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 20, 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, Nurdayana 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 world around it. 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. We want the robot to 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.

Some existing solutions to this problem [1–6] 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. This approach, called 'Reinforcement Learning', is detailed in Section 2.2. Alternatively, engineers can directly generalise the task by presenting it in many different environments and demonstrating to the robot how to complete the task in each case. This however still multiplies the workload of engineers providing large amounts of training data to the system. This method, called 'Imitation Learning', is explored in Section 2.3.

Newer state of the art solutions [7–9] build on the ideas of Imitation Learning, but instead of offloading the work to the engineers, attempt to offload the work to an Artificially Intelligent system. A common approach among these solutions is to capture specialised additional information during training time which allows the task to easily generalise. This additional information, often in the form of images taken by the robot, is something which can be automatically collected with no extra effort from the engineers. In order to process and incorporate the additional information, these papers unanimously opt to rely on Artificial Intelligence models. Either training neural networks from scratch, or incorporating pre-trained vision transformer models. In doing so these solutions achieve a remarkable feat, in that a single human provided demonstration is sufficient to learn a task which generalises to unseen environments. We explore these 'One-shot Imitation Learning' approaches further in Subsection 2.3.3. Figure 1.1 illustrates the problem we aim to solve.

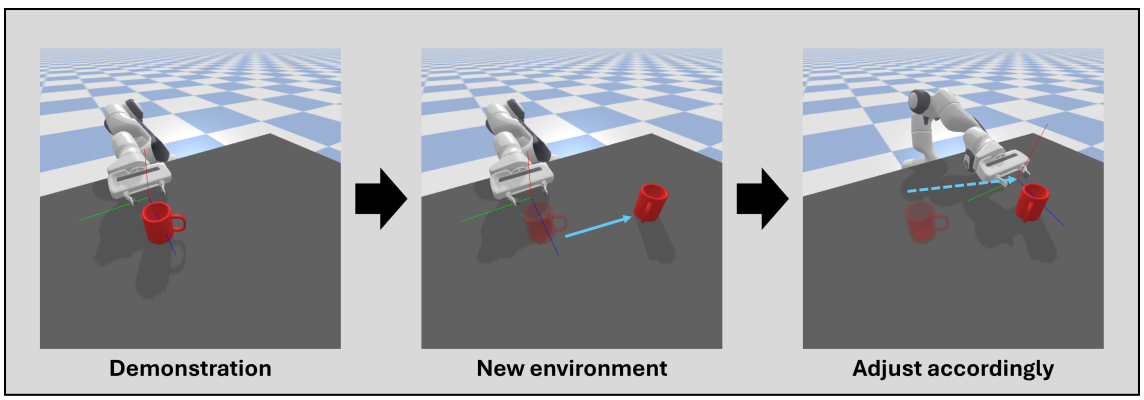


Figure 1.1: The One-shot Imitation Learning Problem

1.1 Contributions

In this paper we consider the limitations of the aforementioned state of the art solutions. In particular we focus on "DINOBot: Robot Manipulation via Retrieval and Alignment with Vision Foundation Models", a system created by N. Di Palo and E. Johns [9] as a case study. N. Di Palo and E. Johns generously publicise their source code for installing and utilising the vision transformer at the heart of their system.

Upon implementing this system it becomes apparent that the physical hardware requirements of this and many other state of the art solutions are immense. The DINObot system requires in excess of 16GB of GPU video RAM. We are unable to find an exact requirement, since this has already reached the maximum hardware specifications of the machines available to us for this project. As such we were not able to implement the DINObot system at any point during this project. This unfortunately limits our ability to directly compare our solution to DINObot, since we cannot test DINObot on the same test suite we use to evaluate our own system. However it does highlight this issue, existing solutions are not accessible to a large portion of potential users.

The driving motivation for this paper is to emulate the methods of state of the art systems, to produce high quality and accurate results while operating under much tighter computational constraints. Our framework should be capable of learning tasks and generalising to novel environments from a single human provided demonstration. It should achieve this with substantially lower memory requirements, while not impacting performance and accuracy.

This paper manages to achieve a such a system which we call 'LiteBot' in reference to its lightweight hardware requirements. LiteBot achieves comparable performance to existing solutions. One of the most significant features of the system presented in this paper is its ability to operate entirely on a CPU. The system performs efficiently without any GPU resources being utilised. Despite this LiteBot is still fast enough for real time inference during testing. Our method combines strategies from existing One-shot Imitation Learning systems, while exchanging memory intensive Artificial Intelligence components for classical algorithms from the field of computer vision. While this may seem irrelevant in an application which already requires specialised robotics hardware, the methods described in this paper have broader applications in physics simulations, video games and pure mathematical settings. In such fields the reduced barrier to entry becomes highly desirable. Furthermore, as shown in this paper, physical robotics hardware has become a luxury as opposed to a necessity, with high quality simulation software providing a solid basis to perform robot learning within.

Moreover, we create a framework which can learn multiple tasks independently. 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

The system presented in this paper does not present any serious ethical concerns.

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, 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, however each of these methods come with some major drawbacks which will be discussed.

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 [10]. 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 [2]. 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. As such 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 so called '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 [1]. 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 [2, 5]. After sufficiently long training, the robot's ideas of which states have a high associated reward 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. The agent will choose the greedy option, the best currently known action in its current state, with proportion ϵ. Otherwise it will choose any other random action with probability $1 - \epsilon$. When choosing the random action, all actions are drawn from a uniform distribution. 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, meaning 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 since 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. Furthermore, the robot's solutions are heavily biased towards the human demon-

strations. The agent will not do much exploring and innovating on the solutions presented by the humans. Depending on the implementation the agent may process the provided demonstrations differently.

Inverse Reinforcement Learning

One approach is to use Inverse Reinforcement Learning to infer the reward function from the provided demonstrations [3]. 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.

Behavioural Cloning

Another approach is to forgo the reward function entirely. In Behavioural Cloning [4, 6], 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.

2.3.1 Training data problem

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.

2.3.2 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 [11]. In this algorithm the agent determines its policy to complete the task, then for every state along this trajectory, the system requests a demonstration from the human. This demonstration shows what the human would have done to complete the task, from this state. This ensures that we provide demonstrations for states which the robot actually visits. There are two main problems with this design. Firstly, we cannot pre-collect all the demonstrations. The robot will propose a policy, and then request more training demonstrations. The agent then needs to retrain the policy on this new training data, taking more time. 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 one full demonstration from every state along the proposed trajectory. This means 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' [12]. 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.

2.3.3 Reducing the dependency on demonstrations

While these previously stated methods help to ensure the demonstrations are diverse, they achieve this by collecting far more demonstrations overall. 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. We wish to reduce the requirement for large amounts of training data by leveraging alternative approaches to generalise the demonstrations.

The work of A. Mandlekar et al. [13] proposes a system to upscale a small number of demonstrations into a larger set. 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" [8]. 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. The above solution 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 [9] 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 relative 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 [8]. 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 [9]. 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 [13]. 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.

Having reviewed the current literature, we now cover some preliminary knowledge which will be necessary in understanding the system we develop in this paper.

2.4 Preliminaries: 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 coordinate 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.

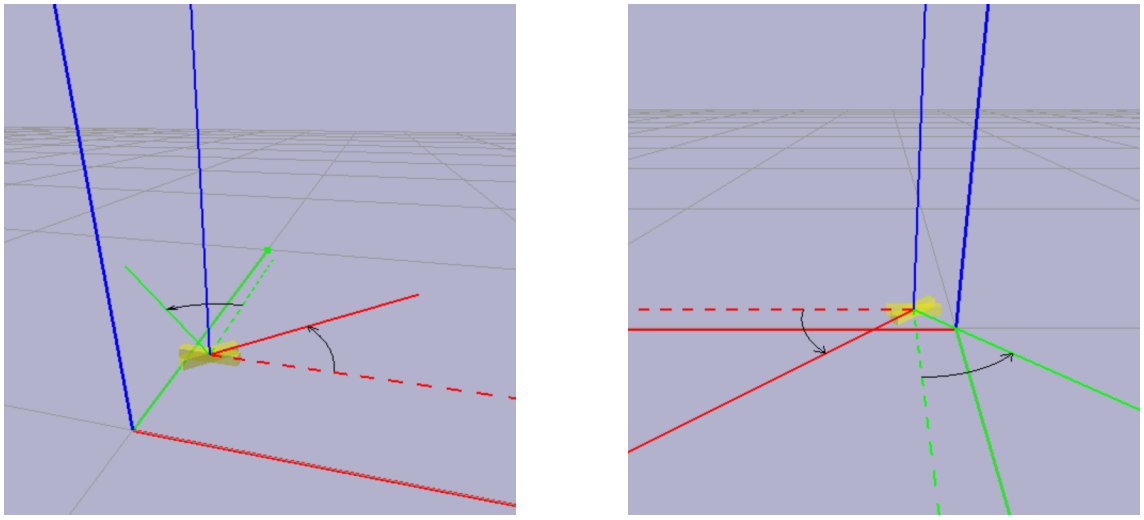


Figure 2.1: An object before and after a rotation of $\frac{\pi}{4}$ about the Z axis [1]

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

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

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. Figure 2.1 shows the exact same setup from two different view points, to better visualise the rotation described.

2.4.1 Quaternions

Quaternions are a 4 dimensional number system [14] which can be used to represent and manipulate orientations and rotations in 3D space [15]. 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 otherwise by simply normalising the quaternion, dividing each component by its L2-norm.

Furthermore, we know by Euler's rotation theorem [16] 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.

Figure 2.2: The right hand rule for rotation

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 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

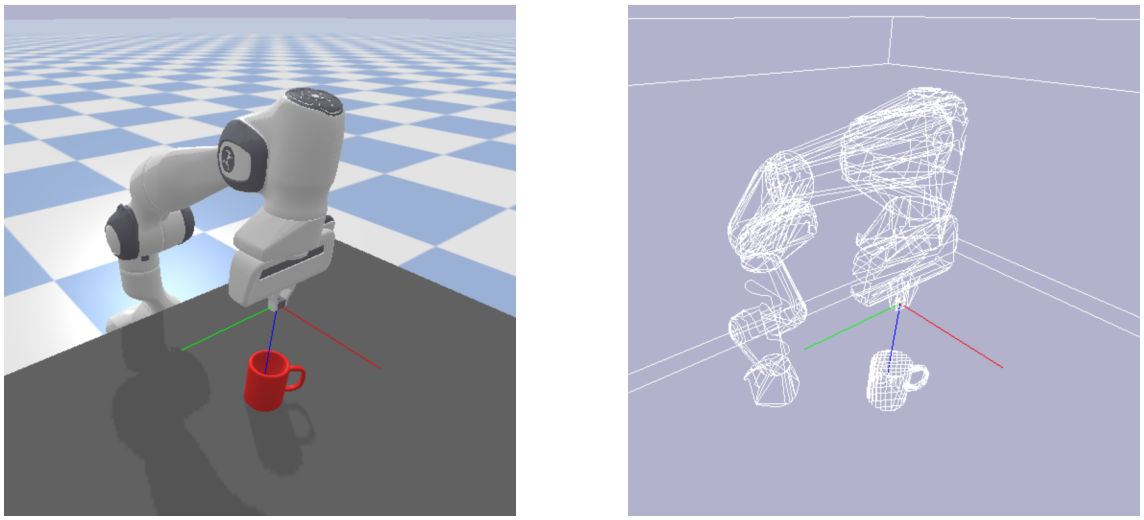
Technical Design

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' [17], 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. This issue is only apparent when using inverse kinematics to calculate the joint angles to reach a desired end effector position, since the inverse kinematics function, `pybullet.calculateInverseKinematics()`, 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. However `pybullet.setJointMotorControlArray()`, the function to control the arm motors, expects one angle for every joint, even those which cannot be moved and so the specified angle is ignored anyway. 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 other inconsistencies in the signature and behaviour 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 [18]. 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.

Finally, in PyBullet orientations are represented using quaternions, [14, 15] an explanation of which is provided in Section 2.4. Despite requiring a fourth dimension, quaternions provide a big advantage over the alternative of Euler angles. This being that quaternions do not suffer from 'gimbal lock' [19, 20].

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 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

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.

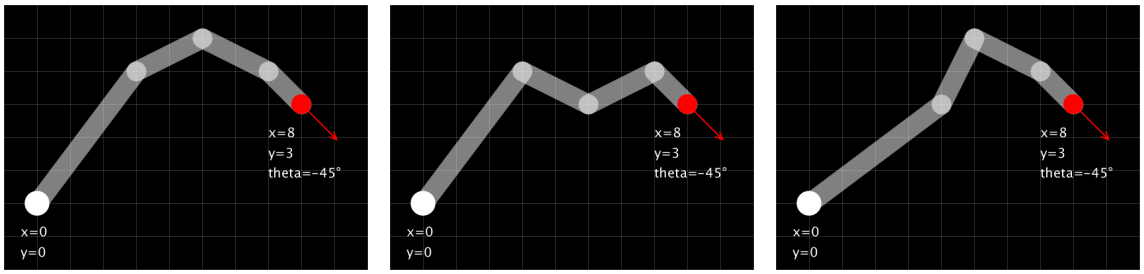


Figure 3.2: An underdetermined inverse kinematics system has multiple solutions

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 Section 3.5 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 inverse kinematics system to our advantage is explored further in Appendix A.

We may optionally decide to also include velocity information in addition to positional information in the robot's state. 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. The specific decision for how to encode the robot's state is not tightly coupled to the methods used in this paper. One could imagine how to modify the ideas presented in this paper to produce a version of LiteBot which encodes the robot state with velocities as opposed to positions.

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 inverse kinematics 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 [8]. 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 factors can influence the environment. If external factors 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' [9] 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 [8].

$$\Sigma = I^{(RGB-D)}$$

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 Section 3.5. 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

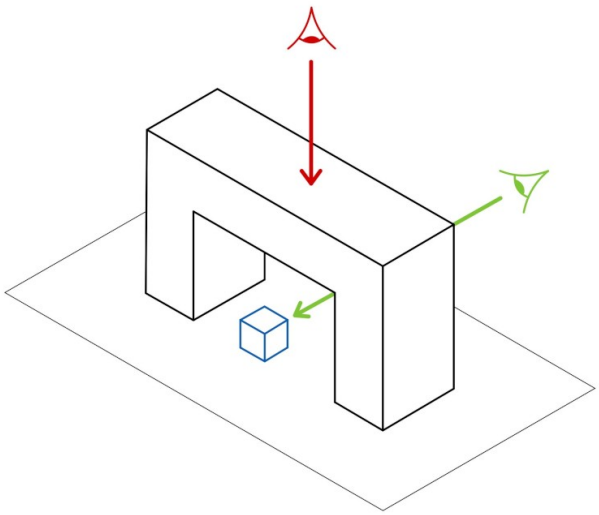


Figure 3.3: Example environment where initial position cannot view the object of interest

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.

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

In practice we make a few modifications to simplify the implementation. As will be seen in Section 3.5, 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' [21]. 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 Overview

Figure 3.4 shows an overview of the pipeline developed in this paper. The first step of the algorithm is to select the demonstration to follow. The robot will look at every demonstration image in its training corpus, and select the one which is most similar to the current environment. It will then compare this demonstration image to the live image taken by the robot's camera. From these two images LiteBot then extracts keypoints, and uses these to compute the transformation which has been applied to the object. This offset is then applied to the end effector, transforming the demonstration trajectory into this new environment. The robot can then execute this transformed trajectory, completing the task. The following sections detail the specifics of each module of the pipeline.

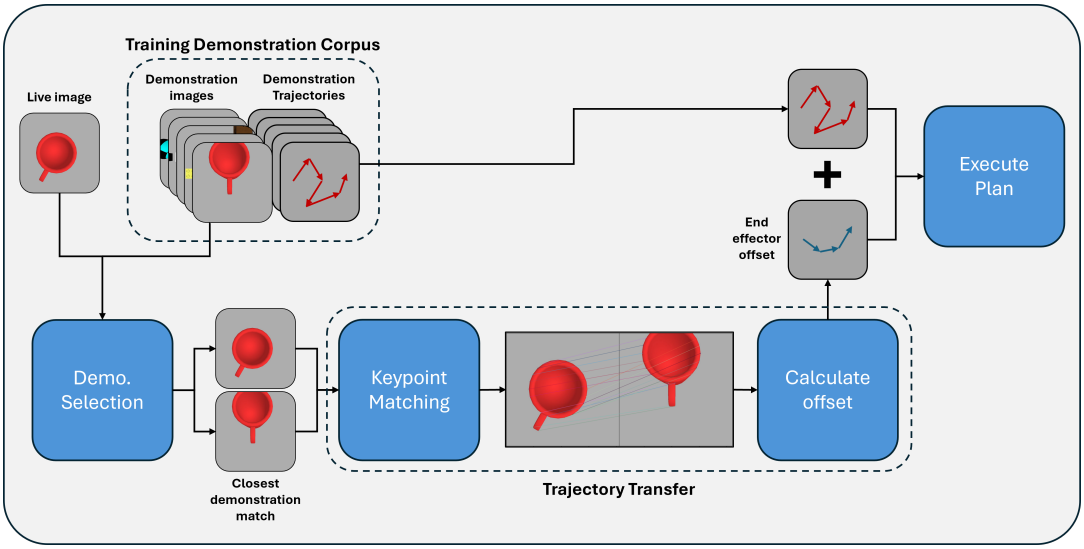


Figure 3.4: Overview of system pipeline

3.4 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 LiteBot 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' [22]. 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.

3.5 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

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

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 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[2].

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

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 Section 3.8. 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.

3.6 Using keypoints 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 keypoint matching figures in this report we will assume the left image is the live image and the right image is the demonstration image.

Figure 3.5 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.

[2]Note that this equation is a single implication, the converse is not necessarily true. From only the RGB data we cannot tell if the object is small and close to the camera or large and far from the camera (and end effector). If we include depth information then we can formulate the converse statement.

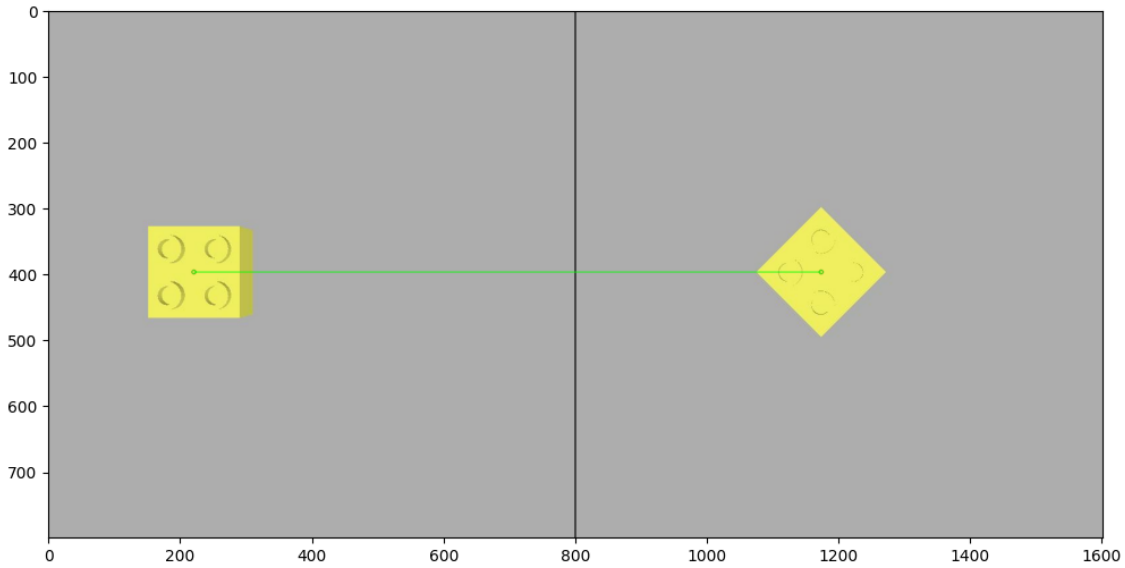


Figure 3.5: Centre of mass matching

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 3.6 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.

Note that this is only one way the keypoints could be matched. Figure 3.7 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

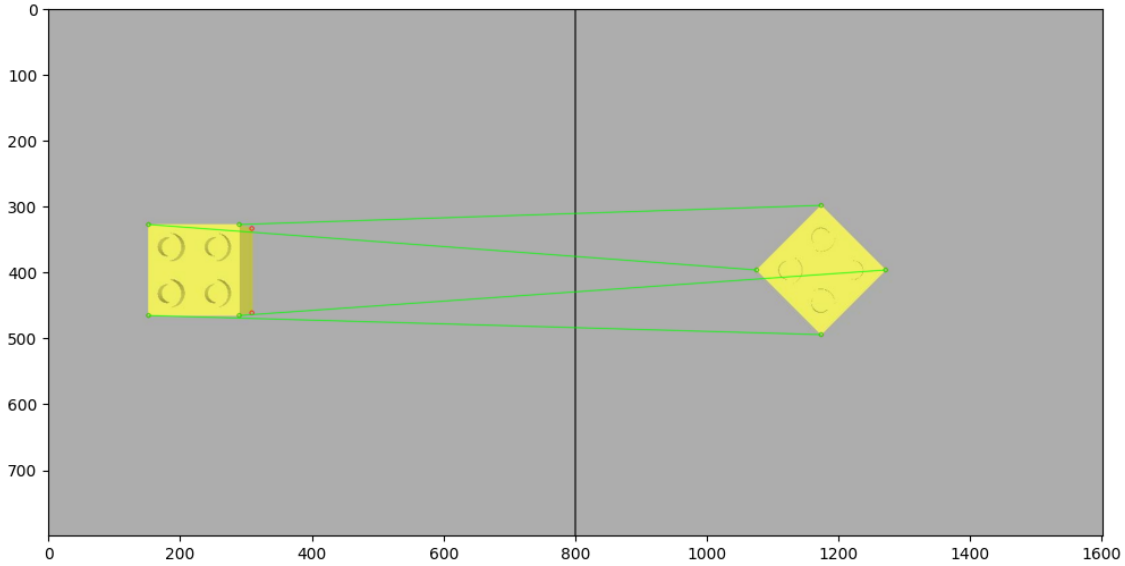


Figure 3.6: Ideal keypoint matching

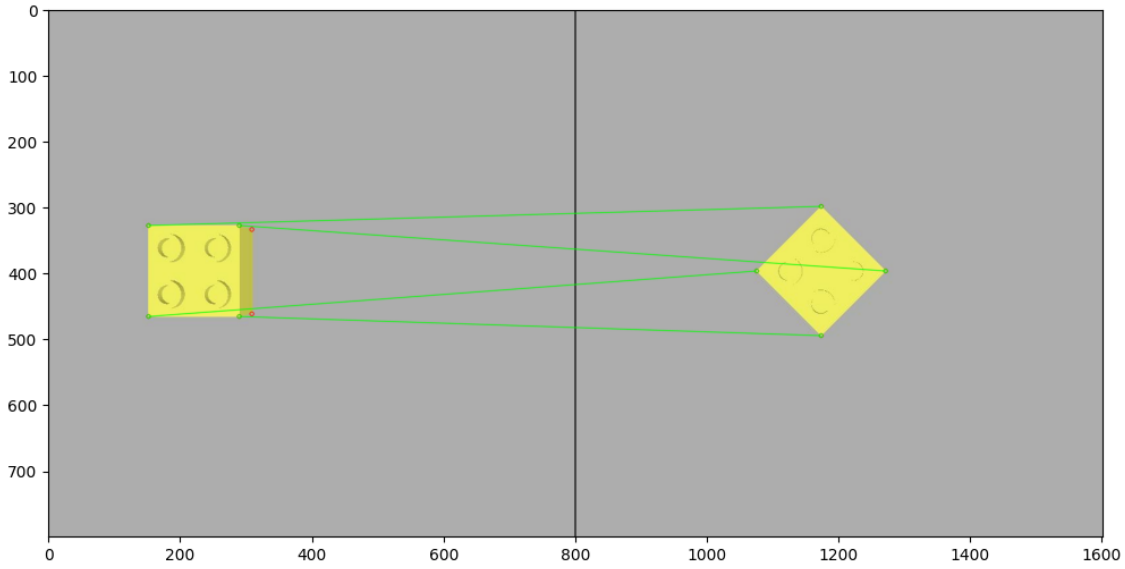


Figure 3.7: Alternative ideal keypoint matching

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. How we achieve this will be explored next.

3.7 Automated keypoint algorithms

Keypoint extraction involves identifying distinctive points or features in an image that can be reliably recognized under different viewing conditions, such as taking the same image from a different angle, zoom, or changing the lighting conditions. These keypoints, often represent corners or edges of objects within the image, since these are easy to recognize across different images. Once we have chosen the keypoints we need some way to identify them. This is achieved through creating feature descriptors, which provide a unique signature for each keypoint based on the image information in the local area around the keypoint. These feature descriptors should be invariant to the viewing conditions described above, so that the same keypoint is assigned the same (or as close as possible within some error) descriptor, even when viewed from different angles, zoom, or lighting conditions. Once keypoints are extracted from different images, keypoint matching techniques allow us to find correspondences between the keypoints in two different images of the same scene. This section explores various keypoint extraction algorithms, how they work, and resulting advantages and disadvantages for our use case. As well as this we discuss the methods used to match these keypoints across images.

3.7.1 Keypoint Extraction

Most keypoint detection algorithms work by identifying points in the image where it sharply changes. These are often the boundaries between an object and the background. Mathematically, these are the points in the image where the derivative peaks (either positively or negatively). The derivative is approximated by the finite difference between pixel intensities, and computed in both the X and Y direction of the image. We can therefore compute the magnitude and direction of the gradient:

$$|g| = \sqrt{g_x^2 + g_y^2}, \qquad \theta = \arctan \frac{g_y}{g_x}$$

where g_x and g_y represent the finite difference (gradient) in the X and Y direction respectively.

SIFT

The first keypoint detection algorithm we look at is Scale-invariant Feature Transform (SIFT), designed by David Lowe and published in 2004 [23].

The extraction process begins with identifying potential keypoints through scale-space extrema detection. This involves creating a series of progressively blurred images using Gaussian filters at different scales and computing the 'difference of Gaussians' (DoG) to highlight areas of interest. Local extrema in these images are points which stand out from their neighbours in both space and size. The keypoints are further localized to sub-pixel accuracy.

Once keypoints are identified we must assign a descriptor. To ensure the descriptors are invariant to image rotation, all local pixels vote for a dominant orientation in a discretised histogram using their gradient orientation. The winning bucket is the dominant orientation. This dominant orientation is used to offset all future orientation calculations. The local area is now split into a 4x4 grid of sub-regions.

Each sub-region produces a histogram of gradient orientations, relative to the dominant orientation, with votes weighted by the gradient magnitude. Each histogram is encoded as the height of each bucket in a vector. All histogram vectors are concatenated creating a final 128-vector descriptor.

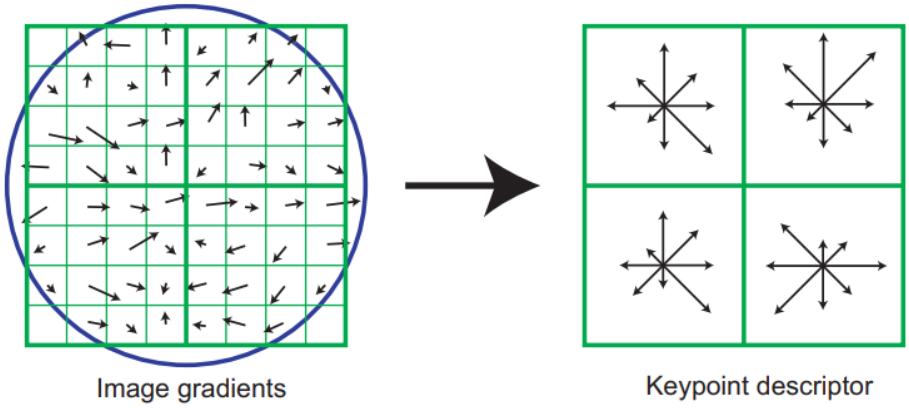


Figure 3.8: SIFT keypoint descriptor [23]

SIFT was a seminal algorithm in computer vision which has been tried and tested since its patent expired. It stands as an excellent baseline for our keypoint extraction.

ORB

ORB (Oriented FAST and Rotated BRIEF) is a keypoint extraction and description algorithm developed by Ethan Rublee et al. in 2011 [24]. ORB combines the FAST (Features from Accelerated Segment Test) keypoint detector [25] with the BRIEF (Binary Robust Independent Elementary Features) keypoint descriptor [26].

FAST identifies keypoints by examining the intensity of pixels in a circular region around each candidate pixel and classifying it as a keypoint if it has a sufficient number of neighbouring pixels that are significantly brighter or darker than the candidate. ORB utilises FAST at different scales to create scale invariant features.

Additionally, ORB assigns an orientation to each keypoint as the direction of the vector from the keypoint to the centroid of the intensities within a local patch. This orientation assignment ensures that the keypoints are invariant to image rotation. BRIEF is then used to assign a descriptor to each keypoint. BRIEF generates a binary string for each keypoint by performing a series of binary intensity difference tests between pairs of pixels at predefined locations relative to the keypoint. Figure 3.9 shows an example of which pixel intensities may be compared to each other. This results in a 256 bit binary string, as opposed to a 128-vector of floating point values with SIFT.

ORB's main advantage is its speed and efficiency. Large amounts of keypoints can be detected and described very quickly, since the majority of calculations are binary tests on pixel intensities.

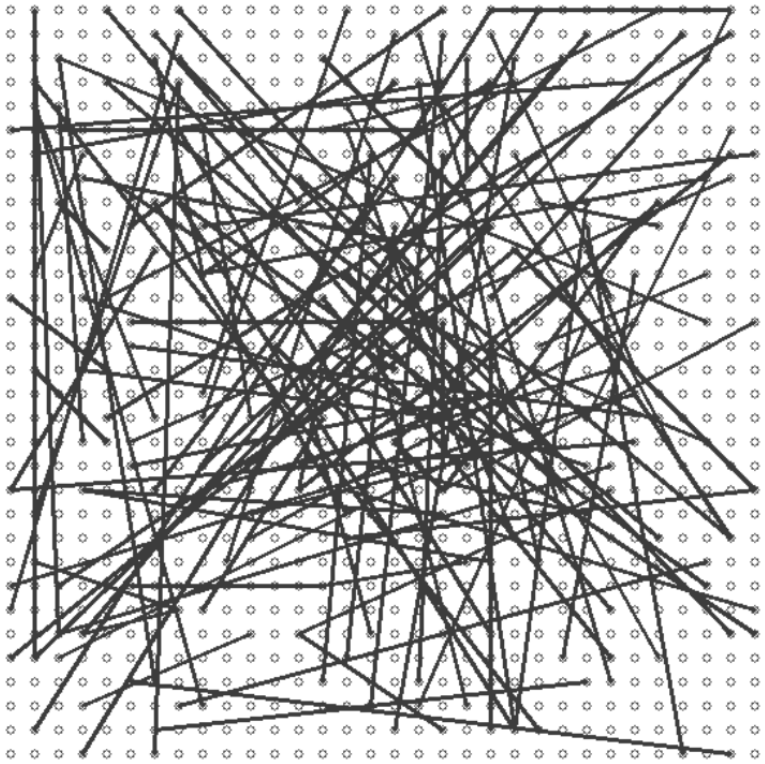


Figure 3.9: Example ORB pixel comparison locations [26]

3.7.2 Keypoint Matching

Once we have extracted keypoints and their descriptors, we need a way to find which keypoints represent the same semantic features across the two images. This is done by defining some notion of distance between keypoint descriptors. A shorter distance means the descriptors are more similar, and they are more likely to refer to the same feature of the image.

Brute Force Matching

The simplest method for keypoint matching is brute force matching. In this approach, each descriptor from one image is compared with all the descriptors from another image to find the best matches. This involves calculating the distance between each pair of descriptors using a suitable metric. For SIFT descriptors this would be the Euclidean distance between the vector descriptors. For ORB this would be the Hamming distance between the bit string descriptors. The descriptor pairs with the smallest distances are considered as potential matches.

Brute force matching can be computationally expensive, especially for large datasets, as it requires a comparison of every possible pair of descriptors, resulting in $O(n^2)$ time complexity. The advantage is that this method is highly accurate and does not rely on any approximations, making it a reliable choice. To improve accuracy we can include additional considerations such as cross checking, where matches are verified by ensuring mutual nearest neighbors, and ratio tests, which filter out ambiguous matches by comparing the distance of the closest match to the second closest match.

Grid-based Motion Statistics

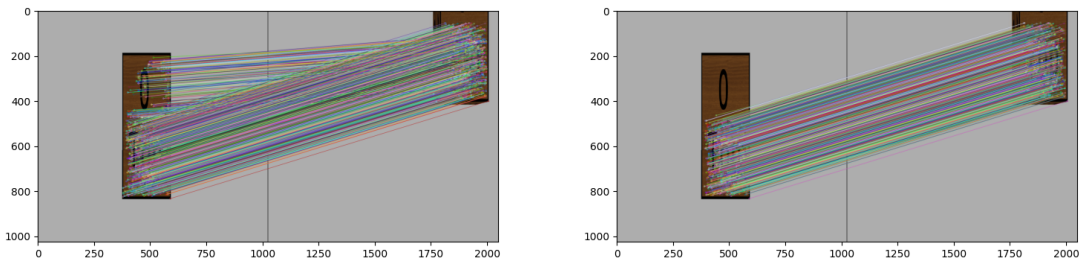
Grid-based Motion Statistics (GMS) is a more sophisticated algorithm developed by Jiawang Bian et al. in 2017 [27]. This paper describes a method for refining keypoint matches leveraging a few key assumptions. If we assume that the two images are of the same object, then we can impose smoothness constraints. Unlike brute force matching, which relies solely on descriptor similarity, GMS incorporates spatial information to filter out false matches effectively.

The key principle of GMS is to divide the image into a grid and analyze the distribution of keypoint matches within each grid cell, ensuring that matches are not only similar in descriptor space but also exhibit consistent motion patterns across the image. The object may have been moved, rotated, zoomed and warped by camera projection, but if it is the same object then there will be consistencies in how near by keypoints are transformed.

This is perfect for our use case since provided the demonstration selection stage has worked as intended, the object will be the same in both images. Furthermore, the assumption GMS relies is one we have already taken. We have already assumed that the object in both images has simply been translated and rotated between the live and demonstration image. We already mentioned in Section 3.6 that we are using these keypoints to try to recover this transformation. As such, by using GMS we filter out keypoint matches which specifically contradict this assumption, leading to much more stable results when computing the transformations. This is demonstrated in Chapter 4.

Outlier Filtering

While GMS is very good at computing consistent keypoint matches, it still can have some difficulty with large groups of inconsistent matches, since GMS only filters out matches which disagree with its local neighbours. This is great as a first pass and serves to remove the majority of high frequency noisy matches. However, the grid based structure of GMS, means that individual grid cells can often disagree with each other. This is low frequency noise, where collections of matches disagree with other collections.



(a) Keypoint matches using GMS

(b) Keypoint matches using GMS + outlier filtering

Figure 3.10: Our method further improves the consistency of keypoints

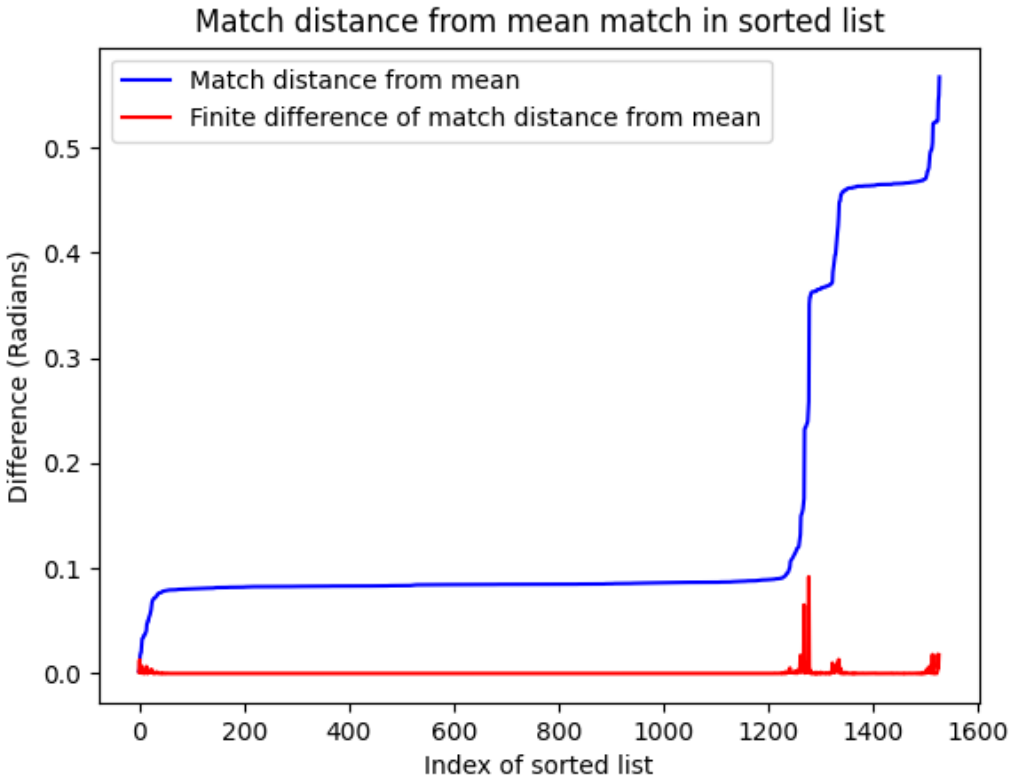


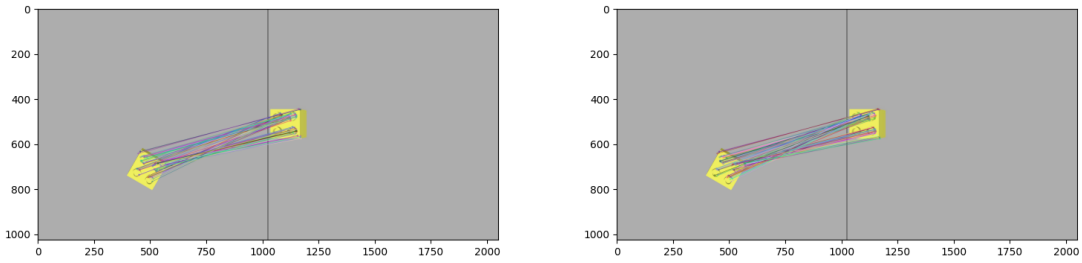
Figure 3.11: Identifying outliers by their distance from the mean vector

Figure 3.10 shows example keypoint matching using GMS, before and after applying outlier filtering. We can see that in Figure 3.10a, there is a collection of keypoints which match incorrectly. The object has moved diagonally, as most matches correctly identify. However, this collection correspond to a more horizontal translation. Our filtering method successfully removes these incorrect matches, while retaining the correct matches.

In this paper we develop an additional measure to improve our keypoint matches. We further eliminate outliers to refine keypoint matches by considering all matches, not just those within a grid cell. After applying GMS to obtain an initial set of matches, this algorithm performs an additional filtering step based on vector analysis of the keypoint coordinates. We start by considering keypoint matches as vectors within image space. If a keypoint at (x, y) in image one, matches to a keypoint at (x', y') in image two, then we consider the match as the vector from (x, y) to $(x'y')$. This is the line you would get by connecting the keypoints if the two images were superimposed on top of each other. By considering keypoint matches as vectors in image space, we can consider these vectors as describing how the object moved between the images. This is not a direct calculation of the translation and rotation, since we make no assumption that the camera which took the images was in the same pose in world space. However, we are still able to perform some filtering on these vectors regardless.

To identify and remove outliers, we examine the gradient of each vector relative to the mean values. Matches that deviate significantly from the mean are considered potential outliers. Specifically, we order the matches based on how much they differ

from the mean vector. We then compute the finite difference between consecutive ordered pairs of match vectors. This allows us to see how quickly the derivative changes. The algorithm looks for spikes in the derivative, which indicate sudden changes in the gradient that are inconsistent with the overall motion pattern. We record the index of all points where the gradient spikes more than some hyperparameter threshold. We then check each spike from left to right ensuring that if we were to split here, we wouldn't remove too many keypoints. The justification for this additional test is that if the potential outliers held a majority or close to it, then we would conclude that these points are not in fact outliers. If a potential split would remove too many keypoints, but there are later spikes, then we check each spike until the first one passes. Only once a spike passes both tests do we split the list at this point, and discard the matches with a mean distance higher than that of the split point. Figure 3.11 shows a plot of distance from the mean and the finite difference for the images shown in Figure 3.10



(a) Keypoint matches using GMS

(b) Keypoint matches using GMS + outlier filtering

Figure 3.12: Our method only removes outlier matches

It is important to analyse the derivative and not just remove the vectors furthest from the mean. This is because if a rotation is present then the vectors will not agree perfectly on their orientation. However, these changes should be smooth in the ordered list of matches, since a rotation would affect all vectors proportional to their distance from the centre of rotation. These matches are not outliers and should not be removed. As such, only when the finite difference changes drastically can we accuse the matches of being outliers. Figure 3.12 shows an object which has been rotated. For this object the matches are very accurate and consistent. Outlier filtering removes very few matches as outliers. We can see in Figure 3.13 that this corresponds to a smooth increase in the distance from the mean. As such the derivative is much more stable.

In Chapter 4 we compare the effectiveness of each keypoint algorithm with each matching algorithm, to determine the best combination of algorithms for use in this project.

3.8 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

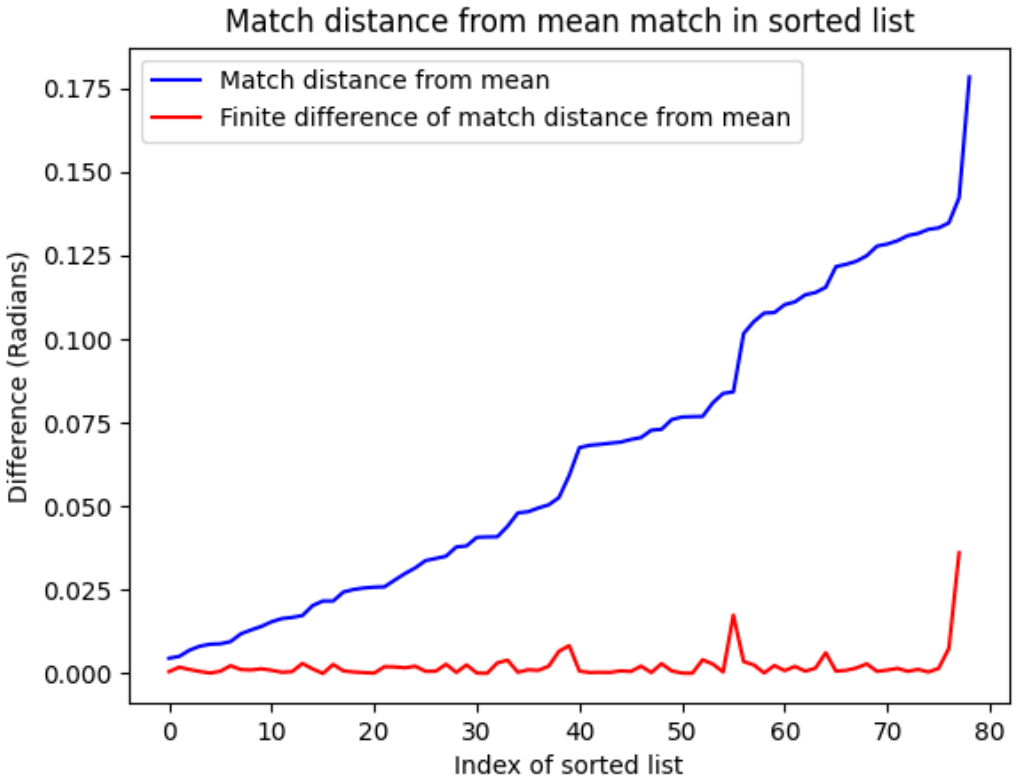


Figure 3.13: A rotated object will have a smooth increase in distance

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' [28]. The algorithm takes in two sets of points, a reference set P, and a test set Q, and computes a translation t, a rotation matrix R and a scale factor c, 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 calculations are needed to compute how far to move the end effector.

Algorithm 1 shows the modified Kabsch-Umeyama algorithm we use in LiteBot. The first step of the algorithm will look familiar to something we tried in Section 3.6. The first step is to compute the centroids of the two sets of points. 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 we computed earlier by considering all pixels of the object. The algorithm then normalises the points so that the centre of mass is at the origin. From here it computes the covariance matrix, H, between the two sets of points. This covariance matrix tells us how the X, Y and Z coordinates vary between the two sets of points.

Next we perform singular value decomposition (SVD) to break H into a composition of 3 elementary transformations. In Figure 3.14 we can see that the effect of multiplying by the matrix M is to first rotate by the matrix V^T, followed by a scaling by the matrix Σ, followed by another rotation by the matrix U. We note that Σ is diagonal, where each element defines how much to scale in each direction. In this figure, we scale by σ_1 in the X direction and σ_2 in the Y direction. Therefore $\Sigma = \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix}$

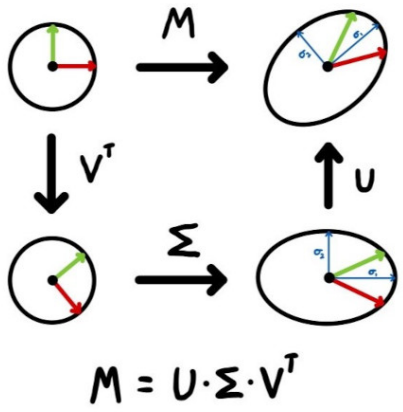


Figure 3.14: Singular Value Decomposition

This notion naturally extends to higher dimensions. Since U, Σ, V^T all have the same shape as the input matrix M, when we pass in our 3×3 covariance matrix, we get the rotation, scaling and second rotation in 3 dimensions. Note that Σ is the conventional notation for this diagonal matrix, it is not the demonstration environment context which we also denoted with capital sigma.

From here we compute the rotation between the points as $R = U \cdot S \cdot V^T$. This is just the combination of the rotation parts of the covariance matrix H with some normalisation. This step involves an additional matrix S which was not present in the original algorithm. The effect of S is to ensure the rotation matrix R has a determinant of 1. In the original algorithm the rotation matrix may include a reflection of the points. For our purposes we want to disallow reflections, since we cannot reflect the object by moving the end effector camera. To achieve this we compute S such that if R is already a pure rotation, then S will be the identity matrix, having no effect. If R would have contained a reflection, then we negate one element of S, so that the resulting R will once again be a pure rotation with no reflection. We can test this by comparing the determinants of U and V^T, without needing to compute R first to find its determinant.

In our modified algorithm, we do not compute the scale factor c, or the transformed points Q'. We do not compute Q' since we only need the transformation so that we can apply it to the end effector. We do not need to know where the transformed points will end up exactly. We do not compute c for the same reason we disallowed reflections. The scale factor lets us move all points further from the centroid $\overline{q}$. However, when moving the end effector we cannot magically cause the object to change size. We can make the object appear slightly bigger or smaller by zooming into or out from the object. However, this would affect the depth to the object and resultantly the Z coordinates of the points in Q. As such we disallow scaling to be considered in the transformation, and let the Z coordinates handle the elevation the robot arm should be at instead.

While this does limit the system's generalisability to novel objects which are different sizes compared to demonstration objects, it is a necessary sacrifice. Even with the same object in both images, the scale factor is rarely computed as exactly 1, due to camera perspective warping the object. This means the calculation of the

scale factor is far too inaccurate to be used reliably. Adding it into the calculation for the translation as per the original algorithm, causes this result to also become wildly inaccurate, decreasing the reliability of the whole system.

The final modification we make is in how we compute the translation. In Algorithm 1 we compute the translation as the difference in the centroids, without the rotation and scaling applied. In the original algorithm translation is computed as $t = \overline{p} - cR \cdot \overline{q}$. This change is a result of how our environment is set up. Rotations are applied about the current object position. This is the case for all objects, but especially important for the end effector. When we rotate the end effector, we do not rotate about the world origin, we rotate about the end effector's current position, effectively spinning it in place. As such we can consider the translation and rotation as two independent transformations. First we spin the end effector in place to match the object's orientation. Then we move the end effector so that it is aligned relative to the object. Because the rotation happens in place, it does not have any effect on how much we need to translate by.

Algorithm 1 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)
  $\overline{p} \leftarrow \frac{1}{N} \sum_i^N p_i$
  $\overline{q} \leftarrow \frac{1}{N} \sum_i^N q_i$
  $H \leftarrow \frac{1}{N} \sum_i^N (p_i - \overline{p})^T (q_i - \overline{q})$ *// Covariance Matrix*
  $U, \Sigma, V^T \leftarrow \text{SVD}(H)$ *// Singular 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 $R^{-1} = R^T$. 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.

3.9 Final Algorithm

With each of these modules defined, we can consider the algorithm as a whole. Following the pipeline shown in Figure 3.4 we produce the following pseudocode:

Algorithm 2 LiteBot: A lightweight One-shot Imitation Learning algorithm

Input: $\mathcal{D} = \{[\tau, I^{(RGB)}, I^{(D)}, V]\}$ *// set of training demonstrations*
$\Sigma' = [I'^{(RGB)}, I'^{(D)}, V']$ *// live environment context*

```
Select demonstration
repeat
    Take live environment context image
    Extract keypoints from demonstration image and live image
    Match keypoints between demonstration image and live image
    Convert matching keypoints to world coordinates
    Calculate t, r from modified Kabsch-Umeyama algorithm
    Move end effector by t and rotate by r
    Calculate mean keypoint error
until keypoint error > threshold
Calculate total amount moved M and R
for all keyframes [P, O] in chosen demonstration do
    Move end effector to P + M and RO
end for
```

Until now we have been considering that we can compute M and R in a single pass. While in an ideal case this is true, due to inaccuracies in keypoint matching, it is sometimes necessary to perform a second or third pass and iterate towards the correct position. As such we place the keypoint detection and end ffector offset code inside a loop. The majority of the time this is not necessary, and a single pass is sufficient to bring the end effector within the threshold tolerance. We compute the mean Euclidean distance between each keypoint and its corresponding matching keypoint as a measure of the error between keypoints. We average this error over all keypoints since we cannot be certain how many keypoint matches we will have. Therefore, it is a fairer approach to use mean error as opposed to the total error, which would disadvantage objects where more keypoints were identified.

This pseudocode also overlooks a number of implementation improvements. For example error handling if no keypoints can be found. This is likely to occur if the object is completely out of frame when the live image is taken.

Chapter 4

Evaluation

In this section, we evaluate the performance of LiteBot 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 keypoint matching algorithms by using them in a full run of the algorithm 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 keypoints in the demonstration image. These keypoints 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 keypoints but in this new image. 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. For further details about the specifics of the test suite, refer to Appendix B.

Now we can convert the manually marked keypoints in the demonstration and live image to world coordinates and pass them 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 human provided, ground truth keypoints is to control any additional noise in the test. If we used our keypoint matching algorithm then

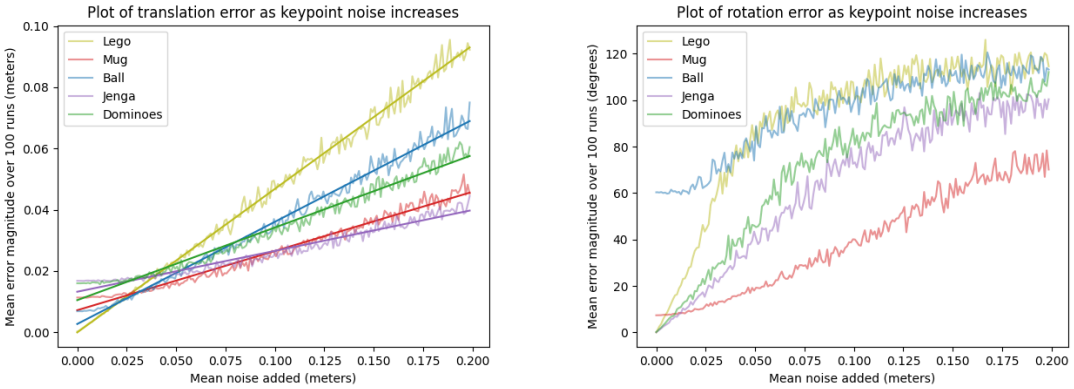
Object	Translation applied	Rotation applied	Number of keypoints
Lego	[-0.1, 0.05, 0]	[0, 0, $\frac{\pi}{3}$]	4
Mug	[0, 0.05, 0]	[0, 0, $\frac{\pi}{6}$]	16
Ball	[-0.1, 0.07, 0]	[0, 0, 0]	7
Jenga	[0.09, 0.15, 0]	[0, 0, $-\frac{\pi}{6}$]	22
Domino	[0.05, 0.07, 0]	[0, 0, -$\frac{\pi}{4}$]	10

Table 4.1: The true transformation between demo and live objects in the test suite

it would be unclear how much error came from our randomly added noise, or from mismatches in the keypoint algorithm. As such we gain clearer results by using our ground truth keypoints. With the method defined, we now add some noise to the ideal keypoints 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.

4.1.1 Coordinate noise

In this first set of tests the noise added is calculated as a random 3D unit vector multiplied by some random magnitude. When computing the random 3D unit vector, we take care to use an "equal area projection of the sphere onto a cylinder" [29]. This allows us to choose a point uniformly from the unit sphere, without experiencing a bunching of points at the poles [30, 31]. This random noise is applied to the world coordinates after they have been computed from the ideal keypoints. This gives us a solid grasp as to just how much noise can be present before the algorithm produces unsatisfactory results.



(a) Noise graph for translation error

(b) Noise graph for rotation error

Figure 4.1: Sensitivity analysis with added keypoint coordinate noise

In these tests noise was added within a range of 0.001 meters. The point plotted at horizontal coordinate x was the result of sampling the noise magnitude from the half open interval $[x - 0.0005,\ x + 0.0005)$. This is with the exception of the data point at $x = 0$. This point is the error when no noise was added, as though sampling the noise magnitude from $[0, 0)$.

Translation error

We see in Figure 4.1a that the magnitude of the error between the true translation and computed translation appear to follow linear trend lines for all of the objects. A key observation is that the lines do not all intersect with the origin. This conveys that even with no added noise, the computed translation is still not the true translation. This systematic error remains constant across all results for the same object, only affecting the vertical intercept of the line, and not the gradient. This systematic error can be attributed to two main causes. Slight numerical precision errors will be present in the algorithm. These are somewhat exacerbated by converting our keypoints to world coordinates first. As a result keypoints, which are often very close to each other, have coordinates which differ only in low order decimal places. However, the much larger contribution to this systematic error is human error. The ground truth keypoint matching was performed by a human, and as a result is subject to large inaccuracies. It can be very difficult to identify the exact pixel which correspond between the demonstration and live image, especially when rotations and perspective distort straight edges.

For our test we are more concerned with how the error changes as a result of increasing the amount of noise the algorithm is subjected to. Therefore, we care more about the gradient of the trend lines than the true values. From this graph we can notice an interesting correlation. The gradient of each line is inversely proportional to the number of keypoints we marked. The Jenga block with the most keypoints is the most stable to additional noise, evident by possessing shallowest gradient. Meanwhile, the Lego piece with the fewest marked keypoints is the most sensitive. This relation holds for all intermediate objects with no outliers.

We also notice that even the worst performing object, the Lego piece, has a gradient less than 1. The line appears to have a gradient of approximately $\frac{1}{2}$, when 20cm of noise is added, the error magnitude is 10cm. This is promising because it is unlikely that the true noise experienced would be as high as 20cm. The graph shows that all objects were very stable up to 5cm of noise. This is a much more reasonable and still fairly generous estimate for how much noise the system will experience.

This result is quite promising for us, since the sensitivity appears to decrease as we mark more keypoints. We also note that the number of keypoints in this test are very low, due to requiring a human to manually place them. We will see in Section 4.2 that certain keypoint algorithms can identify on the order of thousands of keypoint matches. While this is dependent on the object used, it is promising as to the reliability of the system when automatic keypoint algorithms are used.

Rotation Error

Figure 4.1b shows how the rotation error increases as the amount of noise increases. The rotation error is computed as the geodesic distance between the rotations. Formally we consider the true rotation matrix $\hat{R}$ and the computed rotation matrix R as representing orientations within 3D space. We then compute E, the error rotation matrix which rotates from R to $\hat{R}$. E represents how much more rotation we needed to do to get to the correct total rotation. E represents a rotation by some angle θ about some axis. We consider θ to represent the size of this error, and is

therefore our dependent variable.

$$E = R^T \cdot \hat{R}, \qquad \theta = \frac{trace(E) - 1}{2}$$

Unfortunately, Figure 4.1b clearly shows that the algorithm is much more sensitive to noise when computing the rotation matrix. The trend lines in this graph do not appear to be linear for all objects. The error appears to increase more rapidly, before slowing down and reaching a plateau. The mug object appears to be most stable however, even this can reach quite large errors.

We again notice that the lines do not cross the origin due to systematic error. However, this time there is an outlier. The ball object has a rotation error of 60 degrees with the human marked keypoints. This is not a small error due to inaccuracies in the keypoint placement, this is a fundamental issue with the Ball object. As shown in Appendix B, the keypoint placements on this object are not well placed to determine rotation. 6 of the 7 keypoints are placed around the perimeter of the ball. These mark the silhouette of the ball, which since it is a perfect sphere, would not change with a pure rotation. These keypoints are very useful for determining the translational offset, but they are no help in computing the rotation. This leaves a single keypoint placed in the middle of the object. The position of this keypoint, particularly its depth, is the only information the system has for computing how the object has rotated. Obviously this is insufficient, and leads to the very large error in Figure 4.1b. However, we will see in Section 4.2, that even with many more keypoints, the system struggles to determine the rotation of the Ball object for the exact same reason. The majority of detected keypoints are on the edges of objects since this is when the image drastically transitions from the object to the background.

It is to be expected that rotation would be more sensitive to the noise than translation. Consider Figure 4.2 which shows a simplified view of this concept. We can see that for translation, the most unlucky we can get with the noise is for all of the noise vectors to align. If the noise vectors all had length n, then this would produce a translation error of n. This is why in Figure 4.1a, the gradient of the trend lines were all less than 1. The absolute worst case is that the translation error would be equal to the noise magnitude. Since this is very unlikely and we compute an average, the trend lines are much more shallow than this worst case. However, for rotation, the worst case is that the noise vectors are all rotated copies of each other. We can clearly see by inspecting the figure, that this rotates the object by a much more noticeable amount, even when the same magnitude of noise is applied. This is why the rotation error is much more sensitive to the noise than the translation error.

4.1.2 Keypoint noise

In the second round 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 algorithm are the result of small floating point inaccuracies. The source of the error will be predominantly a result of the keypoint matching. As such in this second round of tests, we add noise to the pixel coordinates of the keypoints, before converting them to world coordinates. This is more representative of the type of error we will encounter in this

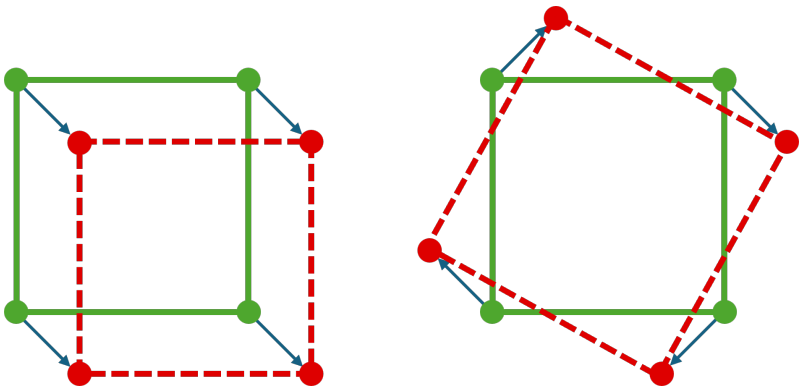
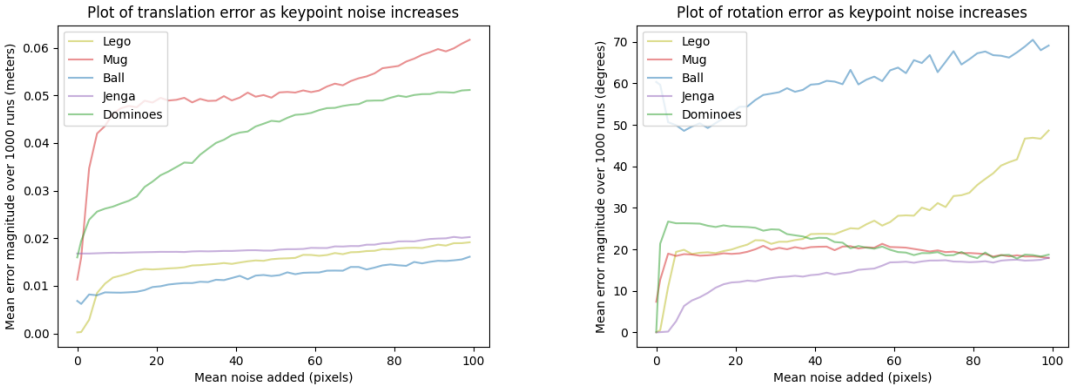


Figure 4.2: Unlucky noise can affect rotation more than translation

system, and so proves a more reliable result. Since the keypoint algorithms used offer sub-pixel precision, we are not limited to only adding integer amounts of pixel noise. As such we use the same random vector method as in the first tests, only this time the vector is 2 dimensional. Since it does not make sense to be adding too small of a fraction of a pixel, as this will have very little effect between runs, we run these tests with a larger granularity of noise magnitude. This does however, allow us to compute the average over 1000 runs, rather than 100, providing more reliable data.



(a) Noise graph for translation error

(b) Noise graph for rotation error

Figure 4.3: Sensitivity analysis with added keypoint pixel noise

Translation error

In Figure 4.3a, objects like the Jenga block and Ball are remarkably stable, with the noise having almost no impact on the computed translation. The Jenga block specifically, exhibits many properties of an ideally marked object for this algorithm. The object has many marked keypoints as discussed earlier. However, it is also one of the largest objects. This means that the noise is proportionally smaller when applied to this object compared to smaller objects like the Lego piece. The object takes up more of the image, and so it takes larger pixel deviations for the noise to have much affect.

For the Mug object the error appears to increase more rapidly. This is likely due

to the specific topology of this object. The keypoints were marked around the rim of the mug and handle. As such, only small deviations are needed for the keypoint to fall off of or into the mug. In either case, the depth of the keypoint will change wildly. This highlights an important insight. Although the X and Y coordinates of the keypoint can only deviate by as much as the noise maximum, the Z coordinate can change by very large amounts depending on the height of the object. The effect of this is that the error is not uniformly applied to each coordinate. It would be analogous to performing the coordinate noise tests, but with a different much larger magnitude range for the Z coordinate specifically. It is this tendency for small image space deviations to lead to very large world space deviations that accounts for the reduced stability in the pixel noise tests. This is shown in Figure 4.4.

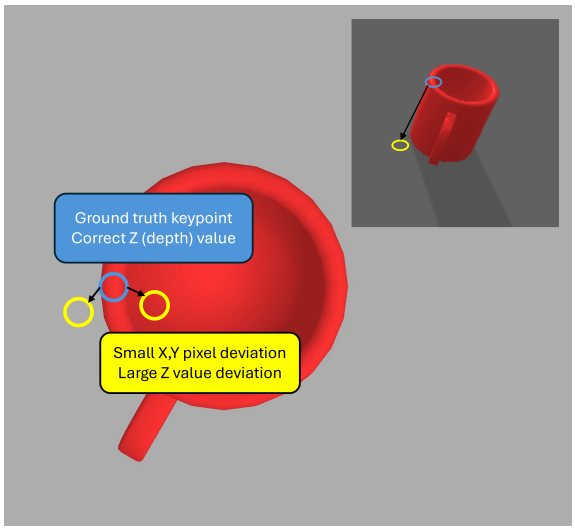


Figure 4.4: Small X,Y deviations can cause large Z deviations

To support this theory consider the keypoints marked for the Jenga block object as shown in Appendix B. Only 2 of the keypoints are marked on the edge of the object. All other keypoints are in a more central position on the object's face. As such, the deviations of up to even 100 pixels, are not enough to move the majority of these keypoints off of the object and onto the background. As such for this object, the Z coordinates are almost always precisely correct.

With regards to how this discovery impacts the true system, it is likely that during normal execution the keypoint matching would not be this poor. The incorrect matches present in the true system are the result of two different keypoints matching to each other. It is unlikely that the system would identify any keypoints which are entirely on the background and not the object. This large Z deviation is a consequence of how the random noise is applied in this test, and is less apparent in usual execution.

Rotation error

The rotation error appears to follow similar trends in both the coordinate noise test (Figure 4.1b) and the pixel noise test (Figure 4.3b). For the same reasons as discussed in the previous section, we notice that the rotation error appears less stable than translation, increasing rapidly with small deviations before levelling out. It

does not make much difference to the sensitivity of the system, whether the keypoints become rotated from their true position in pixel space or in world space (see Figure 4.2). The only affect is that the noise applied in pixel space seems to cause smaller deviations overall. This is evident by comparing the two graphs, in both figures the lines follow approximately the same shape, but in Figure 4.3b the lines plateau to a lower mean rotation error.

Another interesting observation is that the Ball object initially decreases its rotation error as more noise is added. This is certainly unexpected as we would expect noise to make the result worse. This is likely a result of the generally poor rotation calculations for the Ball object. As discussed in Subsection 4.1.1, by nature of the Ball object being a perfect sphere and the test containing very few marked keypoints, it is very difficult for the system to accurately describe the rotation that occurred. As such we hypothesise that some of the manually marked keypoints were poorly placed, and it happens that adding small amounts of noise, can on average move the keypoints closer to where they should have been placed.

4.2 Comparison of keypoint algorithms

In this section we wish to empirically evaluate which keypoint extraction and matching algorithm performs best when used for our system. Any algorithm which can extract and match keypoints could theoretically be used for our system, however this paper focuses analysis on the algorithms discussed in Section 3.7.

We consider two algorithms for keypoint extraction and description, SIFT and ORB. For these rows in the table we apply brute force matching with a suitable distance metric, performing no additional refinement. We add further rows to the table for additional refinement algorithms applied to the initial brute force matches. Despite our outlier filtering originally being designed to be performed after GMS matching, it is possible to apply it immediately to the brute force matches. However, where GMS refines the matches to conform to local smoothness constraints, our method simply removes outliers. Without GMS applied, it is likely that a large number of keypoints will be considered outliers. As such we do not expect this algorithm to perform well, nonetheless it is included to test this hypothesis. The table contains two additional rows, one for applying GMS to the brute force matches, and another for GMS with added outlier filtering.

To conduct the tests we use the same 5 object test suite as with the noise sensitivity tests. For each object we define 5 poses with varying difficulty for the system. We then run LiteBot, using the specified keypoint algorithm to attempt to complete the task on the object in the new pose. Since the whole system is deterministic with respect to the object pose, we do not complete multiple runs with the same object in the same pose. In the table we report the mean translation error, denoted d; the mean rotation error, denoted θ; the mean number of keypoint matches, denoted n; and the success rate of completing the task, denoted s. All averaged across the 5 poses. The results of these tests are presented in Table 4.2.

	Lego	Mug	Ball	Jenga	Domino
SIFT	d=0.0066m θ=142.2° n=9.0 s=0%	d=0.0336m θ=18.6° n=7.0 s=0%	d=0.0217m θ=53.3° n=19.2 s=40%	d=0.0143m θ=126.3° n=42.4 s=20%	d=0.0660m θ=137.1° n=8.0 s=0%
SIFT + filtering	d=0.0089m θ=37.9° n=7.2 s=20%	d=0.0417m θ=26.5° n=5.2 s=0%	d=0.0200m θ=45.6° n=16.4 s=40%	d=0.0171m θ=14.3° n=39.4 s=60%	d=0.0565m θ=166.9° n=6.8 s=0%
SIFT + GMS	d=0.2350m θ=60.0° n=0 s=0%	d=0.0569m θ=45.0° n=0.2 s=20%	d=0.1210m θ=0.0° n=0 s=0%	d=0.0162m θ=0.3° n=19.8 **s=100%**	d=0.0583m θ=108° n=0 s=0%
SIFT + GMS + filtering	d=0.2350m θ=60.0° n=0 s=0%	d=0.0569m θ=45.0° n=0.2 s=20%	d=0.1210 θ=0.0° n=0 s=0%	d=0.0155m θ=0.3° n=19.0 **s=100%**	d=0.0583m θ=108° n=0 s=0%
ORB	d=0.0081 θ=20.6° n=204.4 s=60%	d=0.0112 θ=17.9° n=318.0 s=60%	d=0.0166 θ=82.6° n=223.4 **s=80%**	d=0.0213 θ=8.63° n=2783.8 s=80%	d=0.0341 θ=19.8° n=120.0 s=40%
ORB + filtering	d=0.0067 θ=20.7° n=197.2 **s=80%**	d=0.0251 θ=33.7° n=247.4 s=40%	d=0.0167 θ=80.6° n=208.8 **s=80%**	d=0.0194 θ=6.40° n=2605.2 s=80%	d=0.0289 θ=24.9° n=113.6 s=40%
ORB + GMS	d=0.0051 θ=25.8° n=95.4 **s=80%**	d=0.0113 θ=21.2° n=113.6 **s=80%**	d=0.0207 θ=93.3° n=67.0 **s=80%**	d=0.0172 θ=3.57° n=1531.8 **s=100%**	d=0.0106 θ=22.5° n=21.0 s=40%
ORB + GMS + filtering	d=0.0050 θ=24.7° n=93.6 **s=80%**	d=0.0097 θ=27.5° n=80.2 s=60%	d=0.0207 θ=93.3° n=67.0 **s=80%**	d=0.0176 θ=3.49° n=1289.4 **s=100%**	d=0.0102 θ=18.5° n=18.0 **s=60%**

Table 4.2: Summary statistics for each algorithm

SIFT algorithms

From these results a number of observations are immediately clear. Most obviously, SIFT keypoints perform much worse than ORB features on average, across all objects. The key insight is to notice that the average number of matched keypoints is significantly lower than the same setup using ORB. We noticed in Section 4.1, that the algorithm was more stable when a larger amount of keypoint matches were present. The limited number of keypoints available when using SIFT, contribute to the algorithm's poor performance. On rare occasion, SIFT with GMS can outperform standalone ORB, although this only happens with the Jenga block, which is able to identify a large number of matches whichever algorithm is used. In most cases, applying GMS to SIFT features produces no matches at all. This is because with what few SIFT keypoints we have available, the brute force matching can ap-

pear very erratic, with no clear consensus among the matches. As such GMS severely struggles, and can result in a situation with 0 matches. If this is the case, LiteBot is unable to proceed[1]. For this test we consider this an absolute failure. We consider in this case as the system deciding to not move the end effector at all. As a result the end effector error is recorded as the entire offset between the demonstration and live objects. Interestingly, with regards to the rotation error, these cases are not even the highest error achieved by some test runs using SIFT. The largest average error achieved was an angle magnitude of 166.9 degrees. For this test, the robot kept trying to rotate the arm an enormous amount, often hitting into itself and becoming stuck. This large error is a result of the very few keypoints, leading to incredibly unstable calculations when computing the pose offset.

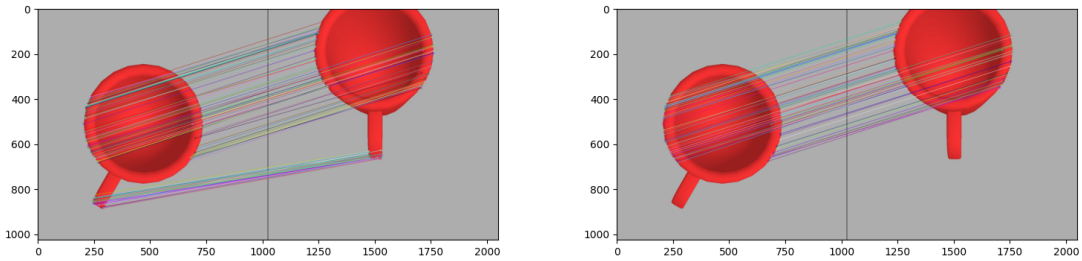
A particularly interesting result was the Mug object with SIFT and GMS applied. For this test the algorithm succeeded a single task. It was able to do this by identifying a single keypoint on the mug, which it matched across the images. This singular keypoint acted like the centre of mass we discussed in Section 3.6. This singular keypoint was enough to compute the translation offset fairly accurately, however, it was unable to extract any information about how the object rotated. As such it did not rotate the end effector at all, so still received the maximum penalty for rotation. In the other 4 runs, the algorithm was unable to identify any keypoints at all.

ORB algorithms

With that covered we can turn our attention to the ORB algorithms. The first thing we may notice is that the number of identified keypoint matches is highest when only using ORB and brute force matching. This is to be expected since all the additional matching algorithms refine the existing matches and remove outliers. We also notice that quite frequently, applying outlier filtering has no effect compared to the row above in the table where the same algorithm was used without filtering. This is the result of our carefully designed system, to prevent removing points unless they are clearly outliers. In a number of cases, the outlier filtering removes no points, and so has no effect on the performance. This may imply that our current requirements to mark outliers are too strict, preventing the filtering from being as effective. However, this needs to be balanced. We can see with the Mug object that filtering actually causes one test to fail, which passed without filtering. In this run there are a number of keypoints which match between the handle of the mug, as shown in Figure 4.5a. These allow the system to more accurately compute the rotation and complete the task. However, in Figure 4.5b we can see that the filtering algorithm removes this collection of matches. While it is true that they do not conform to the gradient of the other matches, fitting our definition of outliers, in this case they were not incorrect matches. With these matches removed, the computed rotation is less accurate, and the robot just barely fails the task, managing to grasp the handle, but dropping the mug as it tries to lift it. As such we conclude that further tweaking to the outlier filtering algorithm would help improve the accuracy of the LiteBot

[1]When no keypoint matches are detected the system will move the end effector randomly and try again. This is designed to try to find the object in the case that the object was fully out of frame. However, in this case the object was in frame, and GMS was just unable to identify any matches. This causes the arm to loop infinitely, moving randomly trying to identify keypoints. After a number of failed attempts we cut the test short and assign a failure.

as a whole. Another observation is that filtering when applied directly to brute force matches has a tendency to increase the end effector pose error. This rarely results in failing more tests (except in the case of the Mug discussed above). This supports our hypothesis from earlier, in that the filtering is designed to be applied after GMS. When the filtering is applied without GMS first refining the matches, too many keypoints fit the definition of outlier. When filtering is applied after GMS, as intended, the end effector pose generally decreases, although we only pass one extra test on the Domino objects.



(a) Keypoint matches using GMS

(b) Keypoint matches using GMS + outlier filtering

Figure 4.5: One mug test fails as a result of incorrectly filtering accurate matches

Across all algorithms, certain objects appear more forgiving with error, and more prone to success. The jenga block for example performs the best out of any object. This is a result of it being a large object with a textured surface. This provides a very large number of keypoints, so many that even SIFT is able to perform well after GMS is applied. It is obvious in retrospect that a larger object occupies more pixels of the image. This means that slight deviations in the exact pixels of keypoints have less of an impact on the resulting world coordinates of keypoint matches. It would also provide more possible keypoints to be identified and matched when using the automatic keypoint algorithms. This is something we can control with our demonstrations. By recording demonstrations where the object of interest occupies the majority of the camera frame (by placing the end effector closer to the object when taking the environment context image), the agent will have an easier time computing the correct offset at inference. We also note that the test suite contains some tests where the object is partially out of frame (refer to Appendix B). This removes a lot of the potential keypoints of the object, since we already established that edges of the object contain the most keypoint matches. This was a deliberate choice to make the test suite harder, with LiteBot overcoming this issue for the most part as seen in Table 4.2. Nonetheless, in a practical use case where the user is not trying to actively stress test the system, demonstrations where the object is fully in frame will improve the accuracy of the computed offset.

Another highly forgiving object is the Ball. This object is extremely forgiving to errors in rotation, since as a sphere the object interacts exactly the same whatever angle it is grabbed from. However, it is very unforgiving when it comes to translation error. If the arm tries to grasp the Ball object from off centre, then the circular nature of it can cause it to shoot away as the gripper closes. The arm needs to ensure that the line between its gripper arms, is as close to a diameter of the sphere as possible, else it will fail to pick up the Ball.

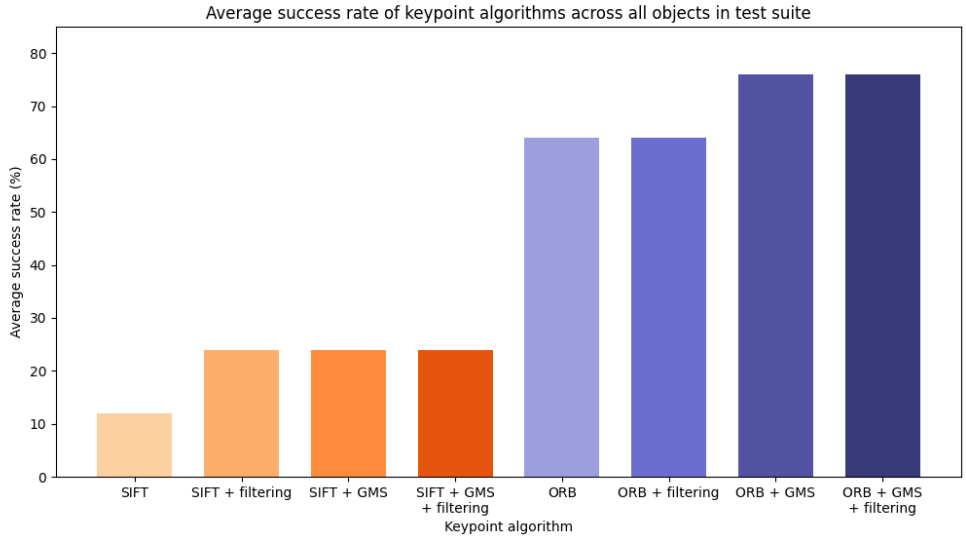


Figure 4.6: Success rate of tests across all objects, for each keypoint algorithm

From these tests it is difficult to decide the outright best algorithm. ORB+GMS and ORB+GMS+filtering both pass the same number of tests. However, with the exception of the Mug object, filtering manages to reduce the end effector pose error on average, even if only slightly. It is clear from these results that the outlier filtering does not have a large impact on the performance of LiteBot. The only time it actively increases the number of tests passed is with SIFT and no GMS. As such it would be fairly reasonable to choose ORB and GMS, with or without outlier filtering as our final algorithm. In this paper, we elect to implement ORB feature extraction and description, followed by brute force matching, GMS matching and outlier filtering into the final version of LiteBot. While the filtering step has very little effect on overall performance, it slightly improves end effector pose error. It is clear from the failing Mug test, that there is further work to be done on the outlier filtering algorithm. With these further improvements, we hope that the filtering step would have a more positive impact on performance.

4.2.1 Average success rate

Having selected our algorithm, we test a further 5 novel object poses for each object. We take the average across all objects to calculate the mean success rate of our framework as **70.0%** on this larger test suite. Since the end effector error and number of matched keypoints depend heavily on the object in question, it would not be appropriate to compute an average for these values. As stated in Chapter 1, we were not able to implement pre-existing One-shot Imitation Learning frameworks, due to their high computational requirements. As a result, we cannot directly compare these algorithms to our own on a consistent test suite. In lieu of this, we compare our results to those produced in the algorithm's original paper.

As DINObot influenced many design choices in this framework, it would be pertinent to compare it to our solution. At a high level, the pipeline of our framework and DINObot share many similarities. As such it serves as an excellent base line for our system. The goal of this project was to show that we can take a standard One-shot Imitation Learning system, and exchange the computationally costly AI components with more space efficient classical algorithms, without experiencing a

significant loss in performance. In their paper, N. Di Palo and E. Johns demonstrate a task success rate of 80% on their test suite. While this value is slightly greater than our own, the decrease is marginal. Furthermore, since LiteBot and DINOBot were not tested on the same test suite, it is difficult to draw conclusions about there comparative effectiveness.

What we can say for sure is that LiteBot is very capable at completing simple manipulation tasks in our test suite. 70% success rate is a remarkably high result given that we have exchanged a state of the art vision transformer developed in 2021, with classical algorithms, some developed as long ago as 2011. Of course it is important to note the comparatively small size of our test suite. This is discussed further in Chapter 5. We would like to test our implementation more extensively, with more objects and more dynamic situations. For example the vast majority of rotations applied to objects were around the Z axis. This is because rotating about the other axis would be unstable for most objects. The only non-Z rotations were applied to the Ball, which as discussed has poor rotation offset performance anyway, and the Lego piece, which had to be rotated 90 degrees to be in a stable orientation which did not fall over. This being a test that all algorithms failed to complete. A more extensive test suite would include situations were objects were placed on a inclined surface. This would allow us to apply smaller rotations around the X and Y axis, while still being a stable orientation for the object.

We would also like to test our algorithms performance more extensively with distraction objects. For each object in our test suite, one of the pose tests involved a distraction object. The distraction object test had mixed results, varying largely between different object tests. However, we would like to test to what extent LiteBot can ignore distractions, by varying the separation between the correct object and the distraction object, and by testing different distraction objects which more closely resemble the main object. Performance in the distraction object test would likely be improved by implementing a segmentation map, as discussed in Chapter 5.

4.3 Memory Requirements

Given that the main motivation for this project was reducing memory requirements, we analyse the amount of RAM a machine would need to run our system. During our previous tests, memory usage peaked at approximately 6.58 GiB across all testing runs. We also emphasise that LiteBot is running entirely on the CPU. No GPU computation time, or VRAM are used. Note, the peak does not represent the total amount of data required by the system. This represents the peak RAM allocated by the system at any one time. Some data, particularly the vision transformer used to initially embed the demonstration and live images, is only needed at the start of the program. After this the memory is freed by garbage collection, and can be re-allocated to later parts of the code. This runtime overhead could be reduced even further by pre-computing embeddings as discussed in Chapter 5. We also note that this value includes the overhead of running the simulation environment. Since currently this is intrinsically linked to the algorithm, there is little way to remove this systematic overhead accurately.

We perform a rough test to estimate the requirements if we implemented pre-computed embeddings, in which we remove the embedding code from the algorithm

and manually tell LiteBot the correct demonstration to use. In this test, the runtime peak RAM usage dropped almost 8-fold to 874 MiB. In a full implementation the embeddings would be loaded from disk and compared, potentially increasing this number slightly. Despite this, it still represents a dramatic runtime decrease. It is however worth noting, that if embeddings are pre-computed this simply moves the problem. When recording and saving demonstrations, the vision transformer would still be required to embed the images once in order to save them to disk, meaning this portion of the framework would still require approximately 6.58GiB. As such the peak memory requirements of the complete system have not changed.

Although, it is worth considering that the majority of time spent using the system would be at inference time. The framework is specifically designed to avoid requiring additional demonstrations to be provided later, unless the user wishes to teach additional tasks. As such we can envision a system where a high memory machine can record demonstrations, and distribute these to other low memory machines which only execute the learned algorithm. This could be the case for an industrial application for example, where each individual robot arm of an assembly line does not to be trained independently. Provided they all posses similar robotic hardware, a single robot arm could be trained, and its policy distributed to all arms in a warehouse or factory.

Furthermore, the term 'high memory machine' is relative. While 6.58 GiB is obviously more than 0.874 GiB, even this is not particularly high for modern computers. According to the 'Steam Hardware and Software Survey', conducted in May of 2024 [32], only 2.90% of Steam users surveyed possessed less than 8GB of RAM. It is important to remember that Steam is a video game store and digital distribution system. As such, all users surveyed will be using their machines for gaming, a use case which often demands high amounts of RAM. While this will skew the results, it remains the most recent survey available at the time of writing. Additionally, Micron "recommend 8GB of RAM for casual computer usage and internet browsing" in 2024 [33]. Implying that most users, not just those using their system for gaming, should have access to at least 8GB of RAM[2]. From this data we can be reasonably sure that our system would fit within the limitations of the majority of machines in use at this time.

In conclusion, the system presented in this paper successfully reduces the minimum memory requirements of a similar One-shot Imitation Learning framework, from some unknown amount greater than 16 GiB, to under 7 GiB. In spite of these reduced memory requirements, LiteBot remains fast, with an average time of 6.11 seconds between taking the live image and aligning to the object, ready to perform the transformed demonstration trajectory. However, this time is dependent on how many demonstrations are learnt and need to be embedded. This would again be reduced, by switching to pre-computed embeddings.

[2] 1GB = 1000MB, 1GiB = 1024MiB. It is unclear when the mentioned sources refer to GB if they are actually meaning GiB. However for this comparison, the difference is largely inconsequential.

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.

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 keypoint matching algorithm by investigating outlier filtering. 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 keypoint matches between different objects. If a keypoint in the live image is at a pixel belonging to object 1 in the segmentation map, then it should not map to a keypoint in the demonstration image, unless that keypoint location also belongs to object 1. Note that the live and demonstration images each have their own segmentation map. This prevents erroneous keypoint 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 keypoint 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 modified Kabsch-Umeyama algorithm is sensitive to inaccuracies in the keypoint matching. Removing matches which are categorically incorrect may reduce this sensitivity. However, in practice it is difficult to tell how many inaccurate keypoint 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 reduce the end effector pose error, 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 [22] 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 an easy change to the system. It has not been implemented because it simply does not provide any useful impact on the accuracy of the system.

5.4 Non-AI 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 we use the DINO vision transformer [22] to embed images into a vector 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) [34]. The design is very similar to SIFT descriptors (discussed in Section 3.7), except that it creates a representation of the entire image, not just the local area around a keypoint. HOG has been shown to be suitable in use for object detection [35] and pose estimation [36].

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 keypoint detection. As such the embeddings it produces encode some notion of the keypoints 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.

5.5 Further testing & real world deployment

The use of a simulation environment has limited the scope of the test suite used to evaluate this project. Creating more tests for the system is not a case of grabbing an interesting object off the shelf, and placing it randomly in the environment. Objects need to instead be modelled as a rigid 3D mesh. Since this is a difficult and time consuming process we find ourselves limited by the built in objects available in the software package.

It would be desirable to test over a larger test suite with more objects and poses, to more rigorously test the generalisability of our system. Furthermore, it remains unexplored how our system handles the 'Sim-to-Real gap'. There was sadly insufficient time to deploy this project onto a physical robot arm, and so we are left to speculate on how this would have affected the performance. Imperfections in objects and lighting changes would likely provide additional identifiable features on objects, potentially allowing the system to detect more keypoints than in simulation. However, there are a number of factors which become impossible to overlook when using a real world system. Issues such as camera calibration and imperfections in joint motor movements, would likely create new challenges for the system to overcome.

Chapter 6

Conclusion

In this paper, we have addressed the challenges inherent in robotic manipulation, particularly the need for robots to generalise tasks across varying environments often necessitating Artificially Intelligent components. Through the development of 'LiteBot', we have successfully demonstrated that it is possible to achieve robust, One-shot Imitation Learning with significantly reduced computational resources compared to currently existing solutions. This work not only replicates the capabilities of state of the art systems like DINOBot [9] but also extends accessibility by operating efficiently on standard CPU hardware, bypassing the need for expensive high-end GPU resources.

In Chapter 4 we demonstrate that while noise and inaccuracies in keypoint matching can introduce discrepancies between the computed and ideal end effector poses, LiteBot consistently manages to complete the majority of tasks successfully regardless of this variation. This robustness is evident in Table 4.2, highlighting LiteBot's capability to perform effectively under less than ideal conditions.

Looking ahead, we propose a number of potential directions to take this project further in Chapter 5. In particular, it remains an ideological goal of this project to remove the last remaining AI component, the embedding transformer. This would allow us to reduce the memory requirements even further, and prove definitively that AI is not a requirement for One-shot Imitation Learning.

In summary, LiteBot represents a significant step forward in making advanced robotic manipulation more accessible and efficient. By balancing high performance with lower hardware requirements, this work lowers the barrier of entry for training and experimenting with a One-shot Imitation Learning system, allowing for use in disciplines where traditional hardware constraints may have previously hindered innovation. As we look to the future, the ideas established in this paper will serve as a foundation for further research and development, broadening the practical application and versatility of robotic systems across academic, commercial, and industrial domains.

Bibliography

[1] P. Read Montague, Peter Dayan, Christophe Person, and Terrence J. Sejnowski. Bee foraging in uncertain environments using predictive hebbian learning. *Nature*, 377(6551):725–728, Oct 1995. ISSN 1476-4687. doi: 10.1038/377725a0. URL https://doi.org/10.1038/377725a0.

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

[3] 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.

[4] Donald Michie, Michael Bain, and J Hayes-Miches. Cognitive models from subcognitive skills. *IEE control engineering series*, 44:71–99, 1990.

[5] Pieter Abbeel and Andrew Y. Ng. *Inverse Reinforcement Learning*, pages 554–558. Springer US, Boston, MA, 2010. ISBN 978-0-387-30164-8. doi: 10.1007/978-0-387-30164-8_417. URL https://doi.org/10.1007/978-0-387-30164-8_417.

[6] Caude Sammut. *Behavioral Cloning*, pages 93–97. Springer US, Boston, MA, 2010. ISBN 978-0-387-30164-8. doi: 10.1007/978-0-387-30164-8_69. URL https://doi.org/10.1007/978-0-387-30164-8_69.

[7] Edward Johns. Coarse-to-fine imitation learning: Robot manipulation from a single demonstration. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2021.

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

[9] 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.

[10] Cosimo Della Santina, Christian Duriez, and Daniela Rus. Model-based control of soft robots: A survey of the state of the art and open challenges. *IEEE Control Systems Magazine*, 43(3):30–65, 2023. doi: 10.1109/MCS.2023.3253419.

[11] Stephane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning, 2011.

[12] Yuchen Cui, David Isele, Scott Niekum, and Kikuo Fujimura. Uncertainty-aware data aggregation for deep imitation learning. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 761–767, 2019. doi: 10.1109/ICRA.2019.8794025.

[13] 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.

[14] 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.

[15] James Diebel. Representing attitude: Euler angles, unit quaternions, and rotation vectors. *Matrix*, 58, 01 2006.

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

[17] 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].

[18] 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].

[19] 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].

[20] 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.

[21] 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.

[22] 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.

[23] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, Nov 2004. ISSN 1573-1405. doi: 10.1023/B:VISI.0000029664.99615.94. URL https://doi.org/10.1023/B:VISI.0000029664.99615.94.

[24] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *2011 International Conference on Computer Vision*, pages 2564–2571, 2011. doi: 10.1109/ICCV.2011.6126544.

[25] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In *Computer Vision – ECCV 2006*, pages 430–443, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-33833-8.

[26] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. Brief: Binary robust independent elementary features. volume 6314, pages 778–792, 09 2010. ISBN 978-3-642-15560-4. doi: 10.1007/978-3-642-15561-1_56.

[27] Jiawang Bian, Wen-Yan Lin, Yasuyuki Matsushita, Sai-Kit Yeung, Tan-Dat Nguyen, and Ming-Ming Cheng. Gms: Grid-based motion statistics for fast, ultra-robust feature correspondence. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2828–2837, 2017. doi: 10.1109/CVPR.2017.302.

[28] 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.

[29] Jim Belk. How to find a random axis or unit vector in 3d? from 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].

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

[31] 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.

[32] Steam Valve Corporation. Steam hardware & software survey: May 2024, May 2024. URL https://store.steampowered.com/hwsurvey/Steam-Hardware-Software-Survey-Welcome-to-Steam. [Accessed on 2024-06-19].

[33] Inc Micron Technology. How much ram do you need for your computer memory?, 2024. URL https://www.crucial.com/articles/about-memory/how-much-ram-does-my-computerneed. [Accessed on 2024-06-19].

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

[35] 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.

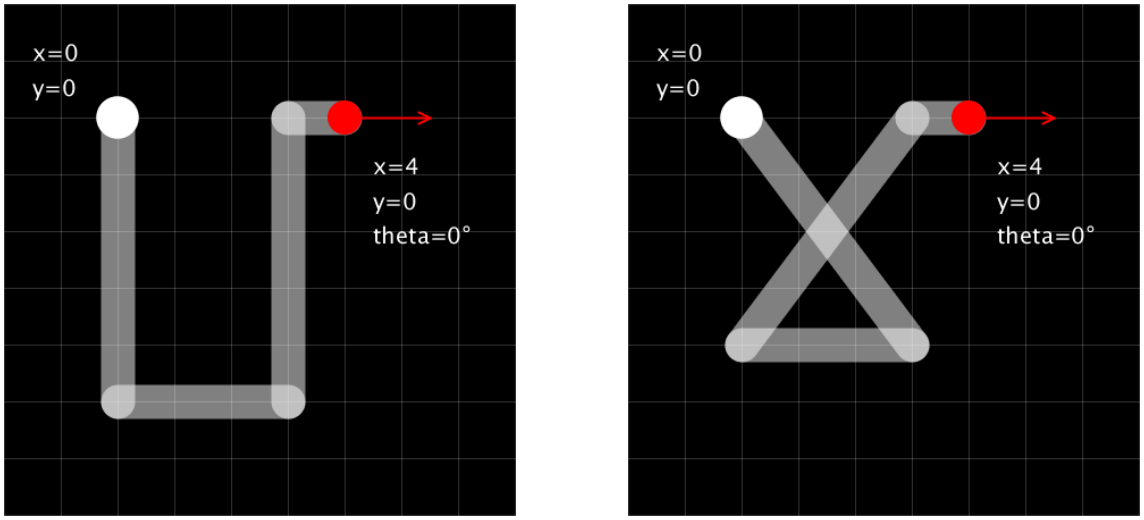
[36] William T. Freeman and Michal Roth. Orientation histograms for hand gesture recognition. Technical Report TR94-03, MERL - Mitsubishi Electric Research Laboratories, Cambridge, MA 02139, 1994. URL https://www.merl.com/publications/TR94-03/.

[37] Travis DeWolf. Robot control part 5: Controlling in the null space, 2013. URL https://studywolf.wordpress.com/2013/09/17/robot-control-5-controlling-in-the-null-space/. [Accessed on 2024-06-19].

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.

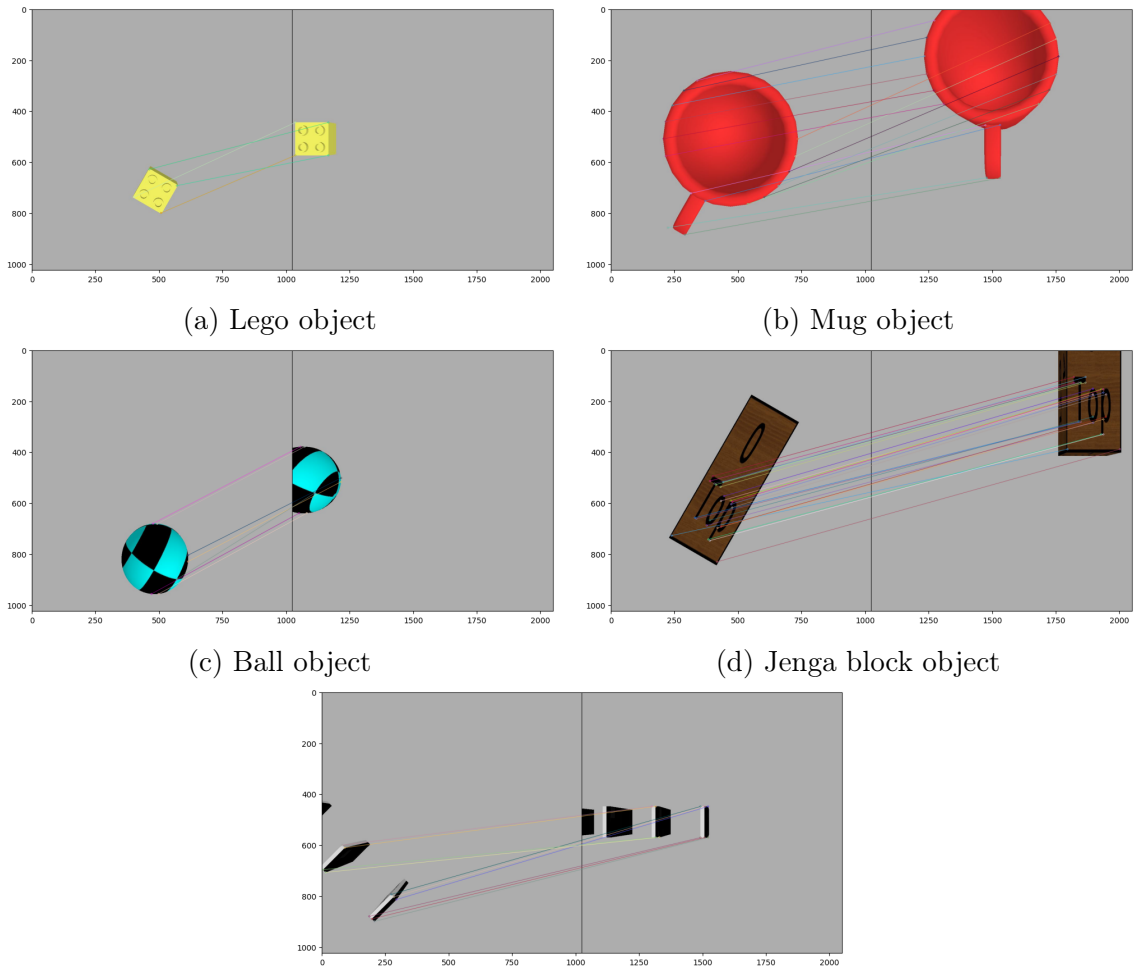
There are two additional types of constraints we can add. Firstly we can impose limits on each of the joints. This defines the maximum and minimum joint angles which can be achieved by a joint. For example, most joints will have a range of $-\pi$ to π. This helps prevent an arm link from getting too close to other links by limiting its range of motion. The second type of constraint is to optimise the distance to a rest pose. The rest pose defines the joint angles in a typical neutral pose for the robot. When we try to move the end effector to a new position, the IK system will try to keep the joints as similar to this rest pose as possible, while still achieving the desired end effector position and orientation. This helps to prevent the IK system from choosing an unexpected pose to solve the system. The solution will be the one which looks most similar to the typical rest pose. We can dynamically decide this rest pose to influence the resulting solution. For example, defining a rest pose which keeps all links of the arm high, to prevent it from knocking over other objects on a table.

This is referred to as null space inverse kinematics [37]. We have a primary goal, to reach a desired end effector position and orientation. As mentioned the system is underdetermined, so there exist many possible solutions. The secondary goal is then achieved by operating within the null space of the first system. By definition, the null space, or kernel of the primary system is the solution space which gets mapped to the zero vector. This means that we can further refine a solution to the secondary goal, without impacting the primary goal, since all the changes we make necessarily have a result of 0 on the primary goal.

Appendix B

Evaluation Test Suite

This appendix details specifics about the test suite used in Chapter 4.



(a) Lego object

(b) Mug object

(c) Ball object

(d) Jenga block object

(e) Domino objects

Figure B.1: Live and demonstration images used in the test suite, with human marked keypoint matches

By analysing these images we can understand why some of the objects are more sensitive to noise, as explored in Section 4.1.

Lego object

In this object the keypoints are marked as the 4 corners. It is very easy to identify these even when the object was rotated. This explains why this object had the lowest systematic error (as discussed in Section 4.1) because the human error was very close to 0.

Mug object

In this object the keypoints are marked at even intervals around the mug, as well as the corners of the handle. However, in the demonstration image, the top of the mug is out of frame. This reduces how many keypoints we can identify. The circular nature of this object makes placing keypoints in the exact correct position more challenging.

Ball Object

This object is surprisingly difficult to mark keypoints given it is a perfect sphere. As such keypoints move in difficult to predict ways for a human keypoint matcher. Furthermore, the pattern of the ball changes quite significantly with the perspective of the camera, even though no rotation was applied.

Jenga block object

This object experienced the largest amount of systematic error by a large margin in Section 4.1. Initially one might think that it should be easy to identify keypoints. Like with the Lego object, we may try to mark the corners of the block. However, in the demonstration image the top of the block is out of frame. This is deliberate to make the test more challenging for the system. As a result we can only mark the bottom two corners as keypoints. Instead we match identifiable points of the text on the object. This allows us to manually identify far more keypoints, the most of any object at 22.

Dominoes

This 'object' is interesting because it is actually not a single object but multiple. The rotation applied in Table 4.1 is not applied about the centre like other objects. The line is rotated about the first domino. This presents some challenges for the robot since in order to knock over the domino line successfully it needs to rotate the vector it approaches at, unlike most objects where the angle of approach does not matter, provided the end effector is spun to the correct angle.

In this test we can only identify a small number of keypoints since the majority of the domino line is out if frame.