A Step-by-Step Guide to 2D Motion Profiling Using Bézier Curves

Mechanics Division, Autonomous Systems Research

June 20, 2025

Abstract

This paper walks through the construction and explanation of a real-time motion profiling system for differential drive robots. It introduces foundational concepts in calculus, curve modeling, numerical integration, and motion planning. All concepts are broken down for high-school level understanding, with annotated code and diagrams to guide the reader through the complete system.

Contents

1 What is Motion Profiling?

Motion profiling answers the question: how should a robot move from point A to point B smoothly and safely?

- How fast should it go?
- When should it turn?
- When should it start slowing down?

The result is a path plus a time-parameterized velocity and angular velocity profile that respects the robot's physical limits.

2 Our Goal: Move Along a Curve

We want to follow a curved path while:

- Respecting maximum speed and acceleration
- Slowing down for turns (based on curvature)
- Hitting specified speeds at key positions (keyframes)

To do that, we describe the path mathematically and then compute how the robot should move along it in real time.

3 Curve Representation Options

There are multiple ways to describe a smooth curve:

- **Quadratic Bézier Curve:** Uses 3 control points (a start point, end point, and one middle control point) to form a smooth curve. It is simple but less flexible in shaping the path.
- **Cubic Bézier Curve:** Uses 4 control points (start, end, and two control points). This gives more control over the curve's shape. We use cubic Béziers here for a good balance of simplicity and flexibility.
- **Cubic Hermite Splines:** Defined by endpoints and specified tangents (slopes) at those endpoints. This means you explicitly set the direction of travel at each end, resulting in a smooth curve that honors those directions.
- **Catmull–Rom Splines:** A type of spline that passes through a series of given waypoints. The curve is calculated to smoothly interpolate through all the points, making it easy to create paths through specific locations.

We chose cubic Bezier for its balance of simplicity and control. The following are the functions that describe the x and y coordinates of the cubic Bezier curve.

$$x(t) = (1-t)^3 x_0 + 3(1-t)^2 t\, x_1 + 3(1-t)\, t^2 x_2 + t^3 x_3$$
$$y(t) = (1-t)^3 y_0 + 3(1-t)^2 t\, y_1 + 3(1-t)\, t^2 y_2 + t^3 y_3$$

4 What are Keyframes?

Keyframes are specific points along the path where we fix the robot's speed. For example:

- Start at 0 m/s
- Reach 1 m/s halfway
- End at 0 m/s

The planner converts those into arc-length positions and then blends between them. In practice, path planning software (such as *PATH.JERRYIO*) provides a visual interface to define keyframes. For example, PATH.JERRYIO includes a "speed graph" where you can add **speed keyframes** at various positions along the path:contentReference[oaicite:1]index=1. Each keyframe corresponds to a specific distance along the path (the x-axis on the graph) and a desired speed (the y-axis). The planner then constructs the speed profile by connecting these points smoothly, ensuring the robot slows down or speeds up at the designated locations.

5 Quick Primer: Derivatives and Integrals

Derivative = Rate of Change

If $s(t)$ is position over time, then

$$v(t) = \frac{ds}{dt},$$

the instantaneous speed.

Integral = Area Under Curve

If $v(t)$ is speed, then

$$s(t) = \int_0^t v(\tau)\, d\tau,$$

the total distance traveled.

6 Arc Length via Gaussian Quadrature

For a parametric curve $\mathbf{r}(t) = (x(t), y(t))$, the arc length from the start up to parameter t is

$$s(t) = \int_0^t \|\mathbf{r}'(\tau)\|\, d\tau,$$

the integral of speed $\|\mathbf{r}'(\tau)\|$ along the curve. We approximate this integral with Gaussian quadrature:

$$\int_a^b f(x)\, dx \approx \frac{b-a}{2} \sum_{i=1}^{n} w_i\, f\Big(\tfrac{b-a}{2} x_i + \tfrac{a+b}{2}\Big).$$

Numerical integration rationale. For most curves, there isn't a simple formula for $s(t)$, so we need to integrate numerically. Gaussian quadrature is one effective method for numerical integration. It chooses special sample points (nodes x_i) and weights w_i to maximize accuracy. An n-point Gauss–Legendre rule is exact for all polynomials up to degree $2n-1$, meaning it integrates any such polynomial perfectly on $[-1,1]$ with only n evaluations of f.

To illustrate, the 2-point Gaussian rule on $[-1,1]$ picks

$$x_1 = -\frac{1}{\sqrt{3}}, \quad x_2 = \frac{1}{\sqrt{3}}, \qquad w_1 = w_2 = 1,$$

so that

$$\int_{-1}^{1} f(x)\,dx \approx f\Big(-\tfrac{1}{\sqrt{3}}\Big) + f\Big(\tfrac{1}{\sqrt{3}}\Big),$$

which integrates any cubic polynomial exactly.

Table of Nodes and Weights. The following table lists the Gauss–Legendre nodes x_i and weights w_i on the standard interval $[-1,1]$ for $n = 2, 3, 4$. These are the constants you plug into the formula above.

n	x_i (nodes)	w_i (weights)
2	$-\frac{1}{\sqrt{3}} \approx -0.577350$	1
	$+\frac{1}{\sqrt{3}} \approx 0.577350$	1
3	$-\sqrt{\frac{3}{5}} \approx -0.774597$	$\frac{5}{9} \approx 0.555556$
	0	$\frac{8}{9} \approx 0.888889$
	$+\sqrt{\frac{3}{5}} \approx 0.774597$	$\frac{5}{9} \approx 0.555556$
4	-0.861136	0.347855
	-0.339981	0.652145
	$+0.339981$	0.652145
	$+0.861136$	0.347855

Application to Arc Length. In our arc-length integral, set

$$f(\tau) = \|\mathbf{r}'(\tau)\|, \quad a = 0, \quad b = t,$$

and use the above x_i, w_i with

$$\tau_i = \frac{b-a}{2}\,x_i + \frac{a+b}{2} = \frac{t}{2}\,x_i + \frac{t}{2},$$

to compute

$$s(t) \approx \frac{t}{2} \sum_{i=1}^{n} w_i\,\|\mathbf{r}'(\tau_i)\|.$$

With just $n = 4$ or 5, this gives a highly accurate approximation of $s(t)$ without needing extremely fine subdivisions.

7 Finding the Next Parameter: Newton–Raphson

To move a small distance Δs along the path, we need to find the new parameter t_{next} such that

$$s(t_{\text{next}}) = s_{\text{current}} + \Delta s.$$

We use Newton–Raphson (a root-finding method) to solve this equation:

$$t_{k+1} = t_k - \frac{s(t_k) - \big(s_{\text{current}} + \Delta s\big)}{v(t_k)}.$$

Why Newton's method? We want to solve

$$F(t) = s(t) - \big(s_{\text{current}} + \Delta s\big) = 0$$

for t. Newton–Raphson is an efficient iterative technique that uses the derivative $F'(t) = s'(t) = v(t)$. Starting from an initial guess t_k, we linearize and obtain

$$t_{k+1} \;=\; t_k - \frac{F(t_k)}{F'(t_k)} \;=\; t_k - \frac{s(t_k) - \big(s_{\text{current}} + \Delta s\big)}{v(t_k)}.$$

Stopping criterion. We iterate $k = 0, 1, 2, \ldots$ until one of the following is met:

$$\big|\, t_{k+1} - t_k \big| < \varepsilon \quad \text{or} \quad \big|s(t_k) - \big(s_{\text{current}} + \Delta s\big)\big| < \varepsilon \quad \text{or} \quad k \geq k_{\max},$$

where $\varepsilon > 0$ is a chosen tolerance (e.g. 10^{-6}) and $k_{\max}$ (e.g. 10) caps the number of iterations to avoid infinite loops. This ensures we stop once the update or the residual is sufficiently small, or when we've reached a safe iteration limit.

8 Acceleration and Deceleration Distances

From basic kinematics:

$$v^2 = v_0^2 + 2a\,\Delta s \quad \Longrightarrow \quad \Delta s = \frac{v^2 - v_0^2}{2a}\,,$$

which tells us the distance needed to change speed under constant acceleration a.

- If $a > 0$: Δs is the distance needed to **accelerate** from v_0 up to v.
- If $a < 0$: Δs is the distance needed to **decelerate** (brake) from v_0 down to v.

Why it works. This formula is derived by integrating acceleration over distance:

$$a = \frac{dv}{dt} \quad \Rightarrow \quad v\,dv = a\,ds \quad \Rightarrow \quad \int_{v_0}^{v} v\,dv = \int_0^{\Delta s} a\,ds \quad \Rightarrow \quad \frac{v^2 - v_0^2}{2} = a\,\Delta s\,,$$

so rearranging gives $\Delta s = (v^2 - v_0^2)/(2a)$.

9 Curvature and Turning Speed

For a 2D path $\mathbf{r}(t) = (x(t), y(t))$, curvature is defined as:

$$\kappa(t) = \frac{|x'(t)y''(t) - y'(t)x''(t)|}{\left(x'(t)^2 + y'(t)^2\right)^{3/2}}, \qquad R(t) = \frac{1}{\kappa(t)},$$

where $R(t)$ is the radius of curvature at that point on the path.

To ensure safe turning (limiting lateral acceleration), we restrict speed using:

$$v_{\text{curve}}(t) = v_{\max} \cdot \frac{R(t)}{R(t) + \frac{w}{2}},$$

where w is the track width (distance between the left and right wheels).

Why it works. We start with the differential drive kinematic equations:

$$V_l = v - r \cdot \omega \quad \text{and} \quad V_r = v + r \cdot \omega,$$

where v is the linear velocity, ω is the angular velocity, and r is half the track width (the distance from the center of the robot to either wheel).

We also know that $\omega = v \cdot \kappa$, by the definition of angular velocity, where κ is the curvature of the path. Substituting into the equations and factoring gives:

$$V_l = v \cdot (1 - r \cdot \kappa), \quad V_r = v \cdot (1 + r \cdot \kappa).$$

Each wheel must stay within its maximum velocity, so we create two inequalities:

$$|v \cdot (1 - r \cdot \kappa)| \leq v_{\max}, \quad |v \cdot (1 + r \cdot \kappa)| \leq v_{\max}.$$

Since $v \geq 0$, we can take v out of the absolute value and divide both sides:

$$\max\left(|1 - r \cdot \kappa|,\ |1 + r \cdot \kappa|\right) \leq \frac{v_{\max}}{v}.$$

Noting that the maximum of $|1 - r \cdot \kappa|$ and $|1 + r \cdot \kappa|$ is always $1 + |r \cdot \kappa|$, we simplify to:

$$1 + |r \cdot \kappa| \leq \frac{v_{\max}}{v} \quad \Rightarrow \quad v \leq \frac{v_{\max}}{1 + |r \cdot \kappa|}.$$

This tells us how fast the center of the robot can move without exceeding the speed limit of either wheel during a turn.

Now, since curvature is defined as $\kappa = \frac{1}{R(t)}$, where $R(t)$ is the radius of curvature of the path, we can substitute $|r \cdot \kappa| = \frac{r}{R(t)}$. Plugging this into the inequality gives:

$$v \leq \frac{v_{\max}}{1 + \frac{r}{R(t)}} = v_{\max} \cdot \frac{R(t)}{R(t) + r}.$$

This gives the final form:

$$v_{\text{curve}}(t) = v_{\max} \cdot \frac{R(t)}{R(t) + r},$$

which is a more geometric way of expressing the speed limit based on curvature and robot geometry.

10 Keyframe Velocity Interpolation

Suppose we have keyframe pairs (s_i, v_i), which define desired velocities v_i at specific arc lengths (distances) s_i along the path. To determine the target velocity at any intermediate position s (where $s_i \leq s \leq s_{i+1}$), we linearly interpolate between the surrounding keyframes:

$$\lambda = \frac{s - s_i}{s_{i+1} - s_i}, \qquad v_{\text{interp}}(s) = v_i + \lambda\big(v_{i+1} - v_i\big)\,.$$

Note: This is just a fancy way of reforming the formula for a linear equation: $f(x) = m \cdot x + b$ where $\lambda = m$, $x = (v_{i+1} - v_i)$, and $b = v_i$.

11 Velocity Planning Algorithm

Ok, now we put it all together. We now follow section 2 that describes our goals.

1. Compute $v_{\max}$: the robot's absolute maximum speed (a constant limit).
2. Compute v_{accel}: the speed limit imposed by acceleration distance (which also includes max acceleration).

   This is pretty easy for acceleration and can be done throughout the motion profile. We can just do $v_{\text{accel}} = v(t) + (a_{\max} * \Delta t)$. Deceleration for any part of the curve can be done similarly $v_{\text{decel}} = v(t) - (a_{\max} * \Delta t)$. However there is another part of deceleration, we have to make sure that the robot decelerates to the exit velocity in time (which can't simply be done with this formula). We want to first find when we should start decelerating. This can be found by rearranging the kinematic equations in section 8. Here is the equation to find the deceleration distance d_{accel} based on $a_{\max}$ and $v_{\max}$ (which are both positive).

$$d_{\text{accel}} = \frac{v_{\text{exit}}^2 - v_{\max}^2}{-2 \cdot a_{\max}}$$

   .

   We want to now find the maximum velocity v such that the robot can still decelerate (when the distance traveled along the path is greater than the d_{accel} to v_{end} using the same a_{decel} (assuming a trapezoidal profile).

   We can take the kinematic equation from section 8:

$$v_{\text{end}}^2 = v^2 + 2 \cdot a_{\text{decel}} \cdot d_{\text{remaining}}$$

   Substitute the expression for a_{decel} from above:

$$v_{\text{end}}^2 = v^2 + 2 \cdot \left(\frac{v_{\text{end}}^2 - v_{\max}^2}{2 \cdot d_{\text{decel}}}\right) \cdot d_{\text{remaining}}$$

   Simplify:

$$v^2 = v_{\text{end}}^2 - \left(\frac{d_{\text{remaining}}}{d_{\text{decel}}}\right)(v_{\text{end}}^2 - v_{\max}^2)$$

   Finally, solve for v:

$$v = \sqrt{v_{\text{end}}^2 - \left(\frac{d_{\text{remaining}}}{d_{\text{decel}}}\right)(v_{\text{end}}^2 - v_{\max}^2)}$$

3. Compute v_{curve}: the speed limit due to curvature at the robot's current position (using the turning formula).

4. Compute v_{interp}: the speed dictated by keyframe interpolation at the current position.

Then we can take the minimum of most of the constraints:

$$v_{\text{accdesired}} = \min\{\, v_{\max},\ v_{\text{accel}},\ v_{\text{curve}},\ v_{\text{interp}} \,\},$$

and combine it with the maximum compared to the max deceleration:

$$v_{\text{desired}} = \max\{\, v_{\text{accdesired}},\ v_{\text{dec}} \,\},$$

You would then combine this with the max global deceleration as explained in step 1.

12 Handling Multiple Path Segments

Many paths consist of several curve segments (splines) joined end-to-end. We can apply the motion profiling approach to each segment in sequence, while ensuring smooth transitions between segments.

At the end of one segment, the next segment begins with the robot continuing at whatever speed it reached. In practice, the end of one spline and the start of the next share a common point and velocity (if planned correctly). To handle the transition smoothly:

- If the remaining distance in the current segment is greater than the distance Δs the robot will travel in the next time step, then the robot simply continues within the same segment (it hasn't reached the end yet).
- If Δs would carry the robot past the end of the current segment, we first move it to the end of that segment (using the portion of Δs needed to cover the last bit of that spline). Then, we use the leftover distance to continue into the next segment (starting the next segment's arc length from 0 at its beginning).

This way, the robot seamlessly continues from one spline to the next. The parameter t resets to 0 at the start of the new segment, and we proceed with the same calculations on the new segment. (If using keyframes, you can also set a keyframe at the segment boundary to enforce a specific speed at that point.)

13 Simplified C++ Example from VMPLib

```
VelocityLayout TrapezoidalProfile::step() {
    if (isFinished()) {
        return { 0.0f, 0.0f, time_accum_ };
    }

    time_accum_ += dt_;
    s_current_ = sFunction(control_, prev_t_);

    float keyframe_lim  = computeKeyframeLimit();
    float curvature_lim = computeCurvatureVelocityLimit(prev_t_);
```

```
    float accel_lim     = computeAccelerationLimit(s_current_);
    float decel_lim     = computeDecelerationLimit(s_current_);

    float desired_linear = std::min({ curvature_lim,
                                      accel_lim,
                                      keyframe_lim,
                                      max_lin_vel_ });
    // float desired_linear = std::max({desired_linear_without_decel,
       decel_lim});
    float deltaS = desired_linear * dt_;
    float next_t = findNextT(s_current_, deltaS);

    float kappa = curvature(control_, next_t);
    float turning_component = kappa * desired_linear;

    Pose newPose = findXandY(control_, next_t);
    poses_.push_back(newPose);

    VelocityLayout vlay{ desired_linear, turning_component, time_accum_ };
    velocities_.push_back(vlay);

    prev_t_    = next_t;
    cur_speed_ = desired_linear;

    return vlay;
}
```

How to use these outputs. Send v_{desired} and ω to your drivetrain's velocity controller (commonly a PID or feedforward velocity controller, which is beyond the scope of this guide). A typical differential-drive conversion is:

$$v_{\text{left}} = v_{\text{desired}} - \frac{\omega\, w}{2}, \qquad v_{\text{right}} = v_{\text{desired}} + \frac{\omega\, w}{2},$$

where w is the track width. For even better tracking, you can wrap this in a RAMSETE or pure-pursuit controller that uses the robot's pose feedback to correct errors and ensure convergence to the planned path.

14 Why Acceleration/Deceleration Distance is an Approximation

The constant-a distance formulas we derived assume the robot accelerates and then decelerates in a perfect "trapezoidal" velocity profile (speeding up, cruising at $v_{\max}$, then slowing down). This assumes the robot has a long enough straight run to reach and maintain that top speed.

However, along a curvy path, we impose extra speed limits based on curvature. This often prevents the robot from ever reaching the planned $v_{\max}$ on that segment of the path. In other words, the velocity profile gets "cut off" or dips in the middle due to a turn. The robot might have to start slowing down for the turn before it ever fully accelerates to $v_{\max}$. The figure below illustrates this: in the curved case, the robot cannot maintain a flat-top speed profile because it must slow down for the turn.

Because of this effect, the simple distance formulas for accelerating and braking become approximations. In practice, the robot might start decelerating earlier (or not accelerate as much)

when a turn is coming up, meaning our calculated distances might slightly overestimate how far it needs to speed up or slow down.

To improve accuracy, we can account for curvature in our motion profile calculations. For example:

- Instead of relying solely on the constant-a formula, we could integrate the actual velocity curve (which already factors in curvature limits) to find the distance covered during acceleration and deceleration.
- We can dynamically adjust target speeds. If a sharp turn is imminent, the algorithm can cap the speed below $v_{\max}$ ahead of time, so that the robot never accelerates beyond what it can comfortably slow down from when it reaches the curve.

15 Ideas for Future Improvement

- **Motor Model**: Acceleration decreases with speed (motors produce less torque at high RPM). For example:
$$a(v) = a_{\max}\Big(1 - \frac{v}{v_{\text{free}}}\Big),$$
where v_{free} is the motor's free (no-load) speed.
- **Friction and Drag**: Include frictional forces (wheel friction, air resistance) in the model. These forces oppose motion and reduce net acceleration, especially at higher speeds.
- **Full Dynamics**: Model the robot's full dynamics (mass, rotational inertia) for more realistic acceleration behavior.
- **Numerical Refinements**: Use more advanced numerical methods (adaptive integration steps, etc.) to improve the accuracy of the motion profile calculations.

16 Further Learning Resources

- **YouTube - "The Continuity of Splines":** Deep dive into spline curves by Freya Holmér (`https://www.youtube.com/watch?v=jvPPXbo87ds`)
- **Khan Academy:** Derivatives and Integrals (introductory calculus lessons)
- **Feynman Lectures on Physics**, Vol. 1 (insightful treatment of motion in Chapter 8)
- **MIT OCW:** 18.01 Single Variable Calculus (free online course materials)

17 Conclusion

We have shown how to build a motion profiler that:

- Defines a path with cubic Béziers
- Uses calculus and numerical methods to track arc-length
- Computes acceleration, curvature, and keyframe limits

- Steps along the path with Newton–Raphson root-finding
- Outputs (v, ω) for closed-loop control (e.g. feeding into a controller like RAMSETE)

This approach blends math, programming, and robotics in an accessible, step-by-step way. By understanding the underlying methods (and their limitations), a student can extend and refine the system to handle more complex real-world conditions.

References

[1] Farin, G. (2002). *Curves and Surfaces for CAGD*. Morgan Kaufmann.

[2] Coulter, R. C. (1992). Implementation of the pure pursuit path tracking algorithm.

[3] Abramowitz, M., & Stegun, I. A. (1964). *Handbook of Mathematical Functions*.