

Numerical Methods for Graphics

Topics: Smoothness – Path Smoother Spline

University of Florence

Armand Palla-7042287



Table of contents:

1. Introduction
2. C1-continuous path
3. C2-continuous path
4. Path Smoother Spline - Implementation
5. Conclusions
6. References



1. **Introduction**

Automated Driving Toolbox provides algorithms and tools for designing, simulating, and testing autonomous driving systems. This toolbox let us import and work with the **smoothPathSpline** and other functions in order to execute our implementation. The smoothness of a function is a property measured by the number of **continuous** **derivatives** it has over some domain. The **derivative** of a function of a real variable measures the sensitivity to change of the function value with respect to a change in its argument. At the very minimum, a function could be considered smooth if it is **differentiable** or **continuous** everywhere. A differentiable function of one real variable is a function whose derivative exists at each point in its domain.

**Differentiability class** is a classification of functions according to the properties of their derivatives. It is a measure of the highest order of derivative that exists for a function. Consider an open set on the real line and a function **f** defined on that set with real values. Let **k** be a non-negative integer. The function **f** is said to be of (differentiability) class **C^k** if the derivatives **f′, f″, ..., f(k)** exists and are continuous. The function **f** is said to be infinitely differentiable, smooth, or of class **C^∞**, if it has derivatives of all orders. A function:



defined on an open set **U** is differentiable at **a ∈ U** if the derivative



exists. This implies that the function is continuous at **a.** This function **f** is differentiable on **U** if it is differentiable at every point of **U**.

Let **f**: R→R be a real function.



Then f(x) is of **differentiability class C^k** if and only if:



where **C** denotes the **class of continuous** real functions.

That is, **f** is in **differentiability class k** if and only if there exists a **k^th** derivative of **f**, which is continuous. If:



is **continuous** for all **k∈N**, then f(x) is of differentiability class **C^∞**(infinitely differentiable).

**Class Zero**

The class **C^0** consists of all continuous functions whether they be differentiable or not.

Ex:

Let **f** be the real function defined as:



Then **f ∈ C^0** but **f ∉ C^1.** Why?

By inspection it is seen that **f** is continuous everywhere. We have that:

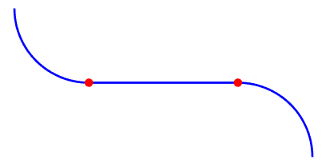


Hence **f′** is not continuous at x=0.

So, by definition of **differentiability class**, **f** is not a member of **C^1**.



1. **C1-continuous path**

A path is **C^1**-continuous if its derivative exists and is continuous. Paths that are only **C^1**-continuous have discontinuities in their curvature. For example, a path may have discontinuities in curvature at the points where the segments join. These discontinuities result in changes in direction that are not smooth enough for driving with passengers. 

Ex: Let **f** be the real function defined as:



Then **f ∈ C^1** but **f ∉ C^2.** By inspection it is seen that **f** is continuous everywhere. We have that:



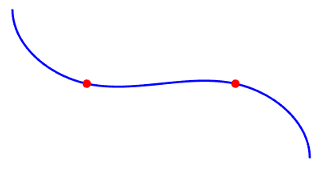
By inspection it is seen that **f′** is continuous everywhere. So, by definition of differentiability class, **f** is a member of **C^1**. Then we have that:



Hence **f′′** is not continuous at x=0.So by definition of differentiability class, **f** is not a member of **C^2**.



1. **C2-continuous path**

A path is considered **C^2**-continuous if its second derivative exists and is continuous. **C^2**-continuous paths have continuous curvature and are smooth enough for driving with passengers.

More generally, a function is said to be of class **C^k** if the first **k** derivatives all exist and are continuous.

If derivatives **f^n** exist for all positive integers n, the function is **smooth** or equivalently, of class **C^∞.**

Ex: Let’s see a general case:

Let a real function **f** be required that has the following properties:



where **C^k** denotes the differentiability class of order **k**.

Then **f** may be defined as:



By inspection it is seen that **f** is continuous everywhere. Then, we have:





By inspection it is seen that **f(n)** is continuous everywhere. So, by definition of differentiability class, **f** is a member of **C^n**.

Then we have that:



Hence **f^(n+1)** is not continuous at x=0. So, by definition of differentiability class, **f** is not a member of **C^(n+1).**



1. **Path Smoother Spline - Implementation**

The Path Smoother Spline block generates a smooth vehicle path, consisting of a sequence of discretized poses, by fitting the input reference path poses to a cubic spline. Given the input reference path directions, the block also returns the directions that correspond to each pose. Let’s see the function **smothPathspline:**

**[poses,directions]=smoothPathSpline(refPoses,refDirections,numSmoothPoses)** generates a smooth vehicle path, consisting of numSmoothPoses discretized poses, by fitting the input reference path poses to a **cubic spline**(a [piecewise](https://www.sciencedirect.com/topics/engineering/piecewise) cubic function that interpolates a set of data points and guarantees smoothness at the data points). This function is used to convert a C1-continuous vehicle path to a C2-continuous path.

Let’s see the function **pathPlannerRRT:**

The **pathPlannerRRT** object configures a vehicle path planner based on the optimal rapidly exploring random tree (RRT) algorithm. An RRT path planner explores the environment around the vehicle by constructing a tree of random collision-free poses.

**planner = pathPlannerRRT(costmap)** returns a **pathPlannerRRT** object for planning a vehicle path, **costmap** is an object specifying the environment around the vehicle. Once the **pathPlannerRRT** object is configured, we use the **plan** function to plan a path from the start pose to the goal.

Let’s discuss the **properties** of the **planner(1x1 struct)** in order to understand also the way how it functions:



* **Costmap: Costmap** of the vehicle environment, specified as a **vehicleCostmap** object.
* **Goal Tolerance:** Tolerance around the goal pose, specified as a **[xTol, yTol, ΘTol]** vector. The path planner finishes planning when the vehicle reaches the goal pose within these tolerances for the (x, y) position and the orientation angle, Θ. The **xTol** and **yTol** values are in the same world units as the **vehicleCostmap**. **ΘTol** is in degrees.
* **Goal Bias:** Probability of selecting the goal pose instead of a random pose, specified as a real scalar in the range [0, 1]. Large values accelerate reaching the goal at the risk of failing to circumnavigate(bypass) obstacles.
* **Connection Method:** Method used to calculate the connection between consecutive poses, specified as '**Dubins**' or **'Reeds-Shepp'**:
  + The '**Dubins**' method contains a sequence of three primitive motions, each of which is one of these types:
    - Straight (forward)
    - Left turn at the maximum steering angle of the vehicle (forward)
    - Right turn at the maximum steering angle of the vehicle (forward)

A vehicle path composed of **Dubins** path segments allows motion in the forward direction only.



* The **'Reeds-Shepp'** method contains a sequence of three to five primitive motions, each of which is one of these types:
  + Straight (forward or reverse)
  + Left turn at the maximum steering angle of the vehicle (forward or reverse)
  + Right turn at the maximum steering angle of the vehicle (forward or reverse)
* **Connection Distance:** Maximum distance between two connected poses, specified as a positive real scalar. **pathPlannerRRT** computes the connection distance along the path between the two poses, with turns included. Larger values result in longer path segments between poses.
* **Min Turning Radius:** Minimum turning radius of the vehicle, specified as a positive real scalar. This value corresponds to the radius of the turning circle at the maximum steering angle. Larger values limit the maximum steering angle for the path planner, and smaller values result in sharper turns. The default value is calculated using a wheelbase of 2.8 meters with a maximum steering angle of 35 degrees.
* **MinIterations:** Minimum number of planner iterations for exploring the **costmap**, specified as a positive integer. Increasing this value increases the sampling of alternative paths in the costmap.
* **MaxIterations:** Maximum number of planner iterations for exploring the **costmap**, specified as a positive integer. Increasing this value increases the number of samples for finding a valid path. If a valid path is not found, the path planner exits after exceeding this maximum.



* **ApproximateSearch:** Enable approximate nearest neighbour search, specified as **true** or **false**. Set this value to true to use a faster, but approximate, search algorithm. Set this value to false to use an exact search algorithm at the cost of increased computation time.

**costmap = vehicleCostmap(combinedMap, 'CellSize', res);**

The **costmap** is used for collision checking of the randomly generated poses. The **vehicleCostmap** object creates a **costmap** that represents the planning search space around a vehicle. The **costmap** holds information about the environment, such as obstacles or areas that the vehicle cannot traverse. To check for collisions, the **costmap** inflates obstacles using the inflation radius specified in the **CollisionChecker** property. The costmap is used by path planning algorithms, to find collision-free paths for the vehicle to follow.

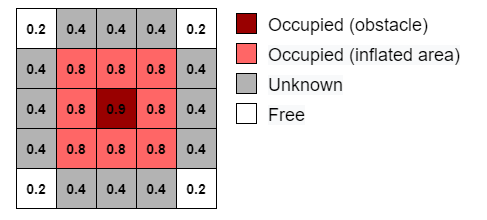
**Properties:**

* **FreeThreshold:** Threshold below which a grid cell is free, specified as a real scalar in the range [0, 1]. In our case it is put by default and the value is **0.2**
* **OccupiedThreshold:** Threshold above which a grid cell is occupied, specified as a real scalar in the range [0, 1]. In our case it is put by default and the value is **0.65**
* **CollisionChecker:** Collision-checking configuration, specified as an **InflationCollisionChecker** object. The fields of this object are:
  + **NumCircles**
  + **VehicleDimensions**
  + **CenterPlacements**
  + **InflationRadius**



* **MapExtent:** Extent of costmap around the vehicle, specified as a four-element, nonnegative integer vector of the form **[xmin xmax ymin ymax].**
  + **xmin** and **xmax** describe the length of the map in world coordinates.
  + **ymin** and **ymax** describe the width of the map in world coordinates.
* **CellSize:** Side length of each square cell, in world units, specified as a positive real scalar.
* **MapSize:** Size of **costmap** grid, specified as a two-element, positive integer vector of the form **[nrows ncols].**
  + - * **nrows** is the number of grid cell rows in the **costmap.**
      * **ncols** is the number of grid cell columns in the **costmap.**

The following figure shows a **costmap** with sample costs and grid cell states:





**Implementation – Smart Parking**

Automatically parking a car that is left in front of a parking lot is a challenging problem. The vehicle's automated systems are expected to take over control and steer the vehicle to an available parking spot. Such a function makes use of multiple on-board sensors. For example:

* Front and side cameras for detecting lane markings, road signs (stop signs, exit markings, etc.), other vehicles, and pedestrians
* Lidar and ultrasound sensors for detecting obstacles and calculating accurate distance measurements
* Ultrasound sensors for obstacle detection

On-board sensors are used to perceive the environment around the vehicle. The perceived environment includes an understanding of road markings to interpret road rules and infer drivable regions, recognition of obstacles, and detection of available parking spots.

As the vehicle sensors perceive the world, the vehicle must plan a path through the environment towards a free parking spot and execute a sequence of control actions needed to drive to it.

**Environment Model**

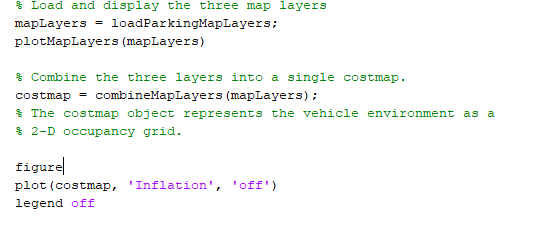
The environment model represents a map of the environment. For a parking system, this map includes available and occupied parking spots, road markings, and obstacles such as pedestrians or other vehicles. Occupancy maps are a common representation for this form of environment model. This example concentrates on a simpler scenario, where a map is already provided by a camera overlooking the entire parking space. It uses a static map of a parking lot and assumes that the self-localization of the vehicle is accurate.



The parking lot example used in this example is composed of three occupancy grid layers:

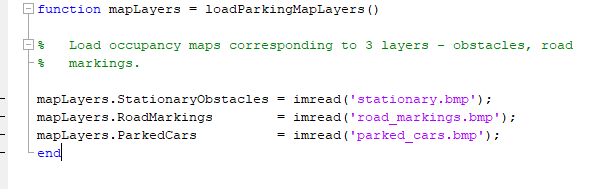
* **Stationary obstacles:** This layer contains stationary obstacles like walls, barriers, and bounds of the parking lot.
* **Road markings:** This layer contains occupancy information pertaining to road markings, including road markings for parking spaces.
* **Parked cars:** This layer contains information about which parking spots are already occupied. Each map layer contains different kinds of obstacles that represent different levels of danger for a car navigating through it. With this structure, each layer can be handled, updated, and maintained independently.

Let’s see the implementation of the code:



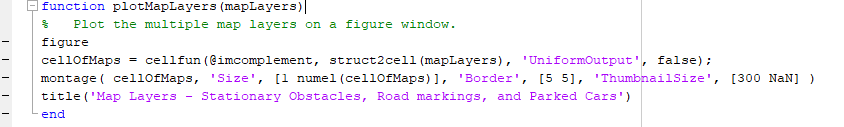


Let’s take a look at **loadParkingMapLayers** function:



For simplicity, I took **ready images** from Matlab and I have used it in my function.

The second is **plotMapLayers** function.



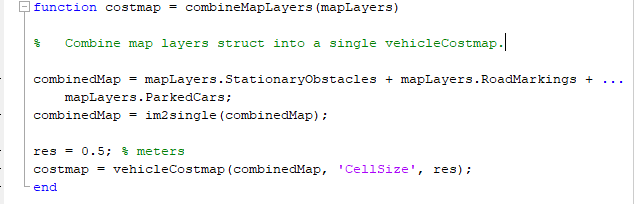
The **cellfun** applies function to each cell in cell array and takes as argument **@imcomplement**, **struct2cell(mapLayers)** and **UniformOutput** will value **false.**

* **@imcomplement** computes the complement of an image and returns the result. In the complement of a binary image, zeros become ones and ones become zeros; black and white images are reversed.
* **struct2cell(mapLayers)** converts a structure into a cell array.
* **'UniformOutput', false** means that **cellfun** returns the outputs ofin cell arrays.The outputs ofthe functioncan have any sizes and different data types.



**Montage** function display multiple image frames as rectangular montage. We have done this in order to understand better what we are the images combined together and this function will serve as a reference for us to work with the third function.

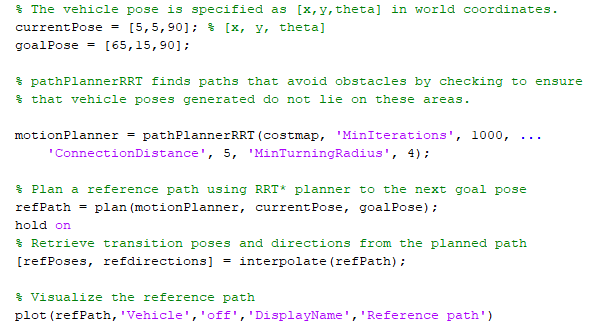
The third function is **combineMapLayers** which is used to combine map layers into a single **vehicleCostmap.**



The variable **res** represents the **CellSize** of the **costmap** object which means the side length of each square cell(in our case is 0.5 meters). The output is the **costmap** object, the most important structure of our implementation that represents the vehicle environment as a 2-D occupancy grid. Each grid in the cell has values between 0 and 1, representing the cost of navigating through the cell. Obstacles have a higher cost, while free space has a lower cost. A cell is considered an obstacle if its cost is higher than the **OccupiedThreshold** property, and free if its cost is lower than the **FreeThreshold** property.



Let’s continue with the rest of the code:



After specifying the **currentPose** and **goalPose** in order to see the reference path and the smoothed path we have used the **pathPlannerRRT** function explained above to configure a path planner using an optimal rapidly exploring random tree (RRT\*) approach. The RRT family of planning algorithms find a path by constructing a tree of connected, collision-free vehicle poses. In our case poses are connected using **Dubins** method because we use just forward movements. Maximum distance between two connected poses is 5 meters and minimum turning radius of the vehicle is 4.

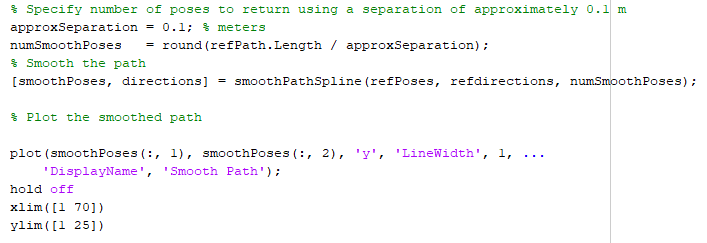
Then, using plan function we return the **refPath** object which has the fields:

* **StartPose**
  + **GoalPose**
  + **PathSegments** as a **DubinsPathSegment** structure.
  + **Length** of the reference path in meters.



**[refPoses, refdirections] = interpolate(refPath)** returns the transition poses and directions from the planned path. Then, we visualize the planned path on the figure.

Let’s see the rest of the code:



**approxSeparation** returns poses spaced about 0.1 meters apart, along the entire length of the path. Then, based on this we find the **numSmoothPoses**. To increase the granularity of the returned poses we need to increase the number of smooth poses. **\*This is a main factor that specify the difference between a reference path and a smoother path.**

After this, we use the **smoothPathSpline** function in order to return the **smoothPoses** as a matrix(in our case with dimesions 673x3) and also the **directions** which together with **velocity** are very important things to park a car. For simplicity, **I have not taken in consideration** these factors.

At the end, we plot the smoothed path putting in the x-axis the x-coordinates of **smoothPoses** and in the y-axis the y-coordinates of **smoothPoses.**



**Parking Maneuver**

At the second part of the project, I have worked with a maneuver of parking in order to illustrate the final part of the automatic parking in Matlab. Again, I am not taking in consideration the velocity of the car. The environment is the same, so the **costmap** object does not change.

In this example I am giving the values of current pose, parked pose and reference path as three **.mat** files:

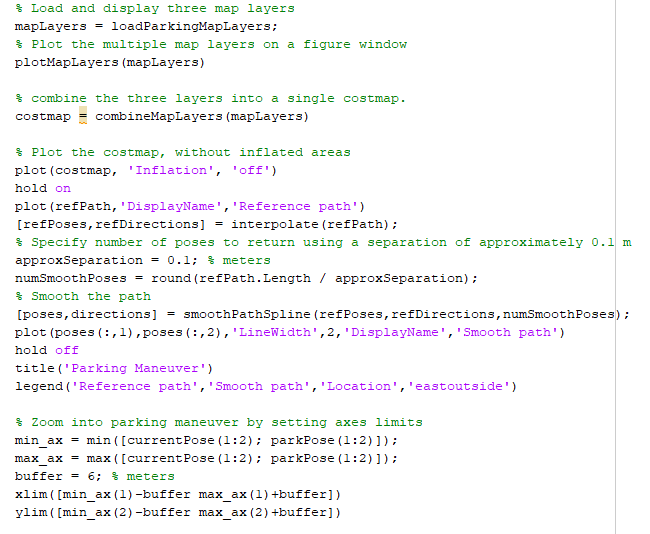
* **currentPose.mat**
* **parkPose.mat**
* **refPath.mat**

The supporting functions are again the same:

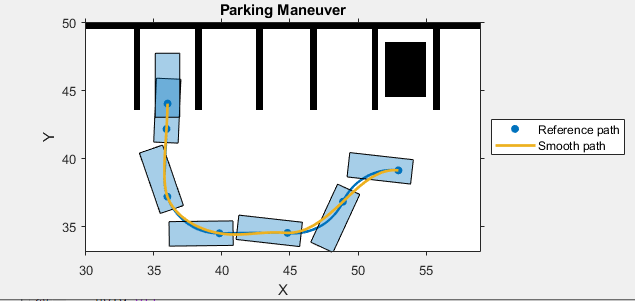
* **combineMapLayers.m**
* **plotMapLayers.m**

The main file is called **plotParkingManeuver.m** and represent a maneuver of a car in order to make the final step of the automated parking. Let’s see the implementation of this exercise in Matlab.





So, the logic of the implementation is the same as in the first part of the project. The difference is that we are using a more specific **Θ(theta)** orientation angle in order to have a more clearly visualization of the view.





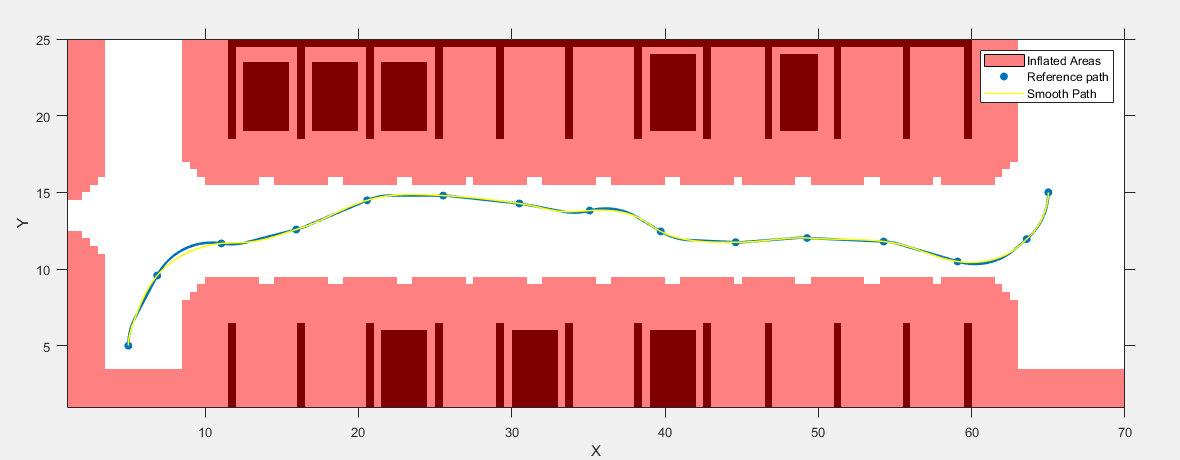
1. **Conclusions**

**This example showed how to:**

* Plan a feasible path in a semi-structured environment, such as a parking lot, using an RRT path planning algorithm.
* Smooth the path using splines and show a parking maneuver.
* Realize different parking behaviours by using different motion planner settings.

The differences between **reference path** and **smoother path** are:

* The reference path includes abrupt changes in curvature that are not suitable for driving with passengers while the smooth path is less sharp and available for a car to drive with passengers because the car can take more comfortable the curves.





* **refPoses** is a matrix with dimensions **(33x3 double)** while **smoothPoses** is a matrix with dimensions **(678x3 double),** this means that the coordinates of the smooth path are more specifically specified.
* The same thing for the **refdirections(30x1 double)** and **smoothdirections(685x1).**
* So, from these conclusions we understand the importance of **smoothPathSpline** function, its properties and the way how it works.

At the end, we have understood the difference between **C^1-continuous path** and **C^2-continuous path** and the reason why is impossible to drive with passengers in a **C^1-continuous path.** A **C^1-continuous path** sometimes have discontinuities in curvature at the points where the segments join. These discontinuities result in changes in direction that are not smooth enough for driving with passengers.

1. **References**

* <https://it.mathworks.com/help/driving/ref/smoothpathspline.html>
* <https://it.mathworks.com/help/driving/ref/pathplannerrrt.html>
* <https://en.wikipedia.org/wiki/Smoothness>
* <https://proofwiki.org/wiki/Differentiability_Class/Examples/Class_Zero_Function>
* <https://github.com>