Informatics

Digital Representation

Claudio Sartori
Department of Computer Science and Engineering
claudio.sartori@unibo.it
https://www.unibo.it/sitoweb/claudio.sartori/

Learning Objectives

- Explain the link between patterns, symbols, and information
- Determine possible binary encodings using a physical phenomenon
- Encode and decode ASCII
- Represent numbers in binary form
- Compare two different encoding methods
- Explain how structure tags (metadata) encode the Oxford English Dictionary

Digitizing Discrete Information

- The dictionary definition of digitize is to represent information with digits
- Digit means the ten Arabic numerals 0 through 9

Limitation of Digits

- A limitation of the dictionary definition of digitize is that it calls for the use of the ten digits, that produces a whole number
  - Digitizing in computing can use almost any symbol
- Having a bigger telephone number does not make you a better person

Alternative Representations

Digitizing can use almost any symbols

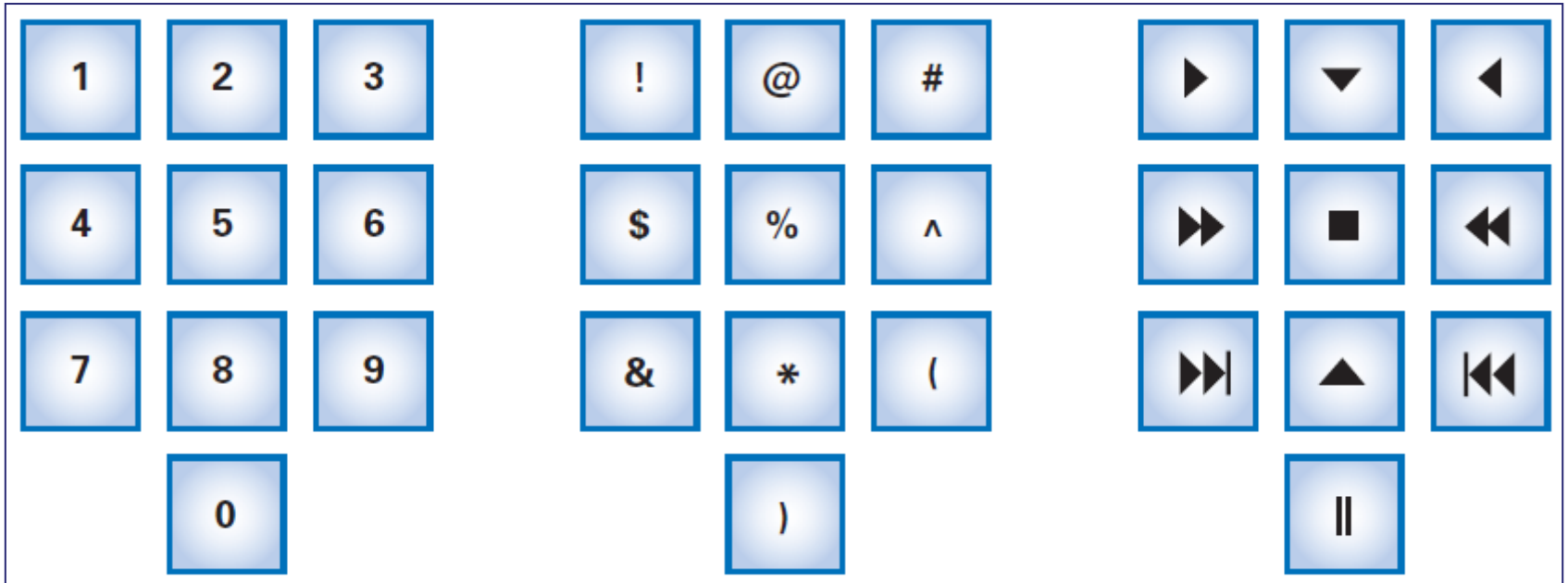


Figure 7.1 Three symbol assignments for a telephone keypad.

Symbols, Briefly

- One practical advantage of digits is that digits have short names (one, two, etc)
- Imagine speaking your phone number the multiple syllable names:
  - "asterisk, exclamation, closing parenthesis"
- IT uses these symbols, but have given them shorter names:
  - exclamation point . . . is ***bang***
  - asterisk . . . is ***star***

Ordering Symbols

- Another advantage of digits is that the items can be listed in numerical order
- Sometimes ordering items is useful
- ***collating sequence:*** placing information in order by using non-digit symbols
  - need to agree on an ordering for the basic symbols
- Today, digitizing means representing information by symbols

Fundamental Information Representation

- A computer represents information by the presence or absence of some physical phenomenon
- This gives two symbols: a binary system
- We name the two states 1 and 0
- We can then build larger symbols using these two basic ones
- The phenomenon can be charge, current, magnetization, or many other things

The Binary Representation

- Only two states are available
- Such a formulation is said to be discrete
- Discrete means "***distinct***" or "***separable***"
  - It is not possible to transform one value into another by tiny gradations
  - There are no "**shades of gray**"

A Binary System

- The binary encoding has two patterns: present and absent
- Two patterns make it a binary system
- There is ***no*** law that says:
  - ***on*** means "***present***"
    off means "***absent***"
- the assignment is arbitrary

Present	Absent
True	False
1	0
On	Off
Yes	No
+	–
Black	White
For	Against
. . .	. . .

Bits Form Symbols

- *In the binary representation, the unit is a specific place (in space and time), where the presence or absence of the phenomenon can be set and detected.*
- The binary unit is known as a ***bit***
- ***Bit*** is a contraction for "***b***inary dig***it***"
- Bit sequences can be interpreted as binary numbers
- Groups of bits form symbols

Bits in Computer Memory

- Memory is arranged inside a computer in a very long sequence of bits
  - the physical phenomenon can be encoded, the information can be ***set*** and ***detected*** to ***present*** or ***absent***

Alternative binary Encodings

- There is no limit to the ways to encode two physical states
  - stones on all squares, but with white (absent) and black (present) stones for the two states
  - multiple stones of two colors per square, more white stones than black means 1 and more black stones than white means 0
  - And so forth

Combining Bit Patterns

- The two-bit patterns gives limited resources for digitizing information
- Only two values can be represented
- The two patterns must be combined into sequences to create enough symbols to encode the intended information

Table 7.2 Number of symbols possible from *n* bits in sequence

n	2^n	Symbols
1	2^1	2
2	2^2	4
3	2^3	8
4	2^4	16
5	2^5	32
6	2^6	64
7	2^7	128
8	2^8	256
9	2^9	512
10	2^{10}	1,024

Binary Explained

- Computers use base-2 to represent numbers using the ***binary number system***
- When counting in binary you are limited to only use 0 and 1
  - 0, 1, 10, 11, 100, 101, 110, 111, 1000, …

Hex Explained

- ***Hex*** digits, short for ***hex***adecimal digits, are base-16 numbers
- Uses decimal numbers, and then the first six Latin letters
  - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F
- There needed to be a better way to write bit sequences…hexadecimal digits

Changing Hex to Binary

- The 32 bits below represent a computer instruction

  1000 1110 1101 1000 1010 0011 1010 0000
- Writing so many 0's and 1's is tedious and error prone
- We can convert each each four-bit group to hex, giving the shorter version:

  8E D8 A3 A0

Hexadecimal Digits

Each hex digit codes a four-bit group:

0000	0	0100	4	1000	8	1100	C
0001	1	0101	5	1001	9	1101	D
0010	2	0110	6	1010	A	1110	E
0011	3	0111	7	1011	B	1111	F

Digitizing Numbers in Binary

- The two earliest uses of binary digits were to:
  - Encode numbers
  - Encode keyboard characters
- Representations for sound, images, video, and other types of information are also important

Place Value in a Decimal Number

- Recall that To find the quantity expressed by a decimal number:
  - The digit in a place is multiplied by the place value and the results are added
- Example, 1,010 (base 10) is:
  - Digit in the 1's place is multiplied by its place
  - Digit in the 10's place is multiplied by its place
  - and so on:
    $(0 \times 1) + (1 \times 10) + (0 \times 100) + (1 \times 1000)$

Place Value in a Binary Number

- Binary works the same way
- The base is 2 instead of 10
- Instead of the decimal place values: 1, 10, 100, 1000, . . . , the binary place values are: 1, 2, 4, 8, 16, . . . ,

Power	Decimal	Binary
0	$1 = 10^0$	$1 = 2^0$
1	$10 = 10^1$	$2 = 2^1$
2	$100 = 10^2$	$4 = 2^2$
3	$1000 = 10^3$	$8 = 2^3$
4	$10{,}000 = 10^4$	$16 = 2^4$
. . .	. . .	. . .

Place Value in a Binary Number

- 1010 in binary:
  - (1 × 8) + (0 × 4) + (1 × 2) + (0 × 1)

Table 7.5 The binary number 1010, representing the decimal number ten = 8 + 2

2^3	2^2	2^1	2^0	Binary Place Values
1	0	1	0	Bits of Binary Number
1×2^3	0×2^2	1×2^1	0×2^0	Multiply place bit by place value
8	0	2	0	and add to get a decimal 10

Table 7.6 Binary representation of the decimal number one thousand ten = 11 1111 0010

2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	Binary Place Values
1	1	1	1	1	1	0	0	1	0	Bits of Binary Number
1×2^9	1×2^8	1×2^7	1×2^6	1×2^5	1×2^4	0×2^3	0×2^2	1×2^1	0×2^0	Multiply place bit by place value
512	256	128	64	32	16	0	0	2	0	and add to get decimal 1,010

Digitizing Text

- The number of bits determines the number of symbols available for representing values:
  - *n bits in sequence yield 2^n symbols*
- The more characters you want encoded, the more symbols you need

Digitizing Text

- Roman letters, Arabic numerals, and about a dozen punctuation characters are the minimum needed to digitize ***English*** text
- What about:
  - Basic arithmetic symbols like +, −, *, /, =?
  - Characters not required for English ö, é, ñ, ø?
  - Punctuation? « », ¿, π, ∀)? What about business symbols: ¢, £, ¥, ©, and ®?

Assigning Symbols

- We need to represent:
  - 26 uppercase,
  - 26 lowercase letters,
  - 10 numerals,
  - 20 punctuation characters,
  - 10 useful arithmetic characters,
  - 3 other characters (new line, tab, and backspace)
  - 95 symbols…enough for English

Assigning Symbols

- To represent 95 distinct symbols, we need 7 bits
  - 6 bits gives only $2^6 = 64$ symbols
  - 7 bits give $2^7 = 128$ symbols
- 128 symbols is ample for the 95 different characters needed for English characters
- Some additional characters must also be represented

Assigning Symbols

- ***ASCII*** stands for American Standard Code for Information Interchange
- ASCII is a widely used 7-bit (2^7) code
- Advantages of a “standard”:
  - Computer parts built by different manufacturers can be connected
  - Programs can create data and store it so that other programs can process it later, and so forth

Extended ASCII: An 8-Bit Code

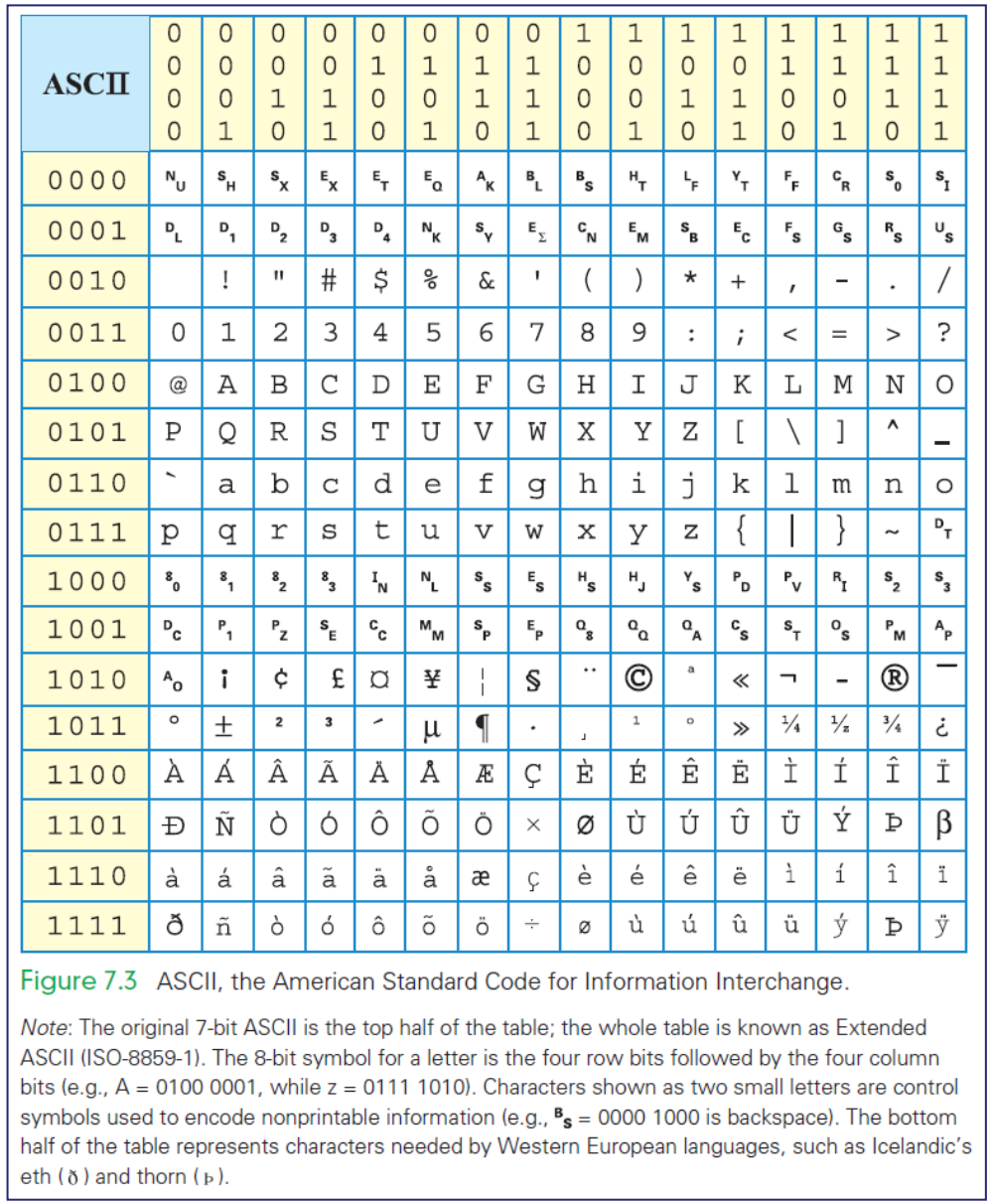
- 7-bit ASCII is not enough, it cannot represent text from other languages
- IBM decided to use the next larger set of symbols, the ***8***-bit symbols (2^{8})
- Eight bits produce $2^8 = 256$ symbols
  - The 7-bit ASCII is the 8-bit ASCII representation with the leftmost bit set to 0
  - Handles many languages that derived from the Latin alphabet

Extended ASCII: An 8-Bit Code

- IBM gave 8-bit sequences a special name, ***byte***
- It is a standard unit for computer memory

Unicode

- The 256 extended ASCII codes cover most Western languages
- Unicode represents many more characters by using up to 32 bits to code characters
- UTF-8 records Unicode by writing long characters as groups of bytes



ASCII	0000 0000	0000 0001	0000 0010	0000 0011	0000 0100	0000 0101	0000 0110	0000 0111	0000 1000	0000 1001	0000 1010	0000 1011	0000 1100	0000 1101	0000 1110	0000 1111
0000	$^{N}_{U}$	$^{S}_{H}$	$^{S}_{X}$	$^{E}_{X}$	$^{E}_{T}$	$^{E}_{Q}$	$^{A}_{K}$	$^{B}_{L}$	$^{B}_{S}$	$^{H}_{T}$	$^{L}_{F}$	$^{Y}_{T}$	$^{F}_{F}$	$^{C}_{R}$	$^{S}_{0}$	$^{S}_{I}$
0001	$^{D}_{L}$	$^{D}_{1}$	$^{D}_{2}$	$^{D}_{3}$	$^{D}_{4}$	$^{N}_{K}$	$^{S}_{Y}$	$^{E}_{\Sigma}$	$^{C}_{N}$	$^{E}_{M}$	$^{S}_{B}$	$^{E}_{C}$	$^{F}_{S}$	$^{G}_{S}$	$^{R}_{S}$	$^{U}_{S}$
0010		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0011	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0100	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0101	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0110	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0111	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	$^{D}_{T}$
1000	$^{8}_{0}$	$^{8}_{1}$	$^{8}_{2}$	$^{8}_{3}$	$^{I}_{N}$	$^{N}_{L}$	$^{S}_{S}$	$^{E}_{S}$	$^{H}_{S}$	$^{H}_{J}$	$^{Y}_{S}$	$^{P}_{D}$	$^{P}_{V}$	$^{R}_{I}$	$^{S}_{2}$	$^{S}_{3}$
1001	$^{D}_{C}$	$^{P}_{1}$	$^{P}_{Z}$	$^{S}_{E}$	$^{C}_{C}$	$^{M}_{M}$	$^{S}_{P}$	$^{E}_{P}$	$^{O}_{S}$	$^{O}_{Q}$	$^{O}_{A}$	$^{C}_{S}$	$^{S}_{T}$	$^{O}_{S}$	$^{P}_{M}$	$^{A}_{P}$
1010	$^{A}_{o}$	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯
1011	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
1100	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
1101	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	β
1110	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
1111	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Figure 7.3 ASCII, the American Standard Code for Information Interchange.

Note: The original 7-bit ASCII is the top half of the table; the whole table is known as Extended ASCII (ISO-8859-1). The 8-bit symbol for a letter is the four row bits followed by the four column bits (e.g., A = 0100 0001, while z = 0111 1010). Characters shown as two small letters are control symbols used to encode nonprintable information (e.g., $^{B}_{S}$ = 0000 1000 is backspace). The bottom half of the table represents characters needed by Western European languages, such as Icelandic's eth (ð) and thorn (þ).

Advantages of Long Encodings

- With computing, we usually try to be efficient by using the shortest symbol sequence to minimize the amount of memory
- Examples of the opposite:
  - NATO Broadcast Alphabet
  - Bar Codes

NATO Broadcast Alphabet

- The code for the letters used in radio communication is ***purposely*** inefficient
- The code is distinctive when spoken amid **noise**
- The alphabet encodes letters as words
  - Words are the symbols
  - “Mike” and “November” replace “em” and “en”
- The longer encoding improves the chance that letters will be recognized
- Digits keep their usual names, except nine, that is known as ***niner***

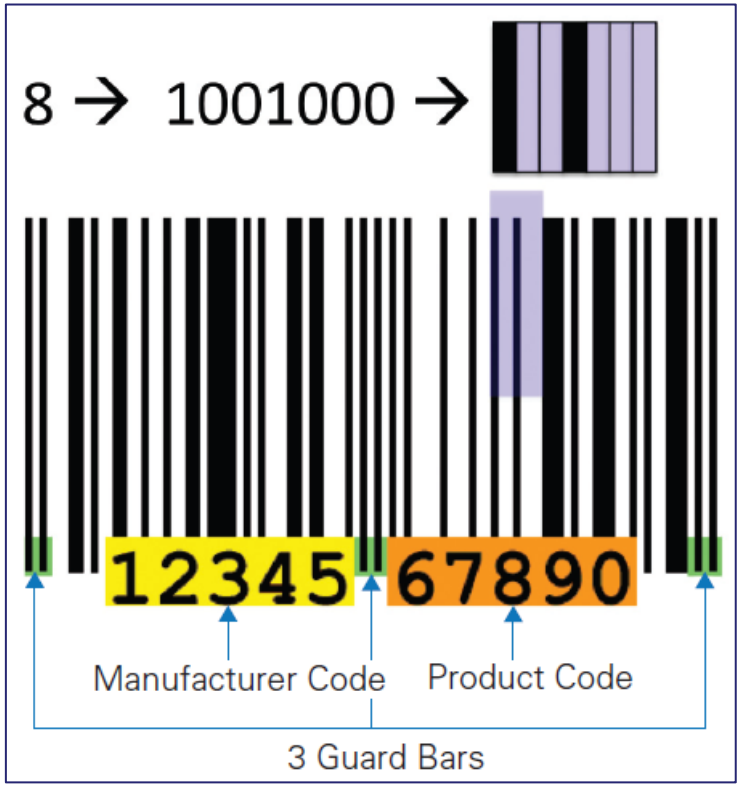
NATO Broadcast Alphabet

Table 7.7 NATO broadcast alphabet designed not to be minimal

A	Alpha	H	Hotel	O	Oscar	V	Victor
B	Bravo	I	India	P	Papa	W	Whiskey
C	Charlie	J	Juliet	Q	Quebec	X	X-ray
D	Delta	K	Kilo	R	Romeo	Y	Yankee
E	Echo	L	Lima	S	Sierra	Z	Zulu
F	Foxtrot	M	Mike	T	Tango		
G	Golf	N	November	U	Uniform		

Bar Codes

- Universal Product Codes (UPC) also use more than the minimum number of bits to encode information
- In the UPC-A encoding, 7 bits are used to encode the digits 0 – 9



Bar Codes

- UPC encodes the manufacturer (left side) and the product (right side)
- Different bit combinations are used for each side
- One side is the complement of the other side
- The bit patterns were chosen to appear as different as possible from each other

Bar Codes

- Different encodings for each side make it possible to recognize whether the code is right side up or upside down

Table 7.8 Bit encoding for bars for the UPC-A encoding; notice that the two sides are complements of each other, that is, the 0's and 1's are switched.

Digit	Left Side	Right Side
0	0001101	1110010
1	0011001	1100110
2	0010011	1101100
3	0111101	1000010
4	0100011	1011100
5	0110001	1001110
6	0101111	1010000
7	0111011	1000100
8	0110111	1001000
9	0001011	1110100

Parity

- Computer memory is subject to errors
- An extra bit is added to the memory to help detect errors
  - A ninth bit per byte can detect errors using parity
- ***Parity*** refers to whether a number is even or odd
  - If the number of 1's is even, set the ninth bit to 0; otherwise set it to 1

Parity

- All 9-bit groups have even parity:
  - Any single bit error in a group causes its parity to become odd
  - This allows hardware to detect that an error has occurred
  - It cannot detect *what* bit is wrong, however

Why “Byte”?

- IBM was building a supercomputer, called Stretch
- They needed a word for a quantity of memory ***between*** a bit and a word
  - A word of computer memory is typically the amount required to represent computer instructions (currently a word is 32 bits)

Why “Byte”?

- Then, why not bite?
- The ‘i’ to a ‘y’ was done so that someone couldn’t accidentally change ‘byte’ to ‘bit’ by the dropping the ‘e’ ”
  - bite bit (*the meaning changes)*
  - byte byt (*what’s a byt?)*

Metadata and the Oxford English Dictionary (OED)

- Converting the content into binary is half of the problem of representing information
- The other half of the problem? Describing the information's properties
- Characteristics of the content also needs to be encoded:
  - How is the content structured?
  - What other content is it related to?
  - Where was it collected?
  - What units is it given in?
  - How should it be displayed?
  - When was it created or captured?
  - And so on…

Metadata and the OED

- Metadata: information describing information
- Metadata is the third basic form of data
- It does not require its own binary encoding
- The most common way to give metadata is with tags (think back to Chapter 4 when we wrote HTML)

Properties of Data

- Metadata is separate from the information that it describes
- For example:
  - The ASCII representation of letters has been discussed
  - How do those letters look in **Comic Sans** font?
  - How they are displayed is metadata

Properties of Data

- Rather than fill a file with the **Times New Roman** font, the file is filled with the letters and tags that describe how it should be displayed
- This avoids locking-in the form of display, that can be changed by changing the metadata
- Metadata can be presented at several levels

Using Tags for Metadata

- The Oxford English Dictionary (OED) is the definitive reference for every English word's meaning, etymology, and usage
- The printed version of the OED is truly monumental
  - 20 volumes
  - 150 pounds
  - 4 feet of shelf space

Using Tags for Metadata

- In 1984, the conversion of the OED to digital form began
- Imagine, with what you know about searching on the computer, finding the definition for the verb ***set***:
  - "set" is part of many words
  - You would find closet, horsetail, settle, and more
- Software can help sort out the words

Structure Tags

- Special tags can be developed to handle structure:
  - **<*hw*>** is the OED's tag for a headword (word being defined)
  - **<*pr*>** handles pronunciation
  - **<*ph*>** does the phonetic notations
  - **<*ps*>** parts of speech
  - **<*hm*>** homonym numbers
  - **<e>** surrounds the entire entry
  - **<*hg*>** surrounds the head group or all of the information at the start of a definition

Structure Tags

- With structure tags, software can use a simple algorithm to find what is needed
- Tags do not print
- They are included only to specify the structure, so the computer knows what part of the dictionary to use
- Structure tags are also useful for formatting…for example, boldface used for headwords
- Knowing the structure makes it possible to generate the formatting information

Summary

- We began the chapter by learning that digitizing doesn't require digits—any symbols will do
- We explored the following:
  - binary encoding, that is based on the presence and absence of a physical phenomenon
  - Their patterns are discrete; they form the basic unit of a bit. Their names (most often 1 and 0) can be any pair of opposite terms

Summary

- We explored the following:
  - A bit's 0 and 1 states naturally encourage the representation of numbers in base 2, that is, binary
  - 7-bit ASCII, an early assignment of bit sequences (symbols) to keyboard characters. Extended or 8-bit ASCII is the standard
  - The need to use more than the minimum number of bits to encode information

Summary

- We explored the following:
  - How documents like the *Oxford English Dictionary are digitized*
    - *We* learned that tags associate metadata with every part of the *OED*
    - Using that data, a computer can easily help us find words and other information.
  - The mystery of the *y in byte*

Representing numbers

decimal vs binary numbers

- decimal numbers are based on *base 10 positional notation* (or *base 10 place-value notation*)
  - ten digits 0 to 9
  - $234.56 = 2x10^{2} + 3x10^{1} + 4x10^{0} + 5x10^{-1} + 6x10^{-2}$
- positional notation can be used with any base; useful bases
  - hexadecimal 16
  - octal 8
  - binary 2 – the only one used in computers
- for technological reasons number representation in computers uses always a fixed number of bits

Types of numbers

Two major types

- integers
- non-integers → floating point
  - float (32 bits)
  - double (64 bits) – the non-integers used in R
  - … other representations
  - float and double share the same principles, the differences are only quantitative, doubles allow a *greater precision*

Types of numbers

The internal representations are significantly different

- integers use less space and are processed at a higher speed
- non-integers require a larger number of bytes and more complex processing
  - e.g. the sum of two integers can be computed in a single cpu clock cycle, while the sum of two double precision requires 4 cpu clock cycles

Numbers

Integers

- 32 bits
  - from $-(2^{31}-1)$ to $2^{31}-1$
- internal binary representation
- negatives represented in *two's complement*

Double Floating point

- 64 bits
- limits can change in different language implementations
- represented as **mantissa* base$^{\text{exponent}}$**
- internal binary representation

mantissa is also known as *significant*

Two's complement for negative integers

- representation of -1
  - represent 1 — 00000000000000000000000000000001
    - 31 zeros followed by 1
  - invert all bits
    - 31 one followed by zero — 11111111111111111111111111111110
  - add one — 11111111111111111111111111111111

Two's complement for negative integers

- representation of $-(2^{31}-1)$

01111111111111111111111111111111

10000000000000000000000000000000

10000000000000000000000000000001

Two's complement: 8 bit example

3	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**1**
inversion	1	1	1	1	1	1	0	0
add 1 this is -3	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**1**
what number is?	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
this is 127								
what number is?	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
inversion	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
add 1 this is -128	**1**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

non negative

negative

Overflow: 8 bit example

127	**0**	**1**	**1**	**1**	**1**	**1**	**1**	**1**
1	0	0	0	0	0	0	0	1
127+1	1	0	0	0	0	0	0	0

when the sum exceeds the maximum integer that can be represented with the actual number of bits per integer

in real cases the bits are 32, but one is reserved to discriminate negative and non-negative

Why is the point *floating*?

because the exponent is adjusted in such a way that

- the mantissa is less than 1
- the first digit to the right of the decimal point is different from zero

Normalized floating point

- any rational number with a finite number of digits can be represented, in a given base B, with two numbers:
  - mantissa (or significand)
  - exponent
- mantissa is a rational number < 1
  - it is normalized, requiring the first digit to the right of the decimal point to be non-zero (*normalization*)
- $X = m * B^e$
- example: B=10

X	<mantissa,exponent>	representation
42.576	<0.42576,2>	$0.42576 * 10^2$
4257.6	<0.42576,4>	$0.42576 * 10^4$
0.00042576	<0.42576,-3>	$0.42576 * 10^{-3}$

double IEEE 754 standard

https://en.wikipedia.org/wiki/Double-precision_floating-point_format

bit	1	11	52
content	sign	exponent	mantissa
position	63	62 52	51 0
numbers represented	expo-nent	0:1023	
	man-tissa	any but 0	
special numbers		mantissa and exponent 0	0
		mantissa 0, exponent 1023	Infinity

Use the reference below to visualize bit contents in the *float* format, that has the same principles of the double but uses only 32 bits in memory http://www.h-schmidt.net/FloatConverter/IEEE754.html. Try 1.4E-45 and 3.4028235E38

Parameters for numbers representation in R

to inspect in R **`print(.Machine$parameter)`**

Parameter	Value	Description
double.eps	2.220446e-16	the smallest positive floating-point number x such that 1 + x != 1
double.neg.eps	1.110223e-16	a small positive floating-point number x such that 1 - x != 1
double.xmin	2.225074e-308	the smallest non-zero normalized floating-point number
double.xmax	1.797693e+308	the largest normalized floating-point number
double.base	2	yes, … they are binary!
double.digits	53	the number of base digits in the floating-point significand: normally 53
double.rounding	5	the rounding action, one of … 5 floating-point addition rounds in the IEEE style, and there is partial underflow.
double.guard	0	truncation in floating point arithmetics
double.ulp.digits	-52	the largest negative integer i such that 1 + double.base ^ i != 1, except that it is bounded below by -(double.digits + 3)
double.neg.ulp.digits	-53	the largest negative integer i such that 1 - double.base ^ i != 1, except that it is bounded below by -(double.digits + 3)
double.exponent	11	the number of bits reserved for the representation of the exponent (including the bias or sign) of a floating-point number
double.min.exp	-1022	the largest in magnitude negative integer i such that double.base ^ i is positive and normalized
double.max.exp	1024	the smallest positive power of double.base that overflows
integer.max	2147483647	the largest integer that can be represented, always 2^31 - 1
sizeof.long	8	byte sizes for C language
sizeof.longlong	8	byte sizes for C language
sizeof.longdouble	16	byte sizes for C language
sizeof.pointer	8	byte sizes for C language

double in R

- the biggest number is 2^{1023}
  - about $9*10^{307}$
- the smallest non-zero is 2^{-1074}
  - about $5 * 10^{-324}$
  - the minimum exponent in base 10 is -308, but the smaller number is reached by renouncing to the normalization of the mantissa and reducing the precision
- the same limits hold, in absolute value, for negatives

advantages of the mantissa/exponent representation

- the number of significant digits does is not influenced by the order of magnitude
- it is possible to represent very big and very small (non-zero) numbers
  - think to the limitations of a representation with a fixed number of digits for the integer and the decimal parts

Observations

- between the two ends only a finite amount of *double* numbers is allowed
- *doubles are a finite subset of the rationals that can be represented with a finite number of digits for mantissa and exponents*
- the concepts of infinite and infinitesimal are substituted by two specific numbers

double: precision

- let us consider two consecutive double numbers

$d_1 = m_1 * 2^e$ e $d_2 = m_2 * 2^e$

- m_1 and m_2 differ only in the last bit
- all the infinite reals between d_1 and d_2 are represented with either d_1 or d_2
  - rounding → the nearest
  - truncation → cut the exceeding, less significant, digits
- the maximum error, in case of truncation, is

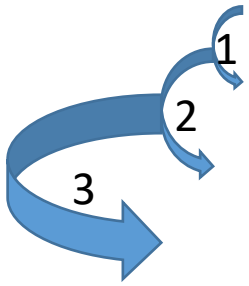
  $(m_1 - m_2) * 2^e$

double: precision (ii)

- the precision changes, depending on the exponent
  - the error generated by the representation of a real number with the truncated double increases with the magnitude of the number
- nevertheless, the relative error is independent from the exponent
  - $(d_1-d_2)/d_1 = (m_1-m_2)/m_1$

alignment and truncation (i)

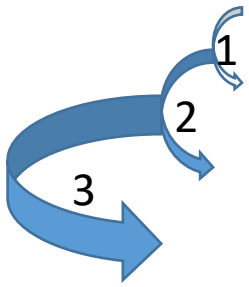
example of sum on base 10 and two significant digits
the sum is executed with a *left to right association*



normalized addends	aligned addends	intermediate results (without truncation)	intermediate results (with truncation)
$0.72*10^{1}$	7.2		
$0.63*10^{-1}$	0.063	7.263	7.2
$0.42*10^{-1}$	0.042	7.305	7.2
$0.63*10^{-2}$	0.0063	7.3113	7.2
error = 0.1113			

alignment and truncation (ii)

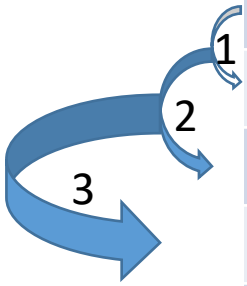
example of sum on base 10 and four significant digits



normalized addends	aligned addends	intermediate results (without truncation)	intermediate results (with truncation)
$0.72*10^{1}$	7.2		
$0.63*10^{-1}$	0.063	7.263	7.263
$0.42*10^{-1}$	0.042	7.305	7.305
$0.63*10^{-2}$	0.0063	7.3113	7.311
error = 0.0003			

alignment and truncation (iii)

example of sum on base 10 and two significant digits with addends of increasing size



normalized addends	aligned addends	intermediate results (without truncation)	intermediate results (with truncation)
$0.63*10^{-2}$	0.0063		
$0.42*10^{-1}$	0.042	0.0483	0.048
$0.63*10^{-1}$	0.063	0.1113	0.11
$0.72*10^{1}$	7.2	7.3113	7.3
error = 0.0113			

The arithmetics of doubles

- the associative property can be violated
  - (x + y) + z *can be different from* x + (y + z)
  - (x * y) * z *can be different from* x * (y * z)
- the distributive property can be violated
  - x * (y + z) *can be different from* x * y + x * z
- changing the order of execution of some operations *can change* the result when using the arithmetics of doubles, while with the arithmetics or *real numbers* it would not change
- the commutative property has the same validity for *double/float* and *real*

Finite Precision in Statistical Computation

- Computers approximate real numbers
- Statistical algorithms operate on approximations
- Why this matters for data analysis

Mathematics vs Computation

- In mathematics:
  - $\mathbb{R}$ *(infinite, exact)*
- In a computer:
  - $\mathbb{F} \subset \mathbb{R}$ *(finite set of representable numbers)*
- Key facts:
  - Only finitely many numbers can be stored.
  - Every stored real number is an approximation.

Floating-Point Model (Conceptual)

- When a real number $x$is stored:
  - $\mathrm{fl}(x) = x(1+\delta)$ *with* $|\delta| \leq \varepsilon$
- Where:
  - ε= machine precision
  - δ= relative rounding error
- Important:
  - Errors are small
  - Errors are unavoidable
  - Errors propagate

Three Types of Numerical Effects

- **Representation error**
  Some numbers (e.g. 0.1) are not exactly representable.
- **Rounding error**
  Each arithmetic operation introduces small error.
- **Error propagation**
  Algorithms may amplify or dampen errors.

Example 1: Loss of Significance

- Consider:
  - $10^8 + 1 - 10^8$
- Mathematically:
  - $= 1$
- Numerically (finite precision):
  - $10^8 + 1$may be stored as 10^8
- Final result becomes 0
- This is called:
  - **Catastrophic cancellation**

Why This Matters for Statistics

Statistical computations involve:

- Large sums
- Differences of similar quantities
- Matrix operations
- Iterative algorithms

These are sensitive to rounding.

Example 2: Variance Formula

- Naive formula:
  - $\mathrm{Var}(X) = E[X^2] - \Big(E[X]\Big)^2$
- Problem:
  - Two large, similar numbers are subtracted.
  - Significant digits may be lost.
- Better approach:
  - Use numerically stable algorithms
  - Center data before squaring

Accumulation of Error in Sums

Sample mean:

- $\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$

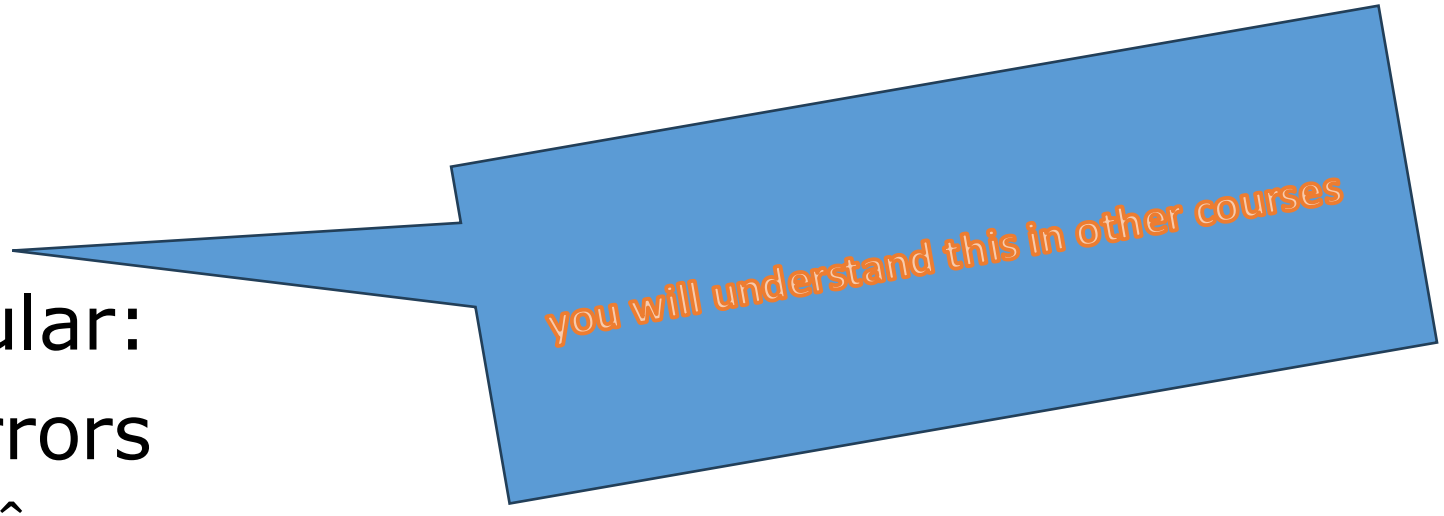
If $n$is large:

- Each addition introduces rounding error
- Total error grows approximately as:
  - $O(n\varepsilon)$
- Order of summation can change the result.

Matrices and Regression

Linear regression:

- $\hat{\beta} = (X^T X)^{-1} X^T y$



If $X^T X$is nearly singular:

- Small numerical errors
- Large variation in $\hat{\beta}$

Key concept:

- Conditioning of a problem

A well-posed statistical model can still be numerically unstable.

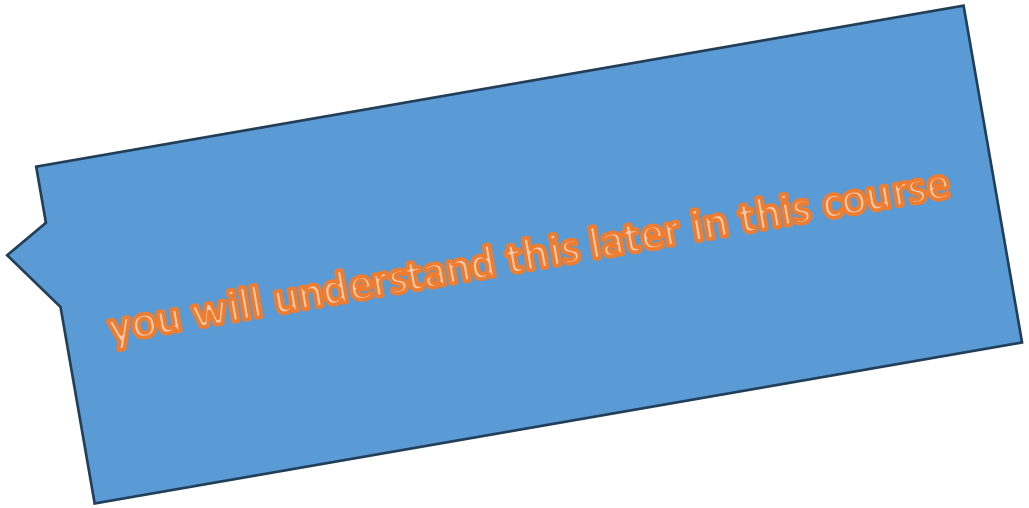
Iterative Algorithms

Many statistical procedures are iterative:

- Newton–Raphson
- EM algorithm
- Gradient descent
- MCMC

Effects of finite precision:

- Accumulated rounding
- Stopping criteria sensitivity
- Apparent non-convergence



Two Types of Error in Statistics

Type	Origin
Statistical error	Sampling variability
Numerical error	Finite precision in arithmetic

Statistical theory assumes exact arithmetic
Computation does not

When Does It Matter?

Finite precision is usually negligible in:

- Small datasets
- Well-conditioned problems

It becomes important in:

- Very large datasets
- Nearly collinear regressors
- Ill-conditioned matrices
- Long simulations
- High-dimensional models

Key takeaways

Computers do not compute with real numbers.

- Every operation introduces rounding.
- Some algorithms amplify errors.
- Statistical reasoning must account for numerical stability.

Statistical inference is not only a probabilistic problem, it is also a numerical problem.