LevelSetPy: A GPU-Accelerated Package for Hyperbolic Hamilton-Jacobi Partial Differential Equations

LEKAN MOLU, Microsoft Research, USA

This article introduces a software package release for geometrically reasoning about the *safety* desiderata of (complex) dynamical systems via level set methods. In emphasis, safety is analyzed with Hamilton-Jacobi equations. In scope, we provide implementations of numerical algorithms for the resolution of Hamilton-Jacobi-Isaacs equations: the spatial derivatives of the associated value function via upwinding, the Hamiltonian via Lax-Friedrichs schemes, and the integration of the Hamilton-Jacobi equation altogether via total variation diminishing Runge-Kutta schemes. Since computational speed and interoperability with other modern scientific computing libraries (typically written in the Python language) is of essence, we capitalize on modern computational frameworks such as `CUPY` and `NUMPY` and move heavy computations to GPU devices to aid parallelization and improve bring-up time in safety analysis. We hope that this package can aid users to quickly iterate on ideas and evaluate all possible safety desiderata of a system via geometrical simulation in modern engineering problems.

CCS Concepts: • **Software and its engineering** → **Software libraries and repositories**; • **Applied computing** → *Physical sciences and engineering*; • **Mathematics of computing** → *Solvers*.

Additional Key Words and Phrases: partial differential equations, level sets, reachability theory

ACM Reference Format:
Lekan Molu. 20XX. LevelSetPy: A GPU-Accelerated Package for Hyperbolic Hamilton-Jacobi Partial Differential Equations. *J. ACM* 00, 0, Article 000 (20XX), 14 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

With the growing complexity of digital, electronic, and cyberphysical interfaces in the modern systems that we develop, the need to guarantee performance as envisioned by the systems architect in the face of uncertainty has become ever more timely. To deploy modern systems in the wild, modern software must be able to process generated data at multiple levels of abstraction within reasonable time yet guarantee consistency in system performance *in spite* of the dangers that may evolve if nominally envisioned system performance falter. To address this concern, we consider software architectures for the numerical analysis of the safety assurance (ascertaining the freedom of a system from harm). In scope, the computation of safety sets is limited to those numerically obtained via levelset methods [22] derived from Hamilton-Jacobi (HJ) partial differential equations (PDEs) [2, 8] .

Level sets [22] are an elegant tool for numerically analyzing the safety of a dynamical system in a reachability context [13]. They entail discretizing the HJ PDE and employing Lax-Friedrichs Hamiltonian integration schemes [6], and the methods of characteristics and lines [19] to advance the numerical solution to the HJ equation on a grid along based on a total variation diminishing Runge-Kutta (TVD-RK) [20] scheme. In practice, this is done by constructing an implicit surface representation for a problem's value function and employing *upwinding* to advance the evolution of

Author's address: Lekan Molu, lekanmolu@microsoft.com, Microsoft Research, 300 Lafayette St, New York, USA, 10012.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 20XX Association for Computing Machinery.
0004-5411/20XX/0-ART000 $15.00
https://doi.org/XXXXXXX.XXXXXXX

the dynamical system across time. As can be imagined, updating the values at different nodal points of the state space as integration advances has an exponential computational cost. Most of the existing solvers solve this integration process on single-threaded CPUs, usually with non-optimized routine loops. This is debilitating for many practical problems which tend to be in higher dimensions and whose safety reasoning requires fast computation times.

The foremost open-source verification software for engineering applications based on HJ equations and levelset methods is the CPU-based `MATLAB®` levelsets toolbox [15] first developed in 2005. Since its development, there has been vast improvements in hardware and software compute architectures that leverage graphical processing units (GPU) and accelerated linear algebra packages for numerical analysis. It therefore seems reasonable to provide a means of accelerating the computation of HJ PDE solutions to the end of resolving the safety of many cyberphysical systems in a faster manner.

In this article, we will describe our efforts in the past three years on designing a `Numpy®` and GPU-accelerated `CuPy` [18] software packages for numerically resolving generalized discontinuous solutions to time-dependent HJ hyperbolic PDEs that arise in many physical and natural problem contexts. `Numpy` and `Cupy` are both object-oriented array programming packages written in the Python language that enable compact and expressive structures for retrieving, modifying, and executing computations on data arranged as vectors, matrices, or higher-order arrays in a computer's memory. Accompanying this package are implicit calculus operations on dynamic codimension-one interfaces embedded on surfaces in $\mathbb{R}^n$, and numerical discretization schemes for hyperbolic partial differential equations. Furthermore, we describe explicit integration schemes including Lax-Friedrichs, Courant-Friedrichs-Lewy (CFL) constrained TVD-RK schemes for HJ equations. Finally, extensions to reachability analyses for continuous and hybrid systems, formulated as optimal control or game theory problems using viscosity solutions to HJ PDE 's is described in an example. While our emphasis is on the resolution of safe sets in a reachability context for verification settings, the applications of the software package herewith presented extend beyond control engineering applications.

1.1 Background and basic notation

Our chief interest is the evolution form of the Cauchy-type HJ equation

$$\begin{aligned} \boldsymbol{v}_t(x,t) + \boldsymbol{H}(t;x,\nabla_x \boldsymbol{v}) = 0 \text{ in } \Omega \times (0,T] \\ \boldsymbol{v}(x,t) = \boldsymbol{g}, \text{ on } \partial\Omega \times \{t=T\}, \boldsymbol{v}(x,0) = \boldsymbol{v}_0(x) \text{ in } \Omega \end{aligned} \tag{1}$$

or the convection equation

$$\begin{aligned} \boldsymbol{v}_t + \sum_{i=0}^{N} f_i(v)_{x_i} = 0, \quad \text{for } t > 0, \boldsymbol{x} \in \mathbb{R}^n, \\ \boldsymbol{v}(\boldsymbol{x},0) = \boldsymbol{v}_0(\boldsymbol{x}), \ \boldsymbol{x} \in \mathbb{R}^n \end{aligned} \tag{2}$$

where Ω is an open set in $\mathbb{R}^n$; $\boldsymbol{x}$ is the state; $\boldsymbol{v}_t$ denotes the partial derivative(s) of the solution $\boldsymbol{v}$ with respect to time t; the Hamiltonian $\boldsymbol{H} : (0,T] \times \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ and f are continuous; $\boldsymbol{g}$, and $\boldsymbol{v}_0$ are bounded and uniformly continuous (BUC) functions in $\mathbb{R}^n$; and $\nabla_x \boldsymbol{v}$ is the spatial gradient of $\boldsymbol{v}$. It is assumed that $\boldsymbol{g}$ and $\boldsymbol{v}_0$ are given. 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 [4]. In the same vein, global analysis is virtually impossible owing to the lack of existence and uniqueness of solutions $\boldsymbol{v} \in C^1(\Omega) \times (0,T]$ even if $\boldsymbol{H}$ and $\boldsymbol{g}$ are smooth [4]. The method of "vanishing viscosity", based on the idea of traversing the limit as $\delta \to 0$ in the hyperbolic equation (1) (where a parameter $\delta > 0$ "endows" the problem in a

viscosity sense as in gas dynamics [10]), allows generalized (discontinuous) solutions [8] whereupon if $\boldsymbol{v} \in W^{1,\infty}_{loc}(\Omega) \times (0,T]$ and $\boldsymbol{H} \in W^{1,\infty}_{loc}(\Omega)$, one can lay claim to strong notions of general existence, stability, and uniqueness to BUC solutions $\boldsymbol{v}^\delta$ of the (approximate) viscous Cauchy-type HJ equation

$$\begin{aligned} \boldsymbol{v}^\delta_t + \boldsymbol{H}(t;\boldsymbol{x},\nabla_x \boldsymbol{v}^\delta) - \delta\Delta\boldsymbol{v}^\delta = 0 \text{ in } \Omega \times (0,T] \\ \boldsymbol{v}^\delta(x,t) = \boldsymbol{g}, \text{ on } \partial\Omega \times \{t = T\}, \boldsymbol{v}^\delta(\boldsymbol{x},0) = \boldsymbol{v}_0(\boldsymbol{x}) \text{ in } \Omega \end{aligned} \tag{3}$$

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 [5] showed that $|\boldsymbol{v}^\delta(\boldsymbol{x},t) - \boldsymbol{v}(\boldsymbol{x},t)| \leq k\sqrt{\delta}$ for $\boldsymbol{x} \in \Omega$ and $t > 0$. For most of this article, we are concerned with *generalized* viscosity solutions of the manner described by (3).

Mitchell [17] connected techniques used in level set methods to reachability analysis in optimal control (that employ the viscosity solution of the HJ PDE) — essentially showing that the zero-level set of the differential zero-sum two-person game in an Hamilton-Jacobi-Isaacs (HJI) setting [8, 11] constitutes the safe set of a reachability problem. Reachability concerns evaluating the *decidability* of a dynamical system's trajectories' evolution 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 [13], the Hamiltonian is related to the *backward* reachable set of a dynamical system [17]. The essential fabric of this work revolves around reachability problems defined on co-dimension-one surfaces. This reachability theory deals with analyzing the adaptability of complex systems under uncertain dynamics to ascertain the satisfaction of all constraints. Reachability theory thus lends vast applications to various disciplines including robotics, game theory, control theory, aerospace engineering, hydrography, financial markets, biology, and economics. Problems posed in reachability contexts can be analyzed using various forms of HJ equations [1, 17].

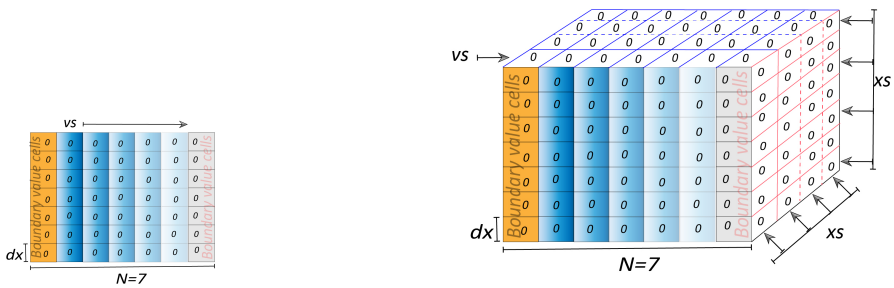
The well-known `LevelSet Toolbox` written by Mitchell [15] is the consolidated `MATLAB®` package that contains the gridding methods, boundary conditions, time and spatial derivatives, integrators and helper functions. While Mitchell motivated the execution of the toolkit in `MATLAB®` for the expressiveness that the language provides, modern data manipulation and scripting libraries often render the original package non-interoperable to other libraries and packages, particularly python and its associated scientific computing libraries such as `Numpy`, `Scipy`, `PyTorch` and their variants — libraries that are becoming prevalent in modern data analysis tooling.

In this regard, we revisit the major algorithms necessary for implicit surface representation of HJ PDEs, write the spatial, temporal, and monotone difference schemes in Python, accelerated onto GPUs via CuPy [18] and present representative numerical examples. A comparison of the performance of our library to the the original one of Mitchell [15] is then provided to show the advantages of our software. All the examples presented in this article conform to the IEEE 754 standard in a 64-bit base-2 format i.e. FP64.

2 GEOMETRY OF IMPLICIT SURFACES AND LAYOUTS

In this section, we discuss how implicit surface functions are constructed, stored on local memory and how they are transferred to GPUs. Throughout, links to `api's`, `routines`, and `subroutines` are highlighted in `blue text` (with a working hyperlink) and we use code snippets in Python to illustrate API calls when it's convenient.

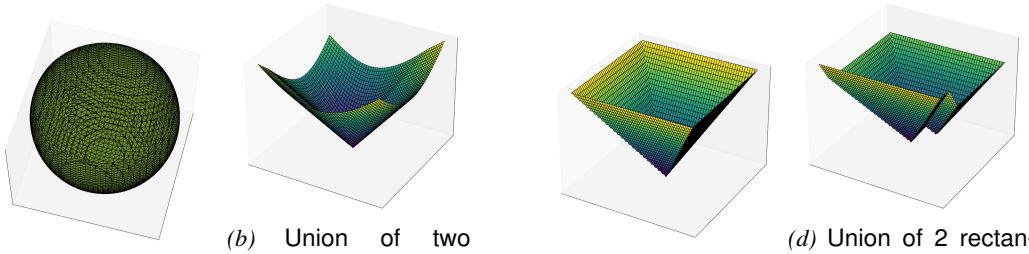
At issue are co-dimension one implicitly defined surfaces on $\mathbb{R}^n$ which represent the interface of a flow, or function e.g. $f(x)$. These interfaces are more often than not the isocontours of some function. This representation is attractive since it requires less number of points to represent a function than



(a) 1D Grid *(b)* Extrapolated 1-D grid into three dimensions

Fig. 1. Creation of a grid with initial attributes N, dx, and vs, the total number of cells per dimension, the size of each cell, and the array of cells in each column.

.



(a) Sphere *(b)* Union of two spheres *(c)* Rectangle *(d)* Union of 2 rectangles

Fig. 2. Examples of implicitly constructed zero levelsets of interfaces of geometric primitives along with Boolean operations on 2D and 3D Cartesian grids. Zero level set of **(a)** a sphere on a 3D grid; **(b)** union of two 3D spheres implicitly constructed on a 2D grid; **(c)** a rectangle on a 2D grid; **(d)** the union of rectangles on a 2D grid.

explicit forms. Relating to the problems of chief interest in this article, the zero isocontour (or the zero levelset) of a reachability optimal control problem is equivalent to the safety set or backward reachable tube. Let us describe the representation of data we employ in what follows.

2.1 Implicit surface containers

Fundamental to the representation of implicit surfaces in our library are `containers` which are implemented as discrete `grid` data structures in our library. The `grid` container is created as an associative memory that is executed as Python dictionaries of keys and corresponding values. Figure 1a is an example of a one-dimensional grid layout in memory. The corresponding keys of note in the grid data structure are the size of cells, `grid.dx` passed by the user[1], the grid resolution, `grid.N`, which specifies the total number of grid points per dimension of $\boldsymbol{v}$ in (1). The attribute `grid.vs` is initialized as a one-dimensional array for all nodal points of $\boldsymbol{v}$ from (3) in an increasingly ordered fashion from the minimum to maximum number of nodal locations. These extrema of nodal points are specified as `min` and `max` in `grid`'s attributes. For problems with multidimensional $\boldsymbol{v}$, the value cells are turned into a multidimensional matrix of cells by repeating entries of `grid.vs` as row and column entries of the resulting $n-$dimensional multi-matrix of cell

[1]All cells are assumed to have a uniform size, however, a user can easily create the field `dx` as a varying-sized array of non-uniform cell sizes.

values of $\boldsymbol{v}$ (see Fig. 1b). The `grid.xs` attribute holds all $n-D$ matrices of coordinates of the values in 3. It is noteworthy to remark that data is stacked in column-major format throughout the memory.

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

Listing 1. Creating a three-dimensional grid.

For every dimension, at the edges of each grid layout is a placeholder array of boundary values. Boundary values are problem specific and users pass a desired boundary value call to the `grid` data structure during its creation. Fig. 2 illustrates a few implicit functions constructed on various grids. Our implementation is in the folder `grids`.

2.2 Advancing integrations via level sets

Osher and Fedkiw [19] advocated embedding the value function on a co-dimension R^{n-1} surface for an n-dimensional problem with signed distance functions. Any geometric primitive such as spheres, cylinders, or the like can then be appropriately constructed on the grid container of §2.1. Specifically, for an implicit surface representation of the continuous geometric function $\boldsymbol{v}$, we treat the coordinates of $\boldsymbol{v}$ on the state space as functional arguments instead of functional values. We do this with a fixed level set of $\boldsymbol{v} : \mathbb{R}^n \rightarrow \mathbb{R}$, implemented as a signed distance function from the query points of moving interfaces point sets described by implicit geometric primitives such as spheres, cylinders, or ellipsoids within the computational domain of the grid. Points inside the computational domain (or interface) possess negative values; points outside the interface possess positive values while points on the interface have zeroes assigned as their value. Then, integrating the PDE (3) consists in moving a front throughout regions of motion in the state space by solving the levelset equation (3) (to be described shortly).

```
e = (g.xs[0])**2 // ellipsoid nodal points
e += 4.0*(g.xs[1])**2
if g.dim==3:
  data += (9.0*(grid.xs[2])**2)
e -= radius // radius=major axis of ellipsoid
```

Listing 2. An `ellipsoid` as a signed distance function in our package.

Figure 1 denotes a few level sets of characteristic geometric primitive implemented in our library. We describe a typical construction of an ellipsoid on a three-dimensional grid in Listing 2: suppose that the zero level set of an implicit surface $\boldsymbol{v}(x,t)$ is defined as $\Gamma = \{x : \boldsymbol{v}(x) = 0\}$ on a grid $G \in \mathbb{R}^n$, where n denotes the number of dimensions. Our representation of Γ on G generalizes a column-major layout. Level sets for several geometric primitives considered are contained in the folder `initialconditions` on our projects page.

3 SPATIAL DERIVATIVES OF THE VALUE FUNCTION: UPWINDING

Consider the levelset equation

$$\boldsymbol{v}_t + \boldsymbol{F} \cdot \nabla \boldsymbol{v} = 0 \tag{4}$$

where $\boldsymbol{F}$ is the speed function, and the implicit function representation of $\boldsymbol{v}$ both denotes and evolves $\boldsymbol{v}$'s solution interface. We are concerned with accurate numerical schemes for approximating the spatial derivatives to $\boldsymbol{v}$ i.e. $\nabla \boldsymbol{v}$. It turns out that with essentially non-oscillatory (ENO) [20] discretization schemes, $\nabla \boldsymbol{v}$ can be "marched forward" in time such that we arrive at monotone

solutions to the levelset equation (4). Suppose that $\boldsymbol{F} = [f_x, f_y, f_z]$ is on a 3D Cartesian grid, expanding (4), we find that $\boldsymbol{v}_t + f_x\boldsymbol{v}_x + f_y\boldsymbol{v}_y + f_z\boldsymbol{v}_z = 0$ captures the implicit function's evolution on the zero levelset since the interface encapsulates $\boldsymbol{v}$. We describe the solution architecture for the `spatialderivatives` procedure in what follows. Consider the differencing schemes on a node within a computational domain that is centered at point i,

$$D^-\boldsymbol{v} = \frac{\partial \boldsymbol{v}}{\partial x} \approx \frac{\boldsymbol{v}_{i+1} - \boldsymbol{v}_i}{\Delta x}, D^+\boldsymbol{v} \approx \frac{\boldsymbol{v}_i - \boldsymbol{v}_{i-1}}{\Delta x}, \tag{5}$$

where Δx is the (uniform) size of each cell on the grid, $\boldsymbol{v}$, and its speed $\boldsymbol{F}$ are defined over a domain Ω. Using the forward Euler method, the level set equation (4) at the n'th integration step becomes $(\boldsymbol{v}^{n+1} - \boldsymbol{v}^n)/\Delta t + f_x^n\boldsymbol{v}_x^n + f_y^n\boldsymbol{v}_y^n + f_z^n\boldsymbol{v}_z^n = 0$. Suppose that the integration is being executed on a one-dimensional domain and around a grid point i, then given that f^n may be spatially varying the equation in the foregoing evaluates to $(\boldsymbol{v}_i^{n+1} - \boldsymbol{v}_i^n)/\Delta t + f_i^n(\boldsymbol{v}_x)_i^n = 0$, where $(\boldsymbol{v}_x)_i$ denotes the spatial derivative of $\boldsymbol{v}$ w.r.t x at the point i. We have followed the naming convention used in the MATLAB toolbox [15] and our implementation is available in the `spatialderivatives` folder.

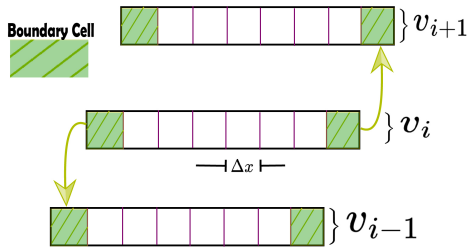


Fig. 3. Upwinding differencing layout.

3.1 First-order accurate discretization

Throughout, we adopt the method of characteristics [19, §3.1]) in choosing the direction of gradients traversal as we compute $\boldsymbol{v}_x$: we approximate $\boldsymbol{v}_x$ with $D^-\boldsymbol{v}$ whenever $f_i > 0$ and we approximate $\boldsymbol{v}_x$ with $D^+\boldsymbol{v}$ whenever $f_i < 0$. No approximation is needed when $f_i = 0$ since $f_i(\boldsymbol{v}_x)_i)$ vanishes.

```
dL, dR = upwindFirstFirst(grid, data, stencil=1) /* data ≡ v */
Host(data) → Device(data) /* transfer data to GPU memory */
ForEach dimension i of v /* stored on grid */
    Patch data according to Fig. 3 /* based on boundary value type in grid.bdry */
    Shift data v_i to v_{i-1} and v_{i+1} according to Fig. 3
    Obtain the left and right directional derivatives, dL, dR by eq. (5)
return dL, dR for every other dimension of v.
```

Listing 3. First-order accurate upwinding directional approximation to $\nabla_x\boldsymbol{v}$.

The routine `upwindFirstFirst` implements the first order derivatives (5) as illustrated in Listing 3.

3.2 Essentially nonoscillatory differencing

The first-order Euler derivatives computed from §3.1 can suffer from inaccurate approximations. Stability, consistence, and convergence are necessary while numerically solving linear PDEs such as (3). Osher and Shu [21] proposed using numerical flux functions of the HJ equation (3), calculated from a divided difference table of the nodal implicit function data. Choosing the most stable numerical flux among computed solutions constitutes the *essentially non-oscillatory* (ENO) polynomial interpolant of the smooth fluxes; this preserves better accuracy as integration of the HJ equation is advanced.

```
1  dL, dR = upwindFirstENOX(grid, v, Δx, stencil=2) /* v ≡ data, Δx ≡ cell size*/
2  Host(data) → Device(data) /* transfer data to GPU memory */
3  ForEach dimension i of v /* stored on grid */
4      Patch data according to Fig. 3 /* based on boundary value type in grid.bdry */
5      Shift data v_i to v_{i-1} and v_{i+1} according to Fig. 3
6      Set the zeroth (spatial) derivative of v as D_i^0 v = v_i
7      /*Compute v's derivatives as the midway between grid nodes */
8       D_{i+1/2}^1 v_i = (D_{i+1}^0 v_i − D_i^0 v_i)/Δx /* 1^st-order divided differences of v */
9       D_i^2 v = (D_{i+1/2}^1 v − D_{i-1/2}^1 v)/2Δx /* Midpoint between the grid nodes */
10      (dL^{0,1}, dR^{0,1}) ← (D_{i+1/2}^1 v_i^-, D_{i+1/2}^1 v_i^+) /* Make two copies of left and right derivatives
    */
11      (dL^0, dL^1)+ = (D_i^2 v_i^-/Δx, D_i^2 v_{i+1}^+/Δx) /* Modify each dL by the 2^nd order derivatives */
12      (dR^0, dR^1)− = (D_i^2 v_{i+1}^-/Δx, D_i^2 v_{i+2}^+/Δx) /* Modify each dR by the 2^nd order derivatives */
13      /* Now, we must determine the least oscillating solution */
14      L^- =| D_{i-1}^2 v |<| D_{i+1}^2 v |, R^- =| D_{i-1}^2 v |>| D_{i+1}^2 v | /* Find smaller of solutions */
15     /* Now choose approximation based on the min|D_2| value. */
16     dL = dL^0 * L^- + dl^1 * R^-,  dR = dR^0 * L^- + dR^1 * R^-
17 return dL, dR.
```

Listing 4. Second-order accurate essentially nonoscillatory upwinding approximation to $\nabla_x \boldsymbol{v}$.

The procedure for calculating the second-order accurate HJ ENO solutions as described in [19, §3.3] is elucidated in Listing 4. For third-order accurate solutions, the procedure proceeds as in Listing 4 but we compute the third divided differences table as $D^3_{i+1/2}\boldsymbol{v} = (D^2_{i+1}\boldsymbol{v} - D^2_i\boldsymbol{v})/3\Delta x$ before Line 10; the essentially non-oscillating (ENO) polynomial and its derivative are constructed as $\boldsymbol{v}(x) = Q_0(x) + Q_1(x) + Q_2(x) + Q_3(x)$, $\boldsymbol{v}_x(x_i) = Q_1'(x_i) + Q_2'(x_i) + Q_3'(x_i)$, with the coefficients $Q_i(x)$ and $Q_i'(x)$ chosen as stipulated in Osher and Fedkiw [19, §3.3.]. As higher order approximations may be susceptible to large gradient oscillations if the interpolant is near this neighborhood, we choose a constant c such that $c^\star = D^3_{k^\star+1/2}$ if $|D^3_{k^\star+1/2}\boldsymbol{v}| \leq |D^3_{k^\star+3/2}\boldsymbol{v}|$, else $D^3_{k^\star+3/2}\boldsymbol{v}$ if $|D^3_{k^\star+1/2}\boldsymbol{v}| > |D^3_{k^\star+3/2}\boldsymbol{v}|$, where $k = i-1$ and $k = i$ for $\nabla_x \boldsymbol{v}^-$ and $\nabla_x \boldsymbol{v}^+$ respectively. We refer readers to the implementation on our online website: `upwindFirstENO3` within the `spatialderivatives` folder.

3.3 Weighted Essentially Nonoscillatory HJ Solutions

Towards improving the accuracy of first-order accurate monotone schemes for the viscosity solution to (1), Jiang and Peng [12] proposed weighted ENO (WENO) schemes which weights substencils of the base ENO stencil according to the solution's relative smoothness on the chosen substencils.

Our implementation of the WENO scheme leverages the computed ENO schemes of §3.2 in addition to computing the smoothness coefficients and weights according to the recommendation in Osher and Shu [20]. Our routine is available in `upwindFirstWENO5a` and called as `upwindFirstWENO5`.

4 THE HAMILTONIAN: A LAX-FRIEDRICHS APPROXIMATION SCHEME

Hyperbolic Hamilton-Jacobi partial differential equations such as

$$\boldsymbol{v}_t + H(t; x, \nabla_x \boldsymbol{v}) = 0 \tag{6}$$

where $H(\nabla \boldsymbol{v})$ is the Hamiltonian are a form of conservative laws which occur in many natural processes. In 3D, they can be written as $\boldsymbol{v}_t + H(\nabla_x \boldsymbol{v}, \nabla_y \boldsymbol{v}, \nabla_z \boldsymbol{v}) = 0$. Resolving this HJ PDE is impossible analytically; however, one can numerically discretize it so that its numerical approximation becomes $\hat{H}(\nabla \boldsymbol{v}^-, \nabla \boldsymbol{v}^+)$ or $\hat{H}(\nabla_x \boldsymbol{v}^-, \nabla_x \boldsymbol{v}^+, \nabla_y \boldsymbol{v}^-, \nabla_y \boldsymbol{v}^+, \nabla_z \boldsymbol{v}^-, \nabla_z \boldsymbol{v}^+)$ in higher dimensions. Then leveraging the classical equivalence between the solutions to scalar conservation laws and HJ

equations in one spatial dimension, a numerically *consistent* solution to H can be found such that $H(\nabla \boldsymbol{v}) \equiv \hat{H}(\nabla \boldsymbol{v})$.

The spatial derivatives in $\hat{H}$ can be approximated with one of the upwinding schemes of §3. To discretize H, the Lax-Friedrichs scheme [6] comes into play. Numerical discretization of the HJ equation proceeds as

$$(\boldsymbol{v}^{n+1} - \boldsymbol{v}^n)/\Delta t + \hat{H}^n(\nabla_x \boldsymbol{v}^-, \nabla_x \boldsymbol{v}^+, \nabla_y \boldsymbol{v}^-, \nabla_y \boldsymbol{v}^+, \nabla_z \boldsymbol{v}^-, \nabla_z \boldsymbol{v}^+) = 0 \tag{7}$$

and *consistency* in the numerical Hamiltonian's solution with the actual system Hamiltonian is enforced via a Courant-Friedreichs-Lewy (CFL) condition which states that the speed of a physical wave be no faster than a numerical wave. Put differently, we require $|\boldsymbol{v}| > \Delta x/\Delta t$ (for a single-dimensional system). In this sentiment, the Lax-Friedrichs approximation scheme for H in 2D is of the form

$$\hat{H} = H\left(\frac{\nabla_x \boldsymbol{v}^+ + \nabla_x \boldsymbol{v}^-}{2}, \frac{\nabla_y \boldsymbol{v}^+ + \nabla_y \boldsymbol{v}^-}{2}\right) - \alpha_x\left(\frac{\nabla_x \boldsymbol{v}^+ - \nabla_x \boldsymbol{v}^-}{2}\right) - \alpha_y\left(\frac{\nabla_y \boldsymbol{v}^+ - \nabla_y \boldsymbol{v}^-}{2}\right) \tag{8}$$

where the quantity of numerical viscosity is managed by the dissipation coefficients α_x, α_y defined as $\alpha_x = \max|H_x(\boldsymbol{v}_x, \boldsymbol{v}_y)|,\ \alpha_y = \max|H_y(\boldsymbol{v}_x, \boldsymbol{v}_y)|$. To preserve stability of the integration scheme, we follow the CFL stability consition of Osher and Fedkiw [19, §5.3] i.e., $\Delta t \max\{|H_x|/\Delta x + |H_y|/\Delta y + |H_z|/\Delta z\} < 1$.

```
1 termLaxFriedrichs(t_init, v, sysData) /* sysData struct: grid info, p, α, Ĥ, ∇Ĥ routines */
2 dL, dR, dC = sysData.CoStateCalc(sysData.grid, v) /* Obtain co-state i.e. left,
3 /* right, and centered derivatives using one of the upwinding schemes */;
4 Ĥ = sysData.hamFunc(t_init, v, dC, sysData) /* compute the numerical Hamiltonian */
5 α, Δt max{|H|/Δx} = sysData.dissFunc(t_init, v, dL, dR, sysData) /* Lax-Friedrichs
6 /* stabilization coefficients and CFL condition */
7 ∇Ĥ = Ĥ − α /* update the implicitly stored data based on the change. */
```

Listing 5. Computational scheme for the Lax-Friedrichs Approximation to $GradH(x, p)$.

Our Lax-Friedrichs implementation scheme is described in the pseudo-code of Listing 5. It takes as input the initial time step for the current integration horizon, the value function data embedded within the grid structure, and a `sysData` data structure that consists of the routines for computing the user-supplied problem's co-state, p, dissipation coefficients (i.e. α_x, α_y), and a function for evaluating the numerical Hamiltonian. The spatial derivatives of $\boldsymbol{v}$ are computed by choosing a desired routine from one of the upwinding schemes described in §3 which are then passed to the internal numerical Hamiltonian calculator as seen on Line 4. For details, we refer readers to the online implementation on this `link`.

5 THE HJ EQUATION: TOTAL VARIATION DIMINISHING RUNGE-KUTTA

So far we have considered numerical methods for solving the spatial derivatives $\nabla_x \boldsymbol{v}$ and the numerical Hamiltonian $\hat{H}(t, x, \nabla_x \boldsymbol{v})$ in the hyperbolic HJ PDE (6). We now shift our attention to spatial discretization methods for resolving the hyperbolic PDE (6) altogether. Ours is an implementation of the non-oscillatory total variation diminishing (TVD) spatial discretization CFL-constrained Runge-Kutta (RK) scheme via method of lines [20, 23]. Under TVD-RK schemes in general, it is required that the total variation of the numerical solution to $\boldsymbol{v}$ i.e. $TV(\boldsymbol{v}) = \sum_j |\boldsymbol{v}_{j+1} - \boldsymbol{v}_j|$ does not increase $TV(\boldsymbol{v}^{n+1}) \leq TV(\boldsymbol{v}^n)$ for all n and Δt such that $0 \leq n\Delta t \leq T$ for first-order in time Euler forward integrators. To maintain the TVD property, it is typical to impose a CFL time step constraint such as $\Delta t \leq c\Delta x$, where c is a CFL coefficient for the higher-order time discretization. Following Osher and Shu [20] observations about the limitations of higher-order schemes with wider stencils that

cause numerical degradation, we limit our implementations to the recommended third-order schemes which are known to work well. For the technical details behind our implementation, we refer readers to Osher and Shu [20].

```
t, v = odeCFL3(func, tspan, v_0) /* func:= Lax-Friedrichs routine */
Host(v_0) → Device(v_0) /* transfer data to GPU memory */
v ← v_0 /* Copy over initial value function */
cfl_factor = 0.32; stepBound = 0
while t_f − t ≥ εt_f /* t_f = final time in tspan; ε > 0 */
    ForEach v_idx in v
        v̇_idx, t_idx = func(t, v) /* Compute the numerical approximation of H(x, p) */
        Δt = min(t_f − t, maxStep, cfl_factor*stepBound) /* CFL timestep bound, up to t_f.*/
        /* First substep */
        t_1 = t + Δt, v_1 = v + Δt * v̇
        /* Second substep */
        v̇_idx, t_idx = func(t_1, v_1) /* Euler forward stepping t_{n+1} → t_{n+2}*/
        t_2 = t_1 + Δt; v_2 = v_1 + Δt * v̇ /* Update time step and values */
        t_{1/2} = (3t + t_2)/4, v_{1/2} = (3v + v_2)/4 /* approximation at t_{n+1/2} */
        /* Third substep */
        v̇_idx, t_idx = func(t_{1/2}, v_1 1/2) /* Euler forward stepping t_{n+1} → t_{n+2}*/
        t_{3/2} = t_{1/2} + Δt; v_{3/2} = v_{1/2} + Δt * v̇ /* Update time step and values */
        t = (t + 2t_{3/2})/3, v = (v + 2v_{3/2})/3 /* approximation at t_{n+1} */
return t, v
```

Listing 6. A 3^{rd}-order TVD-RK Scheme for integrating an hyperbolic HJ PDE by MoL.

Listing 6 describes a representative implementation of the method of lines integration scheme. We refer readers to the `integration` folder for all of our implementations.

6 NUMERICAL EXPERIMENTS

In this section, we will present problems motivated by real-world scenarios and amend them to HJ PDE forms where their numerical solutions can be resolved with our `levelsetpy` toolbox. W All the examples presented in conform to the FP64 double floating precision format. We consider *differential games* as a *collection/family of games*, $\Upsilon = \{\Gamma_1, \cdots, \Gamma_g\}$ where each game may be characterized as a pursuit-evasion *game*, Γ. Such a game terminates when *capture* occurs, that is the distance between players falls below a predetermined threshold. Each player in a game shall constitute either a pursuer ($\boldsymbol{P}$) or an evader ($\boldsymbol{E}$). The essence of our examples is to geometrically (approximately) ascertain the separation between the *barrier hypersurface*, where starting points exist for which escape occurs, capture occurs, and for which the outcome is neutral. The task is to assay the *game of kind* for the envelope of the capturable states. This introduces the *barrier* hypersurface which separates, in the initial conditions space, the hypersurface of capture from those of escape. In this *game of kind* postulation, all optimal strategies are not unique, but rather are a *legion*. Ergo, we are concerned with the set of initial positions on the vectogram where the capture zone (CZ) exists i.e.

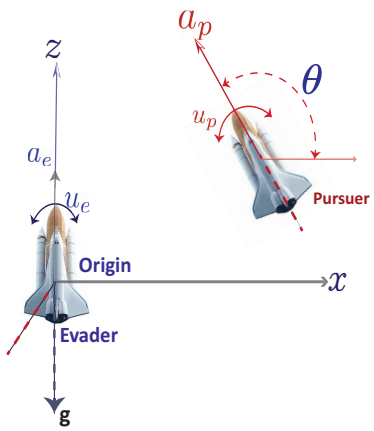


Fig. 4. Two rockets on a Cartesian xz-plane with relative thrust inclination $\theta := u_p - u_e$.

where game termination occurs; and the nature of escape zones (EZ) i.e. zones where termination or escape does not occur – after playing the differential game.

6.1 The Rockets Launch Problem

We consider the rocket launch problem of Dreyfus [7] and amend it to a differential game between two identical rockets, $\boldsymbol{P}$ and $\boldsymbol{E}$, on an (x, z) cross-section of a Cartesian plane. We want to compute the backward reachable tube (BRT) [17] 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 $\boldsymbol{P}$ or $\boldsymbol{E}$ is below zero, and where the *HJI PDE's value functional* is exactly zero.

For a two-player differential game, let $\boldsymbol{P}$ and $\boldsymbol{E}$ share identical dynamics in a general sense so that we can freely choose the coordinates of $\boldsymbol{P}$; however, $\boldsymbol{E}$'s origin is a distance $\boldsymbol{\phi}$ away from (x, z) at plane's origin (see Fig. 4) so that the $\boldsymbol{PE}$ vector's inclination measured counterclockwise from the $\boldsymbol{x}$ axis is θ.

Let the states of $\boldsymbol{P}$ and $\boldsymbol{E}$ be denoted by $(\boldsymbol{x}_p, \boldsymbol{x}_e)$. Furthermore, let the $\boldsymbol{P}$ and $\boldsymbol{E}$ rockets be driven by their thrusts, denoted by (u_p, u_e) respectively (see Figure 4). Fix the rockets' range so that what is left of the motion of either $\boldsymbol{P}$ or $\boldsymbol{E}$'s is restricted to orientation on the (x, z) plane as illustrated in Fig. 4. It follows that the relevant *kinematic equations* (KE) (derived off [7]'s single rocket dynamics) is

$$\dot{x}_{2e} = x_{4e};\ \dot{x}_{2p} = x_{4p};\ \dot{x}_{4e} = a\sin u_e - g;\ \dot{x}_{4p} = a\sin u_p - g \tag{9}$$

where a and g are respectively the acceleration and gravitational accelerations (in feet per square second)[2]. $\boldsymbol{P}$'s motion relative to $\boldsymbol{E}$'s along the $(\boldsymbol{x}, \boldsymbol{z})$ plane includes the relative orientation shown in Fig. 4 as $\theta = u_p - u_e$, the control input. Following the conventions in Fig. 4, the game's relative equations of motion in *reduced space* [11, §2.2] i.e. is $\boldsymbol{x} = (x, z, \theta)$ where $\theta \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right)$ and $(x, z) \in \mathbb{R}^2$ are $\dot{\boldsymbol{x}} = \{\dot{x}, \dot{z}, \dot{\theta}\}$, and

$$\dot{x} = a_p\cos\theta + u_e x,\ \dot{z} = a_p\sin\theta + a_e + u_e x - g,\ \dot{\theta} = u_p - u_e. \tag{10}$$

The capture radius of the origin-centered circle $\boldsymbol{\phi}$ (we set $\boldsymbol{\phi} = 1.5$ ft) is $\|\boldsymbol{PE}\|_2$ so that $\boldsymbol{\phi}^2 = x^2 + z^2$. All capture points are specified by the variational HJ PDE [17]:

$$\frac{\partial \boldsymbol{\phi}}{\partial t}(\boldsymbol{x}, t) + \min\left[0, \boldsymbol{H}(\boldsymbol{x}, \frac{\partial \boldsymbol{\phi}(\boldsymbol{x}, t)}{\partial \boldsymbol{x}})\right] \leq 0, \tag{11}$$

with Hamiltonian given by

$$\boldsymbol{H}(\boldsymbol{x}, p) = -\max_{u_e \in [\underline{u}_e, \bar{u}_e]} \min_{u_p \in [\underline{u}_p, \bar{u}_p].} \begin{bmatrix} p_1 & p_2 & p_3 \end{bmatrix} \begin{bmatrix} a_p\cos\theta + u_e x \\ a_p\sin\theta + a_e + u_p x - g u_p - u_e \end{bmatrix}. \tag{12}$$

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

$$\begin{aligned} \boldsymbol{H}(\boldsymbol{x}, p) &= -\max_{u_e \in [\underline{u}_e, \bar{u}_e]} \min_{u_p \in [\underline{u}_p, \bar{u}_p]} \left[p_1(a_p\cos\theta + u_e x) + p_2(a_p\sin\theta + a_e + u_p x - g) + p_3(u_p - u_e)\right], \\ &\triangleq -a p_1\cos\theta - p_2(g - a - a\sin\theta) - \bar{u}|p_1 x + p_3| + \underline{u}|p_2 x + p_3|, \end{aligned} \tag{13}$$

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

[2]We set $a = 1 ft/sec^2$ and $g = 32 ft/sec^2$ in our simulation.

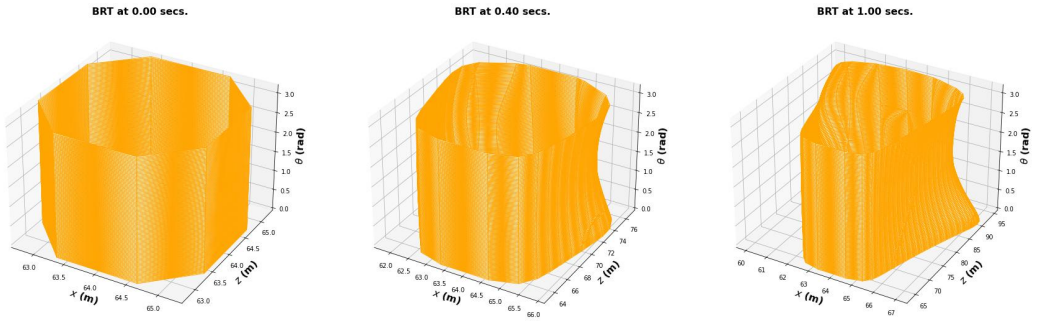


Fig. 5. *(Left to Right)*: Backward reachable tubes (capture surfaces) for the rocket system (cf. Fig. 4) optimized for the paths of slowest-quickest descent in equation (12) at various time steps during the differential game. In all, the BRTs were computed using the method outlined in [3, 16, 19]. We set $a_e = a_p = 1ft/sec^2$ and $g = 32ft/sec^2$ as in Dreyfus' original example.

For the target set guarded by $\boldsymbol{E}$, 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 (13), a TVD-RK discretization scheme [21] based on fluxes is used in choosing smooth nonoscillatory results as described in §5. Denote by (x, y, z) a generic point in $\mathbb{R}^3$ so that given mesh sizes Δx, Δy, Δz, $\Delta t > 0$, letters u, v, w represent functions on the x, y, z lattice: $\Delta = \{(x_i, y_j, z_k) : i, j, k \in \mathbb{Z}\}$.

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

Listing 7. HJ ENO2 computational scheme for the rockets.

The Hamiltonian, upwinding scheme, flux dissipation method, and the overapproximation parameter for the essentially non-oscillatory polynomial interpolatory data used in geometrically reasoning about the *target tube* is set up as seen in Listing 7. The data structure `finite_diff_data` contains all the routines needed for adding dynamics to the original implicit surface representation of $\boldsymbol{v}(\boldsymbol{x}, t)$. The Hamiltonian is approximated with the routine `termLaxFriedrichs` described in §4), and passed to the `innerFunc` query field. During integration, it is calculated as derived in (13) within the `hamFunc` query field. We adopt a second-order accurate essentially nonoscillatory HJ upwinding scheme is used to compute $\nabla \boldsymbol{v}$. Since we are overapproximating the value function, we specify a `True` parameter for the `positive` query field.

Safety is engendered by having the evader respond optimally to the pursuer at various times during the game. The entire safety set over the time interval of play constitutes the backward reachable tube (BRT) [17]; this BRT under the control strategies of $\boldsymbol{P}$ or $\boldsymbol{E}$, is a part of the phase space, $\Omega \times T$. We compute the *overapproximated* BRT of the game over a time span of $[-2.5, 0]$ seconds during 11 global optimization time steps using the CFL-constrained TVD-RK solver for the HJ equation..

The initial value function (leftmost inset of Fig. 5) 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 rockets' orientation (see Fig. 4). This closed and bounded assumption of the target set is a prerequisite of backward reachable analysis (see [17]). It

Table 1. Time to Resolve HJ PDE's. N/A signifies the experiment was not executed in MATLAB due to its memory inefficient programmatic scheme.

Lib / Expt	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

allows us to include all limiting velocities. We reach convergence at the eleventh global optimization timestep (rightmost inset of Fig. 5).

Reachability [13, 14] thus affords us an ability to numerically reason about the behavior of these two rockets before motion execution without closed-form geometrical analysis. This example demonstrates running a CFL-restricted TVD-RK optimization scheme for a finite number of global optimization timesteps. More complicated examples are available on our github repository and we encourage readers to use and test the library for multiple other safety analysis and/or synthesis.

6.2 Computational Time Comparison with LevelSet Toolbox

We compare the solution for recovering the zero level set of the system presented in the previous example against Mitchell [15]'s `LevelSet Toolbox`. Additional results are available within this article's accompanying supplementary materials, and in the online package. In all, we compare the efficacy of running various computational problems using our library on a `CPU` – running `Numpy` versus `MATLAB's`®levelset toolbox [15] – and on a `GPU`. The CPU examples all run on the main thread in order for us to provide a measurable comparison to the original MATLAB implementation. We do not leverage any multiple thread computational tools such as OpenMP or Pthread. Throughout the code, there are parts that execute on the host and the parts that consume a lot of memory are transferred to the device. We do this by design so that we do not create a communication bottleneck in computation between computational operations that require a light memory consumption footprint and large computational operations that could benefit from device computation. For the `CPU` tests, we run the computation on an Intel Core™i9-10885H 16 cores-processor with a 2.4GHz clock frequency, and 62.4GB memory. On `GPUs`, we employed an `NVIDIA Quadro RTX` 4000 with 8.192 GiB memory running on a mobile workstation in all of our `GPU` library accelerations. Table 1 depicts the time it takes to run the total variation diminishing Runge-Kutta scheme for the reachable problems considered. The column `Avg. local` is the average time of running one single step of the TVD-RK scheme (section 5) 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. We see that computation is significantly faster when our schemes are implemented on a GPU in all categories save the low-dimensional double integral plant system. We attribute this to the little amount of data points used in the overapproximated stacked levelsets. For the Air3D game and the two rockets differential game problem, the average local time for computing the solutions to the stagewise HJ PDE's sees a gain of $\sim 76\%$; the global time shows a gain of 76.09% over Mitchell [15]'s MATLAB computational scheme. Similarly, we notice substantial computational gains 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, compared against Mitchell [15] library, 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 Mitchell [15]'s native `MATLAB`®toolbox. We attribute this to the little size of arrays of interest in this problem. The entire target set of the double integral plant exists on a two-dimensional grid whose analytic and approximate time-to-reach-the-origin computational time involves little computational gain in passing data onto the GPU. Nevertheless, we still see noticeable gains in using our CPU implementation as opposed to Mitchell [15]'s native `MATLAB`®toolbox.

On a CPU, owing to efficient arrays arithmetic native to Harris et al. [9]'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 Mitchell [15] LevelSets `MATLAB`® Toolbox library computations across all experiments. The inefficiencies of `MATLAB`®'s array processing routine in the longer time to resolve stagewise BRTs and the effective time to finish the overall HJ PDE resolution per experiment manifests in all of our experimental categories. For `CPU` processing of HJ PDE 's, 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 PDE 's.

In all, there is conclusive evidence that our implementations are faster, extensible to modern libraries, and scalable for modern complex system design and verification problems that arise.

7 CONCLUSION

With the increasing complex systems that are gaining utilization almost every domain of expertise including reinforcement learning, control systems, robotics, modeling, and real-time prediction in sensitive and safety-critical environments, it has become paramount to start considering backup safety filters for these generally unstable large function approximators in safety-critical environments. We have presented all the essential components of the python version in the `levelsetpy` library for numerically resolving HJ PDEs (in a reachability setting and other associated contexts) by advancing co-dimension one interfaces on Cartesian grids. We have motivated the work presented with several numerical examples to demonstrate the efficacy of our library. The examples we have presented in this document provide a foray into the computational aspects of ascertaining the safety (freedom from harm) of the complex systems that we are continually designing and building. This includes safety-critical analysis encompassing simple and complex multiagent systems whose safe navigation over a phase space can be considered in an HJ PDE framework. As complexity evolves, we hope that our library can serve as a useful tool for tinkerers looking for an easy-to-use proof-of-concept toolkit for verification of dynamical systems based on a principled numerical analysis. We encourage users to download the library from the author's github webpage (it is available in `CPU` and `GPU` implementations via appropriately tagged branches) and use it robustly for various problems of interest where speed and scale for the solubility of hyperbolic conservation laws and HJ PDE's are of high importance.

REFERENCES

[1] Anayo K Akametalu, Jaime F Fisac, Jeremy H Gillula, Shahab Kaynama, Melanie N Zeilinger, and Claire J Tomlin. 2014. Reachability-based Safe Learning with Gaussian Processes. In *53rd IEEE Conference on Decision and Control*. IEEE, 1424–1431.

[2] Michael Crandall and Andrew Majda. 1980. The method of fractional steps for conservation laws. *Numer. Math.* 34, 3 (1980), 285–314.

[3] M. G. Crandall, L. C. Evans, and P. L. Lions. 1984. Some Properties of Viscosity Solutions of Hamilton-Jacobi Equations. *Trans. Amer. Math. Soc.* 282, 2 (1984), 487.

[4] Michael G Crandall and Pierre-Louis Lions. 1983. Viscosity solutions of Hamilton-Jacobi equations. *Transactions of the American mathematical society* 277, 1 (1983), 1–42.

[5] Michael G Crandall and P-L Lions. 1984. Two Approximations of Solutions of Hamilton-Jacobi Equations. *Mathematics of Computation* 43, 167 (1984), 1–19.

[6] Michael G Crandall and Andrew Majda. 1980. Monotone Difference Approximations For Scalar Conservation Laws. *Math. Comp.* 34, 149 (1980), 1–21.

[7] Stuart E Dreyfus. 1966. *Control Problems With Linear Dynamics, Quadratic Criterion, and Linear Terminal Constraints*. Technical Report. Rand Corp, Santa Monica Calif.

[8] L.C. Evans and Panagiotis E. Souganidis. 1984. Differential Games And Representation Formulas For Solutions Of Hamilton-Jacobi-Isaacs Equations. *Indiana Univ. Math. J* 33, 5 (1984), 773–797.

[9] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. 2020. Array programming with NumPy. *Nature* 585, 7825 (2020), 357–362.

[10] Eberhard Hopf. 1950. The Partial Differential Equation $u_t + uu_x = \mu^\star_{xx}$. (1950).

[11] R Isaacs. 1999. *Differential Games: A Mathematical Theory with Applications to Warfare and Pursuit, Control and Optimization.* Kreiger, Huntigton, NY.

[12] Guang-Shan Jiang and Danping Peng. 2000. Weighted ENO schemes for Hamilton–Jacobi equations. *SIAM Journal on Scientific computing* 21, 6 (2000), 2126–2143.

[13] John Lygeros. 2004. On reachability and minimum cost optimal control. *Automatica* 40, 6 (2004), 917–927.

[14] Ian Mitchell. 2001. Games of two identical vehicles. *Dept. Aeronautics and Astronautics, Stanford Univ.* July (2001), 1–29.

[15] Ian Mitchell. 2004. A Toolbox of Level Set Methods, version 1.0. *The University of British Columbia, UBC CS TR-2004-09* (July 2004), 1–94.

[16] Ian Mitchell. 2020. A Robust Controlled Backward Reach Tube with (Almost) Analytic Solution for Two Dubins Cars. *EPiC Series in Computing* 74 (2020), 242–258.

[17] Ian M. Mitchell, Alexandre M. Bayen, and Claire J. Tomlin. 2005. A Time-Dependent Hamilton-Jacobi Formulation of Reachable Sets for Continuous Dynamic Games. *IEEE Trans. Automat. Control* 50, 7 (2005), 947–957.

[18] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. 2017. 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)*.

[19] S Osher and R Fedkiw. 2004. Level Set Methods and Dynamic Implicit Surfaces. *Applied Mechanics Reviews* 57, 3 (2004), B15–B15.

[20] Stanley Osher and Chi-Wang Shu. 1988. *Efficient Implementation of Essentially Non-oscillatory Shock-capturing Schemes*. Technical Report 2. Hampton, Virginia. 439–471 pages.

[21] Stanley Osher and Chi-Wang Shu. 1991. High-Order Essentially Nonoscillatory Schemes for Hamilton-Jacobi Equations. *SIAM Journal of Numerical Analysis* 28, 4 (1991), 907–922.

[22] James A. Sethian. 2000. Level Set Methods And Fast Marching Methods: Evolving Interfaces In Computational Geometry, Fluid Mechanics, Computer Vision, And Materials Science. *Robotica* 18, 1 (2000), 89–92.

[23] Chi-Wang Shu and Stanley Osher. 1989. Efficient Implementation of Essentially Non-oscillatory Shock-capturing Schemes, II. *Journal of computational physics* 83, 1 (1989), 32–78.

Received 30 April 2024