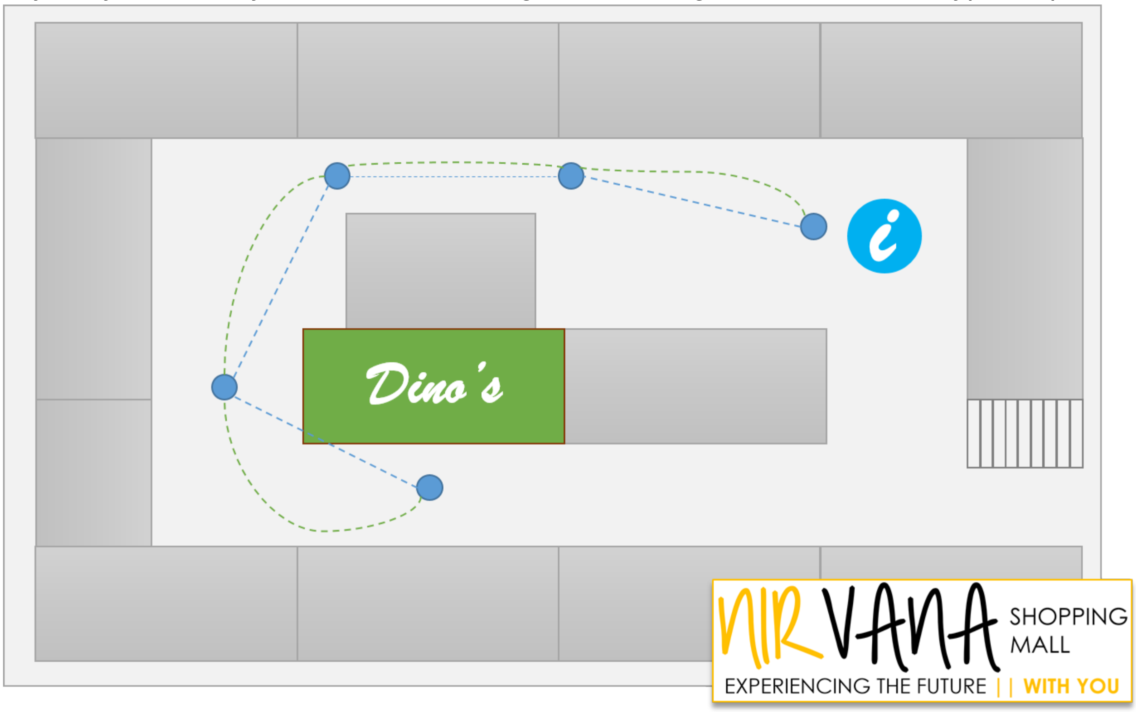
8.2 Motion Planning with Potential Fields - Moving in the mall

After implementing the SLAM algorithm, the robots provided by **UMA-MR** are able to simultaneously build maps of the malls and localize themselves within them. However, the **managers at Nirvana** are looking for a fully operational robot, and something is still missing: the navigation between any two points in the malls. These points could be, for example, an information point, a shop entrance or a shop counter, a rescue point, a restaurant, etc.

From previous developments, our team has an algorithm able to find a sequence of waypoints between the start point and the goal one, that is, to plannify a **global navigation**. So **our mission here** is to develop an algorithm able to command the robot to safely navigate from a start waypoint to a (close) goal one, that is, to carry out **local reactive navigation**.

The image below shows an sketch of the restaurants area in the **Nirvana mall**, along with an example of global navigation (blue waypoints and dotted lines) between the information point and the *Dino's* restaurant. In that example, the green dotted lines correspond to the trajectory followed by a local reactive navigation avoiding obstacles in the waypoints path.



8.2.1 Formalizing the problem

The **reactive navigation** (or **local navigation**) has the objective of moving towards a close destination while avoiding obstacles. For that, it is available sensor data within a specific *look-ahed* as well as the goal point (**inputs**), being the reactive navigation method in charge of producing motor commands (**outputs**) to safely reach such goal.

In this way, reactive navigation methods **does not require neither any kind of map of the environment nor memory of previous observations**. In practice, the last requirement usually arises since in some situations it could be useful to also consider the last sensor observations (e.g. while crossing a door).

Finally, reactive navigation techniques **must run very fast** (i.e. real time or close to it) in order to safely reach the goal point. If not, dynamic obstacles or deprecated motion commands could lead the robot to crash!

In summary:

```
reactive_navigation(current_location,target_location,sensor_readings)
    # Method computations ... so fast!
    return (v_l,v_r) # Motor actuation
```

8.2.2 Potential Fields

Potential Fields is a popular and simple technique for carrying out reactive navigation. Imagine the robot and the objects in its environment (like obstacles, the goal, etc.) are surrounded by invisible fields, similar to magnetic fields around magnets. These fields exert virtual forces on the robot, guiding its movement.

To do this, it consist of defining a **potential (energy) function** over the free space in the robot workspace, which has a **global minimum** at the goal and a maximum at obstacles. Then, in each iteration of the algorithm, the robot moves to a lower energy configuration, similar to a a ball rolling down a hill. To carry out such navigation the robot applies a force proportional to the **negated gradient of the potential field** (recall that the gradient always go in the direction in which the signal increases, and the robot pursues a lower energy, so it has to use the negated gradient).

The **potential (energy) function** defines a **potential field** over the workspace. For each robot position p in such workspace, the energy function is computed as:

$$U(p) = U_{att}(p) + U_{rep}(p)$$

where:

- $U_{att}(p)$ is the **atractive potential field** representing the squared Euclidean distance to the goal, which is retrieved by:

  $$U_{att}(p) = \frac{1}{2} K_{att} d^2_{goal}(p)$$

  being d_{goal} said distance from the robot to the goal: $d^2_{goal}(p) = ||p - p_{goal}||^2$ and K_{att} a given gain, so this potential is higher for far distances,

- and $U_{rep}(q)$ is the **repulsive potential field**, which generates a barrier around obstacles, computed as:

  $$U_{rep}(p) = \begin{cases} \frac{1}{2} K_{rep}(\frac{1}{d(p)} - \frac{1}{d_{max}})^2 & \text{if } d(p) \leq d_{max} \\ 0 & \text{if } d(p) > d_{max} \end{cases}$$

  being d_{max} a given distance threshold, so obstacles far away from the robot does not influence the potential field, and $d(p)$ the distance from the robot to the object so $d^2(p) = ||p - p_{obj}||^2$.

Having defined such potential field, it can be computed a **force field** at the robot position $F(p)$ (a two-element vector) as the gradient of the previous one:

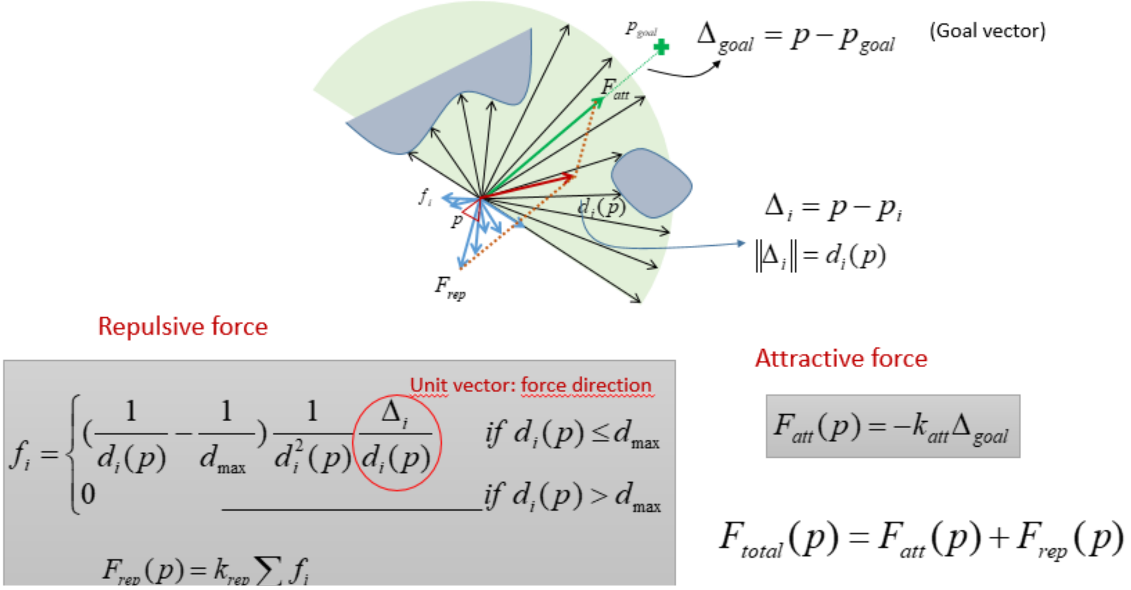
$$F(q) = -\nabla U(p) = -\nabla U_{att}(p) - \nabla U_{rep}(p) = \begin{bmatrix} \partial U / \partial x \\ \partial U / \partial y \end{bmatrix}$$

Where:

- $F_{att}(p) = -\nabla U_{att}(p)$ is also called the **attractive force** and
- $F_{rep}(p) = -\nabla U_{rep}(p)$ the **repulsive force**, so
- $F(p) = F_{att}(p) + F_{rep}(p)$.

Finally, the **robot speed** $[v_x, v_y]$ is set proportional to the force $F(p)$ as generated by the field.

The picture below illustrates all the elements in the computation of $F(p)$ (F_{total} in the image, colored as a red arrow):



8.2.3 Developing the Potential Fields method for Reactive navigation

It's time to develop our own Potential Fields method! For that, you first need to obtain the sum of the forces that apply at a certain robot position, computing for that the attractive and repulsive forces. Then, the total force can be retrieved, and it can be used to apply velocities to the robot wheels! (recall that $F(p) = F_{att}(p) + F_{rep}(p)$)

In [1]:
```
# IMPORTS
import numpy as np
from numpy import random
from scipy import linalg
import matplotlib
matplotlib.use('TkAgg')
from matplotlib import pyplot as plt

import sys
sys.path.append("..")
from utils.DrawRobot import DrawRobot
```

ASSIGNMENT 1: Computing the repulsive force

Let's start with the repulsive force (`FRep`) computation, which is the sum of the repulsive forces yielded by each obstacle close to the object. Recall that forces are 2-elements column vectors.

The `repulsive_force()` function below partially implements this computation. Notice that this function also plots a marker over the obstacles that have influence on this force, and store the handler of that plot in `hInfluentialObstacles` .

Recall that:

$$f_i = \begin{cases} \left(\frac{1}{d_i(p)} - \frac{1}{d_{max}}\right)\frac{1}{d_i(p)^2}\frac{p-p_i}{d_i(p)} & \text{if } d_i(p) \leq d_{max} \\ 0 & \text{if } d_i(p) > d_{max} \end{cases}$$

$$F_{rep}(p) = K_{rep}\sum_i f_i$$

In the code below, $p - p_i$ is stored in `p_to_object` , and $d(p)$ in `d` . Notice that for each f_i , the distance from the robot to the object $d_i(p)$ is a number, while $p - p_i$ is a vector.

In [2]:
```
def repulsive_force(xRobot, Map, RadiusOfInfluence, KObstacles):
    """ Computes the respulsive force at a given robot position

        Args:
            xRobot: Column vector containing the robot position ([x,y
            Map: Matrix containing the obstacles coordinates (size 2x
            RadiusOfInfluence: distance threshold for considering tha
            KObstacles: gain related to the repulsive force

        Returns: Nothing. But it modifies the state in robot
            Frep: repulsive force ([rf_x, rf_y]') (Column vector!)
            hInfluentialObstacles: handler of the plot marking the ob
    """
    p_to_object = xRobot - Map
    d = np.sqrt(np.sum(p_to_object**2, axis=0))
    iInfluential = np.where(d <= RadiusOfInfluence)[0]

    if iInfluential.shape[0] > 0:
        p_to_object = p_to_object[:, iInfluential]
        d = d[iInfluential]
        FRep = KObstacles * np.vstack([np.sum((1/d - 1/RadiusOfInflue

        hInfluentialObstacles = plt.plot(Map[0,iInfluential],Map[1,iI
    else:
        # Nothing close
        FRep = 0
        hInfluentialObstacles = None # Don't touch this! It is ok :)

    return FRep, hInfluentialObstacles
```

In [3]:
```
# TRY IT!
xRobot = np.vstack([[1],[2]])
Map = np.vstack([[1.1, 2.4, 3.5],[2.2, 1.4, 4.5]])
RadiusOfInfluence = 2
KObstacles = 200

FRep, handler = repulsive_force(xRobot, Map, RadiusOfInfluence, KObst

print ('Repulsive force:\n ' + str(FRep))
```

```
Repulsive force:
 [[ -7117.97589183]
 [-14205.83001107]]
```

Expected output:

```
Repulsive force:
 [[ -7117.97589183]
 [-14205.83001107]]
```

ASSIGNMENT 2: Retrieving the attractive force

Next, **you need to compute** the Attractive Force `FAtt` . Do it in the `attractive_force()` function below, taking into account that:

$$F_{att}(p) = -K_{att}d_{goal}(p)$$

Normalize the resultant Force by $||\Delta_{goal}||$ so its doesn't become too dominant. You can take a look at [linalg.norm() (https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.norm.html)](https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.norm.html) for that.

In [4]:
```
def attractive_force(KGoal, GoalError):
    """ Computes the attractive force at a given robot position

        Args:
            KGoal: gain related to the attractive force
            GoalError: distance from the robot to the goal ([d_x d_y]

        Returns: Nothing. But it modifies the state in robot
            FAtt: attractive force ([af_x, af_y]')
    """
    FAtt = -KGoal * GoalError
    FAtt /= np.linalg.norm(GoalError) # Normalization

    return FAtt
```

In [5]:
```
# TRY IT!
KGoal = 1.5
GoalError = np.vstack([[2.3],[1.4]])

FAtt = attractive_force(KGoal, GoalError)

print ('Attractive force:\n ' + str(FAtt))
```

```
Attractive force:
 [[-1.28129783]
 [-0.77992042]]
```

Expected output:

```
Attractive force:
 [[-1.28129783]
 [-0.77992042]]
```

ASSIGNMENT 3: Concluding with the Total Force

Finally you can compute the Total Force `FTotal` . **Do it in the main program below**, considering that:

$$F(p) = F_{att}(p) + F_{rep}(p)$$

In [6]:
```
def main(nObstacles=175,
         MapSize=100,
         RadiusOfInfluence=10,
         KGoal=1,
         KObstacles=250,
         nMaxSteps=300,
         NON_STOP=True):

    Map = MapSize*random.rand(2, nObstacles)

    fig, ax = plt.subplots()
    plt.ion()
    ax.plot(Map[0,:],Map[1,:],'ro', fillstyle='none');

    fig.suptitle('Click to choose starting point:')
    xStart = np.vstack(plt.ginput(1)).T
    print('Starts at:\n{}'.format(xStart))


    fig.suptitle('Click to choose end goal:')
    xGoal = np.vstack(plt.ginput(1)).T
    print('Goal at:\n{}'.format(xGoal))

    fig.suptitle('')

    ax.plot(xGoal[0, 0], xGoal[1, 0],'g*', markersize=10)

    hRob = DrawRobot(fig, ax, np.vstack([xStart, 0]), axis_percent=0.

    # Initialization of useful vbles
    xRobot = xStart
    GoalError = xRobot - xGoal

    # Simulation
    k = 0

    while linalg.norm(GoalError) > 1 and k < nMaxSteps:

        FRep, hInfluentialObstacles = repulsive_force(xRobot, Map, Ra
        FAtt = attractive_force(KGoal, GoalError)

        # Point 1.3
        # TODO Compute total (attractive+repulsive) potential field

        FTotal = FAtt + FRep
        #FTotal /= linalg.norm(FTotal)


        xRobot += FTotal
        Theta = np.arctan2(FTotal[1, 0], FTotal[0, 0])

        hRob.pop(0).remove()
        hRob = DrawRobot(fig, ax, np.vstack([xRobot, Theta]), axis_pe

        if NON_STOP:
            plt.pause(0.1)
        else:
            plt.waitforbuttonpress(-1)

        if hInfluentialObstacles is not None:
            hInfluentialObstacles.pop(0).remove()
```

```
        # Update termination conditions
        GoalError =  xRobot - xGoal
        k += 1
```

8.2.4 Understanding how the technique performs

As a brilliant engineer, you have to provide some indications to the **managers at Nirvana** about how the technique performs and its limitations, which has to be provided in the next ***Thinking about it***. The following code cells help you to execute the implemented technique with different parameters in order to retrieve the required information.

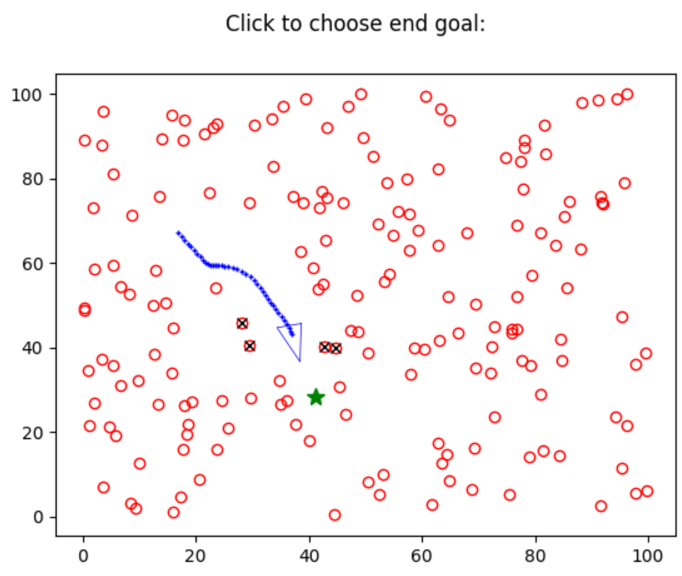
In [ ]:
```
# For considering different gains
main(KGoal=1, KObstacles=250)
```

In [ ]:
```
# For considering different number of obstacles
main(nObstacles=175)
```

Thinking about it (1)

Address the following points to gain insight into how the developed Potential Fields technique performs. You can include some figures if needed.

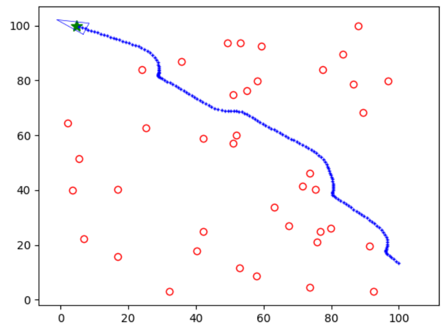
- Discuss the meaning of each element appearing in the plot during the simulation of the *Potential Fields reactive navigation*.

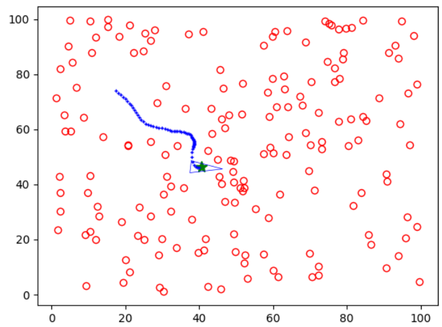


Los puntos rojos son los obstáculos representados en el mapa (que es el espacio bidimensional mostrado en la gráfica). La pose de nuestro robot está representado con el triángulo azul y el recorrido que realiza deja una estela del mismo color. Su objetivo (meta) es la estrella verde. Cuando un obstáculo entra en el rango de consideración de la fuerza repulsiva, aparece tachado con una equis negra.

- Run the program setting different start and goal positions. Now change the values of the goal and obstacle gains (`KGoal` and `KObstacles`). How does this affect the paths followed by the robot?

  Examples with different values for such constants:





El valor de KObstacles determina la fuerza con la que los obstáculos obligan al robot a apartarse de ellos. Si esta fuerza es pequeña, el robot se acercará más a ellos (con el riesgo que eso conlleva) pero será capaz de pasar por caminos que no podría hacer si fuera elevada. Tener una fuerza de repulsión elevada minimiza el riesgo de chocarnos, pero favorece los estancamientos en los estrechamientos.

Por otro lado, KGoal representa la fuerza con la que el objetivo tira del robot para que se aproxime, por lo que a valores elevados nuestro robot se acercará de manera más acelerada (más rápido y directo) hacia el mismo.

- Play with different numbers of obstacles and discuss the obtained results.

  En el caso de tener pocos obstáculos, el robot podrá llegar al objetivo sin tener que realizar muchas variaciones relevantes en su trayectoria y muy probablemente llegue hasta él. Puede darse el caso que tenga obstáculos en la línea recta que une la salida con la meta, o que tenga obstáculos a su alrededor y entren en la zona de consideración de la fuerza repulsiva, por lo que tendrá que variar su ruta.

  Si por el contrario tenemos demasiados obstáculos, aumentan las probabilidades de que se encuentre un mínimo local y nuestro robot se estanque en una posición que no es la meta. De igual manera, podemos ver que tiene que realizar muchos más cambios en su trayectoria y puede no llegar a la meta.

- Illustrate a navigation where the robot doesn't reach the goal position in the specified number of steps. Why did that happen? Could the robot have reached the goal with more iterations of the algorithm? Hint: take a look at the `FTotal` variable.

  Como se ha comentado antes, puede darse el caso de estancamiento del robot a causa de las fuerzas repulsivas y atractivas que se ejercen sobre el robot. Esta sumatoria de fuerzas está almacenada en FTotal y como podemos comprobar no varía una vez el robot se ha estancado, por lo que da igual si dejamos el algoritmo más iteraciones que no se va poder mover el robot (aunque podríamos darle un empujón en alguna dirección aletoria para ver si consigue escapar).