

# Software Architecture & Verification Plan for Adaptive Smoothed-PMP Solver

November 17, 2025

## Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Algorithm Inventory &amp; Traceability</b>                   | <b>2</b>  |
| 1.1      | List of Core Algorithms . . . . .                               | 2         |
| 1.2      | Traceability Summary . . . . .                                  | 5         |
| <b>2</b> | <b>Data &amp; API Specification</b>                             | <b>5</b>  |
| 2.1      | Problem Definition Interface . . . . .                          | 5         |
| 2.1.1    | Dynamics . . . . .  | 5         |
| 2.1.2    | Costs . . . . .   | 6         |
| 2.1.3    | Constraints: Control Set $A$ . . . . .                          | 6         |
| 2.1.4    | State Constraint Set $K$ and Viability . . . . .                | 6         |
| 2.1.5    | Boundary Conditions . . . . .                                   | 7         |
| 2.2      | Hamiltonian and Restricted Hamiltonian Oracles . . . . .        | 7         |
| 2.2.1    | True Hamiltonian . . . . .                                      | 7         |
| 2.2.2    | PA-Bundle Surrogate $\bar{H}$ . . . . .                         | 7         |
| 2.2.3    | Smoothed Hamiltonian $H_\delta$ . . . . .                       | 8         |
| 2.3      | Integrator API (Symplectic Euler and Jacobian Blocks) . . . . . | 8         |
| 2.3.1    | Canonical Residual for One Step . . . . .                       | 8         |
| 2.3.2    | Global Jacobian Assembly . . . . .                              | 9         |
| 2.4      | Shooting / Newton Solver API . . . . .                          | 9         |
| 2.5      | Adaptivity Controller API . . . . .                             | 10        |
| 2.6      | Constraint and Viability Utilities . . . . .                    | 10        |
| 2.7      | Language Mapping and Exceptions . . . . .                       | 10        |
| <b>3</b> | <b>Repository Plan (Parallel Python &amp; MATLAB Structure)</b> | <b>11</b> |
| 3.1      | Python Repository Layout . . . . .                              | 11        |
| 3.2      | MATLAB Repository Layout . . . . .                              | 11        |
| 3.3      | Responsibilities and Dependencies . . . . .                     | 12        |
| <b>4</b> | <b>Numerical Examples: Specifications &amp; Targets</b>         | <b>12</b> |
| 4.1      | Example 1: Linear Quadratic Regulator (LQR) . . . . .           | 13        |
| 4.1.1    | Problem . . . . .   | 13        |
| 4.1.2    | Parameters . . . . .  | 13        |
| 4.1.3    | Expected Behaviour . . . . .                                    | 13        |
| 4.1.4    | Verification Targets . . . . .                                  | 13        |

|       |  |    |
|-------|--|----|
| 4.2   | Example 2: Minimum-Time Double Integrator with Constraints . . . . . | 13 |
| 4.2.1 | Problem . . . . .  | 13 |
| 4.2.2 | Expected Behaviour . . . . .   | 14 |
| 4.2.3 | Verification Targets . . . . .                                       | 14 |
| 4.3   | Example 3: Dubins Car with Bounded Turn Rate . . . . .               | 14 |
| 4.3.1 | Problem . . . . .  | 14 |
| 4.3.2 | Expected Behaviour . . . . .   | 15 |
| 4.3.3 | Verification Targets . . . . .                                       | 15 |

## 1 Algorithm Inventory & Traceability

This section enumerates the key algorithms described in the Manuscript and explains their roles, interfaces, computational complexity, and references. The goal is to ensure that all components proposed in the Manuscript are explicitly mapped to software modules in the Python and MATLAB implementations.

### 1.1 List of Core Algorithms

#### 1. Adaptive Outer Loop (smoothed-PMP)

*Purpose.* Implements the high-level adaptive loop that refines:

- the time grid,
- the PA-bundle (supporting planes for the Hamiltonian),
- the smoothing parameter  $\delta$ .

At each outer iteration it solves the TPBVP via Newton, computes a-posteriori error indicators, and decides which discretisation component to refine next.

*Inputs.*

- Initial time mesh  $\{t_i\}$ .
- Initial PA bundle (initial control planes).
- Initial smoothing parameter  $\delta_0$ .
- Tolerances  $\varepsilon_{\text{time}}, \varepsilon_{\text{PA}}, \varepsilon_{\delta}$ .

*Outputs.*

- Refined state and costate trajectories  $x(t), p(t)$ .
- Control policy  $u(t)$ .
- Final mesh and bundle satisfying all error tolerances.

*Complexity.* Iterative loop. Each outer iteration calls the Newton solver (Algorithm 2.) with cost dominated by solving a sparse linear system; typically only a few outer iterations are required until all indicators fall below tolerances.

*Manuscript reference.* Algorithm 3.1 (p. 5), which uses the error indicators  $\eta_{\text{time}}, \eta_{\text{PA}}, \eta_{\delta}$ .

## 2. Multiple-Shooting Newton (Damped)

*Purpose.* Solves the nonlinear TPBVP obtained from Pontryagin’s Maximum Principle using a multiple-shooting formulation and Newton’s method with damping (Armijo line search). The unknown is the full set of node values for  $x(t)$  and  $p(t)$ .

*Inputs.*

- Current smoothing parameter  $\delta$ .
- Current time mesh and PA bundle.
- Initial guess trajectories  $x^{(0)}(t), p^{(0)}(t)$ .

*Outputs.*

- Converged state and costate trajectories over the mesh.

*Complexity.* Each Newton iteration solves a linear system with Jacobian of size  $\mathcal{O}(2nN)$  (state dimension  $n$ ,  $N$  time steps). The Jacobian is block-banded and solved with sparse linear algebra. A handful of Newton iterations typically suffices.

*Manuscript reference.* Algorithm 3.2 (p. 5). Armijo damping is mentioned inline; the Jacobian structure follows from the symplectic Euler discretisation.

## 3. Symplectic Euler Time-Step

*Purpose.* Single-step integrator for the canonical ODE system (state and costate) using a symplectic Euler scheme. The state is advanced explicitly, while the costate uses an implicit update to preserve the Hamiltonian structure.

*Residual form.* For step size  $\Delta t$ ,

$$r_x = x_{i+1} - x_i - \Delta t \nabla_p H_\delta(p_i, x_i, t_i), \quad r_p = p_{i+1} - p_i + \Delta t \nabla_x H_\delta(p_{i+1}, x_{i+1}, t_{i+1}),$$

which must vanish at the solution.

*Inputs.*  $(x_i, p_i, t_i, \Delta t)$  and the Hamiltonian oracle.

*Outputs.* Residuals and contributions to the global Newton system.

*Complexity.*  $\mathcal{O}(n)$  per step, hence  $\mathcal{O}(nN)$  per trajectory.

*Manuscript reference.* Symplectic Euler is stated as the time discretisation; the time-step error indicator is derived from its residuals.

## 4. PA-Bundle Plane Addition

*Purpose.* Maintains a piecewise-affine surrogate  $\bar{H}$  of the nonsmooth Hamiltonian by adding supporting hyperplanes (planes) corresponding to distinct controls. Each plane refines the surrogate and reduces the model error  $\bar{H} - H$ .

*Inputs.*

- Current bundle with  $M$  planes.
- Trajectory  $x(t), p(t)$ .

*Outputs.*

- Updated bundle with  $M + 1$  planes (or more) improving the approximation.

*Complexity.* For low-dimensional control spaces, adding a plane is cheap: a small minimisation over admissible controls to find a new supporting plane at points of largest gap.

*Manuscript reference.* Appears as the “Add PA planes” step in Algorithm 3.1 and in the discussion of the PA error indicator  $\eta_{\text{PA}} = \int (\bar{H} - H) dt$ .

## 5. Smoothing Function and Gradients

*Purpose.* Replaces the nonsmooth PA surrogate  $\bar{H}$  by a smooth concave approximation  $H_\delta$ , enabling differentiable optimisation (Newton). Typical choice: log-sum-exp soft-min or Moreau–Yosida regularisation.

*Example (log-sum-exp).*

$$H_\delta(p, x, t) = -\delta \log \left( \sum_{i=1}^M \exp(-L_i(p, x, t)/\delta) \right),$$

where  $L_i(p, x, t) = g_i(x, t) \cdot p + d_i(x, t)$ .

*Gradients.* Computed from the soft-min weights; both  $\nabla_p H_\delta$  and  $\nabla_x H_\delta$  are returned by the oracle.

*Inputs.* Bundle planes and current  $\delta > 0$ .

*Outputs.* Value  $H_\delta$  and gradients.

*Complexity.*  $\mathcal{O}(M)$  per evaluation (summing over planes).

*Manuscript reference.* Section 2: smooth concave surrogate converging uniformly to  $\bar{H}$  as  $\delta \rightarrow 0$ .

## 6. State-Constraint Viability ( $H_K$ )

*Purpose.* Enforces state constraints  $x \in K$  by restricting the Hamiltonian minimisation to feasible directions  $f(x, a, t) \in T_K(x)$  (tangent cone). This yields a restricted Hamiltonian  $H_K$  that prevents leaving the feasible set.

*Definition.*

$$H_K(p, x, t) = \min \{ p \cdot f(x, a, t) + \ell(x, a, t) : a \in A, f(x, a, t) \in T_K(x) \}.$$

*Inputs.* Current state, costate and time.

*Outputs.* Value  $H_K$  and the corresponding feasible control.

*Complexity.* Comparable to evaluating the original Hamiltonian, plus viability checks (simple for box constraints).

*Manuscript reference.* Eq. (2.4) (p. 3), defining  $H_K$  and its role in enforcing viability without introducing a free boundary.

## 7. A-Posteriori Error Indicators

*Purpose.* Provide three scalar indicators that drive adaptivity:  $\eta_{\text{time}}$ ,  $\eta_{\text{PA}}$ ,  $\eta_\delta$ .

- $\eta_{\text{time}}$ : time-discretisation error, e.g. based on integration residuals or adjoint error density.

- $\eta_{\text{PA}}$ : error from the Hamiltonian surrogate, approximating  $\int_0^T (\bar{H} - H) dt$ .
- $\eta_\delta$ : smoothing error, approximating  $\int_0^T (H_\delta - \bar{H}) dt$ .

*Inputs.* Converged trajectory  $x(t), p(t)$ , bundle  $\bar{H}$ , and smoothed Hamiltonian  $H_\delta$ .

*Outputs.* Scalars  $\eta_{\text{time}}, \eta_{\text{PA}}, \eta_\delta$ , which are compared against user tolerances and determine refinement actions in Algorithm 3.1.

*Manuscript reference.* Error-estimation logic that informs the adaptive loop and dictates time refinement, plane addition, and smoothing reduction.

## 1.2 Traceability Summary

Algorithms 1–3 implement the main solver loop and the inner Newton solver (Algorithms 3.1 and 3.2 in the Manuscript). Algorithms 4–6 capture the nonsmooth Hamiltonian treatment (PA-bundle, smoothing and state constraints via  $H_K$ ). Algorithm 7 corresponds to the a-posteriori estimates driving adaptivity. The software design will include one module or class for each of these conceptual blocks, ensuring direct traceability from the Manuscript to the code.

## 2 Data & API Specification

This section defines the core data structures and APIs for both the Python and MATLAB implementations. The goal is a unified problem definition and solver interface that mirrors the mathematical formulation.

We distinguish:

- the problem definition interface;
- Hamiltonian and PA-bundle oracles;
- integrator and TPBVP solver APIs;
- the adaptive outer controller;
- constraint and viability utilities.

### 2.1 Problem Definition Interface

The central abstraction is an `OCPPProblem` object describing a particular optimal control problem.

#### 2.1.1 Dynamics

In Python:

```
class OCPPProblem:
    def dynamics(self, x: np.ndarray,
                  a: np.ndarray,
                  t: float) -> np.ndarray:
        """Return state derivative f(x,a,t) as an n-dimensional array."""
```

In MATLAB:

```
% function dx = dynamics(obj, x, a, t)
% Compute f(x,a,t) as an n-by-1 vector.
```

Here  $x \in \mathbb{R}^n$ ,  $a \in A \subset \mathbb{R}^m$ . Units follow the physical meaning of the example at hand.

### 2.1.2 Costs

Running and terminal costs are exposed as:

```
def stage_cost(self, x: np.ndarray,
               a: np.ndarray,
               t: float) -> float:
    """Instantaneous cost (x,a,t)."""

def terminal_cost(self, x_T: np.ndarray,
                  T: float) -> float:
    """Terminal cost g(x_T)."""
```

MATLAB equivalents:

```
% function ell = stage_cost(obj, x, a, t)
% function phi = terminal_cost(obj, x_T, T)
```

If there is no terminal cost, the default can be zero.

### 2.1.3 Constraints: Control Set $A$

Control bounds or more general sets are modelled via helper methods:

```
def control_bounds(self):
    """Return (a_min, a_max) arrays if applicable."""
```

and an admissibility check:

```
def admissible_control(self, a: np.ndarray,
                      x: np.ndarray = None,
                      t: float = None) -> bool:
    """Check whether a ∈ A (possibly depending on x,t)."""
```

In MATLAB this corresponds to properties or functions such as `obj.Amin`, `obj.Amax` or a dedicated validator.

### 2.1.4 State Constraint Set $K$ and Viability

For box constraints we provide

```
def state_bounds(self):
    """Return (x_min, x_max) arrays for box state constraints."""
```

and

```
def admissible_state(self, x: np.ndarray) -> bool:
    """Check whether x ∈ K."""
```

For viability we provide a tangent-cone filter:

```
def tangent_cone_filter(self, x: np.ndarray,
                        f_candidates: np.ndarray) -> np.ndarray:
    """
    Given candidate state velocities (n×M), return indices or a mask
    of those that lie in T_K(x).
    """
```

For box constraints, this checks sign conditions on components that touch their bounds, e.g. at  $x_i = x_{i,\max}$  we require  $f_i \leq 0$ .

### 2.1.5 Boundary Conditions

The problem object holds:

- initial state  $x_0$ ;
- optionally a target state or region;
- horizon  $T$  if fixed (or an initial guess if free).

In Python this can be properties `x0`, `target`, `T`. In MATLAB, fields in a struct or class properties.

## 2.2 Hamiltonian and Restricted Hamiltonian Oracles

### 2.2.1 True Hamiltonian

The true Hamiltonian and its variant with viability are encapsulated as:

```
def compute_H(self, p: np.ndarray,
              x: np.ndarray,
              t: float,
              restricted: bool = False):
    """
    Compute H(p,x,t). If restricted=True, enforce viability and compute H_K.
    Returns (H_value, a_star).
    """
```

MATLAB:

```
% function [Hval, a_star] = compute_H(obj, p, x, t, restricted)
```

For the examples, the control space is low dimensional and we can enumerate candidate controls, apply viability filtering (if restricted) and minimise  $p \cdot f + \ell$ .

### 2.2.2 PA-Bundle Surrogate $\bar{H}$

We represent the piecewise-affine surrogate as a list of planes  $(g_i(x, t), d_i(x, t))$  defining  $L_i(p) = g_i \cdot p + d_i$  and  $\bar{H}(p, x, t) = \min_i L_i(p, x, t)$ .

Key methods:

```

def eval_Hbar(self, p: np.ndarray,
               x: np.ndarray,
               t: float):
    """Return (Hbar_value, active_plane_index)."""

def add_plane(self, a: np.ndarray,
               x: np.ndarray,
               t: float):
    """
    Add a supporting plane corresponding to control a at (x,t).
    Typical choice: g_new = f(x,a), d_new = (x,a).
    """

```

### 2.2.3 Smoothed Hamiltonian $H_\delta$

The smoothed Hamiltonian and its gradients are evaluated via:

```

def eval_H_smooth(self, p: np.ndarray,
                  x: np.ndarray,
                  t: float,
                  delta: float):
    """
    Return (H_delta, grad_p, grad_x) for the chosen smoothing (e.g. log-sum-exp).
    """

```

which implements the soft-min expression, computes soft weights, and forms  $\nabla_p H_\delta$  and (optionally)  $\nabla_x H_\delta$ . Second derivatives can be approximated by finite differences if needed.

## 2.3 Integrator API (Symplectic Euler and Jacobian Blocks)

We provide functions to propagate the canonical system and assemble residuals and Jacobian contributions.

### 2.3.1 Canonical Residual for One Step

```

def canonical_residual(x_i, p_i,
                      x_ip1, p_ip1,
                      t_i, t_ip1,
                      H_oracle, dt):
    """
    Compute residual [r_x; r_p] for a single time interval.
    """

```

with

$$r_x = x_i + dt \nabla_p H_\delta(p_i, x_i, t_i) - x_{i+1}, \quad r_p = p_{i+1} + dt \nabla_x H_\delta(p_{i+1}, x_{i+1}, t_{i+1}) - p_i.$$

### 2.3.2 Global Jacobian Assembly

We assemble the block-banded Jacobian for the full trajectory:

```
def assemble_jacobian(X, P, dt_list, H_oracle):
    """
    Assemble sparse Jacobian of size (2n*N) × (2n*N) for the TPBVP.
    """
```

The blocks encode partial derivatives of  $r_x, r_p$  with respect to  $x_i, p_i, x_{i+1}, p_{i+1}$ , plus final boundary conditions  $p_N + \nabla_x g(x_N) = 0$ .

### 2.4 Shooting / Newton Solver API

The multiple-shooting Newton solver is exposed as:

```
def solve_tpbvp(problem: OCPProblem,
               mesh: np.ndarray,
               bundle,
               delta: float,
               x_init: np.ndarray = None,
               p_init: np.ndarray = None):
    """
    Solve TPBVP for given mesh, PA-bundle and smoothing .
    Return (X, P), arrays of size (N+1, n).
    """
```

In MATLAB:

```
% function [X, P] = solve_tpbvp(prob, mesh, bundle, delta, x_init, p_init)
```

The algorithm:

1. Initialise  $X^{(0)}, P^{(0)}$  from problem data.
2. Form residual vector  $R(X^{(k)}, P^{(k)})$ .
3. Check  $\|R\|_\infty$  against tolerance.
4. Assemble Jacobian  $J(X^{(k)}, P^{(k)})$ .
5. Solve  $J \Delta = -R$ .
6. Perform Armijo line search on  $\Delta$  (with viability checks).
7. Update and repeat until convergence or iteration limit.

On failure (e.g. stagnation) a `NoConvergenceError` (Python) or warning plus flag (MATLAB) is raised and handled by the outer loop.

## 2.5 Adaptivity Controller API

The adaptive outer loop is implemented as:

```
def solve_optimal_control(problem: OCPPProblem,
                          initial_mesh: np.ndarray,
                          tol_time: float,
                          tol_PA: float,
                          tol_delta: float,
                          max_iters: int = 20):
    """
    Adaptive optimal control solver (Algorithm 3.1).
    Returns a dictionary with solution and diagnostics.
    """
```

High-level pseudo-code:

1. Initialise mesh, PA bundle and  $\delta = \delta_0$ .
2. Solve TPBVP for current discretisation.
3. Compute  $\eta_{\text{time}}, \eta_{\text{PA}}, \eta_{\delta}$ .
4. If all indicators  $\leq$  tolerances, stop.
5. If  $\eta_{\text{time}}$  dominates: refine mesh.
6. If  $\eta_{\text{PA}}$  dominates: add planes.
7. If  $\eta_{\delta}$  dominates: reduce  $\delta$ .
8. Repeat until convergence or `max_iters` reached.

The result dictionary includes trajectories, control, cost, final mesh, indicators, and log of iteration diagnostics.

## 2.6 Constraint and Viability Utilities

For box constraints we provide:

```
def enforce_state_bounds(x, x_min, x_max, eps=1e-6):
    """Project x into [x_min, x_max] with small margin eps."""
```

Viability logic is encapsulated in the Hamiltonian computation (via  $H_K$ ). A solver-level switch decides whether to activate viability when states are near the boundary. A post-solve check confirms that  $x(t) \in K$  along the trajectory.

## 2.7 Language Mapping and Exceptions

Python modules:

- `core.problem`: OCPPProblem and specific examples.
- `core.hamiltonian`: true  $H$ ,  $H_K$ , PA bundle, smoothing.

- `core.integrator`: residuals and Jacobian assembly.
- `core.shooting`, `core.newton`: TPBVP set-up and Newton solver.
- `core.adaptivity`: outer loop and error indicators.
- `core.constraints`, `core.utils`: viability helpers, finite-difference checks, logging.

MATLAB mirrors this structure with packages `+core` and `+experiments`. Python uses native exceptions; MATLAB uses `warning/error` codes and flags in result structs.

### 3 Repository Plan (Parallel Python & MATLAB Structure)

The repository is organised to keep a one-to-one correspondence between Python and MATLAB implementations and to separate core solver logic from examples and tests.

#### 3.1 Python Repository Layout

```
python/
src/
  core/
    problem.py          # Problem definition and example subclasses.
    hamiltonian.py      # H, H_K, PA bundle and smoothing.
    integrator.py       # Symplectic Euler, residuals, Jacobian.
    shooting.py         # Multiple-shooting system construction.
    newton.py           # Damped Newton solver.
    adaptivity.py       # Adaptive outer loop (Algorithm 3.1).
    constraints.py      # Viability, state/control checks.
    utils.py            # Utilities (FD checks, logging, etc.).
  experiments/
    ex1_lqr.py
    ex2_double_integrator.py
    ex3_dubins.py
  cli/
    run_example.py
tests/
  test_hamiltonian.py
  test_pa_bundle.py
  test_smoothing.py
  test_integrator.py
  test_newton.py
  test_adaptivity.py
  test_examples.py
README.md
```

#### 3.2 MATLAB Repository Layout

```
matlab/
+core/
```

```

problem.m
hamiltonian.m
pa_bundle.m
integrators.m
shooting.m
newton.m
adaptivity.m
constraints.m
utils.m
+experiments/
  ex1_lqr.m
  ex2_double_integrator.m
  ex3_dubins.m
figs/
README.md

```

### 3.3 Responsibilities and Dependencies

- **problem.py / problem.m:** Define the problem abstraction (dynamics, costs, constraints). Used by Hamiltonian and adaptivity modules and by example scripts.
- **hamiltonian.py / hamiltonian.m:** Implements  $H$ ,  $H_K$ , PA bundle and smoothing. Depends on the problem for  $f$  and  $\ell$  and on constraints for viability.
- **integrator.py / integrators.m:** Time stepping, residuals and Jacobian contributions. Depends on Hamiltonian and optionally problem for Jacobian data.
- **shooting.py / shooting.m:** Assembles global residual and Jacobian for the multiple-shooting system; uses the integrator and Hamiltonian.
- **newton.py / newton.m:** Damped Newton solver using shooting module and constraints for line-search safeguards. Called only by adaptivity.
- **adaptivity.py / adaptivity.m:** Orchestrates the adaptive loop, computes error indicators, refines mesh and bundle and reduces  $\delta$ .
- **constraints.py / constraints.m:** Viability and bound handling, used by Hamiltonian, newton, and adaptivity as needed.
- **utils.py / utils.m:** Non-critical utilities such as finite-difference checks, logging, and plotting helpers.
- **experiments/** scripts set up each example (LQR, double integrator, Dubins car), run the solver, print diagnostics, and generate plots. They serve as integration tests and usage examples.

## 4 Numerical Examples: Specifications & Targets

We implement three benchmark problems corresponding to Examples 1–3 in the Manuscript. For each we specify the setup, expected behaviour and verification targets.

## 4.1 Example 1: Linear Quadratic Regulator (LQR)

### 4.1.1 Problem

Finite-horizon LQR with linear dynamics

$$\dot{x}(t) = Ax(t) + Bu(t), \quad x(0) = x_0,$$

and cost

$$J[u] = x(T)^\top Q_T x(T) + \int_0^T (x(t)^\top Q x(t) + u(t)^\top R u(t)) dt.$$

### 4.1.2 Parameters

- Double-integrator dynamics:  $A = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$ ,  $B = (0, 1)^\top$ .
- Cost weights:  $Q = I_{2 \times 2}$ ,  $R = 10^{-2} I_{1 \times 1}$ , and  $Q_T = Q$  (implicit in the Manuscript).
- Horizon  $T = 1$ .
- Initial state  $x_0 = (1, 0)^\top$ .
- No state or control constraints.

### 4.1.3 Expected Behaviour

The optimal control is a linear feedback  $u^*(t) = -K(t)x(t)$  obtained from a Riccati differential equation. Since control is cheap ( $R = 0.01$ ), the policy aggressively drives the state to zero:  $x_1(t)$  decreases from 1 to 0, and  $x_2(t)$  rises then decays to zero. Both state and control trajectories are smooth.

The PA-bundle degenerates to essentially one active plane (unique control), and smoothing error is negligible, so PA and  $\delta$  refinements are not expected.

### 4.1.4 Verification Targets

- **Cost accuracy.** We compute the analytical optimal cost  $J_{\text{ref}}$  via Riccati and require  $|J - J_{\text{ref}}|/J_{\text{ref}} < 10^{-2}$ .
- **Trajectory agreement.** Plots of  $x_1(t)$ ,  $x_2(t)$ ,  $u(t)$  should visually match the analytical solution.
- **Iteration counts.** Expect  $\leq 2$  outer iterations and roughly 5 Newton steps for convergence.
- **Bundle size.** Final PA bundle size  $M = 1$  (or a small number) with negligible  $\eta_{\text{PA}}$  and  $\eta_\delta$ .

## 4.2 Example 2: Minimum-Time Double Integrator with Constraints

### 4.2.1 Problem

Time-optimal control for the double integrator

$$\dot{x}_1 = x_2, \quad \dot{x}_2 = u,$$

with

- initial state  $x(0) = (-1, 0)^\top$ ,
- target state  $x(T) = (0, 0)^\top$ ,
- control constraint  $u(t) \in [-1, 1]$ ,
- state box constraint (inferred):  $x_1(t) \in [-1, 0]$  (no overshoot beyond 0),  $x_2$  unconstrained.

The objective is to minimise time. In practice, we use a Bolza formulation with running cost  $\ell \equiv 1$  (so  $J \approx T$ ) and a strong penalty on final state deviation (or enforce the final state exactly via transversality).

#### 4.2.2 Expected Behaviour

The classical solution is bang–bang control with one switching time:

$$u^*(t) = \begin{cases} +1, & 0 \leq t < T/2, \\ -1, & T/2 < t \leq T, \end{cases}$$

with minimal time  $T = 2$  (so the switch occurs at  $t = 1$ ). The position  $x_1(t)$  is a piecewise parabolic trajectory from  $-1$  to  $0$ , while the velocity  $x_2(t)$  is piecewise linear (accelerate, then decelerate).

The viability constraint prevents overshoot  $x_1 > 0$ ; the PA bundle should include planes for  $u = \pm 1$ .

#### 4.2.3 Verification Targets

- **Final time.** The numerically inferred final time should be close to 2; we accept  $|T - 2| < 0.1$  (5%).
- **Control pattern.** The reconstructed control  $u(t)$  should have exactly one sign change (one switch) near  $t = 1$ .
- **State constraint.**  $x_1(t) \leq 0$  along the trajectory, with  $x_1(T) \approx 0$ . Violation indicates failure of viability enforcement or adaptivity.
- **Bundle content.** Final PA bundle contains planes for  $u = \pm 1$  (and possibly  $u = 0$ ); gap  $\bar{H} - H$  is small.
- **Adaptivity.** We expect several outer iterations with:
  - PA refinement early (to discover  $u = \pm 1$ );
  - time refinement near the switching point;
  - $\delta$  reduction after time and PA errors are controlled.

### 4.3 Example 3: Dubins Car with Bounded Turn Rate

#### 4.3.1 Problem

Nonholonomic Dubins car model:

$$\dot{x} = v \cos \theta, \quad \dot{y} = v \sin \theta, \quad \dot{\theta} = u,$$

with constant speed  $v = 1$  and control  $u(t) \in [-1, 1]$ . We consider

- initial state  $(x(0), y(0), \theta(0)) = (0, 0, 0)$ ,
- final state  $(x(T), y(T), \theta(T)) = (1, 1, \pi/2)$ ,
- no additional state constraints (other than final condition).

The cost is a soft time and control-effort penalty,

$$J[u] = \int_0^T (\alpha + \beta u(t)^2) dt,$$

with assumed weights, e.g.  $\alpha = 0.1$ ,  $\beta = 1$  (not specified in the Manuscript; we label them as tunable parameters).

### 4.3.2 Expected Behaviour

A minimal Dubins path from the start to the end configuration generally consists of a combination of circular arcs and straight segments, often with one switch in the steering direction. With the chosen cost, we expect:

- the control  $u(t)$  to be close to bang–bang between its bounds  $\pm 1$ , possibly with some smoothing due to the  $u^2$  penalty;
- a single switch in the sign of  $u(t)$ ;
- the trajectory  $(x(t), y(t))$  to approximate a Dubins-like path with total path length around 3–4 (time comparable to this range since  $v = 1$ ).

### 4.3.3 Verification Targets

- **Final state.** The final state should satisfy  $|x(T) - 1|, |y(T) - 1| < 10^{-2}$  and  $|\theta(T) - \pi/2| < 8.7 \times 10^{-3}$  (about  $0.5^\circ$ ).
- **Control structure.** One sign change in  $u(t)$ .
- **Qualitative path.** The path in the plane resembles a reasonable Dubins curve from  $(0, 0)$  to  $(1, 1)$  with heading change from 0 to  $\pi/2$ .
- **Indicators.** Final indicators  $\eta_{\text{time}}, \eta_{\text{PA}}, \eta_\delta < 10^{-3}$ .
- **Bundle and  $\delta$ .** Final bundle includes at least two planes (for  $u = \pm 1$ ), and smoothing parameter  $\delta$  has been reduced so that the smoothed solution is close to the nonsmooth optimal one.