Homework 2 (Python version)

ME570 - Prof. Tron

2023-09-25

Problem 1: Rotations

Consider the following function $\theta \in \mathbb{R} \mapsto R \in \mathbb{R}^{2\times 2}$:

$$R(\theta) = \begin{bmatrix} r_{11}(\theta) & r_{12}(\theta) \\ r_{21}(\theta) & r_{22}(\theta) \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}. \tag{1}$$

The functions r_{ij} represent the i,j-th entry in the matrix.

Question report **1.1.** Show that $R(\theta)$ is a rotation (i.e., it satisfies all the requisites) for any value of θ.

Question provided **1.1.** Write a Python function that implements (1); this simple function will be used in subsequent problems.

> File name: `me570_geometry.py`
> Function name: `rot2d`
> Description: Create a 2-D rotation matrix from the angle `theta` according to (1).
> Input arguments
> - `theta` (dim. $[1 \times 1]$): An angle θ.
>
> Output arguments
> - `rot_theta` (dim. $[2 \times 2]$): A matrix containing $R(\theta)$.

Question report **1.2.** Eq. 1 can be used to build 3-D rotations for some particular axes of rotation. Explain the geometrical meaning (i.e., the rotation axis and the direction of the

rotation for increasing values of θ) of the following 3-D rotations:

$$R_1(\theta) = \mathrm{diag}\big(1, R(\theta)\big) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & r_{11}(\theta) & r_{12}(\theta) \\ 0 & r_{21}(\theta) & r_{22}(\theta) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) \\ 0 & \sin(\theta) & \cos(\theta) \end{bmatrix}, \tag{2a}$$

$$R_2(\theta) = \begin{bmatrix} r_{11}(\theta) & 0 & r_{12}(\theta) \\ 0 & 1 & 0 \\ r_{21}(\theta) & 0 & r_{22}(\theta) \end{bmatrix} = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix}, \tag{2b}$$

$$R_3(\theta) = \mathrm{diag}\big(R(\theta), 1\big) = \begin{bmatrix} r_{11}(\theta) & r_{12}(\theta) & 0 \\ r_{21}(\theta) & r_{22}(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{2c}$$

$$R_4(\theta) = \mathrm{diag}\big(-R(\theta), 1\big) = \begin{bmatrix} -r_{11}(\theta) & -r_{12}(\theta) & 0 \\ -r_{21}(\theta) & -r_{22}(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -\cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & -\cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{2d}$$

$$R_5(\theta) = \mathrm{diag}\big(R(-\theta), 1\big) = \begin{bmatrix} r_{11}(-\theta) & r_{12}(-\theta) & 0 \\ r_{21}(-\theta) & r_{22}(-\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos(-\theta) & -\sin(-\theta) & 0 \\ \sin(-\theta) & \cos(-\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{2e}$$

The operator diag creates a block diagonal matrix with its arguments (in Python, this can be implemented by the function `scipy.lin_alg.block_diag(_)` in the SciPy package). Hints are available for this question.

Problem 2: Grids, function handles, and evaluating functions on grids

In this homework and in future assignments, we will need to evaluate different functions on discretized domains (i.e., on regular grids). To aid in this representation, we will use a class `Grid` with the following constructor.

> Class name: `Grid`
> File name: `me570_geometry.py`
> Description: A function to store the coordinates of points on a 2-D grid and evaluate arbitrary functions on those points.
>
> > Method name: `__init__`
> > Description: Stores the input arguments in attributes.
> > Input arguments
> >
> > - `xx_ticks` (dim. [1 × `nb_grid_x`]), `yy_ticks` (dim. [1 × `nb_grid_y`]): arrays containing, respectively, the x and y values corresponding to the horizontal and vertical lines of the grid.

Function handles. In Python, functions are *first-class objects*, i.e., they can be stored in variables and passed to other functions. For instance, consider the function `math.sin(_)` in the `math` module. You can store[1] it in a variable `myfun` using the assignment `myfun=math.sin`;

[1] Technically, in Python, it would be more accurate to say that you are creating a new label for an object that is already in memory.

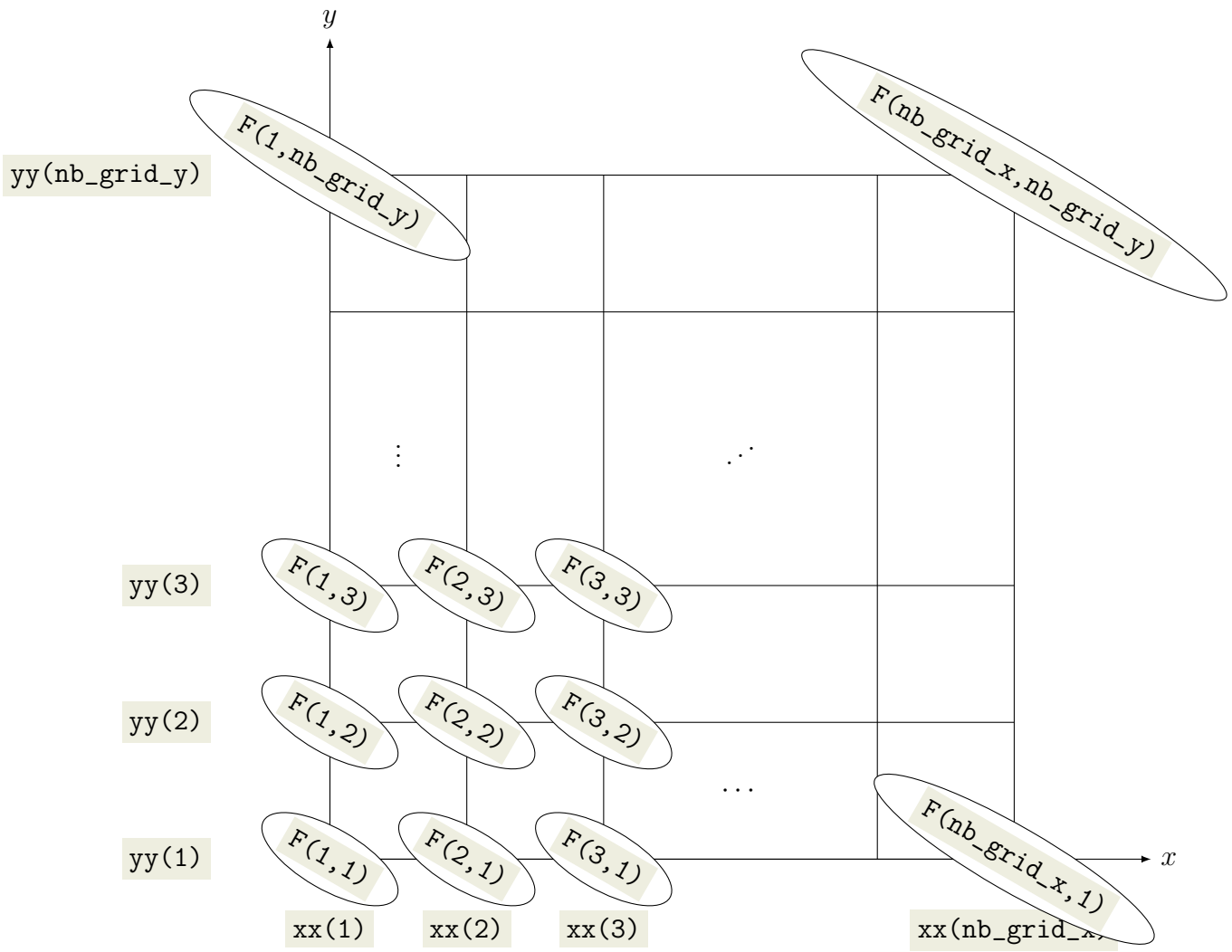

Figure 1: Representation of the computation of a (possibly vector-valued) function on a grid domain. The values in `xx_ticks` and `yy_ticks` (abbreviated as `xx` and `yy` in this figure) contain the x and y coordinates of, respectively, the vertical and horizontal lines defining the grid. The variable `F` contains the value (possibly a vector) associated to each location. Notice the organization of the indexes of `F` with respect to `xx` and `yy`; this organization is transposed with respect to what is expected by `plot_surface(,)` but it has the advantage that the indexes of `F` align more naturally with the indeces of `xx` and `yy`.

notice that, in the assignment, there are no parentheses (otherwise you would *call* the function, not store it). You can pass a function as an argument to another function in a similar way (i.e., using the name of the function without parentheses).

Question **provided** **2.1.** Implement an object that represents a grid and that is used to evaluate functions on this grid.

> Class name: `Grid`
> File name: `me570_geometry.py`
>
> > Method name: `eval`
> > Description: This function evaluates the function `fun` (which should be a function) on each point defined by the grid.
> > Input arguments
> >
> > - `fun` (type `function`): an handle to a function that takes a vector `x` of dimension [2 × 1] as a single argument, and returns a scalar or a vector. An example of this function could be the function `norm(_)`.
> >
> > Output arguments
> >
> > - `fun_eval` (dim. [`nb_grid` × `nb_grid` × `d`]): the function `norm(_)` evaluated on each point of the grid.

We will use this method any time we will need to evaluate a map on a rectangular region. For instance, in Problem 3 `xx` and `yy` will correspond to the two angles θ_1, θ_2 for plotting the torus. You can find a very simple example of how to use it in the provided function `grid_eval_example(_)` in the file `me570_hw2.py`.

Problem 3: Embedding for the torus using rotations

In this question you will consider the torus $\mathbb{T} = \mathbb{S}^1 \times \mathbb{S}^1$ as a manifold, and use the material from the previous question to make charts for the torus and visualize an embedding in $\mathbb{R}^3$ together with some curves in the configuration space.

Question **report** **3.1.** Consider the following map:

$$\phi_{circle}(\theta) = R(\theta) \begin{bmatrix} 1 \\ 0 \end{bmatrix}. \tag{3}$$

Show that $\phi_{circle}(\theta) \in \mathbb{S}^1$ for all $\theta \in \mathbb{R}$, i.e., that (3) maps an angle θ to a point on the circle $\mathbb{S}^1$.

Question **code** **3.1.** Let $\vec{\theta} = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$ be a vector with two angles. Consider the map

$$\phi_{torus}(\vec{\theta}) = R_3(\theta_2) \left(\begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \phi_{circle}(\theta_1) + \begin{bmatrix} r \\ 0 \\ 0 \end{bmatrix} \right), \tag{4}$$

where $r = 3$, R_3 is the same as in (2c) and ϕ is the same as in (3). This map might look a little intimdating at first, but, intuitively, it boils down to setting the circle onto the $x - z$

plane, translating it along the x axis, and then revolving it around the z axis. As you will see later, the map ϕ_{torus} can be used to embed a torus in $\mathbb{R}^3$, and use it to represent the configuration space of the two-link manipulator. Implement the following function, which will be used to embed the torus plot in the Euclidean space $\mathbb{R}^3$.

> Class name: `Torus`
> File name: `me570_geometry.py`
> Description: A class that holds functions to compute the embedding and display a torus and curves on it.
>
> > Method name: `phi`
> > Description: Implements equation (4).
> > Input arguments
> >
> > - `theta` (dim. [2 × `nb_points`]): An array of values for $\vec{\theta}$ (each column corresponds to a different $\vec{\theta}$).
> >
> > Output arguments
> >
> > - `x_torus` (dim. [3 × `nb_points`]): The values of $\phi_{torus}(\vec{\theta})$ corresponding to all the angles in `theta`.

Question provided 3.1. The following function tests `phi(_)` and shows how to plot curves.

> Class name: `Torus`
> File name: `me570_geometry.py`
>
> > Method name: `phi_test`
> > Description: Plots two (almost complete) rings, one with $\theta_1 = 0$, and the other with $\theta_2 = 0$.

Question report 3.2. Run the function `Torus.phi_test(_)` and include the result in your report. It should similar to Figure 2.

Tip: if you use `plt.ion(_)` before plotting, all the subsequent plot functions will display their result immediately (as in Matlab).

Question report 3.3 (2 points). Consider the chart (local coordinates) given by ϕ_{torus} on the domain $U = (0, 2\pi) \times (0, 2\pi)$ (which is an open set).

- Is ϕ_{torus} an homeomorphism between U and $\mathbb{T}$? If not, what modifications to the set U should you make to U so that U and $\mathbb{T}$ have the same topology?
- What is the minimum number of charts that is needed in an atlas for $\mathbb{T}$? Draw a sketch that convincingly illustrates your answer.

Question report 3.4. A function that shows the embedding of the torus given by (4).

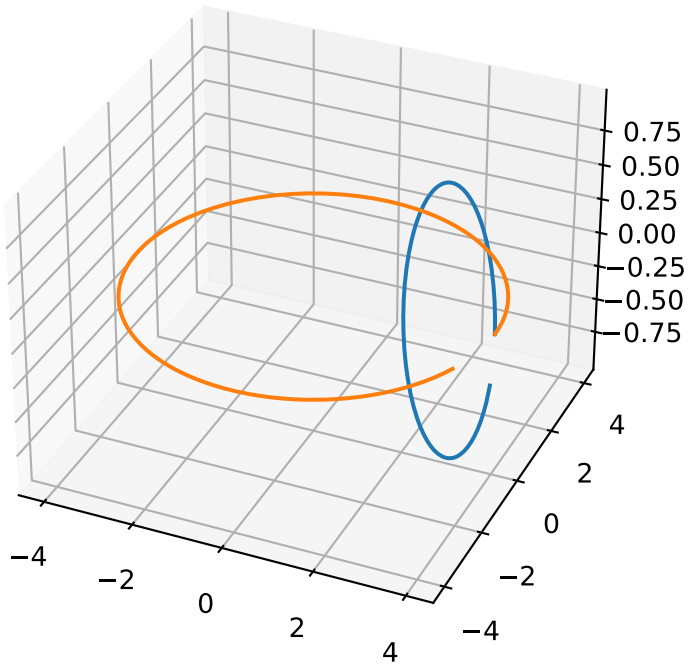

Figure 2: Expected output of `Torus.phi_test()`.

Class name: `Torus`
File name: `me570_geometry.py`

Method name: `plot`
Description: Using the domain U from Question report 3.3:

1) Fill a `grid` structure with fields `xx` and `yy` that define a grid of regular point in U. Use `nb_grid_x=nb_grid_y=33`.

2) Call the function `Grid.eval()` with argument `Torus.phi()`.

3) Plots the surface described by the previous step using the the Matplotlib function `ax.plot_surface()` (where `ax` represents the axes of the current figure).

Include the figure produced by this function in your report

Question provided 3.2. The following function can be used to generate points on a curve $\theta(t)$ that is a straight line.

File name: `me570_geometry.py`
Function name: `line_linspace`
Description: Generates a discrete number of `nb_points` points along the curve $\vec{\theta}(t) =$ (`a_line[0]` $\cdot t +$ `b_line[0]` , `a_line[1]` $\cdot t +$ `b_line[1]`) $\in \mathbb{R}^2$ for t ranging from `t_min` to `tMax` t_max.
Input arguments

- `a_line` (dim. [2 × 1]), `b_line` (dim. [2 × 1]): First and second parameter of the curve.
- `t_min` , `t_max` : Defines the interval for t.
- `nb_points` : Number of points in the interval to generate.

Output arguments

- `theta_points` (dim. [2 × N]): The points generated along the curve $\vec{\theta}(t)$.

Question report 3.5 (0.5 points). Give an analytic expression for the tangent $\dot{\theta}(t)$ of the curve $\theta(t)$ defined in the previous question.

Question code 3.2. This question is based on the idea of generating a curve $\vec{\theta}(t) \in \mathbb{R}^2$ (in fact, a straight line) in the joint space, and "push it through" the embedding ϕ_{torus} to generate the curve $x(t) = \phi_{torus}\big(\vec{\theta}(t)\big) \in \mathbb{R}^3$.

Class name: `Torus`
File name: `me570_geometry.py`

Method name: `phi_push_curve`
Description: This function evaluates the curve $x(t) = \phi_{torus}\big(\vec{\theta}(t)\big) \in \mathbb{R}^3$ at `nb_points=31` points generated along the curve $\vec{\theta}(t)$ using `Line.linspace` ( ) with `tMin` $= 0$ and `tMax` $= 1$, and `a_line` , `b_line` as arguments.
Input arguments

- `a_line` (dim. [2 × 1]), `b_line` (dim. [2 × 1]): Parameters of the curve to be passed to `Line.linspace` ( ).

Output arguments

- `x_points` (dim. [3 × `nb_points`]): Array of points generated by evaluating ϕ_{torus} on `thetaPoints` (i.e., array of points on the curve $x(t)$).

Question report 3.6. The following function combine all the functions from this problem to visualize the embedding of the torus, together with four curves.

Class name: `Torus`
File name: `me570_geometry.py`

Method name: `plot_curves`
Description: The function should iterate over the following four curves:

- `a_line=np.array([[3/4*pi],[0]])`,
- `a_line=np.array([[3/4*pi],[3/4*pi]])`,
- `a_line=np.array([[-3/4*pi],[3/4*pi]])`,
- `a_line=np.array([[0],[-3/4*pi]])`,

and `b_line=np.array([[-1],[-1]])`. The function should show an overlay containing:

- The output of `Torus.plot(_)`;
- The output of the functions `Torus.pushCurve(_)` for each one of the curves.

Requirements: This function needs to use `plot(_)` to show the output of `Torus.pushCurve(_)`. You should see that all the curves start at the same point on the torus. It might be easier to see the curves on the torus by increasing the line width, and making the torus semi-transparent (by changing its opacity, i.e., its *alpha*).

optional Use different colors to display the results of the different curves.

Question report 3.7. Include the output of the function provided in the previous question.

Problem 4: Kinematic map and collision checks for the two-link manipulator

In this problem we will simulate a 2-D two-link manipulator moving in a workspace containing numerous point obstacles, with functions that plot the manipulator and check for collisions.

The 2-D manipulator is composed of two links as shown in Figure 3. The vertices of the

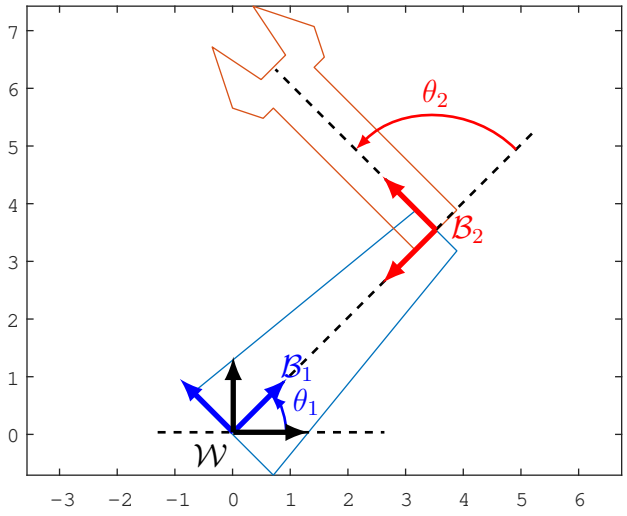

Figure 3: The two-link manipulator for this exercise. The angle of the first link θ_1 is measured from the horizontal x axis, the angle of the second link θ_2 is measured with respect to the axis of the first link.

two links are given by the attribute `polygons` in `me570_robot.py` (which was provided in Homework 1).

We consider three reference frames: $\mathcal{W}$, which is fixed to the world, $\mathcal{B}_1$, which rotates with the first link, and $\mathcal{B}_2$, which rotates with the second link.

The translation of the first link in the world reference frame is zero, i.e., ${}^{\mathcal{W}}T_{\mathcal{B}_1} = 0$, while the rotation ${}^{\mathcal{W}}R_{\mathcal{B}_1}$ has angle θ_1; the translation of the second link in the first link's reference is ${}^{\mathcal{B}_1}T_{\mathcal{B}_2} = \begin{bmatrix} 5 \\ 0 \end{bmatrix}$, while the rotation ${}^{\mathcal{B}_1}R_{\mathcal{B}_2}$ has angle θ_2 (see also the annotations in Figure 3).

Question report 4.1 (1.5 points). Derive two separate expressions that compute the coordinates of a point p expressed in $\mathcal{W}$, ${}^{\mathcal{W}}p$, for each one of these inputs, respectively:

1) The coordinates of the point given in $\mathcal{B}_1$, denoted as ${}^{\mathcal{B}_1}p$;

2) The coordinates of the point given in $\mathcal{B}_2$, denoted as ${}^{\mathcal{B}_2}p$.

Please make sure that the use of rotation matrices is clear, otherwise you might not receive full credit for your answer; although we will use these computations for 2-D points, the same expressions should hold for 3-D points.

Question code 4.1. Create a function to compute the coordinates of the end effector and of the vertices of the manipulator for a given configuration. The end effector is defined to be the point ${}^{\mathcal{B}_2}p_{\text{eff}} = \begin{bmatrix} 5 \\ 0 \end{bmatrix}$.

Class name: `TwoLink`
File name: `me570_robot.py`

Method name: `kinematic_map`
Description: The function returns the coordinate of the end effector, plus the vertices of the links, all transformed according to θ_1, θ_2.
Input arguments

- `theta` (dim. [2 × 1]) A vector containing `theta(1)` $= \theta_1$ and `theta(2)` $= \theta_2$, the two joint angles for the two-link manipulator.

Output arguments

- `vertex_effector_transf` (dim. [2 × 1]): The coordinates of the point ${}^{\mathcal{B}_2}p_{\text{eff}}$ transformed according to θ_1, θ_2 (i.e., the coordinates of the points in the world frame); this point represents the transformed end effector position.
- `polygon1_transf` (type `Polygon`), `polygon2_transf` (type `Polygon`): the polygons given by the variable `me570_robot.polygons`, transformed according to θ_1, θ_2.

Requirements: Use the results from Question report 4.1 to guide your implementation. This function must use `me570_robot.polygons` to obtain the vertices of the polygons of the matrix, and it must use `rot2d(_)` from Question provided 1.1. Note that here we are simply computing the vertices of the transformed polygons, without plotting them. The next function will be used to plot the transformed vertices.

optional Create another method `kinematic_map(_)` in the class `Polygon` to generate the transformed versions of the polygons.

Question provided 4.1. Create a function that plots the manipulator in a given configuration according to the following.

> Class name: `TwoLink`
> File name: `me570_robot.py`
>
> > Method name: `plot`
> > Description: This function should use `TwoLink.kinematic_map(_)` from the previous question together with the method `Polygon.plot(_)` from Homework 1 to plot the manipulator.
> > Input arguments
> >
> > - `theta` (dim. $[2 \times 1]$): A vector containing the configuration angles as used by `TwoLink.kinematic_map(_)`.
> > - `color` (dim. $[1 \times 1]$, type `string`): Color specification (e.g., `'r'`, `'g'`, `'b'`).

Note that, by default, this plotting function does not enforce the same proportions for the horizontal and vertical axes (i.e., the manipulator might appear skewed). To enforce this, you can use `[ax]=matplotlib.pyplot.gca(_)` to get the current axes, and then the command `ax.axis(['equal'])`.

Question code 4.2. Create a function that checks for collisions between the manipulator and a sparse set of points (representing obstacles).

> Class name: `TwoLink`
> File name: `me570_robot.py`
>
> > Method name: `is_collision`
> > Description: For each configuration, returns `True` if *any* of the links of the manipulator collides with *any* of the points, and `False` otherwise. Use the function `Polygon.is_collision(_)` to check each link.
> > Input arguments
> >
> > - `theta` (dim. $[2 \times$ `nb_theta`$]$): An array of configurations where each column represents a different configuration according to the convention used by `TwoLink.kinematic_map(_)`.
> > - `points` (dim. $[2 \times$ `nb_points`$]$): An array of coordinate of points (obstacles). Each column represents a different point.
> >
> > Output arguments
> >
> > - `flag_theta` (dim. $[1 \times$ `nb_theta`$]$, type `list[bool]`): each element of this list should be `True` if the configuration specified in the corresponding column of `theta` causes a collision, and `False` otherwise.

For this question, *do not* consider self-collision (i.e., if the two polygons overlap but they do not cover any of the points, then it is not a collision).

Question report **4.2.** Make a function that combines the functions from the previous two questions.

> Class name: `TwoLink`
> File name: `me570_robot.py`
>
> > Method name: `plot_collision`
> > Description: This function should:
> >
> > *1)* Use `TwoLink.is_collision(_)` for determining if each configuration is a collision or not.
> >
> > *2)* Use `TwoLink.plot(_)` to plot the manipulator for all configurations, using a red color when the manipulator is in collision, and green otherwise.
> >
> > *3)* Plot the points specified by `points` as black asterisks.
> >
> > Input arguments
> >
> > - `theta` (dim. [2 × `NTheta`]), `points` (dim. [2 × `NPoints`]): As used by `TwoLink.is_collision(_)`

All the drawings from the different configurations should overlap on the same figure (it is fine if many configurations overlap). You should use the command `axis equal` if you want to avoid seeing the manipulator "distorted" (e.g., to see right angles as real right angles on the screen).

provided The following function loads data from the file `twolink_testData.mat` (provided with the homework) and it can be used to test and see if the other functions from this problem are working correctly.

> File name: `me570_hw2.py`
> Method name: `twolink_plot_collision_test`
> Description: This function generates 30 random configurations, loads the `points` variable from the file `twolink_testData.mat` (provided with the homework), and then display the results using `twolink_plotCollision` to plot the manipulator in red if it is in collision, and green otherwise.

Include a figure with the output of the function `twolink_plot_collision_test(_)`.

Question optional **4.1.** Write a method `TwoLink.free_space(_)` that loads points from `twolink_testData.mat`, calls `TwoLink.is_collision(f)` or θ_1, θ_2 sampled along a regular fine grid, and stores the results in a matrix of logical values `free_space_map`. Display `free_space_map` as an image. This is an approximate representation of the free configuration space. You can use `matplotlib.pyplot.imshow(_)` to plot the map as an image. Using the functions above, generating the map should be straightforward, but it might take a very long time.

Problem 5: Jacobians and end effector velocities

In this question you will create functions to compute and plot the velocity of the end effector of our two-link manipulator as a function of the position and velocities of the two links. Recall that we defined the end effector of our two-link manipulator to be the point ${}^{\mathcal{B}_2}p_{\text{eff}} = \begin{bmatrix} 5 \\ 0 \end{bmatrix}$.

Tip: On Blackboard, under Class contents › Class notes from previous years and additional material › Jacobians from "Principles of Robot Motion" you can find a copy of the appendix from the book with another explanation of Jacobians.

Question report 5.1. Seeing the coordinates ${}^{\mathcal{W}}p_{\text{eff}}$ as a function from $\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$ to $\mathbb{R}^2$, give an expression for computing $\frac{\mathrm{d}}{\mathrm{d}t}({}^{\mathcal{W}}p_{\text{eff}})$ as a function of $\dot{\theta} = \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \end{bmatrix}$. This expression is a map representing the Jacobian of the end effector. For this question, you can consult the hints (which recommend the method seen in class), or use the Matlab Symbolic Toolbox (for the latter, save your code in the file `twolink_jacobian_sym.m`).

Question optional 5.1. Use the answer to the previous question (Question report 5.1) to find the Jacobian matrix J such that $\frac{\mathrm{d}}{\mathrm{d}t}({}^{\mathcal{W}}p_{\text{eff}}) = J\dot{\theta}$.

Question code 5.1. Make a function that implements the Jacobian map derived in the previous function.

Class name: `TwoLink`
File name: `me570_robot.py`
Description: This class was introduced in a previous homework.

Method name: `jacobian`
Description: Implement the map for the Jacobian of the position of the end effector with respect to the joint angles as derived in Question report 5.1.
Input arguments

- `theta` (dim. [2 × `nb_theta`]) An array where each column contains $\begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$, two joint angles for the two-link manipulator.
- `theta_dot` (dim. [2 × `nb_theta`]): An array where each column contains the time derivatives of the joint angles, $\dot{\theta}$, for each one of the corresponding columns in `theta`.

Output arguments

- `vertex_effector_dot` (dim. [2 × `nb_theta`]): An array where each column contains the time derivative of the end effector position, expressed in the world's frame, $\frac{\mathrm{d}}{\mathrm{d}t}({}^{\mathcal{W}}p_{\text{eff}})$, for the corresponding columns in the input arguments.

Question report 5.2. Assume θ_1 and θ_2 follow the same two sequences as those used to draw the rings in Question provided 3.1. Explain or sketch what paths would the two-link manipulator take.

Question report **5.3 (2 points).** Assume $\theta(t)$ follow a line as defined in Questions provided 3.2 and report 3.5. What is the corresponding value of $\frac{\mathrm{d}}{\mathrm{d}t}({}^{\mathcal{W}}p_{\text{eff}})$ for the following values of θ, $\left\{\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}0\\\frac{\pi}{2}\end{bmatrix}, \begin{bmatrix}\pi\\\frac{\pi}{2}\end{bmatrix}, \begin{bmatrix}\pi\\\pi\end{bmatrix}\right\}$, and the following values of `a_line`, $\left\{\begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}\right\}$; in total, you need to compute eight values for $\frac{\mathrm{d}}{\mathrm{d}t}({}^{\mathcal{W}}p_{\text{eff}})$. You can do the calculations by hand, or use the function `Twolink.jacobian(_)` from the previous question. In addition, use the function `Twolink.plot(_)` from Question provided 4.1 to plot the eight configurations, and use the `quiver(_)` function to plot $\frac{\mathrm{d}}{\mathrm{d}t}({}^{\mathcal{W}}p_{\text{eff}})$ as an arrow with its base at the end effector's position. Make comments on anything interesting that you might notice from the results; in particular, try to explain why for some configurations the end effector velocity might be zero despite the fact that the robot as a whole is moving.

Question report **5.4.** In the following question we analyze the correspondence between the torus and the two-link manipulator from Problem 4.

> File name: `me570_hw2.py`
> Method name: `torus_twolink_plot_jacobian`
> Description: For each one of the curves used in Question report 3.6, do the following:
>
> - Use `Line.linspace(_)` to compute the array `thetaPoints` for the curve (but use `nb_points=7`);
> - For each one of the configurations given by the columns of `thetaPoints`:
>
> *1)* Use `Twolink.plot(_)` to plot the two-link manipulator.
>
> *2)* Use `Twolink.jacobian(_)` to compute the velocity of the end effector, and then use `quiver(_)` to draw that velocity as an arrow starting from the end effector's position.
>
> The function should produce a total of four windows (or, alternatively, a single window with four subplots), each window (or subplot) showing all the configurations of the manipulator superimposed on each other. You can use `matplotlib.pyplot.ion(_)` and insert a `time.sleep(_)` command in the loop for drawing the manipulator, in order to obtain a "movie-like" presentation of the motion.
>
> Include the four figures produced by this function in your report.
> Requirements:
>
> **optional** For each window (or subplot), use the color of the corresponding curve as used in Question report 3.6.

Question report **5.5 (0.5 points).** Comment on the relation between the results of Questions report 5.4 and report 3.6. Instead of focusing of the individual curves, comment on how the two sets of results can be conceptually linked together.

> "If there's a fifty-fifty chance that something can go wrong, nine out of ten times, it will."
> — *Yogi Berra*

Hint for question report 1.2: Consider what happens when you apply each rotation to the standard basis, i.e., to the vectors $e_x = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$, $e_y = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$, $e_z = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$.

Hint for question report 1.2: Remember that a rotation, when applied to the vector $v \in \mathbb{R}^3$ representing its own rotation axis, it does not change it (i.e., $Rv = v$).

Hint for question report 5.1: To derive the map, imagine that θ is a function of time, i.e. $\theta(t)$, and then compute the time derivative of ${}^{\mathcal{B}}p_{\text{eff}}$. The simplest way to derive this expression is by using the derivative of rotation matrices seen in class.