Gripper Grasping Position Classifier

Xavier Parker & Aryan Rishi

December 2025

1 Introduction

The aim of this project is twofold:

- To simulate the grasp of two grippers from various starting positions on two objects.
- To train a classifier to determine whether a grasp attempt will be successful or fail based on its starting position.

Our code can be found here: https://github.com/MightyBrushwagg/COMP0213_Group04

1.1 Pipeline Overview

The pipeline consists of generating starting positions using a spherical distribution with added noise; the orientation of the gripper is determined by the starting positions of the gripper and object. The gripper then simulates whether the grasp will successfully hold the object off the ground and records the outcome as labelling for the starting positions. This data is then used to train a classifier to predict the likelihood of a successful grasp. To test this classifier: 10 grasps are performed and their success is also recorded, we then feed those 10 grasps through the model and compare the outputs in table 4.

1.2 Pipeline Design

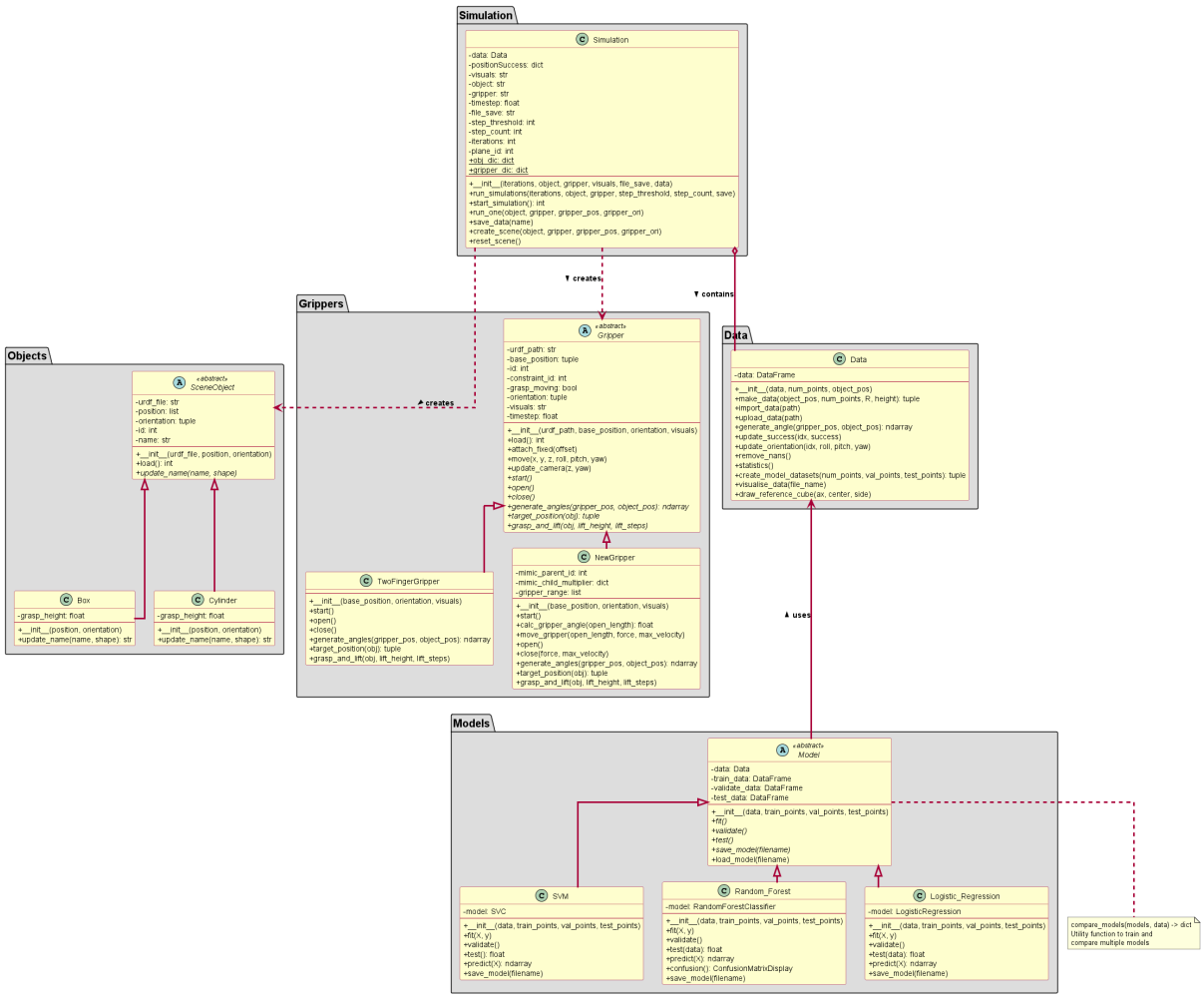


Figure 1: UML

Simulator

The accumulator class of the system is `Simulator` which organises the grippers, objects, and data. Within this class, the `PyBullet` physics simulation environment is created during the initiation step using the `start_simulation` function. The scene is then created by calling the gripper and object of choice, which are loaded in via the `Gripper` and `SceneObjects` abstract method `load` which is abstractly implemented in all grippers and objects. The initial position and orientation of the gripper is produced in the `Data` class and the object is placed at the environment origin. The `Simulator` class then performs the simulations in the `run_simulations` function, first by calling the abstract method `grasp_and_lift` which is customised in all grippers, monitors the contacts between the gripper and object as well as the object and ground, and records whether the gripper successfully held the object off of the ground. It then stores the result using the `Data`'s `update_success` function along with the position and orientation data, and resets the scene for the next iteration. The success criteria for a grasp is: it must maintain ≥ 2 contact points between gripper and object, 0 contact points between the object and ground plane, and it must maintain this for 3 seconds after reaching the target height. Once all the simulations have been run, if the parameter save is True then it saves the data in an external csv file that can be loaded in by `Data` in future runs.

Gripper

The `Gripper` class is an abstract base class which provides methods directly usable via inheritance. It is inherited by the `TwoFinger` (parallel-jaw gripper) and `NewGripper` (mimic joint gripper) classes. The base `Gripper` class loads in the gripper URDF, attaches constraints for movement, and handles camera tracking for visualisation. Each of the individual child classes initialise their joint positions, open their grippers, and calculate their approach orientation by comparing their initial position with the position of the object. The grippers then offset themselves to ensure the object is within the gripper jaws, move to the target position above the object, lower onto it, grasp firmly, then lift to a target height.

The base `Gripper` class has a set of abstract functions: start, open, close, `generate_angles`, `target_position`, `grasp_and_lift`. These functions are then customised in the children classes as each gripper required different instructions to be able to grasp successfully.

Object

The `SceneObject` abstract base class loads in the objects URDF, and generates a name for them. This class is inherited to two child classes for each object: the `Cylinder` class and the `Box` class. Each object defines a `grasp_height` attribute for optimal z-coordinates to pick them up. In addition, `SceneObject` has an abstract method `update_name` which updates the instance attribute name of each class, the children class call the parent function but pass in automatically their shape name.

Data

The `Data` handles all the data in an instance attribute called data that is a Pandas `DataFrame`. The `Data`'s `make_data` function generates random gripper positions on a spherical distribution around the object. Gaussian noise is added to make the distribution more realistic, and then the initial angle is calculated to give the gripper an initial angle to work from before it goes into its specific `grasp_and_lift` function.

`Data` has several functions that abstract away some of the data processing and visualisation. The `update_success` and `remove_nans` functions both keep the data attribute more hidden from other classes while editing common changes needed for the data attribute, the encapsulation here helped in code readability in the `Simulation` class as well as keeping the `DataFrame` storing all the data as far away from the `Simulation` class but still letting the important data be passed to and from each run. `visualise_data` and `draw_reference_cube` both aid in the visualisation of the data while keeping all the data processing within the `Data` class, abstracting it away from other classes. `upload_data` and `import_data` allow datasets to be saved and then downloaded again in future runs. The `create_model_datasets` creates the datasets needed for the classifier training by splitting the data up into 50/50 splits of successes in the designated allocation for training, validation and test datasets with error handling for when there is not enough data.

Models

The abstract class `Model` creates the training, validation and test datasets by calling the passed in data's `create_model_datasets` function and assigning them to instance attributes. `load_model` loads in a downloaded model using the `joblib` library and saves it to an instance attribute. The rest of the functions (`fit`, `validate`, `test`, `save_model`) are all abstract methods that are customised for the specific classifier classes. We used abstract methods because then it was possible to load in and train all the different types of models with the same function names, utilising polymorphism to aid in code complexity.

2 Dataset

2.1 Sampling

Points are sampled in the `make_data` function. The general idea is to create a sphere, add noise, select only valid points, and then generate orientations for all points. The sampling is done in five steps: generate a 3D standard Gaussian for each point using `np.random.normal`, normalise all points to have the same magnitude, then multiply by the radius to form the desired sphere. Next, sample 3D normally distributed noise (mean 0, standard deviation 0.02, this noise was selected as it added a bit of difference without making the challenge too hard for the gripper to grasp) and add it to the sphere points. For each point, run the `generate_angle` function, and then return all positions and orientations.

To generate the angles for each point, the following formula is used. Compute the direction vector by subtracting the current object position from the gripper position, using `NumPy` arrays for efficiency. Normalise the vector by dividing by `np.linalg.norm`, and then create the gripper offset. The pitch is given by pitch $= \arcsin(-\text{direction}[2])$, the yaw by yaw $= \arctan 2(\text{direction}[1], \text{direction}[0])$, and the roll is set to π.

2.2 Grasp execution

Once the angles for the gripper are generated, it points directly towards the object. The offset generated is designed to allow for the object to be positioned within the gripper jaws and displace the gripper a fixed amount based on the gripper size in the orientation of approach. The gripper then moves to it's offset target x and y coordinates but a fixed displacement above the target z. The gripper then lowers onto the object to the object's grasp height (or an offset of it) which is defined in the `Box` and `Cylinder` classes respectively. Once the gripper is at the desired height, it's jaws close firmly; whilst the gripper lifts and holds the object at a defined target height, it continuously reapplies a firm grip to prevent slipping (Fig. 2 & Fig. 3). All offsets were found through tuning to determine the optimal displacement for successful grasping.

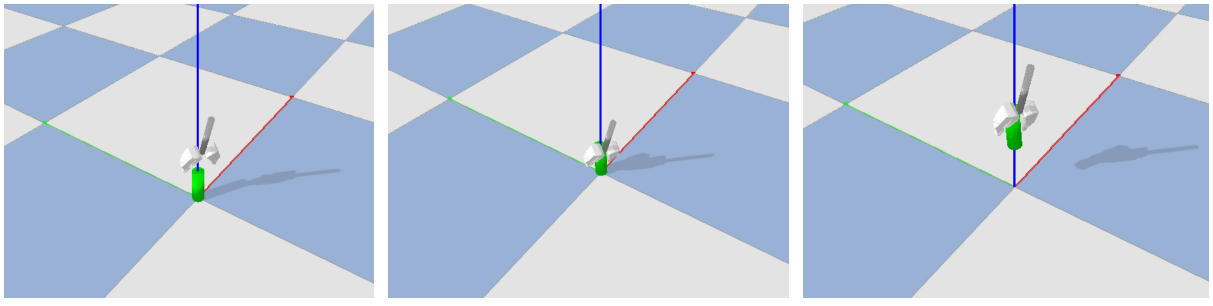


Figure 2: Two Finger Gripper Motion with Cylinder

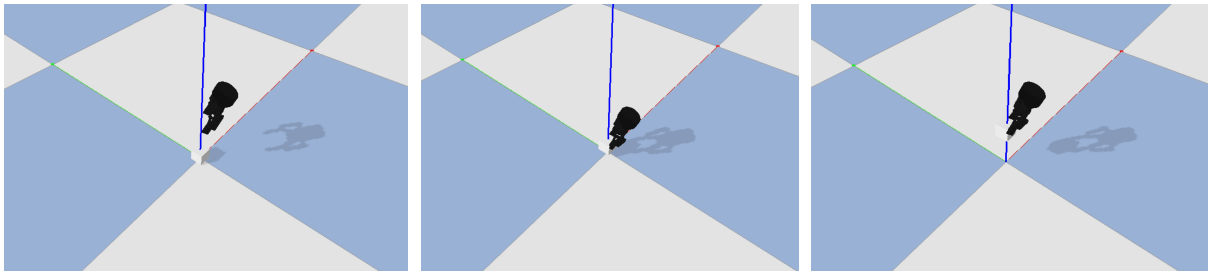


Figure 3: New Gripper Motion with Cube

2.3 Grasp determination

In the `Simulator` class, once the gripper has reached the target height, the object-gripper and object-floor contact points are measured. Whilst there are ≥ 2 contact points between gripper and object and 0 contact points between the object and ground plane, we increment a counter. If this counter reaches 720 iterations (3 seconds of gripping at 240Hz) then the grasp is labelled a success and the outcome is recorded.

2.4 Dataset generation

For the dataset generation, we followed a pipeline. First, more than enough random points around a sphere were generated using the `make_data` function; this was done by creating three times plus one hundred (to cover small cases) the desired number of iterations as around half would be below the plane from the sphere and get cut. Then, the grasp execution and determination were performed for each position one by one, updating the `successful` column in the pandas `DataFrame` storing all the data. The executions were stopped once the desired number of iterations was reached, and all rows with NaN values, corresponding to the extra generated positions, were dropped using `Data`'s `drop_nans` function. Finally, if the user set `save_data` to True in the arguments, the dataset was saved as a CSV file using the data's `upload_data` function, and a plot of successful and unsuccessful grasps was created using `matplotlib.pyplot` and saved and can be seen in Figure 4.

3 Classification

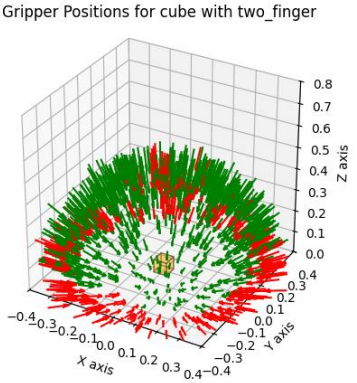
3.1 Chosen classifiers & setup

The three classifiers we went with were Logistic Regression (`sklearn.linear_model.LogisticRegression`), SVM (`sklearn.svm.SVC`) and a random forest classifier (`sklearn.ensemble.RandomForestClassifier`). All three were inherited into an abstracted Model class to grant us the freedom to use the models as we intended, with the `abstractmethods` keeping function naming and use consistent.

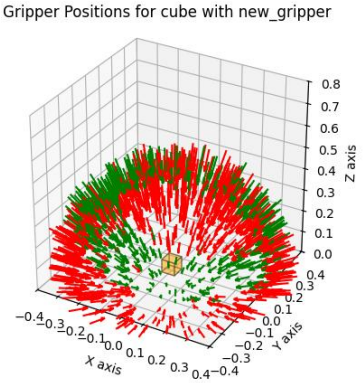
For logistic regression and SVM, there were no hyperparameters to tune, so the default models were used. However, for the random forest classifier, we tested different numbers of estimators to determine which gave the highest accuracy. The results are shown in Table 1. From these results, we selected the model with the highest test accuracy for each combination, favouring the option with fewer estimators in the case of a tie to reduce inference time.

n_estimators	Cube, two finger	Cylinder, two finger	Cube, new gripper	Cylinder, new gripper
50	83	86	90	82
100	85	88	91	83
150	84	87	92	82
200	83	87	90	83

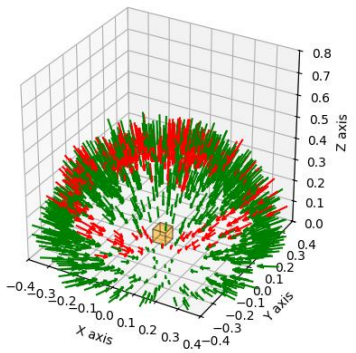
Table 1: Performance for varying numbers of estimators across different objects and grippers for random forest classification.



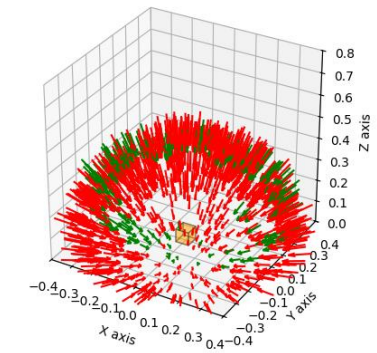
(a) 1000 grasp attempts of the two finger gripper on the cube



(b) 1000 grasp attempts of the new gripper on the cube



(c) 1000 grasp attempts of the two finger gripper on the cylinder



(d) 1000 grasp attempts with the new finger gripper on the cylinder

Figure 4: 1000 grasp attempts for each gripper/object combination. Green marks success and red marks failure.

After choosing to go with 100 estimators for the Random Forest classifier, we compared the different models that were trained and tested on the same dataset. These results can be seen in table 2 and clearly demonstrate the effectiveness that the random forest algorithm has with this problem when the logistic regression and SVM struggle with the new gripper's classifications, we think this is because there are several pockets of successful and unsuccessful rather than a distinct line, which random forest is better at picking out.

Object & Gripper	Logistic Regression	SVM	Random Forest
Cube, Two Finger	78	72	92
Cylinder, Two Finger	63	67	90
Cube, New Gripper	48	55	87
Cylinder, New Gripper	49	49	87

Table 2: Test accuracies for different objects and gripper configurations.

3.2 Training, Validation and Test sets

To create the training, validation and test datasets, Data had a function called `create_model_datasets` which split successful and unsuccessful grasps and created equally successful datasets with the desired number of points for training, validating and testing. If there were not enough points, then the function raised a ValueError saying that there was not enough data. Once the model is trained then it saved

using the `save_model` function in the Model class and can be loaded in future runs using the `load_model` function.

3.3 Average performance

After deciding that 100 estimators produced the best results, we trained three models using the same dataset but it is shuffled each time so a different 120 points are used each time for training, we decided on this number so that we did not over fit the model, and then a different 100 points are used for testing, we chose 100 points as this would give a good reflection on the strength of the model, this required 110 successful and 110 unsuccessful points to make the datasets, to guarantee this we ran each combination for 1000 attempts. From table 3 we can see only once did the accuracy drop below 80%, demonstrating the power of the random forest classifier algorithm with this type of problem and the generalisation of our datasets.

Object & Gripper	**Trial 1**	**Trial 2**	**Trial 3**	**Average**
Cube, two finger	85	90	85	86.67
Cylinder, two finger	93	82	95	90.00
Cube, new gripper	97	93	97	95.67
Cylinder, new gripper	72	85	82	79.67

Table 3: Success rates across three trials for different object-gripper combinations with a random forest classifier (100 estimators). All trials trained using the same data.

3.4 Predictions

To test the classifiers on new predictions, the "test" mode was enabled. First, a model was trained and saved for each classifier. Then, a dataset containing only 10 points was made using the run mode and a custom filename "validation.csv", after which we passed those points into the respective saved models and recorded the accuracy on the 10 samples, which can be seen in table 4.

With these results, we can be confident that our classifier is good at detecting which positions will be successful or not.

Gripper–Object Combination	**Accuracy**
Cube, two_finger	10/10
Cylinder, two_finger	8/10
Cube, new_gripper	8/10
Cylinder, new_gripper	7/10

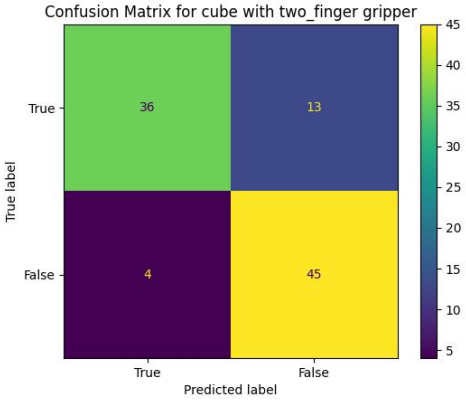
Table 4: Accuracy of grasp success for each gripper–object pairing on 10 new grasps.

3.5 Evaluation metrics

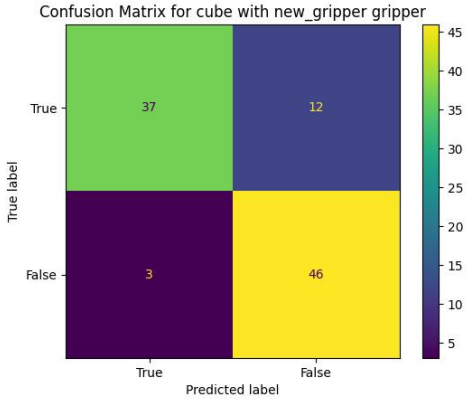
As can be seen in the confusion matrices in figure 5, the models correctly classify most of the time with a tendency to classify negatively when wrong, leading to more false negatives than positives. If this was being used to train a reinforcement learning policy or other grasping policies then this could produce a reliable grasping outcome as it would not think it succeeded when it did not.

4 Discussion and Conclusion

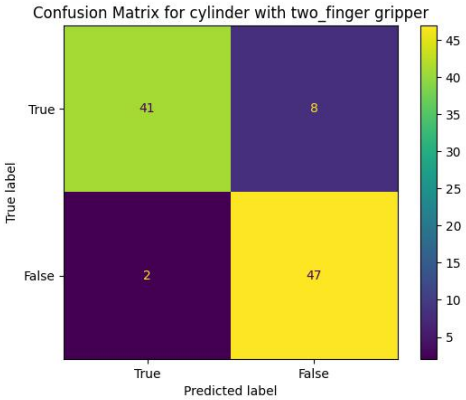
Overall, we successfully implemented the full end to end pipeline with multiple modes for generating data, training or utilising the model. Our biggest strength was planning out every class before starting to write code. This allowed us to split up the tasks and both work on different parts of the project seamlessly and straightforward integration of the two parts. One of the biggest challenges we faced was using GitHub and proper source control, more than once we lost some work as GitHub overwrote the changes before we saved. Another challenge we encountered was getting the new gripper to work as



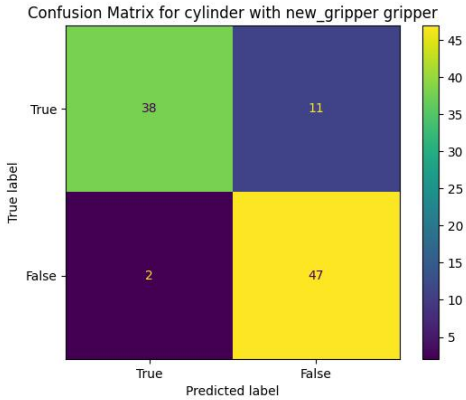
(a) Confusion matrix on 100 test samples with the random forest classifier on the two finger gripper and cube dataset



(b) Confusion matrix on 100 test samples with the random forest classifier on the new gripper and cube dataset



(c) Confusion matrix on 100 test samples with the random forest classifier on the two finger gripper and cylinder dataset



(d) Confusion matrix on 100 test samples with the random forest classifier on the new gripper and cylinder dataset

Figure 5: Confusion matrices on 100 test samples with the random forest classifier with the dour models trained with their respective gripper and object datasets.

intended and although we had moderate success with the new gripper on the cube, it still struggles with the cylinder. However, because of the strength of our random forest classifiers, the new gripper and cylinder make definable areas that the can be picked up in training.