Languages and Algorithms for Artificial Intelligence
Report Project Module 1

Riccardo Falco

Academic Year 2021-2022

1 Introduction

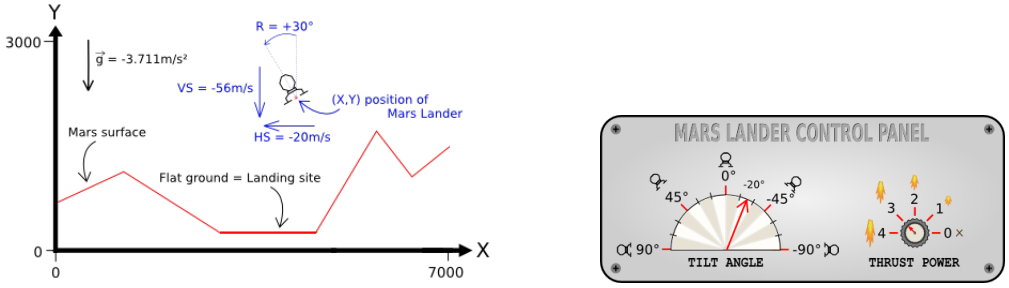
I propose in this project a solution for the coding game challenge Mars Lander - Episode 2using SWI Prolog. The goal of this puzzle is to safely land the spaceship on the platform.

This puzzle is the second level of the "Mars Lander" trilogy. The controls are the same as the previous level but you must now control the angle in order to succeed. In the Section 2 I will explain the rule of the game.

2 Rules

The Mars Lander will be deployed in a well defined and limited region. The zone indeed is of 7000m wide and 3000m high. Note that inside the zone there must be a **unique area of flat ground** which is at least of 1000m wide. This will be the so called *Landing Site*, where the Mars Lander can land. For the sake of simplicity an example image is provided

At each second depending on the current conditions of the lander, the program must provide the new desired *tilt angle* $[-90°, 90°]$ and the new thrust power that goes from 0 to 4.



(a) *Game example* (b) *Control Panel*

Figure 1: (*a.*) Example of game instance. The zone is 7000m wide and 3000m high. There is a unique area of flat ground on the surface of Mars, which is at least 1000m wide. (*b.*) Angle goes from $-90°$ to $90°$. Thrust power goes from 0 to 4.

The game simulates a free fall without atmosphere with Mars Gravity equal to $3.711 m/s^2$. Moreover note that for a **thrust power** of x, an equivalent push force of x m/s^2 is generated and x **liters of fuel** are consumed.

Thus, a thrust power of 4 in almost vertical position is needed in order to compensate the gravity on Mars,

2.1 Landing constraints

For a landing to be successful, the ship must: *i)* land on flat ground, *ii)* land in a vertical position (tilt angle = 0°), *iii)* vertical speed must be limited ($\leq 40 m/s$ in absolute value), *iv)* horizontal speed must be limited ($\leq 20 m/s$ in absolute value)

2.2 Initialization input

An initial number `surfaceN` is provided. It corresponds to the number of points for drawing the surface of Mars, given linking together these points.

Next `surfaceN` lines corresponds to the coordinates of points with integers `landX landY`.

2.3 Input for one game turn

A **single line** with 7 integers is given

`X Y hSpeed vSpeed fuel rotate power`

More in details: *i)* `X`,`Y` are the coordinates of Mars Lander (in meters)., *ii)* `hSpeed` and `vSpeed` are the horizontal and vertical speed of Mars Lander (in m/s). These can be negative depending on the direction of Mars Lander., *iii)* `fuel` is the remaining quantity of fuel in liters. When there is no more fuel, the power falls to zero., *iv)* `rotate` is the angle of rotation of Mars Lander expressed in degrees., *v)* `power` is the thrust power of the landing ship.

2.4 Output for one game turn

A single line with 2 integers:

`rotate power`

`rotate` is the desired rotation angle for Mars Lander. Note that for each turn the actual value of the angle is limited to the value of the previous turn $+/-15°$.

`power` is the desired thrust power. While 0 corresponds to off power, power = 4 is the maximum value that can be provided. Note also that for each turn the value of the actual power is limited to the value of the previous turn $+/-1$.

3 Solution

My solution is composed by two main parts:

1. a simulator of the coding game platform, made in python, which simulates the initialization of the problem and the correctness of each game turn. This part is important because provide the game status at each turn.

2. a Prolog program which solve a single game turn with a given input as defined in Section 2.3.

For what concern the simulator, this perform the same function of the solving platform of Coding Game. It just compute the next turn input to deliver to the Prolog program, check the landing status e finally plot the solution.

As said previously in Section 2, the game simulates a free fall without atmosphere. So the physic rules that describe the motion of the Mars lander are the following

$$\begin{cases} s(t) = s_0 + at \\ x(t) = -\frac{1}{2}at^2 + s_0 t + x_0 \end{cases}$$

with $a = [p\ cos(r + 90), p\ sin(r + 90) - g]$ assuming that p is the thrust power and r is the tilt angle of the Mars lander.

Thus the python script generate the initial input (containing the definition of the Mars Surface) and the conditions (position, speed, actual power, actual angle and remaining fuel) for the next game turn. New actions are printed in standard output stream by the Prolog program which is called by the python script itself with the following instruction:

```
swipl -s .\MarsLander_project\modules\msolve.pl -g predict
```

4 Prolog program

The Prolog program is organised in modules, all located in the directory *modules*:

4.1 mconfigs

This is the configuration file for the problem and it is defined as follow

```
:- module(mconfigs, [speed_eps/1,
                     y_eps/1,
                     p_max_step/1,
                     r_max_step/1,
                     r_max_degree/1,
                     danger_Vspeed/1,
                     danger_Hspeed/1]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%       CONFIGS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
speed_eps(3).
y_eps(20).
p_max_step(1).
r_max_step(15).
r_max_degree(90).
danger_Vspeed(40).
danger_Hspeed(20).
```

The two predicates `p_max_step` and `r_max_step` define the absolute value of the maximum step that can be performed between two different game turns. Instead `p_max_power` define the maximum thrust power while `r_max_degree` define the absolute value of the maximum degree that can be chosen as next actions. With `danger_Vspeed` and `danger_Hspeed` the absolute value of a dangerous speed, thus the speed defined in the landing constraints in Section 2 had been chosen. Finally `speed_eps` and `y_eps` are two values used as a safety margin.

4.2 minput

This is a module which contains the input data of the current instance game

```
:- module(input, [zone/2, bl_landing_site/1, mars_zone/1, g/1]).

%%% The zone is 7000m wide and 3000m high.
zone(7000, 3000).

%%% The limit for the landing site of the shuttle must be at least
bl_landing_site(1000).

%%% The game simulates a free fall without atmosphere. Gravity on Mars is 3.711 m/s .
g(3.711).

%%% Mars surface segments anchor points
mars_zone([
  surface(0, 1500),
  surface(1000, 2000),
  surface(2000, 500),
  surface(3500, 500),
  surface(5000, 1500),
  surface(6999, 1000)
]).
```

This module contains all the input data for the problem, like

- `zone` the dimension of the 2D Mars zone,
- `g` which predicate defines the Mars gravity,
- `bl_landing_site` instead defines the minimum wide size of the landing site,
- finally the predicate `mars_zone` describes Mars segments to draw by a list of coordinates points defined as a list of `surface` predicates.

4.3 mlander

The following module is modified by the python script at each game turn, in order to define the input for the next game turn:

```
:- module(lander, [lander/7]).
%% Start position of the lander and amount of carburants in litres
lander(2500, 2500, 0, 0, 500, 0, 0).
```

As previously said in Section 2.3, at each turn the following line
`X Y hSpeed vSpeed fuel rotate power`
is given in input. Thus the predicate `lander` includes these information.

4.4 mcheck

This is a module which contains some predicates for checking some conditions useful to solve the problem:

```
:- module(mcheck, [checkLandingsite/4,
                   landing_site/3,
                   %%%
                   checkOverLandingSite/1,
                   checkLanding/1,
                   checkSpeedLimit/2,
                   checkOppositeDirection/2,
                   checkHighSpeedH/1,
                   checkLowSpeedH/1]).
:- use_module(minput).
:- use_module(mconfigs).
:- use_module(mutils).
:- dynamic landing_site/3.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                INPUT CHECKS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%% checks the existence and the consistence with the minimum lenght of the landing site
checkLandingsite([], _, _, _):-
    throw("Surfaces must contain a consistent landing site").
checkLandingsite([surface(X, Y)|_], BL, X1, Y1):-
    Y1 == Y, LLS is X - X1, BL < LLS + 1, !,
    asserta(landing_site(X1, X, Y)).
checkLandingsite([surface(X, Y)|T], BL, _, _):-
    checkLandingsite(T, BL, X, Y).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%                GAME CHECKS CASE CONDITIONS
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% true if the lander is passing over the landing site
checkOverLandingSite(Xl):-
    landing_site(X0, X1, _),
    X0 < Xl, Xl < X1.

%%% true if the lander is close to the land
checkLanding(Yl):-
    landing_site(_, _, Y), y_eps(Yeps),
    YplusEps is Y + Yeps,
    Yl < YplusEps.

%%% true if speed is under speed limits
checkSpeedLimit(Sh, Sv):-
    abs(Sh, ShAbs), abs(Sv, SvAbs),
    danger_Hspeed(DSh), danger_Vspeed(DSv), speed_eps(Eps),
    DSh1 is DSh - Eps, DSv1 is DSv - Eps,
    ShAbs =< DSh1, SvAbs =< DSv1.

%%% true if the speed direction is opposite to the direction
%%% from the lander to the landing site
checkOppositeDirection(Xl, Sh):-
    landing_site(X0, _, _), Xl < X0, Sh < 0, !.
checkOppositeDirection(Xl, Sh):-
    landing_site(_, X1, _), X1 < Xl, Sh > 0.

%%% true if the horizontal speed is over the horizontal speed limit
checkHighSpeedH(Sh):-
    abs(Sh, ShAbs),
    danger_Hspeed(DSh), DSh1 is DSh * 4,
    ShAbs > DSh1.

%%% true if the horizontal speed is under the horizontal speed limit
checkLowSpeedH(Sh):-
    abs(Sh, ShAbs),
    danger_Hspeed(DSh), DSh1 is DSh * 2,
    ShAbs < DSh1.
```

First things first, `checkLandingsite` is used to determine the position of the landing site. Since the landing site is designed as a flat region, the list of `surface` predicates is iterated until two following points with same vertical coordinates are found. Next by using the predicate `asserta`, a new fact, namely `landing_site`, is added to the database, with 3 arguments: left and right horizontal coordinates and the vertical one of such points.

Next rule to analyse is `checkOverLandingSite`, which states true if and only if the lander is passing hover the landing site. `checkLanding` is true if and only if the lander is at a certain distance from the land. This is useful to in order to regulate the speed accordingly also to the distance to the land. Another very important rule is `checkOppositeDirection` which is true when the lander is going to the opposite direction respect to the landing site. Please note that all these 3 rules need before the call of the rule `checkLandingsite` in order to assert the fact `landing_site`.

Next step is introduce rules on speed control. For example `checkSpeedLimit` asserts if the horizontal as the vertical speed absolute value are under a certain limit which is what had been defined in the configuration module (see Section 4.1). At the contrary `checkHighSpeedH` and `checkLowSpeedH` assert controls when the horizontal speed is above the speed limit. The first rule avoid to lose the control of the lander without, having the possibility to brake anymore while the second one is used in order to avoid to have a too slow motion, with the risk to go out of fuel.

4.5 mutils

This module contains some utility rules, including 2 private auxiliary predicates.

```
:- module(mutils, [angleDecelerate/3,
                   angleToLandingSite/2,
                   regulateTPower/2,
                   getNextAngle/3,
                   getNextTPower/3]).
:- use_module(minput).
:- use_module(mlander).
:- use_module(mconfigs).
:- use_module(mcheck).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% computes the norm 2 of a vector
norm2_helper([], S, U):-
    sqrt(S, U).
norm2_helper([H|T], S, U):-
    HS is (H * H) + S,
    norm2_helper(T, HS, U).

norm2(V, U):-
    norm2_helper(V, 0, U).

%%% converts an angle in radiants to degree
radToDeg(Rrad, Rdeg):-
    Rdeg is Rrad * 180 / pi.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% returns the best angle to slow down marse lander
angleDecelerate(Sh, Sv, Rround):-
    norm2([Sh, Sv], Snorm),
    Rrad is asin(Sh / Snorm),
    radToDeg(Rrad, Rdeg),
    Rround is round(Rdeg).

%%% returns the angle needed to contrast Mars gravity while
%%% the leander is going to the landing site
get_angle(Xl, R, R1):-
    landing_site(X0, _, _), Xl < X0, !,
    R1 is -R.
get_angle(Xl, R, R1):-
    landing_site(_, X1, _), Xl > X1, !,
    R1 is R.
get_angle(_, _, 0).

angleToLandingSite(Xl, R):-
    g(G),
    Rrad is acos(G / 4),
    radToDeg(Rrad, Rdeg),
    Rround is round(Rdeg),
    get_angle(Xl, Rround, R).

%%% returns the thrust power needed to get a null vertical speed
regulateTPower(Sv, P):-
    Sv > 0, !,
    P is 3.
regulateTPower(_, 4).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% compute the right angle to deliver to the game, according to the limit of +/-15
%%% among two next game turns, the previous angle and the desired one
getNextAngle(Rprev, Rdes, Rdes):-
    Rdiff is Rprev - Rdes,
    abs(Rdiff, RdiffAbs),
    r_max_step(RMaxStep),
    RdiffAbs =< RMaxStep, !.
getNextAngle(Rprev, Rdes, Rnew):-
    Rdes < Rprev, !,
    r_max_step(RMaxStep),
    Rnew is Rprev - RMaxStep.
getNextAngle(Rprev, _, Rnew):-
    r_max_step(RMaxStep),
    Rnew is Rprev + RMaxStep.

%%% compute the right power to deliver to the game, according to the limit of +/-1
%%% among two next game turns, the previous power and the desired one
getNextTPower(Pprev, Pdes, Pdes):-
    Pdiff is Pprev - Pdes,
    abs(Pdiff, PdiffAbs),
    p_max_step(PMaxStep),
    PdiffAbs =< PMaxStep, !.
getNextTPower(Pprev, Pdes, Pnew):-
    Pdes < Pprev, !,
    p_max_step(PMaxStep),
    Pnew is Pprev - PMaxStep.
getNextTPower(Pprev, _, Pnew):-
    p_max_step(PMaxStep),
```

```
86      Pnew is Pprev + PMaxStep.
```

The 2 private predicates are just mathematical tool used by other public predicates in order to solve the problem. Thus, `norm2` computes the euclidean norm of a vector, represented in prolog as list, and `radToDeg` which converts an angle in radians to degree. Indeed it is known that to converts such an angle α_r measured in radians to an angle α_d measured in degree it is possible to follow the proportion:

$$\alpha_d = \frac{\alpha_r \cdot 180}{\pi}$$

Next rule to analyse is `angleDecelerate`. When the speed is to much high it could be important regulate it, but in order to do it, update the tilt angle is necessary. To successfully generate the best rotation angle it is possible to follow an important trigonometry's rule: in a right triangle (like in Figure 2), knowing the length of the sides of the triangle it is possible to determine the wide of the angles α by computing the inverse of the cosine of ratio between the adjacent side and the hypotenuse of the triangle or by computing the inverse of the sine ratio between the opposite side and the hypotenuse, as follow:

$$\alpha = acos\left(\frac{b}{c}\right) \qquad \alpha = asin\left(\frac{a}{c}\right)$$

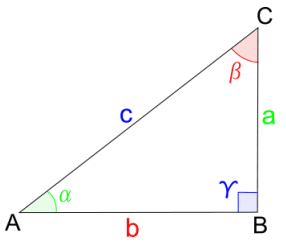
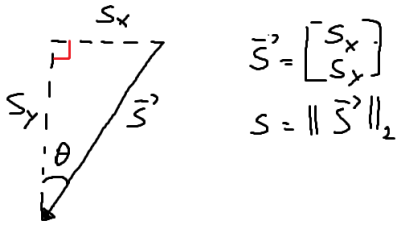


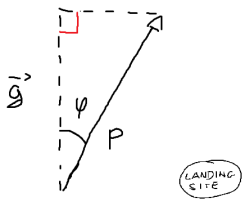
Figure 2: A right triangle

It is possible to apply this concept to the problem of Mars lander for decelerate the lander. Indeed looking at the figure 3 (a), it can be noticed that in order to compute the angle θ it is possible to follow the same approach, with the auxiliary of the euclidean norm it is possible in fact to compute the length of the speed triangle. Thus to compute θ we will have, having S_x and S_y as horizontal and vertical speed:

$$\theta = asin\left(\frac{Sx}{\|[S_x, S_y]^T\|_2}\right)$$



(a) *Speed triangle*



(b) *Gravity and thrust power triangle*

Figure 3: Two angle computation cases

Notice that this procedure will return the desired rotation angle. But in Section 2 it had been already defined the constraints of $+/-15°$ between two following turns. For this reason the rule `getNextAngle` had been defined in order to return the right angle value according to the desired and the previous ones (similarly for thrust power `getNextTPower`).

Please notice that there is no other to do with the angle θ. Indeed, it is possible to demonstrate that this angle will always balance the speed in bring the lander in the opposite direction. Knowing that the norm of a vector is a quantity always positive, two cases can be presented (with $S = [S_x, S_y]$):

- $S_x > 0$: this means that the lander is going to the right. In this case the ratio $\frac{S_x}{\|S\|_2} \geq 0$. Thus, the result of the inverse function of the sine will result in a positive value, meaning that next tilt angle will make the lander rotate to the left, thus reducing its speed,
- $S_x < 0$: it is symmetric to the $S_x > 0$.

Moreover the reason behind the choose to use the horizontal speed as opposite side is motivated by the fact that the angle in this way will be computed respect to the y axis, comporting no other action to the angle to give to the lander.

Next rule is `angleToLandingSite`. This returns the best angle capable to compensate the Mars gravity while the lander is going to the direction where the landing site is located. Also in this case it is possible to see the same approach as before, assuming that the the gravity is vertical side of a triangle and where the minimum thrust power able to contrast the gravity (as previously said in Section 2 it is equal to 4) is its hypotenuse, as shown in Figure 3 (b). In this case the tilt angle will be computed as follow:

$$\phi = acos\left(\frac{g}{4}\right)$$

Also in this case the tilt angle is computed respect to the y axis. Then, depending on the current position of the lander respect to the landing site, the predicate `get_angle` will return the sign of the angle. In particular, assuming R is the tilt angle computed for

the predicate `angleToLandingSite`, `get_angle` will return: *i)* $-R$ if the lander is on the left of the landing site, *ii)* R if the lander is on the right of the landing site, *iii)* 0 if the lander is over the landing site

Finally the last rule in this module is `regulateTPower` which, as its name says, returns the thrust power to which is needed to compensate the negative vertical speed in order to reach the zero. A thrust power of 3 is set if the vertical speed results to be positive in order to limit the amount of fuel used during the landing.

4.6 msolve

Finally, the last module contains the main predicate (public) of the program, namely `predict`, the rule that can solve a single game turn.

```
:- module(msolve, [predict/0]).
:- use_module(minput).
:- use_module(mlander).
:- use_module(mcheck).
:- use_module(mutils).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%% solve predicate
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%% HPA case: Episode 1
predict_HPA(Yl, _, _, Rprev, Pprev):-
  checkLanding(Yl), !,
  getNextAngle(Rprev, 0, Rout),
  getNextTPower(Pprev, 3, Pout),
  write(Rout), write(" "), write(Pout),
  write(" predict_HPA_HP1").

predict_HPA(_, Sh, Sv, Rprev, Pprev):-
  checkSpeedLimit(Sh, Sv), !,
  getNextAngle(Rprev, 0, Rout),
  getNextTPower(Pprev, 2, Pout),
  write(Rout), write(" "), write(Pout),
  write(" predict_HPA_HP2").

predict_HPA(_, Sh, Sv, Rprev, Pprev):-
  angleDecelerate(Sh, Sv, Rdes),
  getNextAngle(Rprev, Rdes, Rout),
  getNextTPower(Pprev, 4, Pout),
  write(Rout), write(" "), write(Pout),
  write(" predict_HPA_HP3").

%%% HPB case: Episode 2
predict_HPB_HP1(Sh, Sv, Rprev, Pprev):-
  angleDecelerate(Sh, Sv, Rdes),
  getNextAngle(Rprev, Rdes, Rout),
  getNextTPower(Pprev, 4, Pout),
  write(Rout), write(" "), write(Pout),
  write(" predict_HPB_HP1").

predict_HPB_HP2(Xl, Rprev, Pprev):-
  angleToLandingSite(Xl, Rdes),
  getNextAngle(Rprev, Rdes, Rout),
  getNextTPower(Pprev, 4, Pout),
  write(Rout), write(" "), write(Pout),
  write(" predict_HPB_HP2").

predict_HPB_HP3(Sv, Rprev, Pprev):-
  regulateTPower(Sv, P),
  getNextAngle(Rprev, 0, Rout),
  getNextTPower(Pprev, P, Pout),
  write(Rout), write(" "), write(Pout),
  write(" predict_HPB_HP3").

%

predict_HPB(Xl, Sh, Sv, R, P):-
  checkOppositeDirection(Xl, Sh), !,
  predict_HPB_HP1(Sh, Sv, R, P).
predict_HPB(_, Sh, Sv, R, P):-
  checkHighSpeedH(Sh), !,
  predict_HPB_HP1(Sh, Sv, R, P).

predict_HPB(Xl, Sh, _, R, P):-
  checkLowSpeedH(Sh), !,
  predict_HPB_HP2(Xl, R, P).

predict_HPB(_, _, Sv, R, P):-
  predict_HPB_HP3(Sv, R, P).

%%% predict predicate, the one that can be used to solve
%%% a game turn of Mars lander
predict:-
  %%% searching the landing_site
  mars_zone(S),
  bl_landing_site(BL),
    checkLandingsite(S, BL, _, _),
  lander(Xl, Yl, Sh, Sv, _, R, P),
  %%% is the lander over the landing site?
  checkOverLandingSite(Xl), !,
  %%% Episode 1
  predict_HPA(Yl, Sh, Sv, R, P),
  halt.

```

```
predict:-
  %%% Episode 2
  lander(Xl, _, Sh, Sv, _, R, P),
  predict_HPB(Xl, Sh, Sv, R, P),
  halt.
```

During the prove of the rule `predict`:

A. In the first attempt `checkLandingSite` assert in the database the predicate `landing_site` that will be used during all the code also by the other modules. The next step will be made in order to check if the lander is already over the landing site (`checkOverLandingSite`, look Section 4.4), in this case it is possible to follow the hypothesis A in order to solve the *Episode 1*, invoking `predict_HPA`. To prove this predict, there are three possibilities:

   1. if `checkLanding` can be proved, the speed can be regulated by returning a sufficient thrust power (3) and a null tilt angle,
   2. if `checkLanding` fails but `checkSpeedLimit` can be proved, it is possible to reduce the amount of fuel consumed during the landing by reducing the thrust power to 2, also in this case it is needed a tilt angle equal to .
   3. if both `checkLanding` and `checkSpeedLimit` fail, this means that there could be the need to regulate the speed also according to the horizontal component, for this reason `anglDecelerate` will return the desired angle to decelerate correctly the lander (note that a thrust power equal to 4 is needed in this case).

B. If instead, in order to prove the predicate `predict`, the predicate `checkLandingSite` fails it can be followed the hypothesis B solving in this way the generalised problem, the *Episode 2*, invoking this time `predict_HPB`:

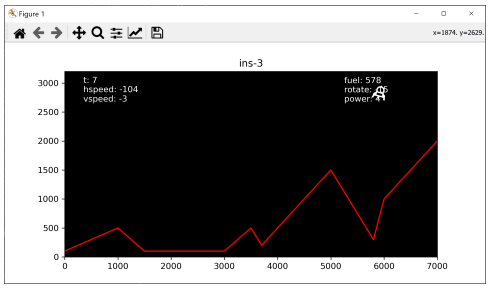
   1. First of all, it will be checked if the direction where the lander is going to is correct according to the position of the landing site or if it is going to fast according to the horizontal speed. This "or" condition can be implemented by proving the predicates `checkOppositeDirection` and `checkHighSpeedH` with the auxiliary of the *CUT* predicate ("!").
      - In this case there is the need to decelerate modifying the tilt angle, so it will be invoked the rule `angleDecelerate` with the maximum available thrust power to decelerate as fast as possible the lander.
   2. Alternatively if the previous or condition fail but `checkLowSpeedH` can be proved:
      - it is possible to compute the desired angle in order to reach the landing site (invoking the predicate `angleToLandingSite`) with the maximum possible thrust power.
   3. if instead `checkOppositeDirection`, `checkHighSpeedH` and `checkLowSpeedH` all fail then
      - the tilt angle can be 0 and the thrust power can be adjusted accordingly to the rule `regulateTPower`.

Please note that both the tilt angles and the thrust power cited before are only the desired one, so before to print the new action for this game turn, `getNextAngle` and `getNextTPower` will computes the new actions to write according to the desired and the previous actions.

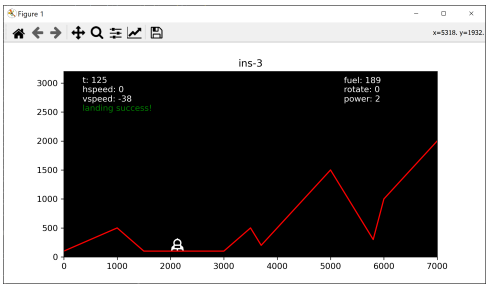
Appendix

To execute the program it is possible to type the following script:

```
python .\MarsLander_project\ML.py --data ins-0 --plot
```



(a) *Initial step*



(b) *Landing success*

Figure 4: ins-3 example