

Mathematical Software Programming (02635)

Lecture 6 — October 9, 2025

Instructor: Martin S. Andersen

Fall 2025



This week

Topics

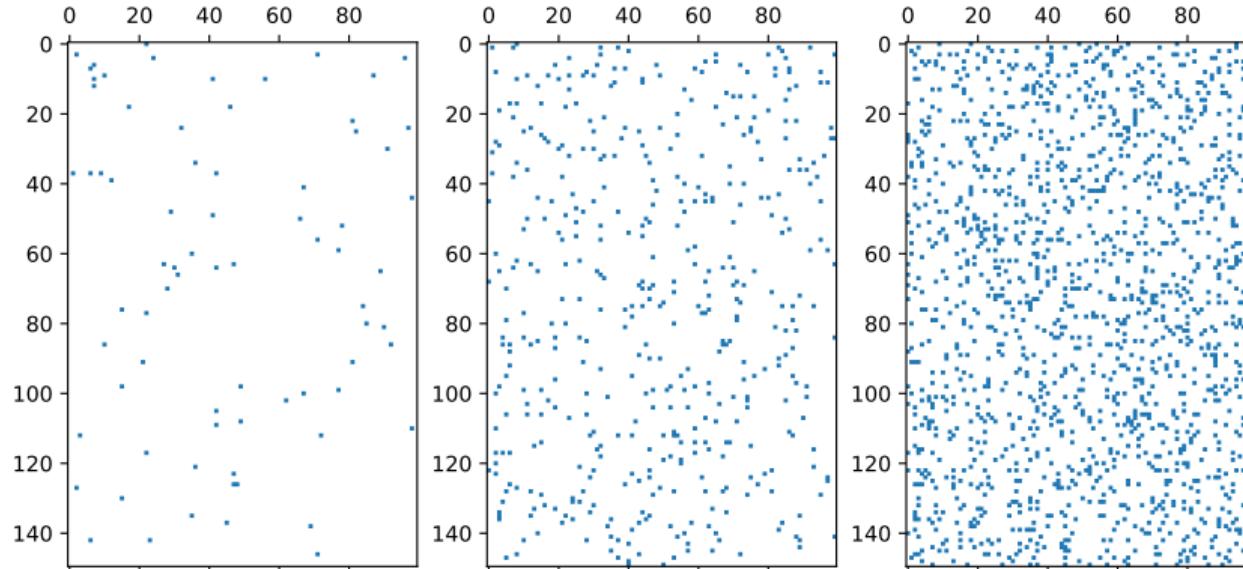
- ▶ Sparse matrices
- ▶ Strings and files

Learning objectives

- ▶ Describe and use **data structures** such as arrays, linked lists, stacks, and queues.
- ▶ Choose appropriate data types and **data structures** for a given problem.

What is a sparse matrix?

A matrix $A \in \mathbb{R}^{m \times n}$ is *sparse* if it contains relatively few nonzero entries.



No precise definition: sparse if there are “*enough zeros that it pays to take advantage of them*”

Sparse matrix representations: coordinate format (COO)

$$A = \begin{bmatrix} 1.0 & 0 & 0 & 4.0 \\ 0 & 2.0 & 3.0 & 0 \\ 5.0 & 0 & 6.0 & 7.0 \end{bmatrix} \quad S = (3, 4)$$

Store shape S and list of row index, column index and value (i, j, v) for each nonzero entry:

$$T = ((1, 1, 1.0), (3, 1, 5.0), (2, 2, 2.0), (2, 3, 3.0), (3, 3, 6.0), (1, 4, 4.0), (3, 4, 7.0))$$

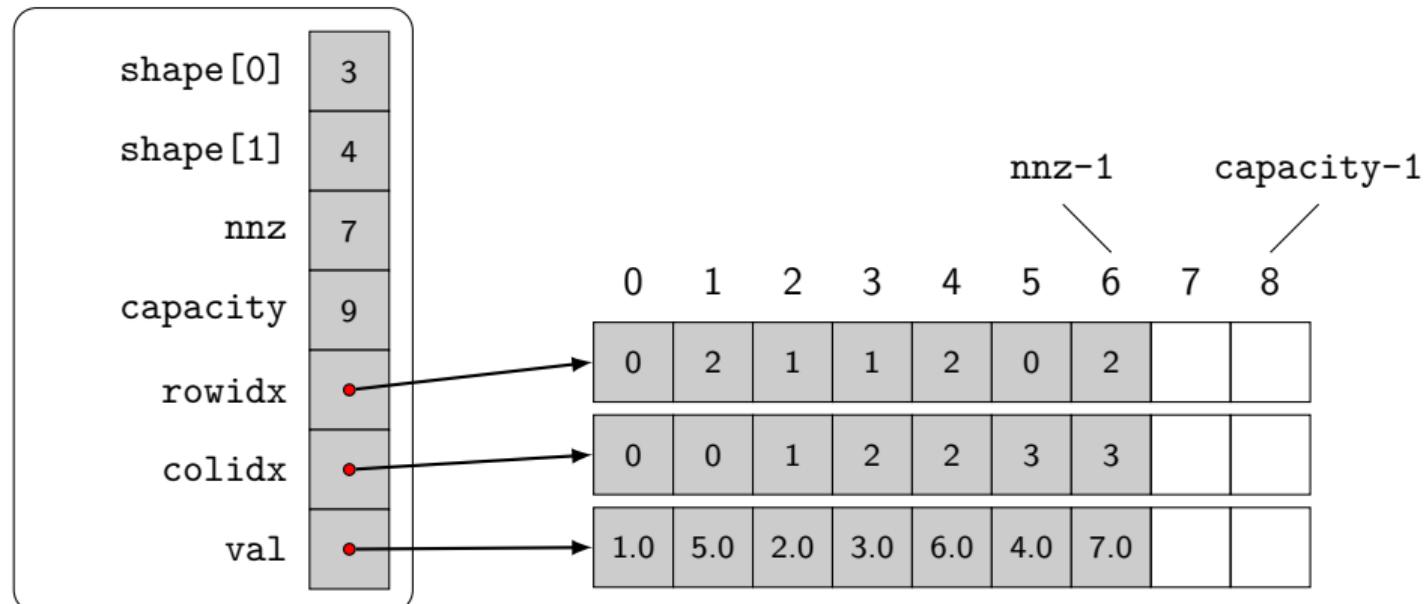
- ▶ several entries with same row/column index typically interpreted additively

$$(3, 1, 1.0) \in T \wedge (3, 1, 4.0) \in T \quad \equiv \quad (3, 1, 5.0) \in T$$

- ▶ T may be unordered or ordered by row/column
- ▶ representation is minimal if $|T|$ is equal to the number of nonzeros in A
- ▶ sum of two sparse matrices represented by lists T_1 and T_2 : $T_1 \frown T_2$
- ▶ pruning may be required to obtain minimal representation

Data structure for coordinate format

Data structure based on dynamic arrays (0-based indices)



Sparse matrix representations: compressed format (CSC/CCS, CSR/CRS)

$$A = \begin{bmatrix} 1.0 & 0 & 0 & 4.0 \\ 0 & 2.0 & 3.0 & 0 \\ 5.0 & 0 & 6.0 & 7.0 \end{bmatrix} \quad S = (3, 4)$$

Compressed column sparse (CCS) or compressed sparse column (CSC)

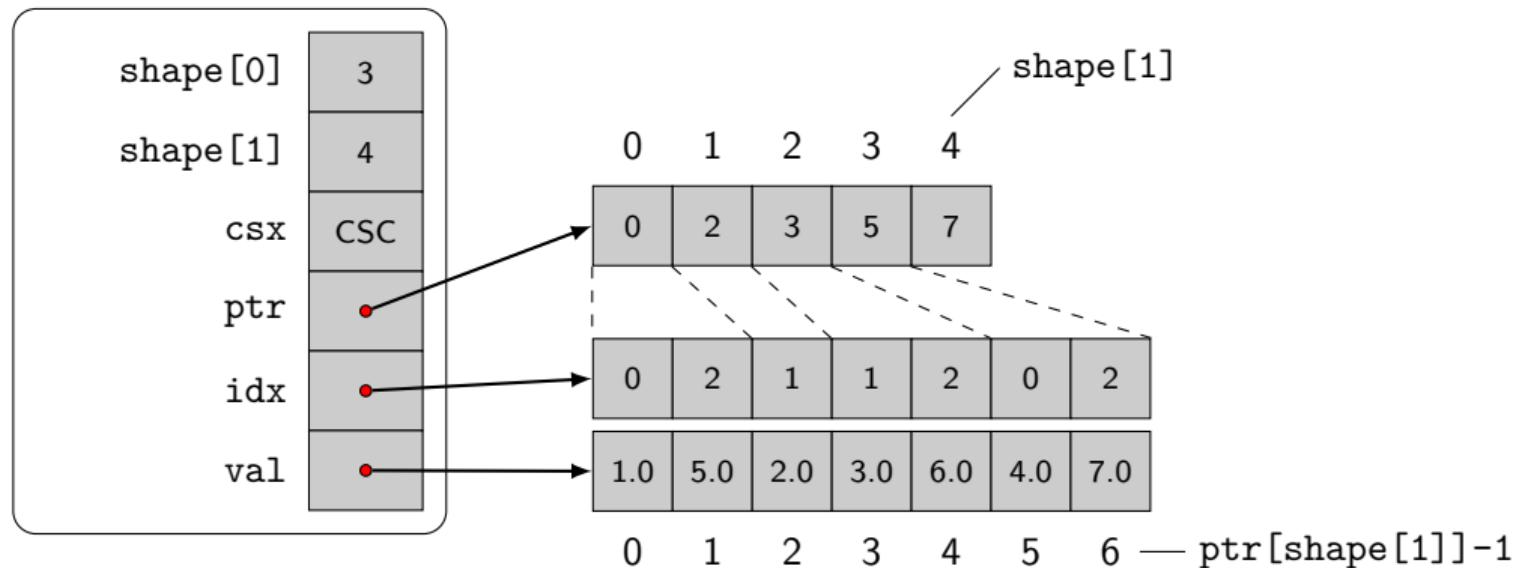
$$C = \left(\underbrace{\left((1, 1.0), (3, 5.0) \right)}_{C_1}, \underbrace{\left((2, 2.0) \right)}_{C_2}, \underbrace{\left((2, 3.0), (3, 6.0) \right)}_{C_3}, \underbrace{\left((1, 4.0), (3, 7.0) \right)}_{C_4} \right)$$

Compressed row sparse (CRS) or compressed sparse row (CSR)

$$R = \left(\underbrace{\left((1, 1.0)(4, 4.0) \right)}_{R_1}, \underbrace{\left((2, 2.0), (3, 3.0) \right)}_{R_2}, \underbrace{\left((1, 5.0), (3, 6.0), (4, 7.0) \right)}_{R_3} \right)$$

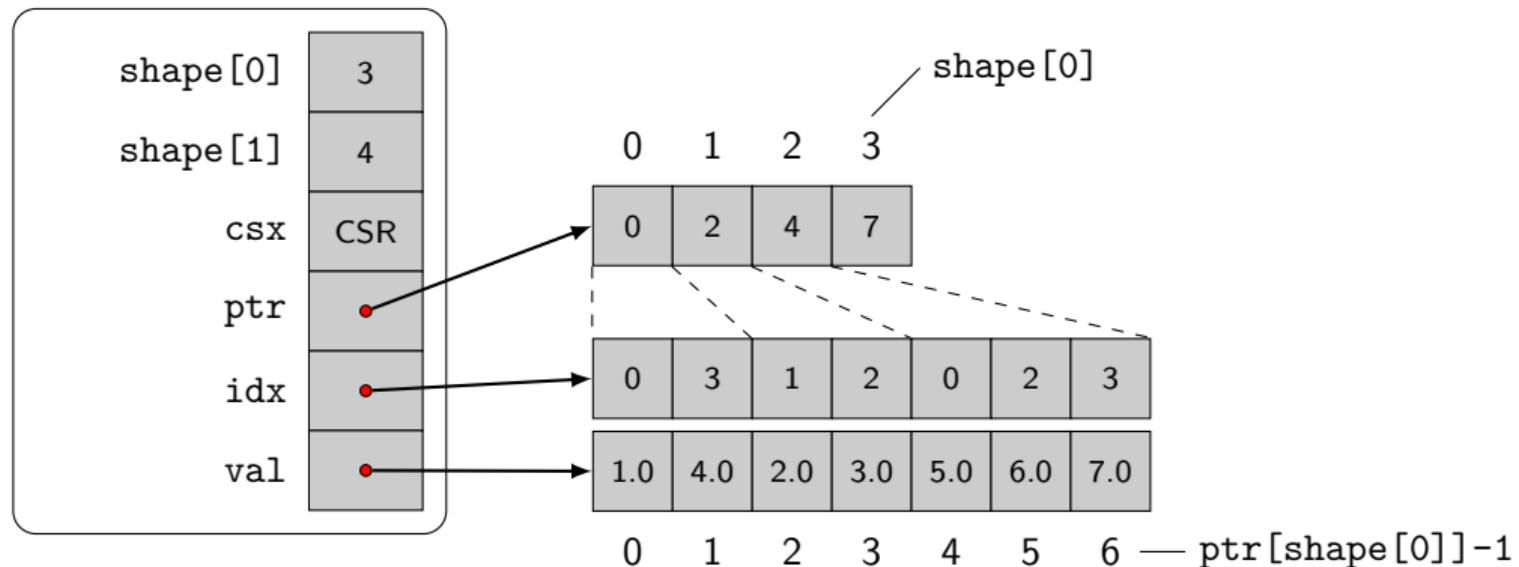
Data structure for compressed format

Data structure based on arrays (0-based indices)



Data structure for compressed format

Data structure based on arrays (0-based indices)



Sparse representations: comparison

- ▶ $A \in \mathbb{R}^{m \times n}$ is sparse with $K \ll mn$ nonzero entries
- ▶ s_i is the size of an index (in bytes)
- ▶ s_v is the size of a value (in bytes)

Representation	Storage	Access
Dense/full	mns_v	$O(1)$
COO	$(2s_i + s_v)K$	$O(K)$
CSC/CCS	$(s_i + s_v)K + (n + 1)s_i$	$O(\max_j C_j)$
CSR/CRS	$(s_i + s_v)K + (m + 1)s_i$	$O(\max_i R_i)$

Example: unit column vector $(1, 0, \dots, 0)$ of length 10^9

- ▶ coordinate representation: $s_v + 2s_i$ bytes
- ▶ compressed sparse column: $s_v + 3s_i$ bytes
- ▶ compressed sparse row: $s_v + (10^9 + 2)s_i$ bytes

Strings

A string is a *null-terminated* array of characters

```
char s[] = "Hello World!";
printf("%s\n",s);
printf("s is a char array of length %zu\n",sizeof(s));
printf("s is a string of length %zu\n",strlen(s));
```

What is the length of the char array s?

What is the length of the string?

- ▶ null-termination character is \0 (called “NUL”)
- ▶ s[0] is the character H, s[11] is the character !, and s[12] is \0
- ▶ the character array may be (much) longer than the string
- ▶ include <string.h> to use functions such as strlen() or strcmp()

Character constants, characters arrays, and string literals

```
int main(void) {
    char c = 'A';      // c is a char
    char s[] = "s is an array of chars";
    char * p = "p points to first character of a string literal";

    printf("%s\n", s); // OK - s is null-terminated
    printf("%s\n", p); // OK - p points to null-terminated char array
    printf("%s\n", c); // Not OK - c is a char, not a string

    s[0] = 'S';          // OK
    p[0] = 'P';          // Not OK - undefined behavior

    return 0;
}
```

Programs with arguments

```
/* main_demo.c */
#include <stdio.h>
int main(int argc, char const *argv[]) {
    printf("The user entered %d strings:\n",argc);
    for (int i=0;i<argc;i++)
        printf("%s\n",argv[i]);
    return EXIT_SUCCESS;
}
```

Running the program with three arguments yields the following output:

```
$ ./main_demo string1 string2 string3
The user entered 4 strings:
./main_demo
string1
string2
string3
```

Unsafe functions

Example: reading a string with `gets()`

```
char name_buffer[8];
printf("What is your name? ");
gets(name_buffer);           // C11: 'gets' removed, 'gets_s' optional
```

- ▶ What happens if the user enters a name with more than 8 characters?
- ▶ Operating system may issue a warning when starting the program:
warning: this program uses `gets()`, which is unsafe.

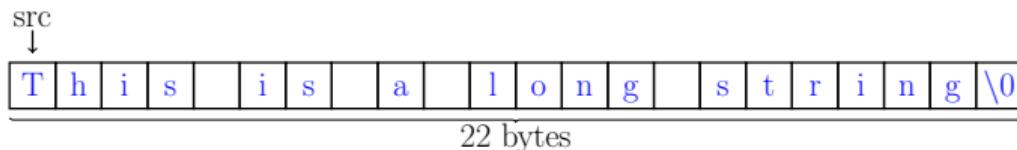
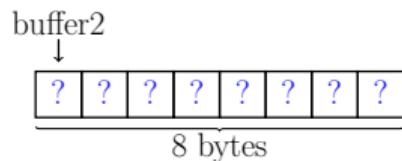
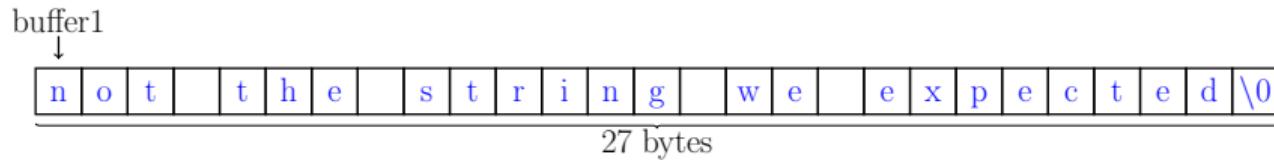
Safer alternative

```
char name_buffer[8];
printf("What is your name? ");
fgets(name_buffer, 8, stdin);
```

What happens now if the user enters a name with more than 8 characters?

String copy example

```
char buffer1[] = "not the string we expected";
char buffer2[8];
char str[] = "This is a long string";
strcpy(buffer2, str); // copy from src to buffer2
```



Working with text files

Opening and closing a file

```
FILE *fp = fopen("data.txt", "r"); // open file for reading  
  
/* ... do something with the file ... */  
  
fclose(fp); // close file  
fp = NULL; // not necessary, but good practice
```

fopen() prototype

```
FILE *fopen(const char * name, const char * mode);
```

- ▶ several modes: reading ("r"), writing ("w"), appending ("a")
- ▶ FILE is a struct defined in stdio.h (*opaque* data structure)
- ▶ returns a FILE* (a pointer to a FILE)

Reading and writing text to a file

Input prototypes

```
/* Read single character from file */
int fgetc(FILE *pfile);
/* Read string from file */
char * fgets(char *str, int nchars, FILE *pfile);
/* Read formattted input from file */
int fscanf(FILE *pfile, const char *format, ...);
```

Output prototypes

```
/* Write single character to file */
int fputc(int ch, FILE *pfile);
/* Write string to file */
int fputs(const char *str, FILE *pfile);
/* Write formatted output */
int fprintf(FILE *pfile, const char *format, ...);
```

Standard streams

Three standard streams of type FILE*

- ▶ standard input: stdin (scanf reads from stdin)
- ▶ standard output: stdout (printf writes to stdout)
- ▶ standard error: stderr

Example: write error message to stderr

```
double * arr = malloc(N*sizeof(*arr));  
if (arr==NULL)  
    fprintf(stderr,"Memory allocation failed.\n");
```

Reading strings with scanf/fscanf

```
FILE *fp;
char buf[32];
// Open file
if ((fp = fopen("data.txt", "r")) == NULL) {
    fprintf(stderr,"Error opening file.\n");
    exit(EXIT_FAILURE);
}
// Read from file
if (fscanf(fp,"%31s",buf)!=1) {
    fprintf(stderr,"Error reading from file.\n");
    exit(EXIT_FAILURE);
}
```

- ▶ fscanf and scanf return the number of input items *assigned*
- ▶ fscanf and scanf need space for the null character \0
- ▶ using %s instead of %[width]s may lead to *buffer overflow*

Working with binary files

Opening a binary file

Mode strings: reading ("rb"), writing ("wb"), appending ("ab")

Input/output prototypes

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *pfile);
size_t fwrite(void *ptr, size_t size, size_t nmemb, FILE *pfile);
```

Maintaining floating-point precision: binary vs. text format

- ▶ text file: format specifier %.17g for double and %.9g for float
- ▶ C11: macros *_DECIMAL_DIG (FLT,DBL,LDBL) defined in float.h
- ▶ binary file: 8 bytes for double and 4 bytes for float

Example: write array to binary file

```
/* Declare double array */
double data[] = {0.1,0.2,-0.1,-2.0,5.0,3.0};

/* Length of array */
size_t n = sizeof(data)/sizeof(double);

/* Open file and write data */
FILE *fp = fopen("data.dat","wb");
if ( fp == NULL ) exit(EXIT_FAILURE);
size_t ret = fwrite(data, sizeof(*data), n, fp);

/* Check return value and close file */
if ( ret != n ) fprintf(stderr,"Ups! Write error...\n");
fclose(fp);
fp = NULL;
```

Example: read array from binary file

```
/* Declare double array */
double data[100];

/* Open file and read (at most) 100 doubles */
FILE *fp = fopen("data.dat", "rb");
if ( fp == NULL ) exit(EXIT_FAILURE);
size_t ret = fread(data, sizeof(*data), 100, fp);
printf("Read %zu doubles.\n", ret);

/* Close file */
fclose(fp);
fp = NULL;
```

Error handling

- ▶ `<errno.h>` defines integer `errno` (initially zero)
- ▶ `<string.h>` defines function `char *strerror(int errnum)`
- ▶ `<stdio.h>` defines function `void perror(const char *s)`

```
int main(void) {
    int errnum;
    FILE *fp;
    if ((fp = fopen("/path/to/file", "r"))==NULL) {
        errnum = errno;
        perror("Error printed by perror");
        fprintf(stderr, "Error opening file: %s\n", strerror(errnum));
        return EXIT_FAILURE;
    }
    /* do something with file */
    fclose(fp);
    return EXIT_SUCCESS;
}
```

Big-endian vs little-endian

Recall that many data types consist of multiple bytes

- ▶ a double consists of 8 bytes
- ▶ a long (typically) consists of 4 bytes or 8 bytes

What is the order of the bytes in memory?

Big-endian

Most significant byte has smallest memory address

Little-endian

Least significant byte has smallest memory address

Endianness

Checking for endianness

```
int i = 1;
char *p = (char *) &i;
if (*p == 1)
    printf("Your system is little-endian.\n");
else if (*(p+sizeof(int)-1) == 1)
    printf("Your system is big-endian.\n");
```

Common predefined macros

```
#define __BYTE_ORDER__ __ORDER_LITTLE_ENDIAN__
#define __ORDER_BIG_ENDIAN__ 4321
#define __ORDER_LITTLE_ENDIAN__ 1234
#define __ORDER_PDP_ENDIAN__ 3412
```

List built-in macros with C preprocessor: `cpp -dM /dev/null`

64-bit programming models

Model	short	int	long	long long	pointer	OS/compiler(s)
LP64	16	32	64	64	64	Most Unix-like systems
LLP64	16	32	32	64	64	Windows/MinGW
ILP64	16	64	64	64	64	

Remark: Most of today's 32-bit systems use ILP32 (int, long, and pointers are 32-bit)

Today's exercises: triplet format for sparse matrices

- ▶ Quiz (strings)
- ▶ Reading/writing data
- ▶ Autolab