

Class Design

For this project, I decided to keep everything in one class because the project requires only one main data structure: a 2D grid storing potential fields. I had an idea of splitting each point into a vector class but felt that increased the complexity without much benefit.

Key Variables:

int r and c: stores number of rows and columns

char **pointer: a dynamically allocated 2D array to mark each cell as goal, obstacle, or empty

double **px and **py: 2D arrays for storing the potential values in x and y directions

double potential: constant K for potential calculations

Keeping good OOP fundamentals, all of these are private (not accessed or modified by outside code) which ensures changes always go through controlled functions which the user must call which also triggers recompute to recalculate the potential forces on the entire grid.

The project requires using dynamically allocated arrays which involve pointer and memory allocation. Some justifications for using them are that we can't use static arrays as we need to dynamically size them based on user's input, gives constant time indexing which is helpful for updating the grid, and keeps operations simple. Adding to that last point, I used two arrays, one in the x and the other in the y direction which is similar to how we break down physics problems.

UML Diagram

Map
- r : int - c : int - pointer : char** - px : double** - py : double** - potential : double
+ create(n,m) : bool + addPoint(t,x,y) : bool + recompute() : void + move(x,y) : bool + clear() : bool + update(p) : bool

Function Design

create(int n, int m): allocates the grid. It first frees old memory to prevent leaks, then initializes all cells to empty. This way, all memory is owned by Map, so no external allocation is needed.

addPoint(char type, int x, int y): adds a goal or obstacle. Instead of leaving potentials stale, this function immediately calls **recompute()**. This ensures correctness at the cost of performance, which is acceptable because the problem size is not that large.

recompute(): central to the project. For every source cell (goal or obstacle), it iterates over the grid and updates potentials. The update rule uses the vector direction (dx, dy) and scales by distance. This design keeps the math simple and ensures that potentials always reflect the current set of sources.

move(int x, int y): returns the potential vector at a given location. Trying to access something out of bounds will return failure. This function is $O(1)$ since it simply looks up precomputed values.

clear(): resets the grid to empty. But instead of freeing memory, it reuses existing arrays to avoid reallocating.

update(double p): changes the potential constant K. Automatically calls **recompute()** after to maintain consistency.

Runtime Analysis

create(n,m): Initializes $n \times m$ cells

Runtime: $O(n \cdot m)$

addPoint(t,x,y): Setting the point is $O(1)$, but **recompute()** dominates

Runtime: $O((n \cdot m)^2)$ in the worst case (each cell updated by each source)

recompute(): For every source, visits every grid cell

Runtime: $O((n \cdot m)^2)$ in the worst case (all cells are sources)

move(x,y): Direct lookup

Runtime: $O(1)$

clear(): Resets all cells

Runtime: $O(n \cdot m)$

update(p): Recomputes potentials

Runtime: $O((n \cdot m)^2)$