Introduction to Mobile Robot Path Planning

Unit 5: Artificial Potential Fields

- Summary -

Estimated time to completion: **3 hours**

This unit covers the fundamentals of the **artificial potential fields** (also known as APF) method applied to the mobile robotic path planning problem. Specifically, you will:

- Grasp the basics of 2D grid path planning with artificial potential fields
- Learn common mathematical models for attractive and repulsive forces
- Build a total potential function
- Write a gradient descent algorithm to create a path
- Get a grip on tuning your algorithm parameters
- Test your work using ROS and Gazebo

- End of Summary -

5.1 Introduction

How fast and accurately a path planning algorithm can produce a route is a highly sought-after characteristic relevant to many use cases in robotics. Why is that? The higher the planning rate, the better the robot can react to changes in the environment and/or perform in large maps.

Previously we saw techniques, such as Dijkstra and A*, which are precise but have high memory requirement and are CPU demanding. Then we turned our attention to the basic RRT implementation, which can be faster but is imperfect, as it creates a zig-zag-like path. **Artificial potential fields** or **virtual potential fields** is yet another approach initially developed precisely because it provides a very fast, direct approach for guiding a robot towards a goal while keeping a safe distance from obstacles.

The basic idea here is to model the map of the environment into a mathematical function f (x, y). It has high values near to obstacles and has its minimum value at the goal location. Once you have constructed such a function, you can use its gradient or slope to build a path to the goal.

One way to visualize the action of this algorithm is to think of some hilly terrain. The planned path will follow the direction that a ball, placed at the robot's start location, would go. Due to the effect of the slope, the ball will roll "downhill" from higher to lower points until it reaches the lowest point.

During this unit, I will walk you through creating an artificial potential field in Python and use it for robotic global path planning. Then you will get to test your implementation thoroughly using RVIZ and Gazebo....and as always, it is going to be great!

Let's get started!

5.2 Creating Potential Fields

Do you want to build an artificial potential field?

A typical artificial potential field requires modeling two different types of forces involved:

- An **attractive field** generated by the goal, and
- A **repulsive field** generated by obstacles

The **total potential field** is generated when both forces, attractive and repulsive, are combined.

Mathematically you let denote U_{total} the total potential field, U_{att} the attractive field, and U_{rep} repulsive field. We also use the letter X to represent a grid map of size (x, y):

$$U_{\text{total}}(X) = U_{\text{att}}(X) + U_{\text{rep}}(X)$$

This formula is telling you that you can divide the construction of an artificial potential field into two separate parts:

- Create an attractive potential field
- Create a repulsive potential field

Once you're done, you have to add both fields up.

Before you move forward, take a small step back and try to grasp the basic idea of a **field**. What is a field? I mean, what is a field *code-wise speaking*?

A **field** can be defined as a spatial representation of the robot's environment. For instance, you have seen how a 2D environment can be represented using a grid map. Conversely, a **potential field** can be coded as a grid map of size (x_max, y_max) that keeps potential values in each of its grid cells. Then each of its grid cell values represents a **potential force** for that particular position in the robot's environment. Let's see that in action.

- Exercise 5.2.1 -

Open the file where you will write your code:

1. Use the code editor window to go into the folder `catkin_ws/src/path_planning_course/unit5`
2. Open `/scripts` and then open the file **unit5_exercise.py**

This exercise aims to complete the `populate_attractive_field(...)` function shown below and add it to that file.

Tasks

- Start by creating a list `field` of size `height * width` that represents the grid map as a 1D array. Pre-fill it with zeros.
- Run a per-column loop nested inside a per-row loop to selectively push data into the `field` by calling the provided function `random_force()`.
- You can use the syntax `field[row + width * col]` to fill each element of `field` correctly.
- Finally, make the function return `field` and save.

Note that `populate_attractive_field(...)` has as parameter `goal_xy`, the goal's (x,y) coordinates, which you didn't need in this exercise, but you will use in the next one!

In [1]:
```
# complete the function below, the add the code to unit5_exercise.py

def populate_attractive_field(height, width, goal_xy):
    """
    Creates an attractive field
    returns: 1D flat grid map
    """




    pass
```

Before running your code you will have to build your catkin workspace with `catkin_make` from the top folder of your catkin workspace.

▶ Execute in WebShell #1

In [ ]:
```
cd ~/catkin_ws
```

In [ ]:
```
catkin_make; source devel/setup.bash
```

Run your code by executing this command and wait for the **Graphical Tools** screen to open:

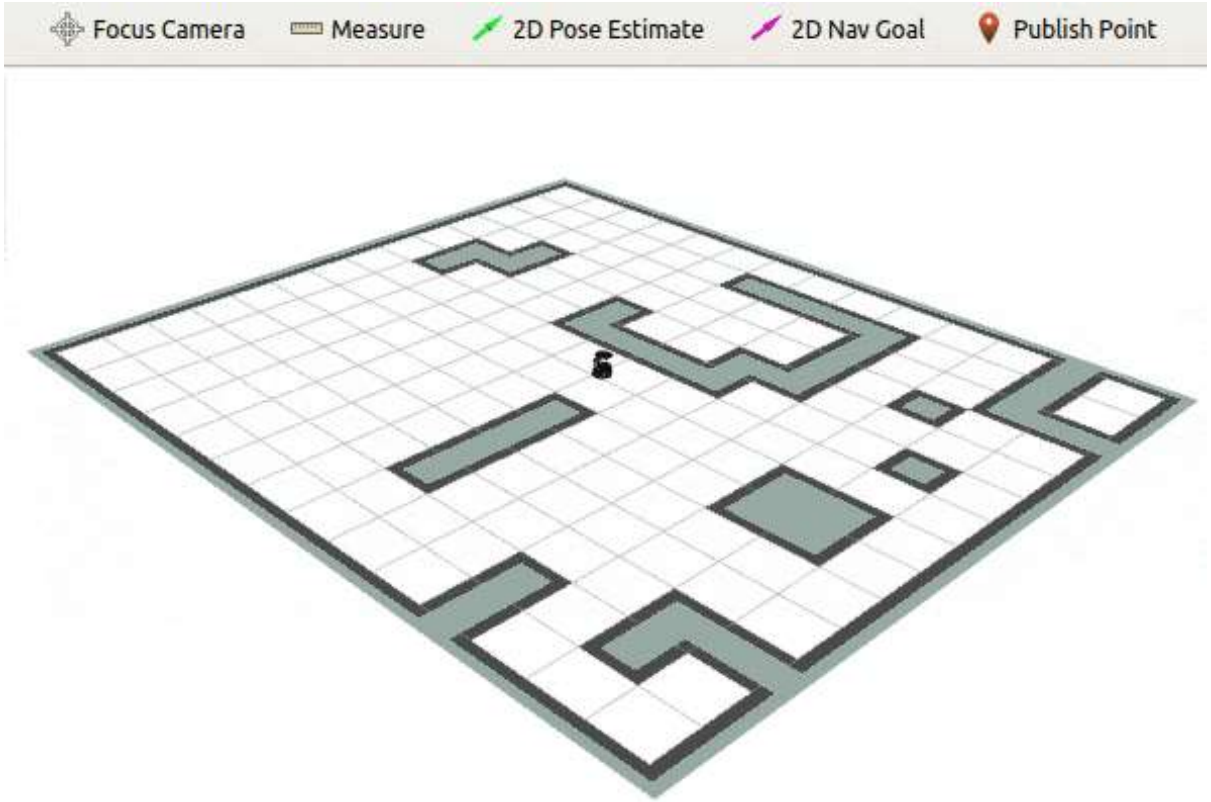
▶ Execute in WebShell #1

In [ ]:
```
roslaunch unit5 unit5_exercise.launch
```

Wait until the **'Ready to accept commands!'** message is displayed. Otherwise, the 2D Nav Goal button will have no effect.

Use the **2D Nav Goal** button to set a navigation goal through RVIZ.

Expected result



Yay! You created your first potential field! Don't worry if it looks like a mess. Just as expected, this potential field leads nowhere because of the many ups and downs caused by the random potential forces in it.

Before continuing, please shut down the script by pressing **CTRL + C** on the WebShell that executed it.

Now it's time to move on and build an attractive force field, something the robot can use!

- End of Exercise 5.2.1 -

5.3 Attractive Potentials

In the attractive field, the basic idea is to apply an attraction force that can lead either the robot or the robot's path towards the goal.

There are several choices on how to implement an attractive potential function. In all of these methods, **the potential values in the attractive field decrease as it gets closer to the goal**. This way, you guarantee that the lowest value is placed at the desired goal location, and there is nothing lower than that. Similarly, the values in the field should increase as you move to positions that are further away from the goal.

In this section, you will examine two different types of commonly used attractive potential functions:

- conical potential function
- quadratic potential function

Conical Potential Function

In textbooks, the **attractive potential field** produced by a **conical potential function** is often written as:

$$U_{\text{att}}(X) = \mathrm{k} \left\| X_{\text{current}} - X_{\text{goal}} \right\|$$

Where k is the attractive potential constant, a positive scaling parameter that you define, $\left\| X_{\text{current}} - X_{\text{goal}} \right\|$ denotes the **distances** between each grid cell and the goal. You model this field by iterating over each of its grid cells and calculating the distance of each to the goal.

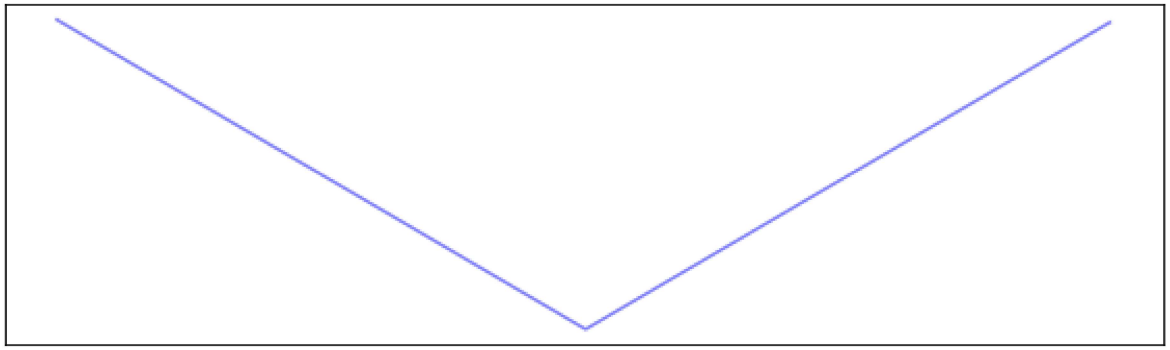
To calculate that distance, you can use the standard Euclidean distance between two points (x_1, y_1) and (x_2, y_2):

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

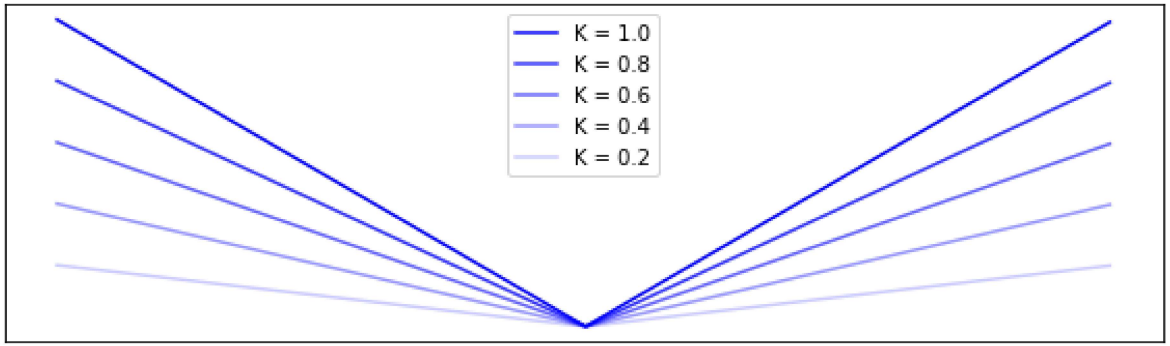
Where (x_1, y_1) position coordinate of the current grid cell and (x_2, y_2) the goal grid cell coordinate.

Perhaps the easiest way to visualize the conic potential function is this very simple side-view of a robot in a one-dimensional map. In this map, the robot can only move along the x-axis, forward or backward. Thus, the conical potential function on this one-dimensional map can be visualized as a cone whose lowest point is directly over the goal and then opens upwards to both sides.

The slope/derivative of the lines represents the intensity of the attractive potential. The attractive potential is zero at the goal position, and if you move to the right or the left, the potential intensity remains constant.



To generate a stronger or weaker attractive force across the full extent of the attractive field, you have to change the value of the attractive potential constant k, as shown in the image below:



- Exercise 5.3.1 -

In the previous exercise you completed, `populate_attractive_field(...)`, which in turn called `random_force()` to set a value for each grid cell in the attractive field. In this exercise, you will complete `conical_attractive_force(...)` and use it instead of `random_force()`.

Have a look at the block of code below. The function shown takes two grid cells as two lists with [x, y] coordinates and the parameter `K`, representing the attractive potential constant. Complete the function's body by completing the following tasks:

Tasks

- Calculate the Euclidean distance between `current_cell` and `goal_cell`
- Add a return statement that includes said distance times the scaling parameter `K`
- Add `conical_attractive_force(...)` to the program file
- Modify `populate_potential_field(...)` to call `conical_attractive_force(...)` instead of `random_force()`
- Note that you have to pass `goal_xy`, the goal's (x, y) coordinates, when calling `conical_attractive_force(...)`
- Save

In [1]:
```
def conical_attractive_force(current_cell, goal_cell, K = 1.0):
    """
    Calculates the linear attractive force for one grid cell with respect to the target
    current_cell: a list containing x and y values of one map grid cell
    goal_cell: a list containing x and y values of the target grid cell
    K: potential attractive constant
    returns: linear attractive force scaled by the potential attractive constant
    """



    pass
```

Test your code by running it and then waiting for the **Graphical Tools** window to open.

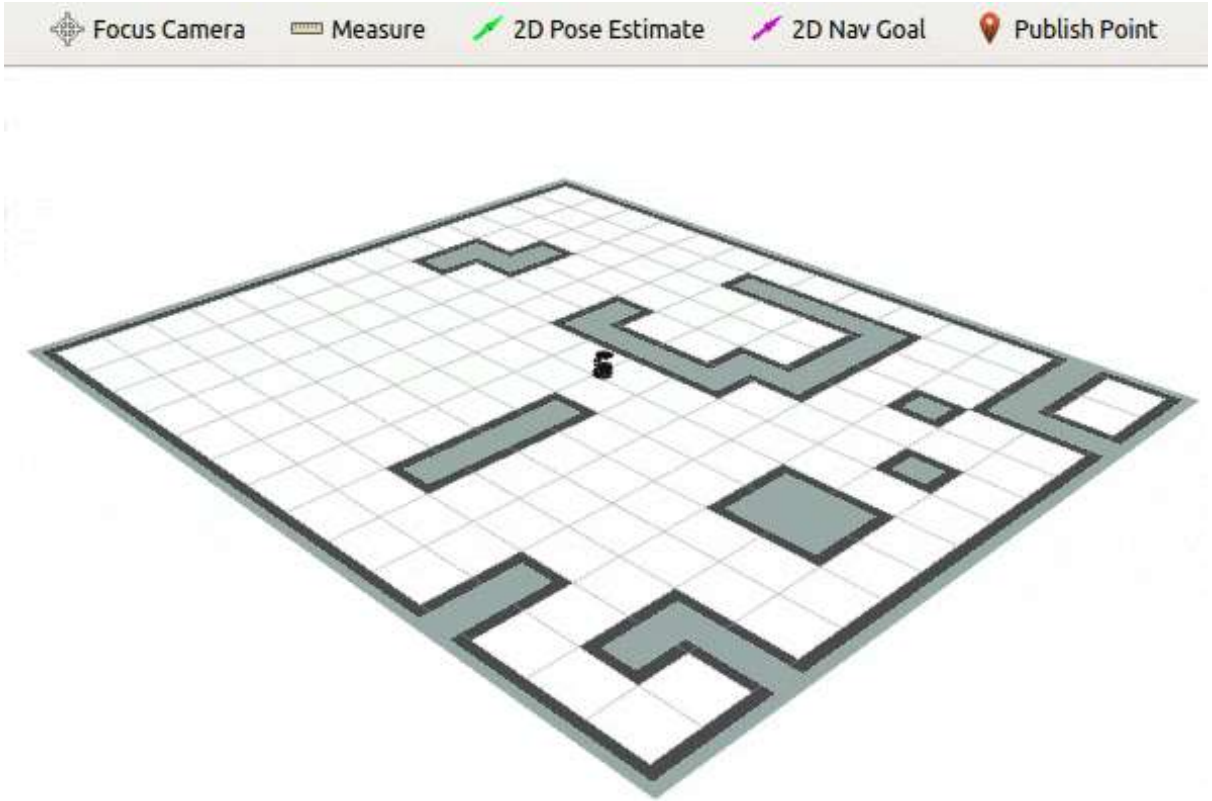
▶ Execute in WebShell #1

In [ ]:
```
roslaunch unit5 unit5_exercise.launch
```

Wait a few seconds. The message *Ready to accept commands!* will be displayed.

Then use the **2D Nav Goal** button to set a goal position through RVIZ.

Expected result



By examining the attractive field's surface, you can see how the conical shape provides the path to follow to get to the goal. If you find it difficult to visualize, try rotating the view with the mouse.

Note, the robot will not move. This is the expected behavior.

Tuning

You can try out different values of `K` to see what difference it makes. This implementation is designed to work with nav_msgs/OccupancyGrid messages, which can only take values from -1 to 100. Therefore, if you set the slope curve to be too steep, the functions value will max out at 100, and the cone will have a portion cut out. Try it out!

Now you can shut down the script by pressing **CTRL + C** on the WebShell that executed it.

- End of Exercise 5.3.1 -

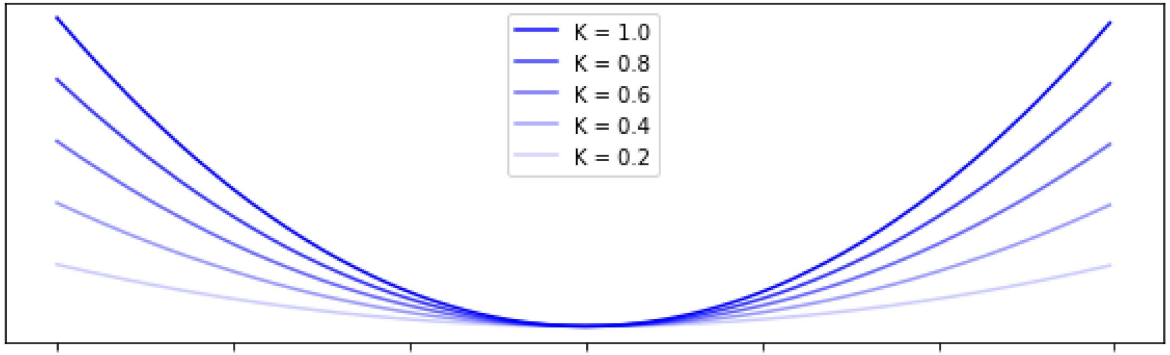
Quadratic Potential Function

Another common function used to construct an attractive potential field is the quadratic potential function or parabolic potential function. You might come across a formula similar to the one below to describe its attractive force field:

$$U_{\text{att}}(X) = \text{k} \left\| X_{\text{current}} - X_{\text{goal}} \right\|^2$$

Here k is again the attractive potential constant that you can tune, and $\left\| X_{\text{current}} - X_{\text{goal}} \right\|$ is, just as before, the **distances** between each grid cell and the goal. What's new is the additional squared symbol (^{2}) which indicates that each distance value in the field gets **squared**.

Let's see how this function varies over a one-dimensional map and how different values of `K` will modify it:



The image above shows a side view of a 1D potential field where it can be seen that the surface has a wide bottom that arcs up as the distance from the goal increases, creating a bowl-shaped figure. A path created following the gradient of this function downhill will guide a robot straight towards the lowest point, the goal's location.

- Exercise 5.3.2 -

Now let's turn back to coding.

You should still have the file from the previous exercise open. If you don't, go ahead and open it now.

Tasks

- Your first task is to fill out the function body of `quadratic_attractive_force(...)` shown below.
- It should return the distance between `current_cell` and `goal_cell` **squared**.
- Add this function to the main script **unit5_exercise.py**.
- Modify `populate_potential_field(...)` to call `quadratic_attractive_force(...)` instead of `conical_attractive_force(...)` .
- Save.

In [ ]:
```
def quadratic_attractive_force(current_cell, goal_cell, K = 0.015):
    """
    Calculates the quadratic attractive force for one grid cell with respect to the target
    current_cell: a list containing x and y values of one map grid cell
    goal_cell: a list containing x and y values of the target grid cell
    K: potential attractive constant
    returns: quadratic attractive force scaled by the potential attractive constant
    """



    pass
```

- End of Exercise 5.3.2 -

- Exercise 5.3.3 -

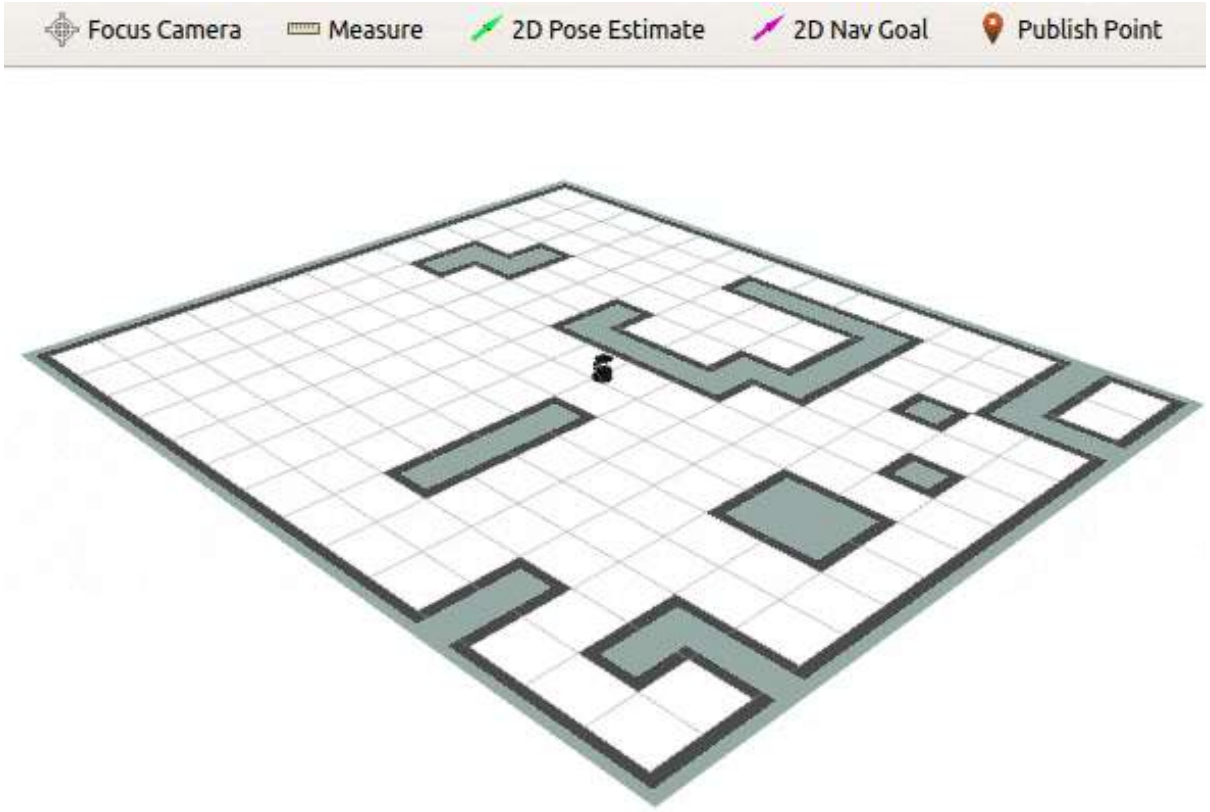
Testing time! As usual, run your code using this command:

▶ Execute in WebShell #1

In [ ]:
```
roslaunch unit5 unit5_exercise.launch
```

Wait a few seconds until you see the message *"Ready to accept commands!"*, then click the **2D Nav Goal** button to set the desired goal.

Expected result



Tuning:

Pay special attention to tuning the `K` parameter. For example, what do you think would happen if you set the value too low?

Important: always stop the program execution using **Ctrl+C** on the original terminal/WebShell before continuing.

- End of Exercise 5.3.3 -

Now it's time to talk about repulsive forces.

5.4 Repulsive Field

In addition to guiding the robot towards the goal, you want to keep the robot at a safe distance from map obstacles. This is why you need repulsive forces.

The basic principle is that the repulsive field pushes the robot path away from obstacles while extending towards the goal.

An ideal repulsive field should have the following properties:

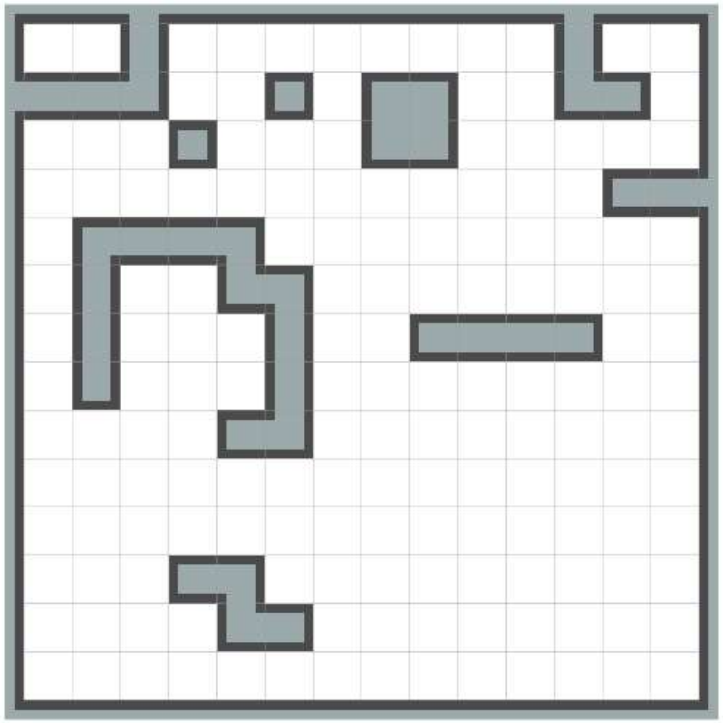
- Never allow a robot to collide with any obstacle
- Reinforce keeping an additional safety distance whenever possible
- Have a limited range of influence

If you think about it, that description matches up pretty close to what a **costmap** does. In fact, before continuing, let me explain why and how **costmaps** and **repulsive fields** are related.

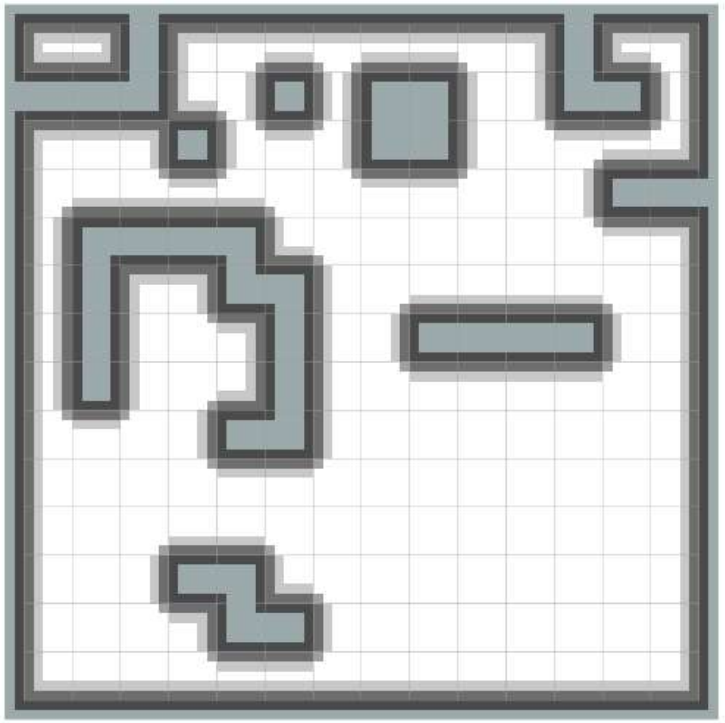
The Relationship Between Costmaps and Repulsive Fields

Perhaps you have heard about **costmaps** in ROS, and maybe you have even tuned **costmap** parameters. Regardless of your previous experience, I can assure you - you have certainly used **costmaps** in this course. While a regular map consists only of occupied, free and unknown grid cells, costmaps contain costs for each cell, as the name suggests. These costs have the highest value as occupied cells and decrease the further away a grid cell is from an obstacle.

Regular map



Costmap



It turns out that **costmaps** are mathematical functions of the environment, which can be modeled using the same procedure steps used to create **repulsive fields**. In fact, costmaps and repulsive fields share the same three properties (see list above). The difference between both solely relies on what the grid cell represents, more specifically, how the values are **interpreted**:

A grid cell value of a **costmap** represents a **movement cost**, whereas in a **repulsive field** it represents a **repulsive force** value.

In the remaining parts of this unit, it will seem as if we were using the terms **costmap** and **repulsive field** almost as synonyms, but now you know that these two concepts **interpret values differently**. Because of this difference, each term is used in a different context and should not be swapped.

Let's go back into coding to see if you can use the costmaps provided by ROS to create a repulsive potential field. After that, you are going to tune some parameters again!

- Exercise 5.4.1 -

In this exercise, you will take the grid cell values in the costmap provided by the ROS Navigation Stack and reuse them as repulsive field values.

Tasks

Complete these tasks by modifying the function `costmap_callback(costmap_msg)` from our **unit5_exercise.py** script from the previous exercises:

- Create a deep copy of the incoming `costmap_msg` and assign it to `repulsive_field`
- Change `repulsive_field.data` to a list, as you will have to modify it next
- Change all values inside `repulsive_field.data` that equal -1 to 100 (set unknown grid cells as occupied)
- Publish `repulsive_field` using `repulsive_field_publisher`

In [1]:

```
# Modify the callback function below to create a Repulsive Potential Field out of a copy of the Costmap

def costmap_callback(costmap_msg):
    global attractive_field, repulsive_field, total_field
    total_field = copy.deepcopy(costmap_msg)
    show_text_in_rviz('Ready to accept commands!')
    attractive_field = copy.deepcopy(costmap_msg)
    attractive_field.data = [0] * attractive_field.info.height * attractive_field.info.width

    ### add your code starting here ###

    # Create a Repulsive Potential Field from a Costmap


    # Change grid map values from unknown to occupied


    # Publish repulsive field
```

To run your code, type in the command shown below:

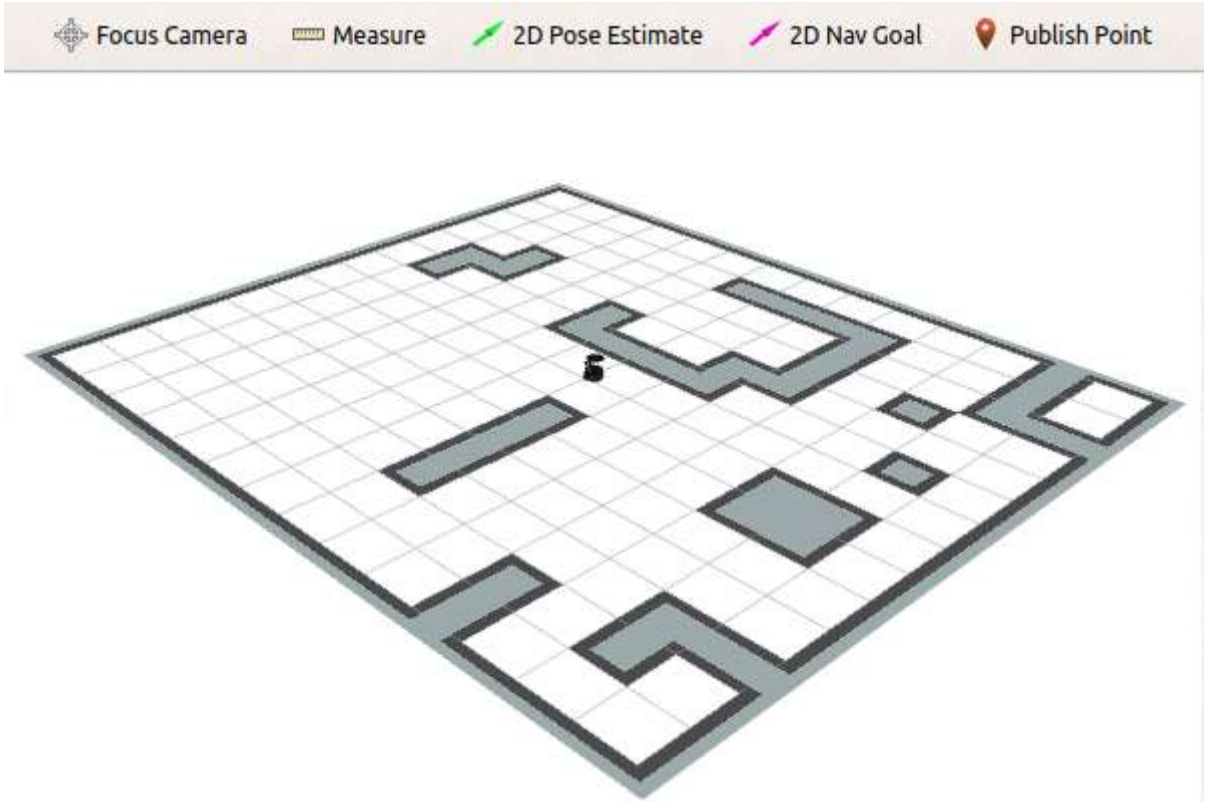
▶ Execute in WebShell #1

In [ ]:
```
roslaunch unit5 unit5_exercise.launch
```

In RVIZ, ensure that the checkmark next to "attractive field" is deactivated and "repulsive field" is set.

You should see the repulsive field as soon as its checkbox is active.

Expected result



The 3D surface represents the resulting repulsive potential field, and the potential forces are the values on the z-axis. The colors show the intensity of the repulsive force: in red regions, the repulsive force is the highest, while in the purple grid cells, there is no repulsive force at all.

Don't forget to close the running script before moving on.

- End of Exercise 5.4.1 -

It's great that ROS already provides costmaps, which you can reuse to create a repulsive potential field. So let's get a bit deeper into the building blocks of repulsive potential fields and their parameters.

5.5 Repulsive Fields Parameters

Repulsive fields can be viewed as the result of two independent building blocks:

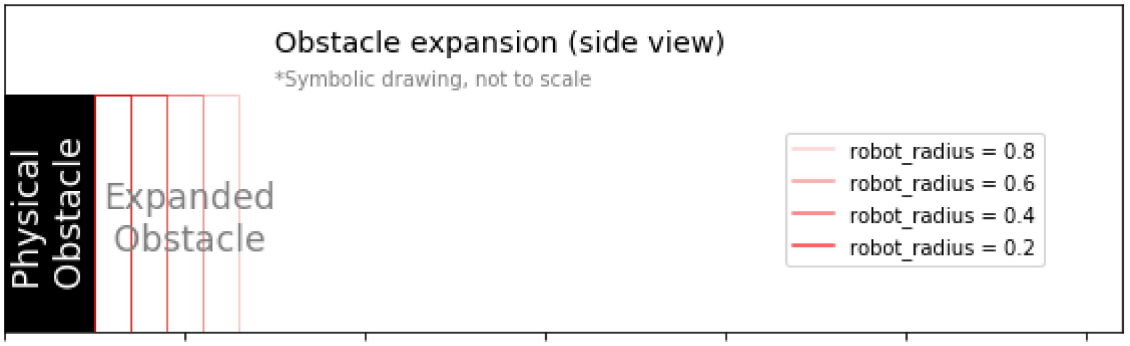
1) A virtual expansion of the size of map obstacles, making them thicker than the physical obstacle.
2) An additional buffer zone that decreases with growing distance from the expanded obstacle.

In conjunction, these two steps, virtual expansion and additional buffer, are commonly called "obstacle inflation" in the [costmap_2d package documentation (http://wiki.ros.org/costmap_2d)](http://wiki.ros.org/costmap_2d).

1) Expanding the size of map obstacles

Typically, obstacles are expanded by at least half of the robot's width. Why? Because a robot's **path** indicates positions where **the center of the robot's base** will be moving. By expanding an obstacle by at least half of the robot's width, you avoid the **path** that gets into the zone of the expanded obstacle, and by implication, the robot's body hits the physical obstacle.

The image below depicts a side view of an obstacle and how big it would appear to the path planner at four different expansion sizes.



The robot center cannot enter an area occupied by the **thicker obstacle** (grid cells inside the red area). If you expand the map obstacles by at least half of the robot's width, you address concern nr. 1: ***"Never allow collision."***

- Exercise 5.5.1 -

In this exercise, you will make obstacles "thicker."

Start by modifying costmap two parameters: `robot_radius` and `inflation_radius` .

Complete the tasks below to see it in action:

Tasks

- Switch back to the code editor window and open the file **costmap_common_params.yaml** (if you haven't already).
- Modify a `robot_radius` and `inflation_radius` , and set them, for instance, to `1.0` .
- Simultaneously increase both to increase the size of map obstacles.

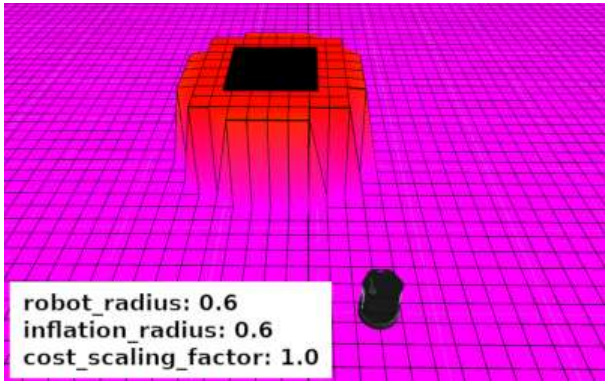
Start the exercise launch file again.

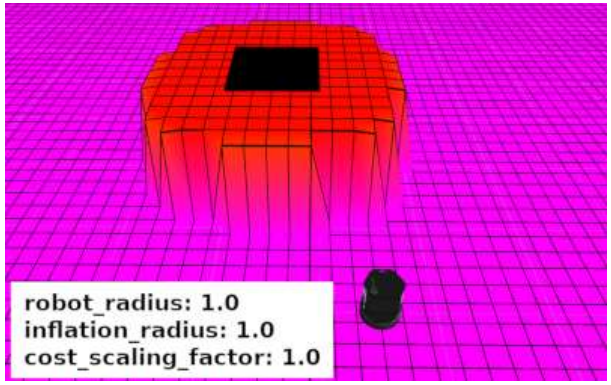
▶ Execute in WebShell #1

In [ ]:
```
roslaunch unit5 unit5_exercise.launch
```

Expected result

The images given below show the expanded obstacle according to two different configuration settings:





Tuning:

- Try to reproduce the examples shown by the images above for yourself.
- Then modify `robot_radius` and `inflation_radius` again. For instance, you could use 0.0, 0.4, or even 2.0.
- Double-check that `robot_radius` and `inflation_radius` both have the **same value**.

What happens to the gaps in between map obstacles when a large amount increases their "virtual size"?

Having completed this exercise, set the value of `robot_radius` back to `0.15`. Be sure to stop the running processes before moving on.

- End of Exercise 5.5.1 -

2) Creating an additional buffer zone

The buffer zone signals that it's preferable to keep an even larger distance from obstacles when possible. If that's not possible, it's okay that the robot's distance to obstacles is lessor. This is achieved by using a decaying function, or a function whose value declines as its distance from an obstacle increases.

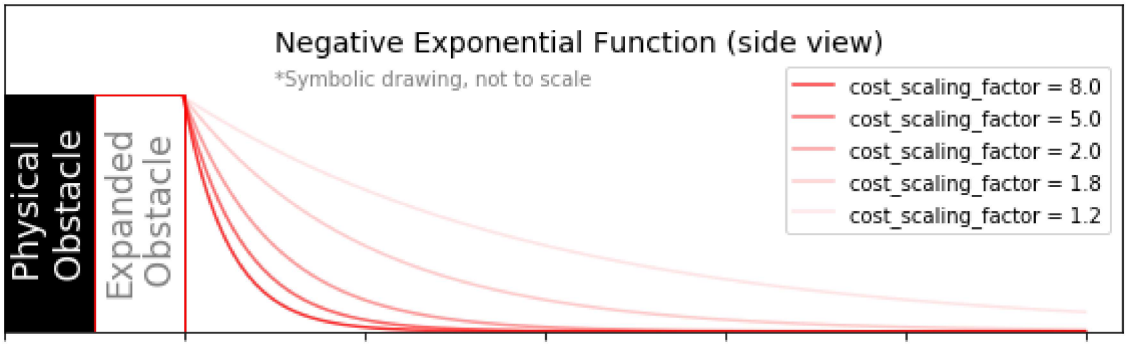
There are different forms of decaying functions. For example, the **negative exponential function** is the one implemented in ROS's Costmap_2d package:

$$e^{k*(r-d)}$$

Where:

- e is a mathematical constant called natural number or Euler's number, approximately equal to 2.71828.
- k is a configurable parameter that determines the **curvature of the slope**. In ROS, this parameter is called `cost_scaling_factor`.
- r indicates the **radius of the expanded obstacle**. In ROS, this parameter is set by `robot_radius`.
- d is the **distance to the closest obstacle**. In ROS, the parameter `inflation_radius` is used to set the **maximum distance of influence** of the buffer zone.

Have a look at the plot below to see how the **slope** of the negative exponential function varies with different values of `cost_scaling_value`. This plot is over one axis only, to more easily visualize what is happening.



From the image above, you can see that setting the `cost_scaling_value` higher will make the buffer zone decay faster.

Also, remember that the `inflation radius` must be set to a value larger than the parameter `robot_radius` for creating the *buffer zone*.

Note that in the costmap_2D package, *buffer zones* are not an official term. Instead, ROS's documentation uses the more general term *inflated obstacles*, referring to both the expanded obstacle and the additional buffer zone around it.

- Exercise 5.5.2 -

In this exercise, you will create and tune a *buffer zone* and control how far away its repulsive force exerts influence.

To complete this exercise, you must edit the `costmap_common_params.yaml` file again.

Tasks

- Keep `cost_scaling_factor: 5.0` and `robot_radius: 0.15` fixed.
- Pick a new value for the `inflation_radius` , save the configuration file and run the script to see the effect in RVIZ.

▶ Execute in WebShell #1

In [ ]:
```
roslaunch unit5 unit5_exercise.launch
```

Wait until the **Graphical Tools** opens. Can you tell what has changed?

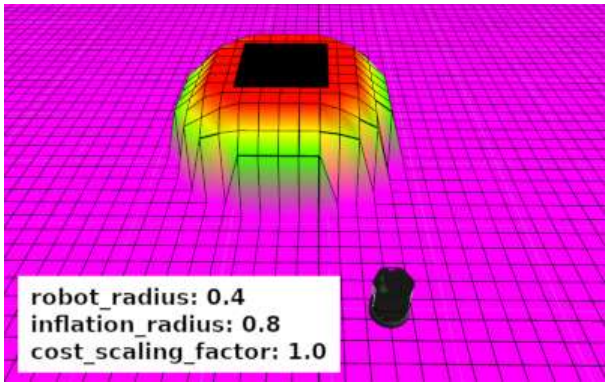
Shut down the script by pressing **CTRL + C**.

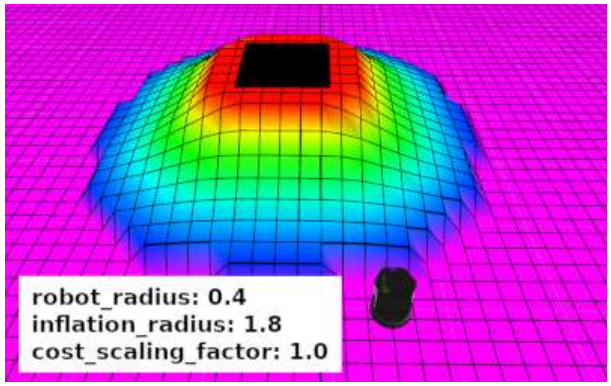
- Then repeat: pick a different `inflation_radius` value, save the file and see the effect in RVIZ.
- Note that in this exercise, `inflation_radius` must always be set to a value larger than `robot_radius` .

Please do the exercise because this way you will learn.

Expected result

The size of the buffer zone changes for different `inflation_radius` parameter values, as shown below.
Note: the images below are just an example and show a `robot_radius: 0.4` and `cost_scaling_factor: 1.0` . Regardless of that, focus on observing different values for the `inflation_radius` parameter!





Areas that are not affected by the repulsive force show up in purple. Can you confirm that anything outside `inflation_radius` is not affected by the repulsive force? Also, what happens to the gaps in between map obstacles when the `inflation_radius` parameter is large?

- End of Exercise 5.5.2 -

- Exercise 5.5.3 -

In this exercise, you will examine the effect of the `cost_scaling_factor` on the repulsive force.

Tasks

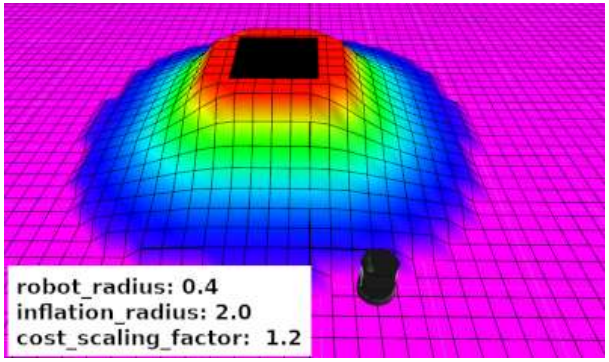
- Switch back to the code editor window and the file `costmap_common_params.yaml`
- Set the `inflation_radius` to a somewhat large value, for instance, `2.0`
- Try different values for the `cost_scaling_factor` (in the range 1 to 10).
- You will have to pick a value, save the file, run the script, and open RVIZ to see the effect. Then repeat for a different value.

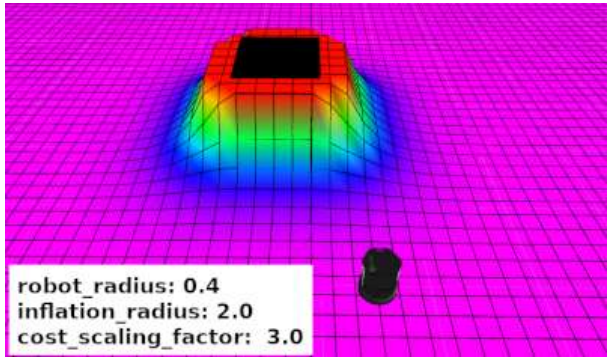
▶ Execute in WebShell #2

In [ ]:
```
roslaunch unit5 unit5_exercise.launch
```

Expected result

Examples for two different values for that parameter:





As you can see, the `cost_scaling_factor` determines the curvature of the slope in the buffer zone. And since it is multiplied by a negative in the formula ($r - d$ to be precise, which is a negative value), increasing its value will decrease the resulting cost values.

Remember to close the script.

Before continuing, please set the parameters back to the original values:

- `robot_radius: 0.15`
- `cost_scaling_factor:  5.0`
- `inflation_radius:     0.4`

- End of Exercise 5.5.3 -

Note: If you are interested in learning more about how costmaps are created, check out The Construct's "Basic Mapping Methods and Map Representations" course to explain this process in detail.

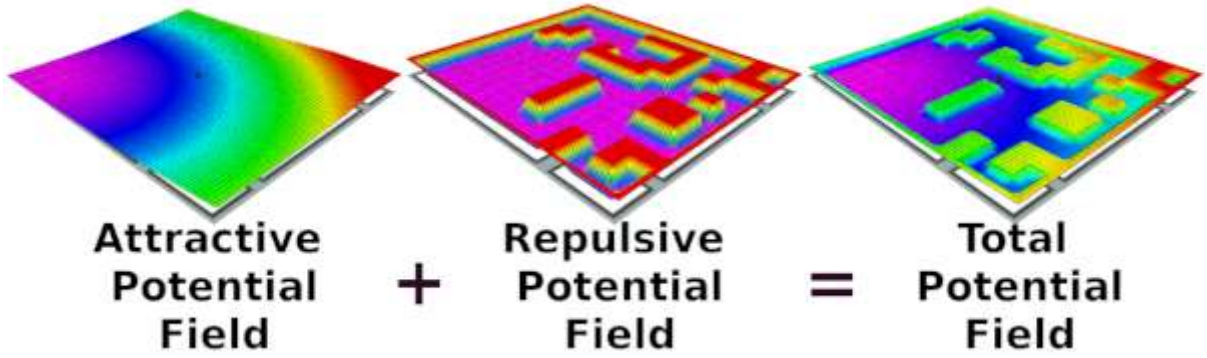
5.6 Generating the Total Potential Field

Now you are in good shape to build the total otential field.

The **total force** acting on the robot is the sum of the forces attracting the robot to the goal and the forces repelling the robot from the obstacles. You can extend this to the whole map and get that the **total potential field** is the sum of the attractive field plus the repulsive field.

$$U_{\text{total}}(X) = U_{\text{att}}(X) + U_{\text{rep}}(X)$$

The images below show how the total potential field in a two-dimensional environment map is obtained by adding the attractive potential field and the respulsive potential field. The resulting total potential field is represented by the 3D surface where the lowest point in the total potential field corresponds to the desired goal location.



- Exercise 5.6.1 -

In this exercise, you will create a total potential field.

This time re-open the file **unit5_exercise.py**
You will have to modify `new_goal_callback(data)` to calculate a new total potential field each time a new goal position is received:

Tasks

- Fill in `total_field.data` by **adding each element** in `attractive_field.data` to the corresponding element in `repulsive_field.data` .

  For example, if `attractive_field.data` has the values [0 0 1 2 1 0 0], and `repulsive_field.data` [5 0 0 0 0 0 5], the contents of `total_field.data` should be [5 0 1 2 1 0 5].

- Use the provided function `rescale_to_max_value(..)` to scale `total_field.data` to the range [0-100]. This will keep the data within the maximum value allowed by a message of type nav_msgs/OccupancyGrid.

In [ ]:

```
# Modify the callback function below to create a Total Potential Field

def new_goal_callback(data):
    global attractive_field, repulsive_field, total_field
    # Convert goal position from Rviz into grid cell x,y value
    goal_grid_map = world_to_grid_map(data.pose)
    # Populate attractive field data
    unbounded_attractive_field = populate_attractive_field(attractive_field.info.height, attractive_field.inf
    # Limit the max value of a grid cell to 100, the max value allowed by a message of type nav_msgs/Occupanc
    attractive_field.data = [100 if x > 100 else int(x) for x in unbounded_attractive_field]
    # Publish attractive field
    attractive_field_publisher.publish(attractive_field)

     ### add your code starting here ###
     # Calculate and publish a Total Potential Field
```

- End of Exercise 5.6.1 -

- Exercise 5.6.2 -

Now, let's start the ROS node again and visualize what the Total Potential Field looks like.

▶ Execute in WebShell #1

In [ ]:

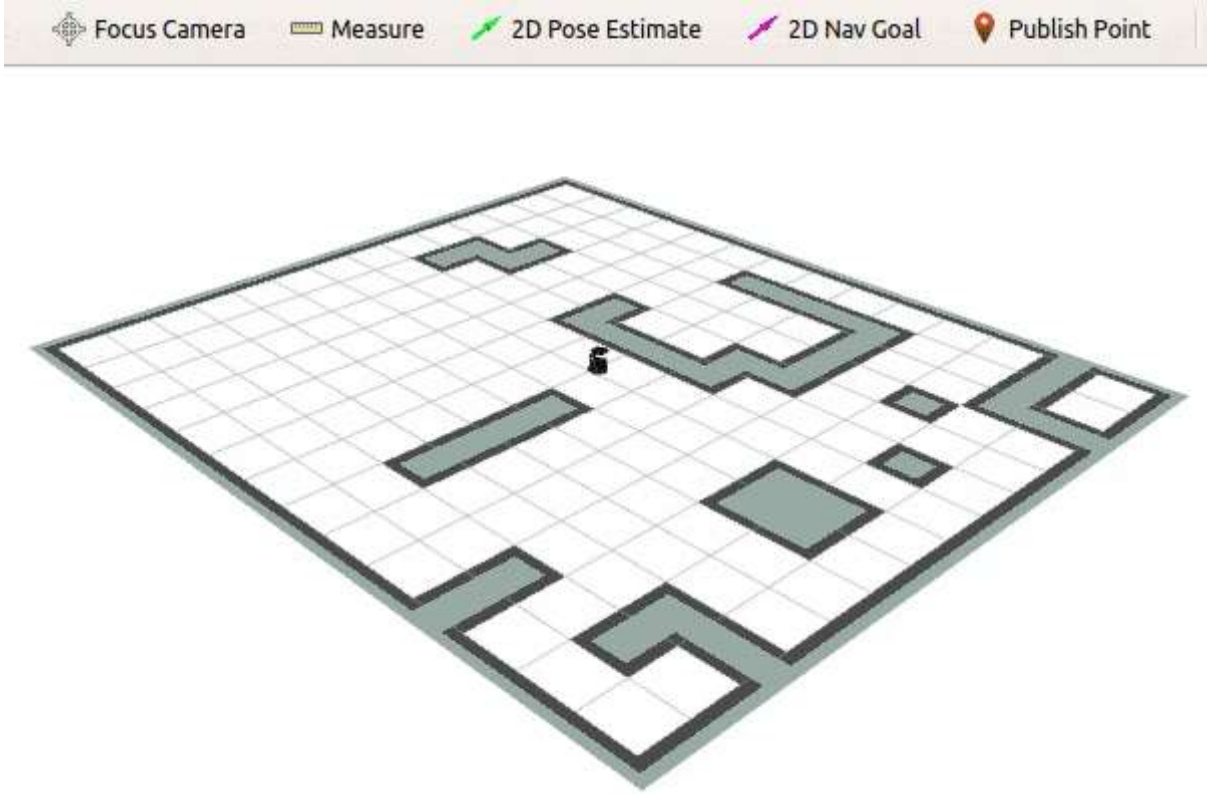
```
roslaunch unit5 unit5_exercise.launch
```

Once the **Graphical Tools** screen has fully loaded:

- Activate the visualization of **Total Field** by activating the corresponding checkmark in RVIZ.
- Deactivate the visualization of the other two potential fields for a better view.

- Set the goal to any map location.

If everything worked, you should see how the Total Potential Field surface appears once you enter a goal position:

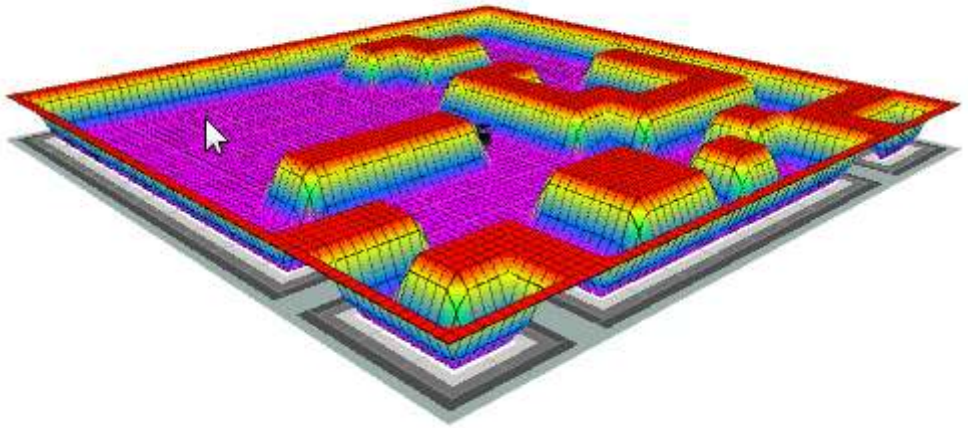


- End of Exercise 5.6.2 -

That's awesome! But you haven't built a path yet! So keep on going, and you will learn how this can be done!

5.7 Gradient Descent

To create a path, you will have to write a gradient descent algorithm. Once you're finished, it'll look something like this:



The black segments on the colored surface show the path constructed from the robot's location to the goal by following the gradient on the total potential field. As you can see, the total potential field guides the expansion of the path towards the goal.

Let's dissect the gradient descent algorithm.

The basic premise of this algorithm is simple: take repeated steps in the direction of the steepest descent. There are only three main steps involved:

1. Compute the gradient of the latest path waypoint
2. Determine a new path waypoint based on that gradient
3. Add it to the path

These steps get repeated over and over.

Computing the Gradient

As you know, the gradient or derivative of a function describes the slope or incline at any given point, and its sign describes the direction of *ascent*.

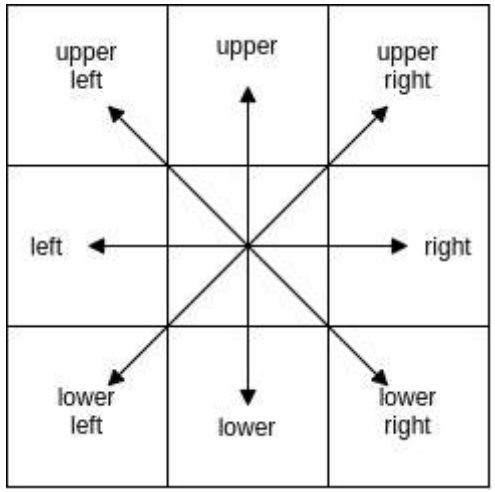
When working continuous functions, you resort to calculus and compute the gradient using the first derivative of the total potential function. Therefore a common notation used to describe the computation of a gradient vector of a two-dimensional field is the following:

$$\nabla F(x, y) = \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{pmatrix}$$

However, when working with grid maps, you only have discrete values and not a continuous function to work with. So the solution is to approximate the required gradient by calculating the discrete value differences between adjacent grid cells.

Suppose you only consider the four adjacent neighbors. In that case, you use the upper and lower grid cells to approximate the gradient in the y-axis direction and the left and right grid cells to calculate the gradient in the x-axis direction.

In your case, you will consider all eight adjacent neighbors; therefore, you will also calculate the differences in the direction of the diagonals (e.g., the differences between the upper left and lower right, as well as upper right and lower left grid cells).



- Exercise 5.7.1 -

For this exercise, you will have to open the **gradient_descent.py** file. You will see that this file already contains a few supporting functions for the main gradient descent algorithm that you will code later on.

In this exercise, you will be adding one additional function that computes the approximate gradient for all eight adjacent neighbors of a grid cell.

Task

1. Inside the code editor window, go to the folder `catkin_ws/src/path_planning_course/unit5/scripts` .
2. Open the **gradient_descent.py** file and paste the code block below.
3. Save

In [ ]:

```
def get_neighbours_and_gradients(index, width, height, potential_field):
    """
    Identifies neighbor nodes and their respective discrete gradient values
    Inspects the 8 adjacent neighbors
    Checks if neighbor is inside the map boundaries
    Returns a list containing [discrete_gradient, index of neighbor node] pairs
    """
    neighbours_and_gradients = []

    # Get the index value of the 8 adjacent neighbors
    upper = index - width
    left = index - 1
    upper_left = index - width - 1
    upper_right = index - width + 1
    right = index + 1
    lower_left = index + width - 1
    lower = index + width
    lower_right = index + width + 1

    # upper neighbor
    if upper > 0:
      discrete_gradient = potential_field[upper] - potential_field[lower]
      neighbours_and_gradients.append([discrete_gradient, upper])

    # left neighbor
    if left % width > 0:
      discrete_gradient = potential_field[left] - potential_field[right]
      neighbours_and_gradients.append([discrete_gradient, left])

    # upper left neighbor
    if upper_left > 0 and upper_left % width > 0:
      discrete_gradient = potential_field[upper_left] - potential_field[lower_right]
      neighbours_and_gradients.append([discrete_gradient, upper_left])
```

```
    # upper right neighbor
    if upper_right > 0 and (upper_right) % width != (width - 1):
      discrete_gradient = potential_field[upper_right] - potential_field[lower_left]
      neighbours_and_gradients.append([discrete_gradient, upper_right])

    # right neighbor
    if right % width != (width + 1):
      discrete_gradient = potential_field[right] - potential_field[left]
      neighbours_and_gradients.append([discrete_gradient, right])

    # lower left neighbor
    if lower_left < height * width and lower_left % width != 0:
      discrete_gradient = potential_field[lower_left] - potential_field[upper_right]
      neighbours_and_gradients.append([discrete_gradient, lower_left])

    # lower neighbor
    if lower <= height * width:
      discrete_gradient = potential_field[lower] - potential_field[upper]
      neighbours_and_gradients.append([discrete_gradient, lower])

    # lower right neighbor
    if (lower_right) <= height * width and lower_right % width != (width - 1):
      discrete_gradient = potential_field[lower_right] - potential_field[upper_left]
      neighbours_and_gradients.append([discrete_gradient, lower_right])

    return neighbours_and_gradients
```

Explanation

The function above takes as input: the index of the current grid cell, map width, map height, and the total potential field. An empty list named `neighbours_and_gradients` is created to keep all neighbor nodes with their respective discrete gradients. Next, each discrete gradient value is calculated one by one and pushed into `neighbours_and_gradients` along with the index of the neighboring grid cell located towards the corresponding direction.

Finally, it returns `neighbours_and_gradients` .

- End of Exercise 5.7.1 -

Next Waypoint Position

Once you have the discrete gradients in all eight directions, you can pick the grid cell located towards the direction of the steepest descent as the next path waypoint.

This is what that looks like expressed in Python, when calling the function `get_neighbours_and_gradients(...)` that you just added to the exercise file:

```
# Neighbor cell with the lowest potential force value
neighbours_and_gradients = get_neighbours_and_gradients(current, width, height, potential_field_data)
new_waypoint = min(neighbours_and_gradients)[1]
```

Creating a Path

By connecting the starting position with its lowest neighbor, and this neighbor, with its own lowest neighbor, repeating the process again and again, the algorithm will eventually reach the goal position.

To create a path, the gradient descent algorithm just needs an empty list, to which it can append the lowest neighbor each time it iterates.

Please be aware that the choice of a **tolerance** to the goal is necesary. Due to the very small gradient differences around the goal locations, sometimes the path might just miss the goal cell and keep oscillating back and forth around it.

Additionally, it is necessary to add a **condition for breaking the loop** to prevent the algorithm from continuing forever in such a situation - something you should avoid.

- Exercise 5.7.2 -

Throughout this unit, you have learned, analyzed, and implemented the essential parts that create the core of a global path planning algorithm based on artificial potential fields and gradient descent.

In this final exercise, you will be coding the body of the gradient descent algorithm.

Now turn back to the **gradient_descent.py** file to start completing the tasks described below.

Tasks

Complete the body of the function `gradient_descent(...)`.

- Create a parameter called `max_iterations` and assign it a value.
- Add `current_iteration` as a variable to keep track of the current iteration number.
- Set a tolerance margin to the goal.
- Initialize a boolean flag variable indicating whether the goal was reached or not
- Create `current` and set it equal to `start_index`. This variable will keep the index of the latest grid cell added to the path.
- Add an empty list to hold the output path from start to goal, name it `path`.
- If you want, you can print a message on the console to inform that the initialization has finished: `rospy.loginfo('Gradient descent: Done with initialization')`.
- Write a loop that iterates as long as `current_iteration < max_iterations`
- Optional: to visualize the gradient descent path as it expands, add this line of code:
  `descent_viz.draw(current, potential_field_data[current])`
- Check if the goal was reached using the tolerance margin.
  Hint: you can use the provided `euclidean_distance(...)` function.
  If the goal has been reached, then:
  - Set the boolean flag variable to `True`
  - Optional: inform the user with a message on the console, for instance using:
    `rospy.loginfo('Gradient descent: Goal reached')`
  - Optional: add the last path segment to the path visualization in RVIZ with:
    `descent_viz.draw(goal_index, 0)`
  - Break out of the loop
- Otherwise, get the neighbor cell in the direction of the steepest descent.
- Add the new waypoint to the path.
- Set the new waypoint as the current waypoint for the next algorithm iteration.
- Increase the number of iteration by one.

When the algorithm exits the main loop:

- Check if the target was found.
- If not, consider informing the user:
  `rospy.logwarn('Gradient descent probably stuck at critical point!!')`
  And return nothing.
- Otherwise, make the function return `path`.

Save your work before continuing.

In [ ]:
```
def gradient_descent(start_index, goal_index, width, height, potential_field_data, descent_viz):
    '''
    Performs gradient descent on an artificial potential field
    with a given start, goal node and a total potential field
    '''


    pass
```

- End of Exercise 5.7.2 -

Great work, you made it! Now it is time to analyze how this method works in practice.

5.8 Testing

- Exercise 5.8.1 -

▶ Execute in WebShell #1

In [ ]:
```
roslaunch unit5 unit5_exercise.launch run_gradient_descent:=true
```

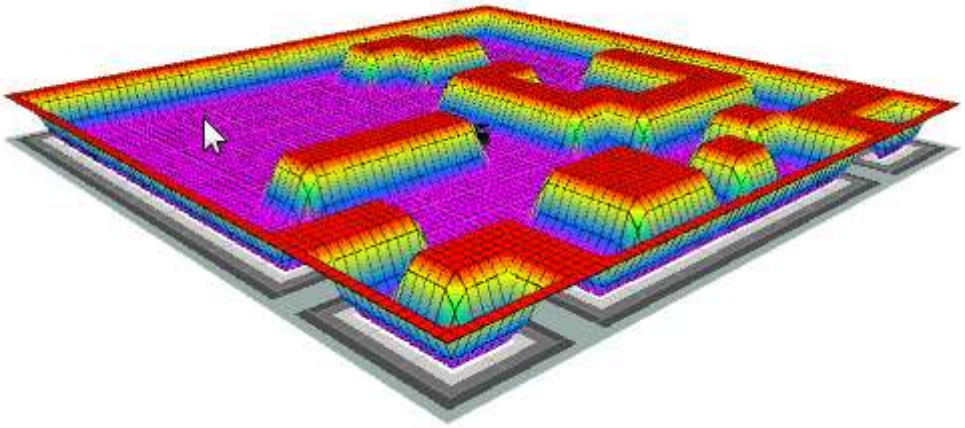
Wait a few seconds until the **Graphical Tools** window gets ready and Rviz is opened.

Use the **2d Nav Goal** button to give a goal position to the robot's path planner.

Note: if your robot doesn't move, double check that you executed the launch command correctly.

For visualizing what the attractive, repulsive and total fields look like, select the corresponding boxes in RVIZ.

Expected Result



The path should advance in the direction of the negative of the gradient.

Set a few different goal positions to test the path planner thoroughly.

Finding the right parameter values is crucial for an optimal behavior of the **gradient descent** path planner, so go back and tune them again if necessary.

Also, **depending on where you set the goal position**, the path might be unable to reach the goal and get stuck.

You will see the following warning:

```
[INFO] [1621957290.274657, 167.238000]: Gradient descent: Done with initialization
[WARN] [1621957290.293676, 167.247000]: Gradient descent probably stuck at critical point!!
[WARN] [1621957290.297218, 167.249000]: Path server: No path returned by gradient descent
```

But why? Let's find out...

- End of Exercise 5.8.1 -

5.9 Pros and Cons of Potential Field Methods

A notable advantage of potential field algorithms is that they are relatively easy to implement, computationally inexpensive, and run really fast. In fact, they are often incorporated into real-time object avoidance systems running at tens of hertz using the robot's sensor data. However, when using this method for global path planning, the path can easily get stuck and never reach the goal.

This situation arises when the structure and geometry of the robot environment produce potential fields where the **attractive and repulsive forces are canceled out**. The following are some examples that typically cause such an event:

- Non-circular obstacles
- Very large obstacles of any shape
- L-shaped obstacles or any dead ends
- Obstacles close to each other in such a way that the robot/path cannot pass through
- Narrow passages where the repulsive forces from either side of the passage cancel out the attractive force
- When the goal is too close to an obstacle

In general, you call those map locations **critical points**. In these places, the gradient is 0 or does not exist.

As a consequence, the artificial potential fields method is not complete, i.e., there is no guarantee that it will find a path to the goal if one exists.

Some solutions help to overcome critical points. However, they go beyond the conventional attractive potential field algorithm and would exceed the scope of this introductory course.

5.10 Summary

Below you will find the most important ideas from this unit. Please ensure that you have mastered the basic concepts:

- An artificial potential field is a mathematical description of the robot's environment used to move a robot from a starting position to a goal location without colliding with obstacles
- A total potential field is the sum of all attractive and repulsive potentials
- Although there exists a plethora of approaches for calculating attractive and repulsive potentials, the basics are: attractive potentials guide towards the goal while repulsive potentials keep the robot away from obstacles
- Whichever potential function you choose to use depends on you as the designer of the algorithm considering the map and the particular situation
- Once you have the total potential function, you can use the gradient descent algorithm to construct a path
- Additionally, there are a number of parameters that must be considered and properly tuned
- The main advantage of this algorithm is the speed of computation
- Accordingly, this method has established itself as one of the most desirable approaches for real-time object avoidance
- This method is prone to getting stuck at critical points, meaning that in some cases, it will not be able to find a path to the goal