UNIVERSITY OF TURIN
FACULTY OF ECONOMICS
SIMULATION MODELS FOR ECONOMICS
PROF. PIETRO TERNA
ACADEMIC YEAR. 2012/2013

ESSAY ON A NETLOGO SIMULATION:

"DYNAMIC PRICING FOR PARKING SPACES IN A CITY ENVIRONMENT"

VALENTINA GALLIMBENI
VALENTINA VARAGONA

SF PARK (http://sfpark.org/)

Our aim is to build a model in Netlogo which will allow us to simulate a new project (which is currently developing) taking place in San Francisco parking environment . This experiment was performed following Donald Shoup's theory over parking availability and its relationship to parking prices. (http://shoup.bol.ucla.edu/Chapter1.pdf first chapter of Donald Shoup's book "The high cost of free parking").

SFMTA (San Francisco Municipal Transportation Agency http://www.sfmta.com) established SFpark to use new technologies and policies to improve parking in San Francisco. Reducing traffic, by helping drivers find parking, benefits everyone. More parking availability makes streets less congested and safer.

SFpark works by collecting and distributing real-time information about where parking is available so drivers can quickly find empty spaces. To help achieve the right level of parking availability, SFpark periodically adjusts meter and garage pricing, up and down, to match demand. Demand-responsive pricing encourages drivers to park in underused areas and garages, reducing demand in overused areas. Through SFpark, real-time data and demand-responsive pricing work together to readjust parking patterns in the City so that parking is easier to find. Access to real-time parking availability information online, via text, and through smartphone apps helps drivers find a space. Longer time limits and new meters that accept credit/debit cards, SFMTA parking cards and coins make parking more convenient and result in fewer parking fines.

How it works

In the first elementary scenery we want to recreate the following situation:

1. agents are created in the corners of the world, corners are their "home";
2. the center represent the "desired" area, where the main parts of the parking areas are located.

We hypothesize that all the agents' aim is to get as close to the center as possible because it represents the most desirable parking spaces. In order to do so we set both a home and parking space variable which pertain to the patches, so that the turtles can distinguish where to go and where to stop.

We create all the different agents with different reservation prices (the maximum price they are willing to pay). Different parking areas have different exogenous prices (at the beginning).

If the agents find a parking area where the price is less or equal than their reservation price, they will stop. If they do not succeed in their purpose they will go back home. To obtain this result we will give the turtles a *parked?* variable.

Going back to the corners the turtles will stop if they find an area with a price equal or less to their reservation price. Therefore we need to set the commands so that, even while going back home, the agents will keep looking for parking, which is also provided by the variable *parked?*. If the car was parked and exceeded its parking time it will go back to one of the "home" patches. If they don't find any free parking space with the right price they will keep looking.

To see how the real SFPark program works we also need to show how prices adjust. We will create a patch variable (*full?*) to check if the parking spaces were full or not.

If during a predetermined amount of time (e.g. 100 ticks), the parking spaces are more full than empty their prices will raise by a given amount (controlled by a slider). If they are more empty instead, their price will drop by a given amount (controlled by a slider). When the parking lot changes price we will reset an internal tick counter which will control if the parking lot will raise or decrease its price.

At the end of the period, we can analyze how the dynamic pricing worked. The change of pricing should highlight a better distribution of full parking spaces. To keep better track of the process we will build a graphical tool.

This model therefore will recreate both the drivers' behavior and the sensibility to prices while looking for parking. The underlying assumption of the real experiment is that free or low priced parking creates a greater demand than it is possible to satisfy give the parking spaces available. Therefore it is only logic, following how every other market adjusts supply and demand, to raise or lower prices depending on the occupancy of the spaces.

Code

```
patches-own [ parking-space
              home-number
              price
              full?
              crowded
              empty
              always-empty ]

turtles-own [ reservation-price
              time-at-arrival
              parked?
              tries
              turtle-home ]
globals [time-counter
         months-passed]
```

The initial part of the code describes all the variables that the agents will own and also two global variables that we will use to simulate the passing of time.

We start with the patches variables that will help us simulate the city environment. The corner patches will be home patches and will have their own identifying number. The center patches, plus some random spots, will be parking spaces. The variables `full?` , `crowded` and `empty` will be used in parking lots to control the adjustment of prices.

The variables of the turtles will help us simulate driver's behavior. Our basic assumption is that all the car's main objective is to get to the "city" center and find a suitable parking space (for their reservation price, which is the maximum price they are willing to pay). Once they exhausted their parking time they will go back to where they were originated.

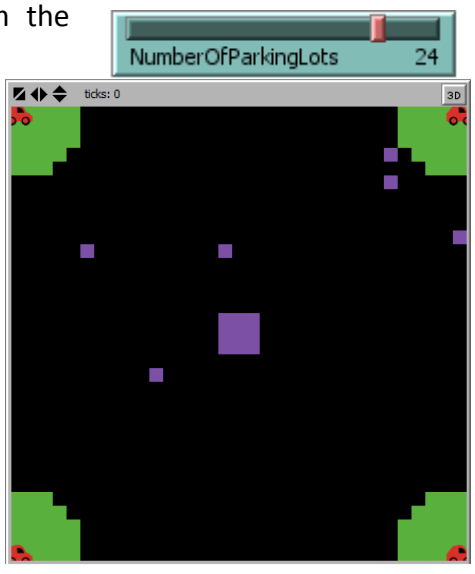
After defining the variables we start with the setup procedure which will create our "city".

```
to setup
   clear-all
   set-default-shape turtles "car"
   cars
   ask patches [ setup-home
                 setup-central-parking ]
   AddParking
   setup-prices
   reset-ticks
end
```

We use default commands as `clear-all` and `reset-ticks` but also some specific procedures. `Clear-all` will eliminate any data from the previous run and `reset-ticks` will set the tick counter to zero.

If we didn't use these two commands we wouldn't be able to start a new experiment with no previous data recorded. In the car procedure (`to car`) we ask each of the corner patches to sprout a quarter of the agents needed. If we had just asked NetLogo to create the desired number of cars the program would have originated them in the center of the world, which is the default procedure in NetLogo.

We also had to separate the central parking creation from the peripheral parking in order to give the user control over the number of parking lots called into existence. The slider which controls the number of parking lots counts both central and peripheral parking lots therefore the user, when deciding the desired number of spaces, needs to consider the fixed 9 central parking spaces. To give even more control to the user we also set up the prices so that they can be different between the outskirts of the "city" and the "city" center, procedure contained in setup-prices and controlled by two sliders.



At the end of the setup procedure our world will look like the picture on the right. All the cars at the corners with the "Home" patches graphically identified by the color green and the parking lots by the color violet.

Next step in our code is the `to go` button.

```
to go
    driving
    time
    CheckParking
    adjust-prices
    tick
end
```

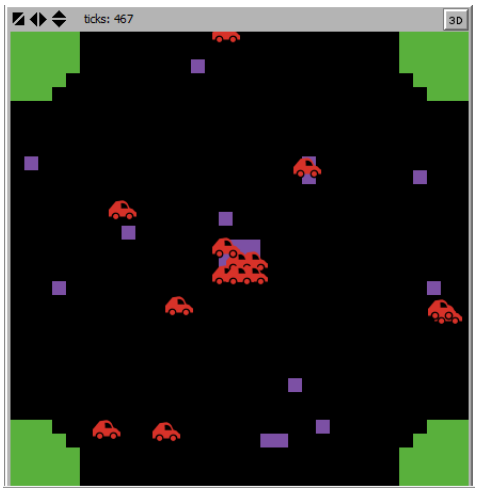
This button incorporates all the main procedures both regarding turtles and patches. We can start analyzing the `driving` procedure which determines the behavior of cars. Cars in fact can be in two states (governed by the variable `parked?`), either parked or looking for parking.

In the first case we defined a procedure `LeaveParking?` which checks whether the car has exceeded its parking time (controlled by a slider). If this is true the procedure `go-home` will be called and the car will go back to where it was generated. In the case where the turtle is looking for parking the code is slightly more complex and it is composed by a "tree" of `ifelse` clauses.

```
to move
   ifelse (pcolor = violet and reservation-price > price and ( patch-here != any? turtles ))
    [set parked? true
     set time-at-arrival ticks ]
    [ifelse  tries < 5
      [move-to one-of patches with [(distancexy 0 0) < 2]
       set tries (tries + 1)]
      [ifelse (pcolor = violet and reservation-price > price and ( patch-here != any? turtles ) and (distancexy 0 0) > 2 )
        [set parked? true
         set time-at-arrival ticks]
        [set heading (random 360)
         fd random 5]]]
end
```

In the first `ifelse` clause we ask the car if the conditions for it to be parked are met. In order to simulate the parking experience the cars can only be parked in a parking lot (which is identified by the color violet), which should be no more expensive than how much the driver is willing to pay and empty (with no other cars parked). If a turtle is parked on a spot another turtle can pass on the same spot (and stop until the next tick) but it won't park in the same spot. This is because just if all the conditions above are satisfied a car will switch its status to parked.

In the next period (from when the car stops to when the parking time is exceeded), if the car found suitable parking it will stop moving until it is time for it to go home (which is part of the procedure we have examined above).



If there aren't the conditions for the car to stop and park it will still look for a parking lot in the city center. To better simulate the behavior of drivers looking for a parking lot we decided that each car will have five tries to find a parking space in the center before moving to the peripheral parking lots. This conduct is a reflection of the drivers willingness to park their cars as close as possible to the place they are going to.

In case they do not succeed in five tries to find a parking lot in their favorite location then, as every driver tired of circling around, they will start looking around in further locations.
The last `ifelse` clause checks if the car is on a suitable patch for parking with almost all the same conditions as the first `ifelse`. The only added condition is that the new parking lot should not be situated in the city center. We have added the last condition since we assume that after the first five tries the drivers will completely give up on the hope to park in the preferred area and will resort to find parking in the rest of the world.
When they find parking in the outskirts their behavior will be identical to the conduct of the cars that find a spot in the city center. They will stop for an amount of time decided by the user and then will directly go back where they were originated.
In case they do not find a parking lot they will look for it randomly in the world.

With this procedure we have finished analyzing the modus operandi of our main agents, the cars, and we can start investigating the behavior of the patches in reaction to the cars'.
Since this model main goal is to analyze a world where dynamic pricing is applied to parking lots the modeling of lifelike agents (cars) is as important as the representation of a fair method for adjusting prices. The following two procedures were designed to create the adjustment of prices process.

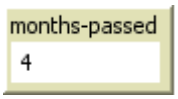
```
to CheckParking
   ask patches with [pcolor = violet]
    [ifelse any? turtles-here [set full? true
                               set parking-space 0]
                              [set full? false
                               set parking-space 1]]
end

to adjust-prices
   ask patches with [pcolor = violet]
    [if full? = false
      [set empty (empty + 1)]
     if full? = true
      [set crowded (crowded + 1)]
     if ((time-counter = 0) and (empty - crowded) > 10)
      [set price (price - decrease-price-by)
       set crowded 0
       set empty 0]
     if ((time-counter = 0) and (crowded - empty) > 10)
      [set price (price + increase-price-by)
       set empty 0
       set crowded 0 ]
     if price < 0
      [set price 0]]
end
```

First of all we ask the patches to check whether a car is parked on them or not. Thanks to this we can give a Boolean value to the variable `full?` .
This doesn't only help us to adjust the prices but gives us an up to date value for the empty parking spaces in the whole city thanks to a monitor and a graphical tool in the interface.
The procedure `adjust-prices` to a first look can be similar to the procedure we used to make the turtles move but there is a substantial difference. Where the first one was a `ifelse` "tree" this one is a sequence of `if` clauses.
This implies that, at every tick, the program will execute all the if to check whether they are true without being influenced by the momentary conditions of the patches the program is asking to.
The first `if` checks if the patch is empty. In that case it will add 1 to the variable empty. With this system we can count for how long in the period we want to consider (e.g. a month) the patch has

been empty. If the patch is full, in contrast with the `ifelse` function, nothing happens while the program is running the first `if` clause.
The same behavior will take place in the second `if` clause. If the patch has a car parked on it then it will set the variable crowded to its value plus one.
In order to explain the third and fourth if clause we first need to take a look at the last procedure of the model.
The only method in Net Logo to count time passing are the ticks. At each tick all the agents move and or perform the action that is required by the program.
In order to fairly adjust prices though we could not base our procedure just on ticks or the price changes would have been extremely biased and not at all resembling what could happen in real life. Therefore we used a procedure we called `time`, since it emulates the passing of time.

```
to time
  set time-counter (time-counter + 1 )
  if time-counter = 100
   [set time-counter 0
    set months-passed (months-passed + 1)]
end
```

This procedure is called by the `to go` button and therefore it is summoned at every tick.
When the procedure is called the `time-counter` variable will augment by one. When this variable reaches the value 100 (that we choose arbitrarily) the `if` clause will be verified and the `time-counter` set to zero.
This will make the `months-passed` variable advance by one. The months passed are visible through a monitor in the interface. Even though it is not essential for the proper running of the program we have added this variable to show how the fictional passing of time effects the results.



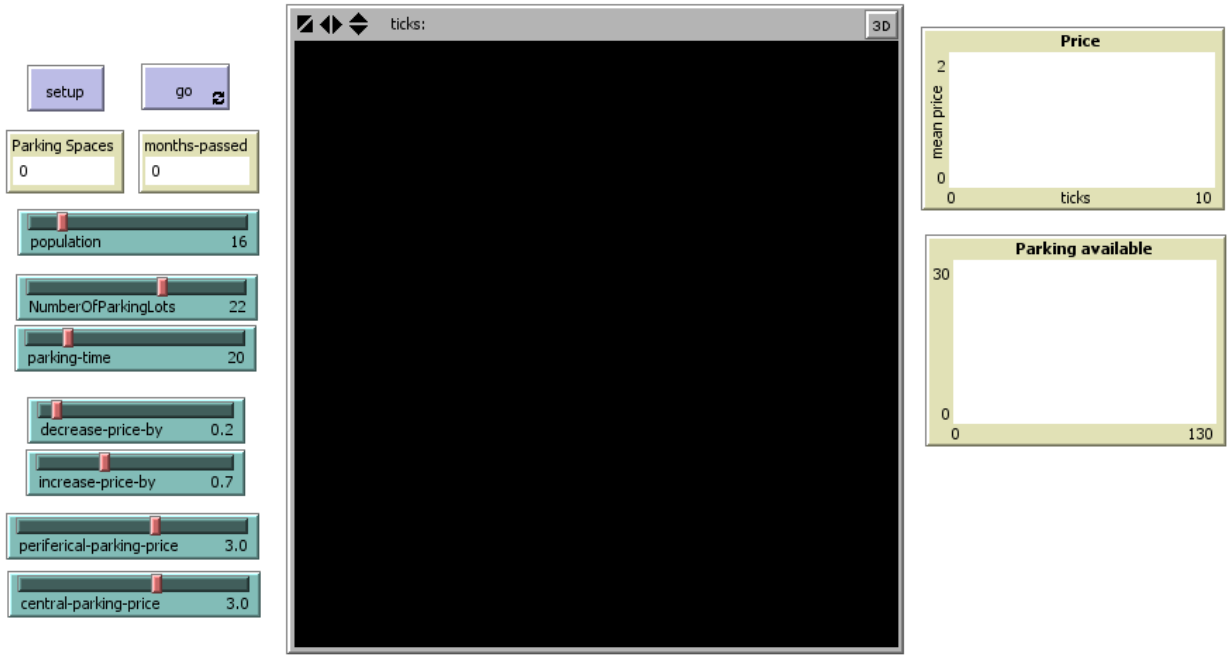
Now that we have explained how we decided to model time we can finish describing the last two `if` clauses which we will rewrite here for completeness. They have the same structure but opposite results.

```
if ((time-counter = 0) and (empty - crowded) > 10)
 [set price (price - decrease-price-by)
  set crowded 0
  set empty 0]
if ((time-counter = 0) and (crowded - empty) > 10)
 [set price (price + increase-price-by)
  set empty 0
  set crowded 0 ]
```

Instead of adjusting the prices continuously, which wouldn't be convenient in real life, we decided to adjust them in what we define as a month in the above procedure (100 ticks).
Each time the `time-counter` variable is reset to zero the program will check for the difference between the two variables crowded and empty.
If we had let the prices adjust on the absolute values of crowed and empty, we wouldn't be able to control the timing of the adjustment. Furthermore with this method we reduce the variations to have a clearer view of the behavior of prices.
The first `if` clause will decrease prices if the time the patch was empty is at least 10% more than the time the patch was full.
Once there is a change in the price of the patch both variables (crowded and empty) are set back to zero. The same happens if `crowded` is 10% bigger than `empty`, then the price will increase. In

case the difference between `crowded` and `empty` is less than zero the price will not be adjusted. Both empty and crowded will continue growing until a period ("month") in which one is bigger than the other.
At the end of the procedure to `adjust-price` we put a non-negativity constraint for prices.
Since the model should respect microeconomic theory it would be impossible to have negative prices (where negative prices mean that the driver would be paid by the parking lot authority to park his/her car in that particular space).

Interface



After commenting most of the code we want to spend a couple more lines on the interface of our program. We decided to divide all the sliders and buttons from the graphical tools in order to give a more clean cut look and to help set up experiments more easily.

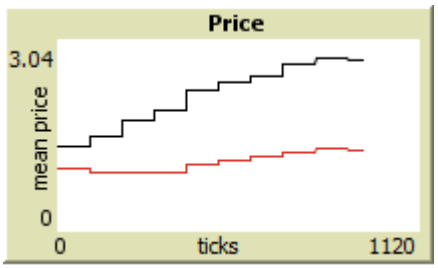
The only two buttons present in the interface are the `setup` and the `go` button. The `go` button runs an infinite cycle, for convenience, but it is also possible to make it run tick by tick (by modifying the button properties).
Underneath the buttons there are two monitors. One shows the amount of parking spaces available at each tick, the other shows the fictional passing of time. Even though the user cannot interact with the monitors they are very useful during experiments as they can be points of reference.

Right below the monitors we find all the sliders. The sliders govern the most important features of the model. Changing their value can lead to different results in the simulations.

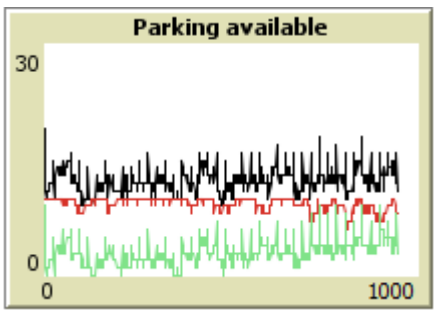
The `population` slider determines the amount of cars present in the "city". The user can augment it 4 by 4, due to the code construction (each corner sprouts one car therefore quantities that are not divisible by 4 cannot be displayed).
The `number-of-parking-lots` slider can be adjusted to add or subtract parking spaces in the world. Its minimum value is 9 since the 9 central parking lots are default, by construction of the model.
The `parking-time` slider sets limits on the amount of time that cars stay parked. Its value can vary from 0 to 100 ticks. Keeping in mind that 100 ticks is the adjustment period for prices, moving this slider can create very interesting effects on behavior of prices.
The `increase-price-by` and `decrease-price-by` sliders set the change that will affect the adjustments of prices. To better simulate real life price adjustments the prices will drop more than increase (since when introducing it in a city it is necessary to consider the population reaction to steep increases in prices), but the user can experiment with any combination of increase and decrease.
The last two sliders regulate prices in central parking and in peripheral parking. Not all the parking lots will assume that exact starting value since they will take as price a random integer between 0 and the slider value. This two sliders can be operated jointly to widen or narrow the differences in price between the outskirts and the "city" center.

The last part of the interface are the two graphical tools on the right.
The first one follows the evolution of prices over time while the second one relates empty parking lots with time.

This first graph contains two different lines. The black one follows the prices of the parking lots in the "city" center.
The red line instead tracks the mean price of all the parking lots. We haven't added a line just for the peripheral parking because one of the main assumptions of this model: given dynamic prices the parking lots medium price will be lower than it was without dynamic pricing.

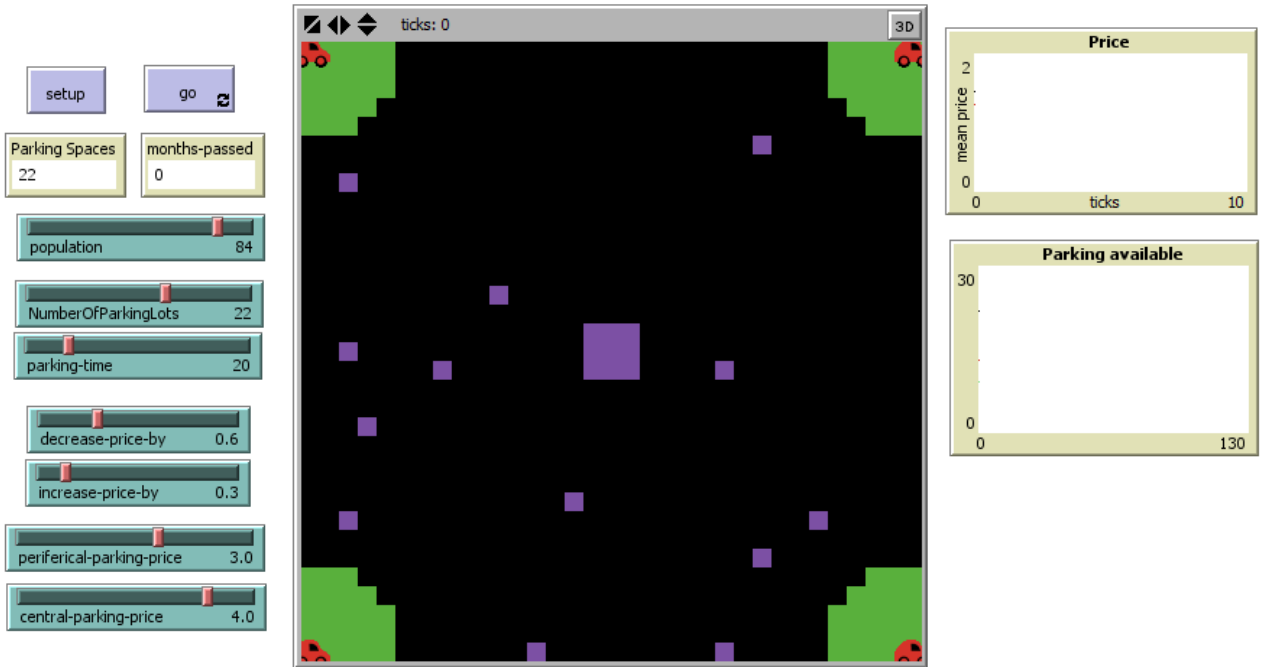


The second graph instead contains 3 lines. The black line is the sum of parking available at all times (both in the center and in the outskirts). The red line is peripheral parking available and the green line is central parking available.
We added this second graph since the other assumption of the model was that, given the right price to reach an equilibrium between offer and demand, there would always be some parking available, even in the more congested areas.



EXPERIMENT PLAN

Our model represents an hypothetical city, with houses at the corners (green coloured), and parking spaces both placed in the middle of the environment and randomly spread around the city (violet coloured).
The aim of this model is to represent the behaviour of agents while searching for a parking space, and the automatic adjustments of the price of our parking space.
As we can see from the interface we can modify, using the sliders, the population (our cars), the number of parking lots, their price (both peripheral and central) and the amount of increase or decrease in price we want.
The thing to notice is that we have added one monitor, in order to keep trace of the time passed.
From now on, we will refer to months (where one month is given by 100 ticks) in order to comment the results of our experiments.
We have also added one plot called "Parking available", in order to show how the availability of parking spaces changes over time.

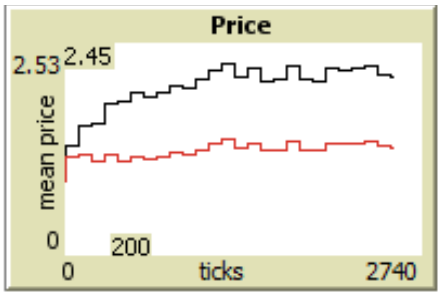


EXPERIMENT 1

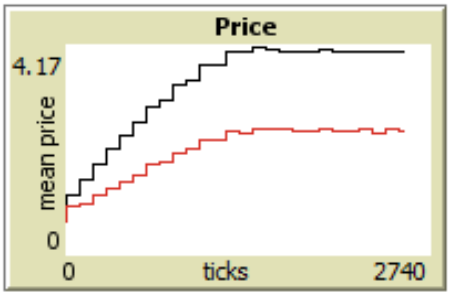
What we do: modify the population through the slider, without touching any other variable.
Aim: study the behaviour of prices, both of peripheral parking spaces and of central parking spaces.

Start with a low number of cars, suppose 20, and an average number of parking lots, suppose 15.
The initial price of the central parking space is set at 2.5 and the one of the peripheral parking space is set at 1.5.

As we can see, after 1 month the prices of the parking spaces start diverging. The black line (that represents the price of the central parking space) starts rising, while the red line (representing the average price of all the parking spaces) seems to be constant.
After the sixth month both the prices are still rising. It seems that they follow the same trend, with the black line always above the red one.
Going ahead with the time of our simulation, the prices keep rising at a constant rate, always maintaining the initial difference in price.
Suppose now to have a higher number of cars in our city, say 60.
Repeating the experiment it can be seen that, as before, the prices start diverging after the first month.
After the sixth month, as before, both the prices are still rising at a constant rate, maintaining their initial difference in price.
Thus we can deduct that changing only the population (the number of cars in the model) does not lead to significant divergence in prices.
As an example, consider the prices after 24 months: as we can see from the two graphs, the only difference is in the level of prices.
In the simulation with 20 agents, the two prices are around 2.5 and 1.4; in the simulation with 60 agents instead, the two prices are around 4.17 and 3.5.
The explanation is clear: if we have more agents in our simulation, the parking spaces (both the central and the peripheral ones) are crowded more often, leading to an increase in prices more visible.



Picture1: Prices after 24 months with 20 cars



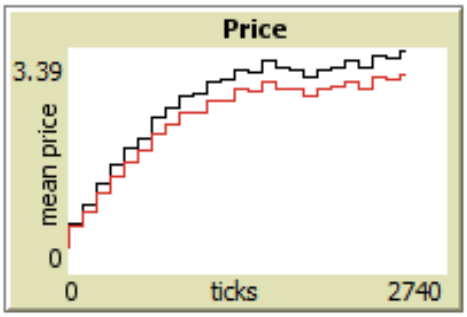
Picture2: Prices after 24 months with 60 cars

EXPERIMENT 2

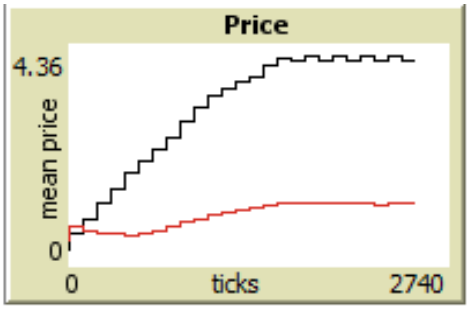
What we do: modify the number of parking lots through the specific slider, maintaining constant the other variables
Aim: study the behaviour of prices, both of peripheral parking spaces and of central parking spaces.

Start with a low number of parking lots, say 10 (which means 9 central parking lot and 1 peripheral), and with an average number of cars, say 40.
The initial price of the central parking space is set at 2.5 and the one of the peripheral parking space is set at 1.5.

As we can see, at the very beginning of our simulation, until the first month, the prices are constant, they do not change over time, since, for construction, we are assuming that the adjustment of prices is carried out every 100 ticks (where 100 ticks is equal to one month).
Right after the first month the prices start rising, at the same constant rate.
Suppose now to have a very high amount of parking spaces, say 30.
While running the simulation, we can immediately see that the prices of the central parking spaces (plotted by the black line) are below the red line, that represents the average price of all the parking spaces.
But right after passing the first month, the two line invert their position and the price of the central parking spaces starts growing, diverging from the red line.
After the sixth month the difference is quite high, with an average price of 2.5 for the central parking lots and only 1 for the overall parking lots.
As we keep running the simulation, the red line start following the black one, rising at the same constant rate, so that the difference in price does not change.
As an example, consider the graphs of the prices after 24 months:



Picture4: Prices after 24 months with 10 parking lots



Picture3: Prices after 24 months with 30 parking lots

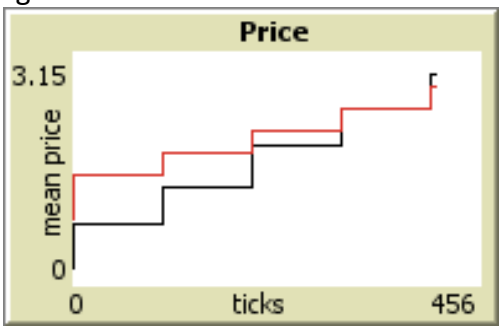
The explanation is clear: as we increase the number of parking lots available, we also increase the possibility to have a divergence in price between the central parking lots and the peripheral ones.

EXPERIMENT 3

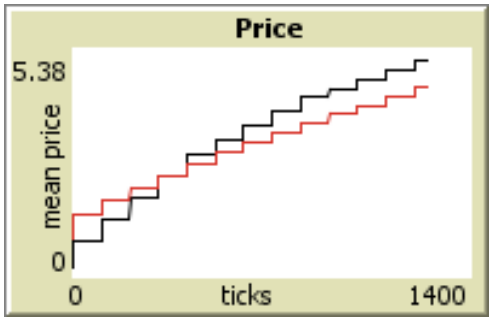
What we do: modify only the central parking price and the peripheral parking price, through the sliders.
Aim: study the behaviour of prices, both of peripheral parking spaces and of central parking spaces.

Start with setting the peripheral parking price at 4.0 and the central parking price at 1.5, and notice that this contrasts with our initial assumption of a higher price for the central parking spaces, with respect to the peripheral ones.
Running the simulation and paying attention to the plot of the prices, we can now notice that the red line (representing the prices of all the parking spaces) is above the black one (representing the prices of the central parking spaces). The difference between the two can vary each time we run the simulation, since the prices are set randomly, with a maximum value given by 4.0 for the peripheral parking spaces and by 1.5 for the central parking spaces.

As we keep running the simulation, we can notice that, on average, at the third month the black line reaches the red one: at this time the two prices are equal.
Then, the prices of the central parking spaces start increasing at a higher growth rate, overtaking the prices of the peripheral parking spaces.
This is an important results for our simulation: it means that our assumption about the preference we have for the central parking spaces still holds in each time we repeat the simulation.
Agents are always looking for the central parking lots, and only in case they do not find a free space there, they will start looking in the peripheral parking lots.
Repeating the simulation simply inverting the values in our initial assumption will not lead to significant results.



Picture5: Prices after 4 months



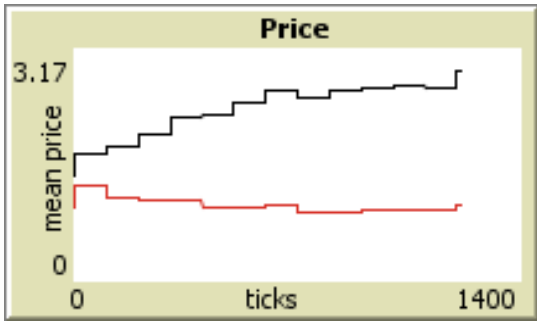
Picture6: Prices after 12 months

As we can notice from the two plots above, during the fourth month there is an overlapping of the two prices, followed by an increase in the prices of the central parking spaces (as we can notice from the second plot). We can notice this trend each time we run the simulation.

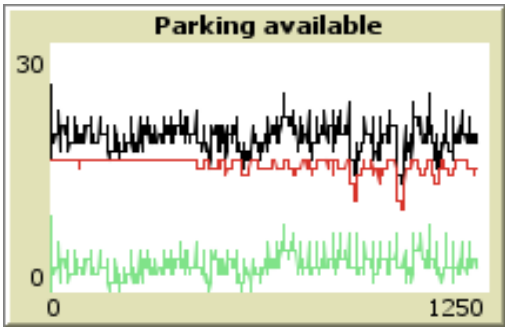
EXPERIMENT 4

What we do: modify the population through the slider, without changing any other variable.
Aim: see how the availability of parking spaces changes over time.

Suppose we have an average number of parking spaces, say 25.
We start the first simulation with a very low number of agents, say 16, and pay attention to the plot called "Parking availability". The black line represents total availability of parking spaces, the red line represents the availability of the peripheral parking spaces and the green line represents the availability of the central parking spaces.
As we start the simulation we can see that, before the first month, the green line varies constantly around a mean, while the red line varies from the mean with infrequent oscillation, maybe due to the fact that, since there are a few number of agents, they are almost always able to find a free parking lots in the central parking spaces.
As we keep running the simulation we see that the trend continues through time, with a relatively large volatility for the availability of the central parking spaces (green line), and a very low volatility of the availability of the peripheral parking spaces (red line).
Only after the fifth month the volatility of the red line seems to increase, possibly because of the contemporaneous adjustments in prices.
Here is the situation:

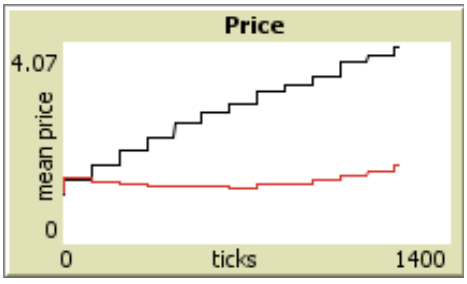


Picture7: Prices after 12 months

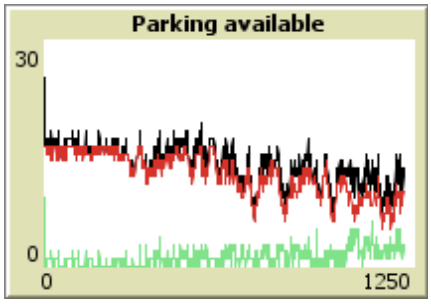


Picture8: Availability of parking spaces after 12 months

Suppose now to have a higher number of agents in our simulation, say 80, with the same number of parking lot, 25.
As we running the simulation, we can immediately see the difference: the red line start oscillating as much as the green one, since now there are more agents that are looking for a parking lot available in the peripheral parking spaces.
It seems, as before, that they both move around a constant mean, which is the number of parking lots we have define earlier.
As we keep running the simulation, we can see that this trend continues, with an equal volatility in all the line.
Always paying attention on the red line, the interesting this to notice is that there is a slight deviation from the mean from the third month on. The volatility increases, and this can be explained by looking also at the automatic adjustments of prices.
It could be possible that, due to the adjustments of prices (with the peripheral parking lots less costly now), the agents, while searching for parking space available, find now more free lots, and above all, more parking lots compatible with their reservation prices.
Here is the situation visible in the two plots:



Picture9: Prices after 12 months



Picture10: Availability of parking spaces after 12 months

As we keep running the simulation we can clearly see that the trend previously explained continues, and, above all, increases in its volatility.