Cranfield University

A. Dixon

Development of a Python Traffic Flow Modelling Tool

Applying existing numerical methodologies, suited to hyperbolic PDEs, to traffic flow problems on networks

School of Aerospace, Transport and Manufacturing
Computational Fluid Dynamics

MSc
Academic Year: 2018-2019

Supervisor: Dr P. Tsoutsanis
August, 2019

Cranfield University

School of Aerospace, Transport and Manufacturing
Computational Fluid Dynamics

MSc

Academic Year: 2018-2019

A. Dixon

Development of a Python Traffic Flow Modelling Tool

Applying existing numerical methodologies, suited to hyperbolic PDEs, to traffic flow problems on networks

Supervisor: Dr P. Tsoutsanis
August, 2019

This thesis is submitted in partial fulfilment of the requirements for the degree of Computational Fluid Dynamics MSc.

Abstract

Traffic flow is a complicated system, and hence is difficult to describe mathematically. Understanding traffic dynamics has important applications in economy, the environment, infrastructure and technology. This work aims to investigate the influence of hyperbolic computational fluid dynamics numerical methods on macroscopic traffic flow simulations. The development of a Python tool including such methods has allowed this to be tested. This tool is capable of executing various Riemann solvers, low and high resolution spatial reconstruction, a 4^{th} order Runge-Kutta update, with a probabilistic traffic network model. Results show the numerical methods of choice are more significant for simulations with rapidly changing density in space and or time, high order WENO schemes require bounded limiting to preserve monotonicity, and the VanLeer MUSCL slope limiter is identified to give an optimal solution.

Keywords

Traffic modelling, Macroscopic, Network, Hyperbolic numerical methods, Python developing, WENO, Riemann problem

Acknowledgements

I would like to thank the staff of Cranfield University, School of Aerospace, Transport and Manufacturing, Computational Fluid Dynamics teaching and administration team.

I must thank my parents, Michael and Katie, friends and Cranfield peers, without whom this research and term of study would not have been possible.

Andrew J. Dixon
s290483
andrew.dixon@cranfield.ac.uk

Last edited, Monday 26th August, 2019 at 22:02.

Contents

List of Figures

List of Tables

Abbreviations

Abbreviation	Description
CFD	Computational Fluid Dynamics
TFM	Traffic Flow Modelling
PDE	Partial Differential Equation
CFL	Courant, Friedrichs, Lewy space-time constraint
GK	Gas Kinetic (model)
LWR	Lighthill, Whitham, Richards (model)
CTM	Cell Transmission Model
TDM	Traffic Distribution Matrix
WENO	Weighted Essentially Non-Oscillatory
ENO	Essentially Non-Oscillatory
MUSCL	Monotonic Upwind Scheme for Conservation Laws
RHS	Right Hand Side
TVD	Total Variation Diminishing
RK	Runge, Kutta scheme
FEM	Finite Element Modelling
FDM	Finite Difference Modelling
API	Application Programming Interface
IDE	Integrated Development Environment
GUI	Graphical User Interface
GPS	Global Positioning System
HLL	Harten, Lax, van Leer (Riemann solver)
HLLC	Harten, Lax, van Leer - Contact wave (Riemann solver)

Other abbreviations are otherwise explained when used.

Nomenclature

Symbol	**Description**	**Units**
f	Traffic flow	# cars·Hr^{-1}
ρ_i^n	Density at x_i, t_n	# cars·Km^{-1}
u	Traffic velocity	Km·Hr^{-1}
x_i	Position displacement at cell i	Km
t_n	Time measure at time step n	Hr
i, j	General indexes	-
n	Time index	-
Δt or $\mathrm{d}t$	Time resolution	Hr
Δx or $\mathrm{d}x$	Spatial resolution	Km
CFL	Courant-Friedrichs-Lewy space-time constraint	-
$\frac{\partial()}{\partial()}$	Partial derivative	-
$\frac{\mathrm{d}()}{\mathrm{d}()}$ or $()'$	Full derivative	-
RK_j	Density solution at Runge-Kutta step j	-
$\mathbf{U}$	Conserved vector	-
$\mathbf{F}$	Flux vector	-
Subscript L, R	Left and right state values	-
S^+, S_L, S_R	Local Riemann problem wave speed	-
$a(\rho)$	Local Riemann problem wave speed	-
Δ_i	2^{nd} order reconstruction minmod limiter	-
ϕ	MUSCL slope limiter function	-
k	Size of the $(2k-1)^{th}$ WENO scheme stencil	-
$a_{i,j}$	Traffic distribution matrix elements	-

Other notation are otherwise explained when used.

Chapter 1

Introduction

Traffic flow modelling is not just about understanding traffic dynamics, but traffic movement has significant links to economy, politics and the environment. Traffic and the mass movement of people and goods through cities and countries is a difficult problem to describe mathematically. The following however, indicates the development of research into traffic modelling, and why this field of study is important.

1.1 History

The first known scientific research into traffic flow theory was from Bruce D. Greenshields, who presented a publication [23] detailing his use of revolutionary photographic measurements and conclusions into flow relationships. Greenshields proposed a linear speed-density relationship and the fundamental $f = \rho \cdot u$ traffic relation [31]. Following World War II, the use of automobiles and traffic infrastructure had seen a significant development. The first international symposium on the theory of traffic flow modelling was held in December 1959 at the General Motors research laboratories in Warren, Michigan. The triennial symposia had to separate due to the expansion of the field of study and now many specialist conferences are held covering various elements of traffic flow modelling [17]. See papers from Kuhne [31] and Dhingra and Ishtiyaq [17], from the 75^{th} anniversary Greenshields' symposium for more detail into the origin, history and development of traffic flow modelling.

1.2 Motivation

Traffic as a concept is widely important and hence requires research in order to understand certain processes and control or predict traffic situations. Accurate and reliable traffic flow modelling tools could benefit the environment by reducing unnecessary journey emissions, improve driver and pedestrian safety with well designed traffic intersections and integration with other infrastructure such as train networks and airways. With a recent increase in companies developing smart tools for road vehicles, an appropriate TFM software could assist autonomous driving decisions and GPS satellite navigation routing algorithms. With a better understanding of

realistic traffic behaviour, city councils and infrastructure projects can be developed with more hindsight and improve road lifetimes helping to cut maintenance costs and road tax. Some of these applications are far away, however the tasks performed in this research are sophisticated enough to be able to run scenario tests. Such tests would be useful to prepare for special events that depend on organised traffic, such as the 200,000 visitors arriving at Glastonbury festival from all over the UK, and the contingency plans from Kent county council around the Port of Dover in the event of Brexit[1].

It is important to study hyperbolic PDE systems as compressible flow equations are formulated in this way [65]. Historically, compressible flows are of importance due to the engineering advances in high speed aerodynamics and aerospace [69]. Compressible flow analysis, hyperbolic PDEs in particular, can be used to solve traffic flow problems where cars along a single lane road can experience shocks and other phenomena from compressible fluid dynamics [62]. Many problems arise such as oscillatory solutions due to shock discontinuities.

1.3 Objectives and Structure

Following this introduction to the origin and importance of modelling traffic flows, Chapter 2 gives a review of many recent aspects of modern TFM, including a classification of modelling approaches with examples and previous studies into network flow modelling. The numerical approach is given in Chapter 3, where appropriate and or selected aspects of compressible fluid dynamics are explained, along with details of the integration of a macroscopic hyperbolic PDE solver and a probabilistic network flow model. These techniques are built into a Python modelling tool, of which the development is described in Chapter 4. Here some guidance on preparing and using the tool are given along with a flow chart for steps of the solution process. This tool is implemented on certain problems and results of such are presented and discussed in Chapter 5. Studies of both theoretical and real road networks are completed and analysed for the influence of utilised numerical techniques on solutions, as well as computational time costs. The study concludes with Chapter 6, where the main results, and suggestions for future development of this tool are given. Following this are the cited references, codes used for research[2] are listed for reference in Appendix A, reconstruction processes detailed in Appendix B, and any other supporting material in Appendix C.

[1] See the term *Dover Tap* for the UK government traffic assessment project (TAP) for port-bound vehicles on the A20 approaching Dover, available at gov.uk.

[2] One should clone the GitHub repository for future development.

Chapter 2

Literature Review

The following review both introduces and discusses the existing numerical approach classification, with example methods belonging to these classifications. The original Greenshields traffic flow stream model is discussed along with other more recent models and variations. We review, previous studies into numerical methods applied to network traffic flow problems, existing TFM software, and more general compressible and hyperbolic fluid dynamics numerical methods.

2.1 Numerical Approach Classification

There are many approaches and families of models that have been developed to simulate and predict traffic flow phenomena. Wageningen-Kessels et al. [76] explore the development and classification of traffic flow modelling, their many features and present a useful model tree.

2.1.1 Fundamental Diagram

The traffic flow fundamental diagram shows the relationships between flow rate, density and velocity. This set of three phase-diagrams can indicate conditions of free, bounded, and congested traffic behaviours. A fundamental diagram can also indicate critical values such as the maximum free flow speed, maximum density, maximum flow rate, critical velocity and critical density[1]. Greenshields [23],[31] was the first to investigate traffic flow theory and is hence often called the *founder of.*

2.1.2 Macroscopic

Macroscopic models describe the flow of traffic as a continuum, such as the continuum model for physical fluid flow. Just as the continuum approximation applies to dynamic fluid particles, the movement of cars along road networks can be characterised with continuous approximate variables for density $\rho(x,t)$, velocity $u(x,t)$, and flow $f(x,t)$, all as functions of time and space. The fundamental relationship

[1]Critical velocity and density are such that $u_{crit} = u(\rho_{crit}) = \max_u \{u(\rho)\}$

of these variables is $f(x,t) = \rho(x,t) \cdot u(x,t)$, which can be easily dimensionally verified. The first macroscopic model was introduced by Lighthill and Williams [38], and Richards [52] independently in 1955 and 1956 respectively.

2.1.3 Microscopic

Microscopic models simulate single-vehicle dynamics with sets of variables for position, velocity and acceleration, resulting in continuous systems of differential equations. The most recent development in microscopic TFM are cellular automaton models, where integers describe dynamical variables and road segments are split into cells which are either occupied or not on a binary measure. Cellular automaton models are numerically simple and efficient, hence can simulate large networks quickly, however they lack spatial accuracy over continuous models.

2.1.4 Mesoscopic

Mesoscopic models bridge the gap between micro and macro models with a hybrid approach to describing traffic dynamics. Vehicle behaviour is given on aggregate by probability distributions, and behavioural rules are prescribed to each individual vehicle. The popular mesoscopic approach is to use gas kinetic PDEs to describe the dynamics of probability distributions for traffic flow variables. Newell [46] criticises gas kinetic models by the inability to model non-free-flow traffic conditions. In comparison to macroscopic models, mesoscopic GK models use lots of unknown parameters taken from empirical observations. Large numbers of independent model variables give rise to increasingly complex numerical scheme implementation.

2.2 Numerical Methods

2.2.1 LWR

Lighthill, Whitham [38] and Richards [52] described the first macroscopic model by a conservation law as used in fluid dynamics, also known as the kinematic wave equation. This simple model is derived from the conservation of vehicle numbers on an infinitesimal road segment, hence traffic flow is governed by a first order hyperbolic PDE and the traffic is analogous to an inviscid compressible fluid. As well as the fundamental variable relationship, the LWR model consists of a PDE and a velocity-density relationship. The hyperbolic conservation law which governs the traffic density variation is

$$\frac{\partial}{\partial t}\rho(t,x) + \frac{\partial}{\partial x}f(t,x) = 0, \tag{2.1}$$

and the final piece of the LWR method is a relationship of traffic velocity in density space. This is also called a stream model where $u = u(\rho)$, see Section 2.3 for a review of some common stream models. Using the fundamental relationship, $f = \rho u$ and

stream model, the spatial derivative can be written as $\partial f/\partial x = (u + \rho u')\partial\rho/\partial x$ by the chain rule. The quantity $(u + \rho u')$ is the rate at which information propagates along waves that occur.

2.2.2 Nagel-Schreckenberg

Nagel and Schreckenberg present a discrete boolean cellular automaton model [45] which tracks each vehicle's individual dynamics explicitly and is hence within the microscopic TFM family. The model is defined by four simple steps that act on a array which splits a road segment into cell sites of occupied or empty cells. each vehicle has its own velocity and location, described by integer quantities. The four steps form a single iteration which acts in parallel on all vehicles of the system,

1. *Acceleration*: *if* the velocity u of a vehicle is less than the speed limit $u_{\max}$ *and* the distance to the next car ahead is larger than $u + 1$ *then* increase the speed by one $(u \leftarrow u + 1)$

2. *Deceleration*: *if* if a vehicle at cell site i sees the next vehicle ahead at site $i + j$ *then* it reduces its speed to $j - 1$ $(u \leftarrow j - 1)$

3. *Randomisation*: with probability p, the velocity of each vehicle (with $u > 0$) is decreased by one $(u \leftarrow u - 1)$

4. *Vehicle motion*: each vehicle advances by u cell sites

This algorithm captures general properties of single lane traffic flow, Nagel and Schreckenberg were able to show non-trivial realistic flow phenomena with these steps alone. The key to realistic flow simulation is held in the random choice of step 3, without this the dynamics are completely deterministic. The parameters for this model are calibrated with reasonable rough arguments and traffic measurements. The original paper shows the computational advantages of this model, and the realisation of important flow aspects such as the transition of laminar flow to stop-start traffic. See Appendix C.2 for some simple results of this algorithm, and the simulation code in Appendix A.3.1.

2.2.3 Payne High Order

Payne's macroscopic approach [48] is an extension of the LWR model, a second order method in which Payne uses an extra PDE to govern the average speed variable. Hence there is no need for a stream model, as the velocity is described by

$$\frac{\partial u}{\partial t} + \underbrace{u\frac{\partial u}{\partial x}}_{convection} = \underbrace{\frac{U^e(\rho) - u}{T}}_{relaxation} - \underbrace{\frac{c_0^2}{\rho}\frac{\partial \rho}{\partial x}}_{anticipation}, \tag{2.2}$$

where U^e is the average equilibrium speed, c_0 is the anticipation constant, and T is the relaxation constant. Payne found that, from observations, average speed is

dependent on the state of neighbouring road sections as well as the local density. The three major influences of average speed dynamics are convection, relaxation and anticipation. The convection term is proportional to the average velocity change in space due to a *gradual* acceleration/deceleration, and to the local average speed. Relaxation describes the tending to an equilibrium speed for all drivers. The anticipation term explains how drivers will anticipate a traffic jam they can see ahead and slow down prior to this. This model is discussed in detail in a review of macroscopic models from Bellemans, De Schutter and De Moor [7], where the previously given terms are discussed in more detail.

2.2.4 Cell Transmission Model

Daganzo proposed a numerical method to solve the LWR kinematic wave equation, where a road is partitioned into homogeneous cells of length equal to the distance travelled by a typical vehicle in one time step. This model assumes vehicles advance to the next cell with each time step, and tracks the transmission of cars through these cells with $n_i(t)$, the number of cars in cell i at time t. Daganzo formulates the CTM model [15] by the flow of cars as follows,

$$n_i(t+1) = n_i(t) + \underbrace{y_i(t)}_{cars\,in} - \underbrace{y_{i+1}(t)}_{cars\,out} \tag{2.3}$$

where the respective flows, $y_i(t)$ of cars *into* cell i at time t, are calculated from

$$y_i(t) = \min\left\{n_{i-1}(t), Q_i(t), N_i(t) - n_i(t)\right\}. \tag{2.4}$$

In this formulation, $Q_i(t)$ is the maximum flow capacity into cell i at time t, and $N_i(t)$ the maximum occupying capacity of cell i at time t. Each term in the minimum statement of Equation 2.4 ensures that the cell transmission flow $y_i(t)$ is realistic and bounded. The flow cannot be larger than the upstream neighbouring cell population $n_{i-1}(t)$, or larger than the flow capacity $Q_i(t)$, nor can the flow be such to overfill the remaining 'empty space' $N_i(t) - n_i(t)$ in cell i. Daganzo further shows that this formulation is consistent with the LWR hydrodynamic model; highway characteristics are independent of space and time due to the assumed homogeneity, and the LWR conservation equation reduces to Equation 2.3. The model form offers four degrees of freedom, free flow speed, maximum flow and density, and the wave speed. As stated by Newell [46], these are the most important parameters for a realistic flow model.

The most recent development of this model is the multi lane CTM from Laval and Daganzo [35]. In this development, each lane satisfies the LWR kinetic wave equation with a lane changing rate source term. The lane changing action is treated as discrete particles which create temporary lane blockages with a finite acceleration, this improves on existing lane changing models that have unrealistic instantaneous accelerating lane changes.

2.2.5 More Macroscopic Models

There are many models for each approach to TFM, Wageningen-Kessels et al. [76] (See Appendix C.2) show the relationships between each developed model. The following are some more recently developed macroscopic models, which are derived and built on earlier research approaches.

Multi Class LWR

Hoogendoorn and Bovy [26] developed a multi class model by applying the gas-kinetic approach to derive continuum macroscopic traffic models. They describe different vehicle classes, each with different desired speed behaviours within in the same density. Each class is represented by their own variables for flow, density and speed, and each follow the LWR model and fundamental flow relationship with their own stream model. Systems of relations are solved with the LWR framework over each class set of variables.

Anisotropic High Order

Payne's high order [48] (See Section 2.2.3) is developed by Aw and Rascle [4]. By using a convective derivative, previous non-physical effects of the Payne model are resolved. This model is shown to perform well in predicting instabilities in very light traffic.

Hybrid High Order CF

Moutari and Rascle [43] develop the Aw and Rascle [4] model into a Lagrangian description which simultaneously solves the microscopic and macroscopic discretisations. The hybrid aspect uses the Aw-Rascle and car following approaches, which allows traffic dynamics to be captured over a large network, still resolving small details in sensitive regions. This model achieves TVD with respect to the space and time for the velocity.

2.3 Stream Models

Under free flow conditions, the three traffic variables are pairwise related and explicitly given by the following models in this section. Reviews from [3], [63], [40], [64] outline each of the following models while commenting on the goodness of fit to empirical traffic data. Each model has advantages and disadvantages over another, for better explaining traffic flow phenomena. The explicit formulations the velocity under each model are given in Table 2.1. Other approaches are multi-regime models where a set of models provide a piecewise description of the traffic flow variables at different ranges of densities, field observations show human behaviour varies for flow at different densities [64].

Table 2.1: Named traffic stream model equations where u_f is the free ($\rho = 0$) speed, ρ_m the maximum *jam* density, u_c the capacity velocity, $\overline{\rho}_{\min}$ the non-zero average minimum density, and b, c general parameters.

Name	Expression for $u(\rho)$
Greenshield	$u_f \left(1 - \frac{\rho}{\rho_m}\right)$
Greenberg	$u_c \ln\left(\frac{\rho_m}{\rho}\right)$
Modified Greenberg	$u_c \ln\left(\frac{\rho_m + \overline{\rho}_{\min}}{\rho + \overline{\rho}_{\min}}\right)$
Underwood	$u_f \exp\left(\frac{-\rho}{\rho_c}\right)$
Bell-Shaped	$u_f \exp\left(\frac{1}{2}\left(\frac{\rho}{\rho_c}\right)^2\right)$
Pipes-Munjal	$u_f \left(1 - \left(\frac{\rho}{\rho_m}\right)^n\right)$
Polynomial	$u_f + b\rho + c\rho^2$
Quadratic	$u_f \left(1 - \frac{\rho^2}{\rho_m^2}\right)$

Greenshields

After traffic observations in 1935, Greenshields proposed the linear velocity-density relationship, parabolic flow-density and flow-speed relationships [23]. The simple model satisfies the conditions of stationary traffic at jam density, and maximum speed at zero density. Due to its simplicity this model rarely fits real data well, the performance at density boundaries does not fit well in the above research reviews.

Greenberg

Using a fluid flow analogy and data from Lincoln Tunnel, New York, Greenberg proposed this logarithmic relation. In his 1959 paper [22], Greenberg provides an analytical derivation from fluid dynamics equations with traffic flow notation to a simple PDE solution. Not suitable for low concentration flow ($u \to \infty$), however this model fits empirical data well for high density congested conditions. To address the low-density flaw of the original model, Ardekani and Ghandehari [2] introduced the non-zero average minimum density, $\overline{\rho}_{\min}$, which means the speed at low density is $u_c \ln\left((\rho_m + \overline{\rho}_{\min})/\overline{\rho}_{\min}\right)$ not ∞ as in the classical Greenberg model.

Underwood

Underwood proposed an exponential model [70], to improve on the Greenshields model which is shown to be true in the above reviews. As an exponential model, the high-density boundary condition is not met, however still performs well when compared to empirical data.

Bell-Shaped (Drake)

Being unimpressed after studying various available traffic models, Drake [19] formulated his own model by transforming an estimated speed-density relation to a speed-flow function. This model is generally a better fit than the Underwood, Greenberg and Greenshields models, however the Underwood is better for congested conditions.

Pipes-Munjal

An alternate approach, Pipes proposed the family of models with parameter n [50], for $n = 1$ this is equivalent to the Greenshields model.

Polynomial, Quadratic and Taylor Expansions

Polynomial models are in a general form defined by two parameters b and c, for $b = 0$ and $c = -1/\rho_m^2$ this gives the quadratic form of the model. These models are found to give realistic results for free flow and congested conditions. By truncating a Taylor expansion of any exponential stream model, the jam density can be found by solving polynomials in ρ for $u = 0$.

2.4 Previous Numerical Studies

Of the recent research into numerical traffic flow simulations, the following apply a fluid dynamics continuum approach. Research into the interaction of this conservative model approach to TFM with road networks as a graph with edges (roads) and nodes (junctions) is presented by Bretti, Natalini and Piccoli [9], and Shi and Guo [58]. Both studies use a similar approach to the definition of road networks, the general junction has a traffic distribution matrix which defines probabilities of flow leaving outgoing roads. Several test cases are selected from Bretti, Natalini and Piccoli's earlier study [8]. Shi-Guo [58] use a third order stability preserving Runge-Kutta time discretisation, obtaining results which satisfy the maximum principle and preserve the 5^{th} order WENO accuracy.

Similar research from [33], and [57] using the LWR [38],[52] model investigate numerical methods abilities to predict shock structures along a single road segment, with applied results for traffic features such as traffic stop/go lights. Lakhanpal recommends a finite element approach, over Godunov, for linear advection and traffic light problems. While FEM introduces oscillations in the presence of shocks, added time relaxation suppresses these [33]. Setiawan, Tarwidi and Umbara successfully implement a finite volume method [57], and are able to control traffic movement by adjusting traffic light timings.

2.5 Software Review

The following comments are a collection from three reviews of traffic flow simulation systems, refer to the reviews for a more in depth analysis of each system and more not mentioned here. Maciejewski [41] compares three specifically microscopic systems on urban roads, results from real road network simulations with TRANSIMS, SUMO and VISSIM systems are discussed. Pell, Meingast and Schauer [49] present the results of an online survey of managers and developers, and gives comments on 17 different TFM software tools from various interviews, for example with traffic planners and PhD candidates. Pell et al., state that no tools are complete with all functionalities, and no system focuses on a single traffic application. Saidallah, El Fergougui and Elalaoui [56] analyse 11 tools each compared over 19 characteristics, assessing the use in planned changes for road networks.

SUMO

Developed by German Aerospace Centre DLR, SUMO (Simulation for Urban MObility) is a free, microscopic system with many available models for example safe distance car following, and lane changing. The space continuous, time discrete system is capable of modelling up to four vehicle types, intersections with or without traffic lights, large networks of over 10,000 links, collisions and accidents, dynamic vehicle routing, public transport and even pedestrians. Extra add ons are available including a 2D graphical visualisation of simulation results and APIs to remotely control simulations.

TrAnSimS

The TRansportation ANalysis and SIMulation System is another free, microscopic tool which boasts its ability to model regional scale transport systems, developed at Los Alamos National Laboratory USA. The system runs each iteration to equilibrium according to the Wardrops first principle [1]. The whole system is a conglomeration of many modules, including a classical traffic microsimulator based on cellular automata theory and the Nagel-Schreckenberg model which governs car following and lane changing. This system has been successfully tested with data from roads in Dallas, Texas and Portland (Oregon).

PTV Vissim

VISSIM[2] is a commercial software that is delivered as a custom tool from PTV depending on their customer requirements. VISSIM models difficult junctions well such as roundabouts which pose difficulties for the classical node-connector definition

[1] Wardrop's first principle: journey times on used travel routes are less than or equal to any journey time for unused travel routes.

[2] VISSIM stands for Verkehr In Stadten - SIMulationsmodell, which is German for 'Traffic in cities - simulation model'

of a road network. The car following model takes psycho-physical driver behaviour into consideration. As a paid commercial software one can expect impressive features, VISSIM simulates city-like processes with multiclass road vehicles, trams and pedestrians with huge impressive 2D and 3D graphics capabilities in high detail. Customers are also able to implement extra user defined functionality though a C++ interface. Multi modal software is multi transport type (road vehicle, pedestrian, public transport), multi class software allows a sub-mode type (vehicle-cars, vehicle-motorbike, pedestrian-old person, pedestrian-child, public transport-tram, public transport-bus), VISSIM is both multi modal *and* multi class.

Other Reviewed Systems

MATSim is used for very large simulations, tested on roads in Zurich, Berlin, Padang and Toronto. Traffic routes are determined by an activity based agent demand generation, rather than typical origin-destination matrices for dynamic assignment.

MiTSimLab analyses the impact of alternate traffic management systems, public transport operations and intelligent transport systems. Widely popular as an open source C++ program, this has been widely and successfully applied in USA, UK, Sweden, Italy, Switzerland, Japan, Korea, Malaysia and Portugal.

Aimsun is able to reproduce real traffic conditions, testing and developing traffic control systems, toll locations and public transport networks. This tool also allows multi network simulation for efficient road network testing.

CorSim is a simulation software mainly used for signal systems, road networks and highway networks. NETSIM and FRESIM represent the environment of traffic on city networks and freeway roads respectively.

The microscopic ParaMicS tool can be scaled for use on single intersections or full city traffic simulations with 2D and 3D visualisation. Able to simulate buses, trams and pedestrians, traffic moves by an origin-destination matrix along a network defined by nodes and connectors on a graph. This has previously used to simulate vehicle movement and predict future traffic implications of proposed infrastructure features.

2.6 Compressible Fluid Dynamics Review

CFD textbooks tend to focus on a few areas or a particular application, the following are useful for understanding certain topics. Variable changes through different types of waves are consistently explained in [1], [34], [47] and [62]. The use of CFD in industry is the focus of [27], with considerable detail of Godunov upwinding. A gas dynamics text from Laney [34] discusses Riemann problems and the approximation of fluxes due to the expensive computation of analytical Riemann problem solutions.

Toro's book [65] formulates the hyperbolic-conservative Euler equations and introduces the requirements of a Riemann problem, and its solution for the accurate flow solution over finite difference cell interfaces. Hyperbolic systems are discussed as difficulties in numerical discretisation are imposed by hyperbolic terms in a PDE. Toro presents Godunovs approach to the solution of conservation laws, along with the HLL Riemann solver and the developed variant HLLC also developed by Toro et al. [66]. The attempt of an encyclopaedic cover of CFD is made by Chung [11], where Godunov's approach is given and the finite difference approach is scrutinised. Chung discusses the disadvantages of simple solver methods, leading to the necessity of higher resolution schemes along with some properties.

Godunov [21] describes the pitfalls of the method of characteristics for numerical fluid simulations, especially for compressible dynamics, and outlines his new method which is described in Section 3.2. Lax presents a flux calculation [36] with the use of conservative form of hydrodynamic equations and a novel differencing method. Rusanov [55] developed a method to improve on Godunov's ideas, a finite difference *through* method which shock capturing ignores discontinuity locations within calculations. To improve on Lax-Wendroff type methods, van Leer [74] developed MUSCL schemes to remove oscillatory solutions with nonlinear instabilities. In a numerical review from Woodward and Colella [78], MUSCL is shown to out perform Godunov methods. A review of Godunov's methods by Quirk [51], explains how Godunov's flaws have gone un-noticed as some errors occur in very high-resolution simulations. Quirk concludes that hybrid Riemann solvers are a better method than artificial dissipation which introduces inaccuracy into a solution. Methods for capturing shock discontinuities developed by proposing shock structures and deriving equations to support, such as the expansion-shock and expansion-contact-shock form of the HLL and HLLC approximate Riemann solvers respectively. Toro developed the HLLC solver from HLL [66], and when HLLC is used with a Godunov-type method, the HLLC outperforms the HLL and (in the Mach reflection case) is virtually identical to the exact Riemann solution. Most approximate Riemann solvers make use of the wave signal velocity bounds, many wave speed estimates are proposed and investigated by Davis [16]. To improve the accuracy of a scheme without introducing dissipative errors, high order schemes are developed. Liu et. al [39] proposed the WENO scheme as a development from the ENO reconstruction method, Liu's results show the new scheme converges to *analytical*[1] solutions. New developments of high resolution schemes include Suresh and Huynh's 5^{th}-Order-Monotonicity-Preserving

[1] Analytical solutions are assumed from Lax-Friedrichs simulations on a very fine grid.

method [60], which is shown in results to resolve discontinuities at high resolution while also being accurate for smooth regions.

To retain a scheme's monotonicity yet eliminate the need for artificial dissipation, flux gradient limiters are used. Barth and Jespersen introduced a gradient limiter for unstructured grids [6], aiming to obtain higher order accuracy by ensuring no new extrema are created during reconstruction. Venkatakrishnan developed Barth and Jespersen's limiter with a continuous approximation function [75]. The Venkatakrishnan limiter reduces to first order accuracy at local extrema. A downfall of second order schemes with gradient limiters are the reduction in order due to the smoothing properties of slope limiters.

The field of methods for compressible dynamics spreads over many areas of fluid dynamics and numerical solutions of PDEs. Cockburn and Shu [12] review Runge-Kutta finite element methods which incorporate compressible and high-resolution-finite-difference methods such as fluxes and slope limiters. A method using discontinuous Galerkin spatial discretisation, Runge-Kutta time stepping, and a slope limiter, is described and found that using higher order reconstruction polynomials increases the resolution of captured discontinuities and increases efficiency in smooth regions.

Chapter 3

Approach

The LWR equation (2.1) is already presented in conservative form as a hyperbolic PDE [65]. The following present this in the framework of Euler gas dynamics equations governing this compressible flow,

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}(\mathbf{U})}{\partial x} = 0, \quad with \quad \mathbf{U} = [\rho], \quad and \quad \mathbf{F} = [\rho u], \tag{3.1}$$

where $\mathbf{U}$ represents the conserved density ρ. In the program code, the conserved state, $\mathbf{U}$, are known and are used with the fundamental flow relationship to calculate the flux, $\mathbf{F}$, hence $\mathbf{F} = \mathbf{F}(\mathbf{U})$. From this formulation, if $\mathbf{U}$ is discretised over the current time, n, and the next time step $n+1$, and $\mathbf{F}$ over left and right cell interfaces, $\mathbf{F}_{i-1/2}$ and $\mathbf{F}_{i+1/2}$ respectively, then an update scheme is

$$\mathbf{U}_i^{n+1} = \mathbf{U}_i^n - \frac{\Delta t}{\Delta x}\left(\mathbf{F}_{i+1/2} - \mathbf{F}_{i-1/2}\right).$$

The values of $\mathbf{F}_{i+1/2} - \mathbf{F}_{i-1/2}$ introduce a difficulty, finite difference approximations construct a piecewise continuous solution profile as shown in Figure 3.1, so interface values are discontinuous and need alternative treatment. To solve $\mathbf{U}$ over cell interface discontinuities, the local Riemann problem is constructed for Equation 3.1 with the initial conditions

$$\mathbf{U}(x, t=0) = \begin{cases} \mathbf{U}_L, & if \quad x < 0, \\ \mathbf{U}_R, & if \quad x > 0, \end{cases} \tag{3.2}$$

where the variables x and t are mapped in the following way

$$(x_{i-1/2}, x_{i+1/2}) \mapsto (-\Delta x/2, \Delta x/2), \quad and \quad (t_n, t_{n+1}) \mapsto (0, \Delta t). \tag{3.3}$$

The following sections discuss more numerical approaches and some Riemann solving methods for the problem in Equations 3.1 and 3.2. This governing formulation is shown in more detail in [11],[65],[67], while many of the following details can be found in [68].

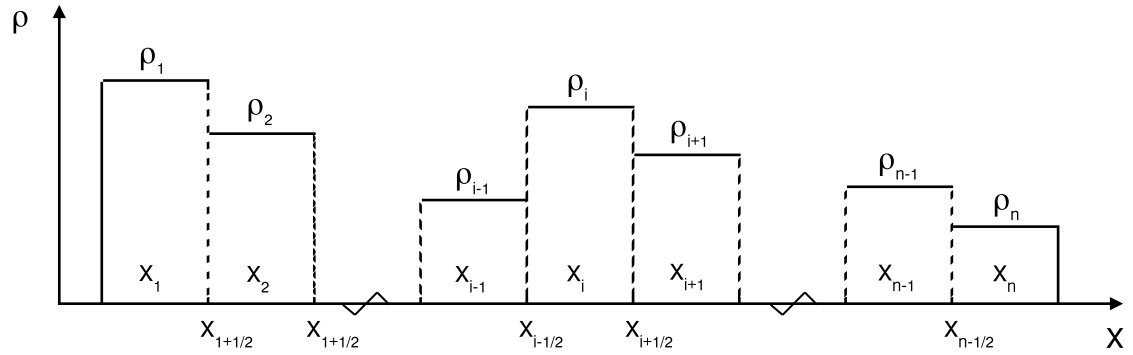


Figure 3.1: Discretisation of a 1D spatial domain for a single time. For $i \in \mathbb{Z}$, x_i represent the control volumes where single Godunov averaged solutions are stored, ρ_i. The cell interfaces are represented at locations $x_{i\pm1/2}$; as single solutions are stored in one cell, there are discontinuities at each cell interface location.

3.1 Compressible Flow Waves

Compressible dynamics numerical methods have been developed to allow flow variables to exhibit discontinuous derivatives in space, these variable jumps are due to the presence of flow waves. There are four types of compressible waves discussed here; normal shock, contact surface, rarefaction, and compression waves. When a region of high pressure and density is separated by a diaphragm from a region of low pressure and density[1], some of these waves occur.

Normal shock waves, such as the shock present on the upper surface of a transonic aerofoil, are normal to a surface. Contact surfaces propagate through a dynamic fluid as the discontinuous interface between two materials. Rarefaction waves represent gradual longitudinal expansion, propagating through the flow. Compression is the opposite of rarefaction, where a compression of characteristics move with the flow.

3.2 Godunov-Type Methods

The Godunov method assumes piecewise constant solution profiles (see Figure 3.1), with discontinuities along cell interfaces that induce many local Riemann problems [65]. Godunov's method improves on central-based schemes by having the capability to distinguish between compression and expansion fan waves [27]. Godunov [21] suggested the following method for computing the interface flux approximations:

1. Construct two *local* Riemann problems over a data pairs (ρ^n_{i-1}, ρ^n_i) and (ρ^n_i, ρ^n_{i+1}),
2. Average the two solutions over $[x_{i-1/2}, x_{i+1/2}]$,
3. Assign a value for ρ^{n+1}_i,

[1]Such as the initial conditions for 1D shock tube and 2D explosion case.

4. Then the Godunov flux is approximated by

$$\mathbf{F}_{i\pm1/2} = \mathbf{F}\left(\mathbf{U}_{i\pm1/2}\right).$$

The two Riemann problems are formulated locally (by scaling the variables as in Equation 3.3) with the equation $\rho_t + f(\rho)_x = 0$, and each set of boundary conditions

$$(\rho^n_{i-1}, \rho^n_i) : \quad \rho(x,0) = \begin{cases} \rho^n_{i-1}, & if \quad x < 0, \\ \rho^n_i, & if \quad x > 0, \end{cases}$$

$$(\rho^n_i, \rho^n_{i+1}) : \quad \rho(x,0) = \begin{cases} \rho^n_i, & if \quad x < 0, \\ \rho^n_{i+1}, & if \quad x > 0. \end{cases}$$

The cell averaging for ρ^{n+1}_i is taken over the cell width, however with the locally formulated Riemann problems this is over $[-\Delta x/2, \Delta x/2]$, for the two Riemann problem solutions $\tilde{\rho}_{i-1/2}$ and $\tilde{\rho}_{i+1/2}$. The new solution is given by

$$\Delta x \cdot \rho^{n+1}_i = \int_{-\Delta x/2}^{0} \tilde{\rho}_{i-1/2} \mathrm{d}x + \int_{0}^{\Delta x/2} \tilde{\rho}_{i+1/2} \mathrm{d}x \tag{3.4}$$

Fluid dynamics problems are usually formulated as a combination of varying PDEs, solutions to which are highly sensitive to numerical methods [11]. Godunov schemes are useful when applied to hyperbolic systems, however have limitations for other types of PDEs. Elliptical problems have no real characteristic curves so flow variable derivatives are smooth with no discontinuities, hence the Godunov method and local Riemann problems are not useful [11]. The major disadvantage of using Godunov schemes with Riemann solvers for shock capturing flow is the extra computational cost over a second order scheme with artificial viscosity [78]. Riemann solvers become complicated to implement if an equation of state cannot be represented with a gamma law. Analytical solutions of Riemann problems exist for the Euler equations however are computationally expensive, hence the Godunov local Riemann problems are solved using approximate methods shown in Section 3.3.

3.3 Approximate Riemann Solvers

The following methods for finding approximate solutions to local Riemann problems at each cell interface are presented for interest, only the Lax-Friedrichs, Rusanov, HLL and Murman-Roe solvers are included in the TFM program code, see Appendix A.1.1 lines [394-470]. As shown in Chapter 5, one of the flaws of this model is the ill defined Riemann solvers. An appropriate definition of parameters involved in the following methods is needed to yield a more realistic solution to the cell interfaces. This process in the simulation is known both as the approximate local Riemann problem solution and the calculation of numerical fluxes. These local problems are too computationally costly to apply exact Riemann solvers, hence approximate methods are used to cut this cost [34].

3.3.1 Rusanov and Lax-Friedrichs Flux

The Rusanov [55] and Lax-Friedrichs [36] fluxes can both be written in the form

$$\mathbf{F} = \frac{1}{2}\left[(\mathbf{F}_L + \mathbf{F}_R) - S^+ (\mathbf{U}_R - \mathbf{U}_L)\right], \tag{3.5}$$

where the wave speed S^+ is calculated from local data in the Riemann problem, where both

$$S^+ = \max\left(|f'(u_L)|, |f'(u_R)|\right), \tag{3.6}$$

$$S^+ = \frac{\Delta x}{\Delta t}, \tag{3.7}$$

represent the maximum wave speed for the Rusanov (Equation 3.6) and Lax-Friedrichs (Equation 3.7) fluxes. These solvers can be found in Appendix A.1.1 lines [412-430], where the left and right state derivatives (Rusanov) are calculated from the chosen stream model.

3.3.2 Murman-Roe

The popular Roe solver defined for a scalar system is named the Murman-Roe solver [44]. Murman defines the wave velocity a as the Rankine-Hugoniot velocity,

$$a(\rho_L, \rho_R) = \frac{f_L - f_R}{\rho_L - \rho_R}, \tag{3.8}$$

and flux,

$$f(\rho_L, \rho_R) = \frac{1}{2}(f_L + f_R) - |a(\rho_L, \rho_R)|(\rho_R - \rho_L), \tag{3.9}$$

if $\rho_L \neq \rho_R$. However if $\rho_L = \rho_R$ then the velocity is defined as,

$$a(\rho_L, \rho_R) = f'_L = f'_R, \tag{3.10}$$

and the flux depends on this velocity as follows,

$$f(\rho_L, \rho_R) = \begin{cases} f_L, & \text{if} \quad a(\rho_L, \rho_R) > 0, \\ f_R, & \text{if} \quad a(\rho_L, \rho_R) \leq 0. \end{cases} \tag{3.11}$$

3.3.3 HLL

Harten, Lax and van Leer [25] suggested a Riemann solver which assumes a wave formation of two waves, however this is incorrect for the Euler equations and only holds for two equation hyperbolic systems [65]. Hence the integral-form conservation equations are split over three regions, left and right states and a single *star* region. Depending on the choice of left and right wave speed values, S_L and S_R, the HLL

flux is given by

$$\mathbf{F} = \begin{cases} \mathbf{F}_L, & if \quad 0 \leq S_L, \\ \mathbf{F}_M = \frac{S_R\mathbf{F}_L - S_L\mathbf{F}_R + S_L S_R(\mathbf{U}_R - \mathbf{U}_L)}{S_R - S_L}, & if \quad S_L \leq 0 \leq S_R, \\ \mathbf{F}_R, & if \quad S_R \leq 0. \end{cases}$$

As well as this formulation, Toro et al. [67] outline the problem with this Riemann solver. Namely the difficulty in finding reliable and simple estimates for the left and right wave speeds. Toro developed the HLL solver further, details of the HLLC Riemann solver are given in Appendix C.1.

3.3.4 Wave Speed Estimations

All previous Riemann solvers have depended on a set of parameters defining the behaviour of each Riemann solver, including the wave speed values of S_L, S_R, S_*. Davis [16] suggested a set of simple estimates for Riemann solvers in compressible gas dynamics where the quantity a represents the local speed of sound. Toro showed that these estimates are however impractical for computations [65],

$$S_L = u_L - a_L, \quad S_R = u_R + a_R,$$

and

$$S_L = \min\left(u_L - a_L, u_R - a_R\right), \quad S_R = \max\left(u_L + a_L, u_R + a_R\right).$$

Davis [16] also uses the Roe-averaged eigenvalues

$$S_L = \tilde{u} - \tilde{a}, \quad and \quad S_R = \tilde{u} + \tilde{a},$$

with Roe-averaged speeds denoted by $\sim$, which lead to a much more effective scheme. Davis also recognised how the Rusanov flux (Equation 3.5) can be recovered from the HLL formulation by setting $S_L = -S^+$ and $S_R = S^+$ for some choice of S^+ mentioned in Section 3.3.1. This aspect of the relationship between solving local Riemann problems in cell reconstruction in TFM is not equivalent to other fields such as gas dynamics, appropriate parameters to describe the local Riemann problems need to be established for the solvers described here to be most effective.

3.4 Spatial Reconstruction

Prior to providing left and right states to a chosen Riemann solver to evaluate the numerical flux and proceed with the iteration, one can reconstruct the cell variable value in many ways. The most simple approach is named 1^{st} order as the values provided for numerical flux calculations are in fact the cell average Godunov values themselves with no reconstruction applied. The next improvement is the 2^{nd} order total variation diminishing scheme. At each cell interface (cell i, left $i - 1/2$, right

$i + 1/2$), the left and right densities are 2^{nd} order TVD reconstructed according to

$$\rho_L = \rho_i + \frac{\Delta_i}{2}, \quad and \quad \rho_R = \rho_{i+1} - \frac{\Delta_{i+1}}{2}, \tag{3.12}$$

$$\Delta_i = \text{minmod}\left(\rho_i - \rho_{i-1}, \rho_{i+1} - \rho_i\right),$$

$$\text{minmod}\left(x, y\right) = \frac{1}{2}\left(\text{sign}\left(x\right) + \text{sign}\left(y\right)\right)\min\left(\left|x\right|, \left|y\right|\right), \tag{3.13}$$

where the slope limit Δ_i is calculated from the minmod limiter function.

3.4.1 High Resolution Schemes

High resolution numerical schemes are used when fluid problems involving shocks and discontinuities are of interest with high accuracy, high resolution methods reduce oscillations to provide monotone solutions [11]. Monotonicity preserving schemes are at most first order accurate according to Godunov's theory, where higher order schemes introduce oscillations around discontinuities [12]. Alternative to the uniform distribution over cells in Godunov's method, the MUSCL scheme which uses linear reconstructions of Godunov cell data. When accuracy greater than second order is required, WENO schemes provide higher accuracy. See Figure 3.2 for the comparison of a 1D problem solution using various discussed schemes.

The first WENO scheme was proposed by Liu, Osher and Chan in 1994 [39], a novel ENO method with a higher order reconstruction. Where the ENO method chooses the smoothest interpolating polynomial, the weighted scheme uses a convex combination of all polynomials with weights specifically chosen to improve on the accuracy of the ENO scheme. ENO schemes attempt higher order accuracy and to avoid oscillations at discontinuities [11]. WENO implementation can be either component-wise or characteristic wise, in terms of the cell reconstruction procedure. A review of reconstruction methods for ENO and WENO schemes are given in full detail in [59]. Component-wise applies scalar reconstruction procedures to each conserved vector component at each cell interface, then applies an exact or approximate Riemann solver to form the high resolution scheme. Characteristic-wise computes average state values, and the left and right Jacobian eigenvectors, then the eigenvectors are used to transform the cell variables to characteristic variables which are then reconstructed using WENO/ENO procedures. The component method is more simple to implement yet the characteristic method is more robust [59]. See Appendix B.2 for the WENO algorithms used.

Proposed by Bram van Leer in 1979 [74] the MUSCL scheme was able to achieve second order accuracy and behaved at least an order of magnitude more efficient that Godunov schemes, equivalent of refining a mesh with factor two. Van Leer's MUSCL approximates the cell density distribution with polynomials of order $n \in \{2, 3\}$, this alone would introduce large oscillations in the presence of shocks. To reduce oscillations and ensure the scheme is TVD, the slope limiter ϕ is applied at left and right states. TVD schemes reduce to first order at local extrema, and are as high order

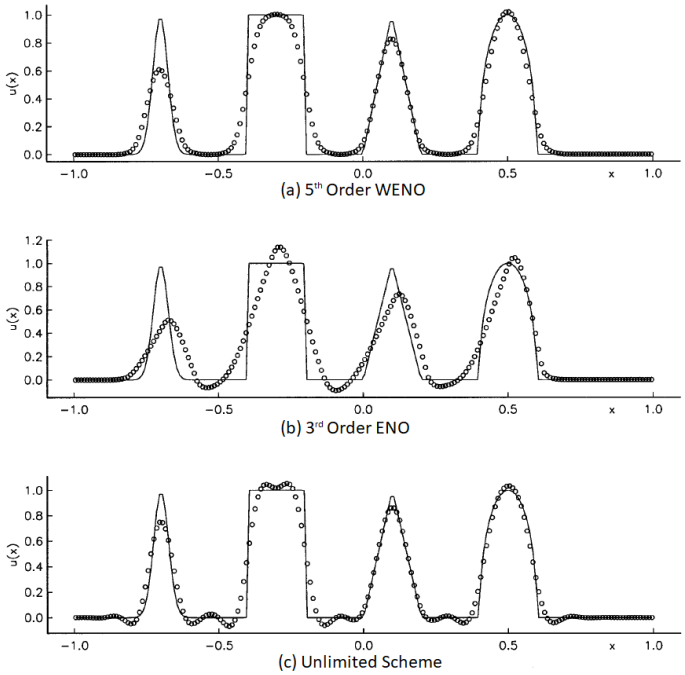


Figure 3.2: 1D advection equation results from Suresh and Huynh [60]. The unlimited scheme (c) is not high resolution, hence oscillations appear in the solution in regions of high gradient $\partial\rho/\partial x$ discontinuities. Using the smoothest polynomial reconstruction in the 3$^{\text{rd}}$-Order ENO scheme (b) finds a monotone improvement from (c). Using a specific weighted average of polynomials, the 5$^{\text{rd}}$-Order WENO (a) scheme has improved accuracy by choosing appropriate polynomials not just the smoothest one, as in ENO (b).

as the general scheme in smooth regions. The 2nd order MUSCL scheme approximates the cell density distribution with a linear slope, whereas the 3rd order scheme approximates with a parabolic slope from a second order interpolation. Oscillations may appear in solutions with large flow variable gradients due to a numerical procedure with no artificial dissipation [69], the latter will reduce flow to a monotone solution however is inaccurate. To keep monotonicity yet eliminate the need for artificial dissipation, flux gradient limiters are used. Barth and Jesperson [6] suggest a slope limiter, ϕ_i, which limits the gradient in the reconstruction

$$R_i\left(x_j - x_i\right) = \rho_i + \phi_i \nabla\rho_i\left(x_j - x_i\right), \quad with \quad \phi \in [0, 1].$$

Mathematical descriptions of two limiters have been summarised from Michalak and Ollivier-Gooch [42], where reconstruction is described with the need of gradient limiters and a useful algorithm for the Barth and Jespersen limiter is presented which can be developed into Venkatakrishnan's limiter. See Appendix B.1 for the MUSCL algorithms used.

3.5 Limitations of Finite Difference Schemes

Finite difference is the approach of describing derivatives by finite discrete values. Chung presents many FDMs and evaluates their performance [11]. FDM can only be used on structured grids[1]. Hyperbolic PDEs are used to model wave propagation; FDM are limited by strict stability criterion restricting spatial and time step sizes, with are due to bounding the dissipative error growth. All *implicit* FDM schemes however are unconditionally unstable. Low-resolution schemes are inaccurate at discontinuities, either over or underestimating with oscillations. Improving the order of accuracy in FDM requires more initial data. Extra solving tools are required to improve the accuracy, including high resolution methods, Riemann solvers and reconstruction procedures with gradient limiters. The higher resolution group of TDV methods are considered higher order but the accuracy is not uniform, TDV schemes may range from first to second order accuracy in different solution regions. Solving systems of hyperbolic PDEs includes the calculations of Jacobians which are computationally inconvenient.

3.6 Time-Update Scheme

The single step discretisation update given in Equation 3.4 will not allow some of the higher resolution schemes to exhibit their advantages over lower resolution methods. A commonly used time-update scheme is the classical 4^{th} order Runge-Kutta process. First proposed by Runge [54] and further developed or finalised into a family of methods by Kutta [32], used by many in a wide field of study the modern Runge-Kutta framework provides a strong foundation for building a sophisticated time update scheme. The classical fourth order update is used in this TFM program, and can be found in lines [552-584] of the code in Appendix A.1.1. This method uses an accumulative weighted average of four different updates, RK_i, around the current time step. The updated density value is given by

$$\rho^{(n+1)} = \rho^{(n)} + \frac{dt}{6}\left(RK_1 + 2RK_2 + 2RK_3 + RK_4\right), \tag{3.14}$$

where the intermediate updates RK_i are given by,

$$RK_1 = f\left(t^{(n)}, \rho^{(n)}\right), \tag{3.15}$$

$$RK_2 = f\left(t^{(n+1/2)}, \rho^{(n)} + \frac{RK_1}{2}\right), \tag{3.16}$$

$$RK_3 = f\left(t^{(n+1/2)}, \rho^{(n)} + \frac{RK_2}{2}\right), \tag{3.17}$$

$$RK_4 = f\left(t^{(n+1)}, \rho^{(n)} + RK_3\right). \tag{3.18}$$

This framework can be extended to adaptive step size based on two approximation errors, larger stability in implicit Runge-Kutta update methods.

[1] More generally FDM can also be used on *transformations* of orthogonal structured grids

3.7 Probabilistic Network Model

Previous studies [9],[58] of the fluid dynamics model application to traffic flow problems on networks provide a clear and useful mathematical description. These studies both use a traffic distribution matrix to describe the amount of flow distributed to any outgoing roads of a junction, from the incoming roads. This links the macroscopic continuum model together in a series of problems for each road segment defined by the network of interest. The macroscopic LWR model is derived by the conservation of traffic from each cell of a road segment, hence the conservation of traffic at junctions is equally as important. This junction conservation can be written as the Rankine-Hugoniot condition,

$$\sum_i f(\rho_i) = \sum_j f(\rho_j), \quad i \in \{1, \ldots, n\}, \quad j \in \{n+1, \ldots, n+m\} \tag{3.19}$$

for the general junction with n incoming roads, m outgoing roads, and where ρ_i and ρ_j respectively represent the densities at the end of incoming roads and start of outgoing roads. The traffic distribution matrix $A = [a_{i,j}]$ has probability-like[1] elements, satisfying the condition $\forall i$,

$$\sum_j a_{i,j} = 1, \tag{3.20}$$

as all flow leaving road i must be distributed to outgoing roads. These definitions and properties are sufficient and consistent over [9] and [58], with implementation given in [14], such that this approach can be applied to the traffic model listed in the junction solver of the code *main.py* in Appendix A.1.1 lines [326-374].

[1] Probabilities have $0 < p < 1$ and $\sum p = 1$.

Chapter 4

Program Development

4.1 Developing Tools

The Python code was developed entirely in the PyCharm IDE [28], this tool allows a user to activate the GitHub version control functions and view the history log from within the editor. Python was chosen as the developing language for this tool due to its compatibility and cross platform benefits, a wide range of functionality can be achieved by importing specific modules. PyCharm allows the importing of many modules from within the editor itself. While developing it can be useful to have a terminal window and Python console at hand, both of which are integrated into PyCharm. As well as providing functional assistance the IDE provides developing tips while coding, PyCharm will suggest syntax choices for automatic fill when using special functions from modules or keywords. Keywords are easy to spot as PyCharm has a colour style (which can be change to many colour schemes) that allows numbers, function definitions, keywords and comments to be identified quickly and easily. The TFM tool developed here uses many different functions, while defining functions it is important to consider the scope of variable names used in the rest of the program, PyCharm will identify a conflicting variable name from within a function as to avoid scope error. See Figure 4.1 for the user interface from within the PyCharm IDE.

All the code used to simulate traffic flow for results in Chapter 5 are present on the TFM_Thesis GitHub repository [18], available at github.com/adj97/TFM_Thesis, see Figure 4.2 for the homepage interface. GitHub is an opensource cloud storage system for program code of any sort. Not only are the current files stored, but a detailed history of every change to the system is noted. This is incredibly useful for adding extra features, where an entire copy of code is made and then developed on a new *branch*, the new feature can then be merged back to the original copy once it has been tested, or alternatively revert back to the original. GitHub is also a useful community of developers, the Facebook of coding, if public then your repository can be viewed, reviewed, tested, cloned by any GitHub user.

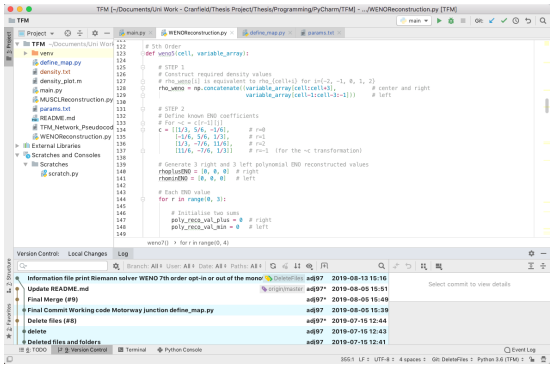


Figure 4.1: The multi-window GUI of the Pycharm IDE, the lower bar shows the version control log, left is the local file directory, and main window for the code editor with files open in different tabs.

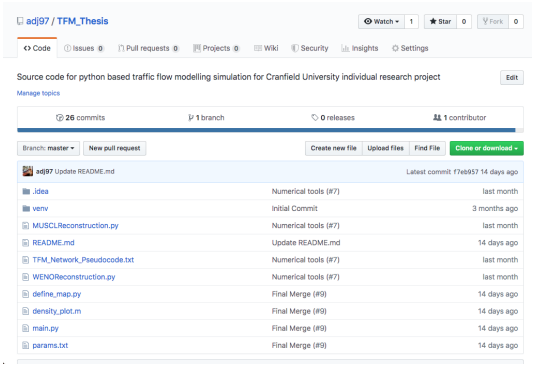


Figure 4.2: The TFM_Thesis repository homepage, showing committed files and information on history of version control. This page links to all other features; issues, project management and all working branches.

4.2 Non-Numerical Features

The complex numerical procedure presented in *main.py* (Appendix A.1.1) is complemented by many non-numerical tools that improve ease of use, and provide extra information to the user. The following give a brief explanation to some of these features,

- Error checks and print statement - Once the program has read in the network and junction_info dictionaries from *define_map.py*, and the parameters from *params.txt*, many aspects of this input is checked for any non-compatible entries. The error types can be found in lines [55-60] of *main.py* in Appendix A.1.1. If any such errors are found then a breakdown of error messages and how many have occurred are printed before the program exits allowing the user to re-input information and try again.

- Internal data structures - There are a wide number of arrays and data objects that have small and large features in the overall solver, listed are the main objects with a small description

  - Dictionaries - Small information structures are useful to store in dictionaries as each entry will have a key tag that can be used to give meaningful names to objects.
  - JSON file - The parameter text file is written in JavaScript Object Notation (JSON), this allows for quick and simple definition of the parameter values after being read in to the main program.
  - Local flows - This is the output of the junction solver, which assigns each in/out road at a junction a flow value.
  - Global flows - The local flows array feeds into this global flows array which acts every time step to define each road's supply and demand value to be used as boundary conditions in the iterated solution.

  - Source/Sink lists - A straight forward vector-type list of numbers defines both the road indexes of source and sink roads in two separate arrays. This is used to loop through and test if a road is a source/sink.
  - Rho - The density solution is stored in this array, the whole network is can be stored road-by-road back to back for every cell for every road.
  - Supply/Demand in junction solver - For the junction solver, this array provides the flow values at the end of in-roads and start of out-roads.
  - Ghost densities - Prior to calculating the reconstructions, some high resolution schemes require values outside of the solution domain. This ghost array provides these values with symmetrical conditions at domain boundaries.
  - Reconstructed - Looping through each cell, this array has two columns that are used to store the left and right reconstructed states for each cell.
  - Cell fluxes - Reading from the reconstructed array (above), the chosen Riemann solver will compute from the two values of reconstruction at a cell interface. This array stores the Riemann problem approximate solution at the right hand side cell interface for each cell.
  - Runge-Kutta arrays - Four separate lists of the density solution are stored, each represent the value after each Runge-Kutta iteration.

- Function : get-start-end - This function is crucial to the data structure approach of storing the whole network solution in a single array. The function will return two indexes for the start and the end of the prescribed road of interest index.

- Loop progress bar - Giving the user more information on the progress of a simulation, this feature, shown in Figure 4.3, is taken from [13] and is available at github.com/tqdm/tqdm.

- Timing Segments - To provide information about the time spent in certain areas of the code, timing trackers are placed and results are printed in simulation output info (next in this list), these results are shown in Section 5.3.

- Output information - Meaningful console (Appendix C.4.2) and information text file (Appendix C.4.1) print statements are provided if requested. The saved text file is placed in new created folder (named as the date and time) in the main working directory, which also includes the final density solution. This allows many simulations to be completed while not losing information on which parameters have been used to generate the solution. Included in the information file is a line count for all code contributing to the program, and a measure of the size of the final density solution output.

Figure 4.3: The information shown on the progress bar is (left to right): overall loop percentage complete, moving partial progress bar, loops completed/total number of loops, total time elapsed, estimated time remaining, loop speed in iterations per second.

4.3 File Structure

It is practical to arrange code in an organised manner, this includes splitting large code chunks into separate files and ordering each file to input or apply when necessary. The run procedure for this TFM program is to execute the *main.py* code, this will call three other Python files, and read in a single parameter text file. Firstly the definition of the road network of interest is entirely contained within *define_map.py*, this means there is no need to alter the code in *main.py* before use. The other two Python files that are used contain the MUSCL and WENO reconstruction procedures, presented as functions that take in the density array and a cell index which identifies the cell to reconstruct. These, like *main.py*, need no code changes before running a simulation. Once a simulation is complete and the *density.txt* output file is saved, any MATLAB postprocessing scripts can be used to analyse the results, for example splitting the array into a density profile for each road and plotting according to the road network.

4.4 Simulation Procedure

4.4.1 Pre-Processing

Prior to executing the simulation code, one needs to plan the study. The input in *define_map.py* requires lots of information about the roads and the junction links. It is useful to sketch a simplified diagram such as in Figure 4.4, which includes the direction for each road, identifies sources and sinks, and which gives indexes to each road and junction. Other information required for input are the each road length, maximum speed, and jam density. This diagram will help establish the road indexes, and allows the junctions to be identified by the road indexes of in and out roads. The only remaining aspect of network definition are each junction's individual TDM, the elements of which have constraints (See Section 3.7) but can be estimated with rational.

4.4.2 Code Algorithm

Once the previous steps have been carried out to define the road network of choice, the code in *main.py* can be executed and the procedure outlined in Figure 4.5 will be carried out resulting in a solution profile and information file being saved to a local results folder.

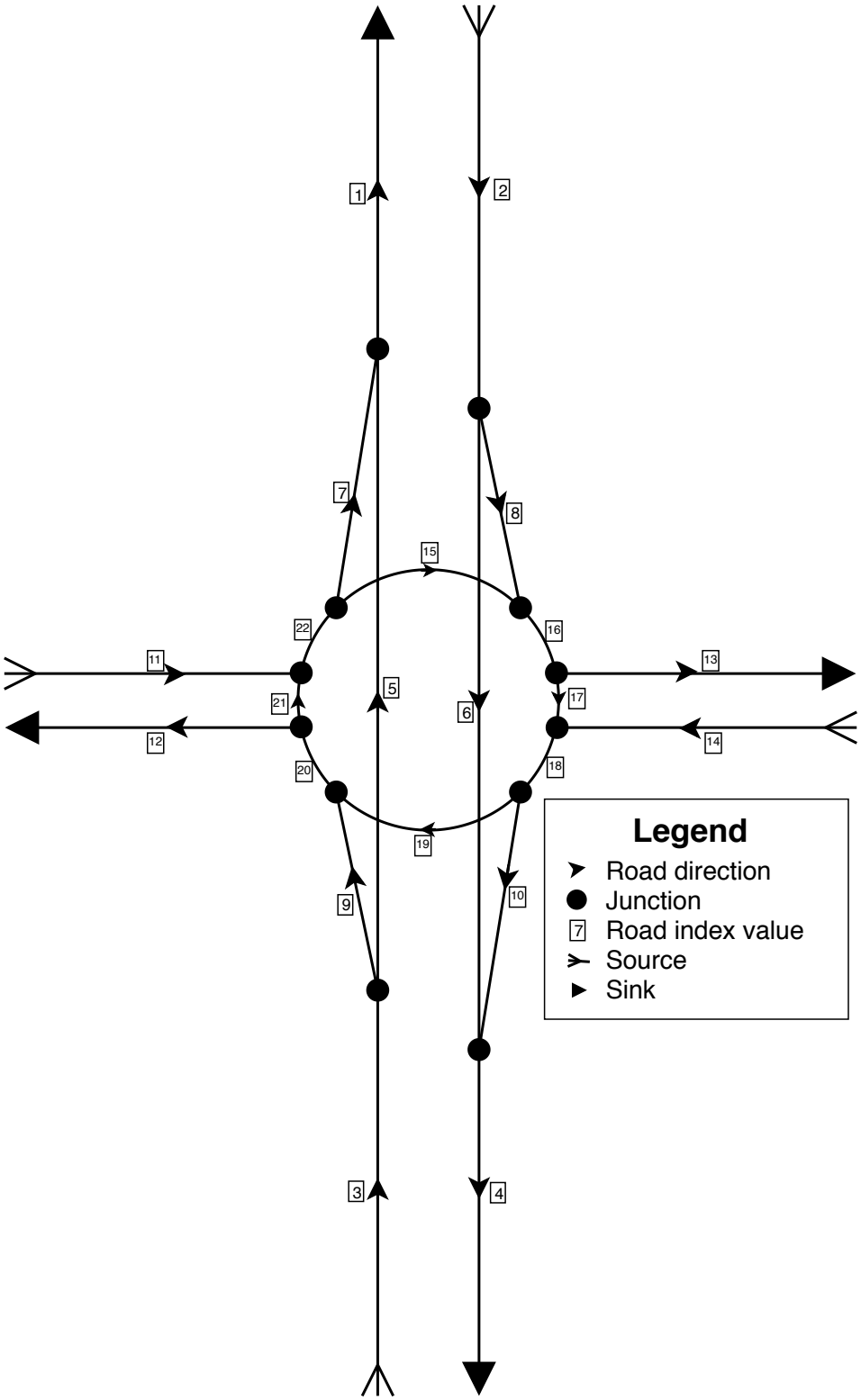


Figure 4.4: A diagram theme for planning the definition of a road network into *define_map.py*, this diagram represents the Wakefield M1 junction 40 network presented in Section 5.2.2.

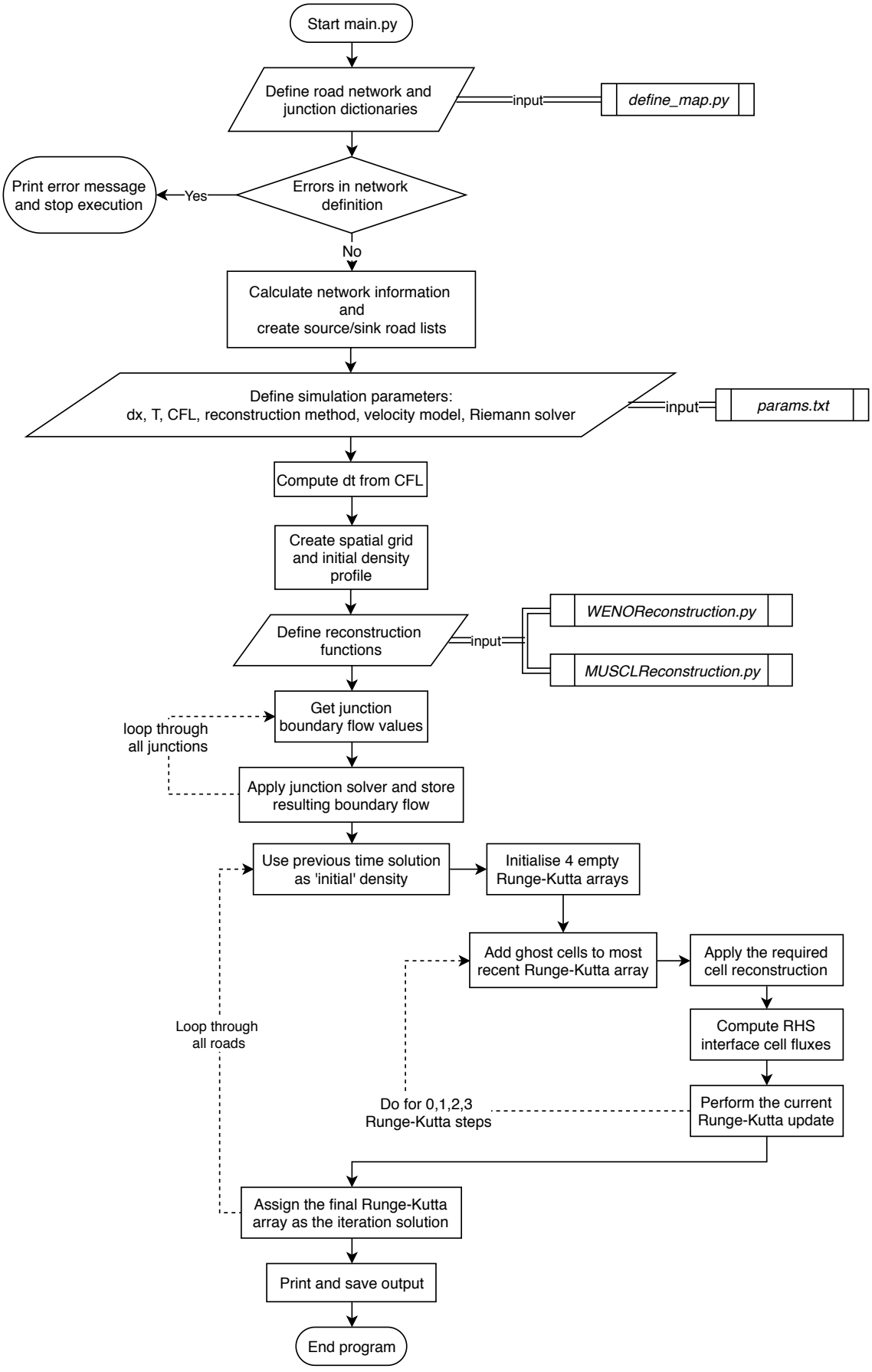


Figure 4.5: Simulation process key steps flowchart.

4.4.3 Postprocessing

The resulting *density.txt* file is difficult to interpret raw; after being read in by a MATLAB script, this array can be split into many smaller objects containing the density for all time steps on a single road section each. Some example MATLAB postprocessing scripts are given in Appendix A.2, these can be used with no change to recreate some results of Chapter 5, or can be used as a template to develop a new script for a new road section.

Chapter 5

Results and Discussion

The following results are organised as the following: simple theoretical road networks used to test numerical features, road sections from two real-world locations showing the application of this model to real situations, and an analysis of various simulation times.

5.1 Simple Road Tests

To begin using the model, the most simple road networks are to be tested. Is it useful to test over-simplified and somewhat unrealistic networks to show the working features of the probabilistic network model and some simple features such as the source and sink roads in action. Here a simple single road and a network, proposed by Bretti, Natalini, and Piccoli [9], known as the traffic circle resembling a common roundabout.

5.1.1 Single Road Segment

The most simple network tested is a straight road with no junctions, made up of a single road segment that is both a source and a sink. The junction solver is not called as there are no defined junctions, hence this test can be executed very quickly due to the simple simulation process.

This simple network is useful to expose individual features of the solver, such as the higher order WENO reconstruction methods. As described in Section 3.4.1 and given explicitly in Appendix B.2, the 5^{th} and 7^{th} order schemes are available as reconstruction methods. The 7^{th} order method is equivalent by process to the 5^{th} and 3^{rd}, with the addition of steps to preserve the monotonic bounds of each cell reconstruction [5],[60]. Figure 5.1 shows the 7^{th} order density profile solution for the single road with 5^{th} order as a reference. The yellow line shows the density profile for the unbounded-7^{th} order WENO scheme. The solution following the 10^{th} time step becomes *numerically* unbounded for the unbounded-7^{th} order WENO scheme. This shows that the procedure presented in Appendix B.2, for the general $(2k-1)^{th}$ scheme cannot be extended to $k \geq 4$ without extra methodology applied such as

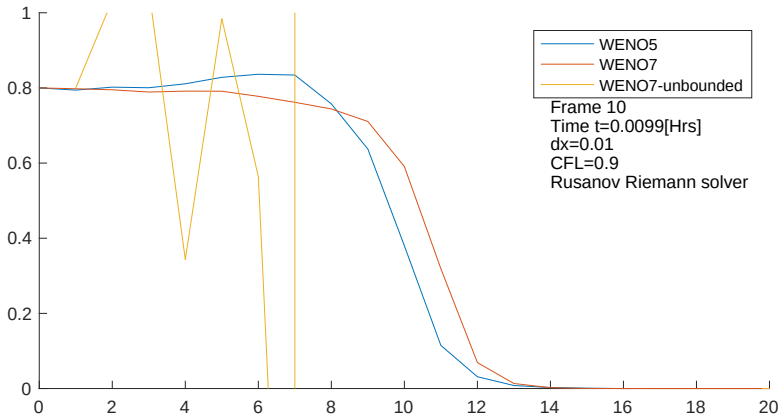


Figure 5.1: The scheme used for 5th and 3rd order WENO reconstruction cannot be extended to 7^{th} order without monotonic bounding applied. The unbounded solution is highly modulated, and becomes numerically unbounded after 10 time steps.

the monotonicity preserving bounds [5]. This figure also highlights the advantage of the 7^{th} order scheme over the 5^{th}, in terms of capturing density jumps to a higher accuracy. The 5^{th} order scheme actually predicts a jump in density earlier than the main density drop, a high density clustering of fast moving cars at the front of the stream. This is unrealistic, and predicted better by the 7^{th} order scheme.

5.1.2 Traffic Circles

The following simulations have reproduced the traffic-circles network and parameters from [9], with 8 roads joined by 4 junctions (as shown in Figure 9 of [9]). Unless otherwise stated in the figure legend, the simulation settings for the results and following discussion are d$x = 0.01$, $CFL = 0.9$, 2^{nd} order reconstruction and Lax-Friedrichs Riemann solver. The defined inlet density from the left and right are 0.25 and 0.4 respectively, this feeds into the four central roads which can be shown by the jump in density along the four internal roads and top and bottom outlet roads, shown in all Figures 5.2, 5.3, 5.4.

It is expected that high order spatial reconstruction results in a more accurate prediction of jumps in traffic density, however such simulation parameters as in Figure 5.2, there is no obvious advantage of the 3^{rd} order WENO over 2^{nd} order TVD reconstruction. The parameter $q_1 = 0.5$ describes equal flow joining and leaving the traffic circle at available junctions. A result of this some density will always remain in the central roads and the density profile will become smoother where the effect of spatial reconstruction is not as significant.

One major fault of the model concerning the numerical flux calculation is the definition of certain parameters used in calculations from Section 3.3. This is concluded from Figure 5.3 where the Rusanov, Murman-Roe and HLL solvers all appear to give an identical solution. The Lax-Friedrichs solution differs from the other solvers but in a less accurate manner in terms of shock resolution.

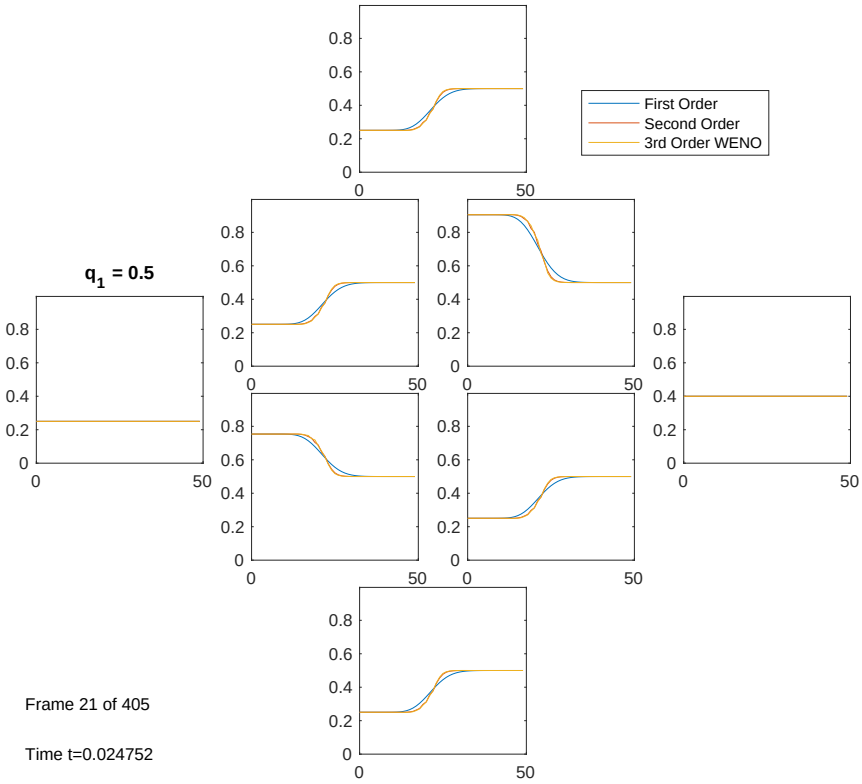


Figure 5.2: Spatial reconstruction influence, lower order reconstruction smooths out local discontinuities.

Another highly influential parameter in the resolution of density jumps is the spatial resolution dx, of which each road segment is split into cells of this size where the density profile is resolved. Figure 5.4 gives another solution to the traffic circle network, with the values of dx for each grid given in the caption. As expected the coarser resolution smooths out density jumps, whereas finer will capture a stronger shock, which is clear from this figure. The coarse and fine simulations give smooth density profiles in comparison to the medium solution which gives a slightly more oscillatory jump.

Another aspect of the flow modelling that is analysed through the traffic circles network is the MUSCL reconstruction slope limiters, discussed in Section 3.4.1 and given in Appendix B.1. All of the 15 listed slope limiters are applied and solutions shown in Figure 5.5, where the top row and bottom row respectively represent limiters applied to the 2^{nd} and 3^{rd} order MUSCL reconstructions. It is immediately clear that the 3^{rd} order MUSCL reconstructed solutions are more oscillatory with all limiters. Of the 2^{nd} order reconstructed solutions, the smoothest profiles arise from the Monotonised Central, UMIST and VanLeer limiters. The VanAlbada2 limiter is numerically unstable for both MUSCL schemes, the solution becomes so oscillatory that after sufficient time steps it is numerically unbounded. This may arise as a result of the violation of Sweby's TVD region [61] for slope limiters, see Appendix Figure B.3.

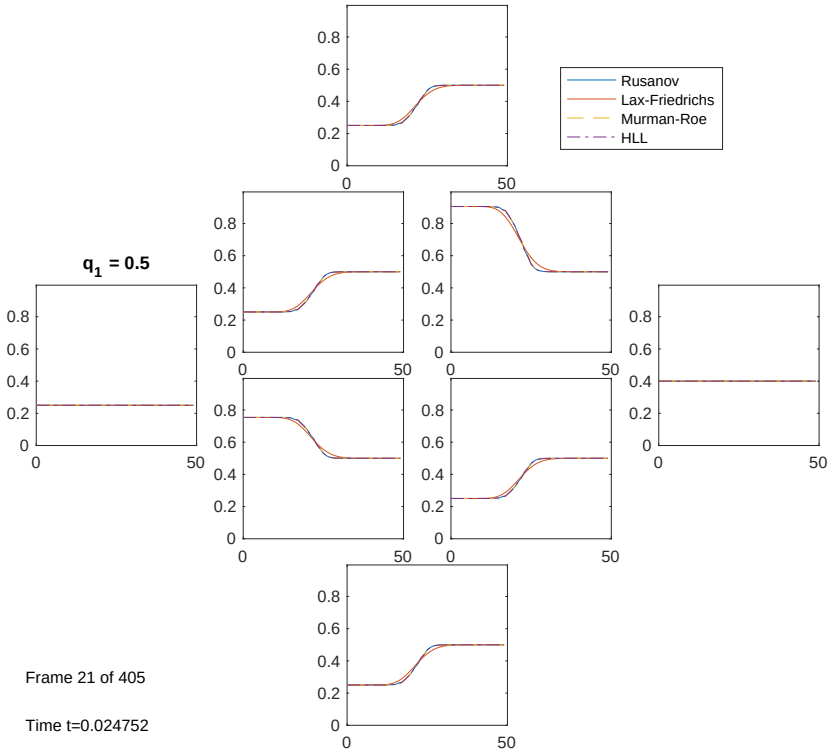


Figure 5.3: Varying the Riemann problem solution method, showing the equivalence of Rusanov, HLL and Murman-Roe.

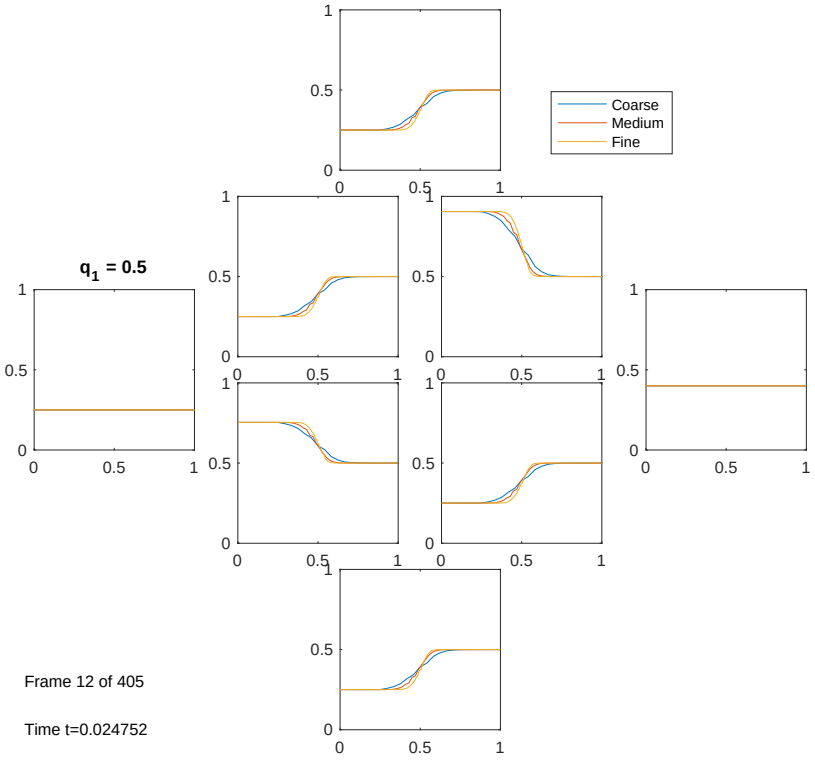


Figure 5.4: Coarse to fine respectively represents $dx = 0.02, 0.01, 0.005$. Increasing the spatial increment causes discontinuities to be resolved more accurately, coarse grids dissipate the density shock.

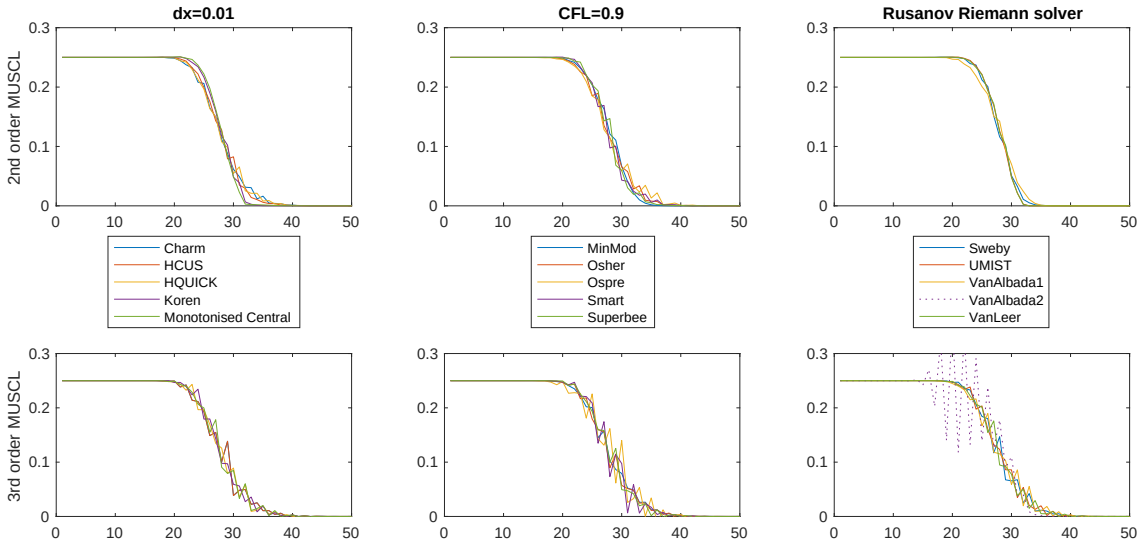


Figure 5.5: Various slope limiters applied to MUSCL reconstruction on the traffic circle case. The top and bottom rows represent the density solution on the leftmost inlet road with second and third order MUSCL reconstructions respectively. The VanAlbada2 limiter becomes numerically unbounded after 10 and 35 time steps for 2nd and 3rd order reconstructions respectively.

5.2 Real Road Networks

Following the theoretical networks used to test the models numerical and probabilistic capabilities, these sections show how the simulations work when reproducing traffic behaviour on real road network sections. The Piazza dei Re Di Roma roundabout in southeast Rome, and a common UK motorway junction taken from the M1 near Leeds.

5.2.1 Re Di Roma Roundabout

This road section is studied in [9], and resembles a more realistic traffic circle as in Section 5.1.2. Figure 5.6 shows the network surrounding the roundabout from Google Maps. As this network is real, the postprocessing involving a real road network simulation needs to be more aesthetic than density profile plots. Figure 5.7 shows the result from an initial simulation using the Re Di Roma roundabout. This figure shows the density not as a distribution but through colour of a physical diagram of the network, the MATLAB code used to generate these figures is given in Appendix A.2.1. Frames from this initial simulation are used to generate the animation *ReDiRoma.mp4*. The parameters used for this simulation result in the solution developing (Figure 5.7a) and reaching a steady state (Figure 5.7b). These results use first order cell reconstruction with the Lax-Friedrichs Riemann solver. The following results test other numerical methods on the Re Di Roma network, only showing the solution for the central roundabout in polar coordinates the density is shown by the distance from the black circle.

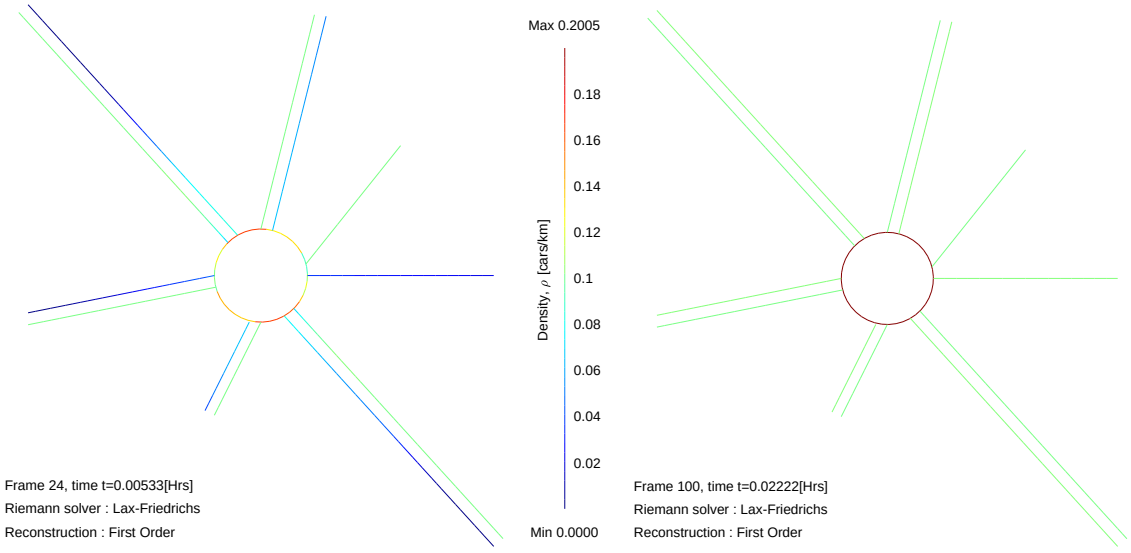
As this real network has road segments of irregular length, choosing an arbitrary spatial increment dx can cause the postprocessing to be out of sync when comparing

Figure 5.6: Central Rome roundabout network. *Google Maps, 2019.*

two solutions. Figure 5.8 shows the roundabout density solution with different spatial resolutions, the coarse simulation is slightly out of sync due to the length of the road network not being divisible into an integer number of cells. For this reason, the coarse profile is smoothed over the density jumps resulting from traffic coming in to the roundabout from source roads. The drops in density are equivalently from sink roads leaving the roundabout. The fine simulation predicts the density jump locations more accurately than both medium and fine. The south-southwest region of the roundabout has an increase of density, shown by the fine and medium solutions, however the coarse solution smooths this small increase in density so much that it could be interpreted differently if analysed without the finer solutions to compare to.

The analysis of reconstruction methods on the Re Di Roma roundabout is shown at two stages during the simulation in Figure 5.9. This physical postprocessing method makes identifying shock capturing accuracy difficult, however is used to show the application to real networks. On the south of the roundabout (left figure), it is evident that the 3^{rd} order WENO reconstruction captures a stronger shock than the second order and first order as expected. It is useful to know these reconstruction methods are behaving as expected even when embedded in a large program that is solving a complicated process defined on an equally complicated real road network. Later in the simulation (right figure) the solution is becoming more smooth so the effect of reconstruction is not as significant.

We have already seen from simulations on the traffic circle, and in Figure 5.3, that the choice of Riemann solver has little influence. Figure 5.10 shows this is also the case for the Re Di Roma roundabout case. The solution for the HLL and Murman-Roe solvers are equivalent with the Lax-Friedrichs solution differing only very slightly. It is clear that the definition of Riemann solvers in traffic flow network modelling simulations needs to be decided more carefully.



(a) Developing flow (b) Steady traffic distribution

Figure 5.7: From the initially empty distribution, traffic flows towards the roundabout (a) and reaches a steady density profile (b).

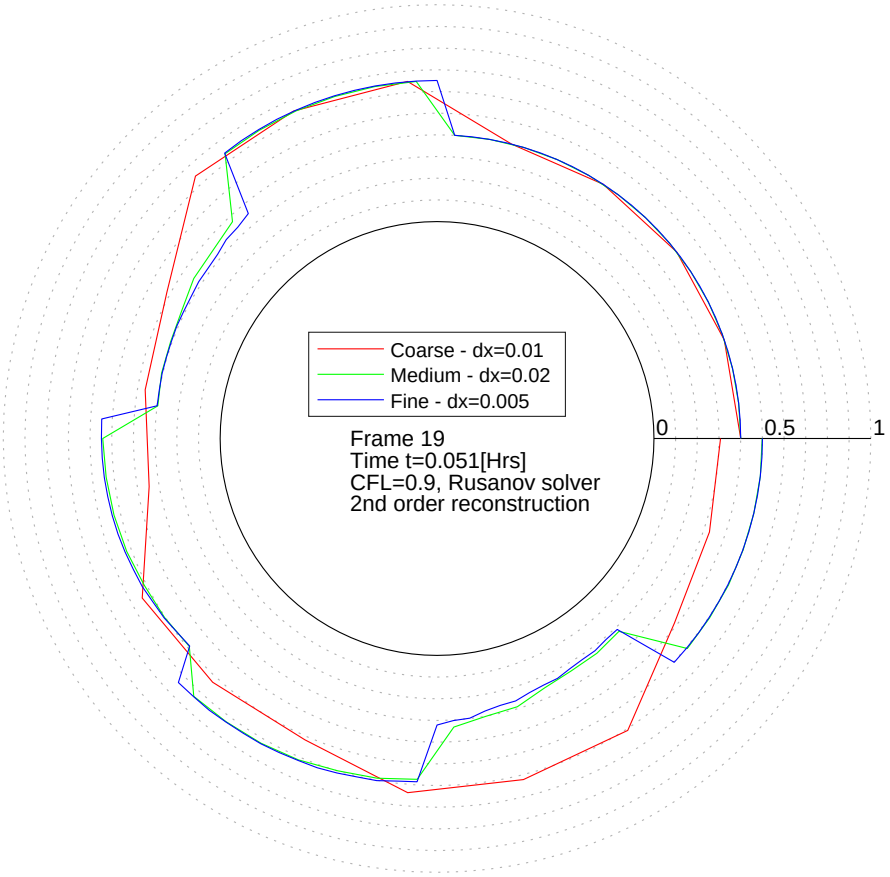


Figure 5.8: As expected the finer grid resolves jumps in density from oncoming roads well, the coarse grid is such that it smooths or even misses density jumps completely due to the resolution of the simulation.

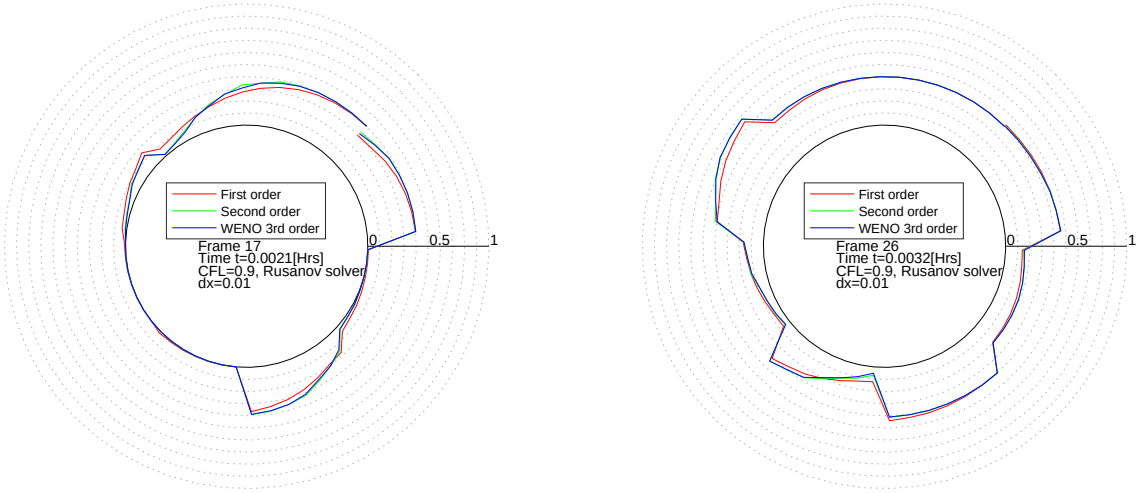


Figure 5.9: Two time steps at frame 17 (left) and 26 (right) of traffic density feeding through the Re Di Roma central roundabout. Lower order reconstruction methods smooth out discontinuities at junctions whereas the higher order methods resolve jumps more accurately.

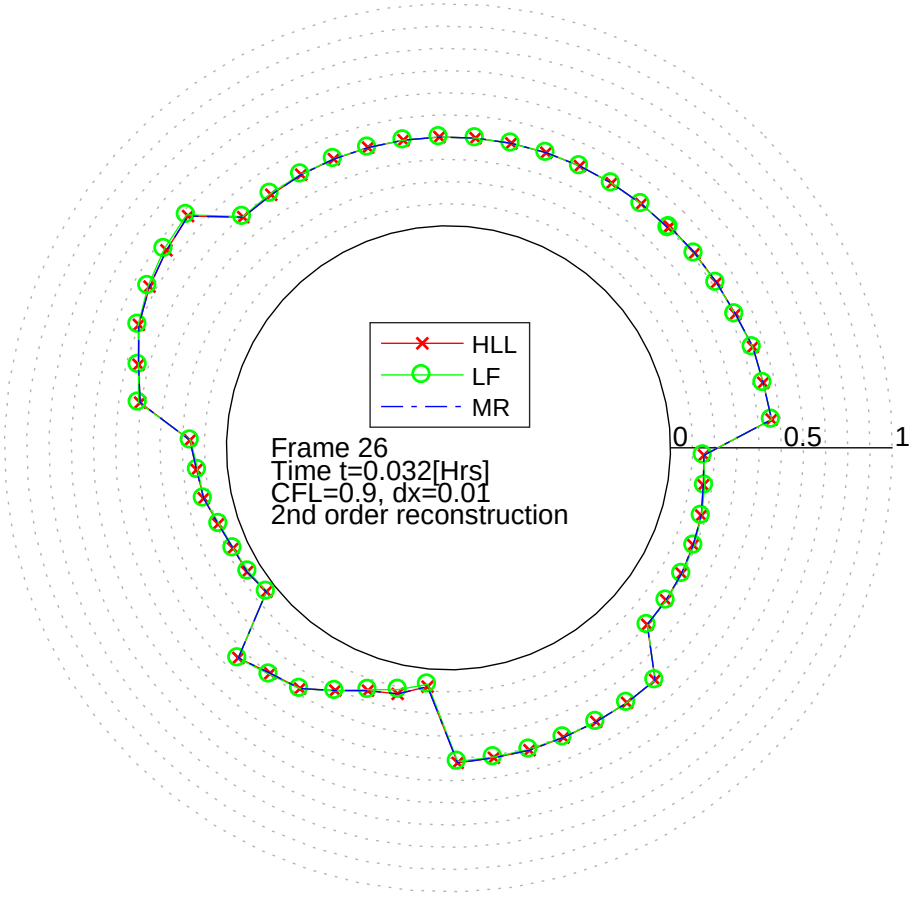


Figure 5.10: The choice of Riemann solver results in a negligible difference in solution. The HLL and Murman-Roe (MR) solver solutions are identical and Lax-Friedrichs (LF) differs only very slightly.

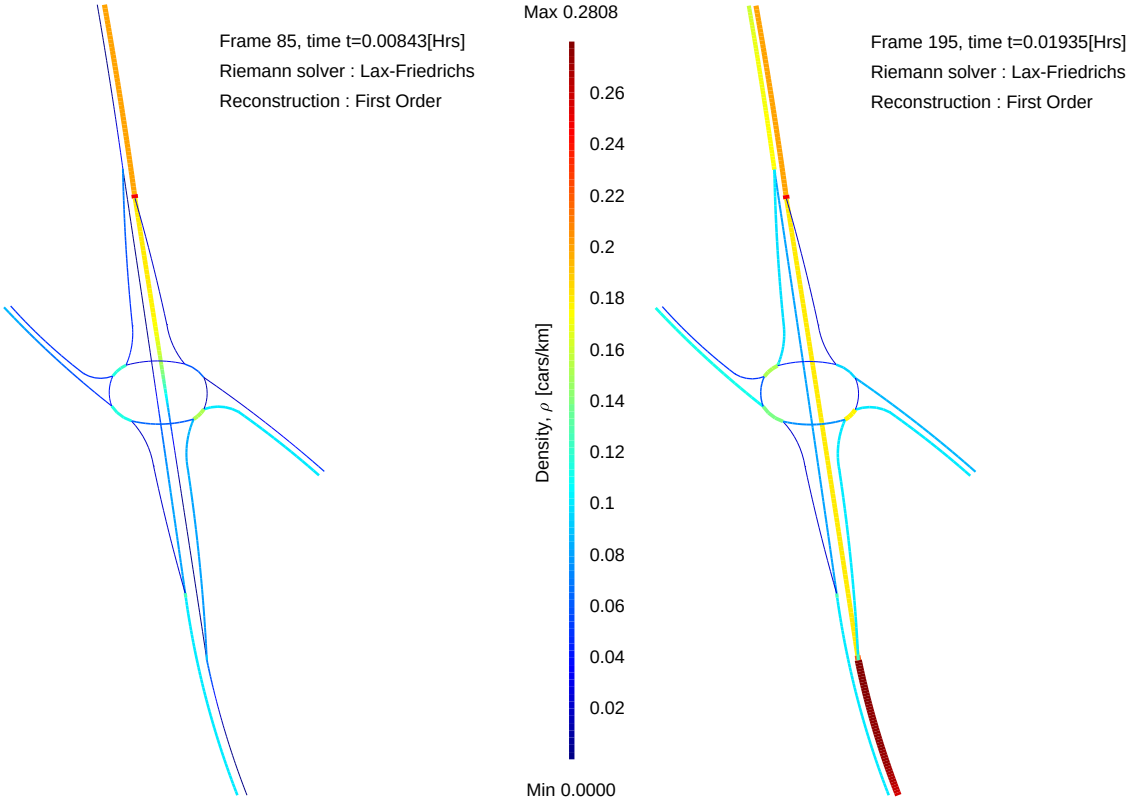
5.2.2 Wakefield M1 Junction 40

The common simple UK motorway junction meets a crossing A-road at right angles and the model junction can be described by junction 40 of the M1 in Wakefield. Figure 5.11 shows the small interchange network surrounding the meeting of a fast high-density motorway taking traffic to and from London, the Midlands and Leeds. This network is a good example of when a time-dependent TDM may be useful to model over a long time period such as 24 hours. During the morning hours lots of traffic may be coming from all directions towards Leeds, and the opposite in the evening rush hour at high densities. This map has been traced in MATLAB and used for postprocessing (Appendix A.2.2)for the initial simulation results, which can be found in Figure 5.12. Not only is the line drawing of the motorway junction coloured by high and low local densities, but the higher density regions are matched with a thicker line. This simulation uses first order reconstruction with Lax-Friedrichs numerical flux calculations. Frames from this initial simulation are used to generate the animation *M1J40.mp4*. This figure allows the clear identification of a high local density on the southbound lane of the motorway after traffic rejoins from the A roads and the roundabout slip road. The following results use more classical postprocessing and analyse the influence of the CFL number, and reconstruction around the motorway and slip lanes.

The southbound motorway lane and its slip lane off to the junction roundabout are shown in Figure 5.13, with 4 profiles for different values of the CFL constraint which defines the time resolution from the spatial resolution. The main motorway section shows a spike in density just before the slip lane begins, after which the slip lane density begins to increase. This is not as expected and may be due to a fault in the junction solver. In terms of the CFL influence on solution accuracy, lower CFL give rise to a more accurate capturing of the density jump along the motorway section that has not departed off the slip lane.

The spike observed in Figure 5.13 cannot be due to the reconstruction interference at junctions as Figure 5.14 shows the same spike for many reconstruction methods. This figure shows the whole southbound M1 lane with off and on slip roads. The two drops in density at 100 and 170 are due to the inlet flow of the southbound motorway not leaving the off-slip road, and flow from the A road joining the motorway from the on-slip road respectively. The 5^{th} order WENO scheme gives a small increase before the main drop in density, this is not seen in the 3^{rd} or bounded 7^{th} schemes. The 3^{rd} order MUSCL scheme appears to be less accurate than the 2^{nd} order, which from Figure 5.5 we can conclude that with a more sensible choice of slope limiter, the 3^{rd} order MUSCL scheme will behave better. A different slope limiter may also reduce some of the oscillations in the density drop with this MUSCL reconstruction. We can also see that the smaller drop in density from 0.3 to 0 gives rise to a less smooth 7^{th} order WENO reconstruction, when compared to the larger earlier jump.

Figure 5.11: Simple and common UK motorway and A-road junction. *Google Maps, 2019.*



(a) Developing (b) Steady

Figure 5.12: Heavy flow southbound from Leeds towards Wakefield and further south. Highlighting the higher density areas of the motorway junction roundabout.

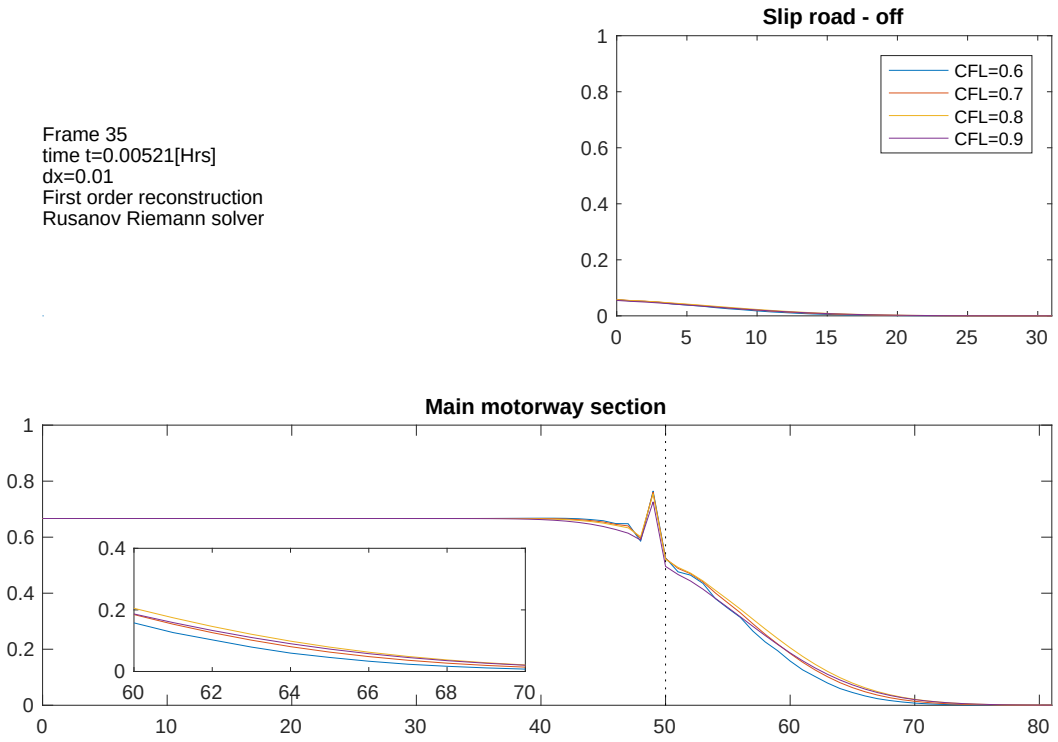


Figure 5.13: The influence of CFL number is little, there are oscillations present at the slip road entrance for all CFL. The value $CFL = 0.6$ appears to capture the traffic density jump with highest gradient. These simulations are analysed by simulation time in Figure 5.18

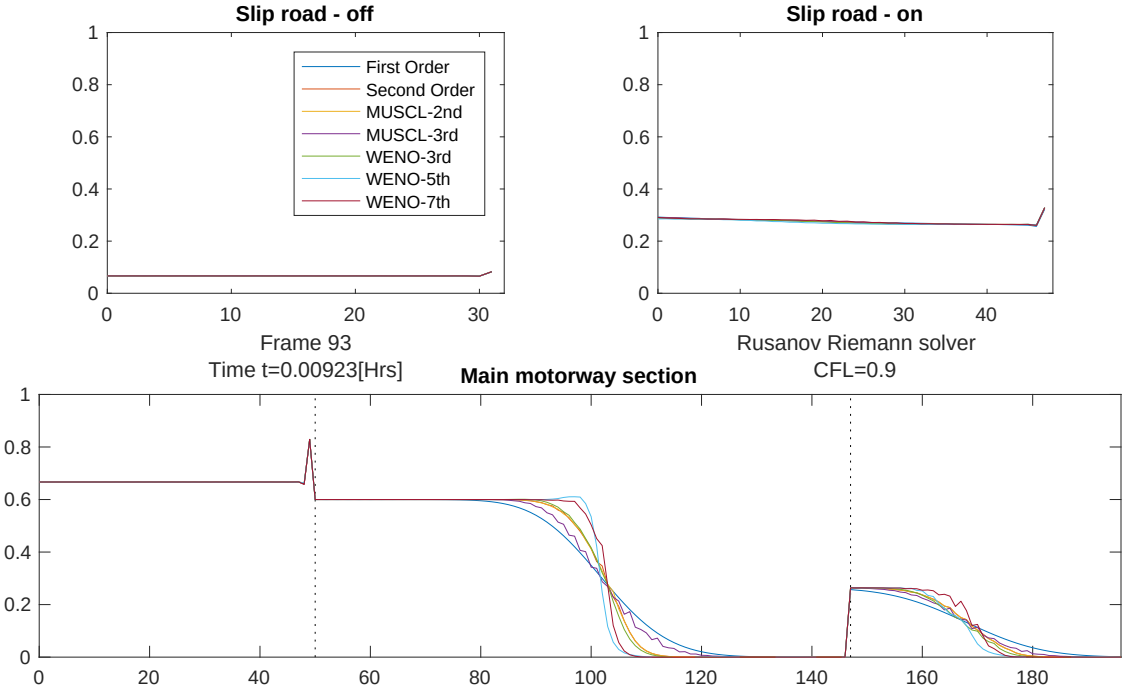


Figure 5.14: $dx = 0.01$. This figure shows the southbound motorway lane with two slip roads towards and away from the roundabout. Reconstruction appears to have a significant influence on the accurate capturing of a jump in traffic density. The general pattern is a more accurate shock resolution for higher order schemes, however this comes at the cost of computational time and oscillations.

5.3 Time Analysis

A high resolution numerical scheme can be highly time-costly, and when such a numerical scheme is coupled with a large road network as the domain this cost is multiplied. It is therefore important to analyse which aspects or parameters of a simulation cause changes in the computational time. The following results are taken from the simulation information output, see Appendix C.4 , with a total time and a breakdown of the time spent on reconstruction and solving local Riemann problems for example.

The analysis of the 7^{th} order WENO scheme in Section 5.1.1 and Figure 5.1 is followed by Figure 5.15. This shows how the higher order WENO schemes vary in terms of total simulation time (orange) and the time-breakdown on simulations of the single road segment. With high resolution reconstruction methods, the reconstruction time will dominate the simulation and the total time is reflected by this. The unbounded 7^{th} order scheme saves time over the bounded solution by not computing the bounds themselves, but as seen in Figure 5.1 this saving is not worth the cost of a meaningless solution. The monotonic bounding from Balsara and Shu [5] is essential for the 7^{th} order WENO reconstruction.

The alternate MUSCL reconstruction method also has an important parameter which contributes to both computational speed and solution accuracy. The MUSCL slope limiter as in Section 3.4.1 and Appendix B.1.3 is a simple tool such that can easily be written into any program to be varied and tested for solution (Figure 5.5) and in Figure 5.16, for computational time. We identified the Monotonised Central [73] and UMIST [37] limiters as the most admissible in terms of density jump capturing accuracy on the traffic circles network, and in this figure it is clear that these two schemes increase the computational cost. The VanLeer [72] limiter can be seen here to be one of the quickest, and in Figure 5.5 is identified as a high accuracy limiter with a smooth and high gradient density drop. This figure also shows the difference in cost when limiters are applied to the 2^{nd} or 3^{rd} order MUSCL schemes, of which all limiters show a similar small increase in time and all 3^{rd} order total times are greater than 3^{rd} order totals as expected. The VanLeer [72] limiter however has a slightly smaller 3^{rd} order speed-up.

The final time analysis from the traffic circles network is that of spatial resolution, the time taken for solutions shown in Figure 5.4 and more values of dx are simulated and the time breakdown is shown here in Figure 5.17. It is expected that with a finer spatial resolution, the road network is described by a greater number of computational cells which will result in more calculations and a greater simulation time. With a second order reconstruction it is interesting to see that it is the local Riemann solutions that begin to dominate the total simulation time for the smallest dx simulated here. At the largest $dx = 0.175, 0.2$ the reconstruction and flux times appear very similar. As with most numerical simulations there is a compromise between solution accuracy and computational time, here a value of $dx = 0.05$ gives a valuable solution and saves 75% of the $dx = 0.025$ simulation time.

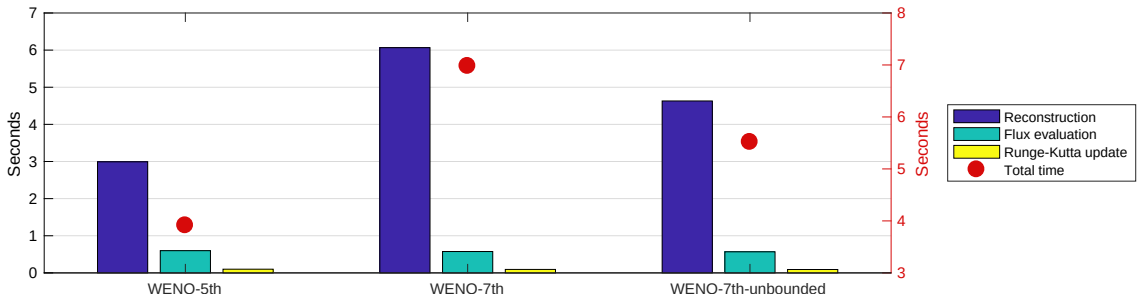


Figure 5.15: As expected the 7^{th} order scheme has an increased computational cost over the 5^{th} order due to calculations over an extra stencil. The unbounded scheme is quicker as no monotonic bounds are calculated but at the cost of an inadmissible solution this cost saving is not valuable.

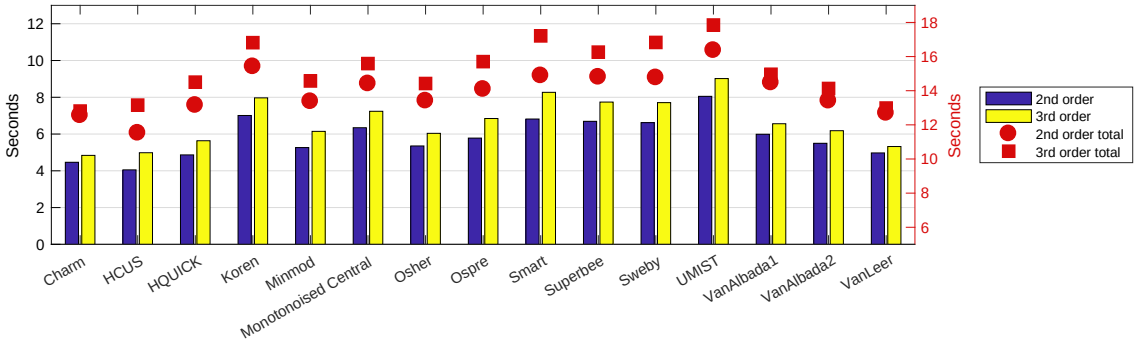


Figure 5.16: For all MUSCL slope limiters, the reconstruction time is shown by the bar chart and total simulation with the orange shaped marker. No limiter has a significantly higher time cost with the 3^{rd} order reconstruction.

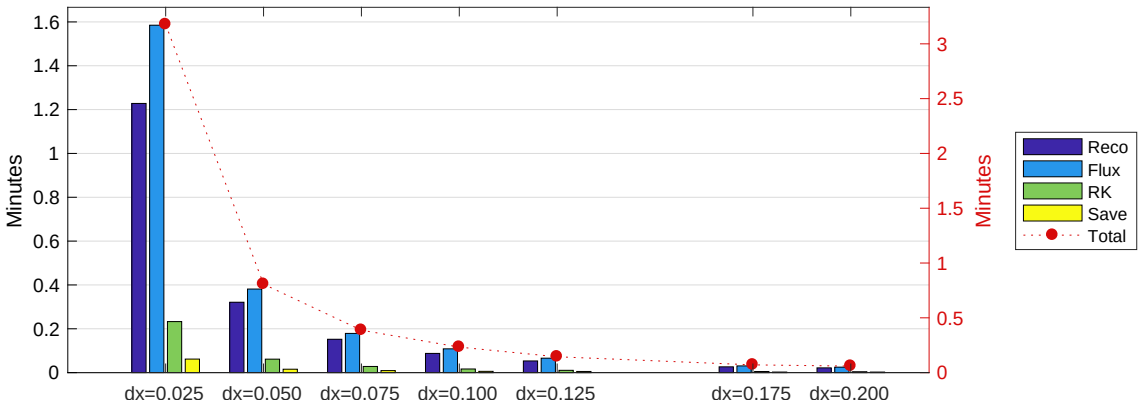


Figure 5.17: A similar time-breakdown shows results including those from Figure 5.4, and more simulations with varying spatial resolution dx.

The next figures use time information from solutions of the Wakefield M1 junction simulations. Figure 5.18 tests the breakdown of varying the CFL constraint. With a constant dx, $CFL \propto dt$, so one would expect lower CFL to give rise to longer computational time. This is the opposite of what is shown in Figure 5.18, computational time increases to a maximum at $CFL = 0.8$. At this simulation setup of first order reconstruction with Rusanov flux calculations (see Figure 5.13) the flux evaluations are dominating the total time, but with little difference between density profiles over each CFL it may be wise to use $CFL = 0.6$ to save time while also capturing the strongest density gradient along the jump.

From the analysis so far, it is not conclusive that the reconstruction influences the simulation time the most. However Figure 5.19 shows the simulation times for all available reconstruction methods in the solver parameters. With similar times for the 2^{nd} order TVD and MUSCL schemes, generally the higher order reconstruction scheme will take longer to complete the reconstruction process so much that the total simulation time appears similar to the individual reconstruction time bars. After the 1^{st} order scheme, all other methods total times are dominated by their reconstruction. This is as expect as higher order schemes apply extra calculations at each cell, which is multiplied over the total number of cells and time steps. The lower plot shows the relationship of the other simulation processes, these appear independent of the choice of reconstruction.

The Re Di Roma roundabout network was simulated for varying Riemann solvers, the solutions shown in Figure 5.10, are timed and shown here in Figure 5.20. With second order reconstruction, the Riemann solvers crucially can be more or less costly than the reconstruction procedure. In the cases of the Lax-Friedrichs and Murman-Roe solvers the flux evaluation is less costly than reconstruction whereas the HLL solver is slightly more costly than its reconstruction. The total time reflects a combination of these two important simulation procedures. Even though the Murman-Roe solver is quicker than HLL, the total times are similar due to the slower Murman-Roe reconstruction. It is hence clear that at the 2^{nd} order level, the flux evaluations and reconstruction times are not independent and do influence each other.

As with all of the time analysis thus far, it is clear that certain parameters of simulation influence the behaviour and the breakdown of total time into the key simulation sections. Figure 5.21 shows pie charts of the reconstruction, junction solver, flux evaluations, Runge-Kutta update and the saving of simulated data. The only other elements of simulation are road network definition, checking for errors and initialising the density profile, these are omitted from the pie charts as they are of the scale of 10^{-3} smaller than the smallest main element, the junction solver. The Wakefield motorway junction network solution with second order reconstruction (top-left) shows how dominating the reconstruction process can be, compared to a first order reconstruction process (top-right). The lower pie shows the increased flux evaluation time with the smallest grid on the traffic circles network tests.

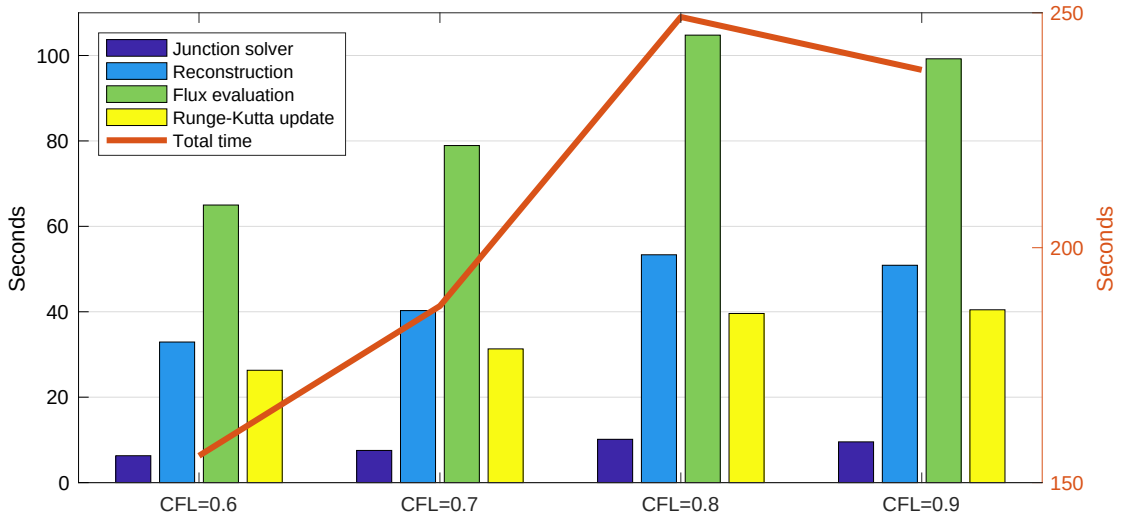


Figure 5.18: Higher CFL number increases computational time in total and is reflected over all timed segments of the code. The higher $CFL = 0.8, 0.9$ show a slightly opposite pattern. This information is joint with density profiles in Figure 5.13.

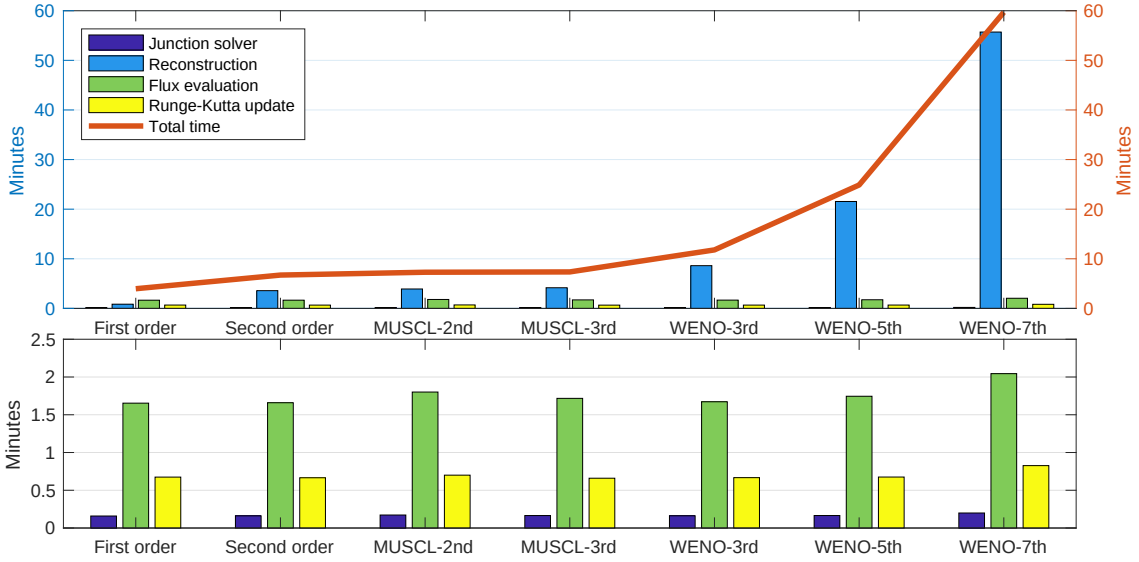


Figure 5.19: As reconstruction applies to every cell at every time step, any extra calculations computed as part of the reconstruction process scales with the number of spatial and time steps. Here it is clear that higher order reconstructions increase the reconstruction time (above), and the other simulation processes are unaffected by this (below).

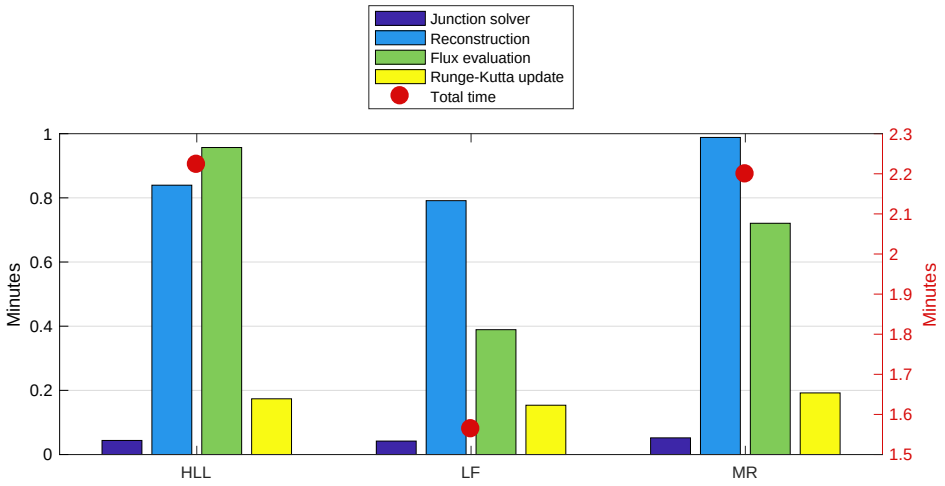


Figure 5.20: The Lax-Friedrichs solver is just over half as quick as the HLL and Murman-Roe solvers. If the choice of Riemann solver has little effect then it may be sensible to minimise computational time with the Lax-Friedrichs solver.

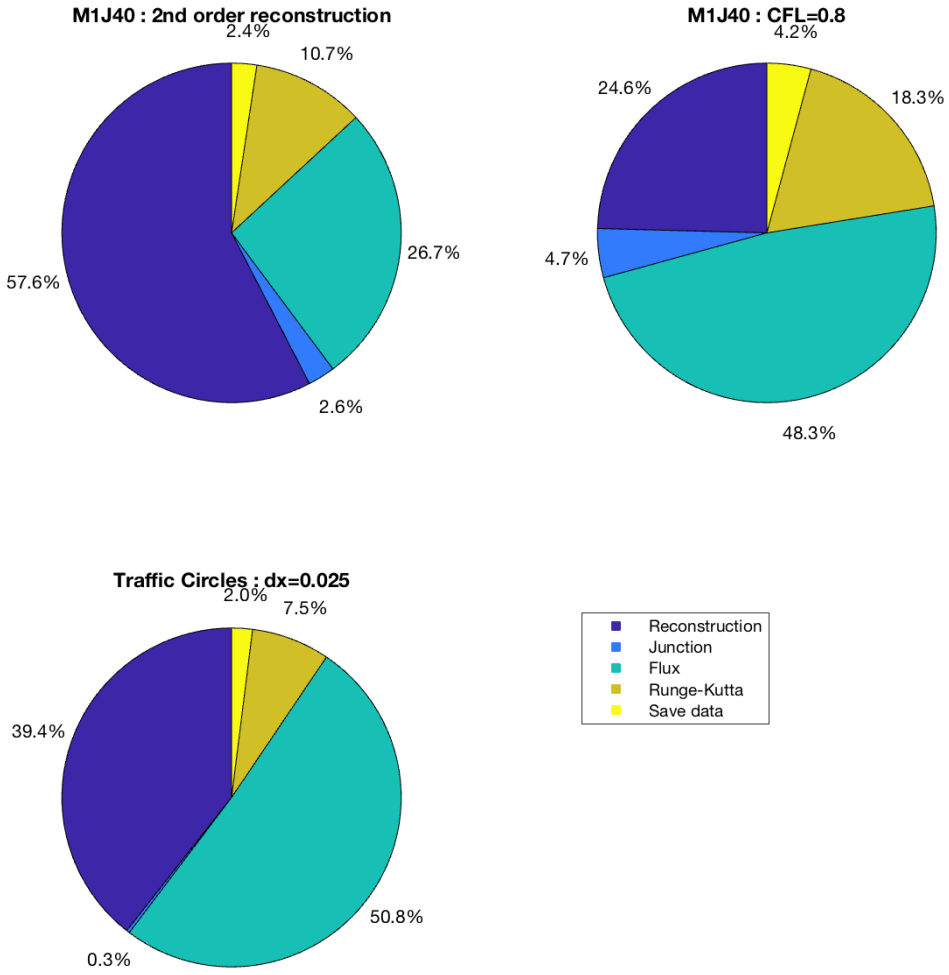


Figure 5.21: Depending on the choice of simulation parameters, the total simulation time is distributed varyingly over the sections shown in the above legend. For higher order reconstruction, this can dominate almost entirely, but other elements of the simulation are more time-influential with low order reconstructions.

Chapter 6

Final Remarks

6.1 Conclusions

The following conclusions can be made about the topics covered:

- There are many approaches to solving compressible flow problems, different combinations of numerical schemes, Riemann solvers and slope limiters all interact and give varying solutions of common problems. Many researchers have reviewed existing methods and developed new ideas to combat the issues of previous schemes.
- The choice of numerical methods in traffic flow simulations is more significant with volatile flows changing quickly, in long simulations where flow changes are very gradual the effect of numerical methods are dulled.
- The general WENO scheme as presented in Appendix B.2 cannot be extended to higher orders of $k \geq 4$ without applying some bounding to preserve monotonicity as in Appendix B.2.1 from Balsara and Shu [5].
- From the analysis of MUSCL slope limiters in Figures 5.5 and 5.16, the Van-Leer [72] limiter is identified as most advantageous, giving a high resolution solution of the density drop, a shorter computational cost, and with little 3^{rd} order speedup.

6.2 Further Ideas

The following recommendations are made for future research:

- More suitable parameters related to traffic flow quantities are required to be used as parameters for Riemann solvers to obtain varying results and to test which Riemann solvers are more or less appropriate for TFM. Once such parameters have been established, the HLLC solver (Appendix C.1) can be implemented and tested.

- Improve MATLAB scripts to read in road network information to split up roads automatically depending on number of roads, their individual length and the spatial step dx.
- An interesting simulation may be over a long time period such as 24 hours with a time-dependent TDM, using the Wakefield M1 junction as a network.
- Revise the junction solver and check for conservation of density, aiming to investigate the spike of density shown in Figure 5.13 and 5.14.
- Implement and test some other stream models as listed in Section 2.3, to identify in which scenarios other stream models perform better. This test can be backed by some empirical data collected on real roads as has been done in the reviews [3], [63], [40], [64].
- A test of the suitability of the LWR model [38],[52] against the Payne-high-order model [48] and some empirical traffic data would be useful to investigate if it is the underlying mathematical description of traffic which influences a simulation results more than the chosen numerical methods to proceed through this description.
- Investigate the effect of other Runge-Kutta time update schemes against the classical fourth order method used here (Section 3.6). Runge-Kutta schemes can be written into an adaptive scheme by comparing a 7^{th} and 8^{th} order update (for example) for the error and adapting the time step accordingly, alternatively such as the third order stability preserving method used in [58],

References

[1] J.D. Anderson. *Modern Compressible Flow : With Historical Perspective*. McGraw-Hill : Boston, 3 edition, 2004.

[2] S. Ardekani and M. Ghandehari. A modified greenberg speed-flow traffic model. *Technical Report, Department of Mathematics, The University of Texas at Arlington, USA*, 357, 2008.

[3] S. Ardekani, M. Ghandehari, and S. Nepal. Macroscopic speed-flow models for characterization of freeway and managed lanes. *Institutul Politehnic din Iasi. Buletinul. Sectia Constructii. Arhitectura*, 57(1):149–159, 2011.

[4] A. Aw and M. Rascle. Resurrection of 'second order' models of traffic flow. *SIAM Journal of Applied Mathematics*, 60(3):916–938, 2000.

[5] D.S. Balsara and C.W. Shu. Monotonicity preserving weighted essentially non-oscillatory schemes with increasingly high order of accuracy. *Journal of Computational Physics*, 160:405–452, 2000.

[6] T.J. Barth and D.C. Jespersen. The design and application of upwind schemes on unstructured meshes. *AIAA*, pages 89–0366, 1989.

[7] T. Bellemans, B. De Schutter, and B. De Moor. Models for traffic control. *Journal A*, 43(3-4):13–22, 2002.

[8] G. Bretti, R. Natalini, and B. Piccoli. Numerical approximations of a traffic flow model on networks. *American Institute of Mathematical Sciences*, 1(1):57–84, 2006.

[9] G. Bretti, R. Natalini, and B. Piccoli. A fluid-dynamic traffic model on road networks. *Archives of Computational Methods in Engineering*, 14(2):139–172, 2007.

[10] S.R. Chakravarthy and S. Osher. High resolution applications of the osher upwind scheme for the euler equations. *Proceedings or AIAA 6th Computational Fluid Dynamics conference*, 363-373:AIAA Paper 83–1943, 1983.

[11] T.J. Chung. *Computational Fluid Dynamics*. Cambridge University Press, 2002.

[12] B. Cockburn and C.W. Shu. Runge-kutta discontinuous galerkin methods for convection-dominated problems. *Journal of Scientific Computing*, 16(3):173–261, 2001.

[13] C.O. Costa-Luis. tqdm: A fast, extensible progress meter for python and cli. *The Journal of Open Source Software*, 4(37):1277, 2019.

[14] G. Costeseque. Traffic-flow-simulator, 11/09/2017. `https://github.com/gcostese/Traffic-flow-simulator` [Accessed 18-August-2019].

[15] C.F. Daganzo. The cell transmission model: A dynamic representation of highway traffic consistent with the hydrodynamic theory. *Transportation Research Part B: Methodological*, 28(4):269–287, 1994.

[16] S.F. Davis. Simplified second order godunov-type methods. *SIAM Journal on Scientific and Statistical Computing*, 9:445–473, 1988.

[17] S.L. Dhingra and G. Ishtiyaq. Traffic flow theory historical research perspectives. *Greenshields 75th Symposium*, pages 45–62, 2011.

[18] A. Dixon. Tfm_thesis, 05/08/2019. `https://github.com/adj97/TFM_Thesis` [Accessed 19-August-2019].

[19] J.S. Drake, J.L. Schofer, and May A.D. A statistical analysis of speed density hypotheses. *Highway Research Board, NRC, Washington D.C.*, 154, 1967.

[20] P.H. Gaskell and A.K.C. Lau. Curvature-compensated convective transport: Smart, a new boundedness-preserving transport algorithm. *International Journal of Numerical Methods of Fluids*, 8(6):617–641, 1988.

[21] S.K. Godunov. A finite difference method for the computation of discontinuous solutions of the equations of fluid dynamics. *Matematicheskii Sbornik*, 47:357–393, 1959, Translated by I. Bohachevsky.

[22] H. Greenberg. An analysis of traffic flow. *Operations Research*, 7(1):79–85, 1959.

[23] B.D. Greenshields. A study of traffic capacity. *Highway Research Board Proceedings*, 14:348–477, 1935.

[24] G.W. Griffiths. Limiter plots, 10/2006. `https://en.wikipedia.org/wiki/File:LimiterPlots1.png` [Accessed 7-August-2019].

[25] A. Harten, P.D. Lax, and B. van Leer. On upstream differencing and godunov-

type schemes for hyperbolic conservation laws. *SIAM Review*, 25(1):35–61, 1983.

[26] S.P. Hoogendoorn and P.H.L. Bovy. Generic gas-kinetic traffic systems modelling with applications to vehicular traffic flow. *Transport Research Part B*, 35(2001):317–336, 1999.

[27] S. Jayanti. *Computational Fluid Dynamics for Engineers and Scientists*. Springer : The Netherlands, 2018.

[28] JetBrains. Pycharm, 19/08/2019. `https://www.jetbrains.com/pycharm/` [Accessed 19-August-2019].

[29] M.J. Kermani, A.G. Gerber, and J.M. Stockie. Thermodynamically based moisture prediction using roe's scheme. *4th Conference of Iranian AeroSpace Society*, Amir Kabir University of Technology, Tehran, Iran, January 2729.

[30] B. Koren. A robust upwind discretisation method for advection, diffusion and source terms. *Vreugdenhil, (eds.), Numerical Methods for AdvectionDiffusion Problems*, page 117, 1993.

[31] R.D. Kuhne. Greenshields' legacy : Highway traffic. *Greenshields 75th Symposium*, pages 3–10, 2011.

[32] W. Kutta. Beitrag zur naherungsweisen integration totaler differentialgleichungen. *Mathematical Physics*, 46:435–453, 1901.

[33] P. Lakhanpal. Numerical simulations of traffic flow models. Master's thesis, Department of Mathematical Sciences, University of Nevada, Las Vegas, USA, August 2014.

[34] C.B. Laney. *Computational Gas Dynamics*. Cambridge University Press, 1998.

[35] J.A. Laval and C.F. Daganzo. Lane-changing in traffic streams. *Transportation Research Part B: Methodological*, 40(3):251–264, 2006.

[36] P.D. Lax. Weak solutions of nonlinear hyperbolic equations and their numerical computation. *Communications on Pure and Applied Mathematics*, 7:159–193, 1954.

[37] F.S. Lien and M.A. Leschziner. Upstream monotonic interpolation for scalar transport with application to complex turbulent flows. *International Journal of Numerical Methods of Fluids*, 19(6):527–548, 1994.

[38] M.J. Lighthill and G.B. Whitham. On kinematic waves ii. a theory of traffic flow on long crowded roads. *Proceedings of the Royal Society A*, 229(1178), 1955.

[39] X.D. Liu, S. Osher, and T. Chan. Weighted essentially non-oscillatory schemes. *Journal of Computational Physics*, 115:200–212, 1994.

[40] Z. Lu and Q. Meng. Analysis of traffic speed-density regression models - a case study of two roadway traffic flows in china. *Proceedings of the Eastern Asia Society for Transportation Studies*, 9, 2013.

[41] M. Maciejewski. A comparison of microscopic traffic flow simulation systems for an urban area. *Transport Problems (Poland)*, 5(4):27–38, 2010.

[42] K. Michalak and C. Ollivier-Gooch. Limiters for unstructured higher-order accurate solutions of the euler equations. *AIAA Aerospace Sciences Meeting and Exhibit*, 46, 2008.

[43] S. Moutari and M. Rascle. A hybrid lagrangian model based on the aw-rascle traffic flow model. *SIAM Journal of Applied Mathematics*, 68(2):413–436, 2007.

[44] E.M. Murman. Analysis of embedded shock waves calculated by relaxation methods. *AIAA Journal*, 12:626–633, 1974.

[45] K. Nagel and M. Schreckenberg. A cellular automaton model for freeway traffic. *Journal de Physique I*, 2(12):2221–2229, 1992.

[46] G.F. Newell. Theory of highway traffic flow: 1945 to 1965. *Institute of Transportation Studies, University of California, Berkely.*

[47] P.H. Oosthuizen. *Compressible Fluid Flow.* McGraw-Hill : New York, 1997.

[48] H.J. Payne. Models of freeway traffic and control. *Mathematical Models of Publication Systems*, 1:51–61, Simulation Council Proceedings, 1971.

[49] A. Pell, A. Meingast, and O. Schauer. Trends in real-time traffic simulation. *Transportation Research Procedia : World Conference on Transport Research - Shanghai 10-15 July 2016*, 25:1477–1484, 2017.

[50] L.A. Pipes. Car-following models and the fundamental diagram of road traffic. *Transportation Research*, 1:21–29, 1967.

[51] J.J. Quirk. A contribution to the great riemann solver debate. *International Journal for Numerical Methods in Fluids*, 18:555–574, 1994.

[52] P.I. Richards. Shock waves on the highway. *Operations Research*, 4(1):42–51, 1956.

[53] P.L. Roe. Characteristic-based schemes for the euler equations. *Annual Review of Fluid Mechanics*, 18:337–365, 1986.

[54] C. Runge. Uber die numerishe auflosing von differentialgleichungen. *Mathe-*

matics Annual Review, 46:167–178, 1895.

[55] V.V. Rusanov. Calculation of interaction of non-steady shock waves with obstacles. *Journal of Computational Mathematical Physics*, 1:267–279, 1961.

[56] M. Saidallah, A. El Fergougui, and A.E. Elalaoui. A comparative study of urban road traffic simulators. *MATEC Web of Conferences ICTTE*, 81, 05002, 2016.

[57] E.B. Setiawan, D. Tarwidi, and R.F. Umbara. Numerical simulation of traffic flow via fluid dynamics approach. *International Journal of Computing and Optimization*, 3(1):93–104, 2016.

[58] Y. Shi and Y. Guo. A maximum-principle-satisfying finite volume compact-weno scheme for traffic flow model on networks. *Applied Numerical Mathematics*, 108:21–36, 2016.

[59] C.W. Shu. Essentially non-oscillatory and weighted essentially non-oscillatory schemes for hyperbolic conservation laws. *NASA Institute for Computer Applications in Science and Engineering*, 97-65, 1997. Available on `https://www3.nd.edu/~zxu2/acms60790S13/Shu-WENO-notes.pdf` [Accessed 6-August-2019].

[60] A. Suresh and H.T. Huynh. Accurate monotonicity-preserving schemes with runge-kutta time stepping. *Journal of Computational Physics*, 136:83–99, 1997.

[61] P.K. Sweby. High resolution schemes using flux-limiters for hyperbolic conservation laws. *SIAM Journal of Numerical Analysis*, 21(5):995–1011, 1984.

[62] P.A. Thompson. *Compressible-Fluid Dynamics*. McGraw-Hill : New York, 1972.

[63] H. Tiwari and A. Marsani. Calibration of conventional macroscopic traffic flow models for nepalese roads (a case study of jadibuti - suryabinayak section). *Department of Civil Engineering, Central Campus, Pulchowk, IOE, TU, Nepal - Proceedings of IOE Graduate Conference*, pages 225–232, 2014.

[64] V.M. Tom. Transportation systems engineering notes - traffic stream models. *Module Material*, February 19, 2014.

[65] E.F. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics : A Practical Introduction*. Springer, 3 edition, 2009.

[66] E.F. Toro, D.H. Peregrine, and G. Watson. Numerical solution of the shallow water equations on a beach using the weighted average flux method. *Computational Fluid Dynamics*, 1:495–502, 1992.

[67] E.F. Toro, M. Spruce, and W. Speares. Restoration of the contact surface in the hll-riemann solver. *Shock Waves*, 4:25–34, 1994.

[68] P. Tsoutsanis. Numerical modelling for steady and unsteady compressible flows. *Module Material*, Cranfield University, February 2019.

[69] J. Tu, C. Liu, and G.H. Yeoh. *Computational Fluid Dynamics : A Practical Approach*. Butterworth-Heinemann : San Diego, 3 edition, 2018.

[70] R.T. Underwood. Speed, volume, and density relationships, quality and theory of traffic flow. *Yale Bureau of Highway Traffic, New Haven*, pages 141–188, 1961.

[71] G.D. Van Albada, B. Van Leer, and W.W. Roberts. A comparative study of computational methods in cosmic gas dynamics. *Astronomy and Astrophysics*, 108:76–84, 1982.

[72] B. Van Leer. Towards the ultimate conservative difference scheme ii. upstream-centred finite-difference schemes for ideal compressible flow. *Journal of Computational Physics*, 14(4):361–370, 1974.

[73] B. Van Leer. Towards the ultimate conservative difference scheme iii. upstream-centred finite-difference schemes for ideal compressible flow. *Journal of Computational Physics*, 23(3):263–275, 1977.

[74] B. van Leer. Towards the ultimate conservative difference scheme v. a second order sequel to godunov's method. *Journal of Computational Physics*, 32:101–136, 1979.

[75] V. Venkatakrishnan. On the accuracy of limiters and convergence to steady-state solutions. *AIAA*, pages 93–0880, 1993.

[76] F. Wageningen-Kessels, H. Lint, K. Vuik, and S. Hoogendoorn. Genealogy of traffic flow models. *EURO Journal on Transportation and Logistics*, 4(4):445–473, 2015.

[77] N.P. Waterson and H. Deconink. A unified approach to the design and application of bounded higher-order convection schemes. *Von Karman Institute Preprint 1995-21*, 1995.

[78] P. Woodward and P. Colella. The numerical simulation of two-dimensional fluid flow with strong shocks. *Journal of Computational Physics*, 54:115–173, 1984.

[79] G. Zhou. Numerical simulations of physical discontinuities in single and multi-fluid flows for arbitrary mach numbers (phd thesis). *Goteborg, Sweden: Chalmers University of Tech*, 1995.

Appendix A

Computer Codes

A.1 Simulation Codes

A.1.1 *main.py*

```
#
#   Andrew Dixon
#
#   Created   03/05/2019
#   Modified  15/07/2019
#
#   Thesis project for Cranfield University
#   MSc in Computational Fluid Dynamics
#
#   ------------------------------------------------------------------
#     demand -->    --    --    --    --    --    --    supply -->
#   ------------------------------------------------------------------
#
#   demand of road i - the flow (veh/hr) demanded by road i from road i-1
#   supply of road i - the flow (veh/hr) supplied to road i+1 from road i
#
#   network demand/supply is inlet/outlet flow in/out of the sources/sinks
#

from __future__ import print_function    # print no new line
import os                                # standard
import datetime
import numpy as np                       # numerical programming
import math                              # mathematical tools
from tqdm import tqdm                    # progressbar in time loop
import time                              # recording execution time
import json                              # read json format parameter file
import define_map                        # separate python file for network and
    junction info
import WENOReconstruction                # |----------"----------| WENO
    reconstructions
import MUSCLReconstruction               # |----------"----------| WENO
    reconstructions

# Start time
time0 = time.time()

# DEVELOPMENT
# PLAYGROUND
playground = False
if playground:
    pass
    exit()
```

```
###############

# From define_map.py file
#   read network and junction info
network = define_map.network
junction_info = define_map.junction_info

# Check for input errors in define_map.py
error_info = {'mis_attr': 0,
              'attr_typ': 0,
              'tdm_row_sum': 0,
              'tdm_shape': 0,
              'rd_id_er': 0,
              'params': 0}
error_description = {'mis_attr': 'missing attribute',
                     'attr_typ': 'wrong attribute type',
                     'tdm_row_sum': 'TDM row sum error',
                     'tdm_shape': 'TDM shape error',
                     'rd_id_er': 'road-junction ID mismatch',
                     'params': 'badly defined parameter(s) in params.txt'}
error_message = ''
jn_inout = {'in': [], 'out': []}

# Allowed data types
allowed_types = (int, float)

for junction in junction_info:

    # Check all junction attributes are present
    junction_keys = list(junction_info[junction].keys())
    junction_keys.sort()
    if junction_keys != ['in', 'out', 'tdm']:
        error_message += 'Badly defined junction {} \n'.format(junction)
        error_info['mis_attr'] += junction_keys != ['in', 'out', 'tdm']

    # Check junction info data types
    for in_road in junction_info[junction]['in']+junction_info[junction]['out']:  #
         in/out roads
        if (not isinstance(in_road, int)) or in_road > len(network) or in_road <=
            0:
            error_info['attr_typ'] += 1
            error_message += 'Badly defined in/out road ID in junction {} \n'.
                format(junction)

    for row in junction_info[junction]['tdm'].tolist():  # tdm elements
        for a in row:
            if not isinstance(a, allowed_types):
                error_info['attr_typ'] += 1
                error_message += 'Badly defined TDM element, {}, in junction {} \n'
                    .format(a, junction)
        # Check row sum
        if abs(1 - sum(row)) >= 1e-3:
            error_info['tdm_row_sum'] += 1
            error_message += 'Incorrect TDM row sum, in junction {} \n'.format(
                junction)

    # Check TDM matches in/out roads
    shape = junction_info[junction]['tdm'].shape
    junction_roads = [len(junction_info[junction]['in']), len(junction_info[
        junction]['out'])]
    if (shape[0] != junction_roads[0]) or (shape[1] != junction_roads[1]):
        error_info['tdm_shape'] += 1
        error_message += 'Mismatch TDM shape and in/out roads, in junction {} \n'.
            format(junction)

    # In/Out list
    for in_id in junction_info[junction]['in']:
        jn_inout['in'].append(in_id)
    for out_id in junction_info[junction]['out']:
        jn_inout['out'].append(out_id)
```

```
104
105 for road in network:
106
107     # Check all road attributes are present
108     network_keys = list(network[road].keys())
109     network_keys.sort()
110     if network_keys != ['demand', 'dmax', 'length', 'sink', 'source', 'supply', '
            vmax']:
111         error_message += 'Badly defined road {} \n'.format(road)
112         error_info['mis_attr'] += network_keys != ['demand', 'length', 'sink', '
                source', 'supply', 'vmax']
113
114     # Check correct road attribute types
115     for attr in network_keys:
116         if attr in ['demand', 'supply']:
117             # Supply and demand should be functions of density
118             if not callable(network[road][attr]):
119                 error_info['attr_typ'] += 1
120                 error_message += 'Badly defined {} in road {} \n'.format(attr, road
                        )
121
122         elif attr in ['length', 'vmax', 'dmax']:
123             # Length and vmax should be real numbers
124             if (not isinstance(network[road][attr], allowed_types)) or network[road
                    ][attr] <= 0:
125                 error_info['attr_typ'] += 1
126                 error_message += 'Badly defined {} in road {} \n'.format(attr, road
                        )
127
128         elif attr in ['source', 'sink']:
129             # Source and sink values can be either 0 or a function
130             if not (network[road][attr] == 0 or callable(network[road][attr])):
131                 error_info['attr_typ'] += 1
132                 error_message += 'Badly defined {} in road {} \n'.format(attr, road
                        )
133
134     # Check road-junction index matching
135     if len(network) > 1:
136         if network[road]['source'] != 0:
137             # Source roads only go IN to junctions
138             if road in jn_inout['out']:
139                 error_info['rd_id_er'] += 1
140                 error_message += 'Bad ID for source road {}'.format(road)
141
142             # Source road goes in to ONE junction only
143             if jn_inout['in'].count(road) != 1:
144                 error_info['rd_id_er'] += 1
145                 error_message += 'Bad ID for source road {}'.format(road)
146
147         elif network[road]['sink'] != 0:
148             # Sink roads only come OUT of junctions
149             if road in jn_inout['in']:
150                 error_info['rd_id_er'] += 1
151                 error_message += 'Bad ID for sink road {}'.format(road)
152
153             # Sink road comes out of ONE junction only
154             if jn_inout['out'].count(road) != 1:
155                 error_info['rd_id_er'] += 1
156                 error_message += 'Bad ID for sink road {}'.format(road)
157
158         else:
159             # Inner roads should be on both sides
160             if jn_inout['in'].count(road)+jn_inout['out'].count(road) < 2:
161                 error_info['rd_id_er'] += 1
162                 error_message += 'Bad ID for internal road {}'.format(road)
163
164 # Parameter file errors
165 with open('params.txt') as file:
166
167     # Read and save parameters
```

```
168     params = json.load(file)
169
170     # Check for each parameter error
171     for element in params:
172         if element == 'dx' and (not isinstance(params[element], allowed_types) or
                params[element] <= 0):
173             error_info['params'] += 1
174             error_message += 'Bad parameter dx'
175
176         if element == 'T' and (not isinstance(params[element], allowed_types) or
                params[element] <= 0):
177             error_info['params'] += 1
178             error_message += 'Bad parameter T'
179
180         if element == 'CFL' and (not isinstance(params[element], allowed_types) or
                params[element] <= 0):
181             error_info['params'] += 1
182             error_message += 'Bad parameter CFL'
183
184         if element == 'velocity_model':
185             if (not params[element] in ['Greenshields']) or (not isinstance(params[
                    element], str)):
186                 error_info['params'] += 1
187                 error_message += 'Bad velocity model'
188
189         if element == 'riemann_solver':
190             if (not params[element] in ['LaxFriedrichs', 'HLL', 'Rusanov', 'Murman-
                    Roe']) \
191                     or (not isinstance(params[element], str)):
192                 error_info['params'] += 1
193                 error_message += 'Bad Riemann solver'
194
195         if element == 'reconstruction':
196             if (not params[element] in ['FirstOrder', 'SecondOrder', 'WENO3', '
                    WENO5', 'WENO7', 'MUSCL2', 'MUSCL3']) \
197                     or (not isinstance(params[element], str)):
198                 error_info['params'] += 1
199                 error_message += 'Bad reconstruction parameter'
200
201         if element == 'limiter':
202             if (not params[element] in ['Charm', 'HCUS', 'HQUICK', 'Koren', 'MinMod
                    ',
203                                         'MonotonizedCentral', 'Osher', 'Ospre',
204                                         'Smart', 'Superbee', 'Sweby', 'UMIST', '
                                                vanAlbada1',
205                                         'vanAlbada2', 'vanLeer']) or (not
                                                isinstance(params[element], str)):
206                 error_info['params'] += 1
207                 error_message += 'Bad slope limiter parameter'
208
209 # Get total number of all errors
210 error_total = 0
211 for error_type in error_info:
212     error_total += error_info[error_type]
213
214 # Print input error and exit program
215 if error_total >= 1:
216     print('ERROR : Found', error_total, 'error(s)')
217     print('\nError breakdown:')
218     for err in error_info:
219         if error_info[err] != 0:
220             print('{} {}'.format(error_info[err], error_description[err]))  #
                    Breakdown
221     print('\nError message(s):')
222     print(error_message)  # Messages
223     exit(1)
224 else:
225     # No errors
226     pass
227
```

```
228
229 # Initialise for road loop
230 global_flows = {}  # To store each road's junction in/outflow
231 #
232 # global_flows is a nested dictionary which stores the supply/demand (or in/outflow
    ) of each road
233 # non-source/sink roads have supply/demand which is updated at each time step
    according to the junction function
234 # source or sink roads have a fixed demand or supply function respectively
235 # global_flows = {road_1: {'supply': ___, 'demand': ___},
236 #                  ...
237 #                 road_n: {'supply': ___, 'demand': ___}
238 #                }
239 #
240 V_max = network[1]['vmax']-1e-3  # Global maximum network speed
241 total_length = 0                 # Total network length
242 sources = []                     # list of source-road indexes
243 sinks = []                       # list of sink-road indexes
244 # Road loop
245 for road in network:
246
247     global_flows[road] = {}  # Each road is a dictionary with demand and supply
248
249     V_max = max(V_max, network[road]['vmax'])  # compare all speeds
250
251     total_length += network[road]['length']  # add up all road lengths
252
253     # Create source and sink lists
254     if network[road]['source'] != 0:
255         sources.append(road)
256     if network[road]['sink'] != 0:
257         sinks.append(road)
258
259 # Floating point error
260 total_length = float(round(total_length, 10))
261
262 # Geometry and junctions (map) defined, errors checked
263 time1 = time.time()
264
265 # Number of road sections
266 nb_link = len(network)
267
268 # Assign parameters
269 dx = params['dx']                               # Spatial resolution [km]
270 T = params['T']                                 # Final time [hr]
271 CFL = params['CFL']                             # Courant-Friedrichs-Lewy (CFL)
    safety factor constraint
272 reconstruction = params['reconstruction']      # Reconstruction scheme
273 velocity_model_id = params['velocity_model']  # Velocity-density model
274 riemann_solver = params['riemann_solver']      # Riemann solver / numerical flux
    calculator
275
276 # Time resolution from CFL constraint
277 dt = dx/(CFL*V_max)  # [hr]
278
279 # Spatial grid
280 x = np.arange(dx/2, total_length+dx/2, dx)
281
282 # Initial density profile
283 Rho_0 = np.zeros(len(x))
284
285 # INPUT
286 # Define initial global density profile
287 for i in range(0, len(x)):
288     Rho_0[i] = 0
289
290 # Set initial density as the first row of Rho array
291 n_t = len(np.arange(0, T, dt))
292 n_x = int(math.ceil(total_length/dx))
293 Rho = np.zeros((n_t, n_x))
```

```
294 Rho[0, ] = Rho_0
295
296 # WENO reconstruction definitions
297 weno3 = WENOReconstruction.weno3
298 weno5 = WENOReconstruction.weno5
299 weno7 = WENOReconstruction.weno7
300
301 # Minmod function
302 minmod = WENOReconstruction.minmod
303
304 # MUSCL reconstruction definitions
305 muscl2 = MUSCLReconstruction.muscl2
306 muscl3 = MUSCLReconstruction.muscl3
307
308 # Specific timers
309 junction_time = 0
310 reconstruction_time = 0
311 RK_update_time = 0
312 RiemProb_time = 0
313
314
315 # Get road start and end index function
316 def get_start_end(road_id):
317     fn_start = 0
318     for fn_road_index in range(1, road_id):
319         fn_start += int(network[fn_road_index]['length'] / dx)
320     fn_end = fn_start + int(network[road_id]['length'] / dx) - 1
321
322     return [fn_start, fn_end]
323
324
325 # Junction flow function
326 def junction(network_section, A, rho_0, junction_number):
327
328     # Get number of in and out roads
329     n_in = len(junction_info[junction_number]['in'])
330     n_out = len(junction_info[junction_number]['out'])
331
332     # Initialise demand(in)/supply(out) array
333     sup_dem = np.zeros(len(rho_0))
334
335     # Get demand and supply functions
336     # Evaluate at the appropriate boundary density
337     # Only need the demand of last cells on in roads
338     #           and supply of first cells on out roads
339     # Road IN - demand f'n
340     for df_i in range(0, n_in):
341         inroad_identifier = junction_info[junction_number]['in'][df_i]
342         sup_dem[df_i] = network_section[inroad_identifier]['demand'](rho_0[df_i])
343
344     # Road OUT - supply f'n
345     for sf_i in range(0, n_out):
346         outroad_identifier = junction_info[junction_number]['out'][sf_i]
347         sup_dem[n_in + sf_i] = network_section[outroad_identifier]['supply'](rho_0[
              n_in + sf_i])
348
349     # Initialise empty flows output array
350     flows = np.zeros(len(rho_0))
351
352     # Fill with values
353     for flow_i in range(0, len(flows)):
354         # Fill each flows element
355
356         if flow_i <= n_in-1:
357             # These are the supply of in-roads
358
359             # Smallest demands of all outgoing roads
360             sup_min_proportion = sup_dem[n_in]/A[flow_i, 0]
361             for out_i in range(1, n_out):
```

```
362                 sup_min_proportion = min(sup_min_proportion, sup_dem[out_i+n_in]/A[
                        flow_i, out_i])
363
364             # Calculate in-road new boundary flow
365             flows[flow_i] = min(sup_dem[flow_i], sup_min_proportion)
366         else:
367             # These are the demands of out-roads
368
369             # Sum of partial flow contribution from each in road
370             flows[flow_i] = 0
371             for in_i in range(0, n_in):  # for all in roads
372                 flows[flow_i] += A[in_i, flow_i-n_in]*flows[in_i]
373
374     return flows
375
376
377 # Velocity model function u=u(rho) and derivative
378 def velocity_model(road_section, density, derivative):
379
380     if velocity_model_id == 'Greenshields':
381
382         # Greenshield's
383         u_f = road_section['vmax']
384         d_m = road_section['dmax']
385         velocity_model_value = u_f*(1-density/d_m)
386
387         # Derivative
388         if derivative == 1:
389             velocity_model_value = - u_f / d_m
390
391     return velocity_model_value
392
393
394 # Compute the numerical flux (Riemann solvers)
395 def compute_flux():
396
397     # Initialise cell flux array
398     CellFluxes = np.zeros(len(rho_0))
399
400     # Loop through all cells
401     for cell_id in range(0, len(CellFluxes)):
402
403         # Reconstruction : Get L and R conserved(C) densities
404         CdL = reconstructed[cell_id, 1]
405         CdR = reconstructed[cell_id+1, 0]
406
407         # Get L and R physical flux(F)
408         FlL = CdL*velocity_model(network[road], CdL, 0)
409         FlR = CdR*velocity_model(network[road], CdR, 0)
410
411         # Calculate flux
412         if riemann_solver == 'LaxFriedrichs':
413
414             # S+ value
415             Splus = dx/dt
416
417             # Flux
418             flux_value = 0.5*((FlL+FlR)-Splus*(CdR-CdL))
419
420         elif riemann_solver == 'Rusanov':
421
422             # Left and right derivatives
423             dfL = velocity_model(network[road], CdL, 0) + CdL * velocity_model(
                        network[road], CdL, 1)
424             dfR = velocity_model(network[road], CdR, 0) + CdR * velocity_model(
                        network[road], CdR, 1)
425
426             # S+ value
427             Splus = max(dfL, dfR)
428
```

```
429             # Flux
430             flux_value = 0.5 * ((FlL + FlR) - Splus * (CdR - CdL))
431 
432         elif riemann_solver == 'Murman-Roe':
433 
434             # Constant interface
435             if CdL == CdR:
436 
437                 # Velocity
438                 a_speed = velocity_model(network[road], CdL, 0) + CdL *
                        velocity_model(network[road], CdL, 1)
439 
440                 # Flux
441                 if a_speed > 0:
442                     flux_value = FlL
443                 else:
444                     flux_value = FlR
445 
446             else:
447 
448                 # Velocity
449                 a_speed = (FlL-FlR)/(CdL-CdR)
450 
451                 # Flux
452                 flux_value = 0.5 * ((FlL + FlR) - abs(a_speed) * (CdR - CdL))
453 
454         elif riemann_solver == 'HLL':
455 
456             # Fastest signals
457             S_L = velocity_model(network[road], CdL, 0) + CdL * velocity_model(
                    network[road], CdL, 1)
458             S_R = velocity_model(network[road], CdR, 0) + CdR * velocity_model(
                    network[road], CdR, 1)
459 
460             # Flux
461             if S_L >= 0:
462                 flux_value = FlL
463             elif S_L < 0 and S_R >= 0:
464                 flux_value = (S_R*FlL - S_L*FlR + S_L*S_R*(CdR-CdL))/(S_R-S_L)
465             else:
466                 flux_value = FlR
467 
468         CellFluxes[cell_id] = flux_value
469 
470     return CellFluxes
471 
472 
473 # Spatial reconstruction
474 def spatial_reco(reconstruction_type):
475 
476     # Create array
477     output_array = np.zeros((len(rho_0)+1, 2))
478 
479     # Loop over all cells
480     for cell_id in range(0, len(output_array)):
481 
482         if reconstruction_type == 'FirstOrder':
483             # First order reconstruction
484             left = rho_ghost[cell_id]
485             right = rho_ghost[cell_id]
486 
487         elif reconstruction_type == 'SecondOrder':
488             # 2nd order minmod reconstruction
489 
490             # Right
491             xmr = rho_ghost[cell_id] - rho_ghost[cell_id - 1]
492             ymr = rho_ghost[cell_id + 1] - rho_ghost[cell_id]
493             right = rho_ghost[cell_id] + 0.5 * minmod(xmr, ymr)
494 
495             # Left
```

```
496             xml = rho_ghost[cell_id] - rho_ghost[cell_id-1]
497             yml = rho_ghost[cell_id + 1] - rho_ghost[cell_id]
498             left = rho_ghost[cell_id] - 0.5 * minmod(xml, yml)
499 
500         elif reconstruction_type == 'WENO3':
501             # 5th-Order Weighted Essentially Non-Oscillatory reconstruction
502 
503             [left, right] = weno3(cell_id, rho_ghost)
504 
505         elif reconstruction_type == 'WENO5':
506             # 5th-Order Weighted Essentially Non-Oscillatory reconstruction
507 
508             [left, right] = weno5(cell_id, rho_ghost)
509 
510         elif reconstruction_type == 'WENO7':
511             # 7th-Order Weighted Essentially Non-Oscillatory reconstruction
512 
513             [left, right] = weno7(cell_id, rho_ghost)
514 
515         elif reconstruction_type == 'MUSCL2':
516             # 2nd-Order Monotonic Upwind reconstruction Scheme for Conservation
                Laws
517 
518             [left, right] = muscl2(cell_id, rho_ghost)
519 
520         elif reconstruction_type == 'MUSCL3':
521             # 3rd-Order Monotonic Upwind reconstruction Scheme for Conservation
                Laws
522 
523             [left, right] = muscl3(cell_id, rho_ghost)
524 
525         output_array[cell_id] = [left, right]
526     return output_array
527 
528 
529 # Network global demand
530 def net_glob_demand(road, t):
531 
532     if road in sources:
533         n_g_d = global_flows[road]['demand'](t)
534     else:
535         n_g_d = global_flows[road]['demand']
536 
537     return n_g_d
538 
539 
540 # Network global supply
541 def net_glob_supply(road, t):
542 
543     if road in sinks:
544         n_g_s = global_flows[road]['supply'](t)
545     else:
546         n_g_s = global_flows[road]['supply']
547 
548     return n_g_s
549 
550 
551 # Runge-Kutta update constants
552 RKc = [[1, 0, 1],
553        [3/4, 1/4, 1/4],
554        [1/3, 2/3, 2/3]]
555 
556 
557 # Runge-Kutta global update loops
558 def RK_global_update(RK_step):
559 
560     # first cell
561     j = 0
562     use = f_demand_upstream - CellFluxes[j]
563     RK_rho[RK_step+1][j] = RKc[RK_step][0] * RK_rho[0][j] + \
```

```
564                             RKc[RK_step][1] * RK_rho[RK_step][j] + \
565                             RKc[RK_step][2] * (dt / dx) * use
566     if road in sources:
567         RK_rho[RK_step+1][j] = global_flows[road]['demand'](0)
568 
569     # internal cells
570     for j in range(1, n_x - 1):
571         use = CellFluxes[j - 1] - CellFluxes[j]
572         RK_rho[RK_step+1][j] = RKc[RK_step][0] * RK_rho[0][j] + \
573                                RKc[RK_step][1] * RK_rho[RK_step][j] + \
574                                RKc[RK_step][2] * (dt / dx) * use
575 
576     # last cell
577     j = n_x - 1
578     outflow = min(CellFluxes[j], f_supply_downstream)
579     use = CellFluxes[j - 1] - outflow
580     RK_rho[RK_step+1][j] = RKc[RK_step][0] * RK_rho[0][j] + \
581                             RKc[RK_step][1] * RK_rho[RK_step][j] + \
582                             RKc[RK_step][2] * (dt / dx) * use
583 
584     return RK_rho[RK_step+1]
585 
586 
587 # Functions and arrays created
588 time2 = time.time()
589 
590 # Time loop
591 # for t in np.arange(dt, T, dt):          # loop without progress bar
592 for t in tqdm(np.arange(dt, T, dt)):  # loop with progress bar
593 
594     # Iteration index from time t
595     i = int(round(t/dt)-1)
596 
597     # Loop for each junction
598     for junction_index in junction_info:
599 
600         # Extract from the junction information dict
601         tdm_array = junction_info[junction_index]['tdm']
602         roads_in = junction_info[junction_index]['in']
603         roads_out = junction_info[junction_index]['out']
604 
605         # Initialise an empty array for boundary densities
606         rho_0 = np.zeros(len(roads_in)+len(roads_out))
607 
608         # Fill these boundary densities
609         #   with the end of in-roads and the start of out-roads
610         for rho_0_index in range(0, len(rho_0)):
611 
612             # Get road value
613             road_identifier = (roads_in+roads_out)[rho_0_index]
614 
615             # Get road start and end indexes
616             [start, end] = get_start_end(road_identifier)
617 
618             # extract appropriate boundary elements
619             if rho_0_index <= len(roads_in)-1:
620                 # in-road end
621                 rho_0[rho_0_index] = Rho[i, end]
622             else:
623                 # out-road start
624                 rho_0[rho_0_index] = Rho[i, start]
625 
626         # Log junction time
627         junction_time -= time.time()
628 
629         # Call junction to get input/output flows
630         local_flows = junction(network, tdm_array, rho_0, junction_index)
631 
632         # Log junction time
633         junction_time += time.time()
```

```
634
635         # Store each road's new supply/demand in global_flows
636         for local_flow_i in range(0, len(local_flows)):
637             local_flow_val = local_flows[local_flow_i]
638
639             if local_flow_i < len(roads_in):
640                 # in-road
641                 road_identifier = junction_info[junction_index]['in'][local_flow_i]
642                 global_flows[road_identifier]['supply'] = local_flow_val
643             else:
644                 # out-road
645                 road_identifier = junction_info[junction_index]['out'][local_flow_i
                        -len(roads_in)]
646                 global_flows[road_identifier]['demand'] = local_flow_val
647
648     # Fill the network source/sink supply/demand values in global_flows
649     for source in sources:
650         global_flows[source]['demand'] = network[source]['source']
651     for sink in sinks:
652         global_flows[sink]['supply'] = network[sink]['sink']
653
654     # Each road separately
655     for road in network:
656
657         # Get start and end indexes
658         [start, end] = get_start_end(road)
659
660         # Update the new 'initial' density as the previous-time-step solution
661         rho_0 = Rho[i, start:end+1]
662
663         # number of x points
664         n_x = int(network[road]['length'] / dx)
665
666         # Get current road supply and demand
667         supply = network[road]['supply']
668         demand = network[road]['demand']
669
670         # types list for callable function or non-callable number
671         allowed_types = (np.float64, int, np.int64)
672
673         # incoming flow
674         f_demand_upstream = net_glob_demand(road, t)
675         f_supply_downstream = net_glob_supply(road, t)
676
677         # initialise RK-dict for density array
678         RK_rho = [[], [], [], []]
679         RK_rho[0] = rho_0.tolist()
680         RK_rho[1] = Rho[i + 1, start:end + 1].tolist()
681         RK_rho[2] = Rho[i + 1, start:end + 1].tolist()
682         RK_rho[3] = Rho[i + 1, start:end + 1].tolist()
683
684         # Runge-Kutta steps
685         for RK in range(0, 3):
686
687             # Chose the most updated array for flux calculation
688             rho_flux = RK_rho[RK]
689
690             # Add in ghost BCs
691             rho_ghost = np.append(rho_flux, [rho_flux[-2:-5:-1], rho_flux[3:0:-1]])
692
693             # Log reconstruction time
694             reconstruction_time -= time.time()
695
696             # Save all reconstructed L and R states in an array size (len(
                    CellFluxes), 2)
697             reconstructed = spatial_reco(reconstruction)
698
699             # Log reconstruction time
700             reconstruction_time += time.time()
701
```

```
            # Log Riemann problem time
            RiemProb_time -= time.time()

            # Compute each cell RHS interface flux
            CellFluxes = compute_flux()

            # Log Riemann problem time
            RiemProb_time += time.time()

            # Log Runge-Kutta update time
            RK_update_time -= time.time()

            # Update
            RK_rho[RK+1] = RK_global_update(RK)

            # Log Runge-Kutta update time
            RK_update_time += time.time()

        Rho[i+1, start:end+1] = RK_rho[3]

# Time loop completed
time3 = time.time()

# Read parameter file
with open('params.txt') as file:
    params = json.load(file)

# Find required slope limiter
chosen_limiter = params['limiter']
print_limiter = False  # default

# Chose to save or not
do_print = False

# Save density profile
if do_print:
    # Create output folder
    if not os.path.exists('Simulation_Results'):
        os.mkdir('Simulation_Results')
    path = 'Simulation_Results/'+str(datetime.datetime.now().strftime("%d-%m-%Y_%H
        -%M-%S"))
    os.mkdir(path)

    # Update the saved density profile in pwd
    density_output = path + '/density.txt'
    np.savetxt(density_output, Rho)

    # Results saved - program complete
    time4 = time.time()

    # Overwrite file (comment out)
    filename = path + '/simulation_info.txt'

    # Write and open file
    info_txt = open(filename, 'w+')
    info_txt.write('TFM Simulation Information \n\n')
    now = datetime.datetime.now()
    info_txt.write(now.strftime("%d/%m/%Y %H:%M:%S \n"))

    # Write reconstruction phrases
    if reconstruction == 'FirstOrder':
        reconstruction = '1st Order single cell average'
    elif reconstruction == 'SecondOrder':
        reconstruction = '2nd Order minmod interpolation'
    elif reconstruction == 'WENO3':
        reconstruction = '3rd Order WENO'
    elif reconstruction == 'WENO5':
        reconstruction = '5th Order WENO'
    elif reconstruction == 'WENO7':
        reconstruction = '7th Order WENO monotonicity preserving bounds'
```

```
771     elif reconstruction == 'MUSCL2':
772         reconstruction = '2nd Order MUSCL'
773         print_limiter = True
774     elif reconstruction == 'MUSCL3':
775         reconstruction = '3rd Order MUSCL'
776         print_limiter = True
777 
778     # Method information
779     info_txt.write('\n\nMethodology:\n')
780     info_txt.write('{:30s} {:.6e}\n'.format('Spatial step', dx))
781     info_txt.write('{:30s} {:.6e}\n'.format('Final time', T))
782     info_txt.write('{:30s} {:.6e}\n'.format('CFL constraint', CFL))
783     info_txt.write('{:30s} {:10s}\n'.format('Reconstruction method', reconstruction
        ))
784     if print_limiter:
785         info_txt.write('{:30s} {:10s}\n'.format('Slope limiter', chosen_limiter))
786 
787     # Time breakdown
788     info_txt.write('\n\nTime breakdown:\n')
789     info_txt.write('{:30s} {:.10s}\n'.format('Code Section', 'Time [s]'))
790     info_txt.write('-----------------------------------------------\n')
791     info_txt.write('{:30s} {:.6e}\n'.format('Defining map and error check', time1-
        time0))
792     info_txt.write('{:30s} {:.6e}\n'.format('Initialising', time2-time1))
793     info_txt.write('{:30s} {:.6e}\n'.format('Time loop', time3-time2))
794     info_txt.write('- - - - - - - - - - - - - - - - - - - - - - - - - - - \n')
795     info_txt.write('{:30s} {:.6e}\n'.format('Junction solver', junction_time))
796     info_txt.write('{:30s} {:.6e}\n'.format('Spatial reconstruction',
        reconstruction_time))
797     info_txt.write('{:30s} {:.6e}\n'.format('Numerical flux calculation',
        RiemProb_time))
798     info_txt.write('{:30s} {:.6e}\n'.format('Runge-Kutta updates', RK_update_time))
799     info_txt.write('- - - - - - - - - - - - - - - - - - - - - - - - - - - \n')
800     info_txt.write('{:30s} {:.6e}\n'.format('Save results', time4-time3))
801     info_txt.write('-----------------------------------------------\n')
802     info_txt.write('-----------------------------------------------\n')
803     info_txt.write('{:30s} {:.6e}\n'.format('Total program time', time4-time0))
804     info_txt.write('-----------------------------------------------\n')
805 
806     # File code line count
807     num_lines = {}
808     files = ['main.py', 'define_map.py', 'params.txt', 'MUSCLReconstruction.py', '
        WENOReconstruction.py']
809     for file in files:
810         num_lines[file] = 0
811         with open(file, 'r') as f:
812             for line in f:
813                 num_lines[file] += 1
814     num_lines['Total'] = 0
815 
816     info_txt.write('\n\nLine Count:\n')
817     info_txt.write('{: <30} {: <20}\n'.format('File', 'Number of Lines'))
818     info_txt.write('-----------------------------------------------\n')
819     for file in num_lines:
820         if file == 'Total':
821             info_txt.write('-----------------------------------------------\n')
822             info_txt.write('{: <30} {: <20}\n'.format(file, num_lines[file]))
823             info_txt.write('-----------------------------------------------\n')
824         else:
825             info_txt.write('{: <30} {: <20}\n'.format(file, num_lines[file]))
826         num_lines['Total'] += num_lines[file]
827 
828     # Write memory information
829     info_txt.write('\n\nOutput File Memory:\n')
830     info_txt.write('{:30} {:}\n'.format('File', 'Size [MB]'))
831     info_txt.write('-----------------------------------------------\n')
832     info_txt.write('{0:30} {1:5.3f}\n'.format('density.txt', os.stat('density.txt')
        .st_size / 1000000))
833     info_txt.write('-----------------------------------------------\n')
834 
```

```
835     # Close file
836     info_txt.close()
837 
838 else:
839     # Developing runs
840     os.remove('density.txt')
841     np.savetxt('density.txt', Rho)
842 
843     # Results saved - program complete
844     time4 = time.time()
845 
846     # Wait 1 second
847     time.sleep(1)
848 
849     # Print basic info to console
850     print('')
851     print('|- - - - - Simulation Complete - - - - -|')
852     print(' {:25s} {:.6e}'.format('Spatial step', dx))
853     print(' {:25s} {:.6e}'.format('Final time', T))
854     print(' {:25s} {:.6e}'.format('CFL', CFL))
855     print(' {:25s} {:6s}'.format('Velocity Model', velocity_model_id))
856     print(' {:25s} {:6s}'.format('Riemann solver', riemann_solver))
857     print(' {:25s} {:6s}'.format('Reconstuction method', reconstruction))
858     if reconstruction == 'MUSCL2' or reconstruction == 'MUSCL3':
859         print(' {:25s} {:6s}'.format('Slope limiter', chosen_limiter))
860     print('|- - - - - - - - - - - - - - - - - - - -|')
861 
862 exit()
```

A.1.2 *define_map.py*

```
import numpy as np           # numerical programming

# Define Road Characteristics
# Each road must have:
#   length - road segment length (in km)
#   vmax   - max speed (in km/h)
#   source - binary selection if source(1)/not(0)
#   sink   - binary selection if sink(1)/not(0)
#   demand - demand function of density
#   supply - supply function of density
#
# If a road is a source or sink:
#   change source(1)/sink(1) to the function
#   demand_upstream (source) or supply_downstream (sink)
#   a function of time and returns a flow (in veh/hr)
#
# Define a new road by network[road] with road = 1, 2, 3, ..., n
# the network object is a nested dictionary
# network = {road_1: {'length_1': ___, 'vmax_1': ___, 'source_1': ___, 'sink_1':
    ___, 'demand_1': ___, 'supply_1': ___},
#             ...
#           road_n: {'length_n': ___, 'vmax_n': ___, 'source_n': ___, 'sink_n':
    ___, 'demand_n': ___, 'supply_n': ___}
#          }

network = {}

# Road Template
# network[1] = {'length': 5, 'vmax': 90, 'dmax': 30, 'source': 1, 'sink': 1}
# def demand(rho): return (90*rho)*(rho <= 30) + 2700*(rho > 30)
# def supply(rho): return 2700*(rho <= 30) + (15*(30-rho)+2700)*(rho > 30)
# network[1]['demand'] = demand
# network[1]['supply'] = supply
# def demand_upstream(): return 0
# network[1]['source'] = demand_upstream
# def supply_downstream(): return 10000000000000
# network[1]['sink'] = supply_downstream

# Demand and Supply - density relations

def demand(rho): return (90*rho)*(rho <= 30) + 2700*(rho > 30)
def supply(rho): return 2700*(rho <= 30) + (15*(30-rho)+2700)*(rho > 30)

# Sink condition

def supply_downstream(t): return 1e12

# MOTORWAY ROAD SOURCE/SINKS

network[1] = {'length': 0.5, 'vmax': 112, 'dmax': 208, 'source': 0, 'sink': 1}
network[1]['demand'] = demand
network[1]['supply'] = supply
network[1]['sink'] = supply_downstream

network[2] = {'length': 0.5, 'vmax': 112, 'dmax': 208, 'source': 1, 'sink': 0}
network[2]['demand'] = demand
network[2]['supply'] = supply
def demand_upstream(t): return 0.2
network[2]['source'] = demand_upstream

network[3] = {'length': 0.5, 'vmax': 112, 'dmax': 208, 'source': 1, 'sink': 0}
network[3]['demand'] = demand
network[3]['supply'] = supply
def demand_upstream(t): return 0.1
network[3]['source'] = demand_upstream

network[4] = {'length': 0.5, 'vmax': 112, 'dmax': 208, 'source': 0, 'sink': 1}
network[4]['demand'] = demand
```

```
67  network[4]['supply'] = supply
68  network[4]['sink'] = supply_downstream
69
70  # MOTORWAY INNER SECTION
71
72  network[5] = {'length': 0.97, 'vmax': 112, 'dmax': 208, 'source': 0, 'sink': 0}
73  network[5]['demand'] = demand
74  network[5]['supply'] = supply
75
76  network[6] = {'length': 0.97, 'vmax': 112, 'dmax': 208, 'source': 0, 'sink': 0}
77  network[6]['demand'] = demand
78  network[6]['supply'] = supply
79
80  # SLIP ROADS
81
82  network[7] = {'length': 0.48, 'vmax': 112, 'dmax': 208, 'source': 0, 'sink': 0}
83  network[7]['demand'] = demand
84  network[7]['supply'] = supply
85
86  network[8] = {'length': 0.32, 'vmax': 112, 'dmax': 208, 'source': 0, 'sink': 0}
87  network[8]['demand'] = demand
88  network[8]['supply'] = supply
89
90  network[9] = {'length': 0.32, 'vmax': 112, 'dmax': 208, 'source': 0, 'sink': 0}
91  network[9]['demand'] = demand
92  network[9]['supply'] = supply
93
94  network[10] = {'length': 0.48, 'vmax': 112, 'dmax': 208, 'source': 0, 'sink': 0}
95  network[10]['demand'] = demand
96  network[10]['supply'] = supply
97
98  # A-ROAD SOURCE/SINKS
99
100 network[11] = {'length': 0.2, 'vmax': 80, 'dmax': 208, 'source': 1, 'sink': 0}
101 network[11]['demand'] = demand
102 network[11]['supply'] = supply
103 def demand_upstream(t): return 0.05
104 network[11]['source'] = demand_upstream
105
106 network[12] = {'length': 0.2, 'vmax': 80, 'dmax': 208, 'source': 0, 'sink': 1}
107 network[12]['demand'] = demand
108 network[12]['supply'] = supply
109 network[12]['sink'] = supply_downstream
110
111 network[13] = {'length': 0.2, 'vmax': 80, 'dmax': 208, 'source': 0, 'sink': 1}
112 network[13]['demand'] = demand
113 network[13]['supply'] = supply
114 network[13]['sink'] = supply_downstream
115
116 network[14] = {'length': 0.2, 'vmax': 80, 'dmax': 208, 'source': 1, 'sink': 0}
117 network[14]['demand'] = demand
118 network[14]['supply'] = supply
119 def demand_upstream(t): return 0.1
120 network[14]['source'] = demand_upstream
121
122 # ROUNDABOUT SECTIONS
123
124 network[15] = {'length': 0.13, 'vmax': 64, 'dmax': 208, 'source': 0, 'sink': 0}
125 network[15]['demand'] = demand
126 network[15]['supply'] = supply
127
128 network[16] = {'length': 0.03, 'vmax': 64, 'dmax': 208, 'source': 0, 'sink': 0}
129 network[16]['demand'] = demand
130 network[16]['supply'] = supply
131
132 network[17] = {'length': 0.08, 'vmax': 64, 'dmax': 208, 'source': 0, 'sink': 0}
133 network[17]['demand'] = demand
134 network[17]['supply'] = supply
135
136 network[18] = {'length': 0.03, 'vmax': 64, 'dmax': 208, 'source': 0, 'sink': 0}
```

```
137 network[18]['demand'] = demand
138 network[18]['supply'] = supply
139
140 network[19] = {'length': 0.13, 'vmax': 64, 'dmax': 208, 'source': 0, 'sink': 0}
141 network[19]['demand'] = demand
142 network[19]['supply'] = supply
143
144 network[20] = {'length': 0.03, 'vmax': 64, 'dmax': 208, 'source': 0, 'sink': 0}
145 network[20]['demand'] = demand
146 network[20]['supply'] = supply
147
148 network[21] = {'length': 0.08, 'vmax': 64, 'dmax': 208, 'source': 0, 'sink': 0}
149 network[21]['demand'] = demand
150 network[21]['supply'] = supply
151
152 network[22] = {'length': 0.03, 'vmax': 64, 'dmax': 208, 'source': 0, 'sink': 0}
153 network[22]['demand'] = demand
154 network[22]['supply'] = supply
155
156 # Define Junction Characteristics
157 # To unambiguously describe a traffic network, a description of the junctions is
      required
158 # Each junction i in (1, n) must be prescribed:
159 #   in  - a list of the road indexes which feed traffic IN to junction i
160 #   out - a lsit of the road indexes which lead OUT from junction i
161 #   tdm - a unique traffic distribution matrix
162 #
163 # The Traffic Distribution Matrix, A
164 #   The general junction with m incoming roads and n outgoing roads
165 #   A has m rows, n columns
166 #   The elements A=(a)_mn describe the proportion of traffic leaving incoming road
      m and travelling on outgoing road n
167 #   The sum of a_mn over index m is 1, as all traffic must leave its current road
      and also be conserved
168 #   The numpy syntax for arrays in 1D is [a, b, c, ...] so for 2D arrays each of a,
       b, c, .. are replaced by arrays:
169 #       [ [a11, a12, a13, ...], [a21, a22, a23, ...], [a31, a32, a33, ...], ... ]
170
171 junction_info = {}
172
173 # Junction template
174 # junction_info[id] = {'in': [ list of roads in by integer],
175 #                      'out': [ list of roads out by integer],
176 #                      'tdm': np.array([[first row of TDM elements], [second row of
      TDM elements], ...])}
177
178 # Junctions
179 junction_info[1] = {'in': [5,7], 'out': [1], 'tdm': np.array([[1], [1]])}
180 junction_info[2] = {'in': [2], 'out': [6,8], 'tdm': np.array([[0.9, 0.1]])}
181 junction_info[3] = {'in': [3], 'out': [5,9], 'tdm': np.array([[0.8, 0.2]])}
182 junction_info[4] = {'in': [6,10], 'out': [4], 'tdm': np.array([[1], [1]])}
183 junction_info[5] = {'in': [22], 'out': [7,15], 'tdm': np.array([[0.8, 0.2]])}
184 junction_info[6] = {'in': [8,15], 'out': [16], 'tdm': np.array([[1], [1]])}
185 junction_info[7] = {'in': [16], 'out': [13,17], 'tdm': np.array([[0.8, 0.2]])}
186 junction_info[8] = {'in': [14,17], 'out': [18], 'tdm': np.array([[1], [1]])}
187 junction_info[9] = {'in': [18], 'out': [10,19], 'tdm': np.array([[0.7, 0.3]])}
188 junction_info[10] = {'in': [9,19], 'out': [20], 'tdm': np.array([[1], [1]])}
189 junction_info[11] = {'in': [20], 'out': [12,21], 'tdm': np.array([[0.7, 0.3]])}
190 junction_info[12] = {'in': [11,21], 'out': [22], 'tdm': np.array([[1], [1]])}
```

A.1.3 *MUSCLReconstruction.py*

```
import json         # read json format parameter file

# Monotonic Upwind Scheme for Conservation Laws
# As formulated in
#    Linear and Parabolic Reconstruction,
#    van Leer, B.
#    1979,
#    Towards the Ultimate Conservative Difference Scheme,
#    V. A Second Order Sequel to Godunov's Method,
#    J. Com. Phys,
#    32,
#    101-136
#
# Available on https://en.wikipedia.org/wiki/MUSCL_scheme


# 2nd Order
def muscl2(cell, variable_array):

    # STEP 1
    # Construct required density values
    # rho_muscl[i] is equivalent to rho_{cell+i} for i={-1, 0, 1}
    rho_muscl = [variable_array[cell],       # center cell
                 variable_array[cell+1],     # right cell
                 variable_array[cell-1]]    # left cell

    # Avoid division by zero
    epsilon = 1e-6

    # STEP 2
    # Compute r_i
    r_i = (rho_muscl[0]-rho_muscl[-1])/(rho_muscl[1]-rho_muscl[0]+epsilon)

    # STEP 3
    # Calculate reconstructed values
    left_reco = rho_muscl[0]-0.5*limiter(r_i)*(rho_muscl[1]-rho_muscl[0])
    right_reco = rho_muscl[0] + 0.5 * limiter(r_i) * (rho_muscl[1] - rho_muscl[0])

    # End function
    return [left_reco, right_reco]


# 3rd Order
def muscl3(cell, variable_array):

    # STEP 1
    # Construct required density values
    # rho_muscl[i] is equivalent to rho_{cell+i} for i={-1, 0, 1}
    rho_muscl = [variable_array[cell],       # center cell
                 variable_array[cell+1],     # right cell
                 variable_array[cell-1]]    # left cell

    # Avoid division by zero
    epsilon = 1e-6

    # STEP 2
    # Compute r_i
    r_i = (rho_muscl[0]-rho_muscl[-1])/(rho_muscl[1]-rho_muscl[0]+epsilon)

    # STEP 3
    # Compute delta values
    delta_r = rho_muscl[1]-rho_muscl[0]
    delta_l = rho_muscl[0]-rho_muscl[-1]

    # Constant
    kappa = 1/3

    # STEP 4a
```

```
69      # Partial reconstructed values
70      left_reco = (1-kappa)*delta_r+(1+kappa)*delta_l
71      right_reco = (1-kappa)*delta_l+(1+kappa)*delta_r
72
73      # STEP 4b
74      # Final reconstructed values
75      left_reco = rho_muscl[0] - 0.25*limiter(r_i)*left_reco
76      right_reco = rho_muscl[0] + 0.25*limiter(r_i)*right_reco
77
78      # End function
79      return [left_reco, right_reco]
80
81
82  # Read parameter file
83  with open('params.txt') as file:
84      params = json.load(file)
85
86  # Find required slope limiter
87  chosen_limiter = params['limiter']
88
89
90  # Slope Limiter
91  def limiter(r):
92      # These limiters are given in https://en.wikipedia.org/wiki/Flux_limiter
93
94      if chosen_limiter == "Charm":
95          # Charm
96          # Zhou, 1995
97          # not 2nd order TDV accurate
98          limited_slope = r*(3*r+1)/((r+1)**2) if r > 0 else 0
99
100     elif chosen_limiter == "HCUS":
101         # HCUS
102         # Waterson & Deconinck, 1995
103         # not 2nd order TDV accurate
104         limited_slope = 1.5*(r+abs(r))/(r+2)
105
106     elif chosen_limiter == "HQUICK":
107         # HQUICK
108         # Waterson & Deconinck, 1995
109         # not 2nd order TDV accurate
110         limited_slope = 2*(r+abs(r))/(r+3)
111
112     elif chosen_limiter == "Koren":
113         # Koren
114         # Koren, 1993
115         # 3rd order TDV accurate for sufficiently smooth data
116         limited_slope = max(0, min(2*r, min((1+2*r)/3, 2)))
117
118     elif chosen_limiter == "MinMod":
119         # MinMod
120         # Roe, 1986
121         # 2nd order TDV accurate
122         limited_slope = max(0, min(1, r))
123
124     elif chosen_limiter == "MonotonizedCentral":
125         # Monotonized Central
126         # van Leer, 1977
127         # 2nd order TDV accurate
128         limited_slope = max(0, min(2*r, (1+r)/2, 2))
129
130     elif chosen_limiter == "Osher":
131         # Osher
132         # Chakravarthy & Osher, 1983
133         # 2nd order TDV accurate
134         b = 1.5
135         limited_slope = max(0, min(r, b))
136
137     elif chosen_limiter == "Ospre":
138         # Ospre
```

```
139         # Waterson & Deconinck, 1995
140         # 2nd order TDV accurate, symmetric
141         limited_slope = 1.5*(r**2+r)/(r**2+r+1)
142 
143     elif chosen_limiter == "Smart":
144         # Smart
145         # Gaskell & Lau, 1988
146         # not 2nd order TDV accurate
147         limited_slope = max(0, min(2*r, (1+3*r)/4, 4))
148 
149     elif chosen_limiter == "Superbee":
150         # Superbee
151         # Roe, 1986
152         # 2nd order TDV accurate, symmetric
153         limited_slope = max(0, min(2*r, 1), min(r, 2))
154 
155     elif chosen_limiter == "Sweby":
156         # Sweby
157         # Sweby, 1984
158         # not 2nd order TDV accurate, symmetric
159         b = 1.5
160         limited_slope = max(0, min(b*r, 1), min(r, b))
161 
162     elif chosen_limiter == "UMIST":
163         # UMIST
164         # Lien & Leschziner, 1994
165         # 2nd order TDV accurate
166         limited_slope = max(0, min(2*r, (1+3*r)/4, (3+r)/4, 2))
167 
168     elif chosen_limiter == "vanAlbada1":
169         # van Albada 1
170         # van Albada, et al., 1982
171         # 2nd order TDV accurate, symmetric
172         limited_slope = (r**2+r)/(r**2+1)
173 
174     elif chosen_limiter == "vanAlbada2":
175         # van Albada 2
176         # Kermani, 2003
177         # not 2nd order TDV accurate
178         # Alternate form for high spatial order schemes
179         limited_slope = (2*r)/(r**2+1)
180 
181     elif chosen_limiter == "vanLeer":
182         # van Leer
183         # van Leer, 1974
184         # 2nd order TDV accurate, symmetric
185         limited_slope = (r+abs(r))/(1+abs(r))
186 
187     else:
188         # Wrong slope limiter specifier
189 
190         print('ERROR: Wrong slope limiter specifier')
191         exit(1)
192 
193     return limited_slope
```

A.1.4 *WENOReconstruction.py*

```
import numpy as np  # numerical programming
import math         # mathematical functions

# Weighted Essentially Non-Oscillatory Reconstruction Schemes
# As formulated in
#    Procedure 2.2,
#    ENO and WENO Schemes for Hyperbolic Conservation Laws,
#    Chi-Wang Shu,
#    NASA/CR-97-206253,
#    ICASE Report No. 97-65,
#    November 1997
#
# Available on
#   https://www3.nd.edu/~zxu2/acms60790S13/Shu-WENO-notes.pdf


# 3rd Order
def weno3(cell, variable_array):

    # STEP 1
    # Construct required density values
    # rho_weno[i] is equivalent to rho_{cell+i} for i={-2, -1, 0, 1, 2}
    rho_weno = [variable_array[cell],                  # center cell
                variable_array[cell+1],      # right
                variable_array[cell-1]]     # left

    # STEP 2
    # Define known ENO coefficients
    # For ~c = c[r-1][j]
    c = [[1/2, 1/2],      # r=0
         [-1/2, 3/2],      # r=1
         [3/2, -1/2]]     # r=-1  (for the ~c transformation)

    # Generate 3 right and 3 left polynomial ENO reconstructed values
    rhoplusENO = [0, 0]  # right
    rhominENO = [0, 0]   # left

    # Each ENO value
    for r in range(0, 2):

        # Initialise two sums
        poly_reco_val_plus = 0  # right
        poly_reco_val_min = 0   # left

        # Each local rho value and respective weight
        for j in range(0, 2):
            poly_reco_val_plus += c[r][j]*rho_weno[j-r]           # right
            poly_reco_val_min += c[r-1][j] * rho_weno[j - r]      # left

        # Store each left or right ENO value
        rhoplusENO[r] = poly_reco_val_plus  # right
        rhominENO[r] = poly_reco_val_min  # left

    # STEP 3
    # Define the known ENO polynomial convex weights
    # \tilde d[r] = d[1-r] with r={0, 1, 2}
    d = [2/3, 1/3]

    # STEP 4
    # Calculate the smoothness indicators
    beta = [0, 0]
    beta[0] = (rho_weno[1] - rho_weno[0]) ** 2
    beta[1] = (rho_weno[0] - rho_weno[-1]) ** 2

    # STEP 5a
    # Calculate 3 alpha weights
    # ~alpha = alpha * (d_{1-r}/d_r)
    alpha = [0, 0]      # right
```

```
69      alpha_t = [0, 0]    # left
70
71      # Avoid division by zero
72      epsilon = 1e-6
73
74      # Each alpha value
75      for r in range(0, 2):
76
77          # Right alpha
78          alpha[r] = d[r]/((epsilon+beta[r])**2)
79
80          # Left alpha (transformation)
81          alpha_t[r] = alpha[r]*(d[1-r]/d[r])
82
83      # STEP 5b
84      # Calculate omega weights
85      # Omega and ~omega separate arrays as no simple transformation
86      omega = [0, 0]        # right
87      omega_t = [0, 0]      # left
88
89      # For each weight
90      for r in range(0, 2):
91
92          # right weights
93          omega[r] = alpha[r]/sum(alpha)
94
95          # left weights
96          omega_t[r] = alpha_t[r]/sum(alpha_t)
97
98      # STEP 6
99      # Evaluate the two 5th order reconstructions
100     # Initialise sums
101     rhoplusWENO = 0   # right
102     rhominWENO = 0    # left
103
104     # Each weight and ENO reconstructed value in the sum
105     for r in range(0, 2):
106
107         # Right
108         rhoplusWENO += omega[r]*rhoplusENO[r]
109
110         # Left
111         rhominWENO += omega_t[r]*rhominENO[r]
112
113     # STEP 7
114     # Assign reconstructed values to output objects
115     left_reco = rhominWENO
116     right_reco = rhoplusWENO
117
118     # End function
119     return [left_reco, right_reco]
120
121
122 # 5th Order
123 def weno5(cell, variable_array):
124
125     # STEP 1
126     # Construct required density values
127     # rho_weno[i] is equivalent to rho_{cell+i} for i={-2, -1, 0, 1, 2}
128     rho_weno = np.concatenate((variable_array[cell:cell+3],           # center and
        right
129                                variable_array[cell-1:cell-3:-1]))     # left
130
131     # STEP 2
132     # Define known ENO coefficients
133     # For ~c = c[r-1][j]
134     c = [[1/3, 5/6, -1/6],      # r=0
135          [-1/6, 5/6, 1/3],      # r=1
136          [1/3, -7/6, 11/6],     # r=2
137          [11/6, -7/6, 1/3]]     # r=-1  (for the ~c transformation)
```

```
138
139     # Generate 3 right and 3 left polynomial ENO reconstructed values
140     rhoplusENO = [0, 0, 0]  # right
141     rhominENO = [0, 0, 0]   # left
142
143     # Each ENO value
144     for r in range(0, 3):
145
146         # Initialise two sums
147         poly_reco_val_plus = 0  # right
148         poly_reco_val_min = 0   # left
149
150         # Each local rho value and respective weight
151         for j in range(0, 3):
152             poly_reco_val_plus += c[r][j]*rho_weno[j-r]           # right
153             poly_reco_val_min += c[r-1][j] * rho_weno[j - r]      # left
154
155         # Store each left or right ENO value
156         rhoplusENO[r] = poly_reco_val_plus  # right
157         rhominENO[r] = poly_reco_val_min  # left
158
159     # STEP 3
160     # Define the known ENO polynomial convex weights
161     # \tilde d[r] = d[2-r] with r={0, 1, 2}
162     d = [3/10, 3/5, 1/10]
163
164     # STEP 4
165     # Define smoothness sum coefficients
166     coeff1 = [1, -2, 1]     # r-invariant
167     coeff2 = [[3, -4, 1],   # r=0
168               [1, 0, -1],   # r=1
169               [1, -4, 3]]   # r=2
170
171     # Calculate the smoothness indicators
172     beta = [0, 0, 0]
173
174     # Each smoothness indicator
175     for r in range(0, 3):
176
177         # Initialise sums
178         sum1 = 0
179         sum2 = 0
180
181         # Each new term
182         for j in range(0, 3):
183             sum1 += coeff1[j]*rho_weno[j-r]
184             sum2 += coeff2[r][j]*rho_weno[j-r]
185
186         # Smoothness indicator form
187         beta[r] = (13/12)*(sum1**2)+(1/4)*(sum2**2)
188
189     # STEP 5a
190     # Calculate 3 alpha weights
191     # ~alpha = alpha * (d_{2-r}/d_r)
192     alpha = [0, 0, 0]       # right
193     alpha_t = [0, 0, 0]     # left
194
195     # Avoid division by zero
196     epsilon = 1e-6
197
198     # Each alpha value
199     for r in range(0, 3):
200
201         # Right alpha
202         alpha[r] = d[r]/((epsilon+beta[r])**2)
203
204         # Left alpha (transformation)
205         alpha_t[r] = alpha[r]*(d[2-r]/d[r])
206
207     # STEP 5b
```

```
208     # Calculate omega weights
209     # Omega and ~omega separate arrays as no simple transformation
210     omega = [0, 0, 0]       # right
211     omega_t = [0, 0, 0]     # left
212 
213     # For each weight
214     for r in range(0, 3):
215 
216         # right weights
217         omega[r] = alpha[r]/sum(alpha)
218 
219         # left weights
220         omega_t[r] = alpha_t[r]/sum(alpha_t)
221 
222 
223     # STEP 6
224     # Evaluate the two 5th order reconstructions
225     # Initialise sums
226     rhoplusWENO = 0   # right
227     rhominWENO = 0    # left
228 
229     # Each weight and ENO reconstructed value in the sum
230     for r in range(0, 3):
231 
232         # Right
233         rhoplusWENO += omega[r]*rhoplusENO[r]
234 
235         # Left
236         rhominWENO += omega_t[r]*rhominENO[r]
237 
238     # STEP 7
239     # Assign reconstructed values to output objects
240     left_reco = rhominWENO
241     right_reco = rhoplusWENO
242 
243     # End function
244     return [left_reco, right_reco]
245 
246 
247 # 7th Order
248 def weno7(cell, variable_array):
249 
250     # STEP 1
251     # Construct required density values
252     # rho_weno[i] is equivalent to rho_{cell+i} for i={-3, -2, -1, 0, 1, 2, 3}
253     rho_weno = np.concatenate((variable_array[cell:cell+4],           # right
            stencil cells
254                                variable_array[cell-1:cell-4:-1]))     # left
                                    stencil cells
255 
256     # STEP 2
257     # Define known ENO coefficients
258     # For ~c = c[r-1][j]
259     c = [[1/4, 13/12, -5/12, 1/12],       # r=0
260          [-1/12, 7/12, 7/12, -1/12],      # r=1
261          [1/12, -5/12, 13/12, 1/4],       # r=2
262          [-1/4, 13/12, -23/12, 25/12],    # r=3
263          [25/12, -23/12, 13/12, -1/4]]    # r=-1  (for the ~c transformation)
264 
265     # Generate 3 right and 3 left polynomial ENO reconstructed values
266     rhoplusENO = [0, 0, 0, 0]  # right
267     rhominENO = [0, 0, 0, 0]   # left
268 
269     # Each ENO value
270     for r in range(0, 4):
271 
272         # Initialise two sums
273         poly_reco_val_plus = 0  # right
274         poly_reco_val_min = 0   # left
275 
```

```
276         # Each local rho value and respective weight
277         for j in range(0, 4):
278             poly_reco_val_plus += c[r][j] * rho_weno[j-r]           # right
279             poly_reco_val_min += c[r-1][j] * rho_weno[j - r]      # left
280 
281         # Store each left or right ENO value
282         rhoplusENO[r] = poly_reco_val_plus  # right
283         rhominENO[r] = poly_reco_val_min  # left
284 
285     # STEP 3
286     # Define the known ENO polynomial convex weights
287     # \tilde d[r] = d[2-r] with r={0, 1, 2}
288     d = [4 / 35, 18 / 35, 12 / 35, 1 / 35]
289 
290     # STEP 4
291     # Define smoothness sum coefficients
292     # b[r,j,l] three indices
293     # r = 0, 1, 2, 3
294     # j = 0, 1, 2, 3
295     # l = 0, ..., 3-j
296     b = [[[2107, -9402, 7042, -1854], [11003, -17246, 4642], [7043, -3882], [547]],
              # r=0
297          [[547, -2522, 1922, -494], [3443, -5966, 1602], [2843, -1642], [267]],
                   # r=1
298          [[267, -1642, 1602, -494], [2843, -5966, 1922], [3443, -2522], [547]],
                   # r=2
299          [[547, -3882, 4642, -1854], [7043, -17246, 7042], [11003, -9402], [2107]]]
                # r=3
300 
301     # Calculate the smoothness indicators
302     beta = [0, 0, 0, 0]
303 
304     # Each smoothness indicator
305     for r in range(0, 4):
306 
307         # Initialise outer sum
308         outer_sum = 0
309 
310         # J loop
311         for j in range(0, 4):
312 
313             # Initialise inner sum
314             inner_sum = 0
315 
316             # L loop
317             for l in range(0, 3-j):
318                 inner_sum += b[r][j][l]*rho_weno[j+l-r]
319 
320             # Multiply inner sum
321             inner_sum *= rho_weno[j-r]
322 
323             # Add to outer sum
324             outer_sum += inner_sum
325 
326         # Allocate smoothness indicator
327         beta[r] = outer_sum
328 
329     # STEP 5a
330     # Calculate 3 alpha weights
331     # ~alpha = alpha * (d_{2-r}/d_r)
332     alpha = [0, 0, 0, 0]       # right
333     alpha_t = [0, 0, 0, 0]     # left
334 
335     # Avoid division by zero
336     epsilon = 1e-6
337 
338     # Each alpha value
339     for r in range(0, 4):
340 
341         # Right alpha
```

```
342         alpha[r] = d[r]/((epsilon+beta[r])**2)
343 
344         # Left alpha (transformation)
345         alpha_t[r] = alpha[r]*(d[3-r]/d[r])
346 
347     # STEP 5b
348     # Calculate omega weights
349     # Omega and ~omega separate arrays as no simple transformation
350     omega = [0, 0, 0, 0]       # right
351     omega_t = [0, 0, 0, 0]     # left
352 
353     # For each weight
354     for r in range(0, 4):
355 
356         # right weights
357         omega[r] = alpha[r]/sum(alpha)
358 
359         # left weights
360         omega_t[r] = alpha_t[r]/sum(alpha_t)
361 
362     # STEP 6
363     # Evaluate the two 5th order reconstructions
364     # Initialise sums
365     rhoplusWENO = 0   # right
366     rhominWENO = 0    # left
367 
368     # Each weight and ENO reconstructed value in the sum
369     for r in range(0, 4):
370 
371         # Right
372         rhoplusWENO += omega[r]*rhoplusENO[r]
373 
374         # Left
375         rhominWENO += omega_t[r]*rhominENO[r]
376 
377     # STEP 7
378     # Monotonicity preserving bounds
379     # A. Suresh and H. T. Huynh,
380     # Accurate monotonicity preserving scheme with Runge-Kutta time-stepping,
381     # J. Comput. Phys.136, 83 (1997).
382 
383     # STEP 7a
384     # Zone center curvature measures
385     d_j = rho_weno[1]-2*rho_weno[0]+rho_weno[-1]
386     d_jp1 = rho_weno[2]-2*rho_weno[1]+rho_weno[0]
387     d_jm1 = rho_weno[0]-2*rho_weno[-1]+rho_weno[-2]
388 
389     # STEP 7b
390     # Minmod of local curvature
391     right_dmd = minmod(d_j, d_jp1)
392     right_dlc = minmod(d_j, d_jp1)
393     left_dmd = minmod(d_j, d_jm1)
394     left_dlc = minmod(d_j, d_jm1)
395 
396     # STEP 7c
397     # Define curvature constants
398     alpha_curv = 2
399     beta_curv = 4
400 
401     # STEP 7d
402     # Median bounds
403     right_md = 0.5*(rho_weno[0]+rho_weno[1])-0.5*right_dmd
404     left_md = 0.5*(rho_weno[0]+rho_weno[-1])-0.5*left_dmd
405     # Upper limit bound
406     right_ul = rho_weno[0] + alpha_curv*(rho_weno[0]-rho_weno[-1])
407     left_ul = rho_weno[0] + alpha_curv*(rho_weno[0]-rho_weno[1])
408     # Large curvature
409     right_lc = rho_weno[0] + 0.5*(rho_weno[0]-rho_weno[-1])+beta_curv*right_dlc/3
410     left_lc = rho_weno[0] + 0.5*(rho_weno[0]-rho_weno[1])+beta_curv*left_dlc/3
411 
```

```
412     # STEP 7e
413     # Define minimum and maximum left and right bounds
414     right_min = max(min(rho_weno[0], rho_weno[1], right_md), min(rho_weno[0],
            right_ul, right_lc))
415     right_max = min(max(rho_weno[0], rho_weno[1], right_md), max(rho_weno[0],
            right_ul, right_lc))
416     left_min = max(min(rho_weno[0], rho_weno[-1], left_md), min(rho_weno[0],
            left_ul, left_lc))
417     left_max = min(max(rho_weno[0], rho_weno[-1], left_md), max(rho_weno[0],
            left_ul, left_lc))
418
419     # STEP 8
420     # Allocate the recosntructed monotonic bounded values
421     right_reco = median(rhoplusWENO, right_min, right_max)
422     left_reco = median(rhominWENO, left_min, left_max)
423
424     # End func
425     return [left_reco, right_reco]
426
427
428 # Minmod function
429 def minmod(arg1,arg2):
430     return 0.5*(math.copysign(1.0, arg1)+math.copysign(1.0, arg2))*min(abs(arg1),
            abs(arg2))
431
432
433 # Median function
434 def median(x,y,z):
435     return x+minmod(y-x, z-x)
```

A.1.5 *params.txt*

```
1 {"dx"   : 0.01,
2  "T"    : 2,
3  "CFL" : 0.9,
4  "velocity_model" : "Greenshields",
5  "riemann_solver" : "LaxFriedrichs",
6  "reconstruction" : "FirstOrder",
7  "limiter" : "MinMod"}
```

A.2 MATLAB Postprocessing Codes

A.2.1 Re Di Roma Roundabout

A.2.1.1 *RomeRoundaboutPlot.m*

```
%% Setup
clc
clear all
close all
set(0,'DefaultFigureWindowStyle','docked')

%% Read in
density=dlmread('density.txt');

%% Split a - lengths

L = [45, 45, 26, 25, 13, 13, 45, 45, 17, 10, 23, 23, 6, 7, 7, 7, 7, 7];
L_cumsum = cumsum(L);

%% Split b - density -> colour transform

eps = 1e-5;
maximum_density = max(max(density));
minimum_density = min(min(density));
gamma = (eps*maximum_density-minimum_density)/(eps-1);
density = (density-gamma)/(maximum_density-gamma);

%% Split c - split

r01 = density(:,1:L_cumsum(1));
r02 = density(:,L_cumsum(1)+1:L_cumsum(2));
r03 = density(:,L_cumsum(2)+1:L_cumsum(3));
r04 = density(:,L_cumsum(3)+1:L_cumsum(4));
r05 = density(:,L_cumsum(4)+1:L_cumsum(5));
r06 = density(:,L_cumsum(5)+1:L_cumsum(6));
r07 = density(:,L_cumsum(6)+1:L_cumsum(7));
r08 = density(:,L_cumsum(7)+1:L_cumsum(8));
r09 = density(:,L_cumsum(8)+1:L_cumsum(9));
r10 = density(:,L_cumsum(9)+1:L_cumsum(10));
r11 = density(:,L_cumsum(10)+1:L_cumsum(11));
r12 = density(:,L_cumsum(11)+1:L_cumsum(12));
rcirc = density(:,L_cumsum(12)+1:371);


%% info

[nframes, tot_length]=size(density);

%% Define road lines

eps=0;
circ_r=1-eps;
circ_t=linspace(83*pi/180,83*pi/180+2*pi,41);
circ_x=circ_r*cos(circ_t);
circ_y=circ_r*sin(circ_t);
r01_x=linspace(-0.5,-5,45);
r01_y=-1.1*(r01_x+0.5)+sqrt(3)/2;
r02_x=linspace(-5.2,-sqrt(2)/2,45);
r02_y=-1.1*(r02_x+sqrt(2)/2)+sqrt(2)/2;
r03_x=linspace(-1,-5,26);
r03_y=0.2*(r03_x+1);
r04_x=linspace(-5,-sqrt(15)/4,25);
r04_y=0.2*(r04_x+sqrt(15)/4)-0.25;
r05_x=linspace(-1/4,-1.2,13);
r05_y=2*(r05_x+0.25)-1;
r06_x=linspace(-1,0,13);
r06_y=2*r06_x-1;
```

```
r07_x=linspace(0.5,5,45);
r07_y=-1.1*(r07_x-0.5)-sqrt(3)/2;
r08_x=linspace(5.2,sqrt(2)/2,45);
r08_y=-1.1*(r08_x-sqrt(2)/2)-sqrt(2)/2;
r09_x=linspace(1,5,17);
r09_y=0*r09_x;
r10_x=linspace(3,sqrt(15)/4,10);
r10_y=1.25*(r10_x-sqrt(15)/4)+0.25;
r11_x=linspace(0.25,1.4,23);
r11_y=4*(r11_x-0.25)+sqrt(15)/4;
r12_x=linspace(1.15,0,23);
r12_y=4*r12_x+1;

%% Plot map lines

    plot(circ_x,circ_y, 'k')
    hold on
    plot(r01_x,r01_y, 'k')
    plot(r02_x,r02_y, 'k')
    plot(r03_x,r03_y, 'k')
    plot(r04_x,r04_y, 'k')
    plot(r05_x,r05_y, 'k')
    plot(r06_x,r06_y, 'k')
    plot(r07_x,r07_y, 'k')
    plot(r08_x,r08_y, 'k')
    plot(r09_x,r09_y, 'k')
    plot(r10_x,r10_y, 'k')
    plot(r11_x,r11_y, 'k')
    plot(r12_x,r12_y, 'k')
    hold off

    pbaspect([1 1 1])
    axis equal
    box off
    axis off

%% Physical Animation

close all
figure

siz = 20;

for i=[1 1:nframes]

    plot([-10 -11], [-10 -11])

    multicollineplot(circ_x,circ_y,rcirc(i,:))
    multicollineplot(r01_x,r01_y,r01(i,:))
    multicollineplot(r02_x,r02_y,r02(i,:))
    multicollineplot(r03_x,r03_y,r03(i,:))
    multicollineplot(r04_x,r04_y,r04(i,:))
    multicollineplot(r05_x,r05_y,r05(i,:))
    multicollineplot(r06_x,r06_y,r06(i,:))
    multicollineplot(r07_x,r07_y,r07(i,:))
    multicollineplot(r08_x,r08_y,r08(i,:))
    multicollineplot(r09_x,r09_y,r09(i,:))
    multicollineplot(r10_x,r10_y,r10(i,:))
    multicollineplot(r11_x,r11_y,r11(i,:))
    multicollineplot(r12_x,r12_y,r12(i,:))

    % colourbar
    multicollineplot(ones(50,1)-8,linspace(-5,5,50),linspace(1e-5,1,50))
    ticks = {0.02,0.04,0.06,0.08,0.1,0.12,0.14,0.16,0.18};
    text(-6.8*ones(9,1),linspace(-4,4,9),ticks)
    strings={sprintf('Max %5.4f',maximum_density),...
             sprintf('Min %5.4f',minimum_density)};
    text([-7 -7],[5.5, -5.5],strings,...
         'HorizontalAlignment','center')
    d=text(-7.5,0,'Density, \rho [cars/km]',...
```

```
133             'HorizontalAlignment','center');
134     set(d,'Rotation',90);
135     text(-5.5,-4.5,sprintf('Frame %d, time t=%6.5f[Hrs]',i,i*0.5/nframes))
136     text(-5.5,-5,'Riemann solver : Lax-Friedrichs')
137     text(-5.5,-5.5,'Reconstruction : First Order')
138 
139     xlim([-8 6])
140     ylim([-6.5 6.5])
141 
142     pbaspect([1 1 1])
143     axis equal
144     box off
145     axis off
146 
147     %pause(0.1)
148 
149     set(gcf,'PaperSize',[siz,siz]);
150     filename=sprintf('frame%04i',i);
151     print(filename,'-dpdf')
152 
153 end
```

A.2.1.2 *multicollineplot.m*

```
1  function multicollineplot(x,y,col)
2  
3      mapping = jet(100);
4  
5      colvec=zeros(length(col),3);
6      for j=1:length(col)
7          colvec(j,:) = mapping(ceil(col(j)*100),:);
8      end
9  
10     hold on
11     for index=1:length(x)-1
12         plot(x(index:index+1),y(index:index+1),...
13             'color',colvec(index,:))
14     end
15     hold off
16 
17 
18 end
```

A.2.2 M1 Wakefield Junction 40

A.2.2.1 *M1J40plot.m*

```
%% Setup
clc
clear all
close all
set(0,'DefaultFigureWindowStyle','docked')

%% Read in
density=dlmread('density.txt');

%% Split a - lengths

L = [50, 50, 50, 50,...
     97, 97,...
     48, 32, 32, 48,...
     20, 20, 20, 20,...
     13, 3, 8, 3, 13, 3, 8, 3];
L_cumsum = cumsum(L);

clear L

%% Split b - density -> colour transform

% time trim
%density = density(50:100,:);

eps = 1e-6;
maximum_density = max(max(density));
minimum_density = min(min(density));
gamma = (eps*maximum_density-minimum_density)/(eps-1);
density = (density-gamma)/(maximum_density-gamma);

clear eps gamma

%% Split c - split

r01 = density(:,1:L_cumsum(1));
r02 = density(:,L_cumsum(1)+1:L_cumsum(2));
r03 = density(:,L_cumsum(2)+1:L_cumsum(3));
r04 = density(:,L_cumsum(3)+1:L_cumsum(4));
r05 = density(:,L_cumsum(4)+1:L_cumsum(5));
r06 = density(:,L_cumsum(5)+1:L_cumsum(6));
r07 = density(:,L_cumsum(6)+1:L_cumsum(7));
r08 = density(:,L_cumsum(7)+1:L_cumsum(8));
r09 = density(:,L_cumsum(8)+1:L_cumsum(9));
r10 = density(:,L_cumsum(9)+1:L_cumsum(10));
r11 = density(:,L_cumsum(10)+1:L_cumsum(11));
r12 = density(:,L_cumsum(11)+1:L_cumsum(12));
r13 = density(:,L_cumsum(12)+1:L_cumsum(13));
r14 = density(:,L_cumsum(13)+1:L_cumsum(14));
r15 = density(:,L_cumsum(14)+1:L_cumsum(15));
r16 = density(:,L_cumsum(15)+1:L_cumsum(16));
r17 = density(:,L_cumsum(16)+1:L_cumsum(17));
r18 = density(:,L_cumsum(17)+1:L_cumsum(18));
r19 = density(:,L_cumsum(18)+1:L_cumsum(19));
r20 = density(:,L_cumsum(19)+1:L_cumsum(20));
r21 = density(:,L_cumsum(20)+1:L_cumsum(21));
r22 = density(:,L_cumsum(21)+1:L_cumsum(22));

clear L_cumsum


%% info

[nframes, tot_length]=size(density);

```

```
clear density

%% Define road lines

close all

figure
grid on
grid minor

xmin = -6; xmax = 6;
h = (xmax-xmin)*(603/544)/2;
ymin = -h; ymax = h;

img = imread('map.png');
image('CData',img,'XData',[xmin xmax],'YData',[ymax ymin])

[r01_x,r01_y] = arcpoints([-0.6122;3.858],[-1.04;6.6],80,51);
[r02_x,r02_y] = arcpoints([-0.4098;3.368],[-0.92;6.6],80,51);
[r03_x,r03_y] = arcpoints([0.4289;-3.226],[1.3;-6.6],13,51);
r02_x = fliplr(r02_x); r02_y = fliplr(r02_y);
r03_x = fliplr(r03_x); r03_y = fliplr(r03_y);
[r04_x,r04_y] = arcpoints([0.7903;-4.343],[1.46;-6.6],13,51);
[r05_x,r05_y] = arcpoints([0.4289;-3.226],[-0.6122;3.858],1000,98);
[r06_x,r06_y] = arcpoints([0.7903;-4.343],[-0.4098;3.368],1000,98);
r06_x = fliplr(r06_x); r06_y = fliplr(r06_y);
[r07_xa,r07_ya] = arcpoints([-0.56;0.58],[-0.453;1.237],1,13);
[r07_xb,r07_yb] = arcpoints([-0.6122;3.858],[-0.453;1.237],30,36);
r07_x = [r07_xb,fliplr(r07_xa)];
r07_y = [r07_yb,fliplr(r07_ya)];
[r08_xa,r08_ya] = arcpoints([0.1215;1.259],[-0.4098;3.368],30,24);
[r08_xb,r08_yb] = arcpoints([0.1215;1.259],[0.42;0.6],1,9);
r08_x = [fliplr(r08_xa),r08_xb];
r08_y = [fliplr(r08_ya),r08_yb];
[r09_xa,r09_ya] = arcpoints([-0.09945;-1.105],[0.4289;-3.226],30,24);
[r09_xb,r09_yb] = arcpoints([-0.09945;-1.105],[-0.47;-0.35],1,9);
r09_x = [fliplr(r09_xa),r09_xb];
r09_y = [fliplr(r09_ya),r09_yb];
[r10_xa,r10_ya] = arcpoints([0.57;-0.32],[0.4751;-1.171],1,13);
[r10_xb,r10_yb] = arcpoints([0.7903;-4.343],[0.4751;-1.171],30,36);
r10_x = [r10_xa,fliplr(r10_xb)];
r10_y = [r10_ya,fliplr(r10_yb)];
[r11_xa,r11_ya] = arcpoints([-1.27;0.442],[-0.78;0.4],0.3,6);
[r11_xb,r11_yb] = arcpoints([-2.486;1.569],[-1.27;0.442],8,15);
r11_x = [r11_xb,r11_xa];
r11_y = [r11_yb,r11_ya];
[r12_x,r12_y] = arcpoints([-2.597;1.547],[-0.8;-0.1],15,21);
[r13_x,r13_y] = arcpoints([2.751;-1.193],[0.74;0.376],20,21);
[r14_xa,r14_ya] = arcpoints([1.32;-0.199],[0.74;-0.155],0.4,6);
[r14_xb,r14_yb] = arcpoints([2.663;-1.259],[1.32;-0.199],15,15);
r14_x = [r14_xb,r14_xa];
r14_y = [r14_yb,r14_ya];
[r15_x,r15_y] = arcpoints([0.42;0.6],[-0.56;0.58],1.5,14);
[r16_x,r16_y] = arcpoints([0.74;0.376],[0.42;0.6],0.4,4);
[r17_x,r17_y] = arcpoints([0.74;-0.155],[0.74;0.376],0.4,9);
[r18_x,r18_y] = arcpoints([0.57;-0.32],[0.74;-0.155],0.4,4);
[r19_x,r19_y] = arcpoints([-0.47;-0.35],[0.57;-0.32],1.5,14);
[r20_x,r20_y] = arcpoints([-0.8;-0.1],[-0.47;-0.35],0.4,4);
[r21_x,r21_y] = arcpoints([-0.78;0.4],[-0.8;-0.1],0.4,9);
[r22_x,r22_y] = arcpoints([-0.56;0.58],[-0.78;0.4],0.4,4);

%% Plot map lines

hold on
    plot(r01_x,r01_y, 'k')
    plot(r02_x,r02_y, 'k')
    plot(r03_x,r03_y, 'k')
    plot(r04_x,r04_y, 'k')
    plot(r05_x,r05_y, 'k')
    plot(r06_x,r06_y, 'k')
```

```
    plot(r07_x,r07_y, 'k')
    plot(r08_x,r08_y, 'k')
    plot(r09_x,r09_y, 'k')
    plot(r10_x,r10_y, 'k')
    plot(r11_x,r11_y, 'k')
    plot(r12_x,r12_y, 'k')
    plot(r13_x,r13_y, 'k')
    plot(r14_x,r14_y, 'k')
    plot(r15_x,r15_y, 'k')
    plot(r16_x,r16_y, 'k')
    plot(r17_x,r17_y, 'k')
    plot(r18_x,r18_y, 'k')
    plot(r19_x,r19_y, 'k')
    plot(r20_x,r20_y, 'k')
    plot(r21_x,r21_y, 'k')
    plot(r22_x,r22_y, 'k')
hold off

xlim([xmin xmax]*1.1)
ylim([ymin ymax]*1.1)

%pbaspect([1 1 1])
axis equal
%box off
%axis off

clear h img
clear r07_xa r07_ya r07_xb r07_yb
clear r08_xa r08_ya r08_xb r08_yb
clear r09_xa r09_ya r09_xb r09_yb
clear r10_xa r10_ya r10_xb r10_yb
clear r11_xa r11_ya r11_xb r11_yb
clear r14_xa r14_ya r14_xb r14_yb


%% Physical Animation

close all
figure

siz = 20;
img = imread('map.png');

for i=[408 408:500]

    plot([-100 -101],[100 101])

    denlineplot(r01_x,r01_y,r01(i,:))
    denlineplot(r02_x,r02_y,r02(i,:))
    denlineplot(r03_x,r03_y,r03(i,:))
    denlineplot(r04_x,r04_y,r04(i,:))
    denlineplot(r05_x,r05_y,r05(i,:))
    denlineplot(r06_x,r06_y,r06(i,:))
    denlineplot(r07_x,r07_y,r07(i,:))
    denlineplot(r08_x,r08_y,r08(i,:))
    denlineplot(r09_x,r09_y,r09(i,:))
    denlineplot(r10_x,r10_y,r10(i,:))
    denlineplot(r11_x,r11_y,r11(i,:))
    denlineplot(r12_x,r12_y,r12(i,:))
    denlineplot(r13_x,r13_y,r13(i,:))
    denlineplot(r14_x,r14_y,r14(i,:))
    denlineplot(r15_x,r15_y,r15(i,:))
    denlineplot(r16_x,r16_y,r16(i,:))
    denlineplot(r17_x,r17_y,r17(i,:))
    denlineplot(r18_x,r18_y,r18(i,:))
    denlineplot(r19_x,r19_y,r19(i,:))
    denlineplot(r20_x,r20_y,r20(i,:))
    denlineplot(r21_x,r21_y,r21(i,:))
    denlineplot(r22_x,r22_y,r22(i,:))

```

```
206     % colourbar
207     denlineplot(ones(100,1)*(-4),linspace(-6,6,100),linspace(1e-10,1,99))
208         ticks = {0.02    0.04    0.06    0.08    0.10    0.12    0.14    0.16
                0.18    0.20    0.22    0.24    0.26};
209     text(-3.7*ones(13,1),linspace(-5.1429,5.1429,13),ticks)
210     strings={sprintf('Max %5.4f',maximum_density),...
211              sprintf('Min %5.4f',minimum_density)};
212     text([-4 -4],[6.5, -6.5],strings,...
213          'HorizontalAlignment','center')
214     d=text(-4.5,0,'Density, \rho [cars/km]',...
215            'HorizontalAlignment','center');
216     set(d,'Rotation',90);
217     text(1,6,sprintf('Frame %d, time t=%6.5f[Hrs]',i,i*2/nframes))
218     text(1,5.5,'Riemann solver : Lax-Friedrichs')
219     text(1,5,'Reconstruction : First Order')
220
221     xlim([xmin xmax])
222     ylim([ymin ymax])
223
224     pbaspect([1 1 1])
225     axis equal
226     box off
227     axis off
228
229     set(gcf,'PaperSize',[siz,siz]);
230     filename=sprintf('frame%04i',i);
231     print(filename,'-dpdf')
232
233 end
```

A.2.2.2 *arcpoints.m*

```
1  function [x,y] = arcpoints(A,B,curv,npoints)
2     d = norm(B-A);
3     R = d/2+curv; % Choose R radius >= d/2
4     C = (B+A)/2+sqrt(R^2-d^2/4)/d*[0,-1;1,0]*(B-A); % Center of circle
5     a = atan2(A(2)-C(2),A(1)-C(1));
6     b = atan2(B(2)-C(2),B(1)-C(1));
7     b = mod(b-a,2*pi)+a; % Ensure that arc moves counterclockwise
8     t = linspace(a,b,npoints);
9     x = C(1)+R*cos(t);
10    y = C(2)+R*sin(t);
11 end
```

A.2.2.3 *denlineplot.m*

```
function denlineplot(x,y,col)

    mapping = jet(100);

    colvec=zeros(length(col),3);
    LWidth = zeros(length(col),1);

    % length(x)=length(col)+1

    hold on
    for index=1:length(col)

        colvec(index,:) = mapping(ceil(col(index)*100),:);
        LWidth(index) = col(index)*4;

        if (length(col) == 99)
            LWidth(index) = 3;
        end

        plot(x(index:index+1),y(index:index+1),...
            'color',colvec(index,:),...
            'LineWidth',LWidth(index))
    end
    hold off


end
```

A.3 Other Codes

A.3.1 Nagel-Schreckenberg Cellular Automation

A.3.1.1 *NS_implementation.m*

```
%% Script to implement the Nagel - Schreckenberg Model
%  As described in Nagel and Schreckenberg (1992 J.Phys)
%
% Andrew Dixon
% Cranfield University 26/04/2019

%% Setup
    clear all                                   % clear variables
    close all                                   % close figures
    clc                                         % clear command window
    set(0,'DefaultFigureWindowStyle','docked') % dock figures

%% Define model parameters and Initialise
    % Model Parameters
        % Simulation iterations
            its=300;

        % Length of 'road' in number of cells
            ncell=100;
        % number of cars
            ncar=50;

        % Limit velocity
            maxvel=5;

        % Human random braking effect
        % p = probability of a random decelleration
            p=0.5;

    % Initialise for Simulation
        % Animation playback frame time
        % No playback for 0
            pause_length=0.01;

        % Initialize arrays
            % Speeds are random integers in [0,maxvel]
                speed=datasample(0:maxvel,ncar);
            % Positions is an array to save history
            % Allocate ncar cars to a unique cell
                position=zeros(its+1,ncar);
                position(1,:)=sort(randperm(ncell,ncar));

        % Plot initial formation on a circle
            % Radius
                r=1;
            % Calculate angles
                theta=2*pi*position(1,:)/ncell;
            % Polar tranformation
                x=r*cos(theta);
                y=r*sin(theta);
            % Plot points
                plot(x,y,'ko','markerfacecolor','k')
                drawCircle(r+0.05)
                drawCircle(r-0.05)
                box off
                xlim([-r r]*1.2)
                ylim([-r r]*1.2)
                pbaspect([1 1 1])

%% Iterate
% Loop over the N-S algorithm for its iterations

```

```
for it=1:its

    %%%%%%%%%%%%%%%%%%%%%%% Algorithm Steps %%%%%%%%%%%%%%%%%%%%%%%

    % 1 Acceleration
    % Increase car's speeds by 1 if speed is less than max
        for i=1:ncar
            if speed(i)<maxvel
                speed(i)=speed(i)+1;
            end
        end

    % 2 Deceleration
    % Reduce a car's speed if it is close to the car infront
        % Calculate separation distances
            separation=[position(it,[2:ncar])-1-position(it,[1:ncar-1]) ncell-
                position(it,ncar)-1+position(it,1)];

        % If separation length is less than the speed
        % Then the next step would cause the trailing car to overlap
        % Fix trailing car speed to move to the space behind
            for i=1:ncar
                if separation(i)<speed(i)
                    speed(i)=separation(i);
                end
            end

    % 3 Randomization
    % Introduce a random human braking element
        for i=1:ncar
            if speed(i)>0
                speed(i)=speed(i)+randsample([0 -1],1,true,[1-p p]);
            end
        end

    % 4 Car motion
    % Update the car positions according to the corresponding speeds
        for i=1:ncar
            position(it+1,i)=position(it,i)+speed(i);
        end

    %%%%%%%%%%%%%%%%%%%%%%% Algorithm Steps %%%%%%%%%%%%%%%%%%%%%%%

    % Plot positions
        % Polar coordinates translation
            theta=2*pi*position(it+1,:)/ncell;
            x=r*cos(theta);
            y=r*sin(theta);
        % Plot points
            plot(x,y,'ko','markerfacecolor','k')
            drawCircle(r+0.05)
            drawCircle(r-0.05)
            box off
            pbaspect([1 1 1])
            xlim([-r r]*1.2)
            ylim([-r r]*1.2)

    % Pause
        if pause_length~=0
            pause(pause_length)
        end
end


%% Position history plot

figure
hold on
for i=1:ncar
    plot(position(:,i),(0:(-1):(-its)),'k');
```

```
132 end
133 hold off
134 xlabel('x')
135 ylabel('Iteration')
136 yticks(-its:50:0)
137 yticklabels(its:(-50):0)
138
139 %% Save PDFs
140
141 siz=[20 20];
142 names=[{'physical','waves'}];
143
144 for i=1:2
145     figure(i);
146     set(gcf,'PaperSize',[siz(i),siz(i)]);
147     print('-bestfit',names{i},'-dpdf')
148 end
```

A.3.1.2 *drawCircle.m*

```
1  function drawCircle(r)
2      theta=0:0.01:2*pi;
3
4      x=r*cos(theta);
5      y=r*sin(theta);
6
7      hold on
8      plot(x,y,'k')
9      hold off
10 end
```

Appendix B

Higher Order Reconstruction Procedures

B.1 MUSCL

For the single cell i, the left and right reconstructed states are given by $\rho^R_{i-1/2}$ and $\rho^L_{i+1/2}$ respectively.

B.1.1 Linear - 2nd Order

The 2nd order MUSCL scheme approximates the cell density distribution with a linear slope. The left and right states are given by

$$\rho^R_{i-1/2} = \rho_i - \frac{1}{2}\phi\left(r_i\right)\left(\rho_{i+1} - \rho_i\right), \tag{B.1}$$

$$\rho^L_{i+1/2} = \rho_i + \frac{1}{2}\phi\left(r_i\right)\left(\rho_{i+1} - \rho_i\right), \tag{B.2}$$

where

$$r_i = \frac{\rho_i - \rho_{i-1}}{\rho_{i+1} - \rho_i}, \tag{B.3}$$

and ϕ is the slope limiting function.

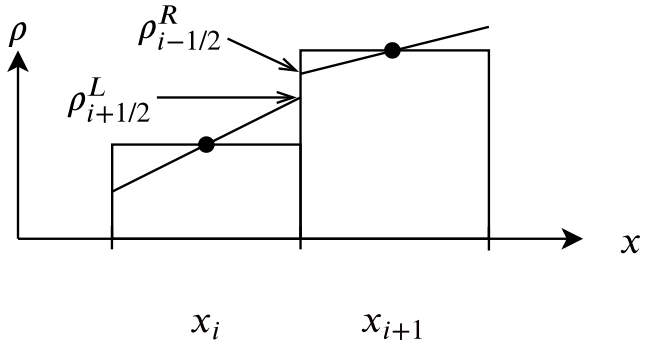


Figure B.1: MUSCL second order piecewise linear reconstruction.

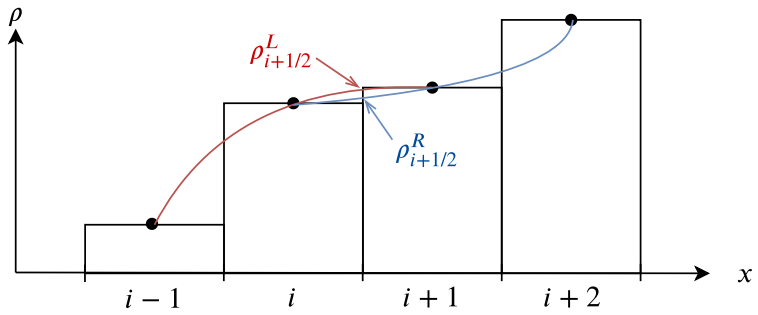


Figure B.2: MUSCL third order piecewise parabolic reconstruction using second order interpolation.

B.1.2 Parabolic - 3rd Order

The 3rd order MUSCL scheme approximates the cell density distribution with a parabolic slope, from a second order interpolation. The left and right states are given by

$$\rho^R_{i-1/2} = \rho_i - \frac{1}{4}\phi\left(r_i\right)\left[\left(1-\kappa\right)\delta\rho_{i+1/2} + \left(1+\kappa\right)\delta\rho_{i-1/2}\right], \tag{B.4}$$

$$\rho^L_{i+1/2} = \rho_i + \frac{1}{4}\phi\left(r_i\right)\left[\left(1-\kappa\right)\delta\rho_{i-1/2} + \left(1+\kappa\right)\delta\rho_{i+1/2}\right], \tag{B.5}$$

with $\kappa = 1/3$, $\delta\rho_{i+1/2} = \rho_{i+1} - \rho_i$, $\delta\rho_{i-1/2} = \rho_i - \rho_{i-1}$, and the slope limiter ϕ.

B.1.3 Slope Limiters

The listed slope limiters apply to MUSCL reconstructions to reduce cell interface oscillations. All present limiters are written in *MUSCLReconstruction.py*, Appendix A.1.3 [lines 90-185].

- CHARM [not 2nd order TVD] [79]

$$\phi\left(r\right) = \begin{cases} \frac{r(3r+1)}{(r+1)^2}, & r > 0 \\ 0, & r \leq 0 \end{cases} \tag{B.6}$$

- HCUS [not 2nd order TVD] [77]

$$\phi\left(r\right) = \frac{3\left(r+|r|\right)}{2\left(r+2\right)} \tag{B.7}$$

- HQUICK [not 2nd order TVD] [77]

$$\phi\left(r\right) = \frac{2\left(r+|r|\right)}{\left(r+3\right)} \tag{B.8}$$

- Koren [3rd order TDV accurate for sufficiently smooth data] [30]

$$\phi(r) = \max\left[0, \min\left(2r, \min\left(\frac{1+2r}{3}, 2\right)\right)\right] \tag{B.9}$$

- MinMod [2rd order TDV accurate] [53]

$$\phi(r) = \max\left[0, \min\left(1, r\right)\right] \tag{B.10}$$

- Monotonized Central (MC) [2rd order TDV accurate, symmetric] [73]

$$\phi(r) = \max\left[0, \min\left(2r, \frac{1+r}{2}, 2\right)\right] \tag{B.11}$$

- Osher [2rd order TDV accurate] [10]

$$\phi(r) = \max\left[0, \min\left(r, \beta\right)\right], \quad (1 \leq \beta \leq 2) \tag{B.12}$$

- Ospre [2rd order TDV accurate, symmetric] [77]

$$\phi(r) = \frac{3\left(r^2 + r\right)}{2\left(r^2 + r + 1\right)} \tag{B.13}$$

- Smart [not 2rd order TDV] [20]

$$\phi(r) = \max\left[0, \min\left(2r, \frac{1+3r}{4}, 4\right)\right] \tag{B.14}$$

- Superbee [2rd order TDV accurate, symmetric] [53]

$$\phi(r) = \max\left[0, \min\left(2r, 1\right), \min\left(r, 2\right)\right] \tag{B.15}$$

- Sweby [not 2rd order TDV, symmetric] [61]

$$\phi(r) = \max\left[0, \min\left(\beta r, 1\right), \min\left(r, \beta\right)\right], \quad (1 \leq \beta \leq 2) \tag{B.16}$$

- UMIST [2rd order TDV accurate] [37]

$$\phi(r) = \max\left[0, \min\left(2r, \frac{1+3r}{4}, \frac{3+r}{4}, 2\right)\right] \tag{B.17}$$

- van Albada 1 [2rd order TDV accurate, symmetric] [71]

$$\phi(r) = \frac{r^2 + r}{r^2 + 1} \tag{B.18}$$

- van Albada 2 - alternate form for high spatial order schemes [not 2rd order TDV] [29]

$$\phi(r) = \frac{2r}{r^2 + 1} \tag{B.19}$$

- van Leer [2nd order TVD accurate, symmetric] [72]

$$\phi(r) = \frac{r + |r|}{1 + |r|} \tag{B.20}$$

Other Limiters

To calculate the gradient limiter, Barth and Jesperson [6] suggested using $\Phi_i = \min(\Phi_{ij})$, where

$$\Phi_{ij} = \begin{cases} \min\left(1, \frac{\delta\rho_i^{\max}}{\rho_{ij} - \overline{\rho}_i}\right), & if \quad \rho_{ij} - \overline{\rho}_i > 0, \\ \min\left(1, \frac{\delta\rho_i^{\min}}{\rho_{ij} - \overline{\rho}_i}\right), & if \quad \rho_{ij} - \overline{\rho}_i < 0, \\ 1, & if \quad \rho_{ij} - \overline{\rho}_i = 0. \end{cases}$$

The values $\delta\rho_i^{\max}$ and $\delta\rho_i^{\min}$ are the maximum and minimum of $\overline{\rho} - \overline{\rho}_i$ respectively, the difference between the concerned volume and nearest neighbours. The value, $\rho_{ij} = R_i(\vec{x}_j - \vec{x}_i)$, is the unlimited reconstructed value. Due to the minimum, maximum, and case selections of Φ_i in the Barth and Jespersen limiter, the reconstructed flux is non-differentiable, which slows the solving convergence. Venkatakrishnan [75] suggested a smooth approximation of the case selection step in the Barth and Jespersen limiter. This approximation uses $\phi(r)$ instead of $\min(1, r)$, where

$$\phi(r) = \frac{r^2 + 2r}{r^2 + r + 2}, \tag{B.21}$$

so for $\rho_{ij} - \overline{\rho}_i < 0$

$$\Phi_{ij} = \phi\left(\frac{\delta\rho_i^{\min}}{\rho_{ij} - \overline{\rho}_i}\right).$$

Venkatakrishnan also suggested avoiding the limiter for regions of uniform flow, i.e. when $\rho_{ij} - \overline{\rho}_i > 0$

$$\Phi_{ij} = \frac{1}{\Delta_-}\left[\frac{\left(\Delta_+^2 + \epsilon^2\right)\Delta_- + 2\Delta_-^2\Delta_+}{\Delta_+^2 + 2\Delta_-^2 + \Delta_-\Delta_+ + \epsilon^2}\right],$$

where $\Delta_- = \rho_{ij} - \overline{\rho}_i$, $\Delta_+ = \delta\rho_i^{\max}$, and $\epsilon^2 = (K\Delta x)^3$ with parameter $K > 0$. The case for $\rho_{ij} - \overline{\rho}_i = 0$ remains unchanged with $\Phi_{ij} = 1$.

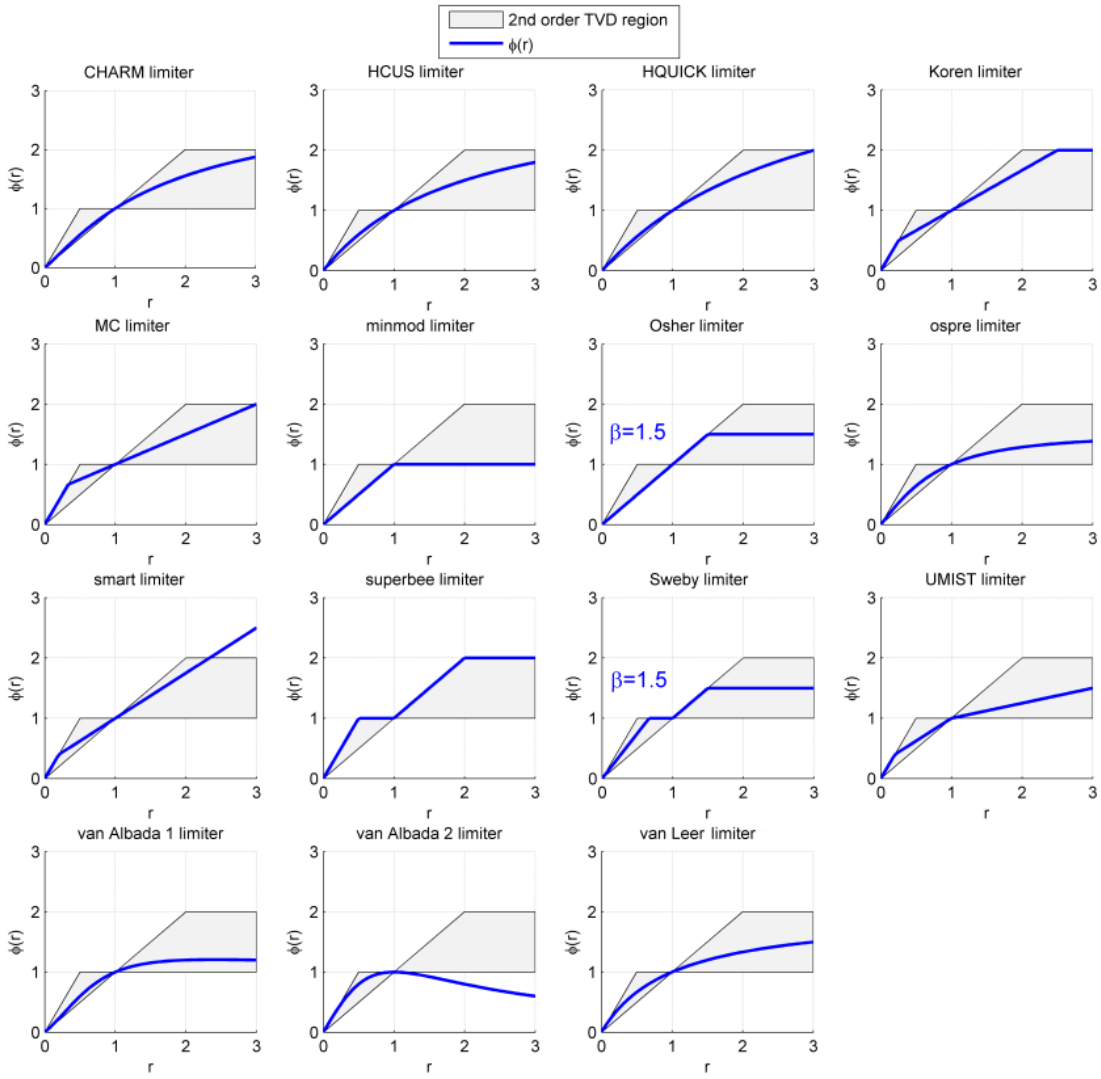


Figure B.3: Plots showing the slope limiter $\phi(r)$ on $r \geq 0$ over the admissible TVD region for second order schemes [61]. Created in MATLAB [24].

B.2 WENO

The general $(2k-1)^{th}$ order WENO reconstruction considers a convex combination of k reconstructions from unique local stencils. See [59] for a more thorough discussion and derivation of WENO and ENO schemes, Procedure 2.2 provides the general WENO scheme steps presented here. The available 3^{rd}, 5^{th}, and 7^{th} order reconstructions in *WENOReconstruction.py* A.1.4 are represented by $k = 2, 3, 4$ respectively. For cell i, we want approximations for the left, $\rho^{+}_{i-1/2}$, and right, $\rho^{-}_{i+1/2}$, density values. The following steps provide an algorithm for computing the WENO reconstructed cell interface values to the required order.

1. Consider the k stencils over $x_{i-(k-1)}, \ldots, x_{i+(k-1)}$, denoted by S_r for $r = 0, \ldots, k-1$,
$$S_r = \left\{x_{i-r}, x_{i-r+1}, \ldots, x_{i-r+(k-1)}\right\}. \tag{B.22}$$

2. Obtain k *right* and k *left* reconstructed values,
$$\rho^{(r)}_{i+1/2} = \sum_{j=0}^{k-1} c_{r,j}\overline{\rho}_{i-r+j}, \quad \text{and,} \quad \rho^{(r)}_{i-1/2} = \sum_{j=0}^{k-1} \tilde{c}_{r,j}\overline{\rho}_{i-r+j} \tag{B.23}$$
using the cell averaged density values $\overline{\rho}$, and weights $c_{r,j}$ and $\tilde{c}_{r,j}$ with $\tilde{c}_{r,j} = c_{r-1,j}$ ($r = -1$ is provided for the left-right transformation purposes) where $c_{r,j}$ are

k	r	j			
		0	1	2	3
2	-1	3/2	-1/2	-	-
	0	1/2	1/2	-	-
	1	-1/2	3/2	-	-
3	-1	11/6	-7/6	1/3	-
	0	1/3	5/6	-1/6	-
	1	-1/6	5/6	1/3	-
	2	1/3	-7/6	11/6	-
4	-1	25/12	-23/12	13/12	-1/4
	0	1/4	13/12	-5/12	1/12
	1	-1/12	7/12	7/12	-1/12
	2	1/12	-5/12	13/12	1/4
	3	-1/4	13/12	-23/12	25/12

3. Define the constants d_r and $\tilde{d}_r = d_{k-1-r}$

r	k		
	2	3	4
0	2/3	3/10	4/35
1	1/3	3/5	18/35
2	-	1/10	12/35
3	-	-	1/35

4. Find the smoothness indicators. For $k = 2$ these can be written in a simple form

$$\beta_r = \left(\overline{\rho}_{i+1-r} - \overline{\rho}_{i-r}\right)^2. \tag{B.24}$$

The smoothness indicator form for $k = 3$ is

$$\beta_r = \frac{13}{12}\left(\sum_{j=0}^{k-1} b_{r,j}^{(0)}\overline{\rho}_{i-r+j}\right)^2 + \frac{1}{4}\left(\sum_{j=0}^{k-1} b_{r,j}^{(1)}\overline{\rho}_{i-r+j}\right)^2, \tag{B.25}$$

with coefficients as below.

r	$b^{(0)}: j$			$b^{(1)}: j$		
	0	1	2	0	1	2
0	1	-2	1	3	-4	1
1	1	-2	1	1	0	-1
2	1	-2	1	1	-4	3

For higher order WENO schemes, the smoothness factors of which are of the form [5]

$$\beta_r = \sum_{j=0}^{k-1}\left[\overline{\rho}_{i-r+j}\cdot\left(\sum_{g=0}^{k-1-j}\hat{b}_{r,j,g}\cdot\overline{\rho}_{i-r+j+g}\right)\right] \tag{B.26}$$

with $\hat{b}_{r,j,g}$ from

r	j	g			
		0	1	2	3
0	0	2107	-9402	7042	-1854
	1	11003	-17246	4642	-
	2	7043	-3882	-	-
	3	547	-	-	-
1	0	547	-2522	1922	-494
	1	3443	-5966	1602	-
	2	2843	-1642	-	-
	3	267	-	-	-
2	0	267	-1642	1602	-494
	1	2843	-5966	1922	-
	2	3443	-2522	-	-
	3	547	-	-	-
3	0	547	-3882	4642	-1854
	1	7043	-17246	7042	-
	2	11003	-9402	-	-
	3	2107	-	-	-

using the cell averaged density values $\overline{\rho}$.

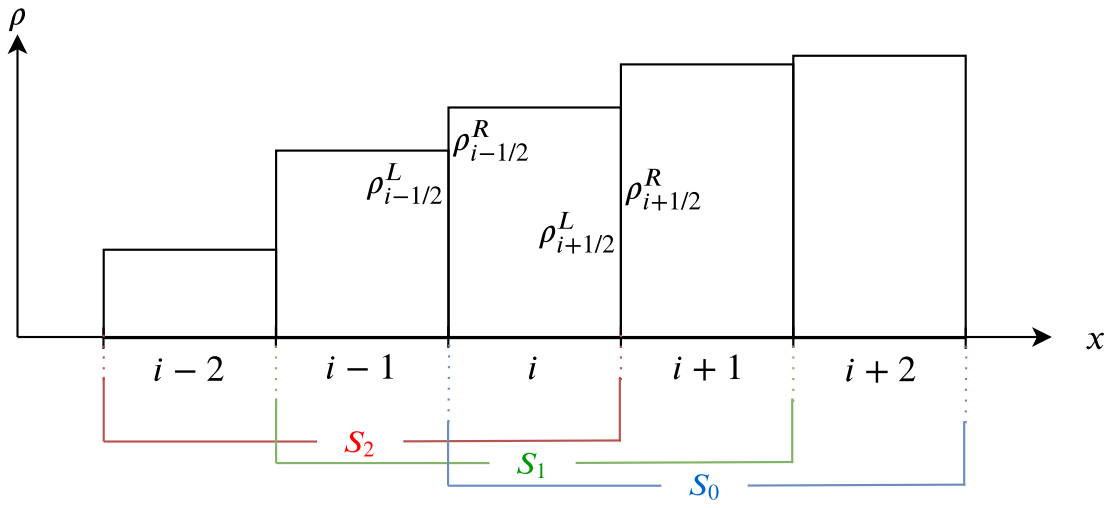


Figure B.4: WENO stencil.

5a Calculate alpha weights

$$\tilde{\alpha}_r = \frac{\tilde{d}_r}{(\epsilon + \beta_r)^2}, \tag{B.27}$$

$$\alpha_r = \frac{d_r}{(\epsilon + \beta_r)^2}, \tag{B.28}$$

using the weights d from step 3, and the smoothness indicators β from step 4. The constant $\epsilon = 10^{-6}$ to avoid a division by zero.

5b Calculate the omega weights

$$\tilde{\omega}_r = \frac{\tilde{\alpha}_r}{\tilde{\alpha}_0 + \tilde{\alpha}_1 + \tilde{\alpha}_2}, \tag{B.29}$$

$$\omega_r = \frac{\alpha_r}{\alpha_0 + \alpha_1 + \alpha_2}, \tag{B.30}$$

using the weights α from step 5a.

6 Evaluate the final interface reconstructions

$$\rho^-_{i+1/2} = \sum_{j=0}^{k-1} \omega_j \rho^{(j)}_{i+1/2}, \tag{B.31}$$

$$\rho^+_{i-1/2} = \sum_{j=0}^{k-1} \tilde{\omega}_j \rho^{(j)}_{i-1/2}, \tag{B.32}$$

using the weights ω from step 5b, and the polynomial reconstructed values $\rho^{(r)}_{i\pm1/2}$ from step 2.

B.2.1 Monotonicity Preserving Bounds

Proposing an improvement on Suresh and Huynh [60], Balsara and Shu [5] give a method to monotonically bound the reconstructed states. Suresh and Huynh [60] found that bounding local extrema reduces the order of accuracy and should be avoided for higher order schemes. The following presents a scheme for the monotonicity preserving bound proposed in [5]. This method is applied to the 7^{th} order WENO reconstruction in Appendix A.1.4. Begin by defining the minmod and median functions,

$$\text{minmod}(x, y) = \frac{1}{2}\left(\text{sign}(x) + \text{sign}(y)\right)\min\left(|x|, |y|\right), \tag{B.33}$$

$$\text{median}(x, y, z) = x + \text{minmod}(y - x, z - x). \tag{B.34}$$

In alignment with the the substeps a to e of step 7 in the WENO code in Appendix A.1.4, step $7a$ defines the local curvature measures d_j, d_{j+1}, d_{j-1} similarly by

$$d_j = \rho_{j+1} - 2\rho_j + \rho_{j-1}. \tag{B.35}$$

Step $7b$ improves on Suresh and Huynh [60] by taking the minmod, allowing local extrema to develop,

$$\rho^{MM}_{j+1/2} = \text{minmod}(d_j, d_{j+1}). \tag{B.36}$$

As well as the minmod evaluation (denoted in the superscript MM), an allowance is made for large curvature (LC) controlled by the parameter β. Step $7c$ defines β and α two curvature constant parameters. The value of β determines the freedom from allowing a large local curvature , and α determines the appropriate CFL number. Suresh and Huynh [60] claim setting $\alpha = 2$ and $\beta = 4$ allow a CFL of at least 0.6 and do not degrade the monotonicity preserving property. Step $7d$ define the median (MD), upper limit (UL) and large curvature (LC) density solutions. The following provide definitions for the right states, the left can be evaluated symmetrically,

$$\rho^{MD}_{j+1/2} = \frac{1}{2}\left(\rho_j + \rho_{j+1}\right) - \frac{1}{2}d^{MD}_{j+1/2}, \tag{B.37}$$

$$\rho^{UL}_{j+1/2} = \rho_j + \alpha\left(\rho_j - \rho_{j-1}\right), \tag{B.38}$$

$$\rho^{LC}_{j+1/2} = \rho_j + \frac{1}{2}\left(\rho_j - \rho_{j-1}\right) + \frac{\beta}{3}d^{LC}_{j-1/2}. \tag{B.39}$$

Step $7e$ defines new bounds for the maximum and minimum reconstructed states before computing the monotonicity preserving value.

$$\rho^{L,\min}_{j+1/2} = \max\left\{\min\left(\rho_j, \rho_{j+1}, \rho^{MD}_{j+1/2}\right), \min\left(\rho_j, \rho^{UL}_{j+1/2}, \rho^{LC}_{j+1/2}\right)\right\}, \tag{B.40}$$

$$\rho^{L,\max}_{j+1/2} = \min\left\{\max\left(\rho_j, \rho_{j+1}, \rho^{MD}_{j+1/2}\right), \max\left(\rho_j, \rho^{UL}_{j+1/2}, \rho^{LC}_{j+1/2}\right)\right\}. \tag{B.41}$$

Finally the monotonicity preserving bounds are calculated and returned as outputs of the WENO function,

$$\rho^{L}_{j+1/2} = \text{median}(\rho^{L}_{j+1/2}, \rho^{L,\min}_{j+1/2}, \rho^{L,\max}_{j+1/2}). \tag{B.42}$$

Within this approach there are many outcomes achieved by setting d^{MD}, d^{LC}, d^{MM} equal in various combinations. The paper from Balsara and Shu [5] tests these monotonicity preserving bounds on standard hyperbolic test cases.

Appendix C

Supplementary Material

C.1 HLLC Riemann Solver

In 1992 Toro [66] introduced a development of the HLL Riemann solver, assuming the two original waves present in HLL and inbetween adding in a contact wave[1].The HLLC fluxes are also given depending on the choice of left and right wave speeds as well as an middle speed given as S_*. The HLLC flux is given by four regions separated by three waves,

$$\mathbf{F} = \begin{cases} \mathbf{F}_L, & if \quad 0 \leq S_L, \\ \mathbf{F}_{*L} = \mathbf{F}_L + S_L \left(\mathbf{U}_{*L} - \mathbf{U}_L \right), & if \quad S_L \leq 0 \leq S_*, \\ \mathbf{F}_{*R} = \mathbf{F}_R + S_R \left(\mathbf{U}_{*R} - \mathbf{U}_R \right), & if \quad S_* \leq 0 \leq S_R, \\ \mathbf{F}_R, & if \quad S_R \leq 0. \end{cases}$$

The middle conserved vector $\mathbf{U}_{*K}$, for $K = L, R$, is expressed in 1D as

$$\mathbf{U}_{*K}^{1D} = \rho_K \left(\frac{S_K - u_K}{S_K - S_*} \right) \begin{bmatrix} 1 \\ S_* \\ \frac{E_K}{\rho_K} + (S_* - u_K) \left(S_* + \frac{p_K}{\rho_K (S_K - u_K)} \right) \end{bmatrix},$$

and in 2D as

$$\mathbf{U}_{*K}^{2D} = \rho_K \left(\frac{S_K - u_K}{S_K - S_*} \right) \begin{bmatrix} 1 \\ S_* \\ v_K \\ \frac{E_K}{\rho_K} + (S_* - u_K) \left(S_* + \frac{p_K}{\rho_K (S_K - u_K)} \right) \end{bmatrix}.$$

The middle wave speed S_* depends on the dimension of the problem, for 1D

$$S_*^{1D} = \frac{p_R - p_L + \rho_L u_L \left(S_L - u_L \right) - \rho_R u_R \left(S_R - u_R \right)}{\rho_L \left(S_L - u_L \right) - \rho_R \left(S_R - u_R \right)},$$

[1]The C in HLLC stands for contact wave.

however in 2D, the approximate Riemann solver has the conditions [65],

$$u_{*L} = u_{*R} = u_*, \quad and \quad S_* = u_*,$$

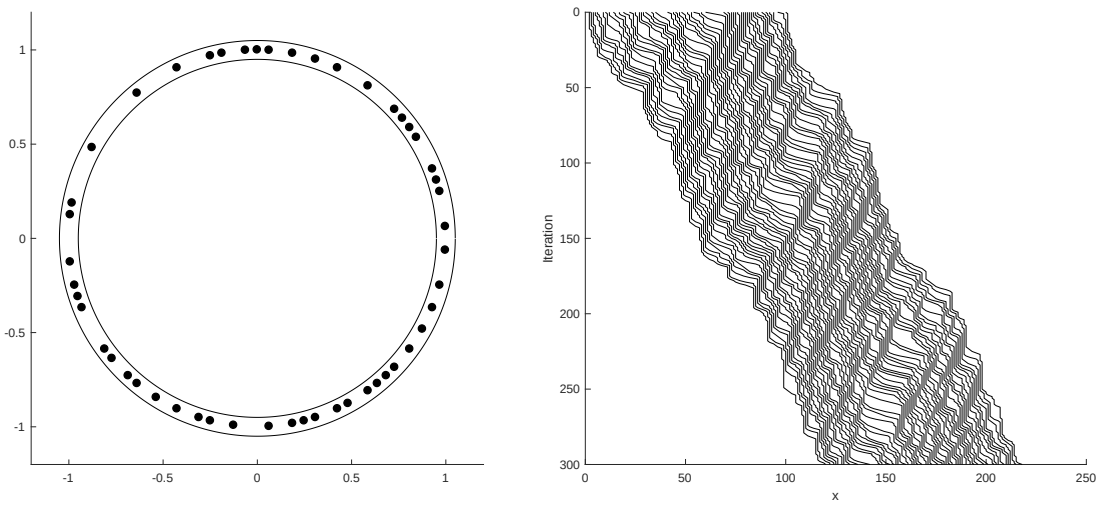
so the middle wave speed in 2D is

$$S_*^{2D} = u_*.$$

All flow variables in the left or right states, including left and right fluxes, can be calculated from the variables given in the conserved vector $\mathbf{U}$.

C.2 Nagel-Schreckenberg Model

See Section 2.2.2, and code in Appendix A.3.1.



(a) Physical location

(b) Pathline clustering waves

Figure C.1: Results from 300 iterations of the Nagel-Schreckenberg [45] model. Figures show physical position on a circular road (a), and displacement x from the road origin traced by iteration to show phantom traffic jams.

C.3 Genealogical Traffic Flow Model Tree

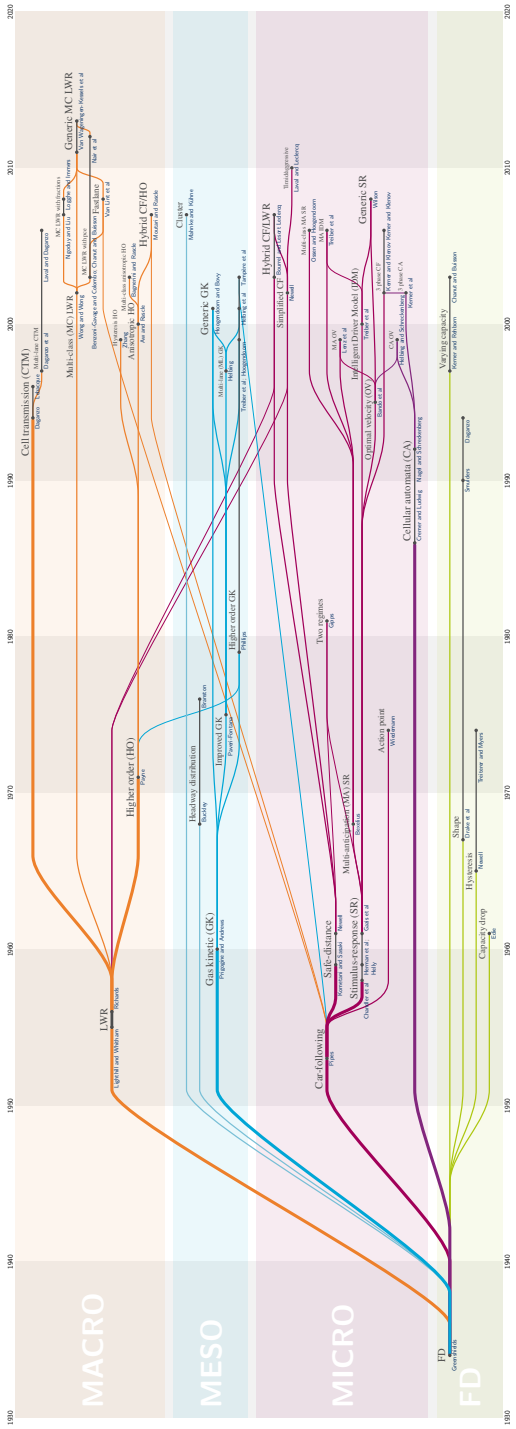


Figure C.2: Part of the online access [76] (doi: `10.1007/s13676-014-0045-5`) includes a full size version complete with references.

C.4 Simulation Information Output

C.4.1 Written Text File

```
TFM Simulation Information

12/08/2019 11:25:38


Methodology:
Spatial step                    1.000000e-02
Final time                      1.000000e+00
CFL constraint                  9.000000e-01
Reconstruction method           1st Order single cell average


Time breakdown:
Code Section                    Time [s]
-----------------------------------------------
Defining map and error check    6.170273e-04
Initialising                    4.893088e-02
Time loop                       2.290685e+02
- - - - - - - - - - - - - - - - - - - - - - - -
Junction solver                 9.541947e+00
Spatial reconstruction          5.088826e+01
Numerical flux calculation      9.922855e+01
Runge-Kutta updates             4.046449e+01
- - - - - - - - - - - - - - - - - - - - - - - -
Save results                    8.729287e+00
-----------------------------------------------
-----------------------------------------------
Total program time              2.378474e+02
-----------------------------------------------


Line Count:
File                            Number of Lines
-----------------------------------------------
main.py                         862
define_map.py                   190
params.txt                      7
MUSCLReconstruction.py          193
WENOReconstruction.py           435
-----------------------------------------------
Total                           1687
-----------------------------------------------


Output File Memory:
File                            Size [MB]
-----------------------------------------------
density.txt                     346.752
-----------------------------------------------
```

C.4.2 Console

```
|- - - - - Simulation Complete - - - - -|
 Spatial step              1.000000e-02
 Final time                1.000000e+00
 CFL                       9.000000e-01
 Velocity Model            Greenshields
 Riemann solver            LaxFriedrichs
 Reconstuction method      FirstOrder
|- - - - - - - - - - - - - - - - - - - - -|
```