Jacob Brown
University of Sussex - Informatics
Computer Science with Artificial Intelligence

"Lizardbot"

A reptile-inspired model of robots optimised to navigate rough terrain

Abstract
Insert abstract here

Submission Year - 2022
Candidate Number - 198732
Project Supervisor - Simon Bowes

Contents

1 Introduction

Add in the intro pretty much directly from the interim report here

2 Project Aims

Why is the robot being modelled instead of physically built?
Why did I choose to use Unity?

2.1 Primary Objectives

2.1.1 Robot Design

Get from interim report - explain overall design and why those decisions were made e.g. simplistic design

2.1.2 Robot Movement

Basic overview of why each component will move the way it does. Tie each point back to how they are founded (or not founded) in natural algorithms.
Include jumping here

2.1.3 Terrain Generation

The terrain will be static - why?
Why will I have three separate terrains? - Octopus
What is the importance of having a smooth terrain?

2.1.4 AI

How will the AI work? Genetic algorithm outline
Dynamic systems theory
Damage minimisation
How do I want the AI to manipulate the relationship between the body and movement?
Why do I want there to be a relationship between the two? - article Simon sent
How will the GA avoid minimas?
Explain triad recombination - what examples are there of this in nature?
Explain about the red robots - why might these be attractive?

2.2 Extension Objectives

2.2.1 Vision

How would a rudimentary visual system reduce damage to the robot?
How would this move the AI from a reactive to proactive mechanism?

2.2.2 Terrain Friction

How do snakes work with different frictions?

2.2.3 Flexible Tail

What are the advantages of having a flexible tail?

3 Project Relevance

3.1 Salamandra Robotica II

Insert from interim report

3.2 Agama Robot

Insert from interim report

3.3 tbc

Find a team that have modelled a robot vs building one

4 Requirements Analysis

Insert from interim report - needs some work
Add section on the constraints of this project

5 Professional and Ethical Considerations

Insert from interim report with more reference to code of conduct

6 Implementation

6.1 Terrain

Three terrains were generated using Procedural Toolkit [Syomus, 2021] to test the performance of the robot across various environments: rough, uneven, and smooth. These categories were inspired by the those used by the Octopus robot [Cianchetti, 2015].

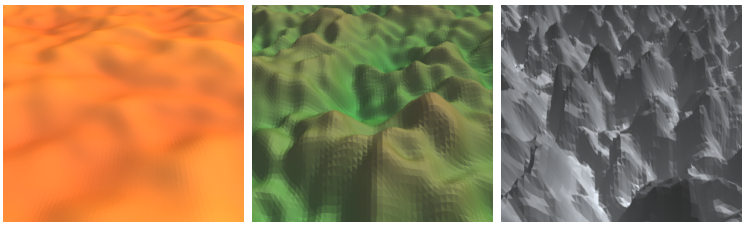

Figure 1: Examples of the three terrain types. (From left to right) Smooth, Uneven, Rough.

At one point the height of the terrain was proportional to the number of sections of the robot, a similar method to that of the Octopus robot. However, as the terrain is a control variable the heights were switched to a static value: $Smooth = 8, Uneven = 16, Rough = 24$.
Overall, the rougher the terrain, the higher and more closed in it is. Most of the development of the robot was conducted on the smooth and uneven terrains, as the rough terrain aims to provide a more extreme environment with which to test the efficacy of the AI.
The smooth terrain is deliberately featureless to test the behaviour of the robot in a simple environment. Herbert Simon provided an elegant example of the importance of this consideration: an ant is observed making its way back to its nest across a beach.
Its route is 'a sequence of irregular, angular segments' that suggests some level of complexity in the ant's behaviour. However, the beach for the ant is a much harsher environment than it is for a human. It is more likely that 'its complexity is really a complexity in the surface of the beach, not a complexity in the ant.' [Herbert, 1996] Thus, the situatedness of the robot could culminate in behaviours that are not of its own making and are instead caused by its relationship with the terrain. The smooth terrain should reduce the role of the environment and allow for emergent behaviours to be prescribed to the robot itself. It is worth noting that the robot is still being tested on three terrains with some common properties (e.g. gravity) and these factors may introduce bias in the AI. This is a reasonable situation as long as applications of the Lizardbot are further modelled on encounterable terrains to allow the AI to adapt the robot accordingly. For proof of concept the sample set of terrains is sufficient.

6.2 Robot

6.2.1 Body

To create a snakelike body, each body module is attached to the previous module by a configurable joint [T, 2020] and has two methods of movement: driving and rotation. The physical design of the robot creates a fluid motion before any complex movement is applied. With the joint structure, the movement of one section is translated to those behind it - similar to dragging a piece of string along the ground. This is shown in figure TODO: the head rotates and, after a delay, creates the same angle in the sections behind it.
The former applies a forward force as determined by the drive velocity parameter whilst the latter applies a velocity to each module using the following equation:

$$\overrightarrow{v_i} = \overrightarrow{v_{i-1}} + \frac{m}{2}\overrightarrow{w}$$

For rotating sections $i = 0, ..., m$, where $m \leq n$, the value of w will be calculated using S or C as specified.

$$S : \overrightarrow{w} = sin\overrightarrow{\theta_{i-1}} + sin\overrightarrow{\theta_i}$$

$$C : \overrightarrow{w} = cos\overrightarrow{\theta_{i-1}} + cos\overrightarrow{\theta_i}$$

This central pattern generator (CPG) approach allows each module to react to the velocity and angle of the previous section. The equation for the CPG originated in Tony Dear's multi-link snake robot: a robot with a similar modular design with passive joints connecting the modules. [Dear et al., 2020] Lizardbot utilises the same math to calculate the velocity with one distinction: Dear's robot split the velocity vector into its axes, using cos for the x axis and sin for the y. Lizardbot instead calculates the vector as a whole and alternates the rotating sections between sin and cos to produce the serpentine motion. For robots with serpentine motion disabled, S and C will be assigned randomly to the rotating modules.

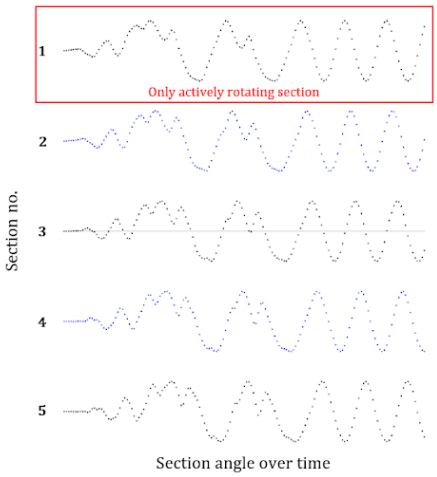

Figure 2: Demonstration of body motion with a single rotating section at the head of the body.

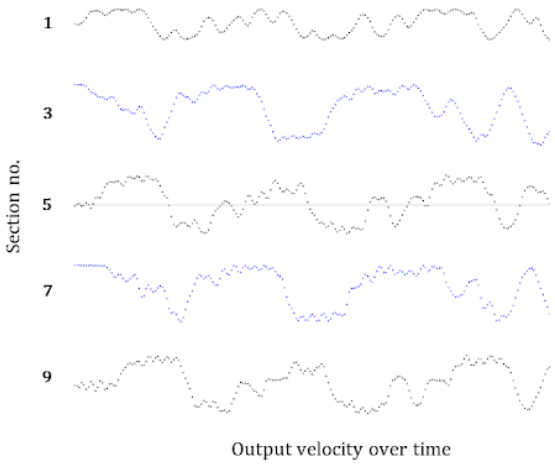

Figure 3: The output velocities generated by each body module with serpentine motion enabled. Modules using S are shown in black, C in blue.

As each section 'reacts' to the previous one whilst using the opposing equation, the velocity applied is almost an inversion of its predecessor. Moving back through the body there appears to be more fluctuation in the values as more noise is introduced through each application of the equation. The advantage of using this recursive approach is the incredibly organic behaviour that it produces. The first prototypes of the project used hardcoded timings and velocities to try and mimic a serpentine motion and the rigidity of the code was evident in the behaviour. With the above equation applied, the motion of the body appears completely natural. As the Lizardbot slithers through troughs in the terrain or wriggles whilst stuck on a ridge, it is easy to forget that it has no awareness of its surroundings. It is simply reacting to the body that came before it.

6.2.2 Tail

The use of a tail has the potential to counterbalance the body and provide stability as the lizardbot moves. The Agama robot used the angular momentum of the body to calculate how to calibrate the tail vertically as the robot jumped. Lizardbot implemented a similar approach in three dimensions by calculating the total momentum of the robot around its centre of gravity (COG) at each frame, and adjusting the velocity of the tail accordingly.

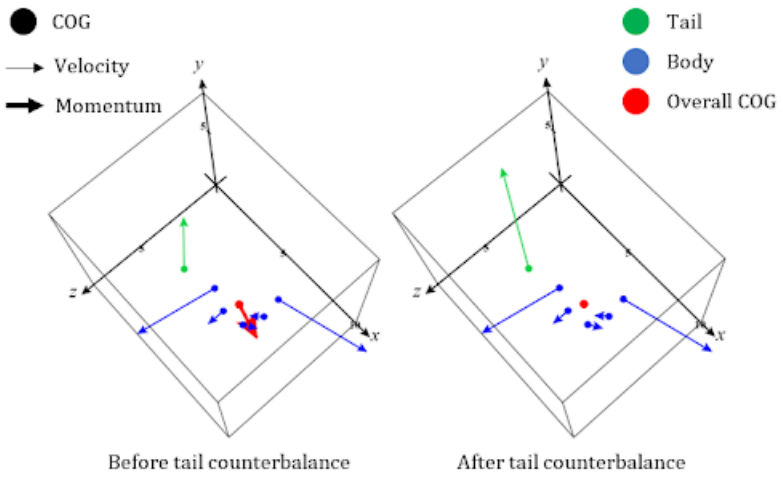

Figure 4: A representation of the tail being adjusted to conserve the angular momentum of a robot. Diagram created using CalcPlot3D [Seeburger, 2021]

For all body parts $i = 0, ..., n$, the radius of the path of motion is the distance from the individual COG x to the COG of the overall robot.

$$r = |x_i - \frac{1}{n} \sum_i^n m_i x_i|$$

The total angular momentum L of the robot is calculated using the above values of r, the mass m and the velocity v of each body part. [Rafay, 2022]

$$L = \sum_i^n r_i m_i v_i$$

To conserve momentum, the velocity of the tail is calculated by inverting L and dividing it by the tail's mass and distance from the overall COG.

$$v_t = -\frac{L}{r_t m_t}$$

Another simplified approach was considered whereby the overall velocity of the robot was counterbalanced instead, however this was found to create sharp changes in the velocity of the tail that could cause it to fling the entire body into the air. Whilst this showed promising behaviour for the basis of a jumping motion, it was counter-productive for a feature whose goal was to stabilise the robot. Additionally, by calculating the magnitude around the COG, any difference in mass between components is taken into consideration. Thus, the tail is able to counterbalance any body structure (assuming that the motion of the tail is not physically blocked by the position of a body part).

The design of the tail assumes that nature has already selected for the optimal location by placing the tail at the back of a creature. This assumption seems intuitive. Most animals, including lizards, are symmetrical and the location of the tail maintains this property, alongside keeping the motion of the tail in the same plane as the rest of the body. For Lizardbot, this assumption could be removed in the future. Since symmetry is not a required property for non-uniform bodies, the tail could be placed anywhere that has equal mass either side of the tail - or placed randomly to see what effect this has on the robot. Who am I to say that a tail cannot be located on the head?

6.2.3 Legs

Legs were added to Lizardbot in an attempt to match the gait of a lizard. As inspired by the design of the Salamandra Robotica [Crespi et al., 2013], each leg is designed to rotate in a circle in order to push the body forward. The physical design of the leg matches the elliptical shape of

the tail and uses the same configuration of customisable mass and length. Currently, there are only two attachment points for a leg on each body module: one each side, perpendicular to the body joints. For a uniform body, the legs are placed symmetrically along the body and given equal length and mass; they are constructed randomly for non-uniform bodies.

The leg rotates around an axis perpendicular to the body section it is attached to, as shown in figure TODO. The size of the circle it follows is determined by the angle it is offset by when attached to the body (α).

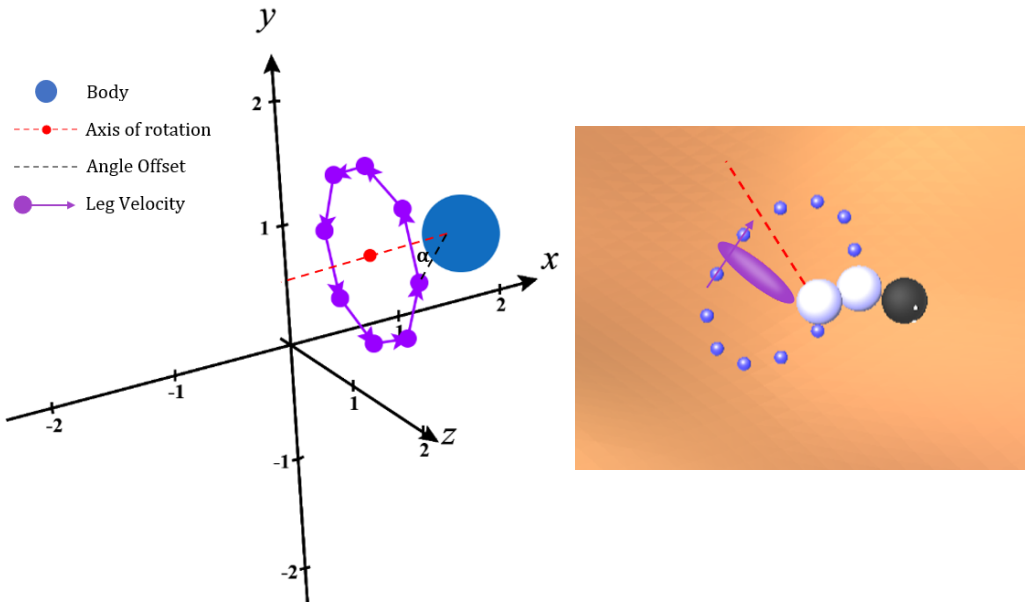

Figure 5: (Left) An illustration of the rotation of a leg around an axis of rotation. Only one attachment point is shown - another would be available on the other side of the body along the x axis. Diagram created using CalcPlot3D [Seeburger, 2021]
(Right) A leg following its path of motion.

The rotation of the leg follows a series of conceptual points spaced 30° apart around the point D (shown by the red point along the axis of rotation in figure TODO). These points are generated using the following equation:

$$P = D + V cos\theta + U cos\theta$$

Where V and U are two vectors perpendicular to each other in a plane through D and the target circle, and $0 \leq \theta < 360$. [K, 2018]

To calculate the desired velocity of the leg, the point P_i closest to the current position of the leg is found. From this, the new velocity can be calculated using the vector to the next point on the circle.

$$v_i = g(P_{i+1} - P_i)$$

If the relevant gene is active, the gait multiplier g increases the velocity when the body is turning away from the leg, decreasing it if the body is turning toward the leg.

The performance of the legs is analysed in section TODO, though it is presumed that with more work the performance could be drastically improved: currently it resembles a lot of twitching. It is possible that an alternative joint mechanism would allow for Lizardbot to stand on its legs and appear more natural. It would also be interesting to explore how the location of the legs

impacts the robot as a whole. Currently the attachment points are based on lizards (indeed, most animals) and the years of evolution that have led to their structure. However, the offer of more variety in the design of the robot may offer alternative solutions to navigate the terrain.

6.3 Artifical Intelligence

A population of Lizardbots can use a genetic algorithm to "evolve" over a series of generations. A genetic algorithm is a form of artificial intelligence that more closely models the characteristics of natural selection. An agent's parameters are broken down into 'genes' that can be manipulated by recombination and/or mutation. The aim of this genetic algorithm (GA) is to improve the performance of the robots over time. The GA is performed when a robot is declared to be stuck by the trapped algorithm.

6.3.1 Performance

The metric used to measure the performance of a robot can heavily influence the outcome of the genetic algorithm. As the desired outcome is a robot capable of navigating the terrain, the base measurement used is the furthest distance (by magnitude) that it has travelled in a given generation from its spawn point within the terrain.

Two additional parameters are used to add context to this base measurement.

The first penalises robots that cause large collisions through a body part. Every time a body part encounters a collision it triggers a method that will measure the force that the body part has experienced. Unity's build-in physics system returns the impulse I of a collision. Using this, the force can be found by dividing the impulse by time: $f = \frac{I}{t}$. [Serlite, 2016] If the force is found to be above a threshold then the robot will be penalised by deducting 10% from the performance. This ensures that the robot is not evolving toward a behaviour that carries it across the terrain efficiently but could be damaging to its structure in the physical world. An example of this is an iteration whereby the robots were observed 'flicking' their tails rapidly downward, causing the robot to be thrown into the air and across the terrain. Whilst this was effective at getting them to the edge of the terrain within seconds, this approach would shatter most robots upon collision with the terrain. This threshold can be tailored to the needs of the robot, such that robots with shielding or soft-bodies robots can be given a higher threshold to allow more risky behaviour.

The second rewards robots that move quickly. The current performance is multiplied by the average speed (distance / time) of the robot since its spawn.

To provide an example of the performance metric, say Robot3 V8 has travelled from (0, 0, 0) to (10, 4, -13) in 21 seconds, and has had two above-threshold collisions. The base performance would be 16.88. Multiplied by its speed, this becomes 13.57. With a 10% deduction for each collision, the final performance is 10.99.
A flaw in this metric is that it does not differentiate between the routes that robots take. If one robot moved quickly but erratically, it may be rewarded equally to one that moved slowly and directly. However, this metric appears to provide a suitable balance between rewarding robots that travel the furthest, further rewarding those that move efficiently, and reducing behaviours that would damage a physical robot.

When referring to the performance of a population, the mean performance of the highest performing 25% is being measured. When a robot is mutated and respawned its performance returns to zero, hence it was found that analysing the entire population added too much noise to the results.

6.3.2 Trapped Algorithm

It is important for the AI to know when the robot is stuck to trigger the termination of the current generation. At this point the AI can mutate the robot before respawning it. This trapped

behaviour can take many forms, from bouncing against the same point in the terrain, circling itself, or looping between the same location(s).

Initially, the maximum - minimum of the baseline performance metric (magnitude of the distance from the origin) over the last t seconds was used. This approach was heavily flawed as it reduced the data set to a single dimension.
Instead, the entire data set over the last t seconds was analysed. The robot was declared stuck when the variance of the robot's 3D coordinates over t seconds converged to 0. This approach correctly identified the robot as stuck but had one major flaw: travelling in a straight line outputs a variance of 0. This behaviour is highly efficient and, whilst it is not something that the AI will train for, is a behaviour that the AI should absolutely not be training against. These false positives prompted a third approach.
The algorithm finds the minimum and maximum values for each axis to draw a conceptual cube around the points the robot has visited in the last t seconds. These cubes depict the worldspace the robot has recently explored. The variance of the data set is calculated for the volume of the cube rather than the coordinates themselves. If the variance of the volumes of the cubes rounds to zero then the robot is considered to be trapped.

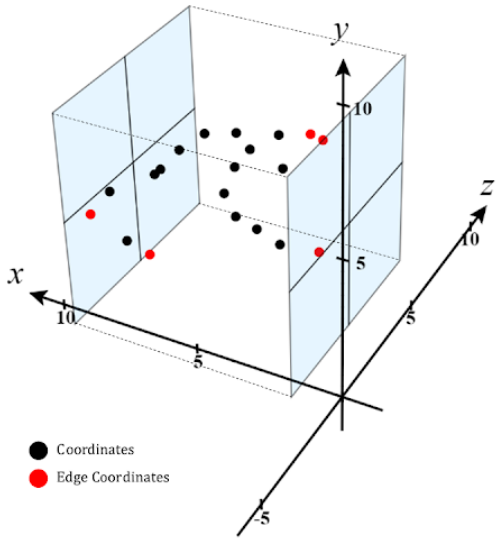

Figure 6: A representation of the cube constructed around the last 20 locations a robot has visited (captured twice a second). Diagram created using CalcPlot3D [Seeburger, 2021]

Finally, the sensitivitiy of the algorithm needed to be determined by adjusting the value of t. The results showed that too low a value of t produced false positives due to it not allowing the robot enough time to turn around. In contrast, if t was too high it took longer for the algorithm to identify when the robot was stuck. This would waste time continuing for an extra few seconds each generation, or could potentially miss instances when the robot became trapped but was able to free itself. A sensitivity of $t = 20$ appeared to provide a balanced value across all three terrains.

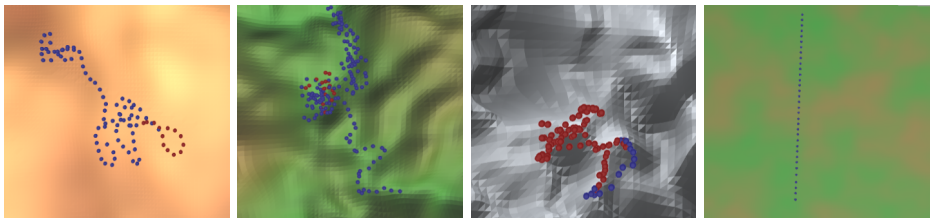

Figure 7: A demonstration of the trapped algorithm using the same robot and $t = 20$ on each terrain.
Each blue point represents the location of the robot being captured. Red indicates that the robot is trapped (and would normally have been disabled and passed to the genetic algorithm).
(From left to right) Smooth, Uneven, Rough, Flat. The latter used a robot with rotation disabled to ensure that a robot travelling in a straight line would not be flagged.

The red points in figure TODO align with behaviours that can reasonably be classed as 'trapped', however it is important to note that this algorithm was implemented with some bias toward certain behaviours (e.g. looping). For this project, the identification of a trapped robot appears sufficient and is expected to produce an AI that will train away from continually exploring the same area for too long. The algorithm has other potential applications in finding looping patterns in any problem that can be assigned a world space. For example, a neural network could be analysed with this algorithm to determine when it is 'stuck' whilst performing a task, or a search algorithm could implement it to avoid excessively exploring within a section of the graph.

6.3.3 Genes

The manipulatable characteristics of a robot are distinguished by a *Gene* class. Each variable is instantiated with a default, minimum, and maximum value. Additionally, they are given a type (established in the enum class *Variable* whereby negative enum values are physical properties and positive are movement) to ensure that the recombination covered in section TODO is combining the same genes with each other.
This class handles any erroneous situations that arise and allows boolean values to be stored as a float. If the *Get* method of a boolean *Gene* is called, then true will be returned for values greater than 0.5. This allows boolean genes to be mutated slowly (e.g. from 0.35 to 0.6) without having to make a single toggle from one value to the other.
When a gene is mutated, if the new value lies outside the valid range then it is 'bounced' (e.g. $max = 0.5, v \to 0.6 \Rightarrow v = 0.4$). If a value outside the max/min values is passed directly then the max/min value would be assigned instead.

6.3.4 Population

To avoid interaction between robots, the body parts of a robot should be capable of colliding between themselves and the terrain but should ignore those of other robots. In Unity, this can be achieved by placing all robot objects into a layer. [no longer trusts SE, 2018] You can then change the Physics system to ignore collisions between different robot layers.
Unity's layers have a limit of 25 available layers. At first, a script was added to every robot scanning around it for other robots in the same layer. Upon detection, the current robot would be moved to another layer that no robots in the vicinity were using. Given that every robot spawned in the same location this script absolutely throttled the performance of the program. With a population of 50 the frame rate would drop from 200fps to 20fps. This is not an acceptable impact so an alternative approach was considered. Multiple terrains are generated and 25 robots placed into each. This has the added benefit of being able to generate a population of robots being tested on all three terrain types simultaneously.

6.3.5 Genetic Algorithm

When the trapped algorithm determines that a robot is stuck it is disabled and passed to the genetic algorithm (GA).

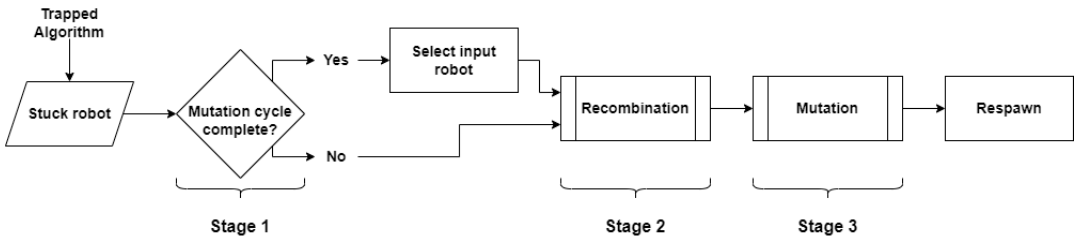

Figure 8: The process followed by the genetic algorithm. Diagram created using draw.io [jgraph, 2022]

The GA first selects which robot should be used as the input for the rest of the process. The default input is the stuck robot, however when a mutatio cycle has been completed this may be overruled. The mutation cycle, exemplified in figure TODO, of a robot refers to how many evolutions should take place before the resultant robot is compared to the robot it initially branched from. When a mutation cycle is complete, the highest performing of the two is selected as the input for the GA. The reasoning behind this is to allow a robot to mutate to a lower performing robot temporarily, as this may allow it to evolve into a more successful robot a few generations later. If the mutation cycle is set to 1 then any single evolution that reduces the performance will be rejected.

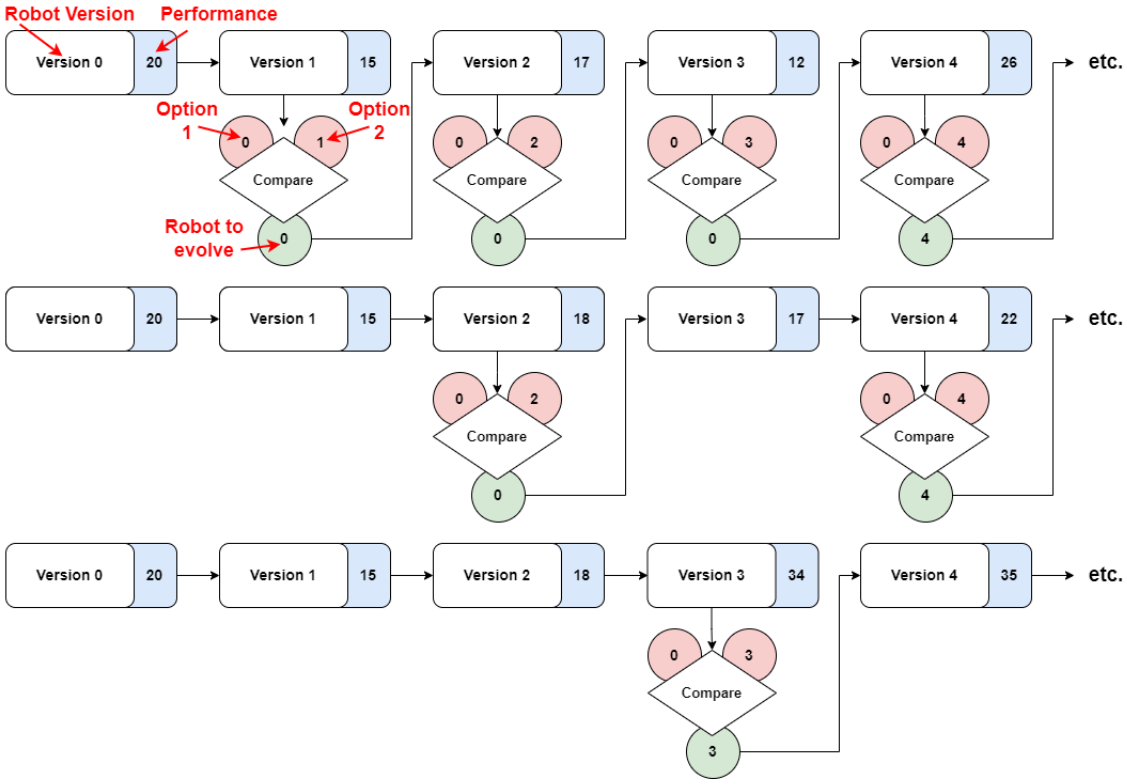

Figure 9: Examples of potential evolutions of a robot. From the top row downwards, the mutation cycle is set to 1, 2, and 3 respectively. Diagram created using draw.io [jgraph, 2022]

Once the input robot has been determined, this robot is then recombined. There are several methods of recombination available but the overall process is: select k robots using a given method and "breed" the genes of one robot in this pool with those of the input robot.
The available recombination methods are:

1. Physical
   The selection pool consists of robots that are physically similar to the input robot. The acceptable margin of difference is increased until k robots are found. This method aims to return robots from the population that have a similar number of each body part, and are matching the input robot as to whether to maintain a uniform body. bullet.

2. Movement
   The selection pool consists of robots that are moving in a similar way to that of the input robot. As above, the acceptable margin of difference is incrementally widened if necessary. The selected robots will be following the same rules for maintaining serpentine motion and the gait of the legs. Additionally, they will have a similar ratio of drive velocity to the number of sections to provide an overview for how powerful the robot is. It allows for differences in the rotation of the body as it is hard to implement a sufficient comparison between them.

3. Triad
   This approach is more artificial than others. Two robots are selected: one each from the physical and movement methods. The genes of these two robots are then recombined with the input robot. As discussed in section TODO, nature does demonstrate examples of more than two agents mating.

4. Lizard

This method aims to mimic a more lizardlike breeding process. A characteristic with no direct bearing on the performance is used to select robots that are nearby the input robot, ignoring those in the population on the other side of the terrain. *Nearby* is relative to the spawn point of the robot, and limited to those in the same terrain type.
The colour of the body was chosen as the desirable trait; Lizardbot loves blue.
In theory, this method should create local 'ecosystems' in the population over time as robots that explore the same area of the terrain each iteration will always interact with the same selection pool.

5. Random
   A random robot is selected from the population, to act as a control method.
6. Any
   Methods 1-4 make an assumption that using the same method for every generation is optimal. This method instead randomly selects one of the above methods for a single generation.

Each of the input robot's n genes is recombined as follows:

$$G(1)_i = R^{[0,1]} < r \longrightarrow \begin{array}{l} R^{[0,1]}{<}0.5 \longrightarrow G(1)_i \\ R^{[0,1]}{\geq}0.5 \longrightarrow G(2)_i \end{array}$$

Where $i = 0, ..., n$. R denotes a randomly generated number in the range $[a, b]$. G(1) refers to the input robot, whilst G(2) is the selected robot. For *Triad* recombination G(2) is randomly chosen from either of the two selected robots, with equal probability.

The recombined robot is then mutated. The genes are divided into physical or movement properties to allow the mutation to be limited to one or the other, or both. As with the recombination, there is also the option for Any mutation, whereby one of the three options is randomly selected every generation. The mutation rate, m, is used to avoid mutating every single gene.
When a gene is mutated its value is adjusted as follows:

$$G_i = R^{[0,1]} < m \longrightarrow (max(G_i) - min(G_i))R^{[0.01,0.1]}G_i$$

The GA contains four major parameters that influence its function: recombination rate, mutation rate, selection size, and mutation cycle. To determine the optimal values for these, 47 iterations were run with randomly generated values and the performance of the population measured. Each iteration contained 50 robots running for 600 seconds. The desired outcome was a performance curve that sloped upwards over time. Peaks and troughs were expected as the successful robots returned to the spawn point; the overall pattern of the performance was key.

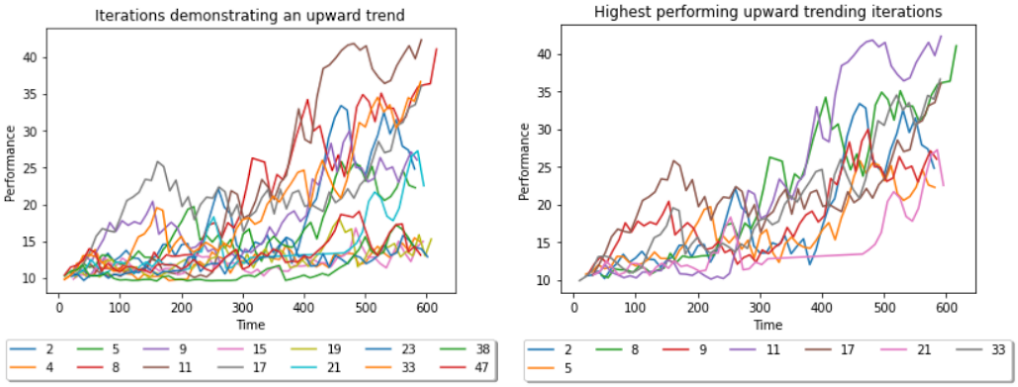

Figure 10: (Left) All iterations demonstrating a promising upward trend.
(Right) Iterations from the left graph filtered to those that reached a performance > 20.
Graphs created using Colab, [Google, 2022] Pandas, [pandas, 2022] MatPlotLib, [matplotlib, 2021] NumPy. [NumPy, 2021]

Upon analysing the four values being used for the highest performing upward-trending iterations it appeared that successful values had been found through an issue in the code. When generating random values, Unity cycles through the same values if not provided with a seed. [Arycama, 2015] As the four values were consistently the first four values generated they had been repeatedly set as $[2, 0.77, 0.31, 9]$. Of the 8 iterations that showed an upwards trend and performed well, 7 were these values. Additionally, there was only a single iteration using these values that did not perform well. Given the apparent success of these GA values, these were selected as the default parameters.

6.4 Dynamic Movement

As outlined in TODO, a dynamic approach uses the historical movement of the robot to 'learn' how to move in a given direction again. Rather than providing rigid rules about how the robot should move, the robot explores how to move in a direction through trial and error. Take a robot that moves left by rolling over. Due to the reactive nature of the algorithms used for movement, it can be assumed that applying the same velocities from the start of the roll would produce another leftward motion. The dynamic movement algorithm saves these velocities for use when a robot needs to move left again.

To achieve this, the worldspace around a robot is divided. "The Fibonacci lattice is a simple way to very evenly distribute points [on the] surface of a sphere". [Roberts, 2020] Using the implementation of the Fibonacci sphere provided by Fnord, [Fnord, 2014] a series of n points were constructed in a sphere around the robot. It was necessary to strike a balance between too low a value of n producing inaccurate results, which could cause the robot to veer off to one side of the intended direction. Meanwhile, too high a value of n would be unnecessarily expensive and result in the majority of the points containing *null* values. $n = 50$ was selected as an appropriate granularity.

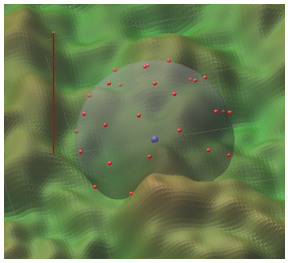

Figure 11: A sphere created using a Fibonacci lattice ($n = 50$) around a centre point shown in blue.

The sphere points are relative to the centre of the sphere, thus acting as vectors in the direction of the point. To track how the robot moved in a direction, every t seconds the velocities of each body part are stored. A conceptual sphere is constructed around the initial position of the robot and rotated to match the rotation of the robot. This way, a point directly to the left of the robot rotated 30 around an axis will be in the same relative location to the robot at 120.
Given another interval of t, a vector is calculated from the initial position to the resultant position. The sphere point closest to this vector is found by comparing the angle between the vector and the points. If there are already values stored for this point then the set of values that moved the robot the furthest distance in the time interval are saved. This process of saving the velocities used to move toward a point is illustrated in figure TODO.

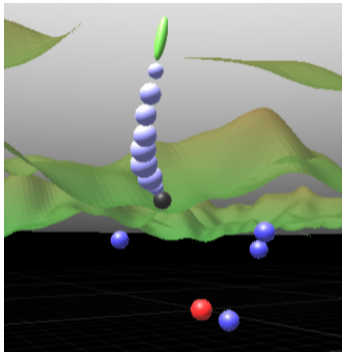

Figure 12: Each blue point represents a set of stored velocities for a robot that moved left before turning right and moving downward.

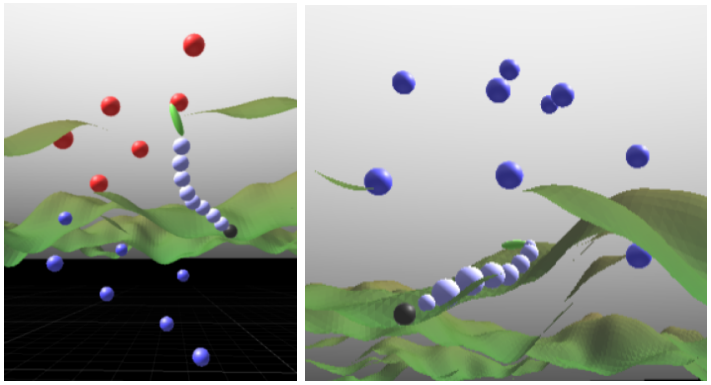

Figure 13: (Left) A wedge of points that would take the robot away from its spawn point. Those shown in red are those remaining after filtering for the height of the terrain.
(Right) An example of the terrain height filtering missing a point.

There are two situations in which the motion of the robot is adjusted:

1. Routine adjustment
   The goal of this adjustment is to keep the robot moving away from its spawn point. Every x seconds the vector between the spawn point and the current position of the robot is calculated. The sphere around the robot is filtered using this vector to output a vertical wedge of points moving away from the origin. Applying any of the velocities saved at these points would theoretically direct the robot further across the terrain.
   The points are further filtered by removing those that have not had any velocities stored yet and those that would move the robot below the level of the terrain. Due to the varying height of the terrain, three measurements are taken at incremental distances from the robot. If the direction of the point at this distance would take it below the height of the terrain then it is removed from the selection. This method aims to take a snapshot of the rough height in a given direction and is liable to miss points that should be eliminated if the peak is between the snapshots.
   From the filtered points, a single one is selected and the velocities applied to the robot.
   Frequent adjustment may affect the ability of the robots to explore new means of movement. To reduce this, the regular adjustment is only enacted if the robot is not already moving toward any of the points in the initial wedge.

2. Scaled adjustment
   This adjustment is used when the robot is found to be stuck by the trapped algorithm.
   At this point it is worth mentioning how the activation of the velocities is implemented. For the routine adjustment the applied velocities will be multiplied by a static activation rate. When a robot is trapped, this activation function is increased exponentially in an attempt to free the robot. Failing that, it is mutated and respawned.

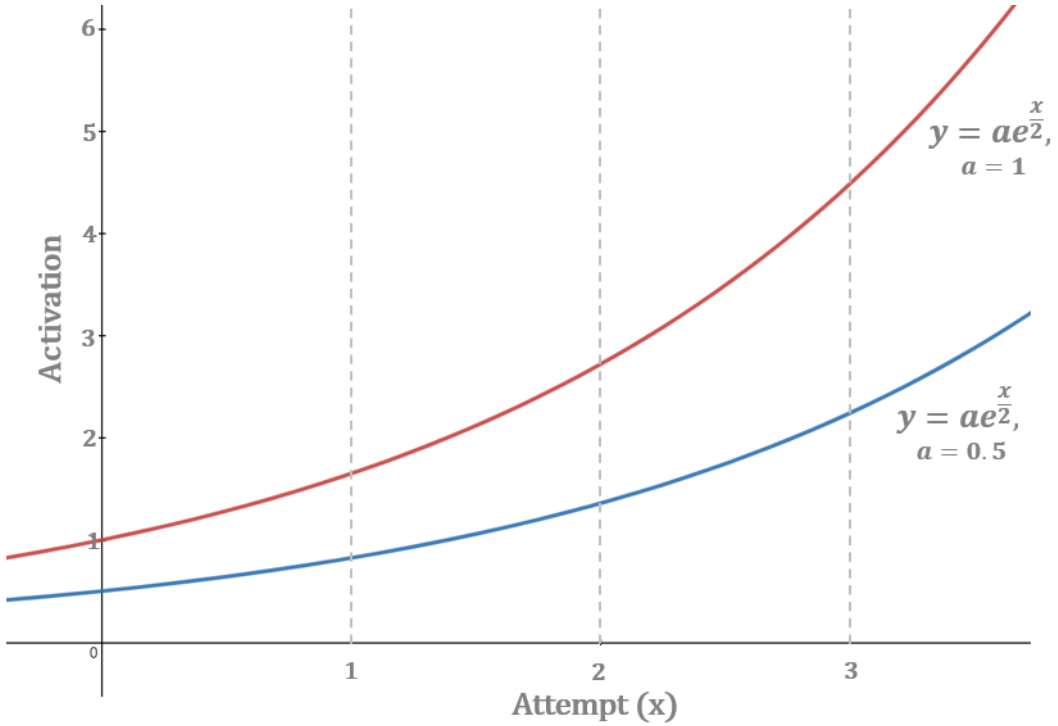

Figure 14: The activation multiplier used for each attempt at scaled adjustment. The static activation rate is shown as α. Graph created using Desmos. [Desmos, 2022]

It is expected that the dynamic movement algorithm will favour robots that are already performing well. All stored velocities are cleared when the robot is mutated, thus the robots will begin "learning" from scratch every time they become trapped. For robots with sufficient data about their movement, the scaled activation will theoretically enable them to progress in the terrain more often, thus reducing the mutation frequency. As they spend longer in the terrain, less successful robots will have longer to recombine with them. It is predicted that the dynamic movement will both improve the performance of any individual robot, and increase the rate of improvement across a population.

6.5 UI

7 Results

How does mutating the body / movement independently work?
How does the addition of the tail help?
How does initiating the body with a serpentine motion established affect the outcome?
What happens when the body is set as static?
What is the outcome when the legs are out of sync from the body?
Starting with defaults, what parameters does the AI mutate to? What is the corresponding behaviour for this?
Does the AI converge on the same parameters when started with different defaults?

Whilst the GA demonstrates promising results, there are still some fundamental issues with it. A major issue is that not all genes have equal weight. If the gene that determines whether a uniform body is maintained mutates from false to true then this has a significant effect on the robot. The physical properties of almost all body and leg parts will be adjusted, resulting in a much larger adjustment to the robot than those caused by mutating other genes. To mitigate this, it would be useful to apply weights to each gene. Thus, the mutation range could be scaled accordingly, or the genes categorised such that a change to a heavily weighted gene would limit any changes to other genes. This would allow an isolated analysis of the effect that such a large mutation has on the performance.
It would be useful to have different GA parameter values tailored to each combination of the recombination and mutation methods. Currently, effective values have been found whilst the Any option was selected for both in an attempt to find parameters suitable for all permutations. However, this may favour certain combinations more than others and there is currently no data on this. Rather, these base values are assumed to be appropriate and the effectiveness of the methods themselves judged relative to them. Ideally, there would be a set of values for each combination, or the ability for the GA parameters to themselves be mutated.
There is an assumption that asynchronous evolution is superior to synchronous evolution. The alternative would be to allow every robot to operate for x seconds before mutating a percentage of the population in parallel. This would ensure that every robot experiences the same number of generations and remove any assumptions in the algorithm used to declare a robot as trapped. Intuitively, the current approach is more appropriate as it does not leave unsuccessful robots continuing past the point where they have been identified as trapped. It is also more natural: we do not wait for a person's 30th birthday before allowing them to have children based on how successful they have been thus far in their lives. Certainly it is more efficient, as waiting for every robot to be declared stuck before mutating a portion of the population would be extremely time consuming. However, it would be useful to gather results of the alternative approach to support this intuition.
TODO anecdotal evidence of complex behaviour emerging

The dynamic movement could be further improved by performing a more comprehensive analysis of the entire terrain around the robot and making a more informed decision about the direction the robot should move in. This could further prevent the robot from becoming trapped as it avoids higher ground or uses a larger activation to "jump" onto higher ground.
Additionally, the algorithm currently makes assumptions about the knowledge the robot has about the world around it. It is assumed that there are sensors capable of detecting the rotation, relative position from the origin, velocities of each body part, and of detecting the height of the terrain. It is not immediately obvious how to offer this functionality without the use of these measurements but an exploration into alternative methods may be valuable if the program were to be applied to a physical robot.

8 Conclusion

9 References

Primary References

Cianchetti, M. (2015). Bioinspired locomotion and grasping in water: the soft eight-arm octopus robot. *Bioinspiration & Biomimetics*.
https://iopscience.iop.org/article/10.1088/1748-3190/10/3/035003
Accessed: 2022-04-10.

Crespi, A., Karakasiliotis, K., Guignard, A., and Ijspeert, A. J. (2013). Salamandra robotica ii: An amphibious robot to study salamander-like swimming and walking gaits. *IEEE*.
https://ieeexplore.ieee.org/document/6416074
Accessed: 2022-04-12.

Dear, T., Buchanon, B., and Abrajan-Guerrero, R. (2020). Locomotion of a multi-link non-holonomic snake robot with passive joints. *The International Journal of Robotics Research*.
http://dx.doi.org/10.1177/0278364919898503
Accessed: 2022-04-11.

Herbert, S. (1996). The sciences of the artificial. *MIT Press*, page 51.
https://monoskop.org/images/9/9c/Simon_Herbert_A_The_Sciences_of_the_Artificial_3rd_ed.pdf
Accessed: 2022-04-11.

Roberts, M. (2020). How to evenly distribute points on a sphere more effectively than the canonical fibonacci lattice. *Extreme Learning*.
http://extremelearning.com.au/how-to-evenly-distribute-points-on-a-sphere-more-effectively-than
Accessed: 2022-04-12.

Software References

Desmos (2022). Graphing calculator - desmos. *Desmos*.
https://www.desmos.com/calculator
Accessed: 2022-04-12.

Google (2022). Colaboratory. *Google*.
https://colab.research.google.com/?utm_source=scs-index
Accessed: 2022-04-12.

jgraph (2022). drawio-desktop. *GitHub*.
https://github.com/jgraph/drawio-desktop/releases/tag/v17.4.2
Accessed: 2022-04-12.

matplotlib (2021). Matplotlib 3.5.1. *matplotlib*.
https://matplotlib.org/stable/index.html
Accessed: 2022-04-12.

NumPy (2021). Numpy 1.22.0. *NumPy*.
https://numpy.org/
Accessed: 2022-04-12.

pandas (2022). Pandas 1.4.2. *pandas*.
https://pandas.pydata.org/
Accessed: 2022-04-12.

Seeburger, P. (2021). Calcplot3d. *LibreTexts*.
https://c3d.libretexts.org/CalcPlot3D/index.html
Accessed: 2022-04-11.

Syomus (2021). Proceduraltoolkit. *GitHub*.
https://github.com/Syomus/ProceduralToolkit
Accessed: 2022-04-10.

Code References

Arycama (2015). Random.range always generates the exact same numbers in start(). *Unity Forums*.
https://answers.unity.com/questions/1072318/randomrange-always-generates-the-exact-same-number.html
Accessed: 2022-04-12.

Fnord (2014). Evenly distributing n points on a sphere. *StackOverflow*.
https://stackoverflow.com/questions/9600801/evenly-distributing-n-points-on-a-sphere
Accessed: 2022-04-12.

K, D. (2018). Plot points around circumference of circle in 3d space given 3 points. *StackExchange*.
https://math.stackexchange.com/questions/3007243/plot-points-around-circumference-of-circle-in-
Accessed: 2022-04-12.

no longer trusts SE, D. (2018). unity - how two objects with rigidbody can pass through each other? *StackOverflow*.
https://stackoverflow.com/questions/48461267/unity-how-two-objects-with-rigidbody-can-pass-thro
Accessed: 2022-04-12.

Rafay, K. (2022). Angular momentum calculator. *Omni Calculator*.
https://www.omnicalculator.com/physics/angular-momentum
Accessed: 2022-04-11.

Serlite (2016). Getting collision contact force. *StackOverflow*.
https://stackoverflow.com/questions/36387753/getting-collision-contact-force
Accessed: 2022-04-12.

T, V. (2020). Luna tech series: A deep dive into unity configurable joints. *Luna Labs*.
https://medium.com/luna-labs-ltd/luna-tech-series-a-deep-dive-into-unity-configurable-joints-96
Accessed: 2022-04-11.

tomvds (2008). Euler, quaternions, radians, degrees... huh? *Unity Forums*.
https://forum.unity.com/threads/euler-quaternions-radians-degrees-huh.53407/
Accessed: 2022-04-11.

10 Appendices

10.1 Code of Conduct