Master ORO
Optimization in robotics

Université de Nantes
2017/2018

Lab session
Path-planning for a simple parallel robots

1 Getting started with python modules

Download the python modules `paving` and `robot` from Madoc. They come in precompiled form, so you cannot see their code or modify it, which will not be needed for your work. Use preferably Python3 versions.

These modules have the following dependences (links available on Madoc) :

— Python modules : `matplotlib.pyplot` for visualization, `numpy` for numerical computations with vectors and matrices, `scipy.optimize` for numerical optimization, `scipy.sparse.csgraph` for graph algorithms, and `rtree` for spatial indexes.

— Shared libraries : `libspatialindex`, the C backend of the rtree python module.

You can acces the modules, classes and functions documentation from the python interpreter in the standard way, e.g., typing `print(paving.Paving.__doc__)` or `print(robot.Robot.__doc__)`.

1.1 The `paving` module

This module comprises three elements : the function `projected_centers_distance`, the class codeBox and the class `Paving`. Only the latter will be used for the moment.

Solve the "simple.mbx"[1] model using IBEX then load the produced output (typically named "simple.mnf") using the paving class :

1. import the paving module : `import paving`
2. create a Paving object : `p = paving.Paving()`
3. load the IBEX output : `p.load_mnf("simple.mnf")`

Check the content of p after that, especially its members `boxes` and `data`.

Display the paving using pyplot :

1. import pyplot : `from matplotlib import pyplot`
2. create a figure : `fig1,ax1 = pyplot.subplots()`
3. display the paving projected on x and y : `p.draw2D(ax1,1,2)`
4. scope the display according to the paving hull : `ax1.axis(p.hull([1,2]))`
5. show the graphic : `fig1.show()`

The paving appears too coarse to be informative. Try to improve it by relaunching IBEX using the options `-e` and `-E` (check ibexsolve command line help).

Reload and display the finer paving, in the three possible projections on three different figures this time. What do you observe ?

Use the box neighborhood graph of the paving to identify distinct components of the solution set :

1. create the graph : `m = p.adjacency_matrix()`
2. import graph library : `from scipy.sparse.csgraph import connected_components`
3. compute components : `nc,l = connected_components(m, directed=False)`

Check the obtained values nc and l. What does their values mean ?

Update the displayed pavings by overlaying components :

[1] First, have a look at this model and try to figure out its solution set.

1. create a subpaving for component 0 : `sp0 = p.subpaving([i for i in range(len(p.boxes)) if l[i]==0])`
2. overlay it in the xy projection : `sp0.draw2D(ax1,1,2,ec=None,fc='yellow')`

Do the same for the other projections, and also overlay the other components with another color.

1.2 The `robot` module

This module comprises two classes : the class `Robot` and the class `FiveBars`. The former is an abstract class the defined the interface of all virtual robots, while the latter is a specialization for the five-bars linkage, a simple yet typical parallel robot which you will study further in your distant work. Check the documentations of these classes and their members

The home commands are `[0,180]`. The assembly mode `0` (respectively `1`) corresponds to the end-effector below (respectively above) the horizontal axis. Compute the corresponding poses using IBEX.

Create your python virtual 5-bars robot :

1. import the robot module : `import robot`
2. create the 5-bars robot : `r=robot.FiveBars([-22.5, 0, 22.5, 0, 17.8, 17.8, 17.8, 17.8],0,seed)`. **Important :** Ask teachers to provide your unique seed for the distant work.

A figure appears displaying your newly created robot. **Important :** The figure and the robot object are interdependent and you should never close the figure manually.

Just like real robots, the virtual robot has small manufacturing imprecisions that nevertheless alter its operations, and its captors also provide slightly noisy measurements. The seed determines the imprecisions in the robot actual architecture and the noise in its captors.

Check the robot pose using the method `measure_pose()`. You can use it several times.

Using IBEX, compute the joint coordinates to reach the pose `x=[5,-20]`.

Actuate the virtual robot to reach this pose using all obtained solutions : For each solution,

1. set the robot in no-draw mode : `r.pen_up()`
2. return the robot to home pose : `r.go_home()`
3. set the robot in draw mode : `r.pen_down()` (you can change the draw color for each command)
4. actuate the robot to reach prescribed command : `r.actuate([t1,t2])`[2]

The figure of the robot accepts overlays : The member `r.ax` allows displaying points, curves, pavings, ... Use it to display the objective pose : `r.ax.plot([5],[-20],color='black',marker='*')`.

Did the robot reach the objective pose in each case ? Can you explain why ?

Same questions for the pose `x=[5,20]`.

Same questions for the pose `x=[5,20]`, but creating the robot with assembly mode `1`.

Build a table that contains $q = (0, 180) + r(\cos t, \sin t)$ for $r = 10$. This is a circle in the joint-space. With pen down, follow this pathSame question for $r \in \{20, 30, 40, 50, 60\}$.

[2]The motion follows a (discretized) straight line in the joint space from the current configuration to `[q1,q2]`. For large displacements, the trace may show some discretization errors.