UniTs - University of Trieste

Faculty of Scientific and Data Intensive Computing

Department of mathematics informatics and geosciences

Advanced Programming for Astrophysics

Lecturers:

Prof. Murante Giuseppe
Prof. Taffoni Giuliano

Author:
Andrea Spinelli

December 14, 2025

github.com/Spina02 andreaspinelli2002@gmail.com

Preface

This document was prepared for the Advanced Programming for Astrophysics course, held by Prof. Murante Giuseppe and Prof. Taffoni Giuliano in the academic year 2025/2026.

Contents

1 Introduction

Operating systems (OS) are the core software that manage a computer's hardware and software resources. Among the most widely used are Microsoft Windows, Apple's macOS, and the open-source Linux. While they differ in many aspects, both macOS and Linux are part of the ***Unix-like*** family of operating systems. These systems are renowned for their stability, security, and a powerful command-line interface known as the ***shell***.
This chapter provides an introduction to the Unix shell, exploring its fundamental role in interacting with the system.

File System

A crucial component of the operating system is the file system, which organizes files and directories on a computer. It is a crucial component of the operating system that allows users to store, retrieve, and manage data efficiently. In unix-like systems, the file system is organized in a hierarchical structure.

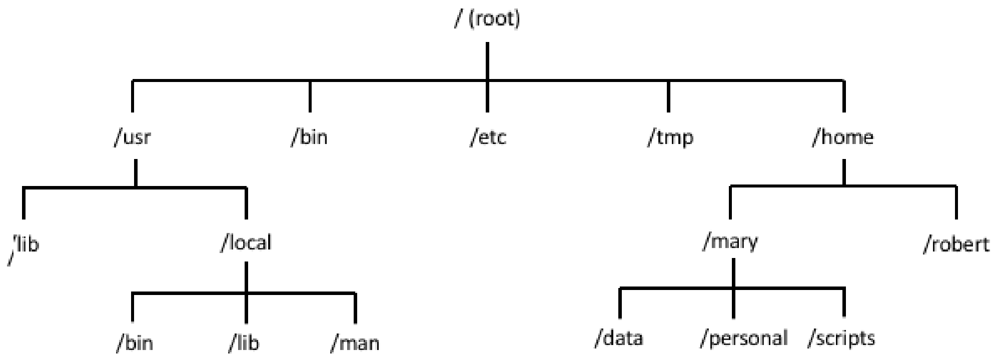

Figure 1.1: The File System

Path

A path specifies the unique location of a file or directory within the file system's hierarchy. Paths can be ***absolute***, starting from the root directory (`/`), or ***relative*** to the current working directory.

`/`	Root Directory	`~`	Home Directory
`.`	Current Directory	`..`	Parent Directory

Example: ***Example of a path***

- a file in the user's Desktop: `/home/user/Desktop/file.txt` or `~/Desktop/file.txt`
- a file in the current directory: `./file.txt`
- a file in the parent directory: `../file.txt`
- a file in the home directory: `~/file.txt`

1.1 The Unix Shell

What is a shell?

The shell is the primary interface between users and the computer's core system. When you type commands in a terminal, the shell interprets these instructions and communicates with the operating system to execute them, making complex system operations accessible through simple commands.

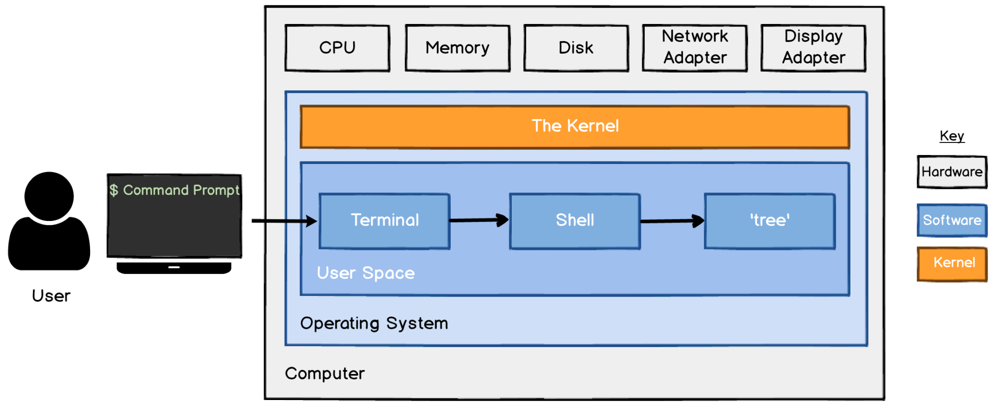

Figure 1.2: The Unix Shell

Definition: *Shell*

A shell is a program that provides the traditional, text-only user interface for Linux and other UNIX-like operating systems. Its primary function is to read commands that are typed into a console [...] and then execute (i.e., run) them. The term shell derives its name from the fact that it is an outer layer of an operating system. A shell is an interface between the user and the internal parts of the OS (at the very core of which is the kernel). *~Linfo*

1.1.1 Basic Commands

A command is nothing more than a program that is executed by the shell. The usual syntax is:

```
command [options] [arguments]
```

The options can be passed setting the corresponding flag e.g. `-h` for help.

Navigation Commands

It is possible to navigate and manage files or directories using the following commands:

Command	Description
`pwd`	Print Working Directory
`cd`	Change Directory
`ls`	List Directory
`mkdir`	Make Directory
`rm`	Remove File or Directory
`cp`	Copy File or Directory
`mv`	Move File or Directory
`touch`	Create File

Recursive Commands

`rm`, `cp` and many other commands can be used recursively into a folder using the flag `-r` (recursive).

Help Flag

Many commands support the `--help` flag to display information about the command and its usage.

Tip: *Hidden Files*

Hidden files are files that are not shown by default when listing the contents of a directory. They are usually marked with a dot (.) at the beginning of their name. e.g. `.config`

Searching Tools

Unix-like systems provide powerful tools for searching files and content. These tools often support regular expressions (regex), which are patterns used to match character combinations in text.

File Search Commands

Command	Description	Example
`find`	Search for files and directories	`find /home -name "*.txt"`
`locate`	Fast file search using database	`locate filename`
`grep`	Search text within files	`grep "pattern" file.txt`
`which`	Find location of executable	`which gcc`
`whereis`	Find binary, source, and manual pages	`whereis python`

Wildcards and Pattern Matching

Wildcards are special characters that represent one or more characters in a filename or text string:

Character	Name	Description
`*`	asterisk	Matches zero or more characters
`?`	question mark	Matches exactly one character
`[]`	brackets	Matches any single character within brackets
`[!]`	negated brackets	Matches any character not in brackets

Example: *Wildcard Examples*

- `*.txt` : matches all files ending with .txt
- `file?.c` : matches file1.c, file2.c, but not file10.c
- `[abc]*` : matches files starting with a, b, or c
- `[!0-9]*` : matches files not starting with digits

Operators

Commands can be combined using operators to manipulate input and output streams:

Operator	Name	Description
`>`	output redirection	Redirects standard output to a file, overwriting its contents
`>>`	append redirection	Redirects standard output, appending it to the end of a file
`|`	pipe	Passes the output of a command as input to another
`&&`	logical AND	Executes the next command only if the previous one succeeds
`||`	logical OR	Executes the next command only if the previous one fails
`;`	separator	Executes commands sequentially, regardless of their outcome

1.1.2 Environment Variables

Environment variables are dynamic values that affect the behavior of processes and programs running on the system. They provide a way to pass configuration information to applications without hardcoding values into the program. These variables are stored in the system's environment and can be accessed by any process.

Variable	Description	Example
`PATH`	Colon-separated list of dirs to search for executables	`/usr/bin:/bin:/usr/sbin`
`HOME`	Path to the user's home directory	`/home/username`
`USER`	Current username	`username`
`SHELL`	Path to the current shell	`/bin/bash`
`PWD`	Current working directory	`/home/username/Desktop`

To view environment variables, use the `env` command or `echo $VARIABLE_NAME`. To set a variable temporarily, use `export VARIABLE_NAME=value`. For permanent changes, modify configuration files like `.bashrc` or `.bash_profile`.

1.1.3 File Permissions

Unix-like systems use a permission system to control access to files and directories. Each file has three types of permissions: read (r), write (w), and execute (x). These permissions are assigned to three categories of users: owner (user), group, and others.

Permission Structure

File permissions are displayed using a 10-character string where the first character indicates the file type, and the remaining nine characters represent permissions for owner, group, and others:

Character	Position	Meaning
`-`	1st	Regular file
`d`	1st	Directory
`r`	2nd, 5th, 8th	Read permission
`w`	3rd, 6th, 9th	Write permission
`x`	4th, 7th, 10th	Execute permission

For instance, `-rwxr-xr--` means:

- Regular file (-)
- Owner: read, write, execute (rwx)
- Group: read, execute (r-x)
- Others: read only (r–)

Octal Notation

Permissions can be represented numerically using octal notation:

(`r`) Read	**4**	`100`
(`w`) Write	**2**	`010`
(`x`) Execute	**1**	`001`

By summing these values for each user category (owner, group, and others), we get a three-digit code. For example, `rwx` permissions translate to $4+2+1=7$, and `r-x` to $4+0+1=5$. Thus, a full permission string like `rwxr-xr-x` can be concisely represented as 755.
Common combinations include: **755** (`rwxr-xr-x`), **644** (`rw-r--r--`), and **600** (`rw———`).

Permission Commands

Command	Description	Example
`chmod`	Change file permissions	`chmod 755 script.sh`
`chown`	Change file ownership	`chown user:group file.txt`
`chgrp`	Change group ownership	`chgrp staff file.txt`
`umask`	Set default permissions	`umask 022`

TODO: notes about file viewing and text processing, few words about multiprocessing and context switching

2 Git, GitHub and SSH

2.1 Git and GitHub

Git is a distributed version control system that allows you to track changes to your code over time. It is a tool that is used to manage your codebase and collaborate with others.
There are different services that provide Git repositories, one of the most popular is GitHub. GitHub is a web-based platform that provides a Git repository hosting service. It allows you to create a repository and invite others to collaborate on it. It also provides a web interface for viewing and editing the code, as well as a command line interface for interacting with the repository.

2.2 Internet and SSH

In this section we briefly explain how the Internet works and how to use SSH securely.

2.2.1 Internet

Computers communicate on the Internet using the TCP/IP protocol suite. An IP address (e.g. `192.168.1.1` in IPv4) identifies a host on the network; services on that host are selected by port numbers (e.g. 22 for SSH, 80/443 for HTTP/HTTPS). DNS translates human-readable domain names (e.g. `www.example.com`) into IP addresses.
Data travels in *packets*. Each packet has a header with routing metadata and a *payload* with application data.

Header	
Source	IP: `192.168.1.10`, Port: `54321`
Destination	IP: `172.217.22.14`, Port: `443`
Protocol	`TCP`
Payload	
Data	`Hello, world!`

Table 2.1: Simplified structure of an Internet packet

Observation: ***Encryption in transit***

If traffic is not encrypted, the payload can be intercepted and read. Encryption in transit is typically provided by TLS (for the web, HTTPS) or by SSH (for remote access and related transfers).

A *LAN* (Local Area Network) covers a limited area (home, office, school). Many LANs use private address ranges (`10.0.0.0/8`, `192.168.0.0/16`, ...) and a router that performs *NAT* (Network Address Translation): it maps private addresses/ports to a public address on the Internet (often using *PAT*, Port Address Translation). This mapping is done by the router and lets many private hosts share one public IP.

2.2.2 SSH

SSH (Secure Shell) is a secure protocol for remote administration and tunneling. It uses port 22 by default and supports public-key authentication.
In the `~/.ssh/` directory you typically find:

- `id_ed25519` and `id_ed25519.pub`: your private and public key (recommended). Historical variants: `id_rsa`, `id_rsa.pub`.
- `known_hosts`: fingerprints of servers you have connected to (helps prevent MITM attacks).
- `config`: client options for specific hosts (alias, port, user, key).
- On a server: `/.ssh/authorized_keys` lists public keys allowed to connect.

There are multiple tools that use SSH as transport:

- `ssh`: opens a remote session to run commands/shell.
- `scp`: copies files between local/remote hosts over SSH.
- `rsync`: efficiently synchronizes directories; often uses SSH as its channel (`rsync -e ssh`).
- `git`: version control system; can use SSH as transport for clone/push.

Practical examples:

```
# Generate a new key pair (Ed25519 recommended)
ssh-keygen -t ed25519 -C "name.surname@example.com"

# Install your public key on the server
ssh-copy-id user@server.example.com

# Test access
ssh user@server.example.com

# File transfers
scp file.txt user@server.example.com:/path/destination/
rsync -avz -e ssh folder/ user@server.example.com:/path/destination/

# Using Git over SSH
git clone git@github.com:organization/repository.git
```

Tip: ***Graphical clients and X forwarding***

Beyond the command line, graphical SSH clients exist. On Unix-like systems you can forward X11 applications with `ssh -X` (if both server and client allow it); on Windows you can use clients like OpenSSH with an installed X server.

3 Intro to C

C is a general-purpose programming language created in the 1970s by Dennis Ritchie. Its design gives programmers direct access to the underlying hardware, making it highly efficient and powerful. It is instrumental in implementing operating systems, device drivers, and protocol stacks.
The easiest code we can write in C is:

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
}
```

- `#include <stdio.h>` : is a preprocessor directive.
- `int main()` : is the main function.
  - `int` : is the return type of the function, if omitted, the function returns an integer.
  - `main()` : is the name of the function.
- `printf(" ... ")` : is the function that prints the string `"Hello, World!"` to the console.
- `{}` : is the block of code that contains the statements of the function.

3.1 Compilation

To compile a C program, use the following command:

```
gcc main.c -o main.x
```

The compiler generates the object code, which is a binary file that contains the machine language translation of the program. The command above actually executes 3 steps:

1. **Preprocessing**: This step expands the macros and includes the header files. e.g. `#include <stdio.h>`

   ```
   gcc -E main.c -o main_preprocessed.c
   ```

2. **Compilation**: The compiler translates the preprocessed code into object code.

   ```
   gcc -c main_preprocessed.c -o main.o
   ```

   During this step, it is possible to set the `-I` flag to include other directories (useful if the headers are not in the default directory). e.g. `-I /path/to/include`
3. **Linking**: The linker combines the object code with the standard library to create an executable.

   ```
   gcc main.o -o main.x
   ```

The `-o` flag is used to set the output file name.

During this step, it is possible to set the `-L` flag to include other directories (useful if the libraries are not in the default directory). e.g. `-L /path/to/lib` and to link against external libraries using the `-l` flag. e.g. `-lmylib`

3.1.1 Libraries

A library is a collection of pre-compiled code, such as functions and data structures, that can be reused across multiple programs. They promote code modularity and prevent developers from having to reimplement common functionalities, such as input/output operations or mathematical calculations. There are two types of libraries:

- `lib___.a` : is a static library.
- `lib___.so` : is a shared library.

The difference between the two is that the static library is linked directly into the executable at compile time, while the shared library is loaded at runtime.

3.1.2 Data Types

C provides several fundamental data types to represent different kinds of data. Understanding these types is crucial for writing efficient and correct programs.
TODO: notes about data types

Increment and Decrement

The `++` and `--` operators are used to increment and decrement the value of a variable.
There are two types of increment and decrement operators:

- `++i` (`--i`): is the prefix increment (decrement) operator. The value of the variable is incremented (decremented) before it is used.
- `i++` (`i--`): is the postfix increment (decrement) operator. The value of the variable is incremented (decremented) after it is used.

3.1.3 Directives

Directives are used to control the compilation process. They are used to define constants, macros, and to conditionally compile code.

Macros

Macros are used to define values that are not expected to change during the execution of the program. They are defined using the `#define` directive.

```c
#define CYCLE 0
```

Conditional Compilation

Conditional compilation is used to compile code only if a certain condition is met. They are defined using the `#ifdef`, `#ifndef`, `#else`, `#elif`, and `#endif` directives.

```c
#ifdef CYCLE
    // code to compile if CYCLE is defined
#endif
```

3.1.4 Functions

Functions are used to group code into ***routines***: reusable units of code.

```
return-type function-name(parameters declarations, if any) {
    declarations
    statements
    return return-value;
}
```

Parameters are passed to the function by value, meaning that a copy of the argument is passed to the function. If the argument is a *pointer* (we will see), the function will modify the original argument.

It is possible to return only a single value. We will see later that it is possible to return a *pointer* or a *struct*, which can be useful to return multiple values from a function.

If the return value is not the same specified in the function declaration, the compiler probably won't notice, so be careful.

Variable scopes

- **Local scope**:
  Variables declared inside a function are said to have ***local scope***. This means they only exist and are accessible from within that specific function. Once the function finishes its execution, these variables are destroyed. This allows different functions to use the same variable names without causing conflicts.
- **Global scope**:
  Variables declared outside any function are said to have ***global scope***. This means they are accessible from any function in the program.

Observation: ***Variable shadowing***

If a local variable is declared with the same name as a global variable, the local variable takes precedence within its scope. This is known as *variable shadowing*. The global variable is temporarily hidden, and any reference to that variable name inside the function will refer to the local one. The global variable remains unaffected outside of this scope.

```c
#include <stdio.h>

int a = 10; // Global variable

void myFunction() {
    int a = 20; // Local variable shadows the global one
    printf("Inside function, a = %d\n", a); // Prints 20
}

int main() {
    printf("Before function call, a = %d\n", a); // Prints 10
    myFunction();
    printf("After function call, a = %d\n", a);  // Prints 10
    return 0;
}
```

3.1.5 Variable Types

In C, every variable has a type, which dictates the size and layout of its memory, the range of values it can store, and the set of operations that can be applied to it. The fundamental types are used to represent integers, floating-point numbers, and characters. The table below shows common data types and their typical sizes on modern systems.

Type	Typical Size	Description
`char`	1 byte	Stores a single character or a small integer.
`short int`	2 bytes	Short integer.
`int`	4 bytes	The most natural size of integer for the machine.
`long int`	8 bytes	Long integer.
`float`	4 bytes	Single-precision floating-point number.
`double`	8 bytes	Double-precision floating-point number.

Tip: ***implementation-defined sizes***

Note that the exact sizes of these types are implementation-defined and can vary across different systems and compilers. You can use the `sizeof` operator to determine the size of a type on your machine.

Integer types () `char`, `short`, `int`, `long`) can be either `signed` (the default) or `unsigned`. A `signed` type can hold both positive and negative values, while an `unsigned` type can only hold non-negative ones. By using the sign bit to store value instead, an `unsigned` type can represent a maximum value twice as large as its signed counterpart. For example, a `signed char` typically ranges from -128 to 127, whereas an `unsigned char` ranges from 0 to 255.

Type conversion

C automatically converts the smaller type to the larger type.

```c
int x = 10;
float y = 20.5;
float z = x + y; // z = 30.5
```

For the other cases, we need to use an explicit cast.

```c
double x = 10.;
int y = 7;
int z = y + (int) x; // z = 17
```

Declaration and initialization

```c
// only declaration
int x, y, z;
char c, line[1000];

// declaration and initialization
int x = 10, y = 20, z = 30;
char line[1000] = "Hello"; // or = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

3.1.6 Operators

C provides a rich set of operators, which are summarized in the table below.

Category	Operator	Description
Arithmetic	`+`	Addition
	`-`	Subtraction
	`*`	Multiplication
	`/`	Division
	`%`	Modulo (remainder of division)
Relational	`==`	Equal to
	`≠`	Not equal to
	`>`	Greater than
	`<`	Less than
	`≥`	Greater than or equal to
	`≤`	Less than or equal to
Logical	`&&`	Logical AND
	`||`	Logical OR
	`!`	Logical NOT
Bitwise	`&`	Boolean AND
	`|`	Boolean OR
	`^`	Boolean XOR
	`~`	Boolean NOT
	`<<`	Left shift (≡ int mul. by 2)
	`>>`	Right shift (≡ int div. by 2)

It is possible to use the compound assignment operators (`+=` , `-=` , ...), which are equivalent to the corresponding arithmetic operator followed by the assignment operator.

```c
int x = 10;
x += 5; // x = 15
```

⚠ Warning: ***Precedence***

Operations have a precedence, which is used to determine the order in which they are evaluated. It is possible to change the order of evaluation using parentheses.

3.2 Control Structures

Control structures are used to control the flow of the program. They are used to execute a block of code only if a certain condition is met.

3.2.1 Conditional statements

If-Else

```c
if (condition1) {
    // code
}
else if (condition2) {
    // code
}
else { // any other condition
    // code
}
```

It is possible to nest if statements one inside the other. If you do, make sure to use braces to avoid ambiguity.

Switch

The `switch` statement is used to execute different blocks of code based on the value of a variable.

```c
switch (expression) {
    case constant1:
        // code
    case constant2:
        // code
    default: // any other value
        // code
}
```

The default behaviour of the `switch` statement is to check all the cases, even if the expression matches one of them. To avoid this, it is possible to use the `break` statement to exit the switch statement.

```c
switch (expression) {
    case constant1:
        // code
        break;
    case constant2:
        // code
        break;
    default:
        // code
        break; // implicit
}
```

3.2.2 Loops

Loops are used to execute a block of code a specified number of times. There are two main types of loops: the `while` loop and the `for` loop.

While and Do While Loops

The `while` loop is used to execute a block of code as long as a specified condition is true.

```c
while (condition) {
    // code
}
```

If the condition is never met, the block of code is not executed. If we need to execute the block of code at least once, we can use the `do while` loop.

```c
do {
    // code
} while (condition);
```

For Loop

The `for` loop is used to execute a block of code a specified number of times.

```c
for (int i = 0; i < 5; i++) {
    // code
}
```

Break and Continue

The `break` statement is used to exit a loop prematurely. The `continue` statement is used to skip the rest of the code in a loop and move to the next iteration.

```c
while (1) { // infinite loop (should be avoided)
    // code
    if (condition_1) {
        break; // exit the loop
    }
    else if (condition_2) {
        continue; // skip the code below and move to the next iteration
    }
}
```

MISSING: 03/10/25 notes

3.3 Memory Management

The C language allows programmers to manage memory manually, providing powerful capabilities for fine-grained control over how memory is allocated, used, and freed. This flexibility enables efficient use of system resources, but also requires careful attention to avoid errors such as memory leaks or accessing invalid memory.

3.3.1 Memory Layout

The memory layout of a program (Figure 3.1) is the way the memory is organized in the computer.

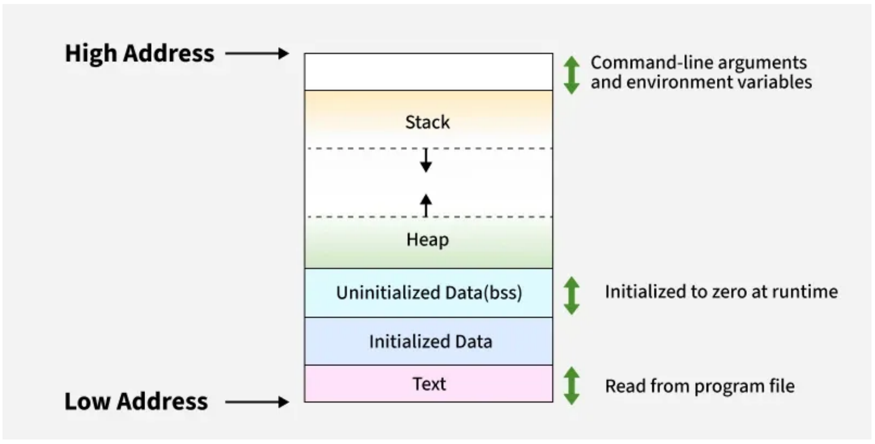

Figure 3.1: The Memory Layout

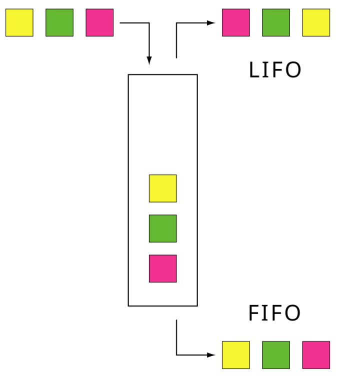

Figure 3.2: FIFO and LIFO

- **Stack**: It is used to store *local variables*, *function parameters*, and *return addresses*. Whenever a function is called, a new ***stack frame*** is created to hold its local data. The stack automatically grows and shrinks as functions are called and return. Variables stored here are automatically removed when the function finishes, following a Last In, First Out (LIFO) order.
- **Heap**: It is used for dynamic memory allocation, which means memory that is explicitly requested at runtime (using functions like `malloc` and `free` in C). Memory allocated on the heap persists until it is explicitly freed by the programmer, and is not automatically cleaned up.
- **BSS (Block Started by Symbol)**: It contains all global and static variables that are declared but not initialized by the programmer. The operating system initializes this region to zero before the program starts running.
- **Data**: It stores global and static variables that are explicitly initialized by the programmer. For example, `int x = 5;` at global scope would be stored here. The values in this segment are set to their initial values when the program starts.
- **Text**: Also known as the code segment, it contains the compiled machine code instructions of the program itself. It is typically marked as read-only to prevent accidental modification of the program's instructions during execution.

⚠ **Warning**: ***Reserved addresses***

Low addresses are reserved for the kernel and other critical system components. Do not use them for your own variables if you don't know what you are doing.

3.3.2 Pointers

All data, variables, and functions in a C program reside in the computer's memory. Each location in memory is identified by a unique address, typically represented as a hexadecimal value. These addresses allow the program to access and manipulate data efficiently. Understanding how to work with memory addresses is fundamental in C, as it enables direct interaction with the underlying hardware and forms the basis for concepts like pointers, dynamic memory allocation, and efficient data structures.
Consider the following code and its corresponding memory diagram:

```c
int a = 1;       // a is 1
a++;             // a is now 2

int* p;          // p is a pointer to an integer
p = &a;          // p now points to a

int value = *p; // value is now 2
```

Memory	Address
a = 2	0x0
p = 0x0	0x1
value = 2	0x2
⋮	⋮

Explanation:

- `a` is an integer variable stored at address `0x0`, and after `a++`, its value is 2.
- `p` is a pointer variable stored at address `0x1`, and it holds the address of `a` (`0x0`).
- `value` is an integer variable stored at address `0x2`, and it is assigned the value pointed to by `p`, which is 2.

The `&` operator is used to get the address of a variable (e.g., `p = &a;`). The `*` operator is used to access the value stored at the address pointed to by a pointer (e.g., `value = *p;`).

Pointers arithmetic

We have seen `*` and `&` operators, used to access the value stored at the address pointed to by a pointer and to get the address of a variable, respectively. Now, let's see how pointers behaves with arithmetic operations.
The actual size of the increment (decrement) is the size of the type pointed to by the pointer.

```c
int a = 1;
int *p = &a;
p++; // this is actually p + 4 bytes

double b = 2.0;
double *q = &b;
q++; // this is actually q + 8 bytes
```

The same applies to the decrement, addition, subtraction, multiplication and division operators (the last two are rarely used with pointers).
We have seen that to access the value pointed to by the pointer, we can use the `*` operator. It is possible to perform arithmetic operations among the values pointed by pointers using the same operator.

```c
int a = 1;
int *p = &a;
(*p) += 2; // a is now 3
```

Pointers and Function Arguments

When a function is called, the arguments are passed by value, meaning that a copy of the argument is passed to the function. If the argument is a pointer, the function can modify the original argument:

```c
void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

The code above swaps the values of the two variables.

Pointers and Arrays

In C, arrays and pointers are closely related. An array name is essentially a pointer to the first element.

```c
int a[5];                   // only declaration
int a[5] = {1, 2, 3, 4, 5}; // declaration and initialization
int *ptr = a;               // pointer to the first element
```

During the declaration, the number in the square brackets is the size of the array.
The expressions `a[i]` and `*(a + i)` are equivalent. Let's see how to

```c

for (int i = 0; i < 5; i++) {
    a[i]++;
}

for (int i = 0; *(a + i) != 0; i++) {
    (*a + i)++;
}

* missing one *
```

The two loops perform the same operation. (The third one, missing, do the same but in a more efficient way - slightly noticeable for "short arrays")

Observation: ***Strings and Pointers***

Strings and pointers are basically the same thing: a string is an array of characters, ending with the null character `'\0'` .

The code below copies the string `t` into the string `s` in a compact and elegant way.

```c
void strcpy(char *s, char *t) {
    while (*s++ = *t++);
}
```

Multidimensional Arrays

Multidimensional arrays are arrays of arrays. They are declared using multiple square brackets.

```c
int a[3][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};
```

The code above declares a 2D array of integers with 3 rows and 4 columns.
The expressions `a[i][j]` and `*(a[i] + j)` are equivalent.

Tip: ***Declaring pointers and 2D arrays***

There are several ways to declare variables related to 2D arrays in C, each with different meanings and use cases:

```c
int a[3][4];    // a true 2D array: 3 rows, 4 columns
int *p[4];      // an array of 4 pointers to int
int **q;        // a pointer to a pointer to int
```

Explanation:

- `int a[3][4];` declares a contiguous block of memory for 12 integers (3 rows, 4 columns). This is the standard way to declare a 2D array.
- `int *p[4];` declares an array of 4 pointers to `int`. Each element of `p` can point to the beginning of a separate row (or any `int`), but the rows themselves are not stored contiguously unless you arrange them that way.
- `int **q;` declares a pointer to a pointer to `int`. This is often used for dynamically allocated 2D arrays, where both the rows and the columns can be allocated at runtime.

Key differences:

- The first declaration (`a`) allocates all the memory for the array at once and ensures the data is stored contiguously.
- The second (`p`) and third (`q`) declarations only allocate space for pointers, not for the actual integers. You must allocate memory for the data separately, typically using `malloc`.
- Only the first declaration (`a`) knows the size of both dimensions at compile time.

We will explore dynamic memory allocation for 2D arrays in more detail later.

3.3.3 Command-Line Arguments

In C, the `main` function can be defined to accept arguments from the command line, allowing users to pass information to your program when it starts.

```c
int main(int argc, char *argv[]) { // Your code here }
```

- `argc` (argument count) is an integer representing the number of command-line arguments passed to the program, including the program's name itself.
- `argv` (argument vector) is an array of pointers to strings (character arrays), where each string is one of the command-line arguments.

Important details:

- `argv[0]` is always the name (or path) of the program as it was invoked.
- The actual user-supplied arguments start from `argv[1]` up to `argv[argc-1]`.
- `argc` is always at least 1, since the program name is always present.

Example: ***Example***

If you run the program as follows:

```
$ ./myprog 1 -2 1
```

Then:

- `argv[0]` is `"./myprog"`
- `argv[1]` is `"1"`
- `argv[2]` is `"-2"`
- `argv[3]` is `"1"`

Warning: ***Arguments as strings***

Arguments passed to the program are treated as strings. You need to convert them to the appropriate type using the `atoi`, `atof`, `atol`, `atoll` functions.

You can use a loop to process all arguments:

```c
for (int i = 0; i < argc; i++) {
    printf("Argument %d: %s\n", i, argv[i]);
}
```

This feature is especially useful for writing flexible and user-friendly command-line programs.

Pointers to functions

Pointers to functions are used to store the address of a function. They are declared using the `typedef` keyword.

```c
typedef int (*func_ptr)(int, int);
```

The code above declares a pointer to a function that takes two `int` arguments and returns an `int`.

MISSING: Variable lenght argument list

3.3.4 Store Management

It is possible to allocate (and deallocate) memory dynamically. To do that we use functions such as `malloc`, `calloc` and `free`:

- `void *malloc(size_t size)`: returns a pointer to `size` bytes of uninitialized storage, or `NULL` if the request cannot be satisfied.
- `void *calloc(size_t n, size_t size)`: returns a pointer to *contiguous* storage for `n` elements of `size` bytes each, or `NULL` if the request cannot be satisfied. *The memory is initialized to zero*.

The pointer returned by these functions is a `void *` and we need to cast it to the appropriate type. When a dynamically allocated pointer is no longer used, release the memory with `free`.

```c
/* allocate n integers */
size_t n = 100;
int *values = malloc(n * sizeof *values);
if (values == NULL) {
    /* handle error */
}

/* ... use values[0..n-1] ... */

/* alternatively, get zero-initialized memory */
int *zeros = calloc(n, sizeof *zeros);
if (zeros == NULL) {
    free(values);
    /* handle error */
}

/* ... use zeros and values ... */

free(zeros);
free(values);
```

Tip: ***Free order***

It's good practice to free dynamically allocated memory in the reverse order that you allocated it. This helps avoid bugs, especially if your allocations depend on each other. While many modern compilers can handle this automatically, freeing memory in the wrong order can sometimes lead to problems like segmentation faults.

3.3.5 Linked Lists

A **linked list** is a *dynamic* data structure that consists of a sequence of nodes, where each node contains a *value* and a *pointer* to the next node.
There are also **doubly linked lists**, where each node contains a value and pointers to both the next and the previous node, allowing traversal in both directions.

```c
typedef struct Node {
  int data;
  struct Node *next;
  // doubly linked list
  // struct Node *prev;
};
```

Here are some functions to create, print and free a linked list (to generalize, I made some of them iterative, some recursive):

```c
// Return a pointer to a new node with the given data
Node* create_node(int data) {
    Node *newNode = (Node*) malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Error: unable to allocate memory.\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Insert a new node at the beginning of the list
void insert_at_head(Node **head, Node *node_to_insert) {
    node_to_insert->next = *head;
    *head = node_to_insert;
}

// Print the list
void print_list(Node *head) {
    for (Node *curr = head; curr != NULL; curr = curr->next)
        printf("%d ", curr->data);
    printf("\n");
}

// Free the list
void free_list(Node *head) {
    if (head == NULL) return;
    free_list(head->next);
    free(head);
}
```

⚠ Warning: ***Memory management***

Always `free` the nodes you allocate when you no longer need them to avoid memory leaks.

Advantages and disadvantages

Pros: The size of a linked list can be changed dynamically at runtime. Inserting and deleting elements is efficient, especially in doubly linked lists.
Cons: Access to the *n*-th element is slow, as you need to traverse the list from the beginning. Each node also requires extra memory for its pointer(s), which increases memory usage.

3.3.6 Binary Trees

A **binary tree** is a hierarchical data structure in which each node contains a *data* and has at most two child nodes, typically referred to as the *left* and *right* child. Each node also stores pointers to these children, creating a branching structure that is fundamentally different from the linear nature of arrays or linked lists.

A particularly common type is the **Binary Search Tree** (BST), which imposes an ordering: for any given node, all values in its left subtree are less, and all values in its right one are greater. This property allows for efficient searching, insertion, and deletion operations, typically in $O(\log n)$ for balanced trees.

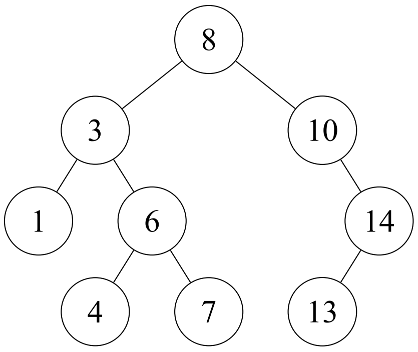

Figure 3.3: Binary search tree

A binary tree in C can be defined with a structure where each node stores data and pointers to its left and right children.
The entire tree is represented by a pointer to its root node. If it is empty, its pointer is set to `NULL`.

```c
struct Node {
    int data;
    struct Node *left;
    struct Node *right;
};
```

To process all the nodes in a binary tree, **recursive** traversal is widely used. The main orders are:
- **Inorder**: Visit the left subtree, then the root, then the right subtree.
- **Preorder**: Visit the root, then the left subtree, then the right subtree.
- **Postorder**: Visit the left subtree, then the right subtree, then the root.

Below is an example of how to add a new element to a binary search tree:

```c
struct Node* add(struct Node* p, int v) {
    if (p == NULL) { // If the tree is empty, create a new node
        p = (struct Node *) malloc(sizeof *p);
        p->data = v;
        p->left = p->right = NULL;
    }
    // If the value is less than the current node, go left
    else if (v < p->data) {
        p->left = add(p->left, v);
    }
    // If the value is greater than the current node, go right
    else if (v > p->data) {
        p->right = add(p->right, v);
    } else { /* The value is already present, do nothing */ }
    return p;
}
```

Advantages and disadvantages

Pros: Binary search trees (especially if balanced) allow for fast operations: searching, insertion, and deletion are all $O(\log n)$ on average. They also keep data automatically sorted, which makes ordered traversals straightforward.

Cons: If the tree becomes unbalanced (e.g., when many values are inserted in sorted order), performance degrades to $O(n)$. Extra care or specific algorithms are needed to keep it balanced.

3.3.7 Hash Tables

Suppose we have a collection of elements, each associated with a unique key, and we want to efficiently store and retrieve them by key—ideally in constant time. A powerful solution for this is the **hash table**.

A **hash table** is a data structure that organizes data based on a computed index, so that insertion, deletion, and lookup operations can be performed very efficiently (typically with average-case $O(1)$ time complexity). Instead of searching through a list, we use a *hash function* to compute an index (or ”address”) in an array (the ”table”) where the element should be stored.

The basic idea is as follows:

- A **hash function** transforms a key (which can be a string, number, etc.) into an integer, which is then mapped to one of the available array indices.
- Each position in the array is called a ***bucket*** or ***slot***.
- Ideally, each key maps to a unique index. However, since there are more possible keys than buckets, multiple keys may map to the same index, a situation known as a **collision**.
- To handle collisions, each bucket can store a list (or another structure) of all elements whose keys hash to the same index. This method is known as **chaining**.

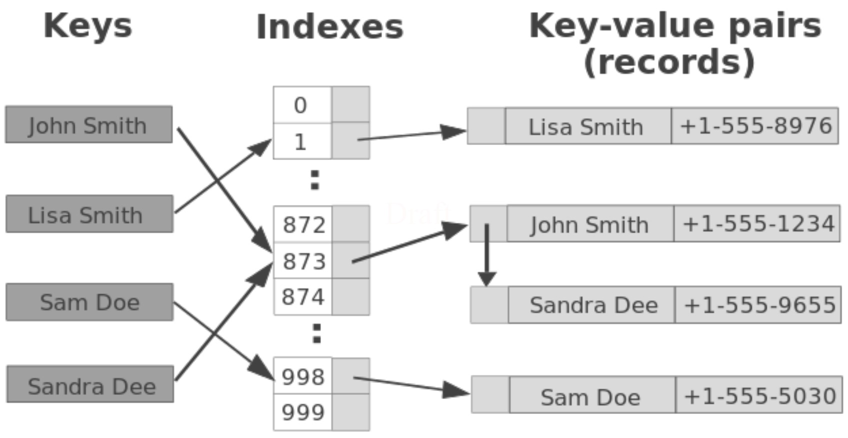

Figure 3.4: Hash table

Pros: Very fast operations (search, insert, delete) with average $O(1)$ time—performance stays the same even if you have 100 or 10 million elements. The fastest data structure for lookups.

Cons: No ordering: data is scattered pseudo-randomly, so there’s no way to print elements in order. Efficiency depends entirely on the quality of the hash function—a poor hash produces lots of collisions and slows everything down.

3.4 Input and Output

Direct Input and Output (I/O) Functions

In C, not all file operations need to be strictly sequential (from start to end). Sometimes, you need to read or write data at specific positions in a file, this is known as ***random access*** or **direct I/O**. Direct I/O functions allow programs to jump to (seek), read from, or write to arbitrary positions in a file. This is especially useful for binary files or large datasets structured in fixed-size records. Here are the most important functions:

- `fread(void *ptr, size_t size, size_t nobj, FILE *stream)`: Reads up to `nobj` objects of size `size` from the file referenced by `stream` into memory at `ptr`. Returns the number of objects actually read (can be less than `nobj` at EOF or on error).
- `fwrite(const void *ptr, size_t size, size_t nobj, FILE *stream)`: Writes `nobj` objects of size `size` from memory at `ptr` to the file referenced by `stream`. Returns the number of objects written.
- `fseek(FILE *stream, long offset, int origin)`: Moves the file position indicator to a given location. `origin` can be `SEEK_SET` (beginning), `SEEK_CUR` (current position), or `SEEK_END` (end of file).
- `ftell(FILE *stream)`: Returns the current position in the file.
- `rewind(FILE *stream)`: Convenient function to set the file position back to the beginning (same as `fseek(stream, 0L, SEEK_SET)`).
- `fgetpos(FILE *stream, fpos_t *ptr)`, `fsetpos(FILE *stream, const fpos_t *ptr)`: Save or restore a file position using a special type (`fpos_t`). Useful for marking positions in a file to come back to later.

Observation: ***Sequential vs. Direct Output***

The standard text file operations you may know (`fgets`, `fprintf`, etc.) always work in order, from start to end. In contrast, direct I/O lets you process files like arrays: you can jump immediately to any position.

Example usage:

```c
/* Write three integers to a binary file */
FILE *fp = fopen("data.bin", "wb");
int arr[3] = {1, 2, 3};
fwrite(arr, sizeof(int), 3, fp);
fclose(fp);

/* Read the second integer only */
fp = fopen("data.bin", "rb");
int x;
fseek(fp, sizeof(int), SEEK_SET); // Skip the first integer
fread(&x, sizeof(int), 1, fp);    // Read the second
printf("%d\n", x);                // Should print 2
fclose(fp);
```

Notes:

- For binary files, direct I/O is safe and efficient.

- For text files, random access may not behave as expected due to encoding/newlines—use only when strictly necessary.
- Always check the return value of these functions to detect errors or EOF.

4 Makefile

A Makefile is a file that contains a set of rules for building a project. It is used to automate the build process and to make it easier to maintain the project.
The syntax of a Makefile is:

```
target: dependencies
    command
```

where:

- `target` : is the name of the target to be built.
- `dependencies` : are the dependencies of the target.
- `command` : is the command to be executed to build the target.

It is possible to concatenate rules adding them as requirements for the target. The following one shows an example of a Makefile that builds a program made of two modules: `main.c` and `math_ops.c` .
It is a common practice to add a `clean` target to the Makefile to remove the compiled files and the executable.

Example: *Example of a Makefile*

```
calculator: main.o math_ops.o
    gcc -o calculator main.o math_ops.o

main.o: main.c math_ops.h
    gcc -c main.c

math_ops.o: math_ops.c math_ops.h
    gcc -c math_ops.c

clean:
    rm -f calculator main.o math_ops.o
```

It is possible to use the `make` command to build the project. The following command will build the project:

```
make

# Output:
gcc -c main.c -o main.o
gcc -c math_ops.c -o math_ops.o
gcc -o calculator main.o math_ops.o
```

Note the chaining of the commands to build the project.

Variables

It is possible to define variables in the Makefile to avoid repeating the same value multiple times. A variable can be defined and used in the Makefile with the following syntax:

```
VARIABLE = value
```

and used in the Makefile with the following syntax:

```
$(variable)
# or
${variable}
```

Example: ***Example of a Makefile with variables***

```
# Compiler
CC = gcc

# Compiler Flags
CFLAGS = -Wall -Wextra -O2

# Libraries Flags
LDFLAGS = -lm

# Target name
TARGET = calculator

# Objects
OBJS = main.o math_ops.o

# Build rules
$(TARGET): $(OBJS)
    $(CC) -o $(TARGET) $(LDFLAGS) $(OBJS)

\%.o: \%.c
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f $(TARGET) $(OBJS)
```

Automatic variables:

- `$@` : is the name of the target.
- `$<` : is the name of the first dependency.
- `$^`: is the names of all the dependencies.
- `$?` : is the names of dependencies newer than the target.

A complete Makefile:

```
# Compiler
CC = gcc

# Directories
```

```makefile
SRC_DIR = src
INC_DIR = include
OBJ_DIR = obj
BIN_DIR = bin

# Base flags
CFLAGS_COMMON = -Wall -Wextra -Iinclude
LDFLAGS = -lm

# Debug flags
CFLAGS_DEBUG = $(CFLAGS_COMMON) -g -O0 -DDEBUG
# Release flags
CFLAGS_RELEASE = $(CFLAGS_COMMON) -O3 -DNDEBUG

# Default to release
CFLAGS = $(CFLAGS_RELEASE)

# Files
TARGET = $(BIN_DIR)/nbody
SRCS = $(wildcard $(SRC_DIR)/*.c)
OBJS = $(SRCS:$(SRC_DIR)/%.c=$(OBJ_DIR)/%.o)
DEPS = $(OBJS:.o=.d)

# Default target
all: $(TARGET)

# Debug build
debug: CFLAGS = $(CFLAGS_DEBUG)
debug: clean $(TARGET)

# Release build (default)
release: CFLAGS = $(CFLAGS_RELEASE)
release: $(TARGET)

# Create directories
$(shell mkdir -p $(OBJ_DIR) $(BIN_DIR))

# Build executable
$(TARGET): $(OBJS)
    $(CC) -o $(TARGET) $(OBJS) $(LDFLAGS)

# Pattern rule
$(OBJ_DIR)/%.o: $(SRC_DIR)/%.c
    $(CC) $(CFLAGS) -MMD -MP -c $< -o $@

# Include dependencies
-include $(DEPS)

# Clean
clean:
    rm -rf $(OBJ_DIR) $(BIN_DIR)

# Phony targets
.PHONY: all debug release clean
```

5 Solving Ordinary Differential Equations (ODEs)

Ordinary Differential Equations (ODEs) are differential equations that involve only one independent variable. They are used to model many physical systems, such as the motion of a particle in a potential field, the flow of a fluid, or the behavior of a circuit.
Problems involving ordinary differential equations (ODEs) can always be reduced to the study of sets of first order differential equations. For example the second-order equation:

$$\frac{d^2y}{dx^2} + q(x)\frac{dy}{dx} = r(x) \qquad \Rightarrow \qquad \begin{cases} \dfrac{dy}{dx} = z(x) \\ \dfrac{dz}{dx} = r(x) - q(x)z(x) \end{cases}$$

We usually need initial conditions (as many as the order of the differential equation) to solve a differential equation.

Euler's method

Midpoint method

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1) \\ y_{n+1} &= y_n + k_2 + O(h^3) \end{aligned}$$

Runge-Kutta 4

RK3 is not stable, therefore we use RK4.

$$\begin{aligned} k_1 &= hf(x_n, y_n) \\ k_2 &= hf(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_1) \\ k_3 &= hf(x_n + \tfrac{1}{2}h, y_n + \tfrac{1}{2}k_2) \\ k_4 &= hf(x_n + h, y_n + k_3) \\ y_{n+1} &= y_n + \tfrac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) + O(h^5) \end{aligned}$$

This method requires to choose an appropriate step size h to ensure the accuracy of the solution.
There exist a variant of this method that consider a variable step size h_n for each step.
the general idea is to estimate the solution of the equation at order h^4 and h^5 and use their difference to control the error.
We can define the scaled error as:

$$\sqrt{\frac{1}{N}\sum_{i=0}^{N-1}\left(\frac{\Delta_i}{scale_i}\right)^2}$$

We accept the step if `err` < 1.

If `err` is much smaller than one the we can **increase** the integration step!
There are several methods to evaluate the integration step size given the error scaling as $O(h^5)$, e.g.:

$$h_0 = h_1 \left| \frac{\Delta_0}{\Delta_1} \right|^{0.2}$$

6 Particle-Mesh

We want to integrate equations for the evolutions of a colisionless fluid acting under the action of gravity. The equations governing the system are the Vlasov-Poisson:

$$\begin{cases} \frac{\partial f}{\partial t} + v \cdot \frac{\partial f}{\partial x} - \nabla V \cdot \frac{\partial f}{\partial v} = 0 \\ \nabla^2 V = 4\pi G(\rho - \bar{\rho}) \end{cases}$$

where the distribution function $f(x,v,t)$ depends on the position x, the velocity v and the time t. The density is given by:

$$\rho(x,t) = \int f(x,v,t) dv$$

Such equations (in 3D) lives in a 6D+1 dimensions, and their direct integration (using e.g. Boltzmann codes) is highly inefficient.
Thus, we can obtain the momentum equation by multiplying VP equations by v and integrating over velocities:

$$\frac{\partial \langle v \rangle}{\partial t} + \langle v \rangle \cdot \nabla \langle v \rangle = -\nabla V - \frac{1}{\rho} \nabla \cdot \mathbb{P}_J$$

where the pressure tensor $\mathbb{P}_J$ is given by:

$$\mathbb{P}_J \equiv \rho \sigma_{ij}^2 = \rho(\langle v_i v_j \rangle - \langle v_i \rangle \langle v_j \rangle)$$

1. We estimate the density field on a grid, as the sum of the mass of the particles in the grid cells, divided by the volume of the grid cells.
2. The density field is then transformed to the Fourier space.
3. The gravitational potential is then computed using the green function of the Laplacian.
4. The gravitational potential is then transformed back to the real space and forces are evaluated.
5. Forces (on the mesh) are then interpolated to the particles.
6. The particles velocities are then updated using the forces, with a chosen time step.
7. The particles positions are then updated using a leap-frog integrator.
8. Cycle is repeated until the desired time is reached.

To make the computation more stable, we need to normalize the units of the simulation. Commonly used units are (in `cgs` units):

- UnitVel = 10^5 km/s
- UnitMass = 1.989 $\times 10^{43} kg(10^{10} M_\odot)$
- UnitLength = 3.085678 $\times 10^{21}$ (1 kpc)

We need the following initial conditions to start the simulation:

- `N_points` :The number of particles.
- `N_grid` : FFT grid size.

- `BoxLength` : Length of the box (in kpc).
- `A_deltaPar` : Maximum density contrast allowed.
- ...

Other options are available, such as `H0` , `rho_crit` , etc.
We will evolve a sinuisodal density contrast:

$$\delta = A \sin(x \cdot 2\pi/L - \pi/2)$$

A will be small because the initial density contrast must be linear. Velocities will be set to zero. Note that using physical UoM means that we have a 3D distribution with only radial (1D) perturbations. Clearly, in reality this setup would not be stable against perturbations in the other two spherical coordinates.

Density computation

…

Force computation and interpolation

After computing the gravitational potential V on the computational grid, the next step is to derive the force field experienced by each particle. The force in one dimension is given by the negative gradient of the potential:

$$F = -\nabla V$$

On a discrete grid, we can approximate this derivative using central finite differences, resulting in the following expression for the force at grid point i:

$$F_i = -\frac{V_{i+1} - V_{i-1}}{2\Delta x}$$

where Δx is the grid spacing. Using a symmetric (central) difference avoids introducing directional biases in the force calculation and ensures second-order accuracy.
However, the physical particles in the simulation typically do not reside exactly at grid points. To obtain the force acting on each particle at its location x_p, we interpolate the force from the grid to the particle position using the same mass-assignment (interpolation) scheme used during the density assignment step. This is crucial to guarantee momentum conservation and minimize numerical artifacts such as ”self-forces” (forces that a particle erroneously exerts on itself):

$$F_p = \sum_{i=1}^{N_{\text{grid}}} W(x_p - x_i) F_i$$

Here, $W(x_p - x_i)$ is the interpolation (assignment) kernel, such as the Nearest-Grid-Point (NGP), Cloud-In-Cell (CIC), or higher-order schemes, evaluated at the distance between the particle and the grid point. Using identical interpolation for both density and force steps ensures consistency and preserves the overall symmetries of the simulation.

Leap Frog

At this point we have the acceleration acting on each particle and we can update velocities and positions:

$$v_i^{t+1/2\Delta t} = v_i^{t-1/2\Delta t} + (F_i/m_i) * \Delta t$$

$$x_i^{t+\Delta t} = x_i^t + v_i^{t+1/2\Delta t}\Delta t$$

7 Hydrodynamics

In astrophysics, **numerical hydrodynamics** plays a crucial role in modeling the formation of baryonic structures within dark matter potential wells during non-linear evolutionary phases.The complexity of realistic astrophysical flows (involving turbulence, shocks, magnetic fields, and radiative processes) demands numerical approaches.

The field encompasses two primary computational paradigms. **Eulerian methods** track the flow of gas and energy through fixed spatial grids, with derivatives computed at stationary points in space; this approach excels at capturing shocks and complex geometries. **Lagrangian methods**, by contrast, follow individual fluid elements as they move through space, with derivatives calculated in a co-moving coordinate system; these methods provide excellent mass conservation and natural resolution adaptivity in regions of high density.

The **Euler equations** form the fundamental framework for describing inviscid fluid motion. This system of coupled PDEs governs the conservation of mass, momentum, and energy:

$$\begin{aligned}
\frac{d\vec{v}}{dt} &= -\frac{\nabla P}{\rho} && \text{(momentum conservation)}\\
\frac{du}{dt} &= -\frac{P}{\rho}\nabla\cdot\vec{v} && \text{(energy conservation)}\\
\frac{d\rho}{dt} + \rho\nabla\cdot\vec{v} &= 0 && \text{(mass conservation)}\\
P &= (\gamma-1)\rho u && \text{(equation of state)}
\end{aligned}$$

where $\vec{v}$ is the fluid velocity field, P represents pressure, u denotes specific internal energy, ρ is the density, and γ is the adiabatic index (equal to $5/3$ for an ideal monoatomic gas).

The total (Lagrangian) derivative captures how quantities change following a fluid element:

$$\frac{d}{dt} = \frac{dx^i}{dt}\frac{\partial}{\partial x^i} + \frac{\partial}{\partial t} = \vec{v}\cdot\nabla + \frac{\partial}{\partial t}$$

This combines the local time rate of change with the advective transport due to fluid motion.

From the principle of mass conservation in Eulerian form, $\partial\rho/\partial t + \nabla\cdot(\rho\vec{v}) = 0$, we can derive the Lagrangian continuity equation. Applying the chain rule $d\rho/dt = \vec{v}\cdot\nabla\rho + \partial\rho/\partial t$ and expanding the divergence yields:

$$\frac{d\rho}{dt} = -\rho\nabla\cdot\vec{v}$$

This shows that density changes are directly related to the divergence of the velocity field: compression ($\nabla\cdot\vec{v} < 0$) increases density, while expansion ($\nabla\cdot\vec{v} > 0$) decreases it.

The energy equation derives from the first law of thermodynamics, $dU = dQ - PdV$. For adiabatic processes ($dQ = 0$), we substitute the specific volume relationship $dV \to d(1/\rho) = -d\rho/\rho^2$ and apply the continuity equation:

$$du = \frac{P}{\rho^2}d\rho \quad\longrightarrow\quad \frac{du}{dt} = \frac{P}{\rho^2}\frac{d\rho}{dt} = -\frac{P}{\rho}\nabla\cdot\vec{v}$$

This demonstrates how internal energy changes are coupled to the compression or expansion of the fluid through the pressure-volume work term.

7.1 Smoothed Particle Hydrodynamics (SPH)

Smoothed Particle Hydrodynamics (SPH) is a Lagrangian method that represents the fluid as a collection of discrete particles, each carrying hydrodynamic properties such as mass, velocity, and internal energy. The fundamental principle is that any quantity at a given point is not taken as a sharp value, but rather *smoothed* over the contributions of neighboring particles within a characteristic length scale h, called the **smoothing length**.

Particles evolve under the Euler equations using these smoothed quantities. At each timestep, positions and velocities are updated, the smoothed fields are recomputed from the new configuration, and the cycle repeats. This approach offers several advantages: automatic adaptivity (particles naturally concentrate in high-density regions), exact conservation of mass and momentum, and straightforward handling of free boundaries.

7.1.1 Smoothing and the kernel

The smoothed estimate of a function $f(\vec{r})$ is defined through a convolution with a **kernel** (or window function) W:

$$\tilde{f}_h(\vec{r}) = \int f(\vec{r}')\, W(\vec{r}-\vec{r}',h)\, d^3r'$$

The kernel must satisfy several properties. First, **normalization**: $\int W(\vec{r},h)\, d^3r = 1$, ensuring that in the limit $h \to 0$ the original function is recovered, i.e. $\lim_{h\to 0} \tilde{f}_h(\vec{r}) = f(\vec{r})$. Second, the kernel should be **radially symmetric** to conserve angular momentum. Third, it should have **compact support** (vanishing beyond some radius) to limit computational cost and avoid unphysical long-range interactions.

To convert the integral into a sum over particles, we multiply and divide by the density field:

$$\tilde{f}_h(\vec{r}) = \int \frac{f(\vec{r}')}{\rho(\vec{r}')} W(\vec{r}-\vec{r}',h)\, \rho(\vec{r}')\, d^3r'$$

Since each particle b occupies a volume element $d^3r' \approx m_b/\rho_b$, we obtain:

$$f(\vec{r}) \simeq \sum_b \frac{m_b}{\rho_b} f_b\, W(\vec{r}-\vec{r}_b,h)$$

Setting $f = \rho$ yields the fundamental SPH density estimator:

$$\boxed{\rho(\vec{r}) = \sum_b m_b W(\vec{r}-\vec{r}_b,h)}$$

This formula, central to SPH, automatically satisfies the continuity equation when particle masses are conserved. Note that the particle itself contributes to its own density estimate (typically as the dominant term, since W peaks at zero separation).

Since derivatives act on smooth functions, we can transfer the gradient operator onto the kernel:

$$\nabla f(\vec{r}) = \sum_b \frac{m_b}{\rho_b} f_b\, \nabla W(\vec{r}-\vec{r}_b,h)$$

This naive form, however, does not vanish for constant f. To enforce this property, we use a symmetrized formulation. Introducing an auxiliary field Φ, the identity

$$\frac{\partial A}{\partial x} = \frac{1}{\Phi}\left(\frac{\partial(\Phi A)}{\partial x} - A\frac{\partial \Phi}{\partial x}\right)$$

leads to the SPH gradient:

$$\left(\frac{\partial A}{\partial x}\right)_a = \frac{1}{\Phi_a}\sum_b m_b \frac{\Phi_b}{\rho_b}(A_b - A_a)\frac{\partial W_{ab}}{\partial x_a}$$

where $W_{ab} \equiv W(|\vec{r}_a - \vec{r}_b|, h)$. This form vanishes identically when A is constant. Common choices are $\Phi = 1$ and $\Phi = \rho$, giving respectively:

$$\left(\frac{\partial A}{\partial x}\right)_a = \sum_b \frac{m_b}{\rho_b}(A_b - A_a)\frac{\partial W_{ab}}{\partial x_a} \qquad \text{or} \qquad \left(\frac{\partial A}{\partial x}\right)_a = \frac{1}{\rho_a}\sum_b m_b(A_b - A_a)\frac{\partial W_{ab}}{\partial x_a}$$

For the divergence (needed in the continuity equation), analogous expressions hold:

$$\frac{d\rho_a}{dt} = \sum_b m_b \vec{v}_{ab} \cdot \nabla_a W_{ab} \qquad \text{or} \qquad \frac{d\rho_a}{dt} = \rho_a \sum_b \frac{m_b}{\rho_b} \vec{v}_{ab} \cdot \nabla_a W_{ab}$$

Differentiating the kernel twice amplifies numerical noise. A more stable approximation for the Laplacian is:

$$(\nabla^2 f)_a = 2\sum_b \frac{m_b}{\rho_b}(f_a - f_b)\frac{W_{ab}}{r_{ab}}$$

Despite this, higher-order derivatives remain problematic in SPH, making diffusion-type equations (thermal conduction, physical viscosity) challenging to handle accurately.
To understand the accuracy of the smoothing procedure, we expand $f(\vec{r}')$ in a Taylor series about $\vec{r}$:

$$f(\vec{r}') = \sum_{k=0}^{\infty} \frac{f^{(k)}(\vec{r})}{k!}(\vec{r}' - \vec{r})^k$$

Substituting into the convolution integral and exploiting the symmetry properties of the kernel, odd-order terms vanish, yielding:

$$\tilde{f}_h(\vec{r}) = f(\vec{r}) + Ch^2 + O(h^4)$$

where the coefficient C contains second-order derivatives of the function. This means that constant and linear functions are reproduced exactly, while the leading error is second-order in the smoothing length.
In practice, the **cubic spline kernel** is the most widely used choice, offering a good balance between accuracy and computational efficiency:

$$W(q) = \frac{\sigma}{h^d}\begin{cases} 1 - \frac{3}{2}q^2 + \frac{3}{4}q^3 & \text{if } 0 \le q \le 1 \\ \frac{1}{4}(2-q)^3 & \text{if } 1 < q \le 2 \\ 0 & \text{if } q > 2 \end{cases}$$

where $q = r_{ab}/h$ is the dimensionless separation, d is the number of spatial dimensions, and σ is a normalization constant: $\sigma = 2/3$ in 1D, $\sigma = 10/(7\pi)$ in 2D, and $\sigma = 1/\pi$ in 3D. The compact support (vanishing for $q > 2$) limits neighbor searches to within a radius of $2h$.
The **Gaussian kernel**, $W(r,h) \propto \exp(-r^2/h^2)$, is sometimes used for theoretical analysis due to its smoothness, but its non-compact support makes it computationally impractical. Higher-order kernels (quintic splines, Wendland functions) offer improved accuracy but at greater computational cost; they are becoming more common in modern implementations.

Tip: ***Self-contribution***

Each particle contributes to its own density estimate, typically as the dominant term since the kernel peaks at zero separation.

7.1.2 SPH equations of motion

With the discretization machinery in place, we can now write the SPH form of the Euler equations. The momentum equation becomes:

$$\frac{d\vec{v}_a}{dt} = -\sum_b m_b \left(\frac{P_a}{\rho_a^2} + \frac{P_b}{\rho_b^2} \right) \nabla_a W_{ab}$$

This symmetric form ensures exact conservation of linear and angular momentum. The energy equation takes the form:

$$\frac{du_a}{dt} = \frac{1}{2} \sum_b m_b \left(\frac{P_a}{\rho_a^2} + \frac{P_b}{\rho_b^2} \right) \vec{v}_{ab} \cdot \nabla_a W_{ab}$$

where the factor of $1/2$ arises from symmetrization. The system is closed by the equation of state $P = (\gamma - 1)\rho u$ and the density estimator derived earlier.

The standard SPH formulation does not explicitly conserve entropy. To address this, one can reformulate the equations using an **entropic function** $A(s)$ such that:

$$P = A(s)\rho^\gamma$$

where s is the specific entropy. From the equation of state for an ideal gas, the specific internal energy becomes:

$$u = \frac{A(s)}{\gamma - 1}\rho^{\gamma - 1}$$

Integrating the evolution equation for $A(s)$ (which changes only due to irreversible processes like shocks) is then equivalent to integrating the energy equation while explicitly tracking entropy generation. This formulation is particularly useful when accurate entropy conservation is important, such as in cosmological simulations.

7.1.3 Shock tube test

The **shock tube** (or Sod problem) is the canonical test case for hydrodynamical codes. Consider a one-dimensional tube divided by a membrane at its center, with two fluid regions having different initial conditions. The left region typically has higher pressure and density, while the right region has lower values of both. Both regions are initially at rest. Typical initial conditions (in SI units) are:

Quantity	Left zone	Right zone
Pressure P	10^5 Pa	10^4 Pa
Density ρ	1 kg/m^3	0.125 kg/m^3
Velocity v	0 m/s	0 m/s

At time $t = 0$, the membrane is instantaneously removed, creating a classical **Riemann problem**. The sudden pressure imbalance drives a complex wave pattern that propagates through the tube. Three distinct phenomena emerge:

Rarefaction fan. On the high-pressure side, a rarefaction wave propagates backward into the undisturbed fluid, creating a smooth transition region where density, pressure, and velocity change continuously. The rarefaction fan spreads out over time, producing the characteristic curved profile in the density distribution.

Contact discontinuity. This interface separates the two original fluids. While pressure and velocity are continuous across this boundary, density exhibits a jump discontinuity. The contact discontinuity moves at the local fluid velocity.

Shock wave. On the low-pressure side, a sharp discontinuity propagates forward into the undisturbed fluid. Across the shock, all fluid properties change abruptly; the shock compresses and heats the fluid it encounters.

The resulting flow pattern consists of four distinct regions separated by these three wave types. This problem serves as an excellent test case because it contains both smooth (rarefaction) and discontinuous (shock, contact) solutions, challenging the numerical scheme's ability to handle different types of flow features accurately.

⚠ **Warning**: ***Euler equations and shocks***

The Euler equations, being inviscid, do not contain the physical dissipation mechanisms that operate within real shock fronts. Without additional treatment, numerical solutions develop spurious oscillations near discontinuities.

7.1.4 Artificial viscosity

To capture shocks correctly in SPH, we introduce an **artificial viscosity** term Π_{ab} that adds dissipation only in regions of compression. The pressure terms in the momentum equation are modified as:

$$\left(\frac{P_a}{\rho_a^2}+\frac{P_b}{\rho_b^2}\right) \longrightarrow \left(\frac{P_a}{\rho_a^2}+\frac{P_b}{\rho_b^2}+\Pi_{ab}\right)$$

The artificial viscosity is activated only when particles approach each other:

$$\Pi_{ab} = \begin{cases} \dfrac{-\alpha \bar{c}_{s,ab}\mu_{ab}+\beta\mu_{ab}^2}{\bar{\rho}_{ab}} & \text{if } \vec{x}_{ab}\cdot\vec{v}_{ab}<0 \\ 0 & \text{otherwise} \end{cases}$$

where the auxiliary quantity μ_{ab} is defined as:

$$\mu_{ab} = \frac{\bar{h}_{ab}\vec{x}_{ab}\cdot\vec{v}_{ab}}{|\vec{x}_{ab}|^2+\varepsilon\bar{h}_{ab}^2}$$

Here, $\vec{x}_{ab}=\vec{r}_a-\vec{r}_b$ and $\vec{v}_{ab}=\vec{v}_a-\vec{v}_b$ are the relative position and velocity, $\bar{h}_{ab}$ and $\bar{\rho}_{ab}$ are arithmetic averages of the smoothing lengths and densities, $\bar{c}_{s,ab}$ is the average sound speed, and $\varepsilon \approx 0.01$ prevents singularities when particles are very close. Typical parameter values are $\alpha = 1$ and $\beta = 2$. The linear term (proportional to α) provides bulk viscosity that smears shocks over a few smoothing lengths, while the quadratic term (proportional to β) prevents particle interpenetration in high Mach number shocks.

A drawback of artificial viscosity is that it also damps legitimate shear flows. The **Balsara switch** reduces this spurious dissipation by weighting the viscosity with a factor that distinguishes between compression and shear:

$$f_a = \frac{|\langle\nabla\cdot\vec{v}\rangle_a|}{|\langle\nabla\cdot\vec{v}\rangle_a|+|\langle\nabla\times\vec{v}\rangle_a|+10^{-4}c_{s,a}/h_a}$$

This factor approaches unity in pure compression (where $\nabla\times\vec{v}=0$) and vanishes in pure shear (where $\nabla\cdot\vec{v}=0$). The modified viscosity becomes:

$$\Pi'_{ab} = \Pi_{ab}\cdot\frac{f_a+f_b}{2}$$

The small term in the denominator prevents division by zero in quiescent regions.

7.1.5 Time integration

The SPH equations of motion must be integrated in time using a stable and accurate scheme. The **leapfrog** (or kick-drift-kick) integrator is the standard choice, offering second-order accuracy, time-reversibility, and excellent energy conservation for Hamiltonian systems.

The timestep must satisfy the **Courant-Friedrichs-Lewy (CFL) condition**, which ensures that information does not propagate more than one resolution element per timestep. For SPH, this becomes:

$$\Delta t < \frac{h}{c_s}$$

where $c_s = \sqrt{\gamma P/\rho}$ is the local sound speed. In practice, a safety factor $K \approx 0.1$–0.15 is applied, and the global timestep is the minimum over all particles:

$$\Delta t = K \cdot \min_a \left(\frac{h_a}{c_{s,a} + 0.6(\alpha c_{s,a} + 2\max_b |\mu_{ab}|)} \right)$$

The additional terms in the denominator account for the artificial viscosity contribution to signal propagation.

The leapfrog integrator proceeds in three phases per timestep:

Kick (half-step): Update velocities and internal energies by half a timestep using the current accelerations and heating rates:

$$\vec{v}^{n+1/2} = \vec{v}^n + \frac{\Delta t}{2}\vec{a}^n$$
$$u^{n+1/2} = u^n + \frac{\Delta t}{2}\dot{u}^n$$

Drift (full step): Update positions by a full timestep using the half-step velocities:

$$\vec{r}^{n+1} = \vec{r}^n + \Delta t\, \vec{v}^{n+1/2}$$

After the drift, recompute densities, pressures, and accelerations at the new positions.

Kick (half-step): Complete the velocity and energy update using the new accelerations:

$$\vec{v}^{n+1} = \vec{v}^{n+1/2} + \frac{\Delta t}{2}\vec{a}^{n+1}$$
$$u^{n+1} = u^{n+1/2} + \frac{\Delta t}{2}\dot{u}^{n+1}$$

The complete SPH simulation loop can be summarized as follows:

1. **Initialization**: Set up initial conditions (positions, velocities, masses, internal energies); compute initial densities and pressures.
2. **Force computation**: Find neighbors within $2h$; compute accelerations and energy rates.
3. **Timestep**: Determine Δt from the Courant condition.
4. **First kick**: Advance velocities and energies by $\Delta t/2$.
5. **Drift**: Advance positions by Δt.
6. **Density update**: Recompute densities and pressures from new positions.
7. **Force update**: Recompute accelerations and energy rates.
8. **Second kick**: Complete velocity and energy update.
9. **Output**: Write snapshot if needed.
10. **Loop**: Return to step 3 until final time is reached.

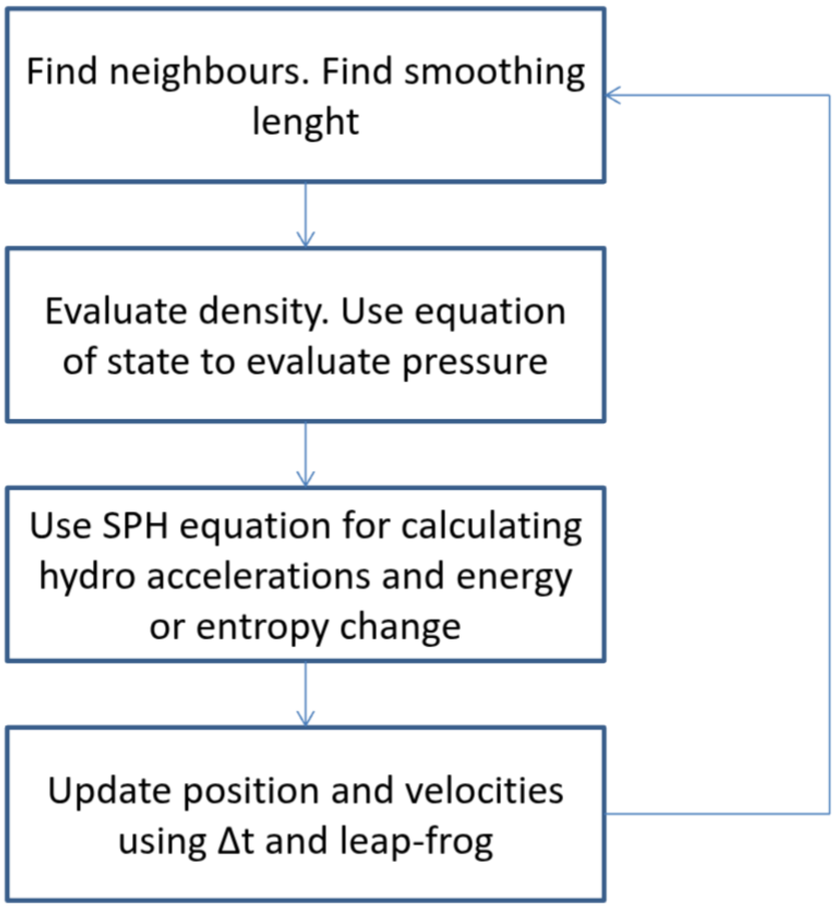
Find neighbours. Find smoothing lenght
Evaluate density. Use equation of state to evaluate pressure
Use SPH equation for calculating hydro accelerations and energy or entropy change
Update position and velocities using Δt and leap-frog

Figure 7.1: Schematic representation of the SPH simulation cycle.

Observation: ***Neighbor search***

Steps 2, 6, and 7 require finding all particles within distance $2h$ of each particle. For large N, a naive $O(N^2)$ search becomes prohibitive. Tree-based algorithms (such as octrees or kd-trees) reduce this to $O(N \log N)$, making SPH practical for millions of particles.

7.1.6 Variable smoothing length

In many astrophysical applications, the density contrast can span many orders of magnitude. Using a fixed smoothing length h would either under-resolve dense regions or waste computational resources in voids. The solution is to let h vary with the local density, typically maintaining a roughly constant number of neighbors N_{ngb}:

$$h_a \propto \left(\frac{m_a}{\rho_a}\right)^{1/d}$$

where d is the number of spatial dimensions. A common choice is $N_{\text{ngb}} \approx 32$–$64$ in 3D.

When h varies spatially, the SPH equations require correction terms (often called "grad-h" terms) to maintain energy conservation. These arise because the kernel normalization now depends on position. Modern SPH implementations include these corrections, though for pedagogical purposes a constant h simplifies the formulation considerably.

Tip: ***Choosing*** h

For a simple 1D simulation with N particles over a domain of length L, setting $h = 32L/N$ ensures each particle has approximately 64 neighbors within the kernel support radius $2h$.