A Toolbox of Level Set Methods
version 1.0

UBC CS TR-2004-09

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

July 1, 2004

Abstract

This document describes a toolbox of level set methods 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.

Copyright

License

Contents

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 ϕ. 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 [13], 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 [15] and [12]. 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 [12]: high order accuracy methods, image processing, computational physics.
- Sethian [15]: 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 [12].

1.1 Contents of the Toolbox

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 equations on fixed, structured Euclidean grids in any number of dimensions. More concretely, the PDE to be solved is of the form

$$
\begin{aligned}
0 = & D_t\phi(x,t) && (1)\\
& + v(x)\cdot\nabla\phi(x,t) && (2)\\
& + a(x)\|\nabla\phi(x,t)\| && (3)\\
& + \text{sign}(\phi(x,0))(\|\nabla\phi(x,t)\|-1) && (4)\\
& + H(x,\nabla\phi) && (5)\\
& - b(x)\kappa(x)\|\nabla\phi(x,t)\| && (6)\\
& + \text{trace}[\mathbf{L}(x)D_x^2\phi(x,t)\mathbf{R}(x)] && (7)\\
& + \lambda(x)\phi(x,t) && (8)\\
& + F(x), && (9)
\end{aligned}
$$

*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.

subject to constraints

$$D_t\phi(x,t) \geq 0, \qquad D_t\phi(x,t) \leq 0, \tag{10}$$

$$\phi(x,t) \leq \psi(x), \qquad \phi(x,t) \geq \psi(x), \tag{11}$$

where $x \in \mathbb{R}^n$ is the state space, $\phi : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}$ is the level set function and $\nabla\phi(x,t) = D_x\phi(x,t)$ is the gradient of ϕ. Note that the time derivative (1) and at least one term involving a spatial derivative (2)–(7) must appear, otherwise the equation is not a hyperbolic PDE. Numerical approximations for each type of term are provided.

- Time derivative (1) is approximated with an explicit total variation diminishing Runge-Kutta integration scheme with order of accuracy between one and three [12, 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 constant velocity field (2), also called advection or convection. The user provides the velocity field $v : \mathbb{R}^n \to \mathbb{R}^n$, and the gradient $\nabla\phi(x,t)$ is approximated with an upwind finite difference scheme with order of accuracy between one and five [12, 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 (3). The user provides the speed of the interface $a : \mathbb{R}^n \to \mathbb{R}$, and $\nabla\phi(x,t)$ is approximated with an upwind finite difference scheme [12, 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 (4). 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 [12, chapter 7.4]. A Godunov scheme for its solution can be found in [5, appendix A.3], which allows this term to be stably approximated with a minimum of artificial dissipation. Note that the initial conditions are used inside the signum function. An example is given in section 2.2.1 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 3.7.3.

- A general Hamilton-Jacobi term (5) can treat a variety of applications, including optimal control and differential games. The user provides the analytic Hamiltonian $H : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$. Upwind finite difference approximations of $\nabla\phi(x,t)$ are provided, and Lax-Friedrichs is used to stably approximate the $H(x,p)$ function (with various options for the degree of localization when calculating the artificial dissipation coefficient) [12, 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 (6). The user provides the speed $b : \mathbb{R}^n \to \mathbb{R}^+$, while the mean curvature $\kappa(x)$ and gradient $\nabla\phi(x,t)$ are approximated by centered second order accurate finite difference approximations [12, chapter 4]. An example is given in section 2.3.1, a description of the toolbox routines for centered finite difference approximations 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 (7), which arises from the Kolmogorov or Fokker-Plank equations when working with stochastic differential equations [6, 10]. The user provides the matrices $\mathbf{L}, \mathbf{R} : \mathbb{R}^n \to \mathbb{R}^{n\times n}$, 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 (8), which arise when solving some types of optimal control problems [1] or stochastic differential equations [10] (in which context they relate to the "killing" process). The user provides the discount factor $\lambda : \mathbb{R}^n \to \mathbb{R}$. This feature has not yet been implemented, but will be available in future releases.

- Forcing terms (9), which the user provides $F : \mathbb{R}^n \to \mathbb{R}$. This feature has not yet been implemented, but will be available in future releases.

- Constraints (10) 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 (11) that the implicit surface should not enter or should not exit another implicit surface. The user provides $\psi : \mathbb{R}^n \to \mathbb{R}$ defining the other implicit surface. 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.

1.2 Using the Toolbox

The best way to start 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 (1)–(11).
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.

1.3 Troubleshooting

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

- 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 (5)), 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.

1.4 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.
- 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.

- 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.

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/addPathToKern` 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 (2)

In this section we examine in detail how to implement motion by an external velocity field (2) 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.

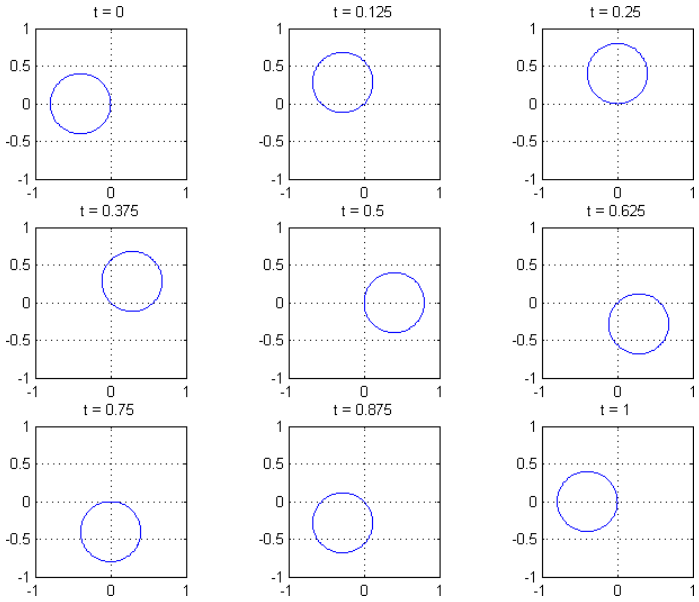


Figure 1: Result of running `convectionDemo('linear', 'medium')`. Shows motion by a constant rotational external velocity field.

- 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.

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.

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

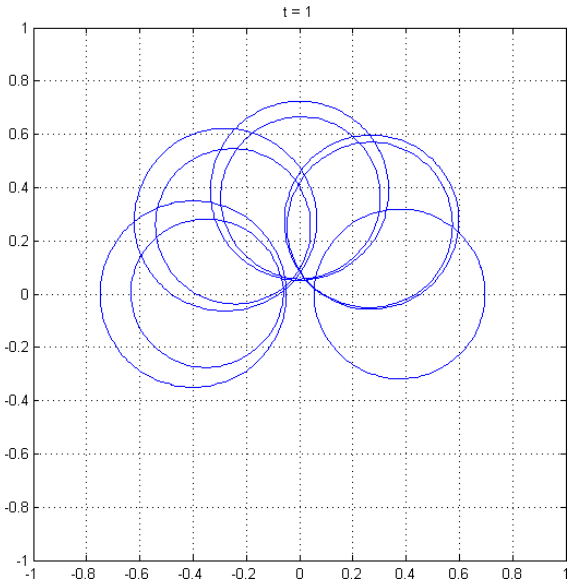


Figure 2: Result of running `convectionDemo('linearRev', 'low')` with internal parameter `useSubplots = 0`. Shows rigid body rotation about the origin, clockwise for the first half of the simulation and then counter clockwise for the remainder. The loss of area associated with using low accuracy methods is obvious from the fact that the two sets of circles do not overlap.

```
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
```

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 ϕ is visualized when using contour plots (in 2D) or surfaces (in 3D).

```
70  %--------------------------------------------------------------------------
71  % Use periodic boundary conditions?
72  periodic = 0;
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. Missing 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
```

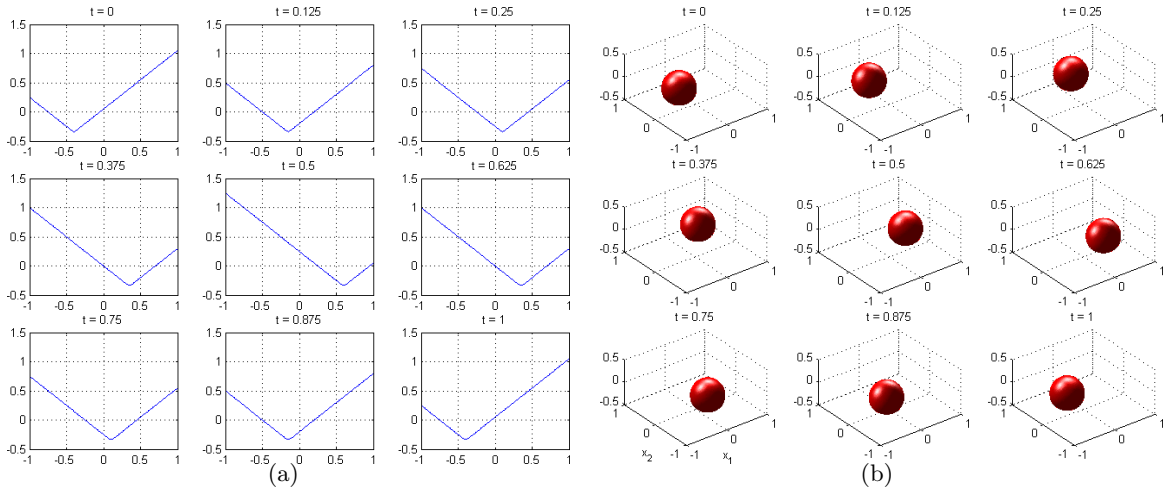


Figure 3: Running `convectionDemo` in other dimensions by modifying the internal parameter `g.dim`. These are not exactly the figures generated during the run: the subplots generated during the run have had their axis bounds adjusted to be consistent across all nine subplots in each case. Figure 3(a): The implicit surface function ϕ 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')`.

```
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 ϕ 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.

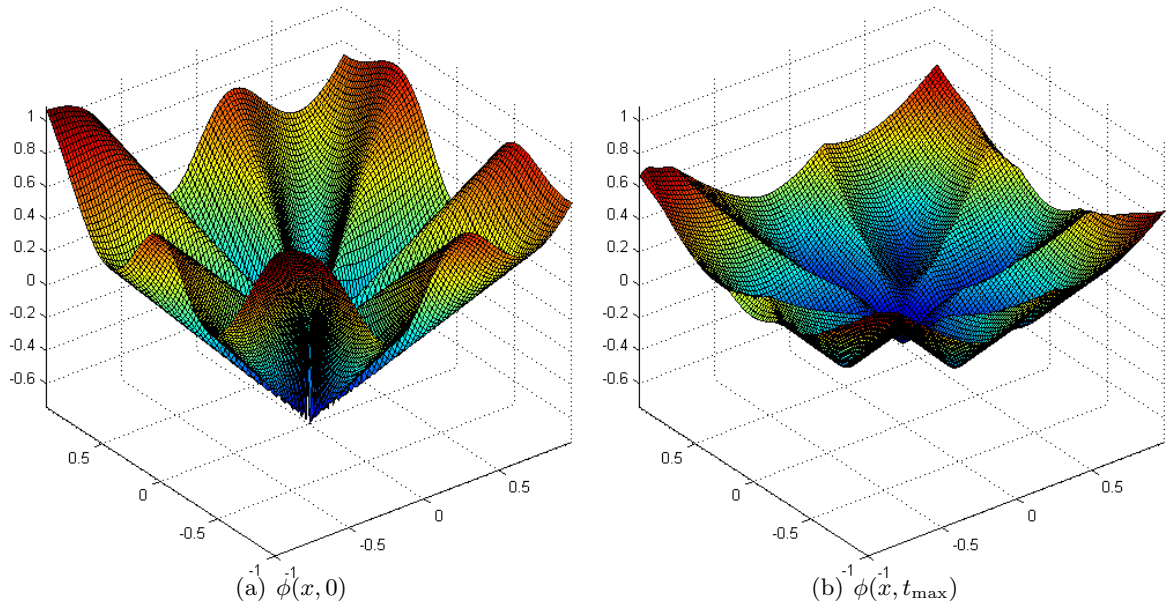
2.2 Basic Examples

This section discusses functions found in the directory `Examples/Basic`. This directory provides an example for each of the types of spatial terms (4)–(11) with the exception of motion by mean curvature (6). Examples for the omitted terms can be found elsewhere: section 2.1 for motion by a constant velocity field (2) and section 2.3 for motion in the normal direction (3) and motion by mean curvature (6). Since terms (8)–(11) do not include a spatial derivative, examples for these terms naturally include a combination with other types of term.

2.2.1 The Reinitialization Equation (4)

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

Reinitialization is the process of modifying an implicit surface function into a signed distance function—modifying ϕ such that $\|\nabla\phi\| \approx 1$ without moving its zero isosurface. One method of reinitialization is to solve the reinitialization equation, which is a general HJ PDE with spatial term (4). Under normal circumstances this task is accomplished with an auxiliary integration routine that hides the details; for example, see `signedDistanceIterative` in section 3.7.3 and `reinitTest` in section 2.7.3. However, for the purposes of demonstrating and testing the term approximation function `termReinit`, we provide the following routine.



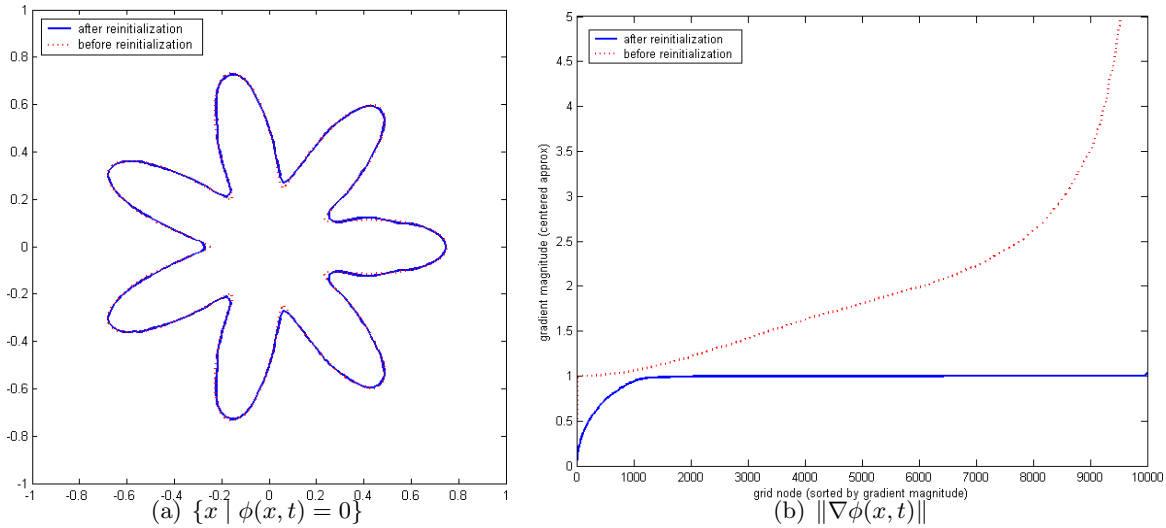
(a) $\phi(x, 0)$ (b) $\phi(x, t_{\max})$

Figure 4: Comparing initial and final implicit surface functions for `reinitDemo('star', 'medium', 'surf')`. Notice how the slope of the final ϕ is much more consistent.

`[ 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`.

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 (13) in section 2.3 for the initial implicit surface equation). Figure 4 shows the results for the `'star'` case, while figure 5 shows how the reinitialization procedure successfully adjusts the gradient magnitude without distorting the zero isosurface too badly. These results were calculated on a relatively coarse grid (`g.dx = 0.02`) using `accuracy = 'medium'`.



(a) $\{x \mid \phi(x,t) = 0\}$ (b) $\|\nabla\phi(x,t)\|$

Figure 5: Examining the effect of reinitialization on the implicit surface (the zero isosurface of $\phi(x,t)$) and the gradient magnitude. The implicit surface has moved only slightly, and at most nodes $\phi(x,t_{\max})$ has close to unit magnitude gradient despite the large gradient of $\phi(x,0)$. Using a higher accuracy scheme would lead to even less movement of the implicit surface.

2.2.2 General HJ Terms (5)

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):D`
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

$$\alpha_i(x) = \max_p \left| \frac{\partial H(x, p)}{\partial p_i} \right| = |v_i(x)|. \tag{12}$$

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 (12) 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: `'low'` = 1, `'medium'` = 4, `'high'` = 12 and `'veryHigh'` = 17.

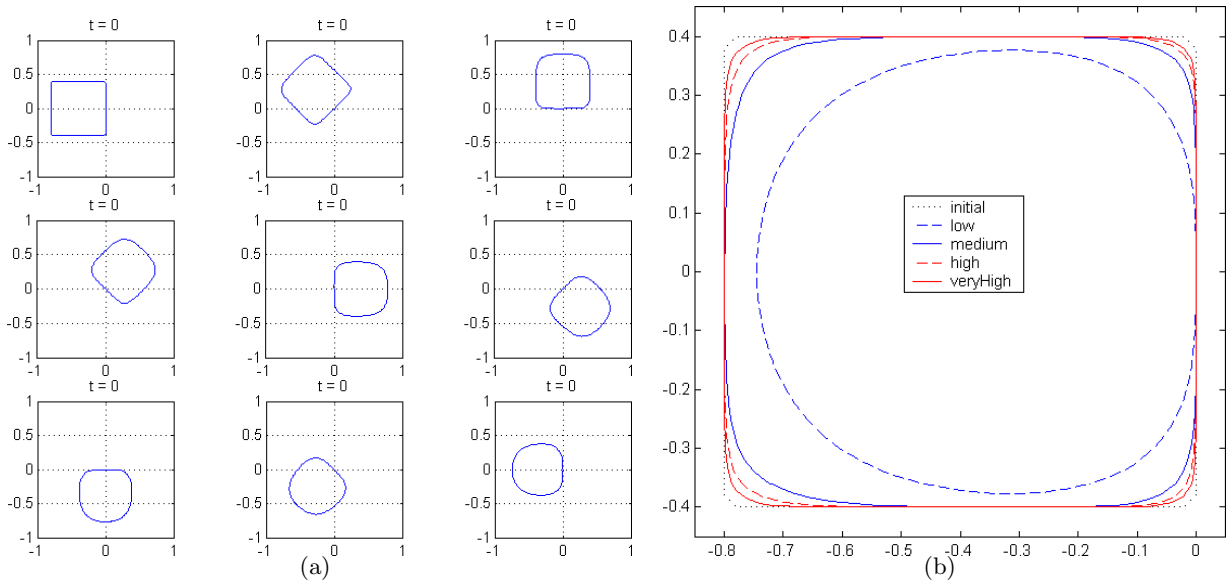


Figure 6: Using Lax-Friedrichs to approximate rotational convective flow in two dimensions with `laxFriedrichsDemo`. Figure 6(a): The individual time steps for `laxFriedrichsDemo('linear', 'cube', 'low')`. Figure 6(b): Comparing the final implicit surface calculated by Lax-Friedrichs when using approximation 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).

2.2.3 Constraints on ϕ (11)

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

Most of the examples deal with terms in the HJ PDE that effect ϕ only through its temporal or spatial dervatives; in contrast, the constraint (11) involves ϕ 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, (11) 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 $\mathcal{S}^{\mathtt{C}}$, 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

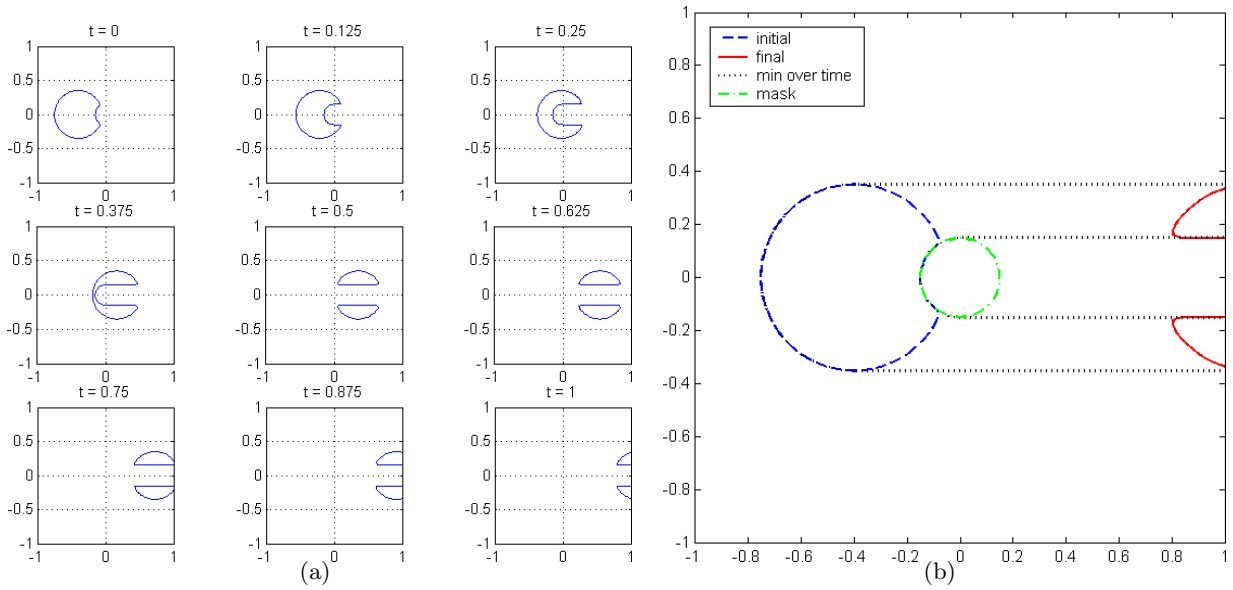


Figure 7: Applications of the `PostTimestep` option of `odeCFLn` to a convection example. Figure 7(a) shows how a constraint of the form (11) can be used to mask out a portion of the state space into which the evolving set cannot enter. Figure 7(b) shows the masking set, as well as $\min_t \phi(x,t)$, which is computed and recorded inside the `PostTimestep` function `maskAndKeepMin`.

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 ϕ to be modified only through its temporal derivative. However, direct modification of ϕ 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 (11) is implemented by modification of `y`, and the constraint function ψ 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 (11). 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 ϕ 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 [12]

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

This section provides routines which recreate some examples from [12]. 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)

$$\phi(x,0) = \|x\| - s\left(\cos\left(\rho \arctan\left(\frac{x_2}{x_1}\right)\right) + \sigma\right) \tag{13}$$

where s is a scale controlling the size of the star, ρ is the number of points, and σ 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.

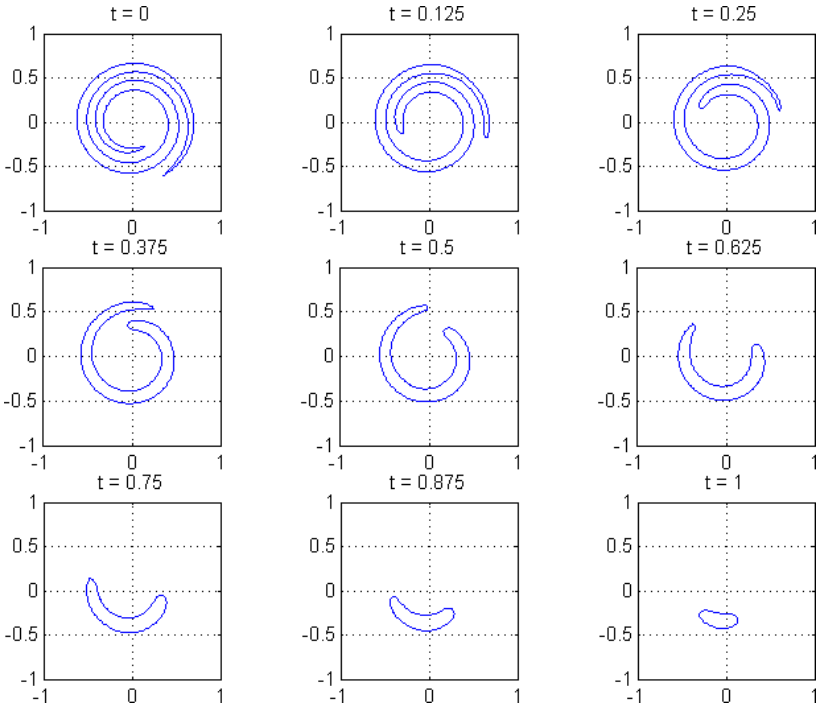


Figure 8: Motion by mean curvature (compare with [12, figure 4.1]). The initial implicit surface function is generated from an ellipse in polar coordinates, rather than the original point cloud description of the problem [13, 12].

2.3.1 Motion by Mean Curvature (6)

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 [13] 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` 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.

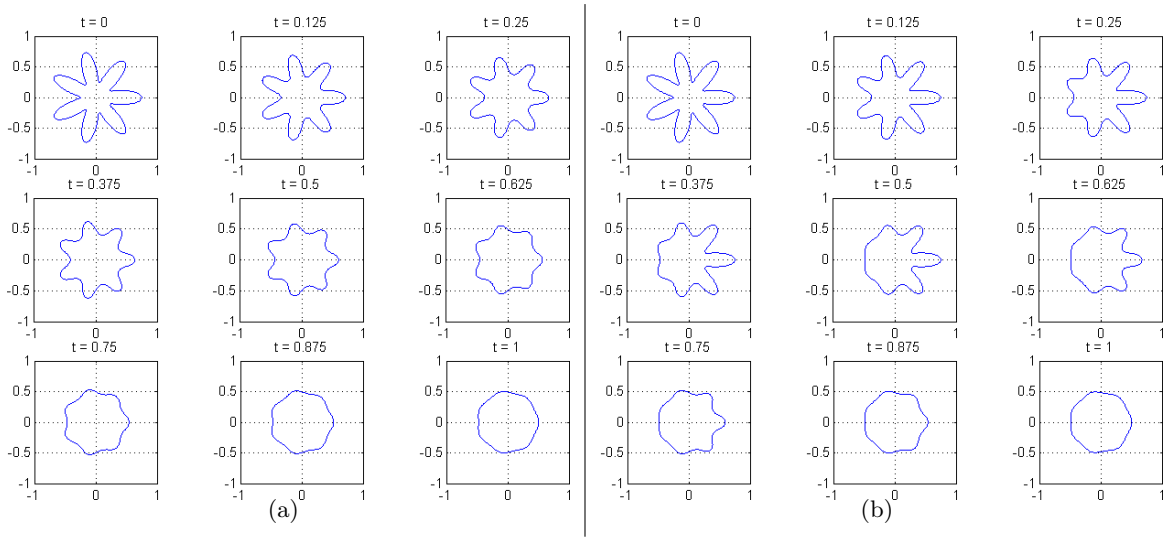


Figure 9: Motion by mean curvature. Figure 9(a) shows motion with constant multiplier b, the result of `curvatureStarDemo` with default parameters (compare with [12, figure 4.2]). Figure 9(b) uses a time and spatially varying multiplier $b(x,t)$ (by choosing `splitFlow == 1`).

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 [13]. 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 (13) are specifically designed for two dimensional grids.

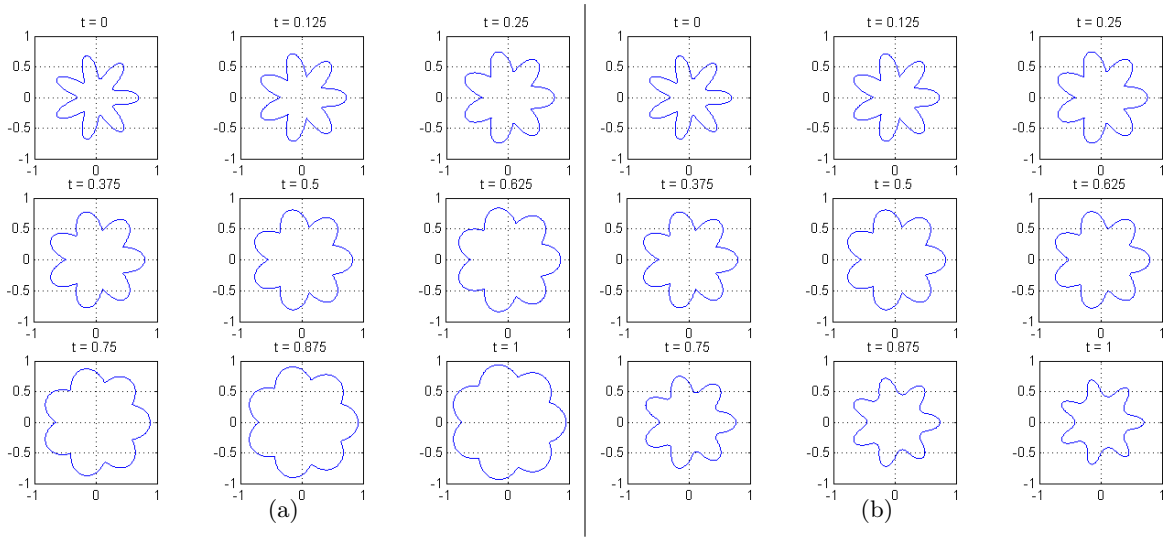


Figure 10: Motion in the normal direction, starting with a star shaped interface. Figure 10(a) shows motion at a constant positive speed, the result of `normalStarDemo` with default parameters (compare with [12, figure 6.1]). Figure 10(b) uses the same speed, but reverses its sign at the midpoint of the simulation (by choosing `reverseFlow == 1`).

The initial conditions and constant multiplier b_0 were chosen to try to match the results of [12, figure 4.2]. For the time and spatially varying case `splitFlow == 1`, the multiplier is given by (the actual implementation uses polar coordinates)

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

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 (3)

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.

`[ 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 (13) are specifically designed for two dimensional grids.

The initial conditions and speed were chosen to try to match the results of [12, 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 (13) and flow fields are specifically designed for two dimensional grids.

Although the caption of [12, 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.

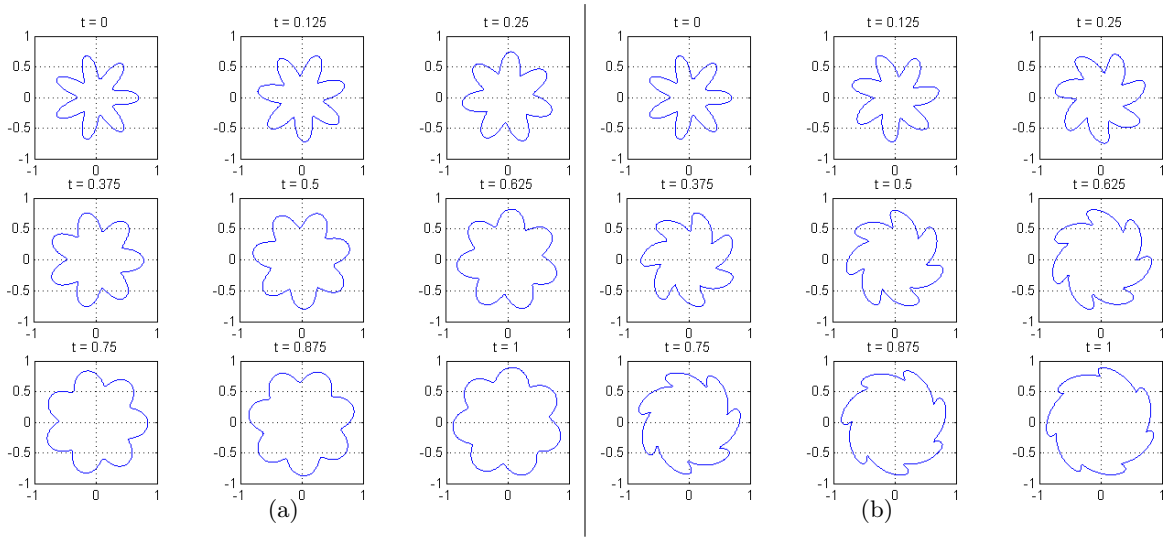


Figure 11: Combining motion in the normal direction with rotational convection. Figure 11(a) shows the results of a rigid body rotation (choose `rigid == 1`). Figure 11(b) multiplies the speed of rotation by the square of the distance to the origin (compare with [12, figure 6.2]). Both figures are generated with `accuracy = 'medium'` on 201^2 grids.

2.4 Examples from Sethian [15]

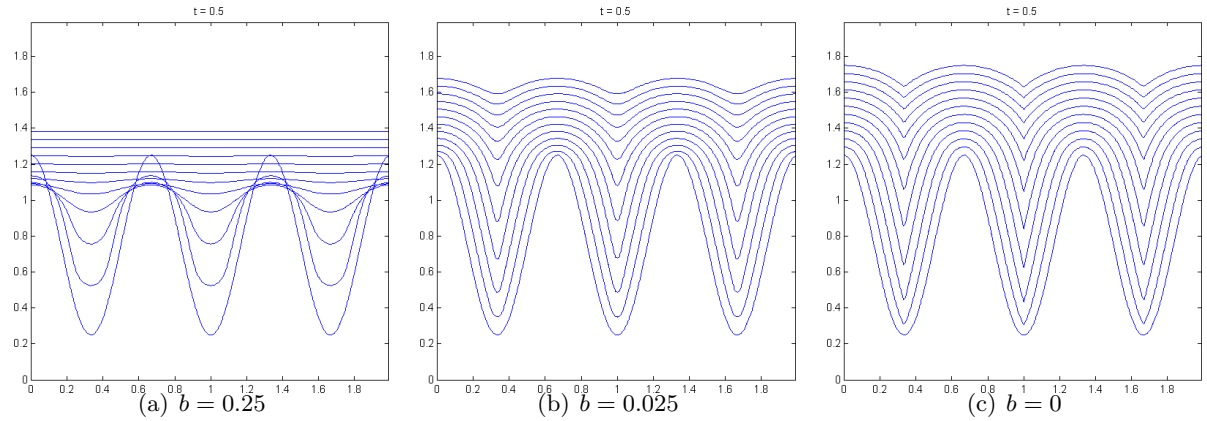
This section provides routines which recreate some examples from [15]. The lack of quantitative parameters in that text—such as figure axis scales with which to reconstruct the initial conditions—makes it challenging to exactly recreate the results.

Before proceeding to the implemented examples, we mention that [15, 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 [15, chapter 2.4] or [4, chapter 10]. In [15, figures 2.6 and 2.7] this claim is examined experimentally on a two dimensional example using motion in the normal direction with speed



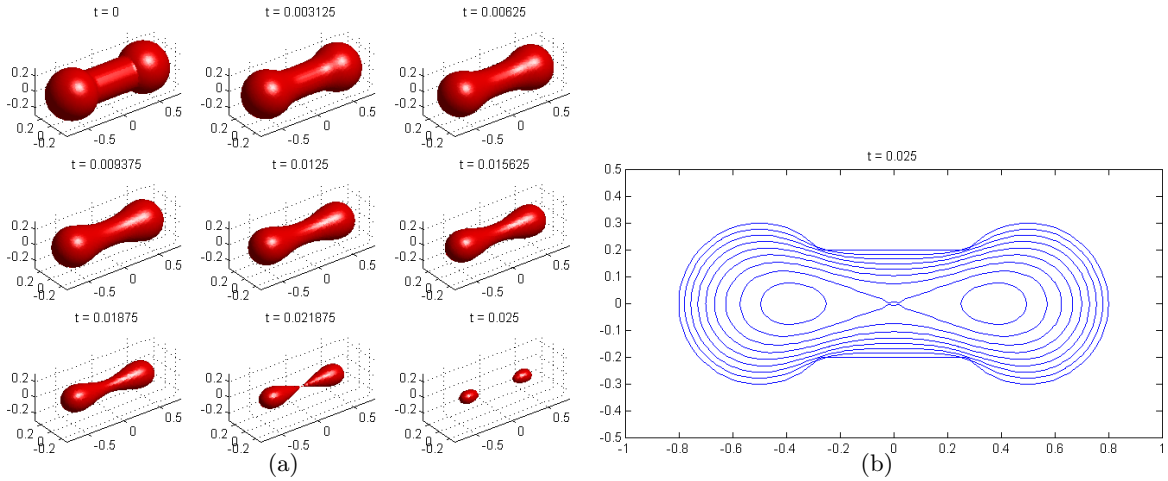
(a) $b = 0.25$ (b) $b = 0.025$ (c) $b = 0$

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 [15, figures 2.6 and 2.7]

$a(x) = 1 - b\kappa(x)$, 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 (3) and (6). 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`.

`[ 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 [15, 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.



(a) (b)

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 [15, figure 14.2]).

2.4.2 Motion by Mean Curvature and Surface Separation

This section describes the function `Examples/Sethian/dumbbell1`.

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 [15, 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.

As an example, the dumbbell is created by

$$\begin{aligned}
\psi_{left}(x) &= \sqrt{(x_1+o)^2+x_2^2+x_3^2}-r,\\
\psi_{right}(x) &= \sqrt{(x_1-o)^2+x_2^2+x_3^2}-r,\\
\psi_{center}(x) &= \max\left[\left(|x_1|-o\right),\left(\sqrt{x_2^2+x_3^2}-w\right)\right],\\
\phi(x,0) &= \min\left[\psi_{left}(x),\psi_{right}(x),\psi_{center}(x)\right],
\end{aligned}$$

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 [13]

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

The method for treating general Hamilton-Jacobi terms (5) adopted by this toolbox and [12] is basically drawn from [13], 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

$$\begin{aligned}
D_t\phi(x,t)+H(\nabla\phi(x,t))=0, \qquad & 1\le x<1,\\
\phi(x,0)=-\cos(\pi x) &
\end{aligned} \tag{14}$$

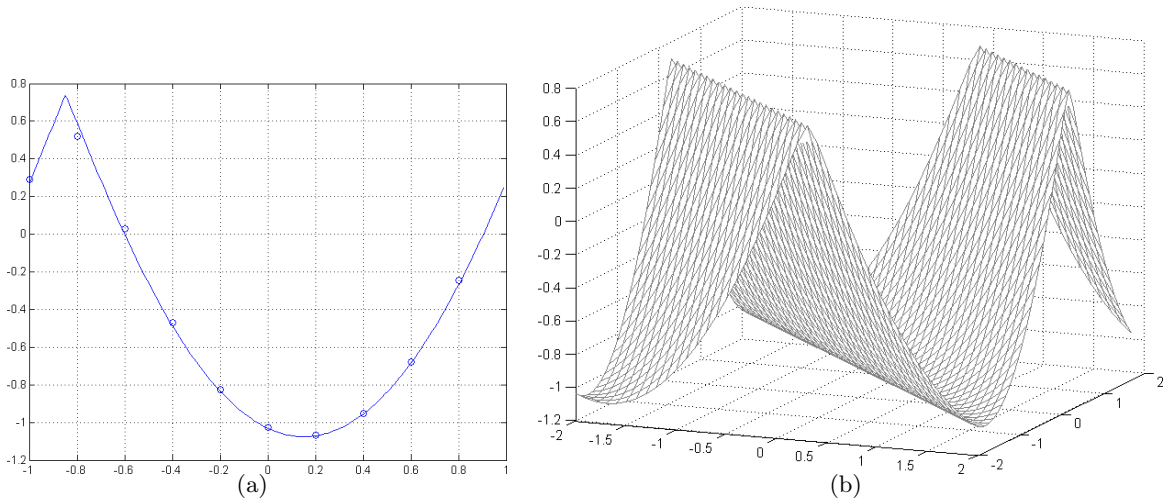


Figure 14: Solving Burgers' equation with Lax-Friedrichs schemes. Figure 14(a) compares the exact solution (solid) with the third order ENO-LLF approximation on a grid of 10 points (circles); compare with [13, figure 1(d)]. Figure 14(b) shows a two dimensional version of Burgers with an ENO-LLF approximation on a 40^2 grid; compare with [13, figure 3(b)].

where $H(p)$ is the convex function

$$H(p) = \frac{\left(\alpha + \sum_{i=1}^{\texttt{grid.dim}} p_i\right)^2}{2}, \tag{15}$$

which makes (14) 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`, `artificialDissipationLLF`, and `artificialDissipationLLLF`.

`[ data, g, data0 ] = burgersLF(accuracy, dissType, gridDim, gridSize, tMax)`: Demonstrates solution of Burgers' equation (14) and (15), 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 [13] 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).

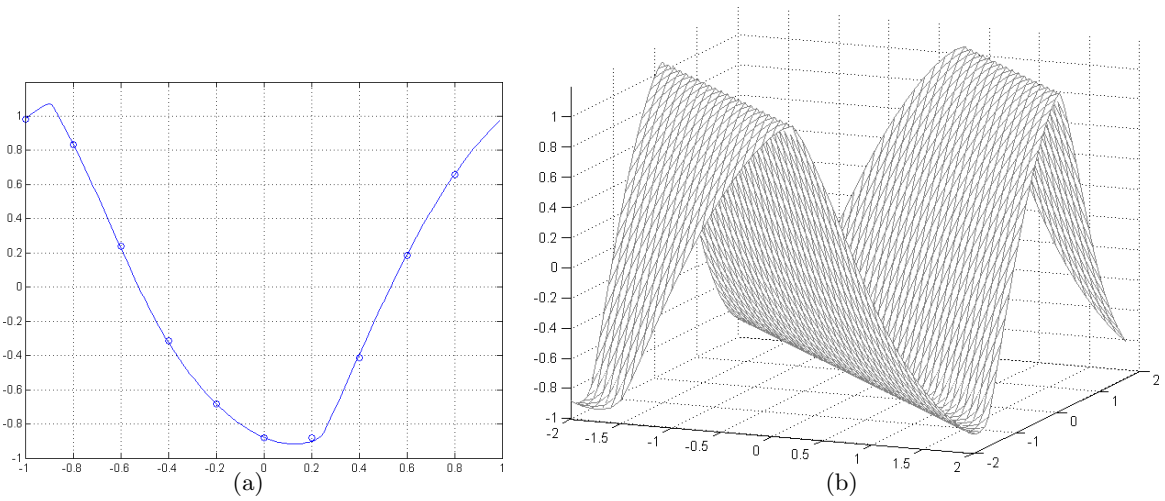


Figure 15: Solving a non-convex general HJ PDE with Lax-Friedrichs schemes. Figure 15(a) compares the exact solution (solid) with the third order ENO-LF approximation on a grid of 10 points (circles); compare with [13, figure 2(d)]. Figure 15(b) shows a two dimensional version of the same equation with an ENO-LF approximation on a 40^2 grid; compare with [13, figure 3(d)]. There may be slightly more dissipation in these solutions than in those of [13] (see the discussion of `nonconvexPartialFunc` below).

Within the file `burgersLF`, the subfunction `burgersHamFunc` implements the `hamFunc` prototype for (15). Subfunction `burgersPartialFunc` implements the `partialFunc` prototype solving (26) with Hamiltonian (15). Note that the dissipation parameter $\alpha_i(x)$ is different from the problem parameter α.

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

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.

2.5.2 Non-Convex Hamiltonian

This section describes the function `nonconvexLF` in the directory `Examples/OsherShu/`, which implements (14), where $H(p)$ is the non-convex function

$$H(p) = -\cos\left(\alpha + \sum_{i=1}^{\texttt{grid.dim}} p_i\right). \tag{16}$$

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 `artificialDissipationLL`

`[ data, g, data0 ] = nonconvexLF(accuracy, dissType, gridDim, gridSize, tMax)`: Demonstrates solution of (14) and (16). 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 [13] 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 (16). Subfunction `nonconvexPartialFunc` implements the `partialFunc` prototype solving (26) with Hamiltonian (16). 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 [18, 7, 9] 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 [9] 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

$$D_t\phi(x,t) + \min[0, H(x, D_x\phi(x,t))] = 0 \tag{17}$$

backwards in time, where

$$H(x,p) = \max_{a\in\mathcal{A}} \min_{b\in\mathcal{B}} p^T f(x,a,b). \tag{18}$$

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 f by -1.

Consideration of (17) reveals that the minimization with zero component is equivalent to constraining the sign of the temporal derivative to be positive (10). 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 (18) degenerates to convection under flow field $v(x) = f(x)$ and can be treated as an example of the form (2). 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 (18) must be treated as a general Hamiltonian (5) 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 (18). 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 (18) 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 (26), which in this context translates to

$$\alpha_i(x) = \max_p \left| \frac{\partial H(x,p)}{\partial p_i} \right| = \max_p \left| \frac{\max_{a\in\mathcal{A}} \min_{b\in\mathcal{B}} p^T f(x,a,b)}{\partial p_i} \right|, \tag{19}$$

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 (19) 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

$$f(x,a,b) = f^x(x) + \mathbf{F}^a(x)a + \mathbf{F}^b(x)b, \tag{20}$$

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 (18) can be determined analytically

$$\begin{aligned} a_i^*(x,p) &= \begin{cases} \underline{\mathcal{A}}_i, & \text{if } \sum_{j=1}^n p_j \mathbf{F}_{ji}^a(x) \le 0; \\ \overline{\mathcal{A}}_i & \text{otherwise;} \end{cases} \\ b_i^*(x,p) &= \begin{cases} \overline{\mathcal{B}}_i, & \text{if } \sum_{j=1}^n p_j \mathbf{F}_{ji}^b(x) \le 0; \\ \underline{\mathcal{B}}_i & \text{otherwise.} \end{cases} \end{aligned} \tag{21}$$

Futhermore, defining

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

the terms (19) for the `partialFunc` routine can be slightly overestimated by

$$\alpha_j(x) \le \left|f_j^x(x)\right| + \sum_{i=1}^{n_a} \left|\mathbf{F}_{ji}^a(x)\right| \mathcal{A}_i^{\max} + \sum_{i=1}^{n_b} \left|\mathbf{F}_{ji}^b(x)\right| \mathcal{B}_i^{\max}. \tag{22}$$

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 [17] and repeated with minor modifications in [18, 9] 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.

[†]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.

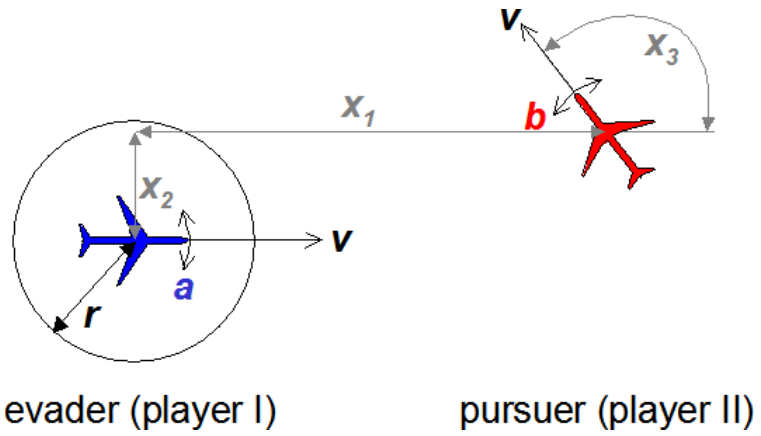


Figure 16: Relative coordinate system for game of two identical vehicles.

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 [18, 9, 7]. 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

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

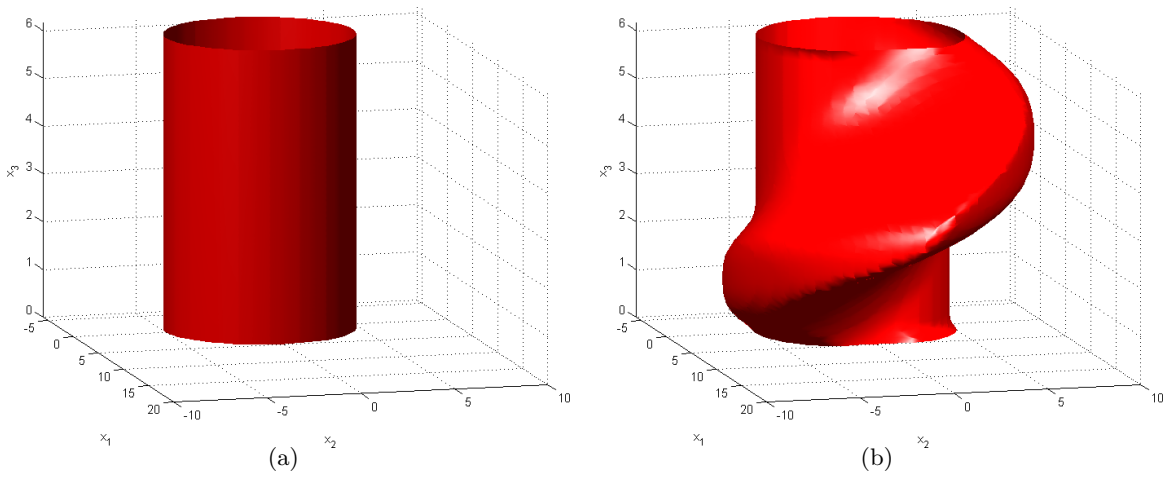


Figure 17: The target and reachable sets for the game of two identical vehicles as visualized by `figureAir3D`. Figure 17(a) shows the target cylinder representing the set of collision states. Figure 17(b) shows the final reachable set at $t = 2.8$, computed by `air3D('medium')`.

relative coordinates with the evader fixed at the origin facing 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

$$\dot{x} = \frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -v_a + v_b \cos x_3 + a x_2 \\ v_b \sin x_3 - a x_1 \\ b - a \end{bmatrix} = f(x, a, b), \tag{23}$$

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.

`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 (23) can be put into the form (20).

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

It is easy to determine from (21) that

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

and the resulting optimal Hamiltonian is

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

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 (22)

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

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

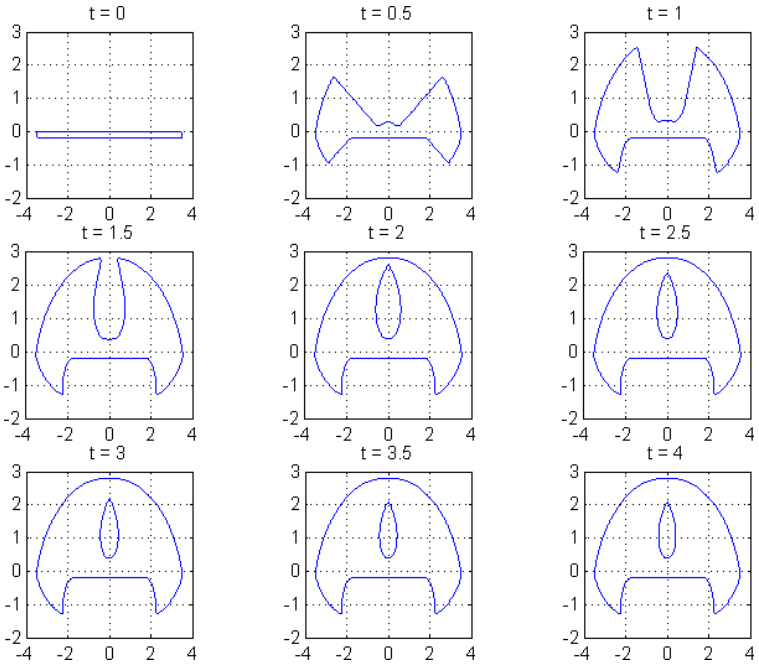


Figure 18: Evolution of the acoustic capture game's reachable set.

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 [2] and we recreate the results in [9]. The reachable set is calculated in relative coordinates with the pursuer fixed at the origin, leading to dynamics

$$\frac{d}{dt}\begin{bmatrix} x \\ y \end{bmatrix} = W_p \begin{bmatrix} 0 \\ -1 \end{bmatrix} + \frac{W_p}{R}\begin{bmatrix} y \\ -x \end{bmatrix} b + 2W_e \min\left(\sqrt{x^2+y^2}, S\right) a = f(z,a,b), \tag{24}$$

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.

Unlike the previous example, (24) cannot be put into the form (20) because the bounds on input a are not dimensionally separable. However, it is relatively easy to find the optimal Hamiltonian

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

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

$$\begin{aligned} \alpha_1(z) &\le \frac{W_p}{R}|y| + 2W_e \min(\sqrt{x^2+y^2}, S), \\ \alpha_2(z) &\le W_p + \frac{W_p}{R}|x| + 2W_e \min(\sqrt{x^2+y^2}, S), \end{aligned}$$

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

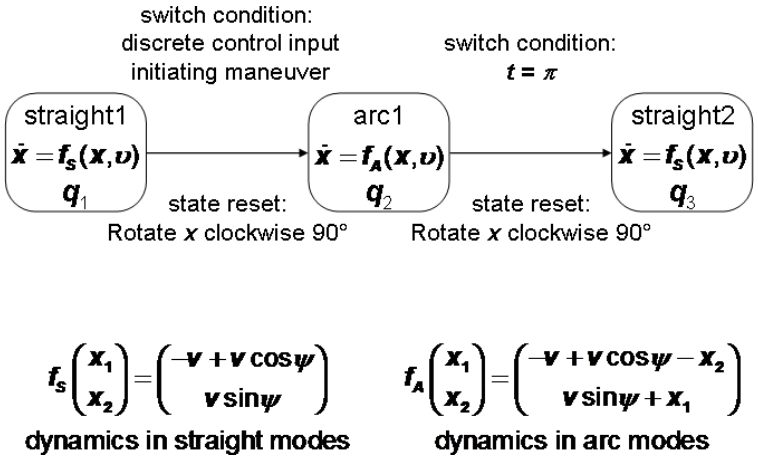


Figure 19: Hybrid automata for the three mode protocol.

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 [8, 9]. 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 σ 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:

- 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).

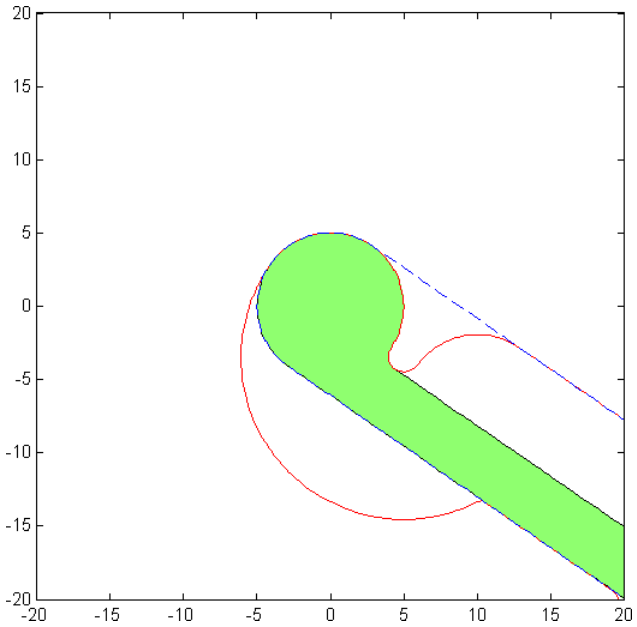


Figure 20: Reachable set in the first mode for the three mode collision avoidance protocol. The solid region is the set of states within which collision is inevitable. Outside the solid contour the protocol can be safely initiated. The dashed contour shows the edges of the unsafe set if no protocol is initiated.

- 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 ω 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`.

`[ reach, g, avoid, data0 ] = airMode(accuracy)`: Demonstrate the three mode collision 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 interaction 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 (11) 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 [18] and the citations within.

2.7 Testing Routines

This section describes functions in `Examples/Test`.

2.7.1 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).

`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.

`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.

2.7.2 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.

`[ 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

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

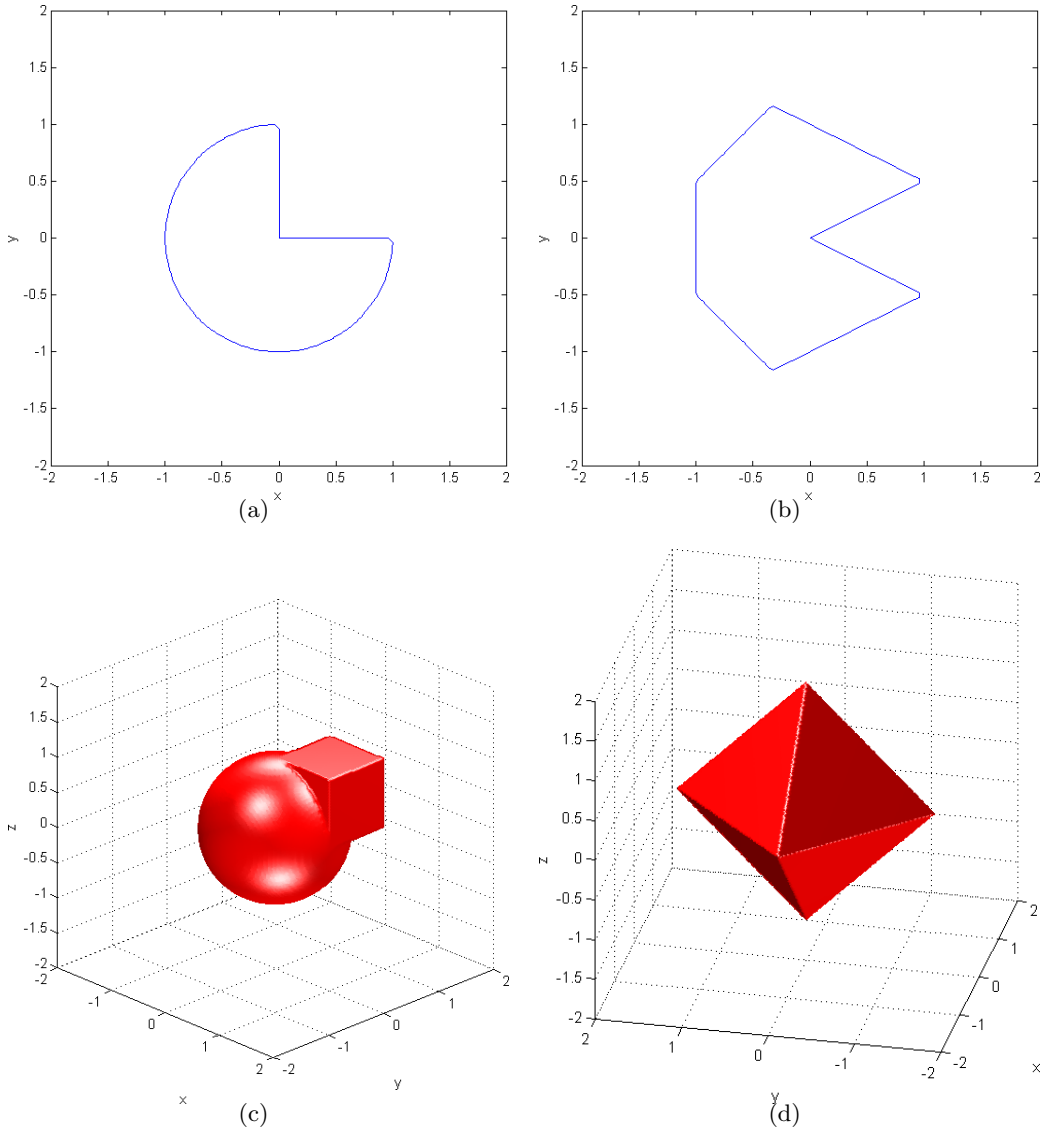


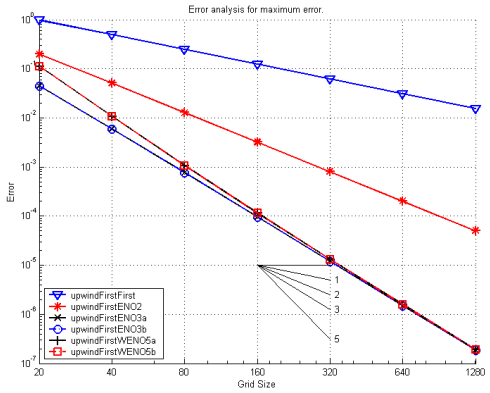
Figure 21: Examples of implicitly defined surfaces and sets built by constructive solid geometry operations from basic shapes. The top row is generated by `initialConditionsTest2D` and the bottom row by `initialConditionsTest3D`. Figure 21(a) shows a square subtracted from a circle, while figure 21(b) shows a nonconvex polygon constructed by intersections and unions of hyperplanes. Figure 21(c) shows the union of a sphere and a cube, and figure 21(d) shows an octohedron constructed by the intersection of eight hyperplanes.

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.

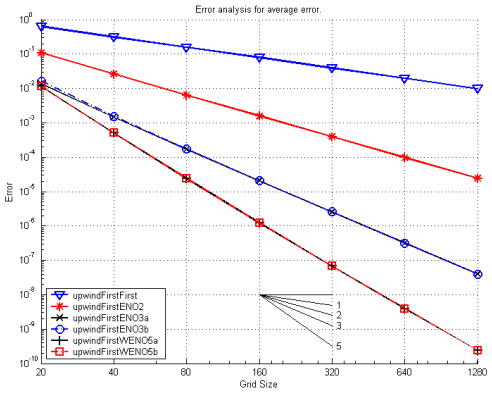
`firstDerivSpatialConverge`: A script file to demonstrate the convergence rate of the various first derivative approximation schemes. The schemes, grid sizes and grid dimensions can be specified inside the script file. The function `firstDerivSpatialTest1` is used to generate the error estimates, although 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 22 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).

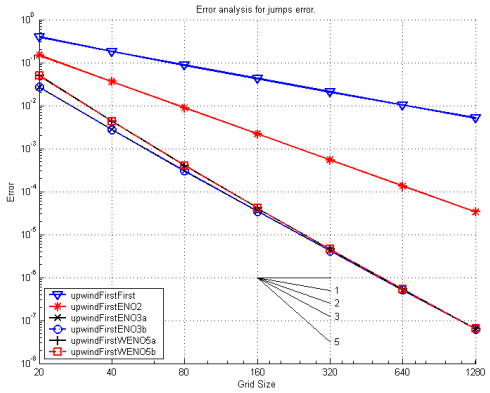
The choice of scheme will be driven primarily by desired accuracy and the need for speed. The relative speeds of the six schemes on the $N = 1280$ grid is shown in table 2, although results will vary depending on the hardware, dimension and grid size. In most simple interface motion examples, the spatial 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.



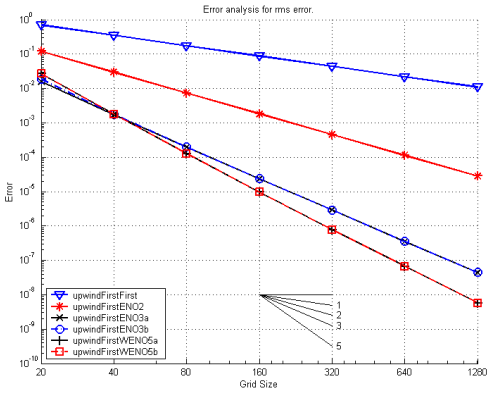
(a) Maximum error.



(b) Average error.



(c) Average error at derivative discontinuities.



(d) Root mean square error.

Figure 22: 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.

2.7.3 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

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

If `data` is a data array defined on `grid`, then `grid.shape == size(data)`.

Notice that manually entering all of these fields 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`. Notice that the `grid.xs` field will generally be much larger than any other, 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 `grid = rmfield(grid, 'xs');` can be used to remove this field and hence enormously reduce the size of the resulting file. The field can be easily regenerated after a load by another call to `processGrid`.

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.

Examination of the code in `shapeHyperplane` provides good evidence of how `cellMatrixMultiply` and `cellMatrixAdd` can be used to simplify spatially 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/Set Operations`.

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 correspondingly 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 (2)–(6) 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 $D^1_{i-1/2}$ and $D^1_{i+1/2}$ entries respectively of the first divided differences of ϕ in dimension d. For more details, see [12, 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 ϕ 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 [12, 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 [12, 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 ϕ is smooth, a fifth order approximation is achieved. In regions where ϕ is not smooth, WENO5 effectively becomes ENO3. For more details, see [12, 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 ϕ. 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 [12, 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 [12, 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 [12, chapter 3.4], although somewhat more difficult to code. The smoothness estimates are constructed from the first divided differences. Several choices of ϵ 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 [12, 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 ϵ term (3.38), all from [12, 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 2.7.2.

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`:

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

where d is the dimension of the grid. The output `curvature` is an array the same size as `data`. For more details, see [12, chapter 1.4] or [15, chapter 6.7]. Since an approximation of the gradient magnitude (using centered first order differences) is computed while finding κ, 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`:

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

where d is the dimension of the grid. The output `laplacian` is an array the same size as `data`. In theory, if ϕ is a signed distance function $\|\nabla\phi\| = 1$, the Laplacian can be substituted for the mean curvature κ, and it is much quicker to calculate. However, since most ϕ 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 [15, chapter 6.7] and the second derivative of curvature (so a fourth order derivative) [15, chapter 14.6].

3.5 Time Derivative Approximations

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

The time derivative (1) is treated by the method of lines. We assume that approximations for all the other terms (2)–(9) 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 (2)–(9) 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 Δt 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: Δt proportional to $(\Delta x)^2$ if there are second order derivative terms (6) or (7), otherwise Δt proportional to Δx. 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 [16] and described in [12, 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 through the `PostTimestep` option discussed in section 3.5.3. The prototype for the function handle `schemeFunc` matches the approximation functions given in section 3.6.

`[ ydot, stepBound ] = schemeFunc(t, y, schemeData)`: Calculate `ydot` $= G(\mathtt{t}, \mathtt{y})$, 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).

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 ϕ 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 (2)–(9) 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 (2)–(7) 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 set using `odeCFLset` in the same manner as MATLAB's `odeset` routine (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.

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

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

The default `[]` indicates that no such routine should be called. If present, this function is called by `odeCFLn` after every full timestep. By modifying `y`, this function can be used to implement constraints of the form (11). 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.

`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 3.7.3.

`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.

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 (2)–(11) 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 (2).

`[ ydot, stepBound ] = 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`. 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 (2)–(4), general first derivative approximations (5), second derivative approximations (6)–(7), and others (8)–(11). 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.

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

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.

3.6.1 Specific Forms of First Derivative

This section discusses functions to approximate the terms which implement convection by a velocity field (2), motion in the normal direction (3), and the reinitialization equation (4). The functions are `termConvection`, `termNormal` and `termReinit` in the directory `Kernel/ExplicitIntegration/Term/`. 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 (5), 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 [12, equation (3.10)])

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

The important fact about this bound is that Δt is proportional to Δx.

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 (2), 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

$$\texttt{v} = \texttt{velocityFunc}(\texttt{t}, \texttt{data}, \texttt{schemeData}),$$

  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 (3). 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 (4). 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. In practice, it is usually used to smooth out excessively steep or shallow gradients in ϕ. The user supplies a copy of the initial conditions $\phi(x, 0)$ (as an array of size `grid.shape`) in `schemeData.initial`. This term will rarely be invoked directly, but will be used indirectly by other routines like `signedDistanceIterative` (see section 3.7.3).

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 (5) 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 [14] and described in [12, 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

$$\hat{H}(x, p^+, p^-) = H\left(x, \frac{p^+ + p^-}{2}\right) - \alpha(x)^T \left[\frac{p^+ - p^-}{2}\right], \tag{25}$$

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 $D_pH(x,p)$, the partial derivative of H with respect to the gradient p. As discussed below, there are several different choices of α 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 (25), 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 $D_pH(x,p)$. 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 α scaling term in (25).

$$\alpha_{\texttt{dim}}(x) = \max_{p \in [\texttt{derivMin}, \texttt{derivMax}]} \left| \frac{\partial H(x,p)}{\partial p_{\texttt{dim}}} \right|. \tag{26}$$

Note that α 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, α 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 (26) 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 (25). 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 [14, 12].

`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 [3]. 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.

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

Once again, Δt is proportional to Δx. Its effect on the choice of CFL restriction reemphasizes the fact that overapproximating α 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 [14, 12]: *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 `termHessian` in the directory `Kernel/ExplicitIntegrati`

The routines for handling terms of the forms (6)–(7) both involve approximations of the second derivative, and both place a stringent bound on the size of explicit timesteps: Δt is proportional to Δx^2. Their `schemeData` structures both require the `schemeData.grid` field, but are otherwise different.

`termCurvature`: Motion by mean curvature (6). The field `schemeData.curvatureFunc` must contain a function handle for a routine that approximates the curvature κ (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 $b(x)$, `b` may be either a scalar (for a constant multiplier) or an array of size `grid.shape` (for a spatially varying multiplier).
- For general multiplier functions, `b` 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`.

Following [12, equation (4.7)], the bound on the timestep is calculated as

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

`termHessian`: Motion by the trace of the Hessian (7). This feature is not yet implemented.

3.6.4 Other Spatial Approximation Terms

Term approximation routines for discounting (8) and forcing (9) have not yet been implemented.

3.6.5 Combining and Restricting Spatial Approximation Terms

This section discusses the functions `termRestrictUpdate` and `termSum` in the directory `Kernel/ExplicitIntegrat`

The term approximation schemes discussed thus far have all dealt with a single term from (2)–(9). In many applications these terms are combined together, or are restricted to a particular sign by constraints of the form (10). 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 (2)–(9).

`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

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

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 (10). 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 $\texttt{stepBound}_{\texttt{sum}}$ is computed from the individual term's timestep restrictions $\texttt{stepBound}_i$ by:

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

Note that `termRestrictUpdate` and `termSum` can be used to wrap each other, and thereby accomplish HJ PDEs more complex than (1)–(10). 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.

The first is an example of the `postTimestepFunc` protocol and implements its most common application, the masking or constraint of ϕ (11).

`[ 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

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

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 remaining 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:

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

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.

3.7.3 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, tMax, errorMax, accuracy)`: Turns an implicit surface function into a signed distance function by iterative solution of the reinitialization PDE (4). 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 `tMax` implicitly defines how many iterations to make, and defaults to a value high enough that the reinitialization front will have reached across the entire grid. 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. 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). 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.

`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 [12, 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 [15].

3.7.4 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 2.7.2 uses `addSlopes` when creating figure 22. 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.

4 Future Features

At the completion of this version of the toolbox, among the extensions which seem useful are:

- Implementation of term approximation routines for motion by the trace of the Hessian (7), discounting (8) and forcing (9).
- 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.
- Implicit time stepping (with MATLAB's ODE suite?).
- Some method to avoid constant reallocation of memory for ghost cells.
- Adaptively refined grids.
- Construction of signed distance functions from point clouds.
- Examples from various application fields.

Do you have any other ideas?

Acknowledgements: The author would like to thank Ronald Fedkiw and Stanley Osher for extensive discussions about the details of numerical schemes for solving the Hamilton-Jacobi PDE. Additional discussions with Jean-Pierre Aubin, David Adalsteinsson, Alexandre Bayen, Doug Enright, L. C. Evans, Chiu-Yen Kao, Alexander Kurzhanski, Doron Levy, John Lygeros, Jianliang Qian, Patrick Saint-Pierre, Jamie Sethian, Steven Ruuth, Pravin Varaiya, Alexander Vladimirsky, Hong-Kai Zhao and many others have contributed to my understanding of the field. Without the support of Claire Tomlin and Shankar Sastry this toolbox would not have been possible. Development and release of the code and documentation for this version is supported by the National Science and Engineering Research Council of Canada and the Department of Computer Science at the University of British Columbia.

References

[1] M. Bardi and I. Capuzzo-Dolcetta. *Optimal Control and Viscosity Solutions of Hamilton-Jacobi-Bellman equations*. Birkhäuser, Boston, 1997.

[2] P. Cardaliaguet, M. Quincampoix, and P. Saint-Pierre. Set-valued numerical analysis for optimal control and differential games. In M. Bardi, T. E. S. Raghavan, and T. Parthasarathy, editors, *Stochastic and Differential Games: Theory and Numerical Methods*, volume 4 of *Annals of International Society of Dynamic Games*, pages 177–247. Birkhäuser, 1999.

[3] M. G. Crandall and P.-L. Lions. Two approximations of solutions of Hamilton-Jacobi equations. *Mathematics of Computation*, 43(167):1–19, 1984.

[4] L. C. Evans. *Partial Differential Equations*. American Mathematical Society, Providence, Rhode Island, 1998.

[5] 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*, 152:457–492, 1999.

[6] Peter E. Kloeden and Eckhard Platen. *Numerical Solution of Stochastic Differential Equations*. Applications of Mathematics. Springer, third edition, 1999.

[7] I. Mitchell, A. Bayen, and C. J. Tomlin. Computing reachable sets for continuous dynamic games using level set methods. Submitted January 2004 to *IEEE Transactions on Automatic Control*.

[8] I. Mitchell and C. Tomlin. Level set methods for computation in hybrid systems. In B. Krogh and N. Lynch, editors, *Hybrid Systems: Computation and Control*, number 1790 in Lecture Notes in Computer Science, pages 310–323. Springer Verlag, 2000.

[9] Ian M. Mitchell. *Application of Level Set Methods to Control and Reachability Problems in Continuous and Hybrid Systems*. PhD thesis, Scientific Computing and Computational Mathematics Program, Stanford University, August 2002.

[10] Bernt Øksendal. *Stochastic Differential Equations: an Introduction with Applications*. Springer, sixth edition, 2003.

[11] S. Osher. A level set formulation for the solution of the Dirichlet problem for Hamilton-Jacobi equations. *SIAM Journal of Mathematical Analysis*, 24(?):1145–1152, 1993.

[12] S. Osher and R. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Springer-Verlag, 2002.

[13] S. Osher and J. A. Sethian. Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics*, 79:12–49, 1988.

[14] S. Osher and Chi-Wang Shu. High-order essentially nonoscillatory schemes for Hamilton-Jacobi equations. *SIAM Journal on Numerical Analysis*, 28(4):907–922, 1991.

[15] J. A. Sethian. *Level Set Methods and Fast Marching Methods.* Cambridge University Press, New York, 1999.

[16] Chi-Wang Shu and Stanley Osher. Efficient implementation of essentially non-oscillatory shock-capturing schemes. *Journal of Computational Physics*, 77:439–471, 1988.

[17] C. Tomlin, J. Lygeros, and S. Sastry. A game theoretic approach to controller design for hybrid systems. *Proceedings of the IEEE*, 88(7):949–970, July 2000.

[18] C. Tomlin, I. Mitchell, A. Bayen, and M. Oishi. Computational techniques for the verification of hybrid systems. *Proceedings of the IEEE*, 91(7):986–1001, July 2003.

Concept Index

Reading the Index: Roman page numbers indicate discussion of the concept or command. Italic page numbers indicate an example of its use.

Command Index

Reading the Index: Roman page numbers indicate discussion of the concept or command. Italic page numbers indicate an example of its use.