MASSEY UNIVERSITY
TE KUNENGA KI PŪREHUROA

A MACHINE LEARNING BASED ROBOTIC SYSTEM FOR APPLE PICKING

A report presented
in partial fulfillment of the requirements
for the degree of Master of Engineering Studies (Mechatronics)
at Massey University

VARUN CHANDRA SHEKAR

School of Food & Advanced Technology
Massey University
Auckland, New Zealand

2020

ACKNOWLEDGEMENTS

This research report has been a challenging journey, one that tested not only my academic capabilities, but also mental fortitude. It has been far from easy to put in the hours to complete this project, all while a pandemic threatens those you care about back home and around the world. In this regard, I would like to show my gratitude to all those who directly or indirectly supported me in these hard times. I hope we come out of this global tragedy as a stronger and more compassionate society.

I am deeply grateful to my mentor and supervisor, Dr. Frazer K. Noble, who's passion for teaching and constant encouragement helped me persevere through this course and find my own passion for robotics and machine learning.

I would like to thank Dr. Khalid Arif for opening my mind to new ways of looking at robots and their capabilities.

To Mr. Kartikay Lal, Mr. Gabriel Phang, Mr. Blair Dixon and all supporting staff at the university, I am thankful to you for being patient while I tried my hand at building my projects in the labs and workshops. I would also like to thank Massey University Health & Counselling for helping me stay healthy enough to make it through stressful times.

To my all my housemates and friends, your kindness and understanding during my time at Massey have been a great source of strength.

Finally, I would like to thank my family here in New Zealand and back home in India. I could not have dreamed of being where I am today and achieving all that I have, had you not been an eternal source of unconditional love, had you not been there for me when days seemed dark and daunting, had you not reassured me things would get better.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

Over the past three years, New Zealand has been facing an acute shortage of labor in the fruit picking sector. For example, in March 2018, only 14 people expressed interest in response to a desperate call for more fruit pickers at Hawke's Bay [1]. The Government of New Zealand was forced to step in and raise the cap on number of seasonal immigrant workers [2].

The situation was not too different in 2019 [3]. Despite a bountiful apple season and strong annual growth, there are few people willing to work in this area [4]. Even though there are more people working on fruit farms and orchards, the numbers are insufficient to meet the need for laborers to harvest increasing volumes of produce. Farmers voiced their concerns, warning that if long-term solutions were not found, a sharp decline in quality, revenue drops, and stagnation in growth was imminent [5]. Unfortunately, an unprecedented global lockdown owing to the COVID-19 pandemic further hurt kiwi fruit and apple farmers, who are now looking towards New Zealanders to help fill labor shortages.

One part of the solution towards meeting the demand for labor would be to use industrial robots for fruit picking and other manual tasks. The robotics sector has been consistently growing over the past decade. The sales of robots in general are predicted to go from 254K units in 2015 to 630K in 2021 [6]. Industrial robots have been used to automate tasks for many decades but were not commonly used on farmlands. This is mainly because robots are precise and efficient tools that are explicitly programmed to perform a specific sequence of actions. A farm on the other hand is a very difficult area to tackle for a programmed system, owing to the extreme set of variations in how plants grow, what the soil conditions may be or what part of the plant fruits could grow on. All these variables are difficult to predict and thereby make it nearly impossible for a programmer to address on a case by case basis.

A great deal of effort and engineering is required to ensure that an automated device operates in a safe manner while preventing damage to people, property and goods. Therefore, robots are popular in facilities which are planned and engineered in such a way that the robot's work area does not overlap or interfere with that of people. When a robot works alongside or nearby humans, it is either remotely controlled by an operator, or operated by hand. This is where Artificial Intelligence (AI) can help by automating several processes. AI is the study of ‘agents’ who acquire information from the environment and performs actions that maximize their chance of successfully achieving their goals.

The main principle that makes AI, and in turn machine learning possible is the neural network. A neural network is a computing system designed to function like neurons in a human brain. Similar to how a human improves their performance of a task with practice, a neural network strengthens and weakens different connection weights during its training to achieve a predefined goal. This technology enables us to design algorithms which identify patterns or iteratively improve the performance of tasks when trained with large amounts of data.

For example, in case of an apple detector, the vision system should be able to clearly distinguish between an apple, an object that is not an apple and the surroundings. To achieve this, the network is trained with many images, apples clearly labelled on each of them. The network breaks down the images and learns different features that make up apples. The kind of features learned depends entirely on the architecture of the network.

Thanks to rapid improvements in storage, processing, sensors and machine learning algorithms, there has been an explosive growth in innovation and investment from researchers and corporations around the world. According to IFI-Claims, the number of Machine Learning patents grew at a 34% Compound Annual Growth Rate between 2013 and 2017, the third fastest growing category among all patents [7]. The investment in AI technologies is predicted to go up from $12B USD in 2017 to $77.6B USD in 2022 [8].

The trend is similar for robots in the farming sector as well. According to Omdia Tractica (a market intelligence firm focusing on emerging technologies), farming robot shipments are growing exponentially - from around 100,000 units in 2019, set to reach 727,000 units annually by 2025 [9].

AIM

The aim of this research project is to develop a machine learning based robotic system for use in apple picking.

OBJECTIVES

The objectives of this research project are to:

- Implement a machine learning algorithm that can use visual data to perceive and identify apples.
- Develop a program that processes the visual data to compute the location of an Apple in 3D space.
- Implement a robotic setup in a simulated environment that can use results generated by the above steps and approach a target point.

REPORT STRUCTURE

First, a systematic and thorough review of the literature related to machine learning, machine vision, robotics and robot learning is carried out. In the second section, the components and materials required to achieve the objectives of the research work are briefly discussed. The methodology of the work and all relevant details are comprehensively laid out in Section 3. The capabilities of each part of the system is determined by performing multiple tests. The results for these tests and the outcome of running the complete system are presented in Section 4. In Section 5, some of the shortcomings of the research project are briefly discussed, followed by suggestions to improve the system in Section 6. Section 7 summarizes the report and discusses its contributions.

1 LITERATURE REVIEW

In the first section of the literature review, a brief overview of machine learning is presented. In the second section, we analyze literature on computer vision using machine learning for object detection and instance segmentation. Finally, we analyze the literature on robotics & robotic learning, a field where robotic manipulation is achieved using control theory, machine learning and/or machine vision.

1.1 MACHINE LEARNING & COMPUTER VISION

One of the techniques that makes machine learning a reality is the Artificial Neural Network (ANN). An ANN is a computing structure vaguely resembling the neural structure in an animal's brain. A neural network uses the principle of backpropagation of errors to adjust the way it processes information to achieve the intended output. The "Neocognitron", proposed by Fukushima [10] is a multilayered ANN which later inspired the development of the Convolutional Neural Network (CNN) by LeCun [11]. A CNN is adept at classifying visual patterns, and therefore formed the foundational approach for Computer Vision.

AlexNet is one of the popular works that kindled modern AI research, making GPUs and Convolutional Neural Networks the go-to toolkit for a variety of machine learning tasks [12], mainly computer vision. Their novel architecture achieved the best results for object detection on the ImageNet dataset in 2010. The network required 1.2 million images and two GPU's to train. This study showed us that favorable results can be obtained on challenging datasets using deep networks and high computational power.

As defined by the Automated Imaging Association (AIA) [13], *"Machine vision encompasses all industrial and non-industrial applications in which a combination of hardware and software provide operational guidance to devices in the execution of their functions based on the capture and processing of images."* The software techniques in machine vision that are of interest in this project are object detection, semantic segmentation and instance segmentation. Object detection networks scan an image and return a bounding shape and corresponding label around different instances of objects it has been trained to detect. Semantic segmentation is the process of associating each pixel of an image with a class label, while instance segmentation is the process of detecting the same instance of an object. There are a multitude of works that implement these techniques in different ways. Some of the popular studies are discussed below.

Long et al. [14] developed a Fully Convolutional Network (FCN) performing pixel-to-pixel semantic segmentation using supervised pre-training, without requiring the use of pre and post-processing complications, such as super pixels, proposals, or post-hoc refinement by random fields or local classifiers. This approach simplifies the detection process, reduces the time needed to infer objects and provided the best segmentation and scene sampling results when applied on PASCAL-VOC 2011 and 2012 datasets.

Ronneberger, et al. [15] present a novel network and training strategy called U-Net which builds upon Fully Convolutional Networks to utilize image samples more efficiently by heavily relying on image augmentation techniques, thereby substantially reducing the number of training images required.

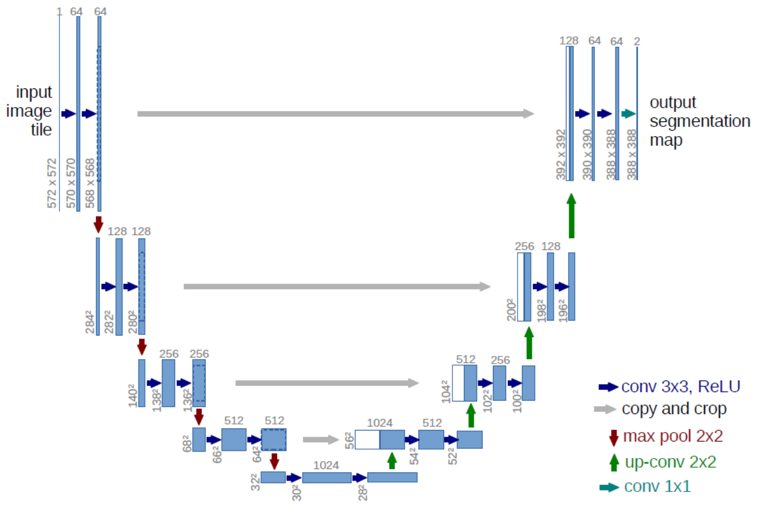


Figure 1: The U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

(Source: [15])

The U-shaped architecture is comprised of two paths - a contracting path and an expansive path. The contracting path consists of repeated 3x3 unpadded convolutions each followed by a rectified linear unit (ReLU) and a 2x2 max pooling operation with stride 2 for down sampling. At each down sampling step, the number of feature channels is doubled. Every step in the expansive path consists of an up sampling of the feature map followed by a 2x2 convolution that halves the number of feature channels, a concatenation with the correspondingly cropped feature map from the contracting path, and two 3x3 convolutions, each followed by a ReLU. The cropping is necessary due to the loss of border pixels in every convolution. At the final layer a 1x1 convolution is used to map each 64-component feature vector to the desired number of classes. In total the network has 23 convolutional layers.

The input images and their corresponding segmentation maps are used to train the network with the stochastic gradient descent implementation. To minimize the overhead and make maximum use of the GPU memory, large input tiles are favored over a large batch size and hence reduce the batch to a single image. A high momentum is used so that many of the previously seen training samples determine the update in the current optimization step.

Invariance and robustness can be achieved in cases where there are few training samples by applying appropriate data augmentation.

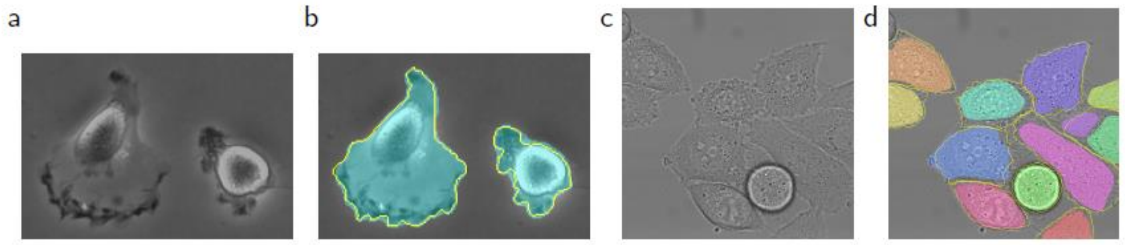


Figure 2: Segmentation results (b, d) as seen on an image from (a) PhC-U373 data set; (c) DIC-HeLa data set respectively (Source: [15])

Name	PhC-U373	DIC-HeLa
IMCB-SG (2014)	0.2669	0.2935
KTH-SE (2014)	0.7953	0.4607
HOUS-US (2014)	0.5323	-
second-best 2015	0.83	0.46
u-net (2015)	**0.9203**	**0.7756**

Table 1: Segmentation results on ISBI cell tracking challenge 2015

(Source: [15])

U-Net achieves very good segmentation results in bio-medical applications and needs only few annotated images to train. This can be adapted and applied on a variety of other image types.

To achieve object instance segmentation (a combination of object detection and semantic segmentation), He et al. [16] developed the Mask Region-CNN (Mask R-CNN) architecture, adding object masks and pixel-to-pixel alignment to Faster R-CNN's class label and bounding box offset.

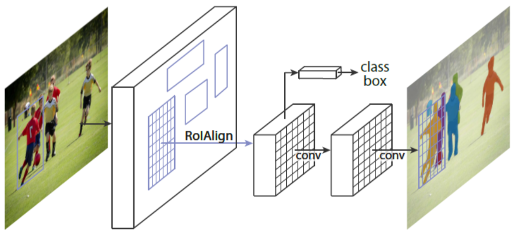


Figure 3: Mask R-CNN framework for instance segmentation

(Source: [16])

Faster R-CNN consists of two stages. The first stage, called a Region Proposal Network (RPN), proposes candidate object bounding boxes. The second stage extracts features using Region of Interest (RoI) Pool from each candidate box and performs classification and bounding-box regression. The features used by both stages can be shared for faster inference. Mask R-CNN has an identical first stage (the RPN). In the second stage, Mask R-CNN returns a binary mask for each RoI, applying bounding-box classification and regression in parallel.

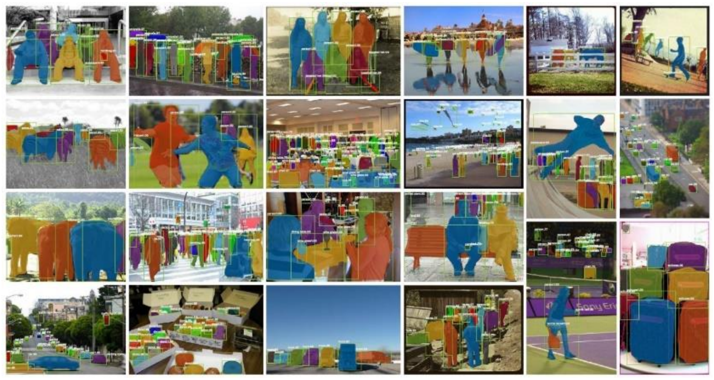


Figure 4: Mask R-CNN output on COCO (Common Objects in Context) test images. Masks are shown in color, along with bounding box, category and confidences

(Source: [16])

	backbone	AP	AP_{50}	AP_{75}	AP_S	AP_M	AP_L
MNC [7]	ResNet-101-C4	24.6	44.3	24.8	4.7	25.9	43.6
FCIS [21] +OHEM	ResNet-101-C5-dilated	29.2	49.5	-	7.1	31.3	50.0
FCIS+++ [21] +OHEM	ResNet-101-C5-dilated	33.6	54.5	-	-	-	-
Mask R-CNN	ResNet-101-C4	33.1	54.9	34.8	12.1	35.6	51.1
Mask R-CNN	ResNet-101-FPN	35.7	58.0	37.8	15.5	38.1	52.4
Mask R-CNN	ResNeXt-101-FPN	**37.1**	**60.0**	**39.4**	**16.9**	**39.9**	**53.5**

Table 2: Comparison of Mask R-CNN performance compared to MNC and FCIS, previous winners of COCO 2015 and 2016 segmentation challenges respectively.

(Source: [16])

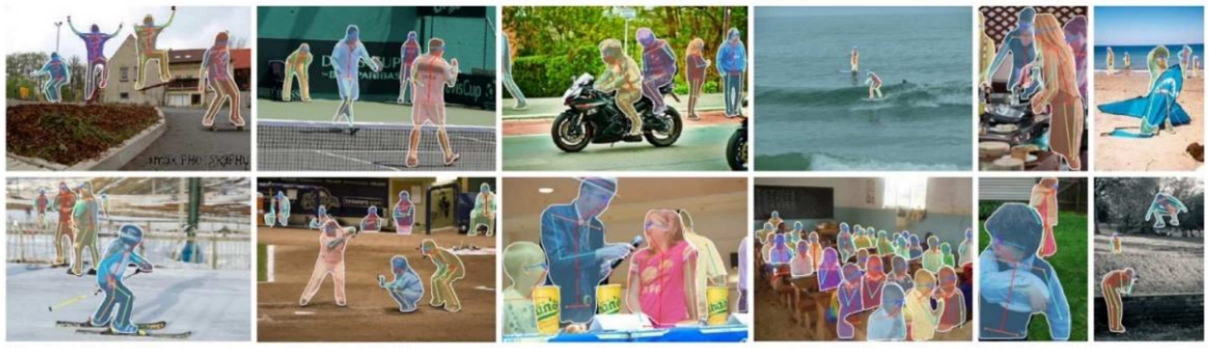


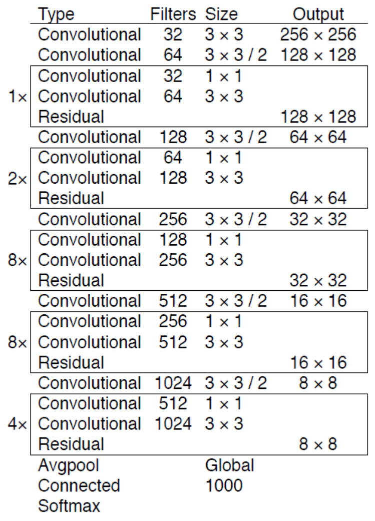
Figure 5: Human Pose Estimation: Key point detection results on COCO images

(Source: [16])

This framework is simple, flexible and can be extended for use in human pose estimation, as demonstrated by the researchers. This study used only minimal domain knowledge of human pose. Provided appropriate training, the network can be improved to better decipher human pose information, which could be a necessity in domestic applications like old-age care and high-risk environments where a robot may be dealing with hazardous materials around humans.

YOLO (You Only Look Once) is a popular object detection architecture that was first developed by Redmon et al. in 2015 [17], then further refined in 2016 (called YOLO9000 [18]) and in 2018 (called YOLOv3 [19]). It is commonly used for its high speed and accuracy. YOLO can be used to achieve object detection at real time speeds in videos.

The YOLO-v3 architecture is quite simple in comparison to other popular detection since object detection is modeled as a single regression problem, i.e. logistic regression is used to predict objectness scores. The network consists of a single CNN (called Darknet-53) that predicts bounding boxes and class probabilities for each box. Each image need only be fed to the trained network during test time and results are presented. YOLO sees the entire image during training and test time, so it implicitly encodes contextual information about classes as well as their appearance and it learns generalizable representations of objects. When trained on natural images and tested on artwork, YOLO outperforms top detection methods like DPM and R-CNN by a wide margin.



	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure 6: Darknet-53

Source: [19]

YOLO-v3 returns five bounding boxes for each grid cell along with an object probability score for each box. It also provides two class probabilities for each grid. Instead of predicting the absolute coordinates of the bounding box centers, YOLO-v3 predicts an offset relative to the coordinates of the grid cell, where (0,0) means top left of that cell and (1,1) means the bottom right. It is trained to predict bounding boxes whose center lies in that cell. We get four coordinates for each bounding box t_X, t_Y, t_W and t_H. The cell is offset from the top left corner of the image by (c_x, c_y) and the bounding box has width and height P_w, P_H then the predictions correspond to the relations for b_X, b_Y, b_W and b_H as shown in the figure below: -

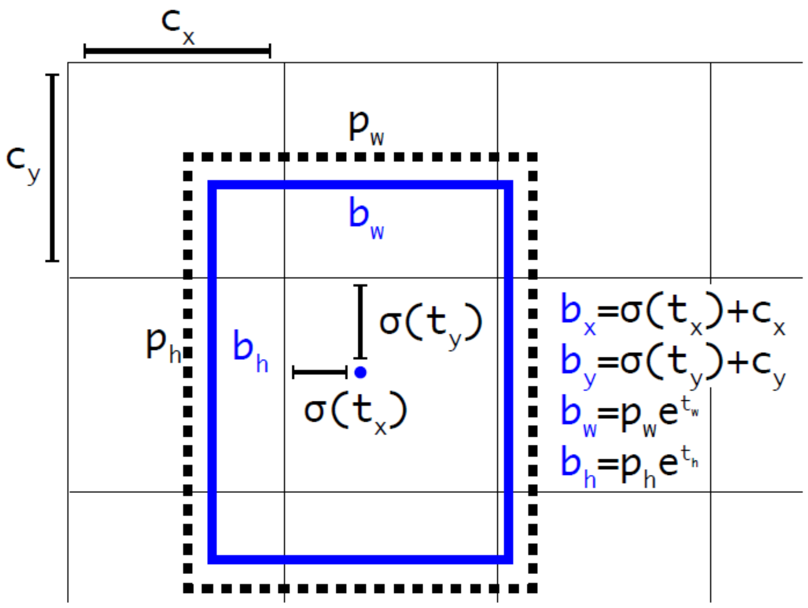


Figure 7: Bounding boxes with dimension priors and location prediction. The width and height of the box are predicted as offsets from cluster centroids. The center coordinates of the box are predicted using a sigmoid function, relative to the location where the filter is applied.

(Source: [19])

Some of the other notable studies in machine vision are briefly discussed below:

1. Noh et al. [20] address some of the shortcomings in Long's study [14], such as the fact that FCNs have a fixed size receptive field. This means if the object is larger or smaller than the receptive field, then the object itself, along with other features which may belong to this object could also be mislabeled. This is ameliorated using a deconvolution network, which brings the added benefit of improved detection of finer features.
2. Chen et al. have published multiple studies. Some of their contributions are as follows: -

    a) DeepLab [21] increases field of view of filters to include larger context, robustly segment objects at different scales and improve the localization of object boundaries by combining methods from DCNNs and probabilistic graphical models
    b) Developed MaskLab [22] which performs foreground/background segmentation by combining semantic and direction prediction (which helps in separating instances of the same semantic class)
3. Pyramid Scene Parsing (PSP) Net (Zhao et al. [23]), augments the U-Net with a Pyramid Pooling Module and manages to overcome the fact that FCNs are lacking in case of scenes with open vocabulary and high object density.

1.2 ROBOTICS & ROBOT LEARNING

Robotics is a field of science and engineering that deals with the design, construction and control of robots. The goal of robotic control is formulating control parameters such that the motor commands efficiently use all degrees of freedom and joint parameters to accurately perform a task, while minimizing the amount of movement and power consumed. This provides fine control, which is important in making the robot a safe part of the human environment. The disadvantage of such a control strategy is that it is very sensitive to the quality of the knowledge of the kinematic and dynamic parameters.

Robot learning seeks to eliminate the need to program a robot from scratch. Robot Learning is an emerging research field which employs both the principles of robotic control and machine learning for developing autonomous devices to perform actions such as object categorization, grasping, locomotion, etc.

Peters and Schaal [24] present a general learning solution for Operational Space Control (Khatib, [25]) that does not require prior knowledge of a robotic system. This involves first modeling operational space control as an optimal control problem. The robot learns to solve this problem by applying reinforcement learning using reward weighted regression.

The authors conducted an experiment in a simulated environment on two robot arms. An initial control policy and initial training data is established in each learning model using random point-to-point movements in joint space using PD control. The measured end effector acceleration served as desired acceleration while remaining controller variables were also measurable. Subsequently, the learning controller was used on-policy with the normally distributed actuator noise serving as exploration.

Both robots were reported to track desired trajectories with high accuracy.

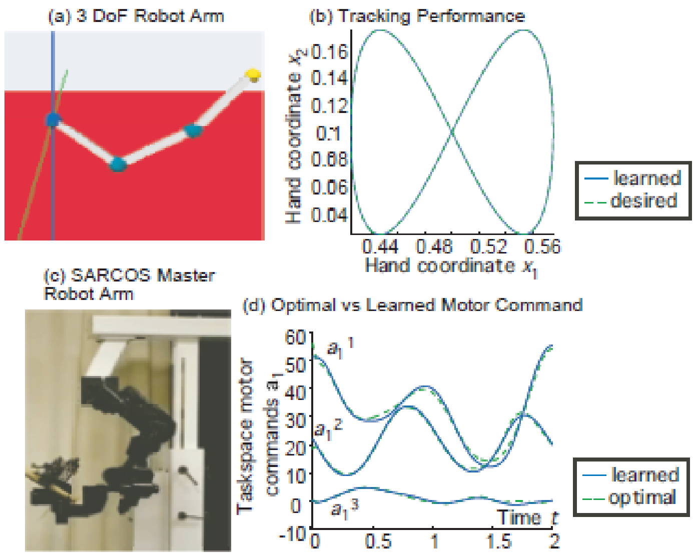


Figure 8: (a) Simulation of 3DoF planar arm (c) SARCOS robotic arm (b) Tracking performance for a planar figure-8 pattern for 3DoF arm (d) comparison between analytically obtained optimal control commands in comparison to the learned ones for one figure-8 cycle of the 3DoF arm

Source: [24]

This study showed that machine learning can be applied to teach a robot how to move optimally, but what about being able to move and orient itself towards an object it can see?

Saxena et al. [26] demonstrated how a robot can move towards an object and orient itself for grasping simply using two or more 2D images. First, a 2D image of the object to be grasped is obtained using a standard stereo camera. This image is fed to an algorithm which triangulates the 3D grasping orientation using a secondary camera mounted to the end effector. The network for this project is trained (supervised training) using simple 3D modeled images of objects with labels for the best two finger pinch-grasping points explicitly labelled on models.

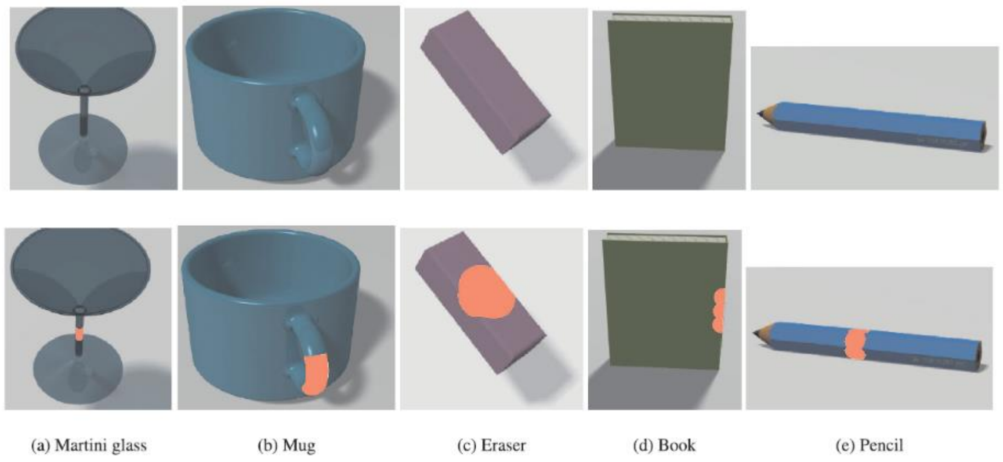


Figure 9:The five object classes used for training, and their corresponding labels identifying best grasping points [26]

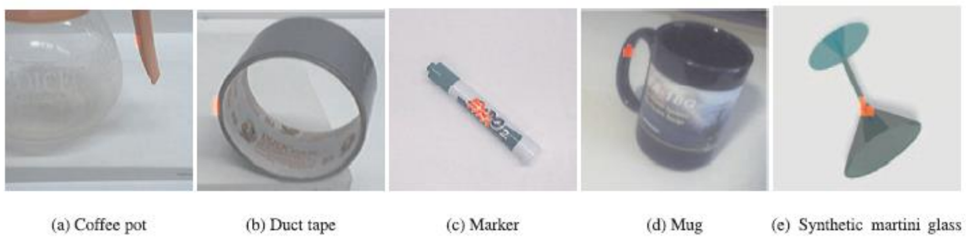
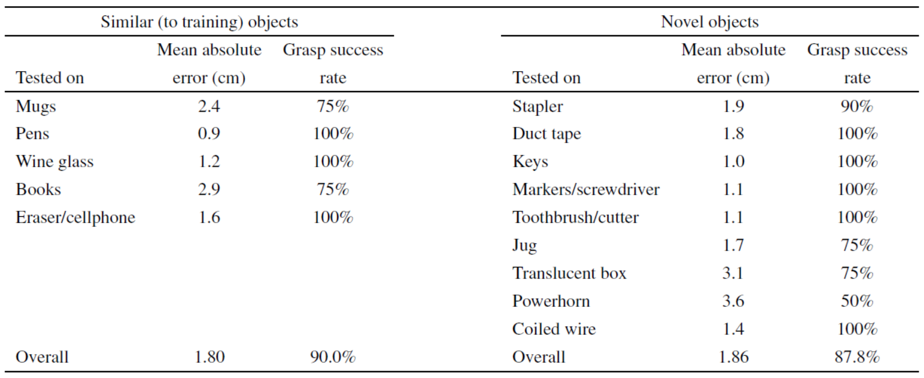


Figure 10: Grasp point classification. The highlighted areas are most likely to be identified as a grasping point [26]



Similar (to training) objects			Novel objects		
Tested on	Mean absolute error (cm)	Grasp success rate	Tested on	Mean absolute error (cm)	Grasp success rate
Mugs	2.4	75%	Stapler	1.9	90%
Pens	0.9	100%	Duct tape	1.8	100%
Wine glass	1.2	100%	Keys	1.0	100%
Books	2.9	75%	Markers/screwdriver	1.1	100%
Eraser/cellphone	1.6	100%	Toothbrush/cutter	1.1	100%
			Jug	1.7	75%
			Translucent box	3.1	75%
			Powerhorn	3.6	50%
			Coiled wire	1.4	100%
Overall	1.80	90.0%	Overall	1.86	87.8%

Table 3: Mean absolute error in locating the grasping point for different objects, as well as grasp success rate for picking up the different objects using the robotic arm [26]

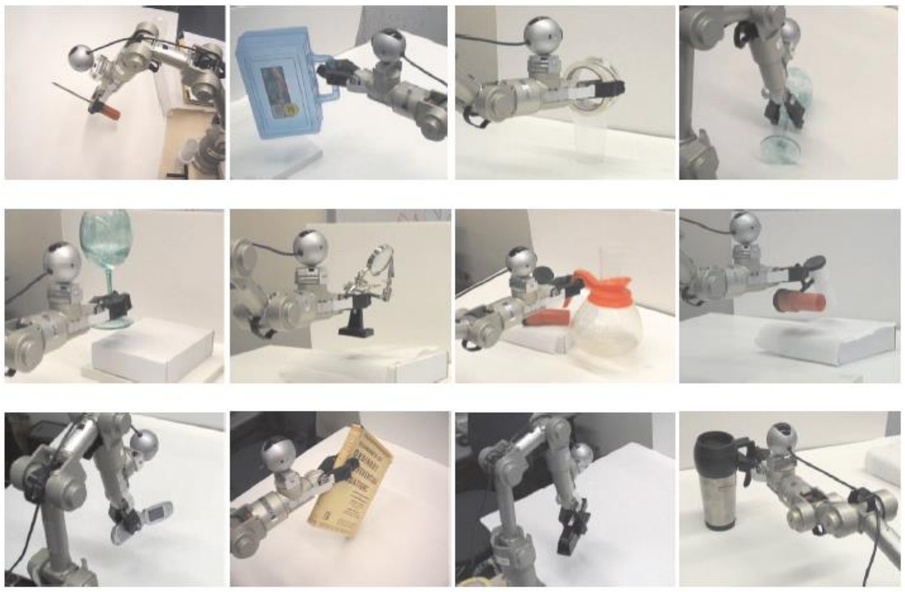


Figure 11: The robotic arm picking up various objects [26]

The next study by Lin et al. [27] uses recent machine vision tools to determine the pose of an object using RGB-D cameras, objects in this case being guavas on a tree. A robot arm is then used to pick the guavas while avoiding the parent branches.

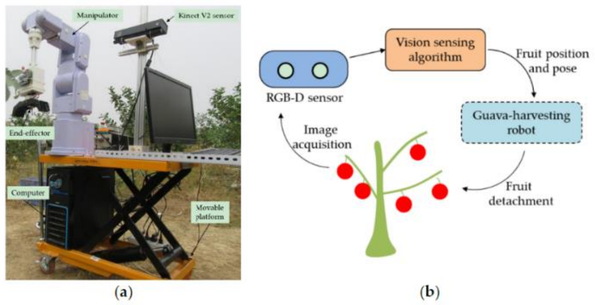


Figure 12: (a)Guava harvesting robot and vision sensing apparatus; (b) simple working of the system [27]

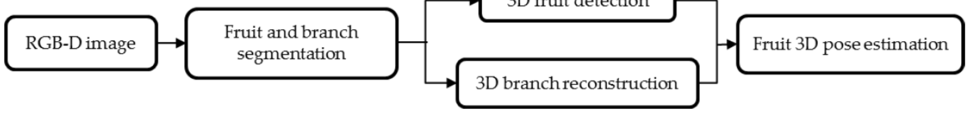


Figure 13: Flow diagram of the vision sensing algorithm [27]

The system processes the RGB-D images in the following manner

1. Use FCN model to segment guava fruits and branches (using the segmentation framework described by Long et al. in [14])

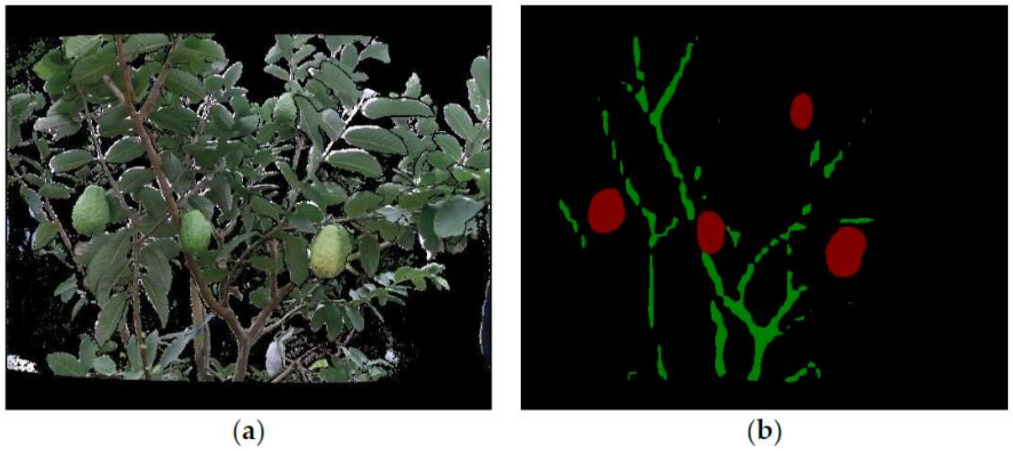


Figure 14: Segmentation results of the FCN. (a) Aligned RGB image where black pixels represent objects outside the working range of the Kinect sensor; (b) Segmentation result where red parts represent fruit and green parts represent the branches. [27]

2. Use Euclidean clustering to obtain all individual fruits from fruit point cloud. Then, determine the center point of the fruit using bounding box (uses mean as the fruit center) or sphere fitting models (uses linear least squares fitting to obtain a sphere, center of which is used as center point). Bounding box is easier to compute while sphere fitting is more accurate.

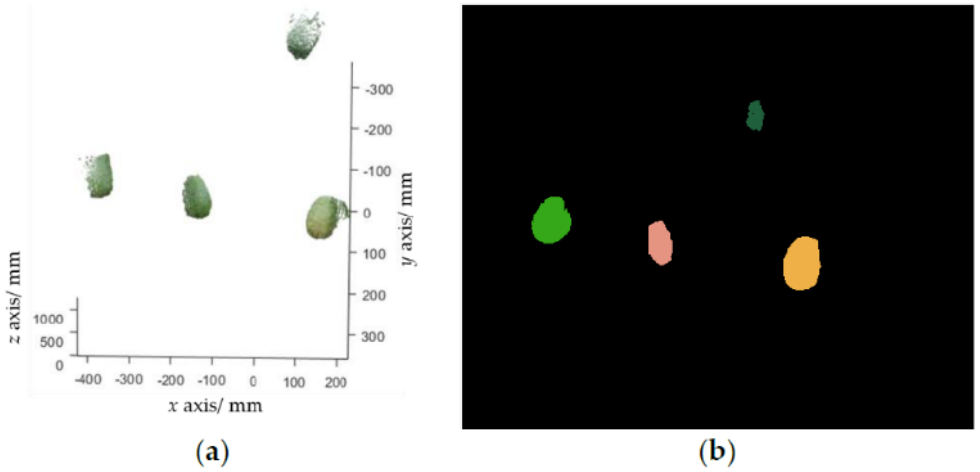


Figure 15: Fruit detection results. (a) Fruit point cloud extracted from the FCN result; (b) clustering result, where each cluster is marked with a random color. [27]

3. Presenting a multiple 3D line segments detection method to reconstruct the branches from branch point clouds

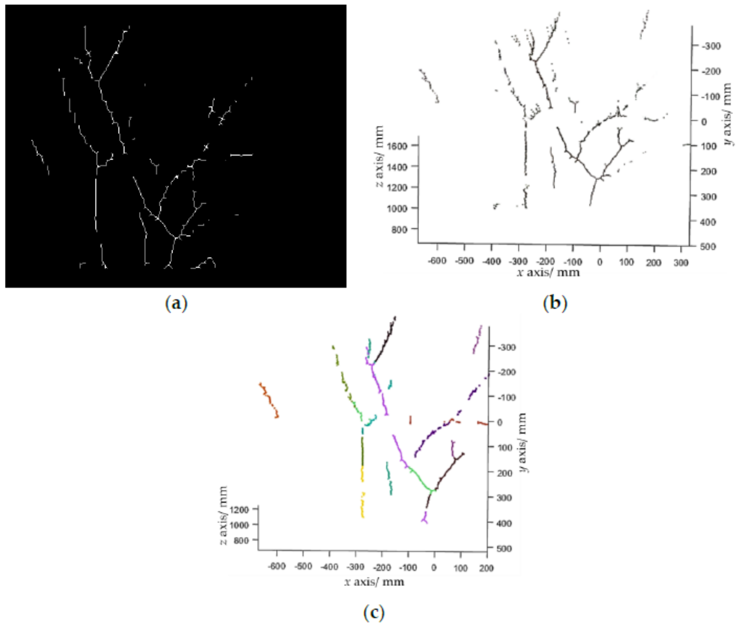


Figure 16: Branch reconstruction process. (a) branch skeletons extracted from the FCN result; (b) branch point per cloud; (c)detected line segments, each marked with a random color [27]

4. Estimating the orientation of the fruit center with respect to its mother branch to allow a robot end effector to pick the fruit along its orientation without branch collisions.

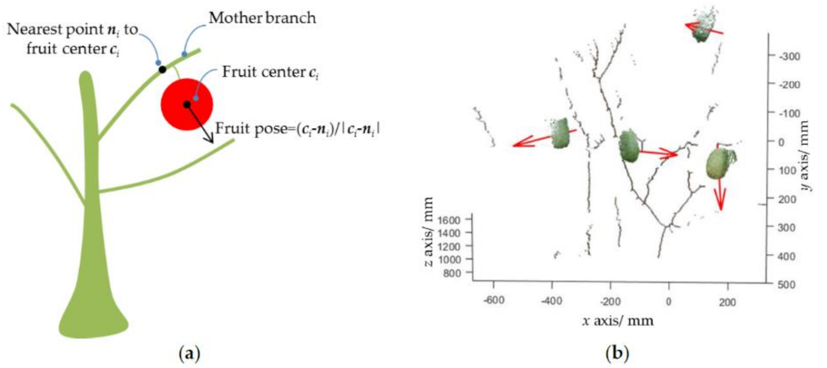

Nearest point n_i to fruit center c_i
Mother branch
Fruit center c_i
Fruit pose=$(c_i-n_i)/|c_i-n_i|$
(a)
(b)
z axis/ mm
x axis/ mm
y axis/ mm

Figure 17: (a) Principle of fruit pose estimation (b) 3D pose estimation result where red arrow represents the pose [27]

Once the nature of an object and its relationship with the surroundings is understood, it is possible to implement object manipulation strategies for robots. Huang et al. [28] propose an approach that applies deep reinforcement learning to policies which teach the robot to be gentle during exploration and task execution. The researchers state that one of the issues with directly applying deep learning on real robots is that it leads to wear and tear of both the robot and its surroundings since a large amount of trial and error is required for policy learning. In case of reinforcement learning, destructive behavior is generally manifested when the robot is trained to maximize a reward signal. The robot agent in this study needs to perform its actions in a reasonable amount of time while minimizing applied force and impact as far as possible.

By giving a robot negative rewards (or as the researchers call it, a "pain" signal), it completely avoids all collisions and does not learn to operate effectively. Therefore, to encourage gentle exploration, the robot is rewarded for "surprising" experiences, i.e. those which contradict the agent's current understanding of the world. This way, the agent makes predictions about the pain penalty that will result from a given state and action, and erroneous predictions deliver a small positive reward. The researchers suggest that this model is not too different from risky play displayed by children, which is essential to test their physical limits and learn to avoid or adapt to dangers in their environment.

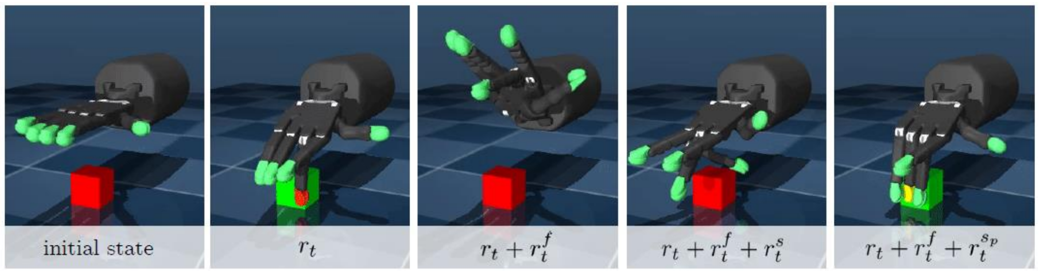


Figure 18: Simulation Task with different policies [28]

The task for the simulated robot (shown in figure 18) is to apply at least 5N force to a block; green block indicates success. The fingertip color shows the amount of impact force, yellow (near zero) to red (10N). trained on only task reward r_t learn how to do the task but use a high impact approach.

In contrast, policies trained on combination of task reward + impact penalty, and penalty-based surprise intrinsic reward ($r_t + r_t^f + r_t^s$), learn to achieve the task in a gentle way by gradually increasing the force applied to the block. Without the penalty-based surprise intrinsic reward ($r_t + r_f$), policies get trapped in local optimum where they avoid contact with the object.

This study paves the way for new research towards studying intrinsic motivations for deep reinforcement learning agents, training robots for complex tasks and opens the possibility of making large autonomous machines a smart and safe companion in domestic environments.

Hämäläinen et al. [29] apply a deep learning network with modular architecture (perception, policy and trajectory generation modules) to a robot. Each module is a distinct neural network trained using synthetic data or simulation. Information is exchanged using low dimensional representations of affordances and trajectories. Visual affordances enable the intermediate representation to capture semantic information regarding what can be done with each part of each object, which creates generality across different tasks.

The perception part is a variational encoder-decoder structure trained for affordance detection from synthetic RGB images. Those images are generated using randomized textures, object shapes, distractor objects and camera viewpoints to maximize the model's capability to generalize to new environments. The encoder part outputs a low-dimensional representation of scene affordances. The trajectory part is a variational auto-encoder trained on a set of trajectories. The policy, trained in a physics simulator, maps latent affordance representations to latent representations of trajectories, which are then passed to the trajectory decoder. Using variational autoencoders ensures smoothness of the latent space, which, together with the low dimensionality of the latent representation, makes it quick to retrain the policy layers using a small amount of training data.

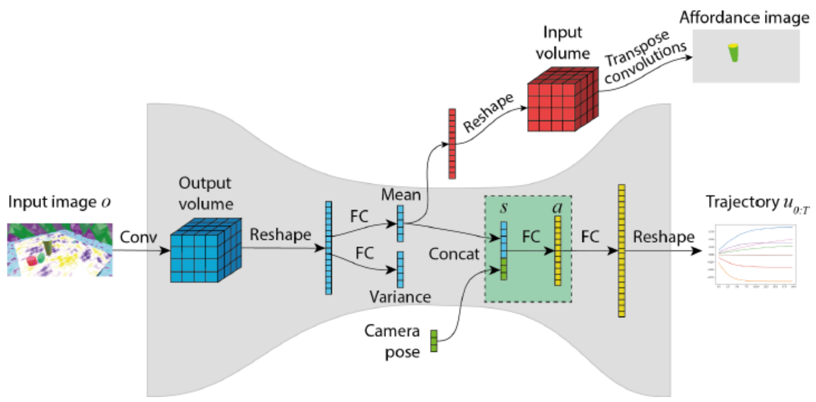


Figure 19: An input image is first processed by the affordance encoder (blue). The policy part (green rectangle) produces action a, *which is then decoded to a robot trajectory* $u_{0:T}$ *The affordance image is also generated by the affordance decoder (red) [29]*

The system was evaluated with and without clutter of various objects, testing criteria being detection accuracy (position error) and task success rate (task being to place a ball inside the objects)

Clutter objects	Rocket			Can			Red		
	d	%	n	d	%	n	d	%	n
0-6	1.25	97	45	1.64	92	28	2.02	94	69
7-12	1.86	96	30	2.93	76	26	2.4	82	64
13-18	2.84	100	6	4.47	73	30	4.76	63	36
19+	2.27	80	15	6.04	35	14	4.84	60	10

*Table 4: Results of experiments in the cluttered environment. (*d *is the average positon error in cm,* % *is the percentage of successful trials and* n *is the total number of trials) [29]*

Cup	Radius	Height	x error	y error	Error	Success
blue	3.45	16.60	0.73	2.48	2.68	82%
can	3.32	10.15	0.76	1.23	1.57	100%
can2	4.22	13.76	0.52	1.48	1.63	100%
green	3.92	9.33	0.74	1.00	1.35	100%
white	4.06	10.44	0.67	2.41	2.53	91%
jar	4.75	16.30	1.52	3.20	3.84	82%
red	4.30	8.08	0.59	1.60	1.76	100%
rocket	4.06	13.06	0.54	0.99	1.17	100%
stack1	3.70	10.52	0.52	1.99	2.07	100%
stack2	4.46	8.361	0.56	3.07	3.14	75%
stack3	2.72	8.76	0.51	1.15	1.31	100%
pastel	3.83	10.67	0.56	1.68	1.84	100%
yellow	4.06	5.69	0.95	3.45	3.69	73%
Average	3.91	10.90	0.71	1.97	2.19	92.5%

Table 5: Experiment results for different positions in a clutterless environment (distance in cm) [29]

This research shows that affordance encodings can be used to train policies successfully. If trained on randomized domain, the policy is invariant to distractor objects and textures, opening the possibility of zero-shot simulation to real system transfer. It also guides us towards further research into how we can incorporate multiple different deep learning algorithms and network them to work together.

To summarize, the literature review analyzed studies in machine learning as applied to machine vision and robot learning and provided a glimpse at how these techniques could be applied towards a series of real-world problems.

In case of machine vision, several different frameworks to achieve classification, object detection, semantic segmentation and instance segmentation were analyzed, each of which is found to enable devices to better perceive surroundings and perform tasks in an intelligent and safe manner. Also discussed were some of the studies which build upon earlier techniques and improve the ability of the machine to perform vision tasks.

For robot learning, this review described five studies where-in respective researchers taught a robot to: -

i. Move optimally using reward weighted regression,
ii. Grasp an object using only a set of 2D images,
iii. Perceive the orientation of an object in a complex visual environment using machine vision,
iv. Explore and execute tasks in gentle manner,
v. Perform perception, trajectory and policy building using a modular neural network architecture

2 MATERIALS

In this section, each of the tools that have been used as part of this project are briefly discussed.

2.1 OPEN COMPUTER VISION

Open Computer Vision (OpenCV) is an open source computer vision and machine learning library that offers thousands of algorithms and tools to implement functions such as basic programmatic image processing such as image color space modifications, image resizing, contour detections, morphological processing, etc. It also provides functions to perform object detection, facial recognition, object tracking, 3D point cloud processing, etc.

The OpenCV Python API (version 4.3.0) is used in this project. [30]

2.2 VOTT (VISUAL OBJECT TAGGING TOOL)

Developed by Microsoft, VoTT is an open source annotation and labeling tool for image and video assets that facilitates end-to-end machine learning pipeline [31]. It can be obtained from the linked GitHub repository.

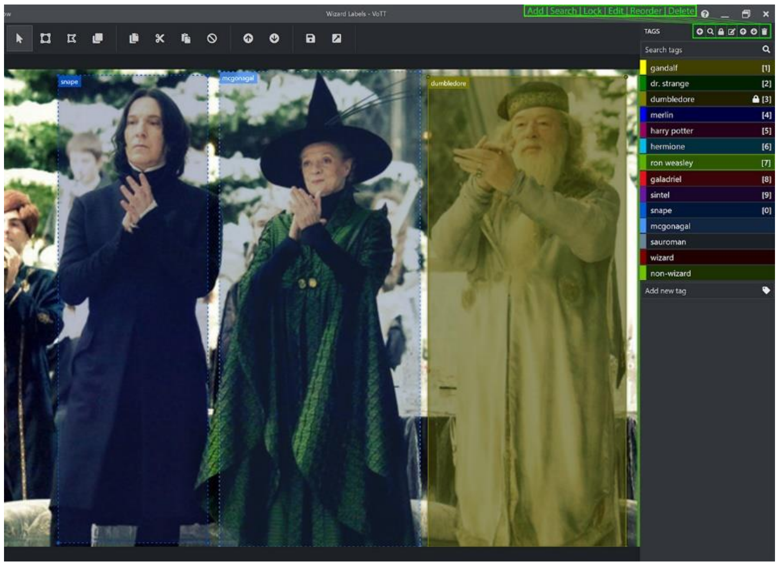


Figure 20: Example of VoTT being used to annotate a video file (Source: [31])

2.3 KERAS

An Application Programming Interface (API) is “a set of functions and procedures to create applications that access the features or data of an operating system, application, or other service.” Keras is a user-friendly, high level neural network API written in Python language [32]. It is possible to train a model developed in Keras on multiple backends like Tensorflow, Theano, CNTK, etc. It also comes packaged with multiple pretrained models and datasets like MNIST, CIFAR10 image dataset, IMDB Movie Reviews, etc. Using the Keras API, we can determine the ideal training method and parameters to obtain the final set of network weights that provides the most accurate object detection.

2.4 TENSORFLOW

TensorFlow (TF) [33] is a machine learning framework developed by Google, primarily for deep learning. TensorFlow accepts data in the form of multi-dimensional arrays called tensors. It provides a variety of tools to help developers design, build, visualize and train machine learning systems. The Keras API is fully integrated into TensorFlow since TF version 2.0.

In this project, the TF Python API version 2.1.0 is used.

2.5 GRAPHICS PROCESSING UNIT (GPU)

GPUs play a key role in training neural networks since they are tailor-made to perform massive numbers of floating-point operations in parallel for rendering graphics intensive applications such as modern video games. Using CUDA (NVIDIA’s API for programming on GPUs), and cuDNN (NVIDIA’s Library of primitives for accelerating Deep Learning related tasks), developers can harness the computational power of a GPU to train their neural networks much faster than on a CPU.

Massey University School of Food and Technology (SFAT) provide several desktop computers equipped with high-end NVIDIA GPU’s which are capable of training neural networks faster than a CPU. One of these systems or any system with similar or higher NVIDIA GPU specifications can be used to train a neural network.

For this project an NVIDIA GTX 1660Ti GPU is utilized along with CUDA version 10.1 and cuDNN version 7.6.

2.6 COPPELIA-SIM

Developed by Coppelia Robotics, CoppeliaSim [34] (formerly V-Rep) is a general-purpose robot simulation environment that comes equipped with an integrated development environment. It is based on a distributed control architecture; each object/model can be individually controlled via an embedded script, a plugin, a ROS or BlueZero node, a remote API client, or a custom solution. Controllers can be written in C, C++, Python, Java, Lua, MatLab or Octave. This makes CoppeliaSim highly versatile.

The remote API client and the programming environment in general are discussed in later sections.

2.7 MICROSOFT VISUAL STUDIO CODE

Microsoft Visual Studio Code [35] is a free, lightweight code editor that supports multiple programming languages and is designed for development operations like debugging, task running, and version control.

2.8 JUPYTER NOTEBOOK

Jupyter Notebook is an open source web application that provides an interactive coding environment, allowing users to implement live code, equations, visualizations, narrative text and much more. It is widely regarded as a robust environment for learning python programming and is also used for data visualization, machine learning, numerical simulation, etc.

3 METHODOLOGY

In this section, each component of the experimental setup is discussed in detail, starting with the coding environment in CoppeliaSim, and how it can be interfaced with Visual Studio Code. The python and Lua scripts used in the project are also explained. Second, we discuss the capabilities of the vision system and how the location of an apple is computed from a 2D image and depth data. Third, the instructions used to move the robot arm are briefly discussed. Finally, the procedure to run the complete setup is provided.

3.1 CODING ENVIRONMENT IN COPPELIASIM

CoppeliaSim is a highly customizable simulator thanks to its suite of six mutually compatible APIs that can be run together. The control entity of a model, scene, or the simulator itself can be implemented using one of the following methods:

1. Embedded Script: This is the most commonly used method which involves writing Lua scripts and allows the user to customize a simulation, a simulation scene, and to a certain extent the simulator itself.
2. Sandbox Script: This method involves writing Lua scripts, allowing us to quickly customize the simulator itself. Add-ons (or the sandbox script) can start automatically and run in the background, or they can be called as functions.
3. Plugins: Oftentimes, plugins are only used to provide a simulation with customized Lua commands, and so are used in conjunction with Embedded Scripts. Plugins are used to provide CoppeliaSim with a special functionality requiring either fast calculation capability, a specific interface to a hardware device, or a special communication interface with the outside world.
4. Remote API Client: This method allows the simulation to interface with an external application, in our case Visual Studio Code.
5. ROS Node: This method allows the simulation to interface with a Robot Operating System Node.

6. BlueZero Node: this method allows an external application to connect to CoppeliaSim via BlueZero.

The figure below illustrates the various customization possibilities in CoppeliaSim:

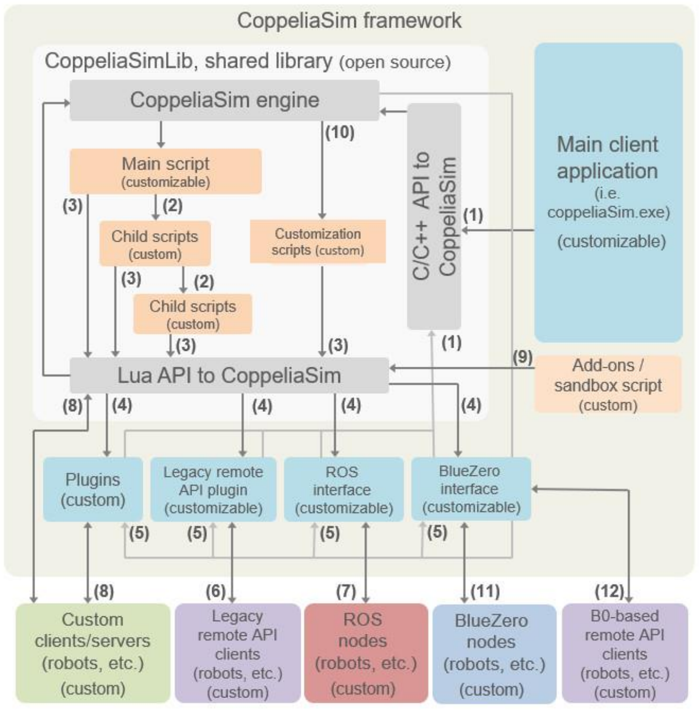

CoppeliaSim framework
CoppeliaSimLib, shared library (open source)
CoppeliaSim engine
Main script (customizable)
Child scripts (custom)
Child scripts (custom)
Customization scripts (custom)
C/C++ API to CoppeliaSim
Main client application (i.e. coppeliaSim.exe) (customizable)
Lua API to CoppeliaSim
Add-ons / sandbox script (custom)
Plugins (custom)
Legacy remote API plugin (customizable)
ROS interface (customizable)
BlueZero interface (customizable)
Custom clients/servers (robots, etc.) (custom)
Legacy remote API clients (robots, etc.) (custom)
ROS nodes (robots, etc.) (custom)
BlueZero nodes (robots, etc.) (custom)
B0-based remote API clients (robots, etc.) (custom)
(1) (2) (3) (4) (5) (6) (7) (8) (9) (10) (11) (12)

Figure 21: CoppeliaSim framework. Colored areas are custom or can be customized [36]

In this project, embedded scripts, sandbox scripts and the Python Remote API Client are used to customize and control the robot during simulation.

3.1.1 Remote API

The CoppeliaSim API consists of one hundred specific functions and one generic function that can be called from a Python script, C/C++ application, Lua script or Java application. The API interacts with CoppeliaSim via socket communication either in a synchronous or asynchronous manner.

A remote API function is called in a similar fashion as a regular API function, however with 2 major differences:

- Most remote API functions return a bit coded return code.
- Most remote API functions require two additional arguments: the operation mode, and the clientID (identifier returned by the simxStart function)

The remote API lets the user choose the operation mode and the way the simulation advances when a function is called. There are four main operation modes:

1. Blocking function calls: a blocking function call is meant for situations where we cannot afford to miss out on a reply from the server
2. Non-blocking function calls: a non-blocking function call is meant for situations when we simply want to send data to CoppeliaSim without the need for a reply
3. Data streaming: the server can anticipate what type of data the client requires. For that to happen, the client has to signal this desire to the server with a "streaming" or "continuous" operation mode flag (i.e. the function is stored on the server side, executed and sent on a regular time basis, without the need of a request from the client). This is a command/message subscription from the client to the server, where the server will be streaming the data to the client.
4. Synchronous operation: Remote API function calls will be executed asynchronously by default. There may be situations where the client needs to be synchronized with the simulation. This can be achieved by using the remote API synchronous mode. The remote API server service needs to be pre-enabled for synchronous operation.

There are two main threads running on the client (Sim application): The main thread (from which remote API functions will be called) and the communication thread/threads (Handling data transfers behind the scenes). The modus operandi of the remote API is shown below.

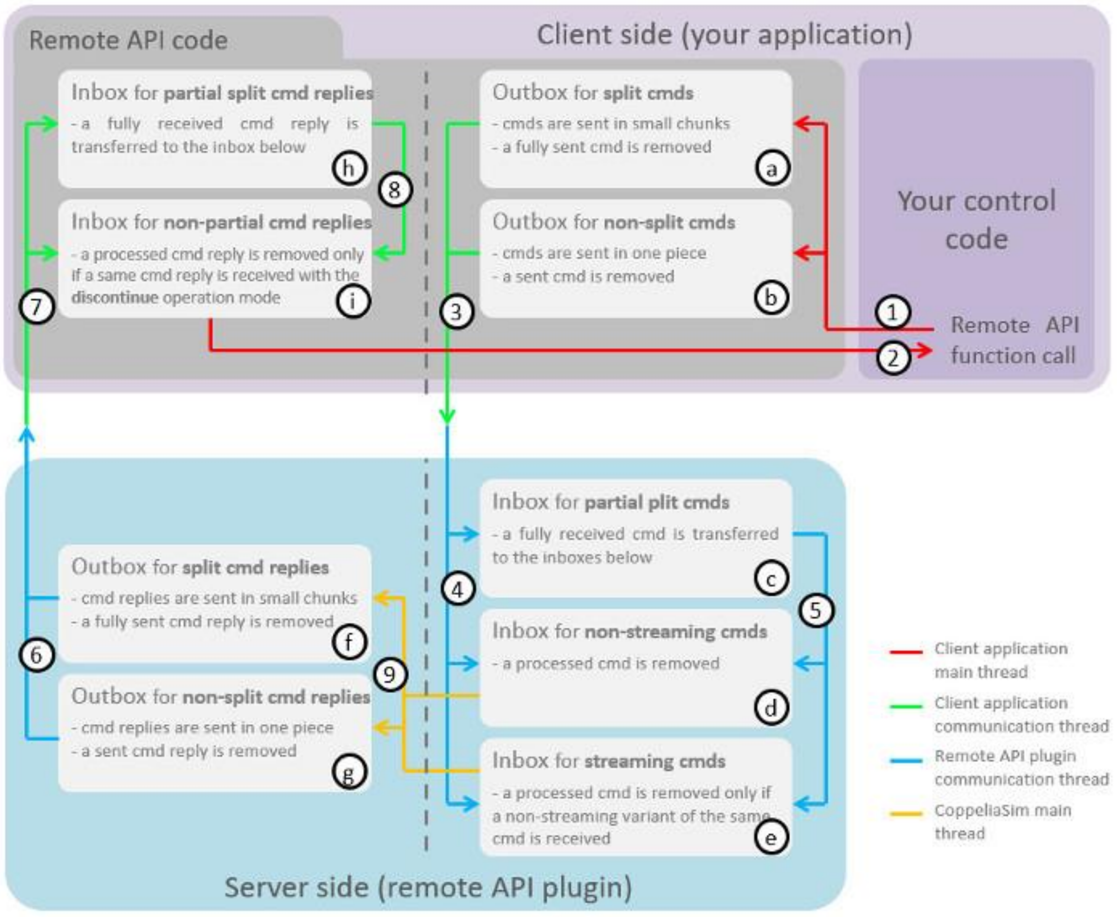

Remote API code
Client side (your application)
Inbox for partial split cmd replies
- a fully received cmd reply is transferred to the inbox below
Inbox for non-partial cmd replies
- a processed cmd reply is removed only if a same cmd reply is received with the discontinue operation mode
Outbox for split cmds
- cmds are sent in small chunks
- a fully sent cmd is removed
Outbox for non-split cmds
- cmds are sent in one piece
- a sent cmd is removed
Your control code
Remote API function call
Inbox for partial plit cmds
- a fully received cmd is transferred to the inboxes below
Inbox for non-streaming cmds
- a processed cmd is removed
Inbox for streaming cmds
- a processed cmd is removed only if a non-streaming variant of the same cmd is received
Outbox for split cmd replies
- cmd replies are sent in small chunks
- a fully sent cmd reply is removed
Outbox for non-split cmd replies
- cmd replies are sent in one piece
- a sent cmd reply is removed
Server side (remote API plugin)
Client application main thread
Client application communication thread
Remote API plugin communication thread
CoppeliaSim main thread

Figure 22: Remote API Functionality Overview [36]

Using these principles, it is possible to send commands to the robot, receive information from the environment or even modify simulation parameters while a simulation is running.

3.1.2 Project Code Explanation

A simple step-by-step explanation of the user-written/modified scripts in this project is provided as follows: -

3.1.2.1 External Python Script & Detection Notebook (Filenames: Kinect_image.py & Yolo_detection_program.ipynb)

Here we discuss the script written for obtaining images from the Kinect, computing the location of the apple, and instructing the robot to appropriately position itself so that it can grasp the apple. The jupyter notebook contains code which detects the apple and returns a bounding box and the pixel location of the bounding box center.

1. Import necessary libraries.

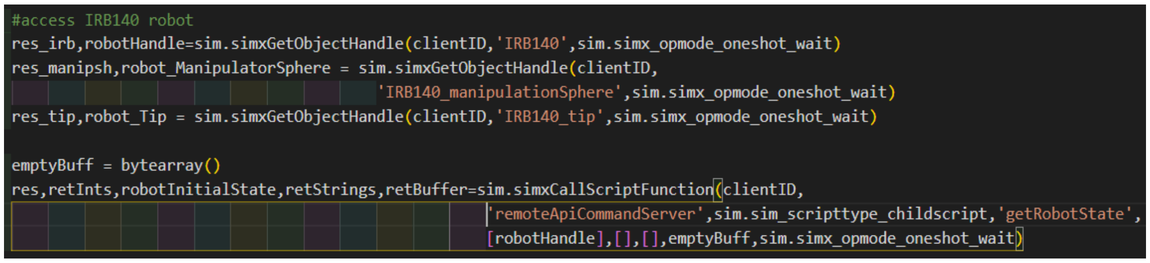
```python
try:
    import sim
except:
    print ('--------------------------------------------------------------')
    print ('"sim.py" could not be imported. This means very probably that')
    print ('either "sim.py" or the remoteApi library could not be found.')
    print ('Make sure both are in the same folder as this file,')
    print ('or appropriately adjust the file "sim.py"')
    print ('--------------------------------------------------------------')
    print ('')

import time
import os
import cv2
import sys
import random
import ctypes
import numpy as np
import array
import math
import skimage.io
from skimage import io
from PIL import Image
import tensorflow as tf
from IPython.display import Image, display
```

2. In the main function, connect to CoppeliaSim (The simulation must be running to achieve successful connection).

```
def main(_argv):

    print ('Program started')

    sim.simxFinish(-1) # just in case, close all opened connections
    clientID=sim.simxStart('127.0.0.1',19999,True,True,5000,5) # Con-
nect to CoppeliaSim

    if clientID!=-1: #if the client-sersver connection is established, exe-
cute the following code
        print ('Connected to remote API server')
```

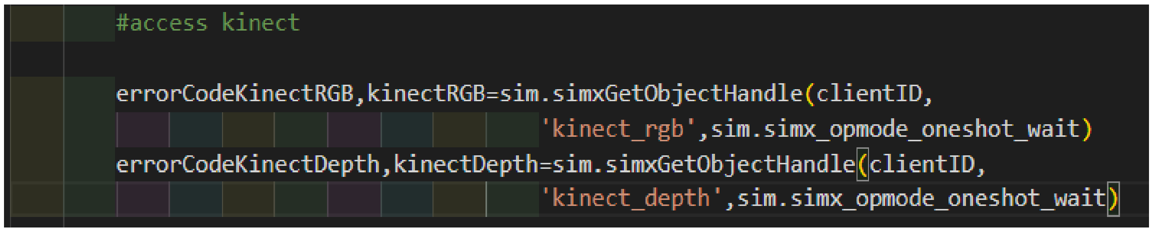
3. Obtain the object handles for the robot. An object handle is an integer value attributed to different objects in the simulation and is stored in a user made variable that can be used to access parameters of or control the corresponding objects in the simulation.

4. Obtain object handles for the relevant parts of the robot. We are concerned with using the IRB140_tip, IRB140_manipulationSphere, and the IRB140 group.



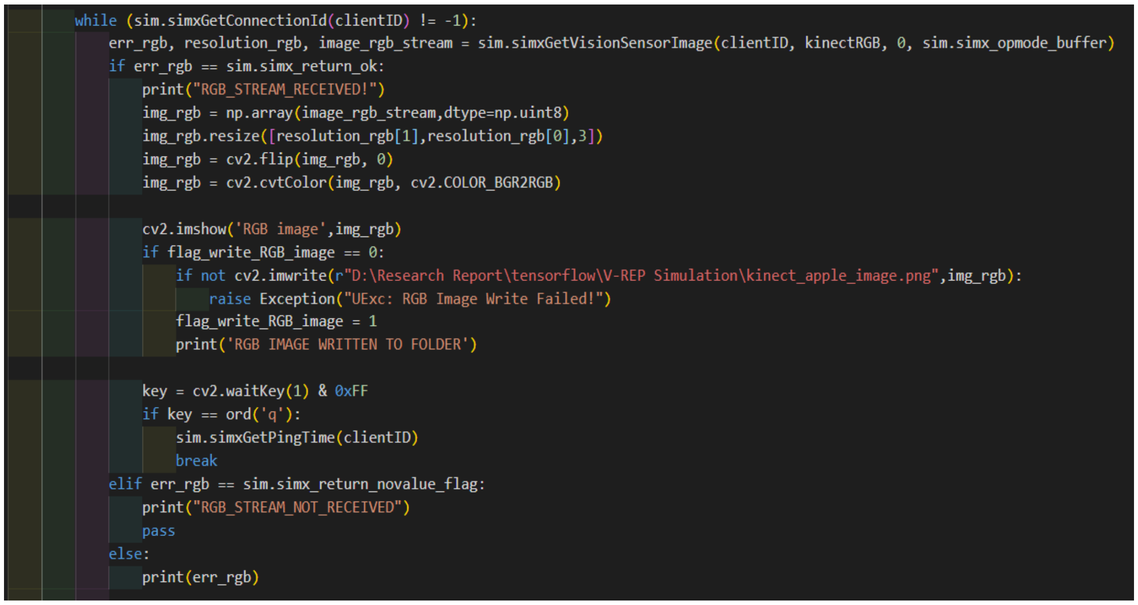
```
#access IRB140 robot
res_irb,robotHandle=sim.simxGetObjectHandle(clientID,'IRB140',sim.simx_opmode_oneshot_wait)
res_manipsh,robot_ManipulatorSphere = sim.simxGetObjectHandle(clientID,
                                        'IRB140_manipulationSphere',sim.simx_opmode_oneshot_wait)
res_tip,robot_Tip = sim.simxGetObjectHandle(clientID,'IRB140_tip',sim.simx_opmode_oneshot_wait)

emptyBuff = bytearray()
res,retInts,robotInitialState,retStrings,retBuffer=sim.simxCallScriptFunction(clientID,
                                                    'remoteApiCommandServer',sim.sim_scripttype_childscript,'getRobotState',
                                                    [robotHandle],[],[],emptyBuff,sim.simx_opmode_oneshot_wait)
```

5. Obtain object handles for the Kinect RGB and Depth cameras.



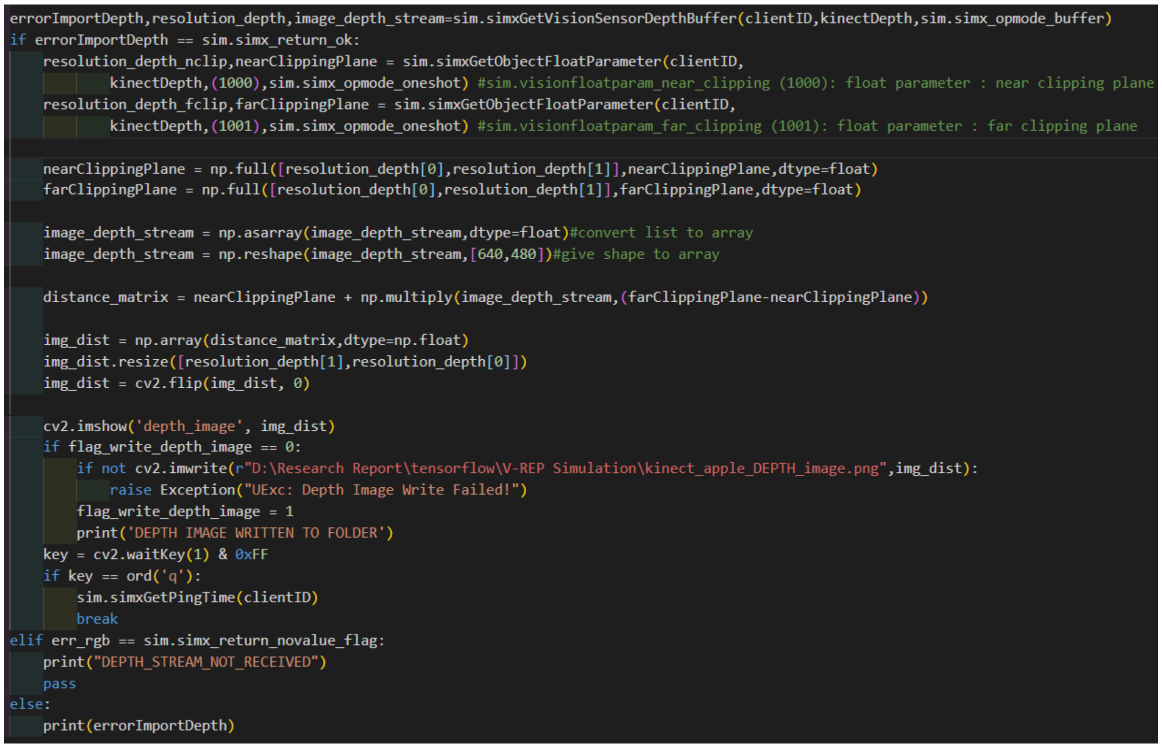
```
#access kinect

errorCodeKinectRGB,kinectRGB=sim.simxGetObjectHandle(clientID,
                                'kinect_rgb',sim.simx_opmode_oneshot_wait)
errorCodeKinectDepth,kinectDepth=sim.simxGetObjectHandle(clientID,
                                'kinect_depth',sim.simx_opmode_oneshot_wait)
```

6. Obtain the RGB image using the "simxGetVisionSensorImage" function in streaming mode and apply OpenCV processing functions like color space change, image flipping, etc. as necessary to get the required image.



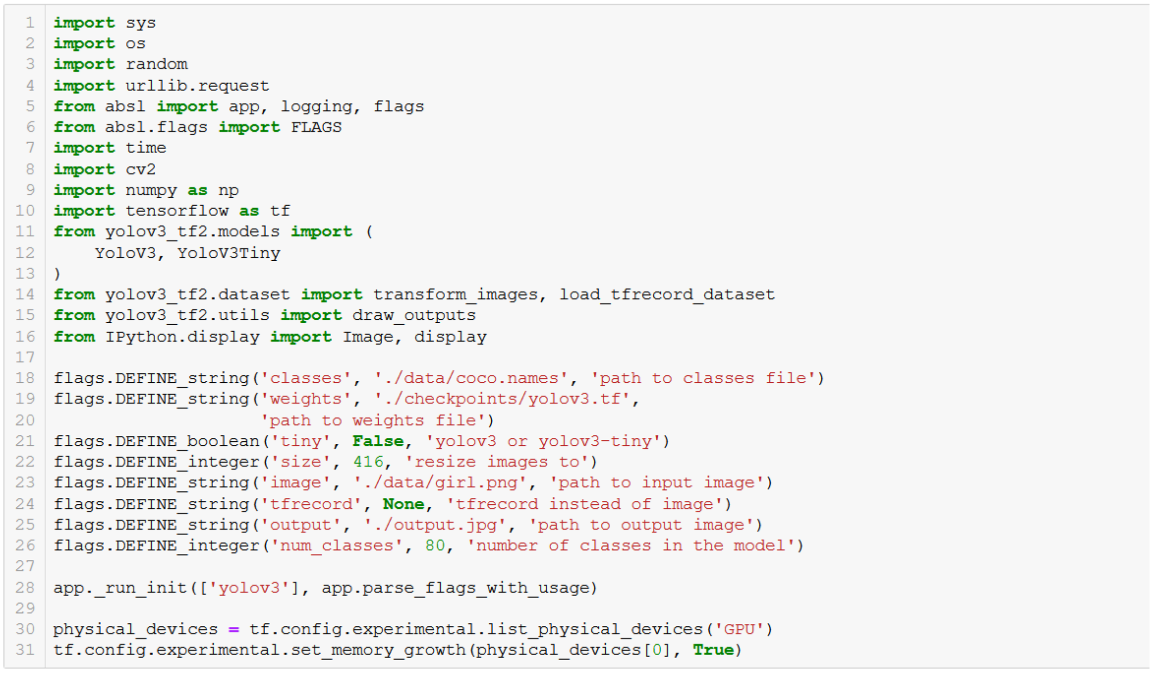
```python
    while (sim.simxGetConnectionId(clientID) != -1):
        err_rgb, resolution_rgb, image_rgb_stream = sim.simxGetVisionSensorImage(clientID, kinectRGB, 0, sim.simx_opmode_buffer)
        if err_rgb == sim.simx_return_ok:
            print("RGB_STREAM_RECEIVED!")
            img_rgb = np.array(image_rgb_stream,dtype=np.uint8)
            img_rgb.resize([resolution_rgb[1],resolution_rgb[0],3])
            img_rgb = cv2.flip(img_rgb, 0)
            img_rgb = cv2.cvtColor(img_rgb, cv2.COLOR_BGR2RGB)

            cv2.imshow('RGB image',img_rgb)
            if flag_write_RGB_image == 0:
                if not cv2.imwrite(r"D:\Research Report\tensorflow\V-REP Simulation\kinect_apple_image.png",img_rgb):
                    raise Exception("UExc: RGB Image Write Failed!")
                flag_write_RGB_image = 1
                print('RGB IMAGE WRITTEN TO FOLDER')

            key = cv2.waitKey(1) & 0xFF
            if key == ord('q'):
                sim.simxGetPingTime(clientID)
                break
        elif err_rgb == sim.simx_return_novalue_flag:
            print("RGB_STREAM_NOT_RECEIVED")
            pass
        else:
            print(err_rgb)
```

7. Obtain the data from the depth buffer using "simxGetVisionSensorDepthBuffer" in buffer mode. The buffer returns a list of depth values corresponding to each pixel.
8. Compute the depth image and distance matrix (process explained in section 3.3.2).



```python
errorImportDepth,resolution_depth,image_depth_stream=sim.simxGetVisionSensorDepthBuffer(clientID,kinectDepth,sim.simx_opmode_buffer)
if errorImportDepth == sim.simx_return_ok:
    resolution_depth_nclip,nearClippingPlane = sim.simxGetObjectFloatParameter(clientID,
            kinectDepth,(1000),sim.simx_opmode_oneshot) #sim.visionfloatparam_near_clipping (1000): float parameter : near clipping plane
    resolution_depth_fclip,farClippingPlane = sim.simxGetObjectFloatParameter(clientID,
            kinectDepth,(1001),sim.simx_opmode_oneshot) #sim.visionfloatparam_far_clipping (1001): float parameter : far clipping plane

    nearClippingPlane = np.full([resolution_depth[0],resolution_depth[1]],nearClippingPlane,dtype=float)
    farClippingPlane = np.full([resolution_depth[0],resolution_depth[1]],farClippingPlane,dtype=float)

    image_depth_stream = np.asarray(image_depth_stream,dtype=float)#convert list to array
    image_depth_stream = np.reshape(image_depth_stream,[640,480])#give shape to array

    distance_matrix = nearClippingPlane + np.multiply(image_depth_stream,(farClippingPlane-nearClippingPlane))

    img_dist = np.array(distance_matrix,dtype=np.float)
    img_dist.resize([resolution_depth[1],resolution_depth[0]])
    img_dist = cv2.flip(img_dist, 0)

    cv2.imshow('depth_image', img_dist)
    if flag_write_depth_image == 0:
        if not cv2.imwrite(r"D:\Research Report\tensorflow\V-REP Simulation\kinect_apple_DEPTH_image.png",img_dist):
            raise Exception("UExc: Depth Image Write Failed!")
        flag_write_depth_image = 1
        print('DEPTH IMAGE WRITTEN TO FOLDER')
    key = cv2.waitKey(1) & 0xFF
    if key == ord('q'):
        sim.simxGetPingTime(clientID)
        break
elif err_rgb == sim.simx_return_novalue_flag:
    print("DEPTH_STREAM_NOT_RECEIVED")
    pass
else:
    print(errorImportDepth)
```

9. The RGB image is fed to the YOLOv3 detection program (running on a separate Jupyter notebook session which is slightly modified to compute the center of the bounding box for each detected object). The center is returned as a pixel value, which is then fed back into VS Code script. This is done manually since there were several technical issues which prevented the functioning of a seamless program running both apple detection and real time robot control.



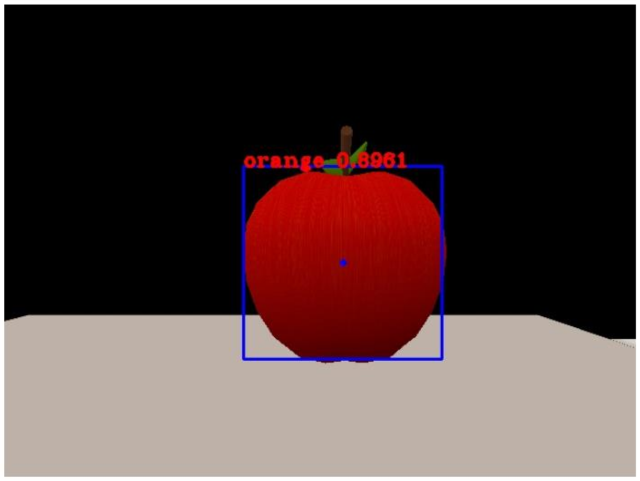
```
import sys
import os
import random
import urllib.request
from absl import app, logging, flags
from absl.flags import FLAGS
import time
import cv2
import numpy as np
import tensorflow as tf
from yolov3_tf2.models import (
    YoloV3, YoloV3Tiny
)
from yolov3_tf2.dataset import transform_images, load_tfrecord_dataset
from yolov3_tf2.utils import draw_outputs
from IPython.display import Image, display

flags.DEFINE_string('classes', './data/coco.names', 'path to classes file')
flags.DEFINE_string('weights', './checkpoints/yolov3.tf',
                    'path to weights file')
flags.DEFINE_boolean('tiny', False, 'yolov3 or yolov3-tiny')
flags.DEFINE_integer('size', 416, 'resize images to')
flags.DEFINE_string('image', './data/girl.png', 'path to input image')
flags.DEFINE_string('tfrecord', None, 'tfrecord instead of image')
flags.DEFINE_string('output', './output.jpg', 'path to output image')
flags.DEFINE_integer('num_classes', 80, 'number of classes in the model')

app._run_init(['yolov3'], app.parse_flags_with_usage)

physical_devices = tf.config.experimental.list_physical_devices('GPU')
tf.config.experimental.set_memory_growth(physical_devices[0], True)
```

```python
APPLE_IMAGE_PATH = r"D:\Research Report\tensorflow\yolov3-tf2-master\yolov3-tf2-master\images"

FLAGS.image = 'images/kinect_apple_image.png

if FLAGS.tiny:
    yolo = YoloV3Tiny(classes=FLAGS.num_classes)
else:
    yolo = YoloV3(classes=FLAGS.num_classes)

yolo.load_weights(FLAGS.weights).expect_partial()
logging.info('weights loaded')

class_names = [c.strip() for c in open(FLAGS.classes).readlines()]
logging.info('classes loaded')

img_raw = tf.image.decode_image(
    open(FLAGS.image, 'rb').read(), channels=3)

img = tf.expand_dims(img_raw, 0)
img = transform_images(img, FLAGS.size)

t1 = time.time()
boxes, scores, classes, nums = yolo(img)
t2 = time.time()
logging.info('time: {}'.format(t2 - t1))

logging.info('detections:')
for i in range(nums[0]):
    logging.info('\t{}, {}, {}'.format(class_names[int(classes[0][i])],
                                        np.array(scores[0][i]),
                                        np.array(boxes[0][i])))

img = cv2.cvtColor(img_raw.numpy(), cv2.COLOR_RGB2BGR)
```

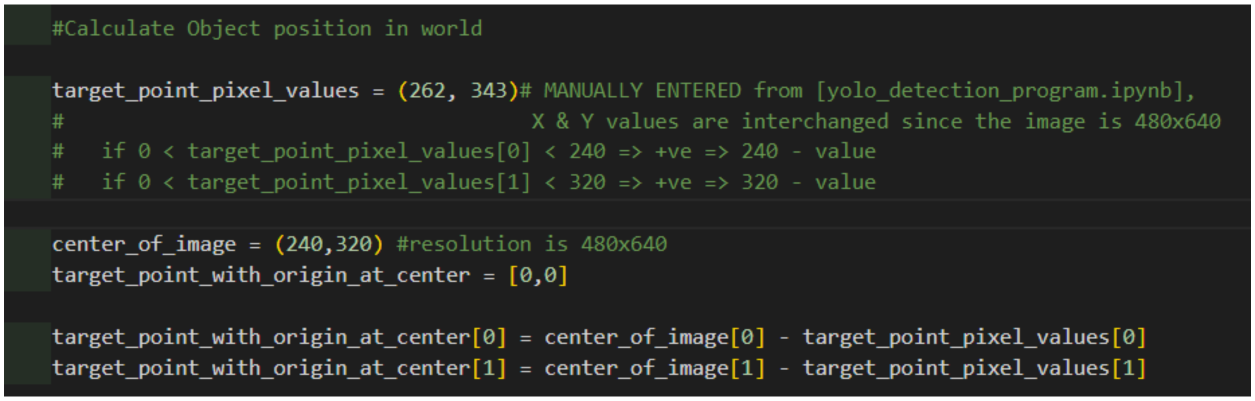
```
I0703 13:01:15.414903 26632 <ipython-input-2-863382dbf565>:16] weights loaded
I0703 13:01:15.423882 26632 <ipython-input-2-863382dbf565>:19] classes loaded
I0703 13:01:20.322913 26632 <ipython-input-2-863382dbf565>:30] time: 4.879072189331055
I0703 13:01:20.322913 26632 <ipython-input-2-863382dbf565>:32] detections:
I0703 13:01:20.328900 26632 <ipython-input-2-863382dbf565>:36]  orange, 0.8961134552955627, [0.3793993  0.34367487 0.
69336617 0.7509531 ]
```

```python
img = draw_outputs(img, (boxes, scores, classes, nums), class_names)
wh = np.flip(img.shape[0:2])

for i in range(nums[0]):
    rect_center_x = (boxes[0][i][0]+boxes[0][i][2])/2
    rect_center_y = (boxes[0][i][1]+boxes[0][i][3])/2
    #print(rect_center_x, rect_center_y)
    rect_center = np.array([rect_center_x,rect_center_y]*wh).astype(np.int32)
    img = cv2.circle(img, tuple(rect_center),2,(255, 0, 0), 2)
    print(rect_center)

display(Image(data=bytes(cv2.imencode('.jpg', img)[1]), width=800))
```

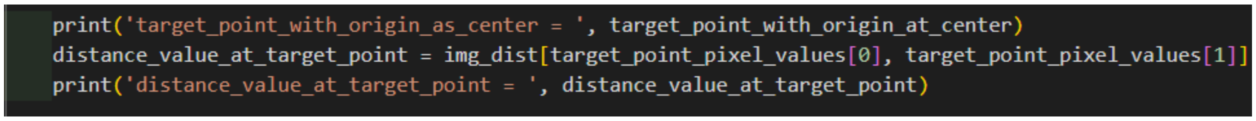
```
[343 262]
```



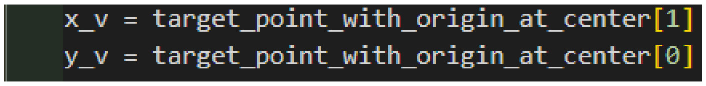
10. Once the pixel values are entered, the script computes the location of the target object in 3D space with respect to the camera frame (process explained in section 3.3.2).



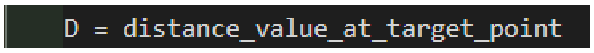
```
#Calculate Object position in world

target_point_pixel_values = (262, 343)# MANUALLY ENTERED from [yolo_detection_program.ipynb],
#                                      X & Y values are interchanged since the image is 480x640
#   if 0 < target_point_pixel_values[0] < 240 => +ve => 240 - value
#   if 0 < target_point_pixel_values[1] < 320 => +ve => 320 - value

center_of_image = (240,320) #resolution is 480x640
target_point_with_origin_at_center = [0,0]

target_point_with_origin_at_center[0] = center_of_image[0] - target_point_pixel_values[0]
target_point_with_origin_at_center[1] = center_of_image[1] - target_point_pixel_values[1]
```



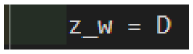
```
print('target_point_with_origin_as_center = ', target_point_with_origin_at_center)
distance_value_at_target_point = img_dist[target_point_pixel_values[0], target_point_pixel_values[1]]
print('distance_value_at_target_point = ', distance_value_at_target_point)
```



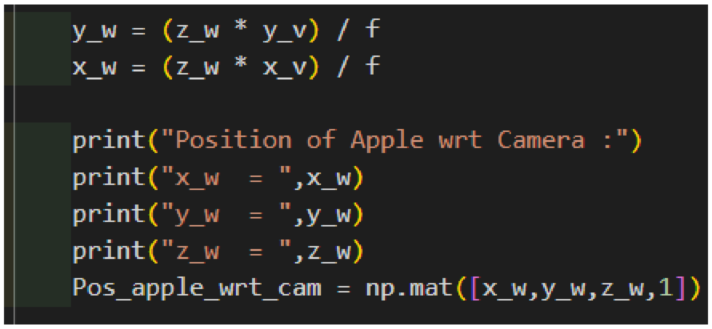
```
x_v = target_point_with_origin_at_center[1]
y_v = target_point_with_origin_at_center[0]
```



```
D = distance_value_at_target_point
```

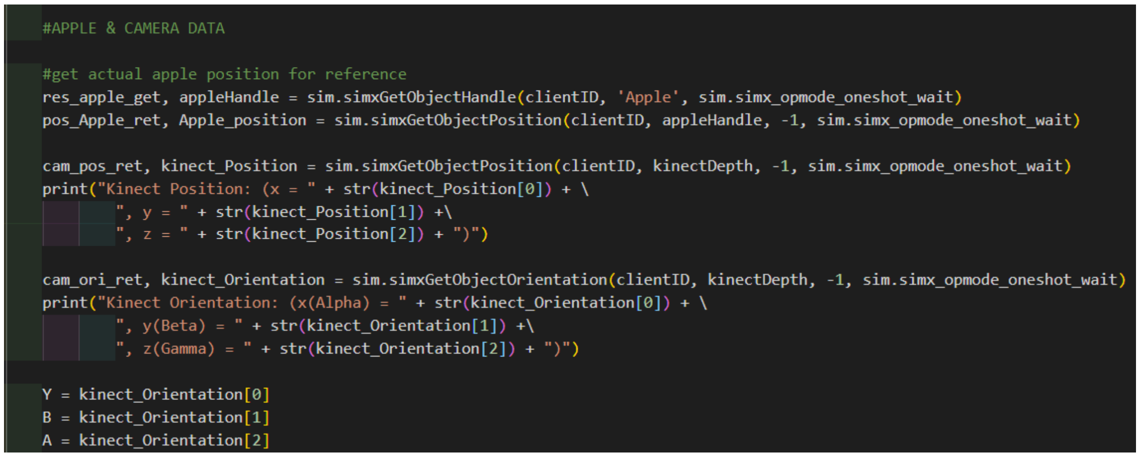


```
z_w = D
```

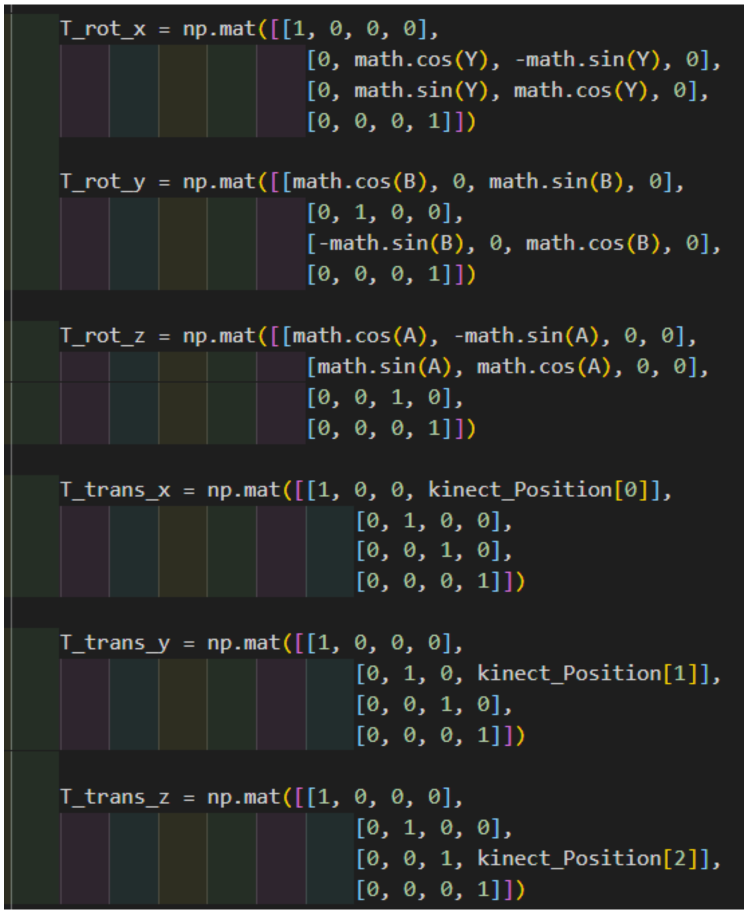


```
y_w = (z_w * y_v) / f
x_w = (z_w * x_v) / f

print("Position of Apple wrt Camera :")
print("x_w  = ",x_w)
print("y_w  = ",y_w)
print("z_w  = ",z_w)
Pos_apple_wrt_cam = np.mat([x_w,y_w,z_w,1])
```

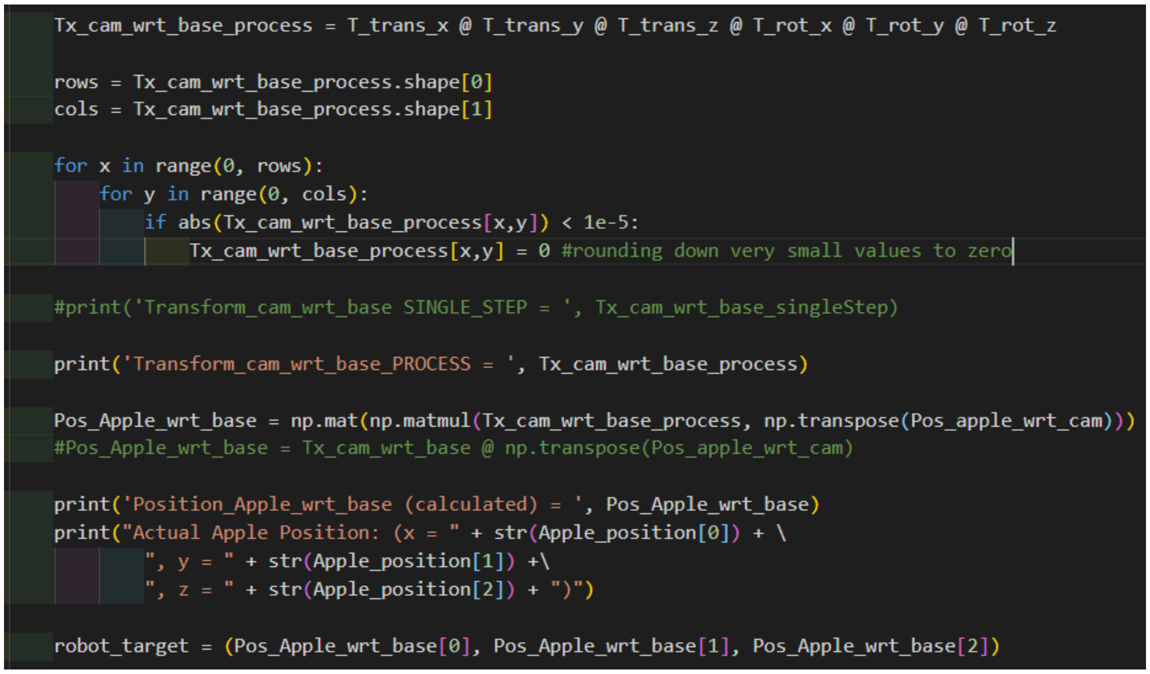
11. The target location is then converted from the camera frame of reference to the base frame using homogeneous transformations.



```
#APPLE & CAMERA DATA

#get actual apple position for reference
res_apple_get, appleHandle = sim.simxGetObjectHandle(clientID, 'Apple', sim.simx_opmode_oneshot_wait)
pos_Apple_ret, Apple_position = sim.simxGetObjectPosition(clientID, appleHandle, -1, sim.simx_opmode_oneshot_wait)

cam_pos_ret, kinect_Position = sim.simxGetObjectPosition(clientID, kinectDepth, -1, sim.simx_opmode_oneshot_wait)
print("Kinect Position: (x = " + str(kinect_Position[0]) + \
        ", y = " + str(kinect_Position[1]) +\
        ", z = " + str(kinect_Position[2]) + ")")

cam_ori_ret, kinect_Orientation = sim.simxGetObjectOrientation(clientID, kinectDepth, -1, sim.simx_opmode_oneshot_wait)
print("Kinect Orientation: (x(Alpha) = " + str(kinect_Orientation[0]) + \
        ", y(Beta) = " + str(kinect_Orientation[1]) +\
        ", z(Gamma) = " + str(kinect_Orientation[2]) + ")")

Y = kinect_Orientation[0]
B = kinect_Orientation[1]
A = kinect_Orientation[2]
```

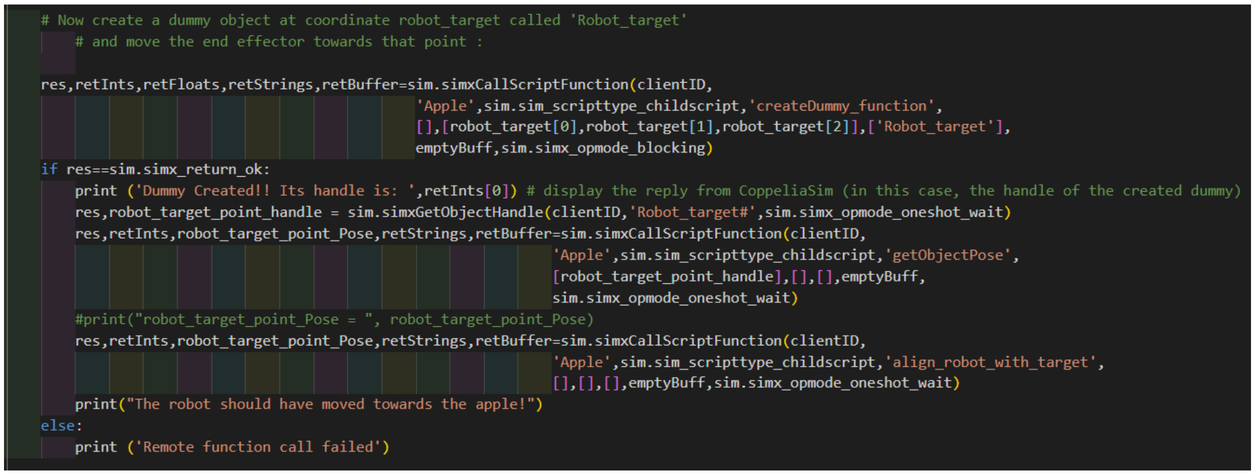


```
T_rot_x = np.mat([[1, 0, 0, 0],
                  [0, math.cos(Y), -math.sin(Y), 0],
                  [0, math.sin(Y), math.cos(Y), 0],
                  [0, 0, 0, 1]])

T_rot_y = np.mat([[math.cos(B), 0, math.sin(B), 0],
                  [0, 1, 0, 0],
                  [-math.sin(B), 0, math.cos(B), 0],
                  [0, 0, 0, 1]])

T_rot_z = np.mat([[math.cos(A), -math.sin(A), 0, 0],
                  [math.sin(A), math.cos(A), 0, 0],
                  [0, 0, 1, 0],
                  [0, 0, 0, 1]])

T_trans_x = np.mat([[1, 0, 0, kinect_Position[0]],
                    [0, 1, 0, 0],
                    [0, 0, 1, 0],
                    [0, 0, 0, 1]])

T_trans_y = np.mat([[1, 0, 0, 0],
                    [0, 1, 0, kinect_Position[1]],
                    [0, 0, 1, 0],
                    [0, 0, 0, 1]])

T_trans_z = np.mat([[1, 0, 0, 0],
                    [0, 1, 0, 0],
                    [0, 0, 1, kinect_Position[2]],
                    [0, 0, 0, 1]])
```



```python
Tx_cam_wrt_base_process = T_trans_x @ T_trans_y @ T_trans_z @ T_rot_x @ T_rot_y @ T_rot_z

rows = Tx_cam_wrt_base_process.shape[0]
cols = Tx_cam_wrt_base_process.shape[1]

for x in range(0, rows):
    for y in range(0, cols):
        if abs(Tx_cam_wrt_base_process[x,y]) < 1e-5:
            Tx_cam_wrt_base_process[x,y] = 0 #rounding down very small values to zero

#print('Transform_cam_wrt_base SINGLE_STEP = ', Tx_cam_wrt_base_singleStep)

print('Transform_cam_wrt_base_PROCESS = ', Tx_cam_wrt_base_process)

Pos_Apple_wrt_base = np.mat(np.matmul(Tx_cam_wrt_base_process, np.transpose(Pos_apple_wrt_cam)))
#Pos_Apple_wrt_base = Tx_cam_wrt_base @ np.transpose(Pos_apple_wrt_cam)

print('Position_Apple_wrt_base (calculated) = ', Pos_Apple_wrt_base)
print("Actual Apple Position: (x = " + str(Apple_position[0]) + \
        ", y = " + str(Apple_position[1]) +\
        ", z = " + str(Apple_position[2]) + ")")

robot_target = (Pos_Apple_wrt_base[0], Pos_Apple_wrt_base[1], Pos_Apple_wrt_base[2])
```

12. Finally, the target location (calculated in reference to the base frame) is fed to the inverse kinematics module in CoppeliaSim. This computes whether it is possible for the robotic arm to reach the target both in terms of position and orientation.



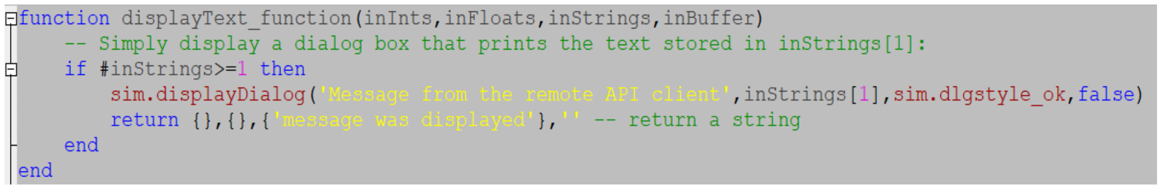
```python
# Now create a dummy object at coordinate robot_target called 'Robot_target'
    # and move the end effector towards that point :

res,retInts,retFloats,retStrings,retBuffer=sim.simxCallScriptFunction(clientID,
                                            'Apple',sim.sim_scripttype_childscript,'createDummy_function',
                                            [],[robot_target[0],robot_target[1],robot_target[2]],['Robot_target'],
                                            emptyBuff,sim.simx_opmode_blocking)
if res==sim.simx_return_ok:
    print ('Dummy Created!! Its handle is: ',retInts[0]) # display the reply from CoppeliaSim (in this case, the handle of the created dummy)
    res,robot_target_point_handle = sim.simxGetObjectHandle(clientID,'Robot_target#',sim.simx_opmode_oneshot_wait)
    res,retInts,robot_target_point_Pose,retStrings,retBuffer=sim.simxCallScriptFunction(clientID,
                                                    'Apple',sim.sim_scripttype_childscript,'getObjectPose',
                                                    [robot_target_point_handle],[],[],emptyBuff,
                                                    sim.simx_opmode_oneshot_wait)
    #print("robot_target_point_Pose = ", robot_target_point_Pose)
    res,retInts,robot_target_point_Pose,retStrings,retBuffer=sim.simxCallScriptFunction(clientID,
                                                    'Apple',sim.sim_scripttype_childscript,'align_robot_with_target',
                                                    [],[],[],emptyBuff,sim.simx_opmode_oneshot_wait)
    print("The robot should have moved towards the apple!")
else:
    print ('Remote function call failed')
```

3.1.2.2 Simulation Scripts

Each script in the simulation and the functions being performed by them is briefly explained as follows: -

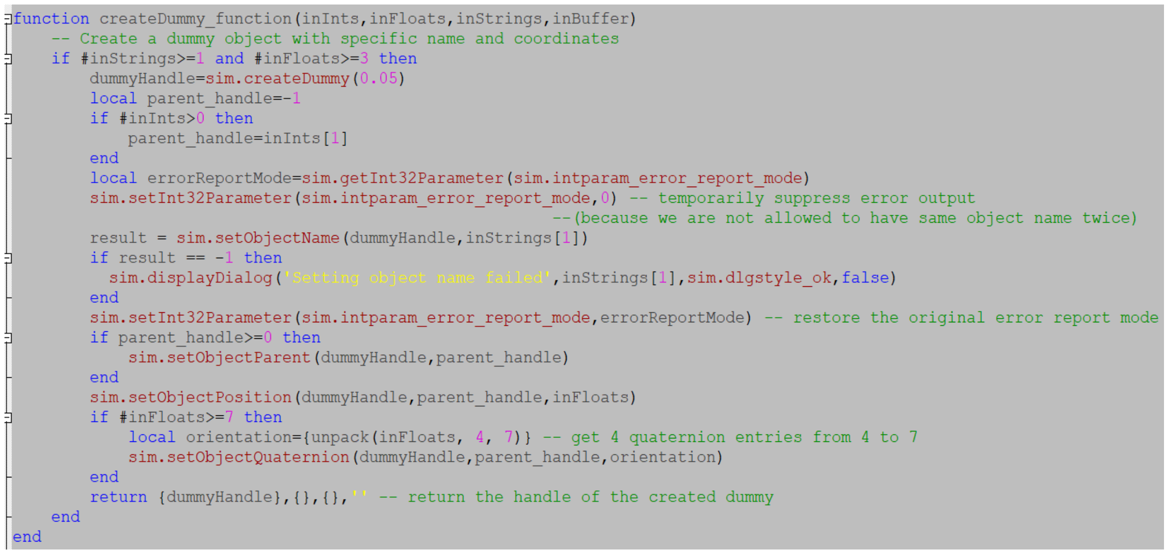
1. The Main Script: Each scene by default has one main script which contains the basic code that enables a simulation to run. It has four main parts: -

   a. The Initialization Function that is executed once when the simulation is started.
   b. The Actuation Function that is run in each simulation pass. It handles actuation functionality such as inverse kinematics, dynamics, etc., and calls the child scripts in the simulation.
   c. The Sensing Function is executed every simulation pass and is dedicated to handling all sensing related functionality such as camera, proximity sensing, collision detection, etc.
   d. The Restoration Function is run once at the end of simulation to ensure all elements of the scene are reset to their initial configurations.
2. Non-Threaded Script “Apple” (attached to the Apple object) has several functions which can be accessed as needed using the remote API via the VS Code script.
   a. Display Text Function can be used to display custom messages during simulation.



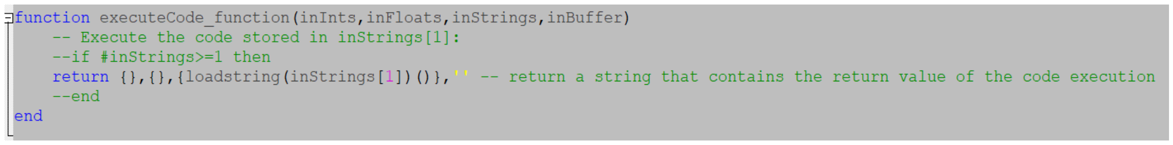
```lua
function displayText_function(inInts,inFloats,inStrings,inBuffer)
    -- Simply display a dialog box that prints the text stored in inStrings[1]:
    if #inStrings>=1 then
        sim.displayDialog('Message from the remote API client',inStrings[1],sim.dlgstyle_ok,false)
        return {},{},{'message was displayed'},'' -- return a string
    end
end
```

   b. Create Dummy Function is used to create a dummy object.



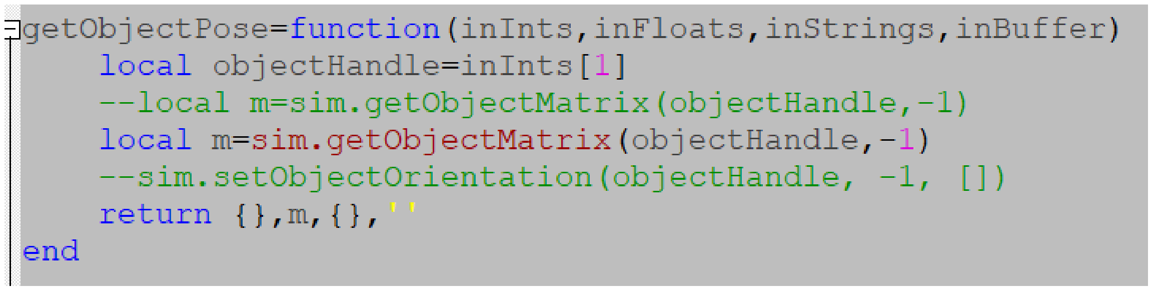
```lua
function createDummy_function(inInts,inFloats,inStrings,inBuffer)
    -- Create a dummy object with specific name and coordinates
    if #inStrings>=1 and #inFloats>=3 then
        dummyHandle=sim.createDummy(0.05)
        local parent_handle=-1
        if #inInts>0 then
            parent_handle=inInts[1]
        end
        local errorReportMode=sim.getInt32Parameter(sim.intparam_error_report_mode)
        sim.setInt32Parameter(sim.intparam_error_report_mode,0) -- temporarily suppress error output
                                                             --(because we are not allowed to have same object name twice)
        result = sim.setObjectName(dummyHandle,inStrings[1])
        if result == -1 then
          sim.displayDialog('Setting object name failed',inStrings[1],sim.dlgstyle_ok,false)
        end
        sim.setInt32Parameter(sim.intparam_error_report_mode,errorReportMode) -- restore the original error report mode
        if parent_handle>=0 then
            sim.setObjectParent(dummyHandle,parent_handle)
        end
        sim.setObjectPosition(dummyHandle,parent_handle,inFloats)
        if #inFloats>=7 then
            local orientation={unpack(inFloats, 4, 7)} -- get 4 quaternion entries from 4 to 7
            sim.setObjectQuaternion(dummyHandle,parent_handle,orientation)
        end
        return {dummyHandle},{},{},'' -- return the handle of the created dummy
    end
end
```

   c. Execute code Function can be used to run Lua code sent to the simulator from the VS Code script.



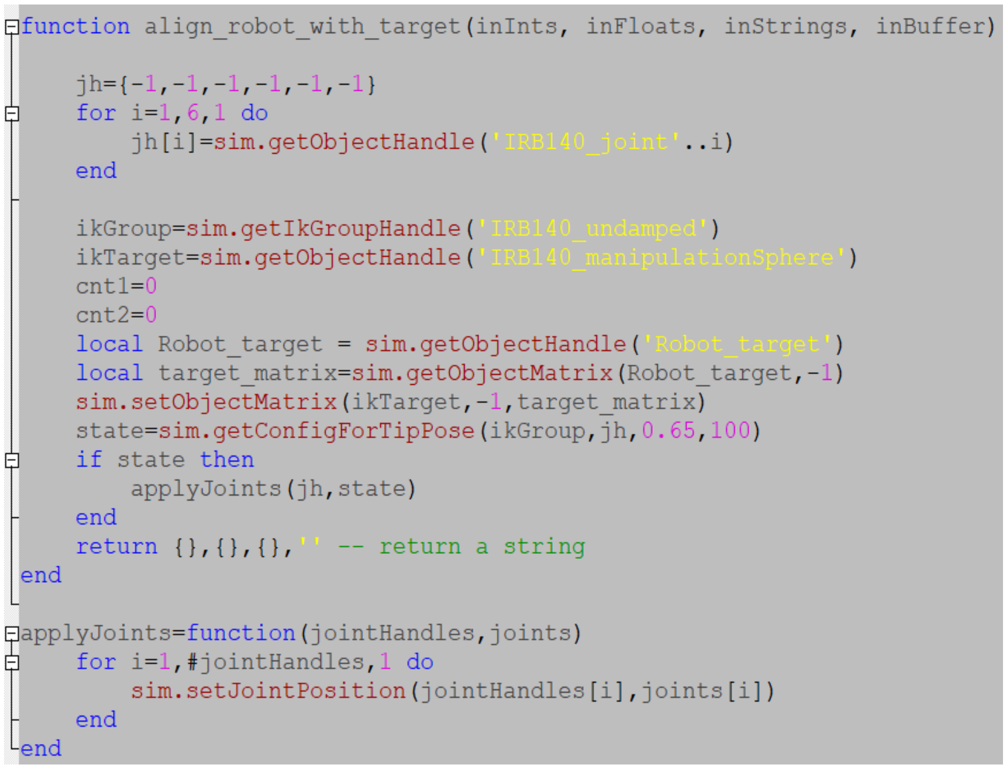
```
function executeCode_function(inInts,inFloats,inStrings,inBuffer)
    -- Execute the code stored in inStrings[1]:
    --if #inStrings>=1 then
    return {},{},{loadstring(inStrings[1])()},'' -- return a string that contains the return value of the code execution
    --end
end
```

d. Get Object Pose Function is used to obtain the transformation matrix of any object in the scene.



```
getObjectPose=function(inInts,inFloats,inStrings,inBuffer)
    local objectHandle=inInts[1]
    --local m=sim.getObjectMatrix(objectHandle,-1)
    local m=sim.getObjectMatrix(objectHandle,-1)
    --sim.setObjectOrientation(objectHandle, -1, [])
    return {},m,{},''
end
```

e. Align Robot with Target Function uses inverse kinematics and searches for a valid configuration for the robot arm, given the target orientation and position. It then moves the robot arm to the final position.



```
function align_robot_with_target(inInts, inFloats, inStrings, inBuffer)

    jh={-1,-1,-1,-1,-1,-1}
    for i=1,6,1 do
        jh[i]=sim.getObjectHandle('IRB140_joint'..i)
    end

    ikGroup=sim.getIkGroupHandle('IRB140_undamped')
    ikTarget=sim.getObjectHandle('IRB140_manipulationSphere')
    cnt1=0
    cnt2=0
    local Robot_target = sim.getObjectHandle('Robot_target')
    local target_matrix=sim.getObjectMatrix(Robot_target,-1)
    sim.setObjectMatrix(ikTarget,-1,target_matrix)
    state=sim.getConfigForTipPose(ikGroup,jh,0.65,100)
    if state then
        applyJoints(jh,state)
    end
    return {},{},{},'' -- return a string
end

applyJoints=function(jointHandles,joints)
    for i=1,#jointHandles,1 do
        sim.setJointPosition(jointHandles[i],joints[i])
    end
end
```

3. The Kinect Script initializes the Kinect's functionality and prepares a display window to show the output image.
4. IRB-140 Script:
    a. Initialization function initializes all the object handles for the robot's links & joints, sets the initial configuration and velocity limits.
    b. Actuation function takes care of forward and inverse kinematics movements of joints.

3.2 VISION SYSTEM

The objective of the vision system is to detect and identify an apple, and determine its position in 3D space. As discussed in the literature review, machine learning based object detection methods are proficient at identifying patterns in visual information, provided the neural network has been trained using the appropriate quality and quantity of training data.

In this work, attempts have been made at adapting three different machine learning algorithms for detecting apples: YOLO-v3, Mask R-CNN and U-Net. (the architectural details of each network are discussed in the literature review section). The implementation procedure and outcomes of YOLO-v3 are discussed here since it is the one being utilized to achieve the objectives of this project.

3.2.1 YOLO-v3

The code for YOLO-v3 is based on the implementation in [38].

The network's weights are pre-trained on the COCO dataset. COCO (Common Objects in Context) [39] is a large-scale object detection, segmentation, and captioning dataset that contains thousands of images, annotated for over 80 classes of objects (including apples).

Attempts were made to fine tune the network using the Fruits-360 dataset. The outcome of training using Fruits-360 is discussed in Section 5 since the quality of the dataset is not suited to the task at hand as the images are of low resolution and requires further work in regards to improved annotations.

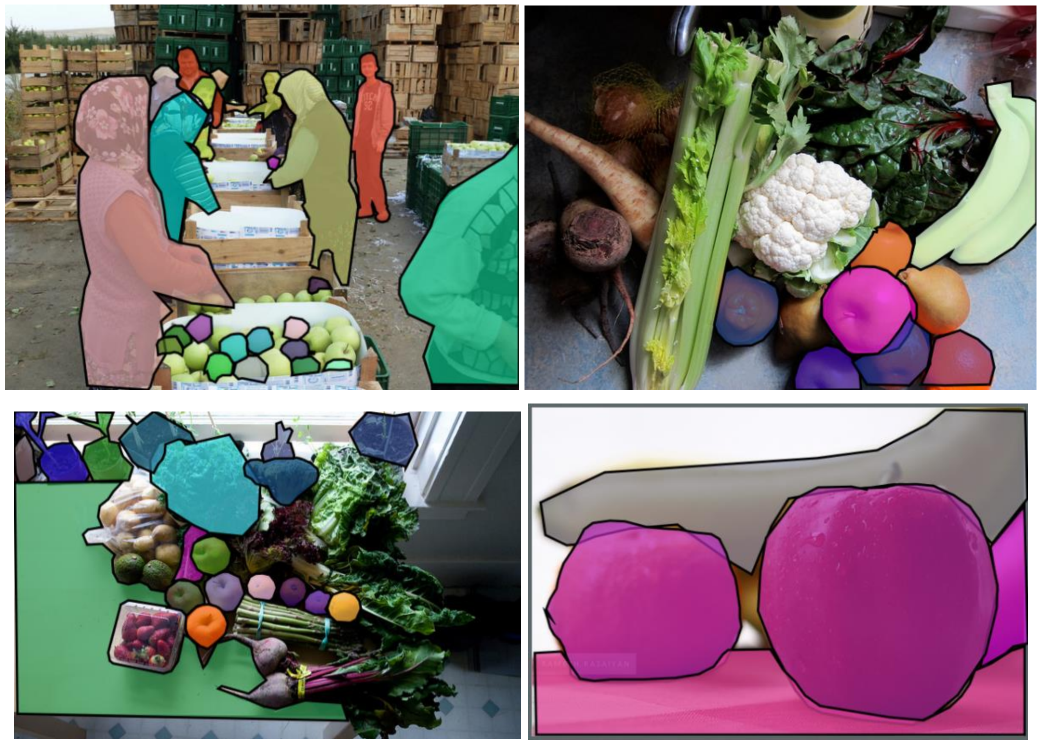


Figure 23: Examples of images with apples and annotations from the COCO dataset

The darknet-53 network can detect red and green apples under standard lighting conditions but is often prone to mis-labeling apples as oranges or labelled as other objects when viewed from different angles (Refer Section 4.1).

3.2.2 APPLE LOCATION COMPUTATION

Once the apple is detected, the network creates a bounding box for each instance (the network does not detect apple instances when a set of similar objects are very close together, such as a pile of apples. It tends to group them together and labels the class as ‘unclassified’). The parameters returned for each detected object box are - the class name, probability score and the coordinates of the upper left and lower right corners of the bounding box.

```
I0703 13:01:15.414903 26632 <ipython-input-2-863382dbf565>:16] weights loaded
I0703 13:01:15.423882 26632 <ipython-input-2-863382dbf565>:19] classes loaded
I0703 13:01:20.322913 26632 <ipython-input-2-863382dbf565>:30] time: 4.879072189331055
I0703 13:01:20.322913 26632 <ipython-input-2-863382dbf565>:32] detections:
I0703 13:01:20.328900 26632 <ipython-input-2-863382dbf565>:36] orange, 0.8961134552955627,
[0.3793993 0.34367487 0.69336617 0.7509531]
```

Table 6: Output from the YOLO-v3 detection program

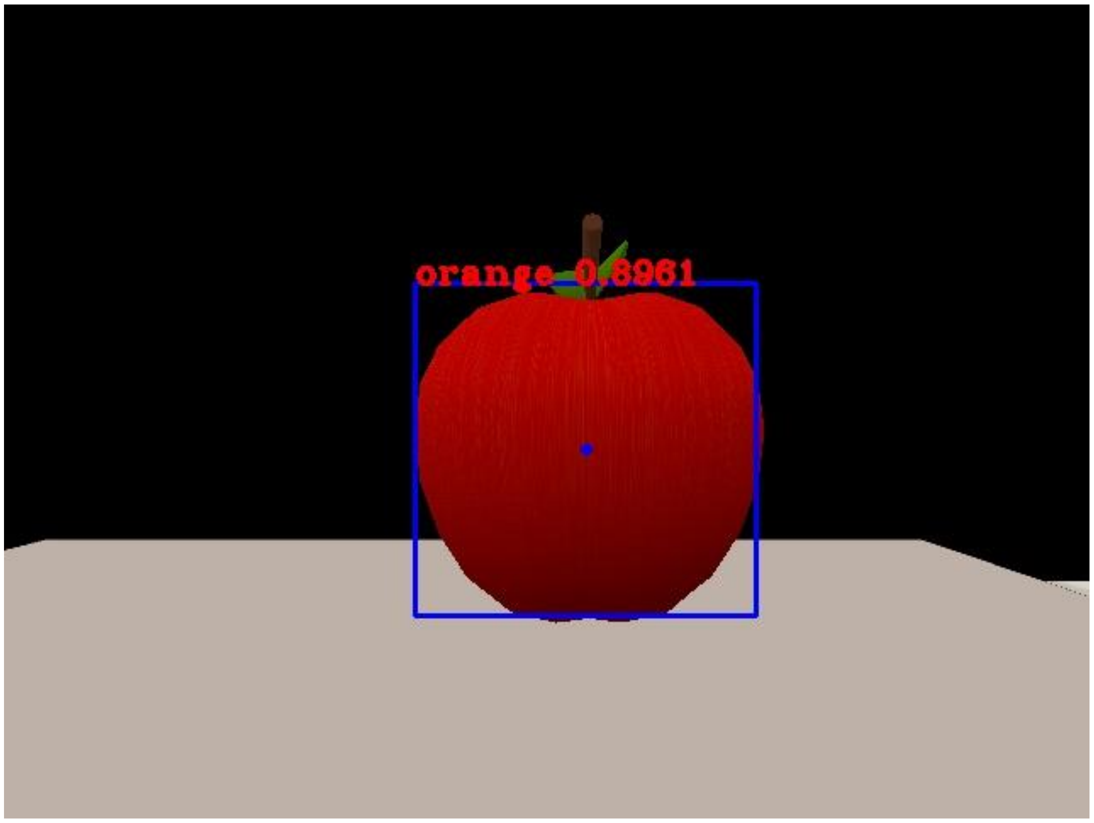


Figure 24: The output image from the YOLO-v3 detection program

Using the two opposing corner coordinates (top left and bottom right corners) of the bounding box, we can calculate the center of the box, which approximately corresponds to the center of the apple. This is the target point for the end effector on our robot arm to approach and orient itself for grasping action.

Once we have the pixel location of the target point on the image, we use depth information to compute a distance matrix and then apply simple geometry to determine target position in 3D space.

The distance matrix is computed as follows (Refer code in Section 3.1.2.1):

1. First resize and reshape the list of depth values obtained from the vision sensor's depth buffer into an array with dimensions equal to the resolution of the image
2. Retrieve the nearClippingPlane and farClippingPlane values. These values give us the range of distances within which the camera can detect objects. Any object outside these planes is not detected by the camera. For the Kinect model being used, nearClippingPlane = 0.1m & farClippingPlane = 3.5m
3. The values in the depth array are usually in the range 0 to 1, where 0 corresponds to the object being on the near clipping plane, and 1 meaning the object is at the far end of the range, or that no object was detected.
4. Multiply the fractional depth value by the difference between the near clipping plane and far clipping plant to obtain the distance of the object from the camera.

$$Distance\,matrix = nearClippingPlane + (image\_depth\_stream \times (farClippingPlane - nearClippingPlane))$$

The process [40] for determining the location of the apple in 3D space using a 2D image of it, is as follows: -

1. Obtain the pixel location of the target point V on the RGB image (target point as seen through the viewport). By default, the YOLO network returns two values, each between 0 & 1 corresponding to the X and Y locations of the object on the image. This is converted to pixel values by multiplying the returned fractional value with the width and height of the image. This gives us (X_v, Y_v)
2. Using the depth array, find the corresponding depth value D at the target point (X_v, Y_v). In this case, since resolution of the depth image and the RGB image provided by the simulated Kinect are same, it is a straightforward operation. This may not the case for a real Kinect or other RGB-D cameras, so it would be advised to refer the respective documentation beforehand.

3. Assume the center of the image is origin of the camera frame. For a 640 x 480 image, the center of the image is (320, 240)
4. Calculate the position of the target point on the image using this new origin, keeping in mind the X, Y, Z axes of the camera frame. In our case, X & Y axes are used to describe the horizontal and vertical displacement with respect to the camera frame's origin, while the Z axis is used to describe how far away the object is from the camera.
5. The focal length constant of the Kinect is considered to be f = 579.83, (calculated based on the field of view of the Kinect). In regard to a real Kinect, two values of 'f ' are calculated - one along the X axis and one along the Y axis. For the simulation, we assume $f = f_x = f_y = 579.83$
6. Calculate the Z_{world} coordinate of the target point using the relation

$$Z_{world} = D \times f / \sqrt{Xv^{\,2} + Yv^2 + f^2}$$

(Note that in case of the simulated Kinect, it is safe to assume $Z_{world} = D$ since the depth value returned is not the Euclidean distance between the camera and the respective point, but rather orthogonal distance to a plane that contains the point of interest)

7. Using the principles of similar triangles, calculate the X_{world} and Y_{world} values

$$Y_{world} = Z\_world \times Yv / f$$

$$X_{world} = Z\_world \times Xv / f$$

Now, we have coordinates of the target point in 3D space with reference to the camera frame. This location can be easily converted to any frame of reference using a mathematical tool such as homogenous transforms, provided we know the location and orientation of the Kinect with respect to a universal frame of reference (called World Frame or Base frame). The location and orientation of the Kinect can be obtained using Remote API commands: simxGetObjectPosition and simxGetObjectOrientation, which returns a list of X-Y-Z coordinates and α-β-γ values respectively.

Homogenous transformation from the base frame to the camera frame is carried out to ascertain the position of the target point in reference to the base frame, which is same as the frame of reference used by the robot. This way, we simply need to feed the desired position and orientation of the robot to the respective function (discussed further in next section) which then performs the necessary operations to reach the target position & orientation. The frame conversion consists of 6 transformations – 3 translations (along X, then along Y and then along Z axes), followed by 3 rotations (about X, then Y and then Z axes). On performing these operations, we obtain the coordinates of the target point in reference to the base frame.

3.3 ROBOT ARM

The IRB-140 comes built-into CoppeliaSim along with scripts that can be modified to control how it behaves.

In this work, the focus is on ensuring the robot can move towards the target point. In this regard, it is shown that the there exists a valid set of joint positions for the end effector to reach the target point.

The end effector is instructed to maintain a certain orientation at the target point by creating a non-interactive dummy object that holds the required orientation at the respective location. This orientation and the computed position are fed to the inverse kinematics function.

The IRB140 is operated in "Inverse Kinematics using Manipulator sphere" mode, which can be activated by changing the stored value in the relevant variable found in the IRB140 script. The green manipulator sphere is considered to be the end effector. Inverse kinematics are computed in CoppeliaSim using the "getConfigForTipPose" command which performs a randomized search for a joint configuration that satisfies the constraints for a given end-effector position & orientation. If the target is attainable, then we can see that the robot instantly moves to show how it's joints would be oriented once the arm has completed the movement action. If the target is outside the workspace of the robot, or there exists no joint configuration to reach the target, an error message "IK Solver Failed" is displayed.

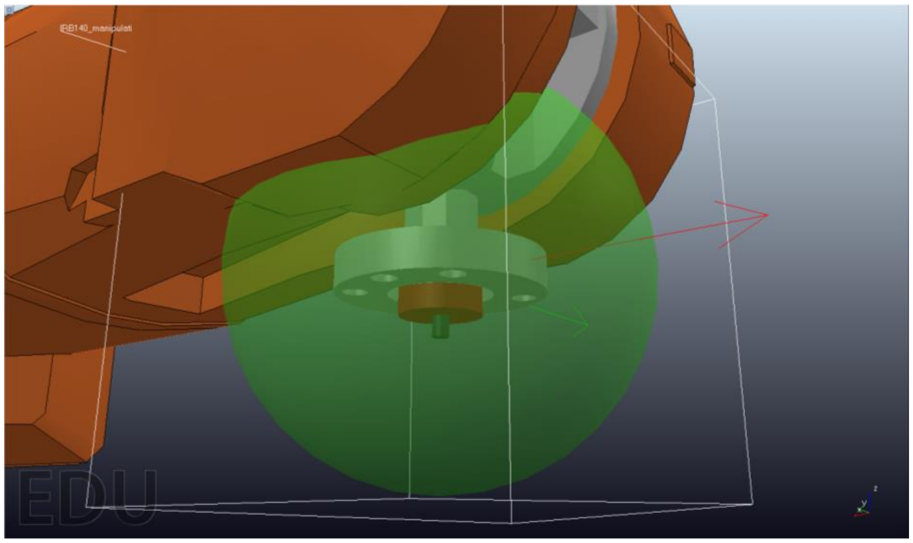


Figure 25: The manipulator sphere of the IRB 140

3.4 SIMULATION SETUP

A simple simulation environment (scene name: IRB140_Apple.ttt) has been set up to demonstrate the working of vision system and how a robot would use results generated by the vision system to approach the target object.

The components used in the environment are discussed in detail in the following sections.

3.4.1 KINECT RGB-D CAMERA

The image being used to detect the apple is obtained from the Kinect RGB-D (Red, Green, Blue - Depth) camera. A real Kinect camera provides two streams of data; one is a standard RGB color image and the other is a depth array; the value of each element of the array corresponding to the distance of the detected objects from the Kinect. The RGB image is obtained from its VGA color camera, and the depth information is generated by a configuration of infrared projector and a monochrome CMOS sensor. Combining these two streams, the camera can create a 3D view of its perceived environment. It also has an accelerometer and microphone, but those features are not relevant to our work.

In case of the simulated Kinect model, the device only provides RGB and depth data.

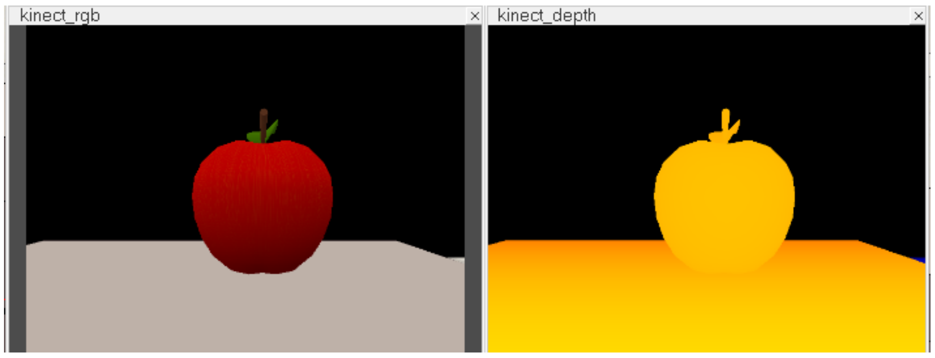
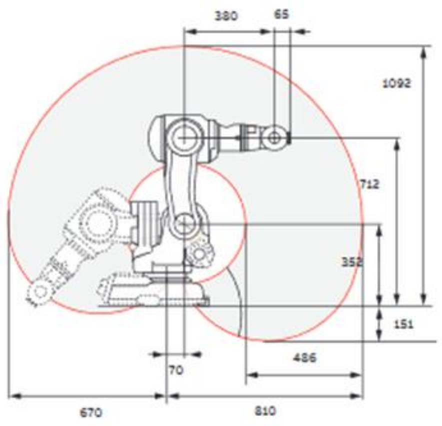


Figure 26: Visual data returned by the simulated Kinect RGB-D camera, as shown within CoppeliaSim

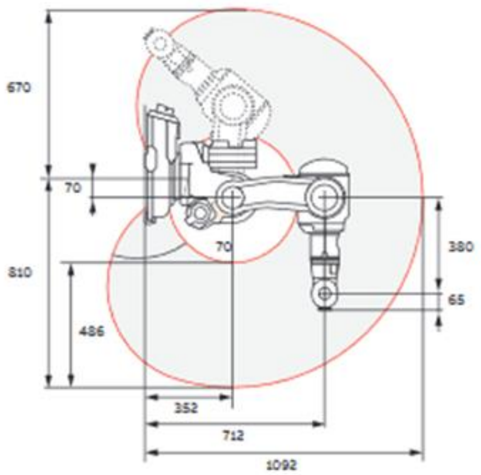
3.4.2 IRB-140 ROBOT

The ABB IRB-140 [37] is a 6-axis industrial robot with a payload of 6 kg, designed specifically for manufacturing industries that use flexible robot-based automation. It has an open structure that is specially adapted for flexible use and can communicate extensively with external systems.

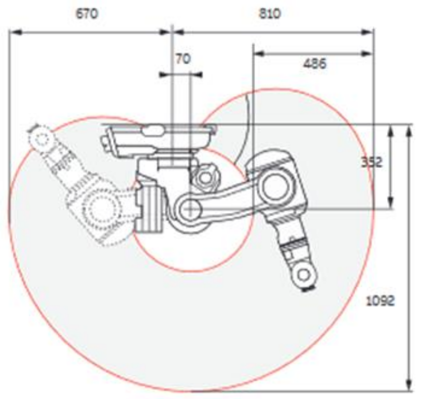
Given below is the IRB-140's range of movement. All target objects in the simulation environment are placed within operating range of the robot arm.



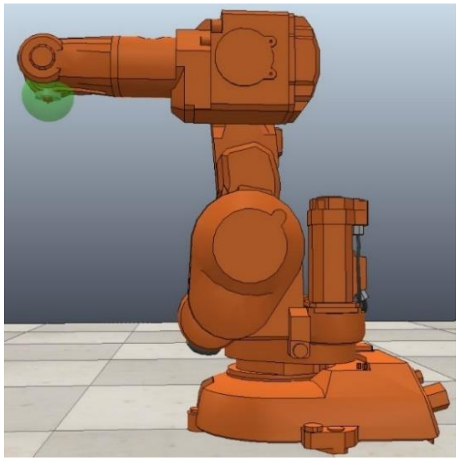
(a)



(b)



(c)



(d)

Figure 27: (a), (b) & (c) show the working range of the IRB 140 from different angles. (d) shows a 3D model of IRB 140 in CoppeliaSim

3.4.3 3D MODEL OF AN APPLE

A simple 3D model of an apple is made using Blender, a free & open source creation suite and saved in the .obj format. It is then imported into CoppeliaSim.

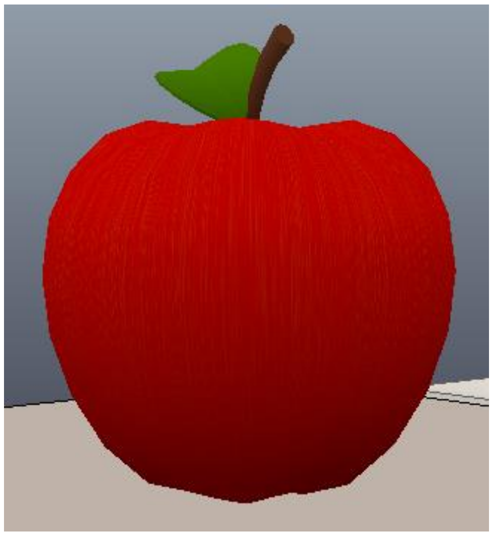


Figure 28: 3D model of an Apple

3.4.4 TABLE

The Kinect and the Apple are placed on a simple 3D model of a table.

The complete simulation setup is as shown below.

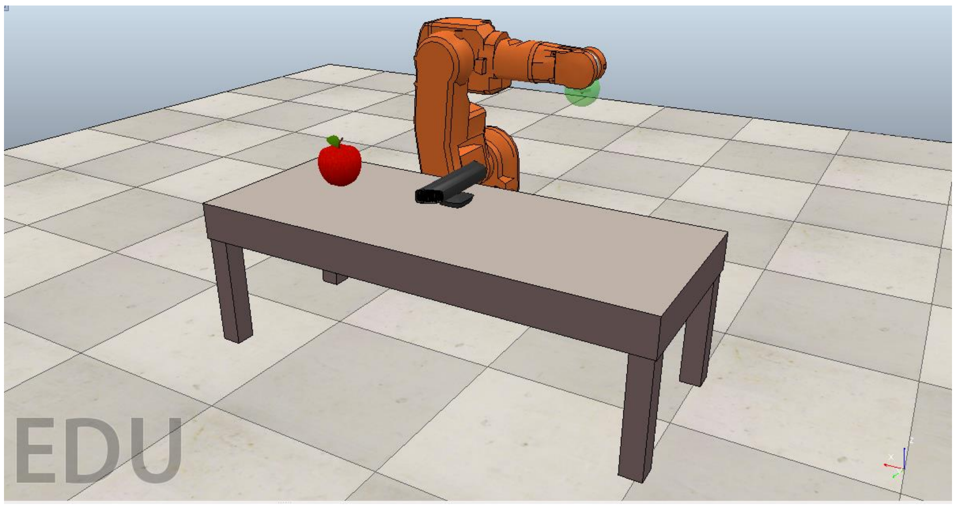


Figure 29: The complete simulation setup

3.5 EXPERIMENTAL PROCEDURE

Once all components are ready, we can begin running the simulation and the python code script in the following manner:

1. Open CoppeliaSim and load the IRB140_Apple.ttt scene
2. Begin the simulation by clicking on the Start/Resume Simulation button in the toolbar
3. Open Visual Studio Code and load the Kinect_image.py script
4. Run the Kinect_image.py script. If the simulation is running, the python script should be able to set up a connection between the script and the simulator.
5. If a connection is established, two windows should pop up with the RGB and Depth images being displayed. Pressing the “q” or “Q” key should close the windows and write the RGB image to a folder. Once the RGB image is written to the respective folder, open the Yolo_detection_program jupyter notebook
6. Ensure the path and name of the image are correct and run the YOLO detection program. If the apple is detected, the program should return an image with a bounding box around the detected apple and the pixel location of the center point of each bounding box.
7. Edit the Kinect_image.py program at the designated line with the returned pixel location of the apple (changing the value stored in the target_point_pixel_values variable).
8. Stop and restart the simulation
9. Run the python script again.
10. Now, the robot arm should have moved towards the target point, provided the target point is within operating range of the arm.

4 TESTING, RESULTS & DISCUSSION

In this section, we test the capabilities of the YOLOv3 detection system as applied to images of real apples and to images of the 3D apple from the simulation environment. Next, the accuracy of the target point location calculations is tested, followed by the outcome of running the complete project setup.

4.1 YOLO-v3 DETECTION PROGRAM

4.1.1 Detection on Images of Real Apples

The following series of images demonstrate the capabilities of the YOLOv3 Darknet at detecting and identifying real apples. These images are fed to the Yolo_detection_program, and the returned results are shown.

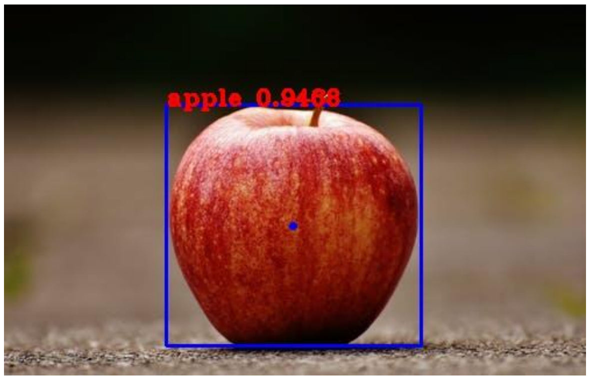


Figure 30: Red Apple (Source: [41])

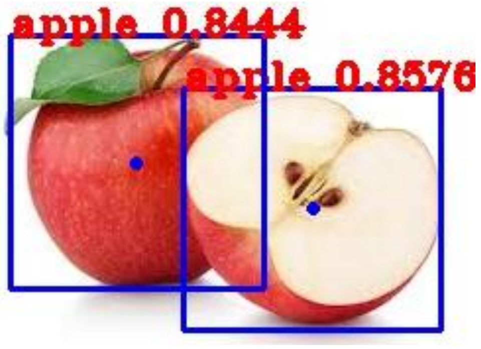


Figure 31: Red Apple and a sliced Red Apple (Source: Shutter Stock)

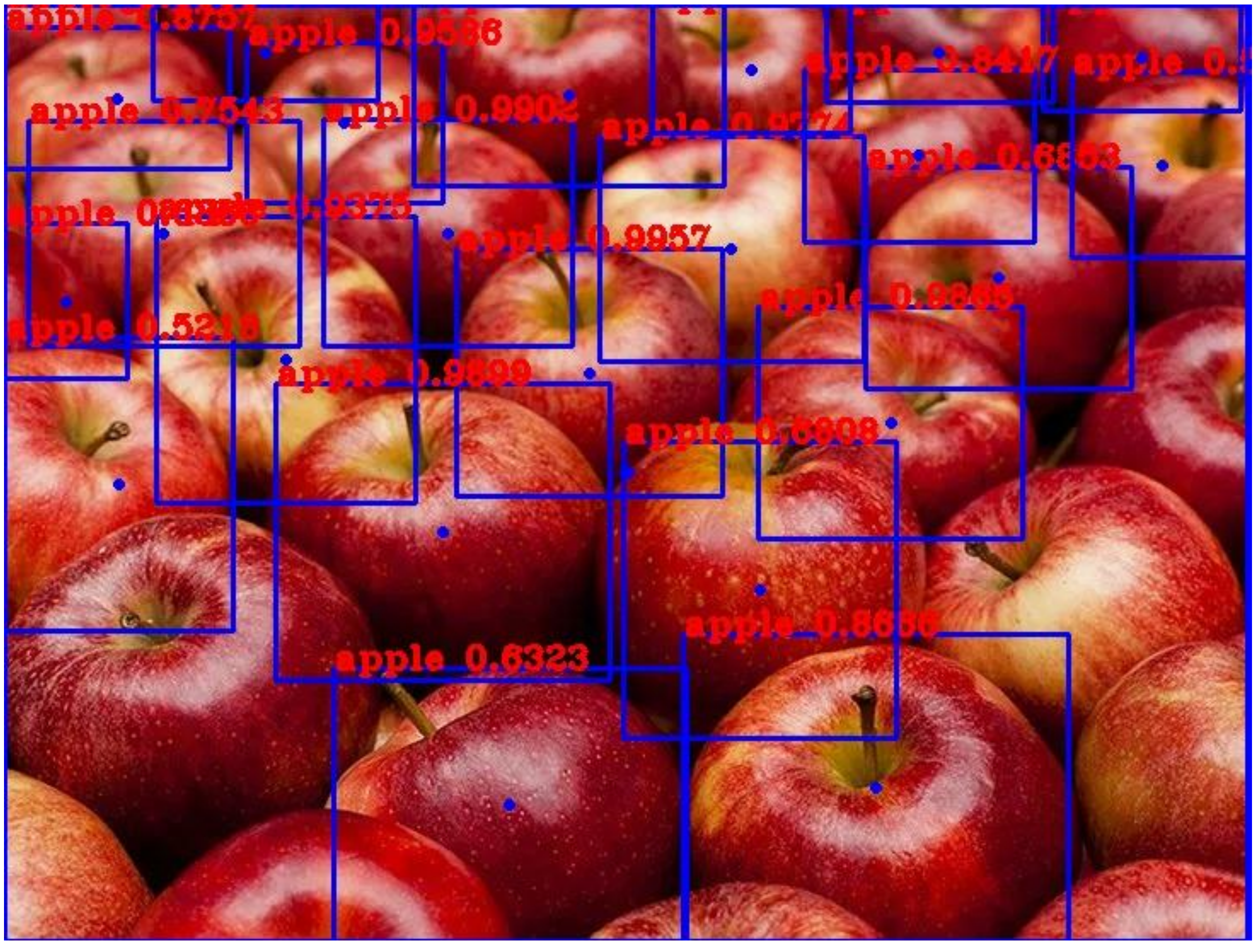


Figure 32: A bunch of red apples (Image Source: Getty Images)

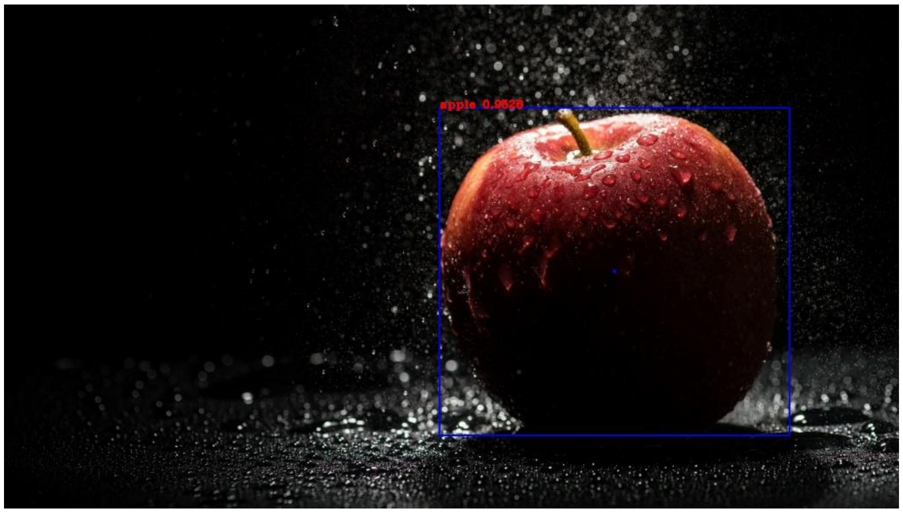


Figure 33: Apple shot under poor lighting conditions, with irregular surface (Source: WallHere Wallpapers)

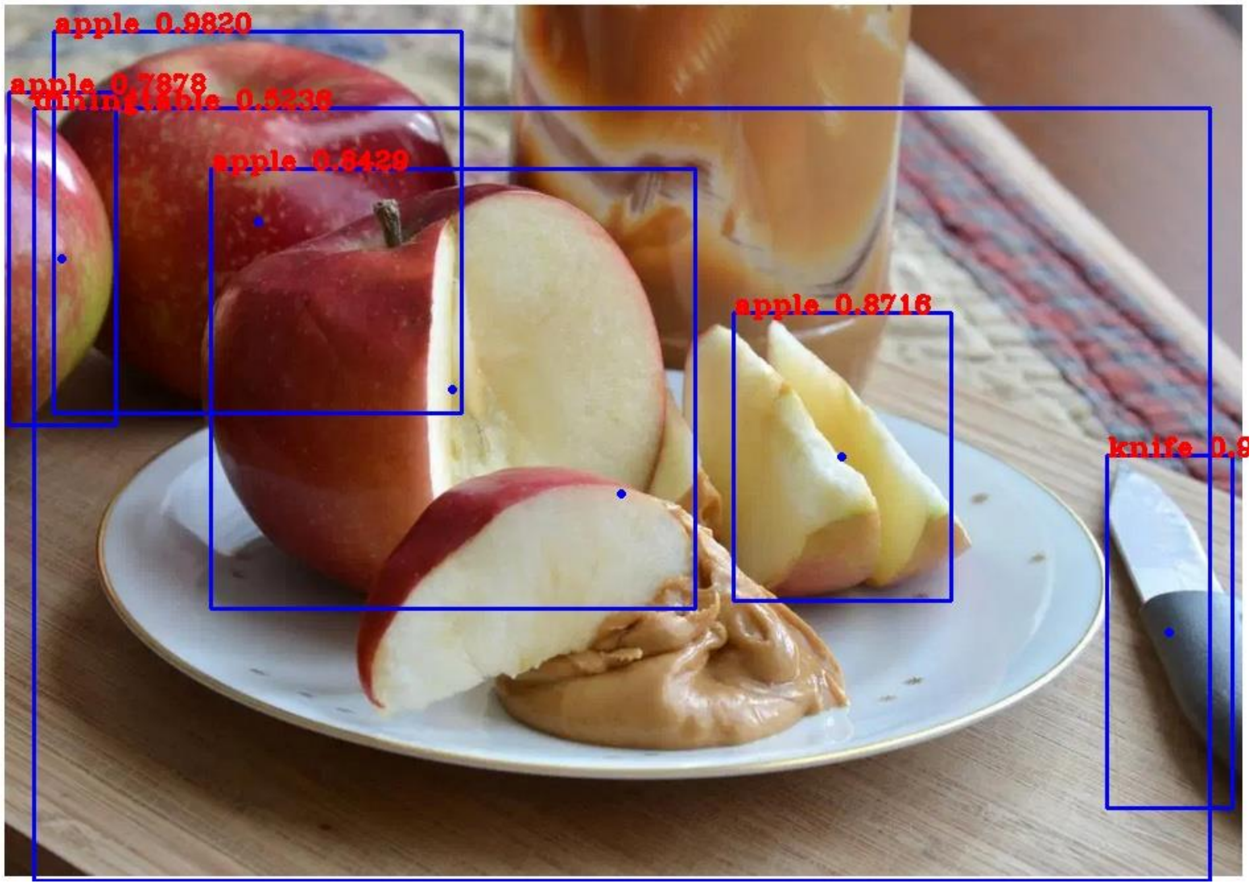


Figure 34: Chopped apples on a dining table (Source: MedicalNewsToday)

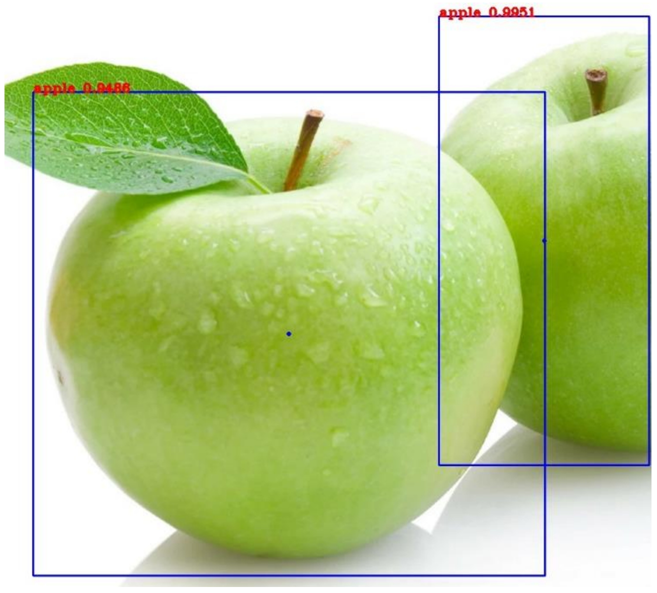


Figure 35: A pair of green apples, one of them being obscured (Source: [42])

4.1.2 Detection on 3D Model of an Apple in Coppeliasim

The following set of images are obtained from the simulated kinect by placing the apple directly in front of the lens and noting the outcome as it is moved further away, starting at 0.5m away from the lens, going upto 2.5m in increments of 0.5m each step. We also note the outcome when the apple model is viewed at a 45degree angle form above and below, and when viewed from directly above and below.

The results from each case is provided, with the heading denoting the position of the apple relative to the camera and the distance of the apple from the camera.

1. Directly ahead, 0.5m.

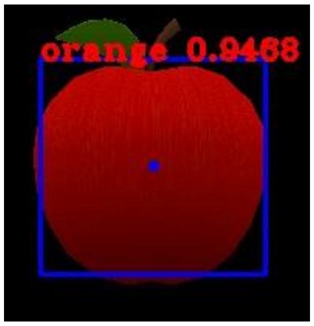


Figure 36: The apple model placed directly in front of the camera at a distance of 0.5m

2. Directly ahead, 1.0m

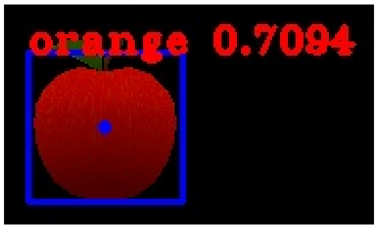


Figure 37: The apple model placed directly in front of the camera at a distance of 1.0m

3. Directly ahead, 1.5m

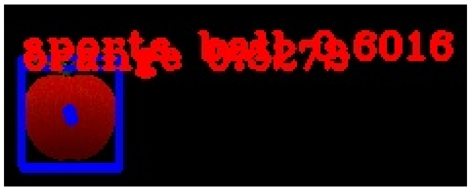


Figure 38: The apple model placed directly in front of the camera at a distance of 1.5m

4. Directly ahead, 2.0m

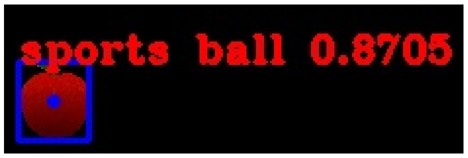


Figure 39: The apple model placed directly in front of the camera at a distance of 2.0m

5. Directly ahead, 2.5m

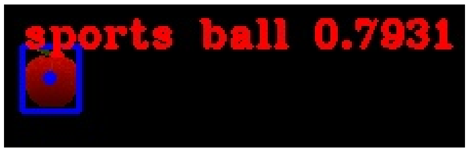


Figure 40: The apple model placed directly in front of the camera at a distance of 2.5m

6. Directly below, 0.5m

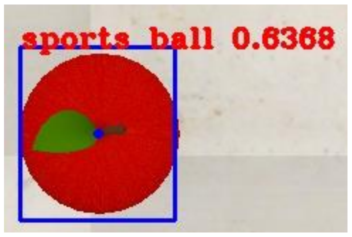


Figure 41: The apple model placed directly below the camera at a distance of 0.5m

7. Below, with the camera looking down at a 45 degree angle, 0.5m

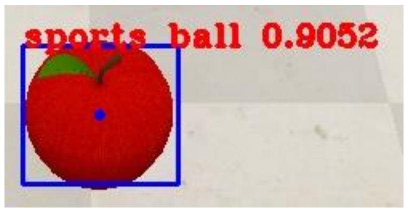


Figure 42: The apple model placed with camera looking down at a 45-degree angle, at a distance of 0.5m

8. Directly above, 0.5m

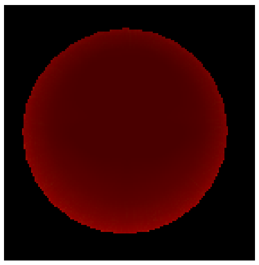


Figure 43: The apple model placed directly above the camera at a distance of 0.5m

9. Above, with the camera looking up at a 45 degree angle, 0.5m

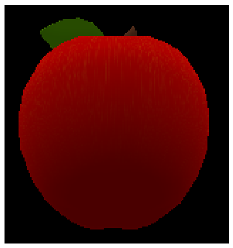


Figure 44: The apple model placed with camera looking up at a 45-degree angle, at a distance of 0.5m

As we can see, the YOLOv3 network fails to identify the apple model correctly or fails to detect it entirely because it does not resemble apples from the training set. When it does detect an object, we could use the location to instruct a robot to perform the relevant operations to pick it up.

4.2 TARGET POINT LOCATION TESTING

Using the simxGetObjectPosition command, the true position of the apple in simulated space can be obtained. We then compare the apple location computations. In this test, the position of the Kinect camera is kept upright, its position constant at (X, Y, Z) = (0, 0, 0.5)m with the depth sensor's positive Z axis facing along the positive Y axis of the world frame. The apple is moved to different positions. In each position, we test the accuracy of computations used to convert the location of the target point on a 2D image to a location in 3D space with reference to the base frame.

The apple positions are chosen such that the system can detect and identify the apple as a fruit.

SERIAL NO.	REAL POSITION (X, Y, Z) in m	CALCULATED POSITION (X, Y, Z) in m
1	(0.4, 1, 0.5)	(0.39, 0.98, 0.49)
2	(-0.4, 1, 0.4)	(-0.38, 0.98, 0.38)
3	(0, 0.45, 0.48)	(-0.05, 0.449, 0.468)

Table 7: Test results for the accuracy of target point location calculations

On comparison, we can see that the obtained values for target point from the transformation will be slightly towards the camera since the detected point is located on the surface of the apple, while the value returned by the API will be the volumetric center of the 3D object.

4.3 PROJECT OUTCOME

To demonstrate that the complete setup works, we follow the procedure described in Section 3.5; if the inverse kinematic computations produce a valid result, and the simulation procedure has been successfully completed, the outcome should be as follows:

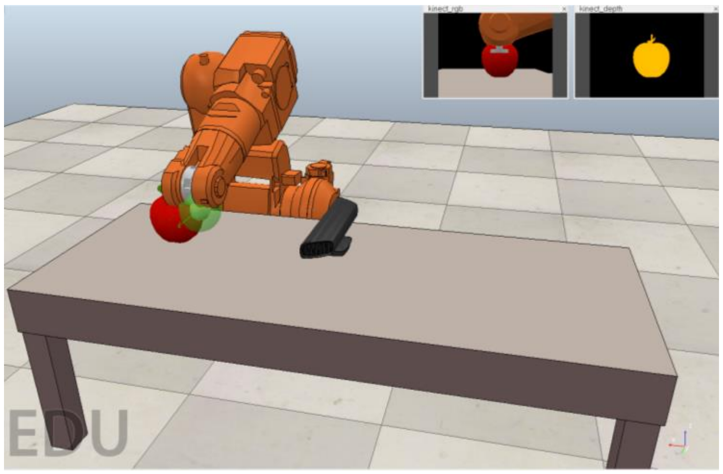


Figure 45: The robot arm has reached the target point

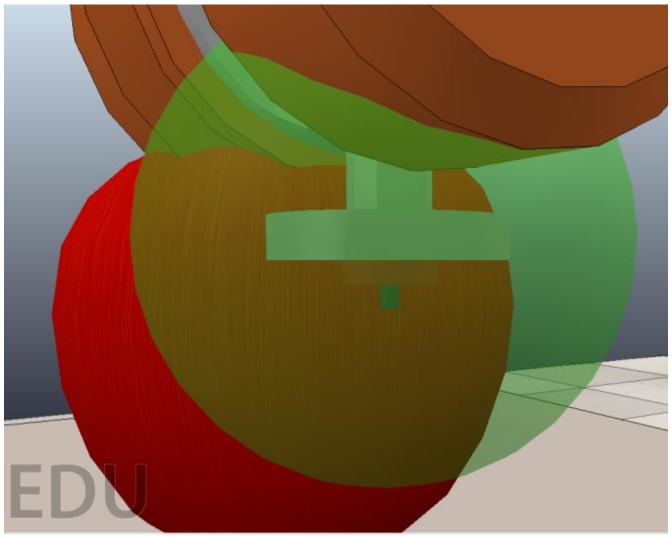


Figure 46: The center of the manipulator sphere is positioned precisely at the calculated target point

Taking a closer look, we can see that the tip of the manipulator sphere has precisely reached the target point. This demonstrates that our calculations are accurate, the robot can compute a valid set of joint parameters and is able to reach the target point with a high degree of accuracy.

5. SHORTCOMINGS

In this section we briefly go over some of the challenges faced in achieving the stated objectives.

Attempts at implementing Mask R-CNN and U-Net for semantic segmentation of apples also failed. Reasons for these failures are unclear, but it is possible that there may have been issues with the written programs as well as with the suitability of the dataset. Code for Mask R-CNN was adapted from [43] while that of U-Net was adapted from [44]. Further tweaking and testing may have provided a solution in this regard. The pretrained YOLOv3 network had to be applied due to time constraints.

As mentioned earlier in Section 3.2, attempts were made to improve YOLO-v3's ability to detect apples by training it on the Fruits 360 dataset. Fruits 360 is a dataset of over 90,000 images consisting of over a hundred fruits and vegetables. The fruits and vegetables were fixed to the shaft of a slowly rotating motor (3 rpm) and a short movie of 20 seconds was recorded.

Training images of apples from the fruits 360 dataset had been annotated with bounding boxes using VoTT. The annotations were saved in the tfrecord format and the network was trained using the “train.py” script.

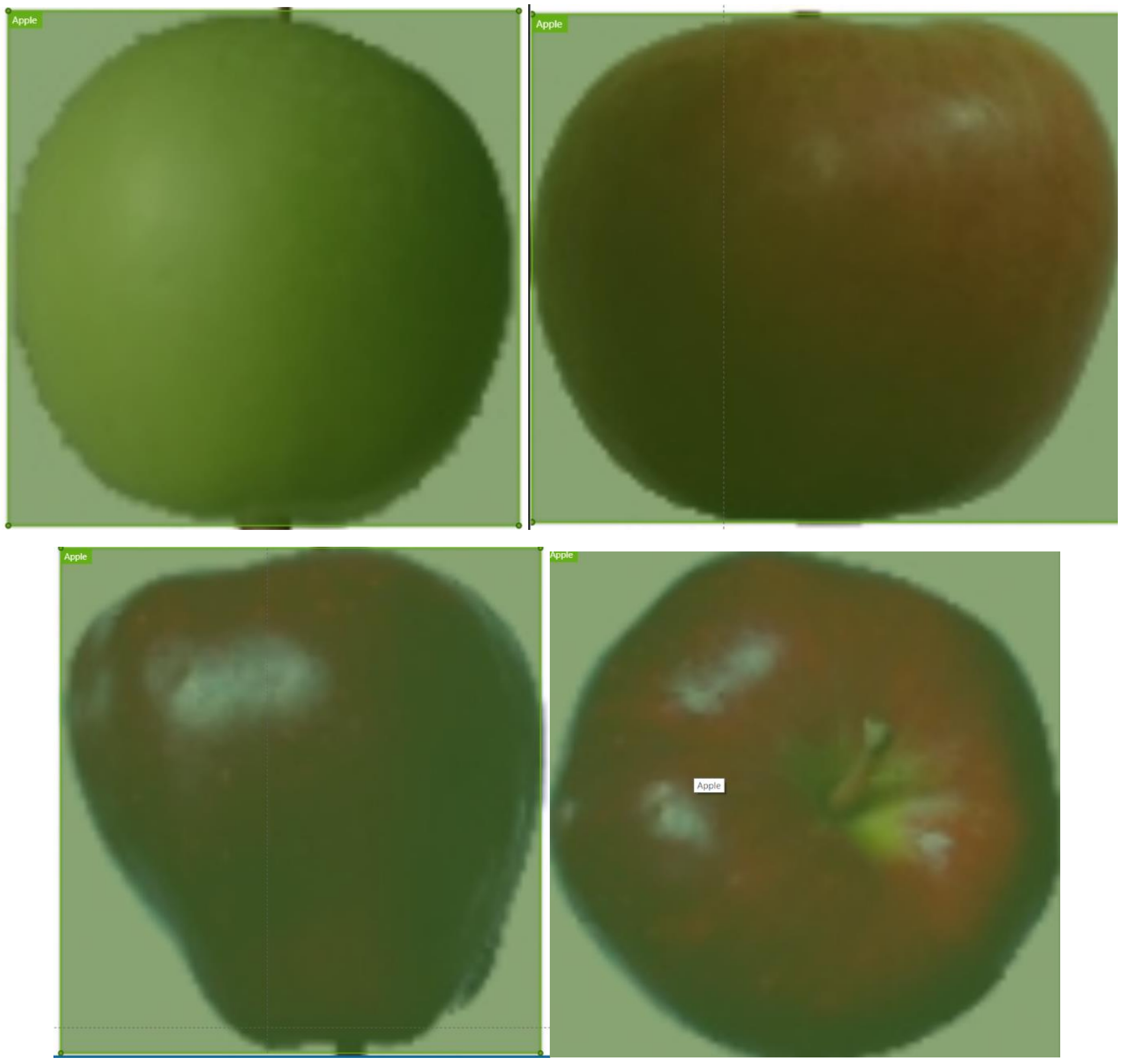


Figure 47: Examples of annotated apples from the Fruits-360 dataset

Training the network with this dataset did not yield meaningful results since the training images are of low resolution (100 x 100 pixels) and do not provide enough detail to enable the network to detect apples in test images of higher resolution than that of the training set. Down-sampling images did not result in apple detection. Successful detections were obtained only on the training set itself. Fruits-360 is better suited to train networks designed for image classification rather than object instance detection.

Attempts at implementing real-time apple detection did not bear fruit due to unexpected technical difficulties with the YOLO program and time constraints that prevented any attempts to overcome these issues.

In the future, a dataset for an image instance segmentation task should ideally consist of high-resolution annotated images of objects in practical scenarios for which the detection system will be applied, such as in our case, apples on trees in apple farms. Another way to compile a comprehensive dataset that could aid in achieving high quality semantic segmentation would be to follow a similar methodology as conducted in Fruits-360, but images obtained need be of higher resolution.

6. IMPROVEMENTS & FUTURE WORK

There remains ample room to improve the current implementation. Some of the objectives to pursue in the future are as follows:

1. Train a more robust neural network to semantically segment apples when viewed from any angle. This would require preparation of a new dataset that would involve a procedure such

as the one followed to create the Fruits-360 dataset. Annotations could be done in either the PASCAL-VOC or COCO formats.

2. Implement real time object detection and semantic segmentation.
3. Upgrade the neural network by training it to decipher pose of the apple such that an end effector can approach it without colliding with surrounding objects.
4. Design an end effector specifically for manipulating apples.
5. Determine actuation parameters for the robot arm to manipulate an apple without damaging it or surrounding objects. Machine learning algorithms for safe exploration can be applied here, such as the work presented in [28].
6. Develop solutions for dealing with obscured targets.
7. Develop a marketable product.
8. Implementation of end to end robot learning for object manipulation.

7. CONCLUSION

This research report describes how machine learning, machine vision and robotics can be applied towards apple picking. By applying the YOLOv3 object detection algorithm to images obtained from an RGB-D camera, apples can be detected and located in 3D space. If the apple is within operating range of a robot arm, it can be manipulated. This is demonstrated in a simulated environment and all details required to recreate the setup from the ground up have been provided.

This system is shown to achieve a level of flexibility that would not be possible to implement on a standard vision system that is programmed on a case by case basis, as can be seen in Section 4.1.1. Having trained a neural network on a robust dataset, it can detect apples in a range of different conditions. Errors that arise in the system can be overcome by further fine tuning the neural network with high quality training images.

This project provides a strong template for developing flexible real-world solutions for a variety of issues using machine learning and robotics. Machine learning based harvesting systems promise to provide a much-needed reprieve to New Zealand’s fruit farmers by reducing dependence on foreign labor. Additionally, machine learning based vision and robotics are poised to play a key role in sectors such as automated agriculture (weed detection [45], soil quality monitoring as seen in [46] & [47], seeding and planting crops.), improved manufacturing, health care, handling hazardous materials, etc. From studies conducted in robot learning, progress is being made towards creating completely autonomous robots that learn various skills without explicit programming. Overall, these technologies promise to deliver powerful tools to bring about a sea-change in how we automate and perform routine tasks.

REFERENCES

1. Urgent call for fruit-pickers in Hawke's Bay saw just 14 people express an interest (2018), https://www.stuff.co.nz/business/farming/102198933/urgent-call-for-fruitpickers-in-hawkes-bay-saw-just-14-people-express-an-interest

2. Recognized Seasonal Employer cap increase (2018), http://www.scoop.co.nz/stories/PA1811/S00044/recognised-seasonal-employer-cap-increase.htm

3. Seasonal labor shortage declared in Hawke's Bay (2019), https://www.immigration.govt.nz/about-us/media-centre/news-notifications/seasonal-labour-shortage-declared-in-hawkes-bay

4. 'Outstanding' apple season blighted by a lack of workers willing to pick them (2019), https://www.stuff.co.nz/business/111820040/outstanding-apple-season-blighted-by-a-lack-of-workers-willing-to-pick-them

5. Apple industry could stop growing if Government doesn't address labor shortage (2019), https://www.stuff.co.nz/business/farming/109836116/apple-industry-could-stop-growing-if-government-doesnt-address-labour-shortage

6. Global Industrial Robot sales doubled over the past 5 years (2018), https://ifr.org/ifr-press-releases/news/global-industrial-robot-sales-doubled-over-the-past-five-years

7. Eight Fastest Growing Technologies (2018), https://www.ificlaims.com/rankings-8-fast-growing.htm

8. Worldwide Semiannual Artificial Intelligence Systems Spending Guide (2018), https://www.idc.com/getdoc.jsp?containerId=IDC_P33198

9. Agricultural Robots (2019), https://www.tractica.com/research/agricultural-robots/

10. Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, *36*(4), 193-202.

11. LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, *1*(4), 541-551.

12. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). *Imagenet classification with deep convolutional neural networks.* Paper presented at the Advances in neural information processing systems.

13. Definition of Machine Vision, http://www.image-https://www.cognex.com/en-nz/what-is/machine-vision/what-is-machine-vision.org/

14. Long, J., Shelhamer, E., & Darrell, T. (2015). *Fully convolutional networks for semantic segmentation.* Paper presented at the Proceedings of the IEEE conference on computer vision and pattern recognition.

15. Ronneberger, O., Fischer, P., & Brox, T. (2015). *U-net: Convolutional networks for biomedical image segmentation.* Paper presented at the International Conference on Medical image computing and computer-assisted intervention.

16. He, K., Gkioxari, G., Dollár, P., & Girshick, R. (2017). *Mask r-cnn*. Paper presented at the Proceedings of the IEEE international conference on computer vision.

17. Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). *You only look once: Unified, real-time object detection.* Paper presented at the Proceedings of the IEEE conference on computer vision and pattern recognition.

18. Redmon, J., & Farhadi, A. (2017). *YOLO9000: better, faster, stronger.* Paper presented at the Proceedings of the IEEE conference on computer vision and pattern recognition.

19. Redmon, J., & Farhadi, A. (2018). Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*

20. Noh, H., Hong, S., & Han, B. (2015). *Learning deconvolution network for semantic segmentation.* Paper presented at the Proceedings of the IEEE international conference on computer vision.

21. Chen, L.-C., Papandreou, G., Kokkinos, I., Murphy, K., & Yuille, A. L. (2018). Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE transactions on pattern analysis and machine intelligence, 40*(4), 834-848.

22. Chen, L.-C., Hermans, A., Papandreou, G., Schroff, F., Wang, P., & Adam, H. (2018). *Masklab: Instance segmentation by refining object detection with semantic and direction features.* Paper presented at the Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.

23. Zhao, H., Shi, J., Qi, X., Wang, X., & Jia, J. (2017). *Pyramid scene parsing network.* Paper presented at the Proceedings of the IEEE conference on computer vision and pattern recognition.

24. Peters, J., & Schaal, S. (2007). *Reinforcement learning by reward-weighted regression for operational space control.* Paper presented at the Proceedings of the 24th international conference on Machine learning.

25. Khatib, O. (1987). A unified approach for motion and force control of robot manipulators: The operational space formulation. *IEEE Journal on Robotics and Automation, 3*(1), 43-53.

26. Saxena, A., Driemeyer, J., & Ng, A. Y. (2008). Robotic grasping of novel objects using vision. *The International Journal of Robotics Research, 27*(2), 157-173.

27. Lin, G., Tang, Y., Zou, X., Xiong, J., & Li, J. (2019). Guava Detection and Pose Estimation Using a Low-Cost RGB-D Sensor in the Field. *Sensors, 19*(2), 428.

28. Huang, S. H., Zambelli, M., Kay, J., Martins, M. F., Tassa, Y., Pilarski, P. M., & Hadsell, R. (2019). Learning Gentle Object Manipulation with Curiosity-Driven Deep Reinforcement Learning. *arXiv preprint arXiv:1903.08542*

29. Hämäläinen, A., Arndt, K., Ghadirzadeh, A., & Kyrki, V. (2019). Affordance Learning for End-to-End Visuomotor Robot Control. *arXiv preprint arXiv:1903.04053*

30. OpenCV, https://opencv.org/

31. VoTT, https://github.com/microsoft/VoTT

32. The KERAS API, https://keras.io/

33. TensorFlow, https://www.tensorflow.org/

34. V-REP, http://www.coppeliarobotics.com/

35. Microsoft Visual Studio Code, https://code.visualstudio.com/

36. CoppeliaSim User Manual, https://www.coppeliarobotics.com/helpFiles/

37. ABB IRB 140, https://new.abb.com/products/robotics/industrial-robots/irb-140

38. YOLO-v3 Implementation, https://github.com/zzh8829

39. COCO Dataset, https://cocodataset.org/#home

40. Convert 2D image position to 3D location, https://stackoverflow.com/questions/17832238/kinect-intrinsic-parameters-from-field-of-view/18199938#18199938

41. Apple Image, http://markethubnews.com/an-apple-a-day-keep-wrinles-at-bay-for-health/

42. Apple Image, https://www.vapebandit.co.nz/products/green-apple-wizard-labs

43. Mask R-CNN (Matterport implementation), https://github.com/matterport/Mask_RCNN

44. U-Net implementation, https://github.com/zhixuhao/unet

45. Lottes, P., Khanna, R., Pfeifer, J., Siegwart, R., & Stachniss, C. (2017, May). UAV-based crop and weed classification for smart farming. In *2017 IEEE International Conference on Robotics and Automation (ICRA)* (pp. 3024-3031). IEEE.

46. Wu, G., Kechavarzi, C., Li, X., Wu, S., Pollard, S. J., Sui, H., & Coulon, F. (2013). Machine learning models for predicting PAHs bioavailability in compost amended soils. *Chemical engineering journal*, *223*, 747-754.

47. Yamamoto, K., Guo, W., Yoshioka, Y., & Ninomiya, S. (2014). On plant detection of intact tomato fruits using image analysis and machine learning methods. *Sensors*, *14*(7), 12191-12206.