Safe Initial Speed Optimization on a Cornering Racing Line via Backward Dynamic Programming

A physics-based vehicle simulation using backward induction

Ishak Palentino Napitupulu - 13524022
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: ishaknapitupulucxr2@gmail.com , 13524022@std.stei.itb.ac.id

***Abstract*— This paper presents a physics-based simulation for determining the maximum safe initial speed of a car on a predefined cornering racing line. The simulator represents the racing line as a discrete sequence of points, estimates local curvature, derives pointwise speed limits from tire grip, road surface grip, mass, braking capability, and downforce, and then applies backward dynamic programming to compute the highest safe initial speed. The algorithmic strategy is backward induction: each racing-line point is treated as a state, and the safe speed at the current state is computed from the physical speed limit at that point and the braking requirement imposed by the next state. The result is a full safe speed profile, where the first value of this profile represents the maximum safe initial speed. A forward simulation is then used to visualize acceleration, braking, local speed limits, and failure conditions when the user-selected speed exceeds the physical constraint.**

Keywords— dynamic programming; backward induction; racing line; vehicle dynamics; physics simulation

I. INTRODUCTION

Choosing the correct speed before entering a corner is a central problem in racing and vehicle motion planning. A racing driver wants to enter the corner as fast as possible, but the car must still be able to stay on the intended racing line. If the initial speed is too low, the car loses time. If the initial speed is too high, the required lateral acceleration may exceed the available tire grip, or the car may be unable to brake early enough before a sharper section of the path. This project models that problem as a safe-speed optimization problem.

The objective of the project is to determine the maximum safe initial speed for a car traveling along a predefined racing line. The racing line is given as input data, not generated by the algorithm. Therefore, the program does not search for the best geometric path; instead, it optimizes the speed profile along an existing path. The central question is: given a car, a track, and a sequence of racing-line points, what is the highest speed at the starting point that still allows the car to pass all following points safely?

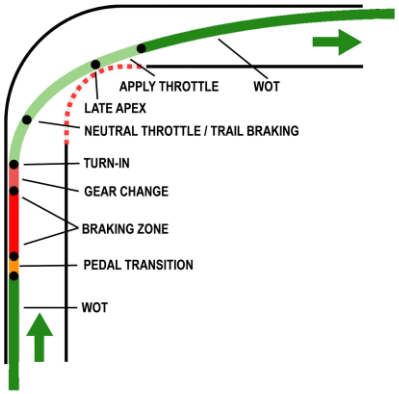

Fig. 1. Braking, cornering, and acceleration phases along a racing line.

Source: https://miro.medium.com/v2/resize:fit:1400/1*6i0KiSjkuFEAULiPbSNNvA.png

The program uses a simplified but consistent physical model. The car is described by mass, tire grip, braking deceleration, acceleration, and downforce coefficient. The track is described by road width, surface grip, centerline points, and racing-line points. These inputs are converted into segment distances, local curvature, local cornering speed limits, lateral acceleration, and speed-dependent downforce. The physics model provides the constraints, while the algorithm decides the maximum safe speed that satisfies those constraints over the whole racing line.

The main algorithm used in the implementation is dynamic programming with backward induction. This strategy is chosen because the safe speed at one point depends not only on the local curvature at that point, but also on future constraints. A point may be locally safe at a high speed, but that speed may still be unsafe if a sharp turn occurs shortly afterward and the car does not have enough distance to brake. Therefore, the computation starts from the last racing-line point and moves backward to the first point. Each state stores the maximum speed that is safe from that point until the end of the path.

II. THEORETICAL FOUNDATION

The first component of the model is the discrete racing line. The racing line is represented as an ordered sequence of two-dimensional points $p_0, p_1, \cdots, p_{n-1}$. Each point has coordinates (x_i, y_i). The distance between two consecutive points is computed using the Euclidean distance formula:

$$d_i = \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}$$

This discretization turns the continuous cornering problem into a sequence of finite states. Each point can be evaluated independently for local curvature, and each segment can be evaluated for acceleration or braking feasibility.

We obtain these two-dimensional points by usng Blender 3D Software and first creating the track centerline and racing line using Blender's curve tool, and then extracting the curves as sets of two-dimensional points using a script.

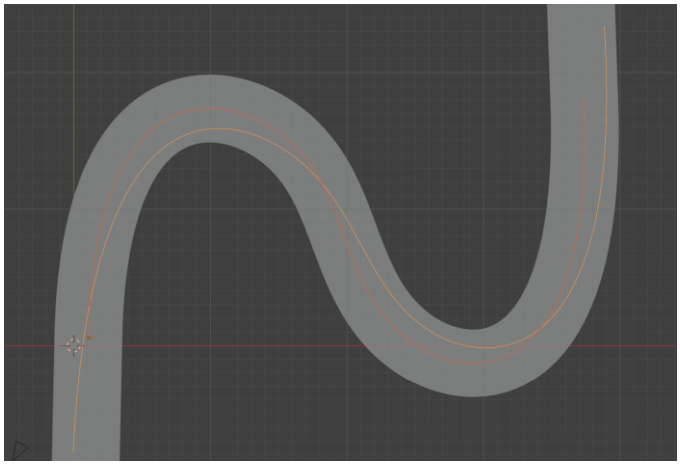

Fig. 2. Racing line (yellow) and Track line (red) made in Blender

The two-dimensional coordinates extracted from these lines are presented below.

```
0.194834 0.24374
0.387884 0.484945
0.579165 0.723648
0.768696 0.959858
0.95649 1.193604
1.142563 1.424906
.....
```

The second component is curvature estimation. Curvature measures how sharply the racing line bends. In the implementation, the curvature at an interior point is estimated from three neighboring points: previous point A, current point B, and next point C. The side lengths of the triangle are computed as $a = |BC|$, $b = |AC|$, and $c = |AB|$. The triangle area is computed from the coordinate cross product. The local curvature is then calculated as:

$$\kappa_i = \frac{4A_\triangle}{abc}$$

If the three points are nearly collinear, or if one of the side lengths is too small, the curvature is treated as zero. This prevents unstable calculations and correctly represents a straight segment as having no significant cornering demand. The first and last points are also assigned zero curvature because they do not have complete neighboring points on both sides.

The third component is lateral acceleration. A car moving with speed v_i on a path with curvature κ_i requires lateral acceleration:

$$a_{\text{lat},i} = v_i^2 \kappa_i$$

This formula shows why sharper turns require lower speed. If κ_i increases, the same speed produces greater lateral acceleration. If speed increases, the lateral acceleration increases quadratically. Therefore, safe cornering speed must be limited by the amount of grip available at the tire-road contact.

The fourth component is effective grip. The program combines tire grip and road surface grip into one value:

$$\mu_{\text{eff}} = \mu_{\text{tire}}\, \mu_{\text{surface}}$$

A car with better tires has a larger μ_{tire}, and a track with better surface condition has a larger μ_{surface}. The product represents the available grip used by the simplified model. This is why the same car can produce different results on different tracks, and why a tire upgrade can increase the computed safe speed.

The fifth component is downforce. The implementation models downforce as a speed-dependent force proportional to v^2:

$$D(v) = C_d v^2$$

Here, C_d is the car downforce coefficient after combining base downforce and spoiler contribution. Downforce increases the normal force on the tires, which increases the lateral force that can be supported. In the simplified model, the available lateral force is proportional to $\mu_{\text{eff}}(mg + C_d v^2)$.

The lateral safety condition can be expressed as:

$$mv_i^2\kappa_i \le \mu_{\text{eff}}(mg + C_d v_i^2)$$

The left side is the lateral force required to follow the curve, while the right side is the simplified available grip force. Rearranging the inequality produces the local speed limit formula used by the program:

$$v_{\text{limit},i} = \sqrt{\frac{\mu_{\text{eff}} g}{\kappa_i - \frac{\mu_{\text{eff}} C_d}{m}}}$$

This formula is only valid when the curvature is positive and the denominator is positive. If the curvature is nearly zero or the denominator is not positive, the implementation assigns a straight-line speed cap. This avoids invalid square-root operations and reflects the idea that straight segments are not constrained by cornering curvature in the same way as curved segments.

The sixth component is longitudinal braking. Local cornering safety is not enough; the car must also be able to slow down before the next point. The program uses the constant-acceleration kinematic equation:

$$v_{\text{next}}^2 = v_{\text{current}}^2 + 2ad$$

For braking, acceleration is negative. If braking deceleration is stored as a positive magnitude b, the maximum current speed that can still brake to the next safe speed over distance d_i is:

$$v_{\text{brake},i} = \sqrt{v_{\text{next}}^2 + 2bd_i}$$

This braking equation is the connection between the physical model and the dynamic programming recurrence. It propagates future speed requirements backward to earlier points.

The dynamic programming state is defined as $\text{dp}[i]$, the maximum safe speed at racing-line point i while still allowing the car to complete the path from point i to the final point. The final state is initialized as:

$$\text{dp}[n-1] = L[n-1]$$

where $L[i]$ is the local speed limit at point i. For each previous point, the recurrence is:

$$\text{dp}[i] = \min\left(L[i], \sqrt{\text{dp}[i+1]^2 + 2bd_i}\right)$$

This recurrence enforces two constraints at the same time. First, the speed cannot exceed the local cornering limit $L[i]$. Second, the speed cannot exceed the maximum value that still allows braking to the next safe speed $\text{dp}[i+1]$. Because each state uses the result of the next state, the method is dynamic programming. Because the computation starts from the final state and moves backward, the technique is backward induction.

The time complexity is $O(n)$ because every racing-line point is processed once in the backward pass. The memory complexity is also $O(n)$ because the program stores one DP value for each point. This efficiency is important because the result must be available interactively in the graphical simulation.

III. IMPLEMENTATION

The source code is organized inside the src directory. The main entry point is driver.py. The program begins by loading car data, loading track data, and opening the Pygame interface. The Asset directory contains the external data used by the simulator, including car specifications, car images, track specifications, track images, and RacingPoints.txt files.

The car model is implemented in car.py. A car stores width, mass, base tire grip, braking deceleration, acceleration, base downforce coefficient, tire option, spoiler option, and asset paths. The property tire_grip returns the sum of base tire grip and installed tire contribution. The property downforce_coeff returns the sum of base downforce and spoiler contribution. This design allows the same base car to produce multiple physical configurations.

```
class Car:
    name: str
    imagePath: str
    simImagePath: str
    SoundPath: str
    width: float
    mass: float
    base_tire_grip: float
    braking_deceleration: float
    acceleration: float
    base_downforce_coeff: float
    tire_option: str = "Main"
    spoiler_option: str = "Main"
    tire_button_position: Tuple[float,
    float] = (0.5, 0.75)
    spoiler_button_position: Tuple[float,
    float] = (0.3, 0.45)
    spoiler: Spoiler =
    field(default_factory=Spoiler)
    tires: Tires =
    field(default_factory=lambda:
```

```
    Tires(0.0))

    def tire_grip(self) -> float:
        return self.base_tire_grip +
        self.tires.tire_grip

    def downforce_coeff(self) -> float:
        return self.base_downforce_coeff +
        self.spoiler.downforce

    def install_spoiler(self, spoiler:
        Spoiler) -> None:
        spoiler.validate()
        self.spoiler = spoiler

    def install_tires(self, tires: Tires) ->
        None:
        tires.validate()
        self.tires = tires
```

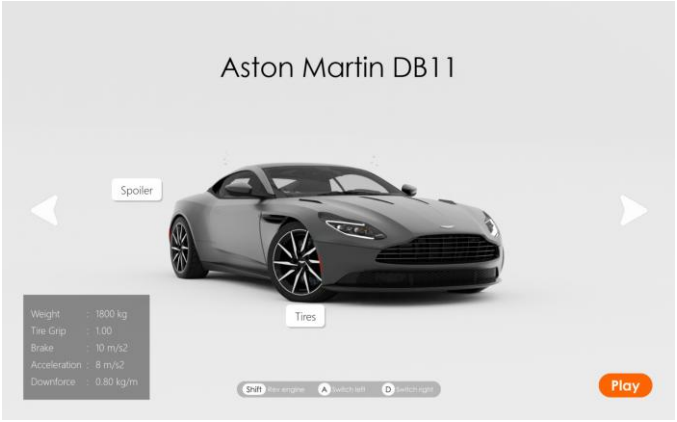

Fig. 3. Graphical user interface for vehicle selection and attributes for each car.

The car_loader.py module reads car folders from Asset/Cars. Each car specification can define tire and spoiler options. The loader creates multiple configurations by combining these options. In the current repository, these variants produce 24 car configurations. A configuration with better tires changes μ_{tire}, while a configuration with an additional spoiler changes C_d.

The track model is implemented in track.py and track_loader.py. A track contains centerline points, racing-line points, road width, name, and surface grip. The loader reads TrackSpec.txt and RacingPoints.txt from each track folder. The centerline and road width can be used for validation, while the racing-line points are the main input for the physics model.

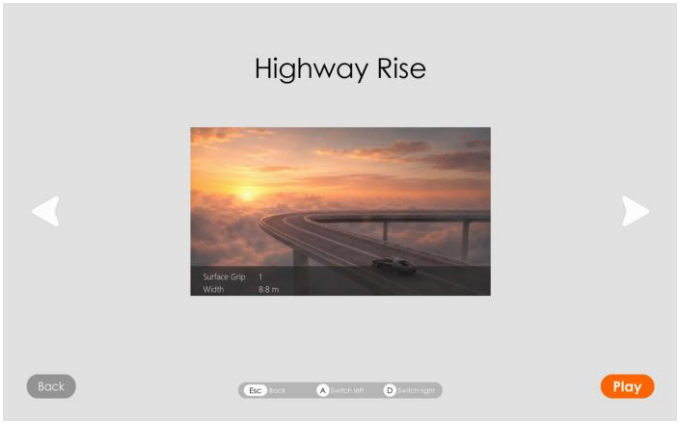

Fig. 4 Graphical user interface for track selection.

The physics_model.py module contains the core physical calculations and the DP algorithm. The compute_distances function computes d_i for each segment. The compute_curvature function computes κ_i for every racing-line point using the three-point curvature formula. The compute_speed_limits function computes $L[i]$ by combining effective grip, gravity, curvature, mass, and downforce. The implementation first calculates μ_{eff} and the denominator $\kappa_i - (\mu_{\text{eff}} C_d)/m$. The speed-limit formula is applied only to valid cornering points.

```
def calculate_curvature_from_three_points(
    A: Sequence[float],
    B: Sequence[float],
    C: Sequence[float],
) -> float:
    ax, ay = float(A[0]), float(A[1])
    bx, by = float(B[0]), float(B[1])
    cx, cy = float(C[0]), float(C[1])

    a = math.hypot(cx - bx, cy - by)
    b = math.hypot(cx - ax, cy - ay)
    c = math.hypot(bx - ax, by - ay)
    if a <= EPSILON or b <= EPSILON or c <=
    EPSILON:
        return 0.0

    area = abs((bx - ax) * (cy - ay) - (by -
    ay) * (cx - ax)) / 2.0
    if area <= EPSILON:
        return 0.0

    return 4.0 * area / (a * b * c)
```

```
def compute_speed_limits(self, car: Car,
track: Track, curvature: np.ndarray) ->
np.ndarray:
   effective_grip = car.tire_grip *
   track.surface_grip
   denominator = curvature -
   (effective_grip *
   car.downforce_coeff / car.mass)
```

```
    speed_limits =
    np.full_like(curvature,
    STRAIGHT_SPEED_LIMIT, dtype=float)
    cornering_mask = (curvature > EPSILON) &
    (denominator > EPSILON)
    speed_limits[cornering_mask] =
    np.sqrt((effective_grip * G) /
    denominator[cornering_mask])
    return speed_limits
```

The compute_backward_dp function is the core algorithmic function. It receives the car, the local speed-limit array, and the distance array. It creates a DP array with the same length as the speed-limit array. The last state is initialized with the last speed limit. Then the function iterates from the second-last index down to zero. For each index, it computes:

$$v_{\text{brake,max},i} = \sqrt{\text{dp}[i+1]^2 + 2bd_i}$$

Then it stores:

$$\text{dp}[i] = \min\left(L[i],\ v_{\text{brake,max},i}\right)$$

The value $\text{dp}[0]$ is the maximum safe initial speed. The full DP array is also useful because it represents the target safe speed profile along the entire racing line.

```
def compute_backward_dp(
        self,
        car: Car,
        speed_limits: np.ndarray,
        distances: np.ndarray,
    ) -> np.ndarray:
        if len(distances) !=
        len(speed_limits) - 1:
            raise ValueError("Jumlah
            distances harus N - 1 dari
            jumlah speed_limits.")

        dp = np.zeros_like(speed_limits,
        dtype=float)
        dp[-1] = speed_limits[-1]

        for i in range(len(speed_limits) -
        2, -1, -1):
            max_speed_before_braking =
            math.sqrt(max(0.0, dp[i + 1] **
            2 + 2.0 *
            car.braking_deceleration *
            distances[i]))
            dp[i] = min(speed_limits[i],
            max_speed_before_braking)

        return dp
```

The implementation flow can be summarized as follows. First, load the selected car and track. Second, convert the racing-line points into a numeric array. Third, compute distances between points. Fourth, estimate curvature. Fifth, compute local speed limits from the physics model. Sixth, run the backward DP recurrence. Seventh, use $\text{dp}[0]$ as the optimal initial speed and use the whole DP array as the target speed profile.

The simulation_profile.py module performs the forward simulation. After the DP profile is computed, the program starts from the user-selected initial speed. At each point, it compares the current speed with the local speed limit. If the speed exceeds the local limit beyond the tolerance, the simulation marks the car as failed. Otherwise, it determines whether the car should brake, accelerate, or coast toward the next DP target speed.

```
dp = model.compute_backward_dp(car,
speed_limits, distances)
current_speed = float(initial_speed)


for index, point in enumerate(points):
   acceleration_value = 0.0
   acceleration_state = "Coasting"
   local_speed_limit =
   float(speed_limits[index])


     if current_speed > local_speed_limit +
     SPEED_LIMIT_TOLERANCE:
         failed_index = index
         failed_message = "Mobil gagal
        mengikuti racing line."
         acceleration_state = "Failed"


    if failed_index is None and index <
    len(points) - 1:
         next_target_speed = float(dp[index
        + 1])
```

The required acceleration for a segment is calculated from the same kinematic model:

$$a = \frac{v_{\text{target}}^2 - v_{\text{current}}^2}{2d}$$

If the value is negative, it is clamped by the car braking deceleration. If the value is positive, it is clamped by the car acceleration. The next speed is then computed using:

$$v_{\text{new}} = \sqrt{v_{\text{current}}^2 + 2ad}$$

For acceleration, the implementation also prevents the new speed from exceeding the next DP target. This keeps the forward simulation aligned with the backward DP profile.

```
def _required_acceleration(current_speed,
target_speed, distance, min_acceleration,
```

```
max_acceleration):
    if distance <= EPSILON:
        return 0.0
    acceleration = (target_speed *
    target_speed - current_speed *
    current_speed) / (2.0 * distance)
    return max(min_acceleration,
    min(max_acceleration, acceleration))

def _next_speed(current_speed,
acceleration_value, distance,
next_target_speed):
    if acceleration_value < 0:
        return math.sqrt(max(0.0,
        current_speed * current_speed + 2.0
* acceleration_value * distance))
    if acceleration_value > 0:
        accelerated_speed =
        math.sqrt(current_speed *
        current_speed + 2.0 *
        acceleration_value * distance)
        return min(accelerated_speed,
        float(next_target_speed))
    return current_speed
```

Each simulation frame stores the point index, position, current speed, local speed limit, optimal DP speed, curvature, acceleration value, acceleration state, elapsed time, lateral acceleration, and downforce. The elapsed time is estimated using distance divided by average segment speed:

$$t_i = \frac{d_i}{\frac{v_i + v_{i+1}}{2}}$$

The gui.py module uses these frames to display the simulation. The user can select a car, select a track, enter an initial speed, and observe whether the car successfully follows the racing line. The interface also displays diagnostic values such as speed, limit, target speed, acceleration state, curvature, lateral acceleration, and final optimality. This makes the DP result easier to interpret visually.

Fig. 5. Simulation interface displaying the vehicle animation and real-time simulation results in Highway Rise track.

IV. Result and Discussion

The current repository contains 24 car configurations and 2 track assets. The tested tracks are Highway Rise and Cherry Blossom. Each combination of car and track is evaluated by building a simulation profile and extracting $dp[0]$, the maximum safe initial speed.

For Highway Rise, the maximum safe initial speed across all available car configurations ranges from 96.80 km/h to 120.53 km/h, with an average of 108.80 km/h. The lowest value is produced by the BMW M5 CS using the main tire and main spoiler configuration. The highest value is produced by the Porsche 911 GT3 using Slick tires and the RS spoiler.

For Cherry Blossom, the maximum safe initial speed ranges from 78.13 km/h to 94.87 km/h, with an average of 86.24 km/h. This result is lower than Highway Rise because the local constraints along the racing line are more restrictive. In the model, lower surface grip, larger curvature, shorter braking distance, or weaker braking capability all reduce the feasible safe speed.

The results are consistent with the physical model. Better tire grip increases μ_{eff} and allows the car to support larger lateral acceleration. Additional downforce increases the speed-dependent normal force and can raise the local cornering limit. Stronger braking deceleration increases $\sqrt{(dp[i+1]^2 + 2bd_i)}$, which means the car may safely enter earlier points faster because it can slow down more effectively before future constraints.

The DP result also explains why the optimal initial speed is not simply the speed limit at the first point. A starting point may have low curvature and therefore a high local speed limit, but a later point may require a much lower speed. Backward induction propagates that later requirement to earlier states. This is why the algorithm is more suitable than a purely local greedy approach.

A failure test supports the interpretation of $dp[0]$ as a safety threshold. When the best Highway Rise configuration is simulated at 20 km/h above its computed safe initial speed, the car fails at racing-line index 109. When the best Cherry Blossom configuration is simulated at 20 km/h above its computed safe initial speed, the car fails at index 19. This shows that speeds above the DP result can violate future physical constraints even if the first point itself appears safe.

The model has limitations. The racing line is predefined, so the program optimizes speed along a fixed path instead of optimizing both path and speed. The physical model is simplified and does not include tire slip angle, load transfer, steering angle limits, suspension behavior, aerodynamic drag,

road banking, tire temperature, or real telemetry validation. However, the implementation is still suitable for demonstrating how physics constraints can be transformed into a dynamic programming recurrence.

Overall, the result shows that the simulator connects physics and algorithm design in a coherent way. The physics model produces local speed limits and braking constraints. The DP algorithm combines those constraints across the entire racing line. The forward simulation then confirms whether a selected initial speed follows or violates the computed safe profile.

V. CONCLUSION

This project implements a physics-based simulation for determining the maximum safe initial speed on a cornering racing line. The model discretizes the racing line into points, computes local curvature, estimates speed limits from grip and downforce, and uses braking kinematics to connect one point to the next.

The main algorithm is backward dynamic programming. The state $dp[i]$ stores the maximum safe speed at point i considering all remaining points. The recurrence $dp[i] = \min(L[i], \sqrt{(dp[i+1]^2 + 2bd_i)})$ combines the local cornering limit and the braking feasibility constraint. Because the computation starts from the final point and moves backward, the implementation uses backward induction.

The algorithm is appropriate because the safe speed at one point depends on future physical constraints. It avoids repeated trial-and-error testing of initial speeds and computes the whole safe speed profile in linear time. The forward simulation and graphical interface then make the result observable through speed, local limit, target speed, acceleration state, and failure behavior.

The results show that different tracks and car configurations produce different maximum safe initial speeds. Better grip, stronger downforce, and stronger braking generally increase the safe speed, while sharper turns and lower surface grip decrease it. Future development could improve realism by generating the racing line automatically, adding a more detailed tire model, modeling aerodynamic drag and load transfer, and validating the simulation using real driving data.

APPENDIX

Github:https://github.com/Ishakpln/ApexDP_13524022
YouTube Video: https://youtube.com/shorts/0TjUrVcyeM8

ACKNOWLEDGEMENT

First and foremost, the author wishes to offer his sincere gratitude to God Almighty for the blessings, perseverance, and guidance granted to him during the preparation and completion of this research paper.

The author would also like to convey his heartfelt appreciation to his beloved family for their unconditional love, constant prayers, and continuous support, which have remained an important source of motivation throughout his academic journey.

Special appreciation is addressed to Dr. Ir. Rila Mandala, M.Eng., Ph.D., along with all members of the teaching staff of the IF2211 Algorithm Strategies course at Institut Teknologi Bandung, for the valuable knowledge, direction, and assistance provided throughout the learning process.

Lastly, the author extends his gratitude to his fellow students in the Informatics Engineering program for their cooperation, encouragement, and companionship during the semester. Their support and contributions have been sincerely valued.

REFERENCES

[1] D. P. Bertsekas, *Dynamic Programming and Optimal Control*, Vol. I, 4th ed. Belmont, MA, USA: Athena Scientific, 2017. [Online]. Available: https://www.mit.edu/~dimitrib/dpbook.html
Diakses pada 15 Juni 2026

[2] R. Munir, "Program Dinamis (Dynamic Programming) Bagian 1," Bahan Kuliah IF2211 Strategi Algoritma, Program Studi Teknik Informatika, STEI-ITB, 2020. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2019-2020/Program-Dinamis-2020-Bagian1.pdf
Diakses pada 15 Juni 2026

[3] M. Guest, "A Beginner's Guide to Dynamic Programming," YouTube. [Online]. Available: https://www.youtube.com/watch?v=oNoILrFOx2k
Diakses pada 13 Juni 2026

[5] R. Rajamani, *Vehicle Dynamics and Control*, 2nd ed. New York, NY, USA: Springer, 2012. [Online]. Available: https://link.springer.com/book/10.1007/978-1-4614-1433-9
Diakses pada 14 Juni 2026

[6] H. B. Pacejka and I. J. M. Besselink, *Tire and Vehicle Dynamics*, 3rd ed. Oxford, UK: Butterworth-Heinemann, 2012. [Online]. Available: https://shop.elsevier.com/books/tire-and-vehicle-dynamics/pacejka/978-0-08-097016-5
Diakses pada 14 Juni 2026

[7] E. Velenis and P. Tsiotras, “Minimum-Time Travel for a Vehicle with Acceleration Limits: Theoretical Analysis and Receding-Horizon Implementation,” *Journal of Optimization Theory and Applications*, vol. 138, no. 2, pp. 275–296, 2008. [Online]. Available: https://doi.org/10.1007/s10957-008-9381-7 Diakses pada 14 Juni 2026

[8] N. R. Kapania, J. Subosits, and J. C. Gerdes, “A Sequential Two-Step Algorithm for Fast Generation of Vehicle Racing Trajectories,” *Journal of Dynamic Systems, Measurement, and Control*, vol. 138, no. 9, Art. no. 091005, 2016. [Online]. Available: https://doi.org/10.1115/1.4033311 Diakses pada 19 Juni 2026

[9] A. Heilmeier, A. Wischnewski, L. Hermansdorfer, J. Betz, M. Lienkamp, and B. Lohmann, “Minimum Curvature Trajectory Planning and Control for an Autonomous Race Car,” *Vehicle System Dynamics*, vol. 58, no. 10, pp. 1497–1527, 2020. [Online]. Available: https://doi.org/10.1080/00423114.2019.1631455 Diakses pada 14 Juni 2026

[10] M. Massaro and D. J. N. Limebeer, “Minimum-Lap-Time Optimisation and Simulation,” *Vehicle System Dynamics*, vol. 59, no. 7, pp. 1069–1113, 2021. [Online]. Available: https://doi.org/10.1080/00423114.2021.1910718 Diakses pada 14 Juni 2026

[11] G. Perantoni and D. J. N. Limebeer, “Optimal Control for a Formula One Car with Variable Parameters,” *Vehicle System Dynamics*, vol. 52, no. 5, pp. 653–678, 2014. [Online]. Available: https://doi.org/10.1080/00423114.2014.889315 Diakses pada 15 Juni 2026

STATEMENT

In this statement, I declare that this paper I have written is my own work, not a duplication or translation of someone else’s paper, and is not plagiarized.

Bandung, 1 Juni 2026

Ishak Palentino Napitupulu 13524022