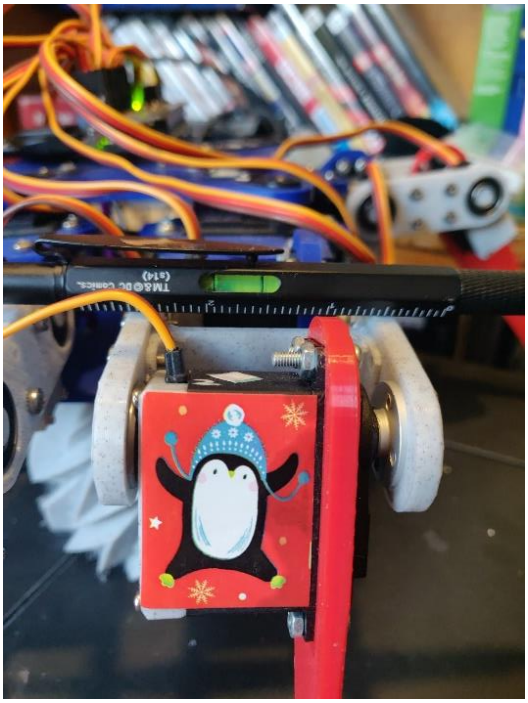


PENGUIN

A technical report on the development and testing of a quadrupedal robot

Benedict Fletcher: 10583327
Benedict.fletcher@students.plymouth.ac.uk

Abstract

This report identifies the design and implementation of a 4-legged walker robot, based around Pulse Width Modulation (PWM) servo control. The project revolves around the implementation of inverse kinematics to recreate a natural gait. The servo control and gait implementation has been designed around the raspberry Pi 0, combined with the Adafruit PWM bonnet. The Raspberry Pi will be powered by a battery pack, whilst the servos will be powered separately by a Turnigy 5000mAH LiPo battery with a XL4015 voltage transformer to reduce the voltage. The robot is designed to explore uneven small places, the large battery and compact design allows it to do this.

The robot has been designed as a low cost, easy to replace product with an estimated cost of under £200. This allows the robot to be sent into dangerous situations where there is a high chance that it will be destroyed in its mission. The low cost will democratise the use of search robots.

The 3D printed, modular design, will promote easy replacement or upgrades to broken parts. This allows for cheap maintenance of the robot and easy upgrades without having to purchase a completely new product. Further promoting accessibility to the robot to all areas of the world. The long feet but short upper leg promotes good ground clearance for walking through rubble, whilst also keeping a high rigidity in the leg to stop it from flexing with each step.

Acknowledgements

The author would like to acknowledge the invaluable information that was provided by his personal tutor, Chenxu Li. The author would also like to express his gratitude to Jake Gibson Shaw-Sutton who provided invaluable advice on whether to use Jacobians to calculate the inverse kinematics or not.

The author would like to thank his parents and friends for their continued support, with particular focus on his father for his assistance with object orientated Python code.

Contents

Abstract 1
Acknowledgements 2
Contents 3
Glossary 6
Introduction 7
Research 8
- Natural disasters 8
- Rescue robots 9
  - Scenarios 9
  - Types of rescue robots 10
- Gaits in nature 12

Iterative prototype design 13
- Components 13
- Design 16
  - MK1 design 17
  - MK2 design 18
  - MK3 design 19
  - MK4 design 20
  - MK5 design 21
  - MK6 - Final Design 22

Programming 27
- Goals of project 27
- Classes 27
  - Robot class 27
  - Leg class 28
  - Servo class 28
- Program logic 28
  - Phases of the gait 29
  - Robot class 31
  - Leg class 32
  - Servo class 35
- Camera code 36

Robot features 39

Remote viewing 39
Remote movement control 39
Testing 40
Calibration 40
Servo limits 41
Offset 42
Gait 43
Future Work 44
Squad robots 44
SLAM 44
WiFi mesh 46
Audio 46
Conclusions 48
References 49
Appendix: supplementary figures 51

Figure 1: Global deaths from natural disasters[21] 8
Figure 2: Number of deaths due to earthquakes[21] 9
Figure 3: Drone in use, surveying disaster area 10
Figure 4: iRAP teleoperative robot[26] 11
Figure 5: Stanford university's snake robot 11
Figure 6: Drawing emphasising body and tail twist[14] 12
Figure 40: Arducam UC-376 16
Figure 7: MK1 design 17
Figure 8: MK2 design 18
Figure 9: MK3 design 19
Figure 10: MK4 design 20
Figure 11: MK5 design 21
Figure 12: Types of servo horn 21
Figure 13: MK6 leg design 22
Figure 14: (1) Front full body, (2) Rear leg, (3) Front leg 22
Figure 15: I-Beam to stop twisting 23
Figure 16: Back plate of shoulder-wrist reinforcement 24
Figure 17: Top-down view of back brace 24
Figure 18: Servo plate of shoulder-wrist joint 25
Figure 19: Servo plate of shoulder-wrist join dimensions 25
Figure 21: Top-down view of servo back 26
Figure 20: Custom servo back with bearing pin 26
Figure 22: Class diagram 27
Figure 23: Robot class overview 27
Figure 24: Leg class overview 28
Figure 25: Servo class overview 28
Figure 26: Main 28
Figure 27: Phases of the gait 29
Figure 29: Forward gait 30
Figure 28: Backward gait 30
Figure 30: Constructor function 31
Figure 31: stepPhases function 31
Figure 32: Step functions 32
Figure 33: Leg constructor 32
Figure 34: moveTo function 33
Figure 35: legLift function 33
Figure 39: Trigonometric diagram of robot leg 34
Figure 36: Servo constructor 35
Figure 37: setAngle function 35
Figure 38: Lynxmotion servo control syntax 36
Figure 42: Initialises the camera code 36
Figure 41: Web architecture 37
Figure 43: Streaming handler class 37
Figure 44: /index.html 38
Figure 51: Servo controller program 40
Figure 52: Sets the servo's angle to 90 41
Figure 54: Physical limits of shoulder joint 41
Figure 53: Min and max check 41

Figure 55: Measuring the offset between two legs 42
Figure 56: Forwards step list of phases 43
Figure 57: Anticlockwise list of phases 43
Figure 45: University of Plymouth's robot football team 44
Figure 46: Robot used SLAM to navigate through room 45
Figure 47: Occupancy grid generated and room it's in 45
Figure 48: Wi-Fi mesh between raspberry Pi's 46
Figure 49: Scary robot 47
Figure 50: Robot to the rescue 47
Figure 58: Side view of back brace 52
Figure 59: Bottom-up view of back brace 52
Figure 60: Side view of servo back 52
Figure 61: Bottom-up view of servo back 52

Glossary

Inverse kinematics: The maths to calculate servo angles based upon a inputted coordinate

PWM: Pulse Width Modulation

DOF: Degrees Of Freedom

PLA: Polylactic acid

DoF: Degrees of Freedom

LDR: Light Dependant Resistor

IR: Infra-red

SLAM: Simultaneously Localisation And Mapping

FPS: Frames Per Second

UAV: Unmanned Aerial Vehicle

LiPo: Lithium Polymer

CAD: Computer Aided Design

Introduction

Global deaths form natural disasters have decreased dramatically over the past decade[1] [2]. This is partly because better precautions are implemented in building structures and sea walls that reduce the damage caused. Tokyo is a good example of this when it got hit by the 8.9 magnitude earthquake in 2011. This earthquake showcased the strength of modern earthquake resistant buildings as video footage surfaced of skyscrapers swaying back and forth yet remaining standing, thus saving thousands of lives[3].

However, these building renovations are expensive and countries with less income will not be able to afford these precautions such as Haiti which was affected to a much greater extent by their earthquake even though it was of a lesser magnitude[3]. It is well known among departments responsible for rescue that it is desirable for a rescue to be within 3 hours, and that the survival rate becomes minimal after 72 hours, referred to as 'the golden 72 hours'[4]. This is where rescue robots would help increase the number of victims located in the golden 72 hours through the combination of several different types of robot. Surveillance drones and helicopters scan a wide area to develop a larger understanding of the scenario and help direct rescuers to the worst affected areas[4]. On-rubble wheeled, and crawler robots are deployed on top of the rubble to get a more detailed scan of the disaster site[4], they will scan for victims trapped and provide structural feedback of the building to aid and keep safe rescue workers. In-rubble robots do much the same but work on an even smaller scale, their job is to pinpoint victims and provide building structure scans of areas covered in rubble[4].

Research

Natural disasters

Natural disasters are an inevitable truth. There is no stopping them, only reducing the damage they cause. Our World in Data[21] reports, in the early-to-mid 20th century, the annual death toll from natural disaster was often reaching over one million per year. This number has significantly dropped in recent decades. In most years there are less than 20,000 deaths, in the most recent decade this has been less than 10,000.

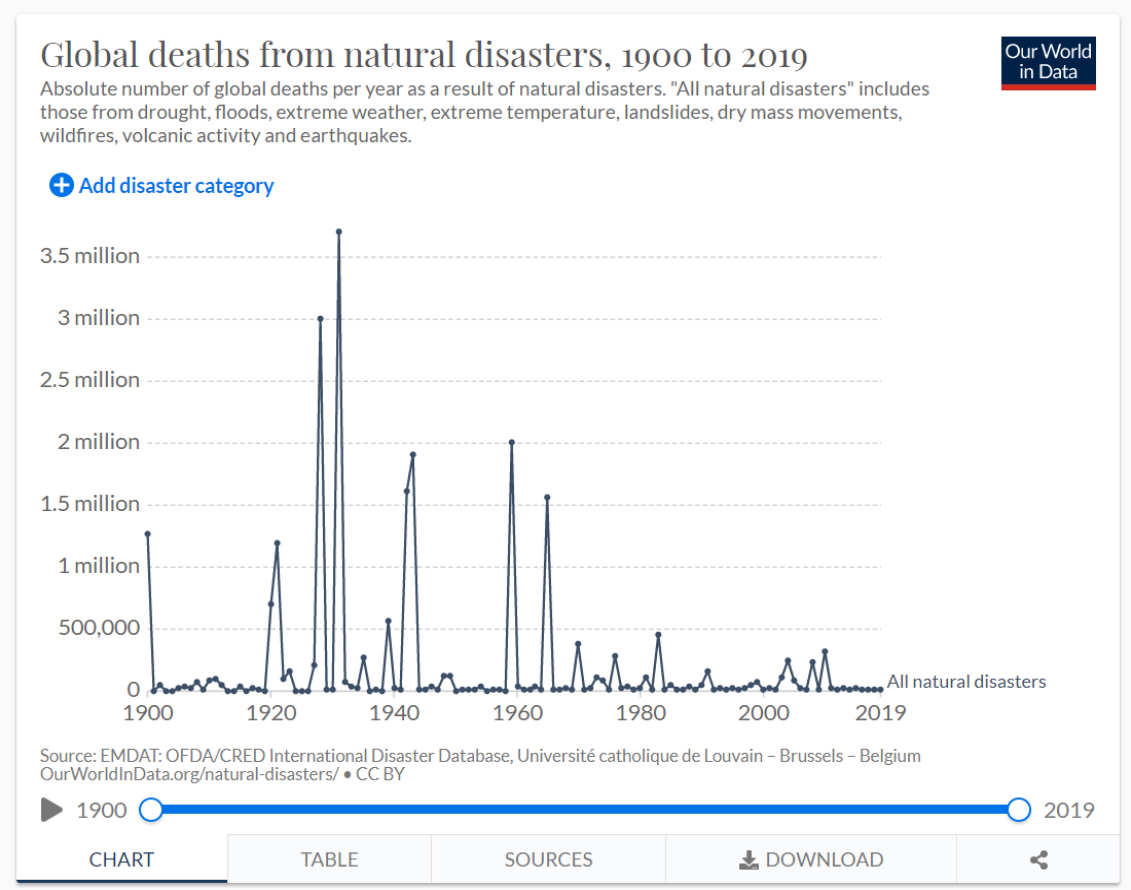


Figure 1: Global deaths from natural disasters[21]

Earthquakes have been a large contributing factor to the annual death toll since records began. This is because they are impossible to predict accurately and are very sudden. They effect taller buildings more because the shaking causes the building's supports to crumble and then the whole building will collapse. There is the possibility that this will cause surrounding buildings to be demolished, causing a cascading effect.

The number of deaths due to earthquakes has been on the rise in recent decades due to the population rise and density of cities. In low-income areas of the world the buildings are not constructed in an earthquake resistant manner. In 2011 Japan suffered a magnitude 9 earthquake and lost 16,000 lives, whereas in 2010 Haiti suffered a magnitude 7 earthquake and lost 200,000 lives : over 12 times more deaths from an earthquake that is 100 times smaller. One of the causes of this drastic difference in deaths is Japan's extremely tight building regulations, earthquake preparedness, and response effectiveness.

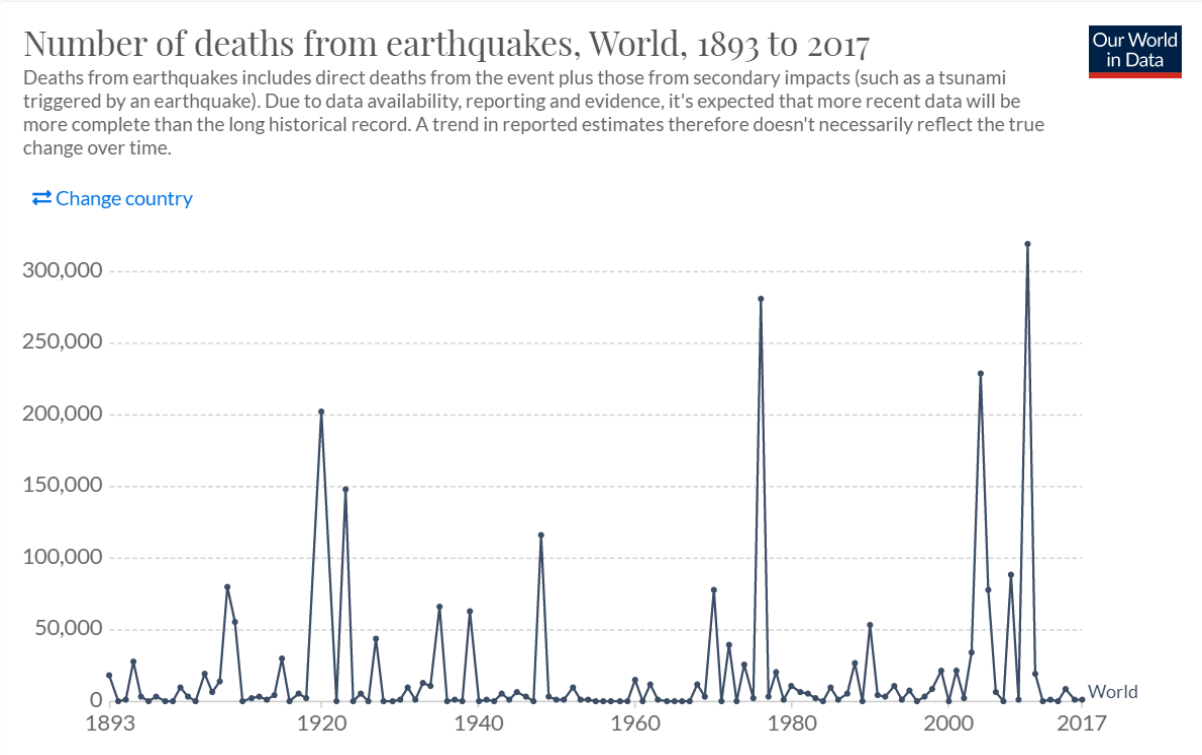


Figure 2: Number of deaths due to earthquakes[21]

This drastic difference in deaths showcases the different effects earthquakes can have on a low-income country compared to a high-income country. In 2010, the year of Haiti's earthquake, Japan made over 150 times more in annual income. It can afford to build all its large buildings to a high standard whereas Haiti cannot. They are also able to afford better search and rescue equipment such as rescue robots.

Rescue robots

Scenarios

Fumitoshi M. et al.[4] states that the process of search and rescue in earthquakes disasters are[22, 23]:

1. Gain awareness of remaining victims:

   Check for voices from rubble piles, information from their family and neighbourhood, and every other information resource.

2. Understanding the situation of the disaster site:

   Understand the risk of collapse, gas leaks, distribution of dangerous objects and fire of the disaster site.

3. Operation planning:
   Plan the procedure, role sharing, and logistics before entering disaster site. Supporting parties, machinery and materials are requested.

4. Search and reconnaissance:

   Detailed positions of victims and structural damage are identified. This information is necessary for human rescue parties.

5. Excavation of rubble piles:

   The search process sometimes requires excavation. This needs careful planning to ensure safety of rescue operatives.

6. Capture victim bodies:

   Retrieval of victim's bodies will commence after excavation if excavation is required.

7. Emergency first aid:

   Diagnosis, first aid, care, and psychological support are performed. Confined space medicine has progressed to give emergency care in rubble piles in the search process of 4.

8. Transfer victims to hospitals:

   Traffic jam and capacity of hospitals are serious problems in large-scale disasters. The police may need to be involved to block off necessary roads to ensure smooth transition of emergency vehicles.

9. Mental care for victims and aiding people

   As symptoms of Post Traumatic Stress Disorder (PTSD) are delayed about a month after traumatic experience, it is hard to make a diagnosis of PTSD. If symptoms are too delayed, and appropriate treatment and help could not be received, stress disorders pass into a chronic state and more serious situations may occur[24].

Types of rescue robots



Figure 3: Drone in use, surveying disaster area

Surveillance – Drones are used for scouting over vast areas of land to increase awareness of the current situation. Drones were used to map the damages in Nepal after it suffered an earthquake in 2015 that killed nearly 9,000 people and left 3.5 million homeless. These maps were used to help re-build their homes quicker and more safely as well as build in such a way to reduce future damage.

On-Rubble robots – Crawler, wheeled and jumping robots are used to collect more detailed scans of close-up rubble. This would be a potentially dangerous job if it were to be performed by a human because they would be exposing themselves directly to a very dangerous situation that could lead to injury or death.

The iRAP robot[26] is a morphing tracked robot that uses six tracks, two large central ones and four flippers that it rotates to allow the robot access to previously blocked areas. This robot has the capability to navigate extreme environments and path over fallen rubble with ease due to its length (1m) and its use of flippers.

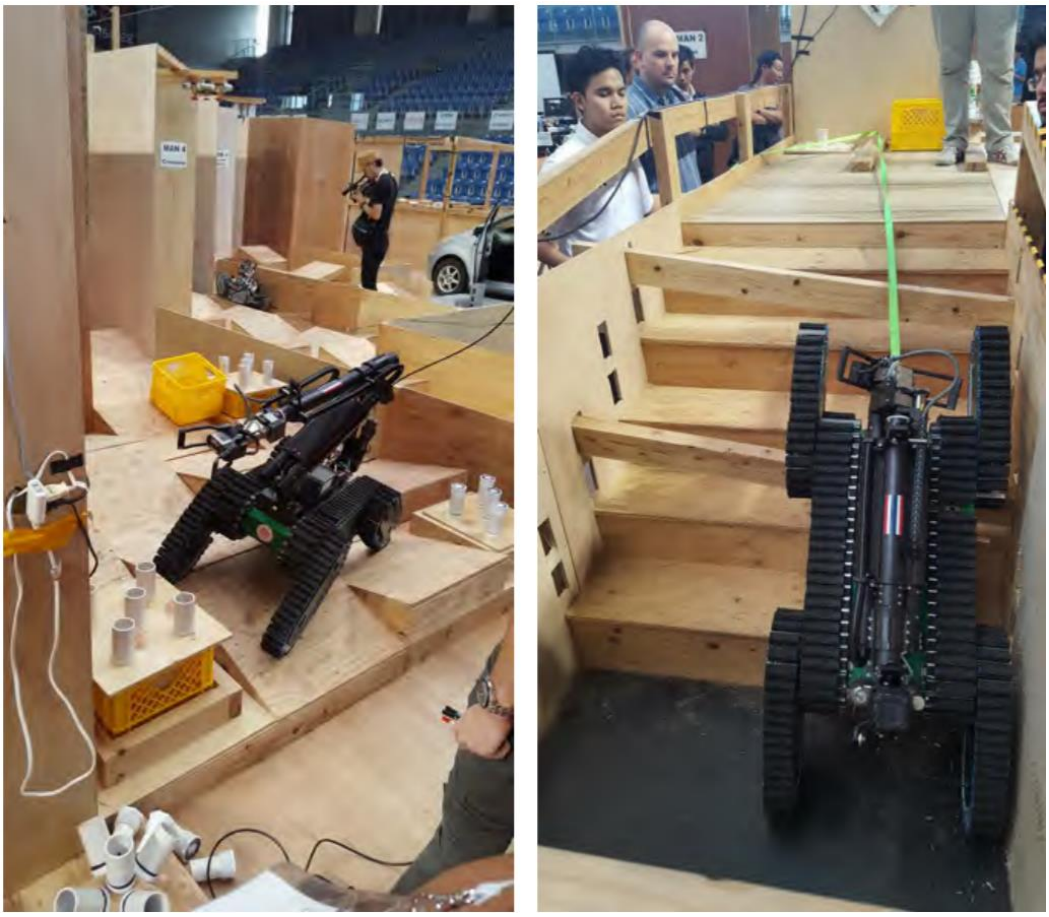


Figure 4: iRAP teleoperative robot[26]

Underground robots – Robots that go inside rubble piles to locate victims and survey the internal structure. An example is the robot developed at Stanford university[25] that can conform its body to fit inside of tight spaces and is highly durable. This robot has a small camera fitted to the end of the head, allowing rescuers to view otherwise unreachable places beneath the rubble.



Figure 5: Stanford university's snake robot

On February the 23rd 2016 Didcot power station, UK, was demolished. Four workers died in an unexpected building collapse part way through. The bodies of the victims took 6 months to be

recovered due to the area being unsafe to navigate. This scenario would be a perfect situation for rescue robots because they could enter the rubble pile without any risk to human life and locate the bodies. This would help the rescue operation because it could be narrowed down in scale to exactly where the bodies are located.

Gaits in nature

Lizard's gait

Research was done into different walking gaits in search of the correct one for the robot. It was decided to base it off the lizard's gait because they have the same leg layout as the robot. According to R. McNeill Alexander[14] lizards walk or run by moving their diagonally opposite feet simultaneously. Their body and tail play a major role in their movement. Both twist to form an S shape, making their centre of balance a straight line down the centre of their body which enables them to lift two legs off the ground at one time.



Figure 6: Drawing emphasising body and tail twist[14]

Crawl gait

The robot cannot do a lizard gait because it does not have a twistable body or tail. The crawl gait was elected instead because it only moves one leg at a time leaving three points of contact on the ground at all times. However, the walking motion is not smooth. It uses a similar principal as bipedal walkers where it catches itself on each step before it has a chance to fall over. This, combined with basic control of the servos has led to a jerky gait.

One solution to this would be to move the body during the gait such that the centre of balance moves inside the other legs. This will ensure that the robot is balanced in all stages of its gaitand smooth the gait, but this will also slow the walk down for what has been found to be a minor issue.

Iterative prototype design

Components

Raspberry Pi 0 W

This project requires a small, low power, light weight embedded system that has enough computational power to handle multiple servos, cameras for object recognition and potential upgrades such as ultra-sonics or lidar for simultaneous localisation and mapping (SLAM). We found the Raspberry Pi 0 W to be the best candidate as it boasted high processor speeds and large amounts of RAM, enough to handle all 12 servos with inverse kinematics and object recognition using mediapipe and openCV. It is WiFi enabled allowing for remote control.

The Arduino Uno is smaller and requires less power to run but the drawbacks in its processor speed, amount of RAM, and memory are too great. The Raspberry Pi 4, whilst possessing the largest amount of RAM and fastest processor speed, was too power hungry and large. It also contained many unnecessary pieces of hardware such as ethernet ports and multiple USB ports.

	RASPBERRY PI ZERO W [5 6]	**ARDUINO NANO** [8]	**RASPBERRY PI 4** [7]
COST (£)	4.80-9.30	17.22	33.9-73.50
PROCESSOR SPEED (GHz)	1	16**MHz**	1.5
RAM	512MB	2KB	2,4,8GB
CORES	1	1	4
POWER REQUIREMENT (mA)	120	19	1010
SIZE (mm)	65x30	45x18	85.6x56.5

Table 1: Embedded system comparison

Servo

This robot required high-torque, cheap servos to fulfil the project requirement of it being a cheap yet robust alternative to modern rescue robots. We researched a few alternatives and elected to use the MG996R servos as they provided suitable torque (9-11KG/CM) to enable the robot to carry additional sensors or potentially medical first aid such as clean clothes or bandages. However, these servos have a high current draw of 500-900mA when in operation, which requires a large battery to achieve extended periods of untethered operation. We decided to supply the servos 5V to account for any voltage fluctuations in the testing process of the hardware.

Large batteries will add a significant amount of extra weight and will require more space. We considered using SG90 servos as they were significantly cheaper to use and did not require large, expensive batteries to operate. We decided to not use these servos because, based on previous project experience, theses servos would not be strong enough to achieve the results required of this robot. The DS5160 servos were ruled out because they were too large and power hungry. To use these servos, we would need a car battery or similar sized battery. They would, however, allow for 7the robot to carry much more equipment or supplies. This could lead to them being used in a different role, that being as a platform that is able to extract victims or carry heavy equipment for rescue workers. Another possibility is to have it act as a sensor hub that coordinates teams of robots and uses its larger size and strength to contain more powerful computers as well as sensors.

	SG90 [9]	MG996R [11]	DS5160 [10]
WEIGHT	14.7g	55g	158g
VOLTAGE	4.8-6V	4.8-6V	6-8.4V
RUNNING CURRENT	170-270mA	500-900mA	3.5A-6.2A
SIZE	23x12x28.5mm	40.7x19.7x42.9mm	65x30x48mm
TORQUE	2.5 KG/CM	9.4-11KG/CM	58-70KG/CM
ROTOTATION RANGE	~180°	~120°	~180° OR ~270°

Servo controller

This robot has four legs, which each have three degrees of freedom (DOF). So a total of 12 servos ae required. The Raspberry Pi zero does not have enough PWM pins to support this number of servos. We overcame this issue by using a servo controller to control the servos. We needed a servo controller that was low cost, compatible with the Raspberry Pi 0 W and can handle the current draw of the servos. The number of servos will lead to the requirement of an external power supply as the Raspberry Pi Zero will not be able to supply the voltage or current.

After researching different boards, we found the Lynxmotion SSC-U32 servo controller that is capable of handling up to 32 servos and includes 8 analogue pins. It can handle 15A per side, ideal for my 6 servos per side. The 8 analogue pins allow for extra sensors for future upgrades.

The Adafruit bonnet and WitMotion were ruled out because they were not able to handle the peak current draw of the servos if they were to get stuck or exert more force than nominal.

	Lynxmotion SSC-U32	Adafruit PCA9685[28]	WITMOTION 16[29]
Number of channels	32	16	16
Size (mm)	76.2 X 58.42	62.5 X 25.4	43.5 X 36
Analogue pins	8	0	0
Maximum motor current (A)	15A per side	5	5
Motor supply voltage (V)	4.8-16	5-12	5-7.2

Battery

Great consideration went into the choice of battery because there are many factors that affect it.

Battery type: A Lithium Polymer (LiPo) battery was chosen because it has a large capacity, therefore it can power the servos for an extended period.

Weight: The battery will be fixed to the robot's body, to reduce the servos current draw we need a battery that weighs as little as possible.

Size: For unimpeded movement, a smaller battery is more desirable. This will also affect the overall size of the robot body as the design will need at accommodate the battery.

Power: We decided upon a search time of 1 hour per battery charge as this would be enough time to locate victims in a collapsed building and foray this information to rescuers. After testing we found that the current draw of the servos on average was 4A when in its walk motion. This led us to choose 4000-5000mAh batteries.

We chose the Turnigy brand because we were recommended them by a technician at the University of Plymouth and they were relatively cheap, reliable batteries.

	TURNIGY 5000MAH	TURNIGY 4000MAH	HRB 5000MAH
COST (£)	31.28	28.02	55.99
SIZE (mm)	147x49x33	148x50x29	155x48x32
WEIGHT (g)	552	452	492
TIME FOR DISCHARGE UNDER CONSTANT USE (MINUTES)	75	60	75
CONSANT DISCHARGE (C)	25	30	50
CELLS	4	4	4
OUTPUT VOLTAGE (V)	14.8	14.8	14.8

Voltage converter

The maximum voltage the Lynxmotion board can receive from external sources through its V+ input is 16V whilst the servos can receive up to 6V. The battery we chose, the Turnigy 5000mAh, is a 4 cell LiPo and, therefore, can output a maximum of 16.8V. The servos have a maximum input voltage of 6V, hence the need for a step-down voltage converter.

The step-down converter needs an input range that included 16.8V and an output range that included 5-6V. It is required to handle a constant 4A current draw with a peak of over 10A. Due to the space constraints the smaller the converter the better.

We decided to use the Pololu 15A step-down voltage regulator because it was cheap, small, and able to handle the high current draw of the servo's peak. We decided against the XL4015 because, whilst it could handle the voltage conversion, it could not supply enough the current to the servos. The LM2596 was disqualified for the same reasons as the XL4015.

	XL4015	LM2596	D24V150F5
COST (£)	5.49	9.59	31.19
INPUT VOLTAGE (V)	8-36	4-40	Up to 40V
OUTPUT VOLTAGE (V)	1.25-32	1.25-37	5
CURRENT (A)	5	2-3	5-20
SIZE (MM)	52x26	65.9x36	43.18x31.75

3D printed parts

The physical structure is an area where a large amount of weight lose can be achieved. Choice in material is responsible for a large part of this as materials have different properties. For this robot the desired material would be:

Light weight – This is to reduce the load on the servos.

Rigid – This is to reduce flexing in the material.

Cheap to manufacture – This Is because we require custom designed pieces which may need to be manufactured specifically for this project.

We decided to use 3D printed PLA using the Creality Ender 3. We reached this decision on the grounds of ease of manufacturing because 3D printing is widely available and does not require large manufacturing plants to construct its shapes. This can lead to the robot being constructed in any country, which reduces import costs and provides extra jobs inside that country as they will require technicians to maintain and operate the printers as well as construct the robots.

This robot is designed to go into dangerous situations where the body is likely to be damaged. Creating replacement parts will be much cheaper with 3D printing because the parts can be rapidly manufactured in a small room. If one printer is used the production rate is 1 robot per 24 hours. However, this can be drastically reduced by adding more printers.

Camera

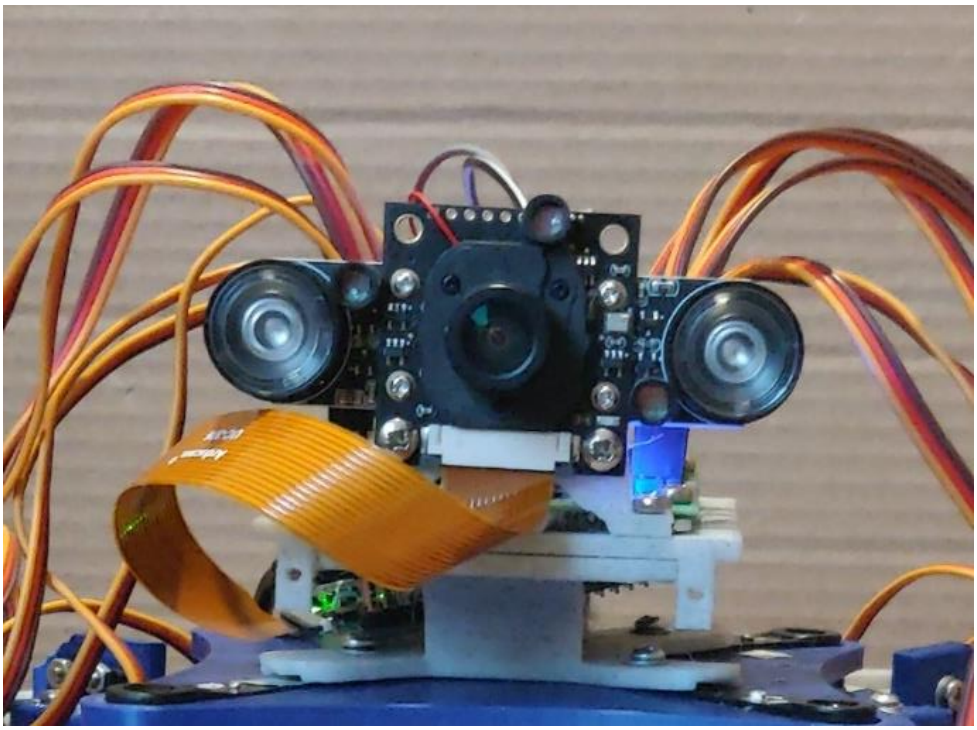


Figure 7: Arducam UC-376

Following research of the requirements for a camera used in rescue operations, we decided that the camera must be:

- Small
- Light weight
- Infra-red (IR) enabled
- Compatible with the raspberry Pi

We sourced the Arducam UC-376[15] which is a small, lightweight raspberry Pi camera. One key feature of this camera is its ability to mechanically switch between IR mode and non-IR mode using a light dependant resistor (LDR). Upon testing this camera has high quality video footage in both modes and has automatic exposure control which will be important for when the robot moves from a dark area to a light area.

Design

We used fusion 360 to design the robot because it was the computer aided design (CAD) software, we were familiar with and owned. We chose this over other CAD software such as Solidworks because we had previous knowledge of Fusion 360, meaning a reduced design time.

The robot developed upon existing robotic designs and insights that we discovered during our previous modules. Here we shall cover the iterative design milestones that we made to get from mk1-mk6, the current design.

MK1 design

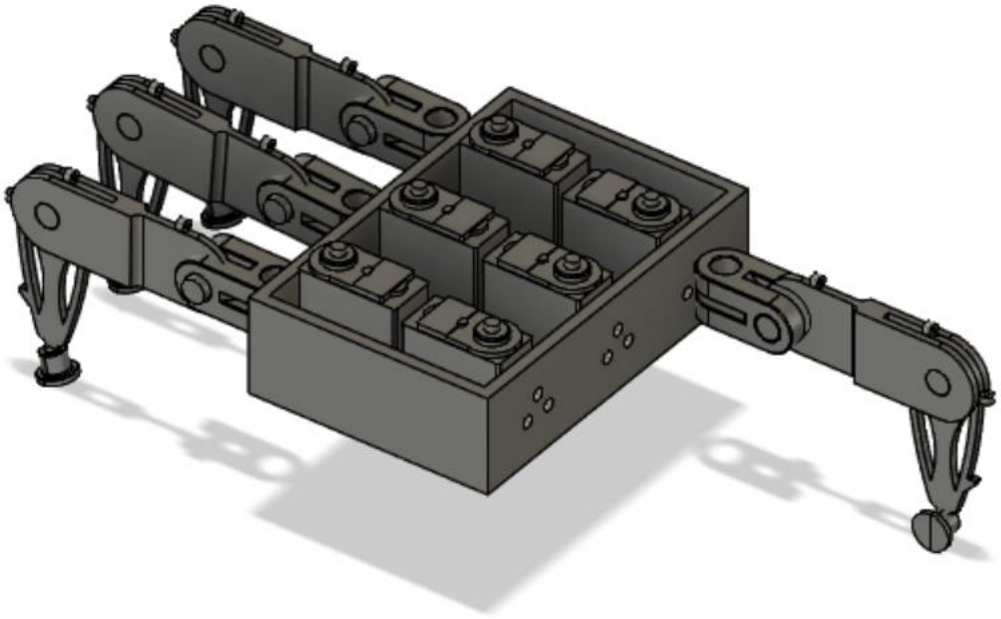


Figure 8: MK1 design

This was the first iteration of our robotic walker and, therefore, contained many areas for improvement. 6-Legs provided greater stability during a gait as there would always be at least 3 points of contact to the ground, forming a stable triangle shape. We decided to use tendons to control the movement of the legs as this meant that we could store all the servos in the body and leave the legs as light as possible, centralising the weight of the robot.

Testing

This design did not get past the concept phase, no testing was completed.

What we learnt and kept

What we learnt:

1. After modelling the servos, we found that they were larger than we initially expected, and the body would have to be much larger than we desired to accommodate all 6 servos. Including six MG996R servo motors would be over our project specification for this module so we reduced the number of legs to four legs.
2. The medium and manufacturing process of the robot was not conducive to the design, therefore, we changed the box design to suit 3D printing more closely.
3. Using tendons would require each servo have enough room to rotate a servo horn. This would further increase the size of the robot and invalidate the box design of the body.

What we kept:

1. We kept the shoulder joint as it was very simple, conducive to 3D printing and performed its role perfectly. We did make small changes to it later but the overall design did not change much.

2. We found that using a passively conforming foot reduces on the need for extra foot control and would increase the amount of grip the robot has.

MK2 design

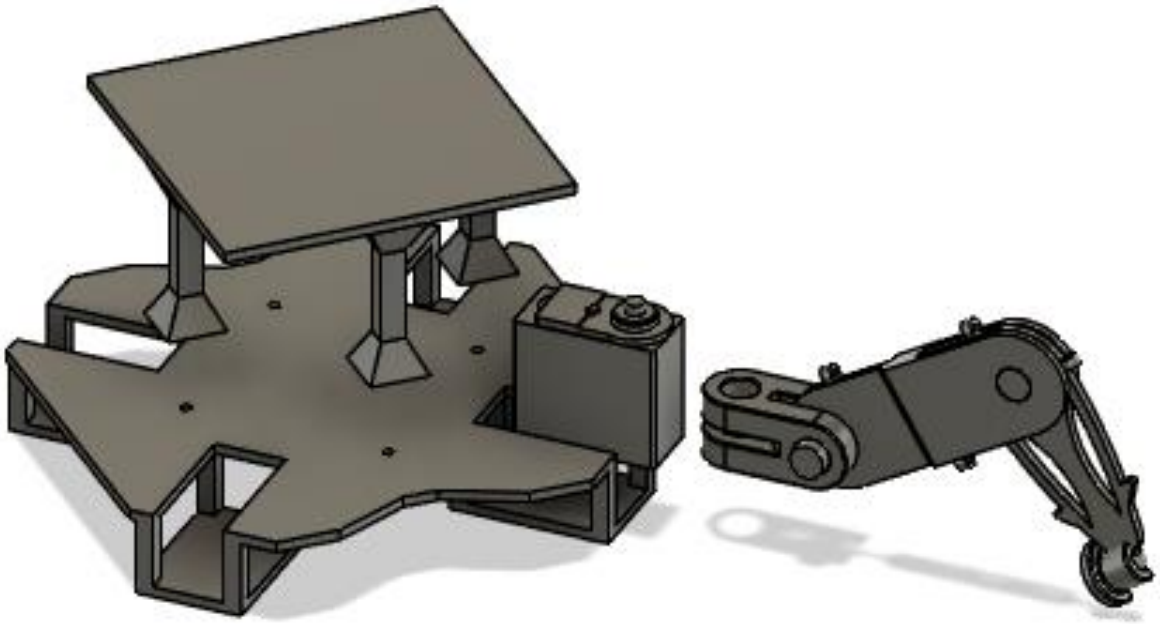


Figure 9: MK2 design

This design was only partially completed and was quickly superseded by the MK3 design. However, it improved upon the MK1 design by reducing the number of legs which both reduced weight and increased the space for the servo horns to rotate. The top panel was a concept for the location of the leg servos. We moved onto the MK3 design before this concept was fully developed.

Testing

This design did not get past the concept phase, no testing was completed.

What we learnt and kept

What we learnt:

1. We decided that it was not worth the time spent to keep trying to store all the servos in one central location. This was due to the complications of achieving the correct mechanical motions required to move the legs.
2. We dropped the idea of using tendons because we had spent too long on them and were no closer to reaching a satisfactory answer on how to implement them.

What we kept:

1. We enlarged the foot design to increase its mechanical strength and ease of manufacture. Other than the increase in scale, the foot design remained unchanged.

MK3 design

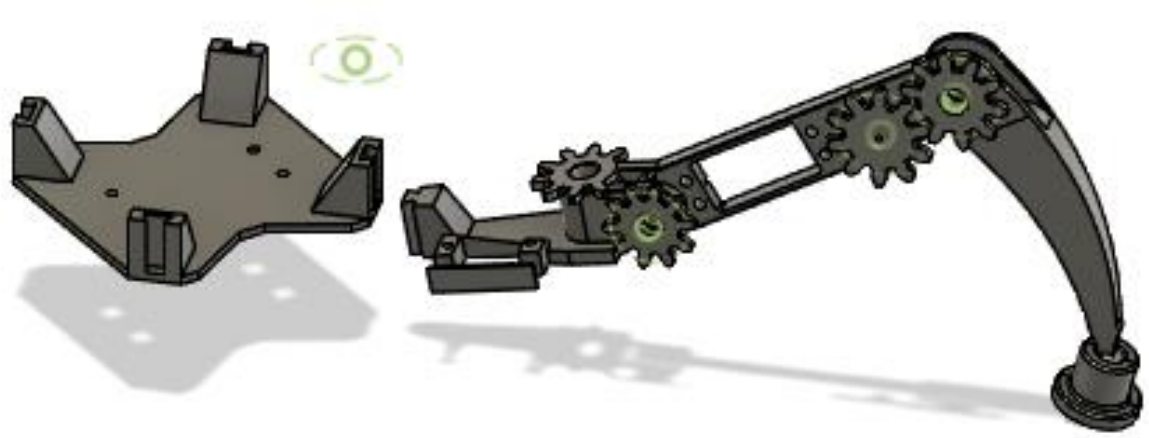


Figure 10: MK3 design

The MK3 design introduces design changes that played a large role in the design of the final MK6 design. It moved the servos solely onto the legs which reduced the size of the body whilst increasing the space that the servos had to operate in. We also switched to a cog design of leg control instead of tendons as we felt this reduced the complexity of the leg.

Testing

We printed and built this robot design. We found that the accuracy of the spur gears was too low and that this, combined with the excessive length of the legs, lead to catastrophic mechanical give. This led to the walker being completely unable to walk when built.

What we learnt and kept

What we learnt:

1. We learnt that using spur gears introduced too much mechanical give, which is ultimately what caused the failure in the robot.
2. The size of the leg could be drastically reduced by moving the servos to different positions within the leg.

What we kept:

1. We decided that embedding the servos into the leg was a good idea and continued with this.

MK4 design

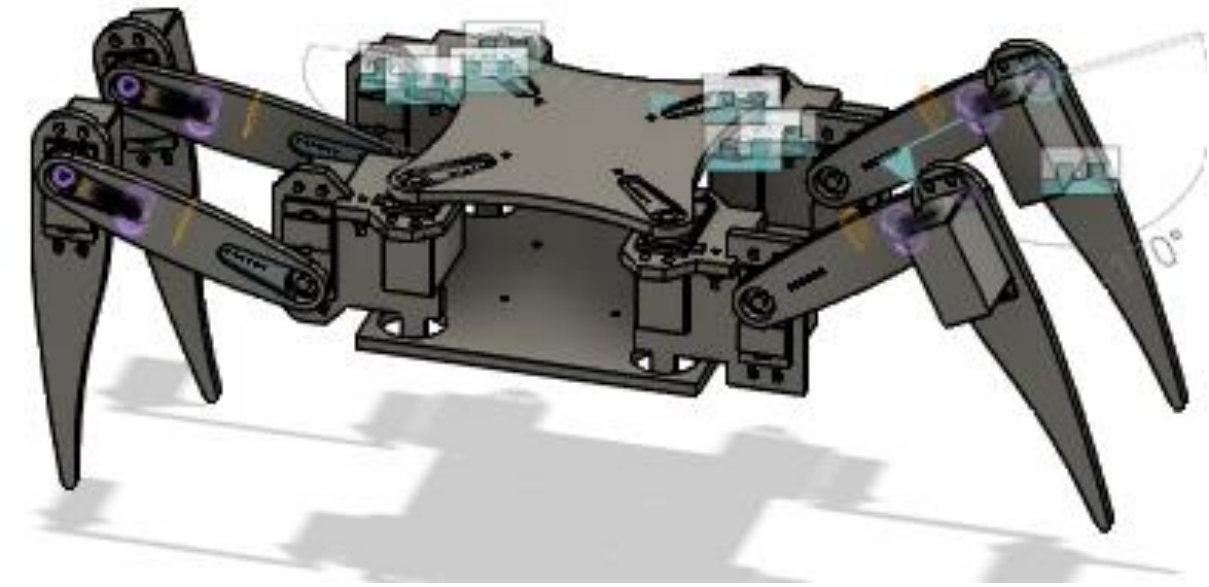


Figure 11: MK4 design

We researched commercial and non-commercial quadruped designs. We discovered the sunfounder crawling quadruped robot kit[13]. We based the design of the shoulder on their robot and found their method for joining the two pieces of the shoulder block together perfect for our application.

After realising the downsides of the MK3 leg design we removed the spur gears entirely. Instead, we fixed the servo horns directly to the 3D printed body. To reduce the overall length of the leg we moved the shoulder servo inside the body, between the top and bottom plates, combined the forearm servo with the body servo into the shoulder block, and moved the wrist servo onto the foot of the robot. This drastically reduced the mechanical give compared to the MK3 design.

Testing

When tested with a basic walking algorithm we found the legs to be too flexible. We discovered this was due to the material of the servo horn flexing when in motion.

What we learnt and kept

What we learnt:

1. We found that the legs had to be extremely sturdy due to the weight of the robot, otherwise unwanted flexing will occur, and this could lead to stress fractures in the components, and hence leading to the overall reduction of component lifetime.
2. Having the servos orientated such that the servo horn was at the top was beneficial as it lowered the centre of gravity of the robot, making it more stable.

What we kept:

1. The shoulder design operated exceptionally well. We decided to go forward with this design.
2. The placement of the servos enabled us to drastically shorten the length of the leg, thus reducing the effect of mechanical give.

MK5 design

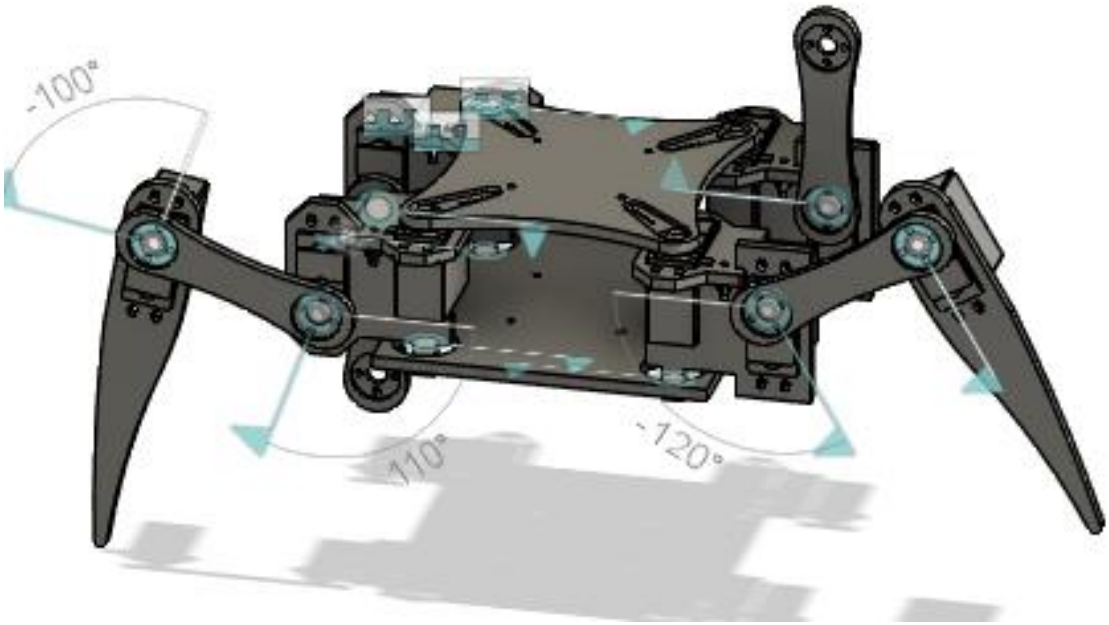


Figure 12: MK5 design

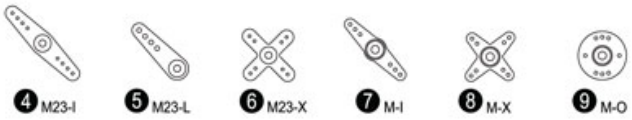


Figure 13: Types of servo horn

The only change this design introduced was to switch from plastic single arm servo horn (number 5 on Figure 13) to metal circular ones (number 9 on Figure 13) that allowed for fastening on both sides of the horn, leading to a more rigid connection.

Testing

This reduced the flexing, but it was not to a satisfactory degree.

What we learnt and kept

What we learnt:

1. The next area of flexing to fix was due to the servo shafts.

What we Kept:

1. We kept the circular metal servo horns were successful in eliminating the flexing introduced by the plastic servo horn.

MK6 - Final Design

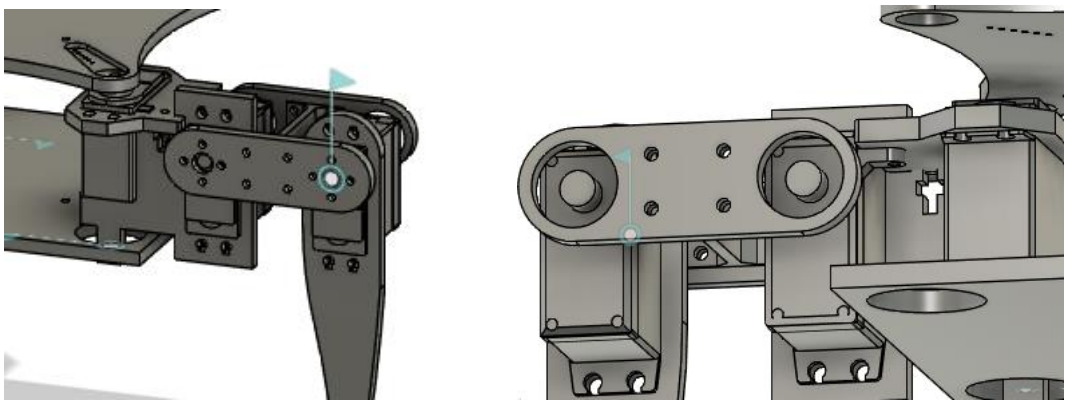


Figure 14: MK6 leg design

In this design we used back braces to eliminate any flexing that was caused by the servo shafts. We found that there was very slight flexing occurring between the two pieces of the shoulder. Custom servo backs were designed and printed to support the back brace as well as add an extra contact point with the shoulder to stop any flexing occurring between the two parts.

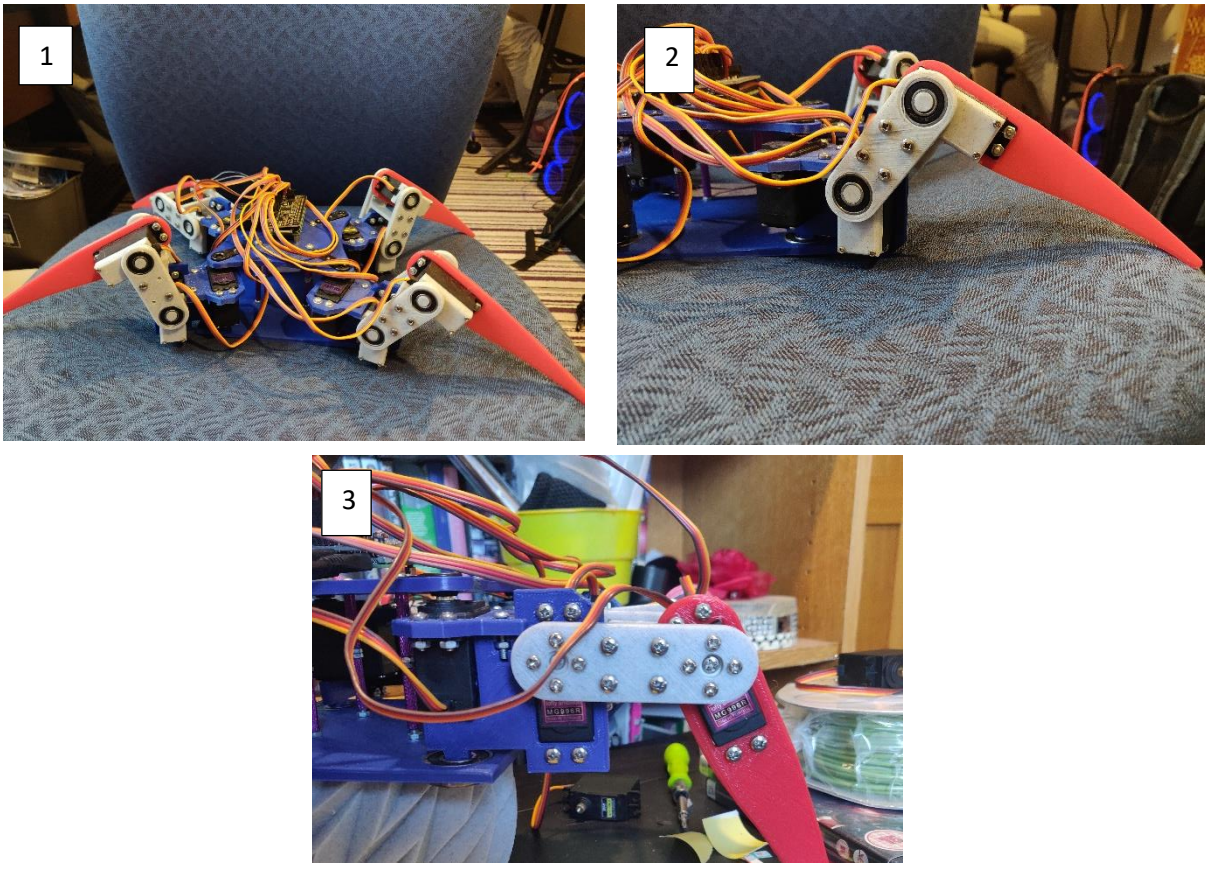


Figure 15: (1) Front full body, (2) Rear leg, (3) Front leg

To fix the issue we faced of the leg flexing, we added more rigidity to our robot by adding a back plate, Figure 17. This solved the issue by adding more support to the servo in the form of a second point of contact. However, we needed a way to fix the back plate to the servos. We decided to design and use a custom back to the servos that included a pin we could fit a bearing over, Figure 22. Using a bearing with an inner diameter of less than 8mm would run the risk of the pin snapping under stress due to the structural limitations of printed Polylactic acid (PLA). We used 1180098 double sealed deep groove ball bearings because they were made from high-quality materials and capable of heavy loads, they also fitted the requirement of being cheap.

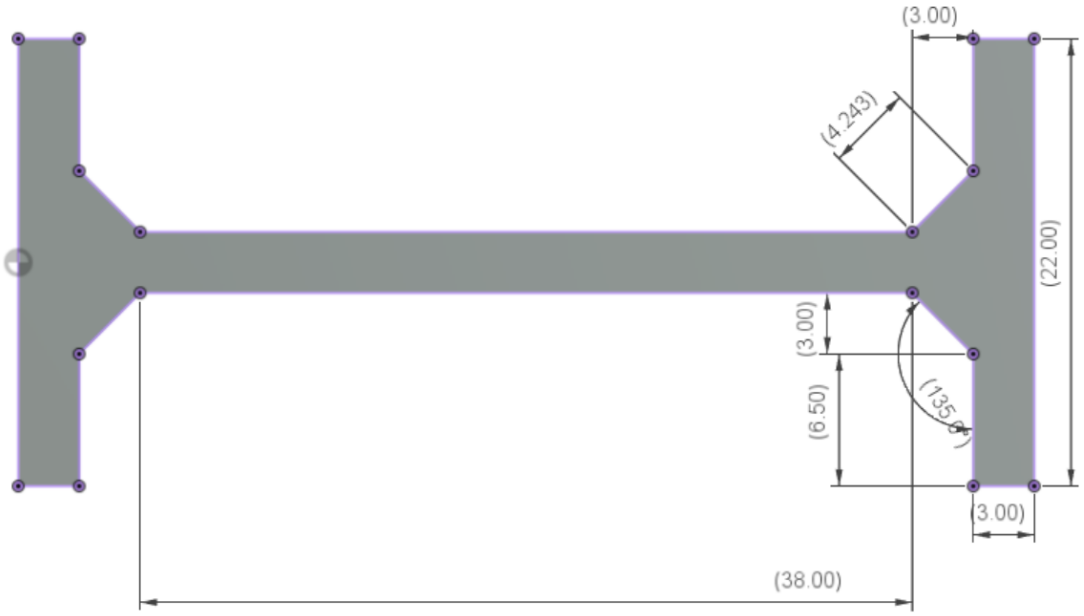


Figure 16: I-Beam to stop twisting

The back plate and fore plate are connected by an I-beam, Figure 16, to eliminate any twisting. The I-beam was specially designed and placed keeping in mind how it will affect the rotation of the wrist and shoulder servos. As it is designed the I-beam does not impede the rotation of the joints in a significant way.

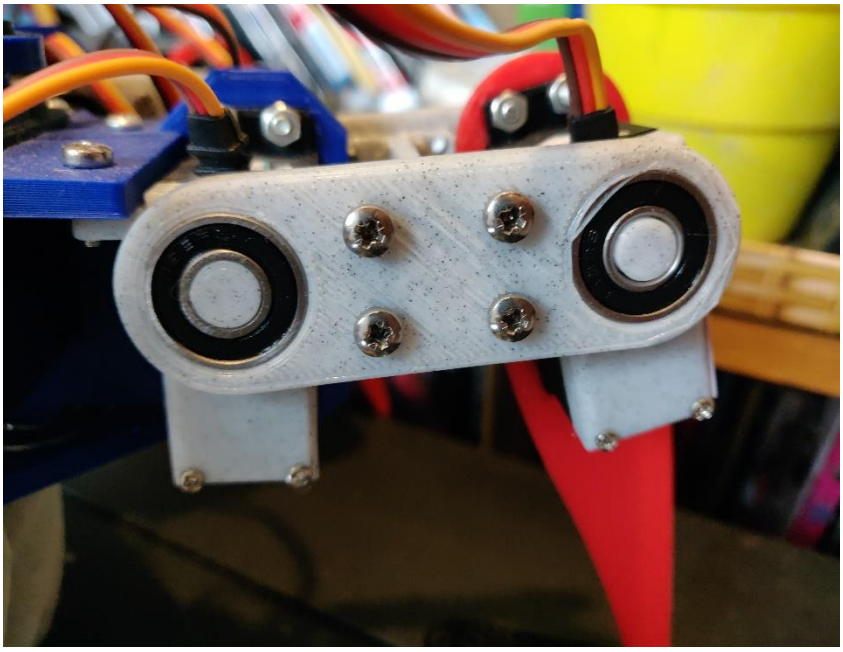


Figure 17: Back plate of shoulder-wrist reinforcement

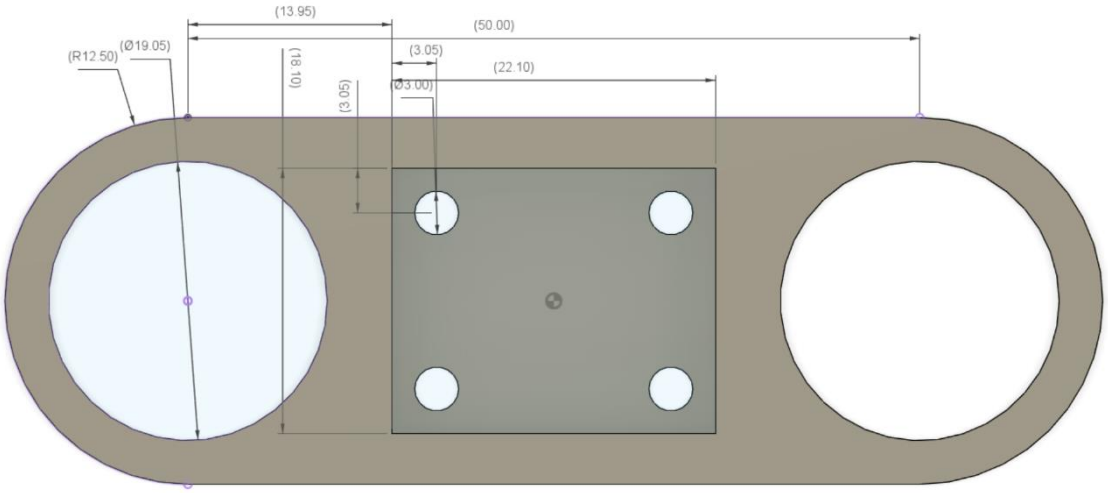


Figure 18: Top-down view of back brace

Figure 18 shows the back brace that has two large holes designed for the bearings to fit in. The large rectangle is a groove for the I-beam to fit in.

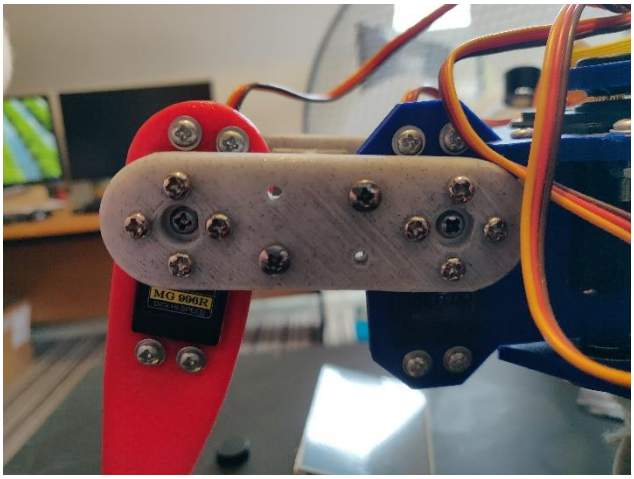


Figure 19: Servo plate of shoulder-wrist joint

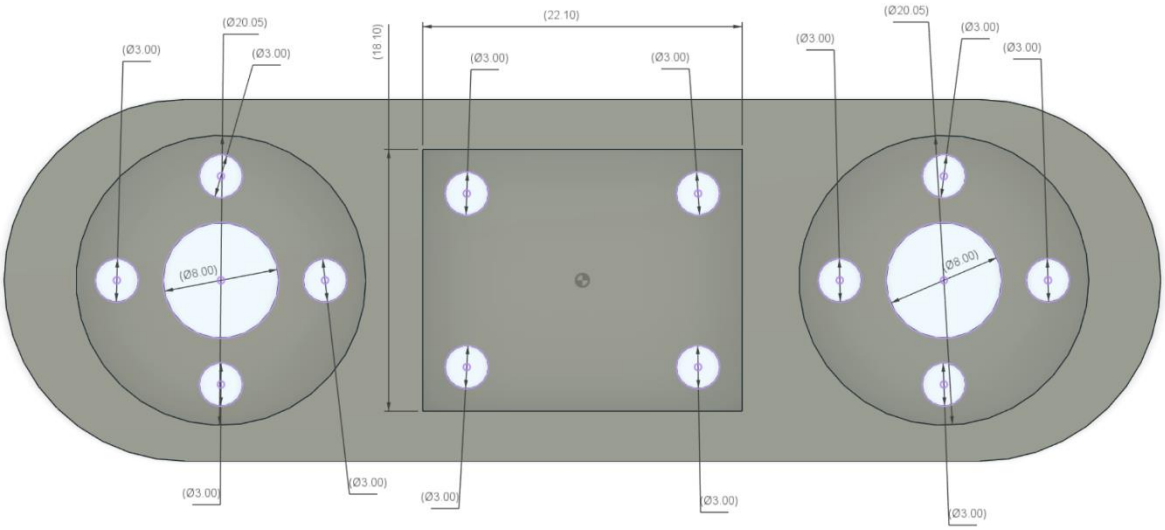


Figure 20: Servo plate of shoulder-wrist join dimensions

Figure 20 shows the servo plate that holds the servo horns in the circular grooves. The rectangular groove is for the I-beam.

We saw a slight amount of flexing occur between the two joints that make up the shoulder, we fixed this by adding an extra fastening point on the shoulder's servo backs that we could screw into through the shoulder plate, Figure 22 (2).

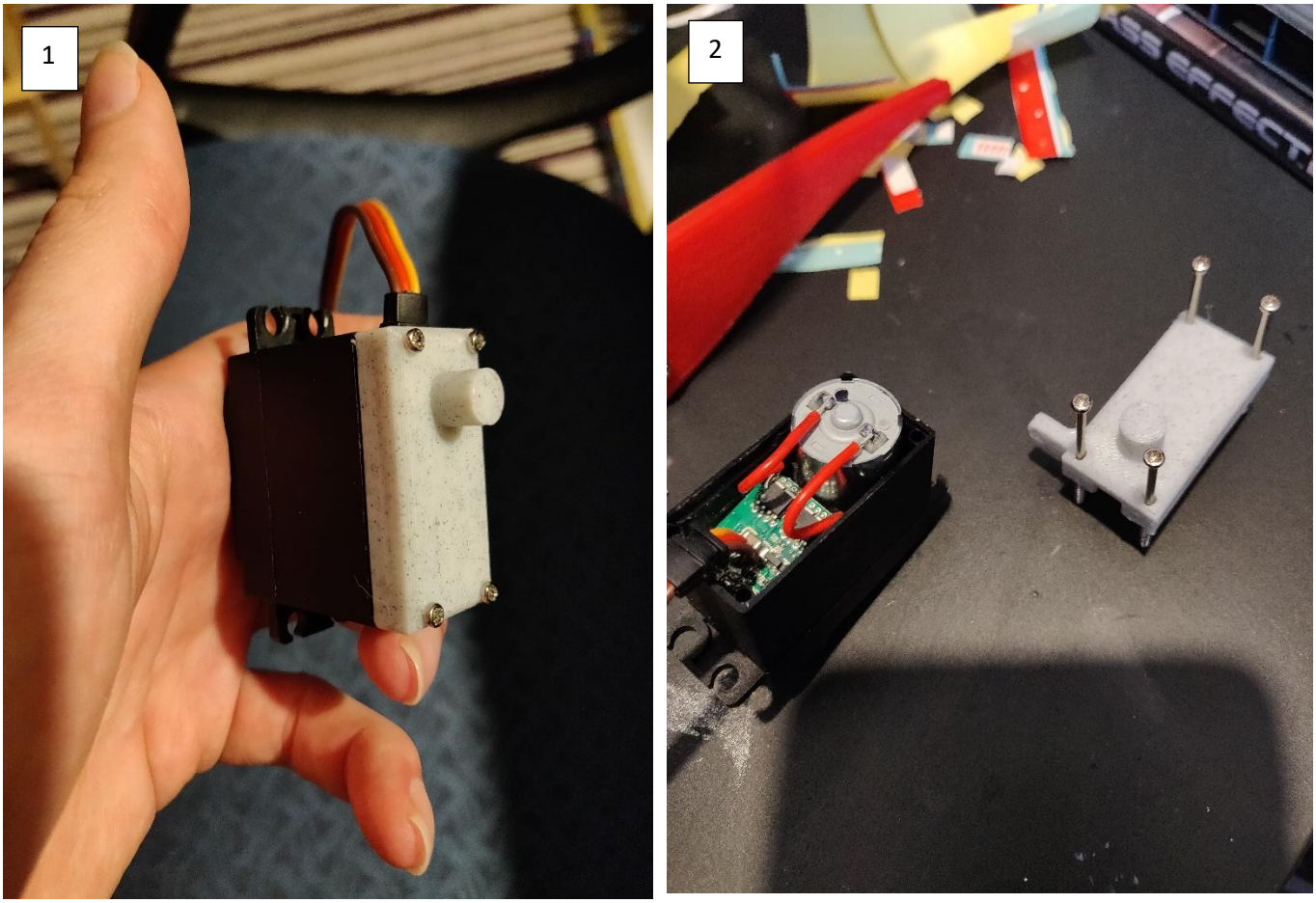


Figure 22: Custom servo back with bearing pin

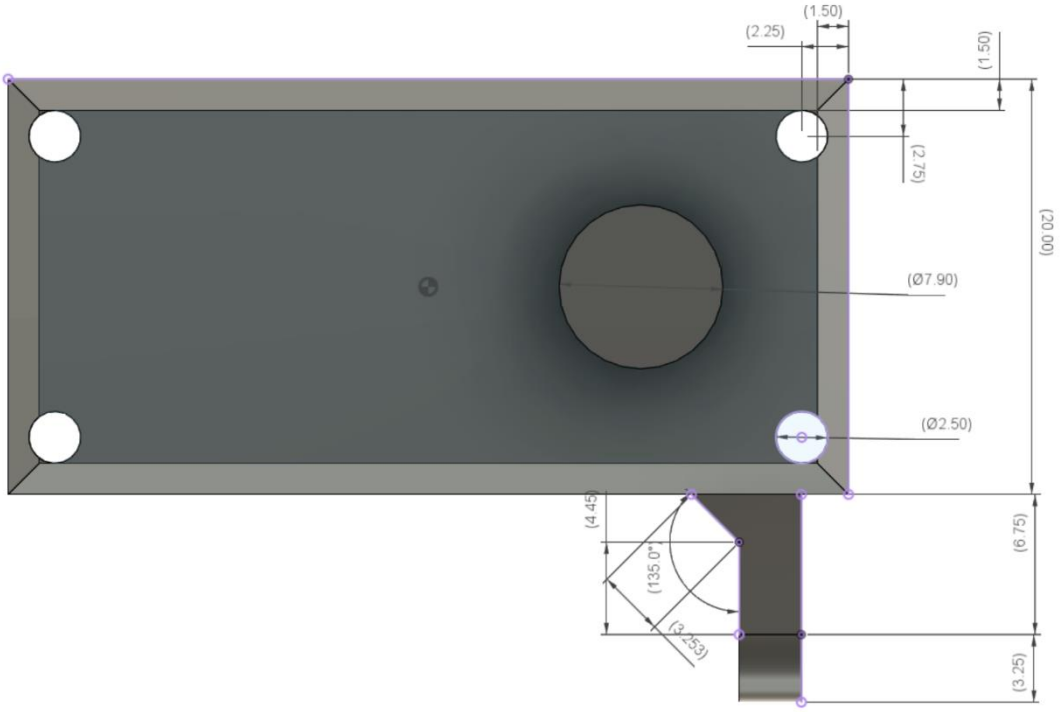


Figure 21: Top-down view of servo back

Programming

Goals of project

The goal of this project is to create a quadrupedal robot that moves forwards, backwards and rotates left and right. It implements object recognition using an Arducam camera and an AI hand recognition and tracking algorithm.

Classes

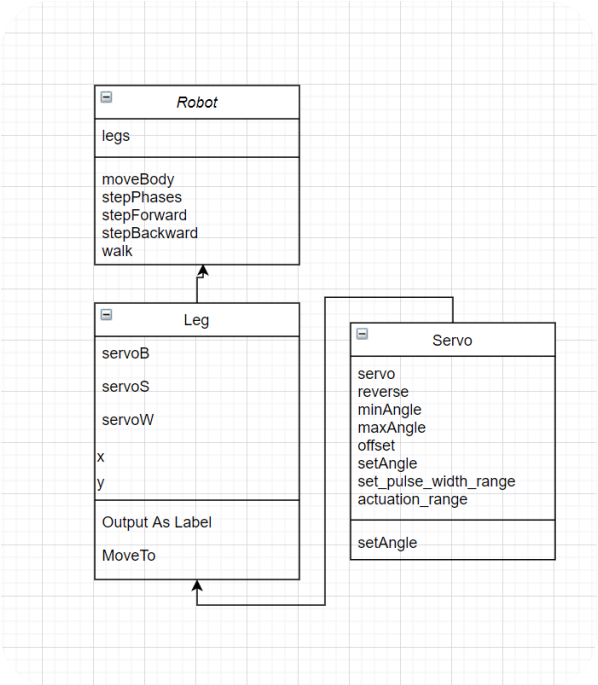


Figure 23: Class diagram

The Python programming language was used to code the robot.

It was decided to use object-oriented architecture because it suited the application immensely. It was possible to abstract each physical element of the robot into a class and represent the interactions between robot components as class methods. The robot is modelled using 3 classes.

Robot class

```
class robot():
    def __init__(self): ···

    def moveBody(self, option, forward): ···


    def stepPhases(self, move): ···
    def stepForward(self): ···

    def stepBackward(self): ···

    def turnRight(self): ···

    def turnLeft(self): ···

    def walk(self): ···
```

Figure 24: Robot class overview

The robot class defines the attributes of the robot, that it has 4 legs, the coordinates it needs to move to and the specific servo limits and offset. It defines the different movement commands used to control the robot.

Leg class

```
class Leg():
    def __init__(self, Servo0,minB,maxB,offsetB, Servo1,minS,maxS,offsetS, Servo2,minW,maxW,offsetW,  reverse):
    def forward(self,angle):

    def lift(self,angle):

    def step(self, angle):

    def moveTo(self,x,y):


    def legLift(self):
```

Figure 25: Leg class overview

The leg class defines what a step is and uses inverse kinematics to calculate the angles of each servo from an input coordinate.

Servo class

```
class Servo():
    def __init__(self, servo, reverse,min,max,offset):


    def setAngle(self, angle):
```

Figure 26: Servo class overview

The servo class defines how to control the servos. The code to write to the servo is dependent on the servo controller.

Program logic

```
while keepGoing:

#   myRobot.walk()
    key= getChar()
    if key == "x":
        keepGoing = False
        break
    else:
        if (key == 'w'):
            myRobot.stepForward()
        elif (key == 's'):
            myRobot.stepBackward()
        elif (key == 'a'):
            myRobot.turnLeft()
        elif (key == 'd'):
            myRobot.turnRight()
```

Figure 27: Main

The abstract nature of the code allows for easier control of the robot because we can call the **stepForward** function from the robot class without having to worry about ensuring the servos rotate correctly and the legs move into the right positions to execute a walk. The robot class knows it has

four legs, but it does not know what makes up a 'leg', it just tells the leg to move to a specific coordinate. Abstraction will help with future proofing the code because if we wish to change a feature about the leg, such as the number of servos, or the type of servos, we are easily able to change a singular class. It enables us to add extra sensors in separate classes and easily integrate them into the main program. This feature lends itself to the modular design we desire.

Phases of the gait

```
#
#####################################################
#    FORWARDS
#####################################################
Fphase1 = [[70,70,True],[70,70, True],[70,1, True],[1,70, True]]        #[X, Y, Is leg lifted?]
Fphase2 = [[70,70, True],[70,70, True],[70,110, True],[1,70, True]]
Fphase3 = [[70,1,False], [110,70, False], [70,70, False], [70,70, False]]
Fphase4 = [[70,1, True], [1,70, True], [70,70, True], [70,70, True]]
Fphase5 = [[70 ,110, True], [1,70, True], [70,70, True], [70,70, True]]
Fphase6 = [[70,70, False], [70,70, False], [70,1, False], [110,70, False]]

forwardStep = [Fphase1, Fphase2, Fphase3, Fphase4, Fphase5, Fphase6]

#####################################################
#    BACKWARDS
#####################################################
Bphase1 = [[70,70,True],[70,70, True],[70,1, True],[110,70, True]]      #[X, Y, Is leg lifted?]
Bphase2 = [[70,110, False],[1,70, False],[70,70, False],[70,70, False]]
Bphase3 = [[70,1,True], [1,70, True], [70,70, True], [70,70, True]]
Bphase4 = [[70,1, True], [110,70, True], [70,70, True], [70,70, True]]
Bphase5 = [[70 ,70, False], [70,70, False], [70,110, False], [1,70, False]]
Bphase6 = [[70,70, True], [70,70, True], [70,1, True], [1,70, True]]

backwardStep = [Bphase1, Bphase2, Bphase3, Bphase4, Bphase5, Bphase6]

#####################################################
#    ANTICLOCKWISE
#####################################################
ACphase0 = [[70,70,True],[70,70, True],[70,70, True],[70,70, True]]
ACphase1 = [[70,70, True], [1, 70, True], [70, 1, True], [70, 70, True]]
ACphase2 = [[70, 1, False], [70, 70, False], [70, 70, False], [1, 70, False]]

antiClockwiseStep = [ACphase0, ACphase1, ACphase2]

#####################################################
#    CLOCKWISE
#####################################################
Cphase0 = [[70,70,True],[70,70, True],[70,70, True],[70,70, True]]
Cphase1 = [[70, 1, True], [70, 70, True], [70, 70, True], [1, 70, True]]
Cphase2 = [[70,70, False], [1, 70, False], [70, 1, False], [70, 70, False]]

ClockwiseStep = [Cphase0, Cphase1, Cphase2]
```

Figure 28: Phases of the gait

We decided the best method for coding the gait would be to write a list of phases. There are 4 elements which correspond to each leg in each phase. The phases are a list that contains the x, y coordinates for the end effect and whether the leg is lifting or not.

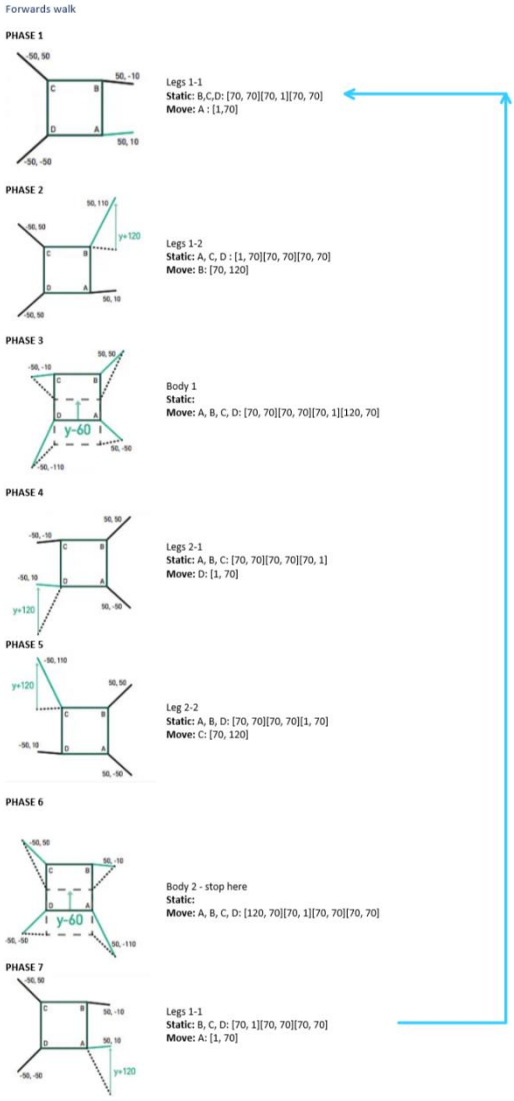


Figure 30: Forward gait

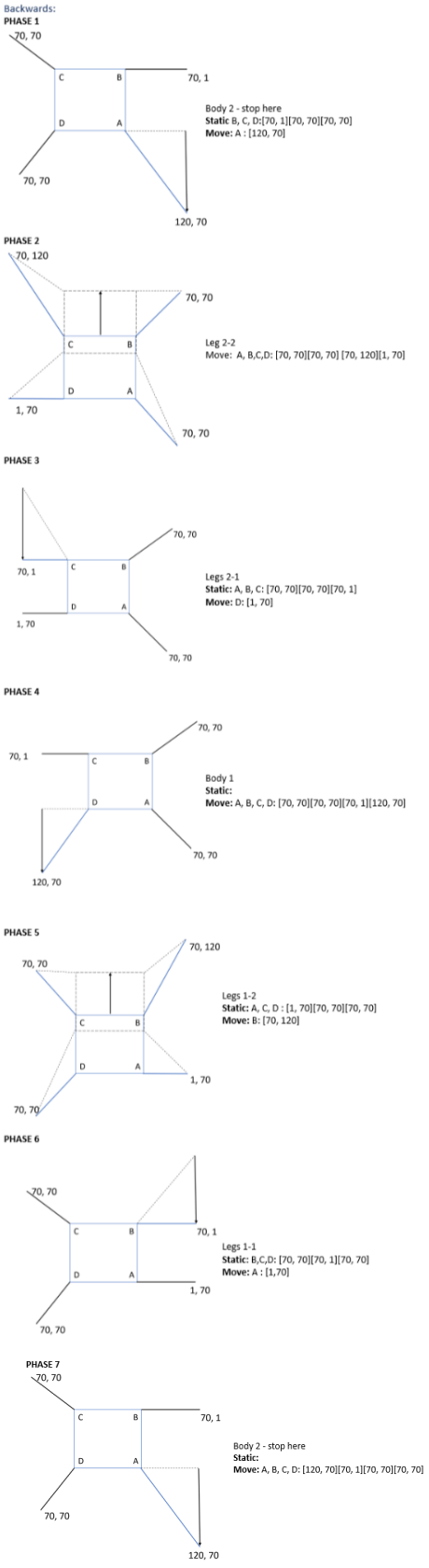


Figure 30: Backward gait

The coordinates were defined with reference to the shoulder, each leg has its own reference system. This coordinate system simplified the understanding of the gait because it was position based, if two legs were in the same position, they would have the same coordinate.

The leg class uses inverse kinematics to convert these coordinates into angles for the servos, it lifts the leg if it receives a True. None of the coordinates are 0, this is because when passed into the **moveTo** function the $\tan^{-1}(x/y)$ calculation returns a divide by zero error. We decided to replace the 0's with 1's because they are near enough to be the same position.

Robot class

```
class robot():
    def __init__(self):
        NUM_SERVOS=16 #define the number of servos

        #self.kit = ServoKit(channels=NUM_SERVOS) #initialise the servos

        self.legs=[]
        #           servo, min, max, offset, servo, min, max, ooffset...
        #       body, shoulder, wrist
        #leg 1, pin 7 for shoulder
        self.legs.append( Leg(12,90,180,0,13,60,180,-4,14,0,150,0,True))        #Front left leg
        #Leg 2
        self.legs.append( Leg(28,90,180,-2,29,55,180,0,30,0,130,0,True))        #Back left leg
        #Leg 3, is labelled 13, 14, 15 on robot
        self.legs.append( Leg(0,0,90,0,1,0,120,10,2,10,180,-15,False))      #Front right leg
        #Leg 4
        self.legs.append( Leg(16,0,90,5,17,0,110,-10,18,10,180,0,False))        #Back right leg
```

Figure 31: Constructor function

The constructor function initialises the robot by creating a list of legs. The legs are created by instantiating a leg object and passing it the servo number, minimum angle and maximum angle, the offset, and whether the leg needs to be reversed or not.

Using this architecture, it is possible to add extra legs inside the constructor function by adding another line with the corresponding pin number, min, and max angle for each servo in the leg. The phases in Figure 28**Error! Reference source not found.** will need editing to accommodate for the e xtra legs. The servo class, and leg class will remain unchanged as they are not affected by the number of legs, instead they solely deal with their specific roles.

```
def stepPhases(self, move):
        #This will be used to move the legs to the next position.
        #The key difference between this and the moveBody function is that in this function I don't need to worry about the
        #servos moving in a straight line and they will be raised and move one at a time.

        for j in range(len(move)):
            for i in range(4):
                #if x coordinate is different AND y coordinate is different.. check on y =0 though?
                if self.legs[i].x != move[j][i][0] or self.legs[i].y != move[j][i][1]:
                    if move[j][i][2]:
                        self.legs[i].legLift()
                    self.legs[i].moveTo(move[j][i][0], move[j][i][1])
            time.sleep(0.1) #sleep between step 1 and 2 of the list
```

Figure 32: stepPhases function

The **stepPhases** function cycles through the selected phase as written in Figure 28. These phases are the different leg positions that make up the gait, only one leg moves at a time so that there is always 3 points of contact with the ground at any time. When one leg moves, the other three stay still. It was decided that it would be easier to command all the legs at once instead of trying to command

one leg at a time. We implemented a check to see if the new coordinate was different from the previous one, if it was the leg would perform a step, if the coordinate was the same then it would remain stationary. This method worked well and helped with understanding and debugging the gait.

```
def stepForward(self):
    self.stepPhases(forwardStep)
    time.sleep(0.5)

def stepBackward(self):
    self.stepPhases(backwardStep)
    time.sleep(0.5)

def turnRight(self):
    self.stepPhases(antiClockwiseStep)
    time.sleep(0.5)

def turnLeft(self):
    self.stepPhases(ClockwiseStep)
    time.sleep(0.5)
```

Figure 33: Step functions

These sets of functions call their corresponding gait phase list. We decided to create these functions to increase the readability of the code. This abstraction means that the main body of the code is very simple, see Figure 27.

Leg class

```
def __init__(self, Servo0,minB,maxB,offsetB, Servo1,minS,maxS,offsetS, Servo2,minW,maxW,offsetW,  reverse):
    self.servoB = Servo(Servo0, reverse, minB,maxB,offsetB)
    self.servoS = Servo(Servo1, reverse, minS,maxS,offsetS)
    self.servoW = Servo(Servo2, reverse, minW,maxW,offsetW)

    self.x = 0
    self.y = 0
    time.sleep(0.1)
```

Figure 34: Leg constructor

The constructor initialises the legs by creating the body, shoulder, and wrist servos. The servos are created by assigning each of the three servos their perspective pin number, whether they are reversed or not, minimum, and maximum angle as well as their offset. There are only three servos shown in Figure 34 whereas there are 12 servos in total. This is because of the abstract nature of the code, each leg in instantiated as separate objects, as such they can re-use the same class's code for each one.

```
def moveTo(self,x,y):

    if (y == 0):
        y = 1

    #Calculate body_servo
    body_servo= degrees(atan(x/y))

    #Calculate Hypotenuse
    H = sqrt(pow(YOFFSET, 2) + pow((y - SHOULDER), 2))

    #Calculate SS2 shoulder_servo2
    SS2 = degrees(acos(YOFFSET/H))

    #Calculate SS1 shoulder_servo1
    SS1 = degrees(acos((pow(HAND, 2) - pow(H,2) - pow(FOREARM, 2))/(-2*H*FOREARM)))

    #Calculate shoulder_servo
    shoulder_servo = SS2 +SS1

    #Calculate wrist_servo
    wrist_servo = degrees(acos((pow(H, 2) - pow(HAND, 2) - pow(FOREARM, 2))/ (-2 * HAND * FOREARM)))

    #Move
    self.x=x
    self.y=y
    self.servoB.setAngle(body_servo)    #Rotates leg
    self.servoW.setAngle(wrist_servo)   #Rotates wrist
    time.sleep(0.1)             #Allows body servo to move leg into correct position
    self.servoS.setAngle(shoulder_servo)    ##puts foot down
    time.sleep(0.1)
```

Figure 35: moveTo function

The **moveTo** function implements trigonometric inverse kinematics to calculate the angle for each servo using a given target coordinate. The order in which a step is performed is: lift leg with shoulder servo (Figure 36), move leg forward with body servo, move wrist with wrist servo, wait 0.1 seconds so that the body and wrist servo have time to move into place, put foot down.

```
def legLift(self):
    self.servoS.setAngle(45)
    time.sleep(0.1)
```

Figure 36: legLift function

The **legLift** function is separated from the **moveTo** function because it needs to be called separately in Figure 32, the **stepPhases** function. This is because we use the **stepPhases** function for both the leg movement and the body movement.

Inverse kinematics is the maths of calculating the joint angles of a system to place the end of the kinematic chain, such as a robot foot, at a specified coordinate location.

We chose to use trigonometric Inverse kinematics over matrix based inverse kinematics because there was no need to use matrix based inverse kinematics when dealing with so few DoF (Degrees of Freedom) and trigonometric inverse kinematics is simpler to understand, therefore easier to debug. Figure 35 shows the code for this process.

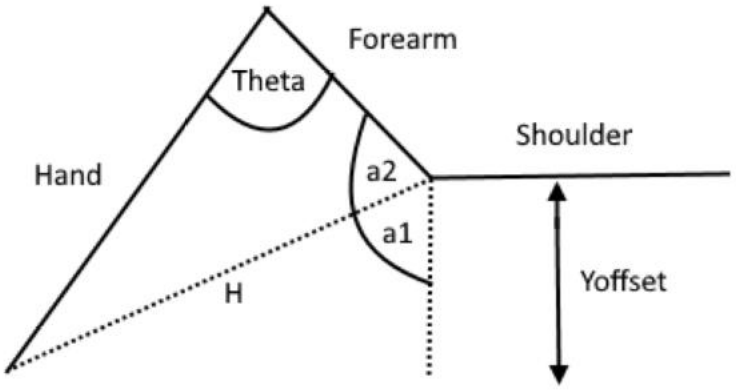


Figure 37: Trigonometric diagram of robot leg

The maths behind the calculations is:

$$H = \sqrt{Yoffset^2 + (y - shoulder)^2}$$

$$a1\ \cos^{-1}\left(\frac{Yoffset}{H}\right)$$

$$a1 = \cos^{-1}\left(\frac{Hand^2 - H^2 - forearm^2}{-2 * hand * forearm}\right)$$

$$Theta = \cos^{-1}\left(\frac{H^2 - Hand^2 - forearm^2}{-2 * hand * forearm}\right)$$

Equation 1: Shoulder and wrist inverse kinematics

The wrist servo angle is theta, and the shoulder servo angle is the sum of a1 and a2.

The body servo calculated as such:

$$body = \tan^{-1}\left(\frac{x}{y}\right)$$

Equation 2

These functions allow for an input of a coordinate to be given to the **moveTo** function and it will calculate the angels for each servo.

Servo class

```
def __init__(self, servo, reverse,min,max,offset):
    self.servo=servo
    self.reverse = reverse
    self.minAngle=min + offset
    self.maxAngle=max + offset
    self.offset=offset
    self.setAngle(90)
```

Figure 38: Servo constructor

The constructor initialises the servos by creating the individual components of the servo. The reverse variable relates to which side of the robot the servo is. If the servo is on the left, then the angle of the servo is calculated as 180 – desired angle. This is because the servo is flipped on the left-hand side with respect to the right-hand side of the robot.

The minimum and maximum angles have an offset added to them because the servo has 24 teeth which caused a few degrees of error to be apparent in the leg, see Figure 49.

The offset was found by setting every servo to 90 degrees before construction and then assembling the robot into a chosen home position. An algorithm that allowed control of a singular servo was written and used to make small changes to the angle of the servo until they were all in a desirable state.

```
def setAngle(self, angle):
    sp = serial.Serial('/dev/ttyUSB0', 115200)
    if self.reverse:    #Only the left legs are reversed
        angle = (180 - angle) + self.offset
    else:
        angle = angle + self.offset
    if angle >= self.minAngle and angle <= self.maxAngle:       #Checks if the angle is in range
        #self.servo.angle = angle               #Sets the servo angle
        servoString= f'#{self.servo} P{angleToPWM(angle)}\r'
        sp.write(servoString.encode())
        print(servoString)
        self.angle = angle                  #Allows me to read the angle the servo is being set
    else:
        print("Servo ", self.servo, " moved too far: ", angle)
```

Figure 39: setAngle function

The **setAngle** function checks if the inputted angle is reversed and that it fits within the range of motion for that specific servo. It then uses the specific syntax of the servo controller to write to the servo.

The Lynxmotion SSC-U32 servo controller required the syntax to be as shown in Figure 40.

 # <ch> P <pw> *S* <*spd*> *T* <time> <cr>

- <ch>: pin / channel to which the servo is connected (0 to 31) in decimal
- <pw>: desired pulse width (normally 500 to 2500) in microseconds
- <*spd*>: servo movement speed in microseconds per second*
- <*time*>: time in microseconds to travel from the current position to the desired position. This affects all servos (65535 max) *
- <cr>: carriage return (ASCII 13)**

Example 1: **#5P1500S750<cr>**

Figure 40: Lynxmotion servo control syntax

This function is a great example of the value of abstraction. The Adafruit bonnet required different syntax to control the servos. When we switched over to the Lynxmotion servo controller we only had to change this one function and the rest of the code remained the same whilst working with the new board.

Camera code

There are two approaches to the writing the camera code. Our first attempt was to try using the FLASK library to implement a lightweight webserver on the Pi, this could then stream the output from the camera. However, whilst this worked for individual images it failed on the Pi when trying to stream images.

We then triad a lower-level solution. For this solution we used the python socketserver, threading and HTTP libraries to create a HTTP webservice and bind it to a TCP/IP port, in this case we used port 8000.

```
with picamera.PiCamera(resolution='640x480', framerate=24) as camera:
    output = StreamingOutput()
    #Uncomment the next line to change your Pi's Camera rotation (in degrees)
    #camera.rotation = 90
    camera.start_recording(output, format='mjpeg')
    try:
        address = ('', 8000)
        server = StreamingServer(address, StreamingHandler)
        server.serve_forever()
    finally:
        camera.stop_recording()
```

Figure 41: Initialises the camera code

This initialises the camera, defines the size of the image captured as 640x480 at 24 frames per second (FPS). We create an output object that uses an in-memory byte stream to store that captured data. The camera recording is then started, and we attempt to start the webserver. If the webserver fails to start, we close the camera and exit the program.

The webserver is set up to handle three different GET requests,'/', '/index', '/stream', any other request returns a 404 page not found error.

```
class StreamingHandler(server.BaseHTTPRequestHandler):
    def do_GET(self):
        if self.path == '/':
            self.send_response(301)
            self.send_header('Location', '/index.html')
            self.end_headers()
        elif self.path == '/index.html':
            content = PAGE.encode('utf-8')
            self.send_response(200)
            self.send_header('Content-Type', 'text/html')
            self.send_header('Content-Length', len(content))
            self.end_headers()
            self.wfile.write(content)
        elif self.path == '/stream.mjpg':
            self.send_response(200)
            self.send_header('Age', 0)
            self.send_header('Cache-Control', 'no-cache, private')
            self.send_header('Pragma', 'no-cache')
            self.send_header('Content-Type', 'multipart/x-mixed-replace; boundary=FRAME')
            self.end_headers()
            try:
                while True:
                    with output.condition:
                        output.condition.wait()
                        frame = output.frame
                    self.wfile.write(b'--FRAME\r\n')
                    self.send_header('Content-Type', 'image/jpeg')
                    self.send_header('Content-Length', len(frame))
                    self.end_headers()
                    self.wfile.write(frame)
                    self.wfile.write(b'\r\n')
            except Exception as e:
                logging.warning(
                    'Removed streaming client %s: %s',
                    self.client_address, str(e))
        else:
            self.send_error(404)
            self.end_headers()
```

Figure 43: Streaming handler class

The supported page requests are:

/

This returns the HTTP code 301 which indicates a permanent redirect to /index.html

/index.html

The returns the HTML code defined in Figure 44.

```
PAGE="""\
<html>
<head>
<title>Raspberry Pi - Surveillance Camera</title>
</head>
<body>
<center><h1>Raspberry Pi - Surveillance Camera</h1></center>
<center><img src="stream.mjpg" width="640" height="480"></center>
</body>
</html>
"""
```

Figure 44: /index.html

/stream.mjpeg

This returns the stream of the data from the camera. This can be requested as part of a webpage as defined above or directly in a remote OpenCV application for image processing.

Implementing support for extra requests would extend the remote functionality of the robot, for example adding support for “/forward” could invoke the robot’s command to step forward.

Any code that opens network ports on a computer brings security considerations. In this case only minimal security has been implemented. There is minimal network support and further access to the system is performed through SSH services. However, in a production environment featuring a more feature rich remote interface a more robust security arrangement would be recommended.

Robot features

Remote viewing

The camera and web interface code are run in their own thread, allowing the robot to continue to operate and move whilst simultaneously presenting live camera feed.

Remote movement control

It moves forwards, backwards, clockwise, and anticlockwise when a keyboard input is given. We have decided to not program it to move the body such that the centre of mass is between the three legs remaining on the ground because it is not necessary for a robot this small. The servo motors are strong and fast enough to move into the correct position before the robot has a chance to fall over.

A video of the gait can be seen here on my YouTube channel:

https://youtu.be/ME_nIa-5qeI



During testing we found that remote control was essential, which is what the raspberry Pi 0 W provided. When we calibrated each servo and found their min and max position, such that they would not crash into the body, we used the remote control to set the angles. This was something that would have been extremely difficult to do if we had to upload new code each time we wanted to move a servo. That could lead to broken servos when they try to move into impossible positions and burn themselves out.

Testing

Calibration

To accurately and confidently control the servos, there needed to be a 'home' position set. All the servos will share this position so that if we tell a servo to move to a specific angle it will move to the same position as all the other servos of the same joint on the other legs.

Using the servo controller program in Figure 45 we collected servo data and complied it into Table 2.

Leg number	Servo number	Minimum	Maximum	Offset
1	12	90	180	0
1	13	60	180	-4
1	14	0	150	0
2	28	90	180	-2
2	29	55	180	0
2	30	0	130	0
3	0	0	90	0
3	1	0	120	10
3	2	10	180	-15
4	16	0	90	5
4	17	0	110	-10
4	18	10	180	0

Table 2: Servo position data

```
sp = serial.Serial('/dev/ttyUSB0', 115200)
#sp.write('#0 P1500\r'.encode())

def angleToPWM(angle):
    return int(((2000/210) * angle + 500))

servos=[0,1,2,12,13,14,16,17,18,28,29,30]
angle=90
for i in range (len(servos)):
    servoString= f'#{servos[i]} P{angleToPWM(angle)}\r'
    print(servoString)
    sp.write(servoString.encode())

while True:
    servoNum=int(input("Servo num: "))
    inAngle= int(input("Enter angle: "))
    #### next two lines do the work
    servoString= f'#{servoNum} P{angleToPWM(inAngle)}\r'
    sp.write(servoString.encode())

    print(servoString)
```

Figure 45: Servo controller program

We concluded that the best angle to set as the home angle was 90° because it didn't matter which orientation the servo was in; this would always be the middle angle. Having the home point as the middle of the servo range simplified the servo control in Figure 39. The program devised to set the servos to 90° was very simple because it had one role to do.

```
for i in range(NUM_SERVOS):
    kit.servo[i].angle=90
    time.sleep(0.1)
```

Figure 46: Sets the servo's angle to 90

Servo limits

Each servo has a physical limit where it will collide with the body, Figure 47. For all the joints these limits are outside of the scope of motion the servos will undergo through a walk cycle. We included a min and max variable in our servo class to prevent out of scope angles from occurring.

If there is a mistake in the coordinates of a new gait that would cause the **moveTo** function to output angles that are out of the scope of movement for each servo, this feature will ensure that the servos are not going to burn themselves out trying to move somewhere that they cannot reach.

```
if angle >= self.minAngle and angle <= self.maxAngle:       #Checks if the angle is in range
```

Figure 48: Min and max check

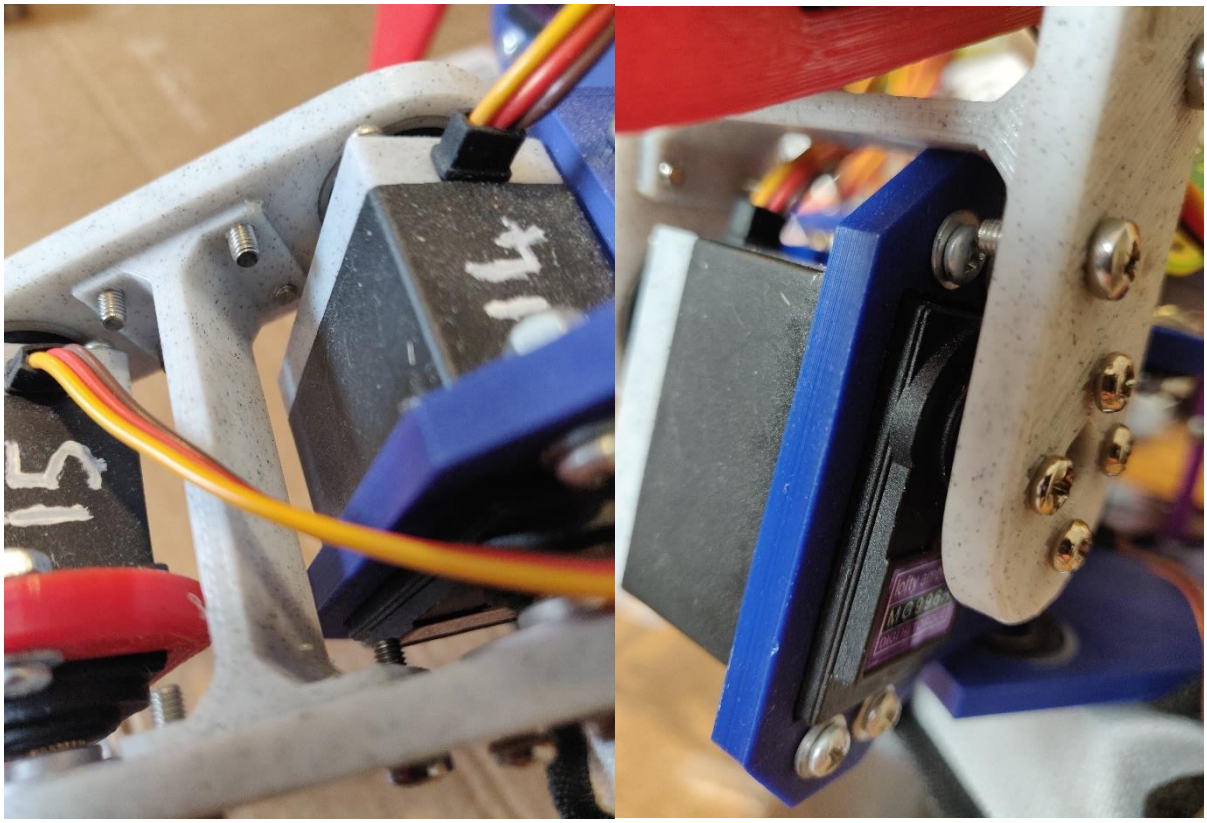


Figure 47: Physical limits of shoulder joint

We found the minimum and maximum of each servo by using a servo control algorithm that granted us control of individual servos. We increased the inputted angle until the body pieces were close to touching together then recorded the angles into Table 2.

Offset

The offset is an unavoidable issue related to the physical limitations brought upon by the servos having 25 teeth and therefore 25 different positions in which I can connect the servos to the servo horns. We found during testing the 'set to 90' program there was a slight difference in the positions of the servos. These could not be fixed by removing the servo horn and rotating it to place the servo in the right alignment. Figure 49 shows the results of the 'set to 90' program, the spirit level shows imbalance in the two legs.



Figure 49: Measuring the offset between two legs

We found that the offsets changed with the new board and would occasionally change after being turned off. Further research will need to be performed to understand and fix this issue.

Gait

We programmed the gait as a series of lists. We found that this provided an easy method to cycle through a series of poses that we predefined as coordinates. The '**True**'s and '**Falses**'s represent whether this phase is one where the legs that move needs to be lifted off the ground or not. If it is **True,** the leg needs to be lifted off the ground, if it's a **False** then it doesn't and it's a body move phase. These phase lists are combined into one list of lists, **forwardStep**. We decided upon this manner of writing the code because it is the best for modifying. If there is a step that we wish to change, or we wish to add another phase to the step, we modify or add a line in this list.

```
#####################################################
#   FORWARDS
#####################################################
Fphase1 = [[70,70,True],[70,70, True],[70,1, True],[1,70, True]]        #[X, Y, Is leg lifted?]
Fphase2 = [[70,70, True],[70,70, True],[70,110, True],[1,70, True]]
Fphase3 = [[70,1,False], [110,70, False], [70,70, False], [70,70, False]]
Fphase4 = [[70,1, True], [1,70, True], [70,70, True], [70,70, True]]
Fphase5 = [[70 ,110, True], [1,70, True], [70,70, True], [70,70, True]]
Fphase6 = [[70,70, False], [70,70, False], [70,1, False], [110,70, False]]

forwardStep = [Fphase1, Fphase2, Fphase3, Fphase4, Fphase5, Fphase6]
```

Figure 50: Forwards step list of phases

During testing of the turning gait, we found that the robot would fall over when transitioning from one gait to another. This was because there was a point between the two gaits where the next step in the gait was not enough to catch the robot and stop its fall. To fix this we added an extra step

```
#########################################################
#   ANTICLOCKWISE
#########################################################
ACphase0 = [[70,70,True],[70,70, True],[70,70, True],[70,70, True]]
ACphase1 = [[70,70, True], [1, 70, True], [70, 1, True], [70, 70, True]]
ACphase2 = [[70, 1, False], [70, 70, False], [70, 70, False], [1, 70, False]]

antiClockwiseStep = [ACphase0, ACphase1, ACphase2]
```

Figure 51: Anticlockwise list of phases

before the rest of the turn that acted as a halfway point and prevented the robot from falling over when it turned. This fix was made easy by writing the code in such a way that I could just add a new phase and the rest would sort itself out.

Future Work

There are a multitude of improvements that could be implemented on my robot. Here we will outline a few of the key ones that we identified.

Squad robots

Ahmed Reza Cheraghi et al. states that the term 'swarm' robotics was first applied in 1988 by G. Beni[17] and Fukuda[18]. G. Beni defines a swarm as system composed of autonomous robots that operates in a n-dimensional cellular space without any central entity. In 2004 G. Beni updated his definition of swarm robotics. He stated that they needed to be simple, identical, and self-organising, and the system must be scalable, and only local communication is available amongst swarm members. G. Beni stated that the number of robots needed for a swarm is between 10^2 - 10^{26}.

This robot is much larger than traditional modern swarm robots but could be used just as effectively to cover a much larger area. You could have one swarm cover many smaller buildings or a particularly large building such as a hotel or skyscraper. Combined these robots could quickly cover a large area and alert all other robots to an identified danger such as a gas leak or structurally weak area.

The squad size of these robots would be closer to the 10s of robots, much like the teams in robocup[20] that uses small teams of collaborative robots that work together as a team to compete with other robot teams in a football match.

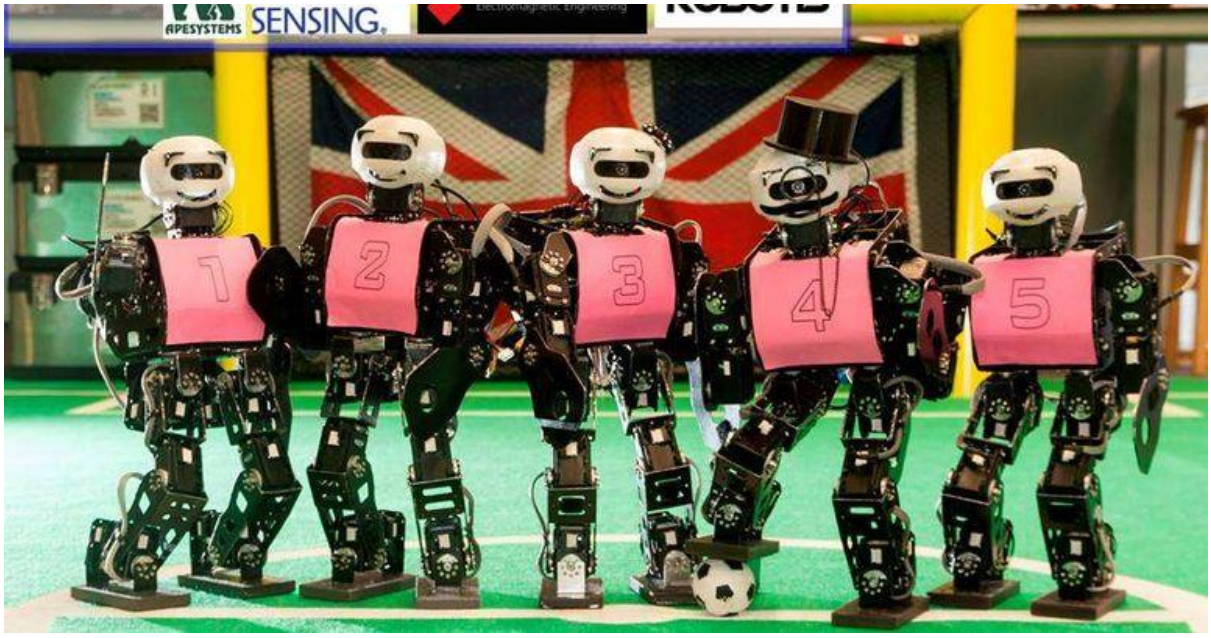


Figure 52: University of Plymouth's robot football team

Instead of football, our robots will be used to collaboratively explore and create a digital map of a disaster site for rescuers to use.

SLAM

Simultaneous Localisation And Mapping (SLAM) is the process of a robot being placed in an unknown environment and incrementally building a map whilst simultaneously determining its location[19]. It uses a series of sensors to calculate the distance from itself to objects and uses these results to create a virtual map of the room it is in.

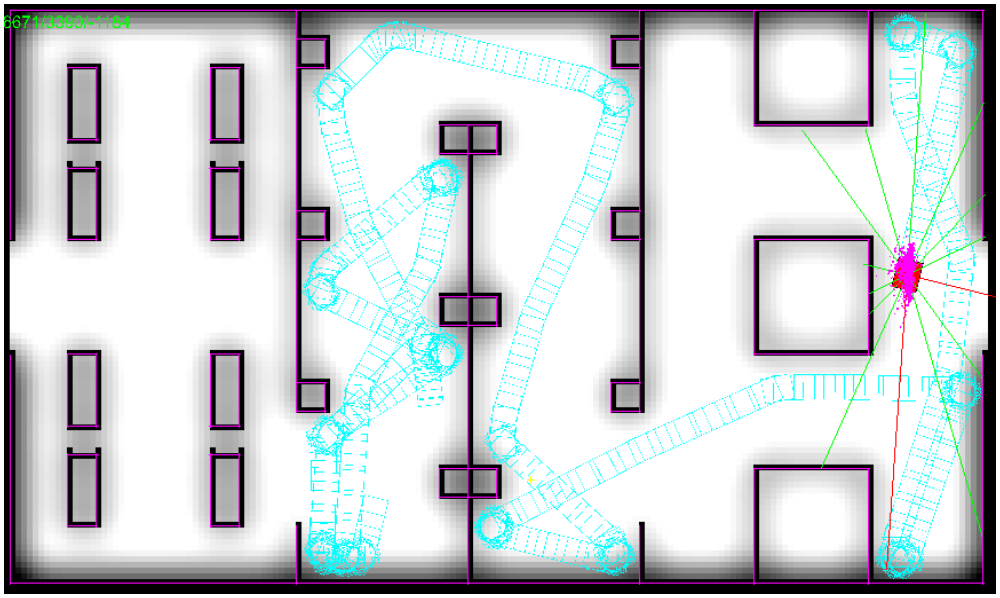


Figure 53: Robot used SLAM to navigate through room

In Figure 53 [30] the green lines represent the sensors detecting the distance from the walls and the pink dots represent the localisation of the algorithm. The localisation algorithm has successfully located where the robot is in the map. The sensor readings are used fill an occupancy grid that is used to build the map for future path planning, Figure 54[30]

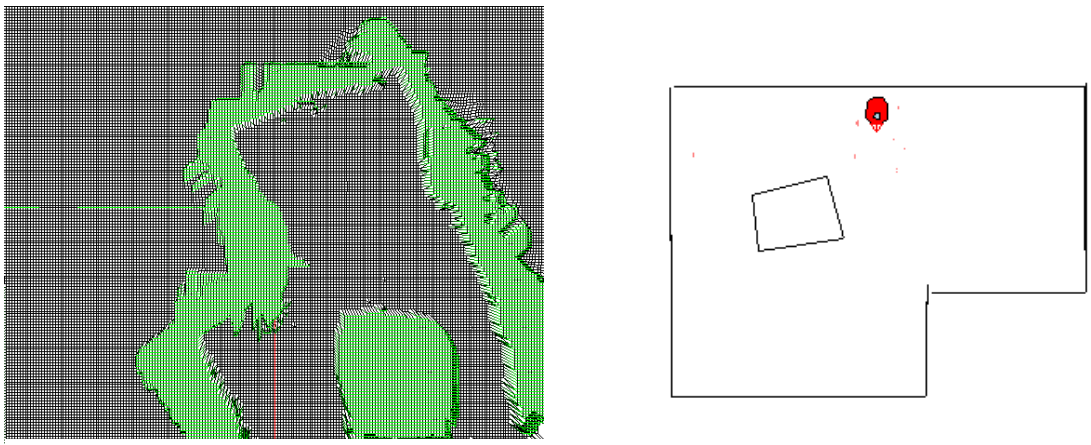


Figure 54: Occupancy grid generated and room it's in

SLAM will allow the robot to navigate through its environment, avoiding obstacles that it is not able to path through. It will form a map of the area as it goes. Combined with a squad of robots this data can be shared and would result in a quick method of creating a map of the entire collapsed building. Including a 360° degree camera would enable the robots to generate a real world view of the room for human rescuers giving them a large advantage when they attempt a rescue.

The Lynxmotion SSC-U32 board has 8 analogue inputs that will allow for multiple sensors, such as ultra-sonics or lidar, to be included. Ultra-sonics can be purchased for minimal cost, keeping with the low-cost design of the robot.

WiFi mesh

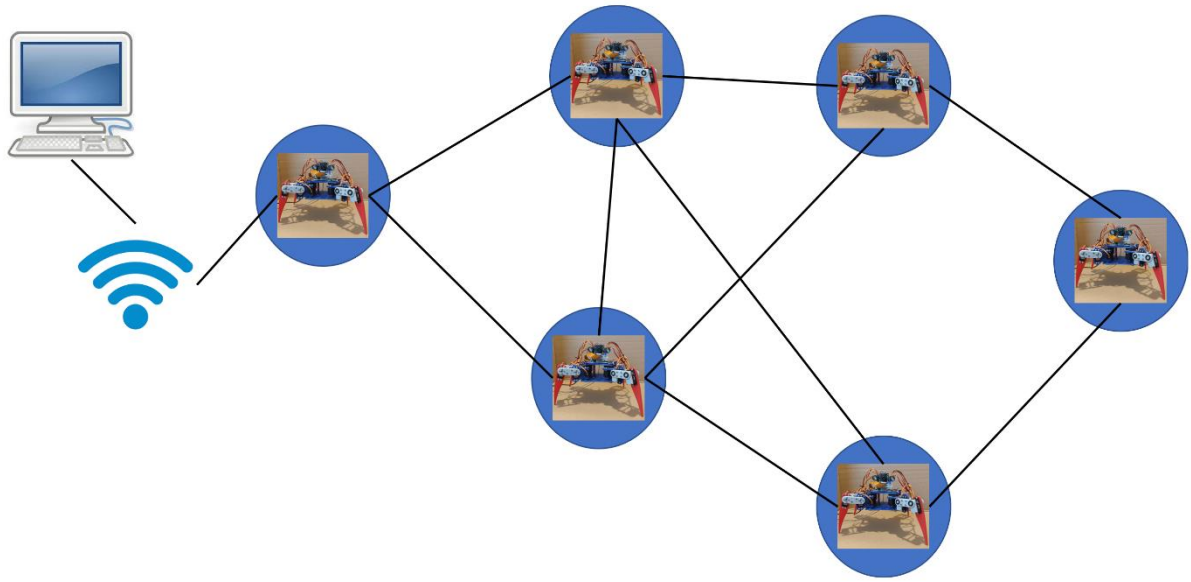


Figure 55: Wi-Fi mesh between raspberry Pi's

An issue that will be more apparent with larger buildings is Wi-Fi connectivity. The rubble of a collapsed building will reduce the signal strength the Pi receives potentially causing them to move out of range and then freeze because they are not receiving any commands. However, if you combine multiple raspberry Pi's together you can form a Wi-Fi mesh where each Pi acts as a relay and shares data in one large network. This method only works in tandem with a squad of robots as you will require multiple robots to form a mesh.

An external computer can combine the SLAM data and camera data together to form a singular map that will give rescuers the opportunity to remotely scout the building using the 3D map generated by the robots, giving them prior knowledge on dangerous areas or blocked paths. This will reduce the risk to rescuers and reduce the time it takes for them to find the victims.

This method of map building is extremely expandable and can use many different types of robots, whether it be drones for large clear areas, or small snake robots for in rubble scouting. Making it expandable opens many doors for future upgrades.

Audio

If you were the victim of a natural disaster you will be scared, and hearing a scuttling sound grow close to you when you are trapped in the dark is likely to add to your fear.

We believe that we can reduce some of the fear felt from our robots by adding speakers onto them that will play a pre-recorded call in the native language of the region for survivors on repeat, until it detects a response at which point it will travel towards the sounds. Once it locates the victim it will switch over to remote audio so that the rescuers can personally alleviate some of the stress by reassuring the victim that help is on its away and give any advice on what to do.



Figure 56: Scary robot



Figure 57: Robot to the rescue

Conclusions

This project showcases the feasibility of combining low-cost materials and manufacturing processes to form a fully functioning rescue robot. It shines hope on low-income areas of the world in that they can afford to mass produce rescue robots that will be able to collaboratively work with human rescuers to reduce the search and rescue time, whilst reducing the risk faced by rescuers.

The project has delivered a fully working prototype robot that can move under independent power source, communicate remotely with its operator, detect victims, and walk through rough terrain. Much was learnt about the mechanical requirements for rigidity of construction and effective joint design. Open sources AI algorithms for hand and body detection were used to provide advanced remote victim recognition capabilities at minimal cost.

The practicality of 3D printing has shone through, both in terms of cost effectiveness and production time. On a domestic 3D printer, each robot body takes less than a day to print and costs less than £10 in materials. The modular design lends itself to multiple different sensor arrays and future upgrades. The camera module implemented in the prototype provides remote viewing and the hand detection aids in searching for victims.

This project demonstrates that a low-cost, 3D printed robot with open source AI detection is a practical proposition.

References

1. 2021. [online] Available at: <https://ourworldindata.org/natural-disasters#:~:text=Summary,from%200.01%25%20to%200.4%25.> [Accessed 15 March 2021].
2. Hannah Ritchie and Max Roser (2014) - "Natural Disasters".
3. https://www.theatlantic.com/international/archive/2011/03/did-engineering-save-lives-japanese-earthquake/348771/
4. F. Matsuno and S. Tadokoro, "Rescue Robots and Systems in Japan," 2004 IEEE International Conference on Robotics and Biomimetics, Shenyang, China, 2004, pp. 12-20, doi: 10.1109/ROBIO.2004.1521744.
5. https://cdn.sparkfun.com/assets/learn_tutorials/6/7/6/PiZero_1.pdf
6. https://www.pidramble.com/wiki/benchmarks/power-consumption
7. https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/
8. https://store.arduino.cc/arduino-nano
9. http://www.ee.ic.ac.uk/pcheung/teaching/DE1_EE/stores/sg90_datasheet.pdf
10. https://www.dgiracing.com/dgi-1-5-steering-servo-ds5160-high-torque-60kg/
11. https://www.electronicoscaldas.com/datasheet/MG996R_Tower-Pro.pdf
12. https://www.adafruit.com/product/3416
13. https://github.com/sunfounder/SunFounder_Crawling_Quadruped_Robot_Kit_for_Arduino/blob/master/CD/User_Manual.pdf
14. R. McNeill Alexander. “Locomotion of Reptiles”, Herpetological Bulletin 2012
15. https://www.robotshop.com/media/files/content/a/adu/pdf/arducam-5mp-ov5647-camera-module-raspberry-pi-w--ir-led-for-night-vision-datasheet.pdf
16. https://arxiv.org/pdf/2101.00671.pdf
17. G. Beni, “The concept of cellular robotic system,” in Proceedings IEEE International Symposium on Intelligent Control 1988, Aug 1988, pp. 57–62.
18. T. Fukuda and S. Nakagawa, “Approach to the dynamically reconfigurable robotic system,” Journal of Intelligent and Robotic Systems, vol. 1, no. 1, pp. 55–72, 1988.
19. H. Durrant-Whyte and T. Bailey, "Simultaneous localization and mapping: part I," in IEEE Robotics & Automation Magazine, vol. 13, no. 2, pp. 99-110, June 2006, doi: 10.1109/MRA.2006.1638022.
20. https://www.facebook.com/University-of-Plymouth-Robot-Football-148285347160/
21. https://ourworldindata.org/natural-disasters#:~:text=Summary,from%200.01%25%20to%200.4%25.
22. S. Tadokoro, T. Takamori, et al., On robotic rescue facilities for disastrous earthquakes -from the Great Hanshin Awaji (Kobe) Earthquake-, Journal of Robotics and Mechatronics, 9(1), pp.46–56, 1997.
23. Tokyo Fire Fighting Department Planning Section, New Fire Fighting Strategies, Tokyo Horei Publ., 2002
24. F. Matsuno and T Kawai, Fusion of Engineering, Medicine and Psychology in Rescue, Proceedings of the 2003 SICE Annual Conference , pp. 1102-1103, 2002.
25. https://news.stanford.edu/2017/07/19/stanford-researchers-develop-new-type-soft-growing-robot/
26. https://robocup-rescue.github.io/team_description_papers/2018/Champ2018_Thailand_iRAP_TDP.pdf

27. https://www.robotshop.com/media/files/pdf2/lynxmotion_ssc-32u_usb_user_guide.pdf
28. https://learn.adafruit.com/16-channel-pwm-servo-driver?view=all
29. http://wiki.wit-motion.com/english/lib/exe/fetch.php?media=smart_control:servomotor:docs:16chs_servo_motor_controller_v2.0.pdf
30. P. M. Fletcher, MSc Intelligent Systems and Robotics, De Montfort University, personal communication

Appendix 1, 3D models

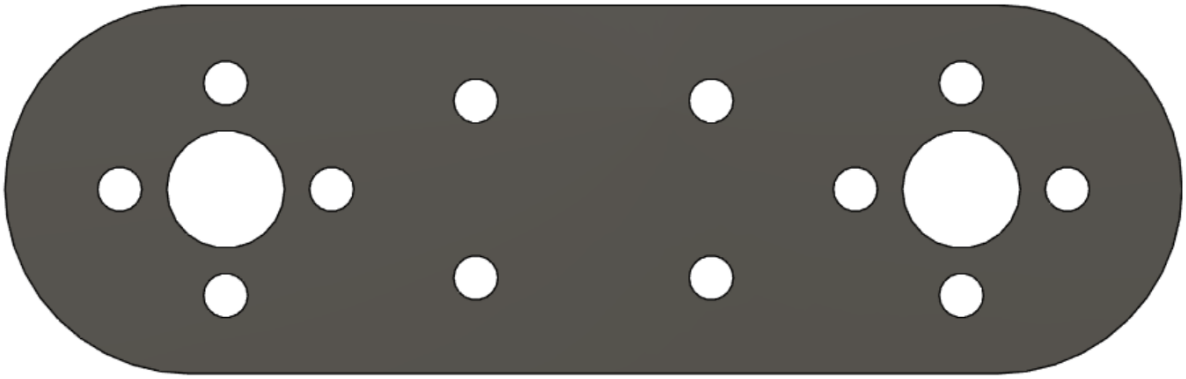


Figure 59: Servo horn plate back

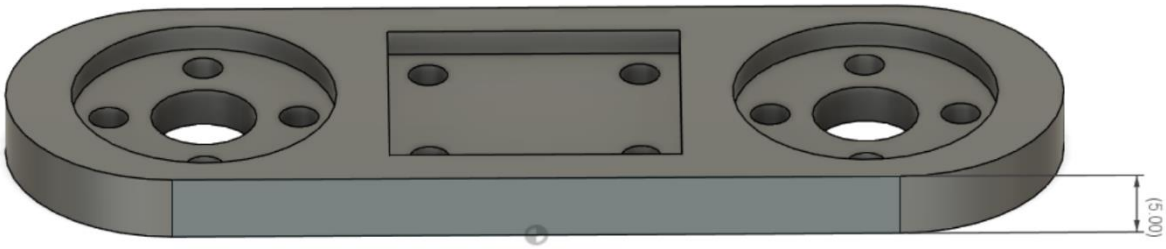


Figure 58: Servo horn plate side

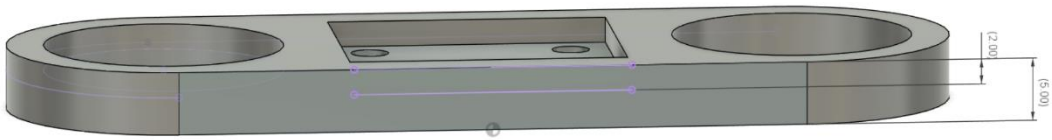


Figure 60: Side view of back brace

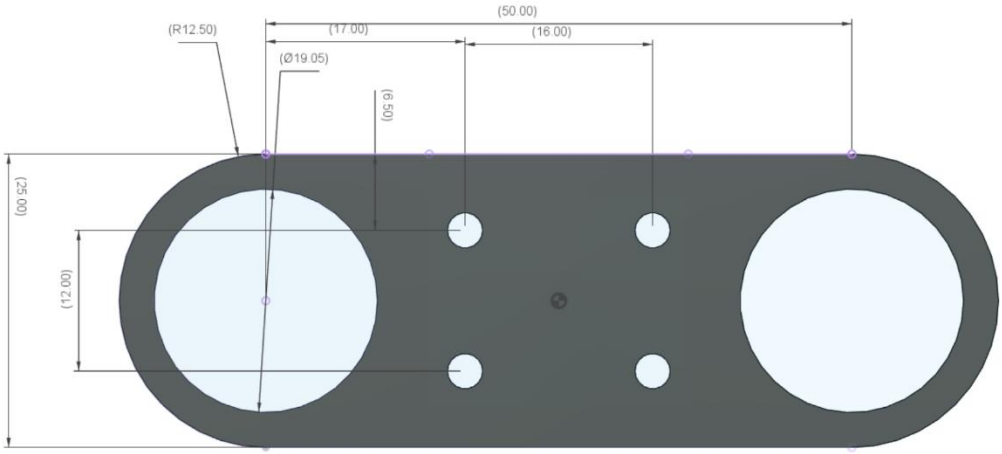


Figure 61: Bottom-up view of back brace

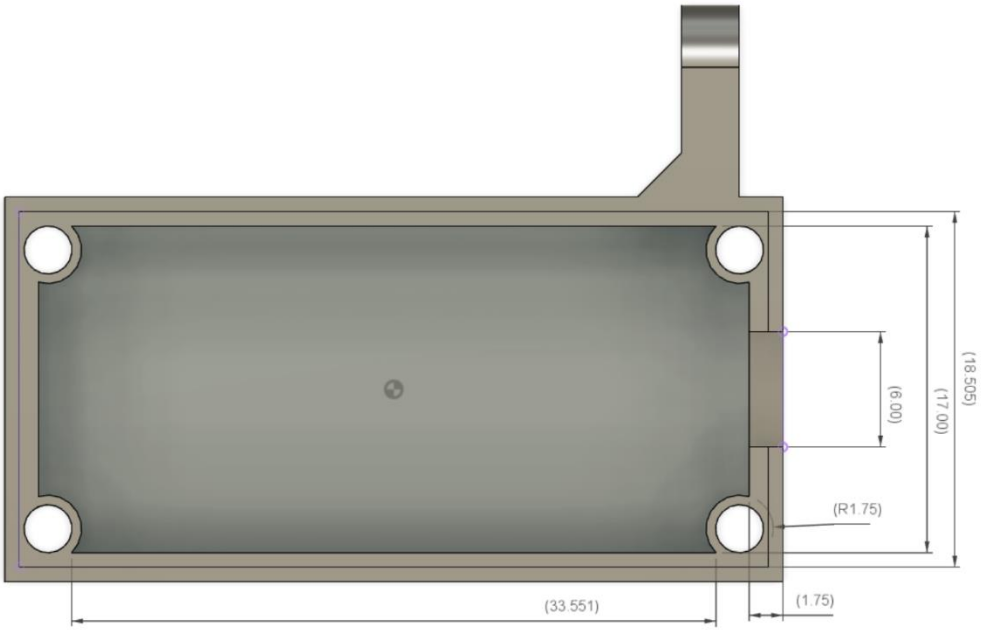


Figure 63: Bottom-up view of servo back



Figure 62: Side view of servo back

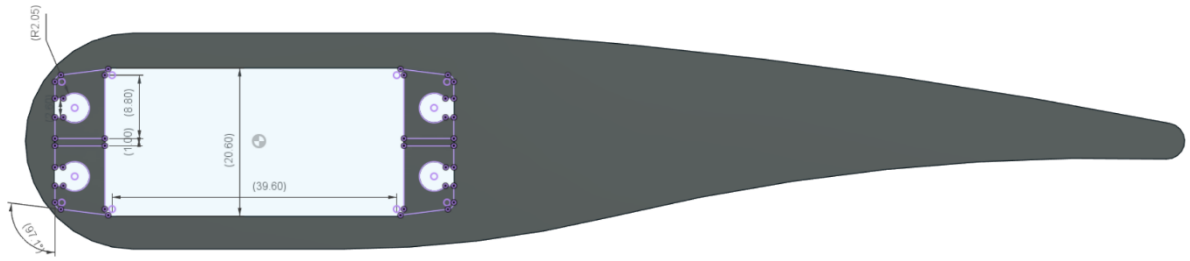


Figure 65: Foot

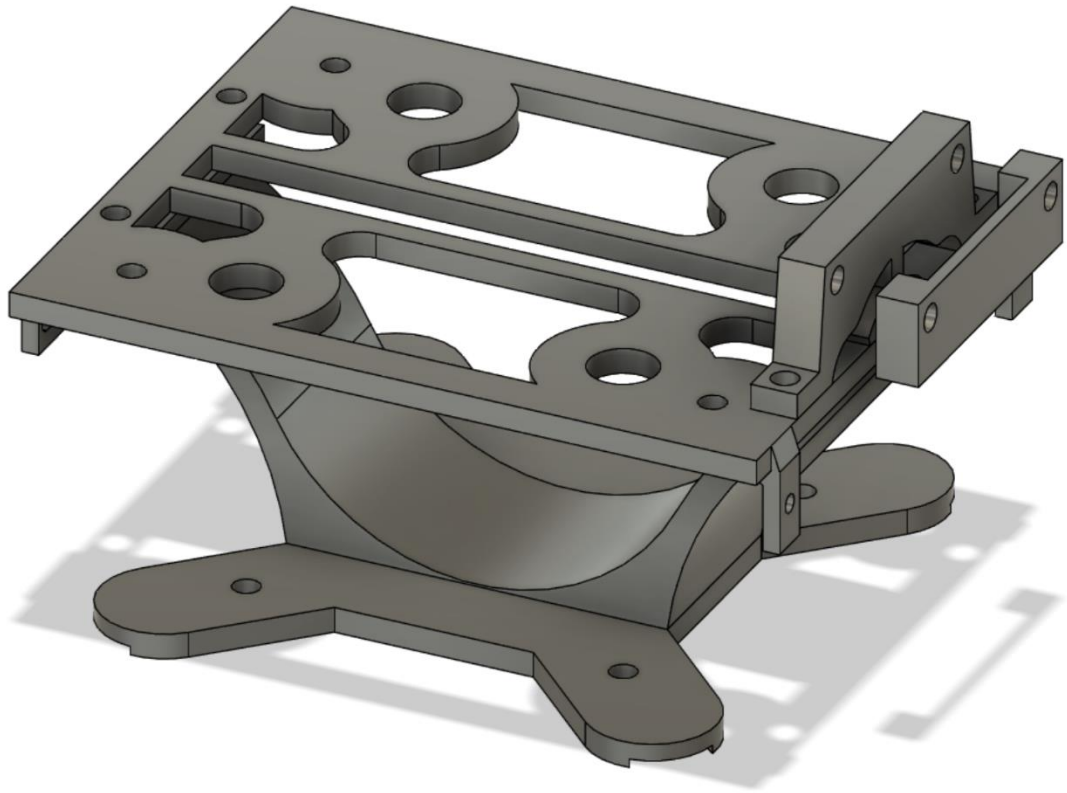


Figure 64: Head diagonal view

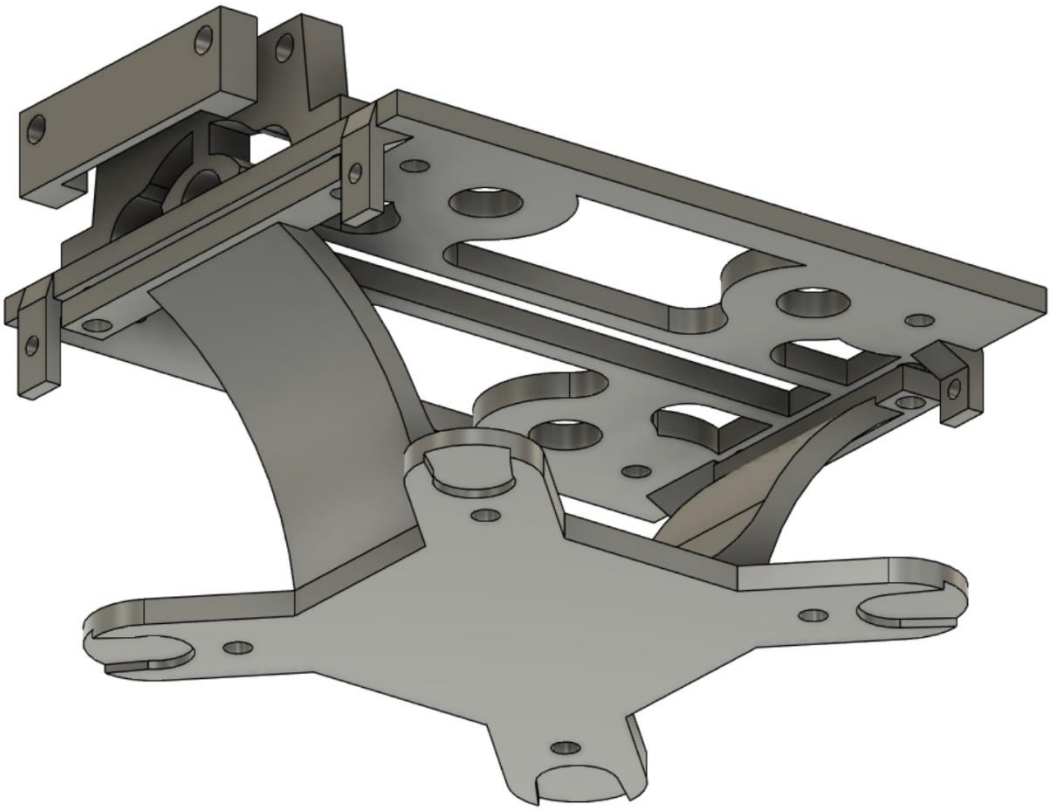


Figure 67: Head under view

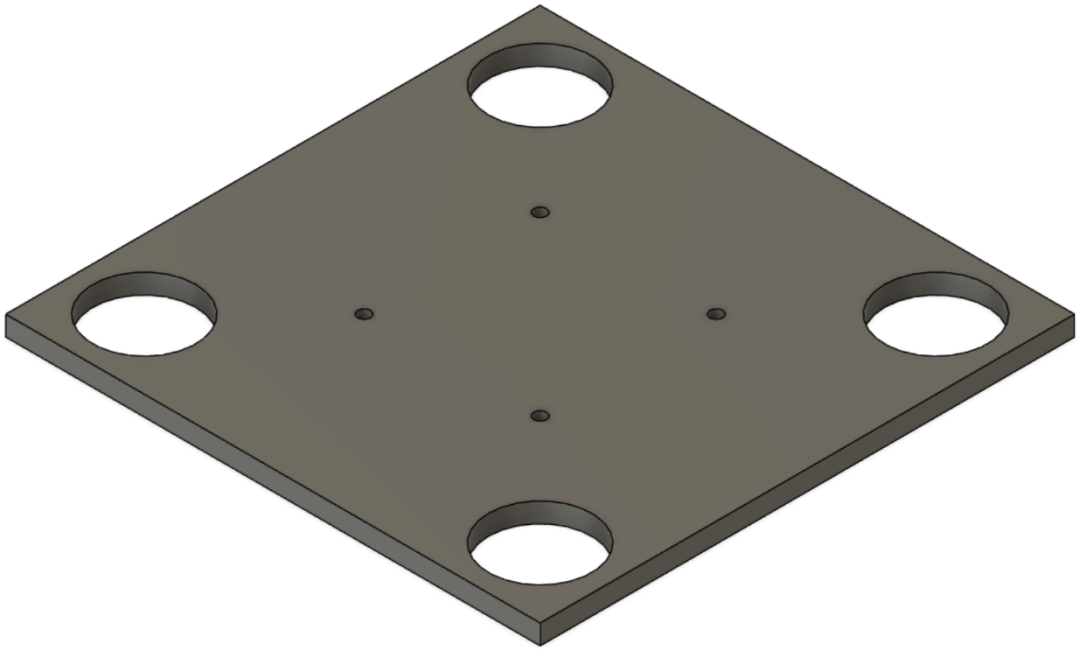


Figure 66: Lower body plate



Figure 68: Upper body plate

Appendix 2, cost breakdown

Component	Quantity	Total prices (incl tax. £)
MG996R servo	12	50.04
Lynxmotion SSC-U32	1	51.54
Raspberry Pi 0 W	1	9.30
Suleve M3AS5	1	2.71
Turnigy 5000mAH 4S 25C LiPo	1	35.42
3D printed body	1	8.31
M3X10MM screws	30	1.92
M3 nuts	30	3.35
M3 washers	30	10.6
M3 x 25mm screws	30	5.59
Metal servo arm 25T disc	8	13
Pololu 5V, 15A step-down voltage regulator D24V150F5	1	31.19
1180098 bearings	8	3.65
Arducam UC-376	1	20.43