JOHNS HOPKINS
UNIVERSITY

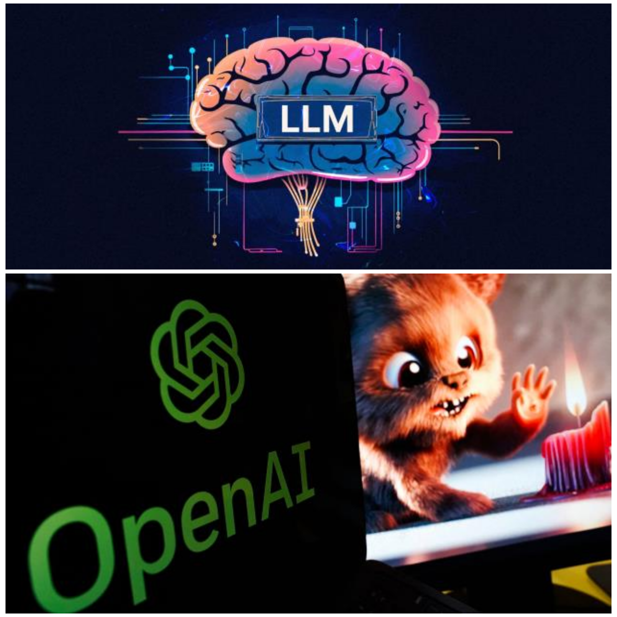
VoxPoser: Composable 3D Value Maps for Robotic Manipulation with Language Models

Johns Hopkins University - Fall 2024

CS 601.771 Advances in Self-supervised Models

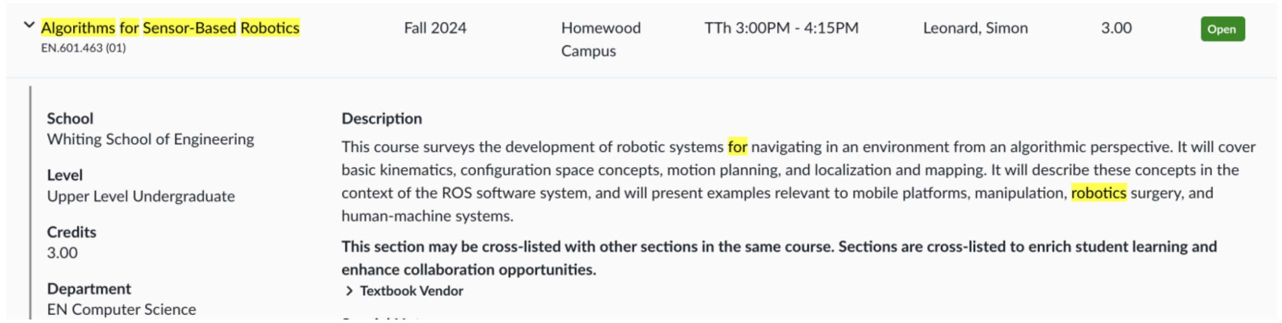
Marvin Gao

Leverage knowledge from LLM/VLM to make the physical world working



Demonstrate the reasoning ability based on the massive human knowledge

Why?



Algorithms for Sensor-Based Robotics
EN.601.463 (01)

Algorithms for Sensor-Based Robotics EN.601.463 (01)	Fall 2024	Homewood Campus	TTh 3:00PM - 4:15PM	Leonard, Simon	3.00	Open

School
Whiting School of Engineering

Level
Upper Level Undergraduate

Credits
3.00

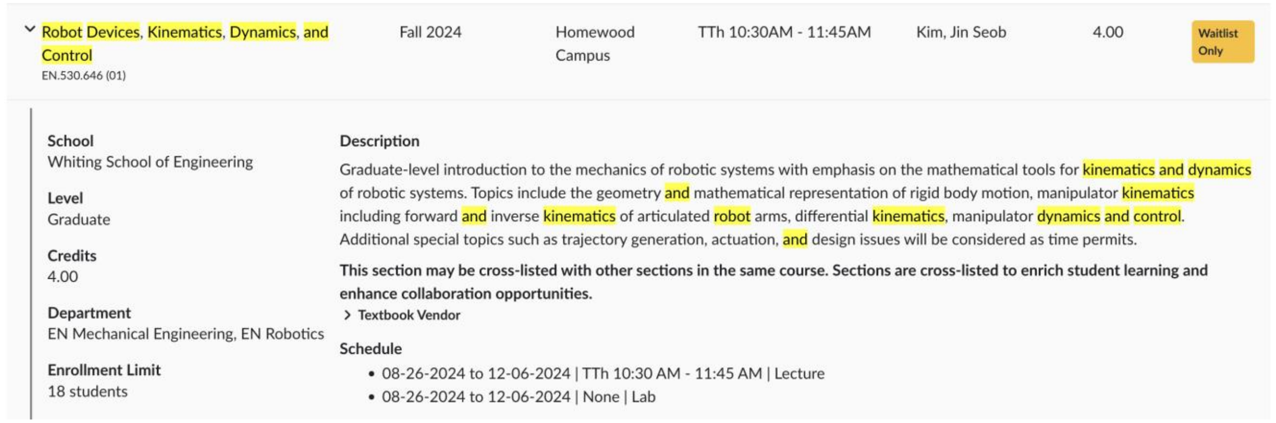
Department
EN Computer Science

Description

This course surveys the development of robotic systems for navigating in an environment from an algorithmic perspective. It will cover basic kinematics, configuration space concepts, motion planning, and localization and mapping. It will describe these concepts in the context of the ROS software system, and will present examples relevant to mobile platforms, manipulation, robotics surgery, and human-machine systems.

This section may be cross-listed with other sections in the same course. Sections are cross-listed to enrich student learning and enhance collaboration opportunities.

> Textbook Vendor



Robot Devices, Kinematics, Dynamics, and Control EN.530.646 (01)	Fall 2024	Homewood Campus	TTh 10:30AM - 11:45AM	Kim, Jin Seob	4.00	Waitlist Only

School
Whiting School of Engineering

Level
Graduate

Credits
4.00

Department
EN Mechanical Engineering, EN Robotics

Enrollment Limit
18 students

Description

Graduate-level introduction to the mechanics of robotic systems with emphasis on the mathematical tools for kinematics and dynamics of robotic systems. Topics include the geometry and mathematical representation of rigid body motion, manipulator kinematics including forward and inverse kinematics of articulated robot arms, differential kinematics, manipulator dynamics and control. Additional special topics such as trajectory generation, actuation, and design issues will be considered as time permits.

This section may be cross-listed with other sections in the same course. Sections are cross-listed to enrich student learning and enhance collaboration opportunities.

> Textbook Vendor

Schedule

- 08-26-2024 to 12-06-2024 | TTh 10:30 AM - 11:45 AM | Lecture
- 08-26-2024 to 12-06-2024 | None | Lab

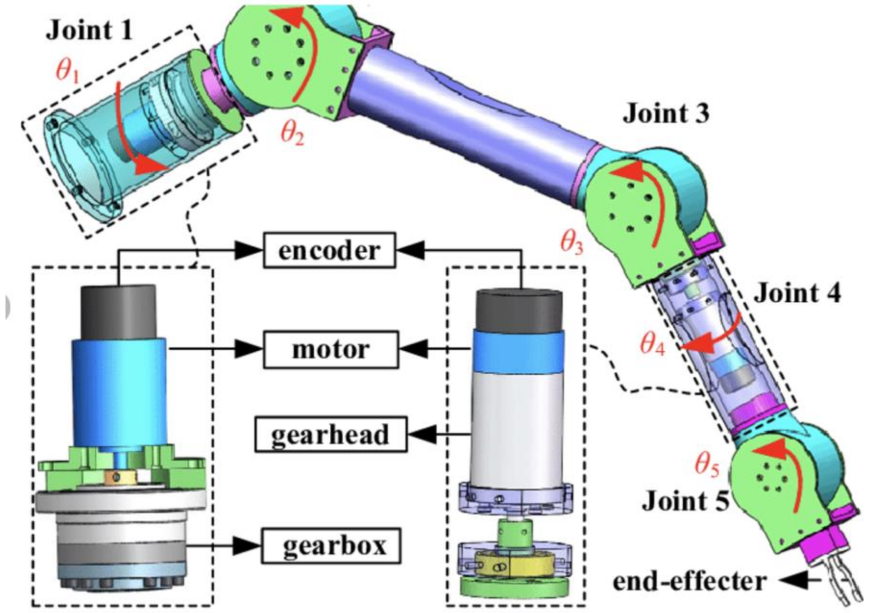
The classical algorithms work for many applications.

However, if the tasks become complex:

1. Complex Pipeline: Given a natural language description task, the robot needs to parse the command the execute the tasks;
2. Complex Tasks: Computer Vision, Perception, Navigation, Motion Planning, Language Understanding …
3. open-set of instructions and open-set of objects

Many Challenges

- LLMs and VLMs are trained by texts and images.
- For Robotics:
  - **Representation Space (Configuration Space)**
    - The positions of robot components are usually represented by a configuration space:
  - **Multi-Mode Observation**
    - Include the data of positions,
    - world frames, forces,
    - etc
  - **Action Space**
    - Action Space is variable for different tasks.



Previous Approaches

- Pre-defined motion primitives:
    - liftArm()
    - moveToLeft()
    - closeGripper()
    - …

- Use LLMs / VLMs to compose new programming.
- Limitation:
    - Heavily depends on human-defined rules
    - How many primitives do we need to define as the problems become challenging?
    - cannot be adapted to different environments
    - cannot sophistically control the robot

Question:

How do we use the LLMs/VLMs knowledge to complete the robotics tasks, such as "turn on the lamp", "Open the top drawer. Please also watch out for that vase! ", given an ***open set of instructions*** and *an* ***open set of objects*** **without** human-defined primitives?

Keywords: Embodied AI, physical intelligence

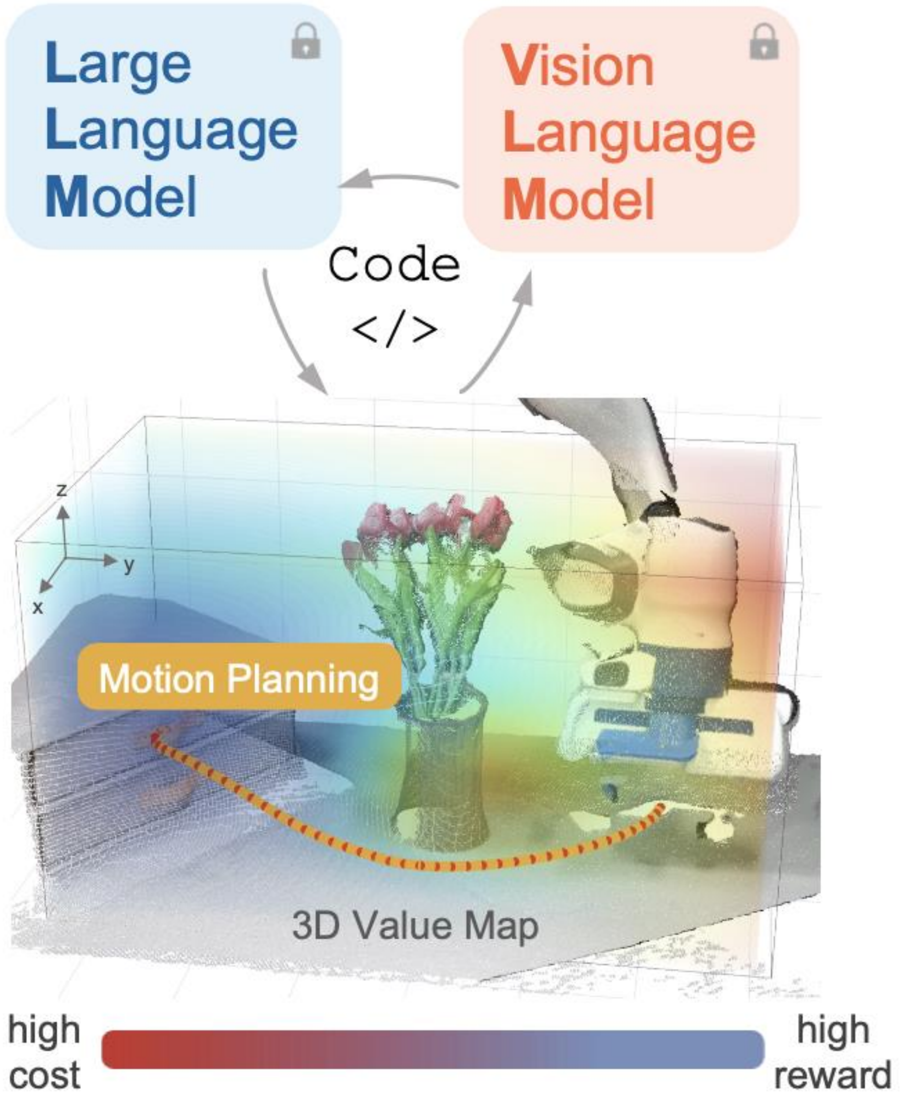
Some Terms

Voxel = Volume x Element

Affordance map: The volume elements that the robot can take action.

Constraint Map: The volume elements that the robot cannot take action.

End-Effort: The part of a robotic system that directly interacts with the environment.

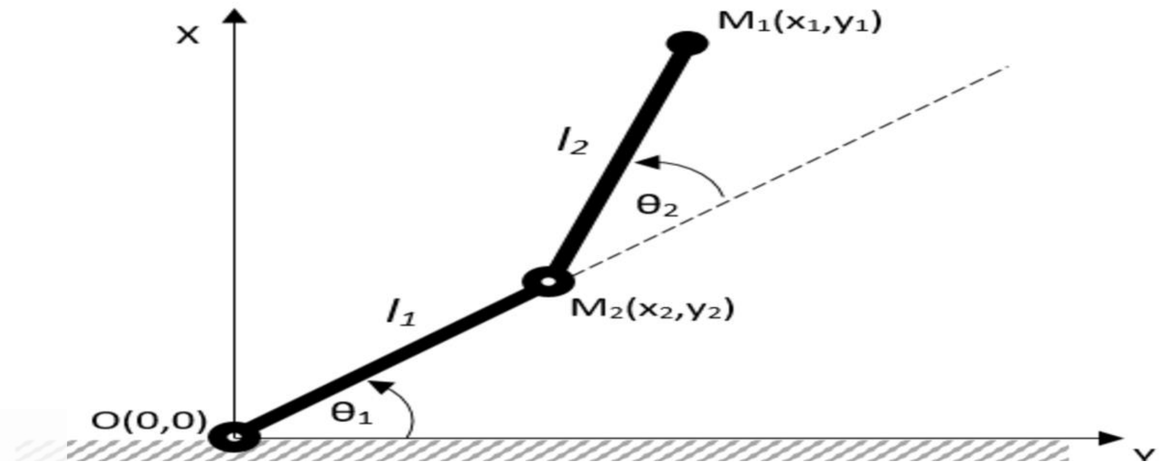


VoxPoser

Robot Kinematics

- Question? How to move the end effort from position A to position B.
- The robot's end effort **x = (xx, xy, xz)** is determined by joint angles, **q** = [q1, q2, .. qN].
- Therefore, **x = f(q)**.
- **dx / dt = df/dq * dq / dt**
- **df / dq** is a matrix, and the elements are the partial values f each joint. Mathematically, this matrix is named the "**Jacobian Matrix**".
- For a simpler example with two joints:
- - We can get the Jacobian Matrix by the combination info of the joints.
- - Then, **dx** / dt is the velocity because the **x** is a vector.
- - Therefore, $\frac{d\,\vec{q}}{dt} = \left(\frac{df}{d\,\vec{q}}\right)^{-1} * \frac{d\,\vec{x}}{dt}$

$$\mathbf{J}_\mathbf{f} = \left[\frac{\partial \mathbf{f}}{\partial x_1} \quad \cdots \right.$$



1. Forward Kinematics

Calculate the position of the end-effector $[x, y]$ based on the current joint angles θ_1, θ_2:

$$x = L_1\cos(\theta_1) + L_2\cos(\theta_1 + \theta_2)$$

$$y = L_1\sin(\theta_1) + L_2\sin(\theta_1 + \theta_2)$$

2. Jacobian Derivation

The Jacobian $\mathbf{J}$ relates joint velocities to end-effector velocities:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial x}{\partial \theta_1} & \frac{\partial x}{\partial \theta_2} \\ \frac{\partial y}{\partial \theta_1} & \frac{\partial y}{\partial \theta_2} \end{bmatrix}$$

For the 2-joint robot, substituting the partial derivatives:

$$\mathbf{J} = \begin{bmatrix} -L_1\sin(\theta_1) - L_2\sin(\theta_1+\theta_2) & -L_2\sin(\theta_1+\theta_2) \\ L_1\cos(\theta_1) + L_2\cos(\theta_1+\theta_2) & L_2\cos(\theta_1+\theta_2) \end{bmatrix}$$

Problem Setting

- Given the language, l_i , such as *"Open the top drawer. Please also watch out for that vase! ",* as a task description.

- We need a trajectory τ_i^r for robot **r** and instruction l_i.

  Each waypoint: [x, y, z, vx, vy, vz, gripper_action[0/1]]

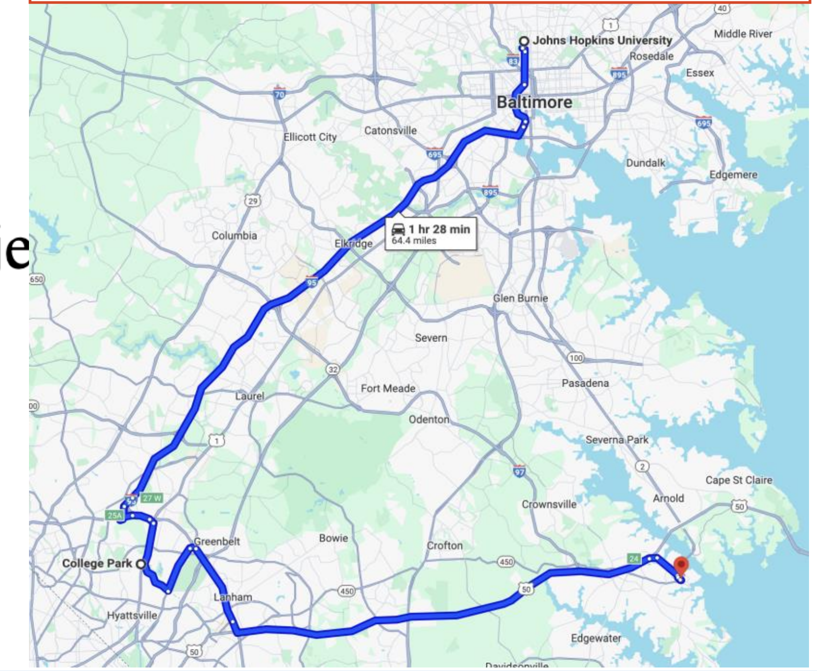
- We formulate an optimization problem defined as follows:

$$\min_{\tau_i^{\mathbf{r}}} \left\{ \boxed{\mathcal{F}_{task}} (\boxed{\mathbf{T}_i}, \ell_i) + \boxed{\mathcal{F}_{control}} (\tau_i^{\mathbf{r}}) \right\} \quad \text{subje}$$

T_i: environment states during the task execution

F_{task}: the cost of the sequential states for task i
$F_{control}$: the cost the trajectory

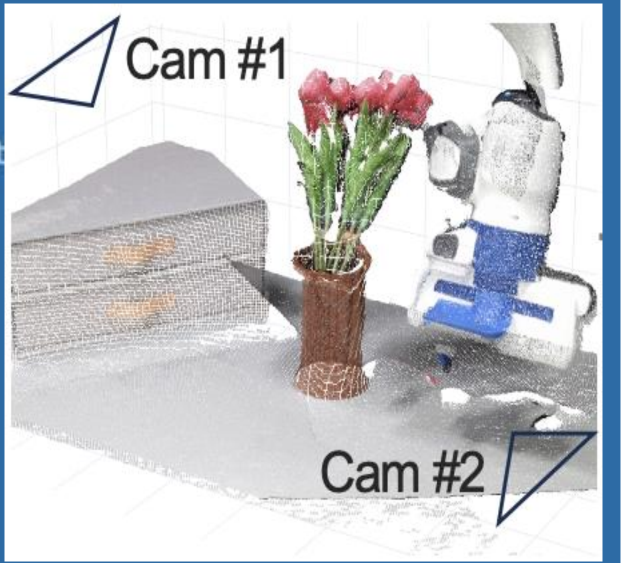
Ttask: How will the state fit to the task
Tcontrol: The time or energy cost when executing the actions.



A complete example – Step1

- Task: *put the sweeter fruit in the tray that contains the bread.*
- Step1: using **planner prompt** let LLM learn and decompose the task into several sub-tasks.

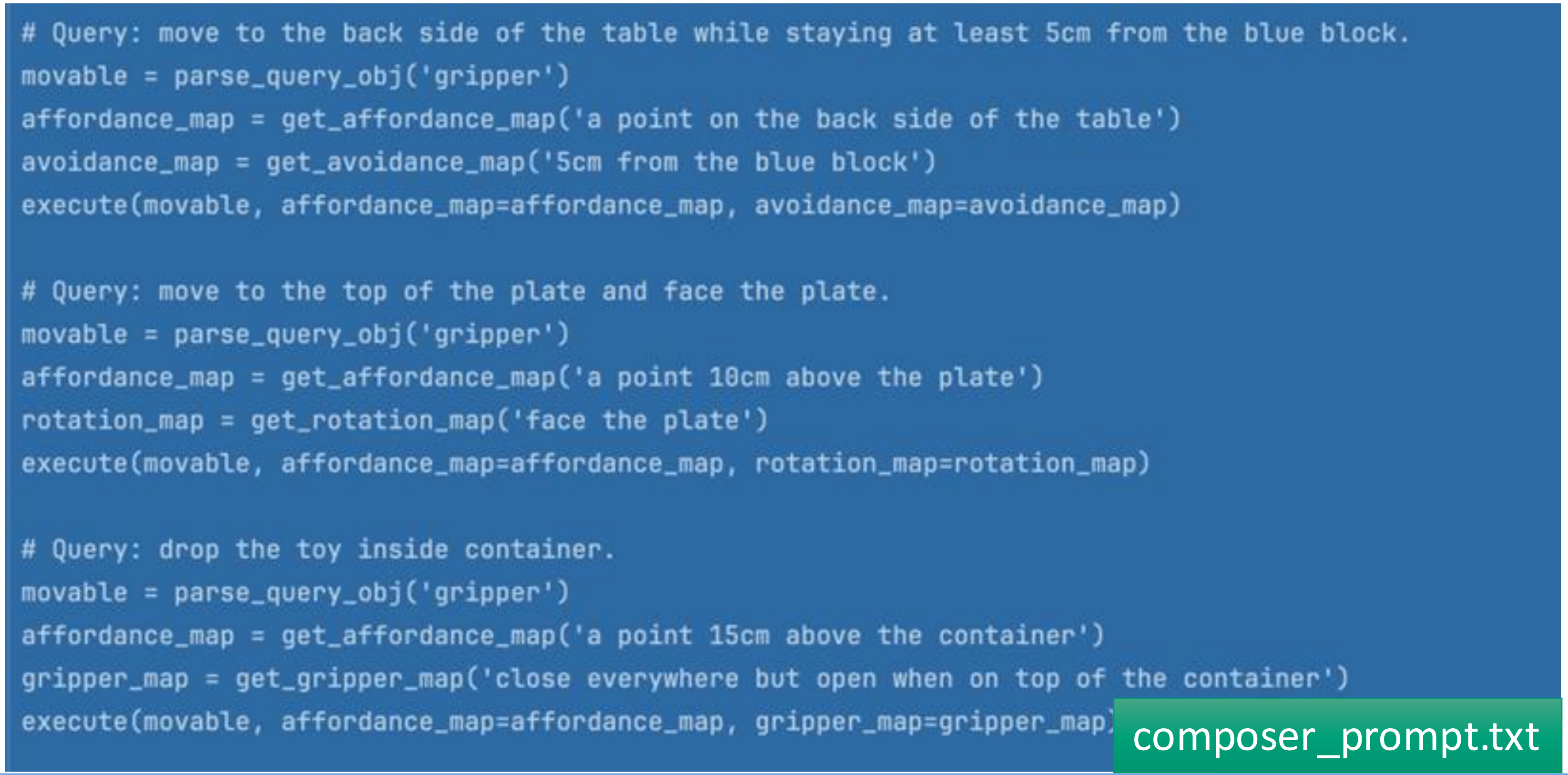
```
objects = ['blue block', 'yellow block', 'mug']
# Query: place the blue block on the yellow block, and avoid the mug at all time.
composer("grasp the blue block while keeping at least 15cm away from the mug")
composer("back to default pose")
composer("move to 5cm on top of the yellow block while keeping at least 15cm away from t
composer("open gripper")
# done

objects = ['airpods', 'drawer']
# Query: Open the drawer slowly.
composer("grasp the drawer handle, at 0.5x speed")
composer("move away from the drawer handle by 25cm, at 0.5x speed")
composer("open gripper, at 0.5x speed")
# done
```



planner prompt.txt

A complete example – Step2

- For each decomposed sub task, the composer_prompt will let LLM with context learning and get corresponding affordance and avoidance maps.

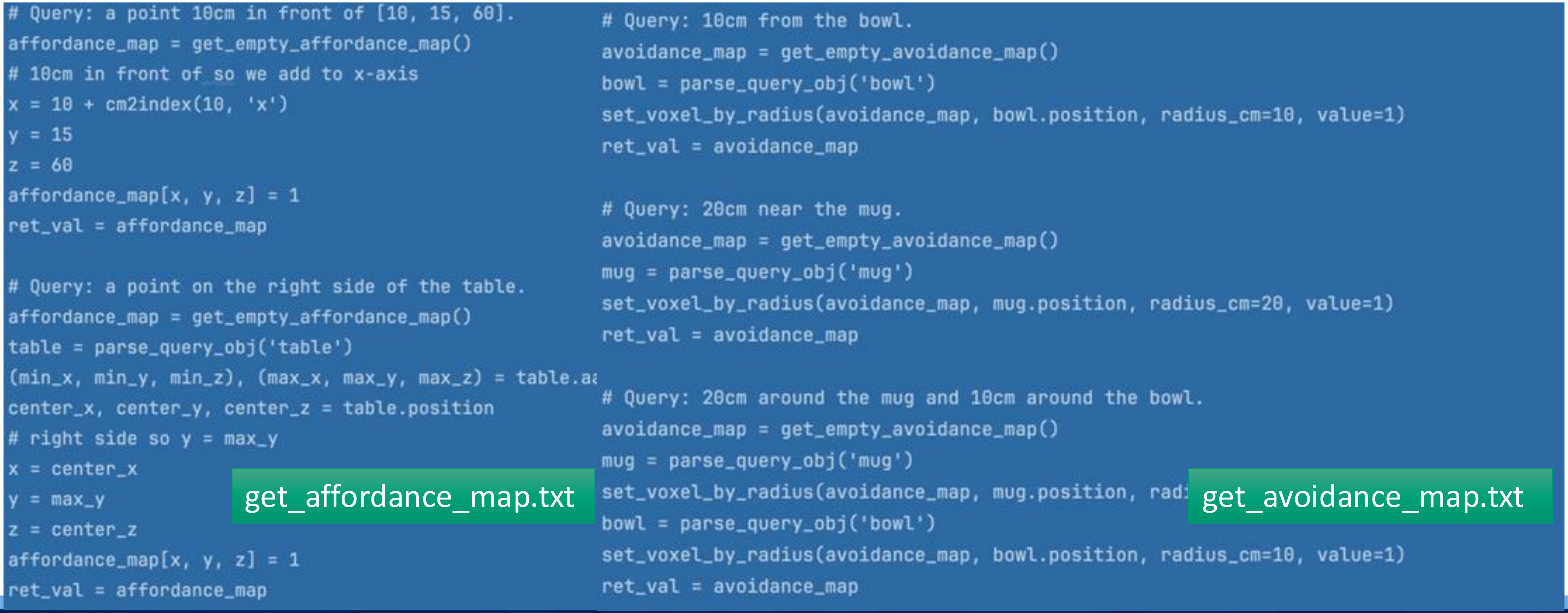


```
# Query: move to the back side of the table while staying at least 5cm from the blue block.
movable = parse_query_obj('gripper')
affordance_map = get_affordance_map('a point on the back side of the table')
avoidance_map = get_avoidance_map('5cm from the blue block')
execute(movable, affordance_map=affordance_map, avoidance_map=avoidance_map)

# Query: move to the top of the plate and face the plate.
movable = parse_query_obj('gripper')
affordance_map = get_affordance_map('a point 10cm above the plate')
rotation_map = get_rotation_map('face the plate')
execute(movable, affordance_map=affordance_map, rotation_map=rotation_map)

# Query: drop the toy inside container.
movable = parse_query_obj('gripper')
affordance_map = get_affordance_map('a point 15cm above the container')
gripper_map = get_gripper_map('close everywhere but open when on top of the container')
execute(movable, affordance_map=affordance_map, gripper_map=gripper_map)
```

composer_prompt.txt

A complete example – Step3

- Generate getting affordance and avoidance maps by LLM context learning.

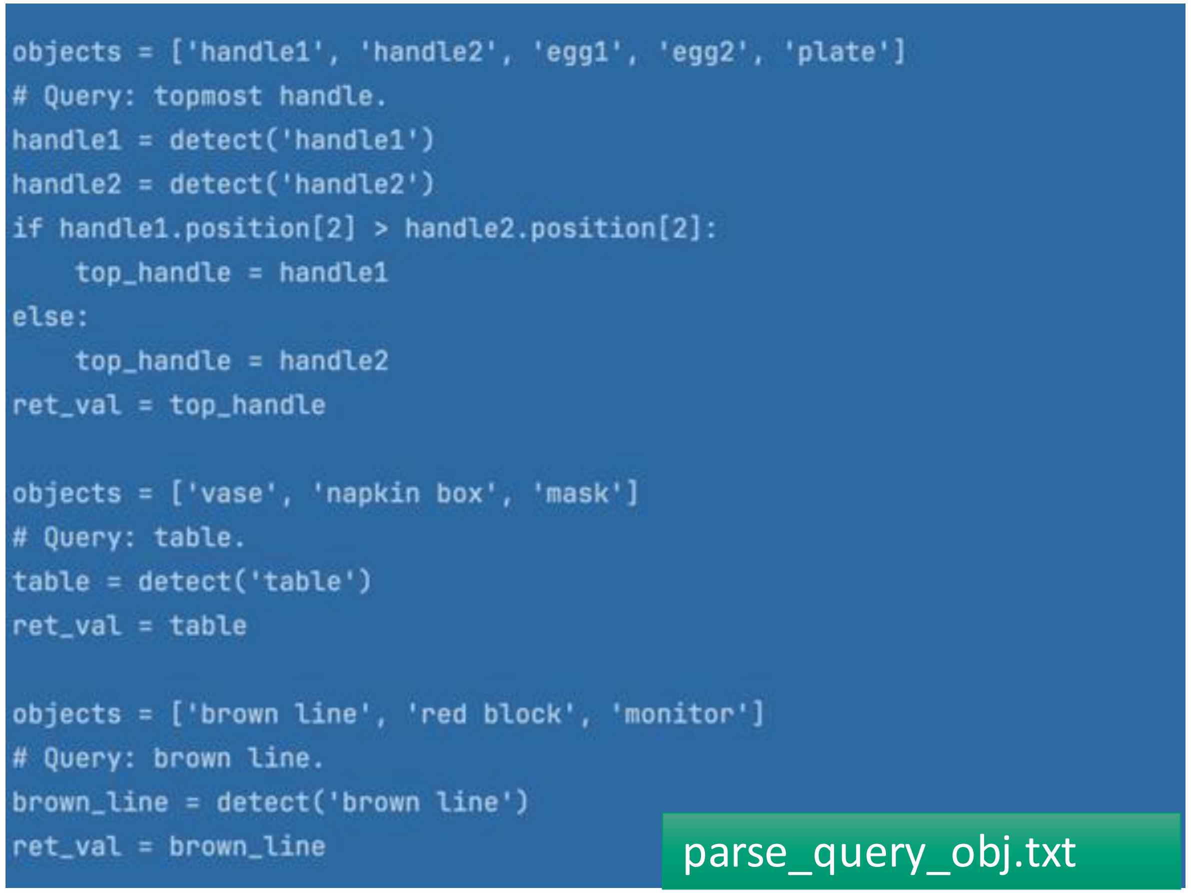


```
# Query: a point 10cm in front of [10, 15, 60].
affordance_map = get_empty_affordance_map()
# 10cm in front of so we add to x-axis
x = 10 + cm2index(10, 'x')
y = 15
z = 60
affordance_map[x, y, z] = 1
ret_val = affordance_map

# Query: a point on the right side of the table.
affordance_map = get_empty_affordance_map()
table = parse_query_obj('table')
(min_x, min_y, min_z), (max_x, max_y, max_z) = table.aa
center_x, center_y, center_z = table.position
# right side so y = max_y
x = center_x
y = max_y
z = center_z
affordance_map[x, y, z] = 1
ret_val = affordance_map
```

```
# Query: 10cm from the bowl.
avoidance_map = get_empty_avoidance_map()
bowl = parse_query_obj('bowl')
set_voxel_by_radius(avoidance_map, bowl.position, radius_cm=10, value=1)
ret_val = avoidance_map

# Query: 20cm near the mug.
avoidance_map = get_empty_avoidance_map()
mug = parse_query_obj('mug')
set_voxel_by_radius(avoidance_map, mug.position, radius_cm=20, value=1)
ret_val = avoidance_map

# Query: 20cm around the mug and 10cm around the bowl.
avoidance_map = get_empty_avoidance_map()
mug = parse_query_obj('mug')
set_voxel_by_radius(avoidance_map, mug.position, rad
bowl = parse_query_obj('bowl')
set_voxel_by_radius(avoidance_map, bowl.position, radius_cm=10, value=1)
ret_val = avoidance_map
```

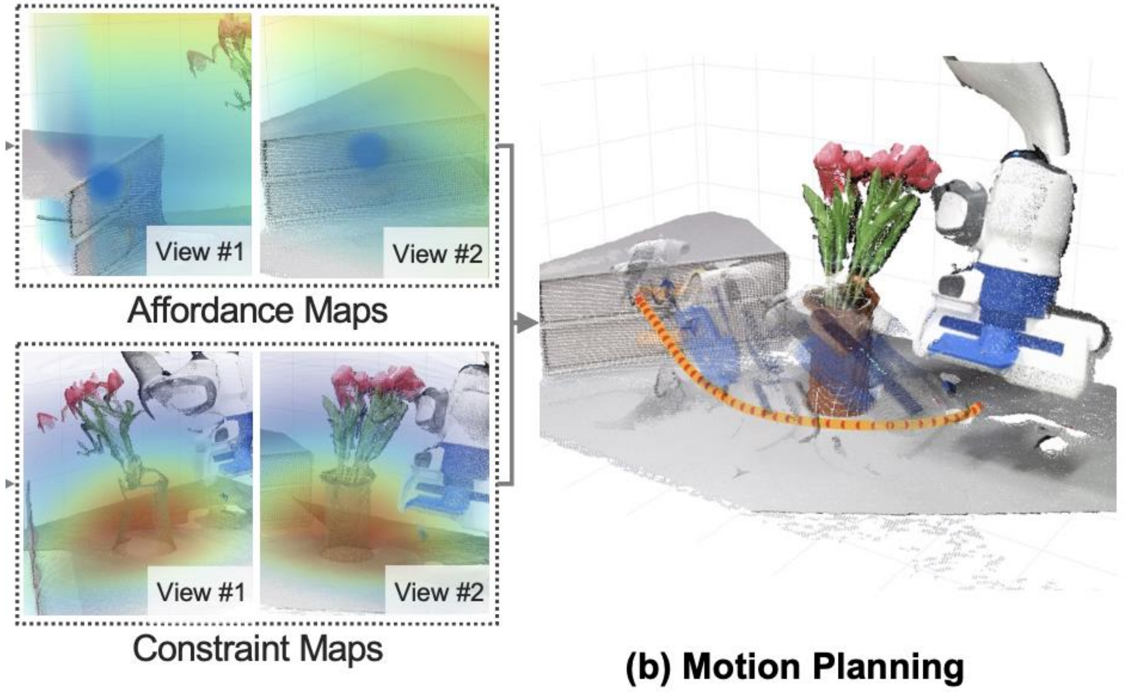
A complete example – Step4

- Generating query objection code by LLM context learning.



```
objects = ['handle1', 'handle2', 'egg1', 'egg2', 'plate']
# Query: topmost handle.
handle1 = detect('handle1')
handle2 = detect('handle2')
if handle1.position[2] > handle2.position[2]:
    top_handle = handle1
else:
    top_handle = handle2
ret_val = top_handle

objects = ['vase', 'napkin box', 'mask']
# Query: table.
table = detect('table')
ret_val = table

objects = ['brown line', 'red block', 'monitor']
# Query: brown line.
brown_line = detect('brown line')
ret_val = brown_line
```

parse_query_obj.txt

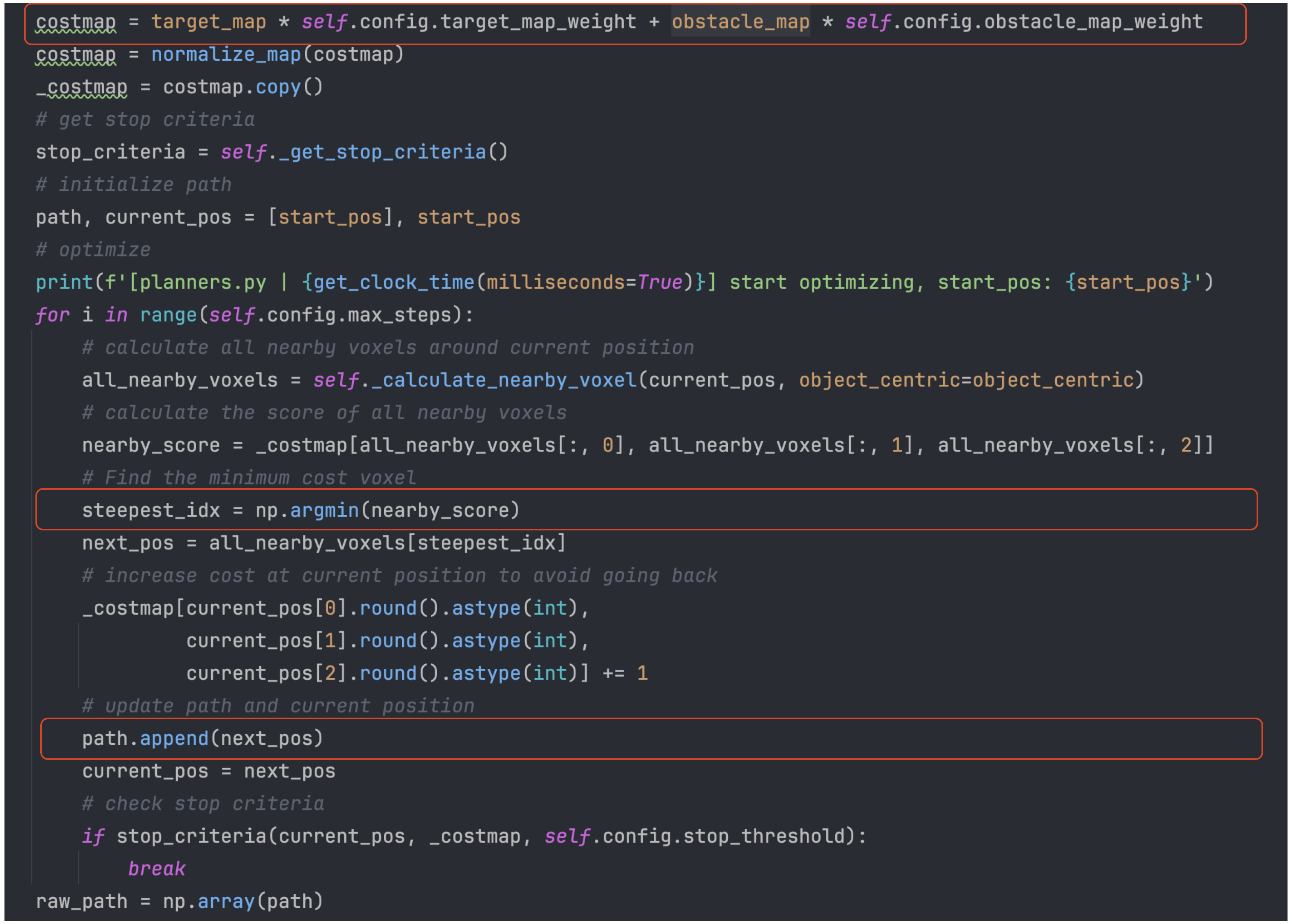


Next Slides will demonstrate how to get the trajectory.

How to get the path?

Ans: argmin *cost map*

After getting the path, use robotic motion kinematics can make the joints moving



```python
costmap = target_map * self.config.target_map_weight + obstacle_map * self.config.obstacle_map_weight
costmap = normalize_map(costmap)
_costmap = costmap.copy()
# get stop criteria
stop_criteria = self._get_stop_criteria()
# initialize path
path, current_pos = [start_pos], start_pos
# optimize
print(f'[planners.py | {get_clock_time(milliseconds=True)}] start optimizing, start_pos: {start_pos}')
for i in range(self.config.max_steps):
    # calculate all nearby voxels around current position
    all_nearby_voxels = self._calculate_nearby_voxel(current_pos, object_centric=object_centric)
    # calculate the score of all nearby voxels
    nearby_score = _costmap[all_nearby_voxels[:, 0], all_nearby_voxels[:, 1], all_nearby_voxels[:, 2]]
    # Find the minimum cost voxel
    steepest_idx = np.argmin(nearby_score)
    next_pos = all_nearby_voxels[steepest_idx]
    # increase cost at current position to avoid going back
    _costmap[current_pos[0].round().astype(int),
             current_pos[1].round().astype(int),
             current_pos[2].round().astype(int)] += 1
    # update path and current position
    path.append(next_pos)
    current_pos = next_pos
    # check stop criteria
    if stop_criteria(current_pos, _costmap, self.config.stop_threshold):
        break
raw_path = np.array(path)
```

Efficient Dynamics Learning with Online Experiences

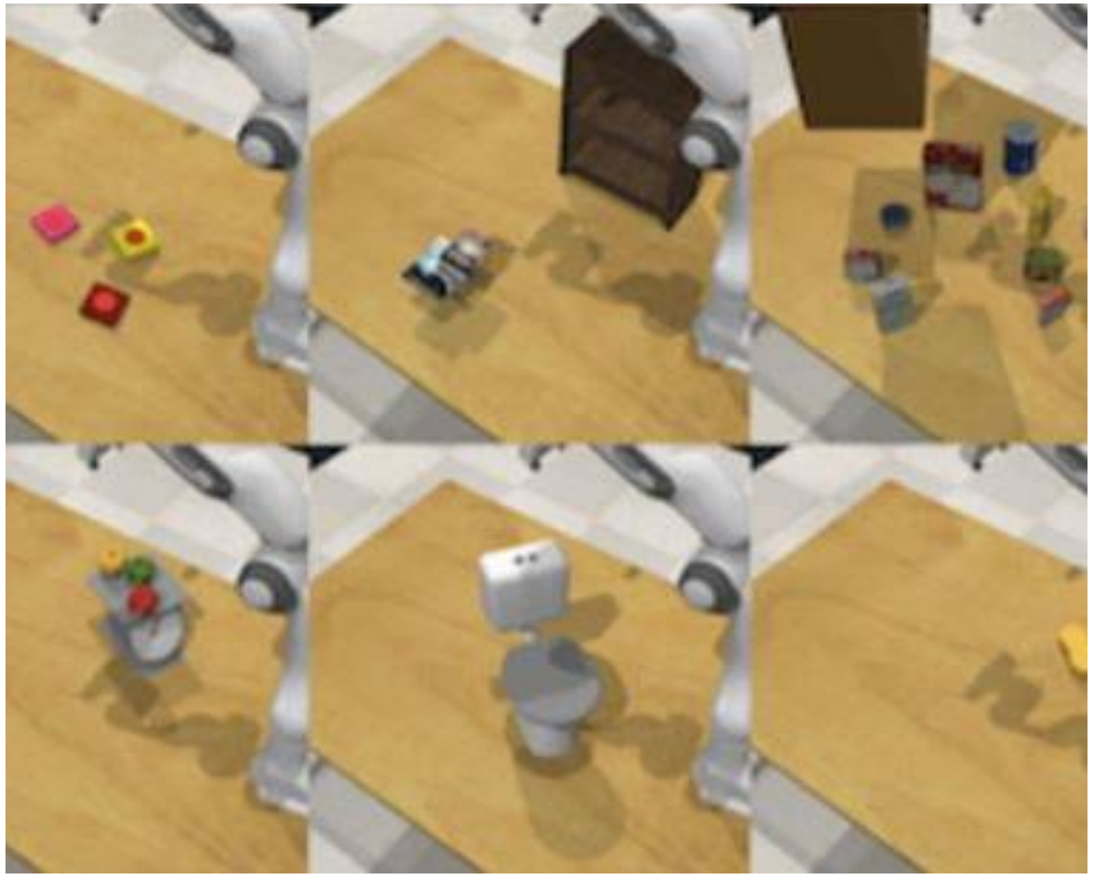
- VoxPoser can get trajectories directly from observation and LLMs/LVMs.

- Can VoxPoser learn where the target is in the next observation?

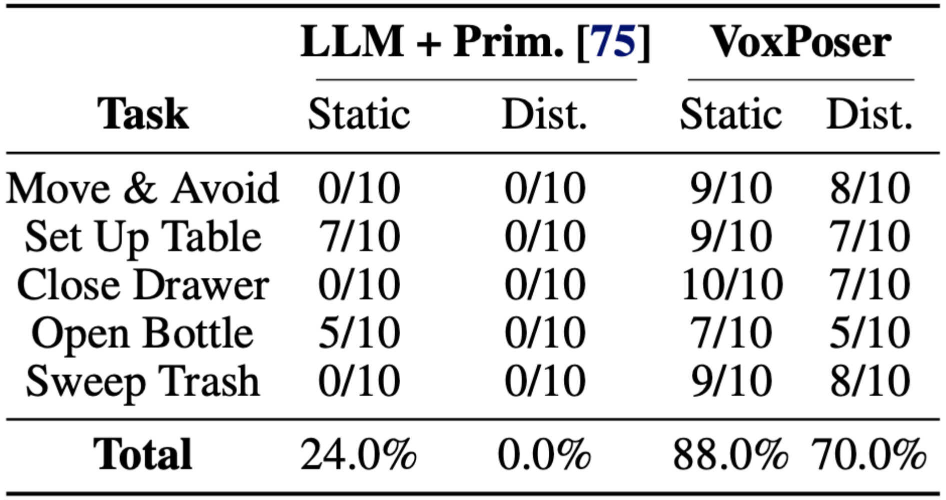
  - Step 1: Given obs and target (x, y, z)s by VoxPoser Affordance and Avoidance map, getting the moving distance and directions. These distances and directions are generated with random sampling. (Because the robot cannot move its end-effort to the target in a single step, it needs to move it gradually)
  - Step 2: Based on the moving distance samplings and directions, we can get the moving directions if is matched with the current target and observation. E.g ., if the gripper direction is +1, and the target position – contact position > 1, it means the direction is correct. Otherwise, the gripper direction is wrong, and we should ignore this motion.
  - Step 3: Calculate the next observations by the moving distances after filtering.
  - Step 4: VoxPoser can predict the trajectory even without really applying the moving actions.

Experiment Setting

- LLMs: GPT-4, which generates code recursively.
- VLMs: OWL-ViT, open-vocab detector, which provides the location by giv ing an object name.
- Environment: RLBench.
- Evaluation Metrics: Success Rate in real world.

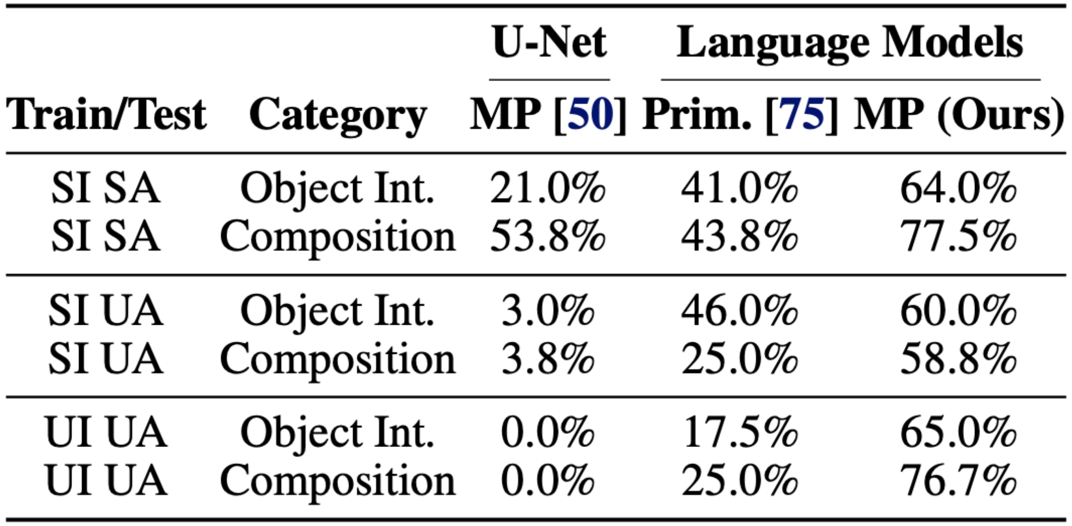


Experiment Results



Task	LLM + Prim. [75]		VoxPoser	
	Static	Dist.	Static	Dist.
Move & Avoid	0/10	0/10	9/10	8/10
Set Up Table	7/10	0/10	9/10	7/10
Close Drawer	0/10	0/10	10/10	7/10
Open Bottle	5/10	0/10	7/10	5/10
Sweep Trash	0/10	0/10	9/10	8/10
Total	24.0%	0.0%	88.0%	70.0%

Table 1: Success rate in real-world domain. VoxPoser performs everyday manipulation tasks with high success and is more robust to disturbances than the baseline using action primitives.



		U-Net	Language Models	
Train/Test	**Category**	**MP [50]**	**Prim. [75]**	**MP (Ours)**
SI SA	Object Int.	21.0%	41.0%	64.0%
SI SA	Composition	53.8%	43.8%	77.5%
SI UA	Object Int.	3.0%	46.0%	60.0%
SI UA	Composition	3.8%	25.0%	58.8%
UI UA	Object Int.	0.0%	17.5%	65.0%
UI UA	Composition	0.0%	25.0%	76.7%

Table 2: Success rate in simulated domain. “SI” and “UI” are seen and unseen instructions. “SA” and “UA” are seen and unseen attributes. VoxPoser outperforms both baselines across 13 tasks from two categories on both seen and unseen tasks and maintains similar success rates.

Efficient Dynamics Learning with Online Experiences

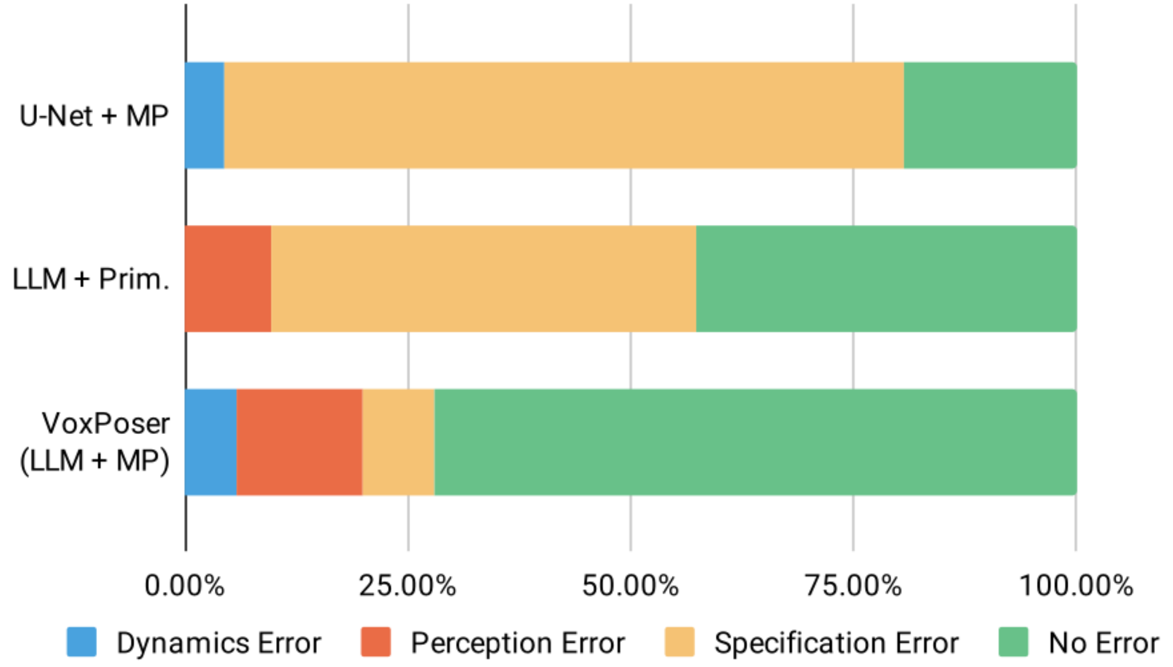
- Zero-Shot
  - Get Planning directly from observation
- No Prior
  - Dynamic Models predict next observation without initial trajectories
- With Prior
  - Dynamic Models predict next observation with k zero-shot trajectories.

	Zero-Shot	**No Prior**		**w/ Prior**	
Task	Success	Success	Time(s)	Success	Time(s)
Door	6.7%$\pm$4.4%	58.3$\pm$4.4%	TLE	88.3%$\pm$1.67%	142.3$\pm$22.4
Window	3.3%$\pm$3.3%	36.7%$\pm$1.7%	TLE	80.0%$\pm$2.9%	137.0$\pm$7.5
Fridge	18.3%$\pm$3.3%	70.0%$\pm$2.9%	TLE	91.7%$\pm$4.4%	71.0$\pm$4.4

Table 3: VoxPoser enables efficient dynamics learning by using zero-shot synthesized trajectories as prior. TLE (time limit exceeded) means exceeding 12 hours. Results are reported over 3 runs different seeds.

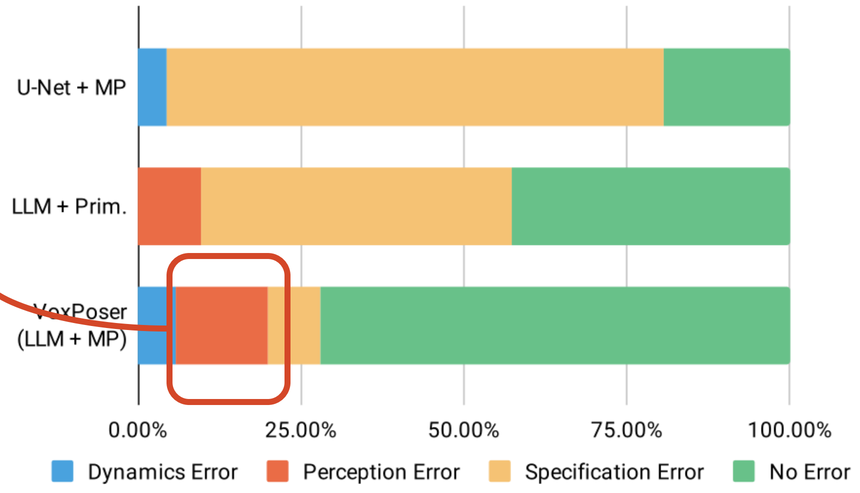
Error Breakdown

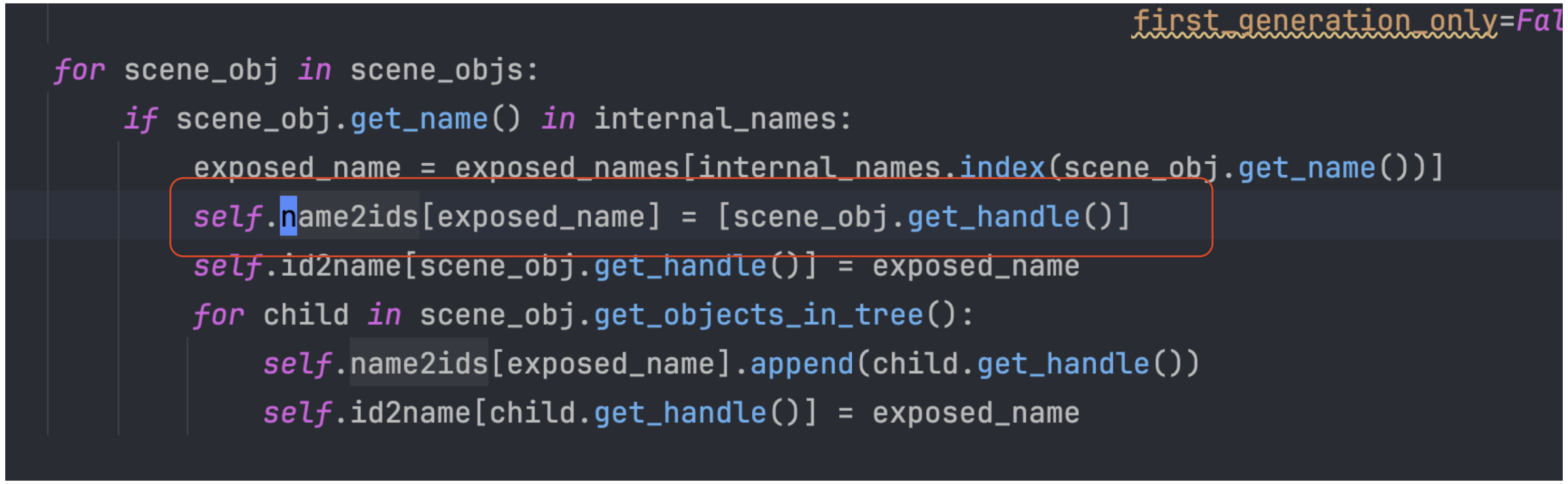
- "Dynamics error" refers to errors made by the dynamics model
- "Perception error" refers to errors made by the perception module
- "Specification error" refers to errors made by the module specifying cost or parameters for the low- level motion planner or primitives.
- VoxPoser achieves lowest "specification error" due to its generalization and flexibility



Limitation and Future

- 1. It relies on external perception modules, which
  holistic visual reasoning or understanding of fine-
- 2. Prompt Engineering.
- 3. More of perception:
  - the implementation of detect() in source code:





```
                                                            first_generation_only=Fal
for scene_obj in scene_objs:
    if scene_obj.get_name() in internal_names:
        exposed_name = exposed_names[internal_names.index(scene_obj.get_name())]
        self.name2ids[exposed_name] = [scene_obj.get_handle()]
        self.id2name[scene_obj.get_handle()] = exposed_name
        for child in scene_obj.get_objects_in_tree():
            self.name2ids[exposed_name].append(child.get_handle())
            self.id2name[child.get_handle()] = exposed_name
```

How to locate objects

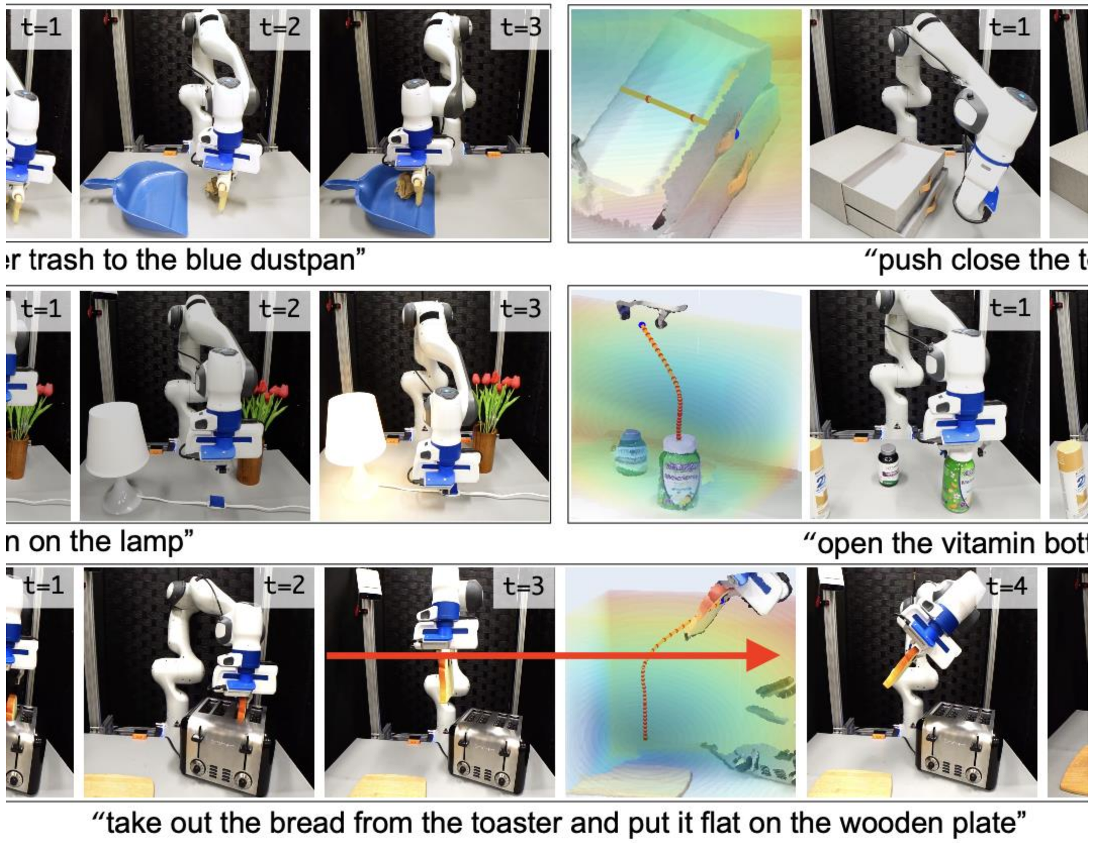
in 3D space is critical and

an open problem.

JOHNS HOPKINS UNIVERSITY

Review

- Prompt Engineering: Demonstrate the code generation for the robot motion planning.
- LLMs and VLMs: Generate corresponding code that drives the whole framework.
- Use Voxel Map and Value Optimization to remove the dependency of the primitive operations.
- Predict the next observation by random sampling motions.


t=1
t=2
t=3
r trash to the blue dustpan"
t=1
"push close the t
t=1
t=2
t=3
n on the lamp"
t=1
"open the vitamin bott
t=1
t=2
t=3
t=4
"take out the bread from the toaster and put it flat on the wooden plate"

- Leverage LLMs / VLMs to provide instructions for robotics problems.
- Recursive code generated by prompt engineering and LLMs.
- Perception is critical for improving the robotics success rate in such problem settings.
- Physical intelligence is fast-forwarding because of the foundation model's success. The future may be exciting.

Insights

Thanks!

Marvin Gao