Robot Path Smoothing through Kalman filtering and Bézier Curves

Rémy Hidra - 119034990010
Project Information Fusion - 2020

***Abstract*— Path planning algorithms are a critical part of every robotic system. The main difficulty is in producing a usable path for the robot. Many optimal and efficient path planning algorithms exist, but they may not produce an easy to follow path for a robot with constrained movement. UAVs have strong constraints related to velocity, wind, and altitude. Autonomous cars present difficulties to make sharp turns.**

To reduce the sharpness of a path, smoothing algorithms are used. In this paper, we introduce a Kalman filter based smoothing algorithm. Very few work exist in this domain. By comparing it to Bézier curves, a state-of-the-art smoothing technique, we show some strengths of our algorithm. Finally, we demonstrate some typical use cases for this algorithm in robotic.

I. Introduction

In robotics, some strong constraints exist to reduce the movement of the robot. For instance, *Unmanned Aerial Vehicles* (UAVs) might seem capable of moving in every direction. However, the direction of their velocity is constrained by their instant acceleration. A classic UAV needs to decelerate to turn correctly. Similarly, autonomous cars need to adapt their acceleration if a sharp turn is ahead. If the angle is too sharp, it is impossible for the car to perfectly follow the path during the turn, hence making a collision with another car or an obstacle possible.

This is critical issue in path planning algorithms. One could solve it by adapting the acceleration and velocity to the characteristics of the future of the path. For example, *kinodynamics* [1] algorithms implement classic path planning algorithms in more than three dimensions, by incorporating dynamic constraints, like acceleration and turning angle.

Such solutions are specific to a robot, hence, can be hard to implement and use for simpler use cases. General solutions exist in order to make a path easily followed. A typical solution can be found in smoothing algorithms. A classic approach is, first, to generate a path, using a path finding algorithm [2], like *A** [3], *Rapid exploring Random Tree* (RRT [4] and RRT* [5]), *Artificial Potential Field* (AFP) [6] or more exotic AI methods [7]. Second, path processing techniques can be applied to clean the path. What to use, depends on the path finding algorithm and its implementation. A* may produce an optimal path with a lot of nodes, which can take a lot of memory. RRT* may create fewer node, but some edges may be a lot longer than others. In all of those algorithms, the turning angle is not considered, so $90°$ and $> 90°$ turn angles can be frequent.

Common path cleaning algorithm can be *path pruning* [8], where redundant nodes, i.e. which are not needed to make the path shorter, are removed. Another common technique is *path smoothing* [9]. When encountering a strong turn angle, the path curve will be modified to add more nodes around this critical point, hence, making the turn progressive along the path. Some important points have to be considered when designing such algorithm. The new path should obviously not go through an obstacle of the scene. It should also not be significantly longer than the original path, with respect to the movement constraints of the robot.

In this work, we introduce *Kalman Filter Smoothing* (KFS), a new path smoothing algorithm, based on Kalman filter equations. Very few prior work exist to implement the Kalman filter in the path planning field of robotic. However, some strong advantages can be found for this algorithm compared to current state-of-the-art algorithm. Those use cases are discussed in section IV.

Many common smoothing algorithms exist [9]. Clothoids are curves with a linear curvature [10] and are widely used in urban planning and road management, but can be hard and expensive to compute, due to numerical integration. Spline fitting interpolate the coefficients of a basis of smooth functions [11]. Bézier curves are a special case of the previous ones [12]. They are parametric polynomial curves, which rely on a specific set of control points. By using the nodes of the computed path as control point, one can easily produce a smooth Bézier curve. They are easy to compute and implement, and have plenty of uses in robotics [8], computer graphics [13], video games and mechanical design.

Our Kalman Filter Smoothing algorithm implements the linear Kalman filter equations for path smoothing. Indeed, several applications of Kalman filtering are in signal smoothing [14]. A path can be seen as a two dimensional signal. In order to evaluate our KFS algorithm, we decided to compare it to a Bézier curve based smoothing algorithm.

In the following section II, we introduce some classic smoothing algorithms based on Bézier curves and the KFS. In section III, we present encouraging results comparing the KFS to state-of-the-art algorithms. Finally, in section IV, we discuss some use cases and limitations of the KFS.

II. Smoothing Algorithms

The objective of a smoothing algorithm is to create a path sufficiently close to the original, while keeping its curvature low at every point, with respect to movement constraints of the robot. Several smoothing algorithms exist. Most of them, interpret the path as a curve [9]. In subsection II-A and II-B, we present two variants of the widely used Bézier curve. In subsection II-C, we introduce the KFS, a Kalman

based smoothing technique, which interpret a path as a two dimensional digital noisy signal.

A. Bézier Curve

At the most basic level, a Bézier curve is a N-polynomial approximation of N control points [12]. Say we have N control points $P_1, P_2, \ldots, P_N \in \mathbb{R}^2$, in a 2D path planning problem. The Bézier curve is parametrically defined by the equation 1.

$$P(t) = \sum_{i=1}^{N} B_i^N(t) P_i \text{ with } 0 \leq t \leq 1 \tag{1}$$

$B_i^n(t)$ is the Bernstein polynomial. It corresponds to the basis of the Bézier curve. It is defined by equation 2.

$$B_i^N(t) = \binom{N}{i} t^i (1-t)^{N-i} \tag{2}$$

From equation 1 and 2, we can see that $P(0) = P_1$ and $P(1) = P_N$. The first and last points P_0 and P_N are the only points by which the curve had to pass. If we want our final curve to have a length of L points, we just need to sample the curve every $t_k = \frac{k}{L}$ for $k = 0, \ldots, L$.

We can already see that L is a parameter which control the smoothness of the curve, and depends on the path complexity. If the original path has a high length and is very rough, L needs to be high too. This will also increase the computational cost of the algorithm. Finally, we can evaluate the cost of the algorithm as $O(NL)$.

B. Piecewise Bézier Curve

To avoid the dependency of L to the original path length and complexity, we can use a different version of the algorithm. In a *Piecewise Bézier Curve* smoothing algorithm [12], we can approximate the original path as multiple small Bézier curves. Indeed, to obtain an optimal curvature, only four control points are needed [15]. Thus, we set $N = 4$, with the control points P_1, P_2, P_3 and P_4. Each piece of smooth curve is linked together. However, the angle between two curves can still be too sharp. Some extended computation can be done to solve this problem [12], but this solution will not be studied here.

This algorithm is much more efficient than the previous one. It does not rely on the expensive computation of binomial coefficients and sum of exponential. The worst case complexity is still $O(NL)$ with N the total number of points in the original path. However, we expect much better performances with this algorithm.

C. Kalman Filter Smoothing

In usual signal processing, Kalman filter is used to merge different sort of information data. In some fields, it can also be used as a noise reduction technique [14]. Thus, we can see the original path produced by the path finding algorithm as a noisy version of the expected smooth path, usable by the robot. If we set up a mathematical model for the next expected point in the smooth path, we can create a Kalman filter to approach an ideal smooth path.

We have plenty of freedom in setting the mathematical model. For an autonomous car, one could expected the next point to be at a distance defined by the car's velocity, at a bounded angle. It would be easy to model constraints on the car wheels turn by defining boundaries for the angle. By using a Kalman filter, it is very easy to constrain the path smoothing model.

Because we want to keep a general model, we decided to apply very few constraints. Thus, we model a random walk path model. Our model is defined by the following equations.

$$\begin{cases} x(t+1) = x(t) + v(t) \\ y(t) = x(t) + e(t) \end{cases} \tag{3}$$

$y(t)$ and $x(t)$ are, respectively, the point of the original path and the point of the smoothed path at the time step t. $v(t)$ and $e(t)$ are noise random variable. It possible to use a variable noise intensity to control dynamically the smooth intensity of the filter. To keep things simple, we set $v(t) \sim e(t) \sim N(0,1)$. Thus, we can implement the filter by using classic Kalman filter equations (eq.4, 5).

$$\begin{cases} \hat{x}(t+1|t) = F\hat{x}(t|t) \\ P(t+1|t) = FP(t|t)F^T + Q \end{cases} \tag{4}$$

$$\begin{cases} K(t) = & \dfrac{P(t|t-1)H^T}{HP(t|t-1)H^T + R} \\ \hat{x}(t|t) = & \hat{x}(t|t-1) + K(t)\left[y(t) - H\hat{x}(t|t-1)\right] \\ P(t|t) = & P(t|t-1) \\ & -K(t)\left[HP(t|t-1)H^T + R\right]K^T(t) \end{cases} \tag{5}$$

According to the model defined in equation 3, for 2D path smoothing problem, we have $F = H = Q = R = I_2$.

This algorithm is very simple to implement. Even by adding constraint, the worst case complexity is $O(N)$, with N the number of points in the original path. A strong advantage of this technique is that the smoothed path can be computed at the same time as the path finding algorithm, in real-time, with an $O(1)$ added complexity. Thus, we expect the KFS implementation to be fast.

III. Results

A. Experimental setup

Before implementing any path smoothing algorithm, we need to generate a path. The main characteristics of a path are its physical length, its number of nodes and its sharpness. We can generate a path, and then use post processing algorithms to change its characteristics. For instance, to reduce its number of nodes, we can apply a path pruning algorithm [8], which remove unnecessary nodes. Moreover, some path are more adapted to some robots than other. A UAV in a cluttered environment will use a very precise 3D path, while an autonomous car can use a 2D path with less precision. Similarly, those two examples usually have a start and end position spread out in the world, but an industrial robot might rotate a lot around the same points.

Because of the wide range of path structure usable, we implemented two types of path generator. The first one is a random walk path. It relies on random choice in a fixed array of possible movements. It can model an industrial robot, making complex movements in an environment. Random walk creates very complex curves, with sharp turns everywhere. Hence, it is very hard to smooth easily.

The second path generator uses the RRT* path finding algorithm [4] [5]. We first generate an environment made of a starting point, a goal, and multiple randomly positioned obstacles. Then, we apply the RRT* to find a path between the two points. It should be noted that none of the algorithms defined in section II take in consideration the obstacles in the environment. Indeed, the new smooth path, because it is different from the original one, could be overlapping with an obstacle. An easy way over that issue, is to over estimate the size of each obstacle. By setting an obstacle bigger than it is, we give the path smoothing algorithm an additional margin to not bother with obstacle detection. If this assumption is not enough for the use case, one could easily add an obstacle detection step in the Kalman filter for the KFS. Finally, we choose the RRT* path finding algorithm because of its ease of use and implementation. It is also widely used in UAV path planning [8], which is the author domain. However, A* or Artificial Potential Fields path finding algorithms could also have been used to produce similar path. After computing the optimal path through RRT*, an over sampling algorithm is applied, to add more nodes in the curve. It is expected to have around the same distance between each node before any smoothing is applied.

To summarize, the random walk generator creates very complex paths, which can be very difficult to smooth if taken as a whole. The RRT* path generator simulates a maze and solve the path finding problem. It can model UAVs and autonomous cars behavior.

As showed in the following sections III-B, III-C and III-D, we studied three different parameters to describe the quality of the smoothing for each of the two types of generated path.

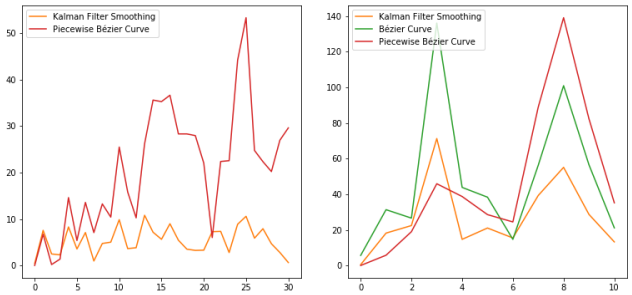
B. Path similarity

By running the smoothing algorithms on the paths we have, we obtain the results shown figure 1.

We did not ran the classic Bézier curve algorithm on the random walk path. Indeed, due to the complexity of this curve, the Bézier curve performs very poorly. As we can see, the rest of the techniques are efficient enough to smooth the raw path curve. In the RRT* path, the KFS, especially, removes nodes that create a wide turn angle.

It is also the only smoothing algorithm which does not overlap with any obstacles. Of course though, this feature is happening randomly here. To evaluate correctly the quality of the smoothed path, with respect to not overlapping with any obstacle, with decided to implement the following test.

The probability of overlapping with an obstacle is intuitively higher if the distance point-to-point between the original and new path is higher. Thus, we evaluate the distance point-to-point between each path in figure 2.



(a) Random walk path generator **(b)** RRT* path generator

Fig. 2: Distance between the original and smoothed path for the example in figure 1

In the example of the path generated in figure 1, we can see in figure 2 that the KFS stays always close to the original path. As we could expect, the Bézier curve is not efficient with this metric. The Piecewise Bézier curve behave sometimes like the Bézier curve, except in intersection of the small curves, as explained in section II-B, where the smooth curve point is equal to the one of the original path.

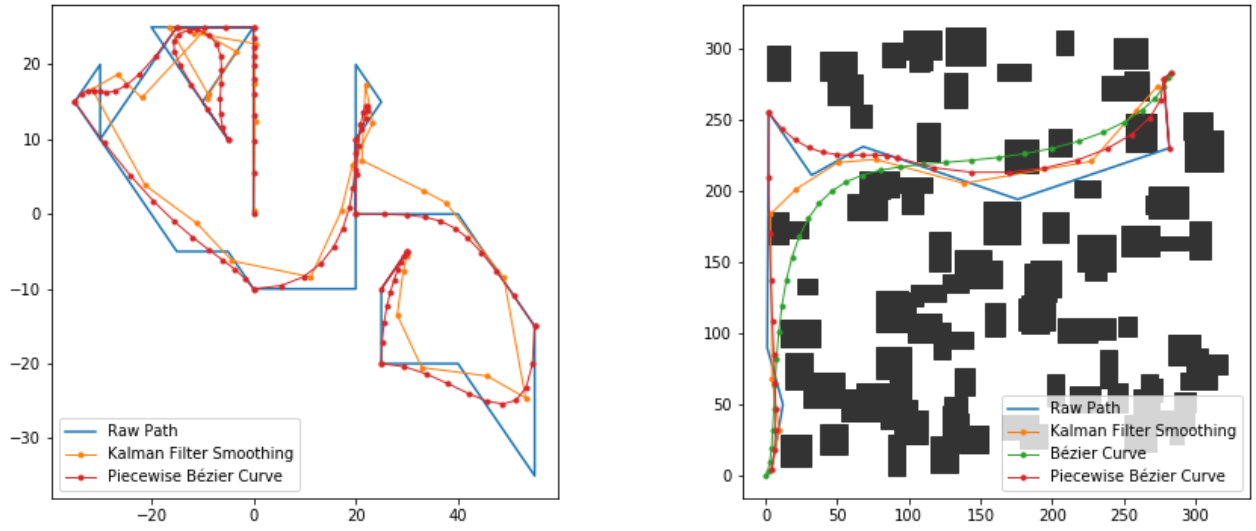
In most of the test we have done, the length of the path or number of obstacles does not change the observation we can make about figure 2.

C. Turn angle

Another interesting element to compare is the turn angle. As explained in section I, the main motivation for building a smoothing algorithm is to constrain the path to the robot mechanical constraints. A constraint that appear with most robot is the turn angle. Indeed, an autonomous car, for example, is unable to rotate its wheels around $360°$. Thus, it is necessary to make sure the path used by the robotic system does not include turn angles too sharp to be physically executed. To measure this metric, we just measure the difference between the absolute angle of the previous node and the absolute angle of the current node. Ideally, in a continuous path, every turn should be progressive, and the turn angle over time should be close to zero along the path.

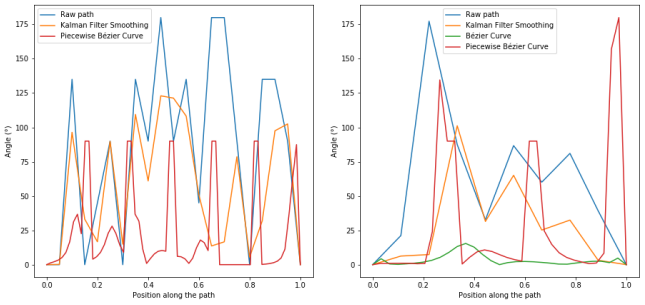
In figure 3, we display the result of the turn angles along a path. The path is a representative randomly generated path, using the random walk generator and the RRT* path generator, as explained in section III-A.

The figures 3a and 3b show the turn angle along the path. The graph of the random walk path (Fig.3a) seems hard to read. But it is actually coherent. The random walk model can only choose the next node at angle of $0°$, $90°$ or $180°$, which creates high varying turn angles. Smoothing algorithms work as expected. The piecewise Bézier curve is very low overall, except, as explained in section III-B, at the intersection points between two smoothed curves. We can also notice that the turn angle of the KFS is almost always lower than the one of the raw path. Sometimes the KFS turn angle reach the level of the piecewise Bézier curve, but it is rare and not significant.



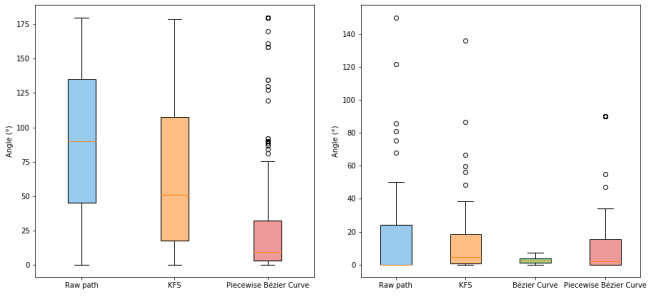
(a) Random walk path generator **(b)** RRT* path generator

Fig. 1: Comparison of path smoothing using different path generator



(a) Random walk path generator **(b)** RRT* path generator

Fig. 3: Turn angle along the path between different smoothing algorithms



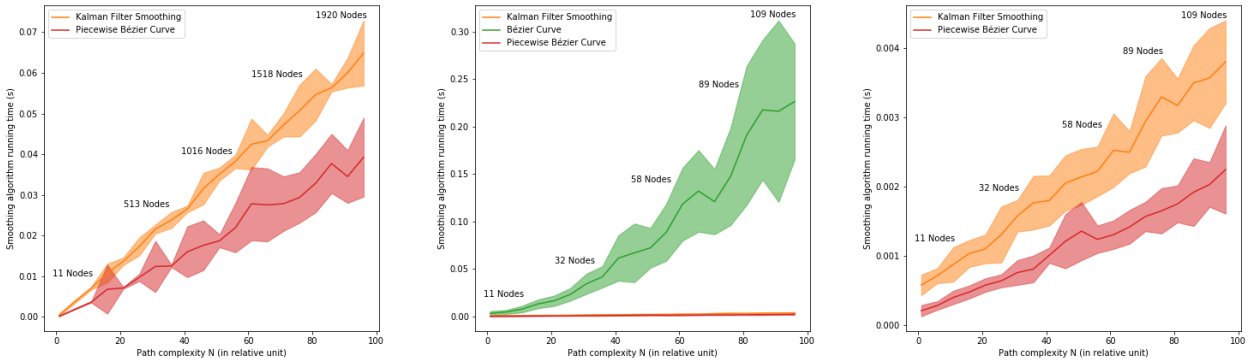
(a) Random walk path generator **(b)** RRT* path generator

Fig. 4: Turn angle distribution between different smoothing algorithms (raw path length: 5799 nodes for the random walk; 3351 nodes for RRT*)

An easier way to read such a graph, is to synthesize this distribution of data in a boxplot. In the figures 4a and 4b, we see the boxplot of the turn angle distribution for the two path generator. To have a relevant plot, the path lengths are much longer than in the previous examples.

With the random walk path (Fig.4a), we can conclude the same observations as earlier. The original path has turn angles randomly spread out between $0°$ and $180°$, with a mean at $90°$. The piecewise Bézier curve creates a path with a much lower median turn angle, at $9.4°$. However, it contains a lot of outsider points, at the extremities of the smooth curves. Those outliers degrade a lot the quality of the curve, with turn angles as high as $180°$, which makes the smoothed curve useless at that point. The KFS path looks similar to the original path. The mean turn angle is lower, at $65°$ against $92°$. The extremities of the boxplots are identical, but the overall shape of the box is skewed toward $0°$. This makes the KFS curve slightly smoother than the original path.

With the RRT* path (Fig.4b), the mean turn angle for each path is centered around $0°$. This is coherent with the fact that this path is similar to a physical path in a maze. In other words, there is a bigger proportion of straight lines (i.e. of $0°$ turn angles) than in the random walk model path. This create some strong outliers toward $180°$ in the raw path. The KFS is very similar to the orignal path, with a graph skewed toward $0°$. However, the median turn angle of the KFS is at $4.4°$ instead of $0°$ for the raw path. This shows how the KFS might correct sharp turns by turning progressively, thus lowering the proportion of straight lines.



(a) Random walk path generator **(b)** RRT* path generator **(c)** RRT* (without Bézier curve)

Fig. 5: Time performance comparison between smoothing algorithms

The same phenomenon is visible with the Bézier curve and the piecewise Bézier curve. The latter has overall the same properties as explained before, with some strong outliers and a low curvature. The natural Bézier curve provides the best result here, with a turn angle systematically under $7.1°$. This was expected with the construction and characteristics of the curve.

D. Performances

The last feature on which we decided to evaluate our KFS algorithm is the computational performance. Indeed, in robotics, most systems are separated from the main computational system, like the ground station for a UAV. Thus, most robots need to carry a small computer, which is mainly used for on-board computation, control computation, telecommunication and sensors signal processing. In some cases, path planning have to be done by the on-board computer. This can happen if the ground station is unreachable or a real-time response is needed. In that situation, only a small quantity of computational resources can be allocated to path planning, in order to let more critical parts of the system work. Therefore, computational performance is a critical factor for an implementation in a fast time response system with low hardware.

The evaluation of the running time performance of each smoothing algorithm is shown figure 5. We designed a parameter N, with $0 < N < 100$, which controls the complexity of the path generated. For the random walk model, it trivially corresponds to the number of samples in the model. For the RRT*, N depends on the maze size, the number of obstacles generated and the maximum size of each obstacle. In the end, N controls the size of the raw path generated. For each N, we compared the smoothing algorithms over 20 different raw path. The graphs figure 5 shows to the mean running time and the standard deviation for the 20 iterations. To compare the results, we also displayed the original path length above the curves. Finally, figure 5c is the same as figure 5b but without the Bézier curve, to make the reading easier.

In figures 5a and 5c, we can see the similar behavior between the KFS and the piecewise Bézier curve. Both algorithm are quite similar. In figure 5a, we can see that the maximum running time, for an average original path length of ~ 2000 nodes, is 73ms for the KFS and 49ms for the piecewise Bézier curve. Both algorithms seem to have a linear time complexity with respect to N.

In figure 5b, we see the Bézier curve smoothing algorithm running time, with the RRT* paths. The obvious result is how slow that algorithm is. For an average raw path length of ~ 100 nodes, the Bézier curve algorithm may take up to 290ms, while the KFS takes only 4.4ms for the same type of path.

IV. Discussion

From the results shown in section III, we can already deduce some observations. All the algorithms presented work in order to smooth a basic curve. However, some are more efficient than others in certain specific cases.

The *natural Bézier Curve* is not very efficient in complex use cases. As stated in section II-A, the Bézier curve is derived from the clothoid curve [9]. This property makes it very efficient to smooth a path while loosely following some control points. Thus, we obtain the result seen in figure 1b, where the Bézier does not follow the original path and overlap with a lot of obstacles. But as seen in figure 3b and 4b, the turn angles encountered in this curve are negligible. This fact makes it ideal for slow turning robot, like the TurtleBot [16], which need a very low turning angle because of mechanical constraints in its wheel. The last important property of the Bézier curve smoothing algorithm is its running time performances. It performs very poorly in comparison of the two other algorithms, certainly because it needs to compute binomial coefficients, powers of floating numbers and sums to sample the Bézier curve. To summarize, the Bézier curve algorithm should not be used in a fast response, real-time, robotic system with low hardware and not a dedicated path planning computational unit. It will,

however, produce a very smooth curve, which may be useful in a wide free space with no obstacle.

The *Piecewise Bézier Curve* is a nice compromise between the KFS and the Bézier curve. Indeed, it performs well to smooth the curve without getting to far away from the original path. As seen in figure 4a and 4b, its main flaws are the intersection between the small curves, which are not smoothed. This is the cause of the outliers and the peaks we see in figure 2a. Some more complex computations [12] can be done to reduce this effect and keep a smooth curve along the whole path. Although, like the Bézier curve, the piecewise algorithm produces curves that are still far from the original path (Fig.2a), which can create collisions in a cluttered environment (Fig.1b). In regards to its running time performances, the piecewise smoothing algorithm is quite efficient. Depending on the implementation, it can be equivalent to the KFS.

The *Kalman Filter Smoothing* algorithm produced some very encouraging result. It works effectively to smooth the path. It has the advantage of being executed point-by-point, thus it can be needed in systems where no sample of the original path can be done. According to figure 4b, it will reduce the mean turning angle along a path, but it could work even better by adding more constraints to its mathematical model. In figure 2b, 1b, we can see that, because it is strongly following the original path, the KFS is less prone to overlap with obstacles, unlike the other two smoothing algorithms. One could even imagine an efficient collision avoidance system directly integrated in the Kalman filter, to remove the need for a post processing overlapping check. Finally, the performances of the KFS are similar to the ones of the piecewise Bézier curve, which makes it usable for real-time and low hardware applications. Its point-by-point processing of data implementation can also be useful in some applications. Paired with a slow but progressive path finding algorithm, the KFS can instantly process the data of the algorithm without waiting for the whole path to be computed.

V. Conclusions

In this work, we introduced the KFS, a new smoothing algorithm relying on a classic Kalman filter to generate a smooth path from the original one. It is flexible and can be customized to the needs of the application. It can perform in real-time applications, with low hardware in an on-board robot computer. With some added computations, the KFS can be aware of mechanical and physical constraints of the robot and the environment, and it can bend the path to follow those constraints as much as possible.

We compared the KFS to two variants of the Bézier curve smoothing algorithm. It performed well in all cases, but does not smooth the curve as much. Thus, some simple cases which require mainly a smooth path curve may find it easier to implement a Bézier curve.

References

[1] B. Zhou, F. Gao, L. Wang, C. Liu, and S. Shen, "Robust and efficient quadrotor trajectory generation for fast autonomous flight," *IEEE Robotics and Automation Letters*, vol. 4, no. 4, pp. 3529–3536, 2019.

[2] L. Yang, J. Qi, D. Song, J. Xiao, J. Han, and Y. Xia, "Survey of robot 3d path planning algorithms," *Journal of Control Science and Engineering*, vol. 2016, 2016.

[3] X. Cui and H. Shi, "A*-based pathfinding in modern computer games," *International Journal of Computer Science and Network Security*, vol. 11, no. 1, pp. 125–130, 2011.

[4] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," 1998.

[5] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The international journal of robotics research*, vol. 30, no. 7, pp. 846–894, 2011.

[6] A. H. Qureshi and Y. Ayaz, "Potential functions based sampling heuristic for optimal path planning," *Autonomous Robots*, vol. 40, no. 6, pp. 1079–1093, 2016.

[7] Y.-L. Kuo, A. Barbu, and B. Katz, "Deep sequential models for sampling-based planning," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 6490–6497, IEEE, 2018.

[8] K. Yang and S. Sukkarieh, "3d smooth path planning for a uav in cluttered natural environments," in *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 794–800, IEEE, 2008.

[9] A. Ravankar, A. A. Ravankar, Y. Kobayashi, Y. Hoshino, and C.-C. Peng, "Path smoothing techniques in robot navigation: State-of-the-art, current and future challenges," *Sensors*, vol. 18, no. 9, p. 3170, 2018.

[10] M. Brezak and I. Petrovic, "Path smoothing using clothoids for differential drive mobile robots,"

[11] J. Pan, L. Zhang, and D. Manocha, "Collision-free and smooth trajectory computation in cluttered environments," *The International Journal of Robotics Research*, vol. 31, no. 10, pp. 1155–1175, 2012.

[12] F. Zhou, B. Song, and G. Tian, "Bézier curve based smooth path planning for mobile robot," *Journal of Information &Computational Science*, vol. 8, no. 12, pp. 2441–2450, 2011.

[13] Y. S. Chua, "Bezier brushstrokes," *Computer-Aided Design*, vol. 22, no. 9, pp. 550–555, 1990.

[14] M. Briers, A. Doucet, and S. Maskell, "Smoothing algorithms for state–space models," *Annals of the Institute of Statistical Mathematics*, vol. 62, no. 1, p. 61, 2010.

[15] M. Lepetič, G. Klančar, I. Škrjanc, D. Matko, and B. Potočnik, "Time optimal path planning considering acceleration limits," *Robotics and Autonomous Systems*, vol. 45, no. 3-4, pp. 199–210, 2003.

[16] M. Wise and T. Foote, "Turtlebot." https://wiki.ros.org/action/show/Robots/TurtleBot, 2020. [Online; accessed 22-Mai-2020].

VI. Acknowledgements

This work was made possible by the Information Fusion course of Shanghai Jiao Tong University, provided by Prof. Anders Lindquist.