

Robotics Toolbox

Peter Corke

June 10, 2026

```
[3]: # import matplotlib.pyplot as plt
import numpy as np
np.set_printoptions(linewidth=100, formatter={'float': lambda x: f"{x:8.3g}" if
    ↪abs(x) > 1e-10 else f"{0:8.3g}"})

import math
```

1 Mobile robotics

1.1 Mobile robot kinematics

We can create a vehicle with bicycle kinematics, with a wheelbase of 2m.

```
[4]: from roboticstoolbox import Bicycle, VehicleIcon

veh = Bicycle(L=2, animation=VehicleIcon("redcar", scale=2), workspace=10)
print(veh)
```

```
Bicycle: x = [ 0, 0, 0 ]
      L=2, steer_max=1.41372, speed_max=inf, accel_max=inf
```

We can find the maximum path curvature it can achieve, which is the reciprocal of the minimum turning radius.

```
[5]: print(veh.curvature_max)
```

```
3.156875757337521
```

The initial state of the vehicle (x, y, θ) , where θ is the heading angle, can be set at construction time but defaults to zero

```
[6]: print(veh.x)
```

```
[      0      0      0]
```

The state derivative $(\dot{x}, \dot{y}, \dot{\theta})$ is a function of current state and the inputs (v, γ) where γ is the angle of the steered wheel.

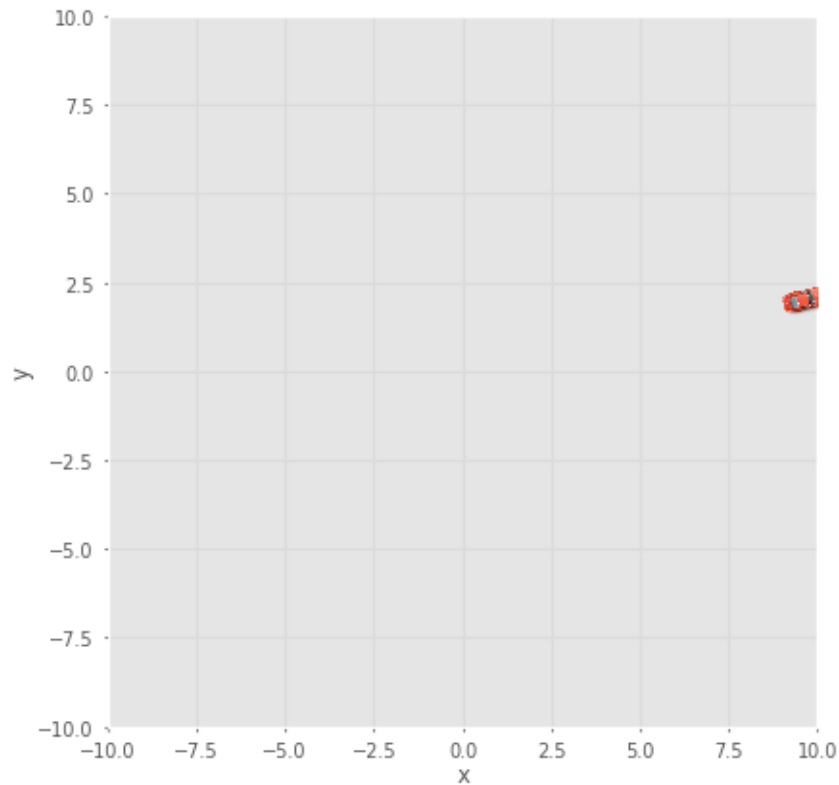
```
[7]: xd = veh.deriv(u=(1, 0.2), x= [0,0,0])
print(xd)
```

```
[      1      0    0.101]
```

which indicates motion in the x-direction and a change of heading angle.

The model supports simple animation of motion directed by a `control` function. In this case the control sets a constant velocity and a steered wheel angle of 0.5 rad for $1 < t < 2$

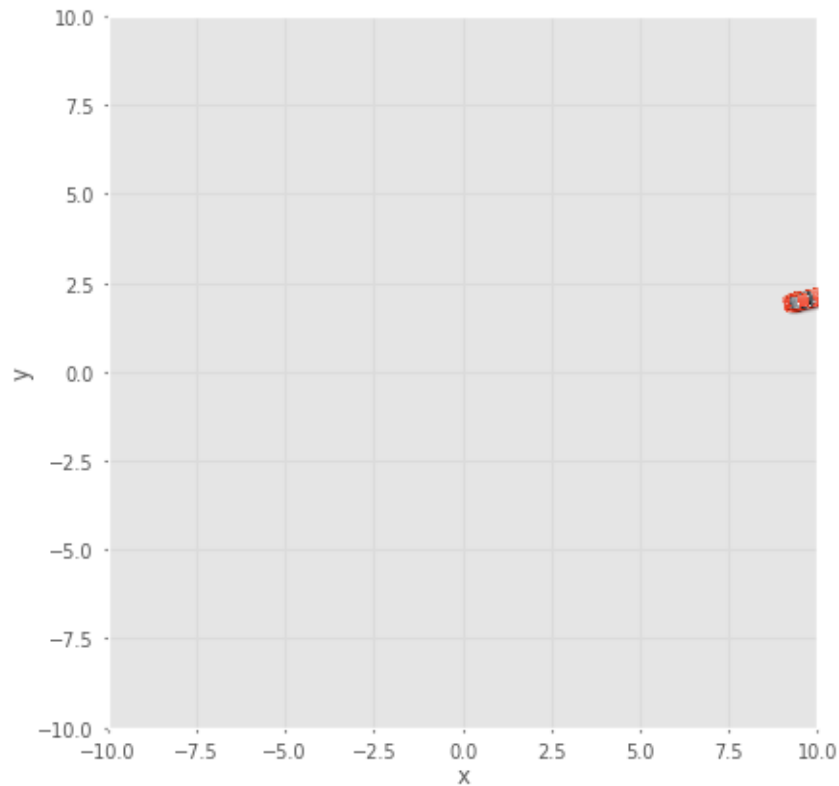
```
[8]: veh.run(10, control=lambda v, t, x: (1, 0.5 if 1<t<2 else 0), animate=True)
```



```
[8]: array([[ 0.1,      0,      0],
 [ 0.2,      0,      0],
 [ 0.3,      0,      0],
 [ 0.4,      0,      0],
 [ 0.5,      0,      0],
 [ 0.6,      0,      0],
 [ 0.7,      0,      0],
 [ 0.8,      0,      0],
 [ 0.9,      0,      0],
 [ 1.0,      0,      0],
 [ 1.1,      0,      0],
 [ 1.2,      0, 0.0273],
 [ 1.3, 0.00273, 0.0546],
 [ 1.4, 0.00819, 0.0819],
```

[	1.5,	0.0164,	0.109],
[	1.6,	0.0273,	0.137],
[	1.7,	0.0409,	0.164],
[	1.8,	0.0572,	0.191],
[	1.89,	0.0762,	0.219],
[	1.99,	0.0979,	0.246],
[	2.09,	0.122,	0.246],
[	2.19,	0.147,	0.246],
[	2.28,	0.171,	0.246],
[	2.38,	0.195,	0.246],
[	2.48,	0.22,	0.246],
[	2.57,	0.244,	0.246],
[	2.67,	0.268,	0.246],
[	2.77,	0.293,	0.246],
[	2.87,	0.317,	0.246],
[	2.96,	0.341,	0.246],
[	3.06,	0.366,	0.246],
[	3.16,	0.39,	0.246],
[	3.25,	0.414,	0.246],
[	3.35,	0.439,	0.246],
[	3.45,	0.463,	0.246],
[	3.54,	0.487,	0.246],
[	3.64,	0.512,	0.246],
[	3.74,	0.536,	0.246],
[	3.84,	0.56,	0.246],
[	3.93,	0.585,	0.246],
[	4.03,	0.609,	0.246],
[	4.13,	0.633,	0.246],
[	4.22,	0.658,	0.246],
[	4.32,	0.682,	0.246],
[	4.42,	0.706,	0.246],
[	4.51,	0.731,	0.246],
[	4.61,	0.755,	0.246],
[	4.71,	0.779,	0.246],
[	4.81,	0.804,	0.246],
[	4.9,	0.828,	0.246],
[	5,	0.852,	0.246],
[	5.1,	0.877,	0.246],
[	5.19,	0.901,	0.246],
[	5.29,	0.925,	0.246],
[	5.39,	0.95,	0.246],
[	5.48,	0.974,	0.246],
[	5.58,	0.998,	0.246],
[	5.68,	1.02,	0.246],
[	5.78,	1.05,	0.246],
[	5.87,	1.07,	0.246],
[	5.97,	1.1,	0.246],

[	6.07,	1.12,	0.246],
[	6.16,	1.14,	0.246],
[	6.26,	1.17,	0.246],
[	6.36,	1.19,	0.246],
[	6.45,	1.22,	0.246],
[	6.55,	1.24,	0.246],
[	6.65,	1.27,	0.246],
[	6.75,	1.29,	0.246],
[	6.84,	1.31,	0.246],
[	6.94,	1.34,	0.246],
[	7.04,	1.36,	0.246],
[	7.13,	1.39,	0.246],
[	7.23,	1.41,	0.246],
[	7.33,	1.44,	0.246],
[	7.42,	1.46,	0.246],
[	7.52,	1.49,	0.246],
[	7.62,	1.51,	0.246],
[	7.72,	1.53,	0.246],
[	7.81,	1.56,	0.246],
[	7.91,	1.58,	0.246],
[	8.01,	1.61,	0.246],
[	8.1,	1.63,	0.246],
[	8.2,	1.66,	0.246],
[	8.3,	1.68,	0.246],
[	8.39,	1.7,	0.246],
[	8.49,	1.73,	0.246],
[	8.59,	1.75,	0.246],
[	8.68,	1.78,	0.246],
[	8.78,	1.8,	0.246],
[	8.88,	1.83,	0.246],
[	8.98,	1.85,	0.246],
[	9.07,	1.87,	0.246],
[	9.17,	1.9,	0.246],
[	9.27,	1.92,	0.246],
[	9.36,	1.95,	0.246],
[	9.46,	1.97,	0.246],
[	9.56,	2,	0.246],
[	9.65,	2.02,	0.246],
[	9.75,	2.04,	0.246]])



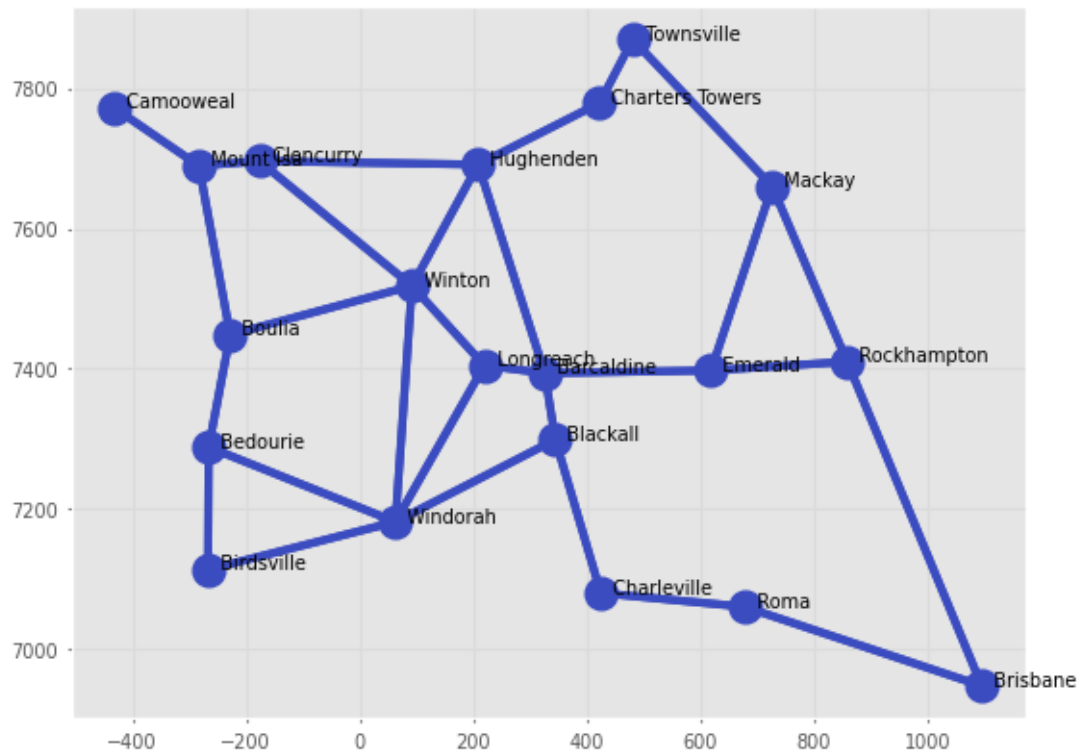
1.2 Graph-based planning

```
[9]: from pgraph import UGraph # from PGraph package
from roboticstoolbox import rtb_path_to_datafile # from Robotics Toolbox package
import json

# load an example route map from the rtb-data package
with open(rtb_path_to_datafile('data/queensland.json'), 'r') as f:
    data = json.loads(f.read())

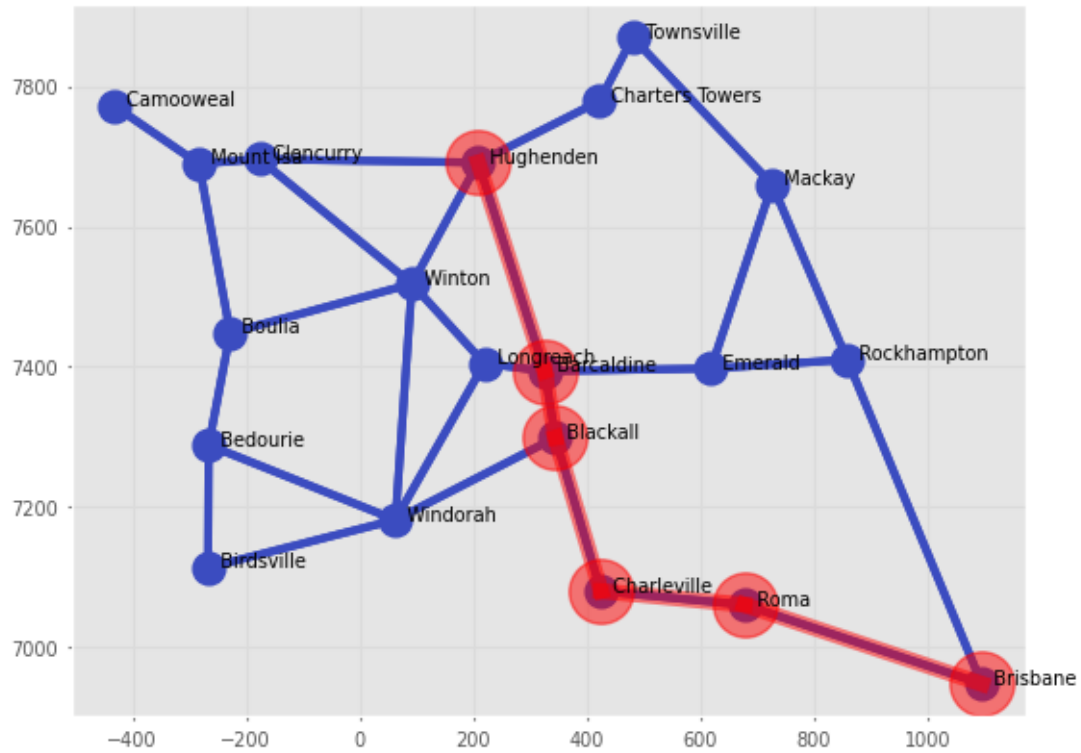
g = UGraph() # create an undirected graph
for name, info in data['places'].items():
    g.add_vertex(name=name, coord=info["utm"]) # add places as vertices
for route in data['routes']:
    g.add_edge(route['start'], route['end'], cost=route['distance']) # add
    ↪ routes as edges

g.plot()
```



```
[10]: path, length, parents = g.path_Astar('Hughenden', 'Brisbane')

g.plot(block=None)
g.highlight_path(path)
```

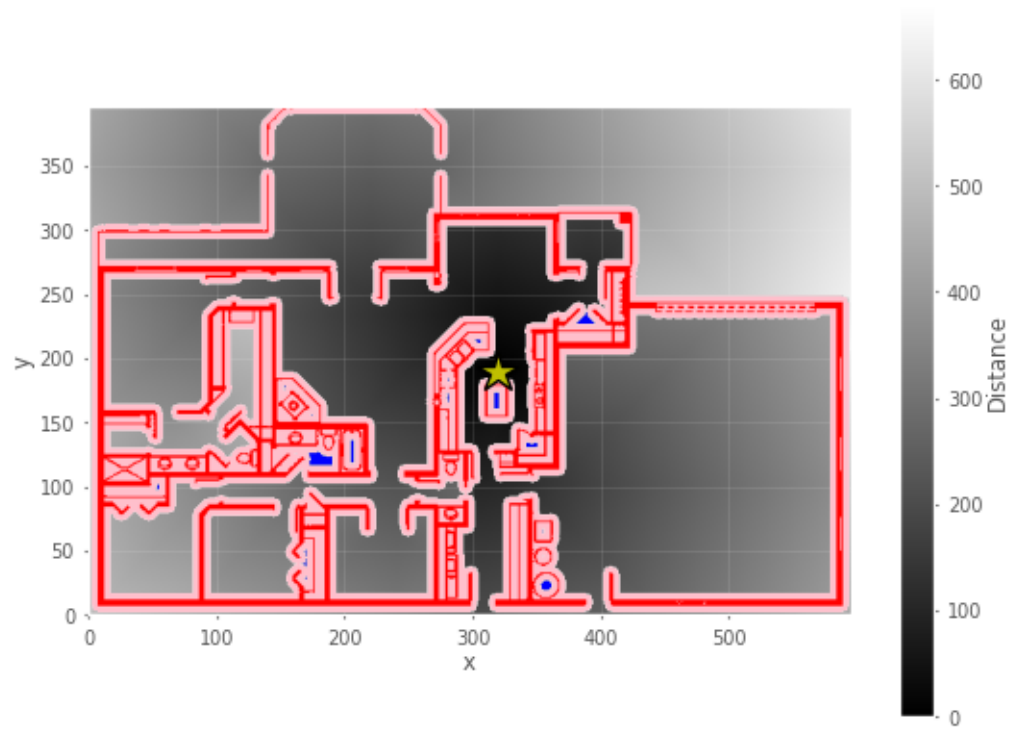


1.3 Occupancy grid path planning

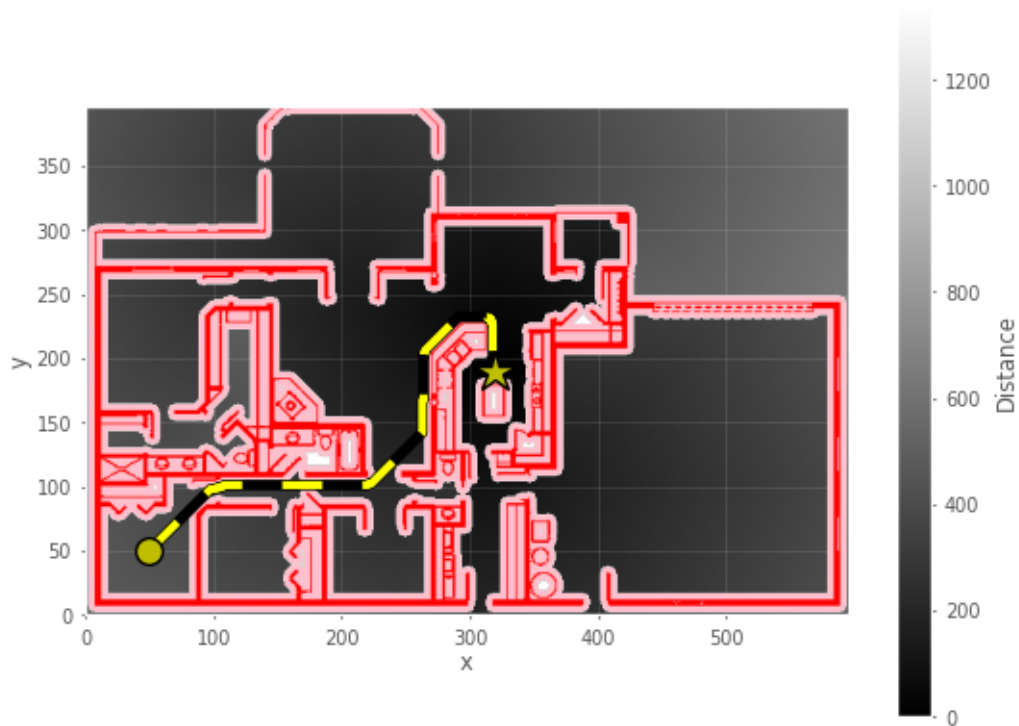
```
[11]: from roboticstoolbox import rtb_load_matfile, DistanceTransformPlanner

house = rtb_load_matfile('data/house.mat')
floorplan = house['floorplan']
places = house['places']

pmarker = dict(markersize=6, color='y')
dx = DistanceTransformPlanner(floorplan, inflate=5)
dx.plan(places.kitchen)
dx.plot();
```



```
[12]: p = dx.query(places.br3)
      dx.plot(p, inflated=True, path_marker=pmarker);
```



1.4 Sample-based planning

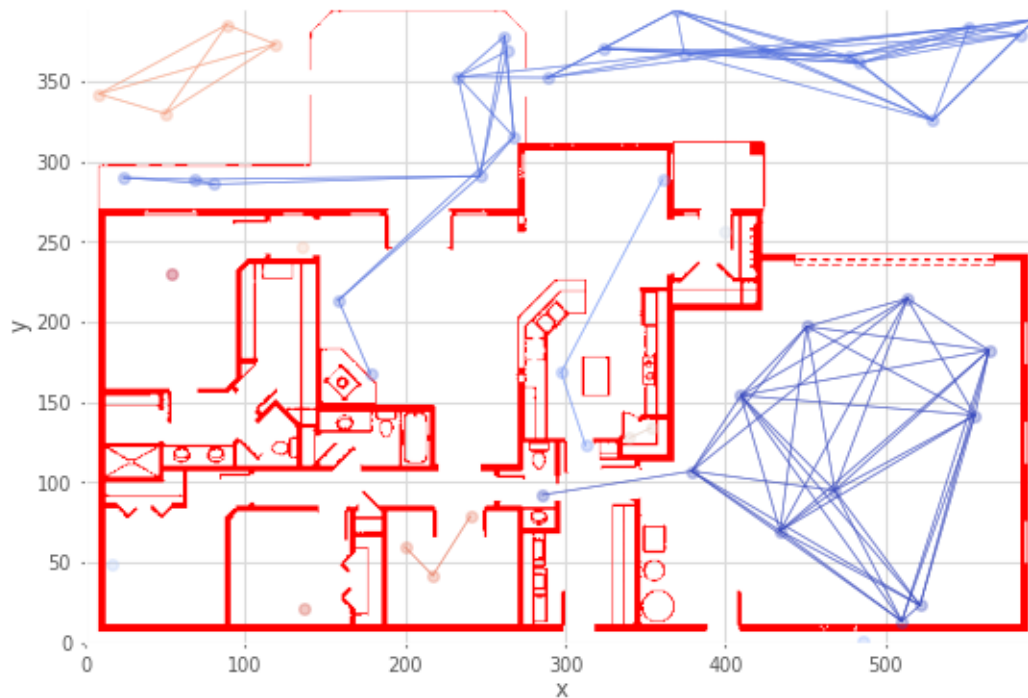
```
[13]: from roboticstoolbox import PRMPlanner

house = rtb_load_matfile('data/house.mat')
floorplan = house['floorplan']
places = house['places']

prm = PRMPlanner(occgrid=floorplan, seed=0)
prm.plan(npoints=50)

prm.plot(edge=dict(alpha=0.3), vertex=dict(alpha=0.3))
print(prm)
```

plotted background



plotted roadmap

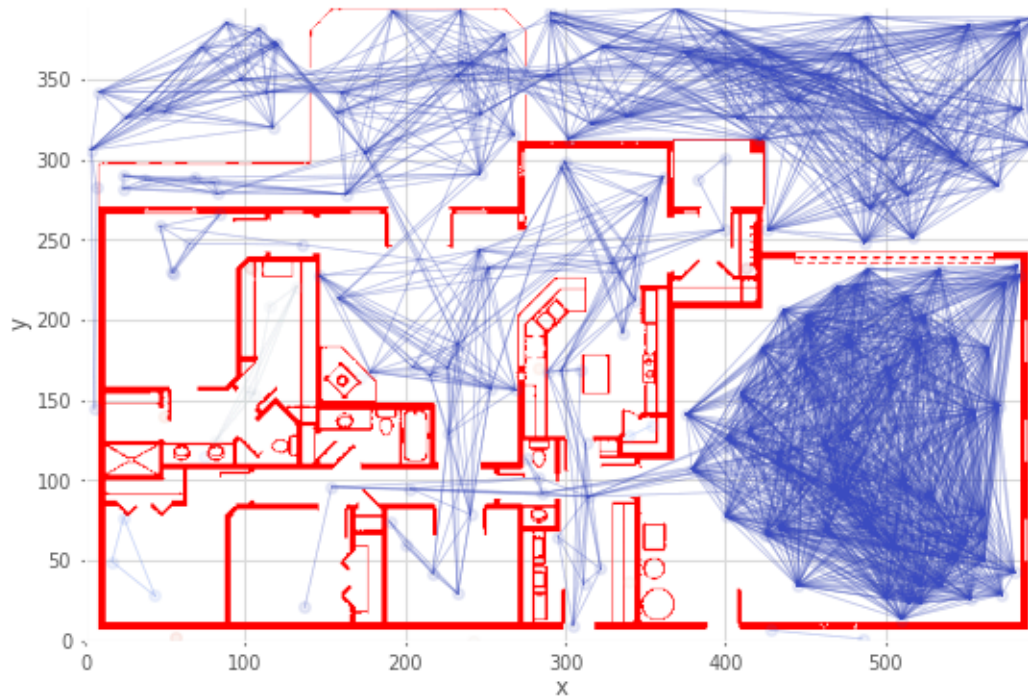
PRMPlanner:

BinaryOccupancyGrid: 596 x 397, cell size=1, x = [0.0, 595.0], y = [0.0, 396.0], 8.8% occupied

UGraph: 50 vertices, 222 edges, 12 components

```
[14]: # reseed the PRN to get a workable solution
# prm = PRMPlanner(occgrid=floorplan, seed=2)
prm.plan(npoints=200)
prm.plot(edge=dict(alpha=0.1), vertex=dict(alpha=0.1))
print(prm)
```

plotted background



plotted roadmap

PRMPlanner:

BinaryOccupancyGrid: 596 x 397, cell size=1, x = [0.0, 595.0], y = [0.0, 396.0], 8.8% occupied

UGraph: 200 vertices, 4464 edges, 14 components

1.5 Planning with motion constraints

1.5.1 Dubbins paths

```
[15]: from roboticstoolbox import DubinsPlanner
```

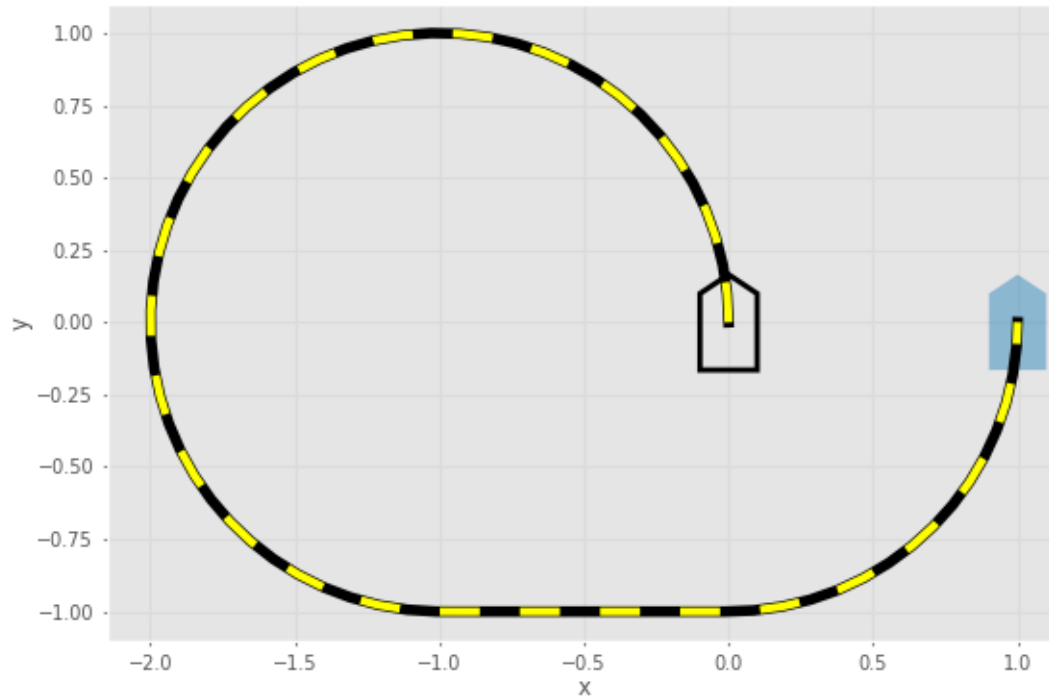
```
qs = (0, 0, math.pi/2)
```

```
qg = (1, 0, math.pi/2)
```

```
dubins = DubinsPlanner(curvature=1)
```

```
path, status = dubins.query(qs, qg)
```

```
dubins.plot(path);
```

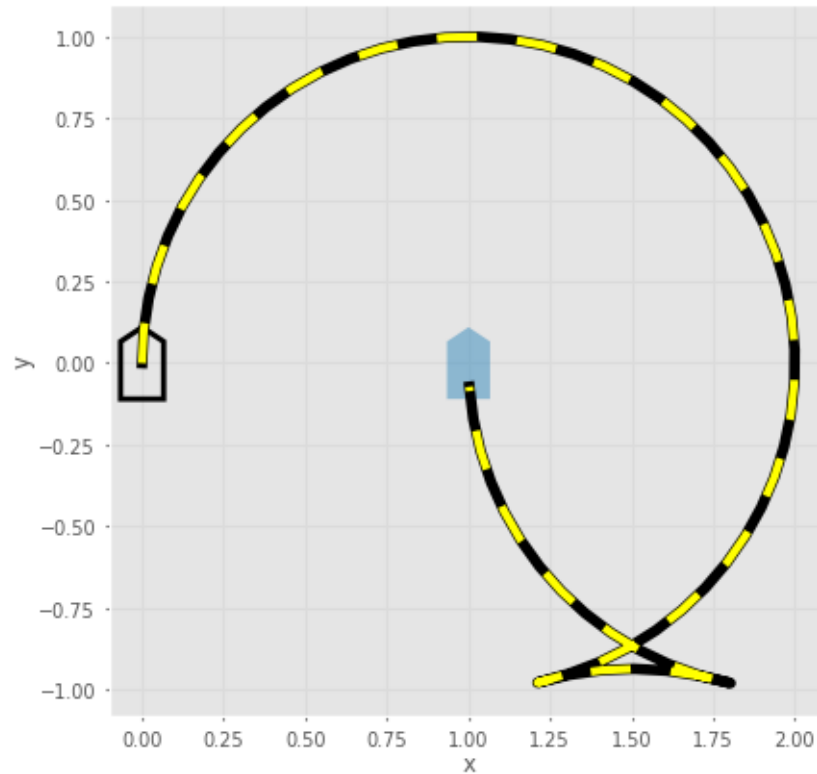


1.5.2 Reeds-Shepp path

```
[16]: from roboticstoolbox import ReedsSheppPlanner

rs = ReedsSheppPlanner(curvature=1)
path, status = rs.query(qs, qg)

rs.plot(path);
```



1.5.3 Configuration-space planning

Let's look at the classic piano mover's problem

```
[17]: from spatialmath import Polygon2
from roboticstoolbox import PolygonMap, RRTPlanner, Bicycle

# start and goal configuration
qs = (2, 8, -math.pi/2)
qg = (8, 2, -math.pi/2)

# obstacle map
map = PolygonMap(workspace=[0, 10])
map.add([(5, 50), (5, 6), (6, 6), (6, 50)])
map.add([(5, 4), (5, -50), (6, -50), (6, 4)])

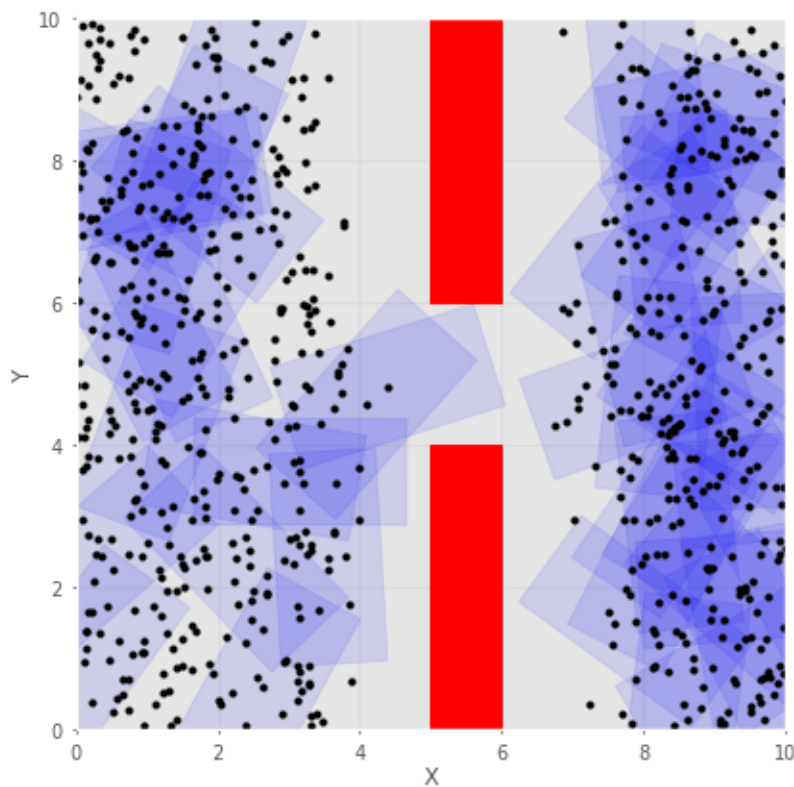
# create a polygon to represent the piano
length, width = 3, 1.5
piano = Polygon2(
    np.array([(-length / 2, width / 2), (-length / 2, -width / 2), (length / 2, -width / 2), (length / 2, width / 2)]).T
)
```

```
# the piano has bicycle kinematics with a maximum steering angle of 1 radian,
↳ and a wheelbase of 2m (ok, it's an odd piano)
vehicle = Bicycle(steer_max=1, L=2, polygon=piano)

rrt = RRTPlanner(map=map, vehicle=vehicle, verbose=False, npoints=50,
↳ showsamples=True, seed=0)
```

We'll build an RRT with its goal `qg` in the lower-right of the figure

```
[18]: map.plot()
rrt.plan(goal=qg)
```



We have found a bunch of piano poses that don't intersect with the red obstacle, a dot represents the position (x, y) and the translucent rectangle indicates its orientation. Then we joined them up using an RRT.

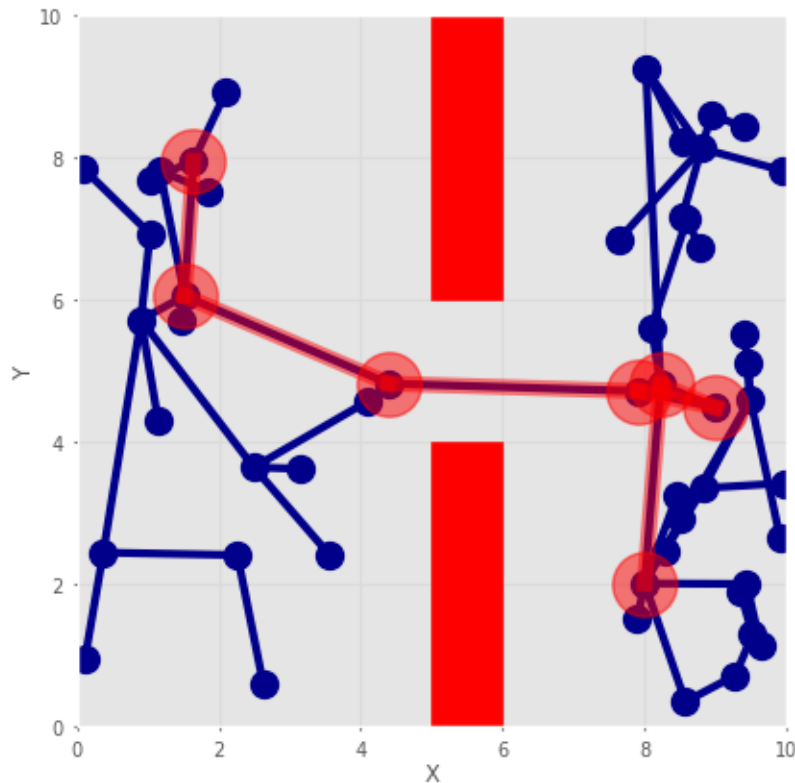
Now we can query for a path to the goal, given a start pose `qs` which is in the upper left of the figure.

```
[19]: path, status = rrt.query(start=qs)
print(status)
```

```
[DVertex[#34, coord=(1.644, 7.95, -2.051)], DVertex[#25, coord=(1.52, 6.047,
-2.131)], DVertex[#23, coord=(4.394, 4.813, 0.3043)], DVertex[#17, coord=(7.913,
4.707, 0.3379)], DVertex[#11, coord=(8.999, 4.479, -0.5868)], DVertex[#3,
coord=(8.224, 4.8, -1.682)], DVertex[#0, coord=(8, 2, -1.571)]]
RRTStatus(length=19.543490758346646, initial_d=np.float64(0.5996466379914136),
vertices=[DVertex[#34, coord=(1.644, 7.95, -2.051)], DVertex[#25, coord=(1.52,
6.047, -2.131)], DVertex[#23, coord=(4.394, 4.813, 0.3043)], DVertex[#17,
coord=(7.913, 4.707, 0.3379)], DVertex[#11, coord=(8.999, 4.479, -0.5868)],
DVertex[#3, coord=(8.224, 4.8, -1.682)], DVertex[#0, coord=(8, 2, -1.571)]]])
```

A path has been found and is described by a set of RRT vertices with pose (x, y, θ) .

```
[20]: map.plot()
rrt.g.plot(colorcomponents=False, text=False, force2d=True,
          vopt=dict(color='darkblue', marker='o', markersize=10),
          eopt=dict(color='darkblue', linewidth=3), block=None)
rrt.g.highlight_path(status.vertices, color='r')
```



The planner returns a smoothed path by fitting Dubins curves between the RRT vertices, resulting in a path that is driveable by the piano

```
[21]: with np.printoptions(threshold=20):
      print(path)
```

```

[[ 1.64427294  7.94970389 -2.05090532]
 [ 1.57019221  7.76428843 -1.85090532]
 [ 1.53442453  7.56785138 -1.65090532]
 ...
 [ 8.01359372  2.20839008 -1.65153633]
 [ 8.00003956  2.00889515 -1.5796916 ]
 [ 8.         2.         -1.57079633]]

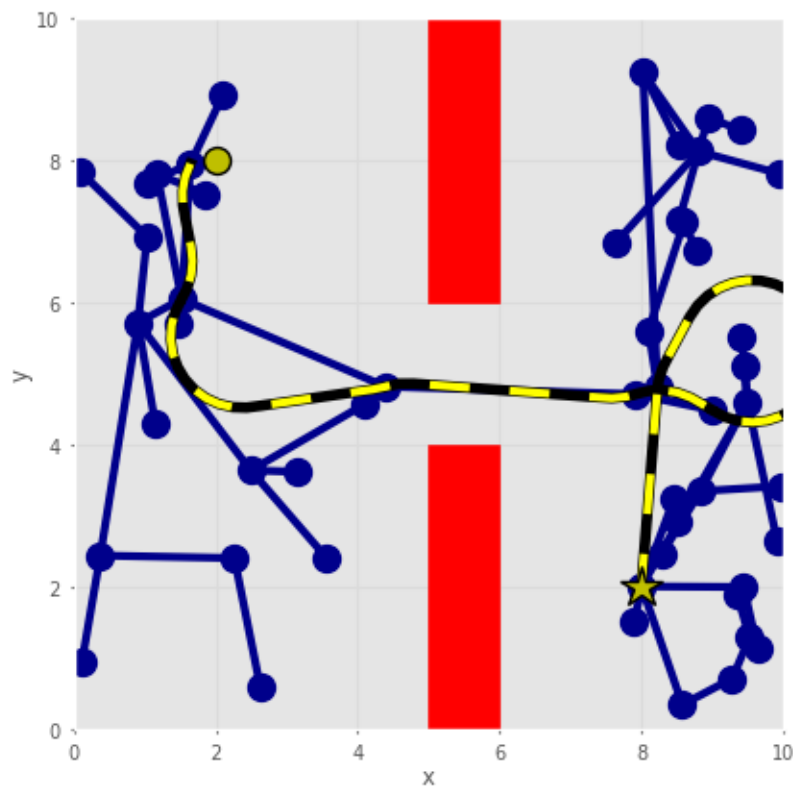
```

We can overlay that on the RRT

```

[22]: map.plot()
rrt.g.plot(colorcomponents=False, text=False, force2d=True,
           vopt=dict(color='darkblue', marker='o', markersize=10),
           eopt=dict(color='darkblue', linewidth=3), block=None)
rrt.plot(path);

```



and animate it as a series of snapshots of the piano on its journey

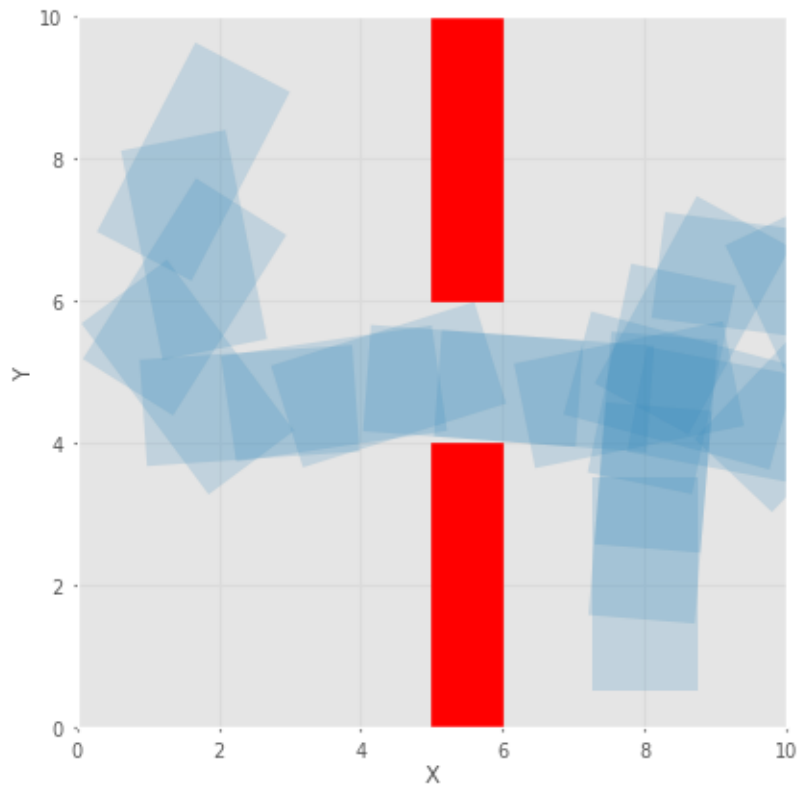
```

[23]: from roboticstoolbox import VehiclePolygon

va = VehiclePolygon(piano)
map.plot(block=None)
for i in np.unique(np.rint(np.linspace(0, len(path) - 1, 20)).astype(int)):

```

```
va.plot(path[i, :], alpha=0.2)
```



1.6 Pose-graph optimization

In this example we load the classic “Killian Court” SLAM dataset from a TORO file. The vertices of the graph represent robot pose estimated from wheel odometry. The edges represent constraints from lidar scan matching of unique landmarks in the scene.

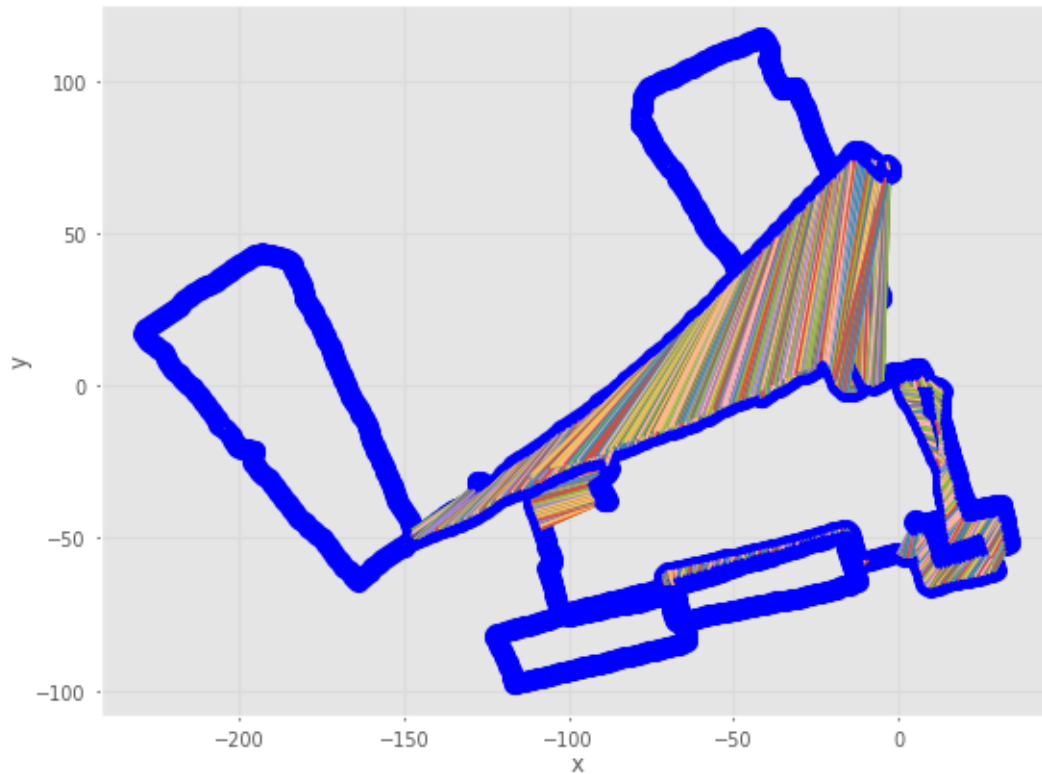
We can load the data and display the vertices (blue circles) and the edges (colored line).

```
[24]: from roboticstoolbox import PoseGraph

pg = PoseGraph('data/killian-small.toro')

pg.plot(text=False, block=None)
```

loaded TORO/LAGO format file: 1941 vertices, 3995 edges



and we can see the effect of odometry pose drift over multiple passes around the corridors.

We can perform pose graph optimization to adjust the vertices so as to bring them into line with the lidar observations.

```
[25]: pg.optimize()

pg.plot(text=False, block=None)
```

```
done in 841.25 msec. Total cost 1.78135e+06
```

```
done in 854.12 msec. Total cost 1.39949e+06
```

```
done in 877.29 msec. Total cost 36.6732
```

```
done in 880.30 msec. Total cost 5.44579
```

```
done in 864.90 msec. Total cost 5.44565
```

```
done in 865.86 msec. Total cost 5.44567
```



2 Arm robotics

We will load a Denavit-Hartenberg model of the Franka “Panda” robot and display it

```
[26]: from roboticstoolbox import models

panda = models.DH.Panda()

print(panda)
```

DHRobot: Panda (by Franka Emika), 7 joints (RRRRRRR), dynamics, geometry, modified DH parameters

a		d	q	q	
0.0	0.0°	q1	0.333	-166.0°	166.0°
0.0	-90.0°	q2	0.0	-101.0°	101.0°
0.0	90.0°	q3	0.316	-166.0°	166.0°
0.0825	90.0°	q4	0.0	-176.0°	-4.0°
-0.0825	-90.0°	q5	0.384	-166.0°	166.0°
0.0	90.0°	q6	0.0	-1.0°	215.0°
0.088	90.0°	q7	0.107	-166.0°	166.0°

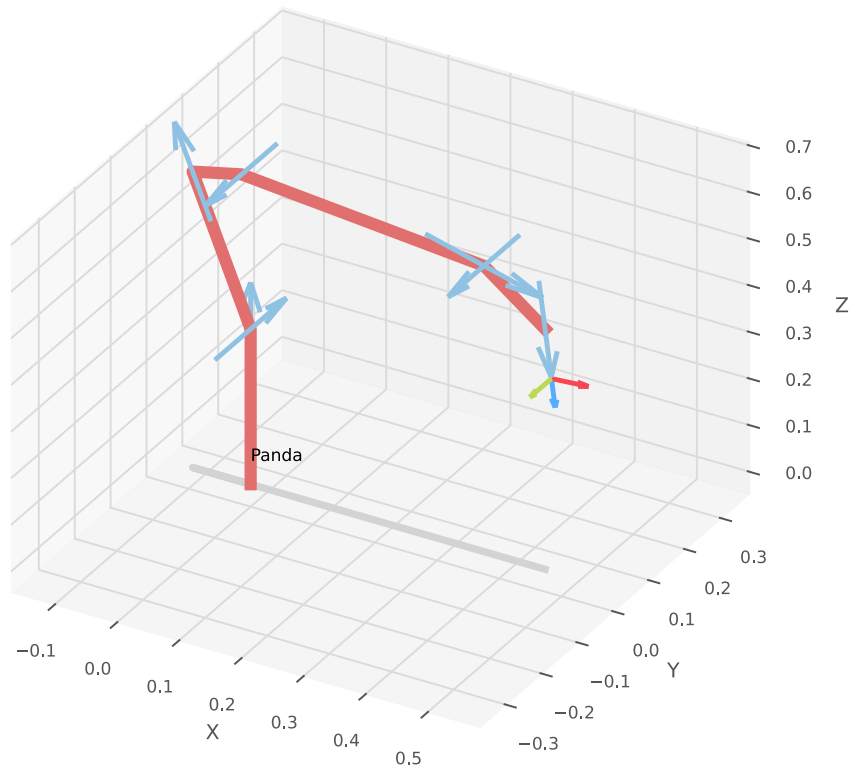
```
tool    t = 0, 0, 0.1; rpy/xyz = -45°, 0°, 0°
```

name	q0	q1	q2	q3	q4	q5	q6
qr	0°	-17.2°	0°	-126°	0°	115°	45°
qz	0°	0°	0°	0°	0°	0°	0°

The model contains a list of links, base and tool transforms, and a set of useful named joint coordinates.

We can display the configuration of the robot at the “ready” coordinate

```
[27]: q = panda.qr
panda.plot(q)
```



```
[27]: PyPlot3D backend, t = 0.05, scene:
      robot: Text(0.0, 0.0, 'Panda')
```

The pose of the tool tip can be computed, it's an SE3 object

```
[28]: panda.fkine(q)
```

```
[28]:  0.995      0      0.09983  0.484
      0      -1      0      0
      0.09983  0     -0.995  0.413
      0      0      0      1
```

Now let's decide we want the tool tip to be at (0.5, 0.2, 0.5) and pointing straight downwards (Z-axis down). We can easily compute the inverse kinematics numerically

```
[29]: from spatialmath import SE3

sol = panda.ikine_LM(SE3.Trans(0.5, 0.2, 0.5) * SE3.Rx(math.pi) )
print(sol)
```

```
IKSolution: q=[-1.558, -1.041, 1.097, -1.619, 0.9719, 1.193, 0.5013],
success=True, iterations=15, searches=1, residual=1.3e-11
```

and the resulting `sol` object contains the status (yes, we were successful), the joint coordinates, as well as information about the residual and number of iterations and restarts. We can validate the solution

```
[30]: q = sol.q
      panda.fkine(q)
```

```
[30]:  1      2.15e-12 -2.671e-12  0.5
      2.15e-12 -1      1.307e-11  0.2
      -2.671e-12 -1.307e-11 -1      0.5
      0      0      0      1
```

We can compute a joint-space trajectory between two sets of coordinates

```
[31]: from roboticstoolbox import jtraj

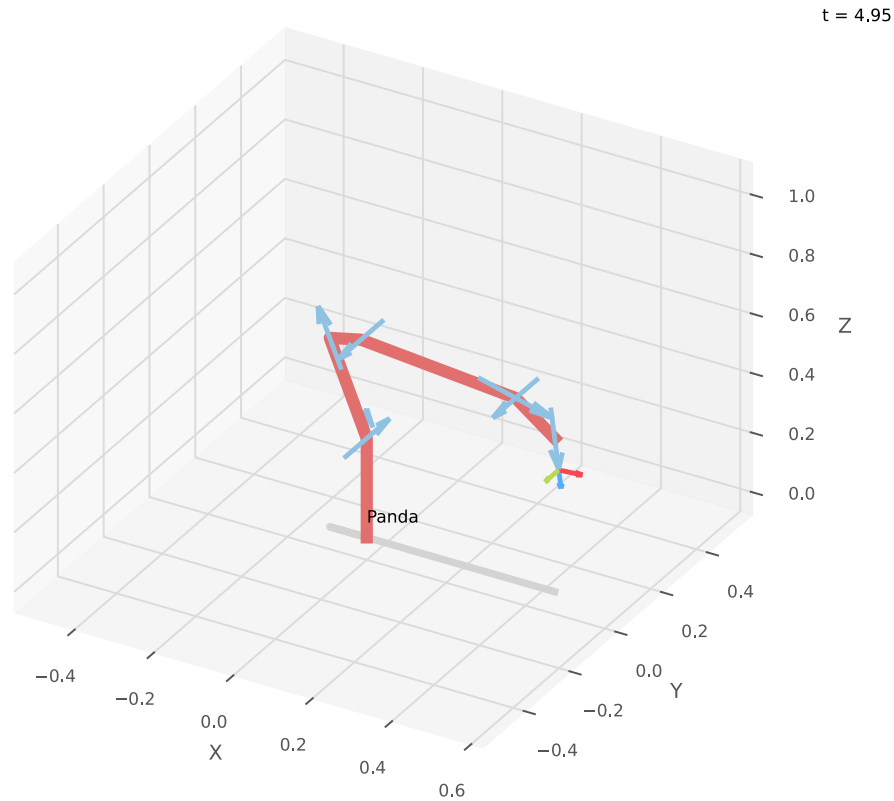
tg = jtraj(panda.qz, panda.qr, 100)
print(tg)
```

Trajectory created by `jtraj`: 100 time steps x 7 axes

The `tg` object contains the joint coordinates, joint rates, and joint accelerations as time series.

We can pass a trajectory (100x7 matrix) to plot and the result will be an animation

```
[32]: panda.plot(tg.q)
```



```
[32]: PyPlot3D backend, t = 4.999999999999999, scene:
      robot: Text(0.0, 0.0, 'Panda')
```

and if we pass the trajectory to `fkine` we get an `SE3` object that contains multiple values, which we can slice as discussed earlier today

```
[33]: T_tg = panda.fkine(tg.q)
      print(len(T_tg))
      print(T_tg[10])
```

```
100
 0.712    0.7022    0.0008808  0.09379
 0.7022   -0.712     0          0
 0.0006271 0.0006185 -1         0.8246
 0         0         0         1
```

I've never really liked DH parameters so we can also create a kinematic model using ETS notation

```
[34]: from roboticstoolbox import models

      panda = models.ETS.Panda()
```

```
panda
```

[34]: ERobot: Panda (by Franka Emika), 7 joints (RRRRRRR)

link	link	joint	parent	ETS: parent to link			
0	link0	0	BASE	tz(0.333)	Rz(q0)		
1	link1	1	link0	Rx(-90°)	Rz(q1)		
2	link2	2	link1	Rx(90°)	tz(0.316)	Rz(q2)	
3	link3	3	link2	tx(0.0825)	Rx(90°)	Rz(q3)	
4	link4	4	link3	tx(-0.0825)	Rx(-90°)	tz(0.384)	Rz(q4)
5	link5	5	link4	Rx(90°)	Rz(q5)		
6	link6	6	link5	tx(0.088)	Rx(90°)	tz(0.107)	Rz(q6)
7	@ee		link6	tz(0.103)	Rz(-45°)		

name	q0	q1	q2	q3	q4	q5	q6
qr	0°	-17.2°	0°	-126°	0°	115°	45°
qz	0°	0°	0°	0°	0°	0°	0°

In ETS the model is just one long string of transforms, some constant, some are joint variables

```
[35]: ets = panda.ets()  
print(ets)
```

```
tz(0.333) Rz(q0) Rx(-90°) Rz(q1) Rx(90°) tz(0.316) Rz(q2)  
tx(0.0825) Rx(90°) Rz(q3) tx(-0.0825) Rx(-90°) tz(0.384) Rz(q4)  
Rx(90°) Rz(q5) tx(0.088) Rx(90°) tz(0.107) Rz(q6) tz(0.103)  
Rz(-45°)
```

which is another list-like sliceable object

```
[36]: ets[2:4]
```

[36]: [Rx(-90°), Rz(q1)]

I can write an ETS string like that for **any robot**. There are none of the DH rule to get in the way.

Then I can turn that into a robot object

```
[37]: from roboticstoolbox import Robot  
  
r = Robot(ets)  
print(r)
```

ERobot: , 7 joints (RRRRRRR)

link	link	joint	parent	ETS: parent to link
0	link0	0	BASE	tz(0.333) Rz(q0)
1	link1	1	link0	Rx(-90°) Rz(q1)
2	link2	2	link1	Rx(90°) tz(0.316) Rz(q2)
3	link3	3	link2	tx(0.0825) Rx(90°) Rz(q3)
4	link4	4	link3	tx(-0.0825) Rx(-90°) tz(0.384) Rz(q4)
5	link5	5	link4	Rx(90°) Rz(q5)
6	link6	6	link5	tx(0.088) Rx(90°) tz(0.107) Rz(q6)
7	@link7		link6	tz(0.103) Rz(-45°)

and compute its forward kinematics

```
[38]: r.fkine(q)
```

```
[38]: 1          2.15e-12 -2.671e-12  0.5
      2.15e-12 -1          1.307e-11  0.2
      -2.671e-12 -1.307e-11 -1          0.5
      0          0          0          1
```

For any robot we can compute Jacobians

```
[39]: panda.jacob0(panda.qr)
```

```
[39]: array([[ 0.,  0.08,  0.,  0.246,  0.,  0.2,  0.],
 [ 0.484,  0.,  0.486,  0.,  0.154,  0.,  0.],
 [ 0., -0.484,  0.,  0.499,  0.,  0.109,  0.],
 [ 0.,  0., -0.296,  0.,  0.946,  0.,  0.0998],
 [ 0.,  1.,  0., -1.,  0., -1.,  0.],
 [ 1.,  0.,  0.955,  0., -0.323,  0., -0.995]])
```

```
[40]: panda.jacobe(q)
```

```
[40]: array([[ -0.2,  0.00211,  0.0429,  0.0534,  0.0638,  0.202,  0.],
 [ -0.5,  0.167, -0.255, -0.319, -0.219,  0.0589,  0.],
 [ 0., -0.194,  0.434, -0.22,  0., -0.088,  0.],
 [ 0.,  1., -0.0109, -0.45,  0.892, -0.28,  0.],
 [ 0., -0.0126, -0.863,  0.456,  0.261,  0.96,  0.],
 [ -1.,  0., -0.506, -0.768, -0.369,  0.,  1.]])
```

and Hessians

```
[41]: H = panda.hessian0(q)
      H.shape
```

```
[41]: (7, 6, 7)
```

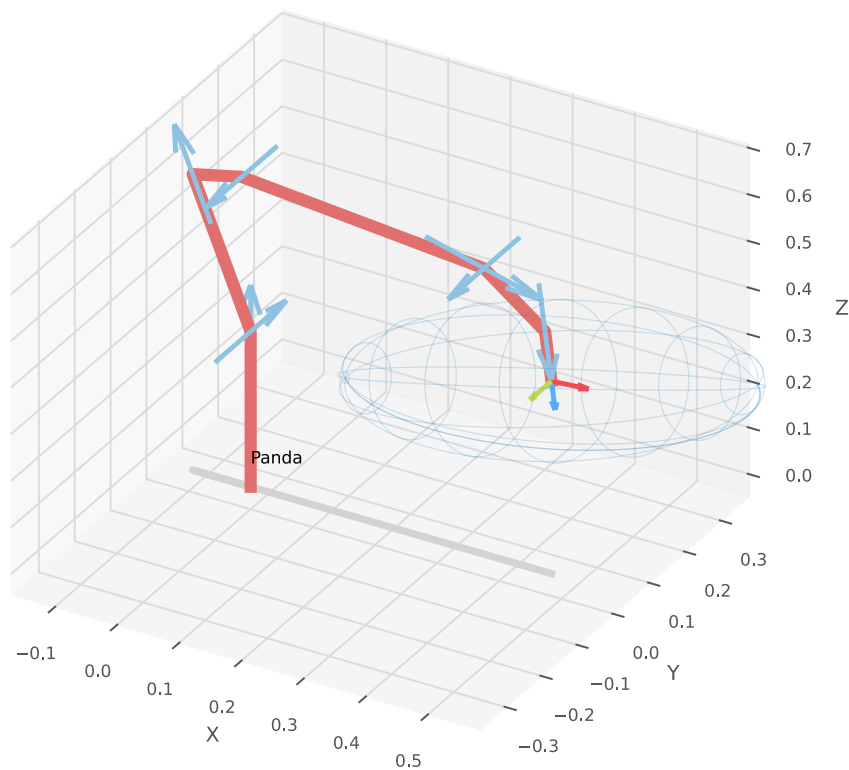
The robot's manipulability in this configuration is summarized by

```
[42]: q = panda.qr
      panda.manipulability(q)
```

```
[42]: np.float64(0.08375150968113343)
```

But we can get a better understanding by displaying the velocity ellipsoid (for translational motion)

```
[43]: panda.plot(q)
      panda.vellipse(q);
```



We can also import a model from URDF files

```
[44]: panda = models.URDF.Panda()
      panda
```

[44]: ERobot: panda (by Franka Emika), 7 joints (RRRRRRR), 1 gripper, geometry, collision

link	link	joint	parent	ETS: parent to link	
0	panda_link0		BASE	SE3()	
1	panda_link1	0	panda_link0	SE3(0, 0, 0.333)	Rz(q0)
2	panda_link2	1	panda_link1	SE3(-90°, -0°, 0°)	Rz(q1)
3	panda_link3	2	panda_link2	SE3(0, -0.316, 0; 90°, -0°, 0°)	Rz(q2)
4	panda_link4	3	panda_link3	SE3(0.0825, 0, 0; 90°, -0°, 0°)	Rz(q3)
5	panda_link5	4	panda_link4	SE3(-0.0825, 0.384, 0; -90°, -0°, 0°)	Rz(q4)
6	panda_link6	5	panda_link5	SE3(90°, -0°, 0°)	Rz(q5)
7	panda_link7	6	panda_link6	SE3(0.088, 0, 0; 90°, -0°, 0°)	Rz(q6)
8	@panda_link8		panda_link7	SE3(0, 0, 0.107)	

name	q0	q1	q2	q3	q4	q5	q6
qr	0°	-17.2°	0°	-126°	0°	115°	45°
qz	0°	0°	0°	0°	0°	0°	0°

[45]: `# panda.plot(q, backend="swift")`

To show the dynamics capability we will load one of the oldest models in the toolbox which has quite an accurate dynamic model.

[46]: `puma = models.DH.Puma560()
print(puma)`

DHRobot: Puma 560 (by Unimation), 6 joints (RRRRRR), dynamics, geometry, standard DH parameters

d	a	q	q
---	---	---	---

q1	0.6718	0	90.0°	-160.0°	160.0°
q2	0	0.4318	0.0°	-110.0°	110.0°
q3	0.15	0.0203	-90.0°	-135.0°	135.0°
q4	0.4318	0	90.0°	-266.0°	266.0°
q5	0	0	-90.0°	-100.0°	100.0°
q6	0	0	0.0°	-266.0°	266.0°

name	q0	q1	q2	q3	q4	q5
qr	0°	90°	-90°	0°	0°	0°
qz	0°	0°	0°	0°	0°	0°
qn	0°	45°	180°	0°	45°	0°
qs	0°	0°	-90°	0°	0°	0°

We see that it is tagged with “dynamics” at the top of the tables above.

The dynamic parameters, rigid-body and friction, are

[47]: `puma.dynamics()`

	j	m	r	I
	Jm	B	Tc	G
link1	0	0, 0, 0	0, 0.35, 0, 0, 0, 0	0.0002, 0.00148, 0.395, -0.435, -62.6
link2	17.4	-0.364, 0.006, 0.228	0.13, 0.524, 0.539, 0, 0, 0	0.0002, 0.000817, 0.126, -0.071, 108
link3	4.8	-0.0203, -0.0141, 0.07	0.066, 0.086, 0.0125, 0, 0, 0	0.0002, 0.00138, 0.132, -0.105, -53.7
link4	0.82	0, 0.019, 0	0.0018, 0.0013, 0.0018, 0, 0, 0	0, 3.3e-05, 7.12e-05, 0.0112, -0.0169, 76
link5	0.34	0, 0, 0	0.0003, 0.0004, 0.0003, 0, 0, 0	0, 3.3e-05, 8.26e-05, 0.00926, -0.0145, 71.9
link6	0.09	0, 0, 0.032	0.00015, 0.00015, 4e-05, 0, 0, 0	0, 3.3e-05, 3.67e-05, 0.00396, -0.0105, 76.7

For a notional pose (not singular) the rigid-body matrix terms can be easily computed

```
[48]: q = puma.qn
```

```
puma.inertia(q)
```

```
[48]: array([[ 3.66, -0.404, 0.101, -0.00252, 0, 0],
 [ -0.404, 4.41, 0.351, 0, 0.00236, 0],
 [ 0.101, 0.351, 0.938, 0, 0.00148, 0],
 [-0.00252, 0, 0, 0.193, 0, 2.83e-05],
 [ 0, 0.00236, 0.00148, 0, 0.171, 0],
 [ 0, 0, 0, 2.83e-05, 0, 0.194]])
```

```
[49]: puma.gravload(q)
```

```
[49]: array([ 0, 31.6, 6.04, 0, 0.0283, 0])
```

```
[50]: puma.coriolis(q, [1, 0, 0, 0, 0, 0])
```

```
[50]: array([[ 0, -0.628, 0.361, 0.000306, 0, 0],
 [ 0.628, 0, 0, -0.000586, 0, -2e-05],
 [-0.361, 0, 0, 3.58e-05, 0, -2e-05],
 [-0.000306, 0.000586, -3.58e-05, 0, 0.000156, 0],
 [ 0, 0, 0, -0.000156, 0, -2e-05],
 [ 0, 2e-05, 2e-05, 0, 2e-05, 0]])
```

Or, in task space they are

```
[51]: puma.inertia_x(q)
```

```
[51]: array([[ 17.3, -2.75, -9.62, 0, 0.28, 0],
 [ -2.75, 11.6, 1.25, 6.71e-05, -0.0703, 6.71e-05],
 [ -9.62, 1.25, 13.3, 0, 0.277, 0],
 [ 0, 6.71e-05, 0, 0.194, 0, 0.194],
 [ 0.28, -0.0703, 0.277, 0, 0.171, 0],
 [ 0, 6.71e-05, 0, 0.194, 0, 0.194]])
```

```
[52]: puma.gravload_x(q)
```

```
[52]: array([ -30.1, 7.58, 53.7, 0, -0.0283, 0])
```

3 What next for Robotics Toolbox?

- play nicely with other packages: Coal, OMPL, ikfast/ssik, Rerun, Pinocchio, CoppeliaSim