5ELEN018W - Tutorial 2 Exercises

1 Grübler's Formula

Implement a Python function which calculates and returns the number of degrees of freedom of a robot manipulator consisting of links and joints. The calculation should be done using the Grübler's formula.

The function should accept 4 arguments: N the number of links (including the ground), J the number of joints, m the degrees of freedom of a rigid body and a list containing the f_i values corresponding to the number of freedoms provided by joints i.

Test the correctness of your implementation by calling the function with appropriate parameters for the 3 examples given in the Lecture Slides (slides 23, 24, 25).

2 Another Example on Grübler's Formula

Calculate the number of degrees of freedom for the robot manipulator shown in Figure 1. Do the calculation on paper, then use the Python function you implemented in Exercise 1.

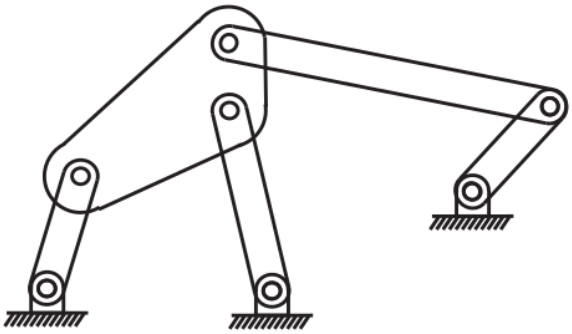

Figure 1: Robot Manipulator for Exercise 2.

3 Python F-Strings

Python version 3.6 introduced support of *f-strings*, i.e. formatted string literals. Notice in the example below that you can embed expressions inside f-strings.

Type these in Jupyterlab and make sure you understand how f-strings work.

```
month = 'October'
year = 2024
print(f"This month is {month} of year {year}. The month of {month} has {24*31} hours.")
```

You can use string formatting as well:

```
a = 11
b = 3
print(f'The result of {a}/{b} is {a/b} or with two decimal digits: {a/b:.2f}')
```

Write a Python program which asks the user their name and displays the name capitalised using f-strings. The conversion to capitals should be done by calling Python string's function `upper()`.

Execute `help(str.upper)` to see the documentation for the function.

4 More on Grübler's Formula

Implement a Python function which when called, it asks the user the number of revolute joints, the number of prismatic joints, the number of helical joints, the number of cylindrical joints, the number of universal joints and the number of spherical joints for a robot manipulator.

Following that, the user is asked about the number of links of the manipulator, and whether it is a planar or a spatial body and how many degrees of freedom f_i each joint has (you should use a loop for the latter).

The function uses Grübler's formula and Python f-strings to return the result as a string exactly as shown in the example below:

```
m = 3 (planar)
J = 4
N = 5 (including the ground)
f_1 = 1
f_2 = 1
f_3 = 1
f_4 = 1
dof = m*(N-1-J) + f_1 + f_2 + f_3 + f_4 = 3*(5 - 1 - 4) + 1 + 1 + 1 + 1 = 4
```

Make sure that your function returns as a string exactly what is shown above in an identical format! The values might change depending on the user input.

5 A Robot Navigating a Maze

A robot tries to navigate the discrete 5x5 maze which is shown in text format below:

```
|------------------|
|   |   |   |   |   |
|------------------|
|   | O |   | G |   |
|------------------|
|   |   | O |   |   |
|------------------|
|   | I |   | O |   |
|------------------|
|   |   |   |   |   |
|------------------|
```

The same maze is shown in diagrammatic form in Figure 2.

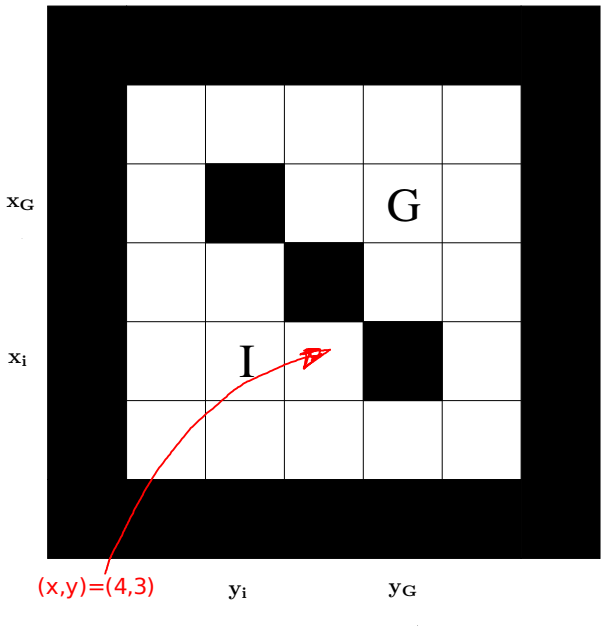

Figure 2: The maze environment for the robot for Exercise 5.

The maze consists of 25 cells. The cells that contain an 'O' are blocked by an obstacle and the robot cannot pass through that cell. At any point in time, the robot can occupy any cell which does not contain an obstacle.

The robot's goal is to reach a goal position G, starting from an initial position I.

At every time step t, the available actions for the robot are move north (`North` action), move east (`East` action), move south (`South` action) or move west (`West` action). If a robot takes an action which would lead it to a cell with an obstacle, then it does not move and remains in the same cell as it was before taking the action.

The robot is quite naive and does not know how to go about the problem, therefore at each time step it decides to take a random move among the 4 possible, i.e. north (`North` action), east (`East` action), south (`South` action) or west (`West` action).

The task for this Exercise is to simulate the navigation of the robot in Python, i.e. the robot starts at the initial position and takes random actions until it reaches the goal position.

1. Implement the maze as a dictionary (see Lecture 1 slides). Each element is a mapping of a tuple (containing the coordinates of a cell) to a string which is either 'O' (for an obstacle) or ' ' (for an empty cell). This means that the key of the dictionary representing the maze is a list of coordinates (x, y) and the value is either 'O' or ' '.

   For example, one element corresponding to the cell indicated as red in Figure 2 is `[4, 3]:'  '`. The element corresponding to the cell (4,4) is `(4,4):'O'`.

2. Write a Python function which simulates the movement of the robot until it reaches the goal position. I.e. at each time step the robot takes a random move and execute it accordingly. Actions which lead to a cell occupied by an obstacle or moving the robot out of the maze will not move the robot out of its current cell.

   For taking a random action you should import the `random` Python module and call its `random.randint(1,4)` function which will return a random number in the range of 1 to 4 inclusive (i.e. $[1, 4]$. You should map 1 to the 'N' action, 2 to the 'E' action, 3 to the 'S' action and '4' to the 'W' action.

   Print in the output all of the steps taken to reach the goal in the format:

   ```
   Location: (4,3), Action 'S' -> (5,3)
   Location: (5,3), Action 'W' -> (5,2)
   ...
   ```

   Hint: Implement the movement of the robot in a single step as a function which takes 2 arguments, the current position of the robot as a tuple of coordinates and a string action. The function returns a list corresponding to the new position of the robot. The skeleton of the function should be:

   ```
   def move(coordinates_list, action):
       // implement the function here
       return ?
   ```

   When the above function is called, it returns the new coordinates of the robot, depending on the current location of the robot and the action, e.g.

   ```
   move((4,3), 'E')  # returns (4,3)
   move((4,3), 'S')  # returns (5,3)
   ```

3. Run a large number of simulations (e.g. 10000) and store the number of steps taken by the robot to reach the target location. Calculate the average number of steps taken by the robot in all the episodes.

   This is called the "Law of Large Numbers"and it is typical for Monte Carlo simulation calculations (ask your tutor what this means). See also the following link:

   https://en.wikipedia.org/wiki/Law_of_large_numbers