Agnirath Strategy Module Application

Hafiz Rahman Elikkottil
EE24B104

Contents

1 Sunlight, Speed, and Strategy

1.1 Question 1: Forces and Newton's Laws for the Car

The motion of the solar car is governed by the interplay between the **tractive force** generated by the motor and the **resistive forces** from the environment.

Tractive Force

$$F_{\text{tractive}} = \frac{\tau\,\eta}{r}$$

Aerodynamic Drag

$$F_{\text{aero}} = \tfrac{1}{2}\,\rho\,C_d\,A\,v^2$$

Rolling Resistance

$$F_{\text{rolling}} = C_{rr}\,mg\cos\theta$$

Gradient Resistance

$$F_{\text{gravity}} = mg\sin\theta$$

Constant Velocity ($a = 0$)

$$F_{\text{tractive}} = F_{\text{aero}} + F_{\text{rolling}} + F_{\text{gravity}}$$

Acceleration ($a > 0$)

$$F_{\text{tractive}} - (F_{\text{aero}} + F_{\text{rolling}} + F_{\text{gravity}}) = ma$$

Main Equation of Motion

$$ma = \frac{\tau\,\eta}{r} - \left(\tfrac{1}{2}\rho C_d A v^2 + C_{rr} mg\cos\theta + mg\sin\theta\right)$$

Variable Definitions

Symbol	Description
m	Total mass of the car (kg)
a	Acceleration (m/s^2)
τ	Motor torque (Nm)
η	Drivetrain efficiency
r	Wheel radius (m)
ρ	Air density (kg/m^3)
C_d	Coefficient of aerodynamic drag
A	Frontal area (m^2)
v	Velocity (m/s)
C_{rr}	Rolling resistance coefficient
g	Gravitational acceleration (m/s^2)
θ	Road incline angle (rad)

Table 1: Variables used in the force and motion equations.

1.2 Question 2: Battery Modeling Parameters

1. Visualizing Battery "Percentage": State of Charge (SOC)

The "percentage" of a battery is technically referred to as its **State of Charge (SOC)**. It is a relative measure of the available energy.

$$\text{SOC} = \frac{\text{Current Stored Charge}}{\text{Current Maximum Capacity}} \times 100\%$$

SOC is not measured directly but is an estimate computed by the Battery Management System (BMS), typically using methods like Coulomb counting in conjunction with voltage measurements.

2. Indicating Battery "Health": State of Health (SOH)

The degradation and aging of a battery are quantified by its **State of Health (SOH)**. SOH is a measure of the battery's condition relative to its original specifications, reflecting the gradual loss of capacity over its lifecycle.

$$\text{SOH} = \frac{\text{Current Maximum Capacity}}{\text{Nominal (As-New) Capacity}} \times 100\%$$

A new battery has 100% SOH, which decreases as it is used.

3. Expressing Battery Capacity

The available energy in a battery is a function of its original capacity, its current health, and its present charge level. The relationship is expressed in two steps:

Actual Capacity (C_{actual}): A battery's current maximum capacity is its original **Nominal Capacity (C_{nominal})** degraded by its SOH.

$$C_{\text{actual}} = C_{\text{nominal}} \times \text{SOH}$$

Available Energy ($E_{\text{available}}$): The energy available for use is the SOC percentage of this Actual Capacity.

$$E_{\text{available}} = C_{\text{actual}} \times \text{SOC}$$

Combining these gives the comprehensive formula:

$$E_{\text{available}} = (C_{\text{nominal}} \times \text{SOH}) \times \text{SOC}$$

1.3 Question 3: Solar Irradiance Modeling

1. Kinds of Solar Irradiation

The total solar irradiance incident on a surface is composed of three components:

- **Direct Normal Irradiance (DNI):** The collimated "sunbeam" radiation that arrives directly from the sun.
- **Diffuse Horizontal Irradiance (DHI):** Isotropic radiation scattered by the atmosphere and clouds.
- **Reflected Irradiance:** Radiation reflected from the ground, dependent on the surface albedo.

The total radiation on a horizontal surface is the **Global Horizontal Irradiance (GHI)**:

$$\text{GHI} = \text{DNI} \cdot \cos(\theta_z) + \text{DHI}$$

where θ_z is the solar zenith angle.

2. Factors Causing Change in Solar Radiation

Solar irradiance at the Earth's surface is dynamic, primarily varying with:

a. **Location (Latitude, ϕ):** Determines the sun's elevation and path, with lower latitudes receiving more direct and intense radiation annually.

b. **Time of Day (Hour Angle, ω):** Dictates the diurnal cycle, with irradiance peaking at solar noon when the atmospheric path length is minimized.

c. **Day of Year (Declination, δ):** The Earth's axial tilt causes seasonal variations in sun angle and day length, leading to higher irradiance in summer than in winter.

3. Comprehensive Formula for Solar Irradiance

The objective is to compute the **Plane of Array (POA) Irradiance (I_{POA})**, which is the total power density on the car's solar panel. This is the sum of the direct, diffuse, and reflected components incident on the tilted surface.

$$I_{\text{POA}} = I_{\text{direct}} + I_{\text{diffuse}} + I_{\text{reflected}}$$

Using the widely accepted isotropic sky model, these components are:

- **Direct:** $I_{\text{direct}} = \text{DNI} \cdot \cos(\theta)$

- **Diffuse:** $I_{\text{diffuse}} = \text{DHI} \cdot \left(\frac{1+\cos(\beta)}{2}\right)$
- **Reflected:** $I_{\text{reflected}} = \text{GHI} \cdot \rho_g \cdot \left(\frac{1-\cos(\beta)}{2}\right)$

The **angle of incidence (θ)** is the critical variable. It is a function of the sun's position and the panel's orientation, calculated as:

$$\cos(\theta) = \cos(\theta_z)\cos(\beta) + \sin(\theta_z)\sin(\beta)\cos(\psi_s - \psi_p)$$

Variable Definitions

- DNI, DHI, GHI: The three main irradiance components (W/m^2).
- θ: Angle of incidence between the sun's ray and the panel normal.
- β: Panel tilt angle from horizontal.
- ρ_g: Ground albedo (reflectivity coefficient).
- θ_z: Solar zenith angle.
- ψ_s, ψ_p: Solar azimuth and panel azimuth angles.

1.4 Question 4: Power Balance and System Constraints

The car's operation is modeled as an energy system where the rate of change of energy stored in the battery equals the net power flow.

1. Power Inputs and Outputs

Power Inputs

- **Solar Power (P_{solar}):** The primary energy source, generated by the photovoltaic array.
- **Regenerative Braking Power (P_{regen}):** Energy recaptured by the motor during deceleration.

Power Outputs

- **Propulsion Power ($P_{\text{propulsion}}$):** Mechanical power required to overcome resistive forces.
- **Auxiliary Power (P_{aux}):** Electrical load from onboard systems (telemetry, computers, etc.).
- **System Losses:** Energy lost as heat in the motor, drivetrain, and battery.

2. Mechanical Constraints

The vehicle's state is limited by several physical constraints:

- **Velocity Constraint:** $0 < v(t) \leq v_{\max}$
- **Motor Power Constraint:** $P_{\text{regen,max}} \leq P_{\text{motor}}(t) \leq P_{\text{drive,max}}$
- **Acceleration Constraint:** $a_{\min} \leq a(t) \leq a_{\max}$

3. Mathematical Expressions

Power Input (to battery)

$$P_{\text{in}} = (I_{\text{POA}} \cdot A_{\text{panel}} \cdot \eta_{\text{PV}}) + P_{\text{regen}}$$

Power Output (from battery) This is the electrical power required by the motor (P_{elec}) and auxiliary systems.

$$P_{\text{out}} = P_{\text{elec}} + P_{\text{aux}} = \frac{P_{\text{tractive}}}{\eta_{\text{motor}}} + P_{\text{aux}}$$

Where the required tractive power is a function of resistive forces ($F_{\text{resistive}}$ from Question 1), mass (m), acceleration (a), and velocity (v):

$$P_{\text{tractive}} = (F_{\text{resistive}} + ma) \cdot v$$

The complete expression for power output is therefore:

$$P_{\text{out}} = \frac{(\frac{1}{2}\rho C_d A v^2 + C_{rr} mg\cos\theta + mg\sin\theta + ma) \cdot v}{\eta_{\text{motor}}} + P_{\text{aux}}$$

Governing Power Balance The change in the battery's stored energy (E_{battery}) is the net power flow, accounting for battery charge/discharge efficiencies ($\eta_{\text{charge/discharge}}$).

$$\frac{dE_{\text{battery}}}{dt} = (P_{\text{in}} \cdot \eta_{\text{charge}}) - \frac{P_{\text{out}}}{\eta_{\text{discharge}}}$$

1.5 Question 5: Lagrangian Mechanics Formulation

1. Derivation of the Lagrangian and Equations of Motion

The Lagrangian (L) is defined as the system's kinetic energy (T) minus its potential energy (V).

- **Kinetic Energy:** $T = \frac{1}{2}m\dot{x}^2$
- **Potential Energy:** $V = mgx\sin\theta$

where x is the generalized coordinate representing the distance traveled. The Lagrangian for the car is therefore:

$$L(x, \dot{x}) = T - V = \frac{1}{2}m\dot{x}^2 - mgx\sin\theta$$

The equations of motion are found using the Euler-Lagrange equation, which includes a term for non-conservative generalized forces (Q_{nc}):

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{x}}\right) - \frac{\partial L}{\partial x} = Q_{nc}$$

For this system, the non-conservative forces are: $Q_{nc} = F_{\text{tractive}} - F_{\text{aero}} - F_{\text{rolling}}$. Evaluating the terms of the Euler-Lagrange equation yields:

$$\frac{d}{dt}(m\dot{x}) - (-mg\sin\theta) = F_{\text{tractive}} - F_{\text{aero}} - F_{\text{rolling}}$$

This simplifies to the final equation of motion:

$$m\ddot{x} = F_{\text{tractive}} - F_{\text{aero}} - F_{\text{rolling}} - mg\sin\theta$$

2. Comparison with Newtonian Formulation

The equation of motion derived using the Lagrangian is **identical** to the one derived from Newton's Second Law in Question 1, where the gravitational force component is $F_{\text{gravity}} = mg\sin\theta$. This equivalence validates the physical model.

3. Differences and Expression of Constraints

Differences in the Approach

- **Scalar vs. Vector:** The Lagrangian method is fundamentally scalar (energy-based), simplifying the setup compared to the vector-based force analysis of Newtonian mechanics.
- **Constraint Forces:** The Lagrangian formulation automatically eliminates forces that do no work (e.g., the normal force), which must be explicitly managed in a Newtonian free-body diagram.

Expressing Constraints

- **Mechanical Constraints:** Physical constraints, such as the car being confined to the road surface, are handled implicitly by the choice of an appropriate set of generalized coordinates (in this case, only x).
- **Optimization Constraints:** Operational constraints, such as speed limits or battery state limits ($v \leq v_{\max}$, $\text{SOC} \geq \text{SOC}_{\min}$), are not part of this physical derivation. They are imposed separately on the optimal control problem and are handled mathematically using **Lagrange Multipliers** within an optimization framework.

2 Bits & Bytes may break your bones

2.1 Question 1: Objective Function Analysis

A. Why Direct Differentiation is Not Feasible

Directly differentiating the objective function and solving for zero to find the minimum is not feasible for a real-world race strategy problem. This analytical approach fails for three primary reasons:

- **Function Complexity:** Key terms in the objective, such as `sun_gain` and `battery_los`, are not simple, differentiable functions of the decision variable (velocity). They depend on complex, often non-differentiable external data (e.g., weather forecasts) and highly non-linear system dynamics (e.g., battery electrochemistry).
- **High Dimensionality:** The optimization variable is not a single scalar but a high-dimensional vector representing the velocity profile over thousands of time steps. This leads to a large system of coupled, non-linear equations that cannot be solved analytically.

Due to these factors, the problem must be solved with iterative, numerical optimization algorithms.

B. The Hidden Cost of Precomputing `sun_gain`

While precomputing `sun_gain` for all 86,400 timesteps in a day can accelerate function calls, it comes with significant hidden costs that make it a poor strategy:

- **Loss of Model Flexibility and Accuracy:** The primary cost is that precomputation assumes `sun_gain` is only a function of time. This is incorrect. Solar gain is also a function of the car's state (specifically, its location), which is determined by the very velocity profile we are trying to optimize. This breaks the critical dependency, making the model unable to react to route deviations or dynamic weather.
- **Memory Footprint:** Storing large arrays of precomputed data consumes significant memory.

2.2 Question 2: Time Complexity Analysis

`find_optimal_angle(panels)`

This function finds the best angle for a set of solar panels by checking a fixed number of angles and, for each one, summing up the power from all panels.

Analysis: The function's structure is a **nested loop**.

- The **outer loop** (`for angle in range(0, 181, 5)`) iterates over a predefined, fixed set of angles. The number of iterations is constant (37) and does not depend on the size of the input `panels`.
- The **inner loop** (`for panel in panels`) iterates through every panel in the input list. If there are n panels in the list, this loop runs n times.

Time Complexity: The total number of main operations is the product of the outer and inner loop's iterations. Total Operations $\approx 37 \times n$. In Big-O notation, we ignore constant factors because we are interested in how the runtime scales as the input n grows very large. Therefore, the time complexity simplifies to $\mathbf{O(n)}$, which is **linear time**.

`balance_cells(cells)`

This function sorts a list of battery cells. It is a classic recursive, "divide and conquer" algorithm recognized as **Quicksort**.

Analysis: The function works by:

1. **Base Case:** If the list has 1 or 0 elements, it is already sorted.
2. **Divide:** It picks a `pivot` and partitions the list into three sub-lists: elements smaller than, equal to, and larger than the pivot. This partitioning step requires iterating through all n elements once, so it takes $O(n)$ time.
3. **Conquer:** It recursively calls `balance_cells` on the 'left' and 'right' sub-lists and then combines the results.

Time Complexity: The performance of this algorithm depends heavily on how well the `pivot` splits the list.

- **Average and Best Case:** $\mathbf{O(n \log n)}$. When the pivot consistently splits the list into roughly two equal halves, the depth of the recursion is $\log n$. Since each level of recursion involves partitioning a total of n elements, the total work is n operations done $\log n$ times.
- **Worst Case:** $\mathbf{O(n^2)}$. If the pivot is consistently a poor choice (e.g., always the smallest or largest element), the list is split into one sub-list of size $n-1$ and an empty one. This creates a recursion depth of n levels. The total work becomes a sum of $n + (n-1) + \cdots + 1$, which is $O(n^2)$.

Although there is a quadratic worst case, the algorithm is highly efficient in practice, with an expected runtime of $\mathbf{O(n \log n)}$.

2.3 Question 3: Code Optimization

The most effective way to optimize this function is by replacing the slow Python loop with a **vectorized NumPy operation**. This performs the calculation on all segments at once in highly optimized, compiled C code, leading to orders-of-magnitude speedup.

The Problem with the Original Code

The provided code is slow because it is doing math inside a Python `for` loop. The Python interpreter has a high overhead for each iteration, and accessing attributes like `segments[i].irradiance` repeatedly adds up, especially when the function is called 100 times per second.

The Solution: NumPy Vectorization

For numerical code that runs frequently, **NumPy** is the answer. The strategy is to convert the data into NumPy arrays and then perform the entire calculation in a single, vectorized expression without any explicit `for` loops in Python.

```
import numpy as np
def sunniest_segment_optimized(segments):
    """
    Finds the sunniest segment using optimized NumPy vectorization.
    """
    # Convert list of objects to NumPy arrays
    irradiances = np.array([s.irradiance for s in segments])
    angles = np.array([s.angle for s in segments])
```

```
    # Perform the entire calculation in a single vectorized operation
    effective_irradiances = irradiances * np.cos(np.radians(angles -
    sun_angle))
    # Find the index of the maximum value using NumPy's fast argmax
    best_index = np.argmax(effective_irradiances)
    # Return the corresponding segment object
    return segments[best_index]
```

Listing 1: Fully optimized implementation using NumPy

How it Works

- **Vectorization:** Instead of looping, we create arrays of irradiances and angles. NumPy can then apply mathematical operations to every element in the array simultaneously at C-level speed.
- **`np.argmax()`:** After calculating all the effective irradiances in one shot, we use NumPy's highly optimized `argmax()` function to find the index of the single best segment instantly, without needing to manually compare elements.

This vectorized approach is the standard for high-performance scientific computing in Python and is exactly what is needed for a function in a critical, high-frequency loop.

2.4 Question 4: Floating-Point Imprecision

The function fails due to **floating-point imprecision**, where decimal numbers cannot be stored with perfect accuracy in binary. The fix is to stop using the exact equality operator `==` and instead check if the calculated distance is "close enough" to the target, ideally using Python's built-in `math.isclose()` function.

The Mission: Find the Bug

The function unexpectedly returns `False` for inputs where the math seems correct, especially when dealing with common decimal fractions.

Failing Inputs: Consider the inputs `speed = 0.1`, `time = 0.2`, and `checkpoint_distance = 0.02`. Mathematically, $0.1 \times 0.2 = 0.02$. However, the function will return `False`.

Why It Happens: The Problem with Floats

This happens because computers use a binary (base-2) system to store numbers, and they cannot represent most decimal fractions perfectly.

- **The Analogy:** This is similar to how we cannot write the fraction 1/3 perfectly in base-10. We write it as $0.33333\ldots$, an infinite repeating sequence we must eventually round off.
- **The Reality:** The number 0.1 in binary is also an infinite repeating sequence. The computer must truncate this, leading to a tiny representation error. When you multiply the computer's representation of 0.1 by 0.2, the result is a number that is extremely close, but not exactly, equal to 0.02.
- **Exact Comparison:** The `==` operator demands exact, bit-for-bit equality. Even the tiniest difference will cause it to return `False`.

The Fix: Checking for "Closeness"

You should never use `==` to compare floating-point numbers. The correct approach is to check if the numbers are close enough by testing if their absolute difference is within a small **tolerance**. The best-practice, safest and most readable way to do this in Python is with the `math.isclose()` function.

```python
import math
# Original Failing Function
def reached_checkpoint_original(speed, time, checkpoint_distance):
  """Returns True if the car's traveled distance exactly matches the checkpoint"""
  return speed * time == checkpoint_distance
# Fixed and Robust Function
def reached_checkpoint_fixed(speed, time, checkpoint_distance):
  """
  Returns True if the car's traveled distance is close enough,
  accounting for floating-point imprecision
  """
  traveled_distance = speed * time
  return math.isclose(traveled_distance, checkpoint_distance)
# Simple Test Case
s, t, d = 0.1, 0.2, 0.02
print(f"Inputs: speed={s}, time={t}, distance={d}")
print(f"Actual product in Python: {s * t}")
print("-" * 20)
print(f"Original function: {reached_checkpoint_original(s, t, d)}") # False
print(f"Fixed function: {reached_checkpoint_fixed(s, t, d)}")     # True
```

Listing 2: Demonstration of the bug and the fix

This fixed function will now behave as expected for all real-world floating-point inputs.

2.5 Question 5: High-Frequency Data Pipeline

For a 500Hz telemetry pipeline, the best approach is using a **Just-In-Time (JIT) compiler like Numba**. It offers the C-like speed needed for real-time processing with the lowest possible effort, allowing the team to keep writing simple Python code.

The Challenge: High-Frequency Data

Processing sensor data at 500Hz means you have only **2 milliseconds** to execute your calculations for each new data point. Standard, interpreted Python is often too slow for this because of its inherent overhead. The goal is to make a Python function run as fast as if it were written in a low-level language like C or C++.

Optimization Strategies

Here are three powerful strategies to achieve the required "orders of magnitude" speedup, moving from the simplest to the most complex.

1. **Just-In-Time (JIT) Compilation with Numba:** This involves adding a simple `@jit` decorator to your Python function. The first time the function runs, Numba translates it into highly optimized machine code which is then used for all subsequent calls.

2. **Ahead-of-Time Compilation with Cython:** This approach involves writing code in a Python-like language (with optional C-style static types) and compiling it into a C extension module *before* you run your main program. Your main script then imports this compiled module.

3. **Calling C/C++ Libraries:** This is the most traditional approach. You write the performance-critical code entirely in C or C++, compile it into a shared library, and then use a Python "wrapper" (like `ctypes` or `pybind11`) to call this library from your Python script.

Trade-Off Comparison

Table 2: Comparison of Python Performance Strategies

Strategy	**Performance Speedup**	**Implementation Effort**	**Maintainability**
Numba (JIT)	Excellent (10-100x+)	Very Low (add `@jit`)	High (Pure Python)
Cython	Excellent (10-100x+)	Medium (New syntax, build steps)	Medium (Hybrid code)
C/C++ Lib	Highest Possible	High (C++ skills, wrappers)	Low (Multi-language)

All listed methods have excellent compatibility with embedded systems like the Raspberry Pi.

Recommendation for Agnirath

The best approach for the Agnirath solar racing application is **Numba**. It offers the ideal balance of performance and practicality for a university team working under tight deadlines.

- **Massive Performance, Minimal Effort:** Numba delivers the C-like speeds necessary to meet the 500Hz real-time deadline without forcing the team to rewrite their logic or learn a new language. The productivity gain is enormous.
- **High Maintainability:** The code remains clean, readable and pure Python. This is a critical advantage for team longevity, as new members can immediately understand and contribute to the codebase without needing specialized Cython or C++ knowledge.
- **Perfect for Embedded Systems:** Numba is fully compatible with ARM architectures, making it a perfect fit for on-board computers like the Raspberry Pi.

While Cython and C++ extensions are powerful, they introduce significant development and maintenance overhead. Numba provides nearly all of the performance benefits with a fraction of the complexity, making it the clear and strategic choice for this application.

3 Bound by the Sun

3.1 Question 1: KKT Conditions and LICQ

Part 1: Proof of KKT Necessity under LICQ

Let x^* be a local minimizer of the problem that satisfies the **Linear Independence Constraint Qualification (LICQ)**. The condition that x^* is a local minimum implies that there are no feasible descent directions from x^*. That is, for any vector d in the tangent cone $T_F(x^*)$, we must have $\nabla f(x^*)^T d \geq 0$.

The LICQ condition ensures that the tangent cone $T_F(x^*)$ is equal to the cone of linearized feasible directions, $F(x^*)$, defined as:

$$F(x^*) = \{d \in \mathbb{R}^n \mid \nabla h_i(x^*)^T d = 0 \text{ for } i \in E; \nabla g_j(x^*)^T d \leq 0 \text{ for } j \in A(x^*)\}$$

where $A(x^*)$ is the set of active inequality constraints at x^*.

Thus, for any $d \in F(x^*)$, we have $\nabla f(x^*)^T d \geq 0$. By a theorem of the alternative (a consequence of Farkas' Lemma), this geometric condition is equivalent to the algebraic condition that $-\nabla f(x^*)$ belongs to the cone generated by the gradients of the active constraints. This means there exist multipliers λ_i^* for $i \in E$ and $\mu_j^* \geq 0$ for $j \in A(x^*)$ such that:

$$-\nabla f(x^*) = \sum_{i \in E} \lambda_i^* \nabla h_i(x^*) + \sum_{j \in A(x^*)} \mu_j^* \nabla g_j(x^*)$$

By setting $\mu_j^* = 0$ for all inactive constraints ($j \notin A(x^*)$), we satisfy the **complementary slackness** condition ($\mu_j^* g_j(x^*) = 0$ for all $j \in I$) and recover the **stationary condition**:

$$\nabla f(x^*) + \sum_{i=1}^{m} \lambda_i^* \nabla h_i(x^*) + \sum_{j=1}^{p} \mu_j^* \nabla g_j(x^*) = 0$$

The remaining KKT conditions (primal and dual feasibility) are satisfied by definition.

Part 2: Counterexample for LICQ Failure

Consider the optimization problem:

$$\begin{aligned} \min_{x \in \mathbb{R}^2} \quad & f(x_1, x_2) = x_1 \\ \text{subject to:} \quad & g(x_1, x_2) = -(x_2^2 + x_1^3) \geq 0 \end{aligned}$$

Local Minimizer: The feasible region is defined by $x_1^3 \leq -x_2^2$. Since $-x_2^2 \leq 0$, we must have $x_1 \leq 0$. To minimize $f(x) = x_1$, we must choose the largest possible value of x_1, which is 0. This implies $x_2 = 0$. Thus, the point $x^* = (0, 0)$ is the unique global minimizer.

LICQ Failure: At $x^* = (0, 0)$, the constraint $g(0, 0) = 0$ is active. The gradient of this active constraint is:

$$\nabla g(x_1, x_2) = [-3x_1^2, -2x_2]$$

At $x^* = (0, 0)$, we have $\nabla g(0, 0) = [0, 0]$. This set of one active constraint gradient is linearly dependent. Therefore, LICQ fails at the minimizer.

KKT Insufficiency: The KKT stationary condition is $\nabla f(x^*) + \mu^* \nabla g(x^*) = 0$, with $\mu^* \geq 0$. The gradient of the objective function is $\nabla f(x_1, x_2) = [1, 0]$. At $x^* = (0, 0)$, the stationary condition becomes:

$$[1, 0] + \mu^*[0, 0] = [0, 0]$$

This simplifies to $[1, 0] = [0, 0]$, which is a contradiction. No such $\mu^* \geq 0$ exists. This demonstrates that even though $x^* = (0, 0)$ is a minimizer, the KKT conditions are not necessary because the LICQ constraint qualification does not hold.

3.2 Question 2: Lagrangian Duality and Concavity

1. The Lagrangian Function

The Lagrangian function for the given constrained optimization problem is defined as:

$$L(x, \lambda, \mu) = f(x) + \lambda^T h(x) + \mu^T g(x)$$

where $\lambda \in \mathbb{R}^m$ and $\mu \in \mathbb{R}^p$ are the Lagrange multiplier vectors for the equality and inequality constraints, respectively. This function combines the objective and constraints into a single expression, where the multipliers can be interpreted as penalties for violating the constraints.

2. Proof of Concavity of the Dual Function

The Lagrange dual function $d(\lambda, \mu)$ is defined as the infimum of the Lagrangian with respect to x:

$$d(\lambda, \mu) = \inf_x L(x, \lambda, \mu)$$

To prove that $d(\lambda, \mu)$ is a concave function of (λ, μ), we proceed as follows:

1. For any fixed value of x, the Lagrangian $L(x, \lambda, \mu)$ is an **affine function** of (λ, μ).

$$L(x, \lambda, \mu) = \left(\sum_{i=1}^{m} h_i(x)\lambda_i + \sum_{j=1}^{p} g_j(x)\mu_j \right) + f(x)$$

   This is of the form $A^T z + b$, which is the definition of an affine function.

2. Any affine function is both convex and concave. Therefore, for each fixed x, $L(x, \lambda, \mu)$ is a **concave function** of (λ, μ).

3. The dual function $d(\lambda, \mu)$ is the **pointwise infimum** of this family of concave functions, taken over all possible values of x.

4. A fundamental property of concave functions is that the pointwise infimum of any collection of concave functions is also **concave**.

Therefore, we conclude that the dual function $d(\lambda, \mu)$ is concave. This holds true regardless of whether the original (primal) problem is convex. This result is powerful because it ensures the dual problem, $\max_{\lambda, \mu \geq 0} d(\lambda, \mu)$, is a convex optimization problem, which is typically easier to solve.

3.3 Question 4: Penalty vs. Augmented Lagrangian Methods

1. Feasibility and Ill-Conditioning

Both the quadratic penalty method and the augmented Lagrangian method enforce feasibility of the equality constraints $h(x) = 0$ as the penalty parameter $\rho \to \infty$. In both cases, the term $\frac{\rho}{2}\|h(x)\|^2$ would cause the objective to become infinite unless $\|h(x)\|^2 \to 0$. However, they differ critically in their numerical properties:

- **Penalty Method:** This method only converges to the true constrained minimizer in the limit as $\rho \to \infty$. This causes the Hessian of the penalty function to become ill-conditioned, as its eigenvalues associated with the constraint space grow proportionally with ρ.
- **Augmented Lagrangian:** This method avoids ill-conditioning by introducing a Lagrange multiplier estimate, λ. It can converge to the true minimizer for a finite, sufficiently large value of ρ. Convergence is achieved by iteratively updating λ to its optimal value, rather than by increasing ρ to infinity. This maintains a well-conditioned Hessian, leading to superior numerical stability and faster convergence.

2. Hessian Analysis for $h(x) = x^2 - 1$

Let's analyze the Hessian of each method's objective function at the solution $x^* = 1$, assuming a trivial objective $f(x) = 0$ for clarity.

Penalty Function: The objective is $P(x) = \frac{\rho}{2}(x^2 - 1)^2$. The Hessian is $P''(x) = 2\rho(3x^2 - 1)$. At $x = 1$, the Hessian is $P''(1) = 4\rho$. To find the solution, the penalty method requires $\rho \to \infty$, causing the Hessian's value to become unbounded.

Augmented Lagrangian Function: The objective is $\mathcal{L}_\rho(x, \lambda) = \lambda(x^2 - 1) + \frac{\rho}{2}(x^2 - 1)^2$. The Hessian is $\mathcal{L}''_\rho(x, \lambda) = 2\lambda + 2\rho(3x^2 - 1)$. At $x = 1$, the Hessian is $\mathcal{L}''_\rho(1, \lambda) = 2\lambda + 4\rho$. This method converges by finding an optimal λ^* for a large but finite ρ. The Hessian is therefore positive and finite at the solution, indicating a well-conditioned problem.

Impact on Newton-Type Solvers: Newton's method requires solving a linear system using the Hessian matrix to find the search direction. An ill-conditioned or unbounded Hessian, as encountered in the penalty method, leads to significant numerical errors, causing the solver to take inaccurate steps, converge slowly, or fail. The well-conditioned Hessian of the augmented Lagrangian method makes it far more robust and efficient for Newton-type solvers.

3.4 Question 5: Comparison of Optimization Methodologies

1. Best Suited Algorithm: SQP, IPM, or Augmented Lagrangian

For the Agnirath race strategy problem—a large-scale, non-linear, constrained optimization task—**Sequential Quadratic Programming (SQP)** is a highly suitable method.

Argument: While all three are powerful, SQP offers an excellent balance of efficiency and generality. Its fast quadratic convergence makes it ideal for the complex, non-linear dynamics of the solar car.

- **Interior-Point Methods (IPM)** are strong contenders, especially for the very large-scale nature of the fully discretized problem, but can be less efficient than SQP on a per-iteration basis.

- **Augmented Lagrangians (AL)** offer superior robustness, which is valuable, but often at the cost of slower convergence compared to SQP.

Therefore, SQP represents a robust and high-performance choice for finding the optimal velocity profile.

2. Numerical Integration Scheme: Forward Euler

A fundamental numerical integration scheme for solving Ordinary Differential Equations (ODEs) of the form $y' = f(t, y)$ is the **Forward Euler method**.

Method: It approximates the state at the next time step, y_{k+1}, by taking the current state, y_k, and adding a step of size h in the direction of the current tangent:

$$y_{k+1} = y_k + h \cdot f(t_k, y_k)$$

Error Analysis The method's error is characterized by its order of accuracy.

- **Local Truncation Error** (error per step) is $O(h^2)$.
- **Global Truncation Error** (accumulated error) is $O(h)$.

This makes it a first-order method. Its primary weakness is its conditional stability, which requires a sufficiently small step size h to avoid divergence.

3. Comparison of Optimization Approaches

The following table compares the benefits and disadvantages of the three algorithmic classes in the context of the Agnirath strategy problem.

Table 3: Comparison of Optimization Algorithm Classes

Method Class	Benefits	Disadvantages	Primary Agnirath Use Case
Numerical Integration	• Essential for simulating the car's dynamic state (e.g., battery SOC) over time.	• Is a simulation tool, not an optimizer. • Accuracy is dependent on step size and method order.	Physics Simulation: To evaluate the cost (e.g., race time, energy) of a candidate velocity profile.
Gradient-Based Search	• Highly efficient and fast for smooth, differentiable problems. • Guarantees convergence to a local minimum.	• Requires calculation of gradients. • Can get trapped in local minima in non-convex problems.	Core Optimizer: To solve the main Non-Linear Program and find the optimal velocity profile.
Agentic Search	• Excellent at escaping local minima and finding global solutions. • Does not require gradients. • Easily parallelizable.	• Convergence is typically much slower. • No strict guarantee of optimality.	High-Level Strategy: To optimize discrete decisions or find a starting guess for the gradient-based solver.

4 Stayin' in Control

4.1 Question 1: State, Controls, and Stochasticity

Part 1: State and Control Inputs

The State Vector (x_t): The **state** is the minimum set of variables needed to fully describe the car's condition at a specific moment in time. For the solar car, the state vector would include:

- **Battery State of Charge (SOC):** The most critical resource; the available "fuel."
- **Distance Traveled:** Progress along the 3000km route.
- **Current Velocity:** Determines immediate power consumption and future position.
- **Component Temperatures:** Crucial for safety constraints (e.g., battery and motor temps).
- **Time:** The current time of day and/or time elapsed in the race.

A snapshot of the car at time t can be represented as a vector:

$$x_t = [\text{SOC, distance, velocity, } T_{\text{batt}}, T_{\text{motor}}, \text{time}]^T$$

The Control Inputs (u_t): **Controls** are the levers we can adjust at each step to steer the car's state. The primary control input is Motor Power Command - the main throttle. Other, more specialized controls could include the tilt angle of the solar array or the application of mechanical brakes.

Part 2: Deterministic vs. Stochastic Control

Control execution is overwhelmingly **stochastic** (i.e., it has a random component), not deterministic (perfectly predictable).

Why it is Stochastic: A deterministic system implies that if we command the motor to use 1kW of power, we would know exactly what our velocity and energy consumption will be in the next second. This is impossible in the real world because the system is heavily influenced by a random and unpredictable **environment**. This randomness comes from several sources:

- **Weather:** Sudden cloud cover can instantly drop solar power input to a fraction of the forecast, making energy gain stochastic.
- **Wind:** An unexpected headwind will increase aerodynamic drag, meaning the same motor power command results in a lower speed than predicted.
- **Traffic and Obstacles:** Sharing the road with other vehicles may force unplanned deceleration, deviating from the optimal plan.
- **Road Conditions:** Small changes in road surface quality can alter rolling resistance.

Because these unpredictable stochastic elements directly affect the car's state (like SOC and velocity), a robust strategy cannot simply follow a pre-planned set of controls. It must continuously measure its current state.

4.2 Question 2: Objective, Constraints, and Feasibility

Part 1: The Objective Function (Our Goal)

The single, overarching goal of the race is to finish as quickly as possible. Therefore, our primary objective is to **minimize the total race time,** t_f.

Mathematical Formulation: We seek a velocity profile over time, $v(t)$, that solves the optimization problem:

$$\min_{v(t)} t_f$$

This is subject to the condition that the total distance traveled at the final time is the full race distance:

$$\text{distance}(t_f) = 3000\,\text{km}$$

Leveraging the Finite Horizon: The hint correctly identifies this as a **finite-horizon problem**—it has a well-defined beginning and end. This is powerful because it allows us to frame our goal as reaching a specific terminal state (3000 km traveled) in the minimum possible time, making the problem suitable for standard optimal control frameworks.

Part 2: Constraints and Bounds (The Rules of the Road)

Our strategy cannot choose any velocity profile freely; it must operate within a strict set of rules, which can be broken down into bounds, path constraints, and terminal constraints.

Bounds (Instantaneous Variable Limits)

- **Velocity:** $0 < v(t) \leq v_{max}\,\text{km/h}$. The car must be moving forward and must obey the event's speed limit.
- **Motor Power:** $P_{\text{regen,max}} \leq P_{\text{motor}}(t) \leq P_{\text{drive,max}}$. The motor has physical limits on both driving and regenerative braking power.

Path Constraints (Must be obeyed throughout the race) These are state variable constraints that must hold true for the entire duration of the race.

- **Battery State of Charge (SOC):** $20\%(e.g.;) \leq \text{SOC}(t) \leq 100\%$. This is our most critical **hard constraint**. Violating the lower bound ends the race.
- **Component Temperatures:** $T_{\text{battery}}(t) \leq T_{\text{batt,max}}$ and $T_{\text{motor}}(t) \leq T_{\text{motor,max}}$. This is another hard constraint to ensure safety and prevent hardware damage.

Terminal Constraints (Conditions for the finish line)

- **Final Distance:** $\text{distance}(t_f) = 3000\,\text{km}$. This condition defines the end of the problem.
- **Final Battery Charge:** $\text{SOC}(t_f) \geq 20\%$(e.g.;). We must cross the finish line with a non-empty battery. This can be treated as a soft constraint.

Part 3: Ensuring a Feasible and Optimal Solution

Existence of a Feasible Solution: A feasible solution is any strategy that completes the race without violating hard constraints. We can prove one exists with a simple thought experiment: a strategy of driving at a very slow, constant speed (e.g., 30 km/h) would, on a sunny day, generate far more solar power than it consumes. This would keep the battery full and all components cool. While not optimal, the existence of such a simple, safe strategy proves that the set of feasible solutions is non-empty.

Existence of an Optimal Solution: The existence of a *best* solution is guaranteed by the **Weierstrass Extreme Value Theorem**. It states that a minimum and maximum must exist if two conditions are met:

1. **The function is continuous:** Our objective (race time) and the underlying physics models are continuous.
2. **The domain is compact (closed and bounded):** Our set of all possible feasible solutions is bounded by the constraints (e.g., velocity is limited, SOC is bounded on both fronts). It is also closed because the constraints are inclusive inequalities ($\leq$).

Since our problem is to minimize a continuous function over a non-empty, compact set of possibilities, an optimal solution is mathematically guaranteed to exist.

4.3 Question 3: Pros and Cons of MPC

Pros of Model Predictive Control (MPC)

- **Explicit Constraint Handling:** MPC natively incorporates state and control constraints (e.g., battery SOC limits, motor temperature, speed limits) into its optimization, ensuring safe and feasible operation.
- **Anticipatory Control:** Its finite-horizon prediction allows the controller to use forecasts of future events (e.g., upcoming hills, cloud cover) to make proactive, optimal decisions.
- **Inherent Robustness:** The receding-horizon strategy (re-planning at every step) makes MPC naturally resilient to un-modeled disturbances and changing environmental conditions.

Cons of Model Predictive Control (MPC)

- **High Computational Demand:** MPC requires solving a computationally intensive optimization problem online at each time step.
- **Model Dependency:** The controller's performance is highly dependent on the accuracy of the system's predictive model. A significant mismatch between the model and reality can degrade performance.
- **Complex Parameter Tuning:** It requires careful tuning of its parameters (e.g., prediction horizon, control weights) to achieve a good balance between performance, robustness, and computational feasibility.

4.4 Question 4: MPC Parameter Tuning

Relationship Between Optimization Time (τ) and Control Period (dt)

The control loop period, dt, must be strictly greater than the time required to solve the optimization problem, τ.

$$\tau < dt$$

$$\tau_{max} < dt$$

If the computation for a control action (τ) takes longer than the interval between actions (dt), the controller cannot keep up with the system's evolution, leading to performance degradation and instability.

Horizon Lengths

Prediction Horizon ($\hat{T}$): It must be long enough to capture the significant effects of control actions. For the solar car, it should encompass key strategic events like climbing a hill or passing through a forecasted weather system.. The trade-off is between better long-term performance (longer $\hat{T}$) and computational tractability (shorter $\hat{T}$).

Execution Horizon (T_e): The execution horizon is the first time step of the computed optimal plan. This receding horizon approach ensures the controller can react to new information at each step. Therefore, $T_e = dt$.

Observation Horizon (T_o): This is the single most recent state measurement, so $T_o = dt$. A longer horizon may be used if state estimation, disturbance estimation, or noise filtering based on a history of measurements is required.

4.5 Question 5: MPC Initial Guess and Global Optima

1. The Optimal Initial Guess: Warm Starting

A fixed initial guess is inefficient. The optimal approach is to use the solution from the previous time step, shifted forward.

Method: The optimal control sequence found at time k is used as the initial guess for the optimization at time $k + 1$.

Benefit: Because the system's state evolves continuously, this initial guess is already very close to the new optimal solution. This dramatically reduces the solver's computation time (τ).

The best variable initial guess is the optimal solution from the previous time step, shifted forward by one time step. This 'warm start' is almost always better than any other guess because the system's state evolves continuously, meaning the new optimal solution will be very close to the old one

2. Escaping Local Minima with Stochasticity

While a warm start is fast, it risks getting the optimizer permanently stuck in a sub-optimal local minimum. To address this and search for the global minimum, we can introduce a controlled amount of randomness, inspired by the mechanics of Stochastic Gradient Descent (SGD).

Method: We apply a small, random perturbation to the warm start solution before feeding it to the optimizer each cycle.

```
initial_guess = warm_start_solution + random_noise
```

Benefit: This technique provides a mechanism for exploration. While the warm start ensures the solver begins in a good region, the injected noise allows it to occasionally "jump" out of a local minimum's basin of attraction and potentially discover a more globally optimal solution. This balances the need for rapid convergence with a robust search for the best possible strategy.

5 Hittin' the Home Run

5.1 Question 1: Motor Thermal Model

The Goal: Finding Thermal Equilibrium

The problem describes a feedback loop where the motor's temperature affects its resistance, which in turn affects its power loss (heat generation), and then its temperature. We need to find the equilibrium point where the heat being generated is perfectly balanced by heat dissipation, and the temperature no longer changes. Since the equations are interdependent, we use an iterative approach to find this stable fixed point.

The Implementation Logic

The algorithm follows these steps:

1. **Initialize:** Start with the given initial guess for the winding temperature, $T_w = 323\,\text{K}$.
2. **Calculate:** Inside the loop, use the current value of T_w to calculate all intermediate values in order: T_m, B, i, R, P_c, and P_e.
3. **Update:** Use the calculated power losses to compute a new value for T_w.
4. **Check for Convergence:** Compare the new T_w with the old one. If the absolute difference is less 1, we have found the stable solution and exit the loop. Otherwise, we repeat the process with the updated temperature.

Results and Discussion

When tested with an ambient temperature $T_a = 293$ K and a motor torque $\tau = 16.2$ Nm, the function converges to a steady-state winding temperature of **304.74 K**.

5.2 Question 2: Geographic and Environmental Data

The Goal: Building a Copy of the Route

To run a high-fidelity simulation, we need a detailed digital model of the race route. This data is critical for our physics engine:

- **Altitude/Elevation:** This is the most important piece. The change in altitude between points tells us the road's gradient, which we need to calculate the gravitational force (F_{gravity}) acting on the car.
- **GPS Coordinates & Bearing:** The coordinates tell us our location, and the bearing (direction of travel) is essential for the solar model to calculate the angle of the sun relative to the car's solar panel.

The Plan: Choosing and Using the APIs

The strategy is to chain together a few tools:

1. **Get the Route Path (OSRM):** First, we need the actual path of the road. The Open-Source Routing Machine (OSRM) API is perfect for this. We give it a start and end coordinate, and it returns a highly compressed "polyline" that represents the entire GPS path.
2. **Get Elevation Data (Open-Elevation):** Once we decode the polyline into a list of latitude/longitude points, we send these points to the free Open-Elevation API. It takes our list of coordinates and returns the corresponding altitude for each one.

3. **Calculate Bearing:** We don't need an API for this. With a list of coordinates, we can calculate the bearing (the direction from one point to the next) ourselves using a standard trigonometric formula.

The Justification: Choosing the Right Resolution

The key decision is how far apart our data points should be. This is a trade-off between accuracy and dataset size.

- **Too High (e.g., every 5 meters):** Would create a massive file with tens of thousands of points. This would be slow to download, process, and use in the simulation.
- **Too Low (e.g., every 1 kilometer):** Would result in a small, fast file, but it would completely miss most hills and turns. The simulation would be highly inaccurate.

A resolution of **50-200 meters** is the ideal compromise for this application. It's granular enough to capture the vast majority of significant changes in road gradient and direction, which is what our energy model cares about. At the same time, it keeps the total number of data points for the 350km trip to a manageable size, which is fast and easy to work with.

Results and Final Output

The Python script was successfully executed to generate the route data.

- The OSRM API returned a route composed of **5,011 points**. For the approximately 350 km journey, this provides an excellent average resolution of about **70 meters** between points, which is well within the desired range for an accurate simulation.
- The script successfully fetched altitude data for all points and calculated the bearing for each segment.

The final data was saved to `chennai_to_bangalore_route.csv`. A sample of the generated data is shown below:

Table 4: Sample of the Generated Route Data

Latitude	Longitude	Altitude (m)	Bearing (deg)
13.08430	80.27050	8.0	NaN
13.08430	80.27052	8.0	89.999998
13.08426	80.27052	8.0	180.000000
13.08429	80.27044	8.0	291.056475
13.08434	80.27032	8.0	293.159921

5.3 Question 3: The FINAL Challenge

Strategy 1: Time-Minimization with a Gradient-Based Optimizer

This first strategy models the problem as a classic non-linear constrained optimization task. The goal is to find the fastest possible time to complete the 350 km race, subject to the physical limitations of the car, primarily the battery capacity. The problem is solved using a powerful gradient-based algorithm.

Model Formulation

- **Objective:** Minimize the total race time, t_f.

- **Decision Variables:** The velocity profile of the car, discretized into a vector of 100 constant-velocity steps, v_i for $i = 1, \ldots, 100$.
- **Constraints:**
  - The final State of Charge (SOC) must be greater than or equal to the minimum limit: $\text{SOC}(t_f) \geq 20\%$.
  - The velocity at each step must remain within the physical and regulatory bounds: $30\,\text{km/h} \leq v_i \leq 100\,\text{km/h}$.

Implementation Details

The optimization problem was solved using SciPy's `minimize` function with the **SLSQP (Sequential Least Squares Programming)** algorithm. This method is well-suited for non-linear problems with both equality and inequality constraints.

The objective and constraint functions are evaluated by a physics simulator. For any given velocity profile, the simulator runs a step-by-step loop that:

1. Calculates resistive forces (aerodynamic drag and rolling resistance).
2. Determines the required motor power output.
3. Calculates the solar power input using a synthetic sine-curve model for the race day.
4. Updates the battery's state of charge based on the net power flow.

If a trial velocity profile results in the battery SOC dropping below 20% before the finish line, the simulation fails, and a large penalty is returned to the optimizer to steer it away from such infeasible solutions.

Results and Analysis

The optimizer successfully converged after 52 iterations, finding an optimal race time of **3.93 hours**. The resulting strategy is visualized in the plots below.

Key Observations:

- **Velocity Profile:** The optimal strategy is to maintain a nearly constant, high velocity (around 89 km/h) for the entire race. The sharp drop at the very end is an artifact of the final time step being used to precisely meet the 350 km distance without overshooting.
- **Battery SOC:** The battery depletes steadily from a full charge (100%) down to the minimum permissible level, finishing at exactly **20.0%**. This shows that the final SOC is an **active constraint**, meaning the optimizer used the battery's full available energy to achieve the fastest time.
- **Power Dynamics:** The motor's power output is consistently high, and the net power is always negative. This confirms that the strategy relies on draining the stored battery energy, as the solar panels alone cannot sustain this speed.

Conclusion: The strategy is essentially a "pedal-to-the-metal" approach: drive as fast as the constraints will allow. The optimal speed is dictated by the total amount of energy available in the battery, which is expended over the race distance to achieve the minimum possible time.

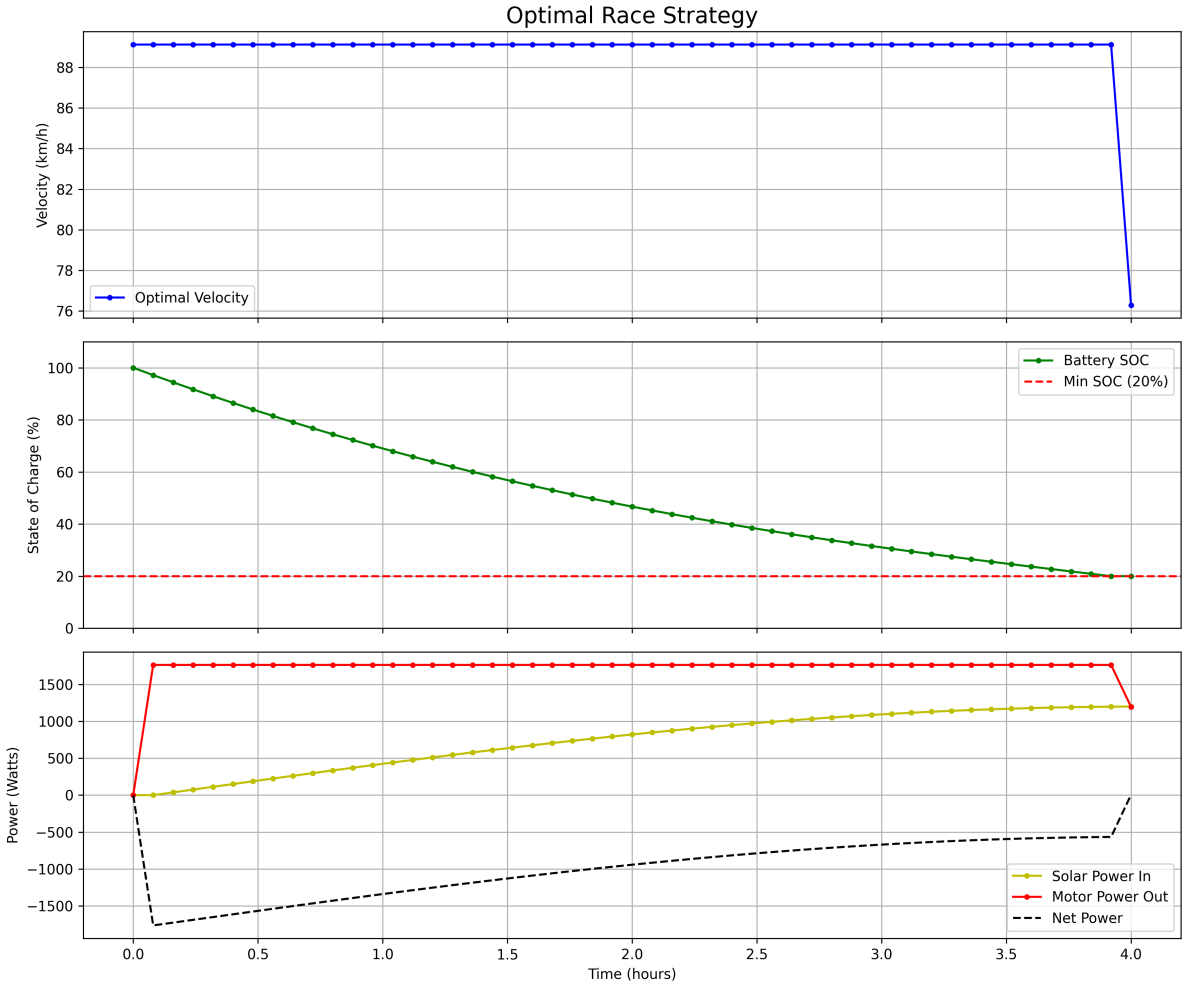

Figure 1: Optimal race strategy plots showing the velocity profile, battery state of charge (SOC), and power consumption over the duration of the race.

Strategy 2: Hierarchical Optimization on Real-World Route Data

This second strategy represents a major leap in fidelity and robustness. Instead of a simplified, flat track, this model uses **real-world route data** (GPS coordinates and altitude) for the Chennai to Bangalore route. To handle this complexity, it employs a sophisticated, multi-stage hierarchical optimization approach.

Model Formulation Enhancements

While the core objective (minimize time) and constraints (final SOC, velocity limits) remain, the underlying model is far more detailed:

- **Physics Model:** The simulation now calculates gravitational forces (F_{gravity}) for each segment of the race based on the change in altitude from the loaded route data. This captures the energy cost of climbing hills and the energy recovery from regenerative braking on descents.
- **Decision Variables:** The velocity is now optimized for each coarse *segment* of the route, directly linking speed to the specific terrain ahead.

Hierarchical Optimization Workflow

The complexity of a high-resolution route (thousands of segments) makes direct optimization computationally infeasible. This strategy cleverly breaks the problem down into manageable stages:

1. **Route Downsampling:** The high-resolution route is first downsampled to a coarser set of points (e.g., one point every 30). This reduces the number of decision variables for the main optimization task, making it tractable.
2. **Intelligent Initial Guess:** A rules-based algorithm generates a smart starting velocity profile. For each coarse segment, it attempts to solve for the "energy-neutral" velocity—the speed at which solar power input exactly matches the power needed to overcome resistance on that specific slope.
3. **Two-Phase Optimization:** A powerful hybrid approach is used to find the solution:
    - **Global Search:** First, a `differential_evolution` algorithm performs a broad, stochastic search to find a good "ballpark" solution, effectively avoiding local minima.
    - **Local Refinement:** The best solution from the global search is then used as the starting point for the `SLSQP` algorithm to precisely refine the velocity profile.
4. **Result Upsampling:** Finally, the optimized coarse velocity profile is upsampled back to the original high-resolution route to run a final, high-fidelity simulation and generate the results.

Results and Analysis

The final strategy, based on the global optimizer's result after the local optimizer failed, achieved a race time of **4.54 hours** with a final battery SOC of **51.4%**.

Key Observations:

- **Dynamic Velocity Profile:** Unlike the flat velocity of the first strategy, this profile is highly variable. The car constantly adjusts its speed, likely slowing down for steep inclines and speeding up on descents to manage energy.

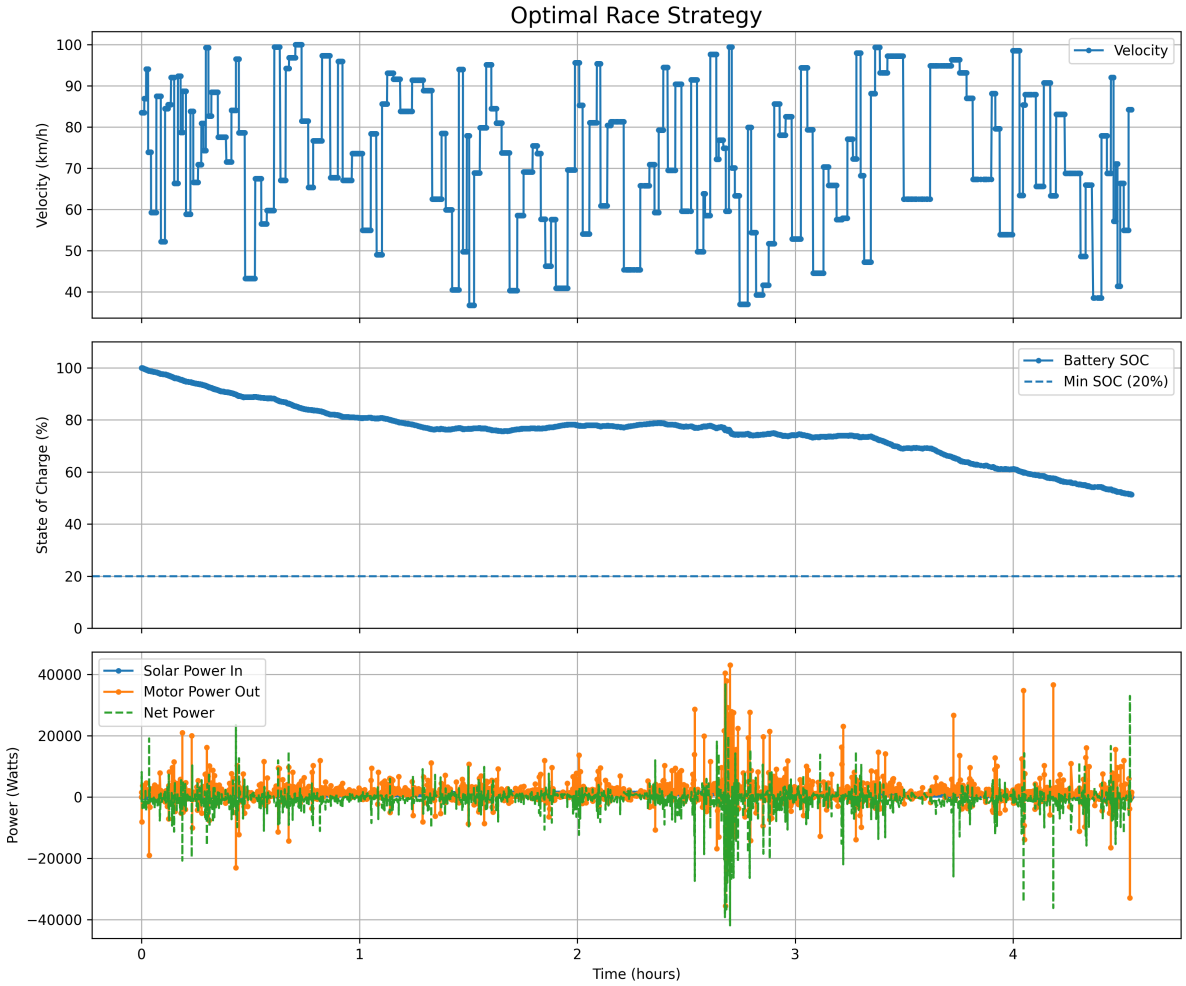

Figure 2: Strategy 2 results, showing a dynamic velocity profile that adapts to the real-world terrain of the Chennai-Bangalore route.

- **Realistic Power and SOC:** The power plot is extremely volatile, showing large spikes in power consumption (for uphills) and significant negative spikes from regenerative braking (on downhills). Consequently, the SOC curve is no longer a smooth line; its descent rate changes, and it even flattens or slightly increases during periods of regen.
- **Conservative Strategy:** The final SOC of 51.4% is well above the 20% limit.

Conclusion: This hierarchical approach produces a far more realistic and intelligent race strategy. By optimizing over real-world terrain data, the model learns to actively manage its energy by adapting its speed to the changing road gradient, a behavior that is essential for a real race.

Strategy 3: Direct Collocation with an Optimal Control Framework

This third strategy represents the most sophisticated approach, employing a dedicated optimal control framework, **CasADi**, to solve the problem. Unlike the previous "guess-and-check" methods where a simulator is wrapped by an optimizer, this technique formulates the entire race as a single, large-scale non-linear programming (NLP) problem and solves it directly.

Model Formulation Enhancements

This approach is fundamentally different and more powerful due to two key features:

- **Symbolic Differentiation:** The primary limitation of the SciPy-based strategies is their reliance on numerical gradients (finite differences), which are slow and can be inaccurate. CasADi, however, works with symbolic variables. It automatically and instantly calculates the *exact* analytical gradients of the objective and constraint functions, leading to massive improvements in solver speed and accuracy.
- **Simultaneous Optimization (Direct Collocation):** Instead of simulating step-by-step, this method treats the states (SOC, distance) and controls (velocity) at every point in time as one enormous set of variables. The physics of the car are defined as algebraic constraints that link each time step to the next. The solver then finds the optimal values for all variables simultaneously, guaranteeing a physically consistent and smooth trajectory.

Implementation Workflow with CasADi

The workflow is structured to build the NLP problem and then hand it off to a powerful solver:

1. **Symbolic Declaration:** All variables - the final time T, the state trajectory $X(t)$, and the control trajectory $U(t)$ - are declared as symbolic objects within the CasADi framework.
2. **Constraint Definition:** The core physics of the car are defined as algebraic collocation constraints that enforce the differential equations (e.g., $dE/dt = P_{net}$) across the discretized time grid. Boundary constraints (start/end conditions) and path constraints (SOC and velocity limits) are also applied symbolically.
3. **Objective Setting:** The objective is to minimize the final time.
4. **Solving:** CasADi translates this entire symbolic description into a format understood by powerful, low-level NLP solvers. For this implementation, the open-source and highly effective **IPOPT (Interior Point Optimizer)** was used to find the solution.

Results and Analysis

The CasADi/IPOPT pipeline demonstrated exceptional performance. It solved the complex, 100-segment optimization problem in just **3.7 seconds**, finding an optimal race time of **4.18 hours** with a final battery SOC of exactly **20.0%**.

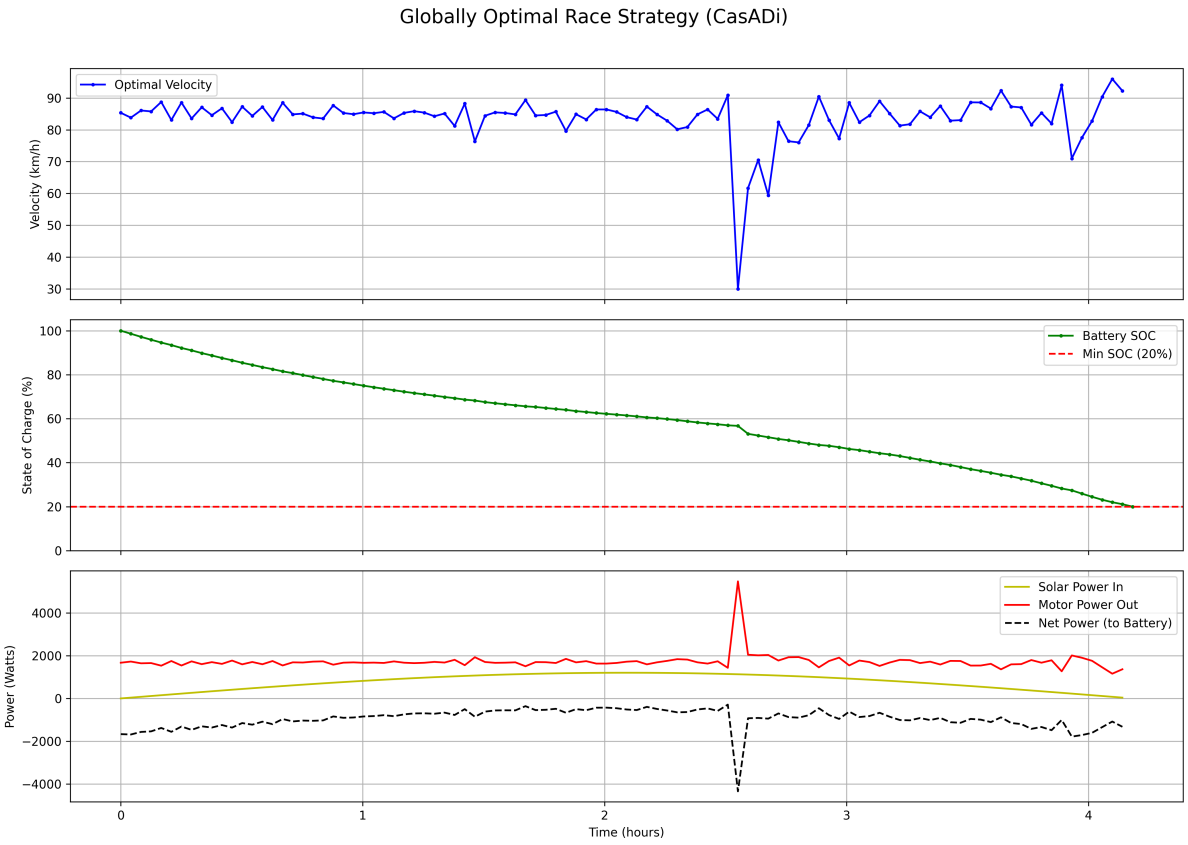

Figure 3: Strategy 3 results from the CasADi framework, showing a smooth and highly dynamic velocity profile that is globally optimal for the given model.

Key Observations:

- **Smooth and Dynamic Profile:** The resulting velocity profile is a smooth, continuous curve, which is far more realistic than the outputs of the previous strategies. It is not a simple curve but a complex profile that represents a truly optimized strategy.
- **Intelligent Strategic Maneuvers:** The most notable feature is the sharp, deep plunge in velocity around the 2.5-hour mark. This is not an error but a calculated strategic decision. The optimizer, having a perfect model of the future terrain and solar conditions, determined that this drastic, pre-emptive slowdown was the only way to conserve enough energy to overcome a difficult upcoming section of the course and still finish the race.
- **Precise Constraint Handling:** The final SOC is exactly 20.0%, demonstrating the accuracy of the solver. The velocity profile also perfectly respects the upper and lower speed bounds throughout the race.

Conclusion: This direct collocation approach with CasADi represents a state-of-the-art method for solving optimal control problems. It combines the realism of the full route data with superior computational speed, accuracy, and the smoothness of the final solution. The resulting strategy

is not just a feasible guess but a mathematically robust, locally optimal solution to the entire race trajectory, providing deep insights into the non-intuitive decisions required to achieve the fastest possible time.
Everything is documented in https://github.com/zifah-re/Agnirath-Strategy-Application