Lunar Lander

Project #2 for Artificial Intelligence Fundamentals

Francisco Silva, Miguel Pereira and Guilherme Rodrigues
Nº 2022213583, 2022232552 and 2022232102

Department of Informatics Engineering
University of Coimbra, Coimbra, Portugal
{francisco.lapamsilva,miguelmpereira0409,gfr04}@gmail.com

May 18th, 2025

Contents

1 Methodology

1.1 Perceptions

We utilize the following perception variables to determine the lander's state:

Symbol	Description
x	Horizontal position (0 at platform center)
y	Vertical position (0 at platform surface)
Vx	Horizontal velocity (positive rightward)
Vy	Vertical velocity (positive upward)
A	Angular direction (positive counter-clockwise)
Va	Angular velocity (positive counter-clockwise)
L	Left leg ground contact (1 if touching)
R	Right leg ground contact (1 if touching)

Table 1: Perception Variables and Descriptions

1.2 Actions

To control the it's descent, the Lander uses two motors: main motor (mM) and lateral motors (lM), whose values range from 0.0→1.0 and -1.0→1.0. These two controls are represented and controlled by the two output neurons, as seen in the "network" function. When the mM's value is greater than 0.5, the motor is on. When the lM's value is lesser than -0.5, the left motor activates, and when it is greater than 0.5, the right motor activates.

Symbol	Description
mM	main Motor
lM	lateral Motor

Table 2: Perception Variables and Descriptions

1.3 Objective Function

The most important mechanism of our evolution is the Objective Function. It is also the part that has undergone the biggest change throughout development. Our Objective Function is designed to evaluate the lander and guide it to a successful landing by rewarding stability, position, and landing.

We started with the simple goal of stabilizing the lander. The first version of the OF looks like this:

```python
def objective_function(observation):
    fitness = 0

    x = observation[0]
    y = observation[1]
    x_speed = observation[2]
    y_speed = observation[3]
    angle = observation[4]
    angle_speed = observation[5]
    left_leg_touching = observation[6]
    right_leg_touching = observation[7]

    #stability
    stable_angle = (25-(angle)**2) / 25
    stable_angle_speed = int(abs(angle_speed) < 1.5)

    stable_horizontal_speed = int(abs(x_speed) < 0.1)    # Neste caso
        melhor discreto
    stable_vertical_speed = (10-(y_speed+0.2)**2) / 10   # Offset de -0.2 (
        velocidade desejada de -0.2)

    stability = (
        500* stable_angle +
        400* stable_angle_speed +
        200* stable_horizontal_speed +
        300* stable_vertical_speed
    )



    fitness = (
        stability
    )

    return fitness, check_successful_landing(observation)
```

Some noteworthy aspects of this code are our early decision to make each rule return values normalized between 0 and 1 that are then multiplied by a weight to balance the importance of different rules, and the utilization of constant functions instead of discrete, to better guide the lander to the desirable state smoothly.

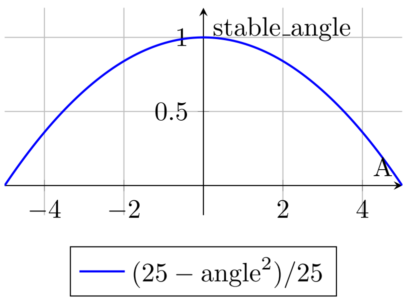

Figure 1: stable_angle

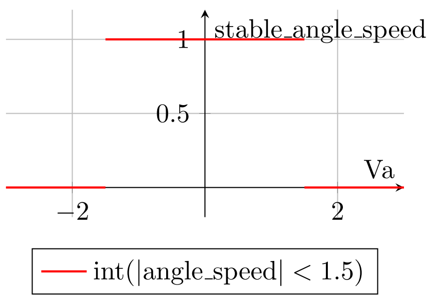

Figure 2: stable_angle_speed

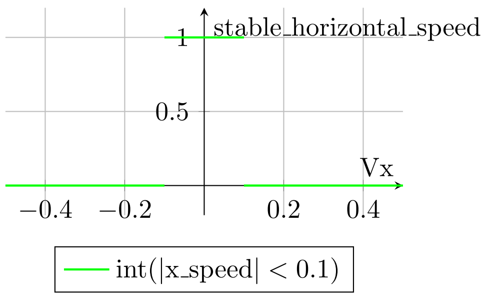

Figure 3: stable_horizontal_speed

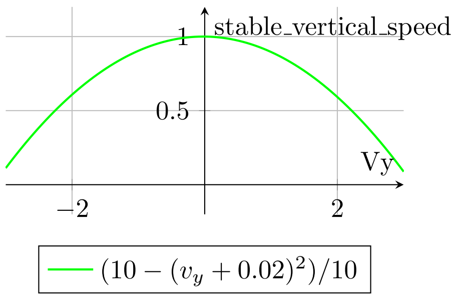

Figure 4: stable_vertical_speed

As shown in Figure 4, in the stable_vertical_speed function, we added a small offset of −0.2 to encourage the lander to descend at a speed of −0.2, which is the desired speed for landing. This adjustment helps prevent the lander from stopping completely or ascending. This allows us to not have to add a separate rule to encourage the lander to descend, as it is already being done by the stable_vertical_speed function.

This version of the code generated a peak landing success rate of 24%, but the evolution process is still very unreliable, and in some tests it remains 1%. This is a problem that will only be resolved in the final version of the code.

Having the stability taken care of, we entered the phase of development. We added a metric to guide the positioning and landing. The code added looks like this:

```
def objective_function(observation):
    fitness = 0

    # ------ existing observations ------- #

    # ---- existing stability rewards ---- #

    positioning_x_centered = (1-(x)**2) / 1
```

```
positioning = (
    40* positioning_x_centered
)


# maximises fitness when both legs are touching, and x==0, using a quadratic function to center the lander
landing = (
    50 * (int(left_leg_touching and right_leg_touching) * ((1-(x)**2)/1))
)



fitness = (
      stability
    + positioning
    + landing
)

return fitness, check_successful_landing(observation)
```

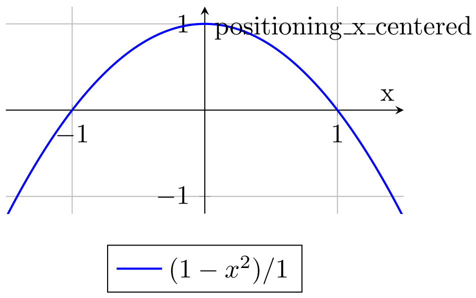

Figure 5: positioning_x_centered

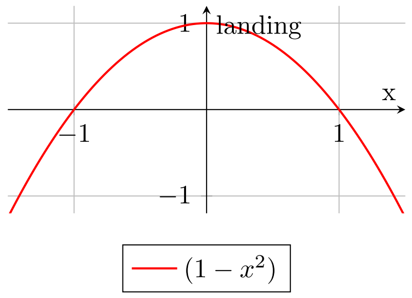

Figure 6: landing (when both legs touch)

This second version could achieve a maximum of 80% success rate, but was also plagued by unreliability in the training process, and often resulted in agents that could not surpass 14% success rate.

In the final stage of development, we tweaked the weights multiplied by each rule, in order to balance the importance of each rule. To resolve the unreliability of the evolution, we decided then to tweak the mutation constants to make the agents try to explore more, and to not get stuck in local minima. By increassing "STD_DEV" from 0.1 to 0.5, we allow the agents to explore more agressively, which finally allows them to reliably reach a success rate of 99% in only 100 generations. We also increased the "EVALS" from 1 to 5, which makes the simulation take longer, but allows us to more accurately evaluate the fitness of each agent. The final OF looks like this:

```
# ...
STD_DEV = 0.5

EVALS = 5
# ...

def objective_function(observation):
    fitness = 0

    x = observation[0]
    y = observation[1]

    x_speed = observation[2]
    y_speed = observation[3]

    angle = observation[4]
    angle_speed = observation[5]

    left_leg_touching = observation[6]
```

```
        right_leg_touching = observation[7]


        stable_angle = (25-(angle)**2) / 25
        stable_angle_speed = int(abs(angle_speed) < 1.5)
        stable_horizontal_speed = int(abs(x_speed) < 0.1)    # Neste caso e
            melhor discreto
        stable_vertical_speed = (10-(y_speed+0.01)**2) / 10  # Offset de
            -0.01 (velocidade desejada de -0.01)

        stability = (
            50* stable_angle +
            40* stable_angle_speed +
            40* stable_horizontal_speed +
            30* stable_vertical_speed
        )

        positioning_x_centered = (1-x**2) / 1
        positioning = (
            40* positioning_x_centered
        )

        landing = (
            50 * (int(left_leg_touching and right_leg_touching) * ((1-(x)
                **2)/1))
        )


        fitness = (
              stability
            + positioning
            + landing
        )

        return fitness, check_successful_landing(observation)

    # ...
```

Finally, this version reliably achieves a success rate of over 90%, often reaching 99%, demonstrating the necessity of increassing the mutation constants to allow the agents to explore more. Even with these simple rules, the lander is able to learn to land in a stable position. Looking at the graph of the evolution of the fitness we see the lander quickly learns to land, in only 100 generations, reaching the maximum fitness of 300.0 in 72 generations.

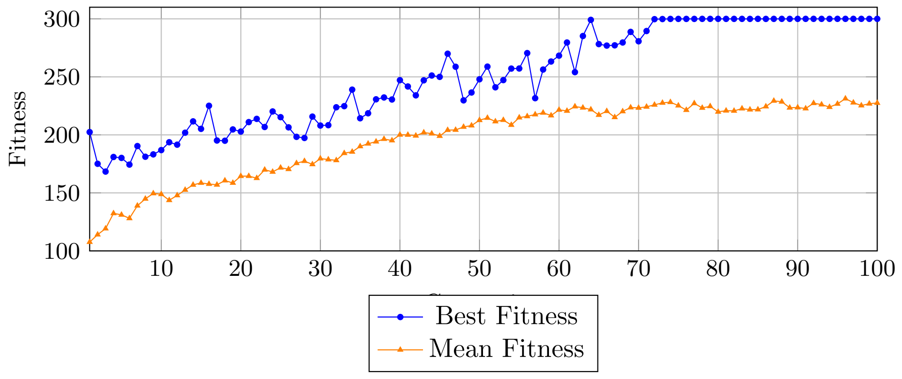

Figure 7: Evolution of fitness over generations.

1.4 Genetic Operators

1.4.1 Parent Selection Methods

Two different strategies for selecting parents were implemented and can be interchanged through the `PARENT_SELECTION` variable.

Fitness-Proportionate Selection (Roulette Wheel) The `parent_selection` function selects individuals with a probability proportional to their fitness. The total fitness of all positively fit individuals is calculated and a random probability is drawn. The algorithm iterates through the population, subtracting each individual's fitness contribution from the probability until one is selected. If the total positive fitness is zero or negative, a random individual is returned to avoid failure. This method ensures that individuals with higher fitness are more likely to be chosen while still giving a chance to less fit individuals.

Tournament Selection The `parent_selection_tournament` function implements a simple tournament selection: two individuals are randomly selected from the population, and the one with the highest fitness is chosen as the parent. This method introduces selective pressure by favoring the fitter of two candidates but without requiring any normalization of fitness values.

Current Strategy The evolutionary process currently uses the `tournament selection` strategy as the active method, set by:

```
PARENT_SELECTION = parent_selection_tournament
```

This can be easily changed to the fitness-proportionate version if desired, providing flexibility to test different selection dynamics. We implemented and tested both approaches and observed that while tournament selection consistently yielded better results, the fitness-proportionate method also performed well in practice. For this reason, we decided to retain both implementations to allow further experimentation and adaptability in different scenarios.

1.5 Crossover

The `crossover` function is responsible for generating a new individual (offspring) from two parent individuals, using a single-point crossover method. This is a standard technique in genetic algorithms where a random point is chosen along the genotype vector, and the offspring inherits genes from one parent up to that point and from the other parent after that point.

The function starts by randomly selecting a crossover point between 1 and `GENOTYPE_SIZE - 1`, ensuring that both parents contribute at least one gene to the offspring. Then, it randomly chooses which parent will contribute the first part and which one will contribute the second. This is done by generating a random number and comparing it with 0.5.

Depending on the order chosen, the function constructs the offspring's genotype by taking the first part from one parent and the second part from the other using list slicing. The result is a new genotype that is a mixture of the genetic material of both parents.

Finally, the function returns a new individual represented as a dictionary with the newly generated genotype and a `None` fitness value, indicating that the individual has not yet been evaluated.

This implementation introduces genetic diversity while preserving genetic information from the parents, which is essential for effective exploration of the solution space.

1.6 Mutation

The `mutation` function is responsible for introducing random genetic variation into an individual. This step is essential in genetic algorithms, as it helps maintain genetic diversity within the population and prevents premature convergence to suboptimal solutions.

The mutation process iterates through each gene in the individual's genotype. For each gene, a random number between 0 and 1 is generated and compared to a predefined mutation probability threshold, given by the constant `PROB_MUTATION`. If the generated number is less than the threshold, the gene is mutated.

Mutation is applied by adding a random value to the gene, sampled from a Gaussian (normal) distribution with a mean of 0 and a standard deviation defined by the constant `STD_DEV`. This method tends to introduce small perturbations, promoting a smoother and more controlled evolutionary process.

The final result is a new individual whose genotype has been slightly modified while maintaining its general structure. This operation enables the exploration of new regions in the solution space that would not be reached by crossover alone. Therefore, mutation plays a fundamental role in balancing exploration and exploitation in evolutionary algorithms.

1.7 Survival Selection

The `survival_selection` function implements an elitist survival strategy. It starts by sorting the `offspring` list in descending order of fitness, ensuring the best individuals are at the front.

Then, the elite individuals from the current population (the top `ELITE_SIZE` individuals) are re-evaluated using the `evaluate_population` function. This accounts for potential changes in evaluation due to stochasticity in the fitness function.

These re-evaluated elite individuals are combined with the rest of the offspring (excluding the worst `ELITE_SIZE` individuals) to form the new population.

Finally, the new population is sorted again by fitness in descending order, and returned. This ensures that the best individuals are preserved and that the population is always composed of the fittest available solutions.

1.8 Evolution

The `evolution` function encapsulates the entire evolutionary process. It begins by creating multiple parallel evaluation processes to improve efficiency during fitness computation.

Next an initial population is generated and evaluated. The best individual is stored to track the algorithm's progress across generations.

For each generation, the algorithm generates offspring through either crossover (with probability `PROB_CROSSOVER`) or direct parent copying. Parents are selected using the active selection strategy and all individuals are then mutated.

Once the offspring population reaches the predefined size, it is evaluated in parallel. After evaluation, the `survival_selection` function is used to determine which individuals survive to the next generation, ensuring elitism is preserved.

The best individual of each generation is recorded alongside success rate and elapsed time for performance monitoring.

At the end of all generations, all evaluation processes are terminated, and the list of best individuals per generation is returned.

1.9 Algorithm Parameters

- Population size: 100 individuals
- Generations: 100
- Crossover probability: 50%
- Elite size: 1 individual

2 Experimentation and Results

2.1 Milestone 1: Windless Environment

Performance comparison of tested configurations:

Experiência	Mutação	Crossover	Elitismo	Tamanho da População	Gerações
1	0.008	0.5	0	100	100
2	0.05	0.5			
3	0.008	0.9			
4	0.05	0.9			
5	0.008	0.5	1		
6	0.05	0.5			
7	0.008	0.9			
8	0.05	0.9			

Table 3: Configuration of the experiments for different parameters.

Experiência	Hitrate	Score Médio (/250)
1	93.3%	245.73
2	99.7%	249.98
3	93.8%	244.94
4	85.3%	241.87
5	96.8%	248.01
6	97.3%	248.63
7	82.1%	239.51
8	97.2%	247.99

Table 4: Results of the experiments with different parameters.

2.2 Milestone 2: Windy Environment

In order to adapt the algorithm to the windy environment, we made the stability rules weight more, something that is easy to do with the way we built the code:

```python
    def objective_function(observation):

        # ...

        stable_angle = (25-(angle)**2) / 25

        stable_angle_speed = int(abs(angle_speed) < 1.5)

        stable_horizontal_speed = int(abs(x_speed) < 0.1)    # Neste caso melhor discreto

        stable_vertical_speed = (10-(y_speed+0.01)**2) / 10  # Offset de -0.01 (velocidade desejada de -0.01)

        stability = (

            50* stable_angle +

            40* stable_angle_speed +

            40* stable_horizontal_speed +

            30* stable_vertical_speed

        ) * 1 + (int(ENABLE_WIND) * 10)

        # ...

        fitness = (

                stability

            + positioning

            + landing

        )

        return fitness, check_successful_landing(observation)

    # ...
```

The way this weight is implemented, allows us to easily turn it on and off, and to adjust the weight of the wind penalty.

Even this trivial implementation allows us to achieve a success rate of 86%, which is significantly better than the 24% we achieved in the first version of the code.

3 Conclusions

We conclude that our approach on a neural network to land a lander on the moon was great although further improvements could be made to account for the presence of wind.

3.1 Objective Function Design

The sucess of the lander heavily depends on the objective function. Its careful tuning while balancing weights for stability, positioning, and landing were crucial. Using continuous reward functions (e.g., quadratic functions for angle and position) proved to be more effective than discrete thresholds in guiding the lander towards optimal behavior.

3.2 Parent Selection

Tournament selection outperformed fitness-proportionate selection in consistency. Retaining both methods allowed for flexibility, but tournament selection was enough for this problem.

3.3 Wind Adaptation

When introducing wind to the environment it required rewarding more horizontal stability, showing the need for environment-specific reward shaping.