Imperial College London

MEng Final Year Project

Imperial College London

Department of Computing

Lightweight One-shot Imitation Learning

Author:
Alfie Chenery

Supervisor:
Dr. Edward Johns

Second Marker:
Dr. Pancham Shukla

June 13, 2024

Abstract

One-shot imitation learning is a leading approach in robotic learning, valued for minimizing the need for time consuming, human-collected demonstrations. However, the computational demands of these algorithms are substantial. What options exist when these requirements surpass available hardware capabilities? This paper explores alternatives to state-of-the-art one-shot imitation learning algorithms and presents a system achieving comparable results with drastically reduced hardware requirements. This advancement lowers the barrier of entry, making such systems accessible to a wider audience, without the need for high-end GPUs.

Acknowledgements

Firstly, I would like to thank my supervisor, Dr. Edward Johns, for his support and guidance throughout this project. His expertise and feedback have been greatly appreciated.

I would like to personally thank my sayang, Dayana Muhd Faisal, for her help proof reading and her unending support through all the challenges of this year and those previous. I would likely not be at this point without her.

Lastly, I would like to thank my friends and family for their support throughout my whole degree.

Contents

Chapter 1

Introduction

Robotic manipulation is a complicated problem in the field of robotics in which we want a robot to interact with and influence it's environment in a specific way to complete a task. This is much more complicated than simple robot locomotion, as we don't just want the robot to move and exist within in its environment, but instead we want the robot to have meaningful interactions with certain objects in the environment. These interactions may be complex involving many moving parts, or multiple individual objects entirely. Furthermore, these interactions may be unpredictable if the robot's actions are prone to failure, or if the robot is interacting with objects it has no prior knowledge of.

As currently described, this problem is not particularly hard to solve. One could spend a few hours hand crafting an exact set of instructions for the robot to complete, such that when run from start to finish the robot completes the task. However, this implementation is missing a crucial feature. It does not generalise to different environments. What we mean by this is the robot should be able to complete the task even when the task is placed in a different environment. Specifically the robot should be able to analyse the state of the environment prior to or during the execution of the task and be able to adapt to this environment in order to complete the task.

Suppose we have a robot arm in which we can control the angle of each joint and an environment which contains a coffee mug. We could easily define the exact joint angles at each time step, such that when the robot executes these positions, it manages to pick up the mug. However, if we now change the environment, and move the mug to the side, then the robot is going to fail the task. It will follow the instructions as before, trying to pick up a mug that is no longer there. This control algorithm does not generalise to different environments, it is hard coded for one specific environment setup. We want a system which can sense the environment in some way, and change it's actions accordingly, making decisions based on the information from its sensors. For example seeing the mug has been moved to the side, and changing the joint angles such that it still manages to pick it up.

It is obvious that hard coding the instructions for every possible task set up is intractable. The general approach to solving such a problem involves trying to teach the robot an understanding of the meaning behind the underlying task itself, abstracting it away from the environment it is performed in. For example, we do not want to teach the robot "How do I pick up ***this*** mug?" We want to teach it

"What does it mean to pick up ***a*** mug?" and "How do I know when I have picked up a mug?". These success criteria questions are something we will refer back to when considering different algorithmic approaches in Chapter 2.

If the robot is able to comprehend this higher level notion of what it truly means to complete the task, without relying on environment specific information, then we have successfully extracted the task out from the environment. In this sense the task can be placed in any environment, and the concept of the task itself has not changed, the robot knows how to pick up a mug in whichever environment it is found in. The robot does not need to work out how to complete the task, it only needs to compute how to apply the already known task in this new unseen environment.

1.1 Contributions

Existing solutions to this problem broadly fall into two categories: Some solutions offload the work of extracting environment agnostic task information to the engineers. This requires us to formalise the task in a complicated mathematical expression which can be applied to any environment. These types of systems are explored in Section 2.2. Alternatively, engineers can directly generalise the task by presenting it in many different environments and showing the robot how to complete the task in each case. This however still multiplies the workload of engineers providing training data to the system. These types of solutions are explored in Section 2.3
Some newer solutions attempt to alleviate this issue by capturing specialised additional information during training time which allows the task to easily generalise. This additional information is something which can be automatically collected with no extra effort from the engineers. However, the downside to this approach is that for the robot to understand and use this additional information requires very large amounts of computational power, often in the form of huge artificial intelligence models. we explore these approaches further in Subsection 2.3.2.

This paper however, manages to achieve a one-shot imitation learning system, with comparable performance to existing solutions, while operating under much tighter computational constraints. The solution presented in this paper successfully utilises existing works in the field of computer vision to drastically reduce the hardware requirements of running such a system.

In our system we also incorporate strategies from state-of-the-art solutions to produce a system which can learn multiple independent tasks. The robot is able to identify which task to complete from its corpus of trained tasks, using only observations of the environment it finds itself in and no external input from a human. It can identify the task to complete by analysing the environment, and then executes this task in the specific test time environment. It achieves all of this with minimal pre-training, each learned task consisting of a single human provided demonstration combined with automatically collected data which facilitates the generalisation to novel environments. This data comes with no additional workload to the demonstrator.

1.2 Ethical Considerations

This project does not work directly with people or animals. However, there could be safety considerations if deploying to a physical robot arm. These potential safety concerns are a consequence of using any robot arm and are not specific to this project. As such normal safety protocol in remaining clear of the robot arm during operation would be sufficient.
There are no legal or licensing concerns with this project. All libraries used are open source and can be installed through the default Python package manager, pip.
A widely generalisable learning agent will be applicable to many scenarios and tasks, some of which may be malicious in nature. While it is possible that work from this project could be taken and misused to teach a robot morally questionable tasks, this is not the intended use or focus of this project.
Finally while this project does not directly focus on environmental issues, we should consider the energy used running the simulations and equipment. While this is a non-zero amount, it is for the purposes of research and well within reasonable limits. Furthermore, this paper specifically focuses on producing a robot learning algorithm with lower hardware requirements to run effectively. This allows our system to run on lower power and more efficient machines than previously possible. While this is undoubtedly a benefit of the solution presented in this paper, reducing the environmental impacts is not the focus of this paper.

Chapter 2

Background

As discussed in Chapter 1, a simple robot learning algorithm will allow the agent to generalise to different environment setups. There are three main techniques which allow us to achieve this.

2.1 Model Based Control

In this first method, in order to teach the robot how to complete the desired task, the human needs to provide to the robot a model of the environment and a reward function. The model is a function which encapsulates the dynamics of the environment, and the ways that actions influence the state. Specifically, it is a map from the current state and chosen action to the resulting next state:

$$\mathcal{P} : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{S}$$

The reward function is some function which maps the current world state and action chosen by the agent to a real number:

$$\mathcal{R} : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$$

Here $\mathcal{S}$ denotes the set of all possible states the system can be in, and $\mathcal{A}$ denotes the set of all possible actions the robot may take.

The robot then tries its best to maximise this reward by choosing the actions which give a high reward now, but that also lead to states with the potential to yield high rewards later. It is able to predict which states it will reach later using the environment model. The priority between getting an immediate reward now and ensuring higher rewards later is controlled through the discount factor $\gamma \in [0, 1]$. If the robot is provided with these aspects, and the model perfectly captures the environment dynamics , then the robot can directly solve the induced Markov Decision Process (MDP), for example using a dynamic programming approach to calculate the expected rewards of being in every state. Then the agent can simply take the actions which ensures it follows the sequence of states with highest expected rewards.

Referring back to the success criteria questions mentioned in Chapter 1, this method teaches the robot "How do I know when I have completed the task?" The task is completed when this given reward function is maximised. It is up to the engineer to ensure that this reward function really does capture the specifics of the underlying task, since the robot has no true understanding of the task. The robot

is merely trying to maximise a number, wherein it has been told that getting the number as big as possible, will complete the task. We have also implicitly defined "What does it mean to complete the task?" To complete the task is to score highly in the reward function.

The problem with this method is it is a lot of work to design the reward function and environment model. In some cases the environment may not even be fully observable. In this case a complete model is simply not possible. As a result Model based control is most often used in specific lab settings where the environment can be meticulously controlled. It is rarely used in 'in the wild' robot learning due to the inability to guarantee control over the environment, resulting in an unreliable environment model. As such, model based control lacks the generalisability we strive for in this paper, and so will not be considered further.

2.2 Reinforcement Learning

The second method at our disposal is Reinforcement Learning. In this method we do not need to provide the agent a model of the environment. However, the agent still requires a reward function. The robot will explore the environment on its own to experience states and see which ones yield high rewards [1]. After sufficiently long training, the robot's ideas of which states have a high associated cost should approach the true values. These values were known outright in model based control thanks to the environment model, however in reinforcement learning the agent explores the state space itself to approximate these values. The robot is then able to select the actions which lead to states it has experienced giving high rewards.

This method answers the success criteria questions in exactly the same way as with model based control. The reward function still solely encodes information as to the completion of the task. The only difference is in how much information the robot has available to pursue improving the reward function, and hence completing the task. Instead of a perfect understanding of the world and how these interactions affect the reward function, the robot only has information it has experienced itself.

While this method is a big improvement over model based control, there is still a problem. We still need to provide the agent a reward function. This can be difficult to formulate, even more so without the environment knowledge of model based control. When we had access to the environment model we had knowledge of the relative positions of objects, and could easily define tasks which involved moving one object to another place. For example the reward function could be the negation of the distance between the object's position and the target position. Alas, without the environment model, the agent cannot know perfectly the position of the objects and the target position. These must be estimated from the environment observations it makes. Such observations are usually in the form of a camera attached to the robot, often mounted in a fixed third person view, or mounted on the wrist for a first person view. From these observations, the robot can estimate the position of objects, given their position in the image, and the known position of the camera when the image was taken.
Despite only having access to estimates through observations, carefully designing a reward function is a reasonable approach. The issue comes with the increased workload of a generalisable system. If we wish to teach the agent multiple tasks we are required to provide the agent one reward function for every task we wish

it to complete. Furthermore, we need some way to decide which task to complete and select the correct reward function. Since each task may be wildly different, it is simply not possible to encode all of these tasks with a single reward function. As such, reinforcement learning is more applicable to fine tuning a solution to a specific task. Since the robot explores the environment itself, there is little bias to the solution, given the robot explores sufficiently within the state space. This is desirable for finding optimal solutions to a problem that the engineers may not have considered. As we will see in the next section, other solutions often introduce large amounts of human bias to the solutions the robot finds. However, poor exploration from the robot can lead to sub-optimal results, if the agent gets stuck exploiting a local optima, when it could have reached a better solution by exploring further. This tendency to get stuck with a sub-optimal solution is obviously undesirable.

It can be difficult to decide a strategy for this 'exploration vs exploitation' problem. A high exploration means the agent can in theory experience more of the state space. However, the exploration can be very unfocused, leaving the robot exploring an area which is not likely to yield good results. Exploitation refers to the agent choosing the best action it can at its current state. A good algorithm balances some exploitation to keep the robot focused in the direction of the currently believed best solution, while allowing exploration to test out close by states, which are more likely to perform well. This balance is often achieved through 'epsilon-greedy exploration', wherein ϵ is a hyperparameter probability. An ϵ proportion of the time the agent chooses the greedy, best currently known action in this state. $1 - \epsilon$ of the time, it chooses any other random action instead. Choosing a good value for this hyperparameter is difficult and largely dependent on the specific problem environment. Additionally, this all assumes the environment is continuous. Specifically that near by states behave similarly to their neighbours. If this is not the case, then keeping the exploration focused about the current optimal solution, offers little benefit.

2.3 Imitation Learning

With these issues in mind, the third common robot learning method is Imitation Learning. This method attempts to simplify the data collection required of the engineers and reduce the impact of hyperparameters. Instead of needing to provide a reward function to the agent, we instead provide demonstrations of how to complete the task. The key paradigm shift is that rather than telling the robot what we want it to do (through the reward function) and leaving it to work out how to do that, we instead show the robot exactly how to perform a task. This generally gives a much faster return on investment. The robot is immediately able to perform the task somewhat well as opposed to learning an optimal solution through time consuming exploration in the environment. This does however have a few downsides. Most notably we, the engineers, need to be able to perform the task ourselves. We cannot teach the robot to perform a task if we fail to show it a successful demonstration. Depending on the implementation the agent may process the provided demonstrations differently.

One approach is to use Inverse Reinforcement Learning to infer the reward function from the provided demonstrations [2]. We may use a function approximator, such as a neural network to replace the reward function. Given the robot's state and chosen action, we want the approximator to output the reward for this action.

In this scenario we use the demonstrations, along with associated, presumably high rewards to train this network. For example stating that all human demonstrations receive a fixed reward of 100, since they accomplish the task. With a reward function now known, the agent can use the previously discussed forward reinforcement learning to solve the task in similar situations.
In this implementation we attempt to again teach the robot "How do I know when I have completed the task?" The difference being how we convey this to the agent. We give the robot a number of demonstrations which are considered as successful completions of the task. From this the robot tries to infer what was common about all these demonstrations which makes them successful at completing the task. The difficulty lies in the fact that often very many reward functions could explain the behaviour. Was the task to grasp the mug or to just move the end effector to the right? To us it may seem obvious that it was probably the first option, but to the robot both of these options are valid.

Another approach is to forgo the reward function entirely. In Behavioural Cloning, the agent instead uses the demonstrations as a blueprint to follow. Similar to Inverse Reinforcement Learning, we will still train a function approximator. However this time we will be using it as the robot's policy, not the reward function. The approximator will take in the robot's current state (or an observation of the state if it is not fully observable), and predict the optimal action. This approximator is trained on the demonstrations so that for each state in the demonstration, the ground truth output is the action which the human took in the demonstration.
In this implementation we are changing how we answer the success criteria questions. This time we are directly teaching the robot "What does it mean to complete the task?" We are teaching the robot exactly how we as humans would solve this task. In this method the robot never gets an answer to the question "How do I know when I have completed the task?" The robot has no actual concept of what it is doing and how to know when it succeeds. All it knows is "This is what the human did, so I should do that too." It trusts that the demonstrations it has been given, will allow it to complete the task.

Both of these approaches to Imitation Learning require a large diverse set of demonstrations. This is because we use the demonstrations to train a neural network. As such we need lots of training data in order to expect the function approximators to produce reasonable results. The trade off is that instead of providing a single but complicated and hard to craft reward function, we instead need to supply many simple to collect demonstrations. Although note that simple does not necessarily imply easy. While the collection of demonstrations can be straight forward, either through teleoperation (remote controlling the robot) or kinesthetic demonstration (physically moving the robot yourself), we need to provide a large amount of them which can be time consuming.

The additional complexity of providing demonstrations is hidden in the fact that for the agent to learn effectively, the demonstrations must be of high quality. For example, we need to ensure the demonstrations actually accomplish the goal, otherwise the robot will likely also fail to achieve the goal. Since the robot does not have a reward function, it has no way to evaluate which demonstrations were good. It instead treats all demonstrations equally, trusting that they achieve the same thing we want the robot to achieve. If a demonstration slightly misses the goal, the robot may think that this near miss state, is actually the true goal state. Problems like this one make the agent very sensitive to the initial conditions, since poor quality

demonstrations may prevent it from learning anything.

2.3.1 State distribution problem

Another complication is that we need the demonstrations to be diverse so that a wide variety of states are reached. If the demonstrations do not sufficiently cover the state space the robot will be operating in, then the robot will likely find itself in a state for which it has no information how to proceed. In this scenario it is likely to perform an action which seems random. This is simply because function approximators struggle to extrapolate to inputs which are out of distribution. If we want our agent to know what to do in a wide variety of states it may find itself in, then we need to show it how to handle these states, by making sure our demonstrations at some point visit these states.

Ensuring the demonstrations cover the state space the robot will be experiencing during testing is a difficult problem. Simply because we cannot know for sure which states the robot will need to traverse in order to reach the goal. While providing a diverse set of demonstrations is a good start, it is always possible the robot will find itself in a state for which it does not know how to proceed. 'DAgger' (Dataset Aggregation) is a Behavioural Cloning technique which aims to solve this problem. The key idea is for the agent to determine its policy to complete the task. For each state along this policy, the system requests a demonstration from the human. This ensures that we provide demonstrations for states which the robot actually visits. There are too main problems with this design. Firstly, we do not pre-collect all the demonstrations. The robot will propose a policy during test time, and then request more training demonstrations. This means the training time is interleaved with the test time. A preferable design would allow us to collect all the demonstrations up front, and then test the agent with no further demonstrations required. The second problem is the sheer volume of required demonstrations. DAgger requires us to collect even more demonstrations than before. Furthermore, the number of required demonstrations is dependent on the length of the trajectory. If the environment involves a continuous state space or a task involves a very long and complex trajectory, then this increases the demonstration burden even further.

Alternatively, rather than requesting a demonstration for every state along the trajectory, we can instead request a demonstration only when the robot is unsure what it should do. This is a strategy employed by 'Uncertainty-Based Imitation Learning'. While this still has the issue of mixing the testing and training time, it requires far less demonstrations overall than DAgger. We may use any method of quantifying uncertainty within the robot's actions. One such approach is to use an ensemble of policy networks. Each network is trained on a subset of the training data, and when the predictions of optimal action differ wildly between each network, it means the agent is unsure what to do.

2.3.2 Reducing the dependency on demonstrations

While these previously stated methods help to ensure the demonstrations are diverse, they generally require collecting far more demonstrations to achieve this. The requirement to collect a large set of human provided demonstrations is the main

bottleneck in an Imitation Learning system as it is a time consuming process. Many papers have attempted to solve this problem through various means.

The work of A. Mandlekar et al. proposes a system to upscale a small number of demonstrations into a larger set [3]. Their system, 'MimicGen', takes as input a small number of human provided demonstrations. It then segments these demonstrations into "object centric subtasks" and constructs new ones by stitching together subtasks. MimicGen also transforms the scene so that each demonstration is diverse in positioning. This system reduces the burden of collecting demonstrations since we now need to collect far fewer demonstrations and can simply up-sample to a larger dataset. In the paper they claim "Image-based agent performance is comparable on 200 MimicGen demos and 200 human demos, despite MimicGen only using 10 source human demos." This means we could collect only 10 demonstrations, up-sample to 200 using MimicGen and have the same accuracy as if we collected 200 demonstrations manually. This dramatically reduces the workload of collecting demonstrations.

Another approach to solve the need for a large set of demonstrations is to provide the agent with alternative information to allow it to generalise to unseen environments thus removing the need for a large dataset of demonstrations entirely. The logical progression of this idea is to reduce the demonstration burden on the engineers as far as possible. So called 'one-shot' algorithms stand as the pinnacle of this design philosophy, by allowing the robot to learn a given task from only a single human provided demonstration.

Specifically, P. Vitiello, K. Dreczkowski and E. Johns utilise an RGB-D image of the environment before the demonstration in their paper titled "One-Shot Imitation Learning: A Pose Estimation Perspective" [4]. This paper creates a system which upon being presented with a new scene, takes a similar picture of this new environment. The system then calculates the transformation which maps the object of interest in the demonstration to its position in the new environment. For example the system may notice by comparing the two images that the object of interest has moved 10cm to the left and rotated 30 degrees clockwise about the Z axis. The agent then applies this same transformation to the robot pose in the demonstration trajectory. The effect of this is to transform the positions at each time step so that the robot will move to a position 10cm to the left with a 30 degree clockwise rotation when compared to the original demonstration. Conceptually what this does is map the demonstration from the old environment to the new one. Now that the demonstration has been mapped to the new environment, the agent can simply execute the transformed demonstration, which will complete the task.
This paper manages to drastically reduce the workload of collecting demonstrations for the imitation learning task. As the name "One-shot imitation learning" suggests, only a single demonstration is required. Furthermore, the additional information required by the agent, the "context vector" as this paper calls it, can be collected automatically by the agent during the demonstration with no additional human input needed.

N. Di Palo and E. Johns propose a similar method [5] which uses a wrist mounted camera as opposed to the previous solution which utilised a third person camera which captured the entire scene. The advantage of a wrist mounted camera is that

the camera is moving with the end effector. This effectively makes the image captured translation and rotation invariant to the end effector pose. The only important information is the relative pose between the end effector and the object in the scene. The advantage of this is that we reduce the need for complicated computer vision methods for pose estimation of the objects between the two images [4]. In the work by N. Di Palo and E. Johns, since the camera moves with the end effector, changes in the end effector position and orientation are reflected very obviously in the captured image [5]. This also means that the agent simply needs to line up the end effector so that the captured image is the same as the demonstration image. If this is the case then the relative position between the end effector and the object is the same as in the demonstration. As such the current position and orientation of the end effector after lining up but before starting the task, can be calculated using forward kinematics and added as an offset to the poses in the demonstration. As before, this transformed trajectory can now be executed which will execute the task as in the demonstration, but mapped into the new scene.

Another advantage is that a wrist mounted camera also reduces the noise within the image. A third person camera which captures the entire scene will capture all objects within the scene, even though only one may be of relevance to the task. This additional noise could cause the agent to incorrectly compute the transformation between the demonstration and current scene, and therefore fail the task. Alternatively, a wrist mounted camera will only capture the focused view around the end effector. This means that during the manipulation task, the object of interest will likely be the only object within the image. This focused view is much less susceptible to noise of other objects within the environment.

There are of course, as with every design decision, some drawbacks to using a wrist mounted camera. Most obviously, due to the focused view failing to capture the entire scene, it can be difficult to locate the object of interest. This means that a single image may no longer be enough as the object we are looking for may not be present in the image if the camera is pointing the wrong way. This necessitates a live feed of the wrist mounted camera as opposed to the single image capabilities of the third person camera, which is a major drawback.

The other large drawback is a consequence of this first one. Since a live feed of the image is now required, any objects blocking the camera will now cause a serious problem. This was not an issue with the third person camera as it could capture the entire scene and everything it needed to know before starting to execute the task. However with the wrist mounted camera, if the robot picks up an oddly shaped object which happens to block some or all of the image feed, then the robot will quite literally be executing the task blind. This is not such a problem for tasks involving a single object of interest, since it is assumed the robot can navigate the arm and line up with the demonstration image while no object is in its gripper, and therefore nothing should be blocking the camera. The real problem comes with multi-stage tasks.
Consider the task which is to pick up a spoon and place it in a mug. The demonstration would pick up the spoon from a particular position and angle relative to the spoon. It would then move to a particular position and angle relative to the mug and release the spoon from its gripper. Now suppose the agent is deployed on a new task with a slightly bigger spoon and the spoon and mug rearranged. The

agent would be able to find the spoon, move to a relative position which matches as in the demonstration, and execute the trajectory to pick up the spoon. However, since the task involves two objects, we can no longer just execute the entire demonstration from this point, since the relative position between the spoon and the mug may have changed. As such we need to partition the trajectory and line up the end effector relative to the object of interest for each segment. It is clear to see that the trajectory can be segmented every time the object of interest changes [3]. This itself generally happens when the gripper closes, picking up the previous object of interest, or when the gripper opens, placing the grasped object in a position relative to the current object of interest. In this example initially the object of interest is the spoon, we move relative to the spoon and execute the segment of the demonstration which picked up the spoon. Now that the spoon is attached to the end effector, the object of interest is now the mug. We want to move to a position which is relative to the mug, the same as in the demonstration. At this point we can execute that segment of the trajectory to drop the spoon. This system works fine given that the camera is a live feed and so we can keep checking the relative position of the end effector to the object we currently care about. The issue however, comes when the object in the gripper is obstructing the view of the camera. If this slightly larger spoon is now blocking the camera, the agent will be unable to find the mug and line up relative to it. In this scenario there is nothing the robot can do. It needs the spoon in its gripper to progress to the next segment of the demonstration, but when it is holding the spoon it cannot see the environment. The robot is unable to know its location relative to the mug, so cannot progress with the next segment. This demonstrates the issue with a wrist mounted camera. There are some objects which when picked up will block the camera, and there is very little the robot can do to work around this. There are just some objects the agent cannot deal with. The affects of this issue can be mitigated by careful placement of the wrist mounted camera. Placing it further up the wrist away from the end effector makes it less prone to being blocked by the object in the gripper. While this is potentially a big problem, it is unlikely that this will happen in practice. This issue could be further prevented by utilising two cameras, perhaps one on each side of the wrist, reducing the chance that an object would block both cameras, rendering the robot blind again. Of course with all these solutions, a contrived scenario with an object shaped perfectly to block the cameras will cause problems. However with everyday regular items, the affects of this problem should be limited.

In this paper we will largely focus on the 'DinoBot' system created by N. Di Palo and E. Johns [5] as a case study for our work. As eluded to in Chapter 1, the limiting factor of this and many similar works is by no means their accuracy and performance. Rather it is the physical hardware requirements to run them. We will focus on utilising some of the core ideas from the 'DinoBot' system, while replacing certain components which proved too computationally costly. We replace these with alternative algorithms which require considerably fewer resources, to produce a system with similarly accurate results. The full design process for our system is covered in Chapter 3.

2.4 Rotations and Orientations

Throughout this report best effort will be made to not confuse the terms 'rotation' and 'orientation'. While mathematically these terms are represented identically,

they have different semantic interpretations. As such the language within this paper will be consistent with the following definitions:
A **rotation** is a function which refers to the circular movement of an object around a central point or axis.
An **orientation** is an angular displacement relative to an agreed upon reference frame. It is the resulting state after a rotation is applied to a body which was initially in the starting reference frame. Typically the base orientation will be such that the object's local reference frame aligns with the world coordinate frame. The object's orientation is then the rotation about the object's local origin, which aligns the world reference frame with the object's local reference frame.

From these definitions we can realise that to find an object's new orientation following a rotation, we compose the applied rotation with the object's previous orientation. This results in the total rotation which maps the object from the base reference frame to its new local frame, which is by our definition, its new orientation.

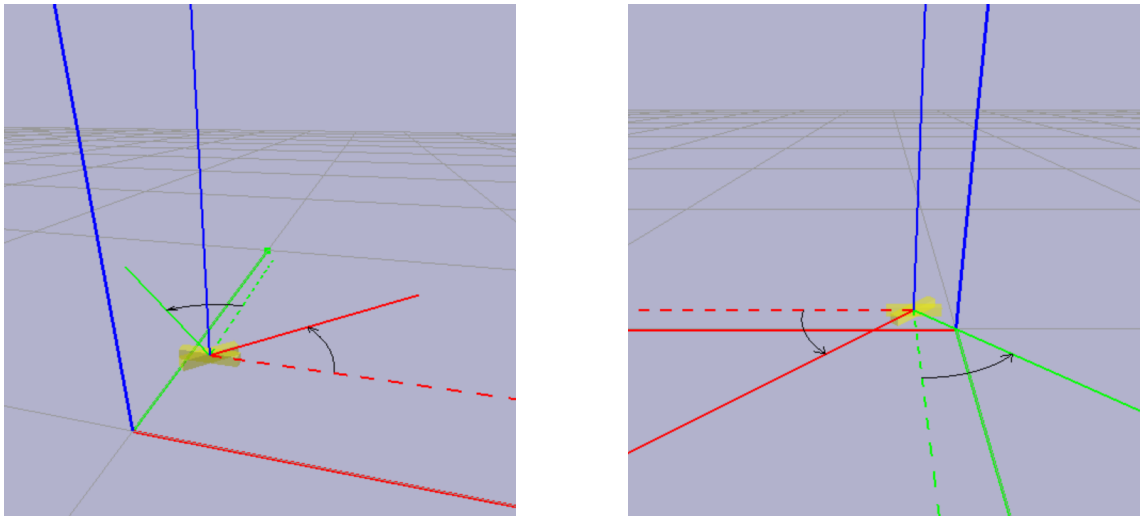


Figure 2.1: An object before and after a rotation of $\frac{\pi}{4}$ about the Z axis [1]

Figure 2.1 showcases an object in two different possible orientations. The dotted lines show the object's local coordinate frame prior to the object being rotated. We can see that the dotted coordinate frame aligns with the world coordinate frame, just translated by the object's position. As such we define this orientation as the initial frame of reference. Therefore we would choose to encode this orientation as the identity rotation since no rotation is needed to align the world frame with the object's local frame.
The solid line coordinate frame shows the object's local coordinate frame after it has been rotated $\frac{\pi}{4}$ radians about the Z axis *from the initial frame of reference*. Therefore we represent this orientation with the rotation $\frac{\pi}{4}$ radians about the Z axis. It would be equally valid to encode this as the rotation matrix, Euler angles or any other encoding of this rotation, but we will see in the next subsection that the method used in this project is quaternions.
The figure shows the exact same setup from two different view points, to better visualise the rotation described.

[1] Refer to Subsection 2.4.2 for an explanation on rotation direction

2.4.1 Quaternions

Quaternions are a 4 dimensional number system [6] which can be used to represent and manipulate orientations and rotations in 3D space [7]. In many ways they are an extension of the complex numbers, adding an additional 2 complex axes. A quaternion consists of a real component and 3 imaginary components which are often grouped together and called the vector component.

$$q \in \mathbb{H}, \qquad q = a + b\,i + c\,j + d\,k, \qquad where\ \ a, b, c, d \in \mathbb{R}$$

The fundamental imaginary units follow the set of rules listed below:

- $i \neq j \neq k$
- $i^2 = j^2 = k^2 = i\,j\,k = -1$
- $i\,j = k, \quad j\,i = -k$
- $j\,k = i, \quad k\,j = -i$
- $k\,i = j, \quad i\,k = -j$

Of relevance to us however, is how these numbers can represent 3D rotations. If we consider a rotation in axis-angle form, that is that we rotate some angle θ about some normalised unit vector $v = (x, y, z)$, then we can produce the unit quaternion:

$$q = \cos\frac{\theta}{2} + (x\,i + y\,j + z\,k)\sin\frac{\theta}{2}$$

This quaternion is an encoding of the specified rotation. For the quaternion to encode a rotation with no scaling, the quaternion must be of unit length:

$$q = a + b\,i + c\,j + d\,k, \qquad |q| = \sqrt{a^2 + b^2 + c^2 + d^2} = 1$$

This property will be satisfied if v is a unit vector, or can be achieved oherwise by simply normalising the quaternion, dividing each component by its L2-norm.

Furthermore, we know by Euler's rotation theorem [8] that for any composition of rotations to a sphere about its centre, there exists a diameter of the sphere which forms an axis, and an angle for which rotating about said axis produces the same net displacement as the composition. We can generalise this to any rigid body by considering the circumscribing sphere centered on the object's position. From this we can convince ourselves that for any orientation the object could be in, we can represent it as a single axis vector and rotation angle, which we can then encode as a quaternion.

Quaternion addition and multiplication mirrors that of the complex numbers. Considering the numbers as expressions and simplifying like terms. However it is important to note that unlike the complex numbers, quaternion multiplication is not commutative. Multiplying quaternions is still associative.

$$qp \neq pq, \qquad pqr = (pq)r = p(qr)$$

If q is a unit quaternion, then the inverse is the same as the conjugate, gained by negating the vector part of q, while leaving the real part unaltered.

$$q = a + b\,i + c\,j + d\,k, \qquad |q| = 1 \implies q^{-1} = q^* = a - b\,i - c\,j - d\,k$$

A unit quaternion q which represents a 3D rotation can be used to rotate a vector in the following manner:
Create the pure quaternion p with real component = 0, and vector component equal to v. Note that v does not necessarily need to be a unit vector.

$$v = [x, y, z], \qquad p = 0 + x\,i + y\,j + z\,k$$

Then the rotated vector v' is given by:

$$v' = [x', y', z'] \quad where$$

$$p' = w' + x'\,i + y'\,j + z'\,k = qpq^{-1}$$

While this is a nice property which makes rotating vectors very computationally easy, it is not the operation we will be performing most often. As mentioned above, we consider orientations as a rotation relative to a starting reference frame. Therefore, to rotate an object, we need to compose the desired rotation quaternion, with the current orientation. If the current orientation is stored as a unit quaternion q, and we wish to rotate the object by some unit quaternion r, then the resulting orientation is given by:

$$q' = rq$$

We can see that this is the correct formula for composition of rotations by using it to rotate a vector.

$$v' = qvq^{-1}, \qquad v'' = rv'r^{-1}$$

$$\implies v'' = rqvq^{-1}r^{-1} \implies v'' = (rq)v(rq)^{-1}$$

Hence rotating v by q and then by r is equivalent to rotating a single time by the composition rotation rq.

The other operation we make use of in this paper is calculating the rotation between two orientations. That is to say given two orientations q and q', we want to find the rotation r such that rotating q by r gives q'. If the orientations were stored as vectors then this is a more complicated procedure. However, as discussed, we represent orientations as rotations relative to a starting reference frame. In this setup, it is obvious to see that given q and q' are themselves rotations we could first rotate by the inverse of q to get back to the starting reference frame, followed by rotating by q'.

$$r = q'q^{-1}$$

We can verify that this satisfies our desired equation:

$$rq = (q'q^{-1})q = q'(q^{-1}q) = q'$$

2.4.2 The right hand rule

The right hand rule refers to a convention defining the positive direction for rotation about an axis. While using your right hand to make a 'thumbs up pose', your thumb represents the positive direction of the axis of rotation, and your fingers curl in the direction of positive rotation. This can be seen in Figure 2.2.

We note that if we were to invert our hand, so that our thumb points in the opposite direction, then the fingers now curl in the opposite direction relative to the

Figure 2.2: The right hand rule for rotation

initial axis. Therefore we can imagine if we were to rotate in the negative direction about this inverted vector, we would achieve the same rotation as before. This is to say a positive rotation about some vector is equivalent to a negative rotation about the negation of the vector.
This is the reason that quaternions are considered a 'double cover' of 3D rotations, since these two constructions describe the same rotation, but are represented by exactly 2 distinct quaternions, q and -q.

$$q = \cos\frac{\theta}{2} + v\sin\frac{\theta}{2} \qquad and \qquad -q = -\cos\frac{\theta}{2} - v\sin\frac{\theta}{2}$$

Chapter 3

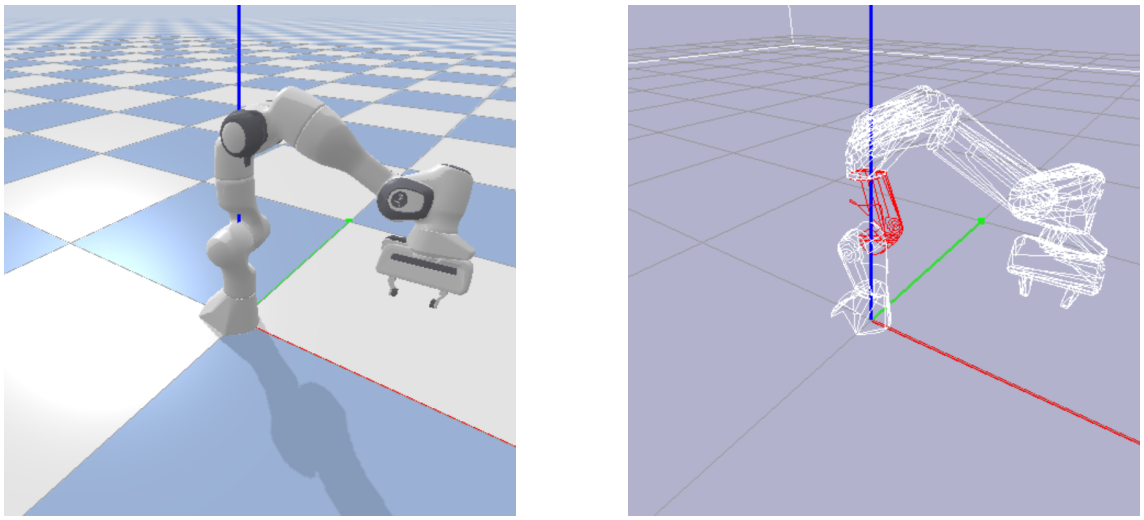
Technical Design

3.1 Project Scope

By design this project aims to create a generalisable robot learning algorithm which will have applications in multiple problem scenarios. The generalisability comes from the ability to provide a robot with whatever demonstrations are relevant to solving the desired task, without requiring any changes to the core system. However, in the interest of keeping this project focused we choose to implement and analyse specifically an agent for generalised object manipulation tasks focused around grasping and moving simple objects.

3.1.1 Environment specifics

For this project we conduct the experiments in a real time physics engine, called 'PyBullet' [9], available as an open source python package. As such the algorithms presented in this paper will be implemented in Python and the experimental results and evaluation will be conducted in this simulation space.
The specific robot arm used throughout this project is a simulation of the 'Franka Panda robot arm', which comes built in with PyBullet. This arm can be seen in Figure 3.1a.



(a) Normal view (b) Wire-frame view

Figure 3.1: Franka panda robot arm in PyBullet simulation

This simulation and arm have a number of intricacies which posed a steep learning curve near the start of this project. This subsection highlights a few of these. The first major hurdle was with the accuracy of the inverse kinematics system built into Pybullet. The initial inaccuracies were not a bug in the simulation, but a misunderstanding of the signature of some of the functions. The Franka panda robot arm has 12 individual joints. However, upon closer inspection it becomes apparent that 3 of these joints are not actuated and are fixed in place relative to the previous joint. As such the arm only has 9 degrees of freedom, while reporting all 12 joints. This issue is only apparent when using inverse kinematics to calculate the joint angles to reach a desired end effector position, since this function returns a list of length equal to the degrees of freedom of the arm. This means we get a list of 9 angles instead of 12. The function to control the arm motors, `pybullet.setJointMotorControlArray()` expects one angle for every joint. As such if we pass the list of joint angles directly, we end up trying to move some of the fixed joints, while leaving the top 3 joints unspecified, resulting in not moving them. While this is not difficult to fix, there are numerous inconsistencies in the signature of different functions, which are not clearly documented. Each of these issues slowing down development while it is debugged.

Another point of note specific to the simulation is how cameras are implemented. The camera's in Pybullet do not use the standard intrinsic and extrinsic matrix to convert world coordinate to pixel coordinates in the image. Pybullet instead uses a projection matrix and view matrix respectively. These broadly serve the same purpose with the view matrix capturing the position and orientation of the camera and the projection matrix capturing the specifics for how points are projected onto the 2D image. The most notable difference is that the projection and view matricies are 4×4 instead of 3×3, since they are applied to homogeneous coordinate points [10]. In this paper we will refer to the projection and view matrix as this is what our simulation uses. However, one could easily use a system which instead requires an intrinsic and extrinsic matrix instead.

In PyBullet, orientations are represented using quaternions, [6, 7] an explanation of which is provided in Section 2.4. Despite adding a fourth dimension, quaternions provide a big advantage over the alternative of Euler angles. This being that quaternions do not suffer from 'gimbal lock' [11, 12].

3.1.2 Collecting demonstrations

In order to generate the trajectories needed for the imitation learning system, a utility program is created to streamline the collection of demonstrations. This program generates trajectories by recording key frame poses throughout the demonstration. The demonstration can be copied by matching each of these key frame poses, with interpolation poses between them. This program allows for the user to control and move the end effector position and orientation in Cartesian world coordinates. Once the arm is positioned in the correct pose, the user can save the current pose as a key frame of the trajectory. From here they can move the arm again and save the next key frame. If a mistake is made, the user can revert the robot arm to the last saved key frame.
The trajectory is stored as a list of end effector poses, the reason for which will be discussed in Section 3.2.

The program is written to interface with a video game controller, so that the user can control the robot using an analogue input method such as the joysticks. This makes the program slightly more user friendly, as moving the joystick only slightly will move the arm very slowly and so allows for finer control.
In addition to controlling the robot, the program allows for camera rotation and zooming to better see the full robot pose, and can also enable a wire-frame view, visible in Figure 3.1b.

3.2 Encoding the problem

In this paper we plan to build upon existing ideas in the field while reducing the barrier to entry formed by using extremely resource intensive AI models. We intend to provide the robot with an example trajectory which successfully accomplishes the skill, in addition to some further context which allows the robot to generalise this trajectory to different environments.

Let us formalise the concept of demonstrations and trajectories, as this will be key in understanding how we can transfer a demonstration from the environment it was given in to other unseen environments. Firstly we decompose the state of the system at time t as the current state of the robot and the state of the environment.

$$s_t = (s_t^{(r)}, s_t^{(e)})$$

We choose to make this separation because the robot state is fully observable and within our control, while the environment is only partially observable and cannot be directly controlled. It can only be influenced by the robot.

3.2.1 Robot state

We have a number of options for how to represent the state of the robot. The most expressive approach is to record the angle of every joint of the robot. This uniquely defines an exact pose of the robot.
Another option is to encode the robot's state as the end effector position and orientation. This option is appealing since for the purposes of performing manipulation tasks, the end effector pose is our main concern. We likely do not care about how the robot achieves this end effector pose, so long as it does. For this reason storing only the end effector pose at each time step, and dynamically computing the joint angles to achieve this using inverse kinematics, appears to be a desirable choice.

However it is worth considering that in this case an end effector pose does not uniquely define the entire robot pose. This is because the end effector pose has 3 degrees of freedom for the position and 3 for the orientation (4 quaternion components however the sum of squares must equal 1, tying down the 4th value once 3 are already chosen). In contrast the robot used in this project has 9 degrees of freedom, as discussed in Subsection 3.1.1. This means that the inverse kinematics system of equations is underdetermined since we have in essence 6 constraints but 9 unknowns to satisfy them with. As such, the inverse kinematics system of equations has infinitely many solutions. What this means for us is that the end effector pose does not uniquely define an entire robot pose. There are in theory infinitely many robot poses which would result in the end effector reaching the position and orientation

we wanted it to.

Figure 3.2 demonstrates an analogous instance in 2 dimensions for easier viewing. In this system the target position is an (x,y) position and the orientation can be defined by a single angle θ. This means we have 3 degrees of freedom. Therefore, a robot arm with 4 or more controllable joints would create an underdetermined system. Figure 3.2 showcases just 3 possible solutions, but one can imagine how infinitely many may exist. One can also extrapolate this idea into 3 dimensions.

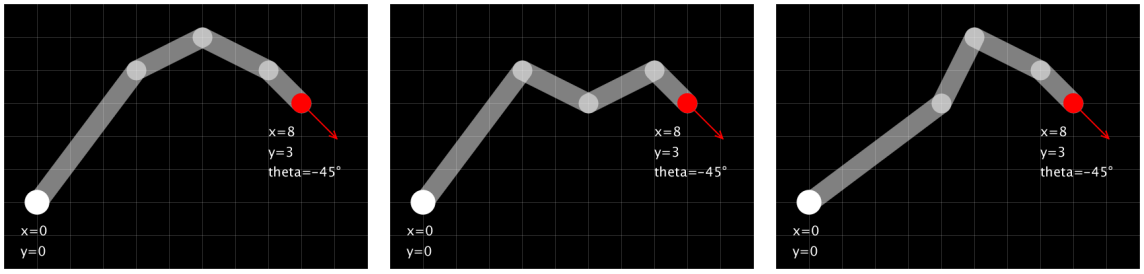


Figure 3.2: An underdetermined inverse kinematics system has multiple solutions

While this initially seems like a big problem, for our purposes we truly do only care that the end effector is in the position and orientation we specify. Any of the satisfying robot poses will do. Working with the end effector pose directly also simplifies a lot of our work later on. We will see in Subsection 3.3.2 that a lot of our calculations will involve the end effector pose. While we could compute this using forward kinematics if the demonstration instead stored joint angles, this would be an unnecessary extra step for no real benefit to the accuracy of the system. A way to use the underdetermined nature of the system to our advantage is explored further in Appendix A.

We may optionally decide to also include velocity information in addition to positional information. This would allow us to specify that the robot should have an exact position, but also describe how it should move in the next time step. While this may be useful for very specific tasks like throwing an object to hit an exact target; for the majority of tasks, velocity information is simply not necessary when we have the position for the next key frame already defined. If we were to save a key frame every time step, then the velocity information would be completely redundant, with velocity being directly computable as the displacement between the two key frames divided by the time step. In practice, for usability, we opt to not do this and only record key frames when deemed necessary by the user during the demonstration. While this does mean velocity information is not directly knowable, we decide that velocity information is not necessary for the simple tasks we are experimenting with in this paper.

One final thing to consider with how to represent the robot's state is the end effector gripper. Since the gripper arms are attached to the end effector, they are not defined by the inverse kinematics result. Moving the gripper arms has no effect on the pose of the end effector, and so any gripper arm positions would constitute a valid IK solution. Therefore, we need to add some information to our state which uniquely decides where the gripper arms are. We could use the angle of each gripper arm, however it is unlikely we would want the grippers at different angles, so we can simplify to a single angle which we mirror for both grippers. Simplifying even

further, we do not anticipate a situation where the gripper would need to be in an intermediate half closed state. We either want the gripper fully closed, applying a constant pressure to the item it is holding, so that the item does not fall out of its grasp; or we want the gripper fully open, so that it can drop the item it is holding. As such we can see that a single boolean value is sufficient to define the gripper state in our situation.

$$g \in \{0, 1\}$$

With these decisions in place we can define the robots state as an 8 vector, containing the position and orientation (as a quaternion) of the end effector, along with the gripper state.

$$s^{(r)} = [x_{eef}, y_{eef}, z_{eef}, a_{eef}, b_{eef}, c_{eef}, d_{eef}, g]$$

3.2.2 Environment State

We define the state of objects similarly to how we defined the end effector pose storing the 3D position and orientation of the object within the scene. Objects do not have an associated gripper, so an objects state is a 7 vector. The state of the whole environment is just the collection of all the object states within the scene.

$$s^{(e)} = \{[x_o, y_o, z_o, a_o, b_o, c_o, d_o]\}_{o=1}^{O}$$

We note that without a perfect model of the environment, the environment's state is not fully observable. In implementation we will not be able to capture the full environment state and instead choose to represent it differently entirely. For example capturing the environment state as an image of the scene using methods similar to those employed by P. Vitiello, K. Dreczkowski, and E. Johns [4]. For now we will continue in the purely theoretical sense, where the environment state can be fully observed.

3.2.3 Trajectories and Demonstrations

Given that we have defined the state of the system, a naive approach would be to define a trajectory as the list of states traversed at each time step of the trajectory.

$$\tau = \{s_t\}_{t=1}^{T}$$

We can now clearly see that a trajectory is a large $T \times D$ matrix where D is the dimensionality of the state. $D = 8 + 7 \cdot O$ in this idealised setting.

$$\tau \in \mathbb{R}^{T \times D}$$

This definition is naive for a number of reasons. Firstly we can see that a trajectory defined in this manor will include a large amount of unnecessary information. In completing a task, we will likely only need to move the robot arm to a few key positions to ensure the task succeeds. However this current definition saves a pose at every time step. We can reduce the dimensionality of a trajectory substantially if instead of saving the robot's state at every time step, we only save the relevant key frames. When executing this trajectory we will then interpolate the poses between each key frame, to recover the pose at each time step. We can now add as many key frames as are required to fully express how to complete the task, without including

unnecessary in between poses. We denote a trajectory as having K key frames.

Another issue with the current trajectory definition is that in addition to the environment state not being fully observable, it is not directly within our control. We cannot choose to change the state of the environment by magically moving an object. We can only influence the environment through the actions of the robot. As such, this approach still includes a large amount of irrelevant information. Furthermore, we want a trajectory to encode a skill. The necessary steps the robot needs to take to complete some task. This task can be completed regardless of the specific environment state and so the environment variables should not be a part of the trajectory. Conceptually, the trajectory encapsulates a skill independent of the environment it is within. The notion of influencing the trajectory based on the environment is considered more a transformation applied to the trajectory, than a component of the trajectory itself.

This is an important difference in definition to a demonstration. A demonstration is a human provided trajectory accompanied with some additional environment information. This additional information is what allows us to compare scenes and transform the trajectory. The trajectory itself does not include any environment specific information.

With these new definitions in mind we can see the trajectory is a $K \times D$ matrix where K is the number of key frame poses and D is only the dimensionality of the robot's state, $D = 8$.

$$\tau = \{s_t^{(r)}\}_{k=1}^{K}$$

A demonstration is a trajectory combined with some environment information, which for now we denote as Σ. Σ must correspond to the exact environment in which τ was presented by the human.

$$\Delta = [\tau, \Sigma]$$

3.2.4 Environment context

There are numerous ways we could choose to define Σ, each with their own benefits and drawbacks. The purpose of Σ is to provide some context as to how the world looked when the trajectory was given. We will then compare this to the environment at test time, and modify the trajectory into the new scene.

$$\tau' = f(\tau, \Sigma, \Sigma')$$

Where f is the main goal of this project. Some function, which we call the 'trajectory transfer function', capable of mapping the trajectory from the original environment to the new environment. How f is defined and implemented is explored thoroughly in Section 3.3.

If the environment were fully observable, then an easy choice for Σ would be the initial environment state.

$$\Sigma = s_{t=0}^{(e)}$$

This perfectly captures what the environment looked like when the trajectory was given by the human. It encodes all the information we need which justifies why the

human provided this exact trajectory to complete the task. Note we only need to store the initial environment state, there is no need to encode the state across all time steps. This is because we assume that the optimal method to complete a task is apparent from a single glance at where all the objects within the scene are. We assume the environment is static, and can only be influenced by the robot's actions itself. In this sense the robot and environment are a closed system. No external actors can influence the environment. If external actors could influence the environment then a single initial snapshot would no longer be sufficient.

This closed-system property is crucial even in the partially observable case. When the environment state is not fully observable, and we cannot know $s^{(e)}$ perfectly, we must make an observation to approximate it. As before, we conclude that if no external forces can be applied, then a single observation is enough. We just need to make sure that our observation is rich enough to infer all the information we need about the environment state.

For this project, we utilise methods presented by N. Di Palo and E. Johns in their implementation of 'Dinobot' [5] to form the basis of our environment context. In their paper they use an RGB-D image taken prior to the trajectory being given. This satisfies the requirement of an observation made at time $t = 0$. From this image, we are able to infer the position of objects within the image. Dinobot uses a first person, wrist mounted camera to capture the RGB-D image. This is in contrast to other papers which use a fixed third person camera [4]. It is important that the object of interest which we will be interacting with is within the image, as the end effector will need to align relative to this object. If the object is not visible within the picture, then the robot will align relative to a different object it can see. Unless this object and the correct object are in the exact same position relative to each other, then this would cause the task to fail. While a third person camera capturing the entire environment would guarantee the object of interest is visible, using a wrist mounted camera leads to lots of nice properties discussed in Subsection 3.3.2. In their paper, N. Di Palo and E. Johns combat this problem by placing the robot in a known initial pose with the end effector and attached camera placed high pointing down onto the scene. This captures a wide view which should encompass any relevant objects.

This however enforces a restriction that the demonstration must always be given from the same initial starting position. This may not be an issue if the initial position is chosen well, but it is possible to create an environment where important information is occluded when viewed in this initial position. Consider Figure 3.3. The task is to pick up the block from underneath the bridge. However, from the initial position in red, the robot is unable to see the object of interest, making it impossible to align correctly. The green line shows an alternative initial position where the object is no longer occluded.

Whatever initial pose we choose for the robot, we could construct an example where the object of interest is not visible. The solution to this is to not fix the initial pose, but allow it to be dynamically chosen by the human giving the demonstration. All we need to do is include the initial pose in the environment context, along with the image it captures. Now rather than the initial pose being fixed in the code, we can use the initial pose defined in the demonstration instead.

$$\Sigma = [I^{(RGB-D)}, s^{(r)}_{t=0}]$$

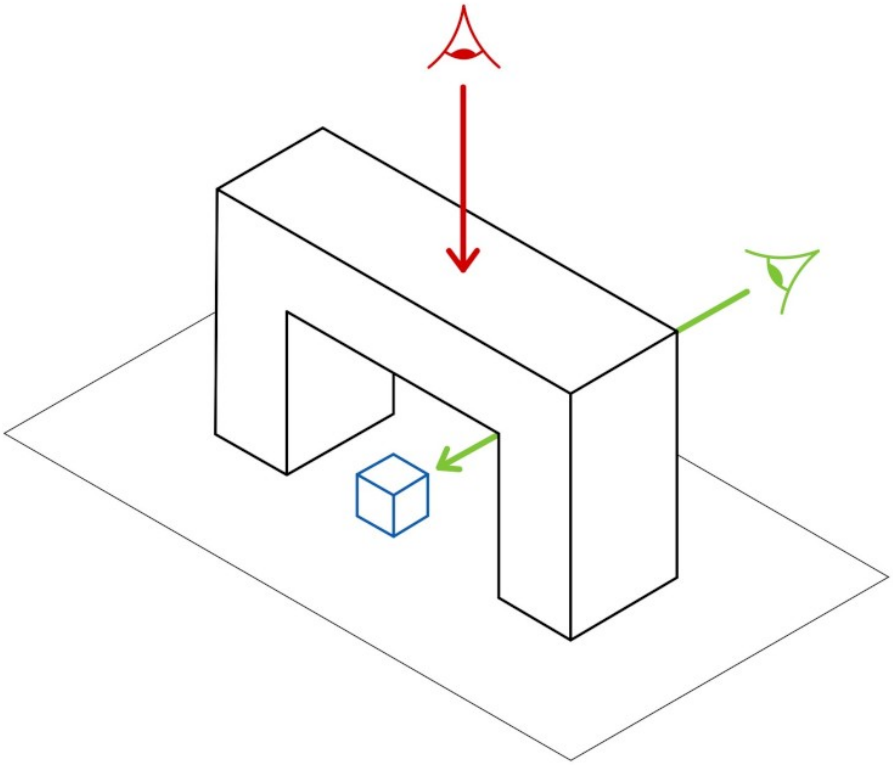


Figure 3.3: Example environment where initial position cannot view the object of interest

In practice we make a few modifications to simplify the implementation. As will be seen in Subsection 3.3.2, the purpose of knowing the initial pose is to know the extrinsics of the camera which took the image. If we know the position and orientation of the camera, we can compute world coordinates from the pixel coordinates of objects in the image. As such we decide to store the extrinsics directly, instead of computing them from the initial pose. As mentioned in Subsection 3.1.1, in Pybullet the extrinsics of the camera are captured by the view matrix, which we denote V.

The other implementation difference is to store the depth information separately to the RGB data. This is because the RGB pixels are integers between 0 and 255, while the depth information is stored in 'Normalised Device Coordinates' [13]. These are float values in the range 0 to 1 where 0 represents an object on the near clipping plane and 1 represents an object on the far clipping plane. Mapping the depth information into integers of the range 0-255 and including as a 4th channel of the RGB image proved to cause too much loss in precision when recovering the true values, leading to very poor performance.

With these differences in mind we can define the final form of the demonstrations:

$$I^{(RGB)} \in (\mathbb{N}_{[0,255]})^{H \times W \times 3}, \qquad I^{(D)} \in (\mathbb{R}_{[0,1]})^{H \times W}, \qquad V \in \mathbb{R}^{4 \times 4}$$

$$\Sigma = [I^{(RGB)}, I^{(D)}, V]$$

$$\Delta = [\tau, I^{(RGB)}, I^{(D)}, V]$$

Where W and H are the width and height of the image captured. These values are not too important provided they remain constant for all demonstrations. In this paper we choose $W = H = 1024$ pixels.

3.3 Implementation

3.3.1 Demonstration Selection

The first step of implementation comes before we can even consider the trajectory transfer function. We first need to consider its inputs. f takes as parameter the task trajectory τ, the original environment context Σ and the context of the new test time environment Σ'. While Σ' can be directly observed from the test time environment, τ and Σ are dependent on the demonstration we choose to follow.

We want the system to be able to learn multiple tasks independent of each other and apply the correct task to a given scene. We do not want the human to have to tell the robot what it should do with the objects. The robot should be able to infer the task from the objects alone. More specifically, from the environment observation Σ'. The robot will have access to a pre-trained corpus of demonstrations. The 'One-shot' nature of this system refers to each task only requiring a single demonstration to be learned. However, to teach multiple tasks, a new demonstration for each task is still required. Each of these demonstrations has associated with it, the environment context Σ. Deciding which task to complete in the new environment is a matter of comparing the context image of each demonstration, to the live image the robot takes upon being presented with the environment. For this purpose, only the RGB image is used, the depth information and view matrix are ignored for now.

In order to compare the images, N. Di Palo and E. Johns opt to embed both images using a vision transformer created by Meta (formerly known as Facebook) called 'Dino' [14]. The transformer produces a 768-vector, and we can directly compare the embedding vector of different images using cosine similarity. Whichever embedding has highest similarity to the live embedding, corresponds to the context image being most similar to the live image. The hope is that the transformer extracts useful information regarding the relevant parts of the image, such as the general make-up of objects within the scene, while ignoring less important information like the positional information of objects.

We can expect the similarity between the embeddings to be greatest when the corresponding images contain the same objects. However, this system naturally extends to generalising to unseen objects as well. If no such demonstration exists on the specific object in the live scene, then we can still select whichever demonstration is closest. Since we directly compare not the images themselves, but the image embeddings from the vision transformer, this translates to selecting the image with the object which the transformer sees as the most similar. For example the robot had a demonstration to pick up a can, it is likely that when faced with a bottle in the current scene that the system would select the demonstration of the can. This is because the objects appear similar as they are both cylindrical items, and the robot would have to interact with them in similar ways, by grasping around the cylinder. As such we can conclude that the demonstration of the can is the closest one to working with this new object. The hope would be that these two objects are

similar enough that the same demonstration would be able to successfully complete the task on this new object.

3.3.2 Trajectory Transfer

Now that we have the necessary parameters, we can begin implementing the trajectory transfer function, $f(\tau, \Sigma, \Sigma')$. In the previous subsection we found a demonstration which shows us how to manipulate the object in the scene (or a similar one). Now we need to consider the pose of the object to transform the trajectory. During the demonstration, the object was at some position with some orientation in world space $\{X_{demo}, \Theta_{demo}\}$. When the object is at this exact position and orientation, we know how to complete the task by executing the demonstration trajectory, τ. In effect we know that for the specific case where the environment has not changed:

$$f(\tau, \Sigma, \Sigma) = \tau$$

However, generally in the live scene the object may have moved to some different position and orientation $\{X_{live}, \Theta_{live}\}$. We define the offset between the two poses as the translation and rotation which maps the demonstration pose onto the live pose. Formally [1]:

$$M = X_{live} - X_{demo} \qquad R = \Theta_{live}\Theta_{demo}^{-1}$$

We need to modify the trajectory to account for this offset, getting τ'. Recall that τ is just a list of end effector positions and orientations through time. So to calculate τ' we add the translation and rotation offset to each pose of the trajectory.

$$\tau = \{P, O\}_{t=1}^{T} \qquad \tau' = \{P + M, RO\}_{t=1}^{T}$$

Since we are currently working in a simulation, the environment state is fully observable. We can query the simulation to get the exact position and orientation of the object of interest during the demonstration and record this along with the demonstration trace. We can then compare this to the environment state in the live scenario and compute the exact offset between the two object's and adjust the demonstration trace by this amount.

However, this information is only afforded to us because we are working in a simulation. We want our solution to be deployable onto real world robots without further modifications. In a real world implementation we would not be able to perfectly know where the object is. We only have access to the observations made by the camera attached to the robots end effector. As such we limit ourselves to a partially observable environment and use the observations alone for our calculations. We will use the observations to produce an estimate of the object's true position and orientation, using this in place of the true value which we no longer have access to. This is why the trajectory transfer function takes as parameter observations of the environment Σ and Σ', not the true environment state $s_{t=0}^{(e)}$ and $s_{t=0}^{\prime(e)}$.

We note that while we chose to use an estimate of the object's world coordinates to compute the offset, this is not absolutely necessary. Our true goal is to align the end effector so that relative to the object, it is in the exact same location as in the demonstration. Once this is the case we can compute the same offset as before, but this time using the end effector's position and orientation. We denote the end

[1] For an explanation of the formula for R, refer to Section 2.4

effector's pose in world coordinates at the start of the demonstration, and during the live scene once alignment has completed as $\{E_{demo}, \Phi_{demo}\}$ and $\{E_{live}, \Phi_{live}\}$ respectively. Given this then we can compute M and R without requiring the position of the object at all.

$$M = E_{live} - E_{demo} \qquad R = \Phi_{live}\Phi_{demo}^{-1}$$

It is important to remember that here E_{live} is the end effector position only once it has aligned to the object. Formally, E_{live} is the end effector position in world coordinates, such that E_{live} and E_{demo} are equivalent when viewed from the reference frame of the object in the live and demo scene respectively.

$$E_{live}^{(obj_{live})} = E_{demo}^{(obj_{demo})}$$

By using this approach we completely remove the need to estimate the object's position and orientation. We only need to compute the end effector's pose, which is trivial using forward kinematics. Furthermore it is relatively easy to decide when the end effector is aligned, since we can just compare the positions in the object reference frame. We know that if the end effector is in the same position relative to the object, then the object is in the same position relative to the end effector. The converse statement is also true.

$$E_{live}^{(obj_{live})} = E_{demo}^{(obj_{demo})} \iff X_{live}^{(eef_{live})} = X_{demo}^{(eef_{demo})}$$

This is why using a wrist mounted camera is useful. Since the camera is attached to the end effector, if the object is in the same position relative to the end effector, it will be in the same position in camera space. And if this is the case it will be in the same position in image space. This means that if the object is in the same position relative to the end effector, it will be in the exact same position in the image taken by the camera.

$$X_{live}^{(eef_{live})} = X_{demo}^{(eef_{demo})} \implies X_{live}^{(image_{live})} = X_{demo}^{(image_{demo})}$$

If this is the case then the image captured in the demonstration and live scene will be exactly the same (barring any background noise such as extra objects). This makes it easy to detect when the end effector is aligned since we just move until the end effector until the live and demonstration image match within a suitably low tolerance. The downside to this method will be more clear after discussing Subsection 3.3.5. While we can still compute the difference between the position and orientation of the object in image space, this results in a translation in a unit of pixels rather than meters which is used by the simulation. This means that while it is easy to know when the end effector has aligned, it is difficult to know by how much we should move to align. This method also makes it much more cumbersome to include depth information, since the depth data is not computed in a unit of pixels, but rather holds the actual depth from the camera to the object in normalised device coordinates. As a result, the x and y coordinates of the object have a different unit to the z coordinate. While these are all issues which are theoretically solvable, it is much easier and debatably more elegant to perform all the calculations in a single reference frame, by estimating the pose of the object in world space. As such this will be the method used going forward.

3.3.3 Using key points to approximate position

As previously mentioned, our goal is to align the live and demonstration image so the object looks the same relative to the end effector camera. It is difficult to work with the image directly, since working with the colours of pixels leaves the system sensitive to lighting conditions and noise in the image. Instead we would like to use the image to estimate the position and orientation of the object in world space, and use this for our calculations instead. One approach could be to identify the bounds of the object using computer vision techniques, to work out exactly which pixels represent the object. This is relatively simple since the object is almost certainly a different colour to the background. By knowing which pixels belong to the object we can estimate the centre of the object and use this as the position of the object in image space.

For all figures in this section we will assume the left image is the live image and the right image is the demonstration image.

Figure 3.4 shows how the centre of mass in the live image is further left than it was in the demonstration image. Therefore the object is too far to the left compared to the demonstration. To fix this we would want to move the end effector camera to the left, so the centre of mass in the live image would move to the right, aligning with the demonstration image.

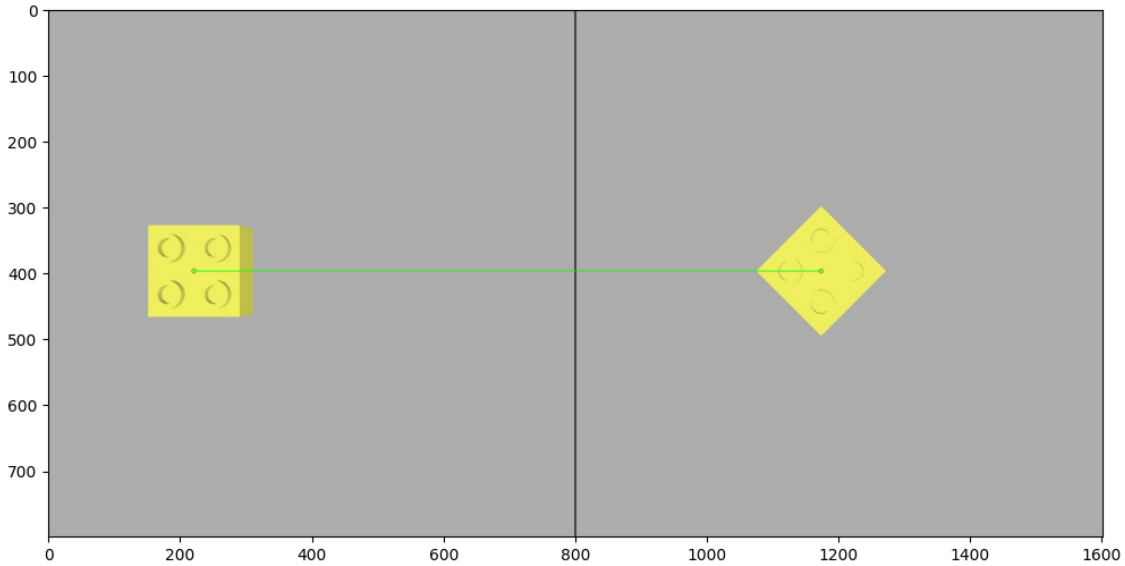


Figure 3.4: Centre of mass matching

However, note that while a single point is enough to compute the translation offset between the object in different scenes, it is not sufficient to compute the orientation offset. By reducing the object in the image to a single point, we have estimated its position, but have no information about its orientation. This is a problem for us since we want the system to be robust to not only moving the object, but also rotating it. Some objects may behave very differently in different orientations if they do not have rotational symmetry. As such we need a way to estimate the orientation of the object in both images.

In order to achieve this we cannot reduce the object to a single central point,

but we can reduce it to a collection of points. Having multiple points which match in the two images can allow us to identify if a translation or rotation have occurred. Figure 3.5 shows how an ideal set of matches allows us to infer both a translation and rotation between the object in the images. In this case we can see the block needs to move to the right of the frame, but now also needs to rotate approximately 45° anticlockwise (a positive rotation according to the right hand rule) about the vertical axis.

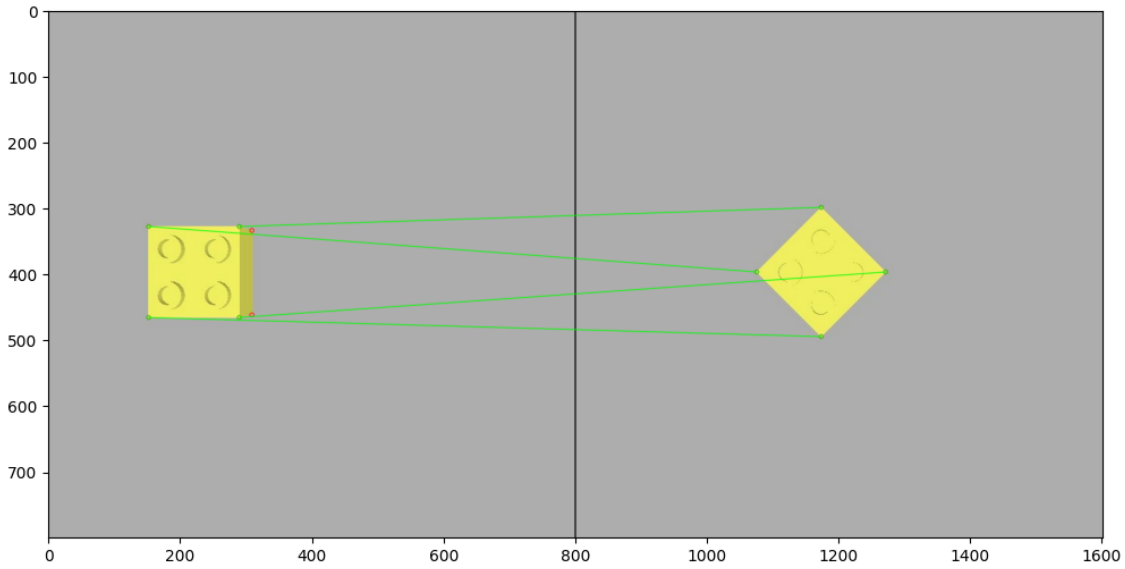


Figure 3.5: Ideal keypoint matching

Note that this is only one way the keypoints could be matched. Figure 3.6 shows another way we may choose to match the keypoints. In this case the live object should move to the right and rotate approximately 45° clockwise (a negative rotation according to the right hand rule) about the vertical axis, in order to align with the demonstration object. While having multiple possible ways to match the keypoints may seem like a problem, this is merely a consequence of the object in question having rotational symmetry. It does not matter which way we rotate the object, because it will behave the same, whichever side we are facing, and each side is indistinguishable from each other. As such, for our purposes it would not matter which matching and subsequent transformation we chose, since in either case the object is aligned to the demonstration image. In practice this situation would not likely occur, since we will have many more keypoints, and the keypoints will very likely not be evenly spaced as in this ideal example.

We can also see that in both figures, the live image contains two additional keypoints marked in red. These keypoints could not be matched to any in the demonstration image so are ignored when calculating the translation and rotation to align the objects. In this case it is because these keypoints become occluded when we view the object in the demonstration image.

These figures are designed to give an intuition for how we can use keypoints to find the desired end effector offset. In these figures we have considered ideal keypoints where the keypoints are extracted on useful points of the image, namely the object corners, and where the matching is perfect with no mistakes. These keypoints have been placed by a human for the purposes of demonstration. Obviously for the complete system we want the keypoint extraction and matching to be automated.

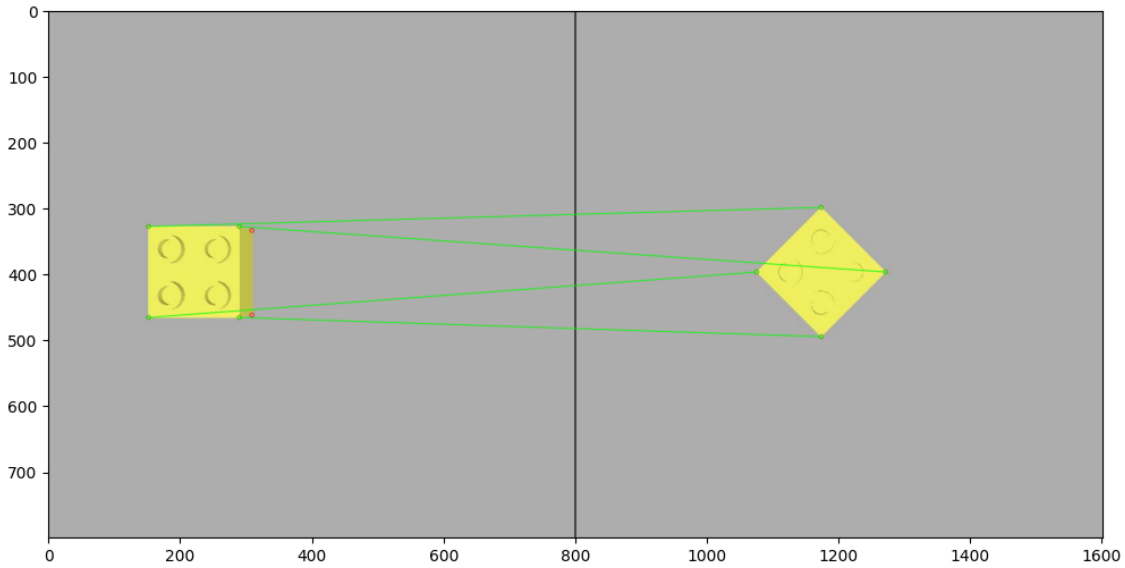


Figure 3.6: Alternative ideal keypoint matching

How we achieve this will be explored next.

here onwards is WIP

3.3.4 Automated keypoint extraction and matching

Keypoint extraction is a popular and well studied problem in the field of computer vision. As such we have many options at our disposal. Our goal is to identify the pixels where the image changes significantly. These may be where ...TODO

3.3.5 Calculating end effector offset

Now that we have a list of matched keypoint pairs, we can begin to calculate the offset between the object in the live and demonstration image. Fortunately there exists an algorithm for computing the translation, rotation and scaling, which best maps a set of points onto another set of points. This algorithm is called the 'Kabsch-Umeyama algorithm' [15]. The algorithm takes in two sets of points, a reference set P, and a test set Q. It computes a translation t, a rotation matrix R and a scale factor s, such that when these transformations are applied to Q, yielding Q', the root mean squared distance between P and Q' is minimized. For our purposes we make a number of important modifications to the original algorithm to better fit our use case.

As mentioned above, we decide to convert all keypoints to world coordinates first. By doing this we can more easily include the depth information of the image. Additionally, this will result in the translation computed already being in world coordinate units, meaning no further calculation is needed to calculate how far to move the end effector.

... The first step of the algorithm will look familiar to something we tried in Subsection 3.3.3. The first step is to compute the centroids of the two sets of points. This is just the average position of all the points in the set. This acts like the centre of mass of the point set. With sufficiently many and well spaced keypoints in our set, this centre of mass will be approximately equal to the centre of mass

Algorithm 1 Kabsch-Umeyama algorithm

Input: $P = [p_1, p_2, \dots p_N], \quad p_i \in \mathbb{R}^M$
$Q = [q_1, q_2, \dots q_N], \quad q_i \in \mathbb{R}^M$

Assert: $|P| = |Q| = N$

1: **procedure** KABSCH-UMEYAMA(P, Q)
2: $\overline{p} \leftarrow \frac{1}{N} \sum_i^N p_i$
3: $\overline{q} \leftarrow \frac{1}{N} \sum_i^N q_i$
4: $\sigma_P^2 \leftarrow \frac{1}{N} \sum_i^N |p_i - \overline{p}|^2$
5: $M \leftarrow \frac{1}{N} \sum_i^N (p_i - \overline{p})^T (q_i - \overline{q})$ *// Covariance Matrix*
6: $U, E, V^T = \text{SVD}(M)$ *// Single Value Decomposition*
7: $c \leftarrow \frac{\sigma_P^2}{trace(E)}$
8: $R \leftarrow U \cdot V^T$
9: $t \leftarrow \overline{p} - cR \cdot \overline{q}$
10: **for** $i = 1 \dots N$ **do**
11: $q_i' \leftarrow t + cR \cdot q_i$
12: **end for**
13: **output** t, R, c, Q'
14: **end procedure**

we computed earlier by considering all pixels of the object. However, the Kabsch-Umeyama algorithm does not use compute the translation as ...

We make a number of modifications to the algorithms which reflect specifics of our use case. The first major difference is in computing the rotation matrix. As can be seen in line 10 of Algorithm 2, we include an additional matrix S in the construction of R. The purpose of this is to prevent reflections in our transformation. In the original Kabsch-Umeyama algorithm, any transformation to map Q onto P is allowed. This includes translation, rotation and scaling, but it also implicitly includes reflections. The reflection is not explicitly computed, rather it is hidden away inside the rotation matrix R. In the original algorithm it is possible for R to have a determinant of -1. If this is the case then the matrix actually represents a rotation and reflection combined. For our purposes we do not want to allow reflections since we cannot reflect the object by moving the end effector, we can only rotate around it. In order to prevent reflections we just need to negate the last row of V^T (or the last column of U) before computing the dot product to find R. To avoid needing to recompute R if we notice it has a negative determinant, we can find the determinant before we compute R, using the fact that $det(A \cdot B) = det(A) \cdot det(B)$. If this determinant product equals -1 then the determinant of R will be -1 and so we need to negate the last row of V. This is achieved with the matrix S. This matrix will be set so that it is the identity matrix if the determinant of R is already going to be 1, having no effect. But if the determinant of R is going to be -1, then we set the bottom right element of S to be -1, which in the dot product calculation will have the effect of negating the last row of V.

Another noticeable difference between Algorithm 1 and Algorithm 2 is that in our version we do not need to compute the scale factor c or the transformed points

Q'. We do not compute Q' simply because we don't actually need these points, we just want the transformation so that we can apply it to the end effector. However, the reason we don't compute a scale factor is more subtle. It is the same reason we do not allow reflections in the rotation matrix, we cannot magically scale points by moving the end effector. The scale factor would allow us to scale all points in Q further from $\overline{q}$. However, this would correspond to physically making the object larger or smaller within the world. This is not something we can do by moving the end effector. Moving the end effector closer to or further from the object, will slightly scale how large it looks in the captured image due to perspective projection, however, this movement would also affect the depth from the camera to the object, impacting the Z coordinates of the key points in the image. As such we do not allow the robot to consider scaling the object as part of the transformation. If the end effector is too close to or too far from the object, then this is handled through the depth information mismatching, which would compute a movement in the Z axis to fix this.

While this does limit the system's generalisability to novel objects which are different sizes compared to demonstration objects, i simply dont care

The final modification we make is in how we compute the translation. ...

Algorithm 2 Modified Kabsch-Umeyama algorithm

Input: $P = [p_1, p_2, \dots p_N], \quad p_i \in \mathbb{R}^M$
$Q = [q_1, q_2, \dots q_N], \quad q_i \in \mathbb{R}^M$

Assert: $|P| = |Q| = N$

procedure MODIFIED-KABSCH-UMEYAMA(P, Q)
  $\overline{p} \leftarrow \frac{1}{N} \sum_i^N p_i$
  $\overline{q} \leftarrow \frac{1}{N} \sum_i^N q_i$
  $M \leftarrow \frac{1}{N} \sum_i^N (p_i - \overline{p})^T (q_i - \overline{q})$ *// Covariance Matrix*
  $U, E, V^T = \text{SVD}(M)$ *// Single Value Decomposition*
  $S \leftarrow I_{M \times M}$ *// $M \times M$ Identity matrix*
  if $det(U) * det(V^T) == -1$ **then**
    $S_{M,M} \leftarrow -1$ *// Set the bottom right corner element to -1*
  end if
  $R \leftarrow U \cdot S \cdot V^T$
  $t \leftarrow \overline{p} - \overline{q}$
  output t, R
end procedure

One final nuance with using this modified function is what order to pass the parameters in. We recall that the function computes the transformation which maps Q onto P. Therefore it would seem logical to pass the demonstration points as P and the live points as Q. This tells us how to move the live points so that they line up with the demonstration points. However, a subtlety is that we cannot directly move the live points by this transformation. We move the end effector and the camera in order to affect where the points will be in the next iteration. If we wish the points to move to the left of the image, then we need to move the camera

to the right. Similarly to rotate the points clockwise about the Z axis, we rotate the camera anti-clockwise about the Z axis. This means after computing t and R, we actually move the end effector by $-t$ and rotate it by R^T. It just so happens that $-t$ and R^T are exactly the values returned from the function if you swap the inputs, computing the transformation from the demonstration points onto the live points. As such this is the order we choose to pass the parameters, to avoid needing additional calculations.

Chapter 4

Evaluation

In this section, we evaluate the performance of our algorithm against a custom test suite. Our evaluation consists of two main tests designed to assess different aspects of the algorithm.
In the first test we examine how well our modified Kabsch-Umeyama algorithm handles varying levels of noise helping us understand its stability and robustness to perform well even in noisy environments.
The second test compares different key point matching algorithms by using them in a full run of the system and measuring their performance under identical environmental conditions. Following this test we will determine the best algorithm for use in our project, finalising the implementation design.

4.1 Sensitivity to noise

In this test we want to understand how robust our modified Kabsch-Umeyama algorithm is to noise in the coordinates it receives. The hope is that small deviations to the coordinate inputs produce very small changes in the output translation and rotation matrix. If this is the case then the algorithm is robust to noise, and is more likely to produce good results when used in our system.

In order to conduct this test we create a test suite of 5 diverse objects and an accompanying demonstration for each one. We then manually mark an ideal set of key points in the demonstration image. These key points are recorded as a list of (x,y) pixel coordinate pairs. We also make sure to record the position and orientation of the object in the demonstration. We then place this same object in a different pose in the environment and save the live image from the robot in this case. Again making sure to note down the position and orientation of the object in this new scene. We then again manually mark the same key points but in this new image. It is important that we mark corresponding key points, so if in the demonstration we marked the corners of a cube, we must also mark the corners in the live image. The pixel location of these corners will just have moved. Table 4.1 shows the transformation between the demonstration and live object pose. We also disclose how many keypoints were manually marked for each object.

Now we can pass the manually marked key points in the demonstration and live image to our modified Kabsch-Umeyama algroithm. This should output the exact translation and rotation between the object from the demonstration to the live scene, within a small tolerance of floating point accuracy. The purpose of using

Object	Translation applied	Rotation applied	Number of key points
Lego	[-0.1, 0.05, 0]	[0, 0, $\frac{\pi}{3}$]	4
Mug	[0, 0.05, 0]	[0, 0, $\frac{\pi}{6}$]	16
Ball	[]	[]	
Block	[0.1, 0.15, 0]	[0, 0, $-\frac{\pi}{6}$]	
Domino	[0.05, 0.07, 0]	[0, 0, -$\frac{\pi}{4}$]	

Table 4.1: The true transformation between demo and live for each object in test suite

human provided, ground truth key points is to control any additional noise in the test. If we used our keypoint matching algorithm then it would be unclear how much error came from our randomly added noise, or from mismatches in the key point algorithm. As such we gain clearer results by using our ground truth key points.

With the method defined, we now add some noise to the ideal key points before passing them to the algorithm. We wish to compare how far the new output deviates from the true output when the noise is added. Since we add random noise, we conduct multiple runs and compute the average.

In this first set of tests the noise added is calculated as a random 3D unit vector multiplied by some random magnitude. The magnitude will be drawn from a uniform distribution, in the range $[a, b]$. Here a and b are used to control the upper and lower bounds on how much noise can be added. This allows us to perform multiple tests and compare the effects of increasingly large deviations.
When computing the random 3D unit vector, we take care to use an "equal area projection of the sphere onto a cylinder" [16, 17, 18]. This allows us to choose a point uniformly from the unit sphere, without experiencing a bunching of points at the poles. This random noise is applied to the world coordinates after they have been computed from the pixel coordinates. This gives us a solid grasp as to just how much noise can be present before the algorithm produces unsatisfactory results.

In the second set of tests we change how we apply the random noise to better emulate our system. Since the pixel to world coordinate calculations use the exact view matrix of the camera, the only errors in this part of the system are the result of very small floating point inaccuracies. The source of the error will be predominantly a result of the key point matching. As such in this second round of tests, we add noise to the pixel coordinates of the key points, before converting them to world coordinates. This is more representative of the type of error we will encounter in this system, and so proves a more reliable result.

4.2 Comparison of keypoint algorithms

We discussed in Subsection 3.3.4 that there are multiple different algorithms we could use for our system. Any algorithm which can extract and match keypoints can be used. In this section we analyse a few different aproaches

Chapter 5

Future Work

The system designed in Chapter 3 forms a solid core concept which demonstrates the capabilities of classical algorithms to create a one-shot imitation learning agent without the dependency on AI technologies. However, as with any project, there remains some aspects which one could consider improvements to the design, left unimplemented. These improvements are simply not the core focus of the project, and given time constraints, priorities had to be chosen. These improvements were largely not prioritized because they do not explore interesting aspects of the system. This chapter details further improvements to the system which we would have liked to implement in the future.

5.1 Multi-stage Tasks

We discussed multi-stage tasks in Chapter 2 where we defined them as a task in which the object of interest changes throughout the execution of the task. When the object of interest changes we need to align the end effector relative to the new object of interest, as though this sub-task was the entire demonstration and we were starting the system from here.

It is clear from this description how the current system could be extended to accommodate multi-stage tasks. During a demonstration we would need to automatically capture a new environment context Σ_n whenever the object of interest changes. We decided earlier that this occurs whenever the gripper state changes. Then during test time, we will align the end effector to compute $\tau' = f(\tau, \Sigma_1, \Sigma_1')$. After aligning we execute τ' until the moment the gripper state changes. At this point we capture a new live environment context Σ_2'. We would re-run the trajectory transfer phase, aligning the current live image with the next stage's demonstration environment context. $\tau'' = f(\tau, \Sigma_2, \Sigma_2')$. We make sure to start executing τ'' from the key frame step we stopped at in τ'. This system could repeat as many times as needed for a n-stage task.

The reason we elected not to implement this feature was because it does not improve the core concept of this project. Undoubtedly this feature would make the system more widely applicable. However, this feature is not fundamental to the idea described in this paper. The limited time on this project was better spent improving the accuracy of the key point matching algorithm, and reducing numerical inaccuracies in the modified Kabsch-Umeyama algorithm. The fact that the entire design for this feature was adequately explained in the single above paragraph, shows

that it is not an interesting problem to solve. It is merely a change to the specifics of the code, it is not an exploration of the concepts described in this paper. With more time this would definitely be a worth while feature to add. Alas given the time constraints of this project, sacrifices needed to be made.

5.2 Segmentation Map

A potential method to improve the accuracy of our system is to incorporate a segmentation map. A segmentation map is an image where every pixel is assigned an ID number based on some categorisation. For our purposes this could be which object the pixel belongs to. The advantage that this offers us is to prune any key point matches between different objects. If a key point in the live image is at a pixel belonging to object 1 in the segmentation map, then it should not map to a key point in the demonstration image, unless that key point location also belongs to object 1. Note that the live and demonstration images each have their own segmentation map. This prevents erroneous key point matches to the wrong object, most commonly to some background noise objects. The Pybullet simulation provides a perfect segmentation map when a camera image is captured. This is already available to us and could be easily incorporated to the key point matching algorithm. However, this segmentation map is a result of working in a simulation. In order to deploy this system on a real world robot arm, we would need some other computer vision system to produce a segmentation map.

We have seen in Section 4.1 that the Kabsch-Umeyama algorithm is very sensitive to inaccuracies in the key point matching. Removing matches which are categorically incorrect may reduce this sensitivity. However, in practice it is difficult to tell how many inaccurate key point matches actually match to completely different objects. As a result it is unclear how much of an improvement this would make to the system. We hypothesise that this change would allow us to remove a small number of obviously incorrect matches. This would make the system more accurate in computing the exact offset between the demonstration and live object. However, due to the seemingly low frequency of these incorrect object matches, it is unlikely that tasks outright fail as a result of a few mismatches. Therefore we hypothesise while this would improve the accuracy of inference, it would likely not lead to more tests passing.
This is however only a hypothesis, and something which would be interesting to explore in future work.

5.3 Pre-compute Embeddings

A small improvement to reduce the amortized memory requirements of this system would be to pre-compute the embeddings of each image and store these as part of the environment context. By doing this we only need to use the Dino vision transformer to embed the image once when the demonstration is recorded. This removes the need to use any artificial intelligence components during inference time. While this is nice in principal it trades off reducing runtime memory requirements with increasing long term storage requirements of each demonstrations. This may or may not be preferable depending on how many demonstrations are saved for a particular agent. Pre-computing embeddings would be a very easy change to the

system. It has not been implemented because it simply does not provide any useful impact on the performance of the system. As such it was an incredibly low priority change which sadly there was not time to accommodate.

5.4 Image Descriptor

An arguably better change would be to remove the need for any artificial intelligence components at all. This would bring the project to its logical conclusion. Currently the system uses the vision transformer to embed images into a vector representation which can be compared. It is therefore clear to see that we could exchange this transformer with a classical algorithm to embed an image into a feature vector. One such algorithm is the 'Histograms of Oriented Gradients' (HOG) [19]. The design is very similar to SIFT descriptors (discussed in Subsection 3.3.4, except that it creates a representation of the entire image, not just the local area around a key point. Hog has been shown to be suitable in use for object detection [20] and pose estimation [21].

The reason this modification was not implemented is that it is a much larger change than it may initially seem. The Dino vision transformer has been specifically trained for semantic key point detection. As such the embeddings it produces specifically encode some notion of the key points within an image. This is why images of different objects with similar interactive properties have similar embeddings (for example, a can and a bottle are both cylindrical objects). This is not a feature immediately available to us by using HOG. Furthermore, we want our embedding to encode just the type of objects within the scene, not their positions, since positional information is handled entirely within the trajectory transfer stage. This is not something HOG can directly achieve since it is a concatenation of smaller scale cells in the image. If an object has moved from one cell to another, the HOG descriptor will be different. As a result we would need to design a more complex comparison function, which can account for differences in HOG descriptors due to a object position. This function would need to be able to output a high similarity if the objects described by the HOG descriptor are similar, even if the overall HOG descriptors are quite different. While this is certainly achievable, it is quite a complex task which is not possible within the time constraints of this project.

Chapter 6

Conclusion

Bibliography

[1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018.

[2] Feiyang Wu, Zhaoyuan Gu, Hanran Wu, Anqi Wu, and Ye Zhao. Infer and adapt: Bipedal locomotion reward learning from demonstrations via inverse reinforcement learning. 2023. URL https://arxiv.org/abs/2309.16074.

[3] Ajay Mandlekar, Soroush Nasiriany, Bowen Wen, Iretiayo Akinola, Yashraj Narang, Linxi Fan, Yuke Zhu, and Dieter Fox. Mimicgen: A data generation system for scalable robot learning using human demonstrations. 2023. URL https://arxiv.org/abs/2310.17596.

[4] Pietro Vitiello, Kamil Dreczkowski, and Edward Johns. One-shot imitation learning: A pose estimation perspective. In *Conference on Robot Learning*, 2023.

[5] Norman Di Palo and Edward Johns. Dinobot: Robot manipulation via retrieval and alignment with vision foundation models. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2024.

[6] Sir William Rowan Hamilton. On quaternions, or on a new system of imaginaries in algebra, 1844-1850. URL https://www.emis.de/classics/Hamilton/OnQuat.pdf.

[7] James Diebel. Representing attitude: Euler angles, unit quaternions, and rotation vectors, 2006. URL https://www.astro.rug.nl/software/kapteyn-beta/_downloads/attitude.pdf.

[8] Leonhard Euler. General formulas for any translation of rigid bodies, 1776. URL https://www.17centurymaths.com/contents/euler/e478tr.pdf.

[9] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning, 2016-2021. URL https://pybullet.org. [Accessed on 2023-11-23].

[10] Tom Dalling. Explaining homogeneous coordinates & projective geometry, 2014. URL https://www.tomdalling.com/blog/modern-opengl/explaining-homogenous-coordinates-and-projective-geometry/. [Accessed on 2024-06-12].

[11] Jonathan Strickland. What is a gimbal – and what does it have to do with nasa?, 2023. URL https://science.howstuffworks.com/gimbal.htm. [Accessed on 2024-06-12].

[12] Evan G. Hemingway and Oliver M. O'Reilly. Perspectives on euler angle singularities, gimbal lock, and the orthogonality of applied forces and applied moments. *Multibody System Dynamics*, 44(1):31–56, 2018. ISSN 1573-272X. doi: 10.1007/s11044-018-9620-0. URL https://doi.org/10.1007/s11044-018-9620-0.

[13] Nathan Reed. World coordinates, normalised device coordinates and device coordinates. [computer graphics stack exchange], 2015. URL https://computergraphics.stackexchange.com/questions/1769/world-coordinates-normalised-device-coordinates-and-device-coordinates.

[14] Mathilde Caron, Hugo Touvron, Ishan Misra, Hervé Jégou, Julien Mairal, Piotr Bojanowski, and Armand Joulin. Emerging properties in self-supervised vision transformers. In *Proceedings of the International Conference on Computer Vision (ICCV)*, 2021.

[15] W. Kabsch. A solution for the best rotation to relate two sets of vectors. *Acta Crystallographica Section A*, 32(5):922–923, Sep 1976. doi: 10.1107/S0567739476001873. URL https://doi.org/10.1107/S0567739476001873.

[16] Eric W Weisstein. "sphere point picking." from mathworld–a wolfram web resource. URL https://mathworld.wolfram.com/SpherePointPicking.html. [Accessed on 2024-06-10].

[17] Jim Belk. How to find a random axis or unit vector in 3d? [maths stack exchange], 2011. URL https://math.stackexchange.com/questions/44689/how-to-find-a-random-axis-or-unit-vector-in-3d. [Accessed on 2024-06-10].

[18] Mervin E. Muller. A note on a method for generating points uniformly on n-dimensional spheres. *Commun. ACM*, 2(4):19–20, apr 1959. ISSN 0001-0782. doi: 10.1145/377939.377946. URL https://doi.org/10.1145/377939.377946.

[19] R. K. McConnell. Method of and apparatus for pattern recognition, Jan 1986. U.S. Patent 4,567,610.

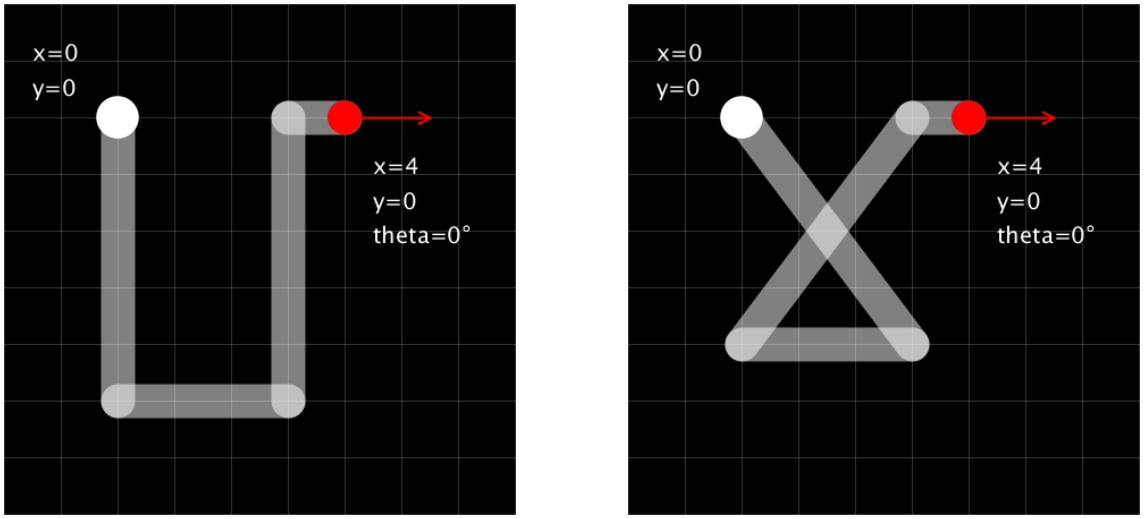
[20] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 886–893 vol. 1, 2005. doi: 10.1109/CVPR.2005.177.

[21] William T. Freeman and Michal Roth. Orientation histograms for hand gesture recognition. Technical Report TR94-03, MERL - Mitsubishi Electric Research Laboratories, Cambridge, MA 02139, 1994. URL https://www.merl.com/publications/TR94-03/.

Appendix A

Null space inverse kinematics

As discussed in Section 3.2, inverse kinematics will be an underdetermined system if the number of controllable joints exceeds the degrees of freedom of the desired end effector pose. This has the effect of having infinitely many, all equally valid robot poses which achieve the desired end effector position. However, this is only in theory. Consider Figure A.1b. Here we can see that the solution requires two of the arm links to intersect. While this is fine in theory and does constitute a solution to the IK system, in the real world the links of the robot arm are not infinitely thin line segments, they are physical parts with thickness. If this robot was in the real world this would require parts of the robot to pass through each other. This is obviously impossible in a real situation.



(a) A valid IK solution

(b) An invalid IK solution

Figure A.1: Not all IK solutions may be valid

Fortunately for us, we saw that by having additional degrees of freedom, we can generate infinitely many solutions to the IK system. This means we could prune these invalid solutions by adding additional constraints. While it is possible this could make all solutions invalid, this is both unlikely and necessary. These invalid solutions remain because the current IK system does not perfectly capture our real world restrictions. If all solutions are removed by our additional constraints, then this simply means the desired end effector pose was not possible in a real world system.

There are two main ...