University of London

Bachelor of Science (Honours) in Computer Science (Machine Learning and Artificial Intelligence)

Individual Mid-Semester Report for Artificial Intelligence Part B

Evolutionary Robotics for Mountain Climbing

<Nguyen Ngoc Quoc Cuong> <SIM ID: 10245826>

Academic Year 2025

Year 3 Semester 2

1. Introduction and Setup:205 words
2. Basic Encoding Parameter Tuning: 434 words
3. Advanced Encoding: Morphological Evolution: 189 words
4. Advanced Encoding: Articulation and Control: 320 words
5. Final Model Configuration: 150 words
6. Extension: Creature's Symmetry and Senses: 368 words
7. GA's Output Comparison and Conclusion: 427 words

Total Word Counts: 2091

Table of Contents:

1. Introduction and Setup **2**

2. Basic Encoding Parameter Tuning **4**

Gene Count 5

Point Mutation Rate 7

Limb Growth and Removal 8

3. Advanced Encoding: Morphological Evolution **10**

The Impact of Limb Shape and Morphological Exploits 10

4. Advanced Encoding: Articulation and Control **13**

Joint Axis Flexibility versus Structural Compensation 13

The Dynamics of Commanded Joint Strength (Force) 15

5. Final Model Configuration **17**

6. Extension: Creature's Symmetry and Senses **18**

Bilateral Symmetry 18

Reactive Touch Sensors 19

7. GA's Output Comparison and Conclusion **21**

1. Introduction and Setup

This report presents the implementation and experimentation for modifying an existing Genetic Algorithm (GA) framework to evolve agents capable of climbing an irregular Gaussian mountain using the PyBullet physics simulator. Key tasks included adapting the environment, revising the fitness function, and adjusting parameters for consistency and fairness.

To enable the evolution of robust mountain-climbing creatures, several modifications were implemented. The environment, including the arena and mountain, was integrated directly into the run_creature method for consistent evaluation.

The mountain's scale was increased (to 19x19x7 m) to require genuine climbing behaviour and discourage morphological exploits, such as creatures evolved having too many limbs or large limbs, achieving high fitness by rolling or simply lying nearer to the peak.

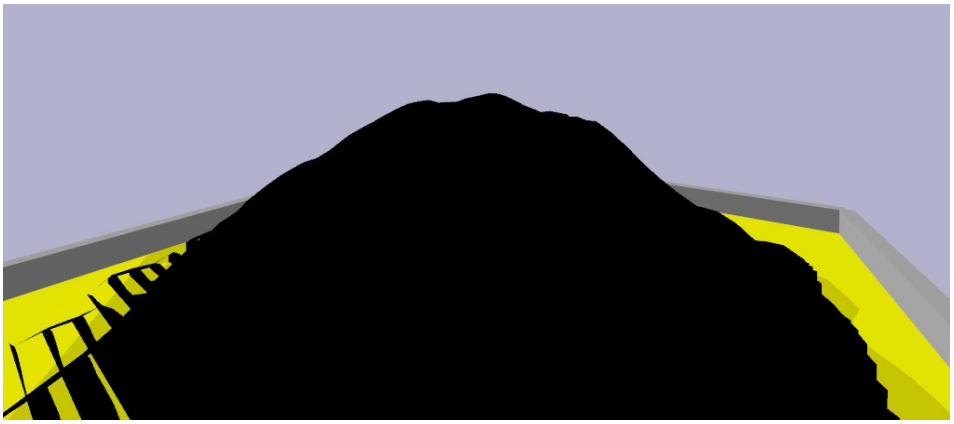


Figure 1.1: Enlarged Mountain

To ensure simulation stability, each creature's URDF was saved to a unique temporary filename and deleted after loading, with all loading operations wrapped in a try-except block to handle malformed creatures.

```
# Initialize creature's position tracking
try:
    init_pos, orn = p.getBasePositionAndOrientation(cid, physicsClientId=pid)
except p.error as e:
    # Despite the changes to the made the p.loadURDF() reading more consistent,
    # The error always pops up here. Thus, Ive decided to just remove the creature that cant be loaded.
    # To ensure that a training session (which can last for 2 days) don't get interrupted.
    print(f"ERROR: getBasePositionAndOrientation failed for CID {cid} (Iteration {current_iter}): {e}")
    print(f"This indicates the creature was loaded but is unstable or PyBullet's state is corrupted.")
    # Assign a very poor fitness for this creature as it cannot be evaluated
    cr.start_position = [0,0,-1000]
    cr.last_position = [0,0,-1000]
    cr.closest_position = [0,0,-1000]
    return # Exit run_creature early
```

Figure 1.2: Snippet of try-except wrapping getBasePositionAndOrientation().

Finally, a standardised setup was enforced: creatures spawn at a fixed position (7, 7, 8), undergo a touched_ground check for a stable start, and are constrained by a boundary "kill zone" to disqualify non-climbing strategies. For the parameter testing, the fitness function was simply defined as 1 / (1 + distance_to_peak), rewarding for being near the mountain's peak.

```
def distance_to_peak(self,height = 6):
    if self.closest_position is None:
        return 999
    p_mountain = np.asarray([0,0,height])
    p_bot = np.asarray(self.closest_position)
    dist = np.linalg.norm(p_mountain-p_bot)
    return dist
```

Figure 1.3: creature.py's distance_to_peak()

```
fits = [ (1 / (1 + cr.distance_to_peak())) for cr in pop.creatures]
```

Figure 1.4: The initial fitness function for parameter testing.

2. Basic Encoding Parameter Tuning

To establish an effective baseline for evolution, the testing Genetic Algorithm (GA) parameters need to be defined. For these experiments, the **base configuration** was used to ensure speed. These are a population of 10 creatures, running for 500 generations. Unless specified, this configuration included an initial gene_count of 3, a point mutation rate of 0.1, shrink/grow rates of 0.25/0.1, a link-recurrence scale of 3, cylinder-shaped limbs, and a joint effort of 1.0.

The base template GA framework deploys an "**elitism" strategy**, ensuring the fittest creature is preserved to the next generation. To fully leverage this and provide a more diverse pool of candidates, a larger population size of 100 will be adopted for the final comparison.

```
for iteration in range(500): #testing 500, final 1500
    print(f"testing iteration no: {iteration}")
    for cr in pop.creatures:
        sim.run_creature(cr, 20 * 240, current_iter=iteration)
    fits = [ cr.fitness_function() for cr in pop.creatures]
    links = [len(cr.get_expanded_links()) for cr in pop.creatures] #total number of independantly moving rigid body segments or parts
    max_fit_log = np.max(fits)
    min_fit_log = np.min(fits)
    mean_fit_log = np.mean(fits)
    std_fit_log = np.std(fits)
    max_link_log = np.round(np.max(links))
    min_link_log = np.round(np.min(links))
    mean_link_log = np.round(np.mean(links))
    print(iteration, "fittest:", max_fit_log, "mean:", mean_fit_log, "mean links", mean_link_log, "max links",max_link_log)
```

Figure 2.1: The code that retrieves the creatures' performance and data.

```
max_fit = np.max(fits)
for cr, fit, link in zip(pop.creatures,fits,links):
    if fit == max_fit:
        new_cr = creature.Creature(gene_count)
        new_cr.update_dna(cr.dna)

        creature_fit_log = fit
        creature_link_log = link
        creature_distance_to_peak = cr.distance_to_peak()
        print(f"Best distance from moutain peak: {creature_distance_to_peak}")
        print(f"best creature data: {cr.highest_z}, {cr.last_position}")

        new_creatures[0] = new_cr
        filename = "extension/elite_"+str(iteration)+".csv"
        genome.Genome.to_csv(cr.dna, filename)
        break
pop.creatures = new_creatures
```

Figure 2.2: The code that applies the "Elitism" strategy and saves the creatures' genome.

Gene Count

The gene_count parameter, setting the initial number of links, was tested with values of 2, 4, and 6. A count of 6 yielded the highest maximum fitness but did so by producing overly complex "bushy" creatures that likely succeeded through morphological exploits (e.g., size or rolling) rather than efficient climbing. To focus evolution on locomotion and avoid rewarding these exploits, a **gene_count of 2 was selected** for the final model.

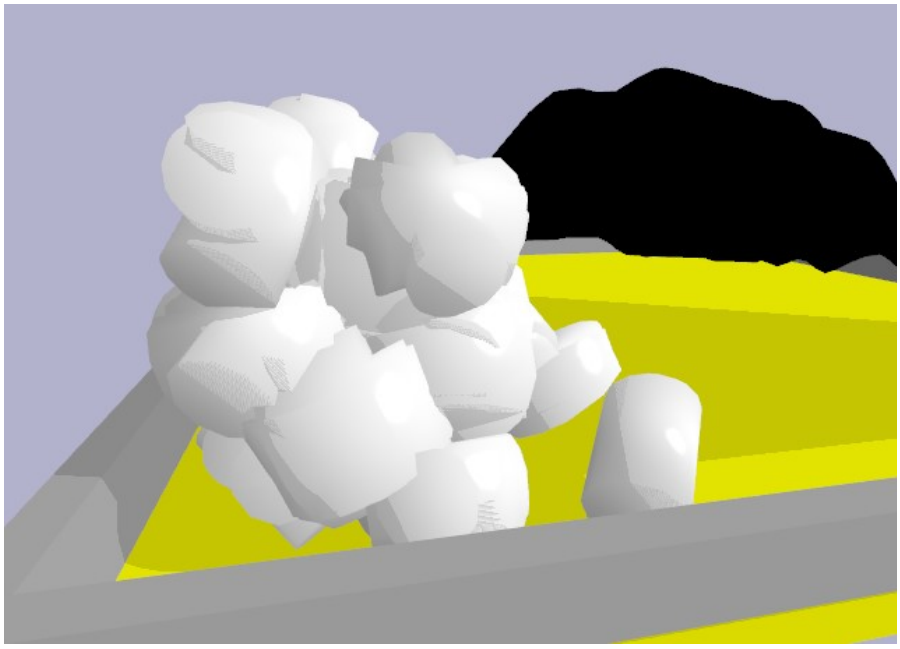


Figure 2.3: An example of a “Bushy” creature with gene_count = 6.

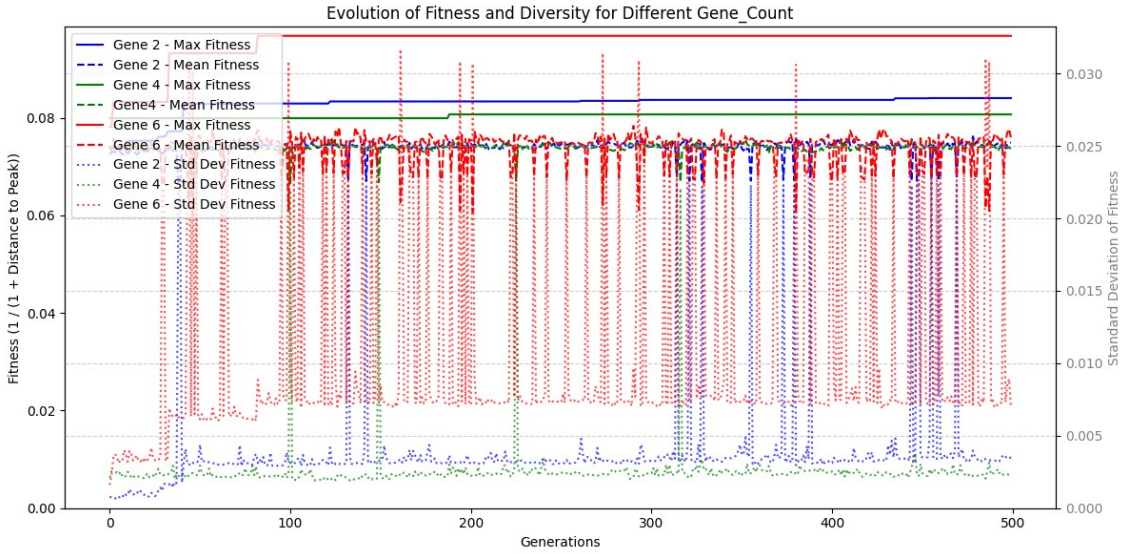


Figure 2.4: Fitness Results of different gene_count values.

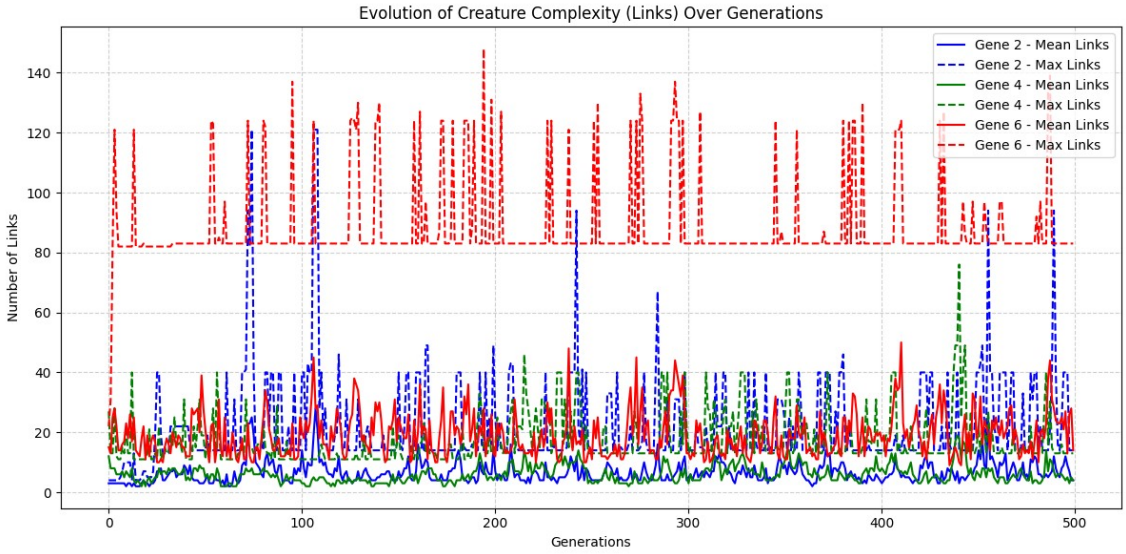


Figure 2.5: Links Count for creatures of different gene_count values.

Comparison between each parameter's best-performing creature of the 500th generation				
Parameters (gene_count)	Fitness	Links	Distance from peak	Comparison to the chosen parameter (fit)
2	0.084018	14	10.902110 (m)	100%
4	0.080676	13	11.395189 (m)	96.02%
6	0.096775	83	9.333237 (m)	115.18%

Point Mutation Rate

The point_mutate_rate (PMR), which controls random changes to individual gene values, was tested at 0.01, 0.05, and 0.2. A rate of 0.05 provided the best balance between introducing new genetic material for exploration and preserving beneficial traits for exploitation. Higher rates introduced excessive genetic disruption, while lower rates limited exploratory potential. A **PMR of 0.05 was therefore chosen** as the optimal setting.

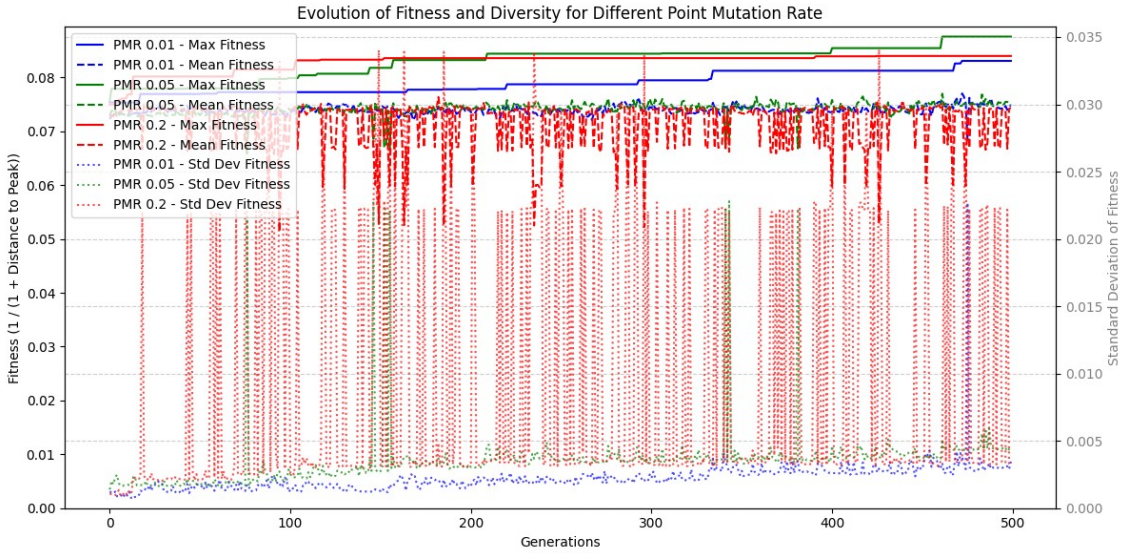


Figure 2.6: Fitness Results of different Point Mutation Rate values.

Comparison between each parameter's best-performing creature of the 500th generation				
Parameters (PMR)	Fitness	Links	Distance from peak	Comparison to the chosen parameter (fit)
0.01	0.083074	11	11.037367 (m)	94.84%
0.05	0.087585	67	10.417395 (m)	100%
0.2	0.083989	33	10.906208 (m)	95.89%

Limb Growth and Removal

The balance between adding new limbs (grow_mutate_rate or GMR) and removing existing ones (shrink_mutate_rate or SMR) is crucial for evolving effective morphologies. Experiments revealed a clear trade-off between exploration and exploitation. A high GMR of 0.3, for instance, leaned heavily towards exploration by rapidly introducing new limbs. As seen in Figure 2.4, this led to early success followed by stagnation, as the algorithm struggled to refine the resulting overly complex and "bushy" creatures. Conversely, a low GMR of 0.05 favoured exploitation, allowing for refinement but lacking the diversity to find better solutions, resulting in slow progression.

A similar pattern emerged for SMR. A low rate of SMR=0.1 allowed for deep exploitation of early, successful body plans but hindered later exploration due to its reluctance to prune, leading to long plateaus (Figure 2.5). An aggressive SMR=0.5 disrupted beneficial structures too frequently.

Therefore, moderate rates of **GMR=0.15** and **SMR=0.3** were found to be suitable. They strike a balance, encouraging consistent fitness growth by managing complexity without overly constraining change. This allows the GA to both explore new morphological avenues and effectively exploit promising designs once they are found.

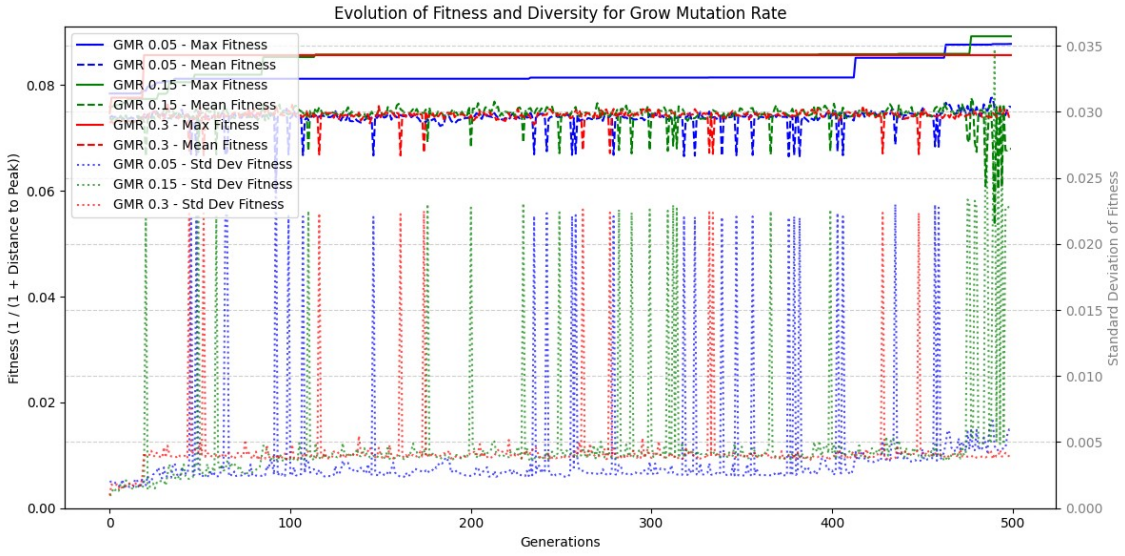


Figure 2.7: Fitness Results of different Grow Mutation Rate values.

Comparison between each parameter's best-performing creature of the 500th generation				
Parameters (GMR)	Fitness	Links	Distance from peak	Comparison to the chosen parameter (fit)
0.05	0.087792	16	10.390490 (m)	98.37%
0.15	0.089240	86	10.205716 (m)	100%
0.3	0.085680	10	10.671270 (m)	96.01%

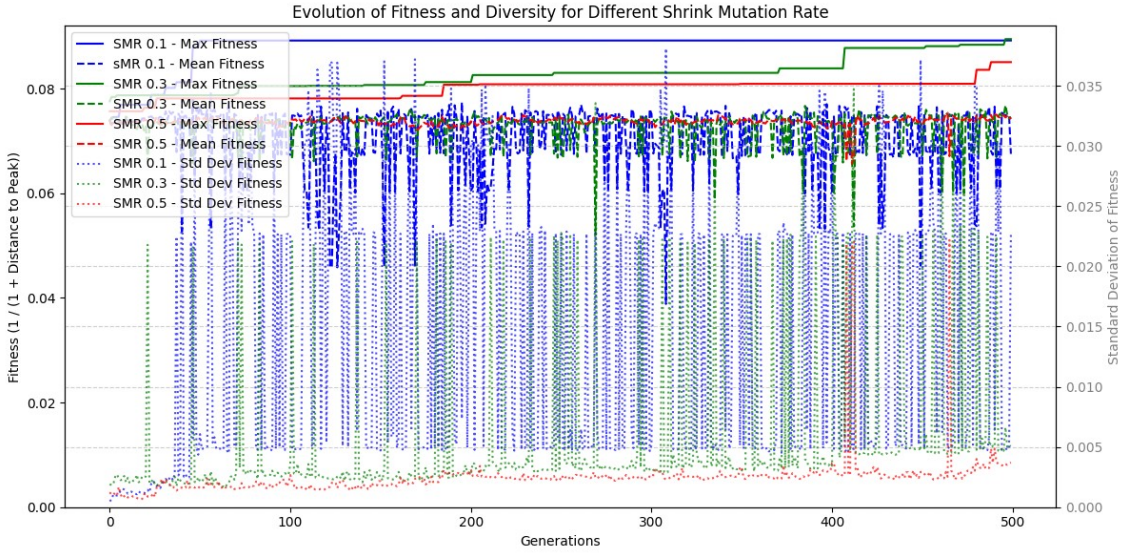


Figure 2.8: Fitness Results of different Shrink Mutation Rate values.

Comparison between each parameter's best-performing creature of the 500th generation				
Parameters (SMR)	Fitness	Links	Distance from peak	Comparison to the chosen parameter (fit)
0.1	0.089212	19	10.209187 (m)	99.70%
0.3	0.089478	23	10.175904 (m)	100%
0.5	0.085075	28	10.754321 (m)	95.07%

3. Advanced Encoding: Morphological Evolution

The Impact of Limb Shape and Morphological Exploits

This experiment investigated how limb geometry influences climbing performance by comparing fixed-shape populations (cuboid, sphere, cylinder) against a population where shape could evolve. Figure 3.1 shows that creatures with fixed cuboid and sphere limbs achieve the highest maximum fitness, indicating a performance advantage for non-cylindrical shapes.

However, Figure 3.2 reveals a critical insight: these high-performing populations also evolved significantly greater structural complexity. The cuboid-limbed population, in particular, exhibits extreme spikes in maximum link counts. This strong correlation suggests that their success stems not from superior climbing capability, but from **morphological exploits**. Large, "bushy" structures can achieve high fitness scores through their sheer size (being taller or wider) or by leveraging environmental features like the arena walls for propulsion, rather than by performing genuine, coordinated climbing.

In contrast, the cylinder ("Base") and evolving-shape ("Random Limbs") populations achieved their fitness but with far simpler structures. The evolving-shape population, in particular, demonstrated a more consistent and reliable evolutionary growth. This indicates that **allowing limb shape to evolve is the better choice for longer training**, as it promotes dependable progress without rewarding the evolution of excessively complex creatures that rely on environmental shortcuts.

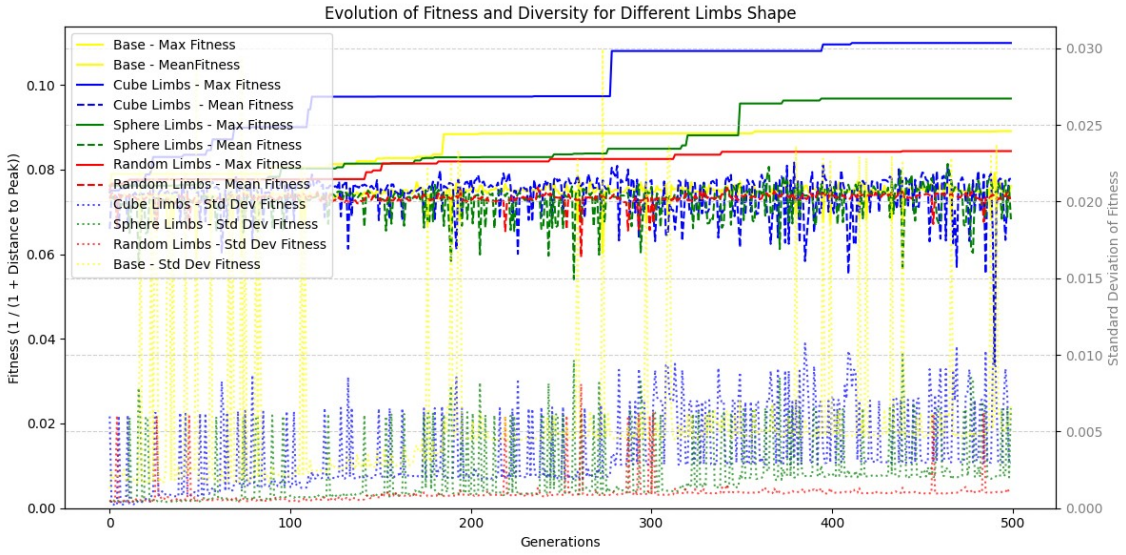


Figure 3.1: Fitness Results of creatures with different limb shapes.

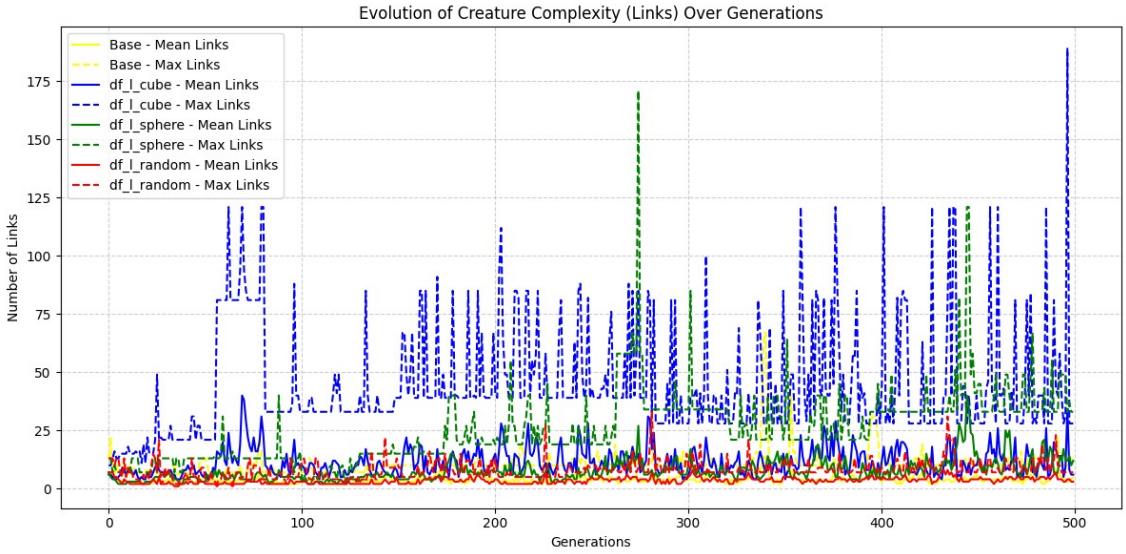


Figure 3.2: Links Count for creatures of creatures with different limb shapes.

Comparison between each limb shape's best-performing creature of the 500th generation				
Limb Shape	Fitness	Links	Distance from peak	Comparison to the chosen parameter (fit)
Base (Cylinder)	0.089051	10	10.229408 (m)	105.58%
Cuboid	0.089478	28	8.100996 (m)	106.09%
Sphere	0.085075	33	9.336089 (m)	100.87%
Random (Evolve)	0.084341	7	10.856621 (m)	100%

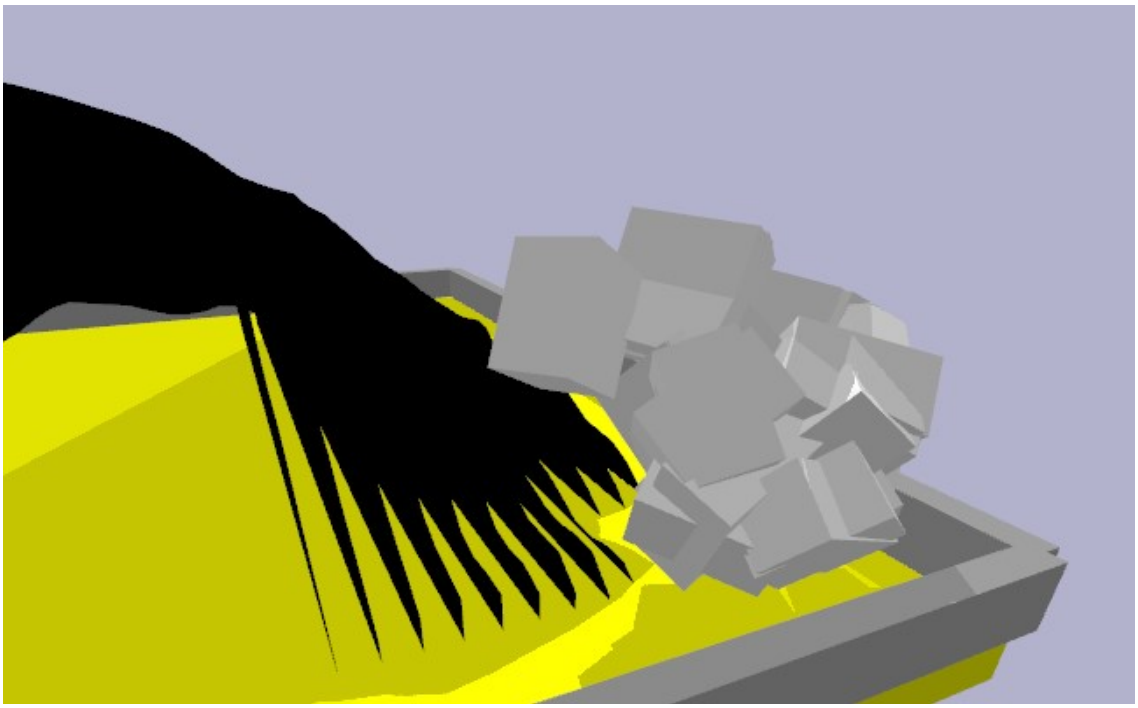


Figure 3.3: Simulation view of the creature with cuboid limbs.

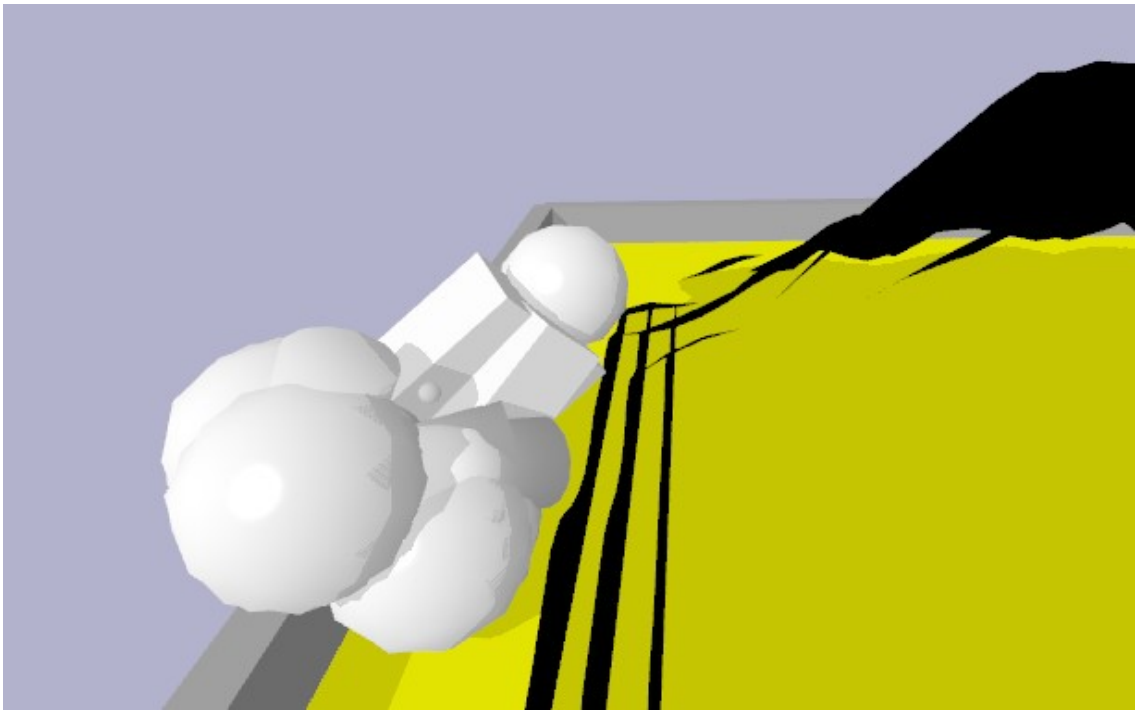


Figure 3.4: Simulation view of the creature with sphere limbs.

4. Advanced Encoding: Articulation and Control

Joint Axis Flexibility versus Structural Compensation

This experiment compared a fixed joint-axis-xyz to an evolving one ("Base") to understand the importance of axis freedom. While the fixed X-axis ("Axisjoint100") setting achieved a high maximum fitness, it did so via a notable compensatory strategy. As seen in Figure 4.2, creatures with fixed axes consistently evolved significantly more limbs than their evolving-axis counterparts.

This demonstrates that the GA **compensated for a lack of joint articulation by increasing structural complexity**. The creatures grew more limbs to achieve the necessary reach and leverage that a more optimally angled joint would otherwise provide. While this is a viable strategy, it leads to inefficient, "bushy" morphologies. The "Base" setting achieved comparable fitness with a much lower link count, indicating it found a more efficient solution by **optimising for limbs' movement angle rather than brute-force structure**. To encourage efficient designs, allowing the joint axis to evolve was selected as the optimal approach.

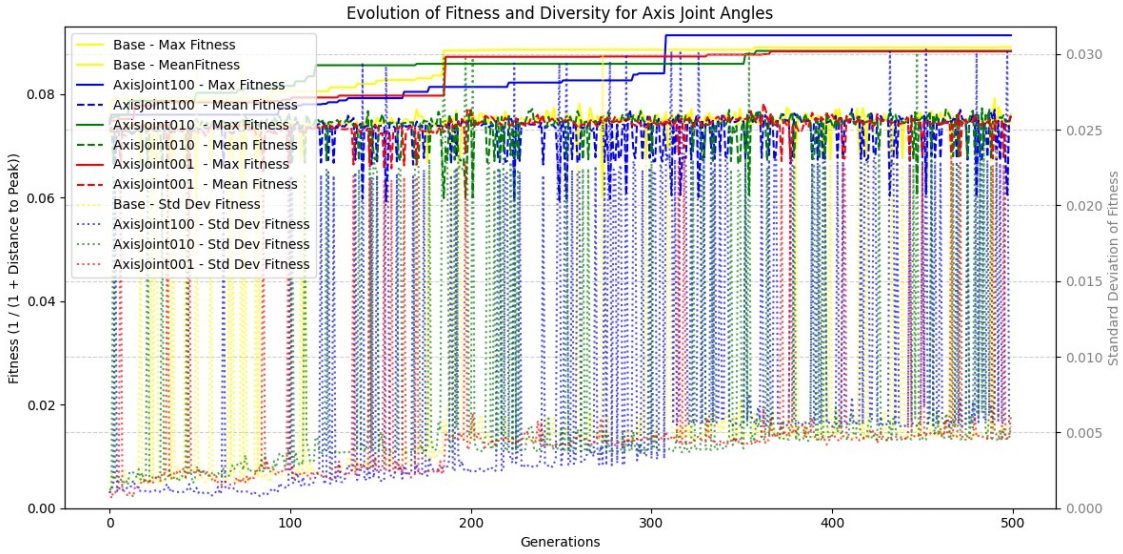


Figure 4.1: Fitness Results of different Axis Joint Angles.

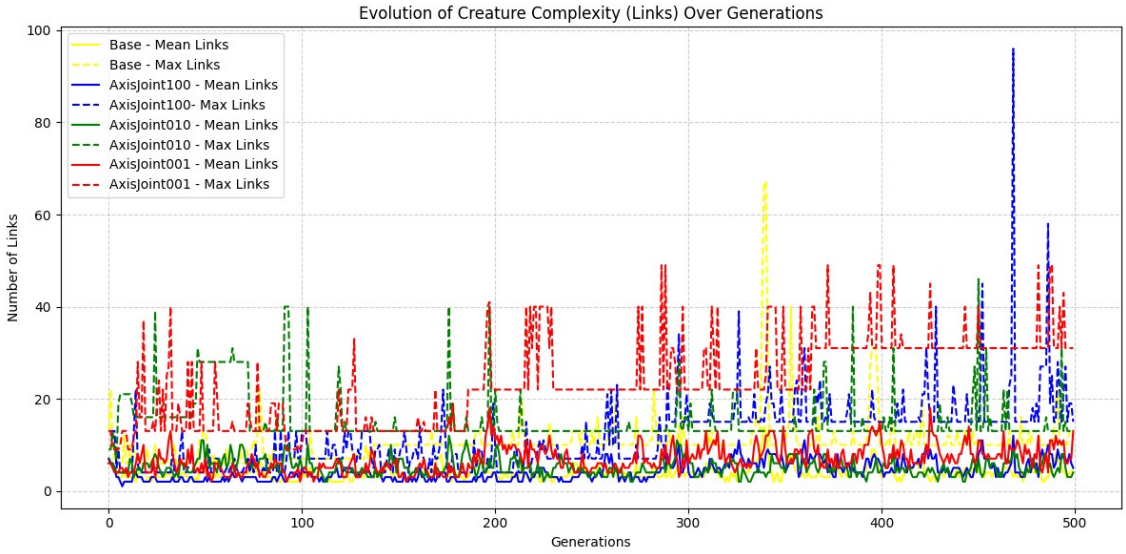


Figure 4.2: Links Count for creatures of different Axis Joint Angles values.

Comparison between each joint angle's best-performing creature of the 500th generation				
Axis joint angle	Fitness	Links	Distance from peak	Comparison to the chosen parameter (fit)
Base (x,y,z)	0.089051	10	10.229408 (m)	100%
100 (x)	0.091326	15	9.949774 (m)	102.55%
010 (y)	0.088307	13	10.324026 (m)	99.16%
001 (z)	0.088222	31	10.334967 (m)	99.06%

The Dynamics of Commanded Joint Strength (Force)

This experiment tested the force parameter within setJointMotorControl2, which dictates the commanded strength of each joint. A high URDF limit effort of 300 was set, and commanded forces of 1 ("Base"), 10, 20, and 40 were tested.

As seen in Figure 4.4, force=20 achieved the highest maximum fitness, followed closely by force=10. Both significantly outperformed the force=40 and force=1 settings, which had similar, lower performance. This suggests that a very low commanded force (1) is insufficient for effective climbing, while an excessive force (40) may introduce instability that hinders progress.

Unfortunately, the success of force=20 is not correlated with its climbing capability. Simulation observation shows that these creatures often evolved very large limbs, unlike having too many limbs like other “bushy” creatures. Thus, achieving high fitness through a morphological exploit (sheer size) rather than coordinated walking. In contrast, force=10 creatures showed more discernible, albeit rudimentary, walking-like behaviours. Therefore, **force=10 was selected as the optimal setting**, as it promotes more genuine movement strategies, representing a better solution to the intended climbing problem.

```
def update_motors(self, cid, cr):
    """
    cid is the id in the physics engine
    cr is a creature object
    """
    for jid in range(p.getNumJoints(cid,
                                    physicsClientId=self.physicsClientId)):
        m = cr.get_motors()[jid]

        p.setJointMotorControl2(cid, jid,
                controlMode=p.VELOCITY_CONTROL,
                targetVelocity=m.get_output(),
                force = 10, #default 5
                physicsClientId=self.physicsClientId)
```

Figure 4.3: The core code for applying the strength of the Links’ movement.

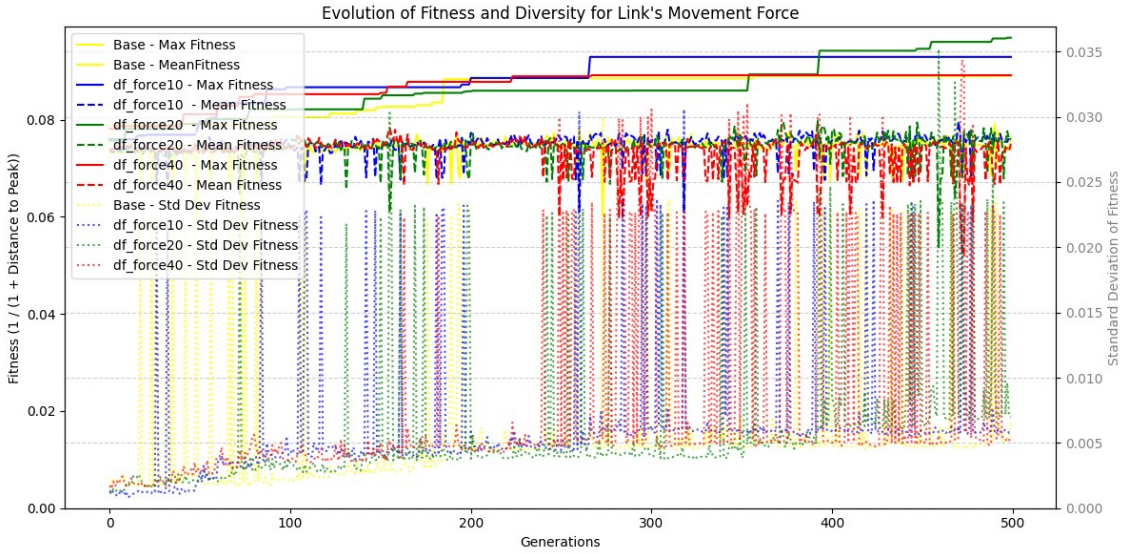


Figure 4.4: Fitness Results of different Force values.

Comparison between each force's best-performing creature of the 500th generation				
Force (300 cap)	Fitness	Links	Distance from peak	Comparison to the chosen parameter (fit)
Base, 5 (1 cap)	0.089051	10	10.229408 (m)	100%
10	0.092935	45	9.760143 (m)	102.55%
20	0.096896	43	9.320287 (m)	99.16%
40	0.089166	34	10.214990 (m)	99.06%

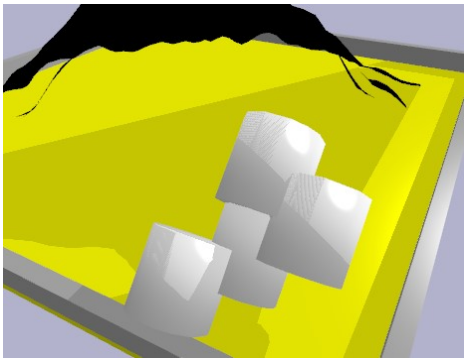


Figure 4.5: A creature trained with force = 20.

5. Final Model Configuration

Based on the series of tuning experiments detailed in the preceding sections, a final set of "optimal" parameters was established to create a robust evolutionary model for the final performance comparison. These parameters were selected to balance exploration with exploitation, encourage efficient morphology, and promote effective control strategies.

The configuration for the **"Final Tuned Model"** is as follows:

- **Population Size:** 100
- **Initial gene_count:** 2
- **Point Mutation Rate:** 0.05
- **Shrink Mutation Rate (SMR):** 0.3
- **Grow Mutation Rate (GMR):** 0.15
- **link-recurrence Scale:** 2
    - Chosen to prevent excessive "bushiness" while allowing for some complexity.
- **Limb Shape:** Allowed to evolve freely.
- **Spawn Point:** Fixed location.
- **Joint Axis:** Allowed to evolve freely.
- **Commanded force:** 10
    - supported by a higher LimitTagEffort of 300.
- **Modified Fitness Function**: Now, with consideration of the creature's stability penalty.

This parameter set was used to run a final, extensive simulation for 1000 generations. The "Extension Model" also used this same configuration.

```
def fitness_function(self,height = 6):
    w_horizontal = 0.5 # This is the penalty factor

    #if no self.final_horizontal_pos detected due to creature removal,
    # the output of np.linalg.norm(p_mountain-current_p_bot) would be 0
    # giving the creature a non-negative fitness due to no penalty value
    # thus, can disrupt the usually negative fitness value during training.
    if self.closest_position == [0,0, -999] or self.final_horizontal_pos is None:
        return -9999

    # Calculate the penalty for being far from the peak, (0,0,6)
    # self.final_horizontal_pos will be a (x,y,z)
    p_mountain = np.asarray([0,0,height])
    current_p_bot = np.asarray(self.final_horizontal_pos)
    horizontal_distance = np.linalg.norm(p_mountain-current_p_bot)
    horizontal_penalty = w_horizontal * horizontal_distance

    # The final fitness is the height score minus the penalty for being off-center.
    # this makes it such that the highest fitness achievable is 1.
        # (1 / (1 + self.distance_to_peak())) - horizontal_penalty = (1/(1 + 0)) - (0.5 * 0) = 1 - 0 = 1
    final_fitness = (1 / (1 + self.distance_to_peak())) - horizontal_penalty

    return final_fitness
```

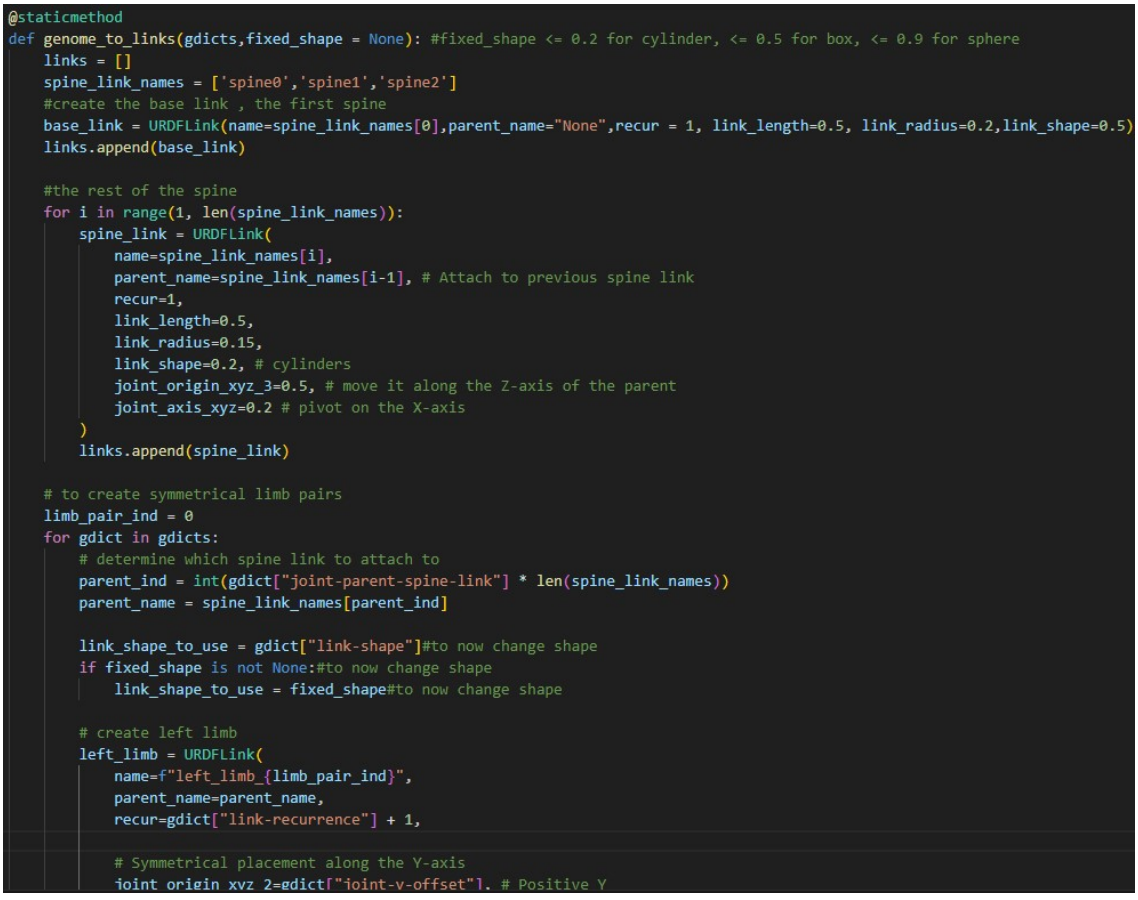
Figure 5.1: The code of the new fitness function.

6. Extension: Creature's Symmetry and Senses

The base template has two fundamental limitations. The GA's generation of "branch-like" morphologies, and the use of a "blind" open-loop motor control system. To address these issues directly, bio-inspired creature designs were implemented as an extension.

Bilateral Symmetry

To move away from growing recursively from one body part, a symmetrical body plan was enforced. This was achieved by altering the genome_to_links function. Instead of each gene defining a randomly-placed limb, on a new hard-coded central "spine" serves as a foundation. Each gene in the creature's DNA now defines a **symmetrical pair of limbs** that attach to this spine. This approach constrains the morphological search space, forcing the evolution of more structured and physically plausible body plans (e.g., resembling insects or quadrupeds), which may evolve a more coordinated locomotion.



```python
@staticmethod
def genome_to_links(gdicts,fixed_shape = None): #fixed_shape <= 0.2 for cylinder, <= 0.5 for box, <= 0.9 for sphere
    links = []
    spine_link_names = ['spine0','spine1','spine2']
    #create the base link , the first spine
    base_link = URDFLink(name=spine_link_names[0],parent_name="None",recur = 1, link_length=0.5, link_radius=0.2,link_shape=0.5)
    links.append(base_link)

    #the rest of the spine
    for i in range(1, len(spine_link_names)):
        spine_link = URDFLink(
            name=spine_link_names[i],
            parent_name=spine_link_names[i-1], # Attach to previous spine link
            recur=1,
            link_length=0.5,
            link_radius=0.15,
            link_shape=0.2, # cylinders
            joint_origin_xyz_3=0.5, # move it along the Z-axis of the parent
            joint_axis_xyz=0.2 # pivot on the X-axis
        )
        links.append(spine_link)

    # to create symmetrical limb pairs
    limb_pair_ind = 0
    for gdict in gdicts:
        # determine which spine link to attach to
        parent_ind = int(gdict["joint-parent-spine-link"] * len(spine_link_names))
        parent_name = spine_link_names[parent_ind]

        link_shape_to_use = gdict["link-shape"]#to now change shape
        if fixed_shape is not None:#to now change shape
            link_shape_to_use = fixed_shape#to now change shape

        # create left limb
        left_limb = URDFLink(
            name=f"left_limb_{limb_pair_ind}",
            parent_name=parent_name,
            recur=gdict["link-recurrence"] + 1,

            # Symmetrical placement along the Y-axis
            joint_origin_xyz_2=gdict["joint-y-offset"], # Positive Y
```

Figure 6.1: The core code to produce bilateral symmetric creatures.

Reactive Touch Sensors

To overcome the limitations of blind, pre-programmed “swinging” movement, a simple control system was introduced. This was implemented by modifying the Motor class and the main simulation loop. During each simulation step, p.getContactPoints is now used to detect if any of the creature's limbs are in contact with the environment. This "touch" information is then passed to the corresponding motor. If a limb's touch sensor is active, its motor overrides its standard oscillating waveform and instead produces a constant, reactive "push away" force. This enables the creature to feel the ground and push off it, transforming the control system from one of blind wiggling to one of reactive behaviour.

By combining these two features, the GA is no longer just evolving random shapes but is now tasked with finding the optimal limb motion to push itself and a stable, symmetrical body.

```python
for step in range(iterations):
    p.stepSimulation(physicsClientId=pid)

    # this dictionary will store which limbs are touching something.
    # keys will be link indices (0, 1, 2, ...), value will be True/False.
    touch_data = {}
    contacts = p.getContactPoints(bodyA=cid, physicsClientId=pid)
    for contact in contacts:
        # contact[3] is the linkIndex of bodyA (our creature) that made contact.
        limb_index = contact[3]
        touch_data[limb_index] = True

    if (step % 24 == 0) and touched_ground == True:
        self.update_motors(cid=cid, cr=cr, touch_data = touch_data)
```

Figure 6.2: The core code to produce creatures that can “sense” their surrounding.

```python
def get_output(self, touch = False):
    # If touch is True, it will generate a constant "push away" force.
    if touch == True:
        return self.amp * -1.0

    self.phase = (self.phase + self.freq) % (np.pi * 2)
    if self.motor_type == MotorType.PULSE:
        if self.phase < np.pi:
            output = 1
        else:
            output = -1

    if self.motor_type == MotorType.SINE:
        output = np.sin(self.phase)

    return output * self.amp #amp to standard oscillation limbs movement
```

Figure 6.3: The core code to produce creatures that can “push” themselves on contact.

Furthermore, to align the evolutionary search with more plausible animal-like locomotion, two additional constraints were applied exclusively to this Extension Model.

Firstly, the "sphere" limb shape was disabled. Creatures in this model can only evolve limbs with cylinder or box shapes, preventing the evolution of "rolling ball" exploits and promoting the development of leg-like structures.

Secondly, recognising that stable climbing with actual limbs is a more complex and time-consuming task than simple movement, the simulation time for this model was extended to **40 seconds (9600 steps)**. This provides a more appropriate window to evaluate the endurance and its effectiveness.

Figure 6.4: The 1st symmetric creature.

Figure 6.5: The trained bilateral symmetric creature, but without the optimal setting.

7. GA's Output Comparison and Conclusion

The outputs of this project reveal the insight that the most significant advancements for this complex task often lie not in parameter tuning, but in fundamental structure changes. A comparison between the Base Model, Final Tuned Model and the Extension Model highlights this distinction.

Model Configuration	**Generations Run**	**Final Elite Fitness**	**Final Elite Distance (m)**	**Final Elite Link Count**
Base Template	500	(Old Fitness Formula)	10.23 m	10
Final Tuned Model	1000	-3.827701	7.880482 m	16
Extension Model	1000	-4.491736	9.179937 m	9

Table 7.1: Final Performance Summary of Key Models.

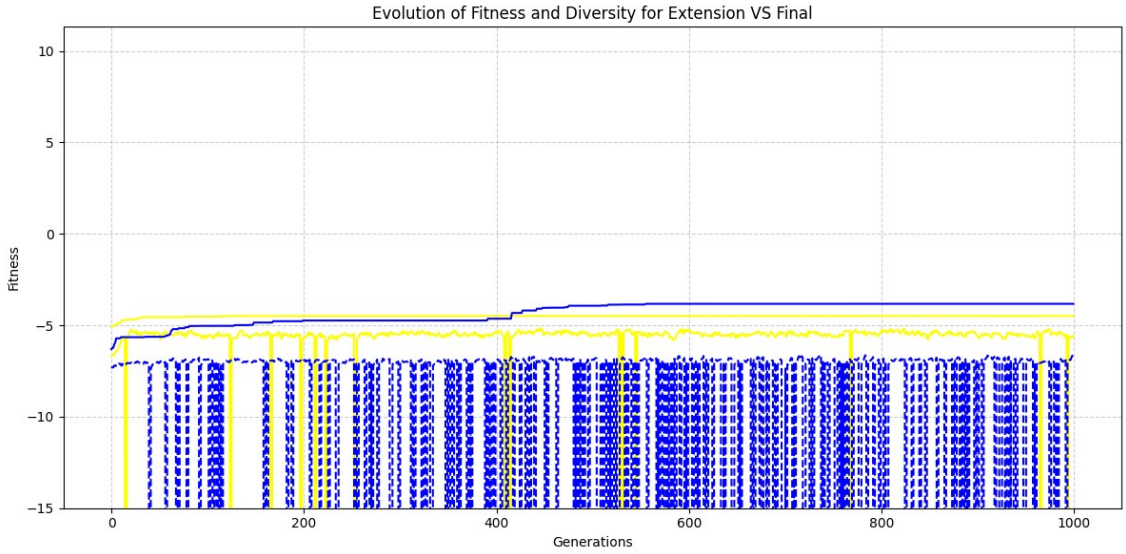


Figure 7.2: The comparison between the Tuned Parameters Model and the Extension Model.

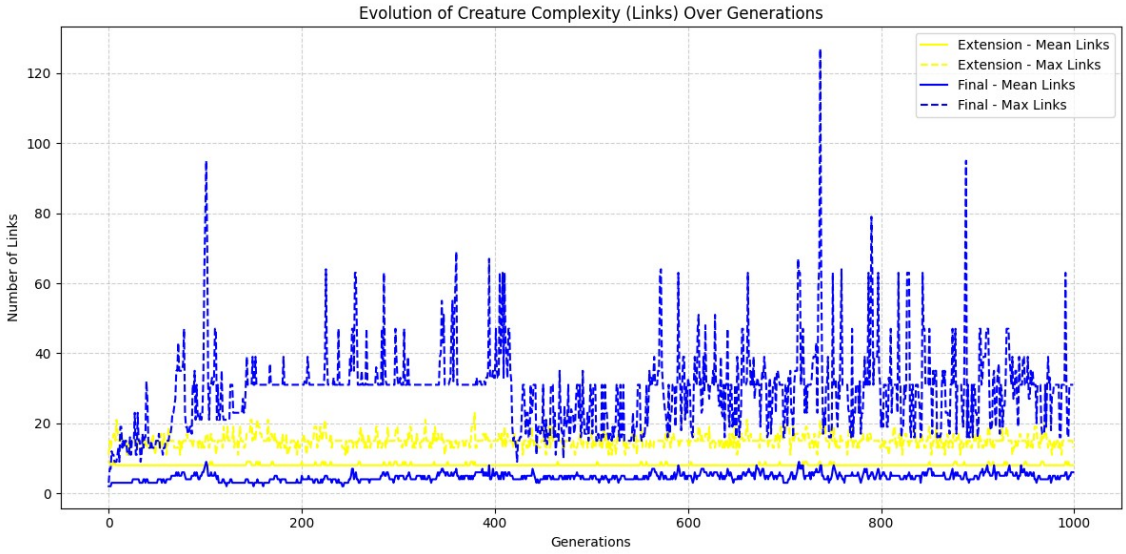


Figure 7.3: Fitness Results of the Extension Model and the Tuned Parameters Model.

The Final Tuned Model, which utilised the optimised parameters, outperformed the other models. It achieved the best numerical result with a fitness of **-3.83** when compared to the Extension Model's -4.49, and came closest to the peak at a distance of **7.88 meters**. However, due to the template's simple motor logic, the model still exhibits "wiggling" motion. Therefore, its only strategy is to increase its reach and size.

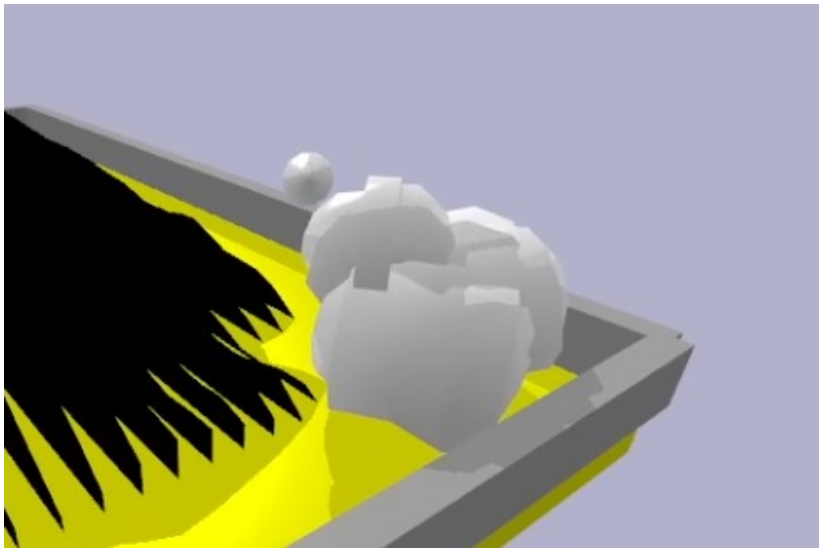


Figure 7.4: The final, trained "Base Template" Creature Design.

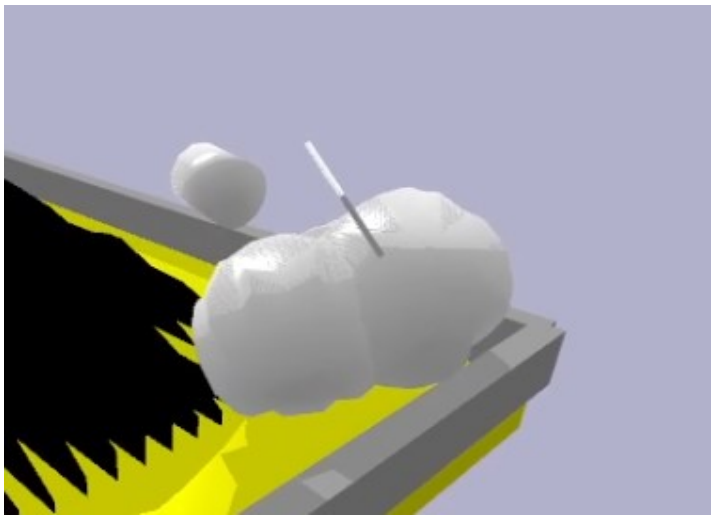


Figure 7.5: The final, trained Tuned-Parameter Creature Design.

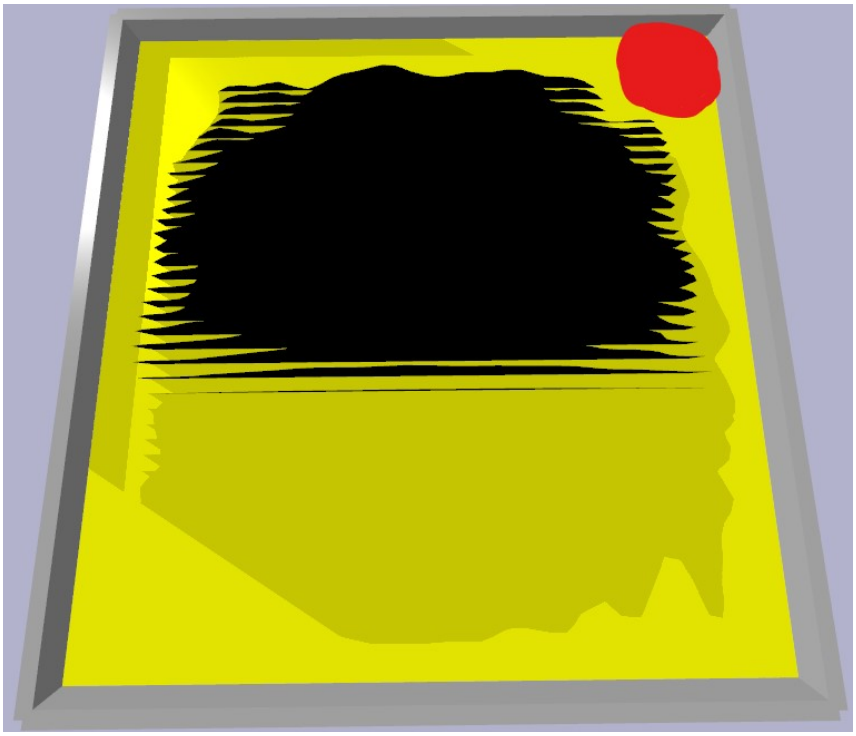


Figure 7.6: The Tuned Model’s creature and Base Model’s creature relative position throughout the demonstration, indicated by the red circle.

In contrast, although the Extension model ultimately achieved a lower performance, its behaviour was found to be more strategic. Instead of attempting a direct vertical ascent, the Extension creature learned to exploit its environment and the fitness function. It discovered that by moving sideways along the mountain's base, a "path of least resistance", it could minimise the horizontal penalty while still improving its score.

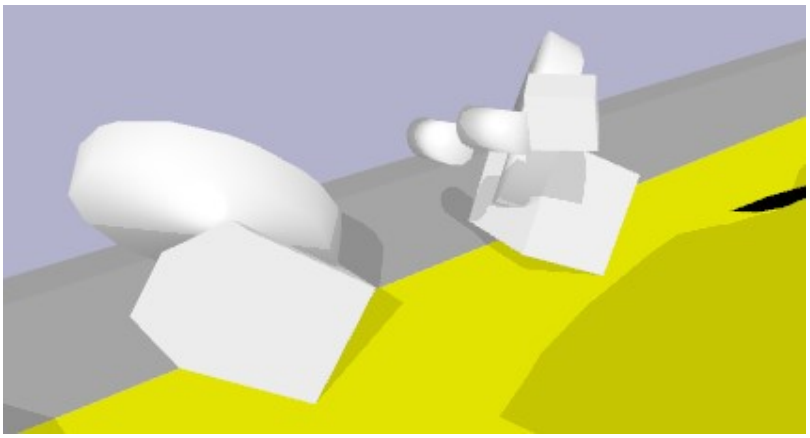


Figure 7.7: The final, trained Symmetric Creature Design.

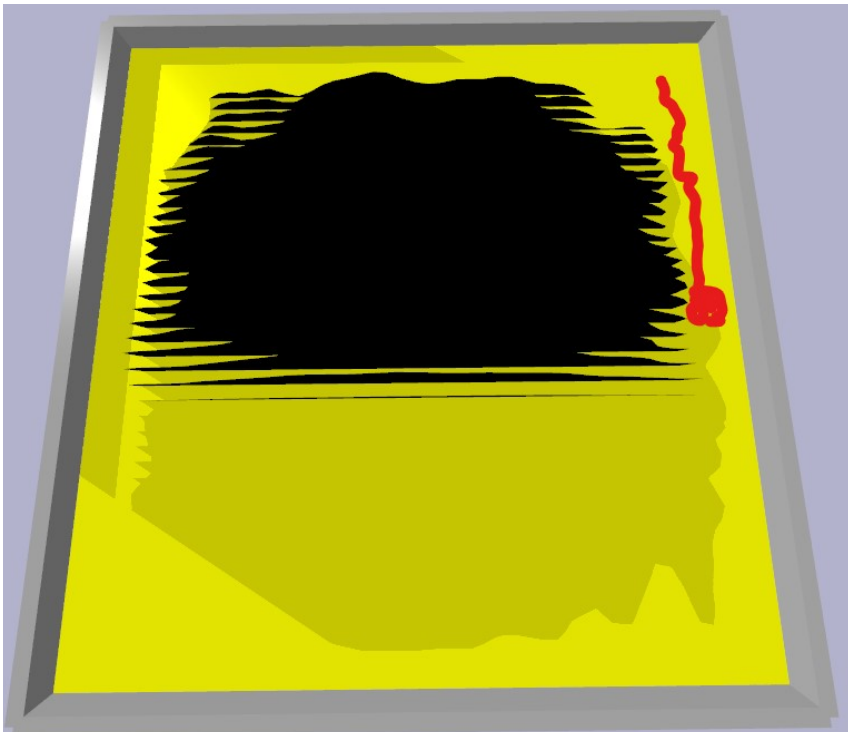


Figure 7.8: The Extension Model’s creature relative position throughout the demonstration, indicated by the red line.

The Extension Model's failure to reach the peak should not be interpreted as an incapable locomotion, but as an adaptation to its training conditions. The creature developed a strategy to exploit the insufficiently planned training environment conditions. The model demonstrated the capacity to "feel" its surroundings and generate purposeful motion, achieving its results through an intelligent, albeit unintended, strategy rather than morphological "cheating."

These outcomes illustrate that while parameter tuning yielded a marginal improvement, it was the fundamental architectural change that demonstrated the potential to break the performance plateau. By enforcing the extension, the GA was tasked with not just creating a shape and modifying its dimensions, but also optimising movement for a body plan capable of strategy.

In hindsight, the project's major limitation was not the evolutionary algorithm, but the training conditions used. If the environment were constrained with walls to prevent lateral movement, and if the fitness function heavily prioritised vertical ascent more, the Extension Model would probably be capable of evolving an effective climbing locomotion strategy.

In conclusion, this project concludes that for complex tasks, designing an effective evolutionary landscape and the creature's fundamental capabilities are far more critical than fine-tuning parameters alone. If given more time, future work would focus on further understanding the code and implementing more bio-inspired structures of an ant (ant’s feelers to feel the surroundings instead, enforcing the rule that each spine must have 1 pair of limbs, etc). Only then, such implementations would evolve a creature that can truly climb the mountain.