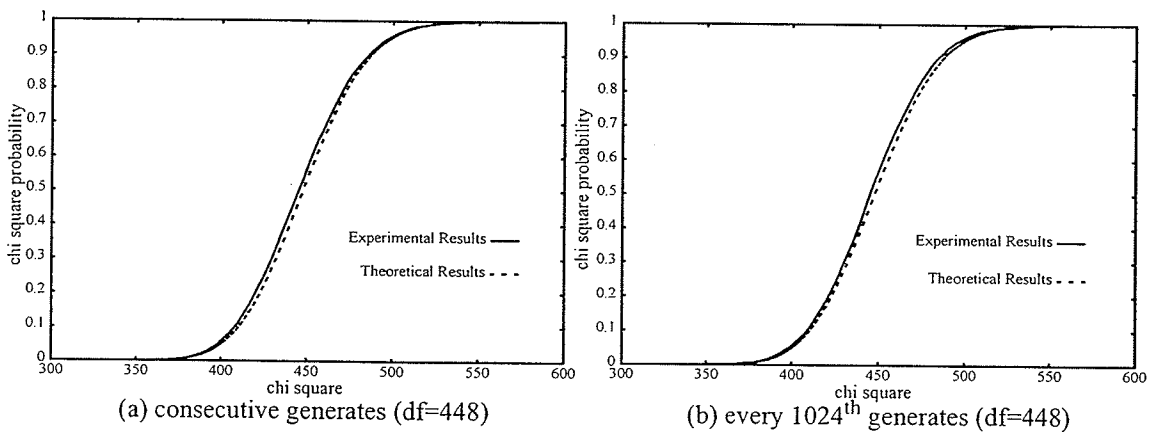


Fig. 5.9. pDES 3LSB 3-tuples χ^2 test, theoretical and experimental results.

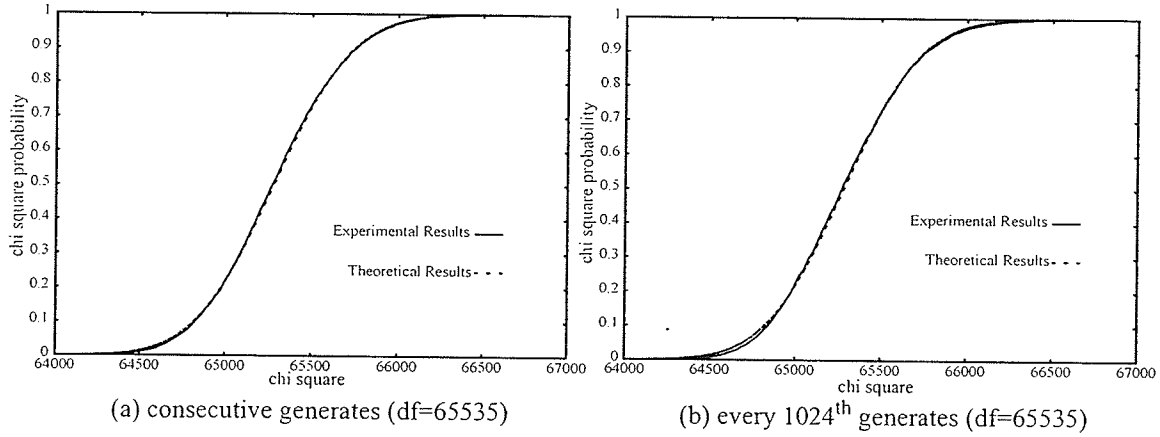


Fig. 5.10. pDES 8LSB pairs χ^2 test, theoretical and experimental results.

performance with different constraint length, then the experimentation of RS codes with different block sizes and close rates; finally, the concatenated code is tested with no interleaver, random interleaver, block interleaver, and symbol block interleaver.

5.2.1 Non Coded Transmission

The following result is the simulation of bit transfer under AWGN using bipolar binary modulation. The AWGN is generated using the ran4 RNG and the Box-Muller algorithm. The theoretical error performance under such conditions is given by [Hayk94]

$$BER = \frac{1}{2} \operatorname{erfc} \left(\sqrt{\frac{E_b}{N_0}} \right)$$

where E_b is the energy per bit, $N_0/2$ is the power spectral density of the Gaussian noise, and erfc is the complementary error function given by

$$\operatorname{erfc}(u) = \frac{2}{\sqrt{\pi}} \int_u^{\infty} \exp(-z^2) dz$$

The error function is generated using the Linux function provided with the mathematical library. Both theoretical results and simulation results are plotted in Fig. 5.11.. It can be observed that there is a close match between the two so the noise and signaling simulations are properly implemented.

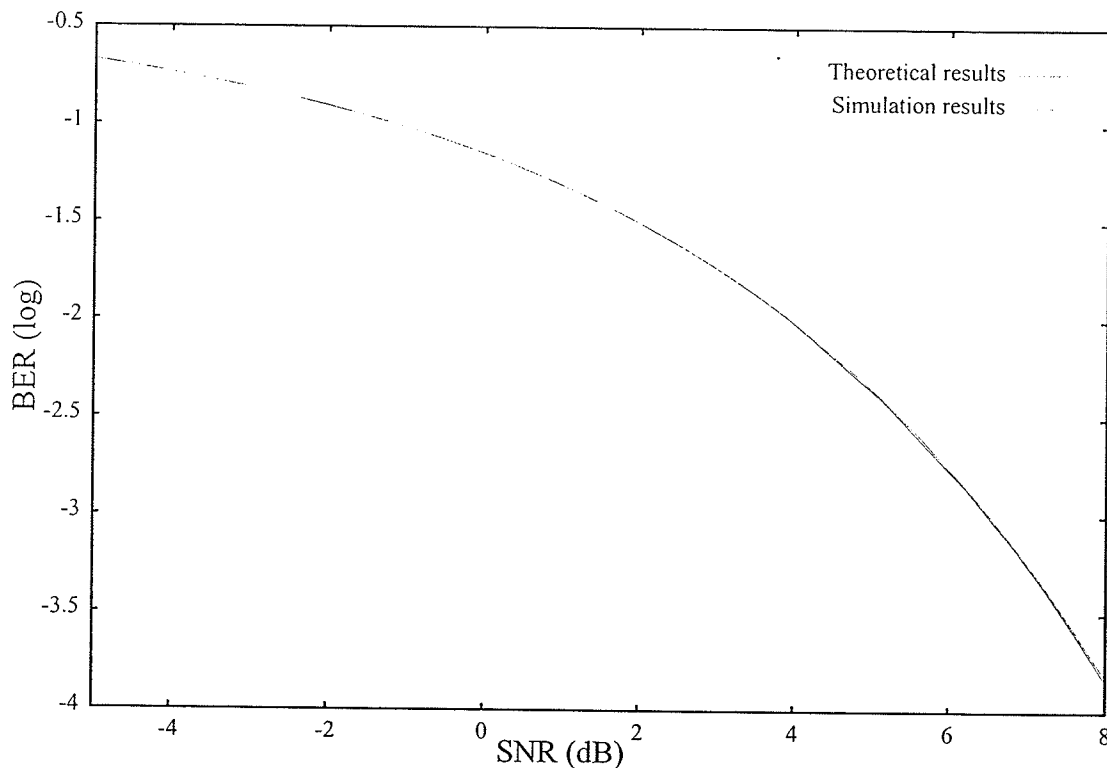


Fig. 5.11. Non-coded data transmission using bipolar binary modulation theoretical and simulation performance

5.2.2 Convolutional Code Performance

(2,1) CSOCs of different constraint lengths 2, 7, and 18 are tested. The graph represents the BER against SNR. CSOCs are decoded using the majority decoding algorithm. The results also include the non coded performance.

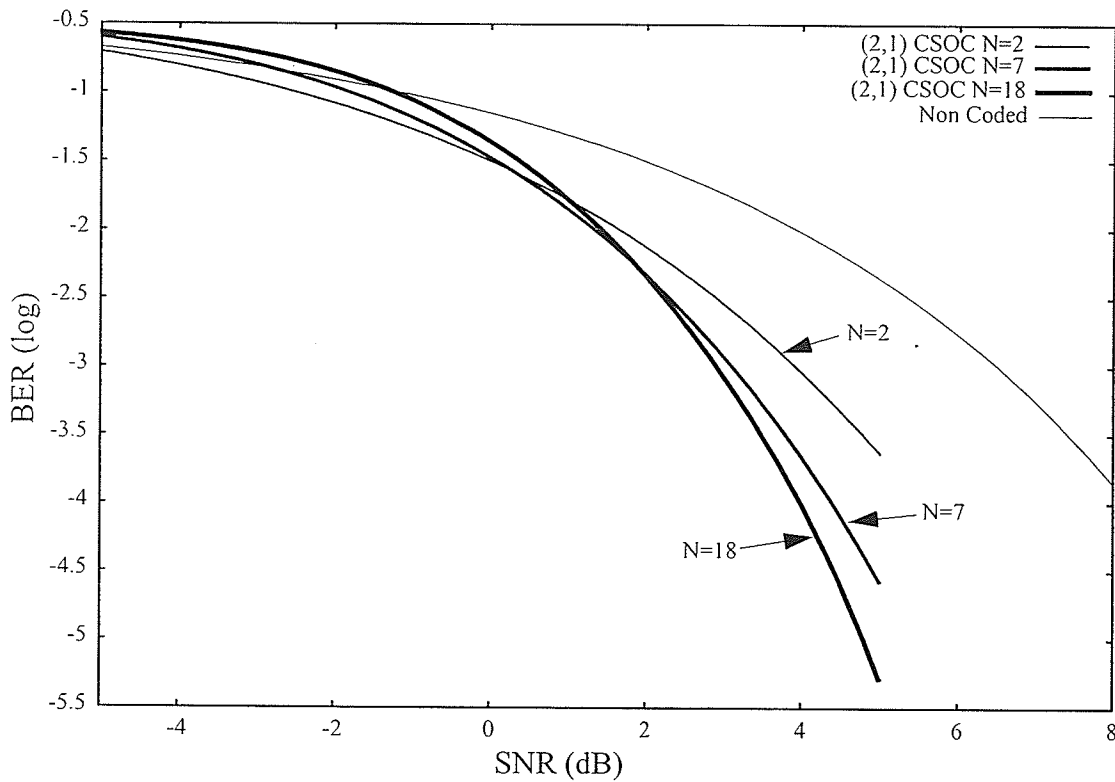


Fig. 5.12. (2,1) CSOC performance with constraint length 2, 7, and 18.

In Fig. 5.12., it can be observed that the longer constraint length codes have better performance at higher SNR by a good margin, e.g. 1dB at BER of 10^{-3} (and it increases) but lower constraint length perform better at low SNR but not by much (approximately 0.2dB.) The crossover of performance happens at around 2dB. Furthermore, a decreasing performance gain is achieved as the constraint length increases. So, for high SNR, it is not necessary to take the largest constraint length code but an optimal length code can be found after which no significant gain is added. The monotony condition necessary for the inner code design is verified by the CSOC (no waterfall performance curve). A BER of 10^{-2} is obtained at around or more than 1 to 2dB which gives us the operating point of the concatenated scheme according to the code design section. In [CaCl81], the author

observed through simulation that 8-level quantized threshold decoding gives an improvement of 1.7dB over majority decoding. The authors also state that feedback decoding increases the performance of approximately 0.6dB in both soft and hard decision decoding. For the concatenated code, all constraint length CSOC perform approximately the same in the operating point region. Of course if a higher SNR can be achieved, the higher constraint length code should be used. However, it should also be considered that a small extra bandwidth expansion of the length of the constraint length is produced by convolutional codes. Furthermore, the gain diminishes as the constraint length increases so 18 looks like a good compromise between decoder complexity and code performance.

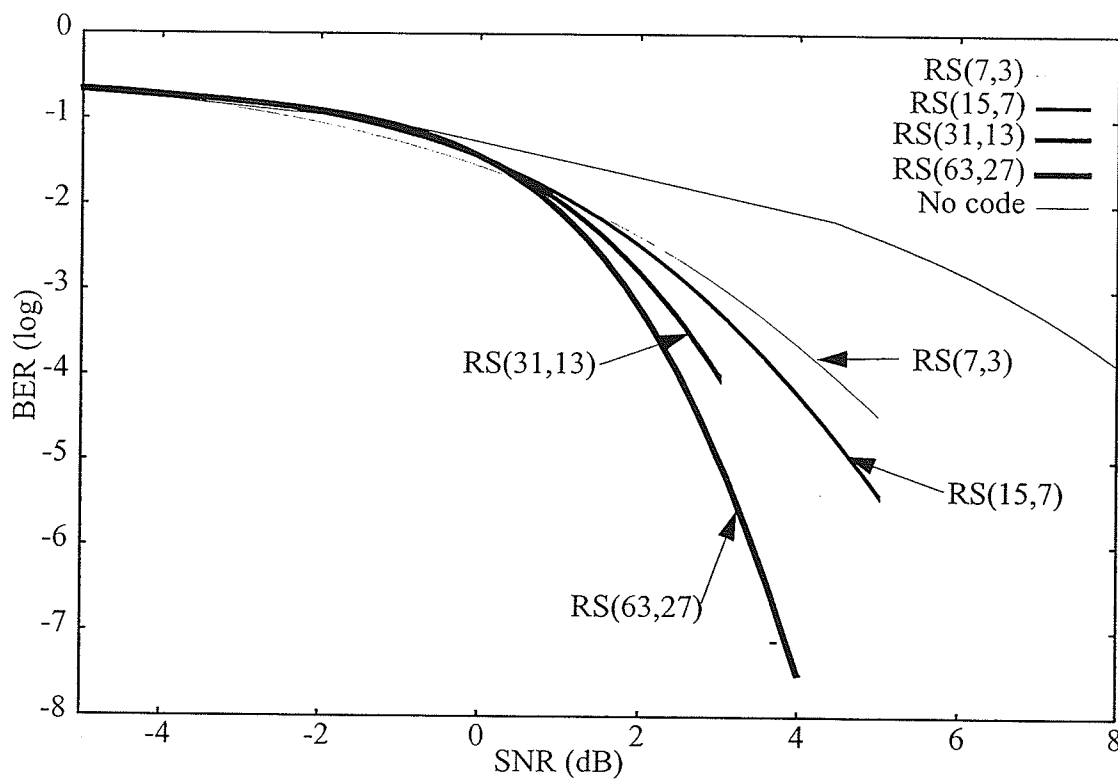


Fig. 5.13. RS performance for RS(7,3), RS(15,7), RS(31,13), and RS(63,27).

5.2.3 RS Performance

The RS codes are characterized by their symbol size and their designed distance (equivalent to their error correcting power as they are MDS codes). The simulation is performed on codes with similar rates: RS(7,3), RS(15,7), RS(31,13), and RS(63,27) with respective rates of 0.428, 0.467, 0.419, and 0.428. The first and last code have identical rates. This experiment is performed to get an idea on which block size should be used for better performance. As seen in Fig. 5.13., similar results apply to the block length of RS codes and constraint length of convolutional codes: at low SNR, lower length codes perform slightly better than longer length codes. However, all codes performance is close to the performance without code which makes the use of codes highly inefficient in such a case; the cross over of all code performance is around 1dB and afterwards, longer size codes perform significantly better. It can also be noted that the performance gain for higher block sized codes does not seem to decrease like the performance of convolutional codes with higher constraint length.

Those results are validated using an upper bound on performance. A decoding error occurs whenever $t+1$ or more symbols are in error. The probability of decoding error is

$$\sum_{i=t+1}^n \binom{n}{i} p_s^i (1-p_s)^{n-i}$$

where p_s is the probability of symbol error and $1-p_s$ is the probability of correct symbol.

The equation illustrates the choice of i positions in n with each chosen i symbols having a

probability of error p_S and each $n-i$ remaining bits having a probability $1-p_S$ of being correct. When a codeword is incorrectly decoded, it leaves at most $i+t$ symbol errors (i are the symbols already in error considering they are all in the information part of the codeword and t is the maximum number of errors that can be introduced when decoding the codeword). The probability of a symbol being in error when a codeword is in error is then $\frac{i+t}{n}$. Considering as an upper bound that all bits in a symbol are in error when a symbol is in error the probability of error bit is then bounded by

$$p_b^{(RS)} \leq \sum_{i=t+1}^n \frac{i+t}{n} \binom{n}{i} p_S^i (1-p_S)^{n-i} \quad (5.9)$$

On the binary symmetric channel, the probability of symbol error is one minus the probability of correct symbol. A symbol is correct if all its bits are correct. If the probability of bit error in the channel is $p_b^{(c)}$ and all bits are probabilistically independent a symbol is in error with a probability of

$$p_S = 1 - (1 - p_b^{(c)})^m \quad (5.10)$$

Using Eq. (5.9) and Eq. (5.10) we finally get

$$p_b^{(RS)} \leq \sum_{i=t+1}^n \frac{i+t}{n} \binom{n}{i} (1 - (1 - p_b^{(c)})^m)^i (1 - p_b^{(c)})^{m(n-i)} \quad (5.11)$$

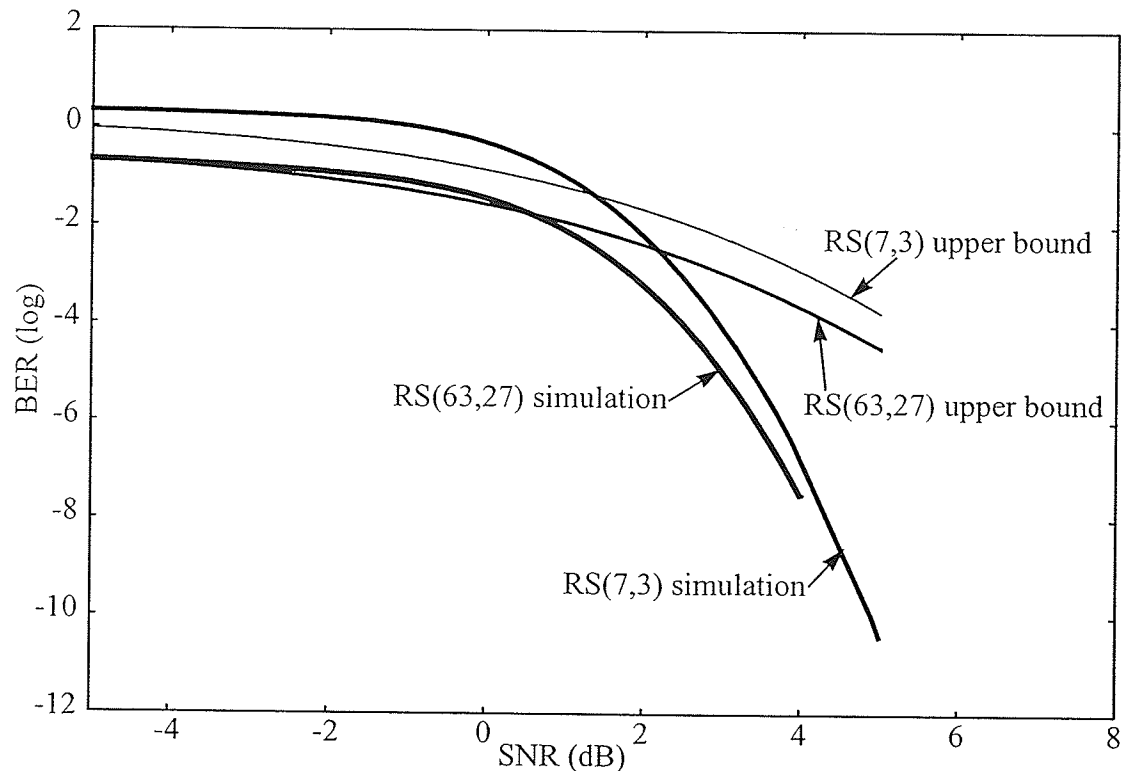


Fig. 5.14. Theoretical bound and simulated RS(63,27) and RS(7,3) performance.

Fig. 5.14. compares the theoretical upper bound and the simulated results. The simulated curve is always under the theoretical which is a validation of our simulation and of the correctness of the error correction power of the code. Even if the bound is not a close bound, it still helps evaluate performance of codes at high SNR (where it is too computationally expensive to simulate) and to compare RS codes between each other.

The overall performance of the code can be estimated by assessing the performance of the convolutional code and tabulating it against the RS code. If the SNR is at 2dB, the convolutional code leaves a BER of 10^{-2} . Considering that the interleaver in between the two codes recreates an independent noise, the BER of 10^{-2} is equivalent to a

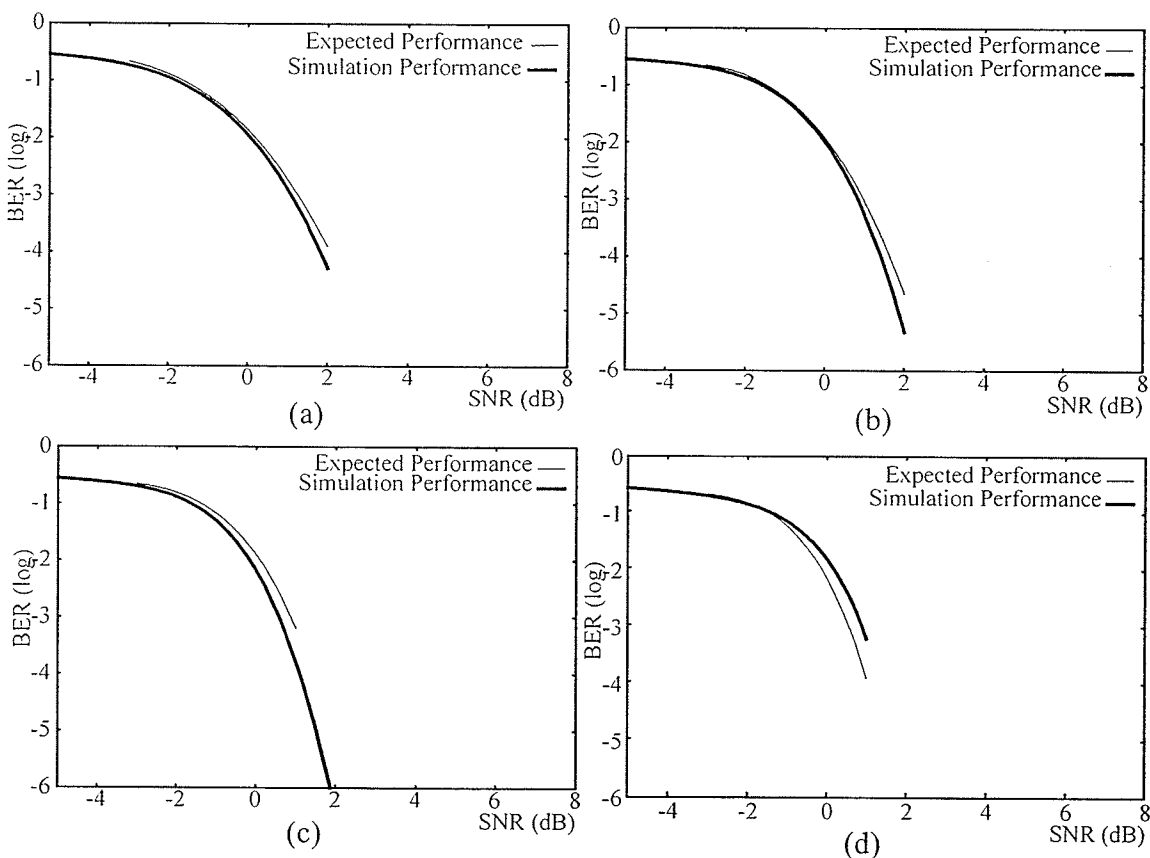


Fig. 5.15. Concatenated codes simulated performance compared to expected performance with (2,1), $N=18$ CSOC, and (a) RS(7,3), (b) RS(15,7), (c) RS(31,13), and (d) RS(63,27).

SNR of 4.5dB according to Fig. 5.11.. The overall code performance at 2dB should then be around the RS performance at 4.5dB.

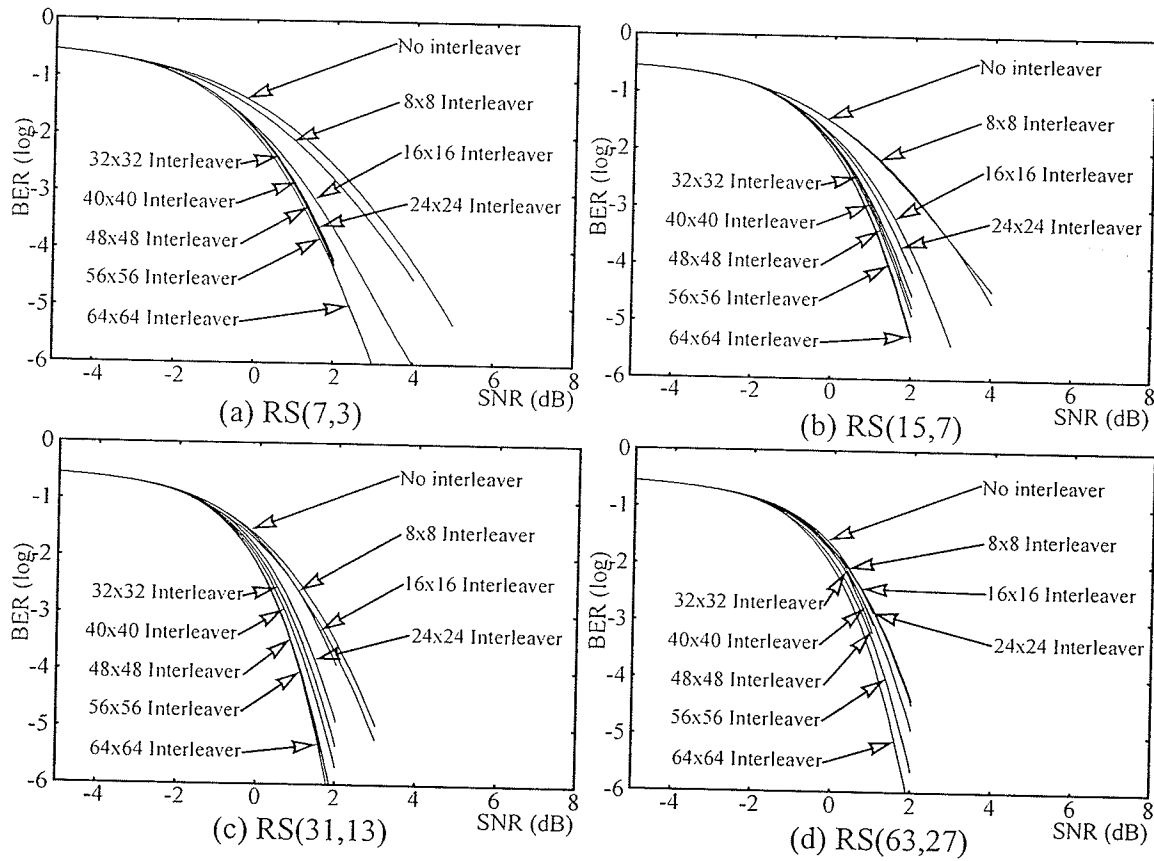


Fig. 5.16. Concatenated codes performance for various block interleavers with (2,1), $N=18$ CSOC, and (a) RS(7,3), (b) RS(15,7), (c) RS(31,13), and (d) RS(63,27).

5.2.4 Interleaver Type and Performance

5.2.4.1 Code Performance with and without Interleaver

The concatenated code performance is assessed without inside interleaver and with block interleaver of size 8x8, 16x16, 24x24, 32x32, 40x40, 48x48, 56x56, and 64x64 (see Fig. 5.16.). The code performs better with an interleaver. As the convolutional code leaves burst noise at the end of its decoder, the interleaver helps break those bursts in several RS blocks and thus enhances the performance of the overall code. The interleaver transforms the burst noise at the output of the convolutional code into random noise so the

performance of the code should tend to the estimated performance using the method described at the end of Section 5.2.3. There is a maximum length of the interleaver after which no additional gain is performed. Figure 5.15 shows a comparison of the expected performance and the performance with a 64x64 interleaver. The simulated code performs better within a few point of a dB for all RS codes except for the (63,27) code. However, the RS(63,27) performance is the only one to still increase with the interleaver of size 64x64 so the expected performance can be reached or outperformed for larger interleaver sizes. This section justified the use of the interleaver between the RS code and the convolutional code. It also showed that the goal performance is reached or outperformed as the interleaver size grows. The next section compares the concatenated code performance with block and random interleaver.

5.2.4.2 Random Interleaver vs. Block Interleaver Performance

Random interleavers and block interleavers working on the same number of bits are compared (e.g., an 8x8 block interleaver is compared with a 64 random interleaver). The random interleaver is based on the rand48 RNG which as we saw in Section 5.1.1 has a period long enough for the numbers generated to be independent. The theory in Section 2.6 is once more verified by experimentation: the random interleaver performs worse or equivalently to the block interleaver at equivalent sizes (see Fig. 5.17.). Random interleaver which are twice the size of block interleavers are needed for equivalent performance (see Fig. 5.18.).

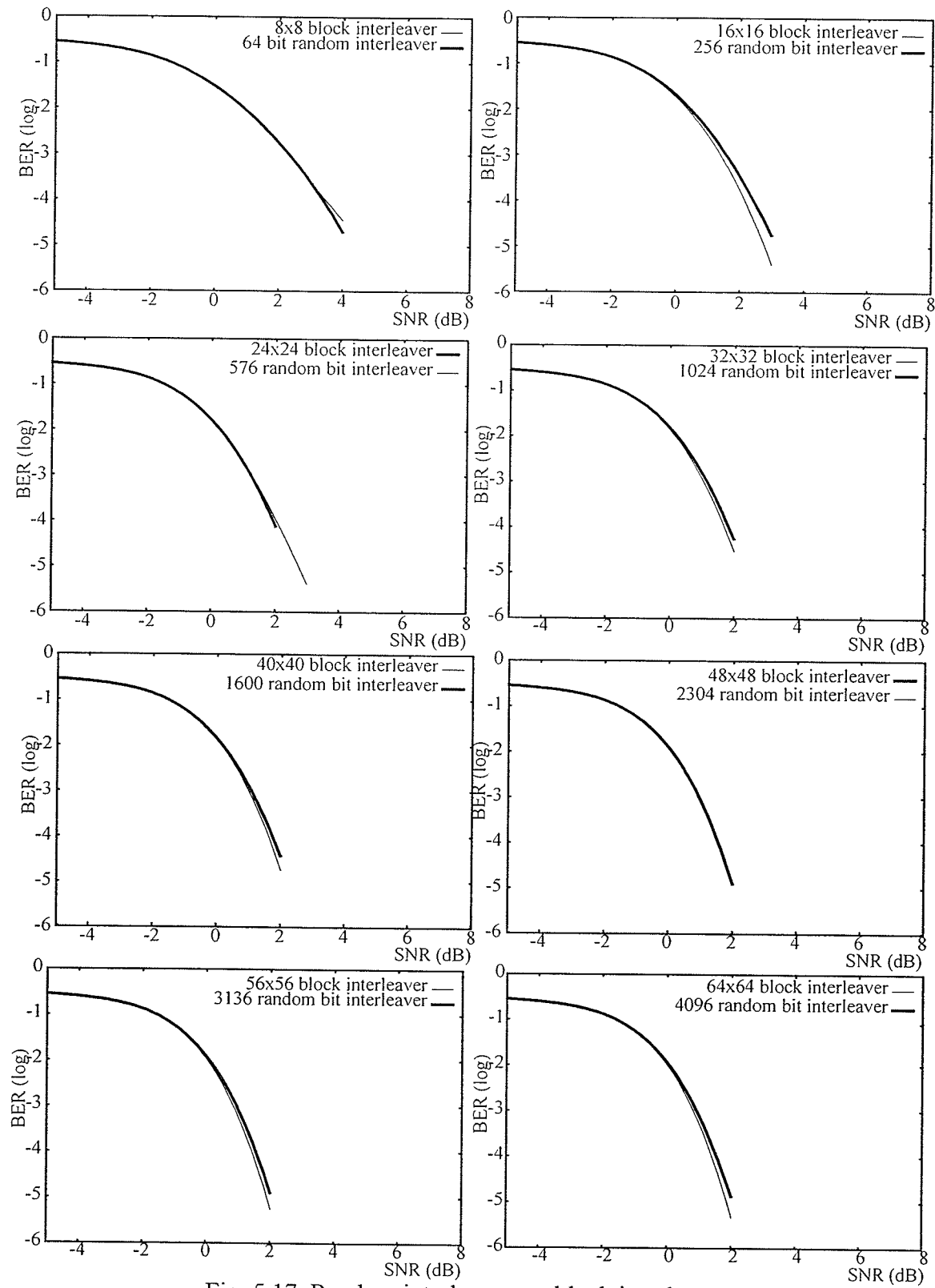


Fig. 5.17. Random interleaver vs. block interleaver.

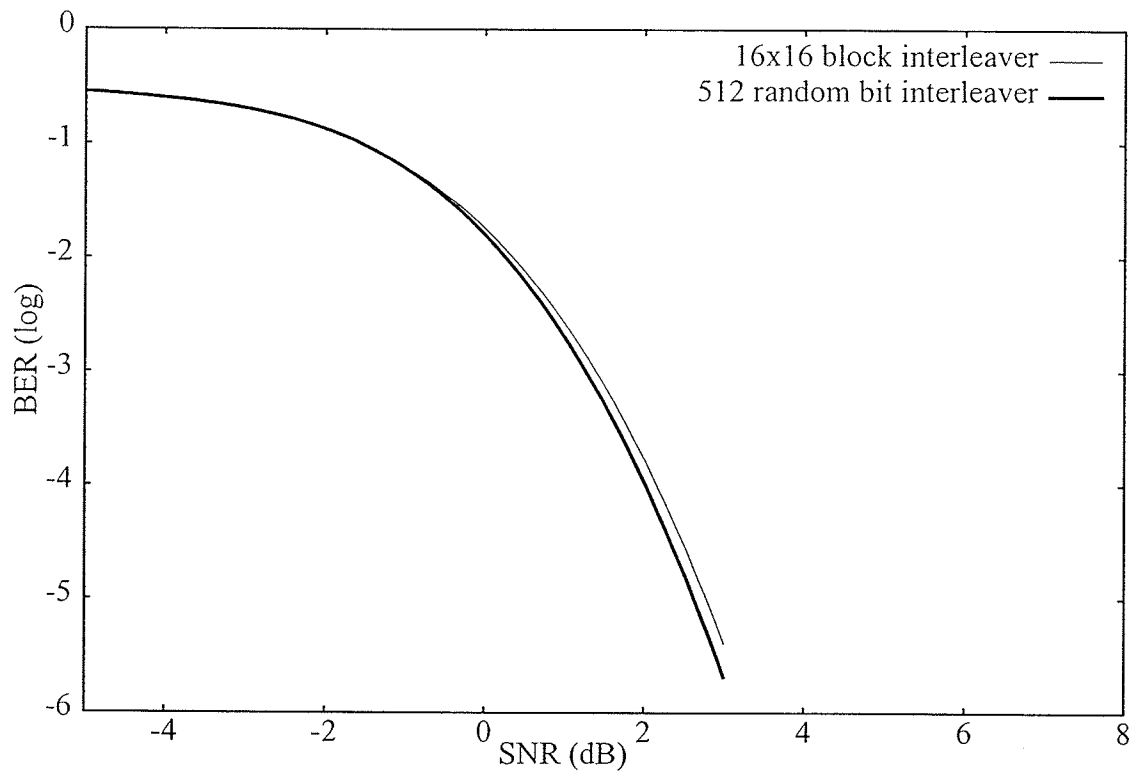


Fig. 5.18. Concatenated code with a double size random interleaver (512 bits) vs. 16x16 block interleaver.

The use of block interleavers is then better than random interleavers for the specific burst noise characteristic at the output of the convolutional decoder. The next section compares the performance of the concatenated code with bit block interleaver and symbol block interleaver.

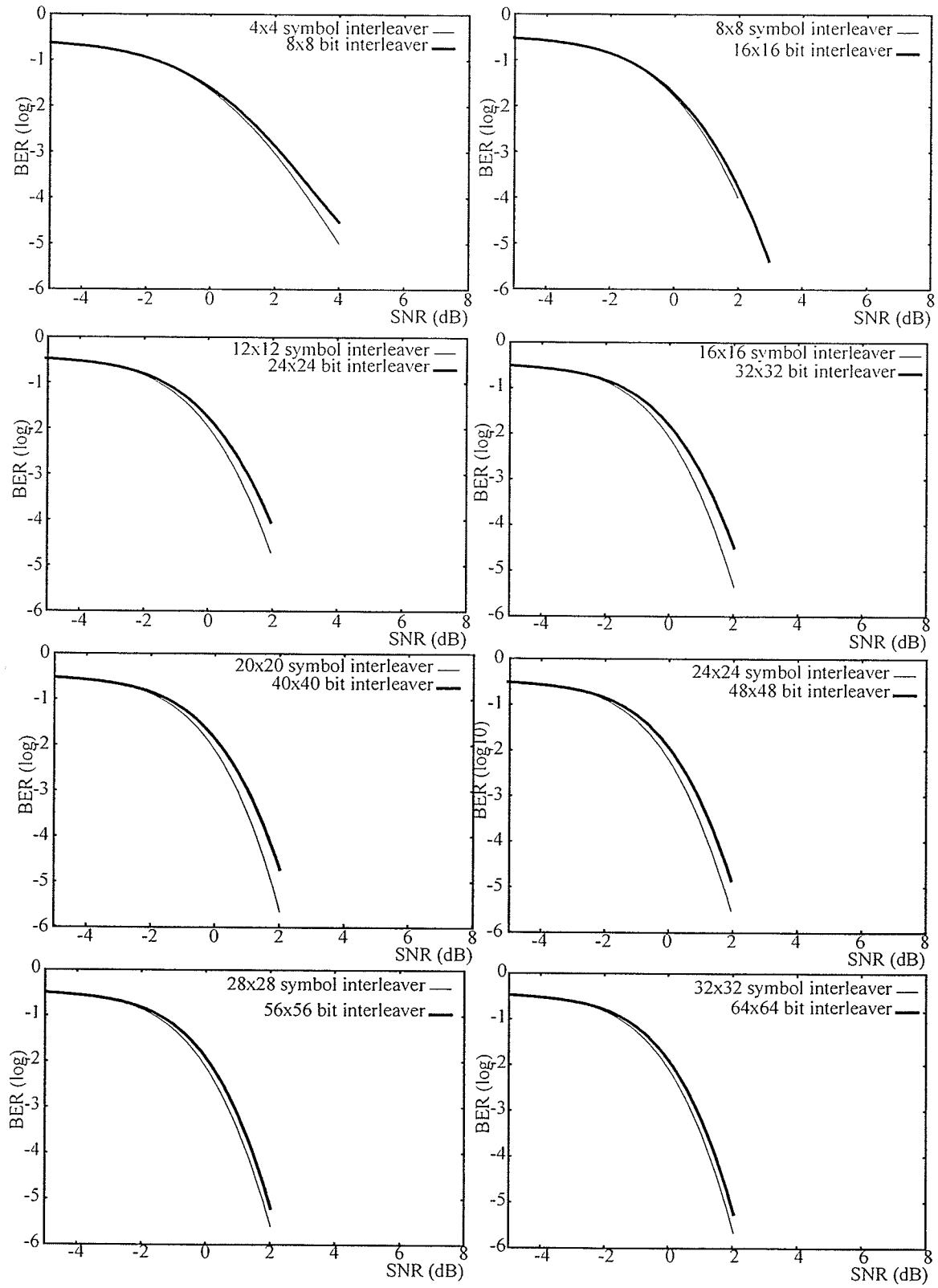


Fig. 5.19. Symbol interleaver vs. block interleaver for RS(15,7), N=18.

5.2.4.3 Bit Block Interleaver vs. Symbol Block Interleaver

Intuitively, the bit block interleaver is not optimum for the RS code decoder that follows it: if a burst of bits of the size of the RS symbol are in error, the bit block interleaver spreads the bits on several symbols which can produce a decoding error if the affected symbols number is higher than the error correcting capability of the code. The goal is then to use an interleaver that works on the RS symbols instead of single bits. The experiment is performed on interleaver working on the same number of bits. For example, a 24×24 bit block interleaver (which processes 576 bit at a time) is compared with a 12×12 symbol block interleaver with symbols of 4 bit (which processes $12 \times 12 \times 4 = 576$ bit at a time). The results using different interleaver sizes with RS(15,7) and CSOC with $N=18$ is shown in Fig. 5.19.. It can be observed that the symbol interleaver always outperforms the bit interleaver even at large interleaver sizes. The use of symbol interleavers is then recommended over bit interleavers.

5.2.5 CSOC and RS choice

Figure 5.20 illustrates the performance of the concatenated code with RS(7,3), bit interleaver of different sizes, and CSOC of constraint length $N=7$ and $N=18$. The lower constraint length CSOC outperforms the longer constraint length with small or no interleaver as the decoding errors at the output of the longer constraint length CSOC are more clustered in longer bursts hence the RS code makes a decoding error. As the interleaver length increases, the code performs closer to the expected performance as the

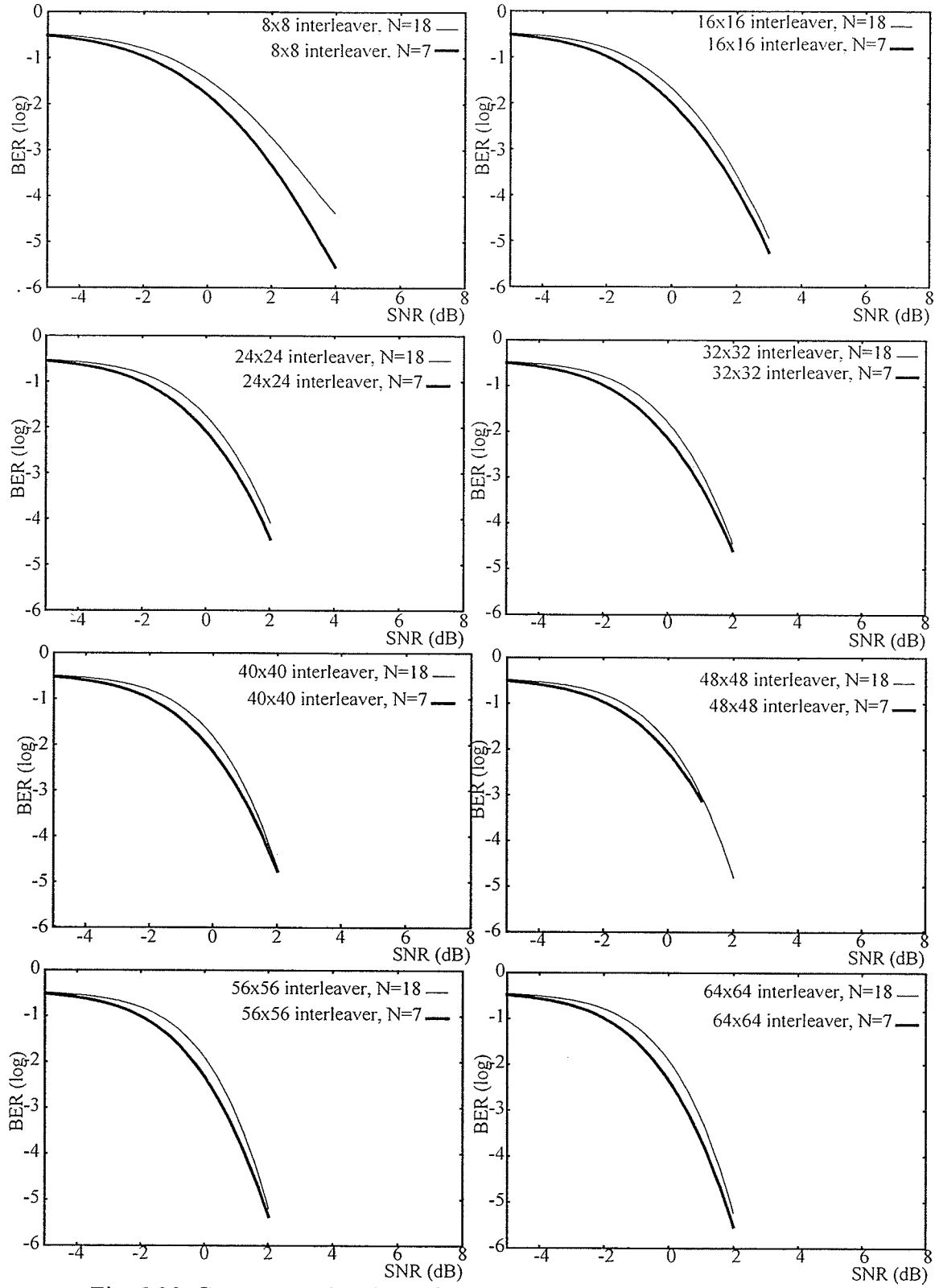


Fig. 5.20. Concatenated code performance with RS(15,7), N=7 and N=18.

output noise of the convolutional code decoder is closer to random. From Fig. 5.12., the cross over performance between the two CSOC codes happens at 2dB; as the interleaver size increases, the cross over point becomes 2dB which illustrates the fact that the code performance approximates better and better the expected performance of the code. The CSOC code should then be chosen according to the expected functioning SNR point of the communication system: if the SNR is less than 2dB, N=7 is a better choice but if the SNR is greater or equal to 2dB, N=18 outperforms largely the other code.

The choice of the RS code does not make a difference for middle SNR less than 0.5dB. If the SNR at the output of the convolutional decoder and the interleaver is considered random, it is equivalent to a BER of $10^{-1.2}$ and for the concatenated code with N=2 it is a SNR less than -1.36dB, for N=7, -0.96dB, and for N=18, -0.45dB. For any SNR larger than the previous values, the bigger the RS block size, the better the performance. As an example, for a SNR of 2dB and CSOC N=18, the concatenated code with RS(7,3) has a BER of $10^{-4.27}$ where the code with RS(31,13) has a BER of $10^{-6.43}$. At 2dB, the performance is multiplied by more than ten each time one bit is added to the symbol size.

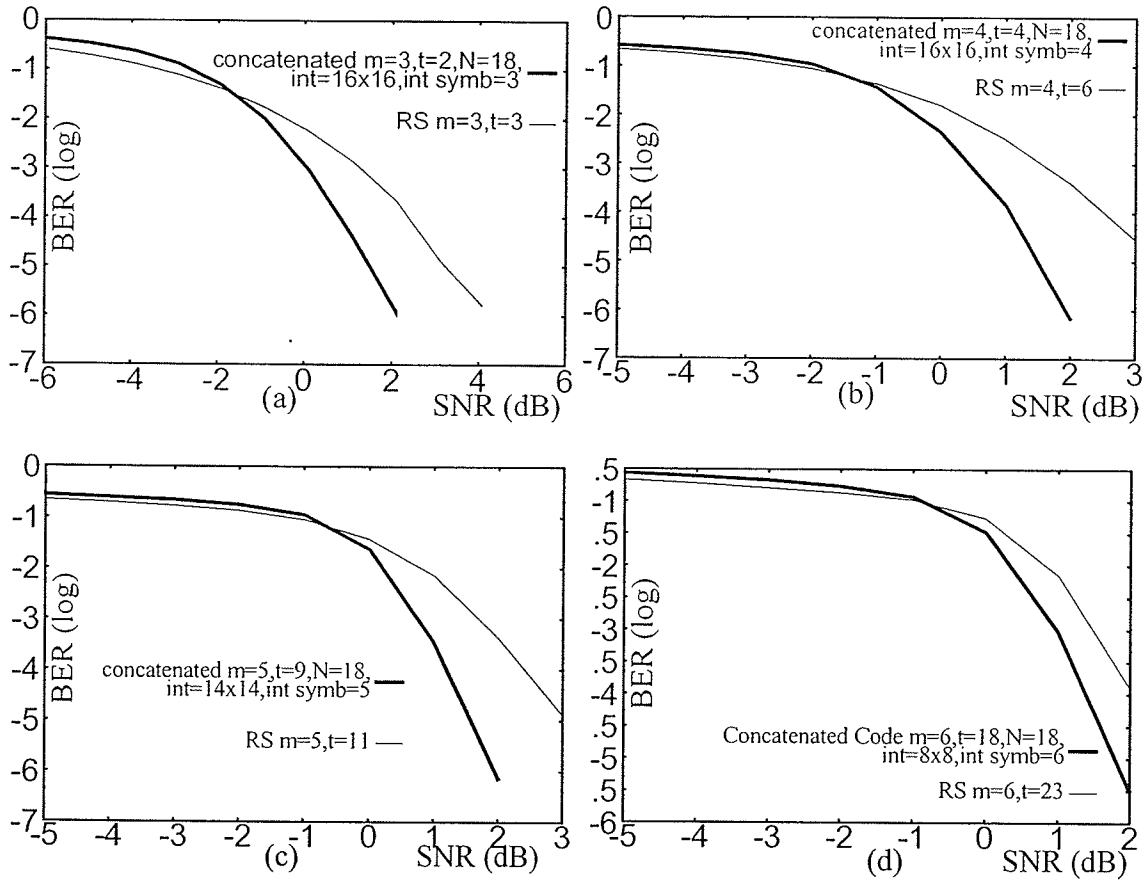


Fig. 5.21. (a) $\text{conc}(m=3, t=2, N=18, I=16, m_i=3)$ with $\text{RS}(m=3, t=3)$, (b) $\text{conc}(m=4, t=4, N=18, I=16, m_i=4)$ with $\text{RS}(m=4, t=6)$, (c) $\text{conc}(m=5, t=9, N=18, I=14, m_i=5)$ with $\text{RS}(m=5, t=11)$, and (d) $\text{conc}(m=6, t=18, N=18, I=8, m_i=6)$ with $\text{RS}(m=6, t=23)$.

5.2.6 RS and Concatenated Code Comparison

The performance under gaussian noise of the concatenated code using $N=18$ and a symbol interleaver is compared to the performance of the RS code alone. Codes with comparable rates are chosen. The concatenated code has an output rate of $(n-2t)/2n$ and the RS code has an output rate of $(n-2t')/n$. The unknown is t' , the number of errors correction capability of the RS code. Solving for t' we get:

$$t' = \frac{n + 2t}{4} \quad (5.12)$$

The codes compared are: conc(m=3,t=2,N=18,I=16,mi=3) with RS(m=3,t=3), conc(m=4,t=4,N=18,I=16,mi=4) with RS(m=4,t=6), conc(m=5,t=9,N=18,I=14,mi=5) with RS(m=5,t=11), and conc(m=6,t=18,N=18,I=8,mi=6) with RS(m=6,t=23) (see Fig. 5.21.).

The concatenated code performs significantly better after 0dB and a little worse below 0dB. However, the code is used at a point where the BER is much less than 10^{-3} so the performance before 0dB is irrelevant.

5.2.7 Concatenated Code and RS Code Timing

To prove the fact that the concatenation of two codes while keeping the same rate can increase performance and speed, both the RS alone and the concatenated code are timed on the coding and decoding of a million bits with AWGN noise. The two codes are selected to have similar rates as described in the previous section. The results are shown in Table 5.2. The concatenated code is always faster at coding and decoding but really makes a difference when decoding under heavy noise conditions: at 0dB, the RS code takes twice as long as the concatenated code to decode the information.

Table 5.2. Timing comparison of RS and concatenated code under AWGN.

Code	SNR	Coding Time (s)	Decoding Time (s)
concm=5, t=7, N=18	5 dB	14	42
RS m=5, t=11	5 dB	17	52
concm=5, t=7, N=18	0 dB	14	61
RS m=5, t=11	0 dB	17	120

5.2.8 Summary

The experimentations allowed to define guidelines for the choice of the best code parameters:

- The code performs best around 2dB;
- The RS code can be chosen according to the targeted BER: the bigger the block length the better the performance;
- The CSOC has a crossover of performance around 2dB after which the higher the constraint length the best the performance. However, the performance gain diminishes with the longer constraint length;
- The best interleaver is the symbol block interleaver which outperforms the bit block interleaver and the random interleaver;
- There exists a ceiling size for the interleaver after which no significant additional gain is obtained; and
- The concatenated code outperforms significantly RS code alone at equivalent rate in both noise protection and speed.

With this code parameters, performance of 10^{-7} can be achieved at 2dB with simple decoder components.

5.3 Decoding Procedure Testing

This section goes through each step of the decoding process and selects the best method to use at each step based on tests performed on live pattern data. The decoding steps tested are: scanning resolution, line detection algorithm in conjunction with edge operators, and demodulation algorithm. The actual decoding of the code is tested in a separate section under noisy conditions. A series of example patterns is shown in Fig. 5.22..

5.3.1 Scanning Resolution of Rotated and Unrotated Pattern

Figure 5.23 shows the scanning in B&W and 8 bit grayscale of the pattern using resolutions equal, double and quadruple of the initial printed resolution. The experiment is performed for a printed pattern at 37.5 ppi (pixel per inch). The first conclusion is that the original printing resolution does not matter as long as the required printing quality can be achieved. The real factor is the ratio between the printing and the scanning resolutions. Second it can be noted that for equivalent results, the B&W image has to have at least twice the resolution of the grayscale image. A slight misalignment in the scanned image

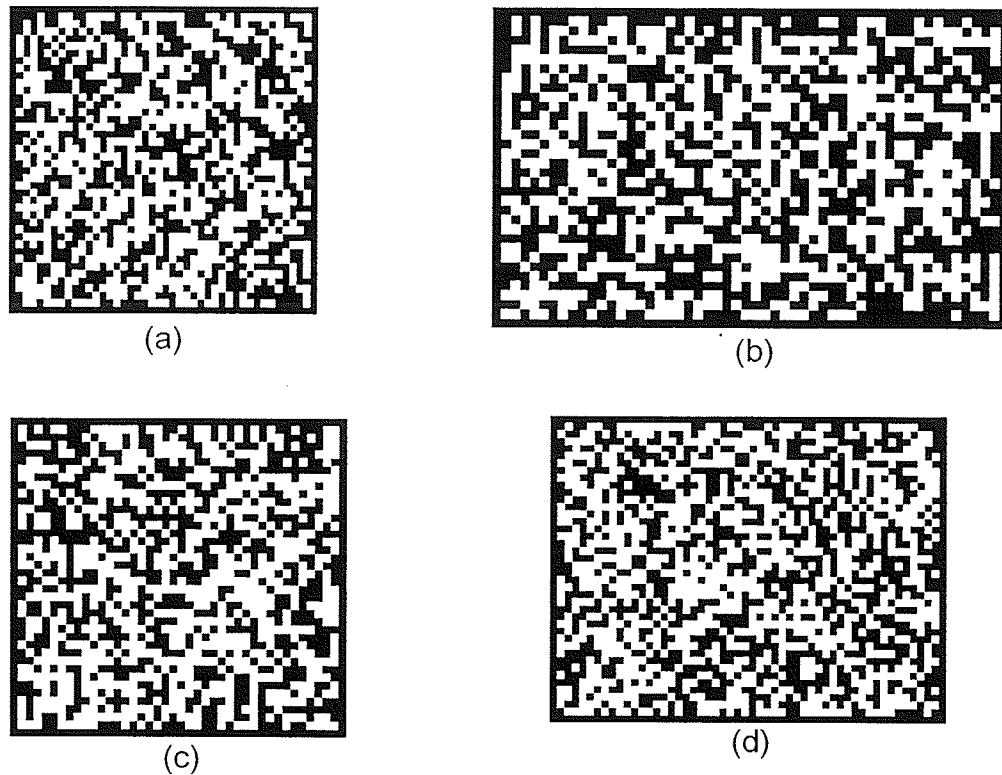


Fig. 5.22. Example of patterns: (a) RS(3,1) Conv(18); (b) RS(3,1) Conv(7); (c) RS(6,5) Conv(2); and (d) RS(4,3) Conv(18).

implies the complete loss of bits in the binary image whereas the grayscale image gives lighter or darker pixels but the original pattern pixel is still very distinguishable. The equal ratio between printing and scanning is too prone to misalignment errors. However, the double ratio is already a workable sample which agrees with [Palm95]: “As expounded by Nyquist, the sampling theorem suggests that it might be possible to build equipment that can successfully decode linear modulated symbols with a Beta (ratio between printed and scanned resolution) approaching 1.0. In the real world scanning situations, this is impractical, and most equipment employs a Beta of 2.0 or greater for width modulated symbologies. 2D Matrix symbologies use even larger values of Beta.”

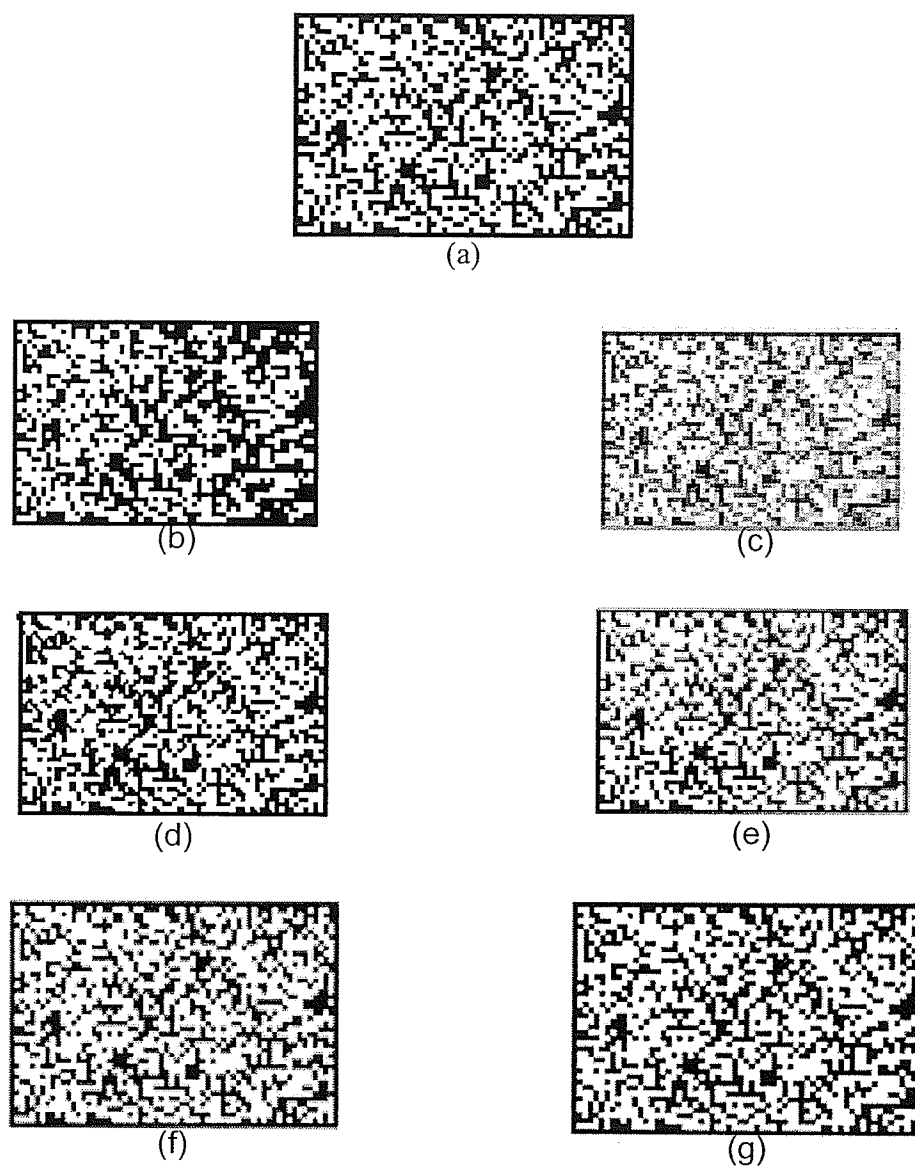


Fig. 5.23. Scanned patterns at different printing and scanning resolutions: (a) original pattern printed at 37 ppi; (b) scanned pattern in B&W at 37 dpi; (c) gray level at 37 dpi; (d) B&W at 75 dpi; (e) gray level at 75 dpi; (f) gray level at 100 dpi; and (g) gray level at 150dpi.

Figure 5.24 illustrates the rotation process applied to a pattern scanned rotated. The transformation has to be performed on a gray scale image for best results. Once more the Beta 2 seems to be a good rule of thumb.

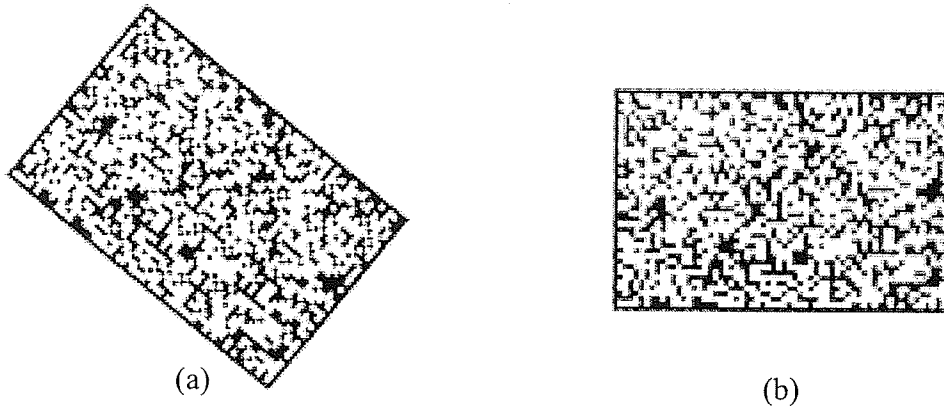
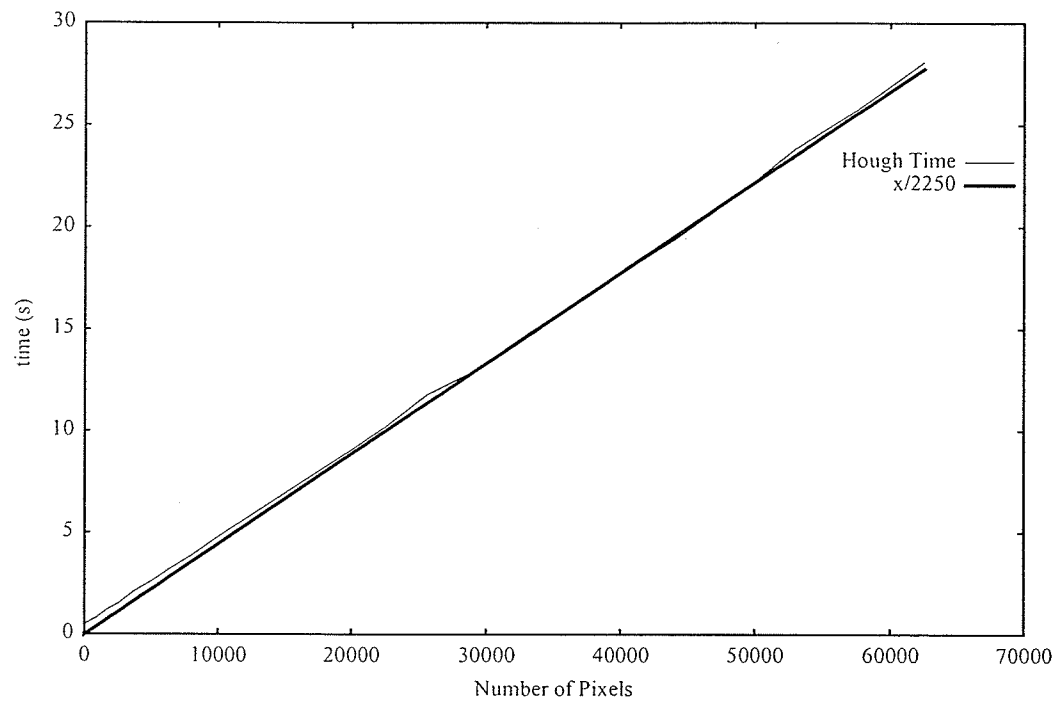


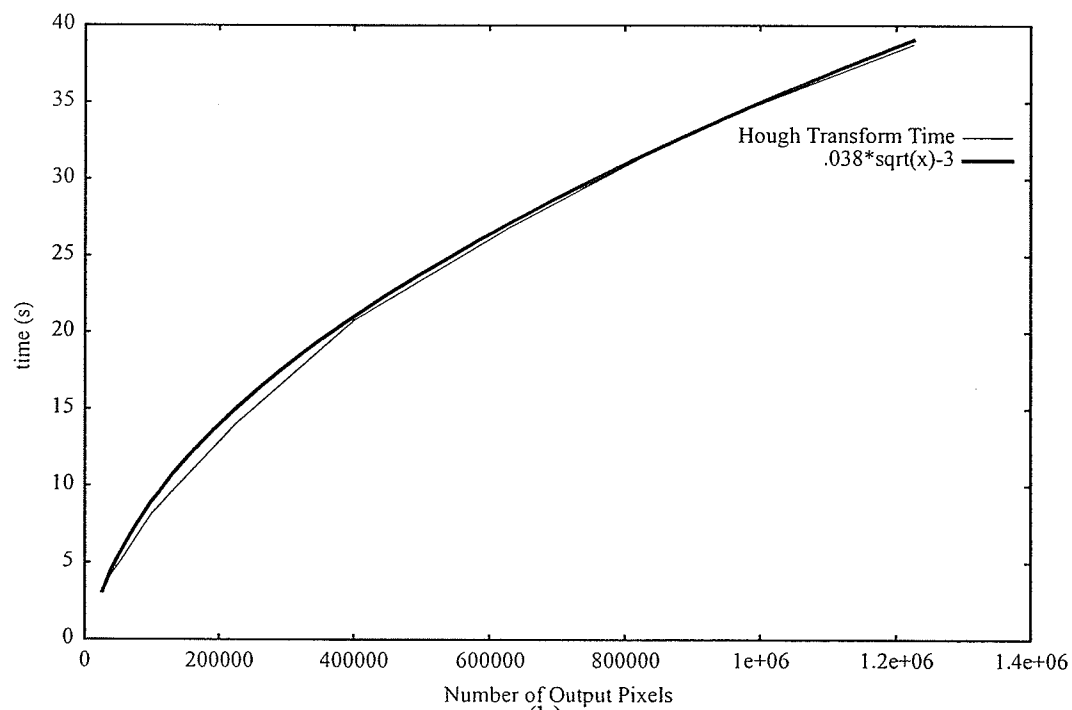
Fig. 5.24. (a) Scanned rotated image and (b) image after transform.

5.3.2 Line Detection Tests

The two line detection algorithms considered in this thesis are the Radon and the Hough transform. The Radon transform is more computationally expensive but gives clearer peaks for the identified lines. The Radon transform cost is based on the size of the original image and the size of the image to generate in the transform space. The Hough transform can be as computationally expensive as the Radon transform but its computational time depends not only on the same factors as the Radon transform but also on the number of non zero pixels in the original image. This is a very important fact because a lot of computation can be saved with a clever way of thresholding the original image.



(a)



(b)

Fig. 5.25. Hough transform time (a) against number of pixels in the input file and (b) number of pixels required in the output file.

The Hough transform is tested with an image containing a square of non zero pixels of increasing dimension. The time to execute the transform is recorded and is plotted against the number of pixels. The output of size of the transform is left relatively large so the time taken by the transform is high enough to be relevant. The result is shown in Fig. 5.25.. There is a linear relationship between the number of non zero pixels and the time taken by the Hough transform. The goal is then to reduce as much as possible the number of pixels in the input image while keeping the lines features. The Radon transform time is the same as the Hough transform time for an image where all pixels are non zero. The second graph illustrates the Hough time to transform an image with all non zero pixels of fixed size but the transform image is of varying size. The time decreases with the size of the output image on a square root profile. The rule is then to decrease the size of the output image while still keeping enough dots to identify the line peaks.

From the first experiment, the goal is to reduce the number of pixels in the input image while keeping the line features. Edge detection operators with thresholding can perform this function. The goal was originally to test several edge detection operators and compare them with one another. However, a Hough transform package developed by Pr. John Gauch at the University of Kansas implemented the idea that I wanted to develop in this thesis with a few interesting additions [KUIM99]. The package performs the following steps:

1. Apply the first order Sobel gradient operator in the x and y direction and calculate the norm of the calculated vector;

2. Threshold the resulting image (the threshold is a function parameter);
3. Apply the Hough transform to the thresholded Sobel image;
4. Use a 5x5 neighborhood operator to threshold the transformed image and threshold the image (the threshold is a function parameter);
5. Apply an averaging operator to the thresholded Sobel image to thicken the lines and fill in any gaps in the lines; and
6. For each pixel in the thresholded Hough transform image, follow the associated line in the blurred Sobel image and record a pixel in the final image when the line hits a series of n non zero pixels in a row (n is a function parameter).

In the Sobel image, edges are enhanced creating a number of non zero pixels around the edge. The transformed edge is not a single point in Hough transformed space, it is an oval. The 5x5 neighborhood operator helps to isolate a local maximum that is not just a dot but a group of pixels.

Usually, the inverse Hough transform results in non localized lines that go through one side of the image to the other. The localization of the lines on the image is performed by blurring the Sobel image and by following the lines in the blurred image which parameters are left in the thresholded Hough transform image. If a segment of non zero pixel with a minimum length is found along the line then the segment is drawn in the final image. The images for each step of the process are shown in Fig. 5.28..

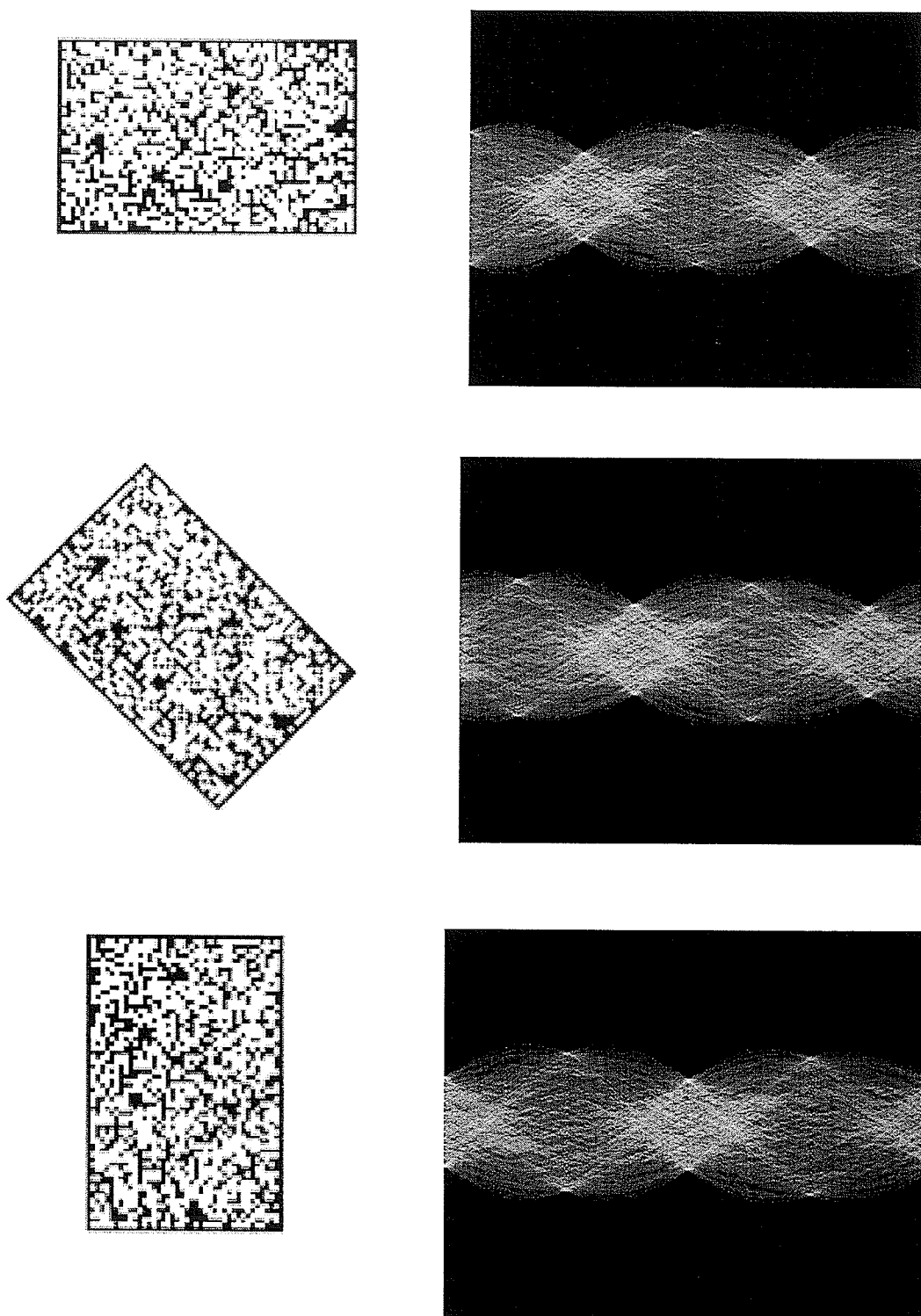
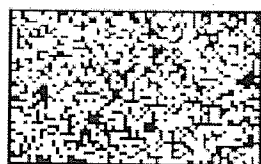
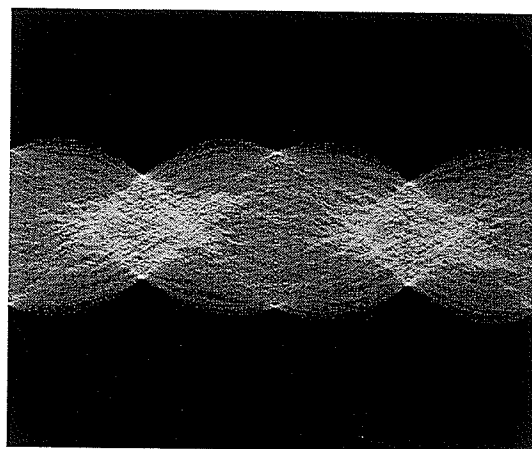


Fig. 5.26. Hough transform of rotated pattern.



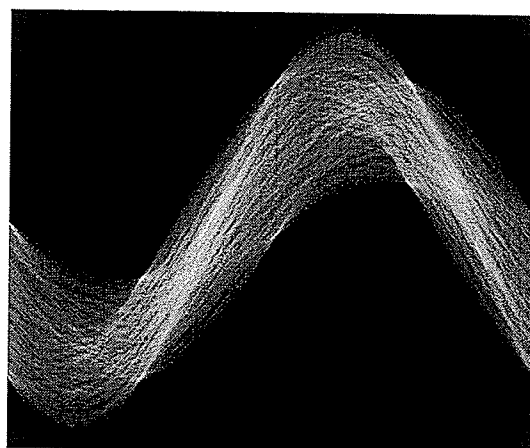
(a)



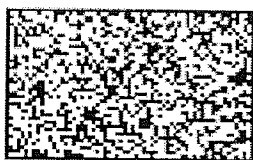
(b)



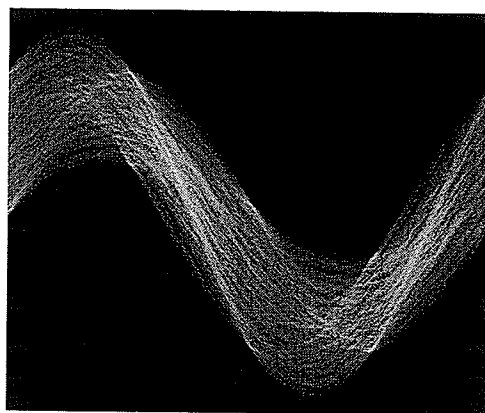
(c)



(d)



(e)



(f)

Fig. 5.27. Hough transform of translated pattern.

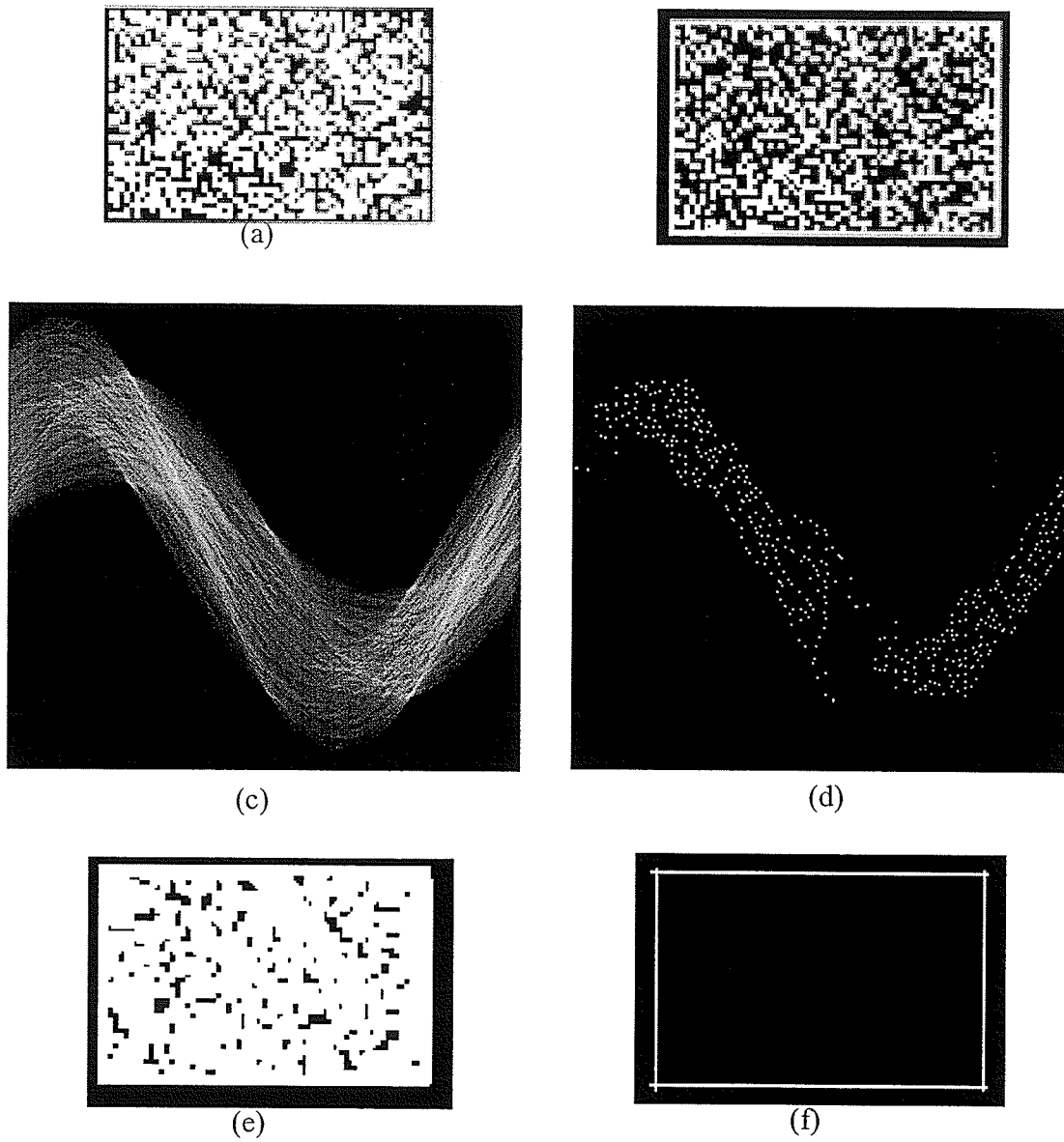


Fig. 5.28. KUIM line identification process: (a) original image, (b) after Sobel, (c) Hough transform, (d) Hough thresholded, (e) image from (b) after averaging, and (f) identified lines.

Fig. 5.26. and Fig. 5.27. correspond to Hough transform of the rotated and translated pattern respectively. The strong dots correspond to the pattern frame. The X axis is the angle parameter of the line and the Y axis is the rho parameter of the line. A

rotation of the pattern means a change in the theta argument (translation in Hough space). The frame dots are still grouped in two parallel pairs spaced 90 degrees in theta parameter. Similarly, the translation leaves the characteristics of the frame dots the same.

It can be noted that specialized DSP processor, FPGA, or ASIC would diminish the time of execution of the transforms specially if the transform is pipelined as described in various papers [BrYo92][CHSA90][GoDr95][KeMa93]. It is advantageous to take the Radon transform if the number of pixels in the final image cannot be reduced or if the extra processing power can be performed in minimal time.

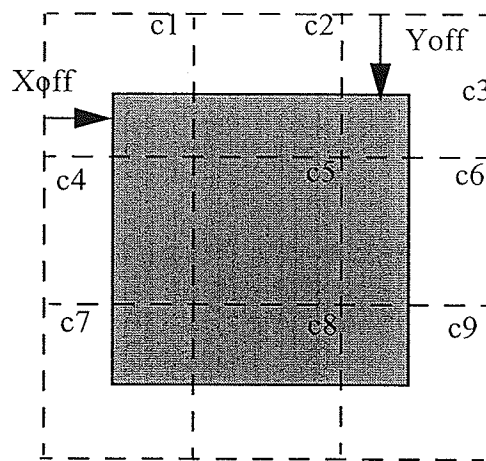


Fig. 5.29. Misaligned pixel and scanning grid when scanning at twice the printing resolution.

5.3.3 Demodulation Theory and Choice

A scanner is a series of photo sensors (usually CCDs) arranged in a square grid that measures the intensity of the light reflected by the scanned material. From the previous paragraph, the document is scanned at twice the resolution it is printed: there are

two scan dots for each printed pixel. The line detection procedure seen in the previous section allows the pattern to be straightened (no more rotation). However it does not avoid the pattern to be imperfectly aligned with the scanner grid. Figure 5.29 shows a scanned pixel (dark dot) and the scanning grid. Each scanning pixel has been numbered.

The following equations give the percentage of the total pixel intensity that is sensed by each scanner dot:

$$\begin{aligned}
 c1 &= (1 - Xoff)(1 - Yoff) \\
 c2 &= (1 - Yoff) \\
 c3 &= Xoff(1 - Yoff) \\
 c4 &= (1 - Xoff) \\
 c5 &= 1 \\
 c6 &= Xoff \\
 c7 &= (1 - Xoff)Yoff \\
 c8 &= Yoff \\
 c9 &= XoffYoff
 \end{aligned} \tag{5.13}$$

Fig. 5.30. illustrates the real scanned pattern and the simulation of the scanning physical process using Eq. (5.13). The simulated result is very close to the scanned result. The only main difference is due to imperfections in the paper, the printed pixels and the eventual non uniformity of the scanning cells and grid. However, the result is satisfying so the model can be used for a back processing algorithm during demodulation.

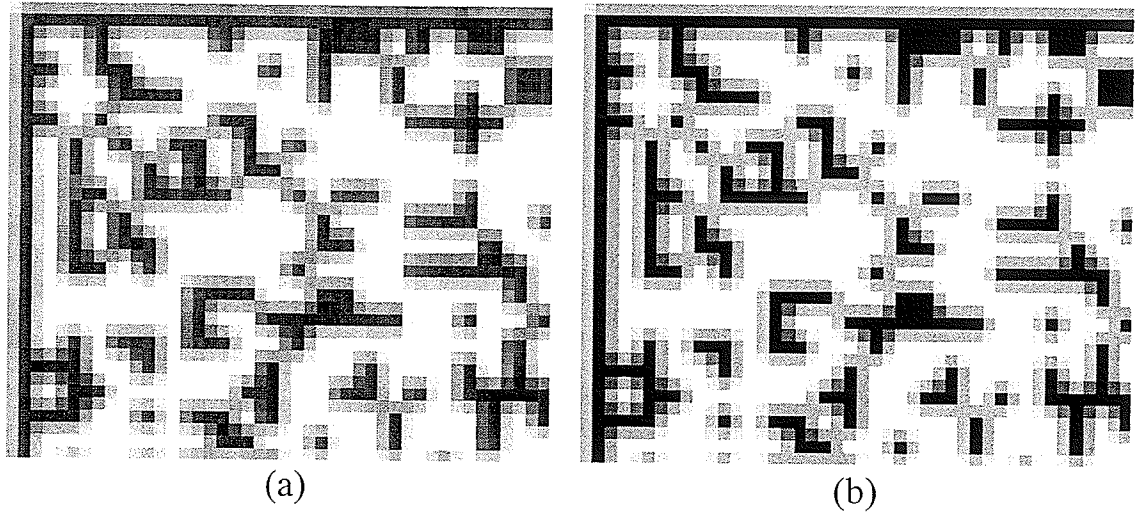


Fig. 5.30. (a) Scanned pattern and (b) simulated scanning pattern.

A1. The algorithm can scan the first contour of the pattern and as it is a complete black pixel contour and the outside of the contour are all white pixels, the X offset and Y offset can be determined by the intensity of the pixels on the outside of the contour. Once the X and Y offsets have been determined, the pattern is demodulated from top down and left to right. The intensity of the top pixel in c7 and c9 is subtracted to the intensity of dot c1 and c2 of the pixel being demodulated; the intensity of dot c3 and c6 of the left pixel are subtracted to the intensity of dot c1 and c4 of the pixel being demodulated; and the intensity of dot c9 of the top left pixel is subtracted to the intensity of dot c1 of the pixel being demodulated. The remaining intensities of dot c1, c2, c4, and c5 of the pixel being demodulated are respectively divided by the coefficients associated to each pixel and a majority vote is performed between

the four. If a tie occurs, a majority vote of c2, c4 and c5 is performed. This algorithm requires the exact identification of which line in the decoded pattern is line c5 otherwise, the wrong coefficients are applied to the wrong scanned dots and this can lead to erroneous demodulation. This algorithm when applied properly is the best demodulation algorithm as it is an exact inverse transform of the scanning process.

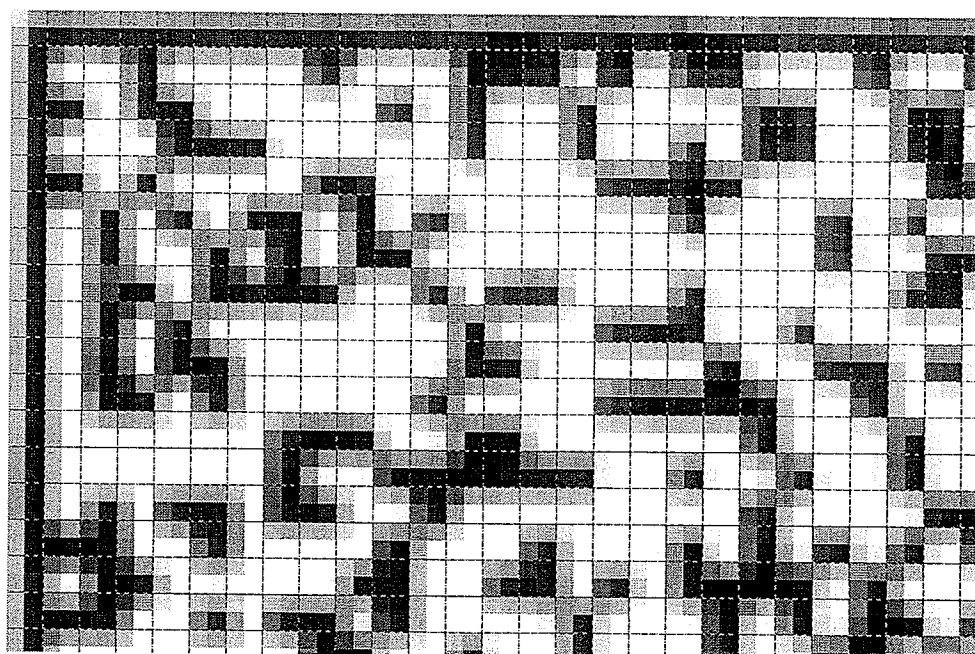


Fig. 5.31. Demodulation grid over the scanned pattern.

A2. Another simplified algorithm that does not depend on the proper identification of c5 is to take four decoded dots arranged in a square, take the average of the four intensities and threshold it with a mid point (Fig. 5.31. shows the demodulation grid over the scanned pattern zoomed on the top left corner).

The pixel c5 is always part of the four dots selected and drives the average of the four pixel under or over the threshold as it always has the intensity of the original pixel.

A3. An even simpler algorithm is to take four dots in a square and identify which of the four is the closest to a 0 intensity or a 1 intensity (this process seeks c5) and decode the pixel to the intensity of that pixel.

The second algorithm (A2) is the one implemented in this thesis.

5.4 Error Altered Pattern Testing

5.4.1 Gaussian Noise Testing

The noise is generated over the scanned image. This kind of noise simulates bad printing quality and poor paper quality (the printed dots in the pattern smear creating a random looking noise in the scanned pattern) as well as the crumbling of the paper throughout its life. Fig. 5.32. illustrates the KUIM Hough transform performance under noise. The Hough transform performs very well still clearly highlighting the lines around the pattern. The algorithm fails to recognize the lines at around 1dB. The performance of the pattern is linked to the analysis of the codes carried earlier in this chapter and will not be performed again. However, the previous decoding step techniques are correctly chosen as they still perform their task under white Gaussian noise.

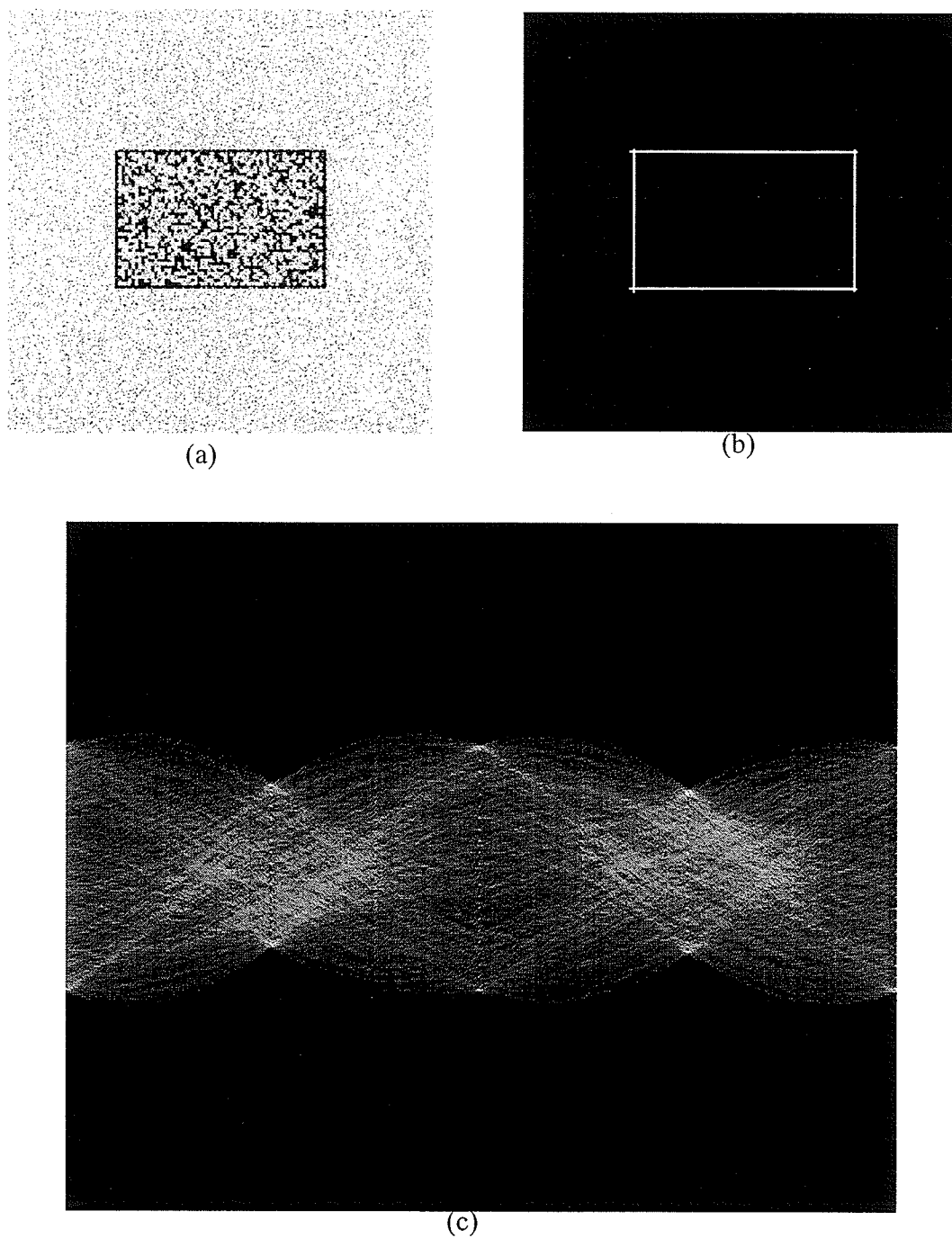


Fig. 5.32. (a) original image, (b) identified lines, and (c) associated Hough transform.

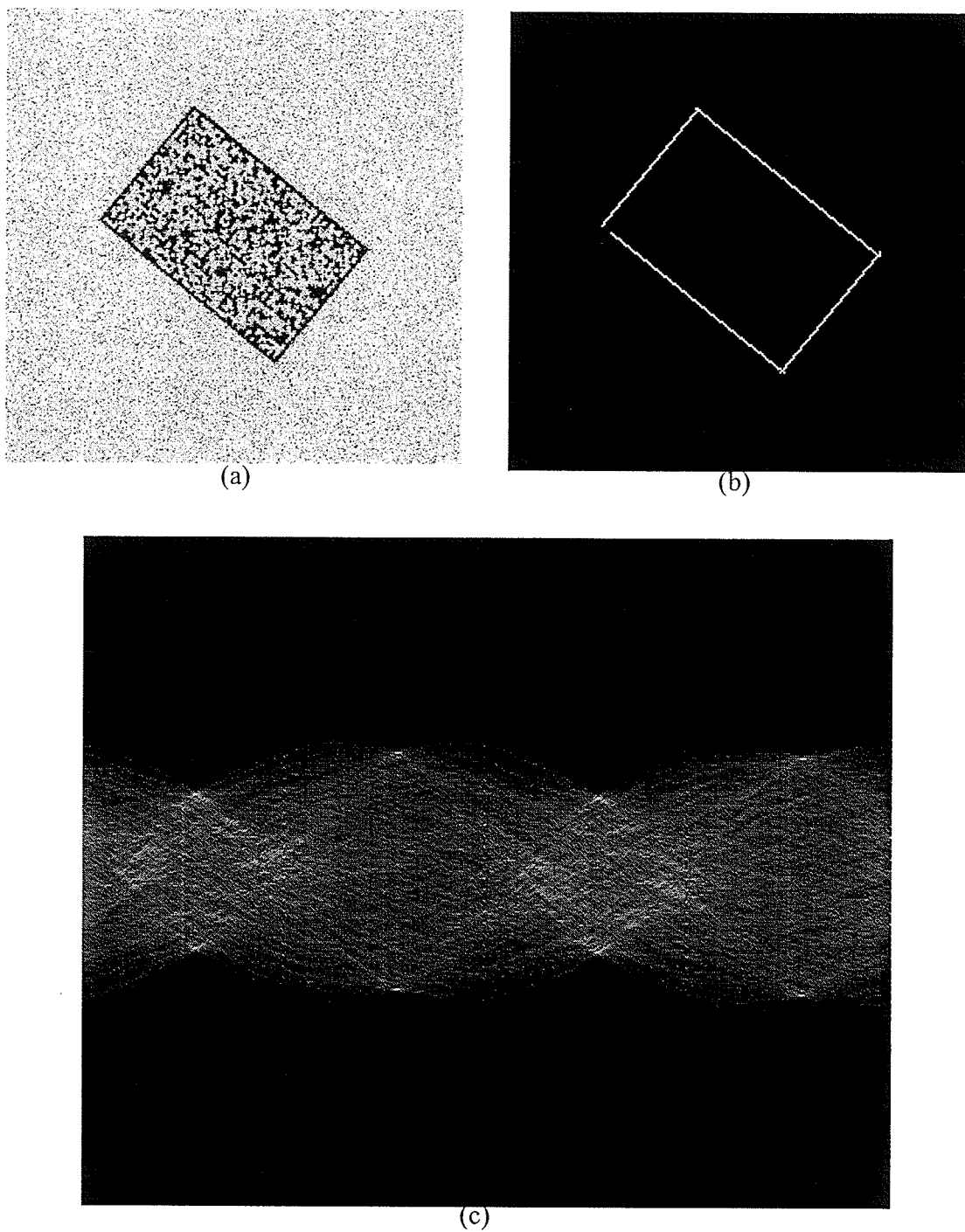


Fig. 5.32. (a) original image, (b) identified lines, and (c) associated Hough transform.

5.4.2 Burst Noise

The pattern is tested under burst noise, The pattern is erased and scanned and demodulated. Examples of the scanned patterns after demodulation are shown in Fig. 5.33. (note all those patterns decode properly).

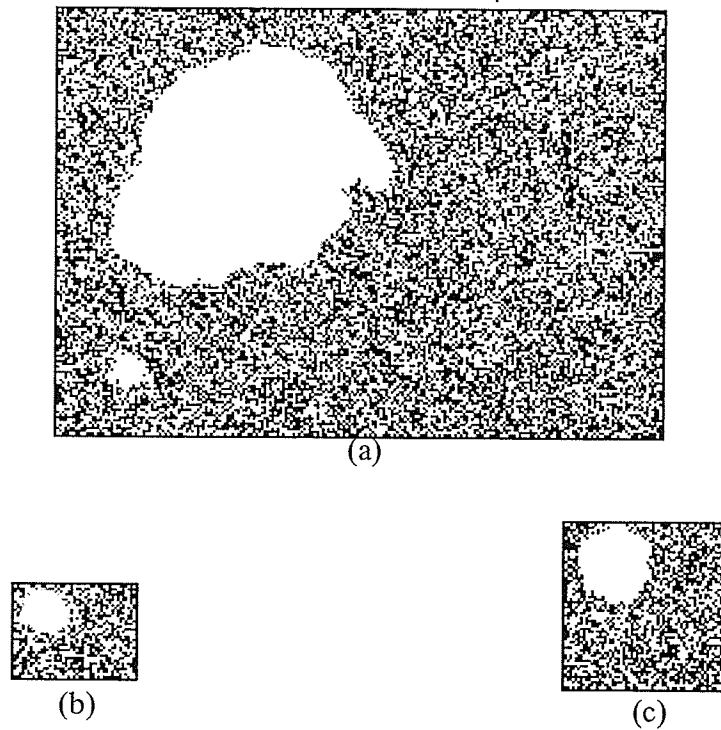


Fig. 5.33. Demodulated patterns after burst noise is applied (a) $m=5$, $t=15$, (b) $m=6$, $t=7$, and $m=6$, $t=20$.

The resulting decoding process is visually illustrated in Fig. 5.35.

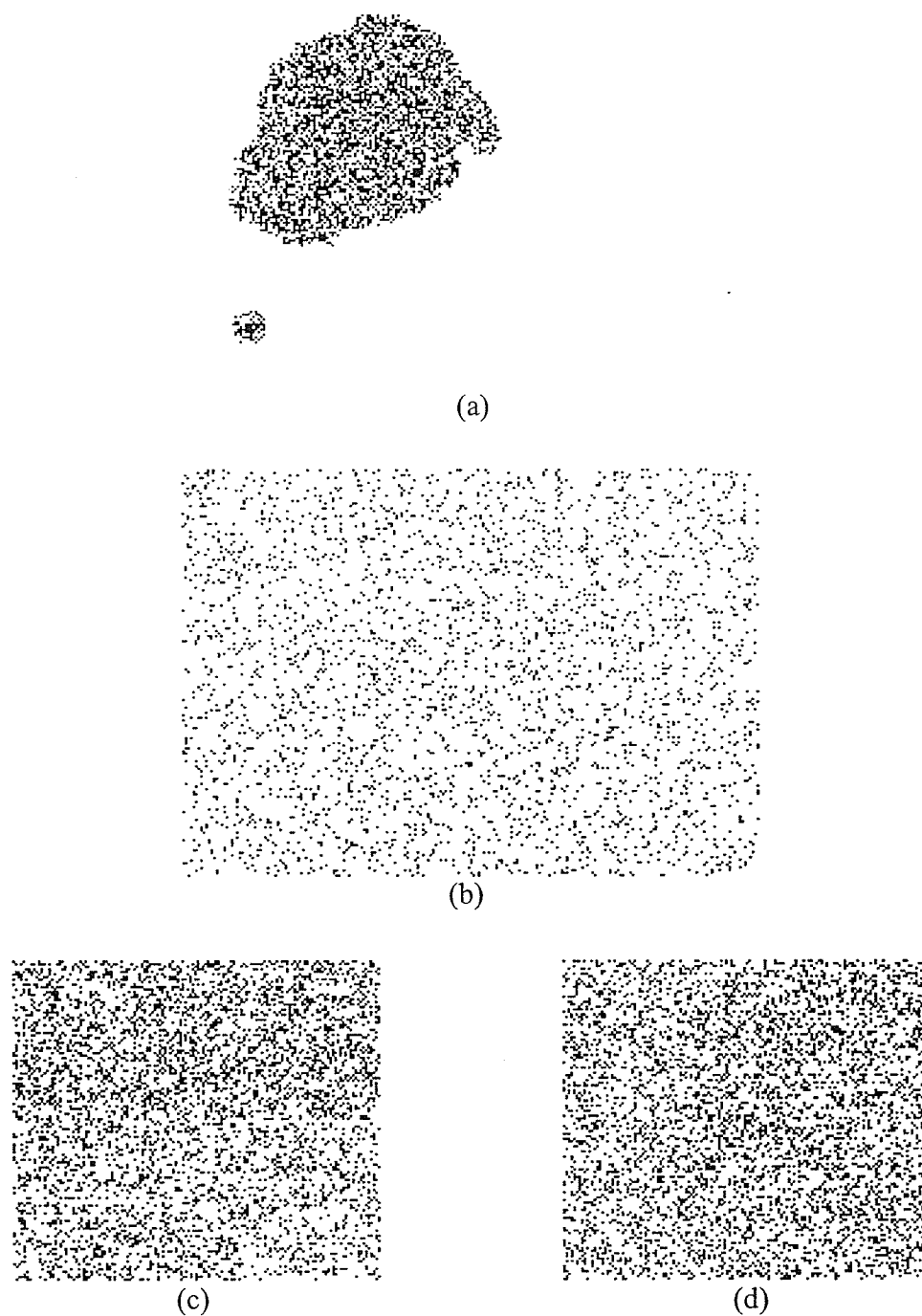


Fig. 5.34. Difference between the decoded patterns with and without noise, (a) for the pattern, (b) after random deinterleaver, (c) after the convolutional decoder, and (d) after the symbol interleaver.

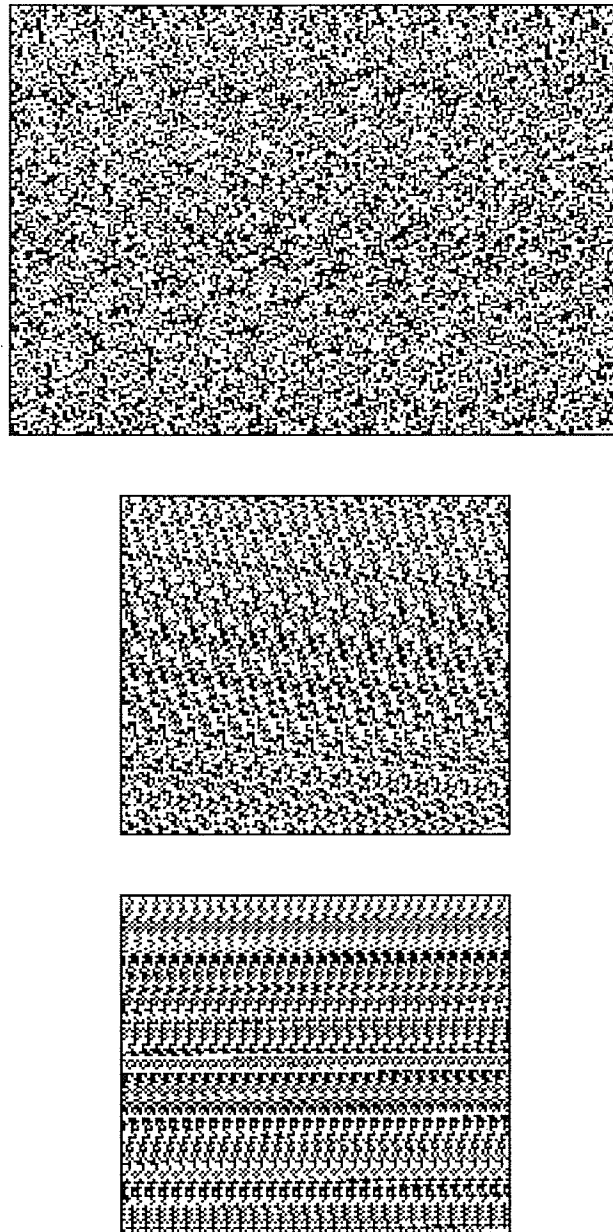


Fig. 5.35. Decoding process of $m=5, t=15$ (see previous figure), (a) after random deinterleaver, (b) after the convolutional decoder, and (c) after the symbol interleaver.

The result after the RS decoder is not shown in this figure as it is not a pattern anymore, it is a text file. The next figure (Fig. 5.34.) shows the subtraction between the decoded patterns without noise and the decoded patterns with noise.

The random interleaver at the beginning of the decoding process does randomize the burst noise as said in the pattern design section. However, after testing all patterns under burst noise, the maximum size of the burst is around 15% of the pattern size. The problem comes from the convolutional code that does not bring any decoding gain, it actually gives a negative gain to the decoding process: the pattern after random interleaving has 3,127 bits in error, after the convolutional decoder, it has 4,470! In this case, the random interleaver and the convolutional code interfere with the good burst decoding properties of the RS code. This remark started a series of testing with the RS code alone as well as the RS with symbol interleaving.

Figure 5.36 shows examples of patterns that decode properly. Similarly to Section 5.2.6, RS code and concatenated codes of identical rates are compared for performance under burst noise. The concatenated code with $m=5$, $t=7$ has a similar rate to the RS code alone with $m=5$, $t=11$ as well as concatenated $m=6$, $t=7$ and RS $m=6$, $t=19$, and finally concatenated $m=6$, $t=20$ and RS $m=6$, $t=26$. For all RS codes a symbol interleaver is used and for RS $m=6$, $t=27$ a pattern without symbol interleaver is also tested. The results of the biggest scratch on each pattern is illustrated in Fig. 5.34..

Table 5.3 Worst case scratch handled by equivalent rate concatenated and RS codes.

Code Type	Max Scratch Size
conc $m=5$, $t=7$, $N=18$	15%
RS $m=5$, $t=11$	27%
conc $m=6$, $t=7$, $N=18$	16%
RS $m=6$, $t=19$	21%
conc $m=6$, $t=20$, $N=18$	15%
RS $m=6$, $t=26$	34%
RS $m=6$, $t=26$ no interleaver	27%

Fig. 5.36. RS pattern with symbol interleaver (a) $m=6$, $t=19$, (b) $m=5$, $t=11$, (c) $m=6$, $t=26$, and without interleaver (d) $m=6$, $t=26$.

From the experiment, in the worst case, the pattern decodes properly until more than 21% of the pattern is scratched and the maximum of the tested patterns is 34% of the pattern. The symbol interleaver helps as the pattern without the interleaver decodes up to 27% scratch when the one with interleaver decodes up to 34% scratch. Furthermore, the shape of the scratch is very constrained as the RS lines are stacked up and the scratch

cannot go any further horizontally without giving a decoding error. The performance of the RS code alone is much better than the performance of the concatenated code (up to a factor of more than 2 for the small set of patterns tested) and the performance of the RS code with an interleaver is better than without an interleaver as it allows for bigger and more isotropic scratches.

5.5 Summary

This chapter guided the selection of the elements used in the pattern design:

1. Selection of the random number generator for noise generation and for random interleaving (rand48);
2. Choice of the optimal code parameters for optimal performance of the concatenated code under AWGN;
3. Testing of the decoding process and selection of the optimal parameters for the decoder; and
4. Testing of the pattern under burst noise and selection of the best code for this kind of noise.

Some of the main results are:

1. Bigger is better for the RS code symbol size, the convolutional code constraint length, and the interleaver size;

2. symbol interleaver with symbols of the size of the RS symbol size perform best;
3. the concatenated code performs much better than RS alone under AWGN; and
4. RS with interleaver performs much better than the concatenated code under burst noise.

The choice between RS with interleaver and the concatenated code depends a lot on the noise type expected during the document's life. For example, on a scratch ticket where the pattern would be printed under the scratch material, it is recommended to use the RS with interleaver code. But in a document that is printed on poor paper with poor ink and that has less chances of being scratched or will suffer smaller scratches the designed concatenated code is recommended.

CHAPTER VI

CONCLUSIONS AND RECOMMENDATIONS

6.1 Conclusions

The validation system developed in this thesis is complementary to the classic forgery prevention techniques as it prevents the creation of new documents and the alteration of the current documents (integrity) but it does not protect against exact duplication of a document. The duplication protection has to be provided by another standard technique (see Section 1.2 for a description of standard printed document security techniques.)

Three levels of security system have been proposed according to the life span and the value of the document to be protected. Increased security comes at the cost of increased bandwidth requirements between the key server and the validation stations.

A line locator algorithm using the gradient edge enhancement operator and the Hough image transform allow for automatic location of the validation pattern. The technique can locate the pattern with SNR as low as 2 dB.

The code used to protect the encrypted signature can perform a BER of $10^{-6.5}$ at an SNR of 2 dB. The best interleaver is the symbol interleaver, and the bigger code components perform the best. However, there is a ceiling with little return for the interleaver size and the convolutional code constraint length. The RS performance

increases linearly with block size. The concatenated code performs up to 2 dB better than the RS code alone at similar code rate for a BER of $10^{-6.5}$. Comparing the same codes for decoding speed, the concatenated decoding can be up to twice as fast than the RS decoding at 0 dB but only 20% faster at 5 dB.

However, the RS decodes with no error with burst error of up to 27% whereas the concatenated code cannot go beyond 15% of erasure due to the poor performance of the outer convolutional code which amplifies the burst noise. Adding a symbol interleaver before the RS code, the performance increased to 34% and the noise applied can be more isotropic. The symbol interleaver breaks the noise into several RS frames without spreading noise between symbols.

The code must be chosen according to the expected noise characteristics: the concatenated code performs best under random noise and the interleaved RS is better at burst noise. However, they both have acceptable performance in the other noise type.

6.2 Contributions

This thesis made the following contributions:

- Design of a entire validation system from the creation of the pattern and its validation, through the management of keys on the network;
- Analysis of the concatenated code with the selection of the optimal parameters for the specific concatenated code;

- Design of a multi level security system based on the value of the document to be validated;
- Modeling of the scanning transformation; and
- Analysis of two random number generators and the selection of the best of the two.

6.3 Recommendations for Future Work

The concatenated code performance was only compared to the interleaved RS. Other codes can be implemented in the pattern to compare performance under Gaussian and burst noise. The performance of concatenated convolutional interleaved with RS code using Turbo decoding should be investigated.

The Radon and Hough transforms have a high computational cost. As mentioned in Section 5.3.2, the transforms can be parallelized and efficiently implemented on DSPs. A hardware implementation should be performed as the current times for the transforms using software only on a generic purpose processor are too high for our automatic validation.

To reduce computation of the Hough transform, the pattern can be processed with low pixelation first, cropped and processed at higher resolution. The large pixelation is a low pass filter which increases the resilience of the location algorithm against noise.

The pattern printing and scanning were ideal (laser printing and flatbed scanning). The pattern should be printed using a low resolution (75-100 dpi) inkjet printer and a handheld scanner (using CCD technology) should be used to acquire the pattern. Those condition would be much closer to real life application.

REFERENCES

- [ABAs00] American Bankers Association, "Deposit account fraud survey report," 2000, 80 pp.
- [ACAI00] American Collectors Association, Inc., "Credit and collections fact sheets: Statistics on checks," Oct. 2000. From <http://www.collector.com/content/press/factsheets/check.html> (available as of June 2001).
- [Benb99] A. Y. Benbasat, "A survey of current optical security techniques," MIT Media Lab. Boston, MA, Apr. 1999. From <http://www.media.mit.edu/~ayb/6637report.pdf> (available as of June 2001).
- [Blah83] R. E. Blahut, *Theory and Practice of Error Control Codes*. Reading, MA: Addison-Wesley, 1983, 500 pp.
- [BrYo92] M. Brady and W. Yong, "Fast parallel discrete approximation algorithms for the Radon transform," *Proc. 4th Annual ACM Symp. on Parallel Algorithms and Architectures*, pp. 91-99, 1992.
- [CaCl81] J. B. Cain and G. C. Clark, Jr., *Error-Correction Coding for Digital Communications*. New York, NY: Plenum Press, 1981, 422 pp.
- [Cann86] J. Canny, "A computational approach to edge detection," *IEEE Trans. Pattern Analysis and Machine Intelligence*, vol. 8, no. 6, pp. 679-698, Nov. 1986.

-
- [CHSA90] W. Current, P. Hurst, E. Shieh, and I. Agi, "An evaluation of Radon transform computations using DSP chips," *Machine Vision and Applications*, vol. 3, pp. 63-74, 1990.
- [Deri87] R. Deriche, "Using Canny's criteria to derive a recursively implemented optimal edge detector," *Int'l J. Computer Vision*, vol. 1, no. 2, pp. 167-187, 1987.
- [Deri98] R. Deriche, "Techniques d'Extraction de Contours," INRIA Sophia-Antipolis, Apr. 1998. From ftp://ftprobotvis.inria.fr/pub/html/Cours/techniques_contours.ps.gz (in French) (available as of June 2001).
- [Enta98] K. Entacher, "A collection of classical pseudorandom number generators with linear structures - Advanced version," June 2000, Ch. C. From <http://crypto.mat.sbg.ac.at/results/karl/server/server.html> (available as of June 2001).
- [FaDe99] J. L. Devore and N. R. Farnum, *Applied Statistics for Engineers and Scientists*. Pacific Grove, CA: Duxbury Press, 1999, 577 pp.
- [GoDr95] W. A. Gotz and H. J. Druckmuller, "A fast digital Radon transform - An efficient means for evaluating the Hough transform," *Pattern Recognition*, vol. 28, pp. 1985-1992, 1995.
- [GoWo92] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*. Reading, MA: Addison-Wesley, 1992, 716 pp.
- [Hayk94] S. Haykin, *Communication Systems* (3rd ed.). New York, NY: Wiley, 1994, 422 pp.
-

-
- [HiPi85] I. Hill and M. Pike, "Remark on Algorithm 299," *ACM Transaction on Mathematical Software*, vol. 11, no. 2, p. 185, 1985.
- [Imag00] "ImageMagick - Convert, edit, and compose images." From <http://www.imagemagick.org> (available as of June 2001).
- [KeMa93] B. T. Kelley and V. K. Madisetti, "The fast discrete Radon transform - Part I: Theory," *IEEE Transactions on Image Processing*, vol. 2, no. 4, pp. 382-400, 1993.
- [Kins95] W. Kinsner, *Fractal and Chaos Engineering*, 24.721 Course Notes, University of Manitoba, 1995.
- [Knut81] Donald E. Knuth, *The Art of Computer Programming*, vol. 2, *Seminumerical Algorithms* (2nd ed.). Reading, MA: Addison-Wesley, 1973.
- [KUIM99] J. Gauch, "Kansas University Image Processing System (KUIM)," Kansas University, 1999. From <http://www.ittc.ukans.edu/~jgauch/research/kuim/html/> (available as of June 2001).
- [LaLi96] J. Lakhal and L. Litzer, "A parallelization of the Deriche filter: A theoretical study and an implementation on the MasPar system," *Proc. Int'l Conference on Parallel Computing and Distributed Processing Techniques (PDPTA'96)*, p. 867, 1996.
- [Leon95] C. W. Leong, *A Concatenated Interleaved Reed-Solomon and Convolutional Self-Orthogonal Coding Scheme: A Forward Error Correction System*. B.Sc. Thesis, University of Manitoba, 1995.
-

-
- [LiWe67] S. Lin and E. Weldon, "Long BCH codes are bad," *Information Control*, vol. 11, pp. 445-451, 1967.
- [Mass63] J. L. Massey, *Threshold Decoding*. Cambridge, MA: MIT Press, 1963, 129 pp.
- [Mand82] B. B. Mandelbrot, *The Fractal Geometry of Nature*. San Francisco, CA: W. H. Freeman, 1982, 465 pp.
- [MaHi80] D. Marr and E. Hildreth, "Theory of edge detection," *Proceedings of Royal Society of London*, vol. 207, pp. 187-217, 1980.
- [MiLe85] A. M. Michelson and A. H. Levesque, *Error-Control Techniques for Digital Communication*. New York, NY: Wiley, 1985, 463 pp.
- [MOVa96] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*. CRC Press, 1996, 816 pp. From <http://www.cacr.math.uwaterloo.ca/hac/> (available as of June 2001)
- [Palm95] R. C. Palmer, *The Bar Code Book, Reading, Printing, Specification, and Application of Bar Code and Other Machine Readable Symbols* (3rd ed.). Peterborough, NH: Helmers Publishing, 1995, 836 pp.
- [PeJS92] H.-O. Peitgen, H. Jurgens, and D. Saupe, *Chaos and Fractals: New Frontiers of Science*. New York, NY: Springer-Verlag, 1992, 984 pp.
- [PTVF92] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: the Art of Scientific Computing*. New York, NY: Cambridge University Press, 1992, 735 pp.
-

-
- [Rene88] R. L. van Renesse, "Paper based document security - A review," *Proc. Int'l Carnahan Conference on Security Technology*, pp. 75-80, 1988.
- [Rene98] R. L. van Renesse (ed.), *Optical Document Security* (2nd ed.). Norwood, MA: Artech House, Jan. 1998, 505 pp.
- [Schn96] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C* (2nd ed.). New York, NY: Wiley, Oct. 1995, 784 pp.
- [Shan48] C. E. Shannon, "A mathematical theory of communications," *Bell Syst. Tech. J.*, vol. 27, pp. 379-423, 623-656, 1948.
- [SLin70] S. Lin, *An Introduction to Error-Correcting Codes*. Englewood Cliffs, NJ: Prentice-Hall, 1971, 330 pp.
- [Toft96] P. Toft, *The Radon Transform, Theory and Implementation*. Ph. D. Dissertation, Technical University of Denmark, 1996. From http://eivind.imm.dtu.dk/staff/ptoft/ptoft_papers.html (available as of June 2001)
-

APPENDIX A

SOURCE CODE

A.1 Debugger

```
// debugger.cc
#include "debugger.h"

int binaire(int nbr, ostream& output, int size)
{
    for (unsigned int i=(1<<(size-1)); i>0; i>>=1)
        output << ((nbr&i)!=0);
}
```

A.2 Buffer

```
// buffer.h
#ifndef _buffer_h
#define _buffer_h

#include <iostream.h>

class buffer
{
protected:
    char          *bufferPtr;
    char          *writePtr;
    char          *readPtr;
    char          *wrapPtr;
    int           inMin, outMin;
    char          full;
    char          firstFill;
    char          eof;
    long          bytesRead;
    unsigned char lastNbrBits;

public:
    buffer(int size, int inMinimum=-1, int outMinimum=-1);
    ~buffer();
}
```

```

    friend buffer& operator>>(buffer& is, buffer& buff);
    friend istream& operator>>(istream& is, buffer& buff);
    friend ostream& operator<<(ostream& os, buffer& buff);
    int read(char* outString,int minRead,int maxRead);
    int testRead();
    int testWrite();
    int get(unsigned char& info);
    int put(unsigned char info);
    int last(unsigned char info, unsigned char nbrBits);
    int reSync();
    void setEOF();
    char atEOF();
    long nbrRead();

    void newData();

    protected:
    int numberRead();
    int numberWrite();
};

#endif

// buffer.cc
#include "buffer.h"

buffer::buffer(int size,int inMinimum,int outMinimum): inMin(inMinimum),
outMin(outMinimum), full(0), firstFill(1),
                bufferPtr(0), writePtr(0), readPtr(0), wrapPtr(0),
eof(0),
                bytesRead(0), lastNbrBits(8)
{
    if (size<=0)
    {
        cerr << "Buffer size must positive." << endl;
        exit(0);
    }

    bufferPtr=new char[size];
    writePtr=readPtr=bufferPtr;
    wrapPtr=bufferPtr+size;
}

buffer::~~buffer()
{
    if (bufferPtr)
        delete bufferPtr;
}

```

```

istream& operator>>(istream& is, buffer& buff)
{
    int bytesToWrite=buff.numberWrite();
    int totalWrite=buff.testWrite();

    buff.bytesRead=0;

    if (buff.inMin>totalWrite)//always false if the "in" minimum is
negative
        return is;//(thus no minimum input)

    if (bytesToWrite==0)
        return is;

    is.read(buff.writePtr,bytesToWrite);
    buff.bytesRead=is.gcount();
    if (buff.bytesRead<bytesToWrite)
        cout << "To read= " << bytesToWrite << " read= " <<
buff.bytesRead << endl;
    buff.writePtr+=buff.bytesRead;

    if (buff.writePtr>=buff.wrapPtr)
    {
        if (buff.firstFill)
            buff.firstFill=0;
        buff.writePtr=buff.bufferPtr;
        buff.full=1;

        if ((bytesToWrite=totalWrite-bytesToWrite)>0)
        {
            is.read(buff.writePtr,bytesToWrite);
            buff.bytesRead=is.gcount();
            buff.writePtr+=buff.bytesRead;

            if (buff.writePtr>=buff.wrapPtr)
            {
                if (buff.firstFill)
                    buff.firstFill=0;
                buff.writePtr=buff.bufferPtr;
                buff.full=1;
            }
        }
    }
    return is;
}

buffer& operator>>(buffer& is, buffer& buff)
{
    int bytesToWrite=buff.numberWrite();
    int totalWrite=buff.testWrite();

```

```

    buff.bytesRead=0;

    if (buff.inMin>totalWrite)//always false if the "in" minimum is
negative
        return buff;//(thus no minimum input)

    buff.bytesRead=is.read(buff.writePtr,buff.inMin,bytesToWrite);
    buff.writePtr+=buff.bytesRead;

    if (buff.writePtr>=buff.wrapPtr)
    {
        if (buff.firstFill)
            buff.firstFill=0;
        buff.writePtr=buff.bufferPtr;
        buff.full=1;

        if ((bytesToWrite=totalWrite-bytesToWrite)>0)
        {
buff.bytesRead=is.read(buff.writePtr,buff.inMin,bytesToWrite);
            buff.writePtr+=buff.bytesRead;

            if (buff.writePtr>=buff.wrapPtr)
            {
                if (buff.firstFill)
                    buff.firstFill=0;
                buff.writePtr=buff.bufferPtr;
                buff.full=1;
            }
        }
    }
    return buff;
}

ostream& operator<<(ostream& os, buffer& buff)
{
    int bytesToRead;
    int bytesWritten=0;

    while ((bytesToRead=buff.numberRead())>0)
    {
        if (buff.outMin>bytesToRead &&//always false if the "out"
minimum is negative
            !buff.eof)//(thus no minimum input)
            return os;

        os.write(buff.readPtr,bytesToRead);
        buff.readPtr+=bytesToRead;

        if (buff.readPtr>=buff.wrapPtr)
        {

```

```
        buff.readPtr=buff.bufferPtr;
        buff.full=0;
    }
}

return os;
}

int buffer::read(char* outputStrg,int minRead,int maxRead)
{
    int maxReadLocal=numberRead();
    int totalRead=testRead();
    int firstRead=(maxRead>maxReadLocal)?maxReadLocal:maxRead;
    int i=0;

    bytesRead=0;

    if (firstRead>=minRead && firstRead>0)
        for (i=0;i<firstRead;i++)
            *(outputStrg++)=*(readPtr++);

    if (readPtr>=wrapPtr)
    {
        readPtr=bufferPtr;
        full=0;
    }

    maxRead-=firstRead;
    totalRead-=firstRead;
    int secondRead=(maxRead>totalRead)?totalRead:maxRead;

    if (secondRead>=minRead && secondRead>0)
        for (i=0;i<secondRead;i++)
            *(outputStrg++)=*(readPtr++);

    if (readPtr>=wrapPtr)
    {
        readPtr=bufferPtr;
        full=0;
    }

    bytesRead=firstRead+secondRead;

    return firstRead+secondRead;
}

long buffer::nbrRead()
{
    return bytesRead;
}
```

```
void buffer::newData()
{
    full=0;
    firstFill=1;
    eof=0;
    writePtr=readPtr=bufferPtr;
}

int buffer::numberRead()
{
    if (readPtr>=wrapPtr)
    {
        readPtr=bufferPtr;
        full=0;
    }

    if (full)
        return (wrapPtr-readPtr);
    else
        return (writePtr-readPtr);
}

int buffer::numberWrite()
{
    if (eof)
        return 0;

    if (writePtr>=wrapPtr)
    {
        if (firstFill)
            firstFill=0;
        writePtr=bufferPtr;
        full=1;
    }

    if (full)
        return (readPtr-writePtr);
    else
        return (wrapPtr-writePtr);
}

int buffer::reSync()
{
    if ((writePtr>=wrapPtr) && (readPtr>=wrapPtr))
    {
        if (firstFill)
            firstFill=0;
        readPtr=bufferPtr;
        writePtr=bufferPtr;
        return 0;
    }
}
```

```
    if (readPtr>=wrapPtr)
    {
        readPtr=bufferPtr;
        full=0;
    }

    if (writePtr>=wrapPtr)
    {
        if (firstFill)
            firstFill=0;
        writePtr=bufferPtr;
        full=1;
    }
}

void buffer::setEOF()
{
    eof=1;
}

char buffer::atEOF()
{
    return (eof&&(testRead()==0));
}

int buffer::testRead()
{
    if (readPtr>=wrapPtr)
    {
        readPtr=bufferPtr;
        full=0;
    }

    if (full)
        return (wrapPtr-readPtr)+(writePtr-bufferPtr);
    else
        return (writePtr-readPtr);
}

int buffer::testWrite()
{
    if (eof)
        return 0;

    if (writePtr>=wrapPtr)
    {
        if (firstFill)
            firstFill=0;
        writePtr=bufferPtr;
        full=1;
    }
}
```

```
    }

    if (full)
        return (readPtr-writePtr);
    else
        return (wrapPtr-writePtr)+(readPtr-bufferPtr);
}

int buffer::get(unsigned char& info)
{
    if (numberRead())
    {
        info=*(readPtr++);
        if (readPtr>=wrapPtr)
        {
            readPtr=bufferPtr;
            full=0;
        }
        if (eof && readPtr==writePtr)
            return lastNbrBits;
        else
            return 8;
    }

    return 0;
}

int buffer::put(unsigned char info)
{
    if (numberWrite())
    {
        *(writePtr++)=info;

        if (writePtr>=wrapPtr)
        {
            if (firstFill)
                firstFill=0;
            writePtr=bufferPtr;
            full=1;
        }

        return 1;
    }

    return 0;
}

int buffer::last(unsigned char info,unsigned char nbrBits)
{
    if (numberWrite())
    {
```

```
        *(writePtr++)=info;
        eof=1;
        lastNbrBits=nbrBits;

        if (writePtr>=wrapPtr)
        {
            if (firstFill)
                firstFill=0;
            writePtr=bufferPtr;
            full=1;
        }

        return 1;
    }

    return 0;
}
```

A.3 coDec

```
// coDec.h
#ifndef _coDec_h
#define _coDec_h

#include "buffer.h"

class coDec
{
    protected:
        buffer* inBuffer;
        buffer* outBuffer;
        char eof;

    public:
        coDec();
        coDec(buffer& in, buffer& out);

        int setInBuffer(buffer& in);
        int setOutBuffer(buffer& out);
        int setIOBuffer(buffer& in,buffer& out);

        virtual int processData();
        virtual void newData();

        char atEOF() {return eof;};
};

#endif
```

```
// coDec.cc
#include "coDec.h"

coDec::coDec(): inBuffer(0), outBuffer(0), eof(0)
{
}

coDec::coDec(buffer& in, buffer& out): inBuffer(&in), outBuffer(&out),
eof(0)
{
}

int coDec::setInBuffer(buffer& in)
{
    if (outBuffer==&in)
    {
        cerr << "the input and the output buffers "
              << "cannot be the same" << endl;
        return 0;
    }

    inBuffer=&in;
}

int coDec::setOutBuffer(buffer& out)
{
    if (inBuffer==&out)
    {
        cerr << "the input and the output buffers "
              << "cannot be the same" << endl;
        return 0;
    }

    outBuffer=&out;
}

int coDec::setIOBuffer(buffer& in,buffer& out)
{
    if (&in==&out)
    {
        cerr << "the input and the output buffers "
              << "cannot be the same" << endl;
        return 0;
    }

    inBuffer=&in;
    outBuffer=&out;
}

int coDec::processData()
{
}
```

```
    if (!inBuffer || !outBuffer)
    {
        cerr    << "Please assign me with some input and output"
                << " buffers before asking me to process data" << endl;
        return -1;
    }

    if (inBuffer->atEOF())
    {
        outBuffer->setEOF();
        return 0;
    }

    unsigned char temp;
    while ((inBuffer->testRead()) &&
           (outBuffer->testWrite()))
    {
        inBuffer->get(temp);
        outBuffer->put(temp);
    }

    return 1;
}

void coDec::newData()
{
    if (inBuffer)
        inBuffer->newData();
    if (outBuffer)
        outBuffer->newData();
    eof=0;
}
```

A.4 RS

```
// RS.h
#ifndef _RS_h
#define _RS_h

#include "coDec.h"
#include <math.h>
#include <limits.h>

class RS : public coDec
{
    //friend int main();
    protected:
```

```

int* alphaOf;
int* indexOf;
int m, GF, t, n, k, r;
        /*  m symbol size
           GF(q^m)
           q=2
           t correcting power
           r=2t number redundant bits
           n number of bits in total (q^m-1)
           k number of data bits
           n=k+r
        */
int bitsInBuffer;

public:
RS(int mValue, int tValue); //GF(q^m) -- q=2 -- t correcting power

int alpha(int index);
int index(int number);
int getGFsize();

protected:
int addOrNull(int operand1, int operand2, int modulus);
int subOrNull(int operand1, int operand2, int modulus);
};

#endif

// RS.cc
#include "RS.h"
#include "debugger.h"
#include "debug.h"

RS::RS(int mValue, int tValue): m(mValue), t(tValue), GF(1), indexOf(0),
alphaOf(0)
{
    if (m<3 || m>6)
    {
        cerr << "m should be in between 3 and 6" << endl;
        exit(1);
    }

    for (int power=1; power<=m; power++)
        GF*=2;//GF=q^m (q=2)
    n=GF-1;
    r=t<<1; // r=2t number of redundant bits

    if (r>=n)
    {
        cerr << "the number of redundant bits r=" << r
            << " is greater than the "

```

```

        << "maximum size n=" << n << endl;
        exit(2);
    }

    k=n-r;
    cout << dec << "code: RS(" << n << "," << k << ")" << endl;

    // construction of the two tables:index -> number (alphaOf[index])
    //                                number -> index (indexOf[number])
    alphaOf = new int[n+1];
    indexOf = new int[n+1];
    alphaOf[0]=1;
    indexOf[0]=n;
    alphaOf[n]=indexOf[1]=0;

    int f[]={0,0x3,0x7,0xB,0x13,0x25,0x43};
        // Galois Field generator polynomials

#ifdef _debug_
    cout << "index\tnbr\tbinary" << endl;
    cout << 0 << "\t" << 1 << "\t";
    binaire(1,cout);
    cout << endl;
#endif

    //generation of the field and its associated indexes:
    for (int index=1; index<n; index++)
    {
        alphaOf[index]=alphaOf[index-1]<<1; // shift the registers
        alphaOf[index]^=(alphaOf[index]>>m)*f[m];
            // feedback last bit through generator polynomial
        indexOf[alphaOf[index]]=index;

#ifdef _debug_
        cout << index << "\t" << alphaOf[index] << "\t";
        binaire(alphaOf[index],cout);
        cout << endl;
#endif
    }
    alphaOf[0]=1;
    indexOf[0]=n;
    alphaOf[n]=indexOf[1]=0;
}

int RS::alpha(int index)
{
    return alphaOf[index];
}

int RS::index(int number)
{

```

```
        return indexOf[number];
    }

int RS::getGFsize()
{
    return GF;
}

int RS::addOrNull(int operand1, int operand2, int modulus)
{
    int temp;
    if (operand1==modulus || operand2==modulus)
        return modulus;

    if ((temp=(operand1+operand2)%modulus)<0)
        return temp+modulus;
    else
        return temp;
}

int RS::subOrNull(int operand1, int operand2, int modulus)
{
    int temp;
    if (operand1==modulus)
        return modulus;

    if (operand2==modulus)
    {
        //cerr<< "Can't divide by zero!!!" << endl;
        return INT_MAX;
    }

    if ((temp=(operand1-operand2)%modulus)<0)
        return temp+modulus;
    else
        return temp;
}
```

A.5 RSEncoder

```
// RSEncoder.h
#ifndef _RSEncoder_h
#define _RSEncoder_h

#include "RS.h"

class RSEncoder: public RS
{
```

```

    unsigned int* generator;//indexes of generator polynomial for the
code
    unsigned int* systematic; // systematic symbols buffer.
    unsigned int* FSR;//feedback shift registers for redundant bits
gen.
    unsigned int SBin;// buffer to build symbols (8bits->mbits)
    int SBinBits;//number of bits in the symbol buffer
    int padBits;
    unsigned int SBout;// buffer to build symbols (8bits->mbits)
    int SBoutBits;//number of bits in the symbol buffer
    int inputCount;
    int outputCount;
    int lastWrite;

    int bytesToOutput;
    int nbrBytesOutput;
    int remainingBits;

    int          symbol,feedback,FSRindex,FSRregister,
                FSRregBit,maxOutput,CBbits;
    unsigned intcharBuffer, output;
    unsigned char tempChar;

    public:
    RSEncoder(int mValue, int tValue);
    ~RSEncoder();
    int processData();
    int processSymbols();

    void newData();
};

#endif

// RSEncoder.cc
#include "RSEncoder.h"
#include "debugger.h"
#include "debug.h"

RSEncoder::RSEncoder(int mValue, int tValue): RS(mValue,tValue),
generator(0),
    systematic(0), FSR(0), SBin(0), SBinBits(0), padBits(0),
SBout(0),
    SBoutBits(0), inputCount(0), outputCount(0),
bytesToOutput(0),
    nbrBytesOutput(0), remainingBits(0), lastWrite(1),
    symbol(0),feedback(0),FSRindex(0),FSRregister(0),
    FSRregBit(0),maxOutput(0),CBbits(0),
    charBuffer(0), output(0),
    tempChar(0)
{

```

```

    int nbrBits;

    nbrBits=k*m; // the output is done with chars
    bytesToOutput=nbrBits/8;// determine the number of bytes in the
    remainingBits=nbrBits%8;// input information (k symbols of m bits)

    generator=new unsigned int[r+1];

    generator[0]=1;
    generator[1]=0;

    int index;
    int power;

    /*
    constructs the code generator polynomial
    */
    for (power=2; power<=r; power++)
    {
        generator[power]=0;
        for (index=power-1; index>=1; index--)
            generator[index]=
                indexOf[ alphaOf[addOrNull(power,generator[index],n)]
                    ^ alphaOf[generator[index-1]] ];
        /*
        the program switches between indexes and numbers because
        addition is easier w/ numbers (XOR) and
        multiplication is easier with indexes (sum of indexes mod
n)
        */

        generator[0]=addOrNull(generator[0],power,n);
    }

    #ifdef _debug_
    for (power=r; power>=0; power--)
        cout << " " << generator[power];
    cout << endl;
    #endif

    systematic=new unsigned int[k];
    FSR=new unsigned int[r];

    for (int FSRindex=0;FSRindex<=r-1;FSRindex++)
        FSR[FSRindex]=n;// alpha[n]=0

    FSRregister=r-1;
}

RSEncoder::~RSEncoder()
{

```

```

        delete generator;
        delete systematic;
        delete FSR;
    }

    int RSEncoder::processData()
    {
        int inNbrBits=0;

        if (!inBuffer || !outBuffer)
        {
            cerr << "Please assign me with some input and output "
                 << "buffers before asking me to process data" << endl;
            return -1;
        }

        dataInput_Process:
        eof=inBuffer->atEOF();
        if ( (inputCount==0) && (outputCount==0) && (SBinBits==0 ||
padBits>0)
                && (SBoutBits<=0) && (eof) )
        {
            outBuffer->setEOF();
            return 1;
        }

        /*
        input the first k symbols (of length m bits) outputs them
        (x^r*i(x))
        and process them through the FSR
        */
        while ( (inputCount<k) && (outBuffer->testWrite()) &&
                ( inBuffer->testRead() || (eof=inBuffer-
>atEOF()) ) )
        {
            if (SBinBits<m)
            {
                // padBits flags when the system started padding the
information
                // (when there is not enough info to fill info block so 0
pad)
                if (eof)
                {
                    SBin=(SBin<<8)+0x0;
                    padBits+=8;
                    inNbrBits=8;
                }
                else
                {
                    inNbrBits=inBuffer->get(tempChar);
                    SBin=(SBin<<inNbrBits)+(tempChar>>(8-inNbrBits));

```

```

        }
        SBinBits+=inNbrBits;
    }

    while (SBinBits>=m)
    {
        if (inputCount>=k)
            goto outputFSR;
        // extract symbol from symbol buffer (SBin):
        SBinBits-=m;
        symbol=SBin>>SBinBits;
        systematic[inputCount]=symbol;
        SBin^=symbol<<SBinBits;
        inputCount++;
#ifdef _debug_
        cout << "new data: " << symbol << " ";
        binaire(symbol,cout,m);
        cout << endl;
#endif
        // FSR:
#ifdef _debug_
        cout << "before: ";
        for (FSRindex=r-1;FSRindex>=0;FSRindex--)
            cout << alphaOf [FSR[FSRindex]] << " ";
        cout <<endl;
#endif

        feedback=indexOf [alphaOf [FSR[r-1]]^symbol];
        for (FSRindex=r-1;FSRindex>=1;FSRindex--)
            FSR[FSRindex]=indexOf [
                alphaOf [addOrNull (feedback,generator [FSRindex],n)]
                    ^alphaOf [FSR[FSRindex-1]]];
        FSR[0]=addOrNull (feedback,generator [0],n);

#ifdef _debug_
        cout << "after: ";
        for (FSRindex=r-1;FSRindex>=0;FSRindex--)
            cout << alphaOf [FSR[FSRindex]] << " ";
        cout <<endl;
#endif
    }
}

/*
    output the content of the Feedback Shift Register
*/
outputFSR:
#ifdef _debug_
cout << endl;
#endif
while ( (inputCount>=k)    &&    (outputCount<n)    &&    (outBuffer-

```

```

>testWrite() )
{
    // output systematic information and then redundant info.
    #ifdef _debug_
    cout << "outputFSR: " << outputCount << endl;
    #endif

    while (SBoutBits<8 && FSRregister>=0)
    {
        if (outputCount<k)
            // systematic
            SBout=(SBout<<m)+systematic[outputCount];
        else
            // redundant
            {
                SBout=(SBout<<m)+(alphaOf[FSR[FSRregister]]);
                FSRregister--;
            }
        SBoutBits+=m;
        outputCount++;
    }

    if (SBoutBits>=8)
    {
        SBoutBits-=8;
        output= (char)(SBout>>SBoutBits);
        outBuffer->put((unsigned char)output);
        SBout^=output<<SBoutBits;
    }
}
// after this stage SBoutBits is strictly less than 8

// if at end of file and still some bits in SBout, output them
if ( (inputCount>=k) && (outputCount>=n) && (SBinBits==0 ||
padBits>0)
                                && (SBoutBits>0) && (eof=inBuffer-
>atEOF()) )
{
    if (lastWrite=outBuffer->testWrite())
    {
        output=(char)(SBout<<(8-SBoutBits));
        outBuffer->last((unsigned char)output,SBoutBits);
        SBoutBits=0;
    }
}

/*
    initialize all the variables and buffers for the next set of
    data to be processed (unless it's the end of the file)
*/
if ( (inputCount>=k) && (outputCount>=n) && (lastWrite) )

```

```
{
    inputCount=0;
    outputCount=0;
    FSRregister=r-1;
    nbrBytesOutput=0;

    for (FSRindex=0;FSRindex<=r-1;FSRindex++)
        FSR[FSRindex]=n;// alpha[n]=0

    goto dataInput_Process;
}

return 1;
}

void RSEncoder::newData()
{
    if (inputCount!=0 && (!eof) )
        cerr << "Warning from Reed Solomon encoder: new data"
              << " requested while the previous data were still "
              << "being processed." << endl;

    RS::newData();

    SBin=0;
    SBinBits=0;
    padBits=0;
    SBout=0;
    SBoutBits=0;
    inputCount=0;
    outputCount=0;
    bytesToOutput=0;
    nbrBytesOutput=0;
    remainingBits=0;
    lastWrite=1;
}

int RSEncoder::processSymbols()
{
    static unsigned int feedback=0,FSRindex=0,FSRregister=r-1,FSRregBit=0,
        maxOutput=0,CBbits=0;
    static unsigned int charBuffer=0, output=0;
    static unsigned char tempChar=0, symbol=0;

    if (!inBuffer || !outBuffer)
    {
        cerr << "Please assign me with some input and output "
              << "buffers before asking me to process data" << endl;
        return -1;
    }
}
```

```

dataInput_Process:
if ( (inputCount==0) && (outputCount==0) && (SbinBits==0 ||
padBits>0)
    && (SBoutBits<=0) && (eof=inBuffer->atEOF()) )
{
    outBuffer->setEOF();
    return 1;
}

/*
input the first k symbols (of length m bits) outputs them
(x^r*i(x))
and process them through the FSR
*/
while ( (inputCount<k) && (outBuffer->testWrite()) &&
    ( inBuffer->testRead() || (eof=inBuffer->atEOF()) ) )
{
    if (eof)
    {
        symbol=0xb;
        padBits=1;
    }
    else
        inBuffer->get(symbol);

    systematic[inputCount]=symbol;
    inputCount++;
#ifdef _debug_
    cout << "new data: " << symbol << " ";
    binaire(symbol,cout,m);
    cout << endl;
#endif
// FSR:
#ifdef _debug_
    cout << "before: ";
    for (FSRindex=r-1;FSRindex>=0;FSRindex--)
        cout << alphaOf[FSR[FSRindex]] << " ";
    cout << endl;
#endif

    feedback=indexOf[alphaOf[FSR[r-1]]^symbol];
    for (FSRindex=r-1;FSRindex>=1;FSRindex--)
        FSR[FSRindex]=indexOf[
            alphaOf[addOrNull(feedback,generator[FSRindex],n)]
                ^alphaOf[FSR[FSRindex-1]]];
    FSR[0]=addOrNull(feedback,generator[0],n);

#ifdef _debug_
    cout << "after: ";
    for (FSRindex=r-1;FSRindex>=0;FSRindex--)

```

```

        cout << alphaOf[FSR[FSRindex]] << " " ;
        cout << endl;
    #endif
}

/*
    output the content of the Feedback Shift Register
*/
outputFSR:
#ifdef _debug_
    cout << endl;
#endif

    while ( (inputCount>=k)  &&  (outputCount<n)  &&  (outBuffer->testWrite()) )
    {
        // output systematic information and then redundant info.
        #ifdef _debug_
            cout << "outputFSR: " << outputCount << endl;
        #endif

        if (outputCount<k)
            // systematic
            output=systematic[outputCount];
        else
            // redundant
            {
                output=alphaOf[FSR[FSRregister]];
                FSRregister--;
            }
        outBuffer->put((unsigned char)output);
        outputCount++;
    }

/*
    initialize all the variables and buffers for the next set of
    data to be processed (unless it's the end of the file)
*/
if ( (inputCount>=k) && (outputCount>=n) )
{
    inputCount=0;
    outputCount=0;
    FSRregister=r-1;
    nbrBytesOutput=0;

    for (FSRindex=0;FSRindex<=r-1;FSRindex++)
        FSR[FSRindex]=n;// alpha[n]=0

    goto dataInput_Process;
}

```

```
        return 1;
    }
```

A.6 RSDecoder

```
// RSDecoder.h
#ifndef _RSDecoder_h
#define _RSDecoder_h

#include "RS.h"
#include <iomanip.h>

class RSDecoder: public RS
{
    int* codeWord;
    int* syndrome;

    int inputCount;
    int outputCount;
    unsigned int SBin;
    int SBinBits;
    char outputDataFlag;
    unsigned int SBout;
    int SBoutBits;

    int blockNbr;

    int codeWordCounter;

    unsigned int*C;
    unsigned int*D;
    unsigned int*Ctp;
    unsigned int**errLoc;
    unsigned int*Xloc;
    unsigned int*errValue;
    int N,L,delta,Di,temp,result,con,pivot,err;

    unsigned char symbol,mask;
    unsigned char tempChar;
    int frameCount;

public:
    RSDecoder(int mValue, int tValue);
    int processData();
    int processSymbols();

    void newData();
};
```

```

};

int printMatrix(int matrix[5][3], RS& GF);
#endif

// RSDecoder.cc
#include "RSDecoder.h"
#include "debugger.h"
#include "debug.h"

RSDecoder::RSDecoder(int mValue, int tValue): RS(mValue, tValue),
codeWord(0),
    syndrome(0), inputCount(0), SBout(0), SBoutBits(0),
    outputCount(0), SBin(0), SBinBits(0), outputDataFlag(0),
    blockNbr(0),
    symbol(0), mask((1<<8)-1), tempChar(0), frameCount(0)
{
    codeWord=new int[n];
    syndrome= new int[r+1];

    for (int syndromeCounter=0; syndromeCounter<r+1; syndromeCounter++)
        syndrome[syndromeCounter]=n;// alphaOf [n]=0

    C=new unsigned int[r+1];
    D=new unsigned int[r+1];
    Ctp=new unsigned int[r+1];
    /*
    errLoc=new unsigned int*[2*(r-1)];
    for (int errLocCount=0; errLocCount<2*(r-1);errLocCount++)
        errLoc[errLocCount]=new unsigned int[r-1];
    Xloc=new unsigned int[r-1];
    errValue=new unsigned int[r-1];
    */
    errLoc=new unsigned int*[2*(n+1)];
    for (int errLocCount=0; errLocCount<2*(n+1);errLocCount++)
        errLoc[errLocCount]=new unsigned int[n+1];
    Xloc=new unsigned int[n+1];
    errValue=new unsigned int[n+1];
}

int RSDecoder::processData()
{
    int syndromeCounter;
    // unsigned int C[r+1], D[r+1], Ctp[r+1], errLoc[2*(r-1)][r-1],
    // Xloc[r-1], errValue[r-1];
    // static unsigned int
    *C=0, *D=0, *Ctp=0, **errLoc=0, *Xloc=0, *errValue=0;

    int cl, col, row, diag, polyPower;
    unsigned char charOut;

```

```

int                                     nbrBits=0;

if (!inBuffer || !outBuffer)
{
    cerr    << "Please assign me with some input and output "
            << "buffers before asking me to process data" << endl;
    return -1;
}

// This algorithm drops any impartially input RS frame at the end
// of the algorithm so it does not matter if it gets the exact
// number of bits from the input because it will drop the extra bits
anyways
startProcess:
if ( eof=inBuffer->atEOF() && (inputCount<n) )
{
    outBuffer->setEOF();
    if (inputCount!=0)
    {
        cout << "RSDecoder: dropped " << inputCount
              << " symbols at the end of the stream\n";
        inputCount=0;
    }
    return 1;
}

while ( (inputCount<n) && (inBuffer->testRead()) )
{
    if (SBinBits<m)
    {
        inBuffer->get(tempChar);
        #ifdef _debug_
            cout << "buffer input:";
            binaire(tempChar,cout);
            cout << endl;
        #endif
        SBin=(SBin<<8)+(tempChar&mask);
        SBinBits+=8;
        #ifdef _debug_
            cout << "data in:";
            binaire(SBin,cout,32);
            cout << endl;
        #endif
    }

    while (SBinBits>=m && inputCount<n)
    {
        // extract symbol from symbol buffer (SBin):
        SBinBits-=m;
        symbol=SBin>>SBinBits;
    }
}

```

```

        #ifdef _debug_
            cout << "new data: ";
            binaire(symbol,cout,m);
            cout << " " << hex << (int)symbol << endl;
        #endif
        SBin^=symbol<<SBinBits;
        codeWord[inputCount]=indexOf[symbol];
        // syndrome calculation:
        for (syndrCounter=1;syndrCounter<r+1;syndrCounter++)
        {
            syndrome[syndrCounter]=
                indexOf[
                    alphaOf[ addOrNull(syndrome[syndrCounter],
                                        syndrCounter,n) ]^
                    alphaOf[ codeWord[inputCount] ] ];
            #ifdef _debug_
                cout << "syndrome " << syndrCounter << " "
                    << alphaOf[syndrome[syndrCounter]] << endl;
            #endif
        }
        inputCount++;
    }

    if (inputCount>=n)
    {
        #ifdef _debug_
            cout << "remaining bits " << SBinBits << endl;
        #endif

        #ifdef _debug_
            for (syndrCounter=1;syndrCounter<r+1;syndrCounter++)
            {
                cout << "syndrome " << syndrCounter << " "
                    << alphaOf[syndrome[syndrCounter]] << endl;
            }
        #endif

        //cout<< "Decoding block #" << (++blockNbr) << endl;
    }

}

int Lold;

if (inputCount>=n && outputDataFlag==0)
{
    /*
        Massey algorithm to calculate the error locator polynomial
    */

```

```

    for (c1=0;c1<=r;c1++)
        C[c1]=D[c1]=Ctp[c1]=n;
    //C[0]=D[1]=L=0;
    C[0]=D[0]=L=0;
    for (N=1;N<r+1;N++)
    {
        #ifdef _debug_
            cout << "iteration #" << N << endl;
            cout << "syndrome " << dec << alphaOf[syndrome[N]] <<
endl;

            #endif

            delta=alphaOf[syndrome[N]];
            #ifdef _debug_
                cout << "delta " << indexOf[delta] << endl;
            #endif
            for (c1=1;c1<=L;c1++)
            {
                #ifdef _debug_
                    cout << "CS " << C[c1] << " " << syndrome[N-c1]
<< " " << addOrNull(C[c1],syndrome[N-c1],n) << endl;
                #endif
                delta=delta^alphaOf[addOrNull(C[c1],syndrome[N-
c1],n)];

                #ifdef _debug_
                    cout << "delta " << indexOf[delta] << endl;
                #endif
            }
            #ifdef _debug_
                cout << "delta " << indexOf[delta] << endl;
            #endif

            if (delta!=0)
            {
                Di=indexOf[delta];
                //
                Ctp[0]=C[0];
                //for (c1=0;c1<=L+1;c1++)
                for (c1=1;c1<=r;c1++)
                {
                    //temp=addOrNull(Di,D[c1],n);
                    temp=addOrNull(Di,D[c1-1],n);
                    #ifdef _debug_
                        cout << "del*D[" << c1-1 << "]= " << temp <<
" C=" << C[c1] << endl;
                    #endif
                    Ctp[c1]=indexOf[alphaOf[C[c1]]^
                        alphaOf[temp]];
                }

                #ifdef _debug_

```

```

        cout << "C* ";
        for (c1=0;c1<=L+1;c1++)
            cout << Ctp[c1] << " ";
        cout << endl;
    #endif

    if (2*L<N)
    {
        for (c1=0;c1<=r;c1++)
            if( (D[c1]=subOrNull(C[c1],Di,n))==INT_MAX
)
            {
                outputDataFlag=1;
                goto outputData;
            }
    #ifdef _debug_
        cout << "D: ";
        for (c1=0;c1<=L+1;c1++)
            cout << D[c1] << " ";
        cout << endl;
    #endif
        L=N-L;
    #ifdef _debug_
        cout << "L=" << L << endl;
    #endif
    }
    else
    {
        for (c1=L+1;c1>=1;c1--)
            D[c1]=D[c1-1];
        D[0]=n;
    }
    for (c1=0;c1<=L;c1++)
        C[c1]=Ctp[c1];

    #ifdef _debug_
        cout << "C ";
        for (polyPower=0;polyPower<=L;polyPower++)
            cout << C[polyPower] << " ";
        cout << endl;
    #endif
}
else
{
    for (c1=L+1;c1>=1;c1--)
        D[c1]=D[c1-1];
    D[0]=n;
}
#ifdef _debug_
    cout << "D: ";
    for (c1=0;c1<=L+1;c1++)

```

```

        cout << D[c1] << " ";
        cout << endl;
    #endif
}

#ifdef _debug_
cout << "C ";
for (polyPower=0;polyPower<=L;polyPower++)
    cout << C[polyPower] << " ";
cout << endl;
#endif

/*
    chien search of the roots of the error locator polynomial
*/
err=0;
#ifdef _debug_
    cout << "error places: ";
#endif

for (int tryPower=0;tryPower<=n-1;tryPower++)
{
    result=n;// 0 (initialization)
    for (polyPower=0;polyPower<=L;polyPower++)
        result=indexOf[alphaOf[C[polyPower]]^
            alphaOf[addOrNull(result,tryPower,n)]];
    if (result==n)// alphaOf[tryPower] is a root
    {
        err++;
        Xloc[err]=tryPower;
        #ifdef _debug_
            cout << err << ":" << Xloc[err] << " ";
        #endif
    }
    if (err==L)
        break;
}
#ifdef _debug_
cout << endl;
#endif

#ifdef _debug_
    cout << "error places: ";
    for (c1=1;c1<=err;c1++)
        cout << c1 << ":" << Xloc[c1] << " ";
    cout << endl;
#endif

// if there is more error than the code can correct, just
// the data without correction

```

output

```

    if (err!=L)
    {
#ifdef _debug_
        cout << "The number of errors is greater"
              << " than the error correction capabilities"
              << " of the code" << endl;
        cout << "number of errors " << err << endl;
        cout << "L " << L << endl;
#endif
        L=err;
        //outputDataFlag=1;
        //goto outputData;
        //return -1;
    }

    /*
    Calculation of the error values (inversion of the error
    locator matrix)
    */
    for (col=1;col<=L;col++)
    {
        for (row=1;row<=L;row++)
        {
            if ((Xloc[col]%n)==0 && Xloc[col]!=0)
                errLoc[col][row]=n;
            else if ((Xloc[col]*row)%n==0)
                errLoc[col][row]=0;
            else
                errLoc[col][row]=addOrNull(Xloc[col]*row,0,n);
            errLoc[col+L][row]=n;
            if (row==col)
                errLoc[col+L][row]=0;
        }
    }

#ifdef _debug_
    for (row=1;row<=L;row++)
    {
        for (col=1;col<=2*L;col++)
        {
            cout << dec << setw(3) << (int)errLoc[col][row] << "
";
        }
        cout << endl;
    }
#endif
    //if (L>0)
    //printMatrix(errLoc,*this);

```

```

        for (diag=1;diag<=L;diag++)
        {
            pivot=errLoc[diag][diag];
            for (col=diag;col<=2*L;col++)
            {
                errLoc[col][diag]=
                    subOrNull(errLoc[col][diag],pivot,n);
                if (errLoc[col][diag]==INT_MAX)
                {
                    outputDataFlag=1;
                    goto outputData;
                }
                //printMatrix(errLoc,*this);
            }
            for (row=1;row<=L;row++)
            {
                if (diag!=row)
                {
                    con=errLoc[diag][row]-errLoc[diag][diag];
                    for (col=diag;col<=2*L;col++)
                    {
                        errLoc[col][row]=
                            indexOf[alphaOf[errLoc[col][row]]^
                                alphaOf[addOrNull(errLoc[col][diag],
                                    con,n)]];
                        //printMatrix(errLoc,*this);
                    }
                }
            }
        }
/*
    for (row=1;row<=L;row++)
    {
        for (col=1;col<=L;col++)
        {
            errLoc[col][row]=n;
            for (int i=1;i<=L;i++)
            {
                errLoc[col][row]=
                    indexOf[alphaOf[errLoc[col][row]]^
                        alphaOf[addOrNull(errLoc[i+L][row],
                            addOrNull(Xloc[col]*i,0,n),n)]];
                cout << alphaOf[errLoc[col][row]] << " ";
            }
            cout << endl;
        }
    }
*/
#ifdef _debug_
    cout << endl << endl;
    for (row=1;row<=L;row++)
    {
        for (col=1;col<=2*L;col++)
        {
            cout << dec << setw(3) << (int)errLoc[col][row] << "

```

```

";
        }
        cout << endl;
    }

    for (syndrCounter=1;syndrCounter<=L;syndrCounter++)
    {
        cout << "syndrome "
              << syndrome[syndrCounter] << endl;
    }
#endif

    /*
     calculates the error correction terms:
    */
    for (row=1;row<=L;row++)
    {
        result=0;
        for (col=1;col<=L;col++)
            result^=alphaOf[addOrNull(syndrome[col],
                                     errLoc[col+L][row],n)];
        errValue[row]=indexOf[result];
#ifdef _debug_
        cout << "error at " << dec << (n-Xloc[row])
              << " is " << hex << alphaOf[errValue[row]] <<
endl;
#endif
    }

    /*
     corrects the input stream
    */
    int errPos;
    for (c1=1;c1<=L;c1++)
    {
        errPos=Xloc[c1];
        codeWord[n-errPos-1]=indexOf[alphaOf[codeWord[n-errPos-
1]]^
                                     alphaOf[errValue[c1]]];
    }

    outputDataFlag=1;
}

outputData:
while ( (outputDataFlag)  && (outputCount<k) &&
      (outBuffer->testWrite()) )
{
    while (SBoutBits<8 && outputCount<k)
    {
        SBout<<=m;
    }
}

```

```

        SBout+=alphaOf [codeWord [outputCount]] ;
        SBoutBits+=m;
        outputCount++;
    }

    if (SBoutBits>=8)
    {
        SBoutBits-=8;
        charOut=SBout>>SBoutBits;
        SBout^=charOut<<SBoutBits;
        outBuffer->put((unsigned char)charOut&0xFF);
#ifdef _debug_
        cout << "output:" << hex
        << (int)(charOut&0xFF)
        << " ";
#endif
    }
}

if ( (outputDataFlag==1) && (outputCount>=k) )
{
#ifdef _debug_
    cout << endl << endl << dec << ++frameCount << "\tnew frame" <<
endl;
#endif

    outputDataFlag=0;
    outputCount=0;
    inputCount=0;

    for (int syndromeCounter=0; syndromeCounter<r+1;
syndromeCounter++)
        syndrome[syndromeCounter]=n;// alphaOf[n]=0

    inBuffer->reSync();
    outBuffer->reSync();

    goto startProcess;
}

#ifdef _debug_
cout << "That's all folks!!!" << endl;
#endif
}

int printMatrix(int matrix[5][3],RS& GF)
{
    for (int row=1;row<=2;row++)
    {
        for (int col=1;col<=4;col++)

```

```

        cout << setw(2) << setfill(' ')
              << GF.alpha(matrix[col][row]) << " ";
    cout << " ";
    for (int col=1;col<=4;col++)
        cout << setw(2) << setfill(' ')
              << matrix[col][row] << " ";
    cout << endl;
}
cout << endl;
}

void RSDecoder::newData()
{
    RS::newData();

    inputCount=0;
    outputCount=0;
    SBin=0;
    SBinBits=0;
    SBout=0;
    SBoutBits=0;
    outputDataFlag=0;
    blockNbr=0;

    for (int syndromeCounter=0; syndromeCounter<r+1; syndromeCounter++)
        syndrome[syndromeCounter]=n;// alphaOf[n]=0
}

int RSDecoder::processSymbols()
{
    static unsigned char ask=(1<<8)-1;
    int syndrCounter;
    static unsigned char tempChar,symbol=0;

    if (!inBuffer || !outBuffer)
    {
        cerr << "Please assign me with some input and output "
              << "buffers before asking me to process data" << endl;
        return -1;
    }

    startProcess:
    if ( eof=inBuffer->atEOF() && (inputCount<n) )
    {
        outBuffer->setEOF();
        return 1;
    }

    while ( (inputCount<n) && (inBuffer->testRead()) )

```

```

{
    inBuffer->get(symbol);
    codeWord[inputCount]=indexOf[symbol];

    // syndrome calculation:
    for (syndrCounter=1;syndrCounter<r+1;syndrCounter++)
    {
        syndrome[syndrCounter]=
            indexOf[
                alphaOf[ addOrNull(syndrome[syndrCounter],
                                   syndrCounter,n) ]^
                alphaOf[ codeWord[inputCount] ] ];
#ifdef _debug_
        cout << "syndrome " << syndrCounter << " "
              << alphaOf[syndrome[syndrCounter]] << endl;
#endif
    }
    inputCount++;

    if (inputCount>=n)
    {
#ifdef _debug_
        for (syndrCounter=1;syndrCounter<r+1;syndrCounter++)
        {
            cout << "syndrome " << syndrCounter << " "
                  << alphaOf[syndrome[syndrCounter]] << endl;
        }
#endif

        cout << "Decoding block #" << (++blockNbr) << endl;
    }

}

//unsigned int C[r+1],D[r+1],Ctp[r+1],errLoc[2*(r-1)][r-1],
//
//                Xloc[r-1],errValue[r-1];

unsigned int *C,*D,*Ctp,**errLoc,*Xloc,*errValue;

C=new unsigned int[r+1];
D=new unsigned int[r+1];
Ctp=new unsigned int[r+1];
errLoc=new unsigned int*[2*(r-1)];
for (int errLocCount=0; errLocCount<2*(r-1);errLocCount++)
    errLoc[errLocCount]=new unsigned int[r-1];
Xloc=new unsigned int[r-1];
errValue=new unsigned int[r-1];

static int N,L,delta,Di,temp,result,con,pivot,err;
int cl,col,row,diag,polyPower;

```

```

unsigned char charOut;

if (inputCount>=n && outputDataFlag==0)
{
    /*
     * Massey algorithm to calculate the error locator polynomial
     */
    for (c1=0;c1<=r;c1++)
        C[c1]=D[c1]=Ctp[c1]=n;
    C[0]=D[1]=L=0;
    for (N=1;N<r+1;N++)
    {
        #ifdef _debug_
            cout << "syndrome " << syndrome[N] << endl;
        #endif

        delta=alphaOf[syndrome[N]];
        for (c1=1;c1<=L;c1++)
            delta=delta^alphaOf[addOrNull(C[c1],syndrome[N-
c1],n)];

        #ifdef _debug_
            cout << "delta " << indexOf[delta] << endl;
        #endif

        if (delta!=0)
        {
            Di=indexOf[delta];
            for (c1=0;c1<=L+1;c1++)
            {
                temp=addOrNull(Di,D[c1],n);
                Ctp[c1]=indexOf[alphaOf[C[c1]]^
                    alphaOf[temp]];
            }

            #ifdef _debug_
                cout << "C* ";
                for (c1=0;c1<=L;c1++)
                    cout << Ctp[c1] << " ";
                cout << endl;
            #endif

            if (2*L<N)
            {
                for (c1=0;c1<=L;c1++)
                    D[c1]=subOrNull(C[c1],Di,n);
                L=N-L;
            }
            for (c1=0;c1<=L;c1++)
                C[c1]=Ctp[c1];
        }
    }
}

```

```

        #ifdef _debug_
        cout << "C ";
        for (polyPower=0;polyPower<=L;polyPower++)
            cout << C[polyPower] << " ";
        cout << endl;
        #endif
    }
    for (c1=L+1;c1>=1;c1--)
        D[c1]=D[c1-1];
    D[0]=n;
    #ifdef _debug_
    cout << "D: ";
    for (c1=0;c1<=L+1;c1++)
        cout << D[c1] << " ";
    cout << endl;
    #endif
}

#ifdef _debug_
cout << "C ";
for (polyPower=0;polyPower<=L;polyPower++)
    cout << C[polyPower] << " ";
cout << endl;
#endif

/*
    chien search of the roots of the error locator polynomial
*/
err=0;
#ifdef _debug_
    cout << "error places: ";
#endif

for (int tryPower=0;tryPower<=n-1;tryPower++)
{
    result=n;// 0 (initialization)
    for (polyPower=0;polyPower<=L;polyPower++)
        result=indexOf[alphaOf[C[polyPower]]^
            alphaOf[addOrNull(result,tryPower,n)]];
    if (result==n)// alphaOf[tryPower] is a root
    {
        err++;
        Xloc[err]=tryPower;
        #ifdef _debug_
            cout << err << ":" << Xloc[err] << " ";
        #endif
    }
    if (err==L)
        break;
}
#ifdef _debug_

```

```

        cout << endl;
    #endif

    #ifdef _debug_
        cout << "error places: ";
        for (c1=1;c1<=err;c1++)
            cout << c1 << ":" << Xloc[c1] << " ";
        cout << endl;
    #endif

    // if there is more error than the code can correct, just
output    // the data without correction
    if (err!=L)
    {
greater"        cout << "Decoding failure:\nThe number of errors is
                << " than the error correction capabilities"
                << " of the code" << endl;
                //outputDataFlag=1;
                //goto outputData;
                return -1;
    }

    /*
        Calculation of the error values (inversion of the error
        locator matrix)
    */
    for (col=1;col<=L;col++)
    {
        for (row=1;row<=L;row++)
        {
            errLoc[col][row]=addOrNull(Xloc[col]*row,0,n);
            errLoc[col+L][row]=n;
            if (row==col)
                errLoc[col+L][row]=0;
        }
    }

    //if (L>0)
        //printMatrix(errLoc,*this);

    for (diag=1;diag<=L;diag++)
    {
        pivot=errLoc[diag][diag];
        for (col=diag;col<=2*L;col++)
        {
            errLoc[col][diag]=
                subOrNull(errLoc[col][diag],pivot,n);

```



```

        //printMatrix(errLoc,*this);
    }
    for (row=1;row<=L;row++)
    {
        if (diag!=row)
        {
            con=errLoc[diag][row]-errLoc[diag][diag];
            for (col=diag;col<=2*L;col++)
                errLoc[col][row]=
                    indexOf[alphaOf[errLoc[col][row]]^
                        alphaOf[addOrNull(errLoc[col][diag],
                            con,n)]];
            //printMatrix(errLoc,*this);
        }
    }
}

/*
calculates the error correction terms:
*/
for (row=1;row<=L;row++)
{
    result=0;
    for (col=1;col<=L;col++)
        result^=alphaOf[addOrNull(syndrome[col],
            errLoc[col+L][row],n)];
    errValue[row]=indexOf[result];
#ifdef _debug_
    cout << "error at " << Xloc[row]
        << " is " << errValue[row] << endl;
#endif
}

/*
corrects the input stream
*/
int errPos;
for (c1=1;c1<=L;c1++)
{
    errPos=Xloc[c1];
    codeWord[n-errPos-1]=indexOf[alphaOf[codeWord[n-errPos-
1]]^
                                alphaOf[errValue[c1]]];
}

outputDataFlag=1;
}

outputData:
while ( (outputDataFlag)  && (outputCount<k) &&

```

```

                                (outBuffer->testWrite()) )
    {
        symbol=alphaOf [codeWord [outputCount]];
        outputCount++;

        outBuffer->put ((unsigned char) symbol&0xFF);
    }
#ifdef _debug_
    cout << endl << endl;
#endif

    if ( (outputDataFlag==1) && (outputCount>=k) )
    {
        outputDataFlag=0;
        outputCount=0;
        inputCount=0;

        for      (int      syndromeCounter=0;      syndromeCounter<r+1;
syndromeCounter++)
            syndrome[syndromeCounter]=n; // alphaOf [n]=0

        inBuffer->reSync();
        outBuffer->reSync();

        goto startProcess;
    }

#ifdef _debug_
    cout << "That's all folks!!!" << endl;
#endif
}

```

A.7 interleaver

```

// interleaver.h
#ifndef _interleaver_h
#define _interleaver_h

#include "RS.h"
#include "coDec.h"

class interleaver: public coDec
{
    int      nbrRow,nbrCol;
    int      rowOut;
    int      colOut;
    int      bufSize;
    int      nbrChar;

```

```

    unsigned char**buffer;
    unsigned charcharIn;
    unsigned charcharOut;

    unsigned int SBin; // buffer to build symbols (8bits->mbits)
    int SBinBits; //number of bits in the symbol buffer
    int padBits;
    int realBits;
    unsigned int SBout; // buffer to build symbols (8bits->mbits)
    int SBoutBits; //number of bits in the symbol buffer
    int lastWrite;

    int bytesToOutput;
    int nbrBytesOutput;
    int remainingBits;

    int inputCount;
    int outputCount;

    public:
    interleaver( int Size );
    interleaver( int _nbrRow, int _nbrCol);
    ~interleaver();

    int processData();

    void newData();
};

#endif

// interleaver.cc
#include "interleaver.h"

interleaver::interleaver(int Size):
    nbrCol(Size),nbrRow(Size),inputCount(0),
    outputCount(0), charIn(0), bufSize(0), rowOut(0),
    colOut(0),
    nbrChar(0), SBin(0), SBinBits(0), SBout(0), SBoutBits(0)
{
    if (Size<=0)
    {
        cerr << "The size of the interleaver is negative or null!!!" <<
endl;
        exit(-1);
    }

    bufSize=Size*Size;
    bytesToOutput=bufSize/8;
    remainingBits=bufSize%8;

```

```
    buffer=new unsigned char*[Size];

    if (!buffer)
    {
        cerr << "Couldn't allocate memory for the interleaver of size "
<< Size << endl;
        exit(-2);
    }

    for (int i=0;i<Size;i++)
    {
        buffer[i]=new unsigned char[Size];

        if (!buffer[i])
        {
            cerr << "Couldn't allocate memory for the interleaver of
size " << Size << endl;
            exit(-2);
        }
    }

    for (int i=0;i<Size;i++)
        for (int j=0;j<Size;j++)
            buffer[i][j]=0;
}

interleaver::interleaver(int      _nbrRow,      int      _nbrCol):
nbrRow(_nbrRow),nbrCol(_nbrCol),inputCount(0),
        outputCount(0),  charIn(0),  bufSize(0),  rowOut(0),
colOut(0),
        nbrChar(0), SBin(0), SBinBits(0), SBout(0), SBoutBits(0)
{
    if (nbrRow<=0 || nbrCol<=0)
    {
        cerr << "The size of the interleaver is negative or null!!!" <<
endl;
        exit(-1);
    }

    bufSize=nbrRow*nbrCol;
    bytesToOutput=bufSize/8;
    remainingBits=bufSize%8;

    buffer=new unsigned char*[nbrCol];

    if (!buffer)
    {
        cerr << "Couldn't allocate memory for the interleaver of size "
<< nbrRow << "x" << nbrCol << endl;
        exit(-2);
    }
}
```

```

    }

    for (int i=0;i<nbrCol;i++)
    {
        buffer[i]=new unsigned char[nbrRow];

        if (!buffer[i])
        {
            cerr << "Couldn't allocate memory for the interleaver of
size " << nbrRow << "x" << nbrCol << endl;
            for (int j=0; j<i; j++)
                delete buffer[j];
            delete buffer;
            buffer=0;
            exit(-2);
        }
    }

    for (int i=0;i<nbrCol;i++)
        for (int j=0;j<nbrRow;j++)
            buffer[i][j]=0;
}

interleaver::~interleaver()
{
    if (buffer!=0)
    {
        for (int i=0; i<nbrCol; i++)
            if (buffer[i]!=0)
                delete buffer[i];
        delete buffer;
    }
}

int interleaver::processData()
{
    int i=0,j=0,colIn,rowIn,bitCount;

    startProcess:

    if ( (eof=inBuffer->atEOF()) && inputCount==0 && outputCount==0)
    {
        outBuffer->setEOF();
        eof=1;
        return eof;
    }

    while ( (inputCount<bufSize) &&
            (inBuffer->testRead() || (eof=inBuffer->atEOF())) )

```

```
{
    if (SBinBits==0)
    {
        if (eof)
            charIn=0;
        else
            inBuffer->get(charIn);
        SBinBits+=8;
    }

    while (SBinBits>0 && inputCount<bufSize)
    {
        rowIn=inputCount/nbrCol;
        colIn=inputCount%nbrCol;

        rowOut=colIn;
        colOut=rowIn;
        if ( (charIn&(0x1<<(SBinBits-1))) )
            buffer[rowOut][colOut]=1;
        else
            buffer[rowOut][colOut]=0;
        inputCount++;
        SBinBits--;
    }
    //cout<< "coucou" << endl;
}

while ( (inputCount>=bufSize) &&
        (outputCount<bufSize) && (outBuffer->testWrite()) )
{
    while (SBoutBits<8 && outputCount<bufSize)
    {
        rowOut=outputCount/nbrRow;
        colOut=outputCount%nbrRow;
        SBout=(SBout<<1)+buffer[rowOut][colOut];
        outputCount++;
        SBoutBits++;
        //cout<< "Gutten tag" << endl;
    }

    if (SBoutBits>=8)
    {
        SBoutBits-=8;
        charOut=SBout>>SBoutBits;
        outBuffer->put(charOut);
        SBout^=charOut<<SBoutBits;
        //cout<< "bonjour" << endl;
    }
    else
    if (SBoutBits<8 && eof)
    {

```

```

        charOut=SBout<<(8-SBoutBits);
        outBuffer->put(charOut);
        SBoutBits=0;
        //cout<< "hello" << endl;
    }
}

if ( inputCount>=bufSize && outputCount>=bufSize )
{
    inputCount=0;
    outputCount=0;
    charIn=0;

    eof=inBuffer->atEOF();
    goto startProcess;
}

return 1;
}

void interleaver::newData()
{
    coDec::newData();

    inputCount=0;
    outputCount=0;
    charIn=0;
    for (int i=0;i<nbrCol;i++)
        for (int j=0;j<nbrRow;j++)
            buffer[i][j]=0;
}

```

A.8 interleaver_rand

```

// interleaver_rand.h
#ifndef _interleaver_rand_h
#define _interleaver_rand_h

#include "coDec.h"
#include <stdlib.h>

class interleaver_rand: public coDec
{
    int            interleaverSize;
    int            rowOut;
    int            colOut;
}

```

```

    int                bufSize;
    int                nbrChar;
    unsigned char*buffer;
    unsigned char*touched;
    unsigned charcharIn;
    unsigned charcharOut;

    unsigned int SBin; // buffer to build symbols (8bits->mbits)
    int SBinBits; //number of bits in the symbol buffer
    int padBits;
    int realBits;
    unsigned int SBout; // buffer to build symbols (8bits->mbits)
    int SBoutBits; //number of bits in the symbol buffer
    int lastWrite;

    int bytesToOutput;
    int nbrBytesOutput;
    int remainingBits;

    int inputCount;
    int outputCount;

    public:
    interleaver_rand( int Size );

    int processData();

    void newData();
};

#endif

// interleaver_rand.cc
#include "interleaver_rand.h"

interleaver_rand::interleaver_rand(int Size):
interleaverSize(Size),inputCount(0),
outputCount(0), charIn(0), bufSize(0), rowOut(0),
colOut(0),
nbrChar(0), SBin(0), SBinBits(0), SBout(0), SBoutBits(0)
{
    if (Size<=0)
    {
        cerr << "The size of the interleaver is negative or null!!!" <<
endl;
        exit(-1);
    }

    bufSize=Size;
    bytesToOutput=bufSize/8;

```

```
        remainingBits=bufSize%8;

        buffer=new unsigned char[Size];
        touched=new unsigned char[Size];

        if (!buffer || !touched)
        {
            cerr << "Couldn't allocate memory for the interleaver of size "
<< Size << endl;
            exit(-2);
        }

        for (int i=0;i<Size;i++)
        {
            buffer[i]=0;
            touched[i]=0;
        }

        srand48(0);
    }

int interleaver_rand::processData()
{
    int i=0,j=0,colIn,rowIn,bitCount,index;

    startProcess:

    if ( (eof=inBuffer->atEOF()) && inputCount==0 && outputCount==0)
    {
        outBuffer->setEOF();
        eof=1;
        return eof;
    }

    while ( (inputCount<bufSize) &&
            (inBuffer->testRead() || (eof=inBuffer->atEOF()))) )
    {
        if (SBinBits==0)
        {
            if (eof)
                charIn=0;
            else
                inBuffer->get(charIn);
            SBinBits+=8;
        }

        while (SBinBits>0 && inputCount<bufSize)
        {
            //index=lrnd48()%interleaverSize;
            do
```

```

        index=lrnd48()%interleaverSize;
        while (touched[index]);
        //cerr<< index << endl;

        touched[index]=1;
        if ( (charIn&(0x1<<(SBinBits-1))) )
            buffer[index]=1;
        else
            buffer[index]=0;
        inputCount++;
        SBinBits--;
    }
    //cout<< "coucou" << endl;
}

while ( (inputCount>=bufSize) &&
        (outputCount<bufSize) && (outBuffer->testWrite()) )
{
    while (SBoutBits<8 && outputCount<bufSize)
    {
        SBout=(SBout<<1)+buffer[outputCount];
        outputCount++;
        SBoutBits++;
        //cout<< "Gutten tag" << endl;
    }

    if (SBoutBits>=8)
    {
        SBoutBits-=8;
        charOut=SBout>>SBoutBits;
        outBuffer->put(charOut);
        SBout^=charOut<<SBoutBits;
        //cout<< "bonjour" << endl;
    }
    else
    if (SBoutBits<8 && eof)
    {
        charOut=SBout<<(8-SBoutBits);
        outBuffer->last(charOut,SBoutBits);
        SBoutBits=0;
        //cout<< "hello" << endl;
    }
}

if ( inputCount>=bufSize && outputCount>=bufSize )
{
    inputCount=0;
    outputCount=0;
    charIn=0;

```

```

        for (i=0;i<interleaverSize;i++)
            touched[i]=0;

        eof=inBuffer->atEOF();
        goto startProcess;
    }

    return 1;
}

void interleaver_rand::newData()
{
    coDec::newData();

    inputCount=0;
    outputCount=0;
    charIn=0;
    for (int i=0;i<interleaverSize;i++)
        buffer[i]=0;
}

```

A.9 deinterleaver_rand

```

// deinterleaver_rand.h
#ifndef _deinterleaver_rand_h
#define _deinterleaver_rand_h

#include "coDec.h"
#include <stdlib.h>

class deinterleaver_rand: public coDec
{
    int            interleaverSize;
    int            rowOut;
    int            colOut;
    int            bufSize;
    int            nbrChar;
    unsigned char*buffer;
    //unsigned char**touched;
    int*           table;
    unsigned charcharIn;
    unsigned charcharOut;

    unsigned int SBin; // buffer to build symbols (8bits->mbits)
    int SBinBits; //number of bits in the symbol buffer
    int padBits;
    int realBits;
}

```

```
    unsigned int SBout; // buffer to build symbols (8bits->mbits)
    int SBoutBits; //number of bits in the symbol buffer
    int lastWrite;

    int bytesToOutput;
    int nbrBytesOutput;
    int remainingBits;

    int inputCount;
    int outputCount;

    public:
    deinterleaver_rand( int Size );

    int processData();
    int initTable();
    int printTable();

    void newData();
};

#endif

// deinterleaver_rand.cc
#include "deinterleaver_rand.h"

deinterleaver_rand::deinterleaver_rand(int Size):
    interleaverSize(Size),inputCount(0),
    outputCount(0), charIn(0), bufSize(0), rowOut(0),
    colOut(0),
    nbrChar(0), SBin(0), SBinBits(0), SBout(0), SBoutBits(0)
{
    if (Size<=0)
    {
        cerr << "The size of the interleaver is negative or null!!!" <<
endl;
        exit(-1);
    }

    bufSize=Size;
    bytesToOutput=bufSize/8;
    remainingBits=bufSize%8;

    buffer=new unsigned char[Size];
    table=new int[Size];

    if (!buffer || !table)
    {
        cerr << "Couldn't allocate memory for the interleaver of size "
<< Size << endl;
    }
}
```

```
        exit(-2);
    }

    for (int i=0;i<Size;i++)
    {
        buffer[i]=0;
        table[i]=0;
    }

    srand48(0);
    initTable();
}

int deinterleaver_rand::processData()
{
    int i=0,j=0,colIn,rowIn,bitCount,index;

    startProcess:

    if ( (eof=inBuffer->atEOF()) && inputCount==0 && outputCount==0)
    {
        outBuffer->setEOF();
        eof=1;
        return eof;
    }

    while ( (inputCount<bufSize) &&
            (inBuffer->testRead() || (eof=inBuffer->atEOF()))) )
    {
        if (SBinBits==0)
        {
            if (eof)
                charIn=0;
            else
                inBuffer->get(charIn);
            SBinBits+=8;
        }

        while (SBinBits>0 && inputCount<bufSize)
        {
            index=table[inputCount];

            if ( (charIn&(0x1<<(SBinBits-1))) )
                buffer[index]=1;
            else
                buffer[index]=0;
            inputCount++;
            SBinBits--;
        }
        //cout<< "coucou" << endl;
    }
}
```

```

    }

    while ( (inputCount>=bufSize) &&
            (outputCount<bufSize) && (outBuffer->testWrite()) )
    {
        while (SBoutBits<8 && outputCount<bufSize)
        {
            SBout=(SBout<<1)+buffer[outputCount];
            outputCount++;
            SBoutBits++;
            //cout<< "Gutten tag" << endl;
        }

        if (SBoutBits>=8)
        {
            SBoutBits-=8;
            charOut=SBout>>SBoutBits;
            outBuffer->put(charOut);
            SBout^=charOut<<SBoutBits;
            //cout<< "bonjour" << endl;
        }
        else
        if (SBoutBits<8 && eof)
        {
            charOut=SBout<<(8-SBoutBits);
            outBuffer->put(charOut);
            SBoutBits=0;
            //cout<< "hello" << endl;
        }
    }

    if ( inputCount>=bufSize && outputCount>=bufSize )
    {
        inputCount=0;
        outputCount=0;
        charIn=0;

        eof=inBuffer->atEOF();
        initTable();
        goto startProcess;
    }

    return 1;
}

void deinterleaver_rand::newData()
{
    coDec::newData();
}

```

```

        inputCount=0;
        outputCount=0;
        charIn=0;
        for (int i=0;i<interleaverSize;i++)
            buffer[i]=0;
    }

int deinterleaver_rand::initTable()
{
    int          i,j;
    int          index;
    int          rowOut,colOut;
    int          row,col;

    for (i=0;i<interleaverSize;i++)
        table[i]=-1;

    for (i=0;i<interleaverSize;i++)
    {
        //index=lrnd48()%interleaverSize;
        do
            index=lrnd48()%interleaverSize;
        while (table[index]!=-1);
        //cerr<< index << endl;

        table[index]=i;
    }
}

int
deinterleaver_rand::printTable()
{
    for (int i=0;i<interleaverSize;i++)
        cout << table[i] << " ";
    cout << endl;
    return 1;
}

```

A.10 interleaver8x

```

// interleaver8x.h
#ifndef _interleaver8x_h
#define _interleaver8x_h

#include "RS.h"
#include "coDec.h"

class interleaver8x: public coDec

```

```
{
    int                interleaverSize;
    int                rowOut;
    int                colOut;
    int                bufSizeinChar;
    int                nbrChar;
    unsigned char**buffer;
    unsigned charcharIn;

    int inputCount;
    int outputCount;

    public:
    interleaver8x( int Size );

    int processData();

    void newData();
};

#endif

// interleaver8x.cc
#include "interleaver8x.h"

interleaver8x::interleaver8x(int                               Size):
interleaverSize(Size),inputCount(0), outputCount(0),
                                charIn(0),  bufSizeinChar(0),
rowOut(0), colOut(0),
                                nbrChar(0)
{
    if (Size<=0)
    {
        cerr << "The size of the interleaver is negative or null!!!" <<
endl;
        exit(-1);
    }

    if ((Size%8)!=0)
    {
        cerr << "The size of the interleaver should be a multiple of 8,
" << Size;
        cerr << " is not a multiple of 8" << endl;
        exit(-3);
    }

    nbrChar=Size/8;
    bufSizeinChar=nbrChar*Size;

    buffer=new unsigned char*[Size];
```

```
        if (!buffer)
        {
            cerr << "Couldn't allocate memory for the interleaver of size "
<< Size << endl;
            exit(-2);
        }

        for (int i=0;i<Size;i++)
        {
            buffer[i]=new unsigned char[nbrChar];

            if (!buffer[i])
            {
                cerr << "Couldn't allocate memory for the interleaver of
size " << Size << endl;
                exit(-2);
            }
        }

        for (int i=0;i<Size;i++)
            for (int j=0;j<nbrChar;j++)
                buffer[i][j]=0;
    }

    int interleaver8x::processData()
    {
        int i=0,j=0,colIn,rowIn,bitCount;

        startProcess:

        if ( (outputCount>=bufSizeinChar && inBuffer->atEOF()) || eof)
        {
            outBuffer->setEOF();
            eof=1;
            return eof;
        }

        while (!eof)
        {
            while ( (inputCount<bufSizeinChar) && (inBuffer->testRead()) )
            {
                inBuffer->get(charIn);

                rowIn=inputCount/nbrChar;
                colIn=(inputCount%nbrChar)<<3;

                for (bitCount=7;bitCount>=0;bitCount--)
                {
                    rowOut=colIn;
```

```

        colOut=rowIn;
        buffer[rowOut][colOut/8]+=( (charIn>>bitCount)&1 ) <<
(7-colOut%8);
        colIn++;
    }
    inputCount++;
}

/*
zero    The buffer used to be reversed so had to push the data as if
in      now the data are put at the right place right away not pushed
        if ( (inputCount<bufSizeinChar) && (eof=inBuffer->atEOF()) )
        {
            for (i=0;i<8;i++)
                buffer[i]<=(8-inputCount);
            inputCount=8;
        }
        */

        while ( (inputCount>=bufSizeinChar || inBuffer->atEOF()) &&
(outputCount<bufSizeinChar) &&
(outBuffer->testWrite()) )
            outBuffer->put(buffer[outputCount/
nbrChar][ (outputCount++)%nbrChar] );

        if ( inputCount>=bufSizeinChar && outputCount>=bufSizeinChar )
        {
            inputCount=0;
            outputCount=0;
            charIn=0;

            for (i=0;i<interleaverSize;i++)
                for (j=0;j<nbrChar;j++)
                    buffer[i][j]=0;

            eof=inBuffer->atEOF();
        }

        if (outputCount>=bufSizeinChar && inBuffer->atEOF())
        {
            outBuffer->setEOF();
            eof=1;
            return eof;
        }

        return 1;
    }
    return 1;

```

```

}

void interleaver8x::newData()
{
    coDec::newData();

    inputCount=0;
    outputCount=0;
    charIn=0;
    for (int i=0;i<interleaverSize;i++)
        for (int j=0;j<nbrChar;j++)
            buffer[i][j]=0;
}

```

A.11 convolEncoder

```

// convolEncoder.h
#ifndef _convolEncoder_h
#define _convolEncoder_h

#include "coDec.h"

// #define _debug_

class convolEncoder: public coDec
{
    unsigned char charIn;
    unsigned char redundBits;
    int          redundCount;
    unsigned int FSR;

    int          _n0; // block length
    int          _k0; // information length
    int          _N;  // Constraint length
    unsigned int generator; // polynomial generator
    unsigned int mask; // mask to clear the output redundant bit from the
FSR

    int inputCount;
    int outputCount;
    char lastDataIn;
    char lastDataOut;

    unsigned char tempChar;

public:
    convolEncoder(int n0, int k0, int N);

```

```

        int processData();

        void newData();
    };

#endif

// convolEncoder.cc
#include "convolEncoder.h"
#include "debugger.h"
#include "debug.h"

convolEncoder::convolEncoder(int n0, int k0, int N):
    _n0(n0), _k0(k0), _N(N),
    charIn(0), redundBits(0), FSR(0), inputCount(0),
    outputCount(0), lastDataOut(0), lastDataIn(0),
    tempChar(0)
{
    // generator polynomials list for (2,1) self-orthogonal codes
    // The index is the constraint length
    // if the generator is 0, it is not defined
    static unsigned int
    generators_list[]={0,0,0x3,0,0,0,0,0,0x53,0,0,0,0,0,0,0,0,0,0x28413};

    if (_n0<=0 || _k0<=0 || _N<=0)
    {
        cerr << "The Convolutional code parameters have to be positive"
              << endl;
        exit(-1);
    }

    if (_N>18 || generators_list[_N]==0 || n0!=2 || k0!=1)
    {
        cerr << "This implementation only supports (2,1) self-
        orthogonal codes with constraint length 2, 7 or 18" << endl;
        exit(-2);
    }
    generator=generators_list[_N];
    mask=(1<<_N)-1;
#ifdef _debug_
    cout << "mask:";
    binaire(mask,cout,32);
    cout << endl;
    cout << "generator:";
    binaire(generator,cout,32);
    cout << endl;
#endif
}

int convolEncoder::processData()

```

```

{
    int                i=0;
    int                inNbrBits;

    startProcess:
    eof=inBuffer->atEOF();

    if ( (lastDataOut) && (eof) && (inputCount==0) && (outputCount==0) )
    {
        outBuffer->setEOF();
        return eof;
    }

    // This loop does atomic input, processing and output.
    while ( (inputCount==0) &&
            ( (inBuffer->testRead()) || (lastDataOut<=_N &&
eof) ) &&
            (outBuffer->testWrite()>=2) )
    {
        redundBits=0;

        if (lastDataOut<=_N && eof)
        {
            charIn=0;
            lastDataOut+=8;
        }
        else
        {
            inNbrBits=inBuffer->get(charIn);
            if (inNbrBits!=8)
            {
                lastDataOut+=(8-inNbrBits);
                eof=1;
            }
        }
        #ifdef _debug_
            cout << endl << "data:" << hex << (int)charIn << endl;
        #endif
        outBuffer->put(charIn);
        for (i=7;i>=0;i--)
        {
            FSR^=generator*((charIn>>i)&1);
            #ifdef _debug_
                cout << "FSR=";
                binaire(FSR,cout,32);
                cout << endl;
                cout << "input bit:" << hex << (int)(charIn) << endl;
            #endif
            FSR<<=1;
            redundBits<<=1;
            redundBits+=(FSR>>_N);
        }
    }
}

```

```

        #ifdef _debug_
        cout << "redundBits=";
        binaire(redundBits,cout);
        cout << endl;
        #endif
        FSR&=mask;
    }
    outBuffer->put(redundBits);
}

// The two following loops execute the previous loop breaking it
// down into
// input loop, processing loop, and output loop so that if the
// output is
// not ready, the processing and input can still happen. This is
// just
// a minor tweaking though

/*while ( (inputCount<1) && ((inBuffer->testRead())>0) ||
(!lastDataOut && eof)) )
{
    if (!lastDataOut && eof)
    {
        charIn=0;
        lastDataOut=1;
    }
    else
        inBuffer->get(charIn);
    #ifdef _debug_
    cout << "data:" << hex << (int)charIn << endl;
    #endif
    inputCount++;
}

while ( (inputCount>=1) && (outputCount<2) && (outBuffer-
>testWrite()) )
{
    switch (outputCount)
    {
        case 0:
            outBuffer->put(charIn);
            break;
        case 1:
            redundBits=0;

            for (i=0;i<8;i++)
            {
                FSR^=generator*(charIn&1);
                FSR<<=1;
                redundBits=redundBits|((FSR>>_N)<<i);
                #ifdef _debug_

```

```

        cout << "redundBits=";
        binaire(redundBits,cout);
        cout << endl;
        #endif
        FSR&=0x7F;
        charIn>=1;
    }
    outBuffer->put(redundBits);
    break;
}
    outputCount++;
}*/

if ( (inputCount>=1) && (outputCount>=2) )
{
    inputCount=0;
    outputCount=0;

    goto startProcess;
}

eof=inBuffer->atEOF();

if ( (lastDataOut) && (eof) && (inputCount==0) && (outputCount==0) )
{
    outBuffer->setEOF();
    return eof;
}
}

void convolEncoder::newData()
{
    coDec::newData();

    charIn=0;
    redundBits=0;
    FSR=0;
    inputCount=0;
    outputCount=0;
    lastDataOut=0;
    lastDataIn=0;
}

```

A.12 convolDecoder

```

// convolDecoder.h
#ifndef _convolDecoder_h
#define _convolDecoder_h

```

```

#include "coDec.h"
// #define _debug_

class convolDecoder: public coDec
{
    unsigned char infoBits;
    unsigned char redundBits;
    unsigned char infoBit;
    unsigned int fifo;
    unsigned char outFifo;
    unsigned int FSR1;
    unsigned char outFSR1;
    unsigned int FSR2;
    unsigned int feedback;

    int          _n0; // block length
    int          _k0; // information length
    int          _N;  // Constraint length
    unsigned int generator; // polynomial generator
    unsigned int feedbackVector; // feedback vector of the syndromes
    unsigned int mask; // mask to clear the output redundant bit from the
FSR
    unsigned int majorityThreshold;
    unsigned int syndromes;
    unsigned int syndrCount;
    unsigned int syndrMask;

    unsigned int charOut;
    int bitsOut;

    int inputCount;
    int outputCount;

public:
    convolDecoder(int n0, int k0, int N);

    int processData();

    void newData();
};

#endif

// convolDecoder.cc
#include "convolDecoder.h"
#include "debugger.h"
#include "debug.h"

convolDecoder::convolDecoder(int n0, int k0, int N):
    _n0(n0), _k0(k0), _N(N),

```

```

        infoBits(0), redundBits(0), fifo(0), FSR1(0), FSR2(0),
        feedback(0), charOut(0), inputCount(0), outputCount(0),
        infoBit(0), bitsOut(1-N)
    {
        // generator polynomials list for (2,1) self-orthogonal codes
        // The index is the constraint length
        // if the generator is 0, it is not defined
        static unsigned int generators_list[] =
            {0,0,0x3,0,0,0,0,0x53,
0,0,0,0,0,0,0,0,0,0,0,0,0x28413};
        static unsigned int feedback_vector_list[] =
            {0,0,0x1,0,0,0,0,0x13,
0,0,0,0,0,0,0,0,0,0,0,0,0x18413};

        if (_n0<=0 || _k0<=0 || _N<=0)
        {
            cerr << "The Convolutional code parameters have to be positive"
                << endl;
            exit(-1);
        }

        if (_N>18 || generators_list[_N]==0 || n0!=2 || k0!=1)
        {
            cerr << "This implementation only supports (2,1) self-
orthogonal codes with constraint length 2, 7 or 18" << endl;
            exit(-2);
        }
        generator=generators_list[_N];
        feedbackVector=feedback_vector_list[_N];
        mask=(1<<_N)-1;
        unsigned int thresholdMask=1;
        majorityThreshold=0;
        for (int i=0; i<_N; i++)
        {
            if ((generator&thresholdMask)!=0)
                majorityThreshold++;
            thresholdMask<<=1;
        }
        majorityThreshold>>=1;
#ifdef _debug_
        cout << "majority threshold: " << majorityThreshold << endl;
#endif
    }

    int convolDecoder::processData()
    {
        int i=0;
        int pos=0;

```

```

int temp;
int nbrToRead=0;

startProcess:
//if ( (inputCount<2) && ((bitsOut-_N+1)<=0) && (eof) )
if ( (inputCount<2) && (eof) )
{
    outBuffer->setEOF();
    return eof;
}

while ( (nbrToRead=inBuffer->testRead()) &&
        (outBuffer->testWrite()) )
{
    // Those two test are there to handle corrupt files with odd
number    // of charaters.
    if (nbrToRead==1 && inputCount==0)
    {
        inBuffer->get(infoBits);
        break;
    }

    if (nbrToRead==1 && inputCount==1)
    {
        inBuffer->get(redundBits);
        break;
    }
    inBuffer->get(infoBits);
    inBuffer->get(redundBits);
#ifdef _debug_
    cout << endl << endl << "infoBits=";
    binaire(infoBits,cout);
    cout << endl;
    cout << "redundBits=";
    binaire(redundBits,cout);
    cout << endl;
    cout << "data:" << hex << (int)infoBits << endl;
#endif
    for (pos=7;pos>=0;pos--)
    {
        infoBit=(infoBits>>pos)&1;
        feedback=0;
        // fifo input & output
        fifo+=infoBit;
#ifdef _debug_
        cout << "fifo=";
        binaire(fifo,cout,32);
        cout << endl;
#endif
        fifo<=1;

```

```

        outFifo=fifo>>_N;
        fifo&=mask;
#ifdef _debug_
        cout << "outFifo=";
        cout << "\t\t" << (int)outFifo;
        cout << endl;
#endif
        // upper FSR input & output
        FSR1^=generator*infoBit;
#ifdef _debug_
        cout << "FSR1=";
        binaire(FSR1,cout,32);
        cout << endl;
#endif
        FSR1<<=1;
        outFSR1=FSR1>>_N;
#ifdef _debug_
        cout << "outFSR1=";
        cout << (int)outFSR1;
        cout << endl;
        cout << "redundant bit=";
        cout << (int)(redundBits&1);
        cout << endl;
#endif
        FSR1&=mask;
        // lower FSR input & output
#ifdef _debug_
        cout << "FSR2=";
        binaire(FSR2,cout);
        cout << endl;
#endif
        FSR2+=((redundBits>>pos)&1)^outFSR1;
#ifdef _debug_
        cout << "FSR2=";
        binaire(FSR2,cout);
        cout << endl;
#endif
        syndromes=FSR2&generator;
        syndrCount=0;
        syndrMask=1;
        for (i=0;i<_N;i++)
        {
            if ((syndromes&syndrMask)!=0)
                syndrCount++;
            syndrMask<<=1;
        }
#ifdef _debug_
        cout << "syndrome count: " << syndrCount << endl;
#endif
        if (syndrCount>majorityThreshold)
            feedback=1;

```

```

        FSR2^=feedbackVector*feedback;
        FSR2<<=1;
        FSR2&=mask;
        if (bitsOut>=0)
        {
            charOut<<=1;
            charOut+=(outFifo^feedback);
#ifdef _debug_
            cout << "charOut=";
            binaire(charOut,cout,32);
            cout << endl;
#endif
        }
        bitsOut++;
#ifdef _debug_
        cout << "original bit:" << (int)outFifo << endl;
        cout << "correction:" << (int)feedback << endl;
        cout << "after correction:" << (int)(outFifo^feedback) <<
endl;
#endif
    }
    if (bitsOut>=8)
    {
        bitsOut-=8;
        outBuffer->put(charOut>>bitsOut);
#ifdef _debug_
        cout << "output:";
        binaire(charOut&0xFF,cout);
        cout << endl;
        cout << "output:" << hex << (charOut>>bitsOut) << endl;
#endif
        //charOut&=(1<<(bitsOut+1)-1);
    }
}

// This is to deal with the case when there is not an even number
// of characters in the input file so program does not loop forever
eof=inBuffer->atEOF();

if ( (inputCount>=2) && (outputCount>=1) )
{
    inputCount=0;
    outputCount=0;

    goto startProcess;
}

}

void convolDecoder::newData()
{
    coDec::newData();
}

```

```

    infoBits=0;
    redundBits=0;
    fifo=0;
    FSR1=0;
    FSR2=0;
    feedback=0;
    charOut=0;
    inputCount=0;
    outputCount=0;
    infoBit=0;
    bitsOut=1-_N;
}

```

A.13 Interleav_symb

```

// Interleav_symb.h
#ifndef _interleav_symb_h
#define _interleav_symb_h

#include "coDec.h"

class interleav_symb: public coDec
{
    // symbol Size
    int m;
    // interleaver size=n*n; interleaverSize=n
    int interleaverSize;
    int rowOut;
    int colOut;
    // n*n
    int bufSizeinChar;
    unsigned char**symbBuffer;

    unsigned long int SBin;// buffer to build symbols (8bits->mbits)
    int SBinBits;//number of bits in the symbol buffer
    unsigned long int SBout;// buffer to build symbols (8bits->mbits)
    int SBoutBits;//number of bits in the symbol buffer

    int inputCount;
    int outputCount;

public:
    interleav_symb( int symblSize, int Size );

    int processData();

    int newData();

```

```
};

#endif

// Interleav_symb.cc
#include "interleav_symb.h"

interleav_symb::interleav_symb(int symbolSize,int Size):
    interleaverSize(Size),inputCount(0),
    outputCount(0),
    bufSizeinChar(0), rowOut(0), colOut(0),
    SBin(0), SBinBits(0),SBout(0),SBoutBits(0),
    m(symbolSize)
{
    int nbrBits;

    if (Size<=0 || symbolSize<=0)
    {
        cerr << "The size of the interleaver or the symbols is negative
or null!!!"
        << endl;
        exit(-1);
    }

    bufSizeinChar=Size*Size;

    symbBuffer=new unsigned char*[Size];

    if (!symbBuffer)
    {
        cerr << "Couldn't allocate memory for the interleaver of size "
        << Size << endl;
        exit(-2);
    }

    for (int i=0;i<Size;i++)
    {
        symbBuffer[i]=new unsigned char[Size];

        if (!symbBuffer[i])
        {
            cerr << "Couldn't allocate memory for the interleaver of
size " << Size << endl;
            exit(-2);
        }
    }

    for (int i=0;i<Size;i++)
        for (int j=0;j<Size;j++)
            symbBuffer[i][j]=0;
```

```

}

int interleav_symb::processData()
{
    static int          symbol=0;
    static unsigned long intcharBuffer=0, output=0;
    static unsigned char tempChar=0;

    int i=0,j=0,colIn,rowIn,bitCount;

    if (!inBuffer || !outBuffer)
    {
        cerr << "Please assign me with some input and output "
              << "buffers before asking me to process data" <<
endl;
        return -1;
    }

    dataInput_Process:
    eof=inBuffer->atEOF();
    if ( inputCount==0 && outputCount==0 && SBinBits==0 && SBoutBits<=0
&& eof )
    {
        outBuffer->setEOF();
        return 1;
    }

    while ( (inputCount<bufSizeinChar) && ( inBuffer->testRead() || eof
) )
    {
        if (SBinBits<m)
        {
            /* you have no more data to give and the rest is padded to zero
automatically
when the buffer is originally initialized */
            if (eof)
            {
                SBin=SBin<<(m-SBinBits);
                SBinBits=m;
                inputCount=bufSizeinChar-1;
            }
            else
            {
                inBuffer->get(tempChar);
                SBin=(SBin<<8)+tempChar;
                SBinBits+=8;
            }
        }

        while (SBinBits>=m)
        {

```

```

        if (inputCount>=bufSizeinChar)
            break;
        // extract symbol from symbol buffer (SBin):
        SBinBits-=m;
        symbol=SBin>>SBinBits;
        // do your symbol interleaving here...
        rowIn=inputCount/interleaverSize;
        colIn=inputCount%interleaverSize;
        rowOut=colIn;
        colOut=rowIn;
        symbBuffer[rowOut][colOut]=symbol;

        SBin^=symbol<<SBinBits;
        inputCount++;
    }
}

while ( inputCount>=bufSizeinChar && outputCount<bufSizeinChar &&
(outBuffer->testWrite())) )
{
    while (SBoutBits<8)
    {
        if (outputCount>=bufSizeinChar)
            break;
        rowOut=outputCount/interleaverSize;
        colOut=outputCount%interleaverSize;
        SBout=(SBout<<m)+symbBuffer[rowOut][colOut];
        SBoutBits+=m;
        outputCount++;
    }

    if (SBoutBits>=8)
    {
        SBoutBits-=8;
        output= (char)(SBout>>SBoutBits);
        outBuffer->put((unsigned char)output);
        SBout^=output<<SBoutBits;
    }
}
// after this stage SBoutBits is strictly less than 8

// if at end of file and still some bits in SBout, output them
if ( (inputCount>=bufSizeinChar) && (outputCount>=bufSizeinChar)
    && (SBinBits==0) && (SBoutBits>0) && (eof) )
{
    if (outBuffer->testWrite())
    {
        output=(char)(SBout<<(8-SBoutBits));
        outBuffer->put((unsigned char)output);
        SBoutBits=0;
    }
}

```

```

    }
}

if ( inputCount>=bufSizeinChar && outputCount>=bufSizeinChar )
{
    inputCount=0;
    outputCount=0;

    for (i=0;i<interleaverSize;i++)
        for (j=0;j<interleaverSize;j++)
            symbBuffer[i][j]=0;

    eof=inBuffer->atEOF();
    goto dataInput_Process;
}

return 1;
}

interleav_symb::newData()
{
    int      i,j;

    if (inputCount!=0 && (!eof) )
        cerr << "Warning from Reed Solomon encoder: new data"
              << " requested while the previous data were still "
              << "being processed." << endl;

    coDec::newData();

    SBin=0;
    SBinBits=0;
    SBout=0;
    SBoutBits=0;
    inputCount=0;
    outputCount=0;

    for (int i=0;i<interleaverSize;i++)
        for (int j=0;j<interleaverSize;j++)
            symbBuffer[i][j]=0;
}

```

A.14 random_test

```

#include "ran4.h"
#ifdef __cplusplus
extern "C" {

```

```

#endif /* __cplusplus */

#define NITER 4
#define NBRIT 100000

long idums=0,idum=1;

void psdes(unsigned long *lword, unsigned long *irword)
{
    unsigned long i,ia,ib,iswap,itmph=0,itmpl=0;
    static unsigned long c1[NITER]={
                                                0xbaa96887L,
                                                0x1e17d32cL,
                                                0x03bcd3cL,
                                                0x0f33d1b2L};

    static unsigned long c2[NITER]={
                                                0x4b0f3b58L,
                                                0xe874f0c3L,
                                                0x6955c5a6L,
                                                0x55a7ca46L};

    for (i=0;i<NITER;i++) {
        ia=(iswap=(*irword)) ^ c1[i];
        itmpl = ia & 0xffff;
        itmph = ia >> 16;
        ib=itmpl*itmpl+ ~(itmph*itmph);
        *irword=(*lword) ^ (((ia = (ib >> 16) |
            ((ib & 0xffff) << 16)) ^ c2[i])+itmpl*itmph);
        *lword=iswap;
    }
}

void sran4(long seedval)
{
    idums=seedval;
    idum=1;
}

unsigned long nran4(long* state0,long* statel)
{
    //void psdes(unsigned long *lword, unsigned long
    *irword);
    unsigned long irword,itemp,lword;
    //extern long idums,idum;
    static unsigned long jflone = 0x3f800000;
    static unsigned long jflmsk = 0x007fffff;

    irword=(*statel);
    lword=(*state0);
    psdes(&lword,&irword);
    ++(*statel);
}

```

```
        return irword;
    }

unsigned long lran4()
{
    //void                psdes(unsigned long *lword, unsigned long
    *irword);
    unsigned long irword, itemp, lword;
    //extern long idums, idum;
    static unsigned long jflone = 0x3f800000;
    static unsigned long jflmsk = 0x007fffff;

    if (idum < 0) {
        idums = -idum;
        idum=1;
    }
    irword=idum;
    lword=idums;
    psdes(&lword, &irword);
    ++idum;
    return irword;
}

float fran4()
{
    //void                psdes(unsigned long *lword, unsigned long
    *irword);
    unsigned long irword, itemp, lword;
    //static long idums = 0;
    static unsigned long jflone = 0x3f800000;
    static unsigned long jflmsk = 0x007fffff;

    irword=idum;
    lword=idums;
    psdes(&lword, &irword);
    itemp=jflone | (jflmsk & irword);
    ++idum;
    return (*(float *)&itemp)-1.0;
}

float eran4(long* state0, long* state1)
{
    //void                psdes(unsigned long *lword, unsigned long
    *irword);
    unsigned long irword, itemp, lword;
    //extern long idums, idum;
    static unsigned long jflone = 0x3f800000;
    static unsigned long jflmsk = 0x007fffff;

    irword=(*state1);
    lword=(*state0);
```

```

        psdes(&lword,&irword);
        itemp=jflone | (jflmsk & irword);
        ++(*state1);
        return (*(float *)&itemp)-1.0;
    }

#ifdef __cplusplus
}
#endif /* __cplusplus */

/*
main(int argc, char** argv)
{
    long int      number;
    unsigned longresult;
    float         result2;
    double        count[3];
    double        countup8[8];
    double        xsq;
    int           i;

    count[0]=0;
    count[1]=0;
    count[2]=0;
    for (i=0;i<8;i++)
        countup8[i]=0;
    number=-1;
    dran4(&number);
    number=1;
    while(number<(NBRIT+1))
    {
        result=dran4(&number);
        count[result%3]++;
        countup8[result>>29]++;
    }
    printf("%.0f %.0f %.0f\n",count[0],count[1],count[2]);
    xsq=(count[0]-NBRIT/3)*(count[0]-NBRIT/3)/(NBRIT/3)+(count[1]-
NBRIT/3)*(count[1]-NBRIT/3)/(NBRIT/3)+(count[2]-NBRIT/3)*(count[2]-
NBRIT/3)/(NBRIT/3);
    printf("%.20f\n",xsq);
    xsq=0;
    for (i=0;i<8;i++)
        xsq+=(countup8[i]-NBRIT/8)*(countup8[i]-NBRIT/8)/(NBRIT/8);
    printf("%.20f\n",xsq);
}
*/

```

A.15 gasdev

```
// gasdev.cc
#include <stdio.h>
#include <iostream.h>
#ifdef _unix_
    #include <stdlib.h>
#else
    #include "rand48.h"
#endif
#include "gasdev.h"
#include "ran4.h"
#include <math.h>
#include <time.h>

float gasdev()
{
    static int iset=0;
    static float gset;
    static long number1=1,number2=100000000;
    float fac,r,v1,v2;

    if (iset == 0)
    {
        do
        {
            v1=2.0*fran4()-1.0;
            v2=2.0*fran4()-1.0;
            r=v1*v1+v2*v2;
        }
        while (r >= 1.0);
        //cout<< r << endl;

        fac=sqrt(-2.0*log(r)/r);
        gset=v1*fac;
        iset=1;
        return v2*fac;
    }
    else
    {
        iset=0;
        return gset;
    }
}

void initSeed()
{
    long int randSeed;
```

```
        time(&randSeed);
        //sran4(0);
        sran4(randSeed);
    }

float gasdev48()
{
    static int iset=0;
    static float gset;
    float fac,r,v1,v2;

    if (iset == 0)
    {
        do
        {
            v1=2.0*drand48()-1.0;
            v2=2.0*drand48()-1.0;
            r=v1*v1+v2*v2;
        }
        while (r >= 1.0);

        fac=sqrt(-2.0*log(r)/r);
        gset=v1*fac;
        iset=1;
        return v2*fac;
    }
    else
    {
        iset=0;
        return gset;
    }
}

void initSeed48()
{
    unsigned long int randSeed;
    time((long int*)&randSeed);
    srand48(randSeed);
}
```

A.16 factor

```
#ifndef _factor_h
#define _factor_h

int factor(unsigned long number, unsigned int factors[], unsigned
maxNbrFactor);
```

```

#endif

#include <iostream.h>
#include <stdio.h>
#include "factor.h"

int factor(unsigned long number, unsigned int factors[], unsigned
maxNbrFactor)
{
    int primeTableIndex=0, factorIndex=0;
    static unsigned primeTable[] =
    {
        2, 3, 5, 7, 11, 13, 17, 19, 23,
        29, 31, 37, 41, 43, 47, 53, 59, 61,
        67, 71, 73, 79, 83, 89, 97, 101, 103,
        107, 109, 113, 127, 131, 137, 139, 149, 151,
        157, 163, 167, 173, 179, 181, 191, 193, 197,
        199, 211, 223, 227, 229, 233, 239, 241, 251,
        257, 263, 269, 271, 277, 281, 283, 293, 307,
        311, 313, 317, 331, 337, 347, 349, 353, 359,
        367, 373, 379, 383, 389, 397, 401, 409, 419,
        421, 431, 433, 439, 443, 449, 457, 461, 463,
        467, 479, 487, 491, 499, 503, 509, 521, 523,
        541, 547, 557, 563, 569, 571, 577, 587, 593,
        599, 601, 607, 613, 617, 619, 631, 641, 643,
        647, 653, 659, 661, 673, 677, 683, 691, 701,
        709, 719, 727, 733, 739, 743, 751, 757, 761,
        769, 773, 787, 797, 809, 811, 821, 823, 827,
        829, 839, 853, 857, 859, 863, 877, 881, 883,
        887, 907, 911, 919, 929, 937, 941, 947, 953,
        967, 971, 977, 983, 991, 997, 1009, 1013, 1019,
        1021, 1031, 1033, 1039, 1049, 1051, 1061, 1063, 1069,
        1087, 1091, 1093, 1097, 1103, 1109, 1117, 1123, 1129,
        1151, 1153, 1163, 1171, 1181, 1187, 1193, 1201, 1213,
        1217, 1223, 1229, 1231, 1237, 1249, 1259, 1277, 1279,
        1283, 1289, 1291, 1297, 1301, 1303, 1307, 1319, 1321,
        1327, 1361, 1367, 1373, 1381, 1399, 1409, 1423, 1427,
        1429, 1433, 1439, 1447, 1451, 1453, 1459, 1471, 1481,
        1483, 1487, 1489, 1493, 1499, 1511, 1523, 1531, 1543,
        1549, 1553, 1559, 1567, 1571, 1579, 1583, 1597, 1601,
        1607, 1609, 1613, 1619, 1621, 1627, 1637, 1657, 1663,
        1667, 1669, 1693, 1697, 1699, 1709, 1721, 1723, 1733,
        1741, 1747, 1753, 1759, 1777, 1783, 1787, 1789, 1801,
        1811, 1823, 1831, 1847, 1861, 1867, 1871, 1873, 1877,
        1879, 1889, 1901, 1907, 1913, 1931, 1933, 1949, 1951,
        1973, 1979, 1987, 1993, 1997, 1999, 2003, 2011, 2017,
        2027, 2029, 2039, 2053, 2063, 2069, 2081, 2083, 2087,
        2089, 2099, 2111, 2113, 2129, 2131, 2137, 2141, 2143,
        2153, 2161, 2179, 2203, 2207, 2213, 2221, 2237, 2239,
    }
}

```

2243, 2251, 2267, 2269, 2273, 2281, 2287, 2293, 2297,
2309, 2311, 2333, 2339, 2341, 2347, 2351, 2357, 2371,
2377, 2381, 2383, 2389, 2393, 2399, 2411, 2417, 2423,
2437, 2441, 2447, 2459, 2467, 2473, 2477, 2503, 2521,
2531, 2539, 2543, 2549, 2551, 2557, 2579, 2591, 2593,
2609, 2617, 2621, 2633, 2647, 2657, 2659, 2663, 2671,
2677, 2683, 2687, 2689, 2693, 2699, 2707, 2711, 2713,
2719, 2729, 2731, 2741, 2749, 2753, 2767, 2777, 2789,
2791, 2797, 2801, 2803, 2819, 2833, 2837, 2843, 2851,
2857, 2861, 2879, 2887, 2897, 2903, 2909, 2917, 2927,
2939, 2953, 2957, 2963, 2969, 2971, 2999, 3001, 3011,
3019, 3023, 3037, 3041, 3049, 3061, 3067, 3079, 3083,
3089, 3109, 3119, 3121, 3137, 3163, 3167, 3169, 3181,
3187, 3191, 3203, 3209, 3217, 3221, 3229, 3251, 3253,
3257, 3259, 3271, 3299, 3301, 3307, 3313, 3319, 3323,
3329, 3331, 3343, 3347, 3359, 3361, 3371, 3373, 3389,
3391, 3407, 3413, 3433, 3449, 3457, 3461, 3463, 3467,
3469, 3491, 3499, 3511, 3517, 3527, 3529, 3533, 3539,
3541, 3547, 3557, 3559, 3571, 3581, 3583, 3593, 3607,
3613, 3617, 3623, 3631, 3637, 3643, 3659, 3671, 3673,
3677, 3691, 3697, 3701, 3709, 3719, 3727, 3733, 3739,
3761, 3767, 3769, 3779, 3793, 3797, 3803, 3821, 3823,
3833, 3847, 3851, 3853, 3863, 3877, 3881, 3889, 3907,
3911, 3917, 3919, 3923, 3929, 3931, 3943, 3947, 3967,
3989, 4001, 4003, 4007, 4013, 4019, 4021, 4027, 4049,
4051, 4057, 4073, 4079, 4091, 4093, 4099, 4111, 4127,
4129, 4133, 4139, 4153, 4157, 4159, 4177, 4201, 4211,
4217, 4219, 4229, 4231, 4241, 4243, 4253, 4259, 4261,
4271, 4273, 4283, 4289, 4297, 4327, 4337, 4339, 4349,
4357, 4363, 4373, 4391, 4397, 4409, 4421, 4423, 4441,
4447, 4451, 4457, 4463, 4481, 4483, 4493, 4507, 4513,
4517, 4519, 4523, 4547, 4549, 4561, 4567, 4583, 4591,
4597, 4603, 4621, 4637, 4639, 4643, 4649, 4651, 4657,
4663, 4673, 4679, 4691, 4703, 4721, 4723, 4729, 4733,
4751, 4759, 4783, 4787, 4789, 4793, 4799, 4801, 4813,
4817, 4831, 4861, 4871, 4877, 4889, 4903, 4909, 4919,
4931, 4933, 4937, 4943, 4951, 4957, 4967, 4969, 4973,
4987, 4993, 4999, 5003, 5009, 5011, 5021, 5023, 5039,
5051, 5059, 5077, 5081, 5087, 5099, 5101, 5107, 5113,
5119, 5147, 5153, 5167, 5171, 5179, 5189, 5197, 5209,
5227, 5231, 5233, 5237, 5261, 5273, 5279, 5281, 5297,
5303, 5309, 5323, 5333, 5347, 5351, 5381, 5387, 5393,
5399, 5407, 5413, 5417, 5419, 5431, 5437, 5441, 5443,
5449, 5471, 5477, 5479, 5483, 5501, 5503, 5507, 5519,
5521, 5527, 5531, 5557, 5563, 5569, 5573, 5581, 5591,
5623, 5639, 5641, 5647, 5651, 5653, 5657, 5659, 5669,
5683, 5689, 5693, 5701, 5711, 5717, 5737, 5741, 5743,
5749, 5779, 5783, 5791, 5801, 5807, 5813, 5821, 5827,
5839, 5843, 5849, 5851, 5857, 5861, 5867, 5869, 5879,
5881, 5897, 5903, 5923, 5927, 5939, 5953, 5981, 5987,
6007, 6011, 6029, 6037, 6043, 6047, 6053, 6067, 6073,

6079,6089,6091,6101,6113,6121,6131,6133,6143,
6151,6163,6173,6197,6199,6203,6211,6217,6221,
6229,6247,6257,6263,6269,6271,6277,6287,6299,
6301,6311,6317,6323,6329,6337,6343,6353,6359,
6361,6367,6373,6379,6389,6397,6421,6427,6449,
6451,6469,6473,6481,6491,6521,6529,6547,6551,
6553,6563,6569,6571,6577,6581,6599,6607,6619,
6637,6653,6659,6661,6673,6679,6689,6691,6701,
6703,6709,6719,6733,6737,6761,6763,6779,6781,
6791,6793,6803,6823,6827,6829,6833,6841,6857,
6863,6869,6871,6883,6899,6907,6911,6917,6947,
6949,6959,6961,6967,6971,6977,6983,6991,6997,
7001,7013,7019,7027,7039,7043,7057,7069,7079,
7103,7109,7121,7127,7129,7151,7159,7177,7187,
7193,7207,7211,7213,7219,7229,7237,7243,7247,
7253,7283,7297,7307,7309,7321,7331,7333,7349,
7351,7369,7393,7411,7417,7433,7451,7457,7459,
7477,7481,7487,7489,7499,7507,7517,7523,7529,
7537,7541,7547,7549,7559,7561,7573,7577,7583,
7589,7591,7603,7607,7621,7639,7643,7649,7669,
7673,7681,7687,7691,7699,7703,7717,7723,7727,
7741,7753,7757,7759,7789,7793,7817,7823,7829,
7841,7853,7867,7873,7877,7879,7883,7901,7907,
7919,7927,7933,7937,7949,7951,7963,7993,8009,
8011,8017,8039,8053,8059,8069,8081,8087,8089,
8093,8101,8111,8117,8123,8147,8161,8167,8171,
8179,8191,8209,8219,8221,8231,8233,8237,8243,
8263,8269,8273,8287,8291,8293,8297,8311,8317,
8329,8353,8363,8369,8377,8387,8389,8419,8423,
8429,8431,8443,8447,8461,8467,8501,8513,8521,
8527,8537,8539,8543,8563,8573,8581,8597,8599,
8609,8623,8627,8629,8641,8647,8663,8669,8677,
8681,8689,8693,8699,8707,8713,8719,8731,8737,
8741,8747,8753,8761,8779,8783,8803,8807,8819,
8821,8831,8837,8839,8849,8861,8863,8867,8887,
8893,8923,8929,8933,8941,8951,8963,8969,8971,
8999,9001,9007,9011,9013,9029,9041,9043,9049,
9059,9067,9091,9103,9109,9127,9133,9137,9151,
9157,9161,9173,9181,9187,9199,9203,9209,9221,
9227,9239,9241,9257,9277,9281,9283,9293,9311,
9319,9323,9337,9341,9343,9349,9371,9377,9391,
9397,9403,9413,9419,9421,9431,9433,9437,9439,
9461,9463,9467,9473,9479,9491,9497,9511,9521,
9533,9539,9547,9551,9587,9601,9613,9619,9623,
9629,9631,9643,9649,9661,9677,9679,9689,9697,
9719,9721,9733,9739,9743,9749,9767,9769,9781,
9787,9791,9803,9811,9817,9829,9833,9839,9851,
9857,9859,9871,9883,9887,9901,9907,9923,9929,
9931,9941,9949,9967,9973,10007,10009,10037,10039,
10061,10067,10069,10079,10091,10093,10099,10103,10111,
10133,10139,10141,10151,10159,10163,10169,10177,10181,

10193,10211,10223,10243,10247,10253,10259,10267,10271,
10273,10289,10301,10303,10313,10321,10331,10333,10337,
10343,10357,10369,10391,10399,10427,10429,10433,10453,
10457,10459,10463,10477,10487,10499,10501,10513,10529,
10531,10559,10567,10589,10597,10601,10607,10613,10627,
10631,10639,10651,10657,10663,10667,10687,10691,10709,
10711,10723,10729,10733,10739,10753,10771,10781,10789,
10799,10831,10837,10847,10853,10859,10861,10867,10883,
10889,10891,10903,10909,10937,10939,10949,10957,10973,
10979,10987,10993,11003,11027,11047,11057,11059,11069,
11071,11083,11087,11093,11113,11117,11119,11131,11149,
11159,11161,11171,11173,11177,11197,11213,11239,11243,
11251,11257,11261,11273,11279,11287,11299,11311,11317,
11321,11329,11351,11353,11369,11383,11393,11399,11411,
11423,11437,11443,11447,11467,11471,11483,11489,11491,
11497,11503,11519,11527,11549,11551,11579,11587,11593,
11597,11617,11621,11633,11657,11677,11681,11689,11699,
11701,11717,11719,11731,11743,11777,11779,11783,11789,
11801,11807,11813,11821,11827,11831,11833,11839,11863,
11867,11887,11897,11903,11909,11923,11927,11933,11939,
11941,11953,11959,11969,11971,11981,11987,12007,12011,
12037,12041,12043,12049,12071,12073,12097,12101,12107,
12109,12113,12119,12143,12149,12157,12161,12163,12197,
12203,12211,12227,12239,12241,12251,12253,12263,12269,
12277,12281,12289,12301,12323,12329,12343,12347,12373,
12377,12379,12391,12401,12409,12413,12421,12433,12437,
12451,12457,12473,12479,12487,12491,12497,12503,12511,
12517,12527,12539,12541,12547,12553,12569,12577,12583,
12589,12601,12611,12613,12619,12637,12641,12647,12653,
12659,12671,12689,12697,12703,12713,12721,12739,12743,
12757,12763,12781,12791,12799,12809,12821,12823,12829,
12841,12853,12889,12893,12899,12907,12911,12917,12919,
12923,12941,12953,12959,12967,12973,12979,12983,13001,
13003,13007,13009,13033,13037,13043,13049,13063,13093,
13099,13103,13109,13121,13127,13147,13151,13159,13163,
13171,13177,13183,13187,13217,13219,13229,13241,13249,
13259,13267,13291,13297,13309,13313,13327,13331,13337,
13339,13367,13381,13397,13399,13411,13417,13421,13441,
13451,13457,13463,13469,13477,13487,13499,13513,13523,
13537,13553,13567,13577,13591,13597,13613,13619,13627,
13633,13649,13669,13679,13681,13687,13691,13693,13697,
13709,13711,13721,13723,13729,13751,13757,13759,13763,
13781,13789,13799,13807,13829,13831,13841,13859,13873,
13877,13879,13883,13901,13903,13907,13913,13921,13931,
13933,13963,13967,13997,13999,14009,14011,14029,14033,
14051,14057,14071,14081,14083,14087,14107,14143,14149,
14153,14159,14173,14177,14197,14207,14221,14243,14249,
14251,14281,14293,14303,14321,14323,14327,14341,14347,
14369,14387,14389,14401,14407,14411,14419,14423,14431,
14437,14447,14449,14461,14479,14489,14503,14519,14533,
14537,14543,14549,14551,14557,14561,14563,14591,14593,

14621,14627,14629,14633,14639,14653,14657,14669,14683,
14699,14713,14717,14723,14731,14737,14741,14747,14753,
14759,14767,14771,14779,14783,14797,14813,14821,14827,
14831,14843,14851,14867,14869,14879,14887,14891,14897,
14923,14929,14939,14947,14951,14957,14969,14983,15013,
15017,15031,15053,15061,15073,15077,15083,15091,15101,
15107,15121,15131,15137,15139,15149,15161,15173,15187,
15193,15199,15217,15227,15233,15241,15259,15263,15269,
15271,15277,15287,15289,15299,15307,15313,15319,15329,
15331,15349,15359,15361,15373,15377,15383,15391,15401,
15413,15427,15439,15443,15451,15461,15467,15473,15493,
15497,15511,15527,15541,15551,15559,15569,15581,15583,
15601,15607,15619,15629,15641,15643,15647,15649,15661,
15667,15671,15679,15683,15727,15731,15733,15737,15739,
15749,15761,15767,15773,15787,15791,15797,15803,15809,
15817,15823,15859,15877,15881,15887,15889,15901,15907,
15913,15919,15923,15937,15959,15971,15973,15991,16001,
16007,16033,16057,16061,16063,16067,16069,16073,16087,
16091,16097,16103,16111,16127,16139,16141,16183,16187,
16189,16193,16217,16223,16229,16231,16249,16253,16267,
16273,16301,16319,16333,16339,16349,16361,16363,16369,
16381,16411,16417,16421,16427,16433,16447,16451,16453,
16477,16481,16487,16493,16519,16529,16547,16553,16561,
16567,16573,16603,16607,16619,16631,16633,16649,16651,
16657,16661,16673,16691,16693,16699,16703,16729,16741,
16747,16759,16763,16787,16811,16823,16829,16831,16843,
16871,16879,16883,16889,16901,16903,16921,16927,16931,
16937,16943,16963,16979,16981,16987,16993,17011,17021,
17027,17029,17033,17041,17047,17053,17077,17093,17099,
17107,17117,17123,17137,17159,17167,17183,17189,17191,
17203,17207,17209,17231,17239,17257,17291,17293,17299,
17317,17321,17327,17333,17341,17351,17359,17377,17383,
17387,17389,17393,17401,17417,17419,17431,17443,17449,
17467,17471,17477,17483,17489,17491,17497,17509,17519,
17539,17551,17569,17573,17579,17581,17597,17599,17609,
17623,17627,17657,17659,17669,17681,17683,17707,17713,
17729,17737,17747,17749,17761,17783,17789,17791,17807,
17827,17837,17839,17851,17863,17881,17891,17903,17909,
17911,17921,17923,17929,17939,17957,17959,17971,17977,
17981,17987,17989,18013,18041,18043,18047,18049,18059,
18061,18077,18089,18097,18119,18121,18127,18131,18133,
18143,18149,18169,18181,18191,18199,18211,18217,18223,
18229,18233,18251,18253,18257,18269,18287,18289,18301,
18307,18311,18313,18329,18341,18353,18367,18371,18379,
18397,18401,18413,18427,18433,18439,18443,18451,18457,
18461,18481,18493,18503,18517,18521,18523,18539,18541,
18553,18583,18587,18593,18617,18637,18661,18671,18679,
18691,18701,18713,18719,18731,18743,18749,18757,18773,
18787,18793,18797,18803,18839,18859,18869,18899,18911,
18913,18917,18919,18947,18959,18973,18979,19001,19009,
19013,19031,19037,19051,19069,19073,19079,19081,19087,

19121,19139,19141,19157,19163,19181,19183,19207,19211,
19213,19219,19231,19237,19249,19259,19267,19273,19289,
19301,19309,19319,19333,19373,19379,19381,19387,19391,
19403,19417,19421,19423,19427,19429,19433,19441,19447,
19457,19463,19469,19471,19477,19483,19489,19501,19507,
19531,19541,19543,19553,19559,19571,19577,19583,19597,
19603,19609,19661,19681,19687,19697,19699,19709,19717,
19727,19739,19751,19753,19759,19763,19777,19793,19801,
19813,19819,19841,19843,19853,19861,19867,19889,19891,
19913,19919,19927,19937,19949,19961,19963,19973,19979,
19991,19993,19997,20011,20021,20023,20029,20047,20051,
20063,20071,20089,20101,20107,20113,20117,20123,20129,
20143,20147,20149,20161,20173,20177,20183,20201,20219,
20231,20233,20249,20261,20269,20287,20297,20323,20327,
20333,20341,20347,20353,20357,20359,20369,20389,20393,
20399,20407,20411,20431,20441,20443,20477,20479,20483,
20507,20509,20521,20533,20543,20549,20551,20563,20593,
20599,20611,20627,20639,20641,20663,20681,20693,20707,
20717,20719,20731,20743,20747,20749,20753,20759,20771,
20773,20789,20807,20809,20849,20857,20873,20879,20887,
20897,20899,20903,20921,20929,20939,20947,20959,20963,
20981,20983,21001,21011,21013,21017,21019,21023,21031,
21059,21061,21067,21089,21101,21107,21121,21139,21143,
21149,21157,21163,21169,21179,21187,21191,21193,21211,
21221,21227,21247,21269,21277,21283,21313,21317,21319,
21323,21341,21347,21377,21379,21383,21391,21397,21401,
21407,21419,21433,21467,21481,21487,21491,21493,21499,
21503,21517,21521,21523,21529,21557,21559,21563,21569,
21577,21587,21589,21599,21601,21611,21613,21617,21647,
21649,21661,21673,21683,21701,21713,21727,21737,21739,
21751,21757,21767,21773,21787,21799,21803,21817,21821,
21839,21841,21851,21859,21863,21871,21881,21893,21911,
21929,21937,21943,21961,21977,21991,21997,22003,22013,
22027,22031,22037,22039,22051,22063,22067,22073,22079,
22091,22093,22109,22111,22123,22129,22133,22147,22153,
22157,22159,22171,22189,22193,22229,22247,22259,22271,
22273,22277,22279,22283,22291,22303,22307,22343,22349,
22367,22369,22381,22391,22397,22409,22433,22441,22447,
22453,22469,22481,22483,22501,22511,22531,22541,22543,
22549,22567,22571,22573,22613,22619,22621,22637,22639,
22643,22651,22669,22679,22691,22697,22699,22709,22717,
22721,22727,22739,22741,22751,22769,22777,22783,22787,
22807,22811,22817,22853,22859,22861,22871,22877,22901,
22907,22921,22937,22943,22961,22963,22973,22993,23003,
23011,23017,23021,23027,23029,23039,23041,23053,23057,
23059,23063,23071,23081,23087,23099,23117,23131,23143,
23159,23167,23173,23189,23197,23201,23203,23209,23227,
23251,23269,23279,23291,23293,23297,23311,23321,23327,
23333,23339,23357,23369,23371,23399,23417,23431,23447,
23459,23473,23497,23509,23531,23537,23539,23549,23557,
23561,23563,23567,23581,23593,23599,23603,23609,23623,

23627,23629,23633,23663,23669,23671,23677,23687,23689,
23719,23741,23743,23747,23753,23761,23767,23773,23789,
23801,23813,23819,23827,23831,23833,23857,23869,23873,
23879,23887,23893,23899,23909,23911,23917,23929,23957,
23971,23977,23981,23993,24001,24007,24019,24023,24029,
24043,24049,24061,24071,24077,24083,24091,24097,24103,
24107,24109,24113,24121,24133,24137,24151,24169,24179,
24181,24197,24203,24223,24229,24239,24247,24251,24281,
24317,24329,24337,24359,24371,24373,24379,24391,24407,
24413,24419,24421,24439,24443,24469,24473,24481,24499,
24509,24517,24527,24533,24547,24551,24571,24593,24611,
24623,24631,24659,24671,24677,24683,24691,24697,24709,
24733,24749,24763,24767,24781,24793,24799,24809,24821,
24841,24847,24851,24859,24877,24889,24907,24917,24919,
24923,24943,24953,24967,24971,24977,24979,24989,25013,
25031,25033,25037,25057,25073,25087,25097,25111,25117,
25121,25127,25147,25153,25163,25169,25171,25183,25189,
25219,25229,25237,25243,25247,25253,25261,25301,25303,
25307,25309,25321,25339,25343,25349,25357,25367,25373,
25391,25409,25411,25423,25439,25447,25453,25457,25463,
25469,25471,25523,25537,25541,25561,25577,25579,25583,
25589,25601,25603,25609,25621,25633,25639,25643,25657,
25667,25673,25679,25693,25703,25717,25733,25741,25747,
25759,25763,25771,25793,25799,25801,25819,25841,25847,
25849,25867,25873,25889,25903,25913,25919,25931,25933,
25939,25943,25951,25969,25981,25997,25999,26003,26017,
26021,26029,26041,26053,26083,26099,26107,26111,26113,
26119,26141,26153,26161,26171,26177,26183,26189,26203,
26209,26227,26237,26249,26251,26261,26263,26267,26293,
26297,26309,26317,26321,26339,26347,26357,26371,26387,
26393,26399,26407,26417,26423,26431,26437,26449,26459,
26479,26489,26497,26501,26513,26539,26557,26561,26573,
26591,26597,26627,26633,26641,26647,26669,26681,26683,
26687,26693,26699,26701,26711,26713,26717,26723,26729,
26731,26737,26759,26777,26783,26801,26813,26821,26833,
26839,26849,26861,26863,26879,26881,26891,26893,26903,
26921,26927,26947,26951,26953,26959,26981,26987,26993,
27011,27017,27031,27043,27059,27061,27067,27073,27077,
27091,27103,27107,27109,27127,27143,27179,27191,27197,
27211,27239,27241,27253,27259,27271,27277,27281,27283,
27299,27329,27337,27361,27367,27397,27407,27409,27427,
27431,27437,27449,27457,27479,27481,27487,27509,27527,
27529,27539,27541,27551,27581,27583,27611,27617,27631,
27647,27653,27673,27689,27691,27697,27701,27733,27737,
27739,27743,27749,27751,27763,27767,27773,27779,27791,
27793,27799,27803,27809,27817,27823,27827,27847,27851,
27883,27893,27901,27917,27919,27941,27943,27947,27953,
27961,27967,27983,27997,28001,28019,28027,28031,28051,
28057,28069,28081,28087,28097,28099,28109,28111,28123,
28151,28163,28181,28183,28201,28211,28219,28229,28277,
28279,28283,28289,28297,28307,28309,28319,28349,28351,

28387,28393,28403,28409,28411,28429,28433,28439,28447,
28463,28477,28493,28499,28513,28517,28537,28541,28547,
28549,28559,28571,28573,28579,28591,28597,28603,28607,
28619,28621,28627,28631,28643,28649,28657,28661,28663,
28669,28687,28697,28703,28711,28723,28729,28751,28753,
28759,28771,28789,28793,28807,28813,28817,28837,28843,
28859,28867,28871,28879,28901,28909,28921,28927,28933,
28949,28961,28979,29009,29017,29021,29023,29027,29033,
29059,29063,29077,29101,29123,29129,29131,29137,29147,
29153,29167,29173,29179,29191,29201,29207,29209,29221,
29231,29243,29251,29269,29287,29297,29303,29311,29327,
29333,29339,29347,29363,29383,29387,29389,29399,29401,
29411,29423,29429,29437,29443,29453,29473,29483,29501,
29527,29531,29537,29567,29569,29573,29581,29587,29599,
29611,29629,29633,29641,29663,29669,29671,29683,29717,
29723,29741,29753,29759,29761,29789,29803,29819,29833,
29837,29851,29863,29867,29873,29879,29881,29917,29921,
29927,29947,29959,29983,29989,30011,30013,30029,30047,
30059,30071,30089,30091,30097,30103,30109,30113,30119,
30133,30137,30139,30161,30169,30181,30187,30197,30203,
30211,30223,30241,30253,30259,30269,30271,30293,30307,
30313,30319,30323,30341,30347,30367,30389,30391,30403,
30427,30431,30449,30467,30469,30491,30493,30497,30509,
30517,30529,30539,30553,30557,30559,30577,30593,30631,
30637,30643,30649,30661,30671,30677,30689,30697,30703,
30707,30713,30727,30757,30763,30773,30781,30803,30809,
30817,30829,30839,30841,30851,30853,30859,30869,30871,
30881,30893,30911,30931,30937,30941,30949,30971,30977,
30983,31013,31019,31033,31039,31051,31063,31069,31079,
31081,31091,31121,31123,31139,31147,31151,31153,31159,
31177,31181,31183,31189,31193,31219,31223,31231,31237,
31247,31249,31253,31259,31267,31271,31277,31307,31319,
31321,31327,31333,31337,31357,31379,31387,31391,31393,
31397,31469,31477,31481,31489,31511,31513,31517,31531,
31541,31543,31547,31567,31573,31583,31601,31607,31627,
31643,31649,31657,31663,31667,31687,31699,31721,31723,
31727,31729,31741,31751,31769,31771,31793,31799,31817,
31847,31849,31859,31873,31883,31891,31907,31957,31963,
31973,31981,31991,32003,32009,32027,32029,32051,32057,
32059,32063,32069,32077,32083,32089,32099,32117,32119,
32141,32143,32159,32173,32183,32189,32191,32203,32213,
32233,32237,32251,32257,32261,32297,32299,32303,32309,
32321,32323,32327,32341,32353,32359,32363,32369,32371,
32377,32381,32401,32411,32413,32423,32429,32441,32443,
32467,32479,32491,32497,32503,32507,32531,32533,32537,
32561,32563,32569,32573,32579,32587,32603,32609,32611,
32621,32633,32647,32653,32687,32693,32707,32713,32717,
32719,32749,32771,32779,32783,32789,32797,32801,32803,
32831,32833,32839,32843,32869,32887,32909,32911,32917,
32933,32939,32941,32957,32969,32971,32983,32987,32993,
32999,33013,33023,33029,33037,33049,33053,33071,33073,

33083,33091,33107,33113,33119,33149,33151,33161,33179,
33181,33191,33199,33203,33211,33223,33247,33287,33289,
33301,33311,33317,33329,33331,33343,33347,33349,33353,
33359,33377,33391,33403,33409,33413,33427,33457,33461,
33469,33479,33487,33493,33503,33521,33529,33533,33547,
33563,33569,33577,33581,33587,33589,33599,33601,33613,
33617,33619,33623,33629,33637,33641,33647,33679,33703,
33713,33721,33739,33749,33751,33757,33767,33769,33773,
33791,33797,33809,33811,33827,33829,33851,33857,33863,
33871,33889,33893,33911,33923,33931,33937,33941,33961,
33967,33997,34019,34031,34033,34039,34057,34061,34123,
34127,34129,34141,34147,34157,34159,34171,34183,34211,
34213,34217,34231,34253,34259,34261,34267,34273,34283,
34297,34301,34303,34313,34319,34327,34337,34351,34361,
34367,34369,34381,34403,34421,34429,34439,34457,34469,
34471,34483,34487,34499,34501,34511,34513,34519,34537,
34543,34549,34583,34589,34591,34603,34607,34613,34631,
34649,34651,34667,34673,34679,34687,34693,34703,34721,
34729,34739,34747,34757,34759,34763,34781,34807,34819,
34841,34843,34847,34849,34871,34877,34883,34897,34913,
34919,34939,34949,34961,34963,34981,35023,35027,35051,
35053,35059,35069,35081,35083,35089,35099,35107,35111,
35117,35129,35141,35149,35153,35159,35171,35201,35221,
35227,35251,35257,35267,35279,35281,35291,35311,35317,
35323,35327,35339,35353,35363,35381,35393,35401,35407,
35419,35423,35437,35447,35449,35461,35491,35507,35509,
35521,35527,35531,35533,35537,35543,35569,35573,35591,
35593,35597,35603,35617,35671,35677,35729,35731,35747,
35753,35759,35771,35797,35801,35803,35809,35831,35837,
35839,35851,35863,35869,35879,35897,35899,35911,35923,
35933,35951,35963,35969,35977,35983,35993,35999,36007,
36011,36013,36017,36037,36061,36067,36073,36083,36097,
36107,36109,36131,36137,36151,36161,36187,36191,36209,
36217,36229,36241,36251,36263,36269,36277,36293,36299,
36307,36313,36319,36341,36343,36353,36373,36383,36389,
36433,36451,36457,36467,36469,36473,36479,36493,36497,
36523,36527,36529,36541,36551,36559,36563,36571,36583,
36587,36599,36607,36629,36637,36643,36653,36671,36677,
36683,36691,36697,36709,36713,36721,36739,36749,36761,
36767,36779,36781,36787,36791,36793,36809,36821,36833,
36847,36857,36871,36877,36887,36899,36901,36913,36919,
36923,36929,36931,36943,36947,36973,36979,36997,37003,
37013,37019,37021,37039,37049,37057,37061,37087,37097,
37117,37123,37139,37159,37171,37181,37189,37199,37201,
37217,37223,37243,37253,37273,37277,37307,37309,37313,
37321,37337,37339,37357,37361,37363,37369,37379,37397,
37409,37423,37441,37447,37463,37483,37489,37493,37501,
37507,37511,37517,37529,37537,37547,37549,37561,37567,
37571,37573,37579,37589,37591,37607,37619,37633,37643,
37649,37657,37663,37691,37693,37699,37717,37747,37781,
37783,37799,37811,37813,37831,37847,37853,37861,37871,

37879, 37889, 37897, 37907, 37951, 37957, 37963, 37967, 37987,
37991, 37993, 37997, 38011, 38039, 38047, 38053, 38069, 38083,
38113, 38119, 38149, 38153, 38167, 38177, 38183, 38189, 38197,
38201, 38219, 38231, 38237, 38239, 38261, 38273, 38281, 38287,
38299, 38303, 38317, 38321, 38327, 38329, 38333, 38351, 38371,
38377, 38393, 38431, 38447, 38449, 38453, 38459, 38461, 38501,
38543, 38557, 38561, 38567, 38569, 38593, 38603, 38609, 38611,
38629, 38639, 38651, 38653, 38669, 38671, 38677, 38693, 38699,
38707, 38711, 38713, 38723, 38729, 38737, 38747, 38749, 38767,
38783, 38791, 38803, 38821, 38833, 38839, 38851, 38861, 38867,
38873, 38891, 38903, 38917, 38921, 38923, 38933, 38953, 38959,
38971, 38977, 38993, 39019, 39023, 39041, 39043, 39047, 39079,
39089, 39097, 39103, 39107, 39113, 39119, 39133, 39139, 39157,
39161, 39163, 39181, 39191, 39199, 39209, 39217, 39227, 39229,
39233, 39239, 39241, 39251, 39293, 39301, 39313, 39317, 39323,
39341, 39343, 39359, 39367, 39371, 39373, 39383, 39397, 39409,
39419, 39439, 39443, 39451, 39461, 39499, 39503, 39509, 39511,
39521, 39541, 39551, 39563, 39569, 39581, 39607, 39619, 39623,
39631, 39659, 39667, 39671, 39679, 39703, 39709, 39719, 39727,
39733, 39749, 39761, 39769, 39779, 39791, 39799, 39821, 39827,
39829, 39839, 39841, 39847, 39857, 39863, 39869, 39877, 39883,
39887, 39901, 39929, 39937, 39953, 39971, 39979, 39983, 39989,
40009, 40013, 40031, 40037, 40039, 40063, 40087, 40093, 40099,
40111, 40123, 40127, 40129, 40151, 40153, 40163, 40169, 40177,
40189, 40193, 40213, 40231, 40237, 40241, 40253, 40277, 40283,
40289, 40343, 40351, 40357, 40361, 40387, 40423, 40427, 40429,
40433, 40459, 40471, 40483, 40487, 40493, 40499, 40507, 40519,
40529, 40531, 40543, 40559, 40577, 40583, 40591, 40597, 40609,
40627, 40637, 40639, 40693, 40697, 40699, 40709, 40739, 40751,
40759, 40763, 40771, 40787, 40801, 40813, 40819, 40823, 40829,
40841, 40847, 40849, 40853, 40867, 40879, 40883, 40897, 40903,
40927, 40933, 40939, 40949, 40961, 40973, 40993, 41011, 41017,
41023, 41039, 41047, 41051, 41057, 41077, 41081, 41113, 41117,
41131, 41141, 41143, 41149, 41161, 41177, 41179, 41183, 41189,
41201, 41203, 41213, 41221, 41227, 41231, 41233, 41243, 41257,
41263, 41269, 41281, 41299, 41333, 41341, 41351, 41357, 41381,
41387, 41389, 41399, 41411, 41413, 41443, 41453, 41467, 41479,
41491, 41507, 41513, 41519, 41521, 41539, 41543, 41549, 41579,
41593, 41597, 41603, 41609, 41611, 41617, 41621, 41627, 41641,
41647, 41651, 41659, 41669, 41681, 41687, 41719, 41729, 41737,
41759, 41761, 41771, 41777, 41801, 41809, 41813, 41843, 41849,
41851, 41863, 41879, 41887, 41893, 41897, 41903, 41911, 41927,
41941, 41947, 41953, 41957, 41959, 41969, 41981, 41983, 41999,
42013, 42017, 42019, 42023, 42043, 42061, 42071, 42073, 42083,
42089, 42101, 42131, 42139, 42157, 42169, 42179, 42181, 42187,
42193, 42197, 42209, 42221, 42223, 42227, 42239, 42257, 42281,
42283, 42293, 42299, 42307, 42323, 42331, 42337, 42349, 42359,
42373, 42379, 42391, 42397, 42403, 42407, 42409, 42433, 42437,
42443, 42451, 42457, 42461, 42463, 42467, 42473, 42487, 42491,
42499, 42509, 42533, 42557, 42569, 42571, 42577, 42589, 42611,
42641, 42643, 42649, 42667, 42677, 42683, 42689, 42697, 42701,

42703,42709,42719,42727,42737,42743,42751,42767,42773,
42787,42793,42797,42821,42829,42839,42841,42853,42859,
42863,42899,42901,42923,42929,42937,42943,42953,42961,
42967,42979,42989,43003,43013,43019,43037,43049,43051,
43063,43067,43093,43103,43117,43133,43151,43159,43177,
43189,43201,43207,43223,43237,43261,43271,43283,43291,
43313,43319,43321,43331,43391,43397,43399,43403,43411,
43427,43441,43451,43457,43481,43487,43499,43517,43541,
43543,43573,43577,43579,43591,43597,43607,43609,43613,
43627,43633,43649,43651,43661,43669,43691,43711,43717,
43721,43753,43759,43777,43781,43783,43787,43789,43793,
43801,43853,43867,43889,43891,43913,43933,43943,43951,
43961,43963,43969,43973,43987,43991,43997,44017,44021,
44027,44029,44041,44053,44059,44071,44087,44089,44101,
44111,44119,44123,44129,44131,44159,44171,44179,44189,
44201,44203,44207,44221,44249,44257,44263,44267,44269,
44273,44279,44281,44293,44351,44357,44371,44381,44383,
44389,44417,44449,44453,44483,44491,44497,44501,44507,
44519,44531,44533,44537,44543,44549,44563,44579,44587,
44617,44621,44623,44633,44641,44647,44651,44657,44683,
44687,44699,44701,44711,44729,44741,44753,44771,44773,
44777,44789,44797,44809,44819,44839,44843,44851,44867,
44879,44887,44893,44909,44917,44927,44939,44953,44959,
44963,44971,44983,44987,45007,45013,45053,45061,45077,
45083,45119,45121,45127,45131,45137,45139,45161,45179,
45181,45191,45197,45233,45247,45259,45263,45281,45289,
45293,45307,45317,45319,45329,45337,45341,45343,45361,
45377,45389,45403,45413,45427,45433,45439,45481,45491,
45497,45503,45523,45533,45541,45553,45557,45569,45587,
45589,45599,45613,45631,45641,45659,45667,45673,45677,
45691,45697,45707,45737,45751,45757,45763,45767,45779,
45817,45821,45823,45827,45833,45841,45853,45863,45869,
45887,45893,45943,45949,45953,45959,45971,45979,45989,
46021,46027,46049,46051,46061,46073,46091,46093,46099,
46103,46133,46141,46147,46153,46171,46181,46183,46187,
46199,46219,46229,46237,46261,46271,46273,46279,46301,
46307,46309,46327,46337,46349,46351,46381,46399,46411,
46439,46441,46447,46451,46457,46471,46477,46489,46499,
46507,46511,46523,46549,46559,46567,46573,46589,46591,
46601,46619,46633,46639,46643,46649,46663,46679,46681,
46687,46691,46703,46723,46727,46747,46751,46757,46769,
46771,46807,46811,46817,46819,46829,46831,46853,46861,
46867,46877,46889,46901,46919,46933,46957,46993,46997,
47017,47041,47051,47057,47059,47087,47093,47111,47119,
47123,47129,47137,47143,47147,47149,47161,47189,47207,
47221,47237,47251,47269,47279,47287,47293,47297,47303,
47309,47317,47339,47351,47353,47363,47381,47387,47389,
47407,47417,47419,47431,47441,47459,47491,47497,47501,
47507,47513,47521,47527,47533,47543,47563,47569,47581,
47591,47599,47609,47623,47629,47639,47653,47657,47659,
47681,47699,47701,47711,47713,47717,47737,47741,47743,

47777,47779,47791,47797,47807,47809,47819,47837,47843,
47857,47869,47881,47903,47911,47917,47933,47939,47947,
47951,47963,47969,47977,47981,48017,48023,48029,48049,
48073,48079,48091,48109,48119,48121,48131,48157,48163,
48179,48187,48193,48197,48221,48239,48247,48259,48271,
48281,48299,48311,48313,48337,48341,48353,48371,48383,
48397,48407,48409,48413,48437,48449,48463,48473,48479,
48481,48487,48491,48497,48523,48527,48533,48539,48541,
48563,48571,48589,48593,48611,48619,48623,48647,48649,
48661,48673,48677,48679,48731,48733,48751,48757,48761,
48767,48779,48781,48787,48799,48809,48817,48821,48823,
48847,48857,48859,48869,48871,48883,48889,48907,48947,
48953,48973,48989,48991,49003,49009,49019,49031,49033,
49037,49043,49057,49069,49081,49103,49109,49117,49121,
49123,49139,49157,49169,49171,49177,49193,49199,49201,
49207,49211,49223,49253,49261,49277,49279,49297,49307,
49331,49333,49339,49363,49367,49369,49391,49393,49409,
49411,49417,49429,49433,49451,49459,49463,49477,49481,
49499,49523,49529,49531,49537,49547,49549,49559,49597,
49603,49613,49627,49633,49639,49663,49667,49669,49681,
49697,49711,49727,49739,49741,49747,49757,49783,49787,
49789,49801,49807,49811,49823,49831,49843,49853,49871,
49877,49891,49919,49921,49927,49937,49939,49943,49957,
49991,49993,49999,50021,50023,50033,50047,50051,50053,
50069,50077,50087,50093,50101,50111,50119,50123,50129,
50131,50147,50153,50159,50177,50207,50221,50227,50231,
50261,50263,50273,50287,50291,50311,50321,50329,50333,
50341,50359,50363,50377,50383,50387,50411,50417,50423,
50441,50459,50461,50497,50503,50513,50527,50539,50543,
50549,50551,50581,50587,50591,50593,50599,50627,50647,
50651,50671,50683,50707,50723,50741,50753,50767,50773,
50777,50789,50821,50833,50839,50849,50857,50867,50873,
50891,50893,50909,50923,50929,50951,50957,50969,50971,
50989,50993,51001,51031,51043,51047,51059,51061,51071,
51109,51131,51133,51137,51151,51157,51169,51193,51197,
51199,51203,51217,51229,51239,51241,51257,51263,51283,
51287,51307,51329,51341,51343,51347,51349,51361,51383,
51407,51413,51419,51421,51427,51431,51437,51439,51449,
51461,51473,51479,51481,51487,51503,51511,51517,51521,
51539,51551,51563,51577,51581,51593,51599,51607,51613,
51631,51637,51647,51659,51673,51679,51683,51691,51713,
51719,51721,51749,51767,51769,51787,51797,51803,51817,
51827,51829,51839,51853,51859,51869,51871,51893,51899,
51907,51913,51929,51941,51949,51971,51973,51977,51991,
52009,52021,52027,52051,52057,52067,52069,52081,52103,
52121,52127,52147,52153,52163,52177,52181,52183,52189,
52201,52223,52237,52249,52253,52259,52267,52289,52291,
52301,52313,52321,52361,52363,52369,52379,52387,52391,
52433,52453,52457,52489,52501,52511,52517,52529,52541,
52543,52553,52561,52567,52571,52579,52583,52609,52627,
52631,52639,52667,52673,52691,52697,52709,52711,52721,

52727,52733,52747,52757,52769,52783,52807,52813,52817,
52837,52859,52861,52879,52883,52889,52901,52903,52919,
52937,52951,52957,52963,52967,52973,52981,52999,53003,
53017,53047,53051,53069,53077,53087,53089,53093,53101,
53113,53117,53129,53147,53149,53161,53171,53173,53189,
53197,53201,53231,53233,53239,53267,53269,53279,53281,
53299,53309,53323,53327,53353,53359,53377,53381,53401,
53407,53411,53419,53437,53441,53453,53479,53503,53507,
53527,53549,53551,53569,53591,53593,53597,53609,53611,
53617,53623,53629,53633,53639,53653,53657,53681,53693,
53699,53717,53719,53731,53759,53773,53777,53783,53791,
53813,53819,53831,53849,53857,53861,53881,53887,53891,
53897,53899,53917,53923,53927,53939,53951,53959,53987,
53993,54001,54011,54013,54037,54049,54059,54083,54091,
54101,54121,54133,54139,54151,54163,54167,54181,54193,
54217,54251,54269,54277,54287,54293,54311,54319,54323,
54331,54347,54361,54367,54371,54377,54401,54403,54409,
54413,54419,54421,54437,54443,54449,54469,54493,54497,
54499,54503,54517,54521,54539,54541,54547,54559,54563,
54577,54581,54583,54601,54617,54623,54629,54631,54647,
54667,54673,54679,54709,54713,54721,54727,54751,54767,
54773,54779,54787,54799,54829,54833,54851,54869,54877,
54881,54907,54917,54919,54941,54949,54959,54973,54979,
54983,55001,55009,55021,55049,55051,55057,55061,55073,
55079,55103,55109,55117,55127,55147,55163,55171,55201,
55207,55213,55217,55219,55229,55243,55249,55259,55291,
55313,55331,55333,55337,55339,55343,55351,55373,55381,
55399,55411,55439,55441,55457,55469,55487,55501,55511,
55529,55541,55547,55579,55589,55603,55609,55619,55621,
55631,55633,55639,55661,55663,55667,55673,55681,55691,
55697,55711,55717,55721,55733,55763,55787,55793,55799,
55807,55813,55817,55819,55823,55829,55837,55843,55849,
55871,55889,55897,55901,55903,55921,55927,55931,55933,
55949,55967,55987,55997,56003,56009,56039,56041,56053,
56081,56087,56093,56099,56101,56113,56123,56131,56149,
56167,56171,56179,56197,56207,56209,56237,56239,56249,
56263,56267,56269,56299,56311,56333,56359,56369,56377,
56383,56393,56401,56417,56431,56437,56443,56453,56467,
56473,56477,56479,56489,56501,56503,56509,56519,56527,
56531,56533,56543,56569,56591,56597,56599,56611,56629,
56633,56659,56663,56671,56681,56687,56701,56711,56713,
56731,56737,56747,56767,56773,56779,56783,56807,56809,
56813,56821,56827,56843,56857,56873,56891,56893,56897,
56909,56911,56921,56923,56929,56941,56951,56957,56963,
56983,56989,56993,56999,57037,57041,57047,57059,57073,
57077,57089,57097,57107,57119,57131,57139,57143,57149,
57163,57173,57179,57191,57193,57203,57221,57223,57241,
57251,57259,57269,57271,57283,57287,57301,57329,57331,
57347,57349,57367,57373,57383,57389,57397,57413,57427,
57457,57467,57487,57493,57503,57527,57529,57557,57559,
57571,57587,57593,57601,57637,57641,57649,57653,57667,

57679,57689,57697,57709,57713,57719,57727,57731,57737,
57751,57773,57781,57787,57791,57793,57803,57809,57829,
57839,57847,57853,57859,57881,57899,57901,57917,57923,
57943,57947,57973,57977,57991,58013,58027,58031,58043,
58049,58057,58061,58067,58073,58099,58109,58111,58129,
58147,58151,58153,58169,58171,58189,58193,58199,58207,
58211,58217,58229,58231,58237,58243,58271,58309,58313,
58321,58337,58363,58367,58369,58379,58391,58393,58403,
58411,58417,58427,58439,58441,58451,58453,58477,58481,
58511,58537,58543,58549,58567,58573,58579,58601,58603,
58613,58631,58657,58661,58679,58687,58693,58699,58711,
58727,58733,58741,58757,58763,58771,58787,58789,58831,
58889,58897,58901,58907,58909,58913,58921,58937,58943,
58963,58967,58979,58991,58997,59009,59011,59021,59023,
59029,59051,59053,59063,59069,59077,59083,59093,59107,
59113,59119,59123,59141,59149,59159,59167,59183,59197,
59207,59209,59219,59221,59233,59239,59243,59263,59273,
59281,59333,59341,59351,59357,59359,59369,59377,59387,
59393,59399,59407,59417,59419,59441,59443,59447,59453,
59467,59471,59473,59497,59509,59513,59539,59557,59561,
59567,59581,59611,59617,59621,59627,59629,59651,59659,
59663,59669,59671,59693,59699,59707,59723,59729,59743,
59747,59753,59771,59779,59791,59797,59809,59833,59863,
59879,59887,59921,59929,59951,59957,59971,59981,59999,
60013,60017,60029,60037,60041,60077,60083,60089,60091,
60101,60103,60107,60127,60133,60139,60149,60161,60167,
60169,60209,60217,60223,60251,60257,60259,60271,60289,
60293,60317,60331,60337,60343,60353,60373,60383,60397,
60413,60427,60443,60449,60457,60493,60497,60509,60521,
60527,60539,60589,60601,60607,60611,60617,60623,60631,
60637,60647,60649,60659,60661,60679,60689,60703,60719,
60727,60733,60737,60757,60761,60763,60773,60779,60793,
60811,60821,60859,60869,60887,60889,60899,60901,60913,
60917,60919,60923,60937,60943,60953,60961,61001,61007,
61027,61031,61043,61051,61057,61091,61099,61121,61129,
61141,61151,61153,61169,61211,61223,61231,61253,61261,
61283,61291,61297,61331,61333,61339,61343,61357,61363,
61379,61381,61403,61409,61417,61441,61463,61469,61471,
61483,61487,61493,61507,61511,61519,61543,61547,61553,
61559,61561,61583,61603,61609,61613,61627,61631,61637,
61643,61651,61657,61667,61673,61681,61687,61703,61717,
61723,61729,61751,61757,61781,61813,61819,61837,61843,
61861,61871,61879,61909,61927,61933,61949,61961,61967,
61979,61981,61987,61991,62003,62011,62017,62039,62047,
62053,62057,62071,62081,62099,62119,62129,62131,62137,
62141,62143,62171,62189,62191,62201,62207,62213,62219,
62233,62273,62297,62299,62303,62311,62323,62327,62347,
62351,62383,62401,62417,62423,62459,62467,62473,62477,
62483,62497,62501,62507,62533,62539,62549,62563,62581,
62591,62597,62603,62617,62627,62633,62639,62653,62659,
62683,62687,62701,62723,62731,62743,62753,62761,62773,

```

62791,62801,62819,62827,62851,62861,62869,62873,62897,
62903,62921,62927,62929,62939,62969,62971,62981,62983,
62987,62989,63029,63031,63059,63067,63073,63079,63097,
63103,63113,63127,63131,63149,63179,63197,63199,63211,
63241,63247,63277,63281,63299,63311,63313,63317,63331,
63337,63347,63353,63361,63367,63377,63389,63391,63397,
63409,63419,63421,63439,63443,63463,63467,63473,63487,
63493,63499,63521,63527,63533,63541,63559,63577,63587,
63589,63599,63601,63607,63611,63617,63629,63647,63649,
63659,63667,63671,63689,63691,63697,63703,63709,63719,
63727,63737,63743,63761,63773,63781,63793,63799,63803,
63809,63823,63839,63841,63853,63857,63863,63901,63907,
63913,63929,63949,63977,63997,64007,64013,64019,64033,
64037,64063,64067,64081,64091,64109,64123,64151,64153,
64157,64171,64187,64189,64217,64223,64231,64237,64271,
64279,64283,64301,64303,64319,64327,64333,64373,64381,
64399,64403,64433,64439,64451,64453,64483,64489,64499,
64513,64553,64567,64577,64579,64591,64601,64609,64613,
64621,64627,64633,64661,64663,64667,64679,64693,64709,
64717,64747,64763,64781,64783,64793,64811,64817,64849,
64853,64871,64877,64879,64891,64901,64919,64921,64927,
64937,64951,64969,64997,65003,65011,65027,65029,65033,
65053,65063,65071,65089,65099,65101,65111,65119,65123,
65129,65141,65147,65167,65171,65173,65179,65183,65203,
65213,65239,65257,65267,65269,65287,65293,65309,65323,
65327,65353,65357,65371,65381,65393,65407,65413,65419,
65423,65437,65447,65449,65479,65497,65519,65521,65537
};

while (primeTable[primeTableIndex]<=number)
{
    if ( (number%primeTable[primeTableIndex])==0 )
    {
        number=number/primeTable[primeTableIndex];
#ifdef _debug_
        cout << primeTable[primeTableIndex] << " ";
#endif
        factors[factorIndex++]=primeTable[primeTableIndex];
        if (factorIndex>=maxNbrFactor)
        {
            cerr << "The number of factors in "
                << number << " is greater than the maximum
("
                << maxNbrFactor << ") allowed\n";
            return -2;
        }
    }
    else
        primeTableIndex++;
}

```

```

#ifdef _debug_
    cout << "\n";
#endif
    return factorIndex;
}

```

A.17 patternDim

```

#ifndef _patternDim_h
#define _patternDim_h

int patternDim(unsigned long nbrBits, unsigned long dimensions[2]);
int getDim(unsigned long* dim, unsigned factors[], unsigned
nbrFactors,
            char factorsI[], char factorsII, char nbrMult);

#endif

#include <iostream.h>
#include <stdio.h>
#include "factor.h"
#include "patternDim.h"

#define MAX_NBR_FACTORS 100

int patternDim(unsigned long nbrBits, unsigned long dimensions[2])
{
    unsigned factors[MAX_NBR_FACTORS];
    unsigned longdim=1,finalDim=1;
    unsigned nbrFactors=0;
    double distance=nbrBits;
    char done=0;

    if ((nbrFactors=factor(nbrBits,factors,MAX_NBR_FACTORS))<0)
    {
        cerr << "Error could not factor the number of bits in the
pattern\n";
        return -1;
    }

    char factorsI[nbrFactors],factorsII=0;

    for (int i=0;i<nbrFactors;i++)
        factorsI[i]=i;

    for (int nbrMult=1;nbrMult<=nbrFactors/2;nbrMult++)
    {
        done=0;

```

```

        while (!done)
        {
            dim=1;
            done=getDim(&dim,factors,nbrFactors,factorsI,0,nbrMult);
            if (abs(dim - nbrBits/dim)<distance)
            {
                distance=abs(dim - nbrBits/dim);
                finalDim=dim;
            }
        }
    }

    // the width is always bigger than the height
    if (finalDim>nbrBits/finalDim)
    {
        dimensions[0]=finalDim;
        dimensions[1]=nbrBits/finalDim;
    }
    else
    {
        dimensions[0]=nbrBits/finalDim;
        dimensions[1]=finalDim;
    }

    return 0;
}

int      getDim(unsigned long* dim, unsigned factors[], unsigned
nbrFactors,
                char factorsI[], char factorsII, char nbrMult)
{
    char      end=0;

    // this should be first because it is the stop condition of the
    // recurrence
    if (factorsI[factorsII]>(nbrFactors-nbrMult))
    {
        // this is to take care of the very first caller in the
recursion
        // otherwise its index stays at the end of the factors list and
        // as no getDim calls it it will not get reset. However, if it
is
        // not the first caller, its index will get overwritten anyway
        // by the caller
        factorsI[factorsII]=0;
        return 1;
    }

    *dim*=factors[factorsI[factorsII]];

    if (nbrMult!=1)

```

```

end=getDim(dim,factors,nbrFactors,factorsI,factorsII+1,nbrMult-1);

    if (end==1 || nbrMult==1)
    {
        factorsI[factorsII]++;
        if (nbrMult!=1)
            factorsI[factorsII+1]=factorsI[factorsII]+1;
    }

    return 0;
}

```

A.18 pattern1

```

// pattern1.h
#ifndef _pattern1_h
#define _pattern1_h

#include "coDec.h"

class pattern1: public coDec
{
    unsigned int* generator;//indexes of generator polynomial for the
code
    unsigned int* systematic; // systematic symbols buffer.
    unsigned int* FSR;//feedback shift registers for redundant bits
gen.
    unsigned int SBin;// buffer to build symbols (8bits->mbits)
    int SBinBits;//number of bits in the symbol buffer
    int padBits;
    unsigned int SBout;// buffer to build symbols (8bits->mbits)
    int SBoutBits;//number of bits in the symbol buffer
    int inputCount;
    int outputCount;
    int lastWrite;
    int bitsInBuffer;
    int nbrLines;
    int maxNbrLines;

    int feedback,FSRindex,FSRregBit,
        maxOutput,CBbits;
    unsigned int charBuffer, output,symbol;
    unsigned char tempChar;
    unsigned char firstTime;
    int done;
    int realBits;

    char sideDone;

```

```

        char                doneData;

        int bytesToOutput;
        int nbrBytesOutput;
        int remainingBits;
        int charPerLine;

        public:
        pattern1(int _nbrCol);
        pattern1(int _nbrCol, int _nbrLines);
        int processData();
        int giveNbrLines();
        int processSymbols();

        void newData();
};

#endif

// pattern1.cc
#include "pattern1.h"
#include "debugger.h"
#include "debug.h"
#include <values.h>

pattern1::pattern1(int _nbrCols): generator(0),
        systematic(0), FSR(0), SBin(0), SBinBits(0), padBits(0),
        SBout(0),
        SBoutBits(0), inputCount(0), outputCount(0),
        bytesToOutput(0),
        nbrBytesOutput(0), remainingBits(0), lastWrite(1),
        nbrLines(0),
        maxNbrLines(MAXINT),
        feedback(0),FSRindex(0),FSRregBit(0),
        maxOutput(0),CBbits(0),
        charBuffer(0), output(0),symbol(0),
        tempChar(0),
        firstTime(0),
        done(0),
        realBits(0),
        sideDone(0),
        doneData(0)

{
        int nbrBits;

        bytesToOutput=(_nbrCols+2)/8;// determine the number of bytes that
the
        remainingBits=(_nbrCols+2)%8;// cols fits in
        if (remainingBits==0)
        {

```

```

        bytesToOutput--; // To take care of the border
        remainingBits=8; // because the last bit is not data, it is
                           // just a high bit
    }
    charPerLine=bytesToOutput+(remainingBits!=0);
    cout << bytesToOutput << " " << remainingBits << endl;
}

pattern1::pattern1(int _nbrCols,int _nbrLines=-1): generator(0),
    systematic(0), FSR(0), SBin(0), SBinBits(0), padBits(0),
    SBout(0),
        SBoutBits(0),        inputCount(0),        outputCount(0),
    bytesToOutput(0),
        nbrBytesOutput(0),        remainingBits(0),        lastWrite(1),
    nbrLines(0),
        maxNbrLines(_nbrLines),
    feedback(0),FSRindex(0),FSRregBit(0),
        maxOutput(0),CBbits(0),
    charBuffer(0), output(0),symbol(0),
    tempChar(0),
    firstTime(0),
    done(0),
    realBits(0),
    sideDone(0),
    doneData(0)
{
    int nbrBits;

    bytesToOutput=(_nbrCols+2)/8; // determine the number of bytes that
the
    remainingBits=(_nbrCols+2)%8; // cols fits in
    if (remainingBits==0)
    {
        bytesToOutput--; // To take care of the border
        remainingBits=8; // because the last bit is not data, it is
                           // just a high bit
    }
    charPerLine=bytesToOutput+(remainingBits!=0);
    cout << bytesToOutput << " " << remainingBits << endl;
    if (maxNbrLines>0)
        maxNbrLines+=2;
}

int pattern1::processData()
{
    if (!inBuffer || !outBuffer)
    {
        cerr << "Please assign me with some input and output "
        << "buffers before asking me to process data" << endl;
        return -1;
    }
}

```

```

    }

    dataInput_Process:

    eof=inBuffer->atEOF();
    if ( done>=charPerLine )
    {
        outBuffer->setEOF();
        return 1;
    }
    // add one line of bits at the beginning of the pattern minus one bit
    // because the right side and the left side are added at the same time
    // by the central routine
    while ( outBuffer->testWrite() && firstTime<charPerLine )
    {
        outBuffer->put(0xff);
        firstTime++;
        if (firstTime>=charPerLine)
            nbrLines++;
        /*
        if ( (SBinBits<8 && outputCount!=bytesToOutput) ||
            (SBinBits<remainingBits
outputCount==bytesToOutput) )
        {
            SBin=(SBin<<8)+0xff;
            SBinBits+=8;
        }

        if      (SBinBits>=8      ||      (SBinBits>=remainingBits      &&
outputCount==bytesToOutput) )
        {
            // extract symbol from symbol buffer (SBin):
            if (outputCount==bytesToOutput)
            {
                if (remainingBits!=0)
                {
                    SBinBits--=(remainingBits-1);
                    symbol=SBin>>SBinBits;
                    SBin^=symbol<<SBinBits;
                    symbol=symbol<<(8-remainingBits);
                    //outBuffer->put((unsigned char)(symbol&0xff));
                    outBuffer->put(symbol);
                }
                outputCount=0;
                nbrLines++;
                firstTime=0;
                SBinBits=0;
                SBin=0;
            }
            else
            {

```

```

        SBinBits-=8;
        symbol=SBin>>SBinBits;
        outBuffer->put(symbol);
        SBin^=symbol<<SBinBits;
        outputCount++;
    }

#ifdef _debug_
    cout << "new data: " << symbol << " ";
    binaire(symbol,cout,m);
    cout << endl;
#endif
}
*/
}

while ( (outBuffer->testWrite()) && firstTime>=charPerLine &&
        ( inBuffer->testRead() || (eof=inBuffer->atEOF()) ) &&
        !doneData )
{
    if (outputCount==0 && !sideDone)
    {
        //cout<< SBinBits << " " << hex << SBin << " ";
        SBin+=(0x1<<(SBinBits++));
        realBits++;
        sideDone=1;
        //cout<< hex << SBin << endl;
    }

    if ( (SBinBits<8 && outputCount!=bytesToOutput) ||
        (SBinBits<remainingBits && outputCount==bytesToOutput) )
    {
        // padBits flags when the system started padding the
information
        // (when there is not enough info to fill info block so 0
pad)
        if (eof)
            SBin=(SBin<<8)+0x0;
        else
        {
            inBuffer->get(tempChar);
            SBin=(SBin<<8)+tempChar;
            realBits+=8;
        }
        SBinBits+=8;
    }

    if (SBinBits>=8 || (SBinBits>=remainingBits &&
outputCount==bytesToOutput) )
    {
        // extract symbol from symbol buffer (SBin):

```

```

        if (outputCount==bytesToOutput)
        {
            //cout<< hex << SBin << " ";
            //if (remainingBits!=0)
            //{
                SBinBits--=(remainingBits-1); // the -1 is to
leave room for
                realBits--=(remainingBits-1); // the border
                symbol=SBin>>SBinBits;
                SBin^=symbol<<SBinBits;
                symbol=(symbol<<(8-remainingBits+1))+(0x1<<(8-
remainingBits));
                //cout<< hex << symbol << " " << hex << SBin <<
" " ;
                //cout<< endl;
                //outBuffer->put((unsigned char)(symbol&0xff));
                outBuffer->put(symbol);
            //}
            outputCount=0;
            nbrLines++;
            if ((eof && realBits<=0) || nbrLines>=(maxNbrLines-
1))
            {
                doneData=1;
                SBin=0;
                SBinBits=0;
                break;
            }
        }
        else
        {
            SBinBits-=8;
            realBits-=8;
            symbol=SBin>>SBinBits;
            outBuffer->put(symbol);
            SBin^=symbol<<SBinBits;
            if (outputCount==0)
                sideDone=0;
            outputCount++;
        }

        #ifdef _debug_
        cout << "new data: " << symbol << " ";
        binaire(symbol,cout,m);
        cout << endl;
        #endif
    }
}

// Add a line at the bottom minus one bit because the middle section has
// already taken care of the left bit

```

```

while ( doneData && done<charPerLine && outBuffer->testWrite() )
{
    outBuffer->put(0xff);
    done++;
    if (done>=charPerLine)
        nbrLines++;
    /*
    if ( (SBinBits<8 && outputCount!=bytesToOutput) ||
        (SBinBits<remainingBits
outputCount==bytesToOutput) )
    {
        SBin=(SBin<<8)+0xff;
        SBinBits+=8;
    }

    if      (SBinBits>=8      ||      (SBinBits>=remainingBits      &&
outputCount==bytesToOutput) )
    {
        // extract symbol from symbol buffer (SBin):
        if (outputCount==bytesToOutput)
        {
            if (remainingBits!=0)
            {
                SBinBits-=remainingBits;
                symbol=SBin>>SBinBits;
                SBin^=symbol<<SBinBits;
                symbol=symbol<<(8-remainingBits);
                //outBuffer->put((unsigned char)(symbol&0xff));
                outBuffer->put(symbol);
                outputCount=0;
            }
            nbrLines++;
            done=1;
            SBinBits=0;
            SBin=0;
        }
        else
        {
            SBinBits-=8;
            symbol=SBin>>SBinBits;
            outBuffer->put(symbol);
            SBin^=symbol<<SBinBits;
            outputCount++;
        }

        #ifdef _debug_
        cout << "new data: " << symbol << " ";
        binaire(symbol,cout,m);
        cout << endl;
        #endif
    }
}

```

```
        */
    }

    #ifdef _debug_
    cout << endl;
    #endif
}

int pattern1::giveNbrLines()
{
    return nbrLines;
}

void pattern1::newData()
{
}

int pattern1::processSymbols()
{
}
```

A.19 depattern1

```
// depattern1.h
#ifndef _depattern1_h
#define _depattern1_h

#include "coDec.h"
#include <fstream.h>

class depattern1: public coDec
{
    unsigned int SBin; // buffer to build symbols (8bits->mbits)
    int SBinBits; // number of bits in the symbol buffer
    int padBits;
    unsigned int SBout; // buffer to build symbols (8bits->mbits)
    int SBoutBits; // number of bits in the symbol buffer
    int inputCount;
    int outputCount;
    int lastWrite;
    int bitsInBuffer;
    int nbrLines;
    int maxNbrLines;

    int bytesToOutput;
    int nbrBytesOutput;
    int remainingBits;
    int bitsLastChar;
```

```

        int                width;
        int                height;
        int                charPerLine;

        ifstream  input;

        int                feedback,FSRindex,FSRregBit,
                           maxOutput,CBbits;
        unsigned intcharBuffer, output,symbol;
        unsigned chartempChar;
        unsigned charfirstTime;
        int                done;
        int                realBits;

        // the first line was already read in the constructor so we are at
line 2
        int                currentLine,currentChar,currentBit;

        public:
        depattern1(char* filename);
        int processData();
        int giveWidth();
        int giveHeight();
        int processSymbols();

        void newData();
};

#endif

// depattern1.cc
#include "depattern1.h"
#include "debugger.h"
#include "debug.h"
#include <values.h>
#include <stdio.h>

depattern1::depattern1(char* filename):
        SBin(0), SBinBits(0), padBits(0), SBout(0),
        SBoutBits(0),          inputCount(0),          outputCount(0),
bytesToOutput(0),
        nbrBytesOutput(0),          remainingBits(0),          lastWrite(1),
nbrLines(0),

maxNbrLines(0),width(0),height(0),bitsLastChar(0),charPerLine(0),
        feedback(0),FSRindex(0),FSRregBit(0), maxOutput(0),CBbits(0),
        charBuffer(0),  output(0),symbol(0),  tempChar(0),  firstTime(0),
done(0),
        realBits(0), currentLine(2),currentChar(0),currentBit(0)
{
        char buffer[256];

```

```

input.open(filename,ios::in);

if (!input)
{
    cerr << "Couldn't open file " << filename << endl;
    exit(-1);
}

input>> buffer >> width >> height;
// passes the carriage return
input.get();

cout << width << "x" << height << endl;

charPerLine=width/8+(width%8!=0);
// number of bits in the last character. If the number of bits are
multiple of 8
// there are 8 bits not zero in the last char
bitsLastChar=width%8;
if (bitsLastChar==0)
    bitsLastChar=8;

// get rid of the top line
input.read(buffer,charPerLine);

//for (int i=0;i<height;i++)
//{
//    input.read(buffer,charPerLine);
//    for (int j=0;j<charPerLine;j++)
//        for (int k=0;k<8;k++)
//        {
//            if ((buffer[j]&(0x80>>k))!=0)
//                cout << "*";
//            else
//                cout << " ";
//        }
//    cout << endl;
//}

}

int depattern1::processData()
{
    char                c=0;

    if (!outBuffer)
    {
        cerr << "Please assign me with some output "
        << "buffer before asking me to process data" << endl;
        return -1;
    }

```

```

    }

    if ( currentLine==height && SBinBits==0 )
    {
        outBuffer->setEOF();
        return 1;
    }

    while ( (outBuffer->testWrite()) && (currentLine!=height ||
SBinBits!=0))
    {
        while ( SBinBits<8 && currentLine!=height)
        {
            c=(char)input.get();
            currentChar++;
            if (currentChar==1)
            {
                //gets rid of the border
                SBin=(SBin<<7)+(c&0x7F);
                SBinBits+=7;
            }
            else if (currentChar==charPerLine)
            {
                // outputs the bits in the last character of the line
                // of the border
                SBin=(SBin<<(bitsLastChar-1))+(c>>(9-bitsLastChar));
                SBinBits+=bitsLastChar-1;
                currentChar=0;
                currentLine++;
                if (currentLine==height)
                    input.close();
            }
            else
            {
                SBin=(SBin<<8)+c;
                SBinBits+=8;
            }
        }

        // the only time this happens is if the end of pattern is
        reached
        if (SBinBits<8)
        {
            c=(char)(SBin<<(8-SBinBits));
            outBuffer->put(c);
            SBinBits=0;
            done=1;
        }
        else
        {

```

```
        c=(char) (SBin>>(SBinBits-8));
        outBuffer->put(c);
        SBinBits-=8;
    }
}
#ifdef _debug_
cout << endl;
#endif
if ( currentLine==height && SBinBits==0 )
{
    outBuffer->setEOF();
    return 1;
}
}

int depattern1::giveWidth()
{
    return width;
}

int depattern1::giveHeight()
{
    return height;
}

void depattern1::newData()
{
}

int depattern1::processSymbols()
{
}
```

A.20 demodulator

```
#ifndef _demodulator_h
#define _demodulator_h

int demodulator (FILE* input, FILE* output,int pixelSize,int threshold);

#endif // _demodulator_h

#include <iostream.h>
#include <stdio.h>
#include <fstream.h>

#ifdef __cplusplus
```

```
extern "C" {
#endif

#include <pgm.h>
#include <pbm.h>

#ifdef __cplusplus
}
#endif

int demodulator (FILE* input, FILE* output,int pixelSize,int threshold)
{
    unsigned intmaxgray;
    int          format;

    if (pixelSize<=0 || pixelSize>=10)
    {
        cerr << "Pixel size should be between 1 and 10, now it is "
              << pixelSize << endl;
        exit(-3);
    }

    bit          **image;
    int          rows=0,cols=0;
    int          outRows=0,outCols=0;
    char          SBout=0;
    gray         *pixelRows[pixelSize],*outRow;
    unsigned long intavg=0;

    pgm_readpgminit(input,&cols,&rows,&maxgray,&format);
    if (rows==0 || cols==0)
    {
        cerr << "Couldn't read image file" << endl;
        exit(-4);
    }

    for (int i=0;i<pixelSize;i++)
        pixelRows[i]= pgm_allocrow(cols);

    outRows=rows/pixelSize;
    outCols=cols/pixelSize;

    pgm_writepgminit(output,outCols,outRows,maxgray,0);
    outRow=pgm_allocrow(outCols);

    int          rowIndex=0,colIndex=0;
    int          pixelCount=0;
    int          horizOffset,vertOffset;
    char         localThresh=3;
```

```

    for (rowIndex=0; rowIndex<outRows; rowIndex++)
    {
        for (horizOffset=0;horizOffset<pixelSize;horizOffset++)

pgm_readpgmrow(input,pixelRows[horizOffset],cols,maxgray,format);
        for (colIndex=0;colIndex<outCols;colIndex++)
        {
            localThresh=3;
            avg=0;
            for
(horizOffset=colIndex*pixelSize;horizOffset<colIndex*pixelSize+pixelSiz
e;
horizOffset++)
                for (vertOffset=0;vertOffset<pixelSize;vertOffset++)
                {
                    avg+=pixelRows[vertOffset][horizOffset];
                    if (pixelRows[vertOffset][horizOffset]<50)
                        localThresh=1;
                    if (pixelRows[vertOffset][horizOffset]>200)
                        localThresh=0;
                }
                avg=avg/(pixelSize*pixelSize);
                if (threshold < 0)
                    outRow[colIndex]=(gray) avg;
                else if (avg>threshold || localThresh==0)
                    outRow[colIndex]=255;
                else if (avg<=threshold || localThresh==1)
                    outRow[colIndex]=0;
            }
            pgm_writepgmrow(output,outRow,outCols,maxgray,0);
        }
    }

    return 0;
}

```

A.21 noisify

```

#ifndef _noisify_h
#define _noisify_h

int      noisifyPbm(FILE* input,FILE* output, float gaussDev);
void noisify (float factor);
double gauss (void);

#endif // _noisify_h

#include <stdlib.h>

```

```
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <iostream.h>
#include "noisify.h"
#include "gasdev.h"

#ifdef __cplusplus
extern "C" {
#endif

#include <pgm.h>
#include <pbm.h>

#ifdef __cplusplus
}
#endif

int      noisifyPbm(FILE* input, FILE* output, float gaussDev)
{
    unsigned intmaxgray;
    int          format;
    int          rows=0, cols=0;
    gray         *inRow, *outRow;

    pgm_readpgminit(input, &cols, &rows, &maxgray, &format);
    if (rows==0 || cols==0)
    {
        cerr << "Couldn't read image file" << endl;
        exit(-4);
    }

    inRow= pgm_allocrow(cols);

    pgm_writepgminit(output, cols, rows, maxgray, 0);
    outRow=pgm_allocrow(cols);

    int          rowIndex=0, colIndex=0;
    int          pixelCount=0;
    int          tempPixel;

    for (rowIndex=0; rowIndex<rows; rowIndex++)
    {
        pgm_readpgmrow(input, inRow, cols, maxgray, format);
        for (colIndex=0; colIndex<cols; colIndex++)
        {
            tempPixel=(int)inRow[colIndex]+(int)(gaussDev*gauss()*127);
            if (tempPixel<0)
                outRow[colIndex]=0;
            else if (tempPixel>maxgray)
                outRow[colIndex]=maxgray;
            else
                outRow[colIndex]=tempPixel;
        }
    }
}
```

```

        outRow[colIndex]=maxgray;
    else
        outRow[colIndex]=(gray) tempPixel;
    }
    pgm_writepgmrow(output,outRow,cols,maxgray,0);
}

return 0;

}

void noisify (float factor)
{
    unsigned char *src_row, *dest_row;
    unsigned char *src, *dest, *dest_data;
    unsigned int nextIndex=0;
    int noise,i=0;
    int count[1000];

    initSeed();

    for (i=0;i<1000;i++)
        count[i]=0;
    noise = 0;

    for (i=0;i<1000000;i++)
    {
        //printf("%d\n", (int) (gauss()*127));
        //printf("%lf ",gauss());
        nextIndex=500+(int) (factor*gauss()*127);
        //nextIndex=500+(int) (factor*gasdev()*127);
        if (nextIndex>=0 && nextIndex<1000)
            count[nextIndex]++;
    }
    //printf ("\n");
    for (i=0;i<1000;i++)
        if (i==500)
            printf("%d\t%d\n",i-500,count[i]/2);
        else
            printf("%d\t%d\n",i-500,count[i]);
    /*
    noise = (int) (nvals.noise[0] * gauss() * 127);
    p = src[b] + noise;
    if (p < 0)
        p = 0;
    else if (p > 255)
        p = 255;
    dest[b] = p;
    */
}

```

```

/*
 * Return a Gaussian (aka normal) random variable.
 *
 * Adapted from ppmforge.c, which is part of PBMPLUS.
 * The algorithm comes from:
 * 'The Science Of Fractal Images'. Peitgen, H.-O., and Saupe, D. eds.
 * Springer Verlag, New York, 1988.
 */
double gauss (void)
{
    int i;
    double sum = 0.0;

    for (i = 0; i < 4; i++)
        sum += rand () & 0x7FFF;

    return sum * 5.28596089837e-5 - 3.46410161514;
}

```

A.22 binToImg

```

// binToImg.h
#ifndef _binToImg_h
#define _binToImg_h

#include "coDec.h"

class binToImg: public coDec
{
    unsigned int* generator;//indexes of generator polynomial for the
code
    unsigned int* systematic; // systematic symbols buffer.
    unsigned int* FSR;//feedback shift registers for redundant bits
gen.
    unsigned int SBin;// buffer to build symbols (8bits->mbits)
    int SBinBits;//number of bits in the symbol buffer
    int padBits;
    unsigned int SBout;// buffer to build symbols (8bits->mbits)
    int SBoutBits;//number of bits in the symbol buffer
    int inputCount;
    int outputCount;
    int lastWrite;
    int bitsInBuffer;
    int nbrLines;

    int bytesToOutput;
    int nbrBytesOutput;

```



```
int remainingBits;

int          feedback,FSRindex,FSRregBit,
             maxOutput,CBbits;
unsigned int charBuffer, output,symbol;
unsigned char tempChar;
int          realBits;

public:
binToImg(int nbrCol);
int processData();
int giveNbrLines();
int processSymbols();

void newData();
};

#endif

// binToImg.cc
#include "binToImg.h"
#include "debugger.h"
#include "debug.h"

binToImg::binToImg(int nbrCols): generator(0),
    systematic(0), FSR(0), SBin(0), SBinBits(0), padBits(0),
    SBout(0), SBoutBits(0), inputCount(0), outputCount(0),
    bytesToOutput(0), nbrBytesOutput(0), remainingBits(0), lastWrite(1),
    nbrLines(0), feedback(0),FSRindex(0),FSRregBit(0), maxOutput(0),CBbits(0),
    charBuffer(0), output(0),symbol(0), tempChar(0), realBits(0)
{
    int nbrBits;

    bytesToOutput=nbrCols/8;// determine the number of bytes that the
    remainingBits=nbrCols%8;// cols fits in
}

int binToImg::processData()
{
    if (!inBuffer || !outBuffer)
    {
        cerr << "Please assign me with some input and output "
            << "buffers before asking me to process data" << endl;
        return -1;
    }
}
```

```

dataInput_Process:
eof=inBuffer->atEOF();
if ( eof && (realBits<=0) )
{
    outBuffer->setEOF();
    return 1;
}

while ( (outBuffer->testWrite()) &&
        ( inBuffer->testRead() || (eof=inBuffer-
>atEOF()) ) )
{
    if ( (SBinBits<8 && outputCount!=bytesToOutput) ||
        (SBinBits<remainingBits                                &&
outputCount==bytesToOutput) )
    {
        // padBits flags when the system started padding the
information
        // (when there is not enough info to fill info block so 0
pad)
        if (eof)
            SBin=(SBin<<8)+0x0;
        else
        {
            inBuffer->get(tempChar);
            SBin=(SBin<<8)+tempChar;
            realBits+=8;
        }
        SBinBits+=8;
    }

    if (SBinBits>=8 || (SBinBits>=remainingBits &&
outputCount==bytesToOutput) )
    {
        // extract symbol from symbol buffer (SBin):
        if (outputCount==bytesToOutput)
        {
            if (remainingBits!=0)
            {
                SBinBits-=remainingBits;
                realBits-=remainingBits;
                symbol=SBin>>SBinBits;
                SBin^=symbol<<SBinBits;
                symbol=symbol<<(8-remainingBits);
                //outBuffer->put((unsigned char)(symbol&0xff));
                outBuffer->put(symbol);
            }
            outputCount=0;
            nbrLines++;
            if (eof&&realBits<=0)

```

```

        break;
    }
    else
    {
        SBinBits-=8;
        realBits-=8;
        symbol=SBin>>SBinBits;
        outBuffer->put(symbol);
        SBin^=symbol<<SBinBits;
        outputCount++;
    }

    #ifdef _debug_
    cout << "new data: " << symbol << " ";
    binaire(symbol,cout,m);
    cout << endl;
    #endif
}

/*
    output the content of the Feedback Shift Register
*/
#ifdef _debug_
cout << endl;
#endif
}

int binToImg::giveNbrLines()
{
    return nbrLines;
}

void binToImg::newData()
{
}

int binToImg::processSymbols()
{
}

```

A.23 rand_test

```

// rand_test.h
#ifndef _rand_test_h
#define _rand_test_h

double xsq_to_stddev(double xsq, double k, double n,int *approx);

```

```

int print2digits(double f);
void diagnostic(double xsq, double k, double n);

#endif

// rand_test.cc
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <time.h>
#include "ran4.h"
#include "gasdev.h"
#include <pLab.h>
#include <values.h>
#include <math.h>
#include "rand_test.h"

#define NBRIT 700000
#define LOG2PI 1.837877 /* log(2*pi) */
#define min(x,y) ( ((x)<(y)) ? (x) : (y) )
#define max(x,y) ( ((x)>(y)) ? (x) : (y) )

/* Evaluate an X-squared variate xsq (for n items placed in k bins,
 * with all bins equi-probable), returning the number of standard
 * deviations the equivalent normal deviate differs from its
 * expectation.
 * That is, if this function returns -2.326, the xsq value is at the
 * p=0.01 point of its distribution.
 *
 * The value of *approx is set to 1 if the calculated stddev has more
 * than
 * 10% relative error.
 *
 * See Thomborson's SODA '93 submission for an explanation of this code.
 */
double xsq_to_stddev(double xsq, double k, double n, int *approx)
{
    double stddev, v, wsq, w, a, keff, twologp, u;

    xsq = xsq + k/n; /* apply a "continuity correction" */
    v = k-1.0; /* degrees of freedom in Chi-squared approximation */
    /* Wallace's approximation */
    wsq = xsq - v - v*log(xsq/v);
    if (wsq<0.0) wsq=0.0; /* avoid sqrt(-0.0), as a result of round-off
errors */
    w = sqrt(wsq);
    a = sqrt(2.0/v)/3.0;
    if (xsq <= v) { /* lower tail: use Wallace's approx */
        stddev = -w+a;
    } else { /* upper tail: min(multinomial approx, Wallace's approx) */

```

```

    keff = k/(1.0+xsq/n) * (1.0 - exp(-(xsq+n)/k));
    twologp = (2.0*n - 1.0 - 2.0*keff) * log(k/keff) - 2.0*keff + LOG2PI;
    /* if twologp is small, make it huge (so it won't affect the output)
    */
    if (twologp < 2.83) twologp = 1000.0;
    /* convert 2*log(p) to (approx.) quantile of normal distribution */
    u = sqrt(twologp - log(twologp - LOG2PI) - LOG2PI);
    stddev = min(u,w+a);
  }
  if (fabs(stddev) > n/sqrt(2.0*v)) {
    *approx = 1;
  } else {
    *approx = 0;
  }
  return(stddev);
}

/* Print a low-precision float, using "%f" format where possible.
 * Uses "%f" format if this would print at most 3 significant digits;
 * otherwise uses "%.2g". Always prints at least 2 significant digits,
 * never prints more than 3.
 */
int print2digits(double f)
{
  double a;
  a = fabs(f);
  if (a > 1000.0)
    printf("%.2g", f);
  else if (a > 10.0)
    printf("%.0f", f);
  else if (a > 1.0)
    printf("%.1f", f);
  else if (a > 0.1)
    printf("%.2f", f);
  else if (a > 0.01)
    printf("%.3f", f);
  else
    printf("%.2g", f);
}

void
diagnostic(double xsq, double k, double n)
{
  double stddev; /* temp for calculating # of std deviations */
  int i; /* loop counter */
  int robust; /* true if we can do a 2-tailed test with p < 1e-6 */
  int accurate; /* true if the stddev estimate is accurate */
  double nrobust, ngiveup; /* min n to give (robust,adequate) test
  results */
  double smallxsq; /* temp for nrobust and ngiveup calculations */

```

```

/* There are two tests for small n. The less restrictive test, below,
* checks whether the computed, continuity-corrected xsq value
* could ever fall below the 5% point on the Chi-squared distribution.
* In all cases, if there is a xsq outcome below the 5% point, there
* are (many) above the 95% point, so a two-sided test is reasonably
* accurate.
*
* The smallest xsq value is  $(n \bmod k)(k - (n \bmod k))/n$ . For the case
* of  $n > k$ , a monotone envelope is
*  $\text{smallestxsq1} = k^2/(4*n)$ 
* For the case of  $n \leq k$ , the smallest xsq value is just
*  $\text{smallestxsq2} = k - n$ 
* We thus use  $\max(\text{smallestxsq1}, \text{smallestxsq2}) =$ 
*  $\text{smallxsq} = \max(k-n, k*k/(4*\max(n,k)))$ 
* as a monotone envelope for this function.
*
* My "ngiveup" calculation evaluates smallxsq for  $n =$ 
 $(3*\sqrt{k})*(1.1)^i$ ,
* for  $i=0,1,\dots$  until the resulting xsq_to_stddev value falls below -
1.645.
* The factor of 1.1 implies that we'll overshoot the smallest possible
* value of ngiveup by at most 10%. (Alternatively, we could
symbolically
* invert the calculations in the xsq_to_stddev() function, perhaps
* approximating in terms of a power series. This would take a lot
* of my time: I'd rather let your CPU do a bit of extra work when
* evaluating ngiveup.)
*
* For large k, ngiveup is  $3*\sqrt{k}$ . For very small k, ngiveup
* is moderately large: 725 at  $k=2$ , 51 at  $k=3$ , 24 at  $k=4$ , 17 at  $k=5$ ,
* 13 at  $k=6$ , ..., 11 at  $k=10$ , then rising with  $3*\sqrt{k}$  for  $k>10$ .
*/

ngiveup = ceil(3.0*sqrt(k));
for (i=0; i<500; i++) {
    smallxsq = max(k-ngiveup, k*k/(4*max(ngiveup,k)));
    if (xsq_to_stddev(smallxsq,k,ngiveup,&accurate) < -1.645) break;
    ngiveup = ceil(ngiveup*1.1);
}

/* My nrobust calculation is similar to the ngiveup calculation,
* except that the cutoff probability is  $0.5*10^{-6}$ , i.e. the
* stddev value must fall below -4.892. This allows an xsq value
* to be rejected at the 1-in-a-million level.
*
* is 60; for  $k=10$ , nrobust is 182; for  $k=5$ , nrobust is about
* 5600; for  $k=3$ , nrobust is approximately  $10^7$ ; for  $k=2$ ,
* nrobust is about  $3*10^{13}$ . For  $k \geq 50$ ,  $\text{nrobust} < k$ .
*

```

```

* Perhaps I should use only a 1-tailed test on xsq for k<4,
* since a 2-tailed test with small failure probability requires
* an ``unreasonably-large'' amount of data.
*
* The statistician's "rule of thumb" for Chi-squared testing
* is  $n \geq 5*k$ , which is much larger than my nrobust for large k,
* and much smaller than my ngiveup for small k. I have found
* no reason to test for  $n \geq 5*k$ . Indeed, Kendall and Stuart
* (Vol 2, 2nd ed, p. 440) say there is "No general theoretical
* basis for this rule."
*/

nrobust = ceil(7.0*sqrt(k));
for (i=0; i<500; i++) {
    smallxsq = max(k-nrobust, k*k/(4*max(nrobust,k)));
    if (xsq_to_stddev(smallxsq,k,nrobust,&accurate) < -4.892) break;
    nrobust = ceil(nrobust*1.1);
}

/* quit if sample is too small */
if (n<ngiveup) {
    printf(" This test requires ");
    print2digits(ngiveup);
    printf(" or more random generates.\n");
    return;
}

if (n<nrobust) {
    printf(" Note: at least ");
    print2digits(nrobust);
    printf(" random generates are needed\n");
    printf(" for a two-tailed test with confidence  $p > 1-1.0e-6$ \n");
};

ChiSquareDistributionchiSquare(k-1);
printf("\tX^2=");
print2digits(xsq);
printf("\n\tDeg Freedom=%.0f",k-1);
printf("\n\tprobability of happening=");
double percent=100*chiSquare.distributionFunctionAt(&xsq);
if (percent>50)
    percent=100-percent;
print2digits(percent);
printf("%%\n");
}

main(int argc, char** argv)
{
    long int counter=1;

```

```

unsigned long result;
float          result2;
double         count[3];
//ChiSquareDistributionchiSquare2(2);
double         countup8[8];
//ChiSquareDistributionchiSquare7(7);
double         equid[1<<10];
double         pairs[16][16];
int            prev;
double         xsq;
int            i,j;
double         m=(double)MAXLONG+1;
long           n=NBRT;
double         freq;

//printf("%x %lf\n",m,m);
count[0]=0;
count[1]=0;
count[2]=0;
for (i=0;i<8;i++)
    countup8[i]=0;
for (i=0;i<(1<<10);i++)
    equid[i]=0;
for (i=0;i<16;i++)
    for (j=0;j<16;j++)
        pairs[i][j]=0;
long int randSeed;
time(&randSeed);
result=randSeed;
//randSeed=0x389890bb;
srand4(randSeed);
srand48(randSeed);
printf("seed: %x\n",randSeed);

int tmp;
while(counter<(NBRT+1))
{
    //srand(result);
    //result=rand();
    result=lrnd4();
    //result=lrnd48();
    counter++;
    count[result%3]++;
    //countup8[result>>29]++;
    countup8[(result>>28)&7]++;
    equid[(result>>21)&0x3ff]++;
    if (counter%2)
        pairs[prev][(result>>26)&0xff];
        //pairs[1][2]++;
    else
        prev=(result>>26)&0xff;
}

```

```

    }
    xsq=0;
    /*freq = n*(ceil(m/3.))/m;
    xsq = ((count[0]-freq)*(count[0]-freq))/freq;
    printf("%lf %lf %lf\n",freq,xsq,count[0]);
    freq = n*(ceil(2.*m/3.))/m - freq;
    xsq += ((count[1]-freq)*(count[1]-freq))/freq;
    printf("%lf %lf %lf\n",freq,xsq,count[1]);
    freq = n-n*(ceil(2.*m/3.))/m;
    xsq += ((count[2]-freq)*(count[2]-freq))/freq;
    printf("%lf %lf %lf\n",freq,xsq,count[2]);*/
    //interpret_xsq(sumfsq,3.0,dn);
    freq=((double)n)*1/3;
    for (i=0;i<3;i++)
        xsq+=((count[i]-freq)*(count[i]-freq))/(freq);
    printf("Test mod 3:\n");
    diagnostic(xsq,3,n);
    xsq=0;
    for (i=0;i<8;i++)
    {
        //printf("%.0lf ",countup8[i]);
        xsq+=(countup8[i]-NBRIT/8)*(countup8[i]-NBRIT/8)/(NBRIT/8);
    }
    printf("\nTest 3 MSB:\n");
    diagnostic(xsq,8,n);
    printf("\nEquidistribution of the 10 MSB\n");
    for (i=0;i<(1<<10);i++)
    {
        //printf("%.0lf ",equid[i]);
        xsq+=(equid[i]-NBRIT/(1<<10))*(equid[i]-NBRIT/(1<<10))/(NBRIT/
(1<<10));
    }
    diagnostic(xsq,1<<10,n);
    for(i=0;i<16;i++)
        for(j=0;j<16;j++)
        {
            printf("%.0lf ",pairs[i][j]);
            xsq+=(equid[i]-NBRIT/(1<<10))*(equid[i]-NBRIT/(1<<10))/
(NBRIT/(1<<10));
        }
    diagnostic(.087,2.0,n);
}
}

```

A.24 treatXsqFile

```
// /home/adenis/Thesis/source/treatXsqFile.main.cc
```

```
#include <pLab.h>
#include <iostream.h>
#include <stdio.h>
#include <fstream.h>
#include <math.h>

#define MAX_SIZE 1024

main(int argc, char** argv)
{
    if (argc!=5)
    {
        cerr << "Usage " << argv[0] << " filename.head filename.data
output_filename degree_freedom" << endl;
        exit (-1);
    }

    ifstreamhead(argv[1],ios::in);
    ifstreamdata(argv[2],ios::in);
    ofstreamoutput(argv[3],ios::out);
    char        buffer[MAX_SIZE];
    float        xsq,percent;
    float        first,last;
    unsigned int  number=55,degFree;

    sscanf(argv[4],"%i",&degFree);

    if (!data || !head || !output)
    {
        cerr << "Couldn't open file one of the files" << endl;
        exit (-2);
    }

    //data.getline(buffer,MAX_SIZE);

    head >> first >> last >> number;
    if (number>100)
        number=100;

    cout << first << " " << last << " " << number << endl;

    float        binSize=(last-first)/(number-2);
    int           i;

    cout << "bin size=" << binSize << endl;

    //for (i=0;i<number;i++)
    //    cout << first+i*binSize << " ";
    //cout<< endl;

    float        yXsq[number],essai=0;
```

```

    int                counter;

    for (int i=0;i<number;i++)
        yXsq[i]=0;

    do
    {
        data.getline(buffer,MAX_SIZE);
        if (data.eof() || data.bad())
            break;
        //cout<< buffer << endl;
        sscanf(buffer,"%f",&xsq);
        //cerr<< xsq << endl;
        //cout<< "Data: " << xsq << " " << (xsq-first)/binSize << endl;
        //cout<< "Before int: " << floor((xsq-first)/binSize) << endl;
        yXsq[(int)floor((xsq-first)/binSize)]++;
        //cout<< "yXsq[" << counter << "]= " << xsq << endl;
    } while (1);

    ChiSquareDistributiondistrib(degFree);
    double                position;

    for (i=1;i<number;i++)
        yXsq[i]+=yXsq[i-1];

    for (i=0;i<number;i++)
    {
        output<< (position=first+i*binSize) << "\t"
            << yXsq[i]/yXsq[number-1] << "\t"
            << distrib.distributionFunctionAt(&position)
            << endl;
    }

    data.clear();
    data.close();
    head.close();
    output.close();
}

```

A.25 scannerSimulator

```

#ifndef _scannerSimulator_h
#define _scannerSimulator_h

int scannerSimulator(FILE* input,FILE* output,float Xoff,float Yoff);

#endif // _scannerSimulator_h

#include <iostream.h>

```

```
#include <stdio.h>

#ifdef __cplusplus
extern "C" {
#endif

#include <pgm.h>
#include <pbm.h>

#ifdef __cplusplus
}
#endif

int scannerSimulator(FILE* input, FILE* output, float Xoff, float Yoff)
{
    unsigned int maxgray;
    int          format;
    int          rows=0, cols=0;
    gray         *inRow[2], *outRow[2];
    float        convol[9];
    int          i, j;
    int          one, two;

    if (Xoff>1.0f || Yoff>1.0f)
    {
        cerr << "One of the offsetd is greater than 1 please try
again." << endl;
        exit(-1);
    }

    pgm_readpgm_init(input, &cols, &rows, &maxgray, &format);
    if (rows==0 || cols==0)
    {
        cerr << "Couldn't read image file" << endl;
        exit(-4);
    }
    inRow[0] = pgm_allocrow(cols+2);
    inRow[1] = pgm_allocrow(cols+2);

    for (i=0; i<cols+2; i++)
        inRow[0][i] = 255;
    inRow[1][0] = 255;
    inRow[1][cols+1] = 255;

    pgm_writepgm_init(output, cols*2+1, rows*2+1, maxgray, 0);
    outRow[0] = pgm_allocrow(cols*2+1);
    outRow[1] = pgm_allocrow(cols*2+1);

    convol[0] = (1-Xoff)*(1-Yoff);
    convol[1] = 1-Yoff;
    convol[2] = Xoff*(1-Yoff);
```

```
convol[3]=1-Xoff;
convol[4]=1;
convol[5]=Xoff;
convol[6]=(1-Xoff)*Yoff;
convol[7]=Yoff;
convol[8]=Xoff*Yoff;

int      rowIndex=0,colIndex=0;
int      pixelCount=0;
int      tempPixel,tempIndex;

two=1;
one=0;

for (rowIndex=0; rowIndex<rows+1; rowIndex++)
{
    if (rowIndex==rows)
        for (i=0;i<cols+2;i++)
            inRow[two][i]=255;
    else

pgm_readpgmrow(input,&(inRow[two][1]),cols,maxgray,format);

        for (colIndex=0;colIndex<cols+1;colIndex++)
        {

outRow[0][colIndex*2]=(gray)(convol[7]*inRow[one][colIndex]+
convol[1]*inRow[two][colIndex]);

outRow[0][colIndex*2+1]=(gray)(convol[8]*inRow[one][colIndex]+
convol[6]*inRow[one][colIndex+1]+
convol[2]*inRow[two][colIndex]+
convol[0]*inRow[two][colIndex+1]);

outRow[1][colIndex*2]=(gray)(convol[4]*inRow[two][colIndex]);

outRow[1][colIndex*2+1]=(gray)(convol[5]*inRow[two][colIndex]+
convol[3]*inRow[two][colIndex+1]);
        }
        pgm_writepgmrow(output,outRow[0]+1,2*cols+1,maxgray,0);
        if (rowIndex!=rows)
            pgm_writepgmrow(output,outRow[1]+1,2*cols+1,maxgray,0);



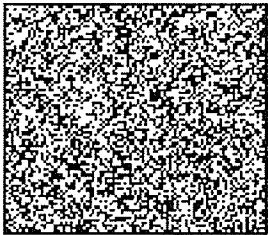





        tempIndex=one;
        one=two;
    }
}
```

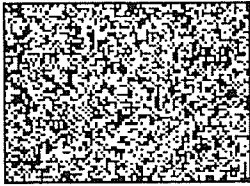
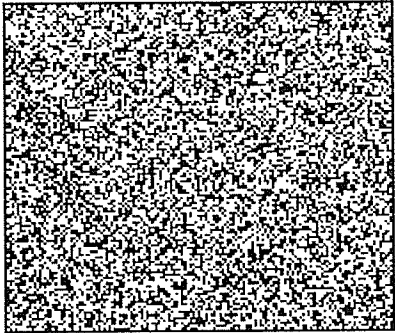










```
        two=tempIndex;  
    }  
    return 0;  
}
```


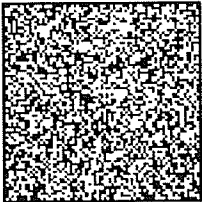
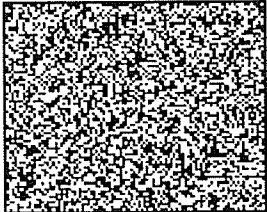
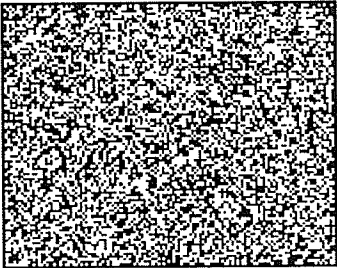
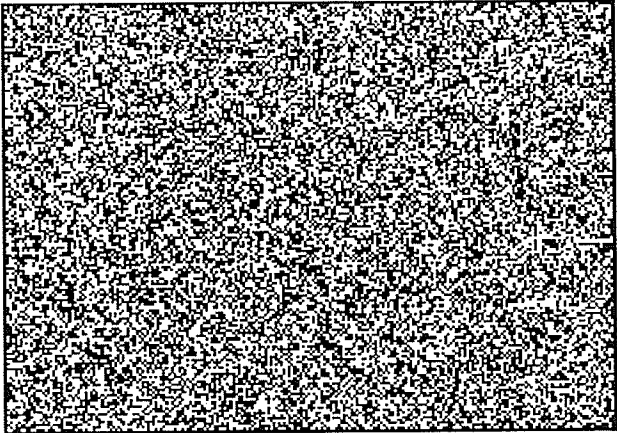




APPENDIX B








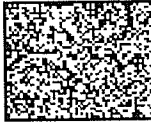






PATTERNS


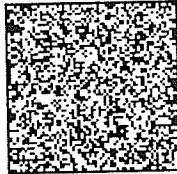
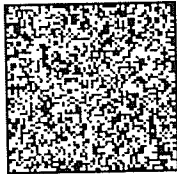

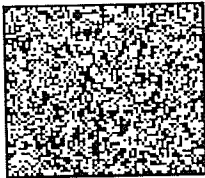
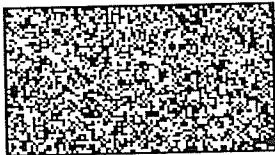

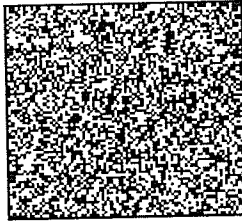
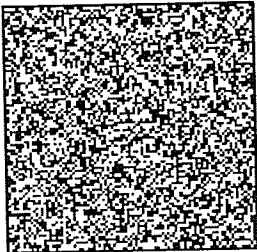
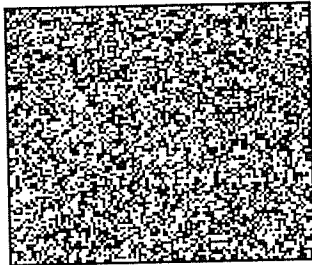
The following are the patterns using the concatenated code with symbol interleaver of the size of the pattern, a random interleaver, the convolutional code with constraint length 18 and the RS code with parameters indicated on top of each pattern.

3-1 	3-2 
3-3 	4-1 
4-2 	4-3 
4-4 	4-5 

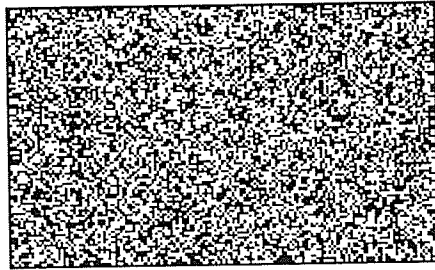
<p>4-6</p> 	<p>4-7</p> 
<p>5-1</p> 	<p>5-2</p> 
<p>5-3</p> 	<p>5-4</p> 
<p>5-5</p> 	<p>5-6</p> 
<p>5-7</p> 	<p>5-8</p> 
<p>5-9</p> 	<p>5-10</p> 

<p>5-11</p> 	<p>5-12</p> 
<p>5-13</p> 	<p>5-14</p> 
<p>5-15</p> 	
<p>6-1</p> 	<p>6-2</p> 
<p>6-3</p> 	<p>6-4</p> 

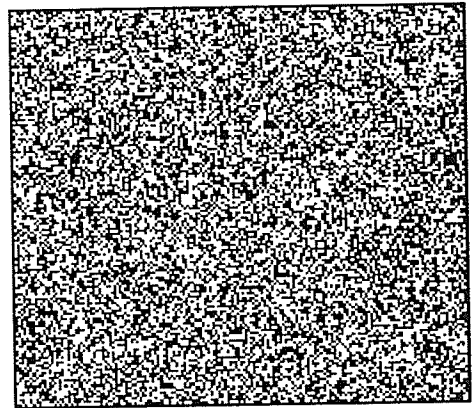
6-5 	6-6 
6-7 	6-8 
6-9 	6-10 
6-11 	6-12 
6-13 	6-14 
6-15 	6-16 
6-17 	6-18 

6-19 	6-20 
6-21 	6-22 
6-23 	6-24 
6-25 	6-26 
6-27 	6-28 

6-29



6-30



6-31

