ChatGPT

Understanding the Math Behind MoMa-LLM: Pipeline & Contributions

Problem Setting and Approach Overview

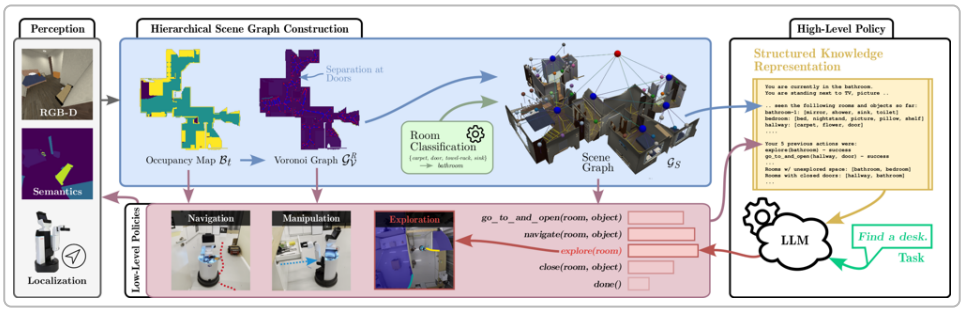

Figure: High-level MoMa-LLM pipeline. The robot builds a semantic map and scene graph from RGB-D sensor data. This scene graph is converted into structured text that grounds a Large Language Model (LLM) in the robot's environment. The LLM suggests high-level actions (plans), which the robot's low-level controllers execute, updating the scene graph as the environment is explored [1] .

At a high level, **MoMa-LLM** enables a robot to **find objects or perform tasks in an unknown environment** by combining **two things**: (1) a **dynamic 3D scene graph** (a structured map of rooms and objects the robot discovers), and (2) a **large language model** that uses this graph-based knowledge to plan actions. The robot starts with no knowledge of the environment and receives a **natural language task** (e.g. *"Find me a cup of tea"*). The system then proceeds as follows in a loop: it **perceives and maps** the environment, **updates the scene graph**, **converts the graph to a text description**, and **prompts the LLM** to suggest the next action. This **grounded LLM** approach ensures the language model's plans are **anchored to the real, physical world** of the robot (avoiding unrealistic or "hallucinated" commands) [2] . The LLM's high-level commands (like "go to the kitchen and open the fridge") are then executed by the robot's low-level navigation or manipulation skills. As the robot moves and interacts (opening doors, entering new rooms, etc.), it discovers more of the environment, **updates the scene graph**, and the cycle repeats until the task is completed (or a failure/timeout occurs).

Step-by-Step Pipeline (Simple Explanation)

Let's break down the **MoMa-LLM pipeline** in clear, step-by-step terms:

1. **Perception & Mapping:** The robot uses an RGB-D camera (color + depth) to observe its surroundings. From these sensor inputs, it **builds a 3D map** of the environment. This map is often stored as a voxel grid or an occupancy grid (think of a 3D grid marking which areas are free space versus occupied by walls or furniture) [3] . Essentially, the robot is figuring out the layout of rooms, walls, and free floor space as it moves around. The map is **dynamic**, meaning it updates continuously as new areas are explored or objects move [4] .

2. **Navigational Graph (Voronoi Graph):** From the occupancy map, the system computes a **navigation graph** to help with path planning. In MoMa-LLM, they use a **Generalized Voronoi**

Diagram (GVD), which is a set of points equidistant from obstacles [5]. In simple terms, imagine drawing lines in the free space such that each line is as far away from walls/obstacles as possible – these lines form a kind of **roadmap** through the environment. The resulting **Voronoi graph** has nodes and edges that represent clear paths the robot can take without hitting obstacles [6]. This graph is later used to estimate distances and navigate robustly (the robot will move along these safe "mid-path" nodes). The graph might be pruned (removing stray bits and keeping the largest connected component centered on where the robot can actually go) [7].

3. **Room Segmentation (Clustering):** The robot doesn't yet know where one room ends and another begins. MoMa-LLM addresses this by **splitting the navigational graph into distinct regions corresponding to rooms**. How? They detect **doorways** as natural separators. Whenever there's a doorway, it likely marks a boundary between two rooms. The system models door locations with a **mixture of Gaussians** – effectively a probability field highlighting areas near detected doors [8]. If an edge in the graph passes through a high-probability "door" zone (above a threshold), that edge is cut, splitting the graph into separate clusters [8]. Each cluster of the graph is treated as a separate **room**. This way, the map is partitioned into room-scale subgraphs without manual floorplans. (Think of it like: if two areas are only connected through a door, they are two different rooms).

4. **Scene Graph Construction:** Now the system builds a **scene graph** – a hierarchical graph representation of the environment. In this graph, **nodes represent entities** (e.g. rooms, objects) and **edges represent relationships** (e.g. "object X is in room Y" or "room A is adjacent to room B"). First, each **room cluster** from step 3 becomes a **Room node** in the scene graph. If two rooms are connected by a doorway, an edge is added between those Room nodes (so the robot knows which rooms are neighbors) [9]. Next, the robot adds **Object nodes**: whenever the robot's perception detects an object (say a **cup**, **fridge**, **bed**, etc.), that object is inserted into the scene graph. The important part is assigning each object to the correct room. MoMa-LLM does this carefully: for each detected object, it finds the nearest node on the navigational graph from which that object was observed (basically the robot's viewpoint for that object) and then assigns the object to whatever room cluster that node belongs to [10]. This method (formalized by an equation with a weighted distance metric) ensures an object is not accidentally assigned to the wrong room through a wall [11]. After this step, we have a **3D scene graph** containing nodes for all discovered rooms and objects, with "contains" edges linking each object to a room, and "adjacent" edges linking rooms via doors.

5. **Open-Vocabulary Room Labeling:** At this point, the scene graph knows "Room 1 contains objects {fridge, oven, table}" but it doesn't know the common-sense name of Room 1 (is it a *kitchen* or *dining room*?). MoMa-LLM treats this as an **open-vocabulary classification** problem. Instead of limiting to a fixed set of room types, they use a **language model (LLM) to name each room** based on its objects [12]. For example, if Room 1 has a fridge, stove, and cabinets, an LLM like GPT might infer "this is a kitchen." They prompt the LLM with the **list of object categories in each room**, and let it output the most appropriate room type [12]. This happens dynamically whenever new objects are discovered in a room. So the scene graph's room nodes get labeled with human-friendly names (kitchen, bedroom, etc.), and this works zero-shot for any odd combination of objects because the LLM has general world knowledge.

6. **Structured Knowledge Extraction (Prompt Preparation):** Now comes a **key novelty**: converting the raw scene graph data into a compact **textual description** for the language model to reason with. The paper emphasizes **structured, concise encoding** of the scene to ground the LLM [2]. Concretely, they generate a structured list in natural language that might read something like: *"Rooms discovered: 1) Kitchen – contains [fridge (closed), oven], 2) Bedroom –*

contains [bed, pillow, closed wardrobe]. Unexplored areas: a closed door leading from Kitchen to an unknown room, an unopened cabinet in Kitchen. Current room: Kitchen. Task: find a tea." (This is a conceptual example to illustrate the style.) The actual encoding follows specific rules:

7. **Room and Object Listing:** They list each known room and the objects in it, **grouping similar objects** and even noting object states (like opened/closed) inline [13]. This provides the LLM with a structured summary of "what is where."
8. **Distance or Adjacency Info:** They include distances in a qualitative way – for example, objects or rooms might be tagged as "near" or "far" relative to the robot or to each other. In fact, they bin the actual path distance from the robot to objects and use adjectives to indicate if something is close by or far away [14]. This helps the LLM understand what can be reached easily.
9. **Unexplored Frontiers:** Critically, because the environment is **partially observable** (unknown areas exist beyond explored space), the prompt also lists **unexplored regions**. Using the scene graph plus map, they identify frontier points – places at the edge of known space or behind closed doors/furniture – and describe them to the LLM [15]. For example, *"There is an unopened door in the hallway (leads to an unknown room)"* or *"There is an unexplored area behind the couch in the living room."* These hints let the LLM consider exploration actions when needed (exploration vs. exploitation trade-off).
10. **Recent Action History:** Instead of giving the LLM the entire dialogue or plan history (which would become too long), they append a **brief history of the last few actions** taken [16]. Importantly, they **align this history with the current state** – e.g. if the LLM previously said "open the door in the kitchen," and that door is now part of the scene graph as an open doorway to the living room, the history might be represented as `go_to_and_open(Kitchen, Door1) → success`. By providing a minimal history (just the last actions and whether they succeeded or failed), the LLM has context of what's been tried recently without being overwhelmed [17].

All this information is formatted as a **prompt to the LLM** (Figure 4 of the paper shows an example prompt structure [18]). The design ensures the LLM stays grounded: it **"knows" the map** of explored rooms, objects, unopened containers, closed doors, etc., and it has the **task goal** as well as the recent context. This structured prompting is crucial to guide the LLM's reasoning and **avoid it making up unseen objects or impossible actions** [2].

1. **High-Level Planning with LLM:** With the structured scene description as context, the **LLM (a pretrained large language model)** is asked to generate the **next high-level action** for the robot. Essentially, the LLM is acting as a planner. Because it has knowledge of the world (from its training) and now a grounded understanding of the robot's current world state (from the prompt), it can suggest sensible next steps. For example, if the task is *"find a tea"* and the LLM knows tea is often in a kitchen or pantry, it might output an action like `navigate(kitchen, cabinet)` (go to the kitchen and check a cabinet) or `go_to_and_open(kitchen, fridge)` if it knows tea might be in a fridge. The set of possible **high-level actions** is predefined (this is the "action space" the LLM can use) [19] [20]. MoMa-LLM's high-level actions include:
2. `navigate(room, object)`: move near a specified object in a given room (e.g. navigate to the table in the living room) [21].
3. `go_to_and_open(room, object)`: go to an object (often a **receptacle** like a door, drawer, or fridge) and open it [22].
4. `close(room, object)`: close an opened object (like shutting a door or drawer) [23].
5. `explore(room)`: explore an unexplored area of a given room (essentially telling the robot to roam within that room to uncover new space) [24].
6. `done()`: conclude that the task is finished (used when the target object has been found or the LLM decides to stop) [25].

These actions are like **subroutine names** that the LLM can output. The LLM's output will typically be something like one of these action names with arguments (room name, object name). For instance: `explore(kitchen)` or `go_to_and_open(living_room, cabinet)`. If the LLM's suggestion is not directly one of these forms, the system will interpret or if it's something unachievable, mark it as invalid.

1. **Low-Level Execution:** The suggested high-level action from the LLM is then executed by the robot's low-level controllers (which are outside the scope of the paper's novelty but crucial for a working system). For example, if the LLM said `navigate(kitchen, fridge)`, the robot invokes its **navigation module** to plan a path on the map (using A *search on the occupancy grid or via the Voronoi graph) to a point in the kitchen near the fridge* [21]. *If the LLM said* `go_to_and_open(kitchen, fridge)`, *the robot would navigate to the fridge and then use a manipulation routine to attempt to open the fridge door* [22]. *These low-level policies handle continuous sensorimotor control: moving the base, avoiding obstacles, controlling an arm to open doors, etc. While executing, the robot's sensors are active, so it might discover new rooms behind opened doors or find the target object inside a cabinet – this new info goes back into updating the map and scene graph** in real time [26]. In MoMa-LLM, the perception (object detection, mapping) is assumed mostly reliable (often ground-truth simulation data is used for focus on planning) [27].

2. **Feedback and Looping:** After the action, the system gives minimal feedback to the LLM for the next cycle. For example, it might tell the LLM that the last action **succeeded** (if, say, the door was opened and new area revealed), or **failed** (if the action couldn't be completed or was invalid) [28]. Notably, they do *not* try to convey every detail of what changed, just a simple success/failure flag, because the updated scene graph (which will be described in the next prompt) inherently contains the new state. If the LLM made an impossible request (like referencing an object that doesn't exist), that's an "invalid" action and they just inform it of failure and ask it to try a different approach [29]. With the scene graph now updated (e.g. a door node might be marked as open, a new room node added if a new area was found, etc.), a new prompt is generated (back to step 6) and given to the LLM for the next decision. This loop continues until the **target object is found and the LLM outputs** `done()`, or a maximum time/step limit is reached.

In summary, those are the **core steps**: mapping → graph → structured prompt → LLM plan → execute → update, in a continuous loop. Even if you "consider yourself a kid" in terms of robotics, the idea is intuitive: *the robot draws a map, writes down what it knows in simple language, asks a knowledgeable friend (LLM) what to do next, and then follows the advice to explore and find things.*

Key Contributions and Novel Components

Understanding which parts of this pipeline are **standard techniques** and which are the **novel contributions of MoMa-LLM** is important. Here are the highlights of MoMa-LLM's contributions:

- **Dynamic Scene Graph with Open-Vocabulary Rooms:** MoMa-LLM introduces a **scalable scene representation** centered on a **dynamic 3D scene graph**. Scene graphs have been used before, but here it's dynamic (updates as the robot explores) and features **open-vocabulary room clustering and classification** [30]. The system doesn't rely on a fixed set of room types or object categories; it leverages a pre-trained vision model and LLM to handle arbitrary object names and room names. The method of splitting the graph by detecting doorways (using Gaussian mixtures) and letting an LLM assign room labels freely is a novel way to avoid manual semantic assumptions [12]. This contribution lies in how the map is abstracted into a graph automatically and labeled in a generalizable way.

- **Structured Knowledge Grounding for LLMs:** Another key contribution is the **structured and compact encoding of scene knowledge to prompt the LLM** [31]. Prior works might either feed raw text descriptions or not use LLMs for such planning at all. MoMa-LLM shows how to **ground an LLM in a large, partially observed environment** by carefully crafting the prompt (rooms, objects, distances, unexplored areas, recent actions) [2]. The paper demonstrates that this structured grounding dramatically reduces LLM mistakes (like hallucinating objects or ignoring unseen areas) and improves plan quality [2] [32]. This is a novel contribution in the way it leverages the LLM's strengths (world knowledge and reasoning) while mitigating its weaknesses (making things up, lacking environmental awareness).

- **Object Search Task & Semantic Reasoning:** MoMa-LLM defines a new **"semantic interactive search" task** in large, realistic indoor environments [33]. Unlike simpler object search tasks, here the robot must not only navigate but also manipulate (open doors, drawers) and reason about where an object *should* be based on semantic context (e.g., coffee might be in the kitchen or maybe in a pantry). They created a benchmark for this task, extending prior interactive navigation tasks with many more objects and realistic object-room relationships [34] [35]. The contribution is partly the **task formulation** (search in unknown house with hidden objects) and a demonstration that combining an LLM with a scene graph is effective at this.

- **New Evaluation Metric (AUC-E) for Search Efficiency:** In evaluating the robot's performance, the authors introduce an **evaluation paradigm using full efficiency curves**, along with a metric called **AUC-E (Area Under the Curve – Efficiency)** [36]. Instead of just saying "did it succeed within 100 steps or not" (which requires picking an arbitrary cutoff), they plot the cumulative fraction of objects found over time and compute the area under this curve [37]. In simple terms, AUC-E captures **how quickly on average the agent finds the target** – higher AUC-E means it finds things faster. A perfect agent that finds the object immediately every time would have AUC-E = 1.0, whereas an agent that never finds the object scores 0 [38]. They typically compute this up to a certain time horizon (e.g. 5000 steps in simulation) and use it to compare methods [38]. This metric was proposed to fairly compare algorithms without biasing to a specific time limit, and it's a contribution of the paper to the evaluation methodology.

- **Zero-Shot and Generalizable:** While not a "math" contribution, it's worth noting that MoMa-LLM's whole approach is **zero-shot and open-vocabulary** – it doesn't train a new model for planning in that specific environment. By using a pre-trained LLM and general perception, it can in principle handle new homes, new objects, or tasks without additional training [39]. This is a contrast to many robotics approaches that rely on training a policy in simulation or require a fixed list of object types.

In summary, the **novel parts** of MoMa-LLM are *how it represents and uses knowledge*: the dynamic scene graph and the LLM grounding mechanism. The math or technical "heavy lifting" in those contributions involves graph algorithms (Voronoi diagrams, Gaussian-based clustering), prompt engineering for LLMs, and designing robust evaluation metrics. Next, we'll dive a bit deeper into some of the mathematical and technical details underpinning these steps to solidify understanding.

Technical Deep Dive: Math and Concepts Under the Hood

Now that we have a high-level grasp, let's explore the **key mathematical and technical concepts** that MoMa-LLM builds upon. We won't go through every equation, but we'll cover the important ones and the general mathematical tools involved:

1. Formal Task Framing (POMDP) and Bayes Reasoning (Conceptual)

MoMa-LLM's problem can be framed as a **Partially Observable Markov Decision Process (POMDP)** [40]. This means: - The robot has a **state** (which includes its position, the layout of the world, locations of objects, etc.), but it can't fully observe the state because the environment is initially unknown or partially hidden. - The robot takes **actions** (move, open, etc.) which change the state. - The robot receives **observations** (camera images, detections) that give partial information about the state. - There's a **goal** or reward for finding the target object.

Understanding this framing helps conceptually: the robot must **balance exploring** (to uncover more of the state) and **exploiting known info** (to move toward where it thinks the object is). While MoMa-LLM doesn't explicitly solve POMDP equations (no belief distributions are computed, unlike classical approaches), the LLM implicitly handles some of this reasoning by deciding when to explore new areas vs. when to go to a likely location of the object. The mention of "exploration-exploitation trade-offs" [15] hints at this classic concept: mathematically, it's about **uncertainty**. The system gives the LLM info about unexplored regions (uncertainty in state), and the LLM's job is to reason if it should reduce that uncertainty (explore) or if it already has enough evidence to go after the goal.

If you're comfortable with probability, you can imagine a Bayesian perspective: initially the "belief" about where the object could be is spread across all rooms. Seeing certain objects (like a **tea box** on a counter) might update the belief (tea is probably near the kitchen). Opening a cupboard gives new observations that might confirm or rule out a location. MoMa-LLM doesn't calculate these probabilities explicitly, but a *human-like reasoning* is outsourced to the LLM. The LLM's world knowledge acts like an internal prior: it "knows" common object-room relations (tea is in kitchen or pantry, not in the bathroom) [41]. So rather than formulas, here the math is hidden inside the LLM's trained parameters. Our job is just to feed it the right facts (via the prompt) so it can apply that knowledge.

2. Mapping and Geometric Computations

Occupancy Grid and Voxel Map: The mapping module uses a **voxel grid** V to integrate depth readings [4]. If you imagine the environment divided into small 3D cubes (voxels), each voxel can be "free" or "occupied." Linear algebra comes in to **transform each depth point into the global frame** (using the robot's pose matrix to go from camera coordinates to world coordinates). Storing points in a grid and projecting to a **bird's-eye view (BEV) occupancy map** involves simple math: for each voxel, record if there's an obstacle above a certain height (that's how you identify walls/furniture in 2D) [42]. The result is a 2D matrix (grid) where 1 = obstacle, 0 = free.

Distance Transform & Voronoi Diagram: To get the Voronoi graph, the system computes an **Euclidean Signed Distance Field (ESDF)** [43]. This is essentially a function $d(x)$ that gives the distance from any point x in the map to the nearest obstacle (distance is positive in free space, maybe negative inside obstacles, hence "signed"). Computing this efficiently can be done by algorithms that sweep over the grid (like dynamic programming) – conceptually, you can think of it as solving a wavefront propagation from all obstacle cells. The **Generalized Voronoi Diagram (GVD)** is then the set of all points that have equal distance to two or more nearest obstacles [5]. In practical terms, they likely threshold the gradient of the distance field: points that are locally maximal distances (ridges in the distance field) form thin lines equidistant from obstacles. This gives a **skeleton** of the free space.

Mathematically, one can define the GVD as: $$\text{GVD} = {\,x \mid \exists p, q \in \text{Obstacles}, p \neq q,\; \mathrm{dist}(x,p) = \mathrm{dist}(x,q) = \min_{r \in \text{Obstacles}} \mathrm{dist}(x, r)\,}.$$ In simpler terms, *x is on the Voronoi diagram if the two closest obstacles to x are exactly the same distance away.* The paper's Equation (1) presumably formalizes that [44]. The output is a set of points or grid cells

which they turn into graph nodes (connecting neighboring Voronoi points into edges yields the Voronoi graph G_V) [45]. They also **remove spurious parts** of this graph (e.g., bits near obstacles or isolated little components) and **sparsify** it to keep it computationally tractable [45]. This involves graph algorithms (finding connected components, filtering by size, etc.), but not an especially complicated formula – more an algorithmic procedure.

Room Separation via Gaussians: To split the graph into rooms, the novel math trick is using a **mixture of Gaussians** over door locations [8]. Let's unpack that: each detected **door** (from perception, they likely know which objects are doors) has a coordinate (x,y). They place a Gaussian bump (a 2D normal distribution) at each door's location. A Gaussian in 2D has the form $G(x,y) = \frac{1}{2\pi |\Sigma|^{1/2}} \exp{-[\begin{smallmatrix}x & y\end{smallmatrix}] \Sigma^{-1} [\begin{smallmatrix}x \ y\end{smallmatrix}]/2}$ – but you can imagine it as just a "hill" centered on the door. Summing these for all doors gives a smooth "door-likelihood field" across the map [46]. Then they take an empirically chosen threshold: any edge of the graph that passes through a region where this door-likelihood is above threshold is cut. In effect, if an edge goes **too close to a door**, they assume that edge actually goes *through* a doorway and thus should be broken to separate two rooms [46]. By doing this, the navigational graph is split into subgraphs $G_{V1}, G_{V2}, ..., G_{Vn}$, each hopefully corresponding to one physical room. This is an elegant use of probability for a geometric purpose (though thresholding the mixture of Gaussians is ultimately just a fancy way to say "cut near doors").

Assigning Objects to Rooms: Once rooms (graph clusters) are defined, each object needs to belong to one. They mention an Equation (3) where they compute the **shortest path** from the object's observation point to all candidate room-nodes and weight part of that distance to favor the node being very close to the object [10]. Essentially, they likely do: for each object, for each node in each room cluster, compute distance = (graph path from robot to node) + (Euclidean distance from node to object)^α + (Euclidean distance from robot to object) (or something along those lines), with an exponent α to emphasize the node-object distance [11]. By setting α > 1, they heavily penalize nodes that aren't near the object, which ensures the object doesn't "jump" rooms. The chosen node with minimum weighted distance gives the room assignment for that object [11]. The exact formula is less important than understanding it's a **nearest-neighbor search on the graph** with a bias toward spatial proximity. The result: each object gets a **"contained-in" edge** to a room node, anchoring it in the scene graph.

3. Graph Structure and Algorithms

The **scene graph** itself is a data structure, but understanding it conceptually requires graph theory basics: - **Nodes and Edges:** Rooms and objects are nodes; "in-room" or "adjacent-to" are edges. This is a simple bipartite structure (rooms connect to objects, and rooms connect to rooms via door adjacency). There's also the navigational graph nodes internally, but those are mostly used for distance calculations, not given to the LLM directly. - **Hierarchical Graph:** It's hierarchical because of different levels (rooms vs. objects). The graph G_S might be considered as two layers (room layer and object layer). This hierarchy is by design for clarity and grounding the actions (an action can specify `navigate(room, object)` which implicitly uses those layers). - **Updating the Graph:** Every time the robot finds a new object or a new room, adding a node and edges is straightforward graph operations. There isn't heavy math in updating – just insert/delete nodes and recompute distances or re-run room classification LLM for new rooms.

One thing to note: **room classification with LLM** effectively attaches a semantic label to each room node. Mathematically, you can think of the LLM as a function $f({o_i}) \to {\text{room name}}$, mapping a set of object categories to a likely room name. It's many-to-one (many combos of objects could yield "kitchen"). It's done in a prompt manner, not a deterministic function, but it's leveraging the LLM's learned probability distribution of words: for example, the probability of the word "kitchen" given the

prompt “The room contains a fridge, an oven, a sink, and a table” is hopefully very high. They used GPT-3.5 for this classification in experiments [47], which shows the system is *part* of the pipeline but not the core contribution (they built on an existing model’s capability).

4. Language Model Grounding and Prompt Design (Math of Prompting)

While prompt design may not look like “math” in the traditional sense (there are no equations), there is a **structure and logic** to it: - **Optimizing Token Usage:** The reason they compress and filter the scene description is because LLMs have a context length limit. If the robot sees 50 objects in 10 rooms, naively listing all could be huge. They compress by grouping identical objects (e.g., “3 chairs” instead of listing “chair” three times) [48], skip listing trivial open doors, and mark states succinctly (“opened fridge” instead of a sentence “the fridge is open”) [48]. All these are like *combinatorial optimization* on words – the goal is to **maximize information per token** given to the LLM. - **Ensuring Markov Property:** By restarting the LLM prompt fresh each cycle with the updated scene and short history, they enforce a Markov property (the next action depends only on current summarized state, not the entire history) [16]. This is conceptually related to how decision processes are modeled in math – by summarizing the necessary history into the state. Here, the “state” for the LLM is the text prompt which encodes everything important so far. - **Few-Shot or Function Call Format:** Although not explicitly detailed in what we saw, often such systems give the LLM a few-shot example or a specific output format (like a JSON or `<function>(args)` style). MoMa-LLM likely used a function-call format (the output looks like `navigate(kitchen, fridge)`) as their description of invalid arguments suggests they parse the LLM output [28]. Designing the prompt to yield a structured output is a bit of a prompt-engineering math/art, ensuring the probability of the LLM spitting exactly the right format is high. This might involve giving the LLM a system message or examples (e.g., “When you decide an action, respond exactly in the format: action(room, object).”).

The **grounding** aspect means the LLM’s generation is conditioned on factual info rather than freeform imagination. In probabilistic terms, if T is the task description and S is the structured scene info, the LLM is essentially sampling an action A from $P(A \mid S, T)$. By structuring S effectively, we narrow $P(\cdot)$ to mostly valid actions. The LLM’s internal knowledge supplies $P(A \mid S,T)$ with bias toward sensible actions (e.g., it “knows” that if the target is tea and there’s a kitchen unexplored, a good action is to explore the kitchen). It’s not solving an equation, but it is performing a kind of **inference** that we usually would approach with probabilistic planning algorithms – here it’s just done with language probabilities.

5. High-Level Planning and Execution Details

The planning loop can be thought of as an **algorithm**:

```
while not done:
    1. Observe environment (update map/graph)
    2. Encode state as prompt
    3. LLM -> propose action
    4. Execute action (with low-level controller)
    5. Feedback success/failure to prompt context
end
```

Steps 1–5 repeat. This is essentially a **closed-loop control system** where the LLM is part of the controller. In classical robotics, one might formulate a reward for finding the object and use a planner or

reinforcement learning. Here, the LLM's objective is implicit, and we rely on its understanding of "done() means found the object which is good."

The **high-level actions** themselves have some mathematical underpinnings: - `navigate(room, object)` uses **A** *pathfinding (which uses graph search algorithms on the grid or graph). A* is optimal path search given a heuristic. The Voronoi graph provides waypoints, and Euclidean distance can be the heuristic. So when the LLM says navigate, the system likely computes a path and distance. The LLM was told distances qualitatively (near/far), which likely came from computing shortest path distances in the graph [14]. - `go_to_and_open(room, object)` involves first navigation then a specific interaction. The success condition might require geometric checks (robot within X meters of object) [21], and if manipulation math is needed (inverse kinematics to open, etc.), that's handled by the low-level skill. - For exploration, choosing a frontier point might involve computing which frontier (unknown grid cell cluster) is within that room and maybe which one is closest – again a distance computation.

The math here is largely geometry and search algorithms. For example, success of navigate is defined as reaching within some radius δ of the target object [49] (so a distance threshold check). Opening a door might be considered successful if the door's state changes to open in the environment.

6. Evaluation Metrics and Efficiency Curves

Finally, the **AUC-E (Area Under Curve - Efficiency)** metric involves understanding **integrals and curves**: - They measure how quickly the agent finds the target object in each trial. Imagine a curve that starts at 0% (not found) and jumps to 100% (found) at the time the object is located. If you normalize time to a fixed horizon, each strategy yields a curve (earlier find means the curve rises sooner). - **Efficiency curve**: If we take the average of those step functions over many test trials, we get a smoother curve from 0 to some value by the horizon [37]. The area under this curve (integral) is higher if the agent tends to find objects early on average [38]. They mention normalizing such that an instantaneous find = 1.0 and never finding = 0.0 for AUC-E [38]. - In practice, they integrate up to a limit (5000 steps) [38]. If $f(t)$ is the fraction of tasks solved by time t, then $\text{AUC-E} = \frac{1}{T_{\max}} \int_0^{T_{\max}} f(t)\,dt$ (with $T_{\max}=5000$ steps for example). Multiplying by a constant, it's basically an average success rate over time. - This is akin to computing an **expected success time** metric. Understanding this requires basic calculus (area under curve) or knowing that this is the **time-averaged success fraction**.

They also use other metrics like **Success Rate (SR)** and **SPL (Success weighted by Path Length)** [37] [50] which are common in navigation tasks. SPL requires understanding a ratio of optimal path length vs. taken path length – but those are standard, and the main new one was AUC-E.

By covering the above points, we've touched on all the mathematical and technical foundations you'd need to read the MoMa-LLM paper comfortably: - Basics of POMDP (for context, though not solved explicitly here), - 3D mapping and coordinate transforms, - Distance fields and Voronoi (computational geometry), - Graph theory for scene graphs and connectivity, - Gaussian mixture models (for door clustering), - Some prompt engineering logic, - Path planning (A*), - and understanding new metrics (integration for AUC-E).

Most of the heavy math lifting in MoMa-LLM is in the **mapping and graph construction** part (Voronoi, Gaussian thresholds, path distance calculations) and in the **design of the prompt** (which is more logic than equation). The *planning* part is learned (inside the LLM) rather than coded as equations – but you

can appreciate that the LLM is leveraging a lot of implicit “probability” and “optimization” over actions via its training.

Finally, to **recap the contributions in math terms**: MoMa-LLM didn’t invent a new equation solving method, but it **innovated in integrating existing tools**: they applied geometric algorithms to get a clean symbolic world model, and used an LLM in a creative way to do decision-making. The “math” of the paper is about structuring information (graphs, distances, probabilities) such that a powerful language model can make use of it to produce valid plans. This showcases how understanding of linear algebra (for mapping), graph theory, and probability can come together in a robotics system that is greater than the sum of its parts.

---

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 [2403.08605] Language-Grounded Dynamic Scene Graphs for Interactive Object Search with Mobile Manipulation
https://ar5iv.labs.arxiv.org/html/2403.08605