# A Toolbox of Level Set Methods
# (Version 1.1)
# UBC CS TR-2007-11

Ian M. Mitchell
Department of Computer Science
University of British Columbia
mitchell@cs.ubc.ca
http://www.cs.ubc.ca/∼mitchell

June 1, 2007

### Abstract

This document describes Version 1.1 of the Toolbox of Level Set Methods, a software package for solving time-dependent Hamilton-Jacobi partial differential equations (PDEs) in the MATLAB programming environment. Level set methods are often used for simulation of dynamic implicit surfaces in graphics, fluid and combustion simulation, image processing, and computer vision. Hamilton-Jacobi and related PDEs arise in fields such as control, robotics, differential games, dynamic programming, mesh generation, stochastic differential equations, financial mathematics, and verification. The algorithms in the toolbox can be used in any number of dimensions, although computational cost and visualization difficulty make dimensions four and higher a challenge. All source code for the toolbox is provided as plain text in the MATLAB m-file programming language. The toolbox is designed to allow quick and easy experimentation with level set methods, although it is not by itself a level set tutorial and so should be used in combination with the existing literature.

This document supercedes the documentation for Version 1.0 of the Toolbox (UBC TR-2004-09).

## Copyright

This Toolbox of Level Set Methods, its source, and its documentation are Copyright ©2007 by Ian M. Mitchell. Use of or creating copies of all or part of this work is subject to the following licensing agreement.

This license is derived from the ACM Software Copyright and License Agreement (1998), which may be found at:

## License

**Commerical or Other Use:** Any User wishing to make a commercial or other use of the Software is encouraged to contact the Developer at `mitchell@cs.ubc.ca` to arrange an appropriate license. Commercial use includes (1) integrating or incorporating all or part of the source code into a product for sale or license by, or on behalf of, the User to third parties, or (2) distribution of a compiled or source code version of the Software to third parties for use with a commercial product sold or licensed by, or on behalf of, the User.

# Contents

# Preface

It has been a little over two years since I made a beta release of Version 1.1 of the Toolbox. I had intended to wait a few months to see if any other bugs appeared in my experiments with the code, test all of the old examples to make sure that none had been broken by the new additions, and then make the official release. My mistake. There was always some more pressing task than performing those regression tests, and so the official release kept getting delayed.

I am happy to finally make this official release available. Over the intervening months since the beta I have made a number of bug fixes, and added two small features that were not present in the beta: a subcell fix in `termReinit` from [27] that significantly reduces the movement of the interface during reinitialization, and a new routine `gridnd2mesh` that converts between the Toolbox's `ndgrid`-based grid definition and the `meshgrid`-based definition required by several of MATLAB's visualization routines. A full list of features added to the Toolbox since Version 1.0 is given in section 4.

Not all of the code that I have written in the past two years is contained in this Version 1.1 release. Where possible—in other words, where the code is clean enough that somebody else might be able to understand it—I have posted additional code at the Toolbox's web site [14]. In particular, code to accompany [16] provides for additional explicit Runge-Kutta integrators from [32] and a new motion by mean curvature approximation from [21]. Those features of this additional code that prove generally useful or instructive will likely be incorporated into the Toolbox kernel or example set in future versions. Keep checking the web site, because I plan to stick with this piecemeal policy for code pre-release in the future rather than risking another beta release which celebrates multiple birthdays.

I would like to thank all of the users who have sent emails with questions, feature requests, support and/or bug fixes over the past few years. Keep them coming! In particular, if you publish articles or use the Toolbox in your business, please let me know. Your messages not only encourage me to keep working on the Toolbox, but also add strength to my grant proposals so that I can support students with whom I can add, test and document features faster than I can alone.

Ian M. Mitchell,
Vancouver, Canada
May, 2007.

# 1 Introduction

*Level set methods* are a collection of numerical algorithms for solving a particular class of partial differential equations (PDEs). They have proven popular in recent years for tracking, modeling and simulating the motion of dynamic surfaces in fields including graphics, image processing, computational fluid dynamics, materials science and many others. Rather than an explicit representation in terms of edges (a one dimensional surface in $\mathbb{R}^2$) or faces (a two dimensional surface in $\mathbb{R}^3$ ), in level set methods the surface is represented implicitly through a *level set function* $\phi(x)$. The surface itself is the zero isosurface or zero level set $\{x \in \mathbb{R}^d \mid \phi(x) = 0\}$. Various types of surface motion can be described by PDEs involving $\phi$. Because of the implicit representation, these methods are sometimes also referred to as *dynamic implicit surfaces.*

Although popularized under the name level set methods, the underlying PDE—a hyperbolic PDE with first order time derivatives often called a *Hamilton-Jacobi* (HJ) PDE—appears in many other branches of mathematics including optimal control, zero sum differential games, mathematical finance and stochastic differential equations.

Level set proponents often claim that a primary advantage of level set methods is their ease of implementation, a claim which we find overly optimistic. PDEs are rarely easy to implement; for example, the base MATLAB installation includes only a PDE solver for one dimensional parabolic-elliptic equations. For simple convective motion (including rigid body motion), it is far easier to implement marker particle or Lagrangian methods for evolving an interface. The advantage of level set methods, however, is that they can accomodate many types of surface motion without any significant increase in theoretical or implementation complexity. Among these capabilities are:

- It is conceptually straightforward to move from two to three and even higher dimensions (although computational cost is exponential in dimension).

- Surfaces automatically merge and separate.

- Geometric quantities are easy to calculate: surface normal, curvature, direction and distance to the nearest point on the surface. Surface motion can depend on these quantities.

In contrast, it is a significant undertaking to implement dynamic surfaces with marker particles in three dimensions with merging, separation and calculation of surface normals and curvatures.

Much of the level set literature has grown out of the seminal paper [25], although dynamic implicit surfaces and the HJ PDE date back much further. Readers interested in using level

set methods for their applications are encouraged to read both of the well written texts [28] and [24]. They discuss the basic concepts in different but complementary ways, and then proceed to cover a variety of additional topics, few of which overlap. In our (probably biased) opinion, the strengths of the two books are their explanations of:

- Osher and Fedkiw [24]: high order accuracy methods, image processing, computational physics.

- Sethian [28]: fast marching methods, unstructured grids, a wide variety of applications.

Because we work with time-dependent equations on structured grids, most of the algorithms and examples in this version of the toolbox are taken from [24].

## 1.1 Toolbox Objectives

The goal of this toolbox is to provide a collection of routines which implement the basic level set algorithms in MATLAB* for any number of dimensions. In using MATLAB we seek to minimize not execution time, but the combination of execution and coding time. In our experience, the visualization, debugging, data manipulation and scripting capabilities of MATLAB make construction of numerical code so much simpler, when compared to compiled languages like C++ or Fortran, that the increase in execution time is quite acceptable. Readers interested in faster implementations should note that for the restricted class of problems that we consider in the toolbox the execution time penalty is relatively small. It is only for more complex problems on unstructured, adaptive or localized grids that a compiled implementation will run significantly faster.

In the jargon of the level set literature, this toolbox provides routines to solve time-dependent Hamilton-Jacobi and related equations on fixed, structured Cartesian grids in any number of dimensions. More concretely, the PDE to be solved is of the form

$$D_t\phi(x,t) + H(x,t,\phi,\nabla\phi,D_x^2\phi) = 0 \text{ for } x \in \mathbb{R}^n, t \geq 0. \tag{1}$$

subject to initial conditions

$$\phi(x,t) = \phi_0(x) \text{ for } x \in \mathbb{R}^n. \tag{2}$$

In this PDE, $x \in \mathbb{R}^n$ is the state space, $\phi : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}$ is the level set function, $D_t\phi$ is the partial derivative of $\phi$ with respect to the time variable $t$, $\nabla\phi = D_x\phi$ is the gradient of $\phi$—the vector of partials of $\phi$ with respect to the state space variables $x$—and $D_x^2\phi$ is the Hessian matrix of second partial derivatives with respect to the state space variables. The initial

---

*MATLAB is a product and trademark of The Mathworks Incorporated of Natick, Massachusetts. For more details see http://www.mathworks.com/products/matlab/. The level set toolbox described in this document was developed by the authors of this document, and is neither endorsed by nor a product of The Mathworks.

conditions $\phi_0$ should be bounded and continuous and $H$ should satisfy the monotonicity requirement [8]

$$H(x, t, r, p, \mathbf{X}) \leq H(x, t, s, p, \mathbf{Y}), \text{ whenever } r \leq s \text{ and } \mathbf{Y} \leq \mathbf{X}, \tag{3}$$

where $\mathbf{X}$ and $\mathbf{Y}$ are symmetric matrices of appropriate dimension. Since the initial conditions (2) may not satisfy (1), they are the limit as $t \to 0$ of the solution $\phi(x, t)$. This PDE is also sometimes called *first order hyperbolic* (if there is no $D_x^2 \phi$ term) or *degenerate parabolic* (if the term involving $D_x^2 \phi$ is not of full rank). Unless $H$ is linear and of full rank in the highest order derivative which is present, even with smooth initial conditions $\phi$ may not remain differentiable and hence (1) will have no classical solution. The algorithms of the toolbox are designed to approximate the viscosity solution [6], which is the appropriate weak solution for many problems that lead to equations of the form (1), although it is not the only possible weak solution.

A key feature of the viscosity solution of (1) is that under suitable conditions $\phi$ remains bounded and continuous for all time. This property may not hold for other types of HJ PDE, such as the stationary (time-independent) equations arising in minimum time to reach problems (see section 2.7). The algorithms in the toolbox make use of the continuity assumption to achieve improved accuracy. The terms presently implemented in the toolbox, and the assumption made about their properties—essentially boundedness and (usually Lipschitz) continuity—are designed to maintain this assumption.

## 1.2   Contents of the Toolbox

General schemes for solving the potentially nonlinear (1) are difficult to implement and often sacrifice speed, accuracy and/or ease of use to achieve convergence. Furthermore, few practical problems require the full generality of this equation. Consequently, the current toolbox implements only a variety of special cases:

$$
\begin{align}
0 =& D_t \phi(x, t) \tag{4} \\
& + v(x, t) \cdot \nabla \phi(x, t) \tag{5} \\
& + a(x, t) \|\nabla \phi(x, t)\| \tag{6} \\
& + \text{sign}(\phi(x, 0))(\|\nabla \phi(x, t)\| - 1) \tag{7} \\
& + H(x, t, \phi, \nabla \phi) \tag{8} \\
& - b(x, t)\kappa(x, t)\|\nabla \phi(x, t)\| \tag{9} \\
& - \text{trace}[\mathbf{L}(x, t)D_x^2 \phi(x, t)\mathbf{R}(x, t)] \tag{10} \\
& + \lambda(x, t)\phi(x, t) \tag{11} \\
& + F(x, t, \phi), \tag{12}
\end{align}
$$

11

subject to constraints

$$D_t\phi(x, t) \geq 0, \qquad\qquad D_t\phi(x, t) \leq 0, \qquad\qquad (13)$$
$$\phi(x, t) \leq \psi(x, t), \qquad\qquad \phi(x, t) \geq \psi(x, t), \qquad\qquad (14)$$

Note that the time derivative (4) and at least one term involving a spatial derivative (5)–(10) must appear, otherwise the equation is not a time-dependent HJ PDE. Numerical approximations for each type of term are provided.

- The time derivative (4) is approximated with an explicit total variation diminishing Runge-Kutta integration scheme with order of accuracy between one and three [24, chapter 3.5]. Because it is an explicit integrator, CFL conditions restrict the size of each timestep. An example is given in section 2.1 and a description of the toolbox routines in section 3.5.

- Motion by a velocity field (5), also called advection or convection. The user provides the velocity field $v : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n$, and the gradient $\nabla\phi(x, t)$ is approximated with an upwind finite difference scheme with order of accuracy between one and five [24, chapter 3]. An example is given in section 2.1, a description of the toolbox routines for upwind finite difference approximations in section 3.4.1, and a description of the toolbox routine for approximating constant velocity flow fields in section 3.6.1.

- Motion in the normal direction (6). The user provides the speed of the interface $a : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}$, and $\nabla\phi(x, t)$ is approximated with an upwind finite difference scheme [24, chapter 6]. An example is given in section 2.3.2 and a description of the toolbox routine in section 3.6.1.

- The reinitialization equation (7). This term is identically zero for signed distance functions, and can be applied to implicit surface functions in order to transform them into signed distance functions [24, chapter 7.4]. A Godunov scheme for its solution can be found in [12, appendix A.3], which allows this term to be stably approximated with a minimum of artificial dissipation. A subcell fix from [27] is optional but is used by default on nodes adjacent to the interface to keep the interface from moving during reinitialization. Note that the initial conditions are used inside the signum function in (7). Examples are given in sections 2.2.1 and 2.10 and a description of the toolbox routine in section 3.6.1. Reinitialization is usually applied as an auxiliary step by itself; a helper routine for this process is described in section 3.7.4.

- A general Hamilton-Jacobi term (8) can treat a variety of applications, including optimal control and differential games. The user provides the analytic Hamiltonian $H : \mathbb{R}^n \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}$. Note that if $H$ depends on $\phi$, the user must ensure that it satisfies the monotonicity requirement (3). Upwind finite difference approximations of $\nabla\phi(x, t)$ are provided, and Lax-Friedrichs is used to stably approximate the $H(x, t, r, p)$

function (with various options for the degree of localization when calculating the arti-
ficial dissipation coefficient) [24, chapter 5]. An example is given in section 2.2.2 and a
description of the toolbox routines in section 3.6.2.

- Motion by mean curvature (9). The user provides the speed $b : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^+$, while
  the mean curvature $\kappa(x, t)$ and gradient $\nabla \phi(x, t)$ are approximated by centered second
  order accurate finite difference approximations [24, chapter 4]. An example is given in
  section 2.3.1, a description of the toolbox routines for centered finite difference approx-
  imations in section 3.4.2, and a description of the toolbox routine for motion by mean
  curvature in section 3.6.3.

- Motion by the trace of the Hessian (10), which arises from the Kolmogorov or Fokker-
  Planck equations when working with stochastic differential equations [13, 22]. The user
  provides the matrices $\mathbf{L}, \mathbf{R} : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^{n \times n}$, while the Hessian matrix of mixed second
  order spatial derivatives $D_x^2 \phi(x, t)$ is approximated by centered second order accurate
  finite difference approximations. This feature has not yet been implemented, but will
  be available in future releases.

- Discounting terms (11), which arise when solving some types of optimal control prob-
  lems [2] or stochastic differential equations [22] (in which context they relate to the
  "killing" process). The user provides the discount factor $\lambda : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}$. This feature
  has not yet been implemented, but will be available in future releases.

- Forcing terms (12), which the user provides $F : \mathbb{R}^n \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. If $F$ depends on $\phi$,
  the user must ensure that it satisfies the monotonicity requirement (3). This feature
  has not yet been implemented, but will be available in future releases.

- Constraints (13) that the implicit surface should not grow or should not shrink. An
  example is given in section 2.2 and a description of the toolbox routine in section 3.6.4.

- Constraints (14) that the implicit surface should not enter or should not exit another
  implicit surface. The user provides $\psi : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}$ defining the other implicit sur-
  face. Unlike most other terms, this constraint is handled in an indirect manner using
  the `postTimestep` option of the time integration routines. The option is discussed in
  section 3.5.3, and an example is given in section 2.2.3.

This collection of terms covers most of the cases arising in applications, although the toolbox
is organized so that adding more types of terms is relatively straightforward.

## 1.3   Using the Toolbox

Running the Toolbox requires MATLAB Version 6.5 or later. Only the base MATLAB is
required; no additional MATLAB toolboxes are used by the kernel code or by any of the

examples in this manual. Assuming that MATLAB is operational on your system, the steps to using the Toolbox of Level Set Methods are to download an archive of the most recent version of the Toolbox from the web site [14] (available in zip or tarball format), unpack the archive, and follow the instructions in the README file.

The best way to start learning how to use the Toolbox is by looking at the examples, in particular the annotated example described in section 2.1. Hopefully, most problems will be similar to one or more of the examples from section 2, so that one of those routines can be modified rather than starting from scratch.

When it comes time to develop code that implements a new application, there are several basic steps that should be followed.

1. Determine the Hamilton-Jacobi equation.

2. Pick out the relevant types of terms from (4)–(14), or implement new ones if necessary.

3. If upwinded approximations of first order derivatives are required, decide on the desired order of accuracy.

4. Provide the other parameters needed by the HJ term approximations (velocities, speeds, matrices, discount factors, etc.).

5. Decide on the desired order of accuracy for the time derivative approximation, and the CFL number.

6. Pick the boundary conditions.

7. Create the grid.

8. Create the initial condition $\phi(x, 0)$.

9. Integrate forward in time, with occasional pauses to display or save the results.


## 1.4   Troubleshooting

Based on the author's experience, common mistakes include:

- Failure to follow the instructions in the README file. If an error of the form `Undefined function or variable 'processGrid'` is encountered when an example is run for the first time, the most likely problem is that MATLAB cannot find the kernel routines because they have not been added to MATLAB's search path using `addpath`. A function `addPathToKernel` is contained in the `Examples/` directory, and it must be edited by the user to provide the *absolute* path to the `Kernel/` directory of the Toolbox before any of the examples can be run (older versions of MATLAB have trouble with relative path references, hence the need to provide an absolute path).

- Too coarse a grid. Static implicit surface functions cannot resolve details of surface features that are smaller than a grid cell. Dynamic evolution of those surfaces using the schemes described here introduces numerical dissipation, so that even features whose size is a few grid cells may be smoothed away. In general, any important features must be at least three to five grid cells wide in each dimension in order for them to be maintained for more than a few timesteps, even when using methods with high order accuracy. In some cases, a sufficiently fine regular grid may be too computationally expensive to evolve and adaptive meshing may be required.

- Poor dimensional scaling. Signed distance functions and the PDE solvers included in this toolbox work best if all the dimensions in the problem are approximately the same size; for example, the grid ranges and cell widths should be within an order of magnitude of one another. If dimensions involve widely different scales—such as radians and thousands of feet—then the problem parameters should be scaled to bring the dimensional ranges closer together. Care must be taken in this process to ensure that all ranges, dynamics and other parameters (such as bounds on partial derivative magnitudes) are scaled by the same amount.

- Incorrect initialization. If no implicit surface can be seen at $t = 0$, two quick checks should be performed. First, make sure that the desired implicit surface falls within the bounds of the computational grid (as defined by the structure members `grid.min` and `grid.max`). Second, make sure that the desired implicit surface is at least two grid cells wide in each dimension (the width of a grid cell is given by the structure member `grid.dx`).

- Numerical instability. The level set function may become highly oscillatory, a behavior which manifests itself by the sudden appearance of many convoluted looking surfaces in two dimensional contour or three dimensional isosurface plots. Instability can be caused by buggy boundary conditions, poor dimensional scaling, incorrect CFL restrictions (for example, if the bounds on the partial derivative of the Hamiltonian are too small when solving a problem with a general HJ term (8)), or bugs in the kernel.

- Sign problems. If the surface seems to be moving in the wrong direction, try switching the sign of the flow.

- Mixing up `ndgrid`-based and `meshgrid`-based grids. The Toolbox *always* uses `ndgrid`-based grids because they can be used in any dimension and because the data they generate is more consistent when exported to other applications. Unfortunately, some routines in MATLAB require the incompatible `meshgrid`-based grids. Even more dangerously, many of MATLAB's visualization routines silently default to `meshgrid`-based grids when no spatial grid argument is provided. Consequently, the grid arguments should *always* be specified with `grid.xs` for any MATLAB visualization routine. For example, the correct call is `surf(grid.xs{1}, grid.xs{2}, data)`. The incorrect call `surf(data)` will silently create a `meshgrid`-based array with indices $1, \ldots,$ `grid.N`$(d)$

for each $d = 1, \ldots, $ `grid.dim`, and may flip some dimensions. The accidental and silent use of `meshgrid` can lead to visualizations with distortions that are very hard to spot; in particular, it plays havoc with the vector fields displayed by `quiver`. See section 3.1.3 for more details and for a routine that converts between `ndgrid`-based and `meshgrid`-based grids.

## 1.5  Advanced Tips for the Toolbox

We heartily endorse attempts to modify the toolbox, add to it, or use some of its more advanced features (such as general Hamilton-Jacobi terms); however, we do have some recommendations.

- Start with a simplified example that is known to work, and add features incrementally with tests until the full version is achieved.

- Start with low order accurate approximations on a reasonably coarse grid. If it works, improve the accuracy. Often it is more efficient to increase the order of accuracy of the approximations than to refine the grid. On the other hand, for most examples the biggest improvement is seen when moving from first to second order of accuracy; additional orders of accuracy may not be worth the computational effort if only qualitative or visual error is important.

- Learn how to use MATLAB's debugging and visualization systems. One of the reasons that structures were used extensively in this version (rather than full blown classes) was to allow their contents to be examined easily during debugging at any level of the stack. Furthermore, the ability to produce contour and isosurface plots at the debugger command line makes debugging of two and three dimensional code merely unpleasant, instead of virtually impossible.

- Learn MATLAB's cell arrays (arrays written with "{}" instead of "()"). In order to create dimensionally independent code, cell arrays were used extensively in the kernel code. In particular, if `data` is an $n$ dimensional (regular) array and `indices` is a cell vector of length $n$ (a two dimensional cell array of size $n \times 1$) each element of which is a regular vector, then the syntax `data(indices{:})` can be used to pick out subsets and slices of `data`. For example, if `data = rand([10 10 10])` and `indices = { 2:9; 4:6; 5 }`, then `data(indices{:}) = data(2:9,4:6,5)`. More generally, the notation `indices{:}` turns the elements of the cell array `indices` into a comma separated list that can be used either to index into an array or as the parameter list for a function; for example, to call `interpn` in a dimensionally independent way. Another very useful function for cell arrays is MATLAB's `deal`; for example, the help text of `deal` shows how to collect the comma separated list of parameters returned by a function into a single cell array.

16

- Learn how to vectorize in the MATLAB sense. Despite working in MATLAB's interpreted programming environment, this toolbox can achieve nearly the performance of compiled code. In order to achieve this performance, it is important never to loop explicitly over the elements of the data array. Instead, all operations on the data array are written as element-wise sums, products (".*") and logical comparisons. The result is not as memory efficient as could be achieved in a carefully constructed compiled code, but it is far better than explicit loops.

- Tell us if you find a repeatable bug.

## 1.6   Other Publications about the Toolbox

Publications currently available describing the Toolbox and/or its applications include:

- Applications in control, simulation and verification of continuous and hybrid systems [20].

- Design and implementation so that the Toolbox can easily be used to test new level set schemes in the field of numerical analysis [16].

Although there is some slight overlap between these papers and this manual, most of the material in these papers is complementary to the material covered here. These other papers may also include schemes and/or examples which are not part of the base Toolbox download discussed here. In those cases, code is available as separate downloads from the Toolbox web site [14].

## 1.7   Citing the Toolbox

It may seem a bit silly to discuss citations, a cross referencing system designed for paper publications, when talking about software available on the Internet. Nevertheless, citations matter to the author's academic career. Consequently, users (particularly those in academia) are encouraged to provide a bibliographic entry in articles which make use of the Toolbox, because alternative methods of attribution—such as plain text footnotes or the Toolbox's URL—are not so easily identified by automated cross-referencing systems.

Users should cite whichever publication is most appropriate to their application from among those available [16, 18, 20]. Thanks!

One additional note about citations. Almost all of the schemes in the original version of the Toolbox were drawn from [24]. In the rush to produce documentation, references to the primary literature on these schemes was mostly omitted. These omissions are definitely a

bug in the documentation, and hence reports of missing citations are welcome. Like all bug reports, it is important to provide as much information as possible; for missing citations, that means specific papers. As the Toolbox is updated, citations to the primary literature will be added for both the new schemes and the old.

# 2  Level Set Examples

Our examples fall into three categories: those that are motivated by specific examples taken from papers or texts, those that demonstrate the basic capabilities of the toolbox, and those designed to test aspects of the implementation. The code implementing most of the examples in the former two categories follows a similar structures, so as a starting point, we provide an extensively annotated script file which shows how to implement motion by an external velocity field.

The first step to running the examples described in this section is to modify the script file `Examples/addPathToKernel` so that it contains the *absolute* path name for the `Kernel` directory. The absolute path name is required because current versions of MATLAB appear unable to create function handles involving relative path names. Once this modification is performed, it should be possible to enter into any of the example subdirectories, start MATLAB, and execute one of the examples by typing its name at the MATLAB prompt.

## 2.1  Getting Started: Convective Motion (5)

In this section we examine in detail how to implement motion by an external velocity field (5) using the file `Examples/Basic/convectionDemo`. The implementation of many of the other examples follows the same basic framework.

[ data, g, data0 ] = convectionDemo(flowType, accuracy, displayType): Demonstrate motion by an external velocity field. The three input parameters are strings; the options for the first two are explained in the help text and the options for `displayType` come directly from the helper routine `visualizeLevelSet`. All three input parameters are optional. The returned parameters are the final $\phi(x, t_{\max})$ function `data`, the computational grid `g` and the initial $\phi(x, 0)$ function `data0`.

Figure 1 shows the results of running `convectionDemo('linear', 'medium')`. Beyond the three input parameters, there are many other options to the way this example runs and is displayed. These options can be easily modified by editing the source of `convectionDemo` directly.

- Initial and final time.
- Whether to display intermediate results. If so, how many intermediate results, whether to display results in a single figure or as a sequence of subplots, whether to pause between visualizations, and whether to remove visualizations from previous timesteps before displaying the next.

19
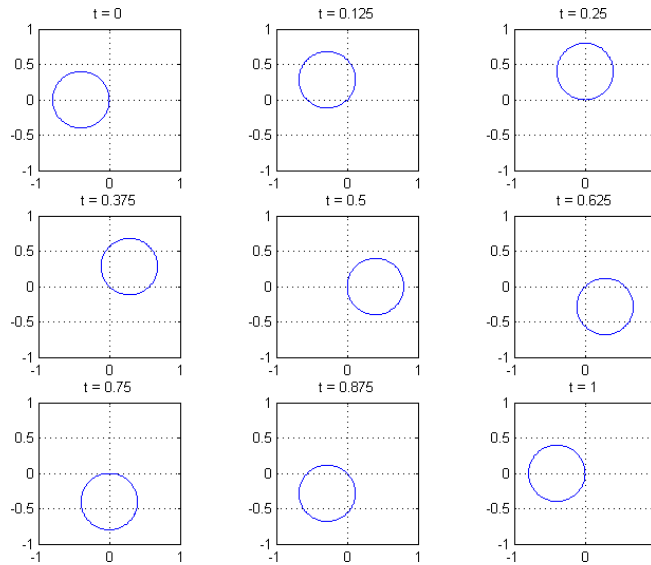
Figure 1: Result of running convectionDemo('linear', 'medium'). Shows motion by a constant rotational external velocity field.

- Grid parameters: dimension, resolution, periodic or extrapolating boundary conditions.

- Details of the velocity field.

- Shape and location of the initial surface.

For more details, see the commentary below. Increasing accuracy will increase execution time. Table 1 shows the execution times for each of the accuracy options with flowType = 'linear'. In order to get better resolution of the execution time, the grid resolution was doubled to g.dx = 0.01 (see below for details on how to make this change). The computational platform was a Pentium 4 with plenty of memory running Matlab 6.5 in Windows XP Professional. Examining the figures, the low accuracy run had clearly lost area by the end of the full rotation (at $t_{\max}$) but the remaining choices were visually indistinguishable. A quantitative error comparison will be performed when somebody has the time to write the scripts.

We now examine the components of the source code for convectionDemo. Notice that most of the file is concerned with initialization, since the toolbox and MATLAB handle the real work.

```
1    function [ data, g, data0 ] = convectionDemo(flowType, accuracy, displayType)
2    % convectionDemo: demonstrate a simple convective flow field.
```

| Accuracy Parameter | Temporal Accuracy | Spatial Accuracy | Execution Time seconds | relative |
|---|---|---|---|---|
| low | 1 | 1 | 140 | 1 |
| medium | 2 | ENO 2 | 684 | 5 |
| high | 3 | ENO 3 | 2433 | 17 |
| very high | 3 | WENO 5 | 2585 | 18 |

Table 1: Execution time for convectionDemo('linear', accuracy) with the various choices of accuracy on a $101^3$ grid with extrapolated boundary conditions.

```
 3    %
 4    % [ data, g, data0 ] = convectionDemo(flowType, accuracy, displayType)
 5    %
 6    % This function was originally designed as a script file, so most of the
 7    %   options can only be modified in the file.
 8    %
 9    % For example, edit the file to change the grid dimension, boundary conditions,
10    %   flow field parameters, etc.
11    %
12    % Parameters:
13    %
14    %   flowType      String to specify type of flow field.
15    %                   'constant'    Constant flow field xdot = k (default).
16    %                   'linear'      Linear flow field xdot = A x.
17    %                   'constantRev' Constant flow field, negate at t_half.
18    %                   'linearRev'   Linear flow field, negate at t_half.
19    %   accuracy      Controls the order of approximations.
20    %                   'low'         Use odeCFL1 and upwindFirstFirst (default).
21    %                   'medium'      Use odeCFL2 and upwindFirstENO2.
22    %                   'high'        Use odeCFL3 and upwindFirstENO3.
23    %                   'veryHigh'    Use odeCFL3 and upwindFirstWENO5.
24    %   displayType   String to specify how to display results.
25    %                   The specific string depends on the grid dimension;
26    %                   look at the helper visualizeLevelSet to see the options
27    %                   (optional, default depends on grid dimension).
28    %
29    %   data          Implicit surface function at t_max.
30    %   g             Grid structure on which data was computed.
31    %   data0         Implicit surface function at t_0.
32
33    % Ian Mitchell, 2/9/04
34
35    %---------------------------------------------------------------------------
36    % You will see many executable lines that are commented out.
37    %   These are included to show some of the options available; modify
```

```
38    %    the commenting to modify the behavior.
39
```

Standard opening comments, including the help text. The blank line 32 ensures that sub-
sequent comment lines are not included in the help entry. Notice the options for input
parameters `flowType` and `accuracy`.

```
40    %---------------------------------------------------------------------------
41    % Make sure we can see the kernel m-files.
42    run('../addPathToKernel');
43
```

To make sense of the function calls and function handles encountered in the remainder of the
file, the kernel directories must be on MATLAB's path. The script `Examples/addPathToKernel`
adds the `Kernel` directory and all its subdirectories to MATLAB's path if they are not already
present (so repeated executions of `addPathToKernel` are safe). We use the functional form
of `run` in order to access the parent directory.

```
44    %---------------------------------------------------------------------------
45    % Integration parameters.
46    tMax = 1.0;                     % End time.
47    plotSteps = 9;                  % How many intermediate plots to produce?
48    t0 = 0;                         % Start time.
49    singleStep = 0;                 % Plot at each timestep (overrides tPlot).
50
51    % Period at which intermediate plots should be produced.
52    tPlot = (tMax - t0) / (plotSteps - 1);
53
54    % How close (relative) do we need to get to tMax to be considered finished?
55    small = 100 * eps;
56
57    %---------------------------------------------------------------------------
58    % What level set should we view?
59    level = 0;
60
61    % Pause after each plot?
62    pauseAfterPlot = 0;
63
64    % Delete previous plot before showing next?
65    deleteLastPlot = 0;
66
67    % Plot in separate subplots (set deleteLastPlot = 0 in this case)?
68    useSubplots = 1;
69
```
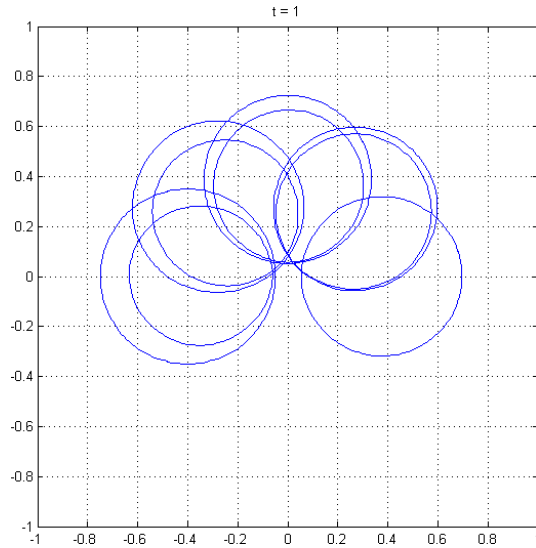
Figure 2: Result of running `convectionDemo('linearRev', 'low')` with internal parameter `useSubplots = 0`. Shows rigid body rotation about the origin, clockwise for the first half of the simulation and then counter clockwise for the remainder. The loss of area associated with using low accuracy methods is obvious from the fact that the two sets of circles do not overlap.

All of these parameters are meant to be modified by the user except `tPlot` and `small`. The difference `tMax − t0` controls the length of the simulation, and `tMax/2` is the time at which the time dependent flow fields `constantRev` and `linearRev` reverse directions (see below). The number of intermediate plots includes the plots of the initial and final conditions, so choose `plotSteps ≥ 2`. The time between plots is controlled by `tPlot` and depends on the length of the simulation and the number of plots. The parameter `small` takes care of the fact that the final timestep often comes up a little short of the final time, but so close that taking another timestep is not worth the effort. The boolean parameter `singleStep` can be turned on to force visualization of the surface after every CFL constrained timestep. It is mostly useful for debugging, and we recommend choosing `deleteLastPlot = 1` and `useSubplots = 0` if you choose `singleStep = 1`. If `useSubplots = 0`, then all visualizations are done in a single full figure axis. Figure 2 shows the results of running `convectionDemo('linearRev','low')` when the source is modified to set the internal parameter `useSubplots = 0`. The parameter `level` chooses which isosurface of $\phi$ is visualized when using contour plots (in 2D) or surfaces (in 3D).

```
70    %---------------------------------------------------------------------------
71    % Use periodic boundary conditions?
72    periodic = 0;
```
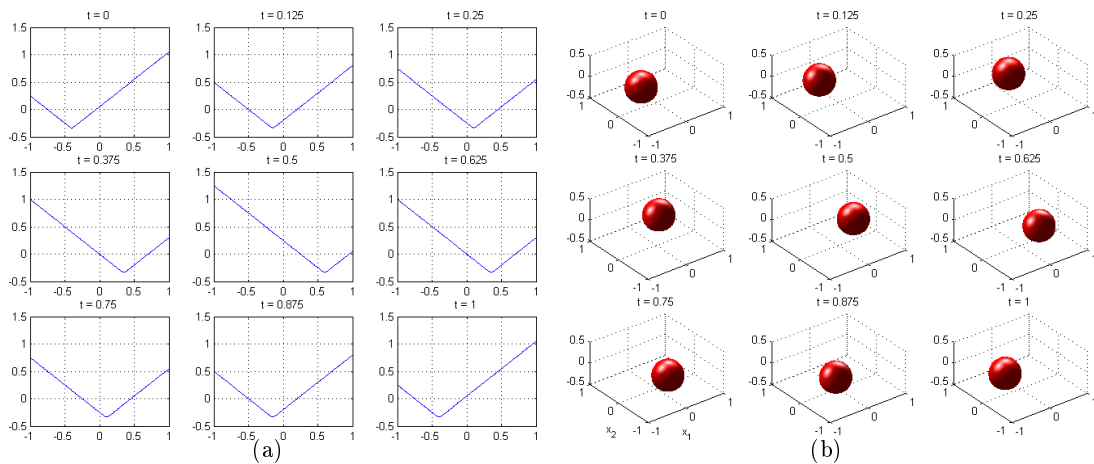
Figure 3: Running `convectionDemo` in other dimensions by modifying the internal parameter `g.dim`. These are not exactly the figures generated during the run: the subplots generated during the run have had their axis bounds adjusted to be consistent across all nine subplots in each case. Figure 3(a): The implicit surface function $\phi$ for a one dimensional example run by `convectionDemo('constantRev', 'veryHigh')`. Figure 3(b): An isosurface plot for a three dimensional example run by `convectionDemo('linear', 'medium')`.

```
73
74   % Create the grid.
75   g.dim = 2;
76   g.min = -1;
77   g.dx = 1 / 50;
78   if(periodic)
79     g.max = (1 - g.dx);
80     g.bdry = @addGhostPeriodic;
81   else
82     g.max = +1;
83     g.bdry = @addGhostExtrapolate;
84   end
85   g = processGrid(g);
86
```

This block of code creates the computational grid. The user may modify the boolean flag `periodic` to choose whether periodic or extrapolation boundary conditions are used (or choose something else by setting `g.bdry`). Dimension is set with `g.dim` and resolution with `g.dx`. Since all dimensions have the same resolution, bounds and boundary conditions, it is only necessary to store scalars and single function handles in the fields. The call to `processGrid` automatically extends all fields (except `g.dim`) to their full vector length. Miss-

ing fields are given inferred values (such as `g.N`) or defaults (such as `g.bdryData`). Figure 3 shows the results of running this example in dimensions one and three.

```
87   %---------------------------------------------------------------------------
88   % Most of the time in constant flow case, we want flow in a
89   %   distinguished direction, so assign first dimension's flow separately.
90   constantV = 0 * ones(g.dim);
91   constantV(1) = 2;
92   constantV = num2cell(constantV);
93
94   % Create linear flow field xdot = A * x
95   linearA = 2 * pi * [ 0 1 0 0; -1 0 0 0; 0 0 0 0; 0 0 0 0 ];
96   %linearA = eye(4);
97   indices = { 1:g.dim; 1:g.dim };
98   linearV = cellMatrixMultiply(num2cell(linearA(indices{:})), g.xs);
99
```

Flow fields are defined by cell vectors. Element $i$ of the cell vector gives the motion in the $i^{th}$ dimensions. Element $i$ can be either a scalar—if the flow field does not depend on $x$—or an array of size `grid.shape`, each element of which gives the motion in dimension $i$ for the corresponding node of the grid. While MATLAB has many ways to generate regular vectors, matrices and arrays, there are few ways to similarly populate cell arrays. This block of code demonstrates a few, including the very useful `num2cell`.

The constant flow field $v(x) = $ `constantV` demonstrates a spatially independent flow field, in this case a flow field with speed two along the first dimension. The linear flow field $v(x) = \mathbf{A}x = $ `linearV` demonstrates the spatially dependent flow field. In order to allow for variable dimension, the array $\mathbf{A} = $ `linearA` is defined up to dimension 4. Line 95 provides a definition of $\mathbf{A}$ which generates rotation about the origin in the $x_1$-$x_2$ plane. Line 96 can be uncommented to generate an exponentially growing surface. The magic is performed in line 98, where `cellMatrixMultiply` computes $\mathbf{A}x$ at every node $x$ in the grid. In particular, the appropriate g.dim $\times$ g.dim subset of `linearA` is picked out by `indices{:}`, which turns the `indices` cell vector into a comma separated list that can be used as an argument to a function or (in this case) an index into an array. This "{:}" construction is used extensively throughout the toolbox to provide dimensionally independent code.

```
100  %---------------------------------------------------------------------------
101  if(nargin < 1)
102    flowType = 'constant';
103  end
104
105  % Choose the flow field.
106  switch(flowType)
```

```
107
108    case 'constant'
109     v = constantV;
110
111    case 'linear'
112     v = linearV;
113
114    case 'constantRev'
115     v = @switchValue;
116     schemeData.one = constantV;
117     schemeData.two = cellMatrixMultiply(-1, constantV)
118     schemeData.tSwitch = 0.5 * tMax;
119
120    case 'linearRev'
121     v = @switchValue;
122     schemeData.one = linearV;
123     schemeData.two = cellMatrixMultiply(-1, linearV)
124     schemeData.tSwitch = 0.5 * tMax;
125
126    otherwise
127     error('Unknown flowType %s', flowType);
128
129   end
130
```

This block of code picks out which velocity field will be used in the run. The default flow field is determined by line 102. The first two cases of flow field 'constant' and 'linear' are straightforward, and show how to create a time independent flow field using a constant cell vector. For time dependent flow fields, a function handle is passed instead. The function switchValue is described below. It requires that the schemeData structure have some additional fields beyond those required by termConvection: one, two, and tSwitch (these additional fields will be ignored by termConvection). Note the use of cellMatrixMultiply with a scalar parameter to reverse the direction of the flow fields for the second half of the simulation.

```
131   %---------------------------------------------------------------------------
132   % What kind of display?
133   if(nargin < 3)
134     switch(g.dim)
135      case 1
136       displayType = 'plot';
137      case 2
138       displayType = 'contour';
139      case 3
140       displayType = 'surface';
```

```
141      otherwise
142        error('Default display type undefined for dimension %d', g.dim);
143      end
144    end
```

The default visualization style for each of dimensions 1–3 is set by this block of code. While the toolbox is almost entirely dimensionally independent, and the version of convectionDemo described here will work computationally for dimensions up to four, visualization is challenging for dimensions greater than three.

```
145    %---------------------------------------------------------------------------
146    % Create initial conditions (a circle/sphere).
147    %   Note that in the periodic BC case, these initial conditions will not
148    %   be continuous across the boundary unless the circle is perfectly centered.
149    %   In practice, we'll just ignore that little detail.
150    center = [ -0.4; 0.0; 0.0; 0.0 ];
151    radius = 0.35;
152    data = zeros(size(g.xs{1}));
153    for i = 1 : g.dim
154      data = data + (g.xs{i} - center(i)).^2;
155    end
156    data = sqrt(data) - radius;
157    data0 = data;
158
```

The initial conditions are a sphere in dimension grid.dim of radius radius centered at center. Note the vectorized use of g.xs to determine the initial implicit surface function (in fact, this is a signed distance function).

```
159
160    %---------------------------------------------------------------------------
161    if(nargin < 2)
162      accuracy = 'low';
163    end
164
165    % Set up spatial approximation scheme.
166    schemeFunc = @termConvection;
167    schemeData.velocity = v;
168    schemeData.grid = g;
169
170    % Set up time approximation scheme.
171    integratorOptions = odeCFLset('factorCFL', 0.5, 'stats', 'on');
172
173    % Choose approximations at appropriate level of accuracy.
```

```
174    switch(accuracy)
175     case 'low'
176      schemeData.derivFunc = @upwindFirstFirst;
177      integratorFunc = @odeCFL1;
178     case 'medium'
179      schemeData.derivFunc = @upwindFirstENO2;
180      integratorFunc = @odeCFL2;
181     case 'high'
182      schemeData.derivFunc = @upwindFirstENO3;
183      integratorFunc = @odeCFL3;
184     case 'veryHigh'
185      schemeData.derivFunc = @upwindFirstWENO5;
186      integratorFunc = @odeCFL3;
187     otherwise
188      error('Unknown accuracy level %s', accuracy);
189    end
190
191    if(singleStep)
192      integratorOptions = odeCFLset(integratorOptions, 'singleStep', 'on');
193    end
194
```

This block sets up function handles for both the spatial approximation scheme `schemeFunc` and the time integration scheme `integratorFunc`. The default accuracy is determined by line 162. The meaning of each level of accuracy is determined by the `switch`/`case` statement. The flow field information which was determined earlier is stored into `schemeData.velocity`. In line 192, notice that an existing `odeCFLn` option structure is modified if single stepping has been requested.

```
195    %---------------------------------------------------------------------------
196    % Initialize Display
197    f = figure;
198
199    % Set up subplot parameters if necessary.
200    if(useSubplots)
201      rows = ceil(sqrt(plotSteps));
202      cols = ceil(plotSteps / rows);
203      plotNum = 1;
204      subplot(rows, cols, plotNum);
205    end
206
207    h = visualizeLevelSet(g, data, displayType, level, [ 't = ' num2str(t0) ]);
208
209    hold on;
210    if(g.dim > 1)
211      axis(g.axis);
```

```
212      daspect([ 1 1 1 ]);
213    end
214
```

This block of code performs basic display initialization. If subplots have been requested, the layout of the subplot array must be determined. Before calling `visualizeLevelSet` to perform the actual visualization, we make current the appropriate figure axis with either `figure` or `subplot`. The current time is passed in a string for use as the title of the figure. As a side effect, `visualizeLevelSet` will finish with a call to `drawnow` to ensure that the results are shown before computation proceeds. Because this call to `drawnow` is performed before the modifications in lines 211–212, they may not be immediately visible.

```
215    %---------------------------------------------------------------------------
216    % Loop until tMax (subject to a little roundoff).
217    tNow = t0;
218    startTime = cputime;
219    while(tMax - tNow > small * tMax)
220
221      % Reshape data array into column vector for ode solver call.
222      y0 = data(:);
223
224      % How far to step?
225      tSpan = [ tNow, min(tMax, tNow + tPlot) ];
226
227      % Take a timestep.
228      [ t y ] = feval(integratorFunc, schemeFunc, tSpan, y0,...
229                        integratorOptions, schemeData);
230      tNow = t(end);
231
232      % Get back the correctly shaped data array
233      data = reshape(y, g.shape);
234
```

This is the heart of the simulation, where all of the work is accomplished. Integration of the underlying PDE is accomplished entirely by lines 228–229. Lines 222 and 233 massage the array `data` that stores the implicit surface function $\phi$ into the shape required by the integrator functions `integratorFunc = @odeCFLn` and back again. Lines 219, 225 and 230 keep track of the passage of simulation time.

```
235      if(pauseAfterPlot)
236        % Wait for last plot to be digested.
237        pause;
238      end
```

```
239
240    % Get correct figure, and remember its current view.
241    figure(f);
242    figureView = view;
243
244    % Delete last visualization if necessary.
245    if(deleteLastPlot)
246      delete(h);
247    end
248
249    % Move to next subplot if necessary.
250    if(useSubplots)
251      plotNum = plotNum + 1;
252      subplot(rows, cols, plotNum);
253    end
254
255    % Create new visualization.
256    h = visualizeLevelSet(g, data, displayType, level, [ 't = ' num2str(tNow) ]);
257
258    % Restore view.
259    view(figureView);
260
261  end
262
263  endTime = cputime;
264  fprintf('Total execution time %g seconds', endTime - startTime);
265
266
267
```

These remaining lines complete the `while` loop that manages simulation time and the `convectionDemo` function as a whole. They are devoted to visualization.

```
268    %---------------------------------------------------------------------------
269    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
270    %---------------------------------------------------------------------------
271    function out = switchValue(t, data, schemeData)
272    % switchValue: switches between two values.
273    %
274    %   out = switchValue(t, data, schemeData)
275    %
276    % Returns a constant value:
277    %            one      for t <= tSwitch;
278    %            two      for t >  tSwitch.
279    %
280    % By setting one and two correctly, this function can implement
```

```
281   %    the velocityFunc prototype for termConvection;
282   %    the scalarGridFunc prototype for termNormal, termCurvature and others;
283   %    and possibly some other prototypes...
284   %
285   % Parameters:
286   %    t              Current time.
287   %    data           Level set function.
288   %    schemeData     Structure (see below).
289   %
290   %    out            Either schemeData.one or schemeData.two.
291   %
292   % schemeData is a structure containing data specific to this type of
293   %    term approximation.  For this function it contains the field(s)
294   %
295   %    .one           The value to return for t <= tSwitch.
296   %    .two           The value to return for t >  tSwitch.
297   %    .tSwitch       The time at which the switch between flow fields occurs.
298   %
299   % schemeData may contain other fields.
300
301      checkStructureFields(schemeData, 'one', 'two', 'tSwitch');
302
303      if(t <= schemeData.tSwitch)
304        out = schemeData.one;
305      else
306        out = schemeData.two;
307      end
308
```

This subfunction `switchValue` within `convectionDemo` is an example of a function satisfying the `velocityFunc` prototype for the term approximation `termConvection` (see section 3.6.1). It implements a time dependent flow field by choosing one of two constant flow fields based on the current time. This simple time dependent function also satisfies the `scalarGridFunc` prototype (assuming that `schemeData.one` and `schemeData.two` are set appropriately), and is used in the examples `normalStarDemo` and `curvatureStarDemo` in section 2.3. Much more complex time dependent velocity fields are possible with this framework.


## 2.2   Basic Examples

This section discusses functions found in the directory `Examples/Basic`. This directory provides an example for each of the types of spatial terms (7)–(14) with the exception of motion by mean curvature (9). Examples for the omitted terms can be found elsewhere: section 2.1 for motion by a constant velocity field (5) and section 2.3 for motion in the normal direction (6) and motion by mean curvature (9). Since terms (11)–(14) do not include a

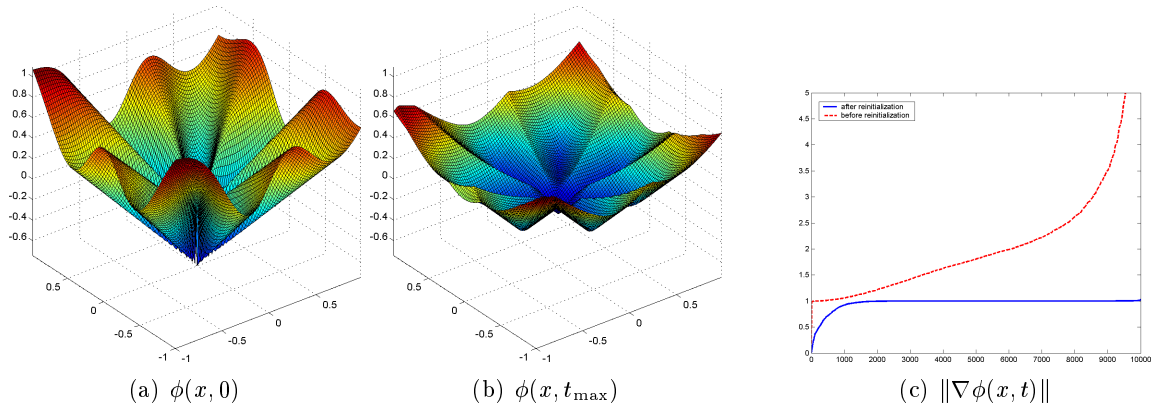(a) $\phi(x,0)$        (b) $\phi(x,t_{\max})$        (c) $\|\nabla\phi(x,t)\|$

Figure 4: Comparing initial (left) and final (center) implicit surface functions for `reinitDemo('star', 'medium', 'surf')`. The right figure shows how the magnitude of the gradient of the final $\phi(x,t_{\max})$ (solid line) is much more consistent and close to unity than that of the initial $\phi(x,0)$ (dashed line). All of these figures were generated with the subcell fix.

spatial derivative, examples for these terms naturally include a combination with other types of term.

### 2.2.1 The Reinitialization Equation (7)

This section describes the function `Examples/Basic/reinitDemo`.

Reinitialization is the process of converting an implicit surface function into a signed distance function—modifying $\phi$ such that $\|\nabla\phi\| \approx 1$ without moving its zero isosurface. Reinitialization was first proposed in [5], and the PDE formulation now called the reinitialization equation was introduced by [33]. The reinitialization equation is a general HJ PDE with spatial term (7), and unlike other methods for reinitialization it does not require explicitly locating the current zero isosurface of $\phi$. The numerical approximation used in the Toolbox for the reinitialization equation is a Godunov scheme from [12, appendix A.3]. By default a first order accurate subcell fix from [27] is applied to the nodes adjacent to the zero isosurface in order to reduce movement of that isosurface during reinitialization, but this fix is optional. For more details on the fix and additional reinitialization examples, see section (2.10).

Under normal circumstances the reinitialization equation is solved in a pseudo-time that takes place between timesteps of the regular dynamic surface integration by an auxiliary integration routine that hides the details; for example, see `signedDistanceIterative` in section 3.7.4 and `reinitTest` in section 2.11.3. However, for the purposes of demonstrating and testing the term approximation function `termReinit`, we provide the following routine.

(a) $\{x \mid \phi(x,t) = 0\}$ (with subcell fix)   (b) $\{x \mid \phi(x,t) = 0\}$ (without subcell fix)
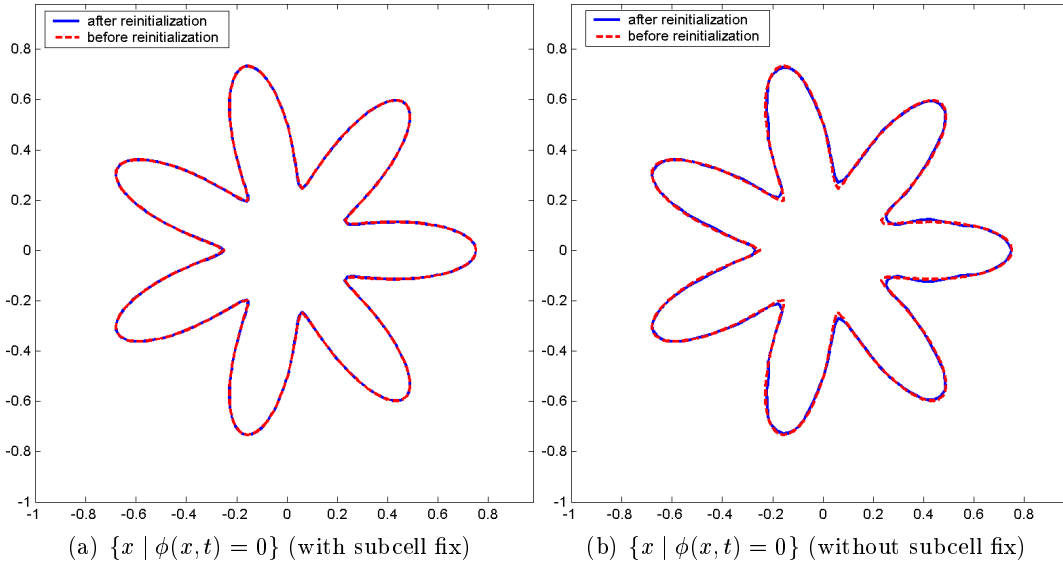
Figure 5: Examining the effect of reinitialization on the implicit surface (the zero isosurface of $\phi(x,t)$). There is much less movement of the interface with the first order accurate subcell fix from [27] than without.

[ data, g, data0 ] = reinitDemo(initialType, accuracy, displayType): Demonstrate the reinitialization equation. The three input parameters are strings; the last two are the same as for convectionDemo. The initialType can be either 'circle' (an off center circle) or 'star' (a centered seven pointed star). All three input parameters are optional. The returned parameters are the final $\phi(x, t_{\max})$ function data, the computational grid g and the initial $\phi(x,0)$ function data0. Edit the file to modify the grid size or to choose not to apply the subcell fix.

The internals of reinitDemo are virtually identical to convectionDemo, so we discuss them no further here.

In the 'circle' case, the initial implicit surface function for an off center circle is not a signed distance function because of the periodic boundary conditions. In the 'star' case, the initial implicit surface function does not have unit magnitude gradient (see (16) in section 2.3 for the initial implicit surface equation). Figure 4 shows the results for the 'star' case, including how the reinitialization procedure successfully adjusts the gradient magnitude to be close to unity. Figure 5 shows the effect of reinitialization on the zero isosurface. Without the subcell fix from [27], there is a small but visually identifiable movement; with the fix the interface appears to be stationary. These results were calculated on a relatively coarse grid (g.dx = 0.02) using accuracy = 'medium'.

A script file `reinitDemoFigures` is provided, which runs `reinitDemo` and generates most of the plots for figures 4 and 5. To generate the contour plot without the subcell fix in figure 5, `reinitDemo` must be edited and run a second time.

### 2.2.2 General HJ Terms (8)

This section describes the function `Examples/Basic/laxFriedrichsDemo`.

General Hamilton-Jacobi equations are challenging but useful in a wide variety of applications. In this section we look at how convective motion can be formulated as a general HJ, which is perhaps the simplest example of such equations. Since the methods for general HJ generally require the addition of artificial dissipation, this formulation is not usually appropriate for convective flow; instead, the specialized upwinded convection schemes should be used (see the example in section 2.1). More ambitious examples of general HJ can be found in sections 2.5 and 2.6.

`[ data, g, data0 ] = laxFriedrichsDemo(flowType, initShape, accuracy, dissType, displayType)`:Demonstrate an implementation of time independent convective flow using a general HJ solver. The four input parameters are strings. The parameters `accuracy` and `displayType` have the same options as the identically named parameters of `convectionDemo`. The parameter `flowType` allows the time-independent flow fields permitted by `convectionDemo`. The parameter `initShape` specifies the shape of the initial implicit surface. The parameter `dissType` specifies which of the types of artificial dissipation functions to use to stabilize the Lax-Friedrichs solver. All five input parameters are optional. The returned parameters are the final $\phi(x, t_{\max})$ function `data`, the computational grid `g` and the initial $\phi(x, 0)$ function `data0`.

The internals of `laxFriedrichsDemo` are the same as `convectionDemo`, with the exception that functions for the prototypes `hamFunc` and `partialFunc` must be provided. In addition, it demonstrates the use of `termLaxFriedrichs` and the routines implementing the `dissFunc` prototype: `artificialDissipationGLF`, `artificialDissipationLLF`, and `artificialDissipationLLLF`.

Formulating convection by flow field $v(x)$ as a general HJ leads to Hamiltonian

$$H(x, p) = v(x) \cdot p$$

This simple dot product is calculated by the subfunction `laxFriedrichsDemoHamFunc` (found in the file `laxFriedrichsDemo`), which implements the `hamFunc` prototype. To scale the dissipation, we need

$$\alpha_i(x) = \max_p \left| \frac{\partial H(x, p)}{\partial p_i} \right| = |v_i(x)|. \tag{15}$$
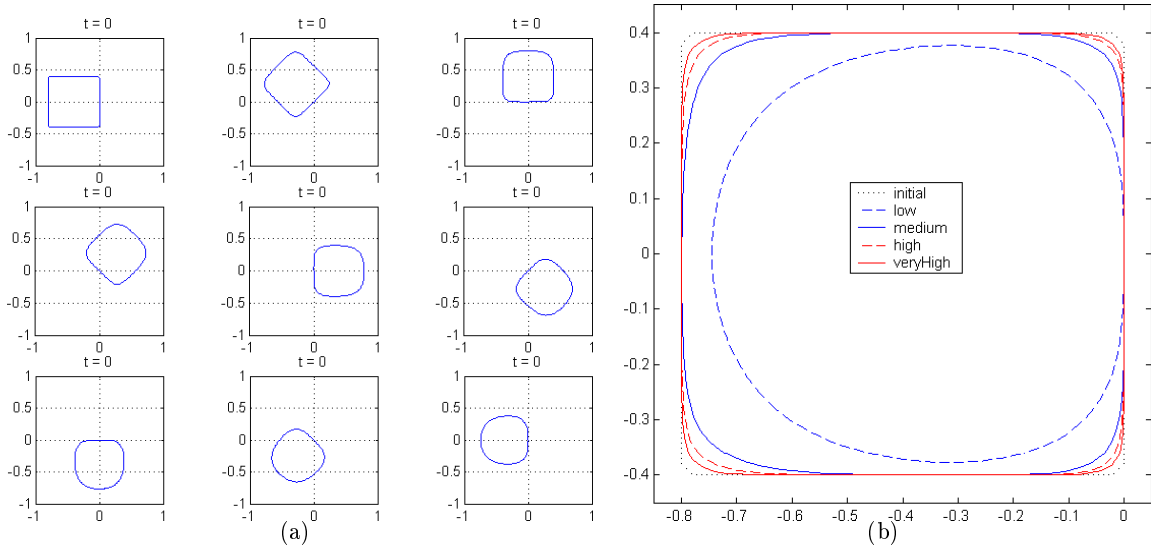
34

Figure 6: Using Lax-Friedrichs to approximate rotational convective flow in two dimensions with `laxFriedrichsDemo`. Figure 6(a): The individual time steps for `laxFriedrichsDemo('linear', 'cube', 'low')`. Figure 6(b): Comparing the final implicit surface calculated by Lax-Friedrichs when using approximation schemes of different accuracies. Note that the results of this example are independent of the artificial dissipation scheme chosen (so the default `dissType = 'global'` was used).

This optimization over partials is performed by the subfunction `laxFriedrichsDemoPartialFunc`, which implements the `partialFunc` prototype. Note that the partials of $H$ with respect to $p$ are independent of $p$; consequently the range of $p$ in the maximization is irrelevant and the different types of dissipation function (chosen by the parameter `dissType` of `laxFriedrichsDemo`) will all produce the same results.

Do not be fooled by the simplicity of these `hamFunc` and `partialFunc` examples. Usually they are much more difficult to compute. In most interesting cases the partial derivative of $H$ with respect to $p$ will depend on $p$ (otherwise the Hamiltonian represents a convective flow field), so the maximization in (15) will be nontrivial. Fortunately, it can be overapproximated if the optimization is too challenging, at the cost of additional dissipation. For more details, see section 3.6.2.

Figure 6 shows the results of running this example in two dimensions for a rigid body rotation of a square. The dissipation which smooths away the corners of the square has two sources: errors in the calculation of the first derivative and the Lax-Friedrichs' artificial dissipation term. By using an approximation scheme of higher order accuracy, the former can be reduced. The approximate execution time (relative to `accuracy = 'low'`) for the four schemes were:
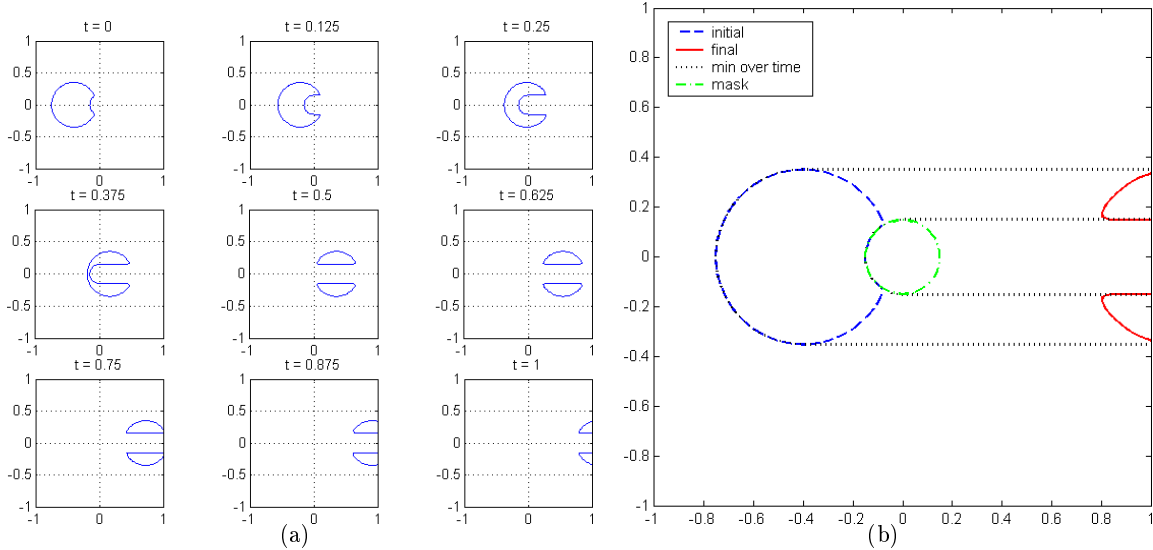
Figure 7: Applications of the `PostTimestep` option of `odeCFLn` to a convection example. Figure 7(a) shows how a constraint of the form (14) can be used to mask out a portion of the state space into which the evolving set cannot enter. Figure 7(b) shows the masking set, as well as $\min_t \phi(x, t)$, which is computed and recorded inside the `PostTimestep` function `maskAndKeepMin`.

'`low`' $= 1$, '`medium`' $= 4$, '`high`' $= 12$ and '`veryHigh`' $= 17$.

### 2.2.3   Constraints on $\phi$ (14)

This section describes the function `Examples/Basic/maskDemo`.

Most of the examples deal with terms in the HJ PDE that effect $\phi$ only through its temporal or spatial dervatives; in contrast, the constraint (14) involves $\phi$ directly. Consequently, it is implemented in a different manner in the toolbox. Users should not be discouraged by its unusual treatment, since this form of constraint has many useful applications, and the mechanism by which it is implemented is even more general than it may first appear.

In its simplest form, (14) can be used to mask out regions of the state space, as shown in figure 7. Suppose that there exists a set $\mathcal{S}$ into which an evolving set—represented by the zero sublevel set of $\phi(x, t)$—should not enter. Given an implicit surface representation $\psi(x)$ for the complement of the forbidden set $\mathcal{S}^{\complement}$, enforcing the constraint $\phi(x, t) \geq \psi(x)$ will ensure that the forbidden set is not entered. In figure 7(b), $\mathcal{S}$ is the small circle centered at the origin. In figure 7(a) the initial circular evolving set is cut in half as it moves to the right

under a constant convective flow field. Because the evolving set is represented implicitly, no special treatment is required when it breaks apart.

The standard `odeCFLn` and term approximation algorithms of the toolbox allow $\phi$ to be modified only through its temporal derivative. However, direct modification of $\phi$ is supported using the `PostTimestep` option of `odeCFLn`, accessed through `odeCFLset` (see section 3.5.3). This option allows the user to specify a function which will be called after each timestep; the function must conform to the `PostTimestepFunc` prototype. The function will have access to the same parameters as a term approximation routine (`t`, `y`, and `schemeData`). It may then modify `y` and/or `schemeData`. The constraint (14) is implemented by modification of `y`, and the constraint function $\psi$ can be stored in `schemeData`.

Figure 7 is generated by the following function, which demonstrates the use of `termConvection` and the `PostTimestep` option of `odeCFLn`. The subfunction `maskAndKeepMin` contained within follows the `postTimestepFunc` prototype.

[ data, g, data0 ] = maskDemo(accuracy, displayType): Demonstrates applications of the `PostTimestep` option of `odeCFLn`, using a simple convective flow field. The parameters `accuracy` and `displayType` are as normal. Plotting routines at the end of the function are specialized to two dimensional grids, and demonstrate the effects of the `PostTimestep` calls. The figure 7(b) is generated by these plotting routines.

The `PostTimestep` mechanism is more general than just constraints of the form (14). Changes to the term approximation parameters in `schemeData` can effect the evolution of the interface; however, there are often ways to achieve the same effect directly in the term approximation routine. A better use is to record information about the changes to $\phi$ during the integration. This application is demonstrated in `maskDemo` as well, where the field `schemeData.min` is used to record $\min_t \phi(x,t)$ as the integration proceeds.

Users should note that modification of `schemeData` can carry a signficant performance penalty, since all of its large fields (such as `schemeData.grid`) will be copied at each timestep. Consequently, this modification mechanism should be used only when no other mechanism can achieve the same result.

## 2.3   Examples from Osher & Fedkiw [24]

This section describes functions in the directory `Examples/OsherFedkiw/`.

This section provides routines which recreate some examples from [24]. Several of these examples involve a star-shaped initial interface. The initial level set function for this curve in $\mathbb{R}^2$ is given by (the implementation uses polar coordinates)

$$\phi(x,0) = \|x\| - s\left(\cos\left(\rho\arctan\left(\frac{x_2}{x_1}\right)\right) + \sigma\right) \tag{16}$$
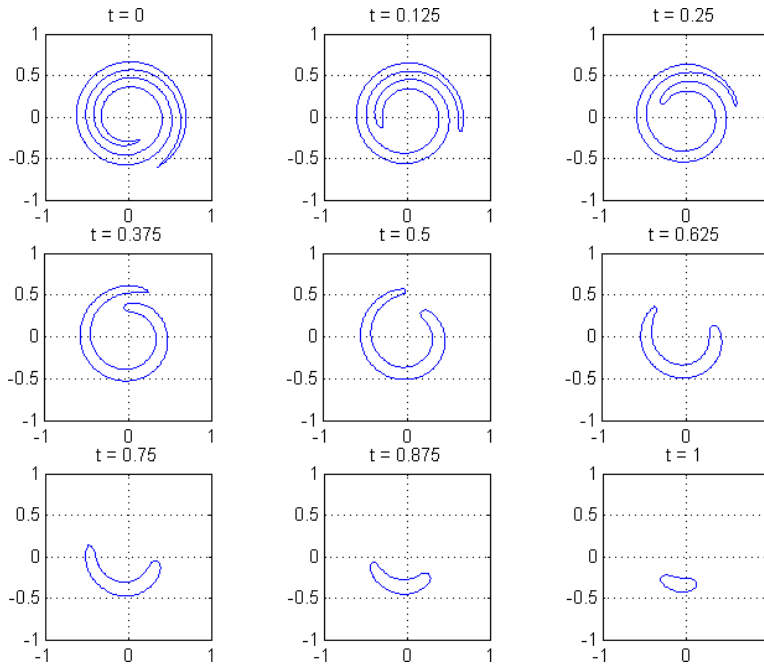
Figure 8: Motion by mean curvature (compare with [24, figure 4.1]). The initial implicit surface function is generated from an ellipse in polar coordinates, rather than the original point cloud description of the problem [24, 25].

where $s$ is a scale controlling the size of the star, $\rho$ is the number of points, and $\sigma$ is an offset the controls the relative size of the points compared to the main body. For the actual parameters chosen, see the example files.

### 2.3.1 Motion by Mean Curvature (9)

This section describes the functions `curvatureSpiralDemo`, `curvatureStarDemo`, `spiralFromEllipse` and `spiralFromPoints` in the directory `Examples/OsherFedkiw/`.

The first example of motion by mean curvature is a classic taken from [25] and shown in figure 8: motion of a two dimensional wound spiral interface. This example and the next demonstrate the use of `termCurvature`.

[ data, g, data0 ] = curvatureSpiralDemo(accuracy, initial, displayType): Demonstrates motion by mean curvature on a two dimensional wound spiral interface. The accuracy and displayType parameters are as normal. The string parameter initial

Figure 9: Motion by mean curvature. Figure 9(a) shows motion with constant multiplier $b$, the result of `curvatureStarDemo` with default parameters (compare with [24, figure 4.2]). Figure 9(b) uses a time and spatially varying multiplier $b(x, t)$ (by choosing `splitFlow == 1`).

> chooses how to construct the initial implicit surface function. The options are `'ellipse'` (the default) and `'points'`. These initial conditions are specifically designed for two dimensional grids.

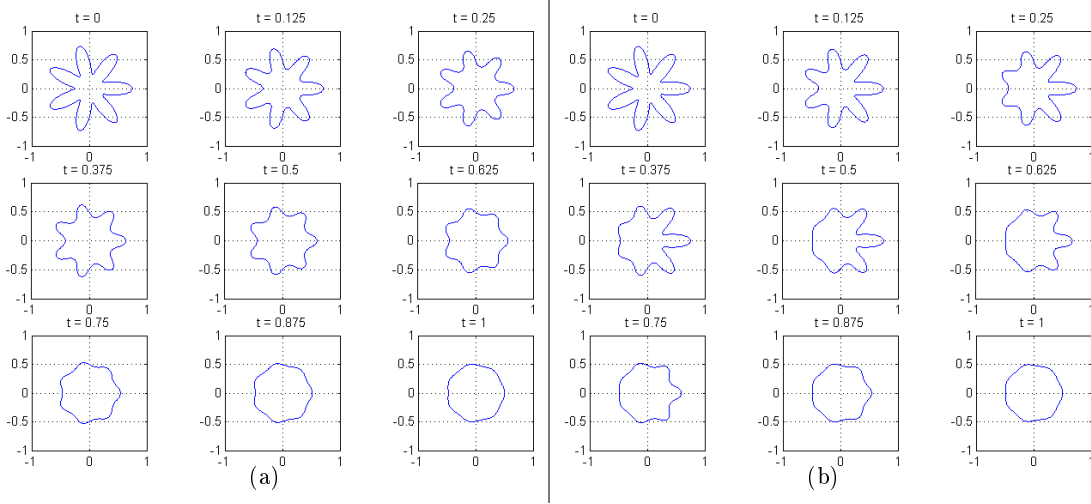Two choices are given for generating the initial implicit surface function. The default choice `initial = 'ellipse'` generates an ellipse in an extended polar coordinate frame, where the parameters of the ellipse were chosen to try to match the shape of the original spiral. The choice `initial = 'points'` uses the original point cloud description of the spiral from [25]. In this release, the latter option is not operational, because the helper routines to generate a signed distance function from a point cloud have not yet been created. The actual generation of the initial implicit surface functions for the spiral is performed in the helper routines `spiralFromEllipse` and `spiralFromPoints`.

The second example of motion by mean curvature is evolution of the star shaped interface, as shown in figure 9. In addition to a different shape, this example shows how to implement a time and spatially varying motion parameter.

[ data, g, data0 ] = curvatureStarDemo(accuracy, splitFlow, displayType): Demonstrates motion by mean curvature with multiplier $b(x)$. The `accuracy` and `displayType` parameters are as normal. The boolean parameter `splitFlow` specifies whether the multiplier should be constant (the default) or varying in time and space. The initial conditions (16) are specifically designed for two dimensional grids.

39
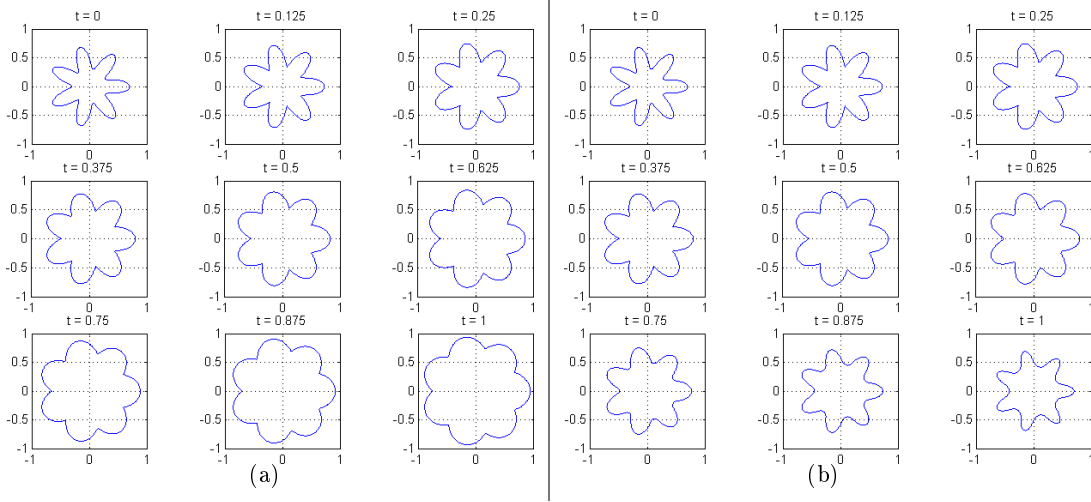
Figure 10: Motion in the normal direction, starting with a star shaped interface. Figure 10(a) shows motion at a constant positive speed, the result of `normalStarDemo` with default parameters (compare with [24, figure 6.1]). Figure 10(b) uses the same speed, but reverses its sign at the midpoint of the simulation (by choosing `reverseFlow == 1`).

The initial conditions and constant multiplier $b_0$ were chosen to try to match the results of [24, figure 4.2]. For the time and spatially varying case `splitFlow == 1`, the multiplier is given by (the actual implementation uses polar coordinates)

$$b(x,t) = \begin{cases} b_0 \left( 1 - \frac{x_1}{\|x\|} \right), & \text{for } t \leq t_s; \\ b_0 \left( 1 + \frac{x_1}{\|x\|} \right), & \text{otherwise.} \end{cases}$$

The switch time $t_s$ is the midpoint of the simulation. In practical terms, this multiplier causes faster motion on the left side of the interface for the first half of the simulation, and then switches sides. The end result should be very similar to the effect of using constant multiplier everywhere. This multiplier is implemented using the subfunction `switchValue`, which follows the `scalarGridFunc` prototype.

### 2.3.2   Motion in the Normal Direction (6)

This section describes the function `Examples/OsherFedkiw/normalStarDemo`.

Evolution of a star shaped interface by motion in the direction normal to the interface is shown in figure 10, and is generated by the following function, which demonstrates the use of `termNormal`. The subfunction `switchValue` contained within follows the `scalarGridFunc` prototype.
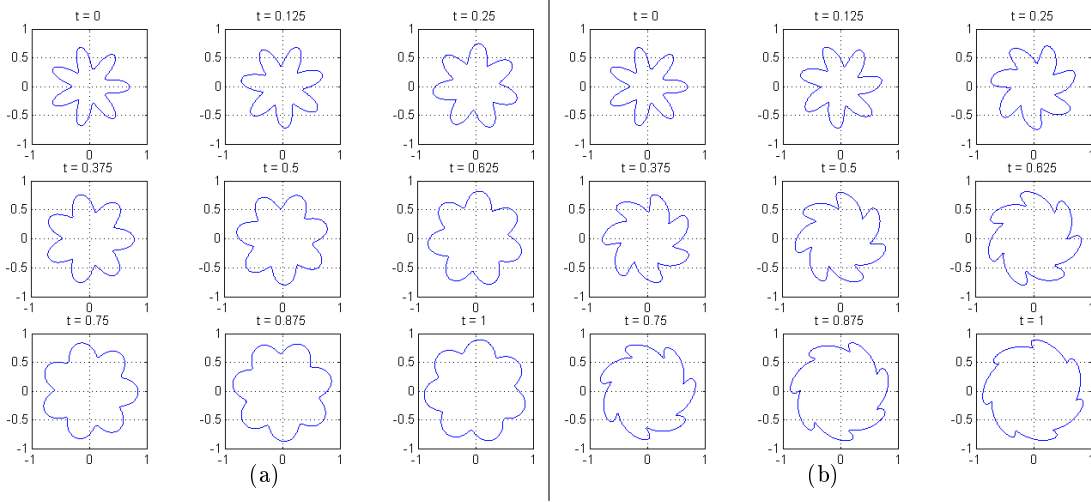
Figure 11: Combining motion in the normal direction with rotational convection. Figure 11(a) shows the results of a rigid body rotation (choose `rigid == 1`). Figure 11(b) multiplies the speed of rotation by the square of the distance to the origin (compare with [24, figure 6.2]). Both figures are generated with `accuracy = 'medium'` on $201^2$ grids.

[ `data, g, data0` ] = `normalStarDemo(accuracy, reverseFlow, displayType)`: Demonstrates motion in the surface normal direction at speed $a(x)$. The `accuracy` and `displayType` parameters are as normal. The boolean parameter `reverseFlow` specifies that the spatially constant speed field should reverse direction halfway through the simulation. The initial conditions (16) are specifically designed for two dimensional grids.

The initial conditions and speed were chosen to try to match the results of [24, figure 6.1] (when `reverseFlow == 0`). Note that when `reverseFlow == 1` is chosen, the initial conditions are not recovered at the final time. This loss of information occurs because of regularization along the concave portions of the front during the first half of the simulation. For another example of this regularization process, see section 2.4.1.

### 2.3.3 Normal Motion Plus Convection

This section describes the function `Examples/OsherFedkiw/spinStarDemo`.

Evolution of a star shaped interface by a combination of rotational convection and motion in the direction normal to the interface is shown in figure 11. It is generated by the following

function, which demonstrates the use of `termSum`, `termNormal`, and `termConvection`. Because `termNormal` and `termConvection` follow the `schemeFunc` prototype, they can be used inside of `termSum`.

> [ data, g, data0 ] = spinStarDemo(accuracy, rigid, displayType): Demonstrates the combination of motion in the normal direction and convective rotation. The `accuracy` and `displayType` parameters are as normal. The boolean parameter `rigid` specifies whether the rotation field should be a rigid body rotation; otherwise, it will be faster further from the origin (the default behavior). The initial conditions (16) and flow fields are specifically designed for two dimensional grids.

Although the caption of [24, figure 6.2] claims that it shows rigid body rotation, the tips of the star are clearly moving faster than the inner portions. Consequently, `spinStarDemo` is designed to show both actual rigid body rotation, or to recreate the figure using a rotational speed that increases as the square of the distance from the origin.

## 2.4 Examples from Sethian [28]

This section provides routines which recreate some examples from [28]. The lack of quantitative parameters in that text—such as figure axis scales with which to reconstruct the initial conditions—makes it challenging to exactly recreate the results.

Before proceeding to the implemented examples, we mention that [28, figure 12.4] uses the same motion as the `flowType = 'linear'` option of the `convectionDemo` routine discussed in section 2.1, and hence could be recreated with minor modifications of that code.

### 2.4.1 Regularization and the Viscous Limit

This section describes the function `Examples/Sethian/tripleSine`.

Many discussions of viscosity solutions of first order HJ PDEs make the point that they are the limit of the classical solutions of a linear second order PDE as the second order term vanishes; for example, see [28, chapter 2.4] or [10, chapter 10]. In [28, figures 2.6 and 2.7] this claim is examined experimentally on a two dimensional example using motion in the normal direction with speed $a(x) = 1 - b\kappa(x)$, where $b \geq 0$ is a constant and $\kappa(x)$ is the local curvature. In the case $b > 0$, this motion is a combination of spatial terms (6) and (9). Figure 12 shows the attempted recreation for three values of $b$. Data for the figure is generated by `tripleSine`, which demonstrates the use of `termNormal`, `termCurvature` and `termSum`.
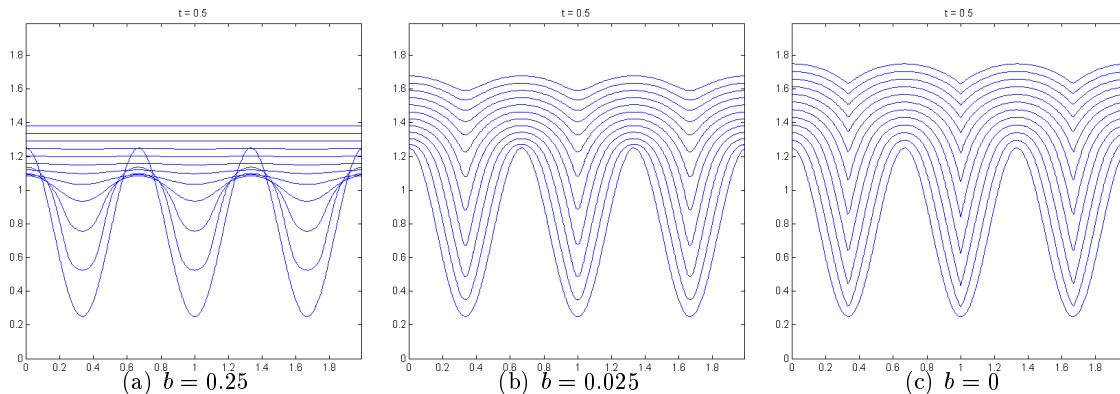
Figure 12: The viscous limit of motion by mean curvature. All three figures show motion in the normal direction with speed $a(x) = 1 - b\kappa(x)$, where each figure uses the specified value for $b$. The initial conditions are the lowest curve, and the remaining curves show the evolution of the implicit surface at equally spaced time intervals. For $b > 0$, the solution remains differentiable for all time. For $b = 0$, the solution quickly develops kinks in the concave regions, but the result can be seen as the limit of the differentiable solution as $b \to 0$. Compare with [28, figures 2.6 and 2.7]

[ data, g, data0 ] = tripleSine(b, accuracy): Demonstrates the evolution of a sine shaped interface under a combination of curvature and normal motion. The accuracy parameter has the usual options. The multiplier for the curvature dependence b must be nonnegative. As $b \to 0$, this function demonstrates how motion in the normal direction is the viscous limit of a curvature dependent motion

The difference between the $b = 0.025$ and $b = 0$ cases is subtle, and lies in the bottom of the valleys of the implicit surface: for the $b = 0$ case, the implicit surface quickly develops a visible sharp corner, while the $b = 0.025$ case remains differentiable for all time. Lagrangian or particle based methods to approximate the motion of the surface in the $b = 0$ case would produce a "swallowtail" solution (see [28, figure 2.3]), which corresponds in some sense to a multivalued solution of the HJ PDE. The upwinded derivatives used in level set methods for motion in the normal direction (the component of the motion independent of $\kappa(x)$) are designed to produce this regularized and single valued viscosity solution, which generates an intersection free implicit surface.

## 2.4.2 Motion by Mean Curvature and Surface Separation

This section describes the function Examples/Sethian/dumbbell1.

43
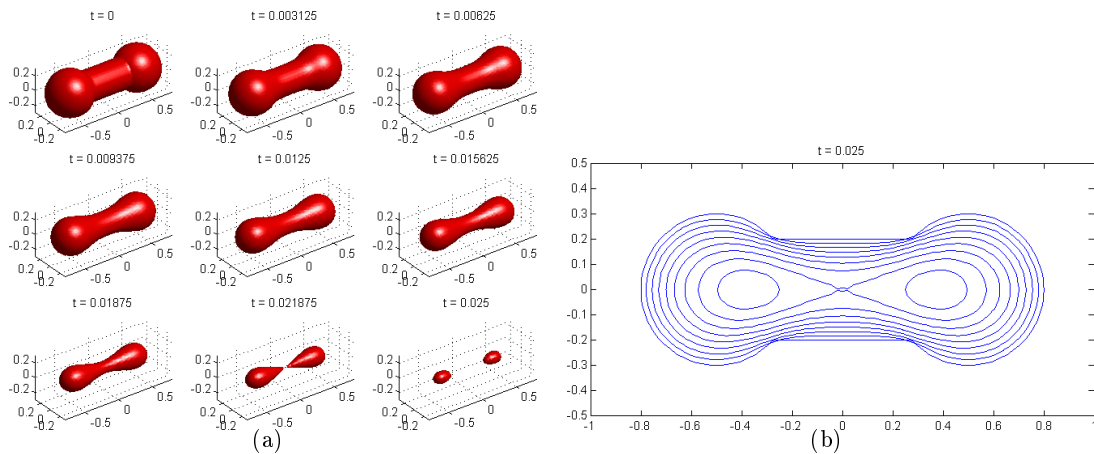
Figure 13: Motion by mean curvature of a three dimensional dumbbell, demonstrating the ability of level set methods to easily handle the separation of implicit surfaces. Figure 13(a) shows how the handle of the dumbbell shrinks faster due to its higher curvature, and hence the implicit surface pinches off into two separate objects. Figure 13(b) shows contour plots at the same timesteps on a slice through the middle of the dumbbell evolving under the same motion (compare with [28, figure 14.2]).

One of the strengths of implicit surface evolution that the level set community often cites is the ability to handle the merging and separation of the surfaces without any mathematical or algorithmic effort. A classic example of the latter is evolution of the dumbbell shape under motion by mean curvature; for example, see [28, figure 14.2]. Figure 13 shows two views of the evolution. Data for the figure is generated by `dumbbell1`, which demonstrates the use of `termCurvature`.

[ data, g, data0 ] = dumbbell1(accuracy): Demonstrates the evolution of a three dimensional dumbbell under motion by mean curvature. The `accuracy` parameter has the usual options. Two figures are produced: a three dimensional isosurface showing the whole dumbbell, and a two dimensional contour of the dumbbell sliced through the middle.

This example also demonstrates another benefit of the implicit surface representation that is not given as much attention. Construction of the three dimensional dumbbell's initial conditions is accomplished in only four lines of code. This feat is possible because simple shapes—such as spheres, polygons and cylinders—can be created by simple mathematical functions, and unions, intersections and complements of implicitly represented sets can be accomplished by taking the minimum, maximum and negation respectively of their implicit surface functions.

44

As an example, the dumbbell is created by

$$\psi_{left}(x) = \sqrt{(x_1 + o)^2 + x_2^2 + x_3^2} - r,$$

$$\psi_{right}(x) = \sqrt{(x_1 - o)^2 + x_2^2 + x_3^2} - r,$$

$$\psi_{center}(x) = \max\left[(|x_1| - o), \left(\sqrt{x_2^2 + x_3^2} - w\right)\right],$$

$$\phi(x, 0) = \min\left[\psi_{left}(x), \psi_{right}(x), \psi_{center}(x)\right],$$

where $o$ is the offset of the center of the lobes of the dumbbell from the origin, $r$ is the radius of the lobes, and $w$ is the radius of the center cylinder. The left and right lobes are constructed from a spherical implicit surface function. The center portion is a cylinder aligned with the $x_1$ axis, capped at the ends by intersection (using the max operator) with halfspaces offset from the origin so as to align with the center of the lobes. The dumbbell as a whole is the union (using the min operator) of these three implicit surfaces.

## 2.5    General HJ Examples from Osher & Shu [26]

This section describes functions in the directory `Examples/OsherShu/`.

The method for treating general Hamilton-Jacobi terms (8) adopted by this toolbox and [24] is basically drawn from [25], and so in this section we provide code for both versions of examples 1 and 2 from that paper.

### 2.5.1    Convex Hamiltonian (Burgers' equation)

This section describes the function `burgersLF` in the directory `Examples/OsherShu/`, which implements

$$D_t\phi(x, t) + H\left(\nabla\phi(x, t)\right) = 0, \qquad 1 \le x < 1,$$
$$\phi(x, 0) = -\cos(\pi x) \tag{17}$$

where $H(p)$ is the convex function

$$H(p) = \frac{\left(\alpha + \sum_{i=1}^{\texttt{grid.dim}} p_i\right)^2}{2}, \tag{18}$$

which makes (17) Burgers' equation. Results in one and two dimensions are shown in figure 14, and are generated by the following function, which demonstrates the use of `termLaxFriedrichs` and the routines implementing the `dissFunc` prototype: `artificialDissipationGLF`, `artificialDissipation` and `artificialDissipationLLLF`.

Figure 14: Solving Burgers' equation with Lax-Friedrichs schemes. Figure 14(a) compares the exact solution (solid) with the third order ENO-LLF approximation on a grid of 10 points (circles); compare with [25, figure 1(d)]. Figure 14(b) shows a two dimensional version of Burgers with an ENO-LLF approximation on a $40^2$ grid; compare with [25, figure 3(b)].

[ data, g, data0 ] = burgersLF(accuracy, dissType, gridDim, gridSize, tMax): Demonstrates solution of Burgers' equation (17) and (18), which in this context is a general HJ PDE with convex Hamiltonian. The accuracy parameter choices are the usual. The dissType parameter must be one of 'global', 'local' or 'locallocal', which choose artificial dissipation using the (regular) Lax-Friedrichs, Local Lax-Friedrichs or Local-Local Lax-Friedrichs schemes from [25] respectively. The gridDim and gridSize inputs specify parameters of the computational grid. The tMax parameter specifies the final time of simulation, and defaults to $1.5/\pi^2$ (when the solution has discontinuous derivative).

Within the file burgersLF, the subfunction burgersHamFunc implements the hamFunc prototype for (18). Subfunction burgersPartialFunc implements the partialFunc prototype solving (47) with Hamiltonian (18). Note that the dissipation parameter $\alpha_i(x)$ is different from the problem parameter $\alpha$.

$$\alpha_j(x) = \max_p \left| \frac{\partial H(p)}{\partial p_j} \right| = \max_p \left| \alpha + \sum_{i=1}^{\texttt{grid.dim}} p_i \right|,$$

where the range over which $p$ is optimized depends on the type of artificial dissipation chosen. For all of the types of artificial dissipation available, the range is a product of intervals, so the optimization over $p$ can be performed by examining each component's interval endpoints independently.
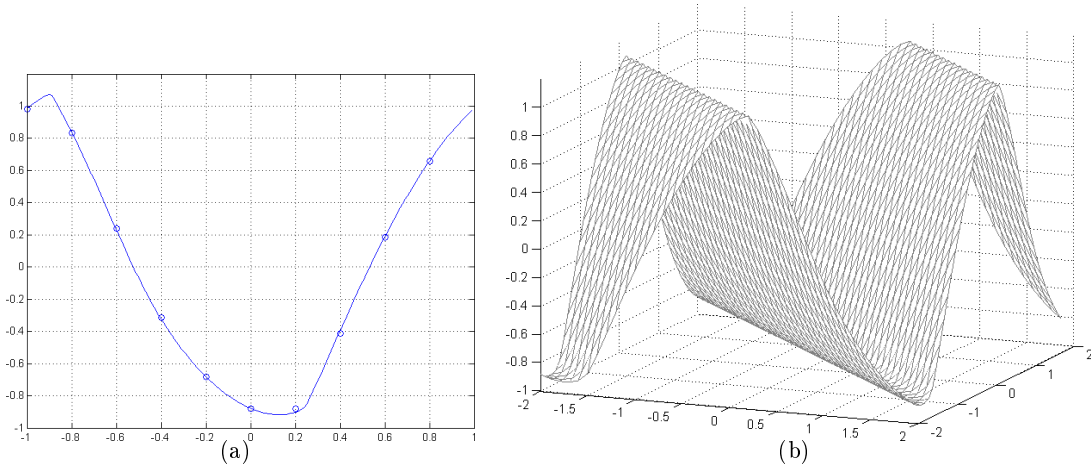
Figure 15: Solving a non-convex general HJ PDE with Lax-Friedrichs schemes. Figure 15(a) compares the exact solution (solid) with the third order ENO-LF approximation on a grid of 10 points (circles); compare with [25, figure 2(d)]. Figure 15(b) shows a two dimensional version of the same equation with an ENO-LF approximation on a $40^2$ grid; compare with [25, figure 3(d)]. There may be slightly more dissipation in these solutions than in those of [25] (see the discussion of nonconvexPartialFunc below).

### 2.5.2 Non-Convex Hamiltonian

This section describes the function nonconvexLF in the directory Examples/OsherShu/, which implements (17), where $H(p)$ is the non-convex function

$$H(p) = -\cos\left(\alpha + \sum_{i=1}^{\texttt{grid.dim}} p_i\right).$$ (19)

Results in one and two dimensions are shown in figure 15, and are generated by the following function, which demonstrates the use of termLaxFriedrichs and the routines implementing the dissFunc prototype: artificialDissipationGLF, artificialDissipationLLF, and artificialDissipationLLLF.

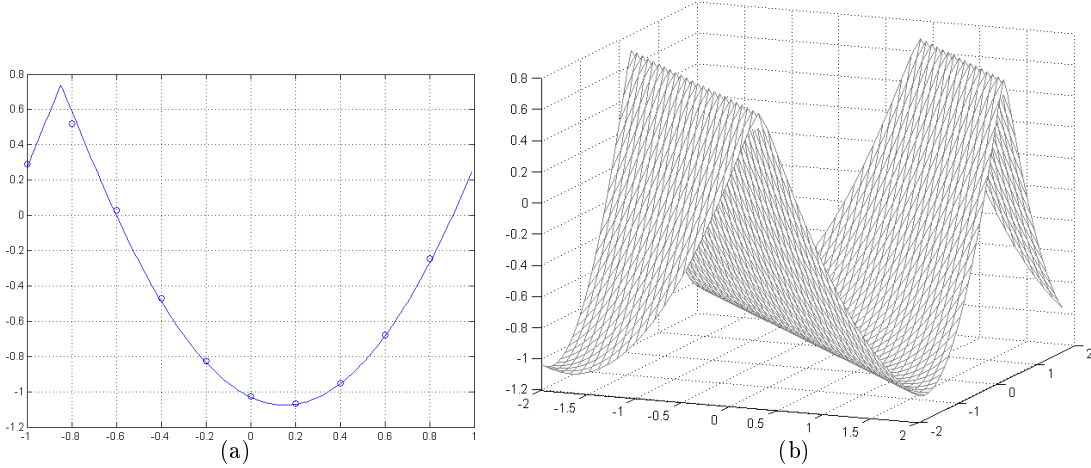[ data, g, data0 ] = nonconvexLF(accuracy, dissType, gridDim, gridSize, tMax): Demonstrates solution of (17) and (19). The accuracy parameter choices are the usual. The dissType parameter must be one of 'global', 'local' or 'locallocal', which choose artificial dissipation using the (regular) Lax-Friedrichs, Local Lax-Friedrichs or Local-Local Lax-Friedrichs schemes from [25] respectively (although the choice turns out to be irrelevant; see the discussion of nonconvexPartialFunc below). The gridDim

and `gridSize` inputs specify parameters of the computational grid. The `tMax` parameter specifies the final time of simulation, and defaults to $1.5/\pi^2$ (when the solution has discontinuous derivative).

Within the file `nonconvexLF`, the subfunction `nonconvexHamFunc` implements the `hamFunc` prototype for (19). Subfunction `nonconvexPartialFunc` implements the `partialFunc` prototype solving (47) with Hamiltonian (19). In this version we conservatively choose

$$\alpha_j(x) = \max_p \left| \frac{\partial H(p)}{\partial p_j} \right| = \max_p \left| \sin \left( \alpha + \sum_{i=1}^{\texttt{grid.dim}} p_i \right) \right| \le 1$$

as an upper bound on the maximum of the magnitude of the partials. This choice is not particularly accurate, but it will maintain numerical stability. Because it does not depend on the range of $p$, all of the dissipation methods will give the same result.

## 2.6  Examples of Reachable Sets

As engineering systems have become more complex, a formal methods community has developed to study methods of validating or verifying the correct behavior of such systems. Model checking is one major thrust of this community, and is a verification method in which the state space of the design is explored in order to determine whether the system—or at least its mathematical model—can enter into an unsafe or incorrect state. Many model checking algorithms attempt to compute a reachable set, which comes in two flavors. The *forwards reachable set* is the set of states that can be reached by system trajectories which start in a given set of initial states. The *backwards reachable set* is the set of states that can give rise to trajectories which subsequently pass through some given set of target states. In [17,19,35] we developed a method of computing robust backwards reachable sets for nonlinear continuous and hybrid systems using an HJ PDE. For more discussion of reachable sets and alternative algorithms for their computation, we suggest [17,19] and the references contained therein.

This toolbox contains several examples of script files to compute reachable sets. We have not yet created an automatic method of computing reachable sets from a SIMULINK block diagram or MATLAB m-file description of a system. Instead, we outline the steps needed to encode a reachable set computation as an HJ PDE in the toolbox.

Consider first the backwards reachable set from a target set $\mathcal{T}$ of a continuous system with dynamics $\dot{x} = f(x, a, b)$, where $x \in \mathbb{R}^n$ is the state of the system, $\mathcal{T} \subset \mathbb{R}^n$, $a \in \mathcal{A} \subset \mathbb{R}^{n_a}$ is an input seeking to keep the system from entering $\mathcal{T}$, and $b \in \mathcal{B} \subset \mathbb{R}^{n_b}$ is an input seeking to drive the system into $\mathcal{T}$. In many examples, $\mathcal{T}$ is an unsafe set so that $a$ should be considered controls keeping the system safe, and $b$ consists of disturbances or model uncertainties which

are assumed to try to make the system unsafe (a robust but conservative treatment). In some examples $a$ and/or $b$ may not be present.

Computation of the backwards reachable set is normally encoded as a terminal value HJ PDE—the same as an initial value PDE, but time runs backwards. The terminal value encodes the target set, so $\phi(x, 0)$ should be an implicit surface function representation of $\mathcal{T}$. Evolution of the backwards reachable set is accomplished by solving

$$D_t \phi(x, t) + \min[0, H(x, D_x \phi(x, t))] = 0 \tag{20}$$

backwards in time, where

$$H(x, p) = \max_{a \in \mathcal{A}} \min_{b \in \mathcal{B}} p^T f(x, a, b). \tag{21}$$

The solution $\phi(x, t)$ is an implicit surface representation of the finite time backwards reachable set. While the toolbox is designed for solving initial value and not terminal value PDEs, for autonomous systems ($f$ does not depend on $t$) converting to the initial value PDE form used in the toolbox simply requires multiplying the output of $H(x, p)$ in (21) by $-1$.

Consideration of (20) reveals that the minimization with zero component is equivalent to constraining the sign of the temporal derivative to be positive (13). This constraint keeps the reachable set from shrinking as time progresses, and is implemented with the spatial term approximation routine `termRestrictUpdate`, which appears in all of the examples below.

If the model involves no inputs or nondeterministic parameters, then (21) degenerates to convection under flow field $v(x) = f(x)$ and can be treated as an example of the form (5). This type of continuous dynamics is encountered in section 2.6.3—although the discrete part of this system has inputs, the continuous part (which gives rise to the HJ PDE) does not. However, most cases involve at least one of the inputs $a$ or $b$, and so (21) must be treated as a general Hamiltonian (8) using `termLaxFriedrichs`. The other examples in this section involve inputs and consequently require the latter treatment.

As described in section 3.6.2, use of `termLaxFriedrichs` requires providing functions which satisfy the `derivFunc`, `dissFunc`, `hamFunc`, and `partialFunc` prototypes. The first is chosen from among the upwind approximations of the first derivative described in section 3.4.1. The second is chosen from among the artificial dissipation functions described in section 3.6.2. The final two must be provided by the user.

The function satisfying the `hamFunc` prototype must compute the solution of (21). Since the optimization over inputs $a$ and $b$ is done for fixed $x$ and $p = \nabla \phi(x, t)$, it can often be performed exactly. If exact optimization over the continuous ranges of $\mathcal{A}$ and/or $\mathcal{B}$ is not possible, they can be sampled discretely. However, users should keep in mind that if $H$ is overestimated—for example, if the truly optimal value of $b$ is not found—then the reachable set will be underestimated. Furthermore, care should be taken if the effects of $a$ and $b$ are not separable. In that case, the order of the optimizations in (21) demands that the value of

$a$ be fixed before the minimization over $b$ is performed (a robust but conservative choice if $a$ is the controls and $b$ is the disturbances).

Coding the function satisfying the `partialFunc` prototype is often the most challenging part of computing a reachable set. This function must solve (47), which in this context translates to

$$\alpha_i(x) = \max_p \left| \frac{\partial}{\partial p_i} H(x, p) \right| = \max_p \left| \frac{\partial}{\partial p_i} \left[ \max_{a \in \mathcal{A}} \min_{b \in \mathcal{B}} p^T f(x, a, b) \right] \right|, \tag{22}$$

where the hyperrectangular range over which $p$ is optimized is an argument to `partialFunc`. The order of the optimizations cannot be modified. Underestimation of this value can lead to numerical instability and toolbox failure. Overestimation will lead to a numerically benign increase in the amount of artificial dissipation introduced by the Lax-Friedrichs approximation. Such dissipation will round sharp corners in the reachable set and, in the worst case, may cause its underestimation; however, since the optimization in (22) can rarely be performed exactly, overestimation is the preferable form of error.

Before proceeding to specific examples, we examine the mathematics of a particularly common form of dynamics. A nonlinear system's inputs enter linearly if we can separate its dynamics into the form

$$f(x, a, b) = f^x(x) + \mathbf{F}^a(x)a + \mathbf{F}^b(x)b, \tag{23}$$

where $f^x : \mathbb{R}^n \to \mathbb{R}^n$, $\mathbf{F}^a : \mathbb{R}^n \to \mathbb{R}^{n_a \times n}$ and $\mathbf{F}^b : \mathbb{R}^n \to \mathbb{R}^{n_b \times n}$.[†] We also assume that the input constraints are hyperrectangles

$$a_i \in \mathcal{A}_i = [\underline{\mathcal{A}}_i, \overline{\mathcal{A}}_i], \quad \mathcal{A} = \prod_{i=1}^{n_a} \mathcal{A}_i,$$

$$b_i \in \mathcal{B}_i = [\underline{\mathcal{B}}_i, \overline{\mathcal{B}}_i], \quad \mathcal{B} = \prod_{i=1}^{n_a} \mathcal{B}_i.$$

Then the optimal inputs to the Hamiltonian (21) can be determined analytically

$$\begin{aligned}
a_i^*(x, p) &= \begin{cases} \underline{\mathcal{A}}_i, & \text{if } \sum_{j=1}^n p_j \mathbf{F}_{ji}^a(x) \le 0; \\ \overline{\mathcal{A}}_i & \text{otherwise;} \end{cases} \\
b_i^*(x, p) &= \begin{cases} \overline{\mathcal{B}}_i, & \text{if } \sum_{j=1}^n p_j \mathbf{F}_{ji}^b(x) \le 0; \\ \underline{\mathcal{B}}_i & \text{otherwise.} \end{cases}
\end{aligned} \tag{24}$$

Futhermore, defining

$$\mathcal{A}_i^{\max} = \max\left( \left| \underline{\mathcal{A}}_i \right|, \left| \overline{\mathcal{A}}_i \right| \right), \qquad \mathcal{B}_i^{\max} = \max\left( \left| \underline{\mathcal{B}}_i \right|, \left| \overline{\mathcal{B}}_i \right| \right),$$

---

[†]Linear systems clearly satisfy this property, since in that case $f^x(x) = \mathbf{A}x$ where $\mathbf{A} \in \mathbb{R}^{n \times n}$, while $\mathbf{F}^a$ and $\mathbf{F}^b$ are constant matrices.

the terms (22) for the `partialFunc` routine can be slightly overestimated by

$$\alpha_j(x) \le \left| f_j^x(x) \right| + \sum_{i=1}^{n_a} \left| \mathbf{F}_{ji}^a(x) \right| \mathcal{A}_i^{\max} + \sum_{i=1}^{n_b} \left| \mathbf{F}_{ji}^b(x) \right| \mathcal{B}_i^{\max}. \tag{25}$$

Section 2.6.1 examines a system which satisfies these separability assumptions.

The extension to hybrid system reachable sets is very much an ad hoc process in the current toolbox. The discrete iteration proposed in [34] and repeated with minor modifications in [17,35] can be coded manually into an m-file, as is done in section 2.6.3. The avoid portion of the reach-avoid operator is implemented by masking the evolving reachable set against the escape set using the `PostTimestep` option of the `odeCFLn` integrators. For autonomous systems, we no longer believe that the escape set itself need be evolved. A revised hybrid system reachable set algorithm—based on variational inequalities—is under development and will be integrated into the toolbox once it is complete.

### 2.6.1 The Game of Two Identical Vehicles

This section describes the functions `air3D` and `figureAir3D` in the directory `Examples/Reachability/`. The game of two identical vehicles has also been called the three dimensional aircraft collision avoidance example.

You've seen this example in virtually every publication on the topic of computing reachable sets using HJ PDEs; recent appearances include [17,19,35]. Now you too can have it running on your very own computer! How much would you pay for this amazing reachable set, you ask? Wait, there's more! Because of recent advances in MATLAB visualization, you can plot not one but two or even three semitransparent isosurface visualizations all in a single figure frame! We'll even throw in a script file to do all the work for you! All this for only a few billion compute cycles! And if you can find a better alternative algorithm, we'll gladly refund 110% of your purchase price![‡]

The coordinate system is shown in figure 16. The vehicles are shown as aircraft, although the simple kinematic model is appropriate to cars or bicycles as well. The state of each vehicle is a position on the plane and a heading. Each vehicle has a fixed forward velocity and an adjustable angular velocity. The game is played between an *evader* vehicle which is trying to escape collision and a *pursuer* which is trying to cause one. Collision occurs if the two vehicles get within a distance $r$ of each other. Because collision only depends on their relative locations, the game is solved in relative coordinates with the evader fixed at the origin facing

---

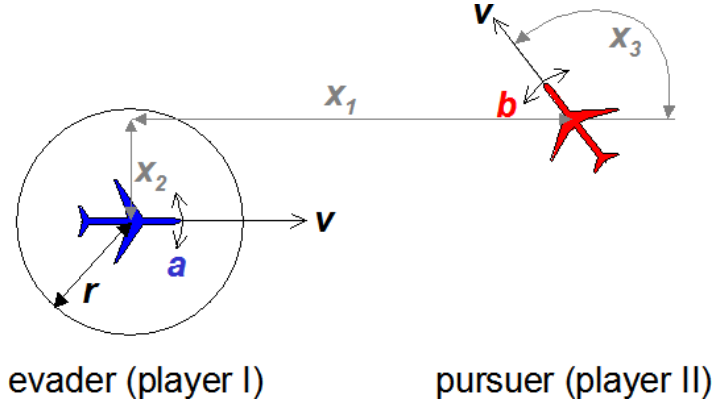[‡]Offer valid only when purchase price is \$0.

51

Figure 16: Relative coordinate system for game of two identical vehicles.

right. The target set $\mathcal{T}$ is the set of collided states, which is a cylinder of radius $r$ centered on the $x_3$ axis. The relative dynamics are

$$
\dot{x} = \frac{d}{dt} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -v_a + v_b \cos x_3 + a x_2 \\ v_b \sin x_3 - a x_1 \\ b - a \end{bmatrix} = f(x, a, b),
\tag{26}
$$

where $v_a \in \mathbb{R}$ is the fixed linear velocity of the evader, $v_b \in \mathbb{R}$ is the fixed linear velocity of the pursuer, $a \in \mathcal{A} \subset \mathbb{R}$ is the angular velocity of the evader and is the "control" input trying to avoid $\mathcal{T}$, and $b \in \mathcal{B} \subset \mathbb{R}$ is the angular velocity of the pursuer and is the "disturbance" input trying to reach $\mathcal{T}$. The routines below assume $v_a > 0$, $v_b > 0$, $\mathcal{A} = [-\mathcal{A}^{\max}, \mathcal{A}^{\max}]$, $\mathcal{A}^{\max} > 0$, $\mathcal{B} = [-\mathcal{B}^{\max}, \mathcal{B}^{\max}]$ and $\mathcal{B}^{\max} > 0$, although the algorithm will work for any combination of parameters. In particular, if $v_a = v_b$ and $\mathcal{A}^{\max} = \mathcal{B}^{\max}$, then the two vehicles are considered identical.

The reachable set for the game of two identical vehicles with $r = 5$, $v_a = v_b = 5$ and $\mathcal{A}^{\max} = \mathcal{B}^{\max} = 1$ is shown in figure 17. The data for the figure is generated by the following function, which demonstrates the use of `termLaxFriedrichs` and `termRestrictUpdate`. Because `termLaxFriedrichs` follows the `schemeFunc` prototype, it can be used inside of `termRestrictUpdate`.

[ `data, g, data0` ] = `air3D(accuracy)`: Demonstrate the (now infamous) three dimensional reachable set for the game of two identical vehicles. The `accuracy` parameter is as usual. The vehicle parameters and visualization technique can be modified within the m-file.

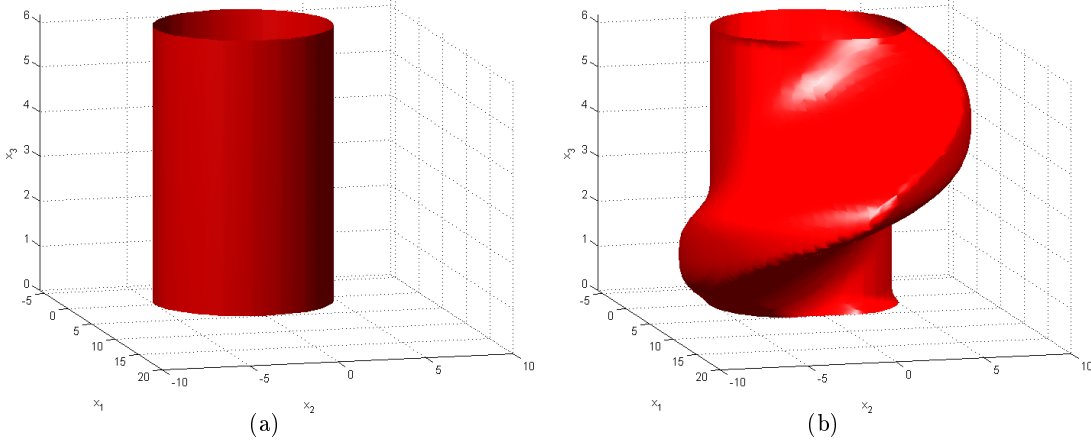The visualization for figure 17 can be recreated by the following routine.

Figure 17: The target and reachable sets for the game of two identical vehicles as visualized by `figureAir3D`. Figure 17(a) shows the target cylinder representing the set of collision states. Figure 17(b) shows the final reachable set at $t = 2.8$, computed by `air3D('medium')`.

`hs = figureAir3D(g, data, data0, superimpose)`: Visualize the three dimensional reachable set, and possibly the initial collision/target set. The first three arguments correspond to the arguments returned by `air3D`. The final argument `superimpose` is a boolean specifying that the target and reachable sets should be displayed in a single figure window using a transparent isosurface for the reachable set. The final two arguments are optional. If `data0` is omitted, no target set is plotted. The default value of `superimpose` is zero. The return value `hs` is a vector of handles to the isosurfaces that were generated.

Before moving on to the next example of reachable sets, we examine the mathematical details of this example a little more. Notice that (26) can be put into the form (23).

$$f^x(x) = \begin{bmatrix} -v_a + v_b \cos x_3 \\ v_b \sin x_3 \\ 0 \end{bmatrix}, \qquad \mathbf{F}^a(x) = \begin{bmatrix} x_2 \\ -x_1 \\ -1 \end{bmatrix}, \qquad \mathbf{F}^b(x) = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}.$$

It is easy to determine from (24) that

$$a^*(x, p) = \mathcal{A}^{\max} \operatorname{sign}(p_1 x_2 - p_2 x_1 - p_3),$$
$$b^*(x, p) = -\mathcal{B}^{\max} \operatorname{sign}(p_3),$$

and the resulting optimal Hamiltonian is

$$H(x, p) = -p_1 v_a + p_1 v_b \cos x_3 + p_2 v_b \sin x_3 + \mathcal{A}^{\max} |p_1 x_2 - p_2 x_1 - p_3| - \mathcal{B}^{\max} |p_3|.$$

53

This Hamiltonian, multiplied by $-1$ to transform the terminal value PDE into an initial value PDE, is implemented by the subfunction `air3DHamFunc`, which implements the `hamFunc` prototype.

The partials of the Hamiltonian can also be determined from (25)

$$\alpha_1(x) \leq |-v_a + v_b \cos x_3| + \mathcal{A}^{\max}|x_2|,$$
$$\alpha_2(x) \leq |v_b \sin x_3| + \mathcal{A}^{\max}|x_1|,$$
$$\alpha_3(x) \leq \mathcal{A}^{\max} + \mathcal{B}^{\max}.$$

These equations are implemented by the subfunction `air3DPartialFunc` which implements the `partialFunc` prototype.

### 2.6.2 Acoustic Capture

This section describes the function `Examples/Reachability/acoustic`.

The example is a variation of the classical homicidal chauffer problem. The version of the game studied here is taken from [4] and we recreate the results in [17]. The reachable set is calculated in relative coordinates with the pursuer fixed at the origin, leading to dynamics

$$\frac{d}{dt}\begin{bmatrix} x \\ y \end{bmatrix} = W_p \begin{bmatrix} 0 \\ -1 \end{bmatrix} + \frac{W_p}{R}\begin{bmatrix} y \\ -x \end{bmatrix} b + 2W_e \min\left(\sqrt{x^2 + y^2}, S\right) a = f(z, a, b), \qquad (27)$$

where the state is $z = (x, y) \in \mathbb{R}^2$ and the problem parameters are the pursuer's speed $W_p$, the evader's speed $W_e$, the pursuer's turn radius $R$ and the evader's radius of maximum speed $S$. The input constraints $a \in \mathcal{A}$ and $b \in \mathcal{B}$ are

$$\mathcal{A} = \{a \in \mathbb{R}^2 \mid \|a\| \leq 1\} \subset \mathbb{R}^2 \qquad \mathcal{B} = [-1, +1] \subset \mathbb{R}.$$

The pursuer's capture set $\mathcal{T}$ is a wide but shallow horizontal rectangle near the origin.

The reachable set for the acoustic capture game with $W_e = 1.3$, $W_p = 1.5$, $R = 0.8$ and $S = 0.5$ is shown in figure 18. The unusual feature of this problem is the development of the hole in the reachable set, a hole which does not anywhere touch the target set $\mathcal{T}$. Because it does not touch $\mathcal{T}$, finding its boundary by Lagrangian methods—for example, by following trajectories backwards from the target set—would prove very challenging.

The figure is generated by the following function, which demonstrates the use of `termLaxFriedrichs` and `termRestrictUpdate`.

[ data, g, data0 ] = acoustic(accuracy): Demonstrate the reachable set for the acoustic capture game. The `accuracy` parameter is as usual. The vehicle parameters and visualization technique can be modified within the m-file.
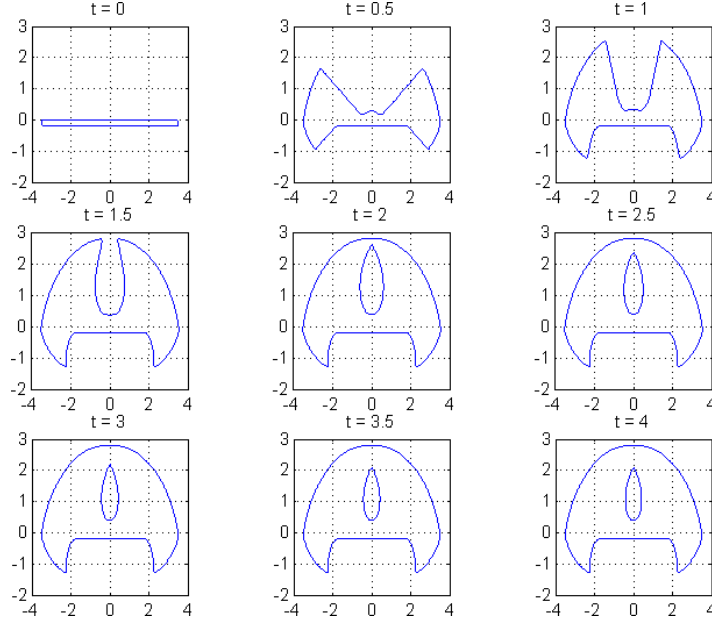
Figure 18: Evolution of the acoustic capture game's reachable set.

Unlike the previous example, (27) cannot be put into the form (23) because the bounds on input $a$ are not dimensionally separable. However, it is relatively easy to find the optimal Hamiltonian

$$
\begin{aligned}
H(z,p) &= \max_{a \in \mathcal{A}} \min_{b \in \mathcal{B}} \left[ p^T f(z,a,b) \right], \\
&= \max_{\|a\| \le 1} \min_{|b| \le 1} \left[ \begin{array}{c} -p_2 W_p + b \frac{W_p}{R} \left( p_1 y - p_2 x \right) \\ + (p^T a)(2W_e) \min \left( \sqrt{x^2 + y^2}, S \right) \end{array} \right], \\
&= -p_2 W_p - \frac{W_p}{R} \left| p_1 y - p_2 x \right| + \|p\|(2W_e) \min \left( \sqrt{x^2 + y^2}, S \right),
\end{aligned}
$$

where we choose inputs

$$
a^*(z,p) = \frac{p}{\|p\|}, \qquad b^*(z,p) = -\operatorname{sign}(p_1 y - p_2 x).
$$

This Hamiltonian, multiplied by $-1$ to transform the terminal value PDE into an initial value PDE, is implemented by the subfunction `acousticHamFunc`, which implements the `hamFunc` prototype.

Computing the partials of the Hamiltonian is also complicated by the dimensionally mixed bounds on input $a$. However, since we only need to overestimate these partials, we can safely

Figure 19: Hybrid automata for the three mode protocol.

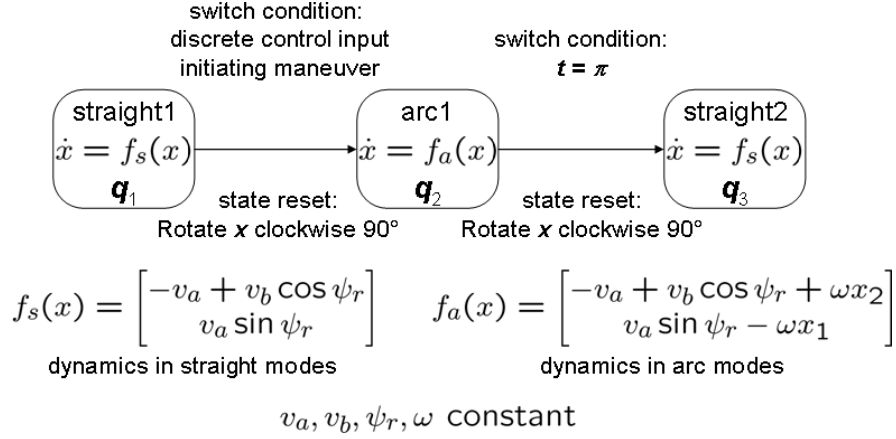assume that the bounds on the norm of $a$ apply to each of its individual components. Then an overestimation of the partials is possible.

$$\alpha_1(z) \leq \frac{W_p}{R}|y| + 2W_e \min(\sqrt{x^2 + y^2}, S),$$

$$\alpha_2(z) \leq W_p + \frac{W_p}{R}|x| + 2W_e \min(\sqrt{x^2 + y^2}, S),$$

These equations are implemented by the subfunction `acousticPartialFunc` which implements the `partialFunc` prototype.

### 2.6.3 Multimode Collision Avoidance

This section describes the function `Examples/Reachability/airMode`.

As an example of a hybrid system reachable set we take the three mode collision avoidance example from [15, 17]. Like the game of two identical vehicles in section 2.6.1, this is a collision avoidance scenario played with two simple kinematic vehicles. In this case, however, the angular velocities of the two vehicles are fixed and equal, so that their relative angle never varies. Therefore the computation can be performed in two dimensions.

The hybrid automata for the example is shown in figure 19. The only input to the system is the decision $\sigma$ to initiate the collision avoidance protocol, and after that point all switches and motion is synchronized between the vehicles. The relative location of the vehicles always follows one of two dynamics:
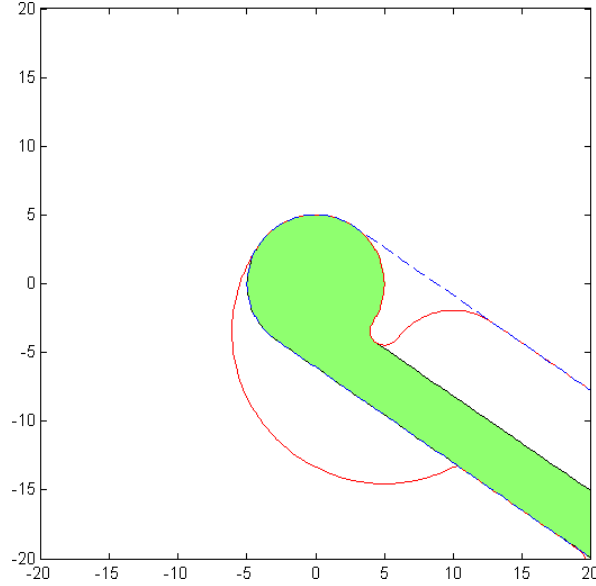
Figure 20: Reachable set in the first mode for the three mode collision avoidance protocol. The solid region is the set of states within which collision is inevitable. Outside the solid contour the protocol can be safely initiated. The dashed contour shows the edges of the unsafe set if no protocol is initiated.

- Straight motion: both vehicles move with constant linear velocities and zero angular velocities. The dynamics are

$$\dot{z} = \frac{d}{dt} \begin{bmatrix} x_r \\ y_r \end{bmatrix} = \begin{bmatrix} -v_a + v_b \cos \psi_r \\ v_a \sin \psi_r \end{bmatrix} = f_s(z),$$

where $v_a$ and $v_b$ are fixed (although not necessarily equal).

- Curved motion: both vehicles move with constant linear velocities and a constant, equal angular velocity. The dynamics are

$$\dot{z} = \frac{d}{dt} \begin{bmatrix} x_r \\ y_r \end{bmatrix} = \begin{bmatrix} -v_a + v_b \cos \psi_r + \omega y_r \\ v_a \sin \psi_r - \omega x_r \end{bmatrix} = f_c(z),$$

where $v_a$, $v_b$ and $\omega$ are fixed.

Because the continuous dynamics involve no inputs, we can simplify the computation by using convection by constant flow fields within each of the individual modes.

The reachable set for this multimode protocol with $v_a = 3$, $v_b = 4$, $\psi = -4\pi/3$ and $\omega = 1$ is shown in figure 20. The figure is generated by the following function, which demonstrates the use of `termConvection` and `termRestrictUpdate`.

57

[ `reach, g, avoid, data0` ] = `airMode(accuracy)`: Demonstrate the three mode colli-
sion avoidance protocol reach set computation. The `accuracy` parameter is as usual.
The vehicle parameters and visualization technique can be modified within the m-file.
The return parameter `reach` is an implicit surface function for the set of states where
a collision is inevitable, and the parameter `avoid` is an implicit surface function for the
set of states in which the protocol can be safely initiated.

The computation of the reach sets in each individual mode is relatively straightforward (all
the more so because of the convective dynamics), and is accomplished by the subfunction
`findReachSet`. The tricky and entirely ad hoc component is how to keep track of the in-
teraction between the modes. For this specific example, four reach set computations are
performed.

- The set of states which lead to collision in the final mode. This is simple convection of
  the target set (a circle) according to the constant linear velocity dynamics $f_s$.

- The set of states which lead to collision in the second mode. This is simple convection
  of the target set according to a constant rotational flow field $f_c$.

- The set of states which, when rotated through the second mode, lead to collision in
  the third mode. This set is computed starting with the third mode's unsafe states and
  using the rotational flow field $f_c$. However, this computation does not restrict the sign
  of the temporal derivative in the HJ PDE. Such a restriction would mark states as
  unsafe if they merely passed into and then out of the third mode's unsafe states while
  still in the protocol's second mode. Instead, states should only be marked as unsafe if
  they pass through the collision set in the second mode, or switch into the third mode
  while in the third mode's unsafe states.

- The set of states in which a collision is inevitable whether the protocol is initiated or not.
  This computation involves the reach-avoid operator. The escape set is all those states
  in which it is safe to initiate the protocol; specifically, the complement of the union of
  the states which lead to collision in the second mode (the second reach set computed)
  and the states which go through the second mode and lead to collision in the third (the
  third reach set computed). This escape set is used to mask the evolution of the reach set
  via a constraint of the form (14) The reach set's evolution is otherwise identical to the
  evolution in the third mode above. The masking is performed by `postTimestepMask`,
  which implements the `postTimestepFunc` protocol.

For more general reach and reach-avoid computation algorithms, see [35] and the citations
within.

## 2.7 Examples of Cost to Go and Time to Reach Stationary HJ Equations

This section describes functions in the directory `Examples/TimeToReach/`.

The time-dependent initial value PDE discussed in sections 1.1 and 1.2 is not the only type of equation often given the moniker "Hamilton-Jacobi." Another common class of equations given this name is exemplified by

$$G(x, \vartheta(x), \nabla\vartheta(x)) = 0 \quad \text{in } \mathbb{R}^d \setminus \mathcal{T},$$
$$\vartheta(x) = 0 \quad \text{on } \partial\mathcal{T}, \tag{28}$$

where $\mathcal{T} \subset \mathbb{R}^d$ is closed. Note that this equation is a boundary value problem with no time variable. This class of *time-independent* equations is also often called *stationary* or *static*. A generalization which allows for second order derivative terms $D_x^2\vartheta(x)$ is called *degenerate elliptic* [8].

Clearly this PDE is not of a form directly supported by the toolbox—it does not even contain a temporal derivative. However, following [23] we can solve an auxiliary time-dependent HJ PDE using the toolbox and extract an approximation of the solution to (28). To summarize those results, assume that the boundary conditions are noncharacteristic

$$\sum_{i=1}^{d} p_i \frac{\partial G(x, \vartheta, p)}{\partial p_i} \neq 0 \text{ on } \partial\mathcal{T}. \tag{29}$$

A time-dependent HJ PDE is found by making the changes of variables

$$\vartheta(x) \leftarrow t \qquad \text{and} \qquad \nabla\vartheta(x) \leftarrow \frac{\nabla\phi(x, t)}{D_t\phi(x, t)}$$

in (28) and algebraic manipulation of the resulting equation into the form

$$D_t\phi(x, t) + H(x, t, \nabla\phi(x, t)) = 0, \tag{30}$$

where (29) ensures that this manipulation is locally feasible. The corresponding initial conditions are v

$$\phi(x, 0) = 0 \text{ on } \partial\mathcal{T},$$
$$\phi(x, 0) < 0 \text{ inside } \mathcal{T}, \tag{31}$$
$$\phi(x, 0) > 0 \text{ on } \mathbb{R}^d \setminus \mathcal{T},$$

with $\phi(x, 0)$ a continuous and strictly monotone function of distance to $\mathcal{T}$ near its boundary. Typically $\phi(x, 0)$ is chosen as a signed distance function for $\mathcal{T}$. Terms (4) and (8) from the toolbox can then be applied to solve (30), and from this solution we can extract

$$\vartheta(x) = \{t \mid \phi(x, t) = 0\}. \tag{32}$$

As a concrete example of this process, let $\mathcal{T}$ be the closed target set of a system evolving according to dynamics $\dot{x} = f(x, b)$. The single input parameter $b \in \mathcal{B}$, where $\mathcal{B} \subset \mathbb{R}^{d_b}$ is compact and $b(\cdot) : [0, T] \rightarrow \mathcal{B}$ is measurable, is attempting to minimize the *cost to go* to arrive at the target

$$\vartheta(x) = \min_{b(\cdot)} \int_0^T \ell(x(t), b(t)) dt, \tag{33}$$

where the running cost $\ell(x, b) > 0$ is continuous and $T = \min\{t \geq 0 \mid x(t) \in \mathcal{T}\}$ is the time of arrival at the target set. If $\ell \equiv 1$, then $\vartheta(x)$ is the *minimum time to reach* function.

Following standard procedures [2] it can be shown that the cost to go function is the viscosity solution of the HJ PDE

$$
\begin{aligned}
\tilde{H}(x, D_x \vartheta(x)) &= \ell(x, b) && \text{in } \mathbb{R}^d \setminus \mathcal{T}, \\
\vartheta(x) &= 0 && \text{on } \partial \mathcal{T}, \\
\tilde{H}(x, p) &= \min_{b \in \mathcal{B}} p \cdot f(x, b).
\end{aligned}
\tag{34}
$$

This equation is an example of (28), and following the procedure outlined above leads to the time-dependent HJ PDE (30) with Hamiltonian

$$H(x, t, p) = \min_{b \in \mathcal{B}} \frac{p \cdot f(x, t)}{\ell(x, b)}. \tag{35}$$

The condition (29) in this case requires that $\nabla \phi(x, 0) \cdot f(x, b) \neq 0$ on $\partial \mathcal{T}$, which is equivalent to requiring that the vector field $f$ not be tangent to the target set. This condition ensures that the resulting $\vartheta$ function is continuous (although not necessarily Lipschitz, as is demonstrated in section 2.7.2).

Interestingly, in the case of cost to go with no discount, it is possible to derive the same HJ PDE (30) with Hamiltonian (35) starting from the reach set theory [19], but without the noncharacteristic assumption (29). The resulting $\phi$ function is still continuous in time and space, but it may be constant with respect to $t$ at fixed $x$; consequently, we cannot uniquely define $\vartheta$ using (32). Choosing $\vartheta(x) = \min\{t \mid \phi(x, t) = 0\}$ is a reasonable alternative, although this $\vartheta$ will no longer be continuous and hence the standard viscosity solution theory does not apply.

A variety of different algorithms have been proposed for approximating minimum time to reach, cost to go or general stationary HJ PDE solutions for systems with inputs and nonlinear dynamics [3, 11, 29, 36]. Because the explicit time-dependent solvers of the toolbox are timestep restricted by a CFL condition, it is likely that the method described above is the slowest of the algorithms. However, it is quite general—although not derived above, this method works for zero sum differential games, where (33) and (35) are modified to include a maximization over an input $a \in \mathcal{A}$ which may appear in both the dynamics $f$ and the running cost $\ell$. The resulting Hamiltonian is nonconvex in $\nabla \phi$. Furthermore, because the

function $\phi$ on which derivative approximations are taken is continuous, this algorithm has the potential for better accuracy than those methods which depend on differentiating the sometimes discontinuous $\vartheta$ function directly. Quantitative comparisons are challenging, because implementations of the other algorithms are not publicly available at the present time.

Solution of (30) requires straightforward application of the toolbox techniques discussed elsewhere in this document. In order to avoid storing the entire time history $\phi(x, t)$, extraction of $\vartheta$ from $\phi$ using (32) can be accomplished on the fly by the `postTimestepTTR` routine (see section 3.7.3) and modification of the `schemeData` structure. The simplest example is provided by `convectionTTR`. The dynamic system in this case has no inputs, so the general Hamiltonian term in (30) can be replaced by motion by convection (5).

[ `mttr, data, gridOut, data0` ] = `convectionTTR(front, accuracy, gridIn)`: Demonstrate time to reach computation for a system without inputs. The target set is everything to the right of a curve running essentially perpendicular to $x_1$. The dynamics are $\dot{x} = f(x) = \begin{bmatrix} 1 & 0 & \cdots & 0 \end{bmatrix}^T$. The parameter `front` can be used to choose a specific shape for the target set. The parameter `accuracy` has the usual string choices. The parameter `gridIn` can be used to modify the default two dimensional grid. Parameter `mttr` is the minimum time to reach function $\vartheta(x)$ out to the maximum integration time $t_{\max}$ (as specified in the source code); nodes that were not reached within $t_{\max}$ are set to `NaN`. Parameters `data` and `data0` are $\phi(x, t_{\max})$ and $\phi(x, 0)$ respectively. Parameter `gridOut = gridIn` (or the default grid, if `gridIn` is unspecified).

The analytic solution to this problem is given by `max(0, data0)`, so this routine can be used to validate the operation of `postTimestepTTR`. Otherwise, `convectionTTR` is not particularly interesting.

The remaining examples in this section treat problems with ideal target sets that are ill-posed for this time-dependent formulation, and the convergence results are not surprisingly somewhat poor. We are still looking for compelling examples with well-posed target sets and analytic solutions.

### 2.7.1 Time to Reach for a Holonomic System

As the first detailed example of a time to reach function, we consider the holonomic two dimensional system

$$\dot{x} = f(x, b) = b$$

for $x \in \mathbb{R}^2$, $b \in \mathbb{R}^2$ with $\|b\|_k \leq 1$, and $k \in \{1, 2, \infty\}$. Trajectories for this system can travel in any direction, with the speed of travel bounded in the $k$ norm. Different results can be generated by choosing different $k$. If the target set $\mathcal{T}$ is the origin, then

- $k = 1$ will generate a $\vartheta$ with diamond shaped level sets.

- $k = 2$ will generate a $\vartheta$ with circular level sets.

- $k = \infty$ will generate a $\vartheta$ with square level sets.

Futhermore, the analytic solution in each of these cases is just $\vartheta(x) = \|x\|_k$.

The stationary formulation of the problem is

$$
\begin{aligned}
\tilde{H}(x, D_x \vartheta(x)) &= 1 && \text{in } \mathbb{R}^d \setminus \mathcal{T}, \\
\vartheta(x) &= 0 && \text{on } \partial\mathcal{T}, \\
\tilde{H}(x, p) &= \min_{\|b\|_k \leq 1} p \cdot b.
\end{aligned}
$$

Note that the noncharacteristic condition (29) will be satisfied for any $\mathcal{T}$. Since $\ell(x, b) \equiv 1$, by (35) the Hamiltonian for the time-dependent equation is identical to that for the stationary equation $H(x, p) = \tilde{H}(x, p)$. After choosing the optimal input $\|b\|_k \leq 1$, the result is

$$
H(x, p) = \begin{cases} -(\max_i |p_i|), & \text{if } k = 1; \\ -\|p\|_2, & \text{if } k = 2; \\ -\sum_i |p_i|, & \text{if } k = \infty. \end{cases}
$$

The resulting time-dependent HJ PDE is implemented in the following routine, which demonstrates the use of `termLaxFriedrichs` and `postTimestepTTR`.

[ mttr, attr, data, gridOut, data0 ] = holonomicTTR(whichNorm, accuracy, gridIn):
  Compute the minimum time to reach function for the simple holonomic system. The input to the system is bounded $\|b\|_k \leq 1$; the argument `whichNorm` provides several methods of specifying $k$. The string argument `accuracy` has its usual options. The `gridIn` argument optionally allows the calling function to specify a grid different from the default—useful for convergence studies (see `convergeHolonomicTTR` below); however, the routine is not designed to work in dimensions other than $\mathbb{R}^2$. Parameter `mttr` is the numeric approximation of the minimum time to reach function $\hat{\vartheta}(x)$ generated by `postTimestepTTR` out to the maximum integration time $t_{\max}$ (as specified in the source code); nodes that were not reached within $t_{\max}$ are set to `NaN`. Parameter `attr` is the analytic minimum time to reach function $\vartheta(x)$ for all nodes of the grid, generated by either `analyticHolonomicTTR` or `analyticSumSquareTTR` (as specified in the source code). Parameters `data` and `data0` are $\phi(x, t_{\max})$ and $\phi(x, 0)$ respectively. Parameter `gridOut = gridIn` (or the default grid, if `gridIn` is unspecified).
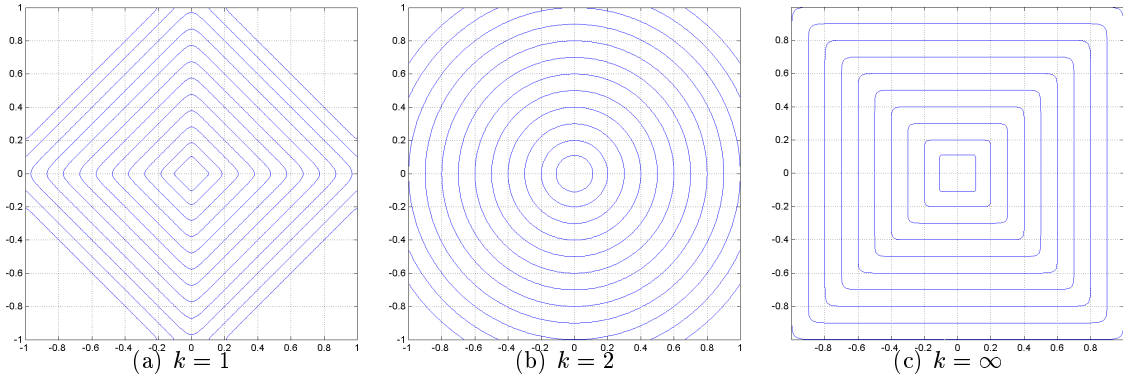
(a) $k = 1$  (b) $k = 2$  (c) $k = \infty$

Figure 21: Contour plots of the numeric approximation $\hat{\vartheta}_r(x)$ of the minimum time to reach function for the simple holonomic system with the three different choices of norm bounds on the input $\|b\|_k \leq 1$. The innermost contour in each case is the target set $\mathcal{T}_r$, and the remaining contours have a spacing of 0.1 units.

Unfortunately, after this translation to time-dependent form, we discover that the initial conditions (31) rule out any target set that cannot be represented by an implicit surface function. The origin of $\mathbb{R}^2$ is of codimension 2, so we must choose a larger set—specifically, a set with an interior. A parameter `whichIC` within the source code for `holonomicTTR` can be modified to choose either a circle or a square of a given size is used as the initial condition; however, the analytic solution for such choices is, in general, no longer so easy to determine.

In order to analyze the accuracy of `postTimestepTTR`, a third choice for `whichIC` is `'analytic'`. This option computes the analytic solution $\vartheta_0(x)$ for the chosen norm $k$ and with the origin as the target set $\mathcal{T}_0$. A target set with nonempty interior is generated by

$$\mathcal{T}_r = \{x \mid \vartheta_0(x) \leq r\},$$

where $r > 0$ is the effective radius of the target set. The analytic solution $\vartheta_r(x)$ with target set $\mathcal{T}_r$ is then simply

$$\vartheta_r(x) = \max(0, \vartheta_0(x) - r),$$

where the maximization is required because minimum time to reach functions are never negative. It is this function $\vartheta_r(x)$ which is returned by parameter `attr` of `holonomicTTR` in the case `whichIC` is set to `'analytic'` in the source code. Note that using `whichIC = 'circle'` and `whichNorm = '2'` or `whichIC = 'square'` and `whichNorm = 'inf'` will produce similar results. Examples of the numerical approximations $\hat{\vartheta}_r(x)$ returned by parameter `mttr` for each of the three choice of $k$ are shown in figure 21. Calculation of the analytic solution $\vartheta_0(x)$ is performed by the following routine.

`mttr = analyticHolonomicTTR(whichNorm, grid)`: Compute the analytic minimum time to reach the origin for the simple holonomic system in two dimensions. Parameter
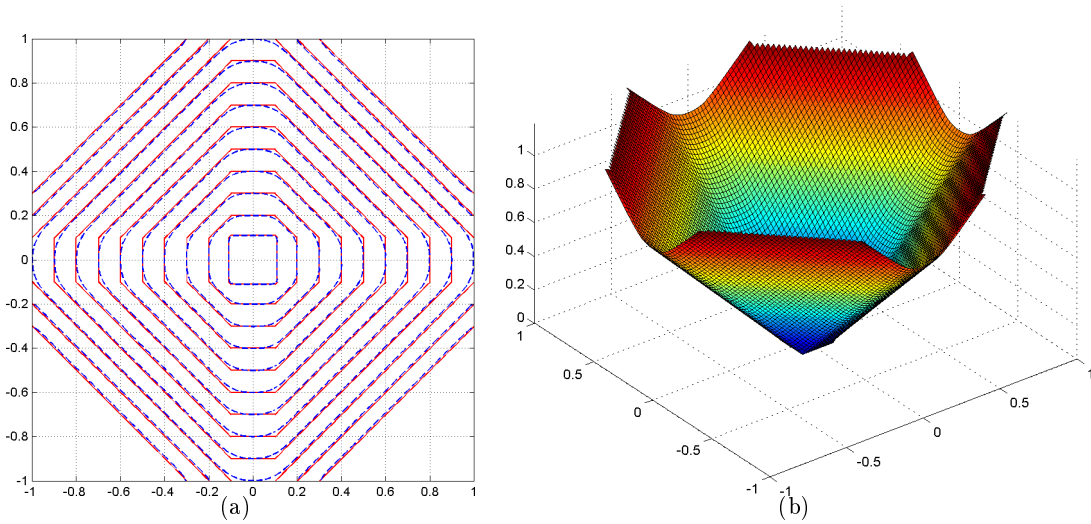
Figure 22: Results for the simple holonomic system with square target set and one norm bounded input. Figure 22(a) shows contour plots of the analytic solution (solid) and numerical approximation (dashed). Figure 22(b) shows a surface plot of the numeric approximation.

whichNorm specifies the norm $k$ in which the system input is bounded $\|b\|_k \leq 1$. The parameter grid should be in $\mathbb{R}^2$. The solution $\vartheta_0(x) = \|x\|_k$ is returned in mttr.

Despite the ease of finding the analytic solution $\vartheta_r(x)$, some readers might object to the validity of a convergence analysis on the approximation of the solution $\hat{\vartheta}_r(x)$ to a stationary primal problem when the analytic solution to that problem is used as the initial condition for the auxiliary time-dependent equation. Consequently, the analytic solution for a slightly different problem was also determined.

mttr = analyticSumSquareTTR(radius, grid): Compute the analytic minimum time to reach a square centered at the origin whose sides are of length 2 * radius for the simple holonomic system with input bounded in the one norm $\|b\|_1 = \sum_i |b_i| \leq 1$. This analytic solution $\vartheta_s(x)$ is returned in mttr. The parameter grid should be in $\mathbb{R}^2$.

The level sets of $\vartheta_s(x)$ are essentially diamonds, except that the corners are replaced by axis aligned segments of the same length as the target square. The user can select either analyticHolonomicTTR or analyticSumSquareTTR to generate the analytic solution inside holonomicTTR by modifying the source code. When using analyticSumSquare, the user should choose whichIC = 'square' and whichNorm = 'sum'. In this case, the initial conditions $\phi(x, 0)$ for the time-dependent equation are not related to the analytic solution of the
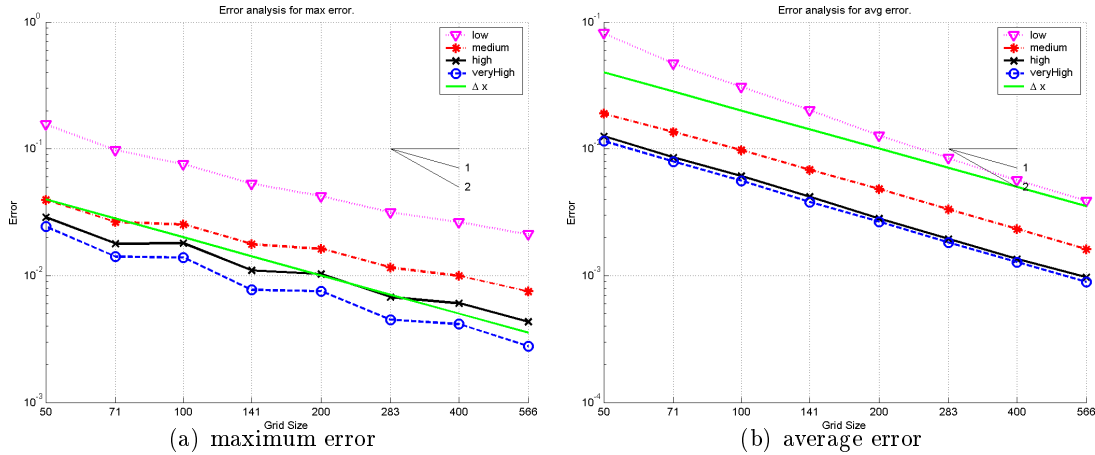
64

(a) maximum error            (b) average error

Figure 23: Convergence of the numeric solution to the analytic solution for the simple holonomic system with square target set and one norm bounded input. A variety of accuracies for the time-dependent solver are shown, as is the grid cell size for comparison. Note that the vertical axes of the two plots are not the same.

problem. The solution is shown in figure 22. Note how the dissipation from the Lax-Friedrichs solver causes the corners of the numeric solution to become rounded. Approximately first order convergence of the numeric solution in both maximum and average error is demonstrated in figure 23. Higher order convergence cannot be expected because `postTimestepTTR` uses linear interpolation to approximate (32). The convergence analysis is performed by the following script.

convergeHolonomicTTR: Script file that runs `holonomicTTR` for a variety of grid resolutions and accuracies, and compares the resulting numeric and analytic minimum time to reach functions in order to demonstrate convergence of the numeric solution to the analytic solution as the grid is refined. Results are stored in the structure `err`. Most problem parameters must be set in advance in the source code for `holonomicTTR`.

This script can be used with either of the analytic solutions `analyticHolonomicTTR` or `analyticSumSquareTTR`, as chosen inside `holonomicTTR`.

65

### 2.7.2 Time to Reach for a Double Integrator

A more challenging example is the well studied double integrator. The two dimensions are position $x_1$ and velocity $x_2$, and the input $b \in \mathcal{B}$ is a scalar. The system parameters are

$$f(x, b) = \begin{bmatrix} x_2 \\ b \end{bmatrix}, \quad \mathcal{T} = \left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\},$$
$$\ell(x, b) = 1, \qquad \mathcal{B} = [-1, +1],$$
$$\phi(x, 0) = \|x\|_2.$$

The analytic solution when the target set is the origin is given by [1]

$$\vartheta(x) = \begin{cases} x_2 + \sqrt{4x_1 + 2x_2^2}, & \text{if } x_1 > \frac{1}{2}x_2|x_2|; \\ -x_2 + \sqrt{-4x_1 + 2x_2^2}, & \text{if } x_1 < \frac{1}{2}x_2|x_2|; \\ |x_2| & \text{if } x_1 = \frac{1}{2}x_2|x_2|. \end{cases} \tag{36}$$

This system satisfies (29) and is small time controllable when the target is the origin, so $\vartheta(x)$ is continuous; however, as can be seen from the square root term in (36), it is not Lipschitz continuous on the curve $x_1 = \frac{1}{2}x_2|x_2|$. Calculation of this analytic solution is performed by the following routine.

**mttr = analyticDoubleIntegratorTTR(grid)**: Compute the analytic minimum time to reach the origin for the double integrator with unit bounded input, as given in (36). This solution is returned in **mttr**, which is defined on the grid given by **grid**.

Returning to the numerical approximation, the Hamiltonian is easy to derive

$$H(x, p) = \min_{|b| \leq 1} (p_1 x_2 + p_2 b) = (p_1 x_2 - |p_2|). \tag{37}$$

The challenge of accurate approximation, as in the previous section, comes down to the target set. The origin is again a codimension 2 object, and cannot be represented by the initial conditions $\phi(x, 0)$. Two options are available: using a small circle around the origin, or using a sublevel set of the analytic solution. The numerical approximation is generated by the following routine, which demonstrates the use of **termLaxFriedrichs** and **postTimestepTTR**. The target set is chosen by setting variable **whichIC** within the source code.

**[ mttr, attr, data, gridOut, data0 ] = doubleIntegratorTTR(accuracy, gridIn)**: Compute the minimum time to reach function for the classical double integrator with unit bounded input. The string argument **accuracy** has its usual options. The **gridIn** argument optionally allows the calling function to specify a grid different from the
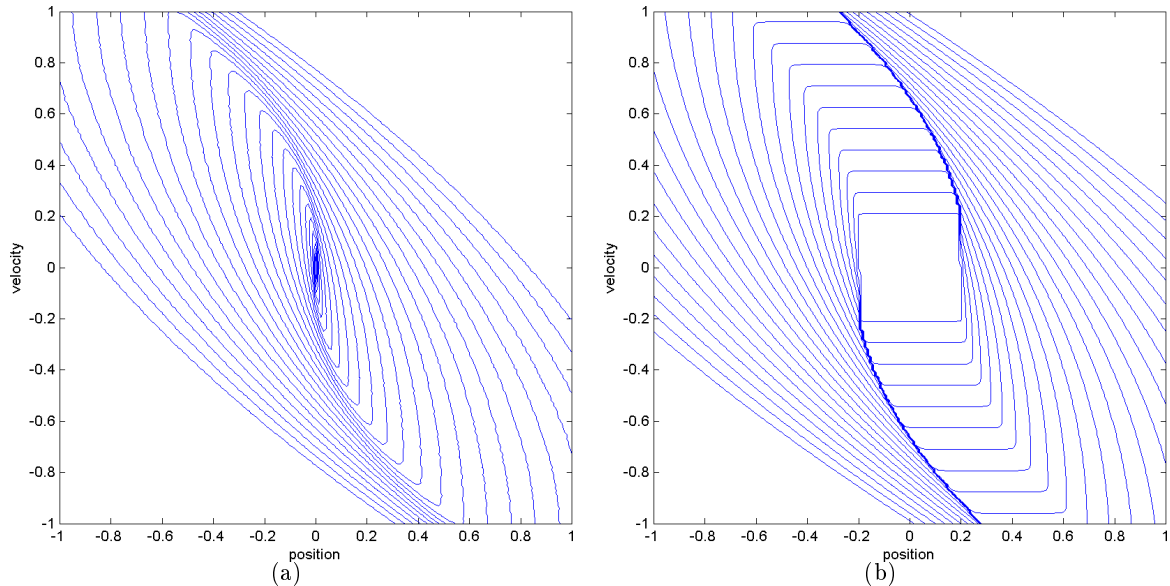
Figure 24: Contour plots of the minimum time to reach a target centered at the origin for a double integrator with unit magnitude input. The largest contour represents a time to reach of 2.4. In figure 24(a), the target is a circle of radius less than a grid cell. In figure 24(b), the target is the square of width 0.4.

default—useful for convergence studies (see `convergeDoubleIntegratorTTR` below); however, the routine is not designed to work in dimensions other than $\mathbb{R}^2$. Parameter `mttr` is the numeric approximation of the minimum time to reach function $\hat{\vartheta}(x)$ generated by `postTimestepTTR` out to the maximum integration time $t_{\max}$ (as specified in the source code); nodes that were not reached within $t_{\max}$ are set to `NaN`. Parameter `attr` is the analytic minimum time to reach function $\vartheta(x)$ for all nodes of the grid, generated by `analyticDoubleIntegratorTTR`. Parameters `data` and `data0` are $\phi(x, t_{\max})$ and $\phi(x, 0)$ respectively. Parameter `gridOut = gridIn` (or the default grid, if `gridIn` is unspecified).

Using a small circle as the target is an appealing solution, since it is easy to create a signed distance function for $\phi(x, 0)$ for such a target. While a target of nonzero size may break assumption (29) and may give rise to a discontinuous $\vartheta(x)$, as discussed earlier it is possible to derive the same time-dependent HJ PDE from reach set theory without this assumption for time to reach examples. Figure 24 shows the results for two target sets: one whose size is small relative to the grid size, and one whose radius is much larger. The contour lines for the latter case become very tightly packed along the curves where the analytic $\vartheta(x)$ would be discontinuous. Unfortunately, the analytic solution (36) no longer applies for a target of nonzero radius and hence quantitative analysis of the accuracy of the approximation for this
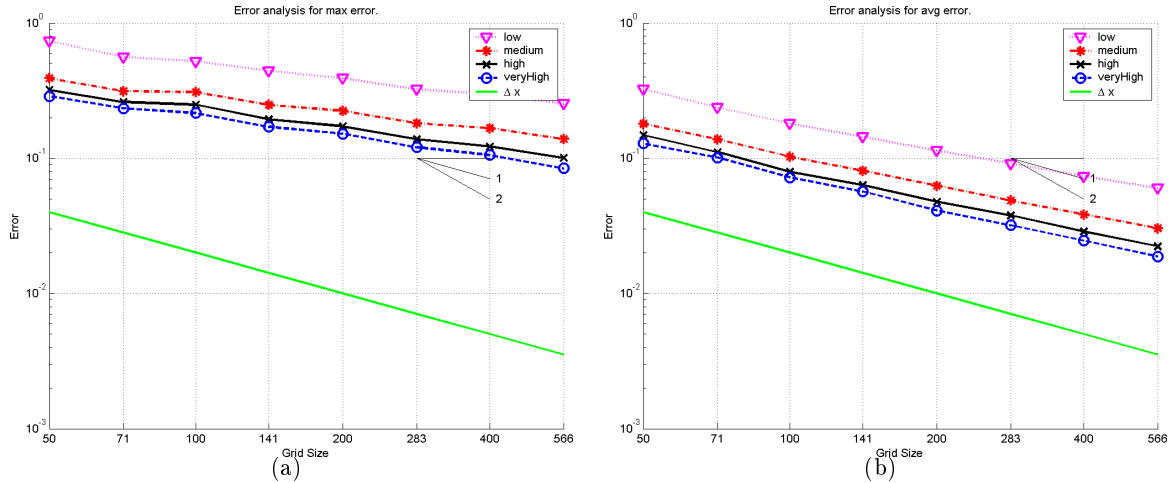
67

Figure 25: Convergence of the numeric solution to the analytic solution for the double integrator when approximating the origin as the target set using the 0.4 sublevel set of the analytic solution. A variety of accuracies for the time-dependent solver are shown, as is the grid cell size for comparison. The error is large, but converging. Possible reasons for the poor results are discussed in the text.

target option is not currently available. Direct quantitative comparison with (36) generates poor results even if the target circle size is reduced toward zero.

For the purposes of demonstrating convergence, we are therefore stuck using a sublevel set of the analytic solution as target. Quite apart from the validity argument presented in the previous section, this option is poor because the analytic solution is not Lipschitz and hence is not a well behaved implicit surface function to use as an initial condition. Attempts to create a signed distance version of this target set using `signedDistanceIterative` have failed due to large motions of the target set's boundary in the regions where the function is not Lipschitz. Direct use of the analytic solution's 0.4 sublevel set generates the convergence results shown in figure 25. The errors are large—usually greater than a grid cell—but are converging. This convergence study was generated with the following routine.

`convergeDoubleIntegratorTTR`: Script file that runs `doubleIntegratorTTR` for a variety of grid resolutions and accuracies, and compares the resulting numeric and analytic minimum time to reach functions in order to demonstrate convergence of the numeric solution to the analytic solution as the grid is refined. Results are stored in the structure `err`. Most problem parameters must be set in advance in the source code for `doubleIntegratorTTR`.

There are at least three potential origins of these poor results. The first is the fact that the initial conditions for the time-dependent HJ PDE were not Lipschitz continuous. The second is that the solution we are trying to approximate is not Lipschitz continuous. The third is the artificial dissipation introduced by the Lax-Friedrichs approximation of the Hamiltonian (37). The true reach set at any fixed time for the double integrator has two very sharp corners in the upper left and lower right. All three of these error sources will tend to blunt the numerical approximation of these corners, and since the optimal trajectories discovered by the HJ PDE from all points in the state space travel through one of these corners, errors introduced here will propagate throughout the state space. In fact, it is likely that all three error sources play a role, and we will continue to investigate this challenging example in the hopes of improving the accuracy of this method of solving stationary HJ PDEs.

## 2.8    Expectations of Stochastic Differential Equations

This section describes functions in the directory `Examples/SDE/`.

Many systems are not amenable to deterministic models because their evolution is influenced by parameters whose exact value is not known. Knowledge about the nondeterministic parameters comes in several different varieties. Sections 2.6 and 2.7 examine systems in which the time varying parameters are drawn at each instant from some bounded set—a model of nondeterminism often used for control signals that steer a system's evolution toward some desired state.

In this section we examine systems in which the nondeterministic parameters are drawn probabilistically from a known distribution. More specifically, the system will evolve according to the *stochastic differential equation* (SDE)

$$dx(t) = f(x(t), t)\, dt + \sigma(x(t), t)\, dB(t), \qquad x(t_0) = x_0, \tag{38}$$

where $B(t)$ is a Brownian motion process of appropriate dimension, the *drift* term $f$ represents the deterministic component of the system evolution, and the *diffusion* term $\sigma\, dB(t)$ represents the probabilistic component of the system evolution. The functions $f$ and $\sigma$ must be continuous in $x$ and $t$. We interpret (38) in the Itô sense [22]. We note in passing that this model can be extended to allow for magnitude bounded time varying parameters, which leads to a stochastic differential game; for more details and examples of using the toolbox to analyze such stochastic systems, see [20]

Given $f$, $\sigma$ and $x_0$, there are methods to simulate sample trajectories of the system (38) [13]. Here, however, we concentrate on expected outcomes. Unfortunately, SDE theory is complex, and this section is not designed as an introduction to it. Instead, we refer the reader to [9,22], and move on to some examples with which we seek to demonstrate the power of the toolbox to those who have selected an appropriate PDE formulation for the expected outcome of their SDE problem.

Actually, the toy routines below are mostly designed to test the toolbox, and are essentially one dimensional. The toolbox is designed to be far more general, and so we would very much like to hear about interesting SDE examples in 1–3 dimensions with which we could demonstrate the toolbox's features, particularly if they have analytic solutions against which to test.

In the examples below, the astute reader will notice that in some cases the sign of the discounting and trace of the Hessian terms are opposite to those given in (11) and (10) respectively. Consequently, the PDEs will appear to violate the monotonicity requirement (3). However, in these cases the stated PDEs are terminal value problems, and so reversing time to put them into the initial value format required by the toolbox will introduce the necessary negative sign.

### 2.8.1  A Linear SDE with Additive Noise

Perhaps the simplest nontrivial case of (38) is the system

$$dx = ax\,dt + b\,dB, \qquad x(t_0) = x_0, \tag{39}$$

where $a$ and $b$ are constants. Given some continuous terminal payoff function $g(x)$, the expected terminal payoff at time $T > t_0$ can be defined as

$$\phi(x_0, t_0) = E\left[g(x(T))\right],$$

where $x(t)$ solves (39). In particular, we can pick out the expected terminal state $E[x]$ with $g(x) = x$ and the variance $\text{Var}(x)$ in the expected terminal state using $g(x) = x^2$ and $\text{Var}(x) = E[x^2] - E[x]^2$.

The expected payoff function $\phi$ is the solution to the second order PDE

$$D_t\phi(x, t) + a\nabla\phi(x, t) + \tfrac{1}{2}\,\text{trace}\left[b^2 D_x^2\phi(x, t)\right] = 0. \tag{40}$$

with terminal condition $\phi(x, T) = g(x)$. Note that we need not appeal to viscosity solutions in this restricted case if $b \neq 0$, because the PDE is linear parabolic. However, we will solve this PDE using the toolbox. The following routine demonstrates the use of `termTraceHessian` and `termConvection`.

[ data, g, data0 ] = linearAdditiveSDE(payoff, a, b, tf, dim, accuracy): Demonstrate the expected outcome of a linear ODE with additive stochastic noise. Input parameter `payoff` specifies the terminal payoff function $g(x)$ whose expectation will be determined: either `'x'` for $g(x) = x$ or `'x^2'` for $g(x) = x^2$. The parameters `a` and `b` respectively specify the drift and diffusion terms for the SDE (39). The time

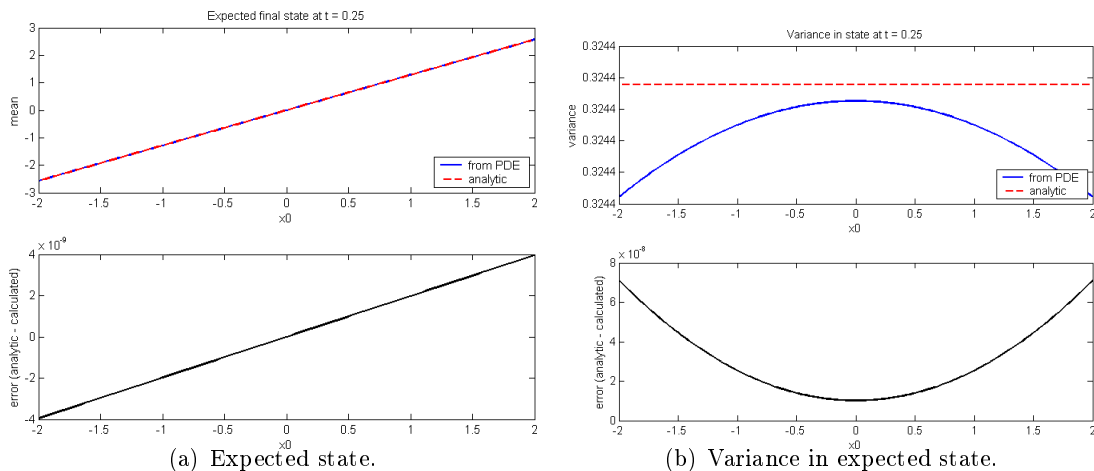(a) Expected state.  (b) Variance in expected state.

Figure 26: Output of `testLinearAdditiveSDE`, showing the expected terminal state and expected terminal variance of the state, as computed with the toolbox and analytically. On the top of the left plot, note that the approximation and the analytic solution are indistinguishable, while on the top of the right plot the scale of the vertical axis is about $10^{-7}$.

interval $T - t_0$ is specified by `tf`. Normally this example is run in one dimension, but for testing purposes it can be run in two dimensions, in which case system evolution is governed by (39) in the $x_1 + x_2$ direction, and no evolution occurs in the $x_1 - x_2$ direction (similar to the two dimensional general HJ examples from section 2.5). The `accuracy` input parameter has its usual choices. The first return parameter `data` is the expected terminal payoff $\phi(x, t)$ for trajectories starting from time $t = T - $ `tf`. The remaining output parameters are the computational grid in `g` and the terminal payoff function $\phi(x, T) = g(x)$ in `data0`.

For this simple SDE (39), there is an analytic solution against which to check our calculations.

$$E[x] = e^{at} E[x(T)],$$
$$\mathrm{Var}[x] = \tfrac{1}{2a} \left[ (b^2 + 2a\,\mathrm{Var}[x(T)]) e^{2at} - b^2 \right].$$

Such testing is performed using the following script, where we assume $E[x(T)] = x$ and $\mathrm{Var}[x(T)] = 0$. The results are shown in figure 26.

`testLinearAdditiveSDE`: Runs the `linearAdditive` routine for a fixed set of parameters, and compares the results against the analytic solution. Intermediate results with `payoff = 'x'` and `payoff = 'x^2'` are shown during simulation, and then the analytic and numeric $E[x]$ and $\mathrm{Var}[x]$ are shown.
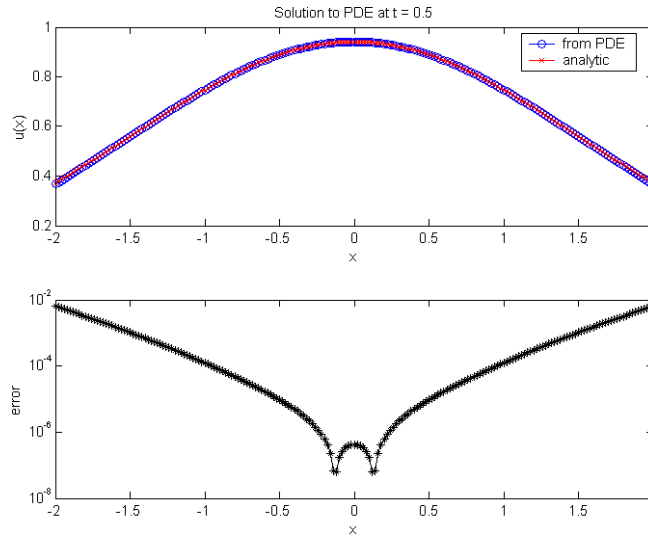
71

Figure 27: Results of running `exerciseKP529`. The analytic and numeric solutions are visually indistinguishable in the upper plot, as can be deduced from the error shown in the lower plot. The pattern of error suggests that the nonphysical boundary conditions are a significant source.

### 2.8.2 An Example from Kloeden & Platen [13]

This example is taken from [13, exercise 17.1.2, p. 531]. Translated into the toolbox's notation, the general PDE is

$$D_t\phi + \tfrac{1}{2}\,\mathrm{trace}\left[D_x^2\phi\right] + V(x,t)\phi = 0 \tag{41}$$

with terminal conditions $\phi(x,T) = g(x)$. Starting from the left, the terms in this equation are a temporal derivative (4), a second order diffusion (10) which is just the Laplacian operator, and a discounting term (11) with potentially state and time dependent rate $V(x,t)$. For this specific exercise, the authors select

$$g(x) = 1, \qquad V(x,t) = -\tfrac{1}{2}x^2,$$

(there is a typo in [13]—the $V(x,t)$ stated there should be negated) and state the solution

$$\phi(x,t) = \exp\left[\frac{1}{2}(T-t) + \frac{1}{2}\left(\frac{1 - e^{2(T-t)}}{1 + e^{2(T-t)}}\right)x^2 + \frac{1}{2}\ln\left(\frac{2}{1 + e^{2(T-t)}}\right)\right]$$

for $x \in \mathbb{R}$ and $0 \le t \le T$.

The following script solves (41) with the toolbox, and compares the approximation against the analytic solution. In the process it demonstrates the application of `termTraceHessian`

72

to approximate a Laplacian (although in this case it is in one dimension), and the application of `termDiscount` with state dependent discount rate. Results are shown in figure 27.

`exerciseKP529`: Approximate the solution to the PDE (41) in [13, exercise 17.1.2, p. 531] numerically using the toolbox. This function is a script, so all parameters must be set by modifying the file and all variables will appear in the top level workspace. A figure showing the evolution of the solution is generated, as is a figure demonstrating the error in the solution at the final time.

### 2.8.3    An Example from Øksendal [22]

This example is technically drawn from [22, exercise 8.6, pp 169–170], although a number of similar SDEs appear in nearby exercises. In connection with the Black-Scholes formula for pricing financial options, we arrive at the PDE

$$D_t\phi + \rho\phi - \alpha x\nabla\phi - \tfrac{1}{2}\beta^2 x^2 D_x^2\phi = 0 \tag{42}$$

with initial conditions $\phi(x,0) = \max(0, x - K)$, where $\rho > 0$, $\alpha$, $\beta$, and $K > 0$ are constants. Starting from the left, the terms in this equation are a temporal derivative (4), a discounting term (11) with constant discount rate $\rho$, a convection term (5) with state dependent velocity $\alpha x$, and a second order diffusion (10) with state dependent rate $\tfrac{1}{2}\beta^2 x^2$. Financially, the constants $\rho$ and $\alpha$ are related to deterministic rates of interest, and $\beta$ to the stock's volatility.

The exercise also gives the solution as the integral

$$\phi(x,t) = \frac{e^{-\rho t}}{\sqrt{2\pi t}} \int_{\mathbb{R}} \max\left[0, x\exp\left(\left[\alpha - \tfrac{1}{2}\beta^2\right]t + \beta y\right) - K\right] \exp\left(\frac{-y^2}{2t}\right) dy. \tag{43}$$

The following routine solves (42) with the toolbox, and (43) with MATLAB's quadrature routine `quad`. In the process it demonstrates the application of `termTraceHessian`, `termDiscount`, and `termConvection`.

`[ data, g, data0 ] = exerciseO169b`: Approximate the solution to the PDE in [22, exercise 8.6, pp. 169–170] numerically using both direct application of the toolbox to the PDE (42), and numerical quadrature on the integral solution (43). This function was designed as a script, but converted into a function only to allow helper subfunctions to be stored in the same file—all parameters must be set within the function itself. The return parameters have their usual interpretation.
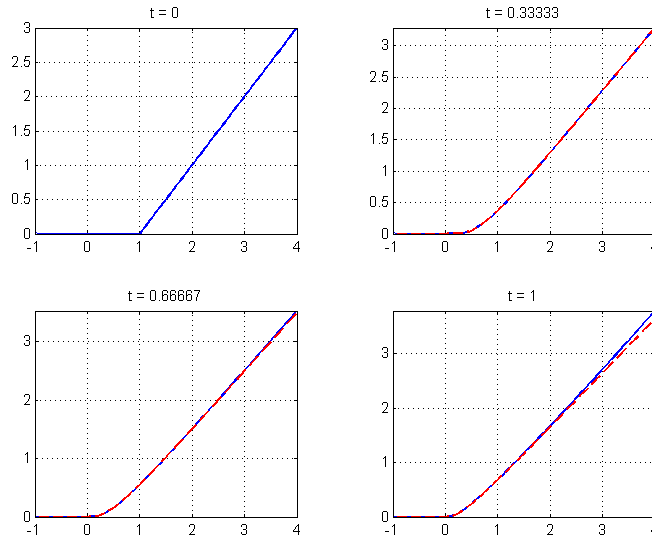
73

Figure 28: Results of running `exercise0169b`. Only the initial condition is plotted for $t = 0$. The PDE and quadrature approximations are indistinguishable until $t = 1$, where they have separated near the upper boundary of the computational domain, likely due to errors introduced by the nonphysical boundary conditions.

The results are shown in figure 28. The toolbox approximation of the solution to the PDE and MATLAB's quadrature approximation to the integral are indistinguishable for small $t$, but begin to separate near the upper boundary of the computational domain by $t = 1$. This error may be due to the dubious boundary condition imposed by `exercise0169b`.

For simulation with smaller plotting intervals, another discrepancy was noted between the two approximations. Near the kink in the initial conditions at $K$, the quadrature solution deviated from the PDE solution and continued to display a kink for small $t > 0$. Since the true solution is known to be smooth for $t > 0$, we believe it is the quadrature approximation that is incorrect rather than the toolbox. This belief is supported by the fact that the discrepancy is occuring near the kink in the initial conditions, where quadrature's interpolating polynomials will do a poor job of approximating the integrand. Further support comes from tests with smooth functions replacing the $\max(0, x - K)$ in the initial conditions and integrand; tests in which the PDE and quadrature approximations agree visually for all time.

**This routine has not been well tested.** This example is closely related to many in financial mathematics, and deserves further exploration and improvements; however, its preliminary form successfully serves the purpose of demonstrating the new features of the toolbox, and so in the interests of early release we have not fully validated or investigated it.

## 2.9 Vector Level Sets

This section describes functions in the directory `Examples/Vector/`.

While they handle topological surface changes very well, the scalar level set functions used in most of the toolbox's examples are restricted in that they can only represent closed surfaces of codimension one. This restriction rules out, for example, closed curves in $\mathbb{R}^3$ (codimension two) or open curves in $\mathbb{R}^2$. As explained in [24, chapter 10], this constraint can be overcome by solving a system of interacting HJ PDEs, and working with intersections of the level and sublevel sets of the solutions.

At the present time, we have only implemented the open curves in $\mathbb{R}^2$ example detailed below. We do not anticipate trouble implementing the closed curves in $\mathbb{R}^3$ example using the same framework, although it will require new term approximation routines for the tangent, normal and binormal vectors. For another example—one that uses the toolbox's vector level set formulation to analyze a stochastic hybrid system model of the Internet's Transmission Control Protocol (TCP)—see [20].

As an initial test of the vector level set code, we compare convective motion by a fixed velocity field (5) using upwinded (`termConvection`) and Lax-Friedrichs (`termLaxFriedrichs`) approximations. The evolution equations for the two level sets do not depend on one another, so this comparison does not technically require vector level sets; however, it serves as a useful test and keeps the two simulations in temporal lockstep.

> `[ dataC, dataH, g, data0 ] = compareTerms(flowType, initShape, accuracy, dissType, displayType)`: Test the vector level set code in `odeCFLn` by two independent simulations of convective motion: one by upwinding with `termConvection` and one by Lax-Friedrichs with `termLaxFriedrichs`. All parameters have defaults. Parameter `flowType` specifies one of a list of possible flow fields. Parameter `initShape` specifies what type of initial level set should be convected. Parameters `accuracy` and `displayType` have their usual options. Return parameter `dataC` contains the final time value of the level set function evolved with `termConvection`, and `dataH` the level set function evolved with `termLaxFriedrichs`. Return parameters `g` and `data0` contain the computational grid and initial conditions respectively.

Figure 29 shows the results of running `compareTerms('linear', 'cube', 'medium')`. Over the timespan $t \in [0, 1]$ and for the options available in `compareTerms`, there appears to be very little difference in results between upwinding and Lax-Friedrichs. Whether this is an indictment of the poor accuracy of upwinding or a testimony to the impressive accuracy of Lax-Friedrichs is not clear.
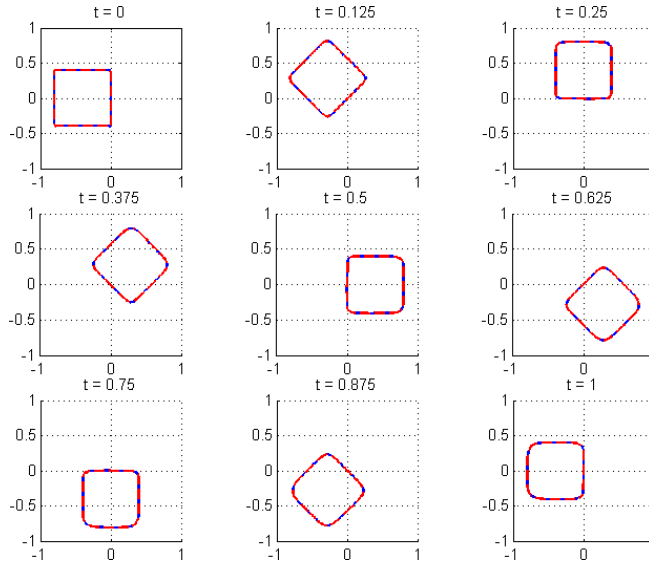
Figure 29: Results of running `compareTerms('linear', 'cube', 'medium')`. The up-winded and Lax-Friedrichs results are indistinguishable. The difference in the values of the two level set functions at $t = 1$ is less than $10^{-10}$, even on a relatively coarse $101^2$ grid.

### 2.9.1  Open Curves for Spiral Crystal Growth

This section describes the routines `smerekaSpirals` and `visualizeOpenCurve` in the directory `Examples/Vector/`.

This example comes from [31] and is mentioned briefly in [24, chapter 10.3]. Crystals are observed to grow in spiral patterns under a variety of growth conditions. The mechanism of this growth is believed to be a step discontinuity of the crystal surface which grows in a direction normal to itself, but is anchored at its ends to screw dislocations in the crystal structure. The model proposed is motion in the direction normal to the step-line at speed

$$v_n(x) = v_\infty (1 - \lambda \kappa(x)),$$

where $v_\infty$ is the speed of a straight step-line, $\lambda$ is a fixed parameter, and $\kappa(x)$ is the local curvature of the step-line. If the step-lines were closed curves, this motion could be easily simulated using the basic toolbox and a scalar implicit surface representation of the step-line in $\mathbb{R}^2$; however, because the step-lines terminate at fixed locations in the state space, we turn to vector level sets.

The idea is simple: use one level set function $\phi(x, t)$ to track the motion of the true step-line curve and an artificial step-line curve, the union of which is a closed curve. The sign of

76

a second level set function $\psi$ is used to distinguish between the physical and the artificial step-lines. Mathematically, the physical step-line is represented by

$$\Gamma(t) = \{x \in \mathbb{R}^2 \mid \phi(x, t) = 0 \wedge \psi(x, t) > 0\}. \tag{44}$$

The screw dislocations which anchor the ends of the physical step-line are those locations where both $\phi$ and $\psi$ are zero. Motion of the step-line is then simulated by solving the pair of HJ PDEs

$$\begin{aligned} D_t \phi - v_\infty \operatorname{sign}(\psi) \left[\lambda \operatorname{sign}(\psi)\kappa(\phi) - 1\right] \|\nabla \phi\| &= 0, \\ D_t \psi - v_\infty \operatorname{sign}(\phi) \left[\lambda \operatorname{sign}(\phi)\kappa(\psi) + 1\right] \|\nabla \psi\| &= 0, \end{aligned} \tag{45}$$

where $\kappa(\xi)$ is the local curvature of the level set function $\xi(x, t)$. Note that (45) is in a slightly different form than [31, equations (9) and (10)]—we have flipped a sign in the latter equation, and then rearranged to regain the symmetry in the two equations. For $\phi$, this PDE is simply motion by mean curvature (9) with speed $b(x, t) = v_\infty \lambda \operatorname{sign}(\psi(x, t))^2$ and motion in the normal direction (6) with speed $a(x, t) = v_\infty \operatorname{sign}(\psi(x, t))$. For $\psi$, the same motions apply after substituting $\operatorname{sign}(\phi(x, t))$ and negating $a(x, t)$. Note that we do not assume $\operatorname{sign}(\xi)^2 = 1$, since we use the mollification

$$\operatorname{sign}(\xi) = \begin{cases} +1, & \xi > \epsilon; \\ \xi/\epsilon, & |\xi| \leq \epsilon; \\ -1, & \xi < -\epsilon. \end{cases}$$

In our simulations, we choose $\epsilon = 4\Delta x$, where $\Delta x = \max_i \texttt{grid.dx}(i)$.

The following routine implements (45) and can be used to recreate selected figures from [31]. In the process it demonstrates the use of `termCurvature`, `termNormal`, and `termSum` with vector level sets. The postTimestep routine `postTimestepReinit` is also demonstrated; it is used to run a single iteration of the reinitialization equation (7) on $\phi$ and $\psi$ after each step of simulating (45). Note that $v_\infty = 1$ in all of these examples.

[ dataCurve, dataMask, g, tPlot ] = smerekaSpirals(whichFig, exactCopy, accuracy, tMax): Demonstrate the evolution of open curves in $\mathbb{R}^2$ using vector level sets. All input parameters have defaults. The $\lambda$ parameter in (45), plot times, and the initial conditions are drawn from among selected figures in [31]; the user may choose with `whichFig`. The user may also specify whether to recreate the exact sample times in the figure or to just show evenly spaced sample times using the boolean parameter `exactCopy`. The usual options are available for `accuracy`. The optional parameter `tMax` can be used to terminate the simulation early or late. The return parameters `dataCurve` and `dataMask` are $\phi$ and $\psi$ respectively. If the user does not request return parameter `tPlot`, `dataCurve` and `dataMask` are arrays containing $\phi$ and $\psi$ at the final time. If the user requests `tPlot`, then `dataCurve` and `dataMask` are cell vectors whose
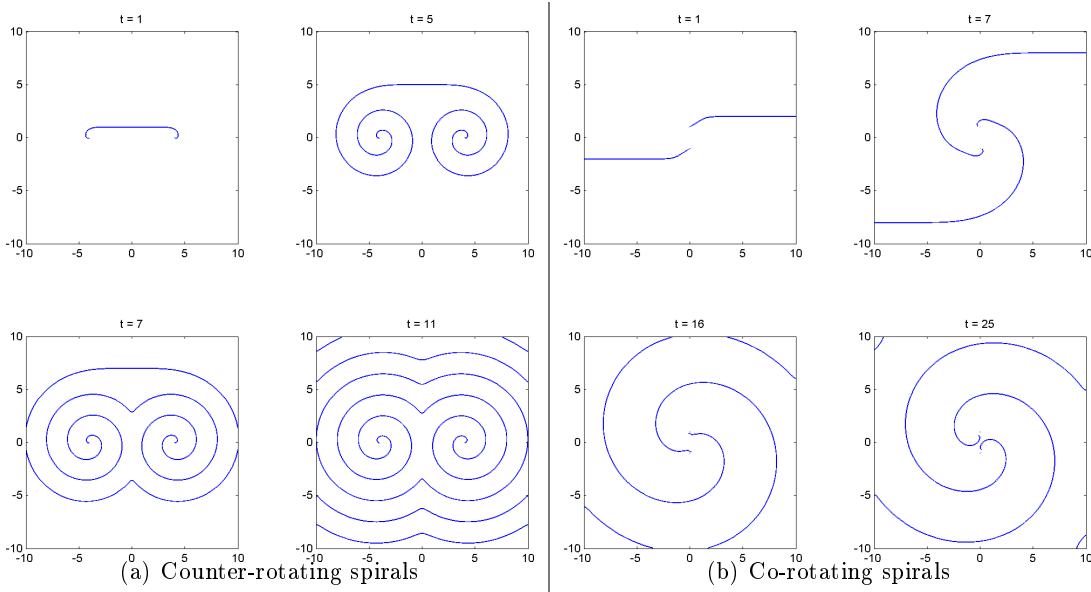
Figure 30: Spiral motions of step lines demonstrating the use of vector level sets for open curve evolution in two dimensions. Compare with [31, figures 4 and 12].

elements are $\phi$ and $\psi$ at each plot time. The return parameter `g` returns the computational grid. The return parameter `tPlot` is a vector of times at which the solution was plotted; requesting it also controls the form of the output parameters `dataCurve` and `dataMask` as described above.

Recreations of [31, figures 4 and 12] are shown in figure 30. Resolution of the tight spiraling patterns demands a fine grid; for the figures we used `grid.N` $= 401^2$. The combination of fine grid and motion by mean curvature (with timestep proportional to $(\Delta x)^{-2}$) means that these systems take a long time to simulate fully—the co-rotating spirals took nearly 24 hours. Consequently, we have not yet tried the resolution of $501^2$ recommended in [31] (which would take roughly twice as long). Nor have we yet tracked down the source of the discrepancy in our recreation of [31, figure 12]. The initial conditions used in `smerekaSpirals` for `whichFig` $= 12$ are not the same as those suggested in [31, equation (21)] (we could not get the definition of $\psi(x,0)$ given there to work), so it is not clear whether the difference in results springs from the slightly coarser grid or from differences in initial conditions.

The spirals in figure 30 are not trivial to generate, since $\Gamma(t)$ in (44) cannot be directly visualized with MATLAB's contour plotters. Following [31], we use MATLAB's `contour` function to visualize the $\frac{1}{2}\Delta x$ isocontour of the function

$$\theta(x,t) = \begin{cases} |\phi(x,t)|, & \psi(x,t) \geq 0; \\ +1, & \psi(x,t) < 0. \end{cases}$$

78

The resulting curves are actually two contour lines very close together, but for the grid resolutions used the two lines are indistinguishable unless we zoom in on the plot. Computing $\theta$ and plotting the result are accomplished with the following routine.

[ hCurve, hAll ] = visualizeOpenCurve(grid, curve, mask, showAll, titleStr): Visualize an open curve defined by two level set functions according to (44). Plotting is performed in the current figure window and axis. The grid is given by grid, and the parameters curve and mask are arrays of size grid.shape containing $\phi$ and $\psi$ respectively. If the optional boolean parameter showAll is true, then the entire zero level set of $\phi$ is plotted in addition to the open curve (the two portions are distinguished by color). The optional string argument titleStr causes a title to be added to the plot. The return arguments hCurve and hAll are graphics handles to the contours created. If showAll is false (or missing) then hAll will be the empty vector [].

## 2.10 Reinitialization Examples from Russo & Smereka [27]

This section describes functions in the directory Examples/RussoSmereka/.

Reinitialization using the reinitialization PDE (7) has a number of advantages over competing methods like fast marching: there is no need to find the front explicitly and schemes with high orders of accuracy can be applied to approximate the temporal and spatial derivatives. Unfortunately, despite smoothing of the signum function there is usually some movement of the interface; for example, see figure 5. In [27], the authors propose a scheme intermediate between explicitly locating the interface and the plain reinitialization equation. In this modified scheme, the reinitialization equation is solved as usual for all nodes except those adjacent to the interface, where adjacent means that they have at least one neighbour node on the other side of the interface. For nodes adjacent to the interface, the distance to the interface is approximated and an alternative update is used that results in less movement of the interface.

The first order accurate version of this "subcell fix" is implemented by termReinit. In this section we demonstrate the advantages of this modification by recreating some of the examples from [27]. All of these codes use termReinit directly (they do not call a helper function such as signedDistanceIterative).

[ data, g, data0 ] = reinit1D(apply_fix, accuracy): Recreates [27, figures 2 and 5]. Adjusting the axis bounds on these figures can generate [27, figures 3 and 6]. The boolean apply_fix parameter determines whether or not the subcell fix is applied. The accuracy parameter choices are as usual; however, it should be noted that the subcell fix is itself only first order accurate. The results of running this code are shown in figure 31.

(a) Without subcell fix.

(b) Without subcell fix (zoomed view).

(c) With subcell fix.
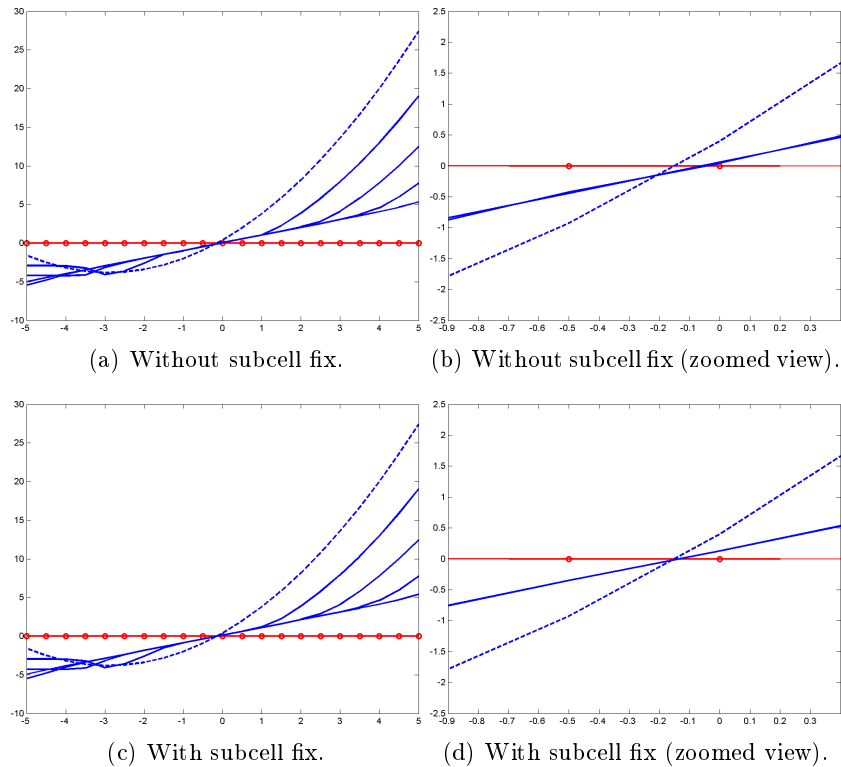
(d) With subcell fix (zoomed view).

Figure 31: One dimensional reinitialization problem from [27, sections 2 and 3], as recreated by `reinit1D` with `accuracy = 'low'`. The dashed line is the initial implicit surface. The horizontal line represents the grid and the circles on this line are the node locations. The other solid lines are iterations 3, 6, 9, and 12 of the reinitialization. Compare with [27, figures 2, 3, 5 and 6].

[ data, g, data0 ] = reinitCircle(apply_fix, accuracy, show_nodes, grid_anisotropy, dim): Recreates [27, figures 7 and 8]. The boolean apply_fix parameter determines whether or not the subcell fix is applied. The accuracy parameter choices are as usual. The boolean show_nodes determines whether the (rather coarse) grid is shown by light grey dots. The double grid_anisotropy parameter allows the example to be run on grids that have different node spacing in the horizontal direction (the analysis and examples in [27] assume an isotropic grid). The integer dim allows for either two or three dimensional runs. Figure 32 shows the results of two sample runs of this code.

[ data_out, g ] = reinitEllipse(apply_fix, accuracy, num_nodes, do_figure_9): Recreates [27, figure 9]. The boolean apply_fix parameter determines whether or not the subcell fix is applied. The accuracy parameter choices are as usual. The num_nodes parameter controls the resolution of the grid; the default setting of 200 is taken from [27]. The boolean do_figure_9 determines whether the user wants graphical output (to recre-
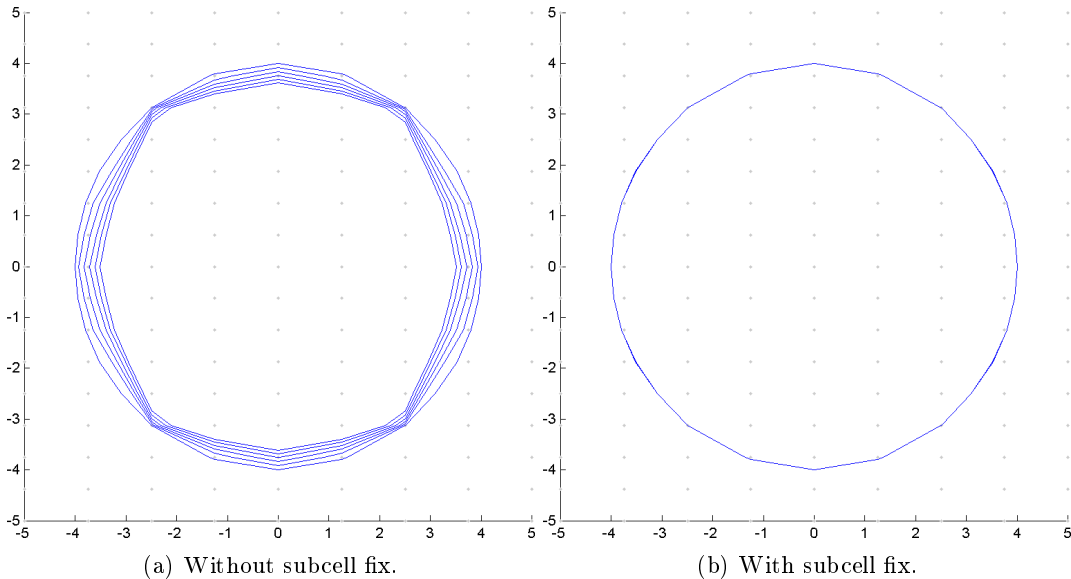
(a) Without subcell fix.  (b) With subcell fix.

Figure 32: A two dimensional reinitialization problem from [27, section 4] as recreated by `reinitCircle(apply_fix, 'low', 1, 2, 2)`, where `apply_fix` $\in \{0, 1\}$. The initial conditions are a signed distance function representing a circle, so reinitialization should not modify the function or its zero isosurface. Both figures show contours at iterations $160i$ for $i = 0, 1, \ldots, 5$. The case without the subcell fix shows that over many iterations the zero isosurface shrinks, while the contours are indistinguishable when the subcell fix is applied. The two subplots correspond to [27, figures 7 and 8], except that the grid is denoted by small dots and the grid nodes have twice the spacing in the horizontal direction as in the vertical.

ate [27, figure 9]), or just error information (to recreate [27, figures 10 and 11]; see `ellipseError` below). The return parameter `data_out` is a structure containing information about the error at each timestep of the reinitialization. The results of running `reinitEllipse` with the default settings are shown in figure 33.

`ellipseError`: A script file which uses multiple calls to `reinitEllipse` to recreate [27, figures 10 and 11]. With some editing, it can also recreate [27, figure 15]. Results are shown in figure 34.

The fact that the recreations are not exact matches has several possible causes. First, the scheme used here for solving the reinitialization equation away from the front is the scheme from [12], which is not the same as what was used in [27]. Second, the default MATLAB visualization methods (linear interpolation) are used to generate the figures here, while [27] used cubic splines in at least a few of the examples. Third, it is not clear in [27] whether the examples use the "robust" distance approximation scheme; all of the recreations above do make use of the robust version. Fourth, some of the parameters given in the text of [27] clearly
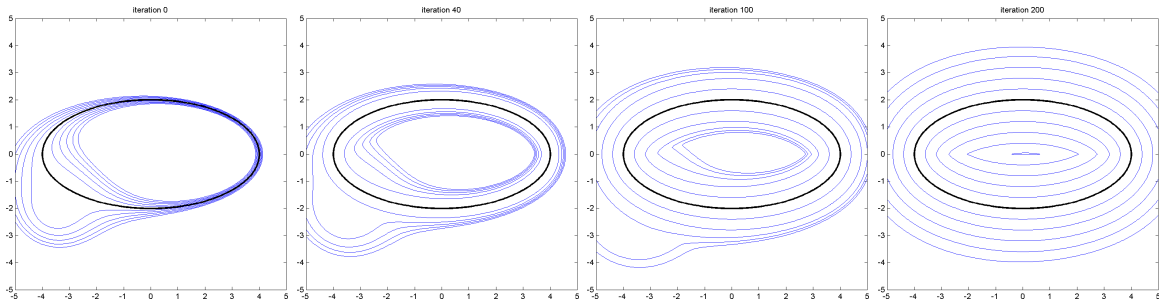
Figure 33: A more challenging two dimensional reinitialization problem from [27, section 4], as recreated by the default settings of `reinitEllipse`. The thicker line in the middle of the contours is the zero isosurface. In the left-most subplot the implicit surface function starts with regions where the gradient magnitude is both large and small, but by the final iteration in the right-most subplot, the contours are evenly spaced and hence the gradient magnitude is roughly equal everywhere. Compare with [27, figure 9].

disagreed with the figures, and so parameter settings were altered to make the recreations closer to the figures (most notably the contours and iterations in figure 33).

The source of the discrepancy in the interface location absolute error metric of the recreation in figure 34 of [27, figure 11] is not known.

## 2.11 Testing Routines

This section describes functions in `Examples/Test`.

### 2.11.1 Initial Conditions

Several script-like functions were written to test the initial condition routines for basic shapes and set operations for constructive solid geometry (see section 3.3). Some of the outputs from these scripts are shown in figure 35.

`initialConditionsTest1D()`: Creates a sequence of shapes defined by implicit surface functions in a one dimensional state space. In one dimension, an implicitly defined shape is always an interval, although one or both endpoints may be infinite. Plotting the intervals is not terribly exciting, so the entire implicit surface function for each shape is displayed as a function plot, state vs function value. The implicitly defined interval for each plot is the region in which the function value is negative.

(a) Average error, first order accuracy throughout.

(b) Interface error, first order accuracy throughout.

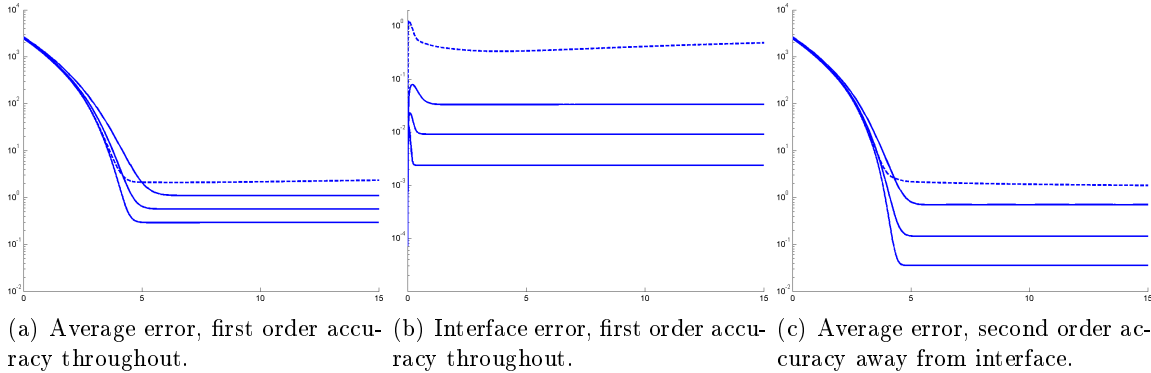(c) Average error, second order accuracy away from interface.

Figure 34: Demonstration that the reinitialization PDE with the subcell fix eventually reaches a fixpoint for the ellipse problem shown in figure 33, a fixpoint which may not be achieved without the fix. Compare with [27, figures 10, 11 and 15], but note that the vertical axes may not be the same. The vertical axes are log-scale error, and the horizontal axes are integration time. The dashed line corresponds to reinitialization without the subcell fix on a $200^2$ grid. The three solid lines correspond to reinitialization with the subcell fix on grids of size $50^2$, $100^2$ and $200^2$. The left two subplots are generated by `ellipseError`, and use first order of accuracy approximations throughout. To generate the right subplot, `ellipseError` must be edited to set `accuracy = 'medium'`, which then uses second order of accuracy in time and for nodes not adjacent to the interface.

`initialConditionsTest2D()`: Creates a sequence of shapes defined by implicit surface functions in a two dimensional state space. The two dimensional implicit surfaces are shown in one figure window by contour plots, while the implicit surface functions themselves appear in a separate window as surface plots.

`initialConditionsTest3D()`: Creates a sequence of shapes defined by implicit surface functions in a three dimensional state space. The three dimensional implicit surfaces are shown as isosurfaces, because the implicit surface functions themselves are rather challenging to visualize.

### 2.11.2 Derivative Approximations

Do the high resolution (high order) approximation schemes live up to their billing? A pair of routines were designed to test the functions (see section 3.4.1) and determine their errors, convergence rates and execution times. Given proper input data, solutions of the time-dependent HJ PDEs that we solve with this toolbox should remain continuous, although they may not be differentiable everywhere. In order to test whether the approximation schemes correctly handle this situation, the test function is chosen to be continuous but with discontinuities in the derivative.
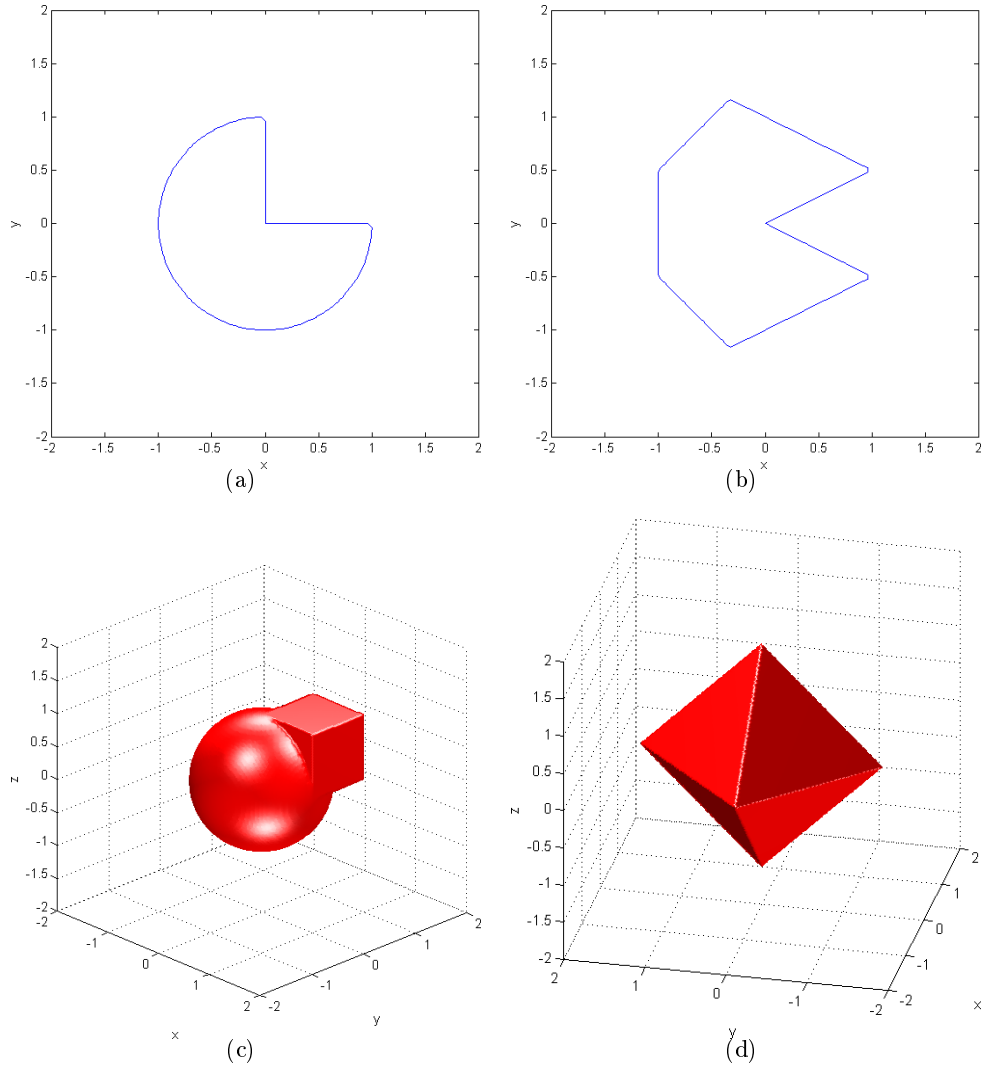
Figure 35: Examples of implicitly defined surfaces and sets built by constructive solid geometry operations from basic shapes. The top row is generated by `initialConditionsTest2D` and the bottom row by `initialConditionsTest3D`. Figure 35(a) shows a square subtracted from a circle, while figure 35(b) shows a nonconvex polygon constructed by intersections and unions of hyperplanes. Figure 35(c) shows the union of a sphere and a cube, and figure 35(d) shows an octohedron constructed by the intersection of eight hyperplanes.

84

[ errorL, errorR, time ] = firstDerivSpatialTest1(scheme, dim, whichDim, dx):
Computes the errors in the left and right approximations for a single scheme on a single grid. The scheme is specified by the function handle scheme. The dim dimensional grid has periodic boundary conditions in every dimension and grid spacing dx. The derivative is taken in dimension whichDim. Letting $x_d$ be the whichDim component of the state vector $x$, the test function is

$$
f(x) = \begin{cases} \sin\left(2\pi x_d + \frac{\pi}{4}\right), & \text{for } 0 \le x_d < \frac{1}{4}; \\ \sin\left(2\pi x_d - \frac{\pi}{4}\right), & \text{for } \frac{1}{4} \le x_d < \frac{1}{2}; \\ \sin\left(2\pi x_d\right) + 1, & \text{for } \frac{1}{2} \le x_d < 1. \end{cases}
$$

Note that the test function is constant in all dimensions other than whichDim. In order to correctly catch the discontinuities, dx should be an integer division of 1/4. Calling this function without output arguments will generate a figure showing the test function, its analytic derivative, and the approximations. Statistics on the quality of the approximation will be displayed. There will be no display if any of the output parameters is requested. The outputs errorL and errorR will be structures with the scalar fields maximum (maximum error over the nodes), average (average error over all nodes), rms (root mean square error over all nodes), and jumps (average error over the three nodes that lie on a jump, assuming that dx was correctly chosen). The output time will be the time (in seconds) required to evaluate scheme, as reported by cputime. This procedure is appended Test1 in the hopes that additional procedures with the same interface but different test functions $f(x)$ will be implemented.

firstDerivSpatialConverge: A script file to demonstrate the convergence rate of the various first derivative approximation schemes. The schemes, grid sizes and grid dimensions can be specified inside the script file. The function firstDerivSpatialTest1 is used to generate the error estimates, although alternative procedures with different test functions could easily be substituted. Four figures are generated, showing the convergence rate in maximum error, average error, root-mean-square error, and average jump error (maximum jump error is not computed, since it is almost always the overall maximum error). Execution times are also displayed.

As a demonstration, figure 36 shows the results of running firstDerivSpatialConverge on all of the upwind approximations from section 3.4.1: upwindFirstFirst, upwindFirstENO2, upwindFirstENO3a, upwindFirstENO3b, upwindFirstWENO5a, and upwindFirstWENO5b. The errors for the two forms of ENO3 and WENO5 turn out to be indistinguishable. The schemes behave as expected, with the exception of the WENO5 schemes. They do not achieve fifth order accuracy, although they do show higher order convergence than the basic ENO3 scheme. Furthermore, although they consistently outperform the ENO3 scheme in average error, the WENO5 schemes are worse in maximum error and errors near the jumps (quantities which tend to be closely related).
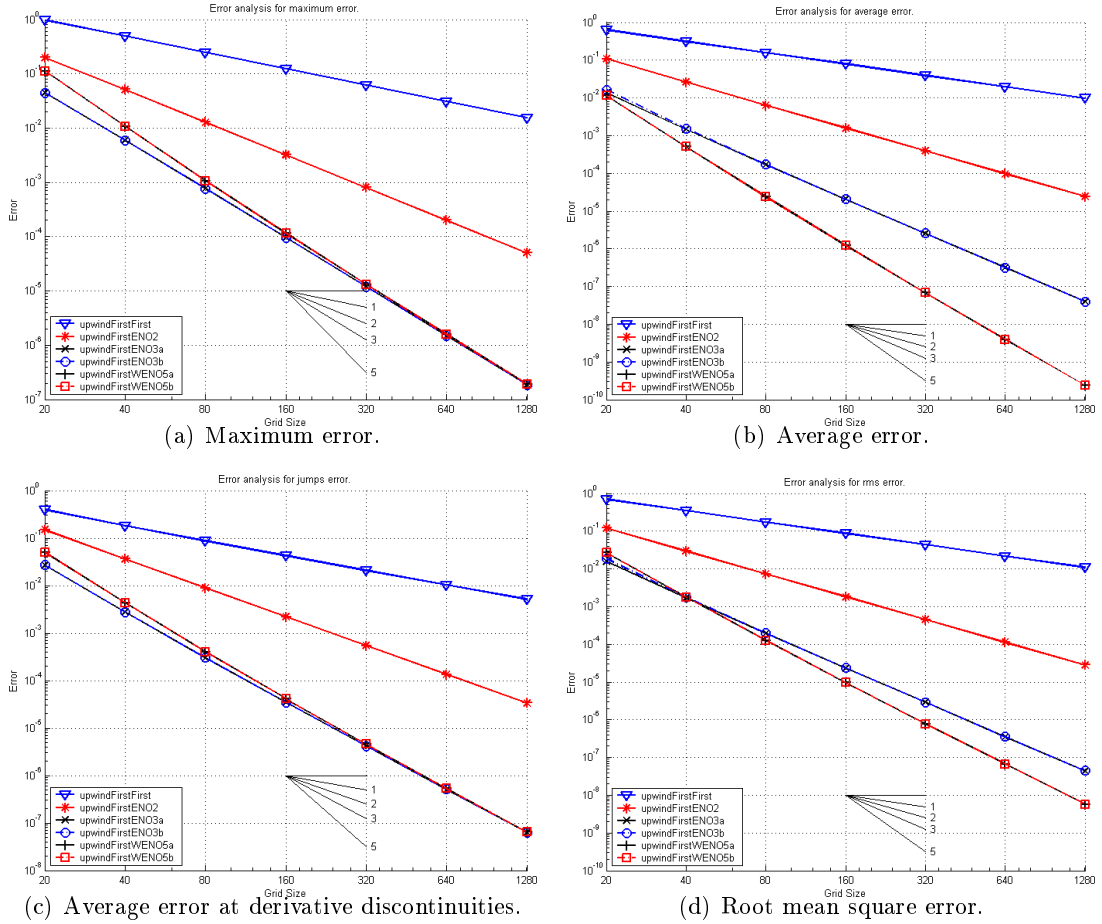
(a) Maximum error.

(b) Average error.

(c) Average error at derivative discontinuities.

(d) Root mean square error.

Figure 36: Convergence rates demonstrated by the various upwind approximations of the first deriviative from section 3.4.1, as generated by `firstDerivSpatialConverge` on the test function in `firstDerivSpatialTest1` in two dimensions. The short lines in the middle of the bottom of each figure show the slopes corresponding to first, second, third and fifth order convergence.

| Scheme | Relative Execution Time |
|---|---|
| `upwindFirstFirst` | 1 |
| `upwindFirstENO2` | 5 |
| `upwindFirstENO3a` | 12 |
| `upwindFirstENO3b` | 25 |
| `upwindFirstWENO5a` | 20 |
| `upwindFirstWENO5b` | 28 |

Table 2: Approximate speeds of the upwind approximation schemes from section 3.4.1 (relative to the speed of `upwindFirstFirst`), as measured by `firstDerivSpatialConverge` on a two dimensional example.

The choice of scheme will be driven primarily by desired accuracy and the need for speed. The relative speeds of the six schemes on the $N = 1280$ grid is shown in table 2, although results will vary depending on the hardware, dimension and grid size. In most simple interface motion examples, the spatial derivative approximation plays the largest roll in determining the overall computation time and the accuracy of the results, so choosing an appropriate scheme is important. Clearly, the ENO3b and WENO3b schemes should not be used for complex examples, since they deliver the same results as ENO3a and WENO5a (respectively) at significantly higher computational cost. For that reason, the functions `upwindFirstENO3` and `upwindFirstWENO5` are wrappers for `upwindFirstENO3a` and `upwindFirstWENO5a` respectively. Beyond that, however, the user must determine the appropriate tradeoff between accuracy and speed. In practice, we often run initial tests with low resolution schemes, and save the high resolution schemes for producing final results.

### 2.11.3 Other Test Routines

Some miscellaneous testing routines.

`[ data, g, data0 ] = reinitTest(initialType, accuracy, displayType)`: Demonstrates the `signedDistanceIterative` helper routine. The parameters and results are identical to those of `reinitDemo` from section 2.2.1, except that this routine uses `signedDistanceIterative` to handle the main loop of the PDE approximation.

`argumentSemanticsTest(loops, matSize)`: MATLAB's programming language uses pass-by-value semantics, but purports to achieve pass-by-reference speed by avoiding the creation of copies until absolutely necessary; for example, when an input argument is modified. This routine can be used to demonstrate the veracity of that claim, as well as test whether array reshaping (through either `reshape` or (`:`)) is inexpensive.

ghostCell: A script file to test the routines for adding ghost cells implementing various types of boundary conditions to data arrays in dimensions one and two. The results must be examined manually to determine whether the correct ghost cell values were added in the correct places. Because the file is a script, parameters can only be modified by editing the file directly; however, all the internal variables of the script are available in the base workspace at the completion of the script (useful for debugging).

# 3 Code Components

This section discusses the routines in the directory `Kernel`. It is designed primarily as a reference, although the best reference is probably the help entries for the routines themselves, which are found at the top of each function's source m-file and can be accessed with MATLAB's `help` command.

## 3.1 Grids

This section discusses functions found in the directory `Kernel/Grids`.

The goal of this toolbox is to allow simple solution of simple interface motion problems. Because the computational grid affects virtually every operation in a hyperbolic PDE solver, nowhere is the decision to pursue simplicity over generality more defining than in our choice of grids. While there are many problems that cannot be solved to high accuracy or within reasonable computational time without resorting to adaptive and/or unstructured grids, the complexity of the data structures for such grids makes them poorly suited for simple problems or the MATLAB interpreted programming environment.

Consequently, we have adopted a very simple grid structure: a fixed rectangular Euclidean mesh. The grid cells are of fixed size, although the spacing for each dimension may be chosen independently. A `grid` is represented by a structure with fields:

`grid.dim`: The dimension of the grid. Typically between one and four, although the code should work in any dimension.

`grid.min`: A column vector specifying the lower left corner of the computational domain.

`grid.max`: A column vector specifying the upper right corner of the computational domain.

`grid.bdry`: A cell column vector. Each element is a function handle pointing to the boundary condition (see section 3.2), which provides data values for nodes which fall outside the computational domain in that dimension.

`grid.bdryData`: A cell column vector. Each element provides parameters for the corresponding `grid.bdry` element.

`grid.N`: A column vector specifying the number of grid nodes in each dimension.

`grid.dx`: A column vector specifying the grid cell spacing in each dimension.

`grid.vs`: A cell column vector. Each element contains a regular column vector giving the node locations in the corresponding dimension. Generated by `grid.vs{d}` `= linspace(grid.min(d), grid.max(d), grid.N(d))`.

grid.xs: A cell column vector. Each element contains an array giving the node locations for each node in the entire grid. Generated by [ grid.xs{1:grid.dim} ] = ndgrid(grid.vs{:}).

grid.axis: A row vector specifying the computational domain boundary in a format suitable to pass to MATLAB's axis command.

grid.shape: A row vector specifying the number of nodes in each dimension in a format suitable to pass to MATLAB's reshape command. Specifically

$$\texttt{grid.shape} = \begin{cases} [\texttt{grid.N} \quad 1], & \text{if } \texttt{grid.dim} = 1; \\ \texttt{grid.N}', & \text{otherwise.} \end{cases}$$

The special case for one dimensional grids is required because MATLAB does not support true one dimensional arrays. If data is a data array defined on grid, then grid.shape == size(data).

### 3.1.1 Creating A Grid

Manually entering all of the fields that define a grid would be tedious and prone to inconsistencies. Therefore, most will be automatically generated by a call to processGrid. Typically, only the fields grid.dim, grid.min, grid.max, grid.bdry, grid.bdryData and one of grid.N or grid.dx need be supplied by the user.

gridOut = processGrid(gridIn, data): Fill in the fields missing from a grid structure. Where possible, missing fields in gridIn will be automatically generated. Some consistency checking is also performed on the fields that already exist. Some fields have default values, which can be seen in the help entry. This function can be safely called multiple times on the same grid structure (the second call will only invoke consistency checks), although it can be rather slow to execute. The optional second argument is only checked to ensure that ndims(data) and size(data) are consistent with gridIn.dim and gridIn.N respectively.

The user should ensure that processGrid is called before a grid structure is passed into any of the other routines in this toolbox. The resulting grid will be a grid.dim dimensional array with grid.N($d$) nodes in dimension $d$ for $d = 1, \ldots,$ grid.dim.

### 3.1.2   Allocating, Passing and Saving Grids

The grid.xs field will generally be much larger than any other in grid, since it will have a total of grid.dim * prod(grid.N) entries. While it is large, alternative schemes for vectorizing the level set computations inevitably lead to allocating multiple copies of similarly large state arrays at different levels in the call stack, and so it was decided to include this single copy of the state array in the grid structure. The large size of this field will not reduce computational efficiency as long as the grid structure and its fields are not modified within any of the functions to which it is passed; so far we have found no reason to do so within any of our examples.

When saving a grid to disk, the command sequence

$$grid = \text{rmfield}(grid,' xs');$$
$$\text{save}('someFilename', grid);$$

can be used to remove this field and hence enormously reduce the size of the resulting file. The field can be easily regenerated when loading by using the command sequence

$$grid = \text{load}('someFilename');$$
$$grid = \text{processGrid}(grid);.$$

### 3.1.3   Other Grid Functions

There are two incompatible methods provided by MATLAB for generating state arrays of the sort stored in the grid.xs field: meshgrid and ndgrid. MATLAB's indexing system grew out of matrix indexing, which is given in the order rows (vertical) and then columns (horizontal). This index ordering is incompatible with the ordering usually used in visualization, which in two dimensions would be $x$ (horizontal) and then $y$ (vertical). To get around this incompatibility, MATLAB's default method for generating location arrays—meshgrid—effectively swaps the order of its first two arguments. This swapping works fine when working in dimensions two and three in MATLAB, but it begins to make less sense when transfering data to external applications and in higher dimensions; in fact, meshgrid only works in dimension three or less. For grids which are indexed consistently in any dimension, MATLAB provides ndgrid, which is equivalent to meshgrid but does not perform the dimension swap. Because it is designed for arbitrary dimension, the Toolbox's grid structure is set up by processGrid using ndgrid.

This choice means that interpn (which requires ndgrid-based grids) should be used when working in dimension two or higher for interpolating data generated by the Toolbox, rather than interp2 or interp3 (which require meshgrid-based grids). In one dimension, interp1 (or even spline) can be used.

The choice of **ndgrid**-based grids also causes two visualization issues. The first is that some MATLAB visualization routines require that their data be laid out in a **meshgrid**-based array; for example, **contourslice**, **slice**, and **isonormals**. To handle such routines, the following function is provided:

[ mesh_xs, mesh_data1, mesh_data2, ...]  = gridnd2mesh(grid, nd_data1, nd_data2, ...): Converts an **ndgrid**-based grid and its associated implicit surface function data arrays into a **meshgrid**-based grid. The **ndgrid**-based grid is specified with a standard Toolbox grid structure in **grid**. The optional **grid.dim** dimensional array(s) **nd_data1**, **nd_data2**, ... store the corresponding implicit surface function(s). The output **mesh_xs** is a cell vector of length **grid.dim** containing the **meshgrid**-based equivalent of **grid.xs**. The decision to return only the **grid.xs** component of the grid structure was purposeful, to try to ensure that all grid structures are **ndgrid**-based in the Toolbox. For the purpose of calling MATLAB visualization routines, **mesh_xs** should be sufficient. Any Toolbox visualization routines will use the **grid** structure directly. The other output parameter(s) **mesh_data1**, **mesh_data2**, ... contain the appropriately modified versions of the input parameter(s) **nd_data1**, **nd_data2**, ... when those are provided.

For an example of how **gridnd2mesh** is used, look at the three dimensional visualization options in the source code of **visualizeLevelSet** (section 3.7.6).

A more subtle problem with the incompatibility between grid types is that almost all MATLAB visualization routines in dimensions two or higher silently call **meshgrid** when they are not provided with grid arguments. Consequently, users should *always* explicitly pass the grid (in the form of the **grid.xs** cell vector) when calling any MATLAB visualization routine. For more comments, see section 1.4.

These grid incompatibility visualization issues do not arise if visualization is performed by **visualizeLevelSet** (see section 3.7.6). Users interested in examples of correct calls to MATLAB's visualization routines should look at the source code for **visualizeLevelSet**.

## 3.2 Boundary Conditions

This section discusses functions found in the directory **Kernel/BoundaryCondition**.

The computational domain is finite, and so the finite difference stencils we use to approximate the spatial derivatives of the HJ PDE will extend beyond the edge of the grid when working on nodes near that edge. In order to manage this process, every face of the computational domain must be associated with a boundary condition. This association is represented by

function handles passed in the `grid.bdry` field of the grid structure described in section 3.1. In general, each dimension can have its own boundary conditions, although the upper and lower boundaries in a particular dimension must use the same boundary condition function.

The boundary condition functions are called by the spatial derivative approximations (see section 3.4). When called for a particular dimension, they add an appropriate number of ghost nodes—the stencil width specified by the spatial derivative approximation—to the upper and lower sides of the data array in that dimension. The values placed in these ghost nodes are determined by the type of boundary condition.

`addGhostPeriodic`: Values from the lower end of the array are copied to the upper ghost nodes, and vice versa. This boundary condition requires no additional parameters.

`addGhostDirichlet`: A constant value is placed into the ghost nodes. Different constants may be chosen for the upper and lower ghost nodes. The values are passed as parameters.

`addGhostNeumann`: The ghost nodes are filled with data linearly extrapolated from the computational boundary so as to have a constant specified derivative normal to the boundary. Different constants may be chosen for the upper and lower ghost nodes. The constants are passed as parameters.

`addGhostExtrapolate`: The ghost nodes are filled with data linearly extrapolated from the computational boundary so as to have a slope towards or away from the zero level set. The choice of towards or away from the zero level set is passed as a parameter. While this is not a traditional PDE boundary condition, it proves quite useful in level set computations for domains with inflow boundaries that have no physically appropriate boundary conditions. By choosing to extrapolate away from zero, the ghost cells will never falsely imply the existence of a "ghost" interface beyond the computational domain, and hence lend stability to a potentially unstable nonphysical computational domain boundary. All of the examples use this boundary condition when the periodic boundary condition cannot be justified.

For more details on the parameters required by each boundary condition function, see the individual help entries. All four boundary condition functions use the same call structure, which we demonstrate with `addGhostExtrapolate`.

`dataOut = addGhostExtrapolate(dataIn, dim, width, ghostData)`: Adds `width` ghost cells in dimension `dim` to the top and bottom of the data array `dataIn`. These ghost cells are filled with data linearly extrapolated from the two nodes nearest the boundary in the appropriate dimension. The sign of the extrapolation is chosen so as to extrapolate away from or towards the zero level set, as specified by the boolean field

`ghostData.towardZero` (defaults to `false`). For example, if `dataIn` is two dimensional of size `grid.shape`, then `dim = 2`, `width = 1` and `ghostData.towardZero = 0` would result in a two dimensional `dataOut` of size `grid.shape` + [ `0, 2` ]$^T$ with values generated by

$$
\begin{aligned}
\texttt{dataOut}(:,1) =& \texttt{dataIn}(:,1) \\
& + \text{sign}(\texttt{dataIn}(:,1)) |\texttt{dataIn}(:,1) - \texttt{dataIn}(:,2)| \\
\texttt{dataOut}(:,2:\texttt{end}-1) =& \texttt{dataIn}(:,1:\texttt{end}) \\
\texttt{dataOut}(:,\texttt{end}) =& \texttt{dataIn}(:,\texttt{end}) \\
& + \text{sign}(\texttt{dataIn}(:,\texttt{end})) |\texttt{dataIn}(:,\texttt{end}) - \texttt{dataIn}(:,\texttt{end}-1)|
\end{aligned}
$$

Function handles to the boundary condition functions described above are passed as the elements of the cell vector `grid.bdry` of the grid structure. Each is called on a single dimension at a time. While this one dimension at a time method reduces the memory requirements of adding ghost cells when working with the one dimension at a time first order spatial derivative approximations in section 3.4.1, it is sometimes necessary to create ghost cells on every side of the data array at once. Two helper routines are provided for this purpose.

`dataOut = addGhostAllDims(grid, dataIn, width)`: Adds `width` ghost cells to the top and bottom of every dimension of the data array `dataIn`, according to the boundary conditions specified in `grid.bdry`.

`[ vs, xs ] = addNodesAllDims(grid, width)`: Creates `vs` and `xs` cell vectors that correspond to those in `grid.vs` and `grid.xs`, but include the states of all the ghost nodes as well as the regular grid nodes. Note that `xs` can be very large, and hence this function can be expensive to evaluate.

The process of creating and releasing the memory for the ghost nodes at each timestep is clearly not the most efficient way to handle boundary conditions. Unfortunately, the alternative would be to preallocate sufficient memory in the data array for the ghost cells. The size of the preallocation would depend on the spatial derivative approximation, and would necessitate an offset indexing system to retrieve the true data from the array. Thus, we decided to use the slower method of repetitive ghost cell allocation rather than destroy the intuitively simple layout of the data array. A future object oriented version of this toolbox may be able to revisit this decision and achieve both goals with a single implementation.

## 3.3  Initial Conditions

This section describes functions in `Kernel/InitialConditions`.

A major advantage of implicit surface representations is the ease with which complex shapes can be created through operations from constructive solid geometry. Simple algebraic functions can create implicit surface functions for basic shapes—circles, spheres, cylinders, squares, cubes, rectangles, hyperplanes, and polygons, to name just a few. These shapes can then be combined by unions, intersections, complements and set differences to form more complex shapes. When sets are represented by implicit surface functions, each of these set operations has a simple corresponding mathematical operation.

In many cases, including most of the examples in this toolbox, the initial conditions involve implicit surfaces so simple that their implicit surface functions are computed explicitly in the main routine. However, for those not so familiar with implicit surface functions, the functions in this section were recently added to the toolbox to simplify the construction of initial conditions. They may also be used for other tasks, such as masking functions (see section 2.2.3).

### 3.3.1  Basic Shapes

This section describes functions in `Kernel/InitialConditions/BasicShapes`.

Routines are currently provided to create implicit surface functions for spheres (including circles), cylinders, rectangles (including cubes and squares) and hyperplanes. Future shapes could include cones and ellipses, among others. At present the cylinders and rectangles must be aligned with the coordinate axes, although that restriction could be removed.

The sphere and cylinder routines both produce signed distance functions. Cylinders must be coordinate axis aligned.

`data = shapeSphere(grid, center, radius)`: Constructs a signed distance function on the computational grid `grid` for a `grid.dim` dimensional sphere centered at `center` of radius `radius`. The parameter `center` should be a vector of length `grid.dim` and `radius` should be a positive scalar. In two dimensions this shape will be a circle, while in one dimension it will be an interval. The default values for `center` and `radius` generate a unit ball centered at the origin.

`data = shapeCylinder(grid, ignoreDims, center, radius)`: Constructs a signed distance function for an unbounded cylinder. In two dimensions this shape will be a slab, while in one dimension it will be an interval. More formally, a cylinder is a prism with a spherical cross-section. It could also be viewed as a sphere in which some dimensions of the state space are ignored. These dimensions are listed in the vector `ignoreDims`; the remaining parameters are the same as for `shapeSphere`. If `ignoreDims` is the empty vector, then a true sphere will be generated. For example, a traditional three

dimensional cylinder oriented vertically with unit radius with axis running through the origin would be created by `shapeCylinder(grid, 3, [ 0; 0; 0 ], 1)`, where `grid` is a three dimensional grid. The default values for `ignoreDims`, `center` and `radius` will generate a unit ball centered at the origin.

Two routines are provided for creating a rectangle, depending in which format the user prefers to describe the rectangle's size and location. Both versions require that the rectangle be aligned with the coordinate axes. Both allow for certain dimensions to be unbounded. Both are implemented using intersection operations on axis aligned hyperplanes, and so do not return true signed distance functions—inside the rectangle the implicit surface function will be a signed distance function, but outside the cones around the corners will not be signed distance (although they will have unit magnitude gradients).

`data = shapeRectangleByCorners(grid, lower, upper)`: Constructs an implicit surface function for an axis aligned (hyper) rectangle on the computational grid `grid`. The vectors `lower` and `upper` (of length `grid.dim`) specify diagonally opposite corners of the rectangle, where `lower(i) < upper(i)`. The rectangle may be unbounded in selected dimensions by choosing components of `lower` as `-Inf` or components of `upper` as `+Inf`. The default values for `lower` and `upper` generate a unit cube whose lower left corner is at the origin.

`data = shapeRectangleByCenter(grid, center, widths)`: Constructs an implicit surface function for an axis aligned (hyper) rectangle on the computational grid `grid`. The vector `center` (of length `grid.dim`) specifies the center of the rectangle, while the vector `widths` (of length `grid.dim`) specifies the full width of each dimension of the rectangle. This function is equivalent to calling `shapeRectangleByCorners` with `lower = center − width/2` and `upper = center + width/2`. The default values for `center` and `width` generate a unit cube centered at the origin.

A hyperplane is defined by its outward normal $n$ and a point through which it passes $x_0$. Given these two parameters, a signed distance function for the hyperplane is given by

$$\phi(x) = \frac{n^T(x - x_0)}{\|n\|}.$$

Hyperplanes can be combined using intersection (see section 3.3.2) to form convex polygons.

`data = shapeHyperplane(grid, normal, point)`: Constructs a signed distance function for a hyperplane on the computational grid `grid`. The vectors `normal` and `point` should be of length `grid.dim`. The vector `normal` specifies the outward normal of the hyperplane, while `point` specifies a point through which the hyperplane passes.

`data = shapeHyperplaneByPoints(grid, points, positivePoint)`: Constructs a signed
distance function for a hyperplane on the computational grid `grid`. Each of the
`grid.dim` rows of matrix `points` is a `grid.dim` dimensional vector specifying a point
through which the hyperplane passes. The direction of the normal vector of the
hyperplane—in other words, on which side of the hyperplane the implicit surface func-
tion is positive—can be specified in one of two ways. First, if the optional `grid.dim`
dimensional vector parameter `positivePoint` is supplied, then the normal direction
is chosen so that the implicit surface function has a positive value at the point rep-
resented by `positivePoint`. Note that `positivePoint` must not be too close to the
hyperplane, and must be within the bounds `grid`. If `positivePoint` is not provided,
then the points in `points` are assumed to be given in clockwise order and the normal
will be chosen to point out of the face of the clock. The output array `data` of size
`grid.size` contains the implicit surface function. **This routine has not been well
tested.**

Examination of the code in `shapeHyperplane` and `shapeHyperplaneByPoints` provides good
evidence of how `cellMatrixMultiply` and `cellMatrixAdd` can be used to simplify spa-
tially dependent matrix algebra, particularly with respect to the vector $x$ which is stored
in `grid.xs`.

### 3.3.2 Set Operations for Constructive Solid Geometry

This section describes functions in `Kernel/InitialConditions/SetOperations`.

Given sets $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_3$ defined by the implicit surface functions $\phi_1(x)$, $\phi_2(x)$ and $\phi_3(x)$
respectively, the set operations of intersection, union, difference and complement have corre-
spondingly simple mathematical descriptions in terms of the implicit surface functions.

$$
\begin{aligned}
\mathcal{G}_3 = \mathcal{G}_1 \cap \mathcal{G}_2 &\iff \phi_3(x) = \max(\phi_1(x), \phi_2(x)), \\
\mathcal{G}_3 = \mathcal{G}_1 \cup \mathcal{G}_2 &\iff \phi_3(x) = \min(\phi_1(x), \phi_2(x)), \\
\mathcal{G}_3 = \mathcal{G}_1 \setminus \mathcal{G}_2 &\iff \phi_3(x) = \max(\phi_1(x), -\phi_2(x)), \\
\mathcal{G}_3 = \mathcal{G}_1^{\complement} &\iff \phi_3(x) = -\phi_1(x).
\end{aligned}
$$

It should be noted that the operations intersection, union and difference do not necessarily
produce signed distance functions even if both of the input shapes are described by signed
distance functions. That said, the outputs of these operations in this case are still implicit
surface functions and, because they retain a gradient of unit magnitude, they are generally
very well behaved numerically.

**`data = shapeIntersection(shape1, shape2)`:** Given implicit surface functions `shape1` and `shape2` (which must be arrays of the same size), returns the implicit surface function for the intersection of the two shapes. If both implicit surface functions are signed distance, then the output function will be signed distance within the intersection, but may not be outside of it.

**`data = shapeUnion(shape1, shape2)`:** Given implicit surface functions `shape1` and `shape2` (which must be arrays of the same size), returns the implicit surface function for the union of the two shapes. If both implicit surface functions are signed distance, then the output function will be signed distance outside of the union, but may not be inside of it.

**`data = shapeDifference(shape1, shape2)`:** Given implicit surface functions `shape1` and `shape2` (which must be arrays of the same size) describing sets $\mathcal{G}_1$ and $\mathcal{G}_2$, returns the implicit surface function for $\mathcal{G}_1 \setminus \mathcal{G}_2 = \mathcal{G}_1 \cap \mathcal{G}_2^{\complement}$. If both implicit surface functions are signed distance, then the output function will be signed distance within the resulting difference, but may not be outside of it.

**`data = shapeComplement(shape1)`:** Given an implicit surface function `shape1`, returns the implicit surface function for its complement. Unlike the binary set operations, with complement if the implicit surface function is signed distance, then the output function will be signed distance.

## 3.4   Spatial Derivative Approximations

This section discusses functions found in the directory `Kernel/SpatialDerivative`.

Level set equations, and more generally HJ PDEs, are first order hyperbolic PDEs related to conservation laws; consequently, care must be taken when computing derivatives in order to keep the numerical solution stable. In particular, certain types of terms—notably those involving the gradient or the surface normal—must either use upwinding or introduce artificial diffusion in order to maintain stability. Derivative approximations for the former case are dealt with in section 3.4.1.

If the HJ PDE contains sufficient diffusion, arising either naturally from second order terms or artificially from methods like Lax-Friedrichs (see section 3.6), then either upwind or centered approximations can be safely employed. Section 3.4.2 treats centered approximations for both first and second order differential terms, including mean curvature.

### 3.4.1 Upwind Approximations of the First Derivative

This section discusses functions found in the directory `Kernel/SpatialDerivative/UpwindFirst`.

The first derivative (in the form of $\nabla\phi(x,t)$) appears in the terms (5)–(9) of the HJ PDE. The last of these terms, curvature dependent motion, includes the dissipative mean curvature $\kappa(x)$, and hence centered differences can be used for the gradient in this case. In the remaining cases—motion by constant velocity, motion in the normal direction, reinitialization, and the general HJ—either upwinded approximations or artificial dissipation must be used in order to maintain stability. If the upwind direction can be determined, upwind approximations will generally yield more accurate results than artificial diffusion.

To take advantage of these cases, a large number of upwind approximations have been developed for the first derivative. This package includes four approximations that operate on each dimension separately (which consequently requires that the upwind direction must be determined for each dimension separately). For each dimension, the *left* approximation is used for flow that comes from nodes with lower index, and the *right* approximation for flow that comes from nodes with higher index. Note that higher order approximations may include nodes on both sides in their stencil. The four approximations provide a range of order of accuracy.

upwindFirstFirst: The basic first order approximation. For dimension $d$, the left $D_d^-\phi(x_i)$ and right $D_d^+\phi(x_i)$ approximations at node $i$ are

$$D_d^-\phi(x_i) = \frac{\phi(x_i) - \phi(x_{i-1})}{\Delta x_d},$$
$$D_d^+\phi(x_i) = \frac{\phi(x_{i+1}) - \phi(x_i)}{\Delta x_d}.$$

These are the $D_{i-1/2}^1$ and $D_{i+1/2}^1$ entries respectively of the first divided differences of $\phi$ in dimension $d$. For more details, see [24, chapter 3.2].

upwindFirstENO2: A second order approximation. The second order correction to the first order approximation is the neighboring entry in the second divided differences of $\phi$ with minimum modulus. In other words, there are two possible second order approximations to both the left and right, and this scheme chooses the least oscillatory of those two. Mathematically, it is equivalent to including up to the $Q_2'(x_i)$ term (3.22) in the derivative approximation (3.18) from [24, chapter 3.3].

upwindFirstENO3: A third order *Essentially Non-Oscillatory* (ENO) approximation. There are three possible third order approximations to both the left and right, and this scheme chooses the least oscillatory among them. Mathematically, it is equivalent to including up to the $Q_3'(x_i)$ term (3.24) in the derivative approximation (3.18) from [24, chapter 3.3].

upwindFirstWENO5: A fifth order *Weighted Essentially Non-Oscillatory* (WENO) approximation. This approximation blends together the three third order approximations from the ENO3 scheme so that in regions where $\phi$ is smooth, a fifth order approximation is achieved. In regions where $\phi$ is not smooth, WENO5 effectively becomes ENO3. For more details, see [24, chapter 3.4].

All four approximation functions use the same call structure, which we demonstrate with upwindFirstENO3.

[ derivL, derivR ] = upwindFirstENO3(grid, data, dim, generateAll): Constructs left and right upwind approximations to the first derivative in dimension dim of the function stored in array data, which exists on grid grid. The approximations are returned in the arrays derivL and derivR respectively, which are the same size as data. The approximations are determined by first constructing three third order approximations in each direction, and then choosing the least oscillatory based on the magnitude of entries in the second and third divided differences of $\phi$. The optional boolean parameter generateAll is primarily used for debugging purposes, and can generally be left at its default value generateAll == 0. If generateAll == 1, then derivL and derivR will be cell vectors of length three, where each cell contains one of the three third order approximations in the appropriate direction (no attempt is made to pick out the least oscillatory approximation in this case).

In addition to the functions listed above, a number of helper functions appear in this directory.

upwindFirstENO3a: Constructs the third order approximations using a divided difference table, which is more efficient than directly applying equations (3.25)–(3.27) from [24, chapter 3.4], although it is somewhat more complicated to code. This function has the same calling sequence as upwindFirstENO3 (in fact, the latter function is just a wrapper for this function).

upwindFirstENO3b: Constructs the ENO3 approximations using equations (3.25)–(3.27) from [24, chapter 3.4]. The least oscillatory approximation is chosen by evaluating the smoothness estimates (3.32)–(3.34) and picking (for each node) the third order approximation corresponding to the smallest smoothness estimate. This function has the same calling sequence as upwindFirstENO3. The algorithm is less efficient than a divided difference table; in particular, it requires that the left and right approximations are independently computed even though they share many of the same terms. However, the code is somewhat easier to understand. The resulting derivative approximation should be equivalent to that produced by upwindFirstENO3a.

**upwindFirstWENO5a:** Constructs the WENO5 approximations using a divided difference table, which is more efficient that directly applying equations (3.25)–(3.41) from [24, chapter 3.4], although somewhat more difficult to code. The smoothness estimates are constructed from the first divided differences. Several choices of $\epsilon$ terms (including one corresponding to (3.38), which is unfortunately rather slow to evaluate) are available by modifying parameters in the file. This function has the same calling sequence as `upwindFirstWENO5` (in fact, the latter function is just a wrapper for this function).

**upwindFirstWENO5b:** Constructs the WENO5 approximations using equations (3.25)–(3.41) from [24, chapter 3.4]. This function has the same calling sequence as `upwindFirstWENO5`. The algorithm is slightly less efficient than a divided difference table, although the speed difference between the two WENO5 schemes is less pronounced than the difference between the two ENO3 schemes. Once again, the code is somewhat easier to understand. The resulting derivative approximation should be equivalent to that produced by `upwindFirstWENO5a`.

**upwindFirstENO3aHelper:** A helper routine that constructs the divided difference table and the third order approximations. It is used by `upwindFirstENO3a` and `upwindFirstWENO5a`.

**upwindFirstENO3bHelper:** A helper routine that constructs the third order approximations according to (3.25)–(3.27), the smoothness estimates according to (3.32)–(3.34) and the $\epsilon$ term (3.38), all from [24, chapter 3.4]. It is used by `upwindFirstENO3b` and `upwindFirstWENO5b`.

**checkEquivalentApprox:** A helper routine that checks whether two derivative approximations are equivalent to within some relative and absolute error bounds. Since the ENO and WENO schemes involve so many different approximations to the first derivative, it should come as no surprise that some of them should be equivalent, in the sense that they include the same terms from the divided difference table. A debugging option that can be set inside the files of these approximation functions will automatically check whether these approximations are actually equivalent. Normally, this check will not be performed.

For a discussion of the relative accuracy and speed of the various approximation schemes, see section 2.11.2.

### 3.4.2 Other Approximations of Derivatives

This section discusses functions found in the directory `Kernel/SpatialDerivative/Other`.

Many of the terms in HJ PDEs require upwind first order derivatives, and it is these terms that cause many of the practical difficulties in numerical solutions. Because there are so many

viable options for approximating these derivatives, the previous section outlined a collection of interchangable routines implementing some of these options.

In contrast, the toolbox at present offers few options for the remaining types of derivative terms. Each of the functions is specialized to a particular type of term, and hence we examine each separately. The first two have corresponding term approximations in section 3.6.

[ second, first ] = hessianSecond(grid, data): Constructs a second order accurate approximation to the mixed second order partial derivative matrix $D_x^2 \phi(x)$ of $\phi(x) =$ data:

$$
D_x^2 \phi(x) = \begin{bmatrix} \frac{\partial^2 \phi(x)}{\partial x_1^2} & \frac{\partial^2 \phi(x)}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 \phi(x)}{\partial x_1 \partial x_d} \\ \frac{\partial^2 \phi(x)}{\partial x_2 \partial x_1} & \frac{\partial^2 \phi(x)}{\partial x_2^2} & \cdots & \frac{\partial^2 \phi(x)}{\partial x_2 \partial x_d} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 \phi(x)}{\partial x_d \partial x_1} & \frac{\partial^2 \phi(x)}{\partial x_d \partial x_2} & \cdots & \frac{\partial^2 \phi(x)}{\partial x_d^2} \end{bmatrix},
$$

where $d$ is the dimension of the grid. Note that $D_x^2 \phi(x)$ depends on $x$, so ideally this function would return a $d \times d$ matrix each of whose entries was an array the size of data. Since that result is challenging to encode in MATLAB, we instead return a $d \times d$ cell matrix, each element of which is an array the size of data containing a second order mixed partial approximation for the entire grid. In other words,

$$
\texttt{second\{i, j\}} = \frac{\partial^2 \phi(x)}{\partial x_i \partial x_j}
$$

over all nodes $x$ in the grid. Because $D_x^2 \phi(x)$ is symmetric, only its lower left half is computed and returned. Therefore, even though second{i,j} = [] for i < j, the appropriate approximation can be found in second{j,i}. Since a centered approximation of the gradient is computed while finding $D_x^2 \phi(x)$, it is optionally returned in the cell vector first. Note that this centered approximation should not be used in place of an upwind approximation for advective terms.

[ curvature, gradMag ] = curvatureSecond(grid, data): Constructs a second order accurate approximation to the mean curvature of the isosurfaces of the function $\phi(x) =$ data:

$$
\kappa(x) = \nabla \cdot \left( \frac{\nabla \phi(x)}{\|\nabla \phi(x)\|} \right),
$$

$$
= \frac{\sum_{i=1}^d \frac{\partial^2 \phi(x)}{\partial x_i^2} \sum_{j \neq i} \left( \frac{\partial \phi(x)}{\partial x_i} \right)^2 - 2 \sum_{j < i} \frac{\partial \phi(x)}{\partial x_i} \frac{\partial \phi(x)}{\partial x_j} \frac{\partial^2 \phi(x)}{\partial x_i \partial x_j}}{\|\nabla \phi(x)\|^3},
$$

where $d$ is the dimension of the grid. The output curvature is an array the same size as data. For more details, see [24, chapter 1.4] or [28, chapter 6.7]. Since an approximation

of the gradient magnitude (using centered first order differences) is computed while finding $\kappa$, it is optionally returned in the array `gradMag`, which is also the same size as `data`. Note that this centered difference approximation of $\|\nabla\phi\|$ should not be used in place of an upwind approximation for motion in the normal direction.

The remaining two derivative approximations do not yet have corresponding term approximations in section 3.6, primarily because we have not yet found a practical use for them. They are provided primarily to demonstrate how additional derivative approximations can be constructed. The main challenge in constructing corresponding term approximation functions is determining the appropriate CFL condition—consider it an exercise left to the reader.

`laplacian = laplacianSecond(grid, data)`: Constructs a second order accurate approximation to the Laplacian of the function $\phi(x) =$ `data`:

$$\Delta\phi(x) = \nabla \cdot \nabla\phi(x),$$
$$= \sum_{i=1}^{d} \frac{\partial^2 \phi(x)}{\partial x_i^2},$$

where $d$ is the dimension of the grid. The output `laplacian` is an array the same size as `data`. In theory, if $\phi$ is a signed distance function $\|\nabla\phi\| = 1$, the Laplacian can be substituted for the mean curvature $\kappa$, and it is much quicker to calculate. However, since most $\phi$ are only approximately signed distance functions, this substitution is not recommended.

`deriv = centeredFirstSecond(grid, data, dim)`: Constructs a centered second order accurate approximation to the first derivative in dimension `dim` of the function $\phi(x) =$ `data`. The output `deriv` is the same size as `data`. Repeated calls with different `dim` can be used to construct an approximation of the gradient; however, since the approximation is centered, it should not be used in place of upwind approximations for advective or similar terms in the HJ PDE.

Other derivative approximations that might prove useful but are not yet coded include the Gaussian curvature [28, chapter 6.7] and the second derivative of curvature (so a fourth order derivative) [28, chapter 14.6].

## 3.5 Time Derivative Approximations

This section discusses functions found in the directory `Kernel/ExplicitIntegration/Integrators`.

The time derivative (4) is treated by the method of lines. We assume that approximations for all the other terms (5)–(12) can be collapsed into a single function $G(x, \phi(x, t))$, and then solve the ODE $D_t\phi(x, t) = G(x, \phi(x, t))$ pointwise at each state $x$. Note that $G$ will have the opposite sign of the terms (5)–(12) because it has been moved onto the opposite side of the equation. Furthermore, $G$ is usually nonlinear, so we will use explicit *Runge-Kutta* (RK) integrators that can determine $\phi(x, t + \Delta t)$ knowing only $\phi(x, t)$ and $G(x, \phi(x, t))$.

The downside of using explicit solvers is that we will need to choose our timestep $\Delta t$ small enough to satisfy a *Courant-Friedrichs-Lewy* (CFL) condition. In practical terms, this means that the timestep will be related to the grid resolution: $\Delta t$ proportional to $(\Delta x)^2$ if there are second order derivative terms (9) or (10), otherwise $\Delta t$ proportional to $\Delta x$. The particular CFL timestep restriction is generated by the term approximations described in section 3.6; the RK integrator routines described below merely enforce these restrictions.

Even if CFL timestep conditions are met, the time integrator must still be chosen carefully in order to guarantee stability. Consequently, we have chosen to use a collection of *Total Variation Diminishing* (TVD) RK schemes proposed in [30] and described in [24, chapter 3.5]. Note that these schemes are only TVD if the underlying spatial approximation is likewise TVD; consequently, they provide no theoretical guarantees when used with ENO and WENO spatial approximations. In practice they seem to work well with all the approximations described in section 3.6.

### 3.5.1  Explicit Integration Routines

The basic first order explicit TVD RK scheme is simply forward Euler

$$\phi(x, t + \Delta t) = \phi(x, t) + \Delta t G(x, \phi(x, t)).$$

The call parameters look very similar to MATLAB's basic ODE suite integrators, such as `ode23` and `ode45`.

[ t, y, schemeData ] = odeCFL1(schemeFunc, tspan, y0, options, schemeData): Integrates the ODE $D_t y = G(t, y)$ from time `tspan(0)` to time `tspan(end)` using a CFL timestep constrained forward Euler integrator that is first order accurate. The function handle `schemeFunc` describes $G(t, y)$, while the initial conditions are provided by `y0`. Integration options—set by a call to `odeCFLset`—are passed in `options`. Parameters for the underlying ODE can be passed in `schemeData`. All arguments are mandatory, but the last two may be replaced with `[]` if they are not needed.

In most circumstances, the `schemeData` parameter will not be modified and therefore its returned value can be ignored. It can be modified either through the `PostTimestep` option

discussed in section 3.5.3, or by the term approximation functions that satisfy the schemeFunc prototype. The prototype for the function handle schemeFunc matches the approximation functions given in section 3.6.

[ ydot, stepBound, schemeData ] = schemeFunc(t, y, schemeData): Calculate ydot = $G(t, y)$, using the parameters provided in schemeData. Note that y is passed as a column vector and ydot should be returned as a column vector. The return scalar stepBound provides the maximum CFL timestep permitted (use stepBound = inf if there is no CFL restriction). The return structure schemeData is usually identical to the input parameter of the same name, but may be a modified version.

Two higher order accurate integrators are also provided, with the same call structure as odeCFL1.

odeCFL2: Second order accurate TVD RK integrator, also known as the midpoint or modified Euler method. It computes two forward Euler steps and hence about twice as much work as odeCFL1.

odeCFL3: Third order accurate TVD RK integrator. It requires three forward Euler steps and hence about three times as much work as odeCFL1.

In the discussion below, we refer to these three integrators interchangably as odeCFLn. TVD RK integrators of fourth and higher order accuracy have been described in the literature, but we have not yet implemented them.

### 3.5.2 Explicit Integrator Quirks

These integrators were designed to be very similar to MATLAB's so as to reduce the learning curve of users and in hopes of leveraging code compatibility in future extensions. However, implementing such compatibility requires the introduction of several nonintuitive quirks to the code.

- In the rest of the toolbox's routines, the implicit surface function $\phi$ is passed in an array data of size grid.shape. When using the method of lines to convert the PDE into an ODE, the value of the implicit surface function at each node becomes the ODE's "state." Since MATLAB's ODE integration routines assume that the current state of the ODE is stored in a column vector y, we must reshape the data array into a column vector of length prod(grid.shape) before passing it to odeCFLn, and the spatial approximation function schemeFunc must both reshape y into data before manipulating it

and return its result in a column vector `ydot`. These shape alterations are accomplished by commands such as `y = data(:)` and `data = reshape(y, grid.shape)` and are essentially free if the underlying data is not subsequently altered; for example, if the right hand side variable is not further modified in the current function. These alterations are performed in all the examples and the term approximations from section 3.6.

- The $G(t, y)$ function appears on the opposite side of the equality as compared to the terms from (5)–(12) that it contains. Consequently, these terms must be negated before inclusion in $G$. This negation is performed in all the term approximations from section 3.6.

- Like MATLAB's ODE integration routines, the `odeCFLn` routines adjust the timestep during integration; however, the method for determining the timestep is completely different. MATLAB's routines adjust the timestep to achieve a given level of local truncation error, as measured by comparing two schemes with different orders of accuracy. In contrast, the `odeCFLn` routines adjust the timestep solely to satisfy a CFL stability restriction, and they never examine the local truncation error. From an ODE error analysis point of view, they behave like fixed timestep integrators. The need for a CFL restriction is the practical source of the requirement that at least one of the terms with a spatial derivative (5)–(10) must be part of $G(t, y)$.

- The state vector `y = data(:)` for a discretized PDE can easily contain millions of elements (one for each node in the grid). Storing versions of this state vector for each of dozens of timesteps in a typical call to an integration routine would quickly fill up memory. Consequently, the contents of return parameters `t` and `y` of `odeCFLn` is determined from `tspan` in a different way than in MATLAB's ODE suite. If `tspan = [ t0, tf ]` contains only two elements, then `y` is a column vector of the state at `t = tf`. If `tspan` contains more than two elements, then `t` is the column vector form of `tspan` and each row of `y` contains the state at the time of the corresponding row in `t`. In both cases, the value of the state at intermediate timesteps is discarded. For discretized PDEs, we recommend use of the first option, since it also avoids making a copy of the initial conditions `y0`.

### 3.5.3   Integrator Options

There are several algorithmic options for `odeCFLn`, which are manipulated using `odeCFLset` and `odeCFLget` in the same manner as MATLAB's `odeset` and `odeget` routines; however, note that the available options are different.

`options = odeCFLset('name1', value1, 'name2', value2, ...)` or `options = odeCFLset(oldopts, 'name1', value1, ...)`: Set options for one of the `odeCFLn` integration routines. The parameters `oldopts` and `options` are option structures. Call `odeCFLset` with no input or output parameters to see the list of available options and their defaults.

`value = odeCFLget(options, 'name')`: Retrieve the value of an option parameter from an option structure created by `odeCFLset`. The parameter `options` is the option structure. The value of the parameter corresponding to the string `'name'` is returned in parameter `value`. Call `odeCFLget` with no input or output parameters to see the list of available options and their defaults.

The currently available options are:

`FactorCFL`: positive scalar, default value 0.5, normally between 0 and 1 exclusive. The actual timestep taken by `odeCFLn` will be `FactorCFL * stepBound`, where `stepBound` is the CFL timestep restriction returned by `schemeFunc`. The default is safe, while a choice of 0.9 would be considered aggressive.

`MaxStep`: positive scalar, default value `realmax`. Upper limit on the size of the timestep taken by `odeCFLn`. Useful to enforce a fixed timestep if `stepBound` is infinite (such as if `schemeFunc` contains no spatial derivative terms).

`PostTimestep`: A function handle to a function with prototype

$$[\mathrm{yOut, schemeDataOut}] = \mathrm{postTimestepFunc(t, yIn, schemeDataIn)}.$$

The default `[]` indicates that no such routine should be called. If present, the post-timestep function is called by `odeCFLn` after every full timestep. By modifying `y`, this function can be used to implement constraints of the form (14). By modifying `schemeData`, this function can record information about the evolution of `y`, or modify parameters for the term approximation routine `schemeFunc` on the fly.

This option may also be a cell vector of `postTimestepFunc` function handles. In this case, each function handle element is called after each timestep. The functions are called in the same order as the cell vector.

`SingleStep`: `'on'` or `'off'`, default value `'off'`. If this option is set to `'on'`, then the integrator will return after a single CFL constrained timestep regardless of whether the final time in `tspan` has been reached or not. In this case, the return parameter `t` will be set to the actual time reached after that single timestep. Useful for debugging or if the calling routine wants to examine the state vector after every timestep; for example, see `signedDistanceIterative` in section 3.7.4.

`Stats`: `'on'` or `'off'`, default value `'off'`. If this option is set to `'on'`, then a few statistics on the integration are displayed on the screen (number of timesteps, CPU time). Useful for debugging.

`TerminalEvent`: A function handle to a function with prototype

$$[\mathrm{value, schemeDataOut}] = \mathrm{terminalEventFunc(t, y, tOld, yOld, schemeDataIn)}.$$

107

The default [] indicates that no such routine should be called. If present, the terminal event function is called by `odeCFLn` after every full timestep. If both post-timestep and terminal event functions are present, all post-timestep functions are called before the terminal event function. The input parameters include the time and data from the present timestep (`t` and `y`) as well as the time and data from the last timestep (`tOld` and `yOld`). The return parameter `value` should be a double scalar, vector or matrix, and must not change size or shape from one timestep to the next. If any element of `value` changes sign between timesteps, then integration is terminated and `odeCFLn` returns immediately. Unlike MATLAB's ODE event system, no attempt is made to accurately locate the time at which that element of `value` passed through zero. Note that integration cannot be stopped until after the second timestep has been taken. It cannot be stopped before or after the first timestep because there is no established sign of the elements of `value` from which a sign change can be detected. **This routine has not been well tested.**

### 3.5.4  Integrator Helper Functions

A number of helper routines appear in this directory that are not intended for direct calls from the user.

`odeCFLcallPostTimestep`: Calls one or more routines registered with the `PostTimestep` option of `odeCFLset`. Since this code was common to all `odeCFLn` routines, it was factored into a separate routine.

`odeCFLmultipleSteps`: Handles the case when `odeCFLn` is called with a `tspan` vector of length more than two (as detailed in section 3.5.2). In this case, `odeCFLmultipleSteps` is called by `odeCFLn`, and then `odeCFLmultipleSteps` makes repeated calls back to `odeCFLn` to collect the solution at the requested times. Since this code was common to all `odeCFLn` routines, it was factored into a separate routine. Normally, using a `tspan` of length more than two is not efficient, so this code is not run. **This routine has not been well tested.**

## 3.6  Approximating the Terms in HJ PDEs

This section discusses functions found in the directories `Kernel/ExplicitIntegration/Term`.

From the perspective of a typical user, it is the routines for approximating the spatial terms (5)–(14) in the HJ PDE that are most interesting among the many routines in this

toolbox, in the sense that it is through these terms that the user controls the motion of the implicit surface. In particular, the user must carefully chose which terms to include, and what parameters to provide to those terms.

All the term approximation functions follow the calling convention established by the integrator functions **odeCFLn** so that these term approximations can be passed as the **schemeFunc** parameter to **odeCFLn**. As an example, consider convective motion by a velocity field (5).

[ ydot, stepBound, schemeData ] = termConvection(t, y, schemeData): Computes an approximation of $G(t, \phi(x,t)) = -v(x) \cdot \nabla\phi(x,t)$, where (the reshaped) $\phi(x,t)$ is contained in the column vector **y** and $G(t, \phi(x,t))$ is reshaped and returned in the column vector **ydot**. The velocity field $v(x)$ is specified as a component of the structure **schemeData**. The maximum CFL timestep is returned in **stepBound**. The return structure **schemeData** is usually identical to the input parameter of the same name, but may be a modified version. For more details, see section 3.6.1.

We divide the term approximation functions into groups and describe each in the sections below. The basic groups are approximations in which the first derivative appears in a specific form (5)–(7), general first derivative approximations (8), second derivative approximations (9)–(10), and others (11)–(14). Among the details discussed for each type of term are the particular parameters for that term (passed in the structure **schemeData**) and the CFL restriction imposed (returned in the scalar **stepBound**). Note that **schemeData** may contain additional fields beyond those discussed below, should the user desire.

Many of the term approximations require the user to provide function handles that will be called on each timestep to provide term parameters throughout the grid. Typically these functions are called once per timestep (or once per dimension per timestep) and return an array (or cell vector of arrays). For efficiency reasons, it is very important that these functions be vectorized in the MATLAB sense—they should not use loops to iterate through the data or derivative arrays. Examples of such vectorization can be found in section 2.

One particular type of function that is allowed by many routines to provide a time dependent scalar term parameter is the **scalarGridFunc** prototype.

$$a = \mathtt{scalarGridFunc}(\mathtt{t}, \mathtt{data}, \mathtt{schemeData})$$

The parameters of **scalarGridFunc** are identical to those of the term approximation routine which calls it, except that **data = y** has been reshaped to its original size. The return parameter **a** must be a scalar or an array of size **grid.shape**, which represents some kind of scalar value for each node in the grid—for example, **termNormal** uses **a** as the speed of motion normal to the front. The user can pass additional information to the function implementing **scalarGridFunc** by including additional fields in the **schemeData** structure.

When designing functions to the `scalarGridFunc` prototype, users should note that the scalar returned is often allowed to depend on the time $t$ and state $x$, but not explicitly on the function $\phi$; for example, `termNormal`, `termCurvature`, `termTraceHessian`, and `termDiscount`. Failure to comply with such restrictions may cause the underlying PDE to become ill-posed, the solution to become discontinuous, and/or the discretization to become unstable. Consequently, in these cases it is recommended that the function satisfying the `scalarGridFunc` prototype not examine the `data` parameter. There are cases, such as `termForcing`, in which dependence on $\phi$ is permitted and the return value may therefore depend on `data`; however, in these cases the user must ensure satisfaction of the monotonicity requirement (3).

### 3.6.1 Specific Forms of First Derivative

This section discusses functions to approximate the terms which implement convection by a velocity field (5), motion in the normal direction (6), and the reinitialization equation (7). The functions are `termConvection`, `termNormal` and `termReinit` in the directory `Kernel/ExplicitIntegration/Te`. These terms are grouped together because they share a number of common features.

Notice that each of these terms could be restated in the form of (8), and hence approximated by the functions discussed in section 3.6.2. Unfortunately, those approximations involve adding artificial dissipation in order to achieve numerical stability. For these specific terms, it is always possible to determine the upwind direction and construct a relatively dissipation free, and hence more accurate, approximation. Because these terms appear so often in practice, it is well worth the effort to build special purpose approximation routines for them.

In addition to the term specific fields discussed below, in every case the parameter structure `schemeData` contains the fields:

`schemeData.grid`: The grid on which the implicit surface function is defined.

`schemeData.derivFunc`: A function handle to a function with prototype

$$[\texttt{derivL}, \texttt{derivR}] = \texttt{derivFunc}(\texttt{grid}, \texttt{data}, \texttt{dim})$$

to compute upwind approximations of the first derivative. This function should generally be chosen from among those described in section 3.4.1. Note that this function must return both left and right approximations to the first derivative.

It turns out that for each of these terms, the approximation algorithm constructs an effective velocity field $v(x)$ and it is this velocity field which determines the CFL timestep constraint (by [24, equation (3.10)])

$$\texttt{stepBound} = \max_{x \in \texttt{grid}} \left( \sum_{i=1}^{\texttt{grid.dim}} \frac{|v_i(x)|}{\texttt{grid.dx}(i)} \right)^{-1}.$$

110

The important fact about this bound is that $\Delta t$ is proportional to $\Delta x$.

We now discuss each of the terms individually. More details can be found in the corresponding functions' help entries.

termConvection: Motion by an externally generated flow field (5), also called convection or advection. The user supplies the flow field $v : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n$ as the field schemeData.velocity in one of two ways.

- For time invariant flow fields $v(x)$, velocity may be a cell vector of length grid.dim, in which case velocity{i} = $v_i(x)$ is either a scalar (for constant velocity) or an array of size grid.shape (for spatially varying velocity) providing component $i$ of the velocity field.

- For general flow fields, velocity may be a function handle to a function with prototype
  $$v = \text{velocityFunc}(t, \text{data}, \text{schemeData}),$$
  where the output v is the cell vector described above and the input arguments are the same as those of termConvection (except that data = y has been reshaped to its original size). The velocityFunc prototype is very similar to the scalarGridFunc prototype, except that it returns a cell vector of arrays. In a similar way to scalarGridFunc, it may be useful to include additional fields in schemeData.

termNormal: Motion in the normal direction (6). The user supplies the speed of the interface $a : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}$ as the field schemeData.speed in one of two ways.

- For time invariant speeds $a(x)$, speed may be either a scalar (for constant speed) or an array of size grid.shape (for spatially varying speed).

- For general speed functions, speed may be a function handle to a function with the scalarGridFunc prototype. The result of evaluating this function at the current time and state will be treated as the scalar/array described above. In this case, it may be useful to include additional fields in schemeData.

termReinit: The reinitialization equation (7). In theory, solving this equation to convergence can turn an implicit surface function into a signed distance function without moving or explicit finding the interface [33]. In practice, it is usually used to smooth out excessively steep or shallow gradients in $\phi$. The user supplies a copy of the initial conditions $\phi(x, 0)$ (as an array of size grid.shape) in schemeData.initial. The upwinding scheme is taken from [12, appendix A.3].

There is one optional field in schemeData. By default, the first order accurate subcell fix from [27] is applied; for more details, see the discussion in section 2.10. The order of accuracy of the subcell fix can be specified by the optional integer schemeData.subcell_fix_order.

At present the only accepted values are 1 (the default) and 0 (in which case the subcell fix is disabled). If the subcell fix is disabled, then a smoothed version of the signum function is used:

$$\text{sign}(\xi) = \frac{\xi}{\sqrt{\xi^2 + \max_i\left[\texttt{grid.dx}(i)\right]^2}}$$

This term approximation routine `termReinit` will rarely be invoked directly, but will be used indirectly by other routines like `signedDistanceIterative` (see section 3.7.4).

### 3.6.2 Approximating General HJ Terms

This section discusses the function `Kernel/ExplicitIntegration/Term/termLaxFriedrichs` and functions found in `Kernel/ExplicitIntegration/Dissipation`.

Terms involving the first derivative in general form (8) are the most challenging to treat numerically, and hence require the most complex term approximation function `termLaxFriedrichs`. This function is based on the framework proposed in [26] and described in [24, chapter 5.3]. The basic idea is to replace the analytic $H(x, p)$ (where $p$ is a placeholder for $\nabla\phi$) with a numerical approximation

$$\hat{H}(x, p^+, p^-) = H\left(x, \frac{p^+ + p^-}{2}\right) - \alpha(x)^T\left[\frac{p^+ - p^-}{2}\right], \qquad (46)$$

where $p^+$ and $p^-$ are the right and left approximations of the gradient respectively. The first term of $\hat{H}$ is simply the analytic Hamiltonian evaluated with a centered approximation to the gradient. By itself, such an approximation will be numerically unstable, so the second term adds some dissipation. The final part of this second term (the difference between $p^+$ and $p^-$) looks like a Laplacian, and provides the stabilizing dissipation. In smooth regions of the solution, the left and right approximations will be similar and this term will be near zero. The scaling portion $\alpha(x)$ of this term depends on $D_p H(x, p)$, the partial derivative of $H$ with respect to the gradient $p$. As discussed below, there are several different choices of $\alpha$ function.

The `schemeData` structure for `termLaxFriedrichs` requires the following fields:

`schemeData.grid`: The grid on which the implicit surface function is defined.

`schemeData.derivFunc`: A function handle to compute upwind approximations of the first derivative, chosen from among those described in section 3.4.1. Note that this function must return both left and right approximations to the first derivative.

`schemeData.dissFunc`: A function handle to one of the dissipation routines with prototype

$$[\texttt{diss}, \texttt{stepBound}] = \texttt{dissFunc}(\texttt{t}, \texttt{data}, \texttt{derivL}, \texttt{derivR}, \texttt{schemeData}),$$

discussed below. Computes the artificial dissipation necessary to stabilize the Hamiltonian approximation calculated with a centered difference approximation of the gradient; in other words, the second term on the right hand side of (46), including the $\alpha(x)$ scaling (which is actually computed by a call to `schemeData.partialFunc` as described below).

`schemeData.hamFunc`: A function handle to a routine that computes the analytic $H(x, p)$. This function is user supplied, and is called directly by `termLaxFriedrichs`.

`schemeData.partialFunc`: A function handle to a routine that computes the extrema (in each dimension) of $D_p H(x, p)$. This function is user supplied and is called by `dissFunc`.

Typically the user will have a mathematical equation for `schemeData.hamFunc` and simply needs to convert it into (vectorized) MATLAB code. Writing `schemeData.partialFunc` is often more challenging. The function prototypes are:

`hamValue = hamFunc(t, data, deriv, schemeData)`: Compute the analytic Hamiltonian $H(x, \nabla\phi)$; in fact, the more general form $H(x, t, \phi, \nabla\phi)$ is allowed. The parameters are the current time `t` (a scalar), the current implicit surface function $\phi = $ `data` (in an array of size `grid.shape`), a cell vector $\nabla\phi = $ `deriv` of length `grid.dim` whose element $i$ is an array of size `grid.shape` containing the $i^{th}$ component of the gradient, and the `schemeData` structure that was passed to `termLaxFriedrichs`. The return value `hamValue` should be an array of size `grid.shape`.

`alpha = partialFunc(t, data, derivMin, derivMax, schemeData, dim)`: Estimate component `dim` of the $\alpha$ scaling term in (46).

$$\alpha_{\texttt{dim}}(x) = \max_{p \in [\texttt{derivMin}, \texttt{derivMax}]} \left| \frac{\partial H(x, p)}{\partial p_{\texttt{dim}}} \right|. \tag{47}$$

Note that $\alpha$ depends on $x$, and so should be evaluated separately at each state (preferably in a vectorized fashion). The gradient range parameters `derivMin` and `derivMax` are each cell vectors of length `grid.dim` whose element $i$ is either a scalar or an array of size `grid.shape`, depending on whether the range of component $i$ of the gradient is constant (for global Lax-Friedrichs) or state dependent (for other types of dissipation). Because the gradient range may depend on the dimension, this function is called once for each dimension `dim` from 1 to `grid.dim`.

In general, $\alpha$ need not be calculated exactly. Too little dissipation will usually lead to instability, but may be tolerable on the occasional timestep. Too much dissipation will

smooth what should be sharp corners in the implicit surface, but is otherwise safe. If the exact optimization in (47) is too complicated or expensive to evaluate, it is reasonable (although somewhat less accurate) to overestimate its value.

There are a number of options for `schemeData.dissFunc` provided by the toolbox. They all have the same prototype

[ diss, stepBound ] = artificialDissipationGLF(t, data, derivL, derivR, schemeData):
   Compute the artificial dissipation in (46). Parameters derivL $= p^-$ and derivR $= p^+$ are the gradient approximations returned by a call to `schemeData.derivFunc`. The returned diss is an array of size `grid.data` containing the appropriate dissipation for each node in the grid. The scalar CFL timestep constraint stepBound is also calculated in the dissipation function.

Apart from calculating the difference between the left and right approximations of the gradient, the dissipation routines' main task is to determine the range of gradient `derivMin` to `derivMax` pass on to `schemeFunc.partialFunc`. The method of calculating this range differs between the dissipation function options, following the framework laid out in [24, 26].

artificialDissipationGLF: *Global Lax-Friedrichs* (GLF) dissipation. Calculate a single range of gradient over the entire grid, as proposed in the original numerical scheme for finding the viscosity solution of an HJ PDE [7]. Because this choice generates the largest range of possible gradients, it will also generate the most dissipation.

artificialDissipationLLF: *Local Lax-Friedrichs* (LLF) dissipation. When considering component $\alpha_i(x)$ of the dissipation scaling $\alpha(x)$, restrict the range of component $i$ of the gradient to the range between left and right approximations of that component at each node individually. The range of the remaining components of the gradient is calculated globally, as with GLF. This restriction is more costly to compute, but can be considerably less dissipative for Hamiltonians that are very close to convective flow.

artificialDissipationLLLF: *Local Local Lax-Friedrichs* (LLLF) dissipation. The range of every component of the gradient is simply the range between left and right approximations of that component at each node individually. This choice leads to the least dissipation and is less expensive to compute than LLF (since the same range is used for every dimension). It is equivalent to LLF if the Hamiltonian is separable

$$H(x, p) = \sum_{i=1}^{\texttt{grid.dim}} H_i(x, p_i).$$

Unfortunately, in those cases where it is not equivalent to LLF, it can be unstable and/or nonmonotonic. Consequently, any approximation it produces may not converge to the true viscosity solution as the grid is refined.

Regardless of which dissipation function is chosen, the user supplied `schemeFunc.partialFunc` will be called `grid.dim` times to compute the components of $\alpha(x)$. Furthermore, even if $H$ is independent of $x$ and GLF is used (so that $\alpha(x)$ is independent of state), the actually dissipation may be state dependent if the left and right approximations of the gradient vary across the grid.

In addition to scaling the dissipation, $\alpha(x)$ is also the effective velocity and is therefore used to compute the CFL timestep restriction.

$$\mathtt{stepBound} = \max_{x \in \mathtt{grid}} \left( \sum_{i=1}^{\mathtt{grid.dim}} \frac{|\alpha_i(x)|}{\mathtt{grid.dx}(i)} \right)^{-1}.$$

Once again, $\Delta t$ is proportional to $\Delta x$. Its effect on the choice of CFL restriction reemphasizes the fact that overapproximating $\alpha$ is safe (although it will lead to smaller timesteps) but regular underapproximation may lead to instability.

Two other approximation schemes for arbitrary Hamiltonians are described in [24, 26]: *Roe with entropy fix* (RF) and *Godunov*. The former uses upwinding when an upwind direction can be determined and some form of Lax-Friedrichs otherwise; thus it will introduce even less dissipation that the LF schemes discussed above. The latter is less dissipative still, but requires solution of a potentially nonconvex optimization at each node. It seems likely that RF could be implemented in the current toolbox framework for general Hamiltonians, but the same is not true for Godunov; however, the approximation schemes in section 3.6.1 are examples of Godunov solvers for specific types of spatial terms.

### 3.6.3 Second Derivatives

This section discusses the functions `termCurvature` and `termTraceHessian` in the directory `Kernel/ExplicitIntegration/Term/`.

The routines for handling terms of the forms (9)–(10) both involve approximations of the second derivative, and both place a stringent bound on the size of explicit timesteps: $\Delta t$ is proportional to $\Delta x^2$. Their `schemeData` structures both require the `schemeData.grid` field, but are otherwise different.

`termCurvature`: Motion by mean curvature (9). The field `schemeData.curvatureFunc` must contain a function handle for a routine that approximates the curvature $\kappa$ (and gradient magnitude $\|\nabla\phi\|$); at present the only such routine in the toolbox is `curvatureSecond` (see section 3.4.2). The user supplies the multiplier $b : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}$ as the field `schemeData.b` in one of two ways.

- For time invariant multipliers $b(x)$, $\mathtt{b}$ may be either a scalar (for a constant multiplier) or an array of size $\mathtt{grid.shape}$ (for a spatially varying multiplier).

- For general multiplier functions, $\mathtt{b}$ may be a function handle to a function with the $\mathtt{scalarGridFunc}$ prototype. The result of evaluating this function at the current time and state will be treated as the scalar/array described above. In this case, it may be useful to include additional fields in $\mathtt{schemeData}$.

Following [24, equation (4.7)], the bound on the timestep is calculated as

$$\mathtt{stepBound} = \left[ \max_{x \in \mathtt{grid}} \left( 2b(x) \sum_{i=1}^{\mathtt{grid.dim}} \frac{1}{\mathtt{grid.dx}(i)^2} \right) \right]^{-1}.$$

$\mathtt{termTraceHessian}$: Motion by the trace of the Hessian (10). The field $\mathtt{schemeData.hessianFunc}$ must contain a function handle for a routine that computes a cell matrix approximation of the Hessian matrix of second derivatives; at present the only such routine in the toolbox is $\mathtt{hessianSecond}$ (see section 3.4.2). The user supplies the multipliers $\mathbf{L}(x,t)$ and $\mathbf{R}(x,t)$ as the fields $\mathtt{schemeData.L}$ and $\mathtt{schemeData.R}$ in one of three ways, where $n = \mathtt{grid.dim}$.

- For time and space invariant matrices, an $n \times n$ matrix. The same matrix will be used for every node of the grid.

- For time invariant but spatially dependent matrices, an $n \times n$ cell matrix, each element of which is an array of size $\mathtt{grid.shape}$. The function $\mathtt{cellMatrixMultiply}$ is used to apply such matrices to the Hessian.

- For general matrices, a function handle to a function with prototype

$$\mathtt{M = matrixGridFunc(t, data, schemeData)},$$

where the output $\mathtt{M}$ is a matrix or a cell matrix as described in the two options above and the input arguments are the same as those of $\mathtt{termTraceHessian}$ (except that $\mathtt{data = y}$ has been reshaped to its original size). In this case it may be useful to include additional fields in $\mathtt{schemeData}$.

The bound on the timestep is calculated as

$$\mathtt{stepBound} = \left[ \max_{x \in \mathtt{grid}} \left( 2 \left| \mathrm{trace}\left[ \mathbf{L}(x,t)\, \mathbf{D}\, \mathbf{R}(x,t) \right] \right| \right) \right]^{-1}, \qquad (48)$$

where the $n \times n$ matrix $\mathbf{D}$ is defined elementwise as $\mathbf{D}_{ij} = \mathtt{grid.dx}(i) * \mathtt{grid.dx}(j)$.

**This routine has not been well tested.** Among the shortcomings in the current version:

- The routine $\mathtt{termTraceHessian}$ has been validated only on one dimensional and faked two dimensional problems (one dimensional problems in a rotated coordinate frame).

- The CFL condition (48) has not been mathematically verified, although it has not yet lead to instability.

Bug reports are always appreciated.

It should be noted that despite their popularity for approximating second order derivatives in degenerate parabolic PDEs—such as $\kappa(x)$ in level set motion by mean curvature—centered difference approximations—such as `curvatureSecond`—are known to be non-monotonic in some cases. Consequently, algorithms based on centered differences may fail to be convergent [21]. Caveat emptor.

### 3.6.4 Other Spatial Approximation Terms

This section discusses the functions `termDiscount` and `termForcing` in the directory `Kernel/ExplicitIntegrat`

The routines for handling terms of the forms (11)–(12) are relatively straightforward. Note that these terms cannot appear alone as the `schemeFunc` passed to `odeCFLn`, since the resulting equation would not be a hyperbolic PDE. Rather, they must be combined with at least one spatial derivative term from sections 3.6.1–3.6.3 using the `termSum` routine described in section 3.6.5.

Since none of these terms involve a spatial derivative, they do not induce a CFL restriction on the timestep size. All of these routines therefore return `stepBound = +Inf`. From a practical point of view, failure to include a spatial derivative term in the (supposed) PDE will result in a timestep of size dictated by the `odeCFLn` option `maxStep` (see section 3.5 for details).

Both of these terms may depend on $\phi$, so care must be taken that they satisfy the monotonicity requirement (3). For `termDiscount`, this requirement translates to $\lambda \geq 0$. For the more general `termForcing`, the onus is on the user to ensure a proper function form.

termDiscount: Discounting or killing term (11). The user supplies the discount factor $\lambda : \mathbb{R}^n \times \mathbb{R} \to [0, +\infty]$ as the field `schemeData.discount` in one of two ways.

- For time invariant discounts $\lambda(x)$, `discount` may be either a scalar (for constant discount) or an array of size `grid.shape` (for spatially varying discount).
- For general discounts, `discount` may be a function handle to a function with the `scalarGridFunc` prototype. The result of evaluating this function at the current time and state will be treated as the scalar/array described above. In this case, it may be useful to include additional fields in `schemeData`.

termForcing: Forcing term (12). The user supplies the forcing function $F : \mathbb{R}^n \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ as the field `schemeData.forcing` in one of two ways.

- For time invariant forcing functions $F(x)$, forcing may be either a scalar (for constant forcing) or an array of size grid.shape (for spatially varying forcing).

- For general forcing functions, forcing may be a function handle to a function with the scalarGridFunc prototype. Unlike most such calls to functions with this prototype, it is reasonable to allow the return value to depend on $\phi$ by examining the data parameter; however, care must be taken in this case to ensure compliance with the monotonicity requirement (3). The result of evaluating this function at the current time and state will be treated as the scalar/array described above. In this case, it may be useful to include additional fields in schemeData.

### 3.6.5 Combining and Restricting Spatial Approximation Terms

This section discusses the functions termRestrictUpdate and termSum in the directory Kernel/ExplicitIntegration/Term/.

The term approximation schemes discussed thus far have all dealt with a single term from (5)–(12). In many applications these terms are combined together, or are restricted to a particular sign by constraints of the form (13). In this section we examine routines to treat these cases.

These routines conform to the term approximation prototype schemeFunc required by the odeCFLn integrators. However, they do not generate updates by themselves, but rather should be thought of as wrappers for update terms from the previous sections. Consequently, their schemeData structures will contain fields referring to other term approximation routines.

schemeData.innerFunc: A function handle (or cell vector of function handles) to a function which conforms to the schemeFunc prototype. Normally this will be a term approximation routine for a term of the form (5)–(12).

schemeData.innerData: A structure (or cell vector of structures) which is the schemeData structure required by the term approximation routine schemeData.innerFunc.

Within the routines below, a call of the form

$$\text{feval(schemeData.innerFunc}, t, y, \text{schemeData.innerData)}$$

will be issued to evaluate the wrapped term approximation routine (or an equivalent call for cell vector members).

termRestrictUpdate: Restrict the sign of a single spatial term, which can be used to implement (13). The spatial term is provided by the function handle schemeData.innerFunc,

and its associated data by the structure `schemeData.innerData`. The sign of the restriction is specified by the boolean `schemeData.positive`, which is true if the update must be greater than or equal to zero (defaults to true). The restriction is calculated independently for each node in the grid, and updates which violate the restriction are clipped to zero. The CFL timestep restriction calculated by `schemeData.innerFunc` is returned without modification, which may be conservative (if the update of the node which induced the timestep restriction has been clipped).

`termSum`: Combine multiple terms by summation. Each of the terms is specified by an entry in the cell vector of function handles `schemeData.innerFunc`, and its associated data by the corresponding entry in the cell vector of structures `schemeData.innerData`. Each term is evaluated independently, and the updates are summed at each node. The overall CFL timestep restriction $\mathtt{stepBound_{sum}}$ is computed from the individual term's timestep restrictions $\mathtt{stepBound}_i$ by:

$$\mathtt{stepBound_{sum}} = \left( \sum_i \frac{1}{\mathtt{stepBound}_i} \right)^{-1}.$$

Note that `termRestrictUpdate` and `termSum` can be used to wrap each other, and thereby accomplish HJ PDEs more complex than (4)–(13). They could even be used to wrap themselves, although we can think of little benefit to be gained from that design.

## 3.7 Helper Routines

This section describes functions in `Examples/Helper`, which are used for various auxiliary tasks.

### 3.7.1 Error Checking

This section describes functions in `Examples/Helper/ErrorCheck`, which are used to check the validity of function arguments.

`checkStructureFields(structure, 'field1', 'field2', ...)`: Checks that the first argument `structure` is a structure and, if so, checks that the subsequent arguments (which should all be strings) are the names of fields in that structure. Causes an error if either check fails. Often used in functions which access the `schemeData` structure.

### 3.7.2 Math

This section describes functions in `Examples/Helper/Math`, which are various types of mathematical operations.

Currently, the functions in this directory implement an extended form of some simple matrix operations. In several parts of the toolbox, it is necessary to represent spatially varying matrices $\mathbf{A}(x)$ or vectors $v(x)$—in fact, $x$ itself is a spatially varying vector. These objects are a challenge to represent, since the toolbox has already adopted the convention that MATLAB's array indices refer to nodes in the spatial grid. Adding more indices to account for the entries in the spatially varying matrix or vector would lead to a great deal of index confusion.

As an alternative, we have chosen to represent such matrices and vectors as cell arrays. A matrix $\mathbf{A}(x) \in \mathbb{R}^{p \times q}$, where $x \in \mathbb{R}^n$ is represented by a two dimensional cell array with $p$ rows and $q$ columns. Each element of this cell array is a regular MATLAB array of dimension $n$, containing elements for every node $x$ in the computational grid. We call this object a *cell matrix*. For example, the field `grid.xs` in the `grid` structure can be thought of as a $n \times 1$ cell matrix description of the vector $x$.

Several operations are provided for cell matrices:

$$\text{addition: } \mathbf{A}(x) + \mathbf{B}(x),$$
$$\text{multiplication: } \mathbf{A}(x)\mathbf{B}(x),$$
$$\text{elementwise maximization: } \max_x \mathbf{A}(x) \text{ or } \max_x |\mathbf{A}(x)|,$$
$$\text{elementwise trace: } \text{trace}\,\mathbf{A}(x).$$

All of the routines also accept a few special cases. If $\mathbf{A}(x) = \mathbf{A}$ is independent of state $x$, then the entries of the cell matrix can be scalars. If $\mathbf{A}(x) = a(x) \in \mathbb{R}$ is a state dependent scalar value, then the corresponding cell matrix should not be a cell object at all, but rather a regular array of the size appropriate for the computational grid. That array will be added to or multiplied by every entry of the cell matrix $\mathbf{B}(x)$, in a manner corresponding to the way that MATLAB treats scalars for regular matrices.

`C = cellMatrixAdd(A,B)`: Returns the spatially varying matrix $\mathbf{C}(x) = \mathbf{A}(x) + \mathbf{B}(x)$, represented as a cell matrix. If they are cell matrices, parameters `A` and `B` must be the same size and of dimension two, and this size is adopted by output `C`. The contents of each cell element of `A` and `B` must also be the same size, since they are added componentwise. Cell elements of `A` and/or `B` may be scalars. If `A` or `B` is a regular array, then `C` adopts the size of the other, and the one which is a regular array is treated as a state dependent scalar.

**maxA = cellMatrixMax(A, takeAbs):** Calculate the elementwise maximum over state space $x$ of spatially varying array $\mathbf{A}(x)$, represented as the cell matrix A. The maximum is returned in the regular matrix maxA, which has the same number of rows and columns as the cell matrix A. If the optional boolean parameter takeAbs is true, then the elementwise maximum $\max_x |\mathbf{A}(x)|$ is computed instead.

**C = cellMatrixMultiply(A,B):** Returns the spatially varying matrix $\mathbf{C}(x) = \mathbf{A}(x)\mathbf{B}(x)$, represented as a cell matrix. If $\mathbf{A}(x) \in \mathbb{R}^{m \times p}$ and $\mathbf{B}(x) \in \mathbb{R}^{p \times q}$, then $\mathbf{C}(x) \in \mathbb{R}^{m \times q}$. Therefore, if they are cell matrices, parameters A and B must be of dimension two, their inner dimensions must agree, and their outer dimensions dictate the size of output C. The contents of each cell element of A and B must either be arrays of the same size or scalars, since they are multiplied componentwise. If A or B is a regular array, then C adopts the size of the other, and the one which is a regular array is treated as a state dependent scalar.

**traceA = cellMatrixTrace(A):** Calculate the elementwise trace at each node for the spatially varying array $\mathbf{A}(x)$, represented as the cell matrix A. The trace is returned in the regular array traceA, which is the same size as each element of A (ie the size of the grid for $x$). Specifically, if $x \in \mathbb{R}$ the operation is

$$\texttt{traceA(i)} = \sum_k \texttt{A\{k,k\}(i)},$$

A similar operation applies in other dimensions. If A is a regular array, it is treated as a spatially varying scalar and hence traceA = A. To represent a spatially independent matrix $\mathbf{A}(x) = \mathbf{A}$, use a cell matrix where every entry is a scalar.

### 3.7.3 Post-Timestep Routines

The routines in this directory are examples of the postTimestepFunc protocol, and implement common tasks that are performed after each timestep of a level set calculation. Because they implement the same protocol, all routines have the same input and output parameters; however, in each case the schemeData structure will contain different fields.

The first routine implements the masking or constraint of $\phi$ (14).

**[ yOut, schemeDataOut ] = postTimestepMask(t, yIn, schemeDataIn):** Constrains the value of $\phi(x,t)$ after each timestep by applying a binary mask operation. The input argument t is ignored, while the input array yIn provides the value of $\phi(x,t)$. The structure schemeDataIn must contain the fields maskFunc and maskData. The output argument schemeDataOut = schemeDataIn (no change), while the modified data array yOut is calculated by

$$\texttt{yOut} = \texttt{feval(schemeDataIn.maskFunc, yIn, schemeDataIn.maskData)}.$$

A typical application of `postTimestepMask` would be to enforce the constraint $\phi(x, t) \geq \psi(x)$. This constraint can be implemented by choosing `maskFunc = @max` and `maskData` to be an array representing $\psi(x)$, reshaped into a column vector.

The second routine reinitializes an implicit surface level set function to be a signed distance function using the reinitialization PDE (7).

[ yOut, schemeDataOut ] = postTimestepReinit(t, yIn, schemeDataIn): Reinitializes an implicit surface function to be an approximate signed distance function. The input argument `t` is ignored, while the input array `yIn` contains the implicit surface function (in vector form). The structure `schemeDataIn` must contain the field `grid`. It may contain the fields `reinitAccuracy`, `reinitSteps` and/or `reinitErrorMax`, which control the accuracy and number of reinitialization iterations. If these fields are not supplied, they have defaults. The output argument `schemeDataOut = schemeDataIn` (no change), while `yOut` is the reinitialized level set function. **This routine has not been well tested.**

The `postTimestepReinit` routine is essentially a wrapper for `signedDistanceIterative` (see section 3.7.4). Note that this PDE algorithm for reinitialization does not explicitly find the interface, and hence may cause that interface to shift during iteration.

The third routine determines the time at which the zero level set crosses each node of the grid. Specifically, for each node $x$, it determines a timestep $\hat{t}$ such that $\phi(x, s) \geq 0$ for $s < \hat{t}$ and $\phi(x, s) < 0$ for $s \geq \hat{t}$. Subject to these constraints, a linear approximation of the exact $t$ such that $\phi(x, t) = 0$ is computed, where $\hat{t} - \Delta t < t \leq \hat{t}$. This routine is used for approximating the solution of stationary HJ [23] and degenerate elliptic PDEs; for example, see the time to reach problems in section 2.7.

[ yOut, schemeDataOut ] = postTimestepTTR(t, yIn, schemeDataIn): Compute the time at which the zero level set crosses each node in the grid. The implicit surface function is unmodified: `yOut = yIn`. The structure `schemeDataOut` contains all the fields of `schemeDataIn`, and the user need not add any special fields before calling this routine. On the first call to this function, the fields `ttr`, `ttrLastY` and `ttrLastT` are added to this structure—the user should ensure that fields of these names are not present in advance (their absence ensures proper initialization). In subsequent calls, these fields in `schemeDataIn` are examined, and then modified versions are returned in `schemeDataOut` (the user should not modify them separately). After each call, the `ttr` field contains the time at which the zero level set reached each node, as determined by the input parameter `t`. Nodes which have not yet be reached are given the value `inf`. In order to generate accurate initialization, it is recommended that the user make a direct call to this routine with the initial time and initial data before starting time integration with the toolbox.

### 3.7.4 Signed Distance Functions

This section describes functions in `Examples/Helper/SignedDistance`. Signed distance functions are a special case of implicit surface functions, and have several useful properties. From a numerical perspective, their gradient has magnitude one, which tends to reduce the error introduced by gradient approximations. From a geometric perspective, at every point in state space the function magnitude measures the distance to the surface and the gradient lies in the direction of the closest point on the surface. For these reasons it is often useful to construct a signed distance function. The routines in this directory are the start of a collection that will build an approximate signed distance function from a variety of initial data types.

`data = signedDistanceIterative(grid, data0, accuracy, tMax, errorMax)`: Turns an implicit surface function into a signed distance function by iterative solution of the reinitialization PDE (7). Both the implicit surface function and signed distance functions are defined on the same computational grid, the parameter `grid`. The implicit surface function is given by input array `data0`, and the signed distance result by output array `data`. The optional parameter `accuracy` has the usual options determining what order of accuracy of spatial and temporal derivative approximations should be used for the reinitialization PDE, and defaults to `'medium'` (second order accurate). Since the reinitialization wave front moves at approximately speed one outward from the zero level set, if the optional parameter `tMax > 0` then enough iterations are taken to move the reinitialization about a distance `tMax` from the front in both directions. The default is the entire distance across the grid (which will be excessive in many cases). If `tMax < 0`, then `-round(tMax)` is taken as the explicit number of CFL constrained timesteps to execute; the CFL factor for these timesteps is typically large ($\sim 0.95$) and can be determined by looking in the source. The optional parameter `errorMax` defines an update magnitude tolerance relative to the longest grid cell edge length `max(grid.dx)`—if the average node update drops below this tolerance on any iteration, the reinitialization is assumed to have converged and the iterations are terminated. The default value of `1e-3` is so tight that iterations rarely converge under the default. Note that the input implicit surface function must be relatively well behaved for this operation to succeed: the function gradient should not change sign or direction too drastically between neighboring nodes near the implicit surface. Even for well behaved implicit surface functions, this operation may shift the implicit surface location slightly. **This routine has not been well tested.**

`data = unsignedDistanceFromPoints(grid, points)`: Creates a function whose value at each grid node measures the distance from that grid node to the nearest of a collection of points. The grid is defined by parameter `grid` and each point is a row (with `grid.dim` columns) of the parameter `points`. The unsigned distance function is returned in array `data`. The unsigned distance function is *not* an implicit surface function. Searching for

the zero level set will prove futile, since all node values will be non-negative. In fact, this routine is simply the first step in turning a collection of surface points into an implicit surface function (for example, see [24, chapter 13]). Furthermore, this implementation uses the brute force, quadratic time pairwise algorithm. In future versions it should be replaced by a much quicker fast marching algorithm for unsigned distance [28].

near = isNearInterface(data, interface_level, strict_opposite): Determines for each node in the data array whether it is near the interface. The interface is defined by an isosurface (level set) of the implicit surface function. A node is "near" the interface if any of its neighbors are on the opposite side of the interface. Array parameter data stores the implicit surface function and may be of any shape. Optional double parameter interface_level specifies which isosurface defines the interface (default 0.0). Optional boolean parameter strict_opposite specifies whether the neighbours of nodes which lie exactly on the interface (nodes whose value in array data is equal to interface_level) count as "near" the interface. The default value 0 (false) specifies that all neighbors of nodes lying on the interface are themselves near the interface. The nodes lying on the interface are always near the interface. The return boolean array near is the same shape as data, and is 1 (true) when the corresponding node in data is near the interface.

### 3.7.5   Terminal Event

The routines in this directory are examples of the TerminalEventFunc protocol.

[ value, schemeDataOut ] = terminalEventConverge(t, y, tOld, yOld, schemeDataIn): Detects whether the implicit surface function is no longer changing; in other words, that the calculation has converged. Define the update as the change between y and yOld. Integration is terminated when the norm of the update falls below some tolerance. Several norm options are supported, and the tolerance is the maximum between an absolute tolerance and a relative tolerance (measured relative to y). The structure schemeDataIn may contain the fields convergeAbsTol, convergeRelTol and/or convergeNorm. The field convergeAbsTol is a scalar double and specifies the absolute tolerance (defaults to $10^{-3}$). The field convergeRelTol is a scalar double and specifies the relative tolerance (defaults to $10^{-6}$). The field convergeNorm is one of the strings 'average', 'maximum' or 'pointwise' and specifies the norm to be used (defaults to 'average'). The corresponding convergence tests are respectively the average update over all nodes, the maximum update over all nodes, or each node's update is tested individually (and hence has its own relative tolerance to satisfy). The input arguments t and tOld are ignored, and the output argument schemeDataOut = schemeDataIn (no change). This helper functions does not work on vector level sets. **This routine has not been well tested.**

### 3.7.6  Visualization

This section describes functions in `Examples/Helper/Visualization`, which are used to simplify various visualization tasks.

**`h = addSlopes(point, width, styles, slopes, labels)`**: Plots one or more lines of specified slope. When plotting experimental convergence rates of algorithms, it is often useful to have comparison lines of specified slope, which correspond to certain theoretical convergence rates. This function is usually called for a figure which has already been created (and hopefully has `hold on` so that `addSlopes` does not destroy the existing figure). For example, the script `firstDerivSpatialConverge` in section 2.11.2 uses `addSlopes` when creating figure 36. Vector parameter `point` (with two elements) specifies a point from which all slope lines emanate to the right. Scalar parameter `width` specifies the extent of the lines in the horizontal direction. Parameter `styles` may be a string or a cell vector of strings, which specifies the line style(s) of the slope lines. Vector parameter `slopes` is a list of slope lines which should be shown. Cell vector parameter `labels` contains one string for each slope line, which is displayed to the right of the end point of the corresponding slope line. The output `h` is a two column array of graphic handles; the first column contains the line handles for the slope lines and the second column the text handles for the labels.

**`spinAnimation(fig, filename, compress)`**: A routine which demonstrates how to use MATLAB's animation facilities to generate an animation of a spinning three dimensional plot. When working with surfaces in three dimensions, it is often difficult to understand the shape without seeing it from several angles. Interactive MATLAB has `rotate3d`, but it is difficult to use during a talk; consequently it is usually better to generate an animation showing the surface from many different angles—if you have seen the author of the toolbox give a talk, then you have probably seen an animation created by this routine. The parameter `fig` is a figure handle to the already created three dimensional plot. The string parameter `filename` is the name of the output animation file (which will have the extension `.avi` appended). The boolean parameter `compress` specifies whether lossy compression should be used to (significantly) reduce the size of the resulting animation, at the expense of some image quality. Remaining parameters, such as animation resolution, number of frames and compression quality, can be set within the source code. Note that this function will probably work only in the Windows environment, since it uses the `avi` file format.

**`h = visualizeLevelSet(g, data, displayType, level, titleStr)`**: Create a visualization of an implicit surface function. At present, dimensions one to three are supported. This function is designed to produce quick visualizations of implicit surface functions, rather than polished figures. While many of the figures in this document started as calls to `visualizeLevelSet`, they were usually then modified by adding labels, improving the viewing angle and/or lighting, or adding more graphical objects.

The visualization is created within the current figure and axis, so this function can be used with `subplot` and to create multiple implicit surfaces in a single plot. The grid structure is given by parameter `g`, and the implicit surface function by array parameter `data`. The string parameter `displayType` specifies which type of visualization to use; the options depend on the dimension of the grid and are given in the `help` entry for this routine. The optional scalar `level` specifies which level set to visualize, and defaults to zero. The optional string `titleStr` creates a title text object for the current axis. The return parameter `h` is a graphics handle to the created graphics object(s), which can be used to modify the appearance using MATLAB's `set` command; for example, the color can be changed. By default, a warning is generated if the requested level set does not exist in `data`. This warning can be disabled by setting `level = '[]'` (useful for types of display that do not plot a level set, such as `'surf'` in two dimensions).

## 3.8 Vector Level Sets

Vector level sets, or systems of Hamilton-Jacobi PDEs are essentially solving (1) with $\phi \in \mathbb{R}^m$ and $H \in \mathbb{R}^m$, where $m$ need not have any particular relationship with the state space dimension $n$. Unfortunately, there is very little theoretical work on such systems of nonlinear PDEs, so care must be taken drawing any conclusions from computed approximations.

However, a number of quite interesting applications make use of vector level sets—for example, open surfaces of codimension one (see section 2.9.1, surfaces of codimension greater than one [24, chapter 10], and analysis of hybrid systems [20]—so the feature has been added to the toolbox. In order to maintain a connection to at least the theory for scalar equations, each element $\phi_i$ and $H_i$ of $\phi$ and $H$ should independently satisfy all the requirements detailed in previous sections.

Vector level sets could be easily implemented as $m$ independent runs of the toolbox, but if the evolution of the $\phi_i$ depend on each other, then the integrations must be kept temporally aligned. Doing so with the `SingleStep` and `MaxStep` options of the `odeCFLn` integrators is technically possible but inefficient and clumsy to implement. Consequently, the integrators and term approximation routines have been rewritten to add direct support for vector level sets.

Integration of scalar HJ PDEs with the toolbox involves three collections of information.

- The function $\phi(x, t)$. This data is commonly referred to by the variable `data`, except inside the time integrators where it is called `y` or `y0` to distinguish the case when it has been reshaped from its usual array of size `grid.shape` into a long vector.

- The term approximation routine for $H$, commonly the variable `schemeFunc`.

- Various auxiliary parameters required by the term approximation routine and any user supplied subroutines, commonly the variable `schemeData`.

In order to handle systems of HJ PDEs of size $m$, each of these data collections is replaced by a cell vector of length $m$. Each element of the cell vector contains the data normally used by a scalar equation.

For example, consider the integration of a codimension two curve in $\mathbb{R}^3$. Such a curve can be represented as the intersection of the zero level sets of two level set functions evolving on the same computational grid [24, chapter 10]. In the toolbox, this would require `data`, `schemeFunc` and `schemeData` to each be cell vectors of length two. Then `data{1}` would be an array of size `grid.shape` containing the first level set function $\phi_1(x, t)$, `schemeFunc{1}` would be a function handle to a term approximation routine controlling the evolution of $\phi_1$, and `schemeData{1}` would be a structure containing fields relevant to the term approximation routine for $\phi_1$. Similarly, `data{2}`, `schemeFunc{2}` and `schemeData{2}` would contain $\phi_2(x, t)$ and information about how to evolve it.

In some cases, the elements of the vector level set may share identical evolution information; in other words, they may all be evolved using the same term approximation routine and/or parameters. In that case, `schemeFunc` may be a single function handle and/or `schemeData` may be a single structure, which will be used for all the elements $\phi_i$.

With this design, the temporal integrators are able to compute the update for all elements of the vector level set, choose a timestep small enough to satisfy all of the CFL restrictions, and update the $\phi_i$ to the next time in lockstep.

The only remaining complication is how the cell vectors are passed through to the term approximation routines, and ultimately on to user code for parameters like the velocity field $v(x, t)$ in (5) or the Hamiltonian $H(x, t, \phi, \nabla\phi)$ in (8). In particular, how do these routines know which element of the cell arrays is to be examined, and how can we make use of existing code for scalar equations in those vector level set cases where the update for $\phi_i$ does not depend on any other $\phi_j$?

The proposed solution taken by the toolbox is to treat each of the cell vectors as a circular list, and rotate this list as each element is processed so that the current element is always at the start of the list. Following on the example descrbed above, when `schemeFunc{1}` is called, the term approximation routine that it points to should operate on `data{1}` (where $\phi_1(x, t)$ is currently stored), possibly referring to `data{2}` (for $\phi_2$) and `schemeData{1}` (the parameters for the evolution of $\phi_1$). So far, not at all surprising.

However, when the function handle passed by the user as `schemeFunc{2}` is called, the `data` and `schemeData` lists will have been rotated. Therefore, the term approximation routine that `schemeFunc{2}` points to should also operate on the array passed to it as `data{1}` (which now

contains $\phi_2(x, t)$), possibly referring to `data{2}` (which now contains $\phi_1$) and `schemeData{1}` (the parameters for the evolution of the level set function currently stored in `data{1}`, which is $\phi_2$). This is perhaps surprising until it is realized that a term approximation routine has no way of determining whether it is called as `schemeFunc{1}` or as `schemeFunc{2}`. Without such knowledge, the easiest way to specify on which element of the cell vectors it should operate is to keep the simple rule that it always operates on the first. The remaining elements are rotated as a circular list to achieve such an ordering.

More generally for cell vectors of length `m`, let $v_{\texttt{init}}$ stand for the cell vectors passed into a time integration routine by the user, and $v_{\texttt{term}}$ be the cell vectors passed on to a term approximation routine, where $v$ is any of the variables commonly called `data`, `schemeFunc` or `schemeData`. When the term approximation routine pointed to by `schemeFunc`$_{\texttt{init}}${`i`} is called, then

$$\texttt{data}_{\texttt{term}} = \texttt{data}_{\texttt{init}}([\texttt{i} : \texttt{m}, 1 : \texttt{i} - 1])$$
$$\texttt{schemeData}_{\texttt{term}} = \texttt{schemeData}_{\texttt{init}}([\texttt{i} : \texttt{m}, 1 : \texttt{i} - 1])$$

The method described above manages to convey to the term approximation routines on which of the cell vector elements they should operate. We would like to use a similar rule for subroutines called by the term approximation routines. However, many of these routines have been written assuming that the `data` and `schemeData` parameters are not cell vectors. Rather than rewriting all such routines (as had to be done with the term approximation routines), it would be nice to provide the option of either passing through the vector level set information (if it is needed) or passing through only a scalar level set (in order to reuse scalar code for motions that are purely scalar in nature).

The resulting protocol depends on an element of the `schemeData` structure called `passVLS`, which is meant to convey "pass through vector level set information." This element is always optional, and its default value is `0` (false). For those term approximation routines that support it, a value of `1` (true) indicates that any user subroutines (such as routines conforming to the `scalarGridFunc` prototype) should be passed the entire cell vectors for `data` and `schemeData`. In this case, the user supplied subroutine will be able to access the current element of the vector level set in `data{1}`, and allow other elements of the vector level set (stored in the other elements of the `data` cell vector) to influence its evolution.

For those routines that do not support `passVLS`, and for those routines that do but are passed a value of `0` (false), only `data{1}` and `schemeData{1}` are passed. In this case, from the subroutine's perspective, the call appears to be for a scalar PDE. This option should be used for all code unless it is specifically written for vector level set evolution.

In version 1.1, the following routines support the `passVLS` protocol: `termCurvature`, `termForcing`, `termNormal`.

The following routines may not support the passVLS protocol (see the help entries for details): termConvection, termDiscount, termLaxFriedrichs, termReinit, termTraceHessian. As described above, for these routines currently act as though passVLS were always false—only data{1} and schemeData{1} are passed through to user subroutines. Unless hidden bugs are discovered in the passVLS protocol, it is expected that it will be added to these routines soon.

There are two exceptions to the passVLS protocol: the routines for combining and restricting term approximations from section 3.6.4. In brief, termRestrictUpdate always passes through the entire cell vector, and termSum passes through the entire cell vector, but modifies the first elements data{1} and schemeData{1}. For full details, see the help entries for these routines.

The protocols for vector level sets may seem complex at first, but examination of the examples in section 2.9 will hopefully clear up most confusion. Several alternative protocols for specifying vector level sets were explored, but the one defined here was the most flexible that we have found so far. We would definitely like to hear about your experiences using this protocol, especially any cases that prove to be impossible to reformulate in this manner. Collection of experiences with this vector level set scheme is one of the reasons for a beta release of this version of the toolbox.

# 4   New and Future Features

The Toolbox is evolving as new schemes, features and examples are added, and new versions will be released as code stability and time permit. Users are encouraged to email comments, bug fixes and feature requests to the author(s).

There are also examples and schemes designed for the Toolbox which are not included in the base download. You may find this other code at the toolbox website [14].

## 4.1   Features New to Version 1.1

Not all of the new and modified routines for this version have appeared in this documentation. However, they should all be listed below, and their help entries should fully document their features. All of the major modifications have been covered in the documentation.

- Implementation of term approximation routines for motion by the trace of the Hessian (10) (section 3.6.3), discounting (11) and forcing (12) (section 3.6.4). For examples, see section 2.8.

- Vector level sets. This feature required modifications to virtually all integration and term approximation routines, but (hopefully) these modifications are completely backward compatible for scalar equations. See section 3.8 for an explanation of how vector level sets are implemented, and section 2.9 for examples of their use.

- The `schemeFunc` prototype has been modified to return the `schemeData` structure as its third argument, and all term approximation routines have been updated to follow the new prototype. Although none of the routines currently make use of this feature, in theory it permits term approximation routines to modify the structure's elements (a capability previously only allowed for routines following the `postTimestepFunc` prototype).

- Support for some types of stationary HJ PDEs using a reformulation to a time-dependent PDE and the `postTimestepTTR` routine. See sections 2.7 and 3.7.3.

- Time dependence for virtually all term parameters. This feature was actually present in version 1.0 of the Toolbox, but was not advertised in the documentation.

- New option `'TerminalEvent'` in `odeCFLset` for early event driven termination of integration. See section 3.5.3.

- Added the subcell fix from [27] as the default behaviour in `termReinit`. This modification seems to significantly decrease the motion of the implicit surface during reinitialization. If this subcell fix option is not used, there is also a new option to increase the degree of smoothing of the signum function in the reinintialization equation. See sections 2.2, 2.10 and 3.6.1.

- The new `postTimestepReinit` routine to support regular reinitialization of the level set during integration. See section 3.7.3. An example of its application appears in section 2.9.1.

- Several new examples: stationary HJ PDEs in section 2.7, expected outcomes of stochastic differential equations in section 2.8, a spiral crystal growth demonstration from [31] of open curves by vector level sets in section 2.9, and the examples from [27] demonstrating the benefits of the subcell fix for reinitialization.

- In `visualizeLevelSet`: Added a `'wireframe'` option in 3D, fixed the `'slice'` and `'contourslice'` options, and modified the behaviour of the `'surface'` option to provide smoother looking surfaces.

- Miscellaneous modified routines: `signedDistanceIterative` and `dumbbell1`.

- Miscellaneous new routines in existing directories. The following routines are described in this manual:
  - `cellMatrixTrace`
  - `gridnd2mesh`
  - `isNearInterface`
  - `odeCFLget`
  - `odeCFLmultipleSteps`
  - `odeCFLcallPostTimestep`
  - `shapeHyperplaneByPoints`

  The following routines are not described in the manual, but should have self-explanatory help entries:
  - `addGhostExtrapolate2`
  - `animateSpinStar`
  - `animateAcoustic`
  - `animateAir3D`
  - `animateDumbbell`

## 4.2 Potential Future Features

At the completion of this version of the toolbox, among the extensions which seem useful are:

131

- More general Dirichlet and Neumann boundary conditions.

- The WENO3 upwind first order spatial derivative scheme.

- Roe-Fix and possibly Gudonov numerical Hamiltonians. Stencil Lax-Friedrichs artificial dissipation.

- ENO/WENO function value interpolation (not just gradients) away from nodes.

- Higher order of accuracy versions of the subcell fix in `termReinit`.

- Implicit time stepping (with MATLAB's ODE suite?).

- Adding the `passVLS` vector level set option to all the term approximation routines.

- Some method to avoid constant reallocation of memory for ghost cells.

- Adaptively refined grids.

- Construction of signed distance functions from point clouds.

- Evolution of closed curves in $\mathbb{R}^3$ using vector level sets.

- Examples from various application fields.

Do you have any other ideas?

# Acknowledgements:

# References

[1] M. Athans and P. L. Falb, *Optimal Control.* New York: McGraw-Hill, 1966.

[2] M. Bardi and I. Capuzzo-Dolcetta, *Optimal Control and Viscosity Solutions of Hamilton-Jacobi-Bellman equations.* Boston: Birkhäuser, 1997.

[3] P. Cardaliaguet, M. Quincampoix, and P. Saint-Pierre, "Optimal times for constrained nonlinear control problems without local controllability," *Applied Mathematics and Optimization*, vol. 35, no. 1, pp. 1–22, 1997.

[4] ——, "Set-valued numerical analysis for optimal control and differential games," in *Stochastic and Differential Games: Theory and Numerical Methods*, ser. Annals of International Society of Dynamic Games, M. Bardi, T. E. S. Raghavan, and T. Parthasarathy, Eds. Birkhäuser, 1999, vol. 4, pp. 177–247.

[5] D. Chopp, "Computing minimal surfaces via level set curvature flow," *Journal of Computational Physics*, vol. 106, pp. 77–91, 1993.

[6] 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, pp. 487–502, 1984.

[7] 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.

[8] M. G. Crandall, H. Ishii, and P.-L. Lions, "User's guide to viscosity solutions of second order partial differential equations," *Bulletin of the American Mathematical Society*, vol. 27, no. 1, pp. 1–67, 1992.

[9] L. C. Evans, "An introduction to stochastic differential equations, lecture notes." [Online]. Available: http://math.berkeley.edu/~evans/SDE.course.pdf

[10] ——, *Partial Differential Equations.* Providence, Rhode Island: American Mathematical Society, 1998.

[11] M. Falcone, "Numerical solution of dynamic programming equations," in *Optimal Control and Viscosity Solutions of Hamilton-Jacobi-Bellman equations.* Birkhäuser, 1997, appendix A of [2].

[12] R. Fedkiw, T. Aslam, B. Merriman, and S. Osher, "A non-oscillatory Eulerian approach to interfaces in multimaterial flows (the ghost fluid method)," *Journal of Computational Physics*, vol. 152, pp. 457–492, 1999.

[13] P. E. Kloeden and E. Platen, *Numerical Solution of Stochastic Differential Equations*, 3rd ed., ser. Applications of Mathematics. Springer, 1999.

[14] [Online]. Available: http://www.cs.ubc.ca/~mitchell/ToolboxLS

[15] I. Mitchell and C. Tomlin, "Level set methods for computation in hybrid systems," in *Hybrid Systems: Computation and Control*, ser. Lecture Notes in Computer Science, B. Krogh and N. Lynch, Eds.  Springer Verlag, 2000, no. 1790, pp. 310–323.

[16] I. M. Mitchell, "The flexible, extensible and efficient toolbox of level set methods," submitted to *Journal of Scientific Computing*, 26 pages, February 2007.

[17] ——, "Application of level set methods to control and reachability problems in continuous and hybrid systems," Ph.D. dissertation, Scientific Computing and Computational Mathematics Program, Stanford University, August 2002.

[18] ——, "A toolbox of level set methods (version 1.1)," Department of Computer Science, University of British Columbia, Vancouver, BC, Canada, Tech. Rep. TR-2007-11, June 2007. [Online]. Available: http://www.cs.ubc.ca/~mitchell/ToolboxLS/toolboxLS.pdf

[19] 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.

[20] I. M. Mitchell and J. A. Templeton, "A toolbox of Hamilton-Jacobi solvers for analysis of nondeterministic continuous and hybrid systems," in *Hybrid Systems: Computation and Control*, ser. Lecture Notes in Computer Science, M. Morari and L. Thiele, Eds. Springer Verlag, 2005, no. 3414, pp. 480–494.

[21] A. Oberman, "A convergent upwind difference scheme for motion of level sets by mean curvature," *Numerische Mathematik*, vol. 99, no. 2, pp. 365–379, 2004.

[22] B. Øksendal, *Stochastic Differential Equations: an Introduction with Applications*, 6th ed.  Springer, 2003.

[23] S. Osher, "A level set formulation for the solution of the Dirichlet problem for Hamilton-Jacobi equations," *SIAM Journal of Mathematical Analysis*, vol. 24, no. 5, pp. 1145–1152, 1993.

[24] S. Osher and R. Fedkiw, *Level Set Methods and Dynamic Implicit Surfaces*.  Springer, 2002.

[25] S. Osher and J. A. Sethian, "Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations," *Journal of Computational Physics*, vol. 79, no. 1, pp. 12–49, 1988.

[26] S. Osher and C.-W. Shu, "High-order essentially nonoscillatory schemes for Hamilton-Jacobi equations," *SIAM Journal on Numerical Analysis*, vol. 28, no. 4, pp. 907–922, 1991.

[27] G. Russo and P. Smereka, "A remark on computing distance functions," *Journal of Computational Physics*, vol. 163, pp. 51–67, 2000.

[28] J. A. Sethian, *Level Set Methods and Fast Marching Methods*. New York: Cambridge University Press, 1999.

[29] J. A. Sethian and A. Vladimirsky, "Ordered upwind methods for static Hamilton-Jacobi equations: Theory and algorithms," *SIAM Journal on Numerical Analysis*, vol. 41, no. 1, pp. 325–363, 2003.

[30] C.-W. Shu and S. Osher, "Efficient implementation of essentially non-oscillatory shock-capturing schemes," *Journal of Computational Physics*, vol. 77, pp. 439–471, 1988.

[31] P. Smereka, "Spiral crystal growth," *Physica D*, vol. 138, pp. 282–301, 2000.

[32] R. J. Spiteri and S. J. Ruuth, "A new class of optimal high-order strong-stability-preserving time discretization methods," *SIAM Journal on Numerical Analysis*, vol. 40, no. 2, pp. 469–491, 2002.

[33] M. Sussman, P. Smereka, and S. Osher, "An improved level set method for incompressible two-phase flow," *Journal of Computational Physics*, vol. 114, pp. 145–159, 1994.

[34] C. Tomlin, J. Lygeros, and S. Sastry, "A game theoretic approach to controller design for hybrid systems," *Proceedings of the IEEE*, vol. 88, no. 7, pp. 949–970, July 2000.

[35] C. Tomlin, I. Mitchell, A. Bayen, and M. Oishi, "Computational techniques for the verification of hybrid systems," *Proceedings of the IEEE*, vol. 91, no. 7, pp. 986–1001, July 2003.

[36] Y.-H. R. Tsai, L.-T. Cheng, S. Osher, and H.-K. Zhao, "Fast sweeping methods for a class of Hamilton-Jacobi equations," *SIAM Journal on Numerical Analysis*, vol. 41, no. 2, pp. 673–694, 2003.

# Concept Index

# Command Index