Lax-Hopf-based LWR solver

—

Matlab implementation: Manual

Pierre-Emmanuel Mazaré, Christian Claudel, Alexandre Bayen

June 15, 2010

This document describes the sample implementation of an exact, grid-free LWR PDE solver in the MATLAB programming environment. The Lighthill-Whitham-Richards Partial Differential Equation (LWR PDE) is a seminal equation in traffic flow theory. It leads to simple yet widely used traffic flow models for highways. This package proposes a sample implementation for a LWR solver using a new Lax-Hopf method. The LWR PDE is typically solved using the Cell Transmission Model, a Godunov scheme, which requires a grid to compute the solution numerically. This grid-free implementation yields faster computation and exact results. It is publicly available for download at `http://traffic.berkeley.edu/downloads`.

1 Copyright and License

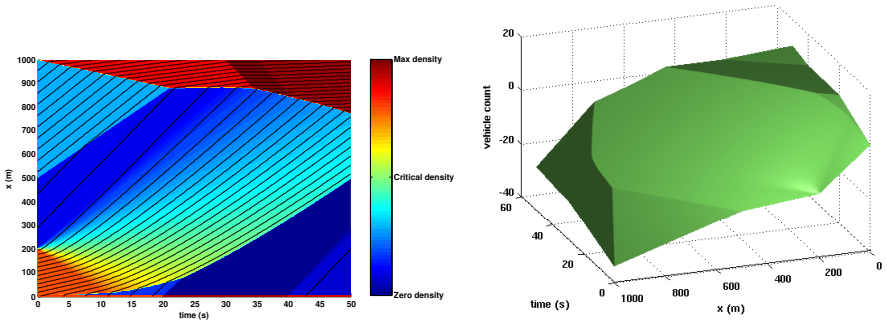

Figure 1: Plots generated by running the file `script.m`. **Left:** 2D representation of the trajectories (black lines), along with a color representation of the density. **Right:** 3D representation of the count function.

CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

2 Publishing results using the software

If you publish results using the MATLAB software, please refer to the following article in your work:

P.-E. MAZARÉ, C. G. CLAUDEL and A. M. BAYEN. Analytical and grid-free solutions to the Lighthill-Whitham-Richards traffic flow model. *Submitted to Transportation Research*, 2010.

The corresponding pdf is included in the zip file which you downloaded.

3 A sample script

This section focuses on a simple example given in the file `script.m` of the package. This MATLAB script calls a preexisting fundamental diagram, uses it to solve a simple initial-boundary condition problem and outputs different plots associated with the solution. The generated plots are shown Fig. 1.

3.1 Using a predefined fundamental diagram

Two general shapes of flux functions are built in the package: a triangular and a Greenshields (parabolic) fundamental diagram. As the solver works only on uniform stretches of road, the fundamental diagram is assumed to remain

the same throughout the problem. The fundamental diagram is created at the beginning of `script.m` and is then stored as an object `fd`.

To instantiate a Greenshields fundamental diagram with free-flow speed `vf` and maximum density `kappa`, one calls `fd=LH_Greenshields(vf,kappa)`. Similarly, to instantiate a triangular fundamental diagram with free-flow speed `vf`, congestion wave speed `w` and maximum density `kappa`, one should call `fd=LH_Tfd(vf,w,kappa)`.

The corresponding class definitions `LH_Greenshields.m` and `LH_Tfd.m` take care of the creation of the full fundamental diagram objects, including explicit formulations for several derived functions, such as the inverse of the fundamental diagram. The fact that fundamental diagrams are objects inheriting from a general fundamental diagram class enables anyone to create new fundamental diagram classes corresponding to different fundamental diagram shapes. See section 4 for more information regarding customized fundamental diagrams.

To define a initial and boundary condition traffic flow problem, one not only needs to define the fundamental diagram, but also the length of the road. The class `LH_general.m` enables the user to encapsulate all this information in a single object we name a *problem environment*. Given a fundamental diagram `fd`, a minimum space index `xmin` and a maximum space index `xmax`, the call `pbEnv=LH_general(fd,xmin,xmax)` creates the corresponding problem environment `pbEnv`.

3.2 Setting initial and boundary conditions

Due to the resolution method, the package only uses the Moskowitz function internally, and not flow or density. The natural way to impose initial and boundary conditions is therefore by assigning the Moskowitz specific values on the boundaries of the domain. Since most data from real-life problems do not come as vehicle count values, helper functions were developed to be able to assign piecewise constant initial densities and upstream and downstream flows to the system.

Given a problem environment `pbEnv`, one can assign initial densities by using the command `pbEnv.setIniDens([x0 x1 ... xn], [k1 k2 ... kn])` where `x0 x1 ... xn` are increasing values, and `k1 k2 ... kn` are within the interval $[0, \kappa]$. This command assigns the constant density `ki` to every point between `x(i-1)` and `xi` at time $t = 0$. Upstream densities can be imposed via `pbEnv.setUsFlows([t0 t1 ... tm], [q1 q2 ... qm])`, with `t0 t1 ... tm` increasing values and `q1 q2 .. qm` are within $[0, q_{\max}]$: in that case, every point belonging to the upstream boundary (`x=xmin`), with time between `t(i-1)` and `ti` is assigned the constant flow `qi`. Similarly, for downstream conditions, the call `pbEnv.setDsFlows([t0 t1 ... tm], [q1 q2 ... qm])` will impose piecewise constant flows on the downstream boundary (`x=xmax`). Note that the time discretizations `[t0 t1 ... tm]` do not need to be identical for upstream and downstream boundary conditions.

An important thing to note about these helper functions is that they need to be called in a particular order: as flows and densities are integrated to become vehicle counts, the system needs to know the constant of integration. If initial conditions are the first ones to be set, the functions will decide of an arbitrary offset (such that the Moskowitz function is equal to 0 in `(0,xmin)`). In that case, the constants for the upstream and downstream conditions are determined

easily by the system. Imposing the value conditions in an order such that the system cannot find the constant of integration deterministically will generate a warning (`warning: could not determine the correct offset for the value condition.`) from the solver and may lead to inconsistent results.

3.3 Solver and output

The solver is divided into two parts: the first one compares the values of all solution components and returns the global solution at any point (t, x), along with the index of the active solution component. Calculating the density then requires an additional step.

The function `pbEnv.explSol(t,x)` calculates the global solution along with the active solution component. It takes advantage of MATLAB vectorization capabilities, which means that `t` and `x` may be vectors, or more generally matrices. These matrices must be the same size, and the solution will be computed on every point `(t(i,j),x(i,j)`.

The output `result` of the call `pbEnv.explSol(t,x)` is a two-by-one cell array. Both elements of `result` are matrices the same size as `t` and `x`. The first element `result{1}` contains the value of the Moskowitz function at every point (or `NaN` when the function is not defined at that point, such as at $t < 0$, before any initial and boundary condition are active), whereas the second element `result{2}` contains the index of the active boundary condition.

Densities on particular components `component` can be calculated by calling `pbEnv.explSol(t,x,component)`. This function will return a density matrix k, also the same size as `t`, `x` and `component`. Flows can then be calculated using the fundamental diagram standard functions: `q=fd.flow(k)`.

Note that, as stated earlier, these functions take advantage of MATLAB vectorization capabilities, and therefore yield optimal performance only when called on matrices. Calling them inside loops may lead to disappointing computation times.

3.4 Plotting tools

The package is provided with two basic plotting functions defined in the files `LH_plot2D.m` and `LH_plot3D.m`. Both take as parameters the variables `t, x, N, k, fd` which were previously defined. The first one displays vehicle trajectories as thin black lines over a colormap of the density. The second one displays the Moskowitz function as a three-dimensional surface. These functions were used to generate plots for our introduction paper, and should be taken as very simple examples of displays that can be generated from the solution of the LWR PDE.

4 Creating customized fundamental diagrams

For various reasons, the MATLAB programming environment is not widely known for its object-oriented capabilities. This package was however implemented following a basic object-oriented design so that users could create their own settings easily, without interfering with the predefined classes we provide.

Fundamental diagrams are defined as classes that all inherit from the generic `LH_fundDiag` class. This inheritance process, noted as `classdef myNewClass <`

`LH_fundDiag`, implies that the new class should *implement* several functions required by the LWR solver. These functions are called `flow`, `wspeed`, `R`, `density` and `densities`. For examples on how to implement these classes, see the provided `LH_Tfd.m` and `LH_Greenshields.m`