

The Python LevelSet Toolbox (LevelSetPy)

Lekan Molu

<https://github.com/robotsorcerer/levelsetpy>

Abstract—This paper describes open-source scientific contributions in python surrounding the numerical solutions to hyperbolic Hamilton-Jacobi (HJ) partial differential equations viz., their implicit representation on co-dimension one surfaces; dynamics evolution with levelsets; spatial derivatives; total variation diminishing Runge-Kutta integration schemes; and their applications to the theory of reachable sets. They are increasingly finding applications in multiple research domains such as safety-critical problems in reinforcement learning, robotics, control engineering and automation among others. We describe a hierarchy of library components and illustrate usage with a representative numerical example. This GPU-accelerated package allows for easy portability to many modern libraries for the numerical analyses of the HJ-Bellman and HJ-Isaacs equations.

I. OVERVIEW

The *reliability* of the modern automation algorithms that we design has become paramount given the dangers that may evolve if nominally envisioned system performance falters. Even so, the need for scalable and faster numerical algorithms in software for *verification* and *validation* has become timely given the emergence of complexity of contemporary systems. The foremost open-source verification software for engineering applications based on Hamilton-Jacobi (HJ) equations [1, 2] and levelset methods [3, 4] is the CPU-based MATLAB®-implemented levelsets toolbox [5], developed before computing via graphical processing units (GPU) became pervasive. Since then, there has been significant improvements in computer hardware and architecture design, code parallelization algorithms, and compute-acceleration on modern GPUs.

This paper describes a python-based GPU-accelerated scientific software package for numerically resolving generalized discontinuous solutions to Cauchy-type (or time-dependent) HJ hyperbolic partial differential equations (PDEs). HJ PDEs arise in many contexts including (multi-agent) reinforcement learning, robotics, control theory, differential games, flow, and transport phenomena. We focus on the numerical tools for safety assurance (ascertaining the freedom of a system from harm) in a verification sense in this paper. Accompanying the package are implicit calculus operations on dynamic codimension-one interfaces embedded within \mathbb{R}^n surfaces, and spatial and temporal discretization schemes for HJ PDEs. Furthermore, we describe explicit integration schemes including the Lax-Friedrichs, Courant-Friedrichs-Lewy (CFL), and total variation diminishing Runge-Kutta (or TVD-RK) conditioning schemes for

HJ Hamiltonians of the form $\mathbf{H}(\mathbf{x}, \mathbf{p})$, where \mathbf{x} is the state and \mathbf{p} is the co-state. Finally, extensions to reachability analyses for continuous and hybrid systems, formulated as optimal control or game theory problems using viscosity solutions to HJ PDEs are described.

All data transfers to the GPU are based on CuPy [6] framework. In all, we closely follow the Python Enhancement Proposals (PEP) 8 style guide¹; however, in order not to break readability with respect to the original MATLAB®code, we err in consistency with the MATLAB®project layout. The Python package and installation instructions are available on the author’s github repository: [levelsetpy](https://github.com/robotsorcerer/levelsetpy). The CPU implementation (in Python) is on the [cpu-numpy](https://github.com/robotsorcerer/cpu-numpy) tree of the repository. Extensions to other python GPU programming language are straightforward (as detailed in the [CuPy interoperability document](#)). While our emphasis is on the resolution of safe sets in a reachability verification context, the applications of this package extend beyond control engineering.

II. BACKGROUND AND MOTIVATION

Our interest is in the evolution form of the HJ equation

$$\begin{aligned} v_t(\mathbf{x}, t) + \mathbf{H}(t; \mathbf{x}, \nabla_{\mathbf{x}} v) &= 0 \text{ in } \Omega \times (0, T) \\ v(\mathbf{x}, t) &= \mathbf{g}, \text{ on } \partial\Omega \times \{t = T\}, v(\mathbf{x}, 0) = v_0(\mathbf{x}) \text{ in } \Omega \end{aligned} \quad (1)$$

or its convection counterpart

$$\begin{aligned} v_t + \sum_{i=0}^N f_i(u)_{x_i} &= 0, \text{ for } t > 0, \mathbf{x} \in \mathbb{R}^n, \\ v(\mathbf{x}, 0) &= v_0(\mathbf{x}), \mathbf{x} \in \mathbb{R}^n \end{aligned} \quad (2)$$

where \mathbf{x} is the state within an open set, $\Omega \subseteq \mathbb{R}^n$; v_t denotes the partial derivative of the solution v with respect to time t ; the Hamiltonian $\mathbf{H} : (0, T] \times \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}$ and f are continuous functions; \mathbf{g} , and v_0 are bounded and uniformly continuous (BUC) functions in \mathbb{R}^n — assumed to be given; and $\nabla_{\mathbf{x}} v$ (sometimes represented $v_{\mathbf{x}}$) is the spatial gradient of v .

Solving problems described by (1) under appropriate boundary and/or initial conditions using the method of characteristics is limiting as a result of crossing characteristics [7]. In the same vein, global analysis is virtually impossible owing to the lack of existence and uniqueness of solutions $v \in C^1(\Omega) \times (0, T]$ even if \mathbf{H} and \mathbf{g} are smooth [7]. The method of “vanishing viscosity”, based on the idea of traversing the limit as $\delta \rightarrow 0$ in the hyperbolic equation (1)

¹Python PEP 8 style guide: peps.python.org/pep-0008/

allows generalized (discontinuous) solutions [8] whereupon if $\mathbf{v} \in W_{loc}^{1,\infty}(\Omega) \times (0, T]$ and $\mathbf{H} \in W_{loc}^{1,\infty}(\Omega)$, one can lay claim to strong notions of general existence, stability, and uniqueness to BUC solutions \mathbf{v}^δ of the (approximate) viscous Cauchy-type HJ equation

$$\begin{aligned} \mathbf{v}_t^\delta + \mathbf{H}(t; \mathbf{x}, \nabla_{\mathbf{x}} \mathbf{v}^\delta) - \delta \Delta \mathbf{v}^\delta &= 0 \text{ in } \Omega \times (0, T] \\ \mathbf{v}^\delta(\mathbf{x}, t) &= \mathbf{g}, \text{ on } \partial\Omega \times \{t = T\}, \mathbf{v}^\delta(\mathbf{x}, 0) = \mathbf{v}_0(\mathbf{x}) \text{ in } \Omega \end{aligned} \quad (3)$$

in the class $\text{BUC}(\Omega \times [0, T]) \cap C^{2,1}(\Omega \times (0, T])$ i.e. continuous second-order spatial and first order time derivatives for all time $T < \infty$. Crandall and Lions [9] showed that $|\mathbf{v}^\delta(\mathbf{x}, t) - \mathbf{v}(\mathbf{x}, t)| \leq k\sqrt{\delta}$ for a small $k > 0$. Throughout the rest of this paper, we are concerned with *generalized* viscosity solutions of the manner described by (3).

Reachability concerns evaluating the *decidability* of a dynamical system’s evolution of trajectories throughout a state space. Decidable reachable systems are those where one can compute all states that can be reached from an initial condition in *a finite number of steps*. For inf-sup or sup-inf optimal control problems [10], the Hamiltonian is related to the *backward* reachable set [11] of a dynamical system. Mitchell [12] connected techniques used in levelset methods to reachability analysis in optimal control, essentially showing that the zero-levelset of the differential zero-sum two-person game in an HJ-Isaacs (HJI) setting [8, 13] constitutes the safe set of a reachability problem [10]. We refer interested readers on the technicalities of the theory to Mitchell’s paper [12].

The well-known LevelSet Toolbox [5] is the consolidated MATLAB® package that contains the grid methods, boundary conditions, time and spatial derivatives, integrators and helper functions. While Mitchell motivated the execution of the toolkit in MATLAB® based on its expressiveness, modern high-dimensional research and engineering problems often render the original package limiting in computational scalability given its single computer processor implementation, lack of interoperability with many modern computing and scripting languages such as Numpy, Scipy, PyTorch and their variants. In this regard, we revisit the major algorithms necessary for implicit surface representation for HJ PDEs, re-write the spatial, temporal, and monotone difference schemes in Python, and accelerate these schemes on modern GPUs via CuPy [6]. Our **contributions** are as follows:

- 1) we describe the levelset python toolkit, starting with the common implicit surfaces that are used as initial conditions to represent $\mathbf{v}(\mathbf{x}, t)$;
- 2) we describe our implementations of the upwinding spatial derivative, temporal discretization via method of lines schemes based on (approximate) total variation diminishing (TVD) Runge-Kutta (RK), and stabilizing Lax-Friedrichs schemes for multidimensional monotone Hamiltonians of HJ equations or scalar conservation laws;
- 3) we then conclude with a representative example, namely, the barrier surface for two adversarial rockets

traveling on an xz -plane. Further examples with execution clock times abound on the online code repository and in this article’s [journal submission version](#).

The rest of this paper is structured as follows. We describe the geometry of (and Boolean operations on) implicit function representations of continuous-time value functions described by (1) using Cartesian grids in section III. Spatial derivatives to scalar conservation laws are discussed in section IV, and temporal discretization schemes for these conservation laws follow thereafter. In section VI we formulate a didactic two-rockets game and show how to define the numerical safe backward reachable sets and tubes amenable to HJ PDEs in a geometrical verification framework. We conclude the paper in section VII. Additional python examples, jupyter notebooks, and representative problems are provided in the online package.

III. THE LEVELSETPY PYTHON PACKAGE

Let us now describe how solutions to the HJ in (1) and (3) are constructed in our software package.

A. Geometry of Implicit Surfaces and Layouts

As stated before, solutions to the HJ equation (1) are implicitly represented on co-dimension one surfaces in \mathbb{R}^n . We discuss implicit surface functions’ construction in levelsetpy, CPU memory, and GPU transfers. Throughout, links to API’s, routines, and subroutines are hyperlinked and highlighted in [blue text](#) and we use code snippets in Python to illustrate API calls when it’s convenient.

Implicit interfaces are typically isocontours of some function, $f(x)$ — attractive as they require fewer points to construct a function than explicit representations. The zero isocontour (or levelset) of a reachable optimal control problem is equivalent to the safety set or backward reachable tube [14]; and for a differential game, it is the *usable part’s boundary* of the *barrier surface* between the *capture* and *escape zones* for all trajectories that emanate from a system.

B. Grids Layout

Fundamental to implicit surface representations are Cartesian grids in our library. Packages that implement ‘grid’ data structures are in the folder [grids](#). A grid \mathbf{g} is [created](#) by specifying minimum and maximum axes bounds, $[\mathbf{g}_{min}, \mathbf{g}_{max}]$, along every Cartesian coordinate axes n (see lines 3 and 4) of [Listing 1](#); a desired number of discrete points N is passed to the grid data structure — specifying the number of grid nodes and the grid spacing in each dimension as (line 5) listed in [Listing 1](#). On line 7, the grid data structure is constructed and all input parameters to the API are checked for consistency.

```

1 from math import pi
2 import numpy as np
3 gmin = np.array((-5, -5, -pi)) // lower corner
4 gmax = np.array((5, 5, pi)) // upper corner
5 N = 41*ones(3, 1) // number of grid nodes
6 pdim = 3; // periodic boundary condition, dim 3
7 g = createGrid(gmin, gmax, N, pdim)

```

Listing 1: Creating a three-dimensional grid.

A grid [data structure](#), \mathbf{g} , (implemented in [Listing 1](#))

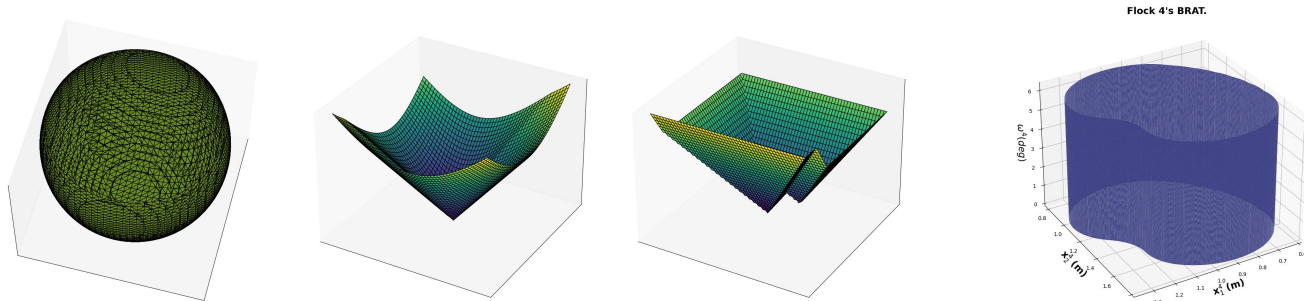


Fig. 1: Implicitly constructed geometric shapes in our library: (a) a sphere on a 3D grid; (b) union of two 3D spheres implicitly constructed on a 2D grid; (c) the union of rectangles on a 2D grid; (d) the union of multiple cylinders on a 3D grid.

has the following fields: (i) discretized nodes of the state(s) \mathbf{x} in (3), denoted as 1-D vectors $\mathbf{g}.vs$; (ii) given the 1-D vectors $\mathbf{g}.vs$, an n -dimensional array of coordinates over n -dimensional grids is computed with matrix-based indexing; this generates a mesh for all state nodal points on the grid $\mathbf{g}.xs$ as a list across all the dimensions of the grid; (iii) grid dimension $\mathbf{g}.dim$, denoting the number of Cartesian axes needed for representing the state \mathbf{x}^2 ; (iv) boundary conditions of the relevant HJ equation to be solved are grafted in by populating the corresponding grid dimension with ghost cells (to be introduced shortly).

C. Implicit Surface Representations: Levelsets

We treat coordinates as functional arguments using a fixed levelset of continuous function $v : \mathbb{R}^n \rightarrow \mathbb{R}$. We use signed distance functions to represent the dynamics throughout. When the signed distance function is not numerically possible, we describe where the implicit surface representations are smeared out in every routine's documentation. The query points for moving interfaces are grid point sets of the computational domain described by implicit geometric primitives such as spheres, cylinders, ellipsoids and even polyhedrons such as icosahedrons. All of these are contained in the folder `initialconditions` on our project page.

The zero levelset of an implicit surface $v(\mathbf{x}, t)$ is defined as $\Gamma = \{\mathbf{x} : v(\mathbf{x}, t) = 0\}$ on a grid $G \in \mathbb{R}^n$, where n denotes the number of dimensions, and the representation of Γ on G generalizes a row-major layout. An example representation of an ellipsoid on a three-dimensional grid is illustrated in Listing 2.

```

1 e = (g.xs[0])**2 // ellipsoid nodal points
2 e += 4.0*(g.xs[1])**2
3 if g.dim==3:
4     \texttt{e} += (9.0*(grid.xs[2])**2)
5 e -= radius // radius=major axis of ellipsoid

```

Listing 2: An `ellipsoid` as a signed distance function.

D. Calculus on Implicit Function Representations

Geometrical operations on implicitly defined functions carry through in the package as follows. Let $v_1(\mathbf{x})$ and $v_2(\mathbf{x})$ be two signed distance representations, then the union of

the interior of both is simply $\min(v_1(\mathbf{x}), v_2(\mathbf{x}))$ (illustrated in Fig. 1b and d). The intersection of two signed distance functions' interiors is generated by $\max(v_1(\mathbf{x}), v_2(\mathbf{x}))$ (illustrated in Fig. 1c). The complement of a function is found by negating its signed distance function i.e. $-v(\mathbf{x})$. The resultant function as a result of the subtraction of the interior of one signed distance function v_2 from the another one, say, v_1 is defined $\max(v_1(\mathbf{x}), -v_2(\mathbf{x}))$. All of these are implemented in the module `shapeOps`.

IV. SPATIAL DISCRETIZATION: UPWINDING

In this section, we discuss higher-order upwinding schemes that mimic high-order essentially non-oscillatory (ENO) [15] schemes for computing the spatial derivatives v_x for the numerical viscosity solutions to levelset PDEs of the *Eulerian form* introduced in (4). Routines for procedures herewith described are in the folder `spatialderivative`. Using the Eulerian form of the levelset equation,

$$v_t + \mathbf{F} \cdot \nabla v = 0 \quad (4)$$

where \mathbf{F} is the speed function, the implicit function representation v_t (see §III) is used both to denote and evolve the interface. Suppose that the interface speed \mathbf{F} is a three-vector $[f_x, f_y, f_z]$ on a three-dimensional Cartesian grid, expanding (4) the evolution of the implicit function on the zero levelset yields the Eulerian form

$$v_t + f_x v_x + f_y v_y + f_z v_z = 0 \quad (5)$$

of the interface evolution given that the interface encapsulates the implicit representation v . In our implementation, we define v throughout the computational domain Ω .

Let us first construct the general form of a spatial upwinding scheme i.e.

$$D^- v = \frac{\partial v}{\partial x} \approx \frac{v_{i+1} - v_i}{\Delta x}, D^+ v \approx \frac{v_i - v_{i-1}}{\Delta x}, \quad (6)$$

where v and its speed \mathbf{F} are defined over a domain Ω (this is the Cartesian grid in our representation). Using the forward Euler method, the levelset equation (5) becomes $(1/\Delta t) \cdot v^{n+1} - v^n + f_x^n v_x^n + f_y^n v_y^n + f_z^n v_z^n = 0$.

Now, suppose that we are on a one-dimensional surface and around a grid point i . Given that f^n may be spatially

²This parameter is useful when computing signed distance to every nodal point on the state space in the implicit representation of v

varying, the foregoing equation becomes

$$\frac{v_i^{n+1} - v_i^n}{\Delta t} + f_i^n(v_x)_i^n = 0 \quad (7)$$

where $(v_x)_i$ denotes the spatial derivative of v w.r.t x at the point i . We now discuss the specific implementations of the upwinding schemes.

A. First-order accurate upwinding discretization

If $f_i > 0$ in (7), the values of v are traversing from left to right so that in order to update v at the end of the next time step, we must look to the left (going by the method of characteristics [4, §3.1]) and vice versa if $f_i < 0$. We therefore follow the standard upwinding method by using (6): we approximate v_x with D^-v whenever $f_i > 0$ and we approximate v_x with D^+v whenever $f_i < 0$. No approximation is needed when $f_i = 0$ since $f_i(v_x)_i$ vanishes. This scheme is accurate within $O(\Delta x)$ given the first order accurate approximations D^-v and D^+v . We have followed the naming convention in [5] and in our `spatialderivative` folder, we name this function `upwindFirstFirst`.

B. ENO Polynomial Interpolation of Solutions

Using a divided differencing table, essentially non-oscillatory (ENO) polynomial interpolation of the discretization [15] of the levelset equation is known to generate improved numerical approximations to D^-v and D^+v . Suppose that we choose a uniform mesh discretization Δx . Define the zeroth divided differences of v at the grid nodes i as $D_i^0 v = v_i$, and the first, second, and third order divided differences of v as the midway between grid nodes i.e.

$$D_{i+1/2}^1 v = \frac{D_{i+1}^0 v - D_i^0 v}{\Delta x}, \quad D_i^2 v = \frac{D_{i+1/2}^1 v - D_{i-1/2}^1 v}{2\Delta x}, \quad (8a)$$

$$D_{i+1/2}^3 v = \frac{D_{i+1}^2 v - D_i^2 v}{3\Delta x}. \quad (8b)$$

Then, an essentially non-oscillating polynomial of the form

$$v(x) = Q_0(x) + Q_1(x) + Q_2(x) + Q_3(x) \quad (9)$$

can be constructed. In this light, the backward and forward spatial derivatives of v w.r.t x at grid node i is found in terms of the derivatives of the coefficients $Q_i(x)$ in the foregoing i.e.

$$v_x(x_i) = Q_1'(x_i) + Q_2'(x_i) + Q_3'(x_i). \quad (10)$$

Define $k = i - 1$ and $k = i$ for v_x^- and v_x^+ respectively. Then the first order (i.e. first-order upwinding) accurate polynomial interpolation is essentially

$$Q_1(x) = (D_{k+1/2}^1 v)(x - x_i), \quad Q_1'(x_i) = D_{k+1/2}^1 v. \quad (11)$$

We follow Osher and Fedkiw's recommendation [4] in avoiding interpolating near large oscillations in gradients. Therefore, we choose a constant c such that

$$c = \begin{cases} D_k^2 v & \text{if } |D_k^2 v| \leq |D_{k+1}^2 v| \\ D_{k+1}^2 v & \text{otherwise} \end{cases} \quad (12)$$

so that

$$Q_2(x) = c(x - x_k)(x - x_{k+1}), \quad Q_2'(x_i) = c(2i - 2k - 1)\Delta x$$

is the second-order accurate upwinding solution for the polynomial interpolation. This is implemented as `upwindFirstENO2` in the `spatialderivative` folder.

To obtain a third-order accurate solution, we choose c^* as follows

$$c^* = \begin{cases} D_{k^*+1/2}^3 v & \text{if } |D_{k^*+1/2}^3 v| \leq |D_{k^*+3/2}^3 v| \\ D_{k^*+3/2}^3 v & \text{if } |D_{k^*+1/2}^3 v| > |D_{k^*+3/2}^3 v|. \end{cases} \quad (13)$$

Whence, we have

$$Q_3(x) = c^*(x - x_{k^*})(x - x_{k^*+1})(x - x_{k^*+2}) \quad (14a)$$

$$Q_3'(x_i) = c^*(3(i - k^*)^2 - 6(i - k^*) + 2)(\Delta x)^2 \quad (14b)$$

for the third-order accurate correction to the approximated upwinding scheme (9). This is implemented as a routine in `upwindFirstENO3aHelper` and called as `upwindFirstENO3` in the `spatialderivative` folder.

C. HJ Weighted Essentially Nonoscillatory Solutions

Here, we focus on weighted ENO (WENO) schemes with the same stencil as the third-order ENO scheme; however, its accuracy reaches up to fifth-order in the solution's smooth parts. Our results closely follow the presentation of Jiang and Peng in [16]. These WENO schemes approximate spatial derivatives at integer grid points as opposed to at half-integer grid values as we did in the ENO scheme of the previous section.

The third-order accurate ENO scheme essentially employs one of three substencils on a grid, namely $\{i-3, i-2, \dots, i\}$, $\{i-2, i-1, \dots, i+1\}$, and $\{i-1, \dots, i+3\}$ on the stencils range $\{i-3, i-2, \dots, i+3\}$.

$$v_{x,i}^{-,0} = \frac{1}{3}D^+v_{i-3} - \frac{7}{6}D^+v_{i-2} + \frac{11}{6}D^+v_{i-1} \quad (15a)$$

$$v_{x,i}^{-,1} = -\frac{1}{6}D^+v_{i-2} + \frac{5}{6}D^+v_{i-1} + \frac{1}{3}D^+v_i \quad (15b)$$

$$v_{x,i}^{-,2} = -\frac{1}{3}D^+v_{i-1} + \frac{5}{6}D^+v_i - \frac{1}{6}D^+v_{i+1} \quad (15c)$$

Suppose that the spatial derivative v_x is to be found using the left-leaning substencil: $\{i-3, i-2, \dots, i\}$, then the third-order ENO scheme chooses one from (15) where $v_{x,i}^{-,p}$ denotes the third-order p 'th substencil to $v_x(x_i)$ for $p = 0, 1, 2$. The WENO approximation to $v_x(x_i)$ leverages a convex weighted average of the three substencils so that

$$v_{x,i}^- = w_0 v_{x,i}^{-,0} + w_1 v_{x,i}^{-,1} + w_2 v_{x,i}^{-,2}. \quad (16)$$

In smooth regions of the phase space, $w_0 = 0.1$, $w_1 = 0.6$, and $w_2 = 0.3$ yield the optimally accurate fifth order WENO approximation, we have for $v_{x,i}^-$

$$\frac{D^+}{30}v_{i-3} - \frac{13}{60}D^+v_{i-2} + \frac{47}{60}D^+v_{i-1} + \frac{9}{20}D^+v_i - \frac{D^+}{20}v_{i+1}$$

the fifth-order approximation $\mathbf{v}_x(\mathbf{x}_i)$ and provides the smallest truncation error on a six-point stencil.

To account for weights in non-smooth regions, however, the smoothness of the stencils (15) can be estimated as recommended in [4, §3.4] so that if

$$\alpha_1 = 0.1/(\sigma_1 + \epsilon)^2, \alpha_2 = 0.6/(\sigma_2 + \epsilon)^2, \alpha_3 = 0.1/(\sigma_3 + \epsilon)^2 \quad (17)$$

for

$$\sigma_1 = \frac{13}{12}(D^+ \mathbf{v}_{i-3} - 2D^+ \mathbf{v}_{i-2} + D^+ \mathbf{v}_{i-1})^2 + \frac{1}{4}(D^+ \mathbf{v}_{i-3} - 4D^+ \mathbf{v}_{i-2} + 3D^+ \mathbf{v}_{i-1})^2, \quad (18a)$$

$$\sigma_2 = \frac{13}{12}(D^+ \mathbf{v}_{i-2} - 2D^+ \mathbf{v}_{i-3} + D^+ \mathbf{v}_i)^2 + \frac{1}{4}(D^+ \mathbf{v}_{i-2} - D^+ \mathbf{v}_i)^2, \quad (18b)$$

$$\sigma_3 = \frac{13}{12}(D^+ \mathbf{v}_{i-1} - 2D^+ \mathbf{v}_i + D^+ \mathbf{v}_{i+1})^2 + \frac{1}{4}(3D^+ \mathbf{v}_{i-1} - 4D^+ \mathbf{v}_i + D^+ \mathbf{v}_{i+1})^2, \quad (18c)$$

and

$$\epsilon = 10^{-6} \max\{D^+ \mathbf{v}_{i-3}, D^+ \mathbf{v}_{i-2}, D^+ \mathbf{v}_{i-1}, D^+ \mathbf{v}_i, D^+ \mathbf{v}_{i+1}\} + 10^{-99} \quad (19)$$

then, we may define the weights for the WENO scheme as

$$w_1 = \alpha_1 / \sum_{i=1}^3 \alpha_i, w_2 = \alpha_2 / \sum_{i=1}^3 \alpha_i, w_3 = \alpha_3 / \sum_{i=1}^3 \alpha_i. \quad (20)$$

This approximates the optimal weights $w_0 = 0.1$, $w_1 = 0.6$ and $w_2 = 0.3$ for decently smooth σ_k that can be dominated by ϵ . Our implementation is the routine `upwindFirstWENO5a` which can be called as `upwindFirstWENO5`.

D. Lax-Friedrichs Monotone Difference Schemes

We now describe a convergent monotone difference spatial approximation scheme for scalar conservation laws of the form

$$\mathbf{v}_t + \sum_{i=1}^N f_i(\mathbf{v})_{x_i} = 0 \text{ for } t > 0, \mathbf{x} = (x_1, \dots, x_N) \in \mathbb{R}^n$$

$$\mathbf{v}(\mathbf{x}, 0) = \mathbf{v}_0(\mathbf{x}), \text{ for } \mathbf{x} \in \mathbb{R}^n \quad (21)$$

Suppose that $N = 1$, let us define $\lambda_x = \Delta t / \Delta x$, $\Delta_x^+ = \mathbf{v}_{j+1} - \mathbf{v}_j$, and $\Delta_x^- = \mathbf{v}_j - \mathbf{v}_{j-1}$. Then at the n th time step, the Lax-Friedrichs scheme is [17]

$$\mathbf{v}_j^{n+1} = \mathbf{v}_j^n - \frac{\lambda_x}{2} \Delta_x^0 f(\mathbf{v}_j^n) + \frac{1}{2} \Delta_x^+ \Delta_x^- \mathbf{v}_j^n. \quad (22)$$

Furthermore, if we define the flux on the state space as

$$g(\mathbf{v}_j, \mathbf{v}_{j-1}) = \frac{f(\mathbf{v}_j) + f(\mathbf{v}_{j-1})}{2} - \frac{1}{2} \lambda_x (\mathbf{v}_j - \mathbf{v}_{j-1}), \quad (23)$$

we may write $\mathbf{v}_j^{n+1} = \mathbf{v}_j^n - \lambda_x^+ g(\mathbf{v}_j, \mathbf{v}_{j-1})$.

The Lax-Friedrichs scheme is monotone on the interval $[a, b]$ if the CFL condition $\lambda_x \max_{a \leq v \leq b} |f'(v)| \leq 1$ for

$(a, b) > 0$ and the upwind differencing scheme for a nondecreasing f is $\mathbf{v}_j^{n+1} = \mathbf{v}_j^n - \lambda_x \Delta_x^+ f(\mathbf{v}_{j-1}^n)$. For a non-increasing f , we have $\mathbf{v}_j^{n+1} = \mathbf{v}_j^n - \lambda_x \Delta_x^+ f(\mathbf{v}_j^n)$. Our Lax-Friedrichs implementation is implemented in the `explicitintegration/term` folder.

V. TEMPORAL DISCRETIZATION: METHOD OF LINES

Here, we describe further improvements on the numerical derivatives of HJ equations by further improving the fifth order accurate HJ WENO schemes presented in section IV. We adopt the *method of lines* (MOL) used in converting the time-dependent PDEs to ODEs. Our presentation follows the total variation diminishing (TVD) Runge Kutta (RK) schemes with Courant-Friedrichs-Lewy (CFL) conditioning imposed for stability as presented in [18] and implemented in MATLAB® in [5].

A. Higher-Order TVD-RK Time Discretizations

To adopt the method of lines, the N -dimensional levelset representation of \mathbf{v} is first rolled into a 1-D vector and an adaptive integration step size, Δt , is chosen to guarantee stability following the recommendation in [19]. The forward Euler algorithm thus becomes

$$\mathbf{v}(\mathbf{x}, t + \Delta t) = \mathbf{v}(\mathbf{x}, t) + \Delta t \Upsilon(\mathbf{x}, \mathbf{v}(\mathbf{x}, t)) \quad (24)$$

where Υ is now the function to be integrated.

A standard MOL can then be applied for the integration similar to ODEs (we have followed Mitchell's [5] code layout to provide consistency for MATLAB users). We implement TVD-RK MOL schemes up to third-order accurate forward Euler integration schemes and the calling signature is as described in Listing 3.

```
odeCFLx(schemeFunc, tspan, y0, options, schemeData)
```

Listing 3: CFL-constrained method of lines routines.

where x could be one of 1, 2, or 3 to indicate one of first-order, second-order, or third-order accurate TVD-RK scheme. The routine `schemeFunc` is typically one of the Lax-Friedrichs approximation routines (implemented as `termLaxFriedrichs`) in the folder `explicitintegration/term`. It approximates the HJ equation based on dissipation functions (shortly introduced).

The first-order accurate TVD (it is total variation bounded [TVB] actually) together with the spatial discretization used for the PDE is equivalent to the forward Euler method. We implement this as `odeCFL1`.

The second-order accurate TVD-RK scheme follows the RK scheme by evolving the Euler step to $t^n + \Delta t$,

$$\frac{\mathbf{v}^{n+1} - \mathbf{v}^n}{\Delta t} + F^n \cdot \nabla \mathbf{v}^n = 0. \quad (25)$$

A following Euler step to $t^n + 2\Delta t$ follows such that

$$\frac{\mathbf{v}^{n+2} - \mathbf{v}^{n+1}}{\Delta t} + F^{n+1} \cdot \nabla \mathbf{v}^{n+1} = 0 \quad (26)$$

before a convex combination of the initial value function and the result of the preceding Euler steps is taken in

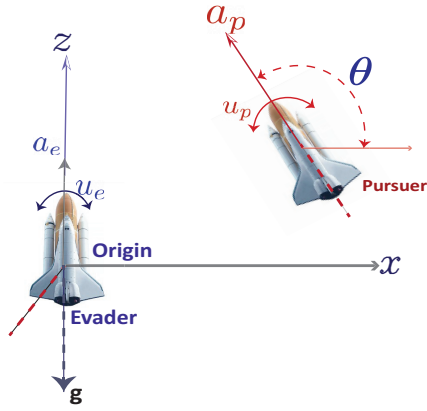


Fig. 2: Motion of two rockets on a Cartesian xz -plane with a thrust inclination in relative coordinates given by $\theta := u_p - u_e$.

the following averaging step, $\mathbf{v}^{n+1} = \frac{1}{2}\{\mathbf{v}^n + \mathbf{v}^{n+2}\}$. The equation in the foregoing produces the second-order accurate TVD approximation to \mathbf{v} at $t^n + \Delta t$, implemented as `odeCFL2`.

With the third-order accurate TVD-RK scheme, the first two advancements in forward Euler schemes are computed but with a different averaging scheme, $\mathbf{v}^{n+1/2} = \frac{1}{4}\{3\mathbf{v}^n + \mathbf{v}^{n+2}\}$ which averages the previous two solutions at $t^n + \frac{1}{2}\Delta t$. The third Euler advancement step to $t^n + \frac{3}{2}\Delta t$ is

$$\frac{\mathbf{v}^{n+\frac{3}{2}} - \mathbf{v}^{n+\frac{1}{2}}}{\Delta t} + F^{n+\frac{1}{2}} \cdot \nabla \mathbf{v}^{n+\frac{1}{2}} = 0 \quad (27)$$

together with the averaging scheme, $\mathbf{v}^{n+1} = \frac{1}{3}\{\mathbf{v}^n + 2\mathbf{v}^{n+\frac{3}{2}}\}$ to produce a third-order accurate approximation to \mathbf{v} at time $t^n + \Delta t$, implemented as `odeCFL3`.

VI. NUMERICAL VALIDATION

In this section, we provide a representative problem and amend it to an HJ PDE form that can be resolved with our toolbox. We consider a *collection/family of differential games*, $\Upsilon = \{\Gamma_1, \dots, \Gamma_g\}$, where each game may be characterized as a pursuit-evasion game, Γ . Each player in a game shall constitute either a pursuer (\mathbf{P}) or an evader (\mathbf{E}) and such a game terminates when *capture* occurs.

A. The Rockets Launch Problem

We consider the rocket launch problem of Dreyfus [20] and amend it to a differential game between two identical rockets, \mathbf{P} and \mathbf{E} , on an (x, z) cross-section of a Cartesian plane. We want to compute the backward reachable tube (BRT) [12] of the *approximate* terminal surface's boundary for a predefined target set over a time horizon (i.e. the target tube). The BRT entails the state-space regions for which min-max operations over either *strategy* of \mathbf{P} or \mathbf{E} is below zero, and where the *variational HJI PDE* is exactly zero.

For a two-player differential game, let \mathbf{P} and \mathbf{E} share identical dynamics in a general sense so that we can freely

choose the coordinates of \mathbf{P} ; however, \mathbf{E} 's origin is a distance ϕ away from (x, z) at plane's origin (see Fig. 2) so that the \mathbf{PE} vector's inclination measured counterclockwise from the x axis is θ .

Let the states of \mathbf{P} and \mathbf{E} be denoted by $(\mathbf{x}_p, \mathbf{x}_e)$. Furthermore, let the \mathbf{P} and \mathbf{E} rockets be driven by their thrusts, denoted by (u_p, u_e) respectively (see Figure 2). Fix the rockets' range so that what is left of the motion of either \mathbf{P} or \mathbf{E} 's is restricted to orientation on the (x, z) plane as illustrated in Fig. 2. It follows that the relevant *kinematic equations* (KE) (derived off Dreyfus' [20] single rocket dynamics) is

$$\dot{x}_{2e} = x_{4e}; \quad \dot{x}_{2p} = x_{4p}, \quad (28a)$$

$$\dot{x}_{4e} = a \sin u_e - g; \quad \dot{x}_{4p} = a \sin u_p - g \quad (28b)$$

where a and g are respectively the acceleration and gravitational accelerations (in feet per square second)³.

As long as \mathbf{E} remains within this target region or *backward reachable tube* (or BRT), \mathbf{P} cannot cause damage or exercise an action of deleterious consequence on, say, the territory being guarded by \mathbf{E} . Setting up \mathbf{E} to maximize a payoff quantity with the largest possible margin or at least frustrate the efforts of \mathbf{P} with minimal collateral damage while the pursuer minimizes this quantity constitutes a terminal value *optimal* differential game: there is no optimal pursuit without an optimal evasion.

\mathbf{P} 's motion relative to \mathbf{E} 's along the (x, z) plane includes the relative orientation, the control input, shown in Fig. 2 as $\theta = u_p - u_e$. Following the conventions in Fig. 2, the game's relative equations of motion in *reduced space* [13, §2.2] is $\mathbf{x} = (x, z, \theta)$ where $\theta \in [-\frac{\pi}{2}, \frac{\pi}{2})$ and $(x, z) \in \mathbb{R}^2$ are

$$\dot{\mathbf{x}} = \begin{cases} \dot{x} &= a_p \cos \theta + u_e x, \\ \dot{z} &= a_p \sin \theta + a_e + u_e x - g, \\ \dot{\theta} &= u_p - u_e. \end{cases} \quad (29)$$

The capture radius of the origin-centered circle ϕ (we set $\phi = 1.5$ ft) is $\|\mathbf{PE}\|_2$ so that $\phi^2 = x^2 + z^2$. All capture points are specified by the variational HJ PDE [12]:

$$\frac{\partial \phi}{\partial t}(\mathbf{x}, t) + \min \left[0, \mathbf{H} \left(\mathbf{x}, \frac{\partial \phi(\mathbf{x}, t)}{\partial \mathbf{x}} \right) \right] \leq 0, \quad (30)$$

with Hamiltonian given by

$$\mathbf{H}(\mathbf{x}, p) = - \max_{u_e \in [\underline{u}_e, \bar{u}_e]} \min_{u_p \in [\underline{u}_p, \bar{u}_p]} \begin{bmatrix} p_1 & p_2 & p_3 \\ a_p \cos \theta + u_e x \\ a_p \sin \theta + a_e + u_p x - g \\ u_p - u_e \end{bmatrix}. \quad (31)$$

Here, p are the co-states, and $[\underline{u}_e, \bar{u}_e]$ denotes extremals that the evader must choose as input in response to the extremal controls that the pursuer plays i.e. $[\underline{u}_p, \bar{u}_p]$. Rather than resort to analytical *geometric reasoning*, we may analyze possibilities of behavior by either agent via a principled numerical simulation. This is the essence of this work. From

³We set $a = 1 \text{ ft/sec}^2$ and $g = 32 \text{ ft/sec}^2$ in our simulation.

(31), set $\underline{u}_e = \underline{u}_p = \underline{u} \triangleq -1$ and $\bar{u}_p = \bar{u}_e = \bar{u} \triangleq +1$ so that $\mathbf{H}(\mathbf{x}, p)$ is

$$\begin{aligned} & - \max_{u_e \in [\underline{u}_e, \bar{u}_e]} \min_{u_p \in [\underline{u}_p, \bar{u}_p]} \begin{bmatrix} p_1(a_p \cos \theta + u_e x) + \\ p_2(a_p \sin \theta + a_e + \\ u_p x - g) + p_3(u_p - u_e) \end{bmatrix}, \\ & \triangleq -ap_1 \cos \theta - p_2(g - a - a \sin \theta) - \bar{u}|p_1 x + p_3| \\ & \quad + \underline{u}|p_2 x + p_3|, \end{aligned} \quad (32)$$

where the last line follows from setting $a_e = a_p \triangleq a$.

For the target set guarded by \mathbf{E} , we choose an implicitly constructed cylindrical mesh on a three-dimensional grid. The grid's nodes are uniformly spaced apart at a resolution of 100 points per dimension over the interval $[-64, 64]$. In numerically solving for the Hamiltonian (32), a TVD-RK discretization scheme [15] based on fluxes is used in choosing smooth nonoscillatory results as described in §V. Denote by (x, y, z) a generic point in \mathbb{R}^3 so that given mesh sizes $\Delta x, \Delta y, \Delta z, \Delta t > 0$, letters u, v, w represent functions on the x, y, z lattice: $\Delta = \{(x_i, y_j, z_k) : i, j, k \in \mathbb{Z}\}$.

```

1 finite_diff_data = {"innerFunc": termLaxFriedrichs,
2 "innerData": {"grid": g, "hamFunc": rocket_rel.ham,
3 "partialFunc": rocket_rel.dissipation,
4 "dissFunc": artificialDissipationGLF,
5 "CoStateCalc": upwindFirstENO2},
6 "positive": True} // direction of approx. growth

```

Listing 4: HJ ENO2 computational scheme for the rockets.

The Hamiltonian, upwinding scheme, flux dissipation method, and the overapproximation parameter for the ENO polynomial interpolatory data used in geometrically reasoning about the *target tube* is as seen in Listing 4. The data structure `finite_diff_data` contains all the routines needed for adding dynamics to the original implicit surface representation of $v(\mathbf{x}, t)$. The monotone spatial upwinding scheme used (here `termLaxFriedrichs` described in §IV-D) is passed into the `innerFunc` query field. The explicit form of the Hamiltonian (see (32)) is passed to the `hamFunc` query field, and the grid is passed to the `grid` field. We adopt a second-order accurate upwinding scheme together with the a `Lax-Friedrichs` conditioner for numerical stability. To indicate that we intend to overapproximate the value function, we specify a `True` parameter for the `positive` query field.

Using our GPU-accelerated toolbox, we compute the *overapproximated* BRT of the game over a time span of $[-2.5, 0]$ seconds during 11 global optimization time steps (the global steps constitute the time-horizon over which the BRT is computed). The initial value function (leftmost inset of Fig. 3) is represented as a (closed) dynamic implicit surface over all point sets in the state space (using a signed distance function) for a coordinate-aligned cylinder whose vertical axes runs parallel to the orientation of the rockets depicted in Fig. 2. The two middle capture surfaces indicate the evolution of the capture surface (here the zero levelset) of the target set upon the optimal response of the evader to the pursuer. We reach convergence at the eleventh global

optimization timestep (rightmost inset of Fig. 3). The BRTs at representative time steps in the optimization procedure is depicted in Fig. 3.

Reachability [10, 14] thus affords us an ability to numerically reason about the behavior of these two rockets aforesaid without closed-form geometrical analysis. To do this, we have passed relevant parameters to the package as shown in Listing 4 and run a CFL constrained optimization scheme (as in Listing 3) for a finite number of global optimization timesteps.

VII. COMPARISONS AND CONCLUSION

We compare evaluation times among our GPU-implementation, Mitchell's [5], and our Numpy CPU implementation for various problems. We refer readers to detailed problem description in this article's [journal submission version](#). Table I depicts the time it takes to run the TVD-RK scheme for other reachable problems solved with our library in comparison to Mitchell's toolbox [5] and other implementations. The column `Avg. local` is the average time of running one single step of the TVD-RK scheme (`crefsec:temporal`) while the `Global` column denotes the average time to compute the full TVD-RK solution to the HJ PDE. Each time query field represents an average over 20 experiments. Computation is significantly faster with our GPU implementation in all categories. In `Air3D` and the rockets launch problems, the average local time for computing the solutions to the stagewise HJ PDEs is an improvement of $\sim 76\%$; the global time is a gain of 76.09% over Mitchell's [5]'s `MATLAB` scheme. Similarly, substantial computational gains are achieved for the two rockets differential game problem: 89% faster global optimization time and 88.62% average local computational time compared to our CPU implementations in `Numpy`. For the rockets game, we notice a speedup of almost 92% in global optimization with the GPU library versus an 89.32% gain using our CPU-`NUMPY` library. Notice the exception in the `Double Integrator` experiment, however: local and global computations take a little longer compared to deployments on the `Numpy CPU` implementation and [5]'s native `MATLAB@toolbox`. We attribute this to the little arrays' sizes. Nevertheless, we still see noticeable gains in using our CPU implementation as opposed to [5]'s `MATLAB@implementation`.

On a CPU, owing to efficient arrays arithmetic native to [22]'s `Numpy` library, the average time to compute the zero levelsets per optimization step for the `odeCFLx` functions is faster with our `Numpy` implementation compared against [5] `LevelSets MATLAB@ Toolbox` library across all experiments. The inefficiencies of `MATLAB@`'s array processing routine becomes pronounced in the time to finish the overall HJ PDE resolution per experiment. For CPU processing of HJ PDEs, it is reasonable, based on these presented data to expect that users would find our library far more useful for everyday computations in matters relating to the numerical resolution of HJ PDEs.

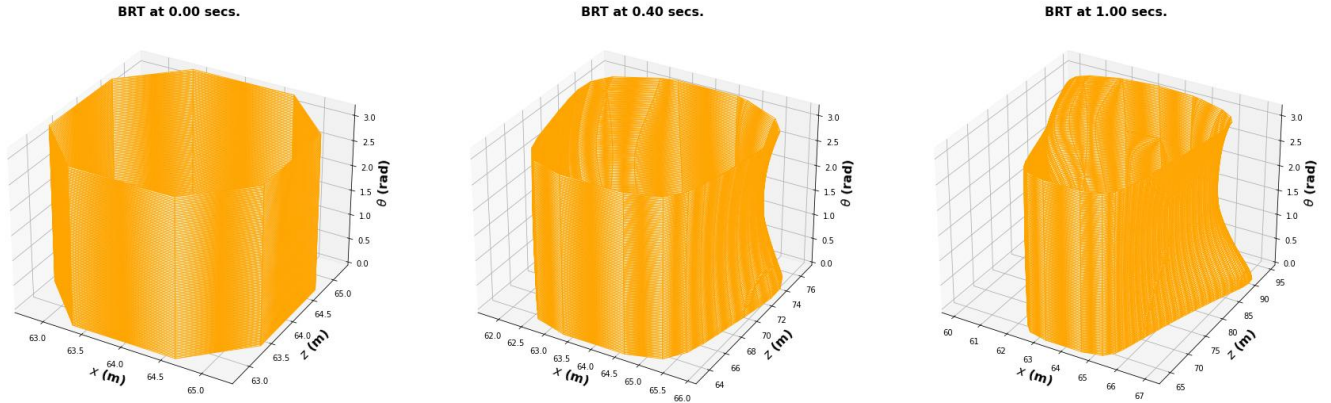


Fig. 3: (Left to Right): Backward reachable tubes (capture surfaces) for the rocket system (cf. Fig. 2) optimized for the paths of slowest-quickest descent in equation (31) at various time steps during the differential game. In all, the BRTs were computed using the method outlined in [4, 11, 21]. We set $a_e = a_p = 1 \text{ ft/sec}^2$ and $g = 32 \text{ ft/sec}^2$ as in Dreyfus’ original example.

TABLE I: Time to Resolve HJ PDEs.

Expt \ Lib	levelsetpy GPU Time (secs)		levelsetpy CPU Time (secs)		MATLAB CPU (secs)	
	Global	Avg. local	Global	Avg. local	Global	Avg. local
Rockets	11.5153 ± 0.038	1.1833	107.84 ± 0.42	10.4023	138.50	13.850
Doub. Integ.	14.7657 ± 0.2643	1.5441	3.4535 ± 0.34	0.4317	5.23	0.65375
Air 3D	30.8654 ± 0.1351	3.0881	129.1165 ± 0.13	12.6373	134.77	16.8462
Starlings	8.6889 ± 0.8323	0.42853	15.2693 ± 0.167	7.4387	N/A	N/A

HJ PDE’s are increasingly becoming a useful tool in control and learning applications. We have presented all the essential components of the python version of the `LevelSet` toolbox for numerically resolving HJ PDEs and for advancing co-dimension one interfaces on Cartesian grids. We have motivated the work presented with a numerical example to demonstrate the efficacy of our numerical implementation and we have provided a compendium of comparisons with other implementations that are available. It stands to reason that our implementation is the fastest available to the best of our knowledge.

REFERENCES

- [1] S. Kruzkov, “First Order Quasilinear Equations In Several Independent Variables,” *Mathematics of the USSR-Sbornik*, 1970. 1
- [2] L. C. Evans, *Partial Differential Equations*. American Mathematical Society, 2022, vol. 19. 1
- [3] J. A. Sethian, “Level Set Methods And Fast Marching Methods: Evolving Interfaces In Computational Geometry, Fluid Mechanics, Computer Vision, And Materials Science,” *Robotica*, vol. 18, no. 1, pp. 89–92, 2000. 1
- [4] S. Osher and R. Fedkiw, “Level Set Methods and Dynamic Implicit Surfaces,” *Applied Mechanics Reviews*, vol. 57, no. 3, pp. B15–B15, 2004. 1, 4, 5, 8
- [5] I. Mitchell, “A toolbox of level set methods, version 1.0,” *The University of British Columbia, UBC CS TR-2004-09*, pp. 1–94, July 2004. 1, 2, 4, 5, 7
- [6] R. Okuta, Y. Unno, D. Nishino, S. Hido, and C. Loomis, “Cupy: A numpy-compatible library for nvidia gpu calculations,” in *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Thirty-first Annual Conference on Neural Information Processing Systems (NIPS)*, 2017. 1, 2
- [7] M. G. Crandall and P.-L. Lions, “Viscosity solutions of hamilton-jacobi equations,” *Transactions of the American mathematical society*, vol. 277, no. 1, pp. 1–42, 1983. 1
- [8] L. Evans and P. E. Souganidis, “Differential Games And Representation Formulas For Solutions Of Hamilton-Jacobi-Isaacs Equations,” *Indiana Univ. Math. J.*, vol. 33, no. 5, pp. 773–797, 1984. 2
- [9] M. G. Crandall and P.-L. Lions, “Two Approximations of Solutions of Hamilton-Jacobi Equations,” *Mathematics of Computation*, vol. 43, no. 167, pp. 1–19, 1984. 2
- [10] J. Lygeros, “On reachability and minimum cost optimal control,” *Automatica*, vol. 40, no. 6, pp. 917–927, 2004. 2, 7
- [11] I. Mitchell, “A Robust Controlled Backward Reach Tube with (Almost) Analytic Solution for Two Dubins Cars,” *EPIC Series in Computing*, vol. 74, pp. 242–258, 2020. 2, 8
- [12] I. M. Mitchell, A. M. Bayen, and C. J. Tomlin, “A Time-Dependent Hamilton-Jacobi Formulation of Reachable Sets for Continuous Dynamic Games,” *IEEE Transactions on Automatic Control*, vol. 50, no. 7, pp. 947–957, 2005. 2, 6
- [13] R. Isaacs, *Differential Games: A Mathematical Theory with Applications to Warfare and Pursuit, Control and Optimization*. Kreiger, Huntington, NY, 1999. 2, 6
- [14] I. Mitchell, “Games of two identical vehicles,” *Dept. Aeronautics and Astronautics, Stanford Univ.*, no. July, pp. 1–29, 2001. 2, 7
- [15] S. Osher and C.-W. Shu, “High-Order Essentially Nonoscillatory Schemes for Hamilton-Jacobi Equations,” *SIAM Journal of Numerical Analysis*, vol. 28, no. 4, pp. 907–922, 1991. 3, 4, 7
- [16] G.-S. Jiang and D. Peng, “Weighted eno schemes for hamilton-jacobi equations,” *SIAM Journal on Scientific computing*, vol. 21, no. 6, pp. 2126–2143, 2000. 4
- [17] M. G. Crandall and A. Majda, “Monotone Difference Approximations For Scalar Conservation Laws,” *Mathematics of Computation*, vol. 34, no. 149, pp. 1–21, 1980. 5
- [18] S. Osher and C.-W. Shu, “Efficient Implementation of Essentially Non-oscillatory Shock-capturing Schemes,” Hampton, Virginia, Tech. Rep. 2, 1988. 5
- [19] C.-W. Shu and S. Osher, “Efficient Implementation of Essentially Non-oscillatory Shock-capturing Schemes, II,” *Journal of computational physics*, vol. 83, no. 1, pp. 32–78, 1989. 5
- [20] S. E. Dreyfus, “Control Problems With Linear Dynamics, Quadratic Criterion, and Linear Terminal Constraints,” Rand Corp, Santa Monica Calif, Tech. Rep., 1966. 6
- [21] M. G. Crandall, L. C. Evans, and P. L. Lions, “Some Properties of Viscosity Solutions of Hamilton-Jacobi Equations,” *Transactions of the American Mathematical Society*, vol. 282, no. 2, p. 487, 1984. 8
- [22] C. R. Harris, K. J. Millman, S. J. Van Der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, et al., “Array programming with numpy,” *Nature*, vol. 585, no. 7825, pp. 357–362, 2020. 7