Scientific computing

Kylie A. Bemis

Northeastern University
Khoury College of Computer Sciences

Northeastern University

Goals for today

- Representing numbers
- Matrices and arrays
- Introduction to NumPy

REPRESENTING NUMBERS

Storing numbers

- We represent numbers using a **base 10** system
  - 1, 10, 100, 1000, etc.
- Computers only store **bits** (0s and 1s)
- How to store *integers* and *real numbers* using only patterns of *bits*?

Patterns of bits

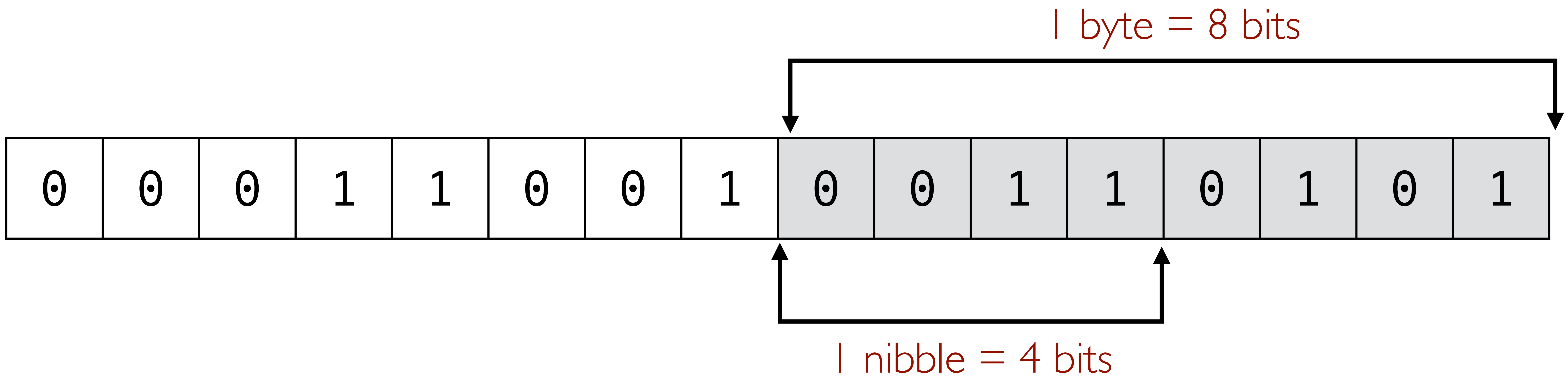
- All computer data is a sequence of bits
- How many unique values can be stored?

# of bits	Possible sequences	# of sequences
1	`0, 1`	2
2	`00, 01, 10, 11`	4
3	`000, 001, 010, 011` `100, 101, 110, 111`	8

N bits can store 2^N possible values!

Bits and bytes

- Computers rarely work on individual bits
- Instead, operate on *chunks of bits*
- Typically, operate on **bytes** of 8 bits



Bytes and storage

- Bytes are the building blocks of data types
- How many values can be stored per byte?

# of bytes	# of bits	# of values
1	8	256
2	16	65,536
4	32	4,294,967,296
8	64	1.844674×10^{19}
16	128	3.402823×10^{38}

Storing integers

- **Bits** can only have two values (1s and 0s)
- Represent integers using **base 2** system

Bit	7	6	5	4	3	2	1	0	
Value	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
	0	0	0	0	0	0	0	0	= 0
	0	0	0	0	0	0	0	1	= 1
	0	0	0	0	0	0	1	0	= 2
	0	0	0	0	0	0	1	1	= 3
	0	0	0	0	0	1	0	0	= 4

Storing integers (2)

- **Bits** can only have two values (1s and 0s)
- Represent integers using **base 2** system

Bit	7	6	5	4	3	2	1	0	
Value	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
	`0`	`0`	`1`	`1`	`0`	`1`	`0`	`1`	= 53

`0x00110101` $= 0\times2^7 + 0\times2^6 + 1\times2^5 + 1\times2^4 + 0\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$

$= 0 + 0 + 32 + 16 + 0 + 4 + 0 + 1$

$= 53$

Storing integers (3)

- **Bits** can only have two values (1s and 0s)
- Represent integers using **base 2** system

Bit	7	6	5	4	3	2	1	0	
Value	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
	`0`	`1`	`1`	`0`	`1`	`1`	`0`	`0`	=108

`0x01101100` $= 0\times2^7 + 1\times2^6 + 1\times2^5 + 0\times2^4 + 1\times2^3 + 1\times2^2 + 0\times2^1 + 0\times2^0$

$= 0 + 64 + 32 + 0 + 8 + 4 + 0 + 0$

$= 108$

Bytes and hex

- *<u>Need:</u>* Print human-readable byte data
- *<u>But:</u>* Bits are difficult for humans to read
- Solution: Use **hexadecimal**
  - Base 16 system using characters 1-9 and A-F
  - Represent <u>one byte</u> with <u>two hex digits</u>

Hexadecimal

- Base 16 numeric system
  - Represent numbers 0-9 with "0"-"9"
  - Represent numbers 10-15 with "A"-"F"
- Represent one byte with two hex digits
  - 0000 0000 becomes 0x00
  - 1111 1111 becomes 0xFF
  - 0000 1111 is 0x0F

Decimal to hex

- Split byte into two half-bytes (nibbles)
- Find hex representation for each half

Bit	7	6	5	4	3	2	1	0	
Value	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
	0	1	1	0	1	1	0	0	= 6C

2^3	2^2	2^1	2^0	
0	1	1	0	= 6

2^3	2^2	2^1	2^0	
1	1	0	0	= C

Representations

- Decimal
  - Typical base 10 system
  - E.g., **108**
- Hexadecimal
  - Compact representation of bytes using base 16
  - E.g., **6C**
- Binary
  - How data is actually stored in computers
  - E.g., **01101100**

What about negatives?

- Straightforward to encode *unsigned* numbers
- How to encode ***signed*** numbers?
  - Sign and magnitude
  - One's complement
  - Two's complement
- Need to choose a representation

Sign and magnitude

- One bit stores sign (+ or -)
- Other bits store magnitude

Bit	7	6	5	4	3	2	1	0	
Value	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
	0	0	1	1	0	1	0	1	= 53
	1	0	1	1	0	1	0	1	=-53

Sign bit

One's complement

- Apply bitwise NOT operator
- Negative is "complement" of positive

Bit	7	6	5	4	3	2	1	0	
Value	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
	0	0	1	1	0	1	0	1	= 53
Invert each bit	1	1	0	0	1	0	1	0	=-53

Two's complement

- Apply bitwise NOT operator and add 1
- One's complement plus one

Bit	7	6	5	4	3	2	1	0	
Value	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
	0	0	1	1	0	1	0	1	= 53
Invert each bit	1	1	0	0	1	0	1	1	=-53

Add 1

Signed representations

- Sign and magnitude
  - *Different* arithmetic for positive and negative
  - +0 and -0 are both represented
- One's complement
  - *Similar* arithmetic for positive and negative
  - +0 and -0 are both represented
- Two's complement
  - *Same* arithmetic for positive and negative
  - Single zero

Signed representations

- Sign and magnitude
  - *Different* arithmetic for positive and negative
  - +0 and -0 are both represented
- One's complement
  - *Similar* arithmetic for positive and negative
  - +0 and -0 are both represented
- Two's complement
  - *Same* arithmetic for positive and negative
  - Single zero

Typical implementation

Integer overflow

- Can only store so many values
- Larger magnitudes "overflow"

8-bit integer

Decimal	Binary
127	`0111 1111`
126	`0111 1110`
...	...
2	`0000 0010`
1	`0000 0001`
0	`0000 0000`

127+1

Decimal	Binary
-1	`1111 1111`
-2	`1111 1110`
...	...
-126	`1000 0010`
-127	`1000 0001`
-128	`1000 0000`

= -128

Common integer representations

- 32-bit integer
  - Unsigned max: $2^{32}-1 = 4{,}294{,}967{,}296$
  - Signed range: -2,147,483,648 to +2,147,483,647
- 64-bit integer
  - Unsigned max: $2^{64}-1$
  - Signed range: -2^{63} to $+2^{63}-1$

What about real numbers?

- Difficult to represent versus integers
- Infinite range of possible values
  - Not all values can be stored precisely
  - Need to trade off between range and precision
- What about "special" values?
  - E.g., $+\infty$, $-\infty$, and NaN

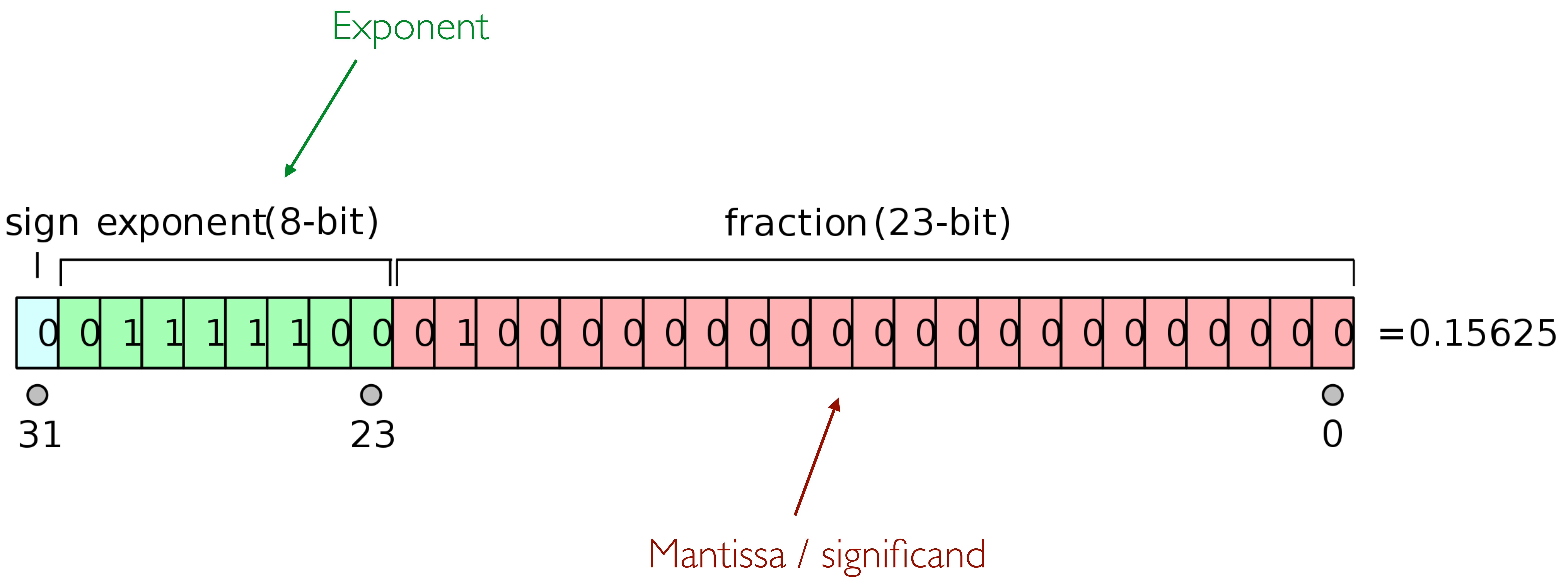
Motivation: Scientific notation

- Similar to floating point representation
- Represent numbers as $\mathbf{m \times 10^n}$, where
  - **n** is an integer called the *exponent*
  - **m** is a real number (typically between 1 and 10)
  - **m** is called the *significand* or *mantissa*
- E.g., $1{,}234 = 1.234 \times 10^3$

IEEE 754: Floating point

- Standard for representing real numbers
- Storage similar to scientific notation
  1. Sign bit
  2. Exponent
  3. Mantissa/significand
- Special sequences for $+\infty$, $-\infty$, and NaN

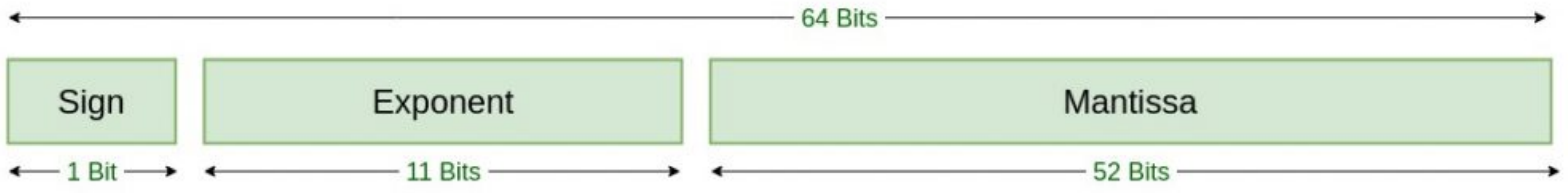
"Single" precision: 32-bit float



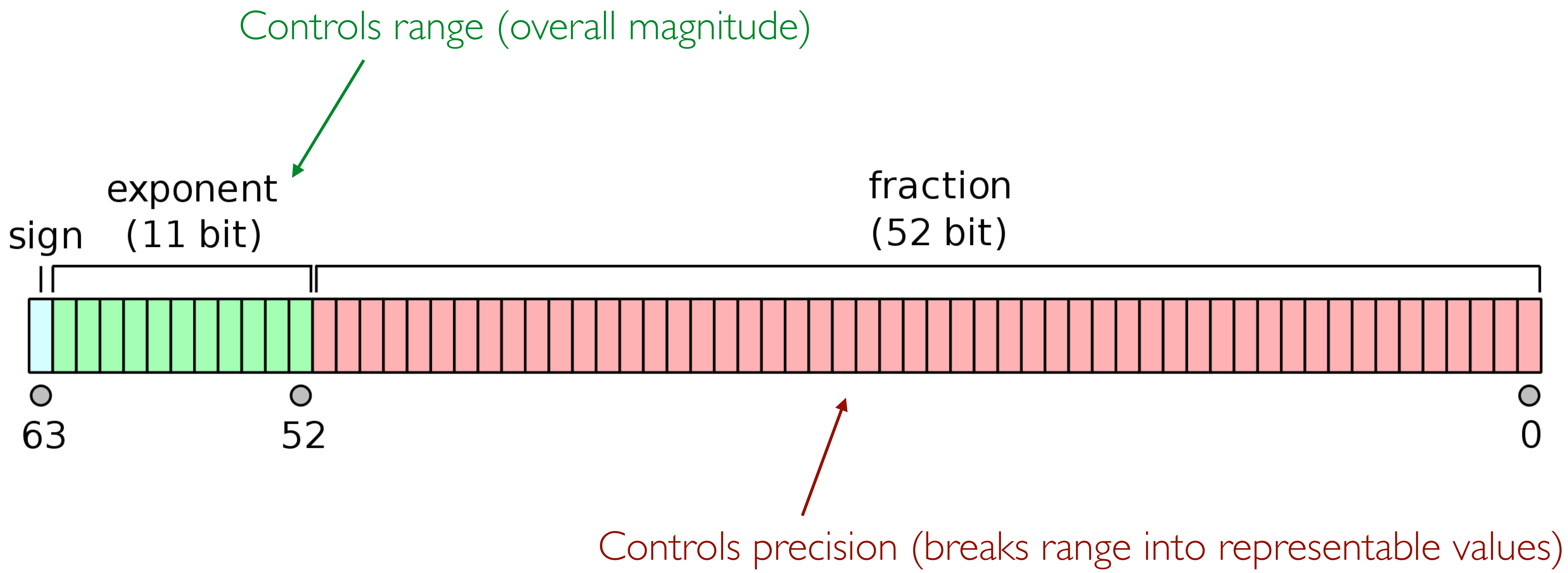
https://en.wikipedia.org/wiki/IEEE_754-1985#/media/File:IEEE_754_Single_Floating_Point_Format.svg

Floating point representation

- Exponent controls range
  - Scales overall magnitude of the number
  - Allows for very large and very small numbers
- Mantissa controls precision
  - Breaks overall range into finite number of points
  - These are the points that can be represented



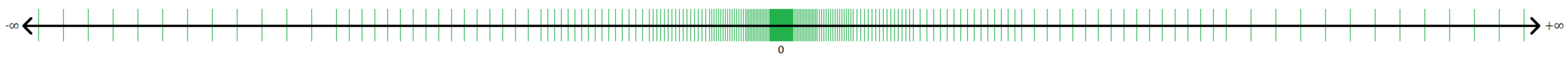
"Double" precision: 64-bit float



https://en.wikipedia.org/wiki/IEEE_754-1985#/media/File:IEEE_754_Double_Floating_Point_Format.svg

Trade-off between range and precision

- Floating point can store <u>either</u>:
  - Very large numbers, OR
  - Very small numbers
- Cannot precisely store both at once
  - Larger numbers are separated by wider "gaps"
  - For sufficiently large **x**, floating point says **x + 1 = x**
- *Caution: consider scale of your computations*



Precision in floating point arithmetic

- Many values cannot be precisely represented
- Small floating point errors can compound or underflow/overflow over many computations
- Use caution with floating point arithmetic:
  - Never check for perfect equality — always some error
  - Transform unstable operations (e.g., product vs. sum of logs)
  - Use algorithms with greater **numeric stability**

"Special" values

- Positive and negative infinity
  - Exponent is all 1s
  - Mantissa is all 0s
- Not-a-number (NaNs)
  - Exponent is all 1s
  - Any part of mantissa is non-0s
- "Subnormal" numbers
  - Exponent is all 0s
  - Ensure small differences $x - y \neq 0$ when $x \neq y$

Common float representations

- 32-bit float ("single"-precision)
  - 8-bit exponent and 23-bit mantissa
  - ~7 digits of decimal precision
- 64-bit float ("double"-precision)
  - 11-bit exponent and 52-bit mantissa
  - ~16 digits of decimal precision

MATRICES AND ARRAYS

Numeric computing

- Need to represent matrices and *N-D* arrays
  - Linear algebra
  - Machine learning
- Representation must be efficient
  - O(1) access of numeric elements
  - Compact storage
  - Fast traversal
  - Data size is typically fixed

Considerations for matrices

- ## Patterns of access
  - *Locality in memory* improves performance
  - Prefer to store **rows** or **columns** as major dimension?
- ## Patterns of data
  - Does the matrix have a **structure**? (e.g., *diagonal*)
  - How **dense/sparse** is the matrix?
    - Sparse matrices (mostly 0 elements) common in some applications
    - Storing only non-zero elements could save a lot of space

Storing dense matrices

- Store data as an **array** + **dimensions**
- Column-major vs. row-major order

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

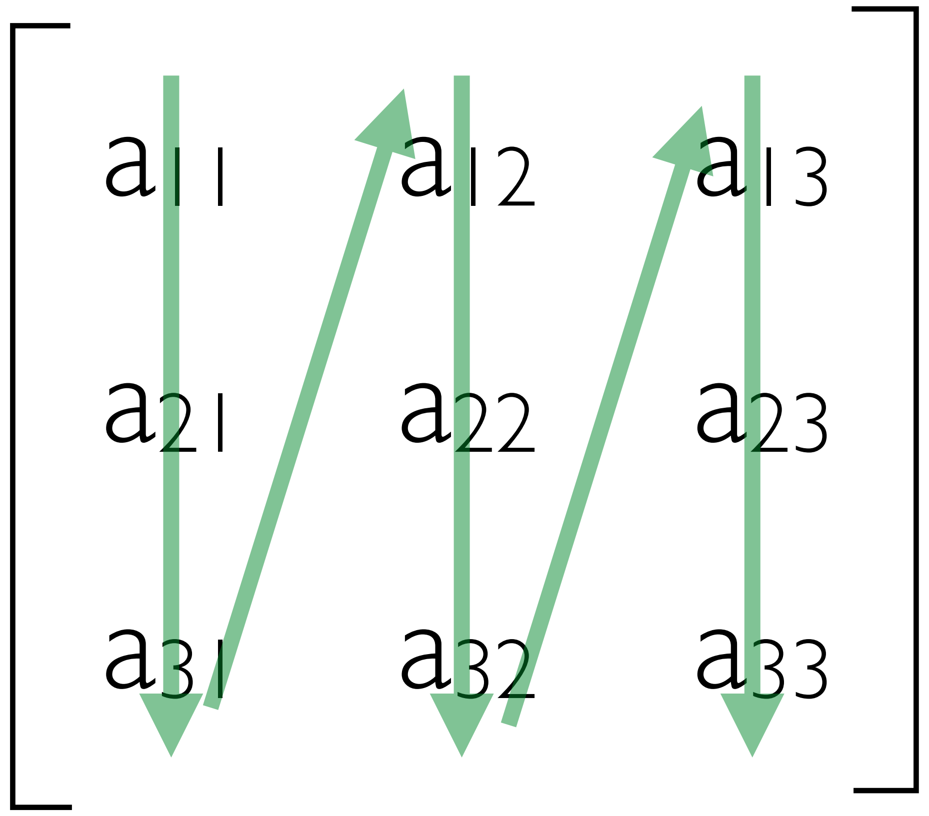
Column-major order

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

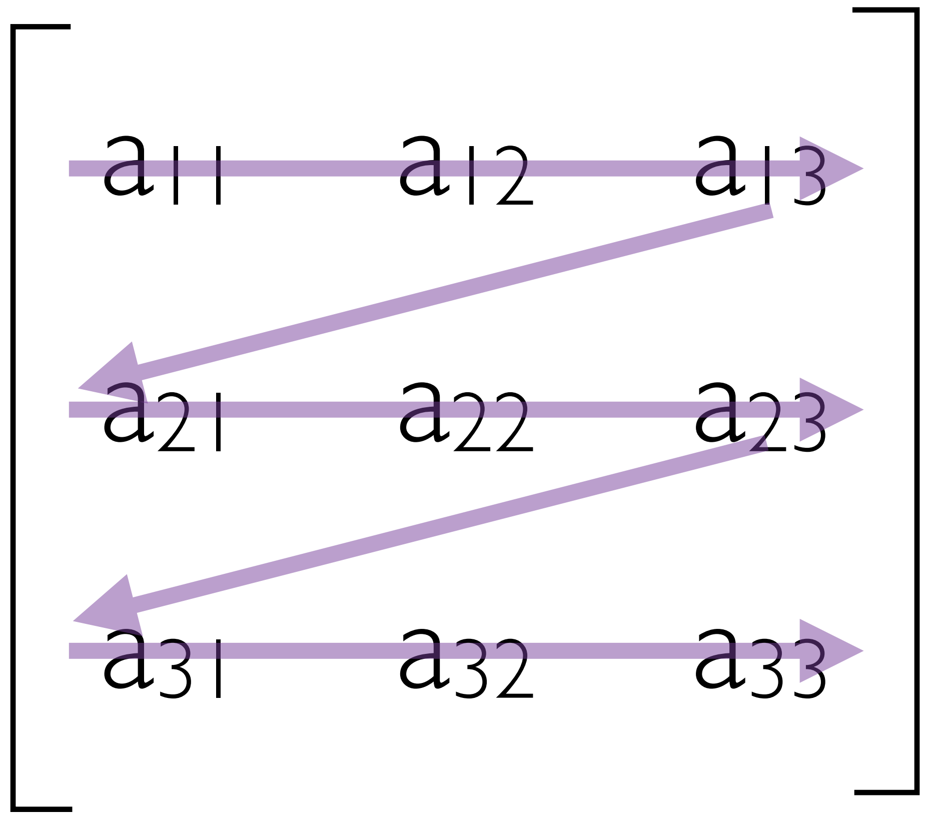
Row-major order

Storing dense matrices (2)

- Why arrays?
  - Most efficient for numeric computing
  - Arrays provide O(1) access
  - Arrays provide *locality in memory*



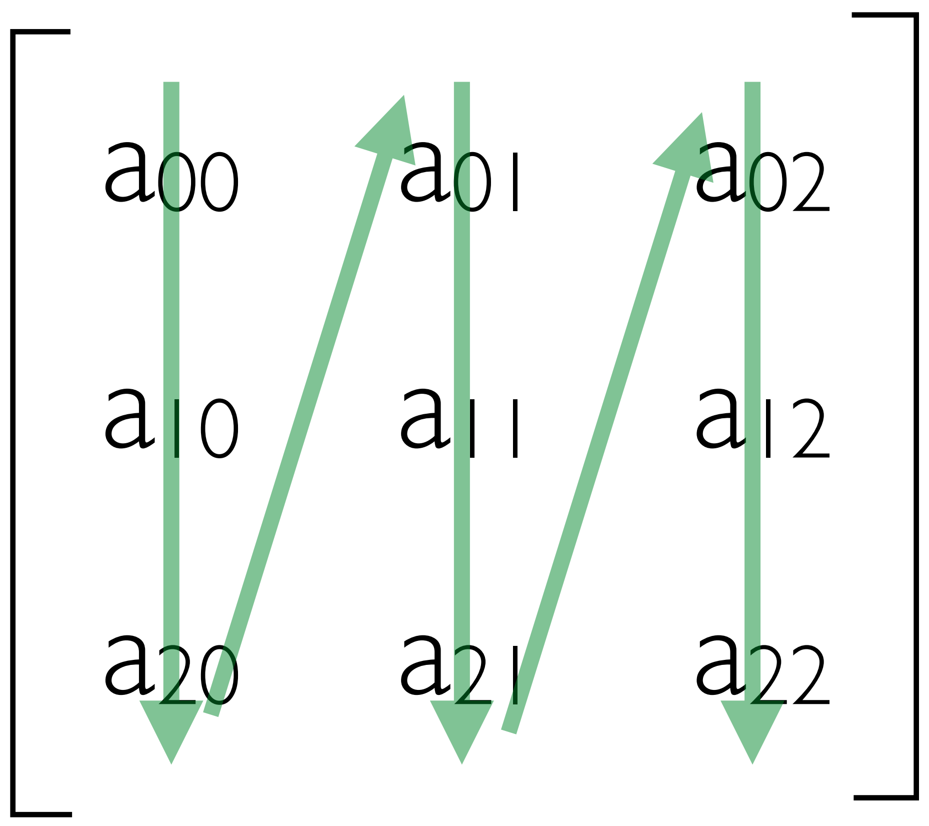
Column-major order



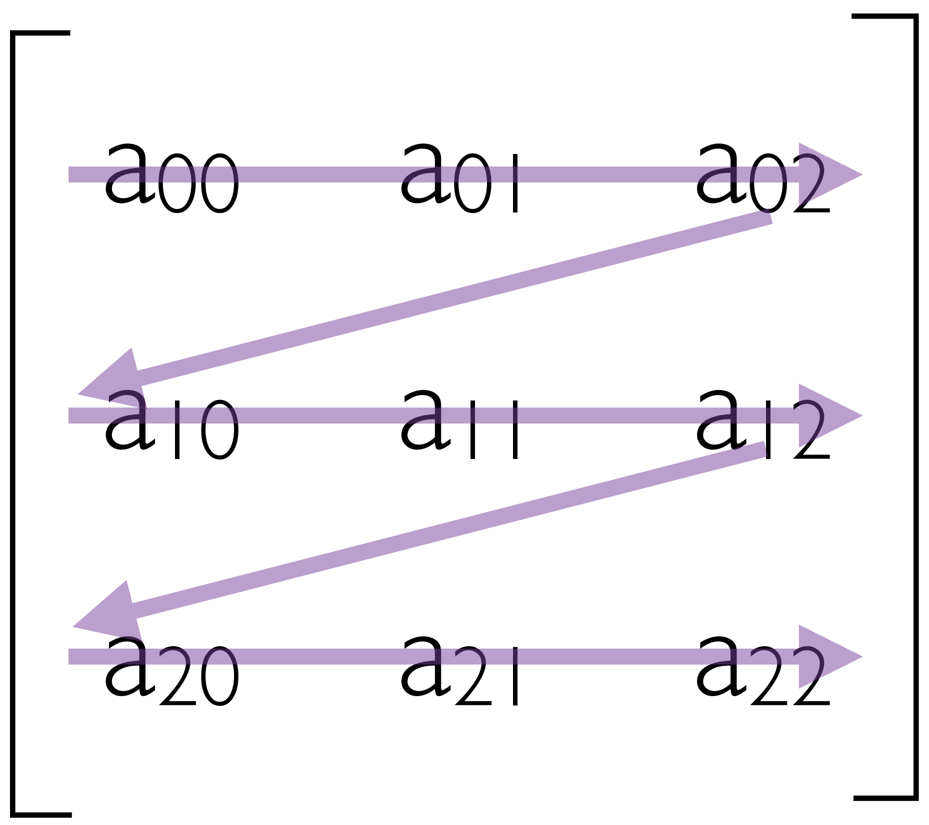
Row-major order

Accessing dense matrix elements

Column-major order



Row-major order

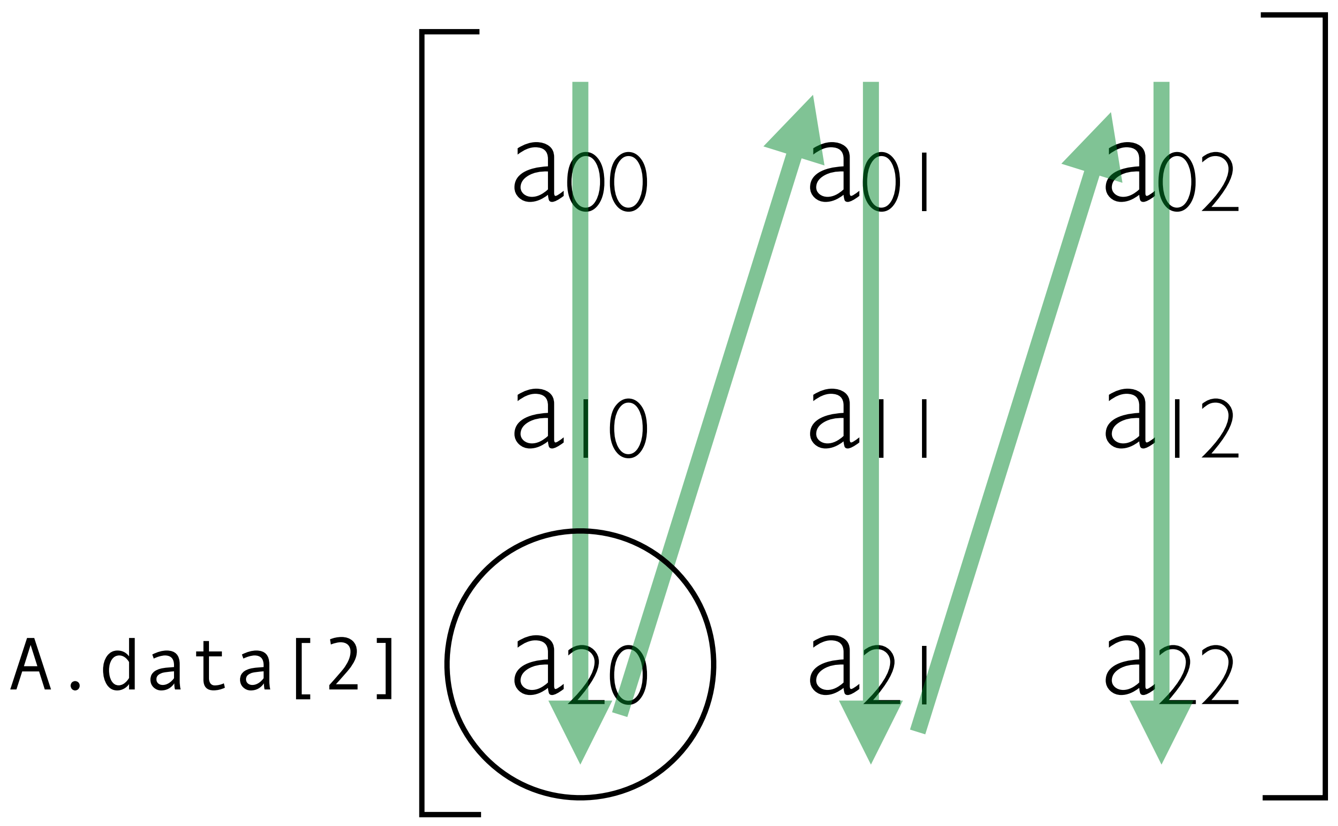



```
A[i,j] = A.data[j * nrow + i]
```

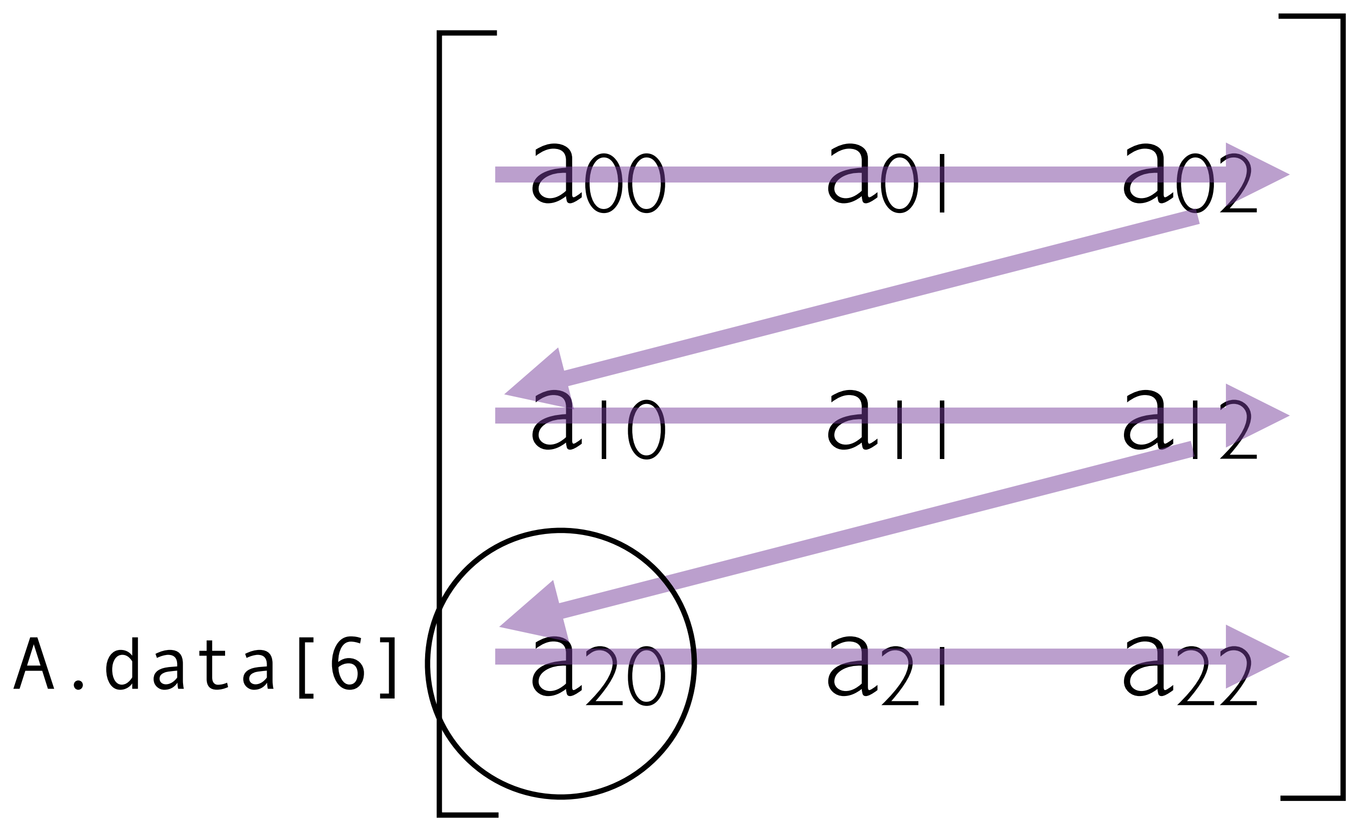
```
A[i,j] = A.data[i * ncol + j]
```

Accessing dense matrix elements (2)

Column-major order



Row-major order




```
A[i,j] = A.data[j * nrow + i]

A[2,0] = A.data[0 * 3 + 2]
       = A.data[2]
```

```
A[i,j] = A.data[i * ncol + j]

A[2,0] = A.data[2 * 3 + 0]
       = A.data[6]
```

Considerations for sparse matrices

- Most data elements are 0
  - E.g., a *document-term matrix* for text modeling
  - Storing all elements is **not** space efficient
  - More compact to store *only nonzero elements*
- Sparse compression
  - How easy to construct/modify?
  - How easy to access/compute on?

Sparse matrix representations

- Ease of construction
    - Dictionary of keys (DOK)
    - List of lists (LIL)
    - Coordinate list (COO)
- Ease of computation
    - Compressed sparse row (CSR)
    - Compressed sparse column (CSC)

Sparse matrix representations (2)

- Dictionary of keys
  - Keys are tuples of coordinates
  - (row, column) → value
- List of lists
  - Store a list of nonzero elements in each row/column
  - [(row, value), (row, value), etc.]
- Coordinate list
  - Store array of coordinate for each element
  - (row, column, value)

Sparse matrix representations (3)

- Compressed sparse row (CSR)
  - Compress rows
  - Fast access to whole **rows**
  - Difficult to construct **columns**
- Compressed sparse column (CSC)
  - Compress columns
  - Difficult to construct **rows**
  - Fast access to whole **columns**

Compressed sparse row (CSR)

- Consider ***m x n*** matrix
- Store only ***nnz*** non-zero elements
  - *Data array* of non-zero elements (length *nnz*)
  - *Index array* of column indices (length *nnz*)
  - *Pointer array* of row slices in *index array* (length *m+1*)

$$\begin{bmatrix} 0 & 11 & 0 & 22 \\ 33 & 0 & 0 & 44 \\ 0 & 55 & 0 & 0 \end{bmatrix}$$

```
data = [11, 22, 33, 44, 55] # data values
ind  = [ 1,  3,  0,  3,  1] # col indices
ptr  = [ 0,  2,  4,  5]     # row slices
```

nnz

Working with CSR

- Consider ***m x n*** matrix
- Store only ***nnz*** non-zero elements
  - *Data array* of non-zero elements (length *nnz*)
  - *Index array* of column indices (length *nnz*)
  - *Pointer array* of row slices in *index array* (length *m+1*)

$$\begin{array}{ll} \text{Row 0} & \\ & \end{array}\begin{bmatrix} 0 & 11 & 0 & 22 \\ 33 & 0 & 0 & 44 \\ 0 & 55 & 0 & 0 \end{bmatrix}$$

```
data = [11, 22, 33, 44, 55]
ind  = [ 1,  3,  0,  3,  1]
ptr  = [ 0,  2,  4,  5]

row0_data = data[ptr[0]:ptr[0+1]]
          = [11, 22] # values
row0_ind  =  ind[ptr[0]:ptr[0+1]]
          = [ 1,  3] # col indices
```

Working with CSR (2)

- Consider ***m x n*** matrix
- Store only ***nnz*** non-zero elements
  - <u>*Data*</u> *array* of non-zero elements (length *nnz*)
  - <u>*Index*</u> *array* of column indices (length *nnz*)
  - <u>*Pointer*</u> *array* of row slices in *index array* (length *m+1*)

$$
\begin{array}{l} \\ \text{Row 1} \\ \\ \end{array}
\begin{bmatrix} 0 & 11 & 0 & 22 \\ 33 & 0 & 0 & 44 \\ 0 & 55 & 0 & 0 \end{bmatrix}
$$

```
data = [11, 22, 33, 44, 55]
ind  = [ 1,  3,  0,  3,  1]
ptr  = [ 0,  2,  4,  5]

row1_data = data[ptr[1]:ptr[1+1]]
          = [33, 44] # values
row1_ind  =  ind[ptr[1]:ptr[1+1]]
          = [ 0,  3] # col indices
```

Working with CSR (3)

- Consider ***m x n*** matrix
- Store only ***nnz*** non-zero elements
  - <u>*Data*</u> *array* of non-zero elements (length *nnz*)
  - <u>*Index*</u> *array* of column indices (length *nnz*)
  - <u>*Pointer*</u> *array* of row slices in *index array* (length *m+1*)

$$
\begin{array}{r}
\\
\\
\text{Row 2}
\end{array}
\begin{bmatrix}
0 & 11 & 0 & 22 \\
33 & 0 & 0 & 44 \\
\boxed{0 \quad 55 \quad 0 \quad 0}
\end{bmatrix}
$$

```
data = [11, 22, 33, 44, 55]
ind  = [ 1,  3,  0,  3,  1]
ptr  = [ 0,  2,  4,  5]

row2_data = data[ptr[2]:ptr[2+1]]
          = [55] # values
row2_ind  =  ind[ptr[2]:ptr[2+1]]
          = [ 1] # col indices
```

Compressed sparse column (CSC)

- Consider ***m x n*** matrix
- Store only ***nnz*** non-zero elements
  - <u>Data</u> *array* of non-zero elements (length *nnz*)
  - <u>Index</u> *array* of row indices (length *nnz*)
  - <u>Pointer</u> *array* of column slices in *index array* (length n+1)

$$\begin{bmatrix} 0 & 11 & 0 & 22 \\ 33 & 0 & 0 & 44 \\ 0 & 55 & 0 & 0 \end{bmatrix}$$

```
data = [33, 11, 55, 22, 44] # data values
ind  = [ 1,  0,  2,  0,  1] # row indices
ptr  = [ 0,  1,  3,  3,  5] # col slices
```

nnz

Working with CSC (2)

- Consider ***m x n*** matrix
- Store only ***nnz*** non-zero elements
  - <u>*Data*</u> *array* of non-zero elements (length *nnz*)
  - <u>*Index*</u> *array* of row indices (length *nnz*)
  - <u>*Pointer*</u> *array* of column slices in *index array* (length n+1)

$$\begin{bmatrix} 0 & 11 & 0 & 22 \\ 33 & 0 & 0 & 44 \\ 0 & 55 & 0 & 0 \end{bmatrix}$$

Column 0

```
data = [33, 11, 55, 22, 44]
ind  = [ 1,  0,  2,  0,  1]
ptr  = [ 0,  1,  3,  3,  5]

col0_data = data[ptr[0]:ptr[0+1]]
          = [33] # values
col0_ind  =  ind[ptr[0]:ptr[0+1]]
          = [ 1] # row indices
```

Working with CSC (3)

- Consider ***m x n*** matrix
- Store only ***nnz*** non-zero elements
  - *Data array* of non-zero elements (length *nnz*)
  - *Index array* of row indices (length *nnz*)
  - *Pointer array* of column slices in *index array* (length n+*1*)

$$\begin{array}{cccc} & \text{Column 1} & & \\ \end{array}$$

$$\begin{bmatrix} 0 & 11 & 0 & 22 \\ 33 & 0 & 0 & 44 \\ 0 & 55 & 0 & 0 \end{bmatrix}$$

Column 1

```
data = [33, 11, 55, 22, 44]
ind  = [ 1,  0,  2,  0,  1]
ptr  = [ 0,  1,  3,  3,  5]

col1_data = data[ptr[0]:ptr[0+1]]
          = [11, 22] # values
col1_ind  =  ind[ptr[0]:ptr[0+1]]
          = [ 0,  2] # row indices
```

Working with CSC (4)

- Consider ***m x n*** matrix
- Store only ***nnz*** non-zero elements
  - <u>Data</u> *array* of non-zero elements (length *nnz*)
  - <u>Index</u> *array* of row indices (length *nnz*)
  - <u>Pointer</u> *array* of column slices in *index array* (length n+*1*)

$$\begin{bmatrix} 0 & 11 & 0 & 22 \\ 33 & 0 & 0 & 44 \\ 0 & 55 & 0 & 0 \end{bmatrix}$$

Column 2

```
data = [33, 11, 55, 22, 44]
ind  = [ 1,  0,  2,  0,  1]
ptr  = [ 0,  1,  3,  3,  5]

col2_data = data[ptr[0]:ptr[0+1]]
          = [] # values
col2_ind  =  ind[ptr[0]:ptr[0+1]]
          = [] # row indices
```

Working with CSC (5)

- Consider ***m x n*** matrix
- Store only ***nnz*** non-zero elements
  - <u>Data</u> *array* of non-zero elements (length *nnz*)
  - <u>Index</u> *array* of row indices (length *nnz*)
  - <u>Pointer</u> *array* of column slices in *index array* (length n+*1*)

$$
\begin{bmatrix} 0 & 11 & 0 & 22 \\ 33 & 0 & 0 & 44 \\ 0 & 55 & 0 & 0 \end{bmatrix}
$$

Column 3

```
data = [33, 11, 55, 22, 44]
ind  = [ 1,  0,  2,  0,  1]
ptr  = [ 0,  1,  3,  3,  5]

col3_data = data[ptr[0]:ptr[0+1]]
          = [22, 44] # values
col3_ind  =  ind[ptr[0]:ptr[0+1]]
          = [ 0,  1] # row indices
```

N-dimensional arrays

- Principles generalize to dense N-D arrays
  - Store data as a single array (as in data structure)
  - Store array shape (size in each dimension)
  - Calculate location of elements on-the-fly (cheap)
- Sparse N-D arrays more challenging
  - Simple to store in COO format, but poor performance
  - Which dimension to compress?

INTRO TO NUMPY

Numerical Python

- NumPy provides efficient matrices and arrays
  - ***ndarray*** implements N-dimensional arrays
  - Attribute **shape** gives the dimensions
  - Attribute **dtype** gives data type (*int32*, *float64*, etc.)
- Sparse matrices also supported
  - **csr_matrix** for compressed sparse row matrices
  - **csc_matrix** for compressed sparse column matrices

Matrices in NumPy

- Default to *row-major* order
  - Can be changed with **order** attribute
  - Row-major is "C" or C-style
  - Column-major is "F" or Fortran-style
- Specify **shape** and **dtype**
  - Matrices/arrays can be *reshaped* on demand
  - All elements are same data type

Integer array

```
In : import numpy as np

In : A = np.array([[1, 2], [3, 4]], dtype=np.int32)

In : A
Out:
array([[1, 2],
       [3, 4]], dtype=int32)
```

Float array

```
In : import numpy as np

In : A = np.array([[1, 2], [3, 4]], dtype=np.float32)

In : A
Out:
array([[1., 2.],
       [3., 4.]], dtype=float32)
```

Scalar addition

```
In : import numpy as np

In : A = np.array([[1, 2], [3, 4]], dtype=np.float64)

In : 100 + A
Out:
array([[101., 102.],
       [103., 104.]], dtype=float32)
```

Scalar multiplication

```
In : import numpy as np

In : A = np.array([[1, 2], [3, 4]], dtype=np.float64)

In : 100 * A
Out:
array([[100., 200.],
       [300., 400.]], dtype=float32)
```

Unary operations

```
In : import numpy as np

In : A = np.array([[1, 2], [3, 4]], dtype=np.float64)

In : np.log(A + 1)
Out:
array([[0.6931472, 1.0986123],
       [1.3862944, 1.609438 ]], dtype=float32)
```

Elementwise matrix addition

```
In : import numpy as np

In : A = np.array([[1, 2], [3, 4]], dtype=np.float64)

In : A + A
Out:
array([[2., 4.],
       [6., 8.]], dtype=float32)
```

Matrix multiplication

```
In : import numpy as np

In : A = np.array([[1, 2], [3, 4]], dtype=np.float64)

In : A.dot(A)
Out:
array([[ 7., 10.],
       [15., 22.]], dtype=float32)
```

Slicing a matrix

```
In : B = np.array(np.arange(16), dtype=np.float64)
In : B.shape = (4, 4)
In : B[:,:]
Out:
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.]])
```

Slicing a matrix (2)

```
In : B = np.array(np.arange(16), dtype=np.float64)
In : B.shape = (4, 4)
In : B[0,:]
Out:
array([ 0.,  1.,  2.,  3.])
```

Slicing a matrix (3)

```
In : B = np.array(np.arange(16), dtype=np.float64)
In : B.shape = (4, 4)
In : B[:,0]
Out:
array([ 0.,  4.,  8.,  16.])
```

Slicing a matrix (4)

```
In : B = np.array(np.arange(16), dtype=np.float64)
In : B.shape = (4, 4)
In : B[:3,:3]
Out:
array([[ 0.,  1.,  2.],
       [ 4.,  5.,  6.],
       [ 8.,  9., 10.]])
```

"Broadcasting" in NumPy

- Operations expect matrices with same shape
- **Broadcasting** relaxes this constraint
- Dimensions are *compatible* if:
  - They are equal, or
  - One of them is 1
- "Broadcast" smaller matrix over larger one

"Broadcasting" in NumPy

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} - \begin{bmatrix} 6 & 7 & 8 & 9 \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} - \begin{bmatrix} 6 & 7 & 8 & 9 \\ 6 & 7 & 8 & 9 \\ 6 & 7 & 8 & 9 \\ 6 & 7 & 8 & 9 \end{bmatrix}$$

"Broadcasting" in NumPy (2)

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} - \begin{bmatrix} 1.5 \\ 5.5 \\ 9.5 \\ 13.5 \end{bmatrix}$$

$$\rightarrow \begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix} - \begin{bmatrix} 1.5 & 1.5 & 1.5 & 1.5 \\ 5.5 & 5.5 & 5.5 & 5.5 \\ 9.5 & 9.5 & 9.5 & 9.5 \\ 13.5 & 13.5 & 13.5 & 13.5 \end{bmatrix}$$

Summarization and "broadcasting"

```
In : B = np.array(np.arange(16), dtype=np.float64)
In : B.shape = (4, 4)
In : B
Out:
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.]])

In : B - B.mean(0) # [6., 7., 8., 9.]
Out:
array([[-6., -6., -6., -6.],
       [-2., -2., -2., -2.],
       [ 2.,  2.,  2.,  2.],
       [ 6.,  6.,  6.,  6.]])
```

Summarization and "broadcasting"

```
In : B = np.array(np.arange(16), dtype=np.float64)
In : B.shape = (4, 4)
In : B
Out:
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.],
       [12., 13., 14., 15.]])

In : B - B.mean(1).reshape((4,1)) # [1.5, 5.5, 9.5, 13.5]
Out:
array([[-1.5, -0.5,  0.5,  1.5],
       [-1.5, -0.5,  0.5,  1.5],
       [-1.5, -0.5,  0.5,  1.5],
       [-1.5, -0.5,  0.5,  1.5]])
```

Advanced NumPy

- Random number generation
- Linear algebra
  - Matrix inversion
  - Singular value decomposition
  - Linear equation solver
- Sparse matrices
- Memory-mapped arrays