The Discontinuous Galerkin Method for Conservation Laws

Michael, Crabb

2010

MIMS EPrint: **2015.63**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

THE DISCONTINUOUS GALERKIN METHOD FOR CONSERVATION LAWS

A dissertation submitted to the University of Manchester
for the degree of Master of Science
in the Faculty of Engineering and Physical Sciences

2010

Michael Crabb
School of Mathematics

Contents

Word count 15653

List of Tables

List of Figures

The University of Manchester

Michael Crabb
Master of Science
The Discontinuous Galerkin Method For Conservation Laws
October 14, 2010

The aim of this project is to study discontinuous Galerkin methods applied to coupled systems of partial differential equations in conservative form in 1D and 2D.

In 1D, a formulation was successfully implemented to solve continuous problems for the advection and shallow water equations. Discontinuous problems for the inviscid Burgers' equation and a breaking dam problem were also investigated and the effectiveness of h- and p-refinement discussed. An alternate set of shallow water equations were derived yielding equivalent results for a continuous problem but different numerical solutions for the breaking dam problem. These anomalous results highlight the importance of enforcing conservation of the correct conserved physical variables in cases when solutions exhibit shocks.

A 2D slope limiter, applicable to quadrilateral elements, is implemented and numerical results obtained for a smoothed breaking dam problem in 2D. A comparison is made between these results and those from a finite volume method (results by Chris Johnson) and indicate that, for this particular problem, both methods resolve the shock over the same length scale.

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright Statement

i. Copyright in text of this dissertation rests with the author. Copies (by any process) either in full, or of extracts, may be made **only** in accordance with instructions given by the author. Details may be obtained from the appropriate Graduate Office. This page must form part of any such copies made. Further copies (by any process) of copies made in accordance with such instructions may not be made without the permission (in writing) of the author.

ii. The ownership of any intellectual property rights which may be described in this dissertation is vested in the University of Manchester, subject to any prior agreement to the contrary, and may not be made available for use by third parties without the written permission of the University, which will prescribe the terms and conditions of any such agreement.

iii. Further information on the conditions under which disclosures and exploitation may take place is available from the Head of the School of Mathematics.

Acknowledgements

A special thank you to my supervisor, Dr. Andrew Hazel, for developing my understanding of the `oomph-lib` and for his useful discussions and comments during the past five months. I would also like to thank Prof. Matthias Heil for his help during the project and Chris Johnson for his two dimensional dam-break results, which were essential for validation of my computational results.

Abbreviations

The following abbreviations are commonly used in this report:

- DG - Discontinuous Galerkin.
- FEM - Finite element method.
- FVM - Finite volume method.
- PDE - Partial differential equation.
- ODE - Ordinary differential equation.
- GLL - Gauss-Lobatto-Legendre.
- RK - Runge-Kutta.
- CFL - Courant-Friedrichs-Levy.
- TVDM - Total variation diminishing in the means.
- TVBM - Total variation bounded in the means.

Chapter 1

Introduction

1.1 Conservative PDEs in one dimension

In this report, coupled systems of partial differential equations (PDEs) in conservative form will be studied. First, consider a system of one conserved field, w, in one spatial dimension, x. A simple conservation law for this single field states that "the rate of change of the total amount of w inside a region $[L, R]$ equals the amount of w that flows into $[L, R]$ minus the amount that flows out of $[L, R]$". This statement is expressed mathematically below, where f represents the flux, or flow, and t is the time:

$$\frac{d}{dt}\int_L^R w(x,t)\,dx = f(L,t) - f(R,t). \tag{1.1}$$

If f is continuously differentiable, then the integral form (1.1) can be written

$$\frac{d}{dt}\int_L^R w\,dx = -\int_L^R \frac{\partial f}{\partial x}dx,$$

and if the field, w, is continuously differentiable we can write

$$\int_L^R \Big(\frac{\partial w}{\partial t} + \frac{\partial f}{\partial x}\Big)dx = 0.$$

The integral above is true for any control region $[L, R]$ and this is only possible if the integrand equals 0. We recover the PDE form of the conservation law,

$$\frac{\partial w}{\partial t} + \frac{\partial f}{\partial x} = 0. \tag{1.2}$$

1.2 Abstract system of conservative PDEs

Systems of PDEs relating m conserved fields in n spatial dimensions will be considered. In the spirit of the 1D conservation law (1.2), a general system of conserved PDEs can be written in the form,

$$\frac{\partial \mathbf{w}(\mathbf{x},t)}{\partial t} + \nabla.F(\mathbf{w}(\mathbf{x},t),\mathbf{x},t) = \mathbf{0} \quad \mathbf{x} \in \Omega, \tag{1.3}$$

where $\Omega \subset \mathbb{R}^n$ is the problem domain, $\mathbf{x} \in \Omega$ is a spatial location, t is the time, $\mathbf{w} \in \mathbb{R}^m$ is a vector of the unknown fields and $\mathbf{F} \in \mathbb{R}^{m \times n}$ is the flux matrix. The vector of partial derivatives, $\nabla = (\frac{\partial}{\partial x_1}, \ldots, \frac{\partial}{\partial x_n})$, is defined to operate across rows of the flux matrix.

Conserved PDEs will be considered with Dirichlet initial conditions (1.4) and either periodic boundary conditions or Dirichlet boundary conditions (1.5) where $\partial\Omega$ is the boundary of the domain and t_0 is the starting time of the system,

$$\mathbf{w}(\mathbf{x},t_0) = \mathbf{f}(\mathbf{x}) \quad \mathbf{x} \in \Omega, \tag{1.4}$$

$$\mathbf{w}(\mathbf{x},t) = \mathbf{g}(\mathbf{x},t) \quad \mathbf{x} \in \partial\Omega. \tag{1.5}$$

For specific initial and boundary conditions, solutions to PDEs in conservative form are known to develop shocks, in which the solution becomes multi valued at a single point [1]. A classic finite element method, which imposes continuity through the domain, is destined to fail because the exact solution lies outside the space of continuous functions. A number of methods have been implemented to overcome these discontinuities such as the finite volume method (FVM) [2]. In this method, the domain is split into individual cells, and in each of these cells, a constant approximation is imposed for the field. A flux function, defined on the cell boundary, is then used to distribute information from the cell to its neighbours. During the timestepping, "spurious" oscillations can appear near discontinuities of the solution due to the Gibbs phenomenon, where errors occur due to approximating a discontinuous function by a continuous function [3, 4]. A number of methods have been devised to remove the spurious oscillations in the approximate solution after each timestep. If the mesh has been appropriately refined to accommodate the local physical features of the solution, then the FVM is known, through numerical experiments, to perform well. However,

the method does not allow high order polynomial approximation within a cell, which could be a useful property for the approximate solution to exhibit in smooth regions.

A discontinuous Galerkin finite element method (DGFEM) is the natural successor to the finite volume method. In this method the domain is split into elements and in each element a polynomial approximation is used. Continuity of the approximate solutions is not enforced between neighbouring elements. Instead, a numerical flux function is defined on the element edges to distribute information between the element and its neighbours. A timestepping scheme is then used to progress the approximate solution in each element at discrete points in time. Numerical schemes have been developed, including aliasing, filtering and limiting [3], to remove oscillations that appear due to the Gibbs phenomenon at each timestep. In this report, slope limiting will be used to stabilise the solution by enforcing the approximate solution to lie in the space of functions with bounded variation.

The slope limiting process works by altering the approximation within each element after each timestep. If an element requires limiting, high order approximation in an element will be replaced with a linear approximation. If the gradients of the element and its neighbours differ in sign, a constant approximation will be used within an element, and, if not, a linear approximation is used with the smallest gradient in absolute value. A modified limiter can also be implemented to ensure that the solution remains high-order accurate in regions where smooth extrema are present.

1.3 Report Outline

A discontinuous Galerkin formulation of the general system of conservative PDEs (1.3) will be derived in chapter 2, including space and time discretization. This includes a discussion of slope limiting in one spatial dimension and the concept of total variation bounded solutions. In chapter 3, computational results will be given for problems with continuous analytic solutions for the one dimensional advection equation and one dimensional shallow water equations. Chapter 4 has a theoretical discussion of one dimensional shock propagation and formation with specific examples relating to the inviscid Burgers' equation. Theoretical discontinuous analytic solutions will be compared to computational results highlighting the importance of

a slope limiter to remove oscillations. A one dimensional dam-break problem of the shallow water equations (see section 5.1) will be analysed, which has a known analytic solution. The analysis will include looking at two different fluxes, the Roe average and Lax-Friedrichs flux, and the effectiveness of performing h- and p-refinement with slope limiting. This chapter will include a discussion on the importance of enforcing conservation of the correct conserved physical variables when solutions exhibit discontinuities. See section 4.2.4 for further details.

Finally, in chapter 5, a two dimensional dam-break problem will be studied, including a discussion of a two dimensional slope limiter applicable to rectangular elements with a linear approximation. The discontinuous Galerkin finite element method results obtained will be compared to those obtained using a finite volume method devised by Jiang *et al.* [5].

Unless otherwise specified, computational results are obtained using the object oriented multi physics library (`oomph-lib`), written in C++. For more details of the library see [6]. The implementation of the numerical flux function for the shallow water equations is described in Appendix B, a two dimensional slope limiter in Appendix C and a third order Runge-Kutta timestepper in Appendix D. Appendix E also describes a stand-alone original C++ code to solve a continuous test problem of the shallow water equations in one dimension.

The computations are performed on an AMD Turion(tm) X2 Dual-Core Mobile RM-74 550MHz processor, running Ubuntu Linux 10.04 LTS, and code is compiled with the GCC 4.5.1 compiler.

Chapter 2

Preliminaries

2.1 A discontinuous Galerkin formulation

To model a set of coupled PDEs in conservative form we consider a weak formulation of (2.1),

$$\frac{\partial \mathbf{w}(\mathbf{x},t)}{\partial t} + \nabla . F(\mathbf{w}(\mathbf{x},t),\mathbf{x},t) = \mathbf{0} \quad \mathbf{x} \in \Omega, \tag{2.1}$$

where $\Omega \subset \mathbb{R}^n$ is the domain, $\mathbf{w} \in \mathbb{R}^m$ is a vector of the unknown fields and $\mathbf{F} \in \mathbb{R}^{m \times n}$ is the flux matrix. The vector of partial derivatives, $\nabla = (\frac{\partial}{\partial x_1}, \ldots, \frac{\partial}{\partial x_n})$, is defined to operate across rows of the flux matrix. The computational domain, Ω_h, is split into K geometry conforming elements, indexed by e. We look for an approximation, $\mathbf{w}_h(x,t)$, to the true solution, $\mathbf{w}(x,t)$, by a direct sum of local solutions, $\mathbf{w}_h^e(x,t)$, defined in each element,

$$\mathbf{w}(x,t) \approx \mathbf{w}_h(x,t) = \bigoplus_{e=1}^{K} \mathbf{w}_h^e(x,t) \in \mathbf{V}_h = \bigoplus_{e=1}^{K} \text{span}\{\Psi_l(D^e)\}_{l=1}^{N_p} ,$$

where an n-dimensional polynomial basis, $\{\Psi_l(D^e)\}_{l=1}^{N_p}$, is defined in each element. The local approximation, $\mathbf{w}_h^e(x,t)$, is then expressed as

$$\mathbf{w}_h^e(\mathbf{x},t) = \sum_{i=1}^{N_p} \mathbf{w}_h^e(\mathbf{x}_i^e,t) l_i^e(\mathbf{x}) = \sum_{i=1}^{N_p} (\mathbf{w}_h)_i^e l_i^e(\mathbf{x}), \tag{2.2}$$

where $l_i^e(\mathbf{x})$ is the Lagrange interpolating polynomial defined on a set of grid points, $\mathbf{x}_i^e$, in element D^e. The normal interpolation property (2.3) holds in element e:

$$l_i^e(\mathbf{x}_j^e) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases} \tag{2.3}$$

The choice of the position of the grid points or nodes is not unique and although equidistant grid points are the most simple to imagine, more complex positioning of nodes can lead to better computational performance, as discussed in section 2.5. For now, assume the nodal positions have been fixed, and in each element form the residual,

$$\mathbf{R}_h^e(\mathbf{x},t) = \frac{\partial \mathbf{w}_h^e}{\partial t} + \nabla.F_h^e(\mathbf{w}_h^e,\mathbf{x},t).$$

If $\mathbf{R}_h^e(\mathbf{x},t) = \mathbf{0}\ \forall \mathbf{x} \in \Omega$ and $\forall t$ then the PDE (2.1) is satisfied. The residual is enforced to be orthogonal to all test functions $v_h^e \in \mathbf{V}_h^e$, resulting in the local statement:

$$\int_{D^e} \mathbf{R}_h^e(\mathbf{x},t) v_h^e(\mathbf{x}) d\mathbf{x} = \mathbf{0}.$$

In a Galerkin formulation, the test functions are precisely the Lagrange polynomials used to interpolate the unknowns

$$\int_{D^e} \Big(\frac{\partial \mathbf{w}_h^e}{\partial t} + \nabla.F_h^e(\mathbf{w}_h^e,\mathbf{x},t) \Big) l_i^e(\mathbf{x}) d\mathbf{x} = \mathbf{0} \qquad i = 1 : N_p.$$

To simplify notation, it is assumed that the element indexed by e is under consideration and that a particular discretization choice has been made, so the labels e and h can be dropped. Using Green's Theorem on the second term, on each of the m rows of the flux matrix

$$\int_{D^e} \Big(\frac{\partial \mathbf{w}}{\partial t} l_i(\mathbf{x}) - F.\nabla l_i(\mathbf{x}) \Big) d\mathbf{x} = -\int_{\partial D^e} F.\hat{\mathbf{n}} l_i(\hat{\mathbf{x}}) d\hat{\mathbf{x}} \qquad i = 1 : N_p,$$

where $\hat{\mathbf{n}}$ is the outer unit normal on ∂D^e. The term on the right hand side represents an integral of the flux function on the boundary of the element so in 2D this will be a line integral and in 3D a surface integral. In 1D, the analysis is simplified because the outer unit normal is either $+1$ on the right hand side of the element or -1 on the left hand side of the element, and an integration need not be performed.

The essential concept behind a discontinuous formulation is that continuity of the fields between adjacent elements is not enforced. At an element's edge, the field can be multivalued and so the flux at the edge can also be multivalued. Consider two elements with a common edge, then the surface integral of the normal flux along the edge of one element can be different to the neighbouring element's surface integral of the flux along the same edge. A choice must be made to which flux, or combination

of fluxes, is the best approximation to the true flux at an element's edge. In the discontinuous Galerkin formulation, at an element's edge, the flux in the direction of the outer unit normal, $F.\hat{\mathbf{n}}$, is replaced by a numerical flux, $\mathbf{f}^*$. This results in a weak formulation, find $\mathbf{w} \in \mathbf{V}_h^e$ such that:

$$\int_{D^e} \Big(\frac{\partial \mathbf{w}}{\partial t} l_i(\mathbf{x}) - F.\nabla l_i(\mathbf{x}) \Big) d\mathbf{x} = - \int_{\partial D^e} \mathbf{f}^* l_i(\hat{\mathbf{x}}) d\hat{\mathbf{x}} \qquad i = 1 : N_p. \tag{2.4}$$

2.2 Numerical Flux

Before constructing a numerical flux, the geometry of the field at element edges must be understood. Consider two adjacent elements, K^+ and K^-, a point $\mathbf{x}$ on their common edge, and let $\mathbf{n}_{K^\pm}$ be the outer unit normal from element $K^\pm$. The limits of the approximate solution at the edge of each element are

$$\mathbf{w}_h^\pm(\mathbf{x}) = \lim_{\epsilon \to 0^\pm} \mathbf{w}_h(\mathbf{x} - \epsilon \mathbf{n}_{K^\pm}). \tag{2.5}$$

The geometry, for rectangular elements, can be seen in figure 2.1 below.

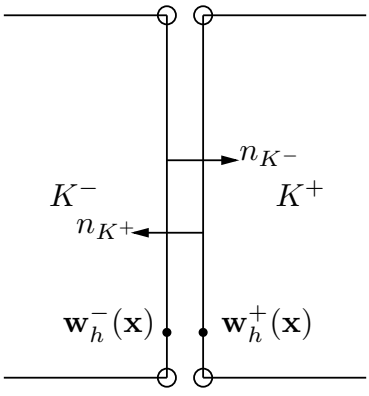

Figure 2.1: *Illustration of the outer unit normals and interior and exterior fields,* $\mathbf{w}_h^-(\mathbf{x})$ *and* $\mathbf{w}_h^+(\mathbf{x})$ *on a common edge of two adjacent elements* K^- *and* K^+.

Cockburn *et al.* [7] construct the numerical flux, $\mathbf{f}^*$, an approximation to the true flux in the direction of the outer unit normal at an element's boundary, by considering the following:

1. The numerical flux is taken to be a function only of the limits (2.5). This defines a numerical flux at the boundary regardless of the polynomial space chosen for

the solution,

$$\mathbf{f}^*(\mathbf{w}_h)(\mathbf{x}) = \mathbf{f}^*(\mathbf{w}_h^+(\mathbf{x}), \mathbf{w}_h^-(\mathbf{x})).$$

2. For continuous problems, the numerical flux must be equivalent to the normal flux at the boundary: $\mathbf{f}^*(\mathbf{w}, \mathbf{w}) = F(\mathbf{w}).\mathbf{n}$.

3. If a piecewise constant approximation is used, the discretization results in a monotone finite volume scheme. This is ensured if we have a conservative flux,

$$\mathbf{f}^*(\mathbf{w}_h^+(\mathbf{x}), \mathbf{w}_h^-(\mathbf{x})) + \mathbf{f}^*(\mathbf{w}_h^-(\mathbf{x}), \mathbf{w}_h^+(\mathbf{x})) = \mathbf{0},$$

the mapping $\mathbf{w} \mapsto \mathbf{f}^*(\mathbf{w}, .)$ is non decreasing and the mapping $\mathbf{w} \mapsto \mathbf{f}^*(., \mathbf{w})$ is non increasing.

A number of fluxes exist that display the properties above including the Lax-Friedrichs, Godunuv and Engquist-Osher fluxes [3, 7, 8]. For the following definitions, assume that $\mathbf{w}_h^-(\mathbf{x})$, $\mathbf{w}_h^+(\mathbf{x})$ are the fields on the interior and exterior of the element boundary and $\mathbf{n}$ is the outward unit normal.

2.2.1 Central Flux

The central flux at an element boundary is defined as the average of the fluxes at the element boundary in the direction of the outer unit normal,

$$\mathbf{f}^*{}_{\text{CEN}}(\mathbf{w}_h^-(\mathbf{x}), \mathbf{w}_h^+(\mathbf{x})) = \frac{1}{2}(\mathbf{F}(\mathbf{w}_h^-(\mathbf{x})) + \mathbf{F}(\mathbf{w}_h^+(\mathbf{x}))).\mathbf{n}. \tag{2.6}$$

Although this is the simplest and perhaps most intuitive flux, it does not necessarily satisfy the third desired property of a numerical flux and is generally only useful for continuous problems.

2.2.2 Lax-Friedrichs Flux

The Lax-Friedrichs flux, defined in (2.7), is the flux average in the direction of the outer unit normal plus a term that is a constant multiplied by the difference between the internal and external fields,

$$\mathbf{f}^*{}_{LF}(\mathbf{w}_h^-(\mathbf{x}), \mathbf{w}_h^+(\mathbf{x})) = \mathbf{f}^*{}_{\text{CEN}}(\mathbf{w}_h^-(\mathbf{x}), \mathbf{w}_h^+(\mathbf{x})) - \frac{C_{LF}}{2}(\mathbf{w}_h^+(\mathbf{x}) - \mathbf{w}_h^-(\mathbf{x})),$$

$$C_{LF} = \max_{\mathbf{w} \in (\mathbf{w}_h^-(\mathbf{x}), \mathbf{w}_h^+(\mathbf{x}))} |\lambda(B)|, \tag{2.7}$$

where $B = \mathbf{n}.\frac{\partial F}{\partial \mathbf{w}} \in \mathbb{R}^{m\times m}$ is the contraction of the Jacobian of the flux matrix, a rank 3 tensor, with the outer unit normal. Using the Einstein summation convention, the components are

$$B_{ik} = \frac{\partial F_{ij}}{\partial w_k} n_j \qquad i,k = 1:m, \quad j = 1:n.$$

The eigenvalues of the normal matrix, B, are computed and the Lax-Friedrichs constant, C_{LF}, is chosen as the largest eigenvalue in absolute value over all possible choices of the field between $\mathbf{w}_h^-(\mathbf{x})$ and $\mathbf{w}_h^+(\mathbf{x})$.

The flux adds an extra diffusive term to the central flux in an attempt to smear out discontinuities and shocks arising from the solution of the conserved PDEs.

2.2.3 Roe average Flux

The Roe average flux is an alternative method to assign numerical fluxes at element boundaries. Consider the conservative initial value problem in one spatial dimension

$$\frac{\partial \mathbf{w}}{\partial t} + \frac{\partial \mathbf{f}}{\partial x} = \mathbf{0}, \tag{2.8}$$

$$\mathbf{w}(x,0) = \mathbf{w}_0(x). \tag{2.9}$$

The Riemann problem is the initial value problem with discontinuous data

$$\mathbf{w}(x,0) = \mathbf{w}_L \ \ (x<0), \qquad \mathbf{w}(x,0) = \mathbf{w}_R \ \ (x>0). \tag{2.10}$$

Roe [9] solved a linearised version of the Riemann problem, at each element edge, to find a field that approximately solves the Riemann problem and hence determine a numerical flux. He introduced so called Roe average states, and used the jump conditions at the discontinuity (see equation (4.17)), to find approximate solutions to the Riemann problem. Specifically, for the shallow water equations in 1D (3.9), this flux can be written in the form,

$$\mathbf{f}^*{}_{RA}(\mathbf{w}_h^-(\mathbf{x}), \mathbf{w}_h^+(\mathbf{x})) = \mathbf{f}^*{}_{\mathrm{CEN}}(\mathbf{w}_h^-(\mathbf{x}), \mathbf{w}_h^+(\mathbf{x})) - \frac{C_{RA}}{2}(\mathbf{w}_h^+(\mathbf{x}) - \mathbf{w}_h^-(\mathbf{x})), \tag{2.11}$$

where C_{RA} is the Roe average constant. For the shallow water equations, the conserved field is $\mathbf{w} = (h, hu)^T$, and by introducing the Roe average states at each edge,

$$\bar{h} = \frac{h_- + h_+}{2}, \qquad \bar{u} = \frac{\sqrt{h_-}u_- + \sqrt{h_+}u_+}{\sqrt{h_-} + \sqrt{h_+}}, \tag{2.12}$$

the constant is calculated as

$$C_{RA} = \max_{\bar{h},\bar{u}} |\bar{u} \pm \sqrt{g\bar{h}}|, \tag{2.13}$$

which is a simple optimization problem over two different values. For further details of the theoretical justification of this numerical flux see [4, 9, 10].

A general implementation of the numerical flux into the `oomph-lib` is described in more detail in Appendix B.

2.3 Elemental linear system

Returning to the weak form (2.4) and substituting the local solution approximation in element e (2.2), there are N_p equations to be solved for each component of the field corresponding to the values at the N_p grid points,

$$\int_{D^e} \frac{\partial \mathbf{w}_j}{\partial t} l_j(\mathbf{x}) l_i(\mathbf{x}) d\mathbf{x} = \int_{D^e} F.\nabla l_i(\mathbf{x}) d\mathbf{x} - \int_{\partial D^e} \mathbf{f}^* l_i(\hat{\mathbf{x}}) d\hat{\mathbf{x}} \quad i = 1 : N_p, \tag{2.14}$$

where $\mathbf{w}_j$ is a vector of the unknown fields at the nodal position $\mathbf{x}_j$. Using the Einstein summation convention, it is useful to define an element mass matrix, M, flux matrix, F, and numerical flux matrix, F^*:

$$M_{ki} = \int_{D^e} l_k(\mathbf{x}) l_i(\mathbf{x}) d\mathbf{x} \quad k, i = 1 : N_p \tag{2.15}$$

$$F_{ki} = \int_{D^e} F_{kj} \frac{\partial l_i(\mathbf{x})}{\partial x_j} d\mathbf{x} \quad i, j = 1 : N_p \quad k = 1 : m \tag{2.16}$$

$$F^*_{ki} = \int_{\partial D^e} f^*_k l_i(\hat{\mathbf{x}}) d\hat{\mathbf{x}} \quad i = 1 : N_p \quad k = 1 : m \tag{2.17}$$

Equation (2.14) is equivalent to a system of $m \times N_p$ ordinary differential equations (ODEs), that must be solved in every element. These ODEs can be written in the form,

$$\begin{pmatrix} M & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & M \end{pmatrix} \begin{pmatrix} \dot{\mathbf{w}}^1 \\ \vdots \\ \dot{\mathbf{w}}^m \end{pmatrix} = \begin{pmatrix} \mathbf{f}_1 \\ \vdots \\ \mathbf{f}_m \end{pmatrix} + \begin{pmatrix} \mathbf{g}^*_1 \\ \vdots \\ \mathbf{g}^*_m \end{pmatrix}, \tag{2.18}$$

where M is the element mass matrix defined in (2.15) and the vectors $\mathbf{f}_i$ and $\mathbf{g}^*_i$ are the i^{th} column of the flux and numerical flux matrices defined in (2.16) and (2.17)

respectively. $\mathbf{w}^i$ is a vector of the nodal values across the element corresponding to the i^{th} field[1].

2.3.1 Element coupling and boundary conditions

In the elemental linear system (2.18) the only communication between elements is through the evaluation of the numerical flux matrix (2.17). The entries of this matrix are line integrals of the numerical flux along the element boundary and so coupling is required between adjacent faces in the mesh.

In the implementation, for 2D rectangular elements, every element in the mesh, named a bulk element, is assigned an index e. Every bulk element is assigned a pointer to each of the four faces, named face elements, and on each of these faces, a pointer is assigned to the neighbouring face. In every bulk element the entries of the numerical flux matrix are computed by moving to one of the faces, finding this face's neighbour, and then evaluating the numerical flux at every integration knot point along the face as illustrated in figure 2.2. See section 2.4 for integration scheme.

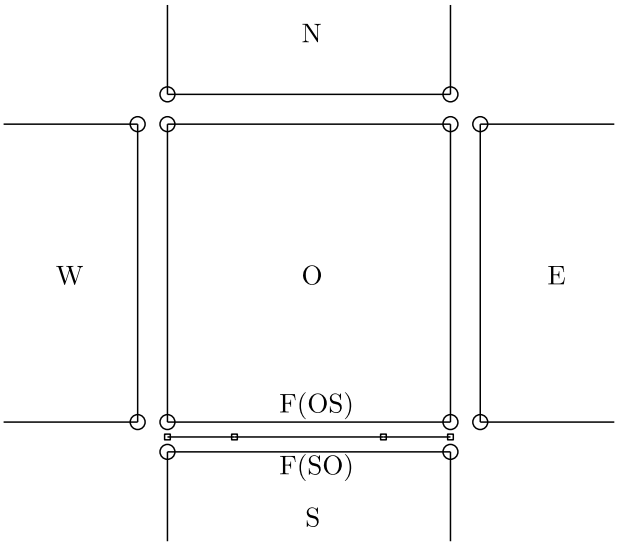

Figure 2.2: *The figure illustrates a rectangular (bulk) element in the mesh, O, and it's neighbours N, E, S and W. One of the faces $F(OS)$ and its neighbouring face $F(SO)$ are illustrated along with the numerical flux integration knot points (squares) along the edge for a Q_4 finite element (bi-cubic approximation).*

[1]The superscript notation for the unknowns, $\mathbf{w}^i$ $i = 1 : m$, corresponds to a vector of unknowns across the nodal positions of the element for a given field component i, whereas the subscript notation, $\mathbf{w}_j$ $j = 1 : N_p$, corresponds to a vector of the unknown fields at a given nodal point $\mathbf{x}_j$

Boundary Conditions

For an element lying on the domain boundary, one or more of the faces also lies on the domain boundary and so there is no neighbouring face to assign a pointer to. This is not a problem for periodic boundary conditions because the neighbouring face is simply set as the face on the opposite side of the mesh.

For Dirichlet boundary conditions the field at the boundary is known. The pointer to the neighbouring face is set to point to the face element itself. This ensures the numerical flux function will return the physical normal flux, $\mathbf{f}^*(\mathbf{w}, \mathbf{w}) = F(\mathbf{w}).\mathbf{n}$, because of the second property of a numerical flux in section 2.2. The field value itself is fixed, as required by the boundary condition.

2.4 Evaluation of Integrals

In general, the integrals in equations (2.15)-(2.17) are computed numerically. For 2D rectangular elements, the standard procedure of mapping element e to the reference element $[-1, 1] \times [-1, 1]$, is achieved through an isoparametric mapping. For more details of reference elements and mappings see [4, chapter 4]. The mass matrix (2.15) and flux matrix (2.16) will then be computed through an appropriate quadrature rule. For the flux and mass matrix, the $P \times P$ Gauss-Lobatto-Legendre (GLL) rule will be used,

$$\int_{-1}^{1}\int_{-1}^{1} f(\epsilon, \eta) d\eta d\epsilon \approx \sum_{l,m=1}^{P} f(\epsilon_l, \eta_m)\omega_l\omega_m,$$

where ϵ_m and η_l are the GLL knot points associated with the weights w_m and w_l respectively [4]. The $P \times P$ GLL rule (which has $P \times P$ integration points) has the property that it exactly integrates all tensor product polynomials, f, up to order $2P - 3$ [4].

The numerical flux matrix, (2.17), is a line integral, and a 1D quadrature rule can be used. In the same spirit as the surface integrals, each element edge will be mapped to the reference line segment $[-1, 1]$, and the integrals computed through the P GLL quadrature rule,

$$\int_{-1}^{1} f(\epsilon) d\eta \approx \sum_{l=1}^{P} f(\epsilon_l)\omega_l, \tag{2.19}$$

where ϵ_l are the Gauss knot points associated with the weights w_l and is exact for polynomials f up to order $2P-3$.

For a 1D method the analysis is simplified because only a 1D GLL quadrature rule is required for the mass and flux matrix. Integration is not even required for the numerical flux matrix because this is just a function evaluation at either side of the element.

2.5 Orthogonal polynomial basis

Previously, it was stated that different choices of the basis functions, and hence nodal positions, can lead to improved computational performance. To solve the set of ODEs (2.18) numerically, the inverse of the block diagonal mass matrix must be computed. At high orders, equidistant node polynomials are nearly orthogonal, resulting in ill conditioned mass matrices and hence reducing the accuracy of a computed solution [3]. Also, using non-diagonal mass matrices mean that solving the linear system of ODEs (2.18) can be slower than using a diagonal system.

Consider a scalar field, in one spatial dimension, with the Legendre polynomials as basis functions. It is possible to use their L^2-orthogonality condition,

$$\int_{-1}^{1} P_l(s)P_{l'}(s)ds = \frac{2}{2l+1}\delta_{ll'},$$

by representing the approximate solution, in element j, as:

$$w_h^j(x,t) = \sum_{i=1}^{N_p} w_i^j \psi_i^j(x), \quad \psi_i^j(x) = P_i(2(x-x_{j+1/2})/\Delta_j), \quad \Delta_j = x_{j+1} - x_j. \quad (2.20)$$

The mass matrix is diagonal, $M_{ij} = \frac{2l+1}{\Delta_j}\delta_{ij}$, and so the matrix inversion, for this particular choice of basis, is trivial by inverting each of the diagonal components in turn. In the `oomph-lib`, an orthonormal polynomial basis, with the Lagrange interpolation nodes (2.3) defined on the GLL quadrature points, is implemented.

For any sensible choice of basis functions, the matrix inversion is always possible, and the system of ODEs (2.18) can be written in the form (2.21), where $\mathbf{w}$ is a vector of the nodal values across an element over the fields and $\mathbf{L}_h$ is the discretized

approximation to $-M^{-1}\nabla_h.F$, where M is the total mass matrix,

$$\dot{\mathbf{w}} = \mathbf{L}_h(\mathbf{w}). \tag{2.21}$$

2.6 Runge-Kutta Time Discretization

To solve the system of ODEs (2.21), the explicit Runge-Kutta class of timesteppers will be used to march the solution through time. In this report the flux matrix, and therefore the vector $\mathbf{L}_h(\mathbf{w})$ (2.21), will have no explicit time dependence, simplifying the classes of timesteppers required to solve the ODEs. The classic four stage Runge-Kutta (RK-4) timestepper and a class known as the total variation diminishing Runge-Kutta (RK-TVD) timesteppers are outlined in this discussion.

A general explicit Runge-Kutta scheme to solve the ODEs (2.21), at step n, with a fixed step size Δt, can be written [11]:

1. Set $\mathbf{w}_h^{(0)} = \mathbf{w}_h^n$
2. For $i = 1 : k$ compute explicit Euler steps
$$\mathbf{w}_h^{(i)} = \sum_{l=0}^{i-1} \alpha_{il}\mathbf{v}_h^{il} \text{ where } \mathbf{v}_h^{il} = \mathbf{w}_h^{(l)} + \frac{\beta_{il}}{\alpha_{il}}\Delta t\mathbf{L}_h(\mathbf{w}_h^{(l)}))$$
3. Set $\mathbf{w}_h^{n+1} = \mathbf{w}_h^{(k)}$ (2.22)

The initial condition for the timestepping, $\mathbf{w}_h^0$, at t=t_0, is the projection of the actual initial condition, $\mathbf{f}(\mathbf{x})$, onto the nodal values of the fields across the element.

2.6.1 Explicit RK-4 Method

The classic four stage explicit Runge-Kutta (RK-4) method is perhaps the most popular explicit method for numerically solving ODEs and is described in (2.23) below:

$$\mathbf{w}^{n+1} = \mathbf{w}^n + \frac{\Delta t}{6}(\mathbf{k}^1 + 2\mathbf{k}^2 + 2\mathbf{k}^3 + \mathbf{k}^4), \qquad t_{n+1} = t_n + \Delta t, \tag{2.23}$$

where $\mathbf{w}^{n+1}$ is an approximation to $\mathbf{w}(\mathrm{t}_{n+1})$ and the intermediate evaluations are

$$\mathbf{k}^1 = \mathbf{L}_h(\mathbf{w}^n), \qquad \mathbf{k}^2 = \mathbf{L}_h(\mathbf{w}^n + \frac{1}{2}\Delta t\mathbf{k}^1),$$
$$\mathbf{k}^3 = \mathbf{L}_h(\mathbf{w}^n + \frac{1}{2}\Delta t\mathbf{k}^2), \qquad \mathbf{k}^4 = \mathbf{L}_h(\mathbf{w}^n + \Delta t\mathbf{k}^3). \tag{2.24}$$

For stable timesteps, this method has total accumulated error of $O(\Delta t^4)$ [12, p.969].

2.6.2 Explicit RK-TVD Method

The RK-4 method, however, does not have a property of bounding the growth of spurious oscillations that can occur in the numerical solution during the timestepping for discontinuous problems. To overcome this, an alternative class of explicit timesteppers, known as the total variation diminishing explicit Runge-Kutta class (RK-TVD), is introduced.

An RK-TVD scheme restricts the general Runge-Kutta coefficients (2.22), to satisfy the three conditions [7]:

$$\beta_{il} \neq 0 \implies \alpha_{il} \neq 0$$
$$\alpha_{il} \geq 0 \tag{2.25}$$
$$\sum_{l=0}^{i-1} \alpha_{il} = 1$$

To discuss stability of an RK-TVD scheme, the concept of a semi-norm must be introduced. A semi-norm on a vector space W is a real valued function, $|.| : W \to \mathbb{R}$, such that $\forall \mathbf{a}, \mathbf{b} \in W$, the following conditions hold for any scalar s:

$$|\mathbf{a}| \geq 0$$
$$|s\mathbf{a}| = |s|.|\mathbf{a}|$$
$$|\mathbf{a} + \mathbf{b}| \leq |\mathbf{a}| + |\mathbf{b}| \tag{2.26}$$

Assume in some arbitrary semi-norm that the following bound can be achieved at each explicit Euler step of the general Runge Kutta scheme (2.22),

$$|\mathbf{v}_h^{il}| \leq |\mathbf{w}_h^{(l)}|, \tag{2.27}$$

then the following argument gives a bound on $\mathbf{w}_h^{(i)}$

$$\begin{aligned}
|\mathbf{w}_h^{(i)}| &= |\sum_{l=0}^{i-1} \alpha_{il} \mathbf{v}_h^{il}| \\
&\leq \sum_{l=0}^{i-1} |\alpha_{il} \mathbf{v}_h^{il}| \qquad \text{semi-norm triangle inequality (2.26)} \\
&\leq \sum_{l=0}^{i-1} \alpha_{il} |\mathbf{v}_h^{il}| \qquad \text{RK-TVD assumption 2 (2.25)} \\
&\leq \sum_{l=0}^{i-1} \alpha_{il} |\mathbf{w}_h^{(l)}| \qquad \text{semi-norm bound assumption (2.27)} \\
&\leq \max_{0 \leq l \leq i-1} |\mathbf{w}_h^{(l)}| \qquad \text{RK-TVD assumption 3 (2.25)} \qquad (2.28)
\end{aligned}$$

By induction, $|\mathbf{w}_h^n| \leq |\mathbf{w}_h^0| \ \forall n \geq 0$, and so the solution remains bounded at step n in this arbitrary semi-norm and the RK-TVD scheme is said to be stable.

If the total variation semi-norm is used (2.31) then the approximate solution is said to be total variation diminishing in the means and physically this condition means that spurious oscillations of the numerical solution are bounded at step n of the timestepping.

However, the underlying assumption that $|\mathbf{v}_h^{il}| \leq |\mathbf{w}_h^{(l)}|$ at each explicit Euler step is by no means naturally satisfied. This is enforced by slope limiting the solution after each intermediate Euler step, as discussed in section 2.7.

Gottlieb *et al.* [11] describe second and third order RK-TVD schemes, satisfying the conditions on the coefficients α_{il}, β_{il}:

RK-TVD2

$$\begin{aligned}
\mathbf{w}^{(1)} &= \mathbf{w}^n + \Delta t \mathbf{L}_h(\mathbf{w}^n), \\
\mathbf{w}^{n+1} &= \frac{1}{2}\mathbf{w}^n + \frac{1}{2}\mathbf{w}^{(1)} + \frac{1}{2}\Delta t \mathbf{L}_h(\mathbf{w}^{(1)}).
\end{aligned}$$

For stable timesteps, this method has total accumulated error of $O(\Delta t^2)$.

RK-TVD3

$$\begin{aligned}\mathbf{w}^{(1)} &= \mathbf{w}^n + \Delta t \mathbf{L}_h(\mathbf{w}^n),\\ \mathbf{w}^{(2)} &= \frac{3}{4}\mathbf{w}^n + \frac{1}{4}\mathbf{w}^{(1)} + \frac{1}{4}\Delta t \mathbf{L}_h(\mathbf{w}^{(1)}),\\ \mathbf{w}^{n+1} &= \frac{1}{3}\mathbf{w}^n + \frac{2}{3}\mathbf{w}^{(2)} + \frac{2}{3}\Delta t \mathbf{L}_h(\mathbf{w}^{(2)}).\end{aligned} \tag{2.29}$$

For stable timesteps, this method has total accumulated error of $O(\Delta t^3)$ and the implementation of this scheme into the `oomph-lib` is described in Appendix D.

2.7 1D Slope Limiting

2.7.1 Stability and sign conditions

In the previous section, it was stated that an RK-TVD method ensures the approximate solution remains bounded in an arbitrary semi-norm, $|\mathbf{w}_h^n| \leq |\mathbf{w}_h^0| \; \forall n \geq 0$, if the intermediate explicit Euler steps, $\mathbf{w}_h^{(l)} \to \mathbf{v}_h^{il}$, remain bounded at every intermediate step in this arbitrary semi-norm, $|\mathbf{v}_h^{il}| \leq |\mathbf{w}_h^{(l)}|$.

Consider an explicit Euler step, $w_h^n \to v_h^n$, in 1 spatial dimension. Cockburn *et al.* [7] show that if polynomials of degree 0 are used, the discontinuous Galerkin method results in a monotone scheme and stability is ensured in the total variation semi-norm,

$$|v_h^n|_{TV} \leq |w_h^n|_{TV}, \tag{2.30}$$

where the TV semi-norm is defined as,

$$|w_h|_{TV} = \sum_{e=1}^{N} |\bar{w}_{e+1} - \bar{w}_e|. \tag{2.31}$$

It is natural to ask for such a property for general approximation orders in one spatial dimension. Cockburn *et al.* [7, p.192] show that the intermediate explicit Euler steps are stable in the TV semi-norm, if the timestep is sufficiently small (see section 2.7.6) and the following sign conditions on the fields at the edge of an element can be achieved,

$$\begin{aligned}\text{sign}(w_{j+1,L} - w_{j,L}) &= \text{sign}(\bar{w}_{j+1} - \bar{w}_j),\\ \text{sign}(w_{j,R} - w_{j-1,R}) &= \text{sign}(\bar{w}_j - \bar{w}_{j-1}),\end{aligned} \tag{2.32}$$

where $w_{j,R/L}$ is the value of the field on the right or left edge of element j, and $\bar{w}_j$ is the average value of the field in element j. These sign conditions are enforced by applying the slope limiter, denoted $\Lambda\Pi_h$, after each explicit Euler timestep.

2.7.2 Minmod slope limiter

A slope limiter must be constructed such that conservation is not violated, the sign conditions given by (2.32) are obeyed and high order accuracy of the method remains in smooth regions of the solution (see section 2.7.4).

Consider, in one spatial dimension, piecewise linear approximation of the solution w in element j, where $\bar{w}_j$ is the field at the central coordinate of the element, $x_{j+1/2}$, and $(w_j)_x$ is the approximate solution gradient,

$$w_j = \bar{w}_j + (x - x_{j+1/2})(w_j)_x, \tag{2.33}$$

and define the minmod function,

$$m(a_1, a_2, a_3) = \begin{cases} s \min |a_i| & \text{if } s = \text{sign}(a_1) = \text{sign}(a_2) = \text{sign}(a_3) \\ 0 & \text{otherwise.} \end{cases} \tag{2.34}$$

In this report the MUSCL limiter, introduced by VanLeer [13], denoted $\Lambda\Pi_h^{MUSCL}$, will be used,

$$\Lambda\Pi_h^{MUSCL} w_j = \bar{w}_j + (x - x_{j+1/2}) m\Big((w_j)_x, \frac{\bar{w}_{j+1} - \bar{w}_j}{\Delta x_j}, \frac{\bar{w}_j - \bar{w}_{j-1}}{\Delta x_j}\Big), \tag{2.35}$$

where Δx_j is the width of element j. The limiter operates by using the three argument minmod function. The first argument is the gradient of the solution in element j. The other two arguments are an "average" gradient between element j-1 and j and an "average" gradient between element j and j+1. If any of the signs of these gradients differ, the minmod function returns 0, and a piecewise constant approximation is used, resulting in a local finite volume method. If the signs of the gradients are the same, a linear approximation is used in element j, with the smallest gradient in absolute value.

Cockburn *et al.* [7] show that if the MUSCL limiter is used, the sign conditions (2.32) are indeed met. The approximate solution is thus stable in the TV semi-norm at each explicit Euler step and the approximate solution is total variation diminishing in the means for an RK-TVD timestepper, from the argument in (2.28).

2.7.3 High Order Approximation

The MUSCL limiter thus operates on a solution that is piecewise linear and ensures stability in the TV semi-norm. However, it is a desirable property for limiters to be applicable to higher order polynomial approximation. Consider element j, with degree n polynomial approximation, $w_j^{(n)}$, and define the following boundary fields on the left and right of the element,

$$u_j(x_l) = \bar{w}_j - m(\bar{w}_j - w_j^{(n)}(x_l), \bar{w}_j - \bar{w}_{j-1}, \bar{w}_{j+1} - \bar{w}_j), \tag{2.36}$$

$$u_j(x_r) = \bar{w}_j + m(w_j^{(n)}(x_r) - \bar{w}_j, \bar{w}_j - \bar{w}_{j-1}, \bar{w}_{j+1} - \bar{w}_j). \tag{2.37}$$

Hesthaven *et. al* [3, p.152] use the following algorithm to perform slope limiting in element j, $\Pi_h : w_j^{(n)} \to v_j$, as is implemented in the `oomph-lib`:

- The edge values, $u_j(x_l)$ and $u_j(x_r)$ are calculated through (2.36) and (2.37).
- If $u_j(x_l) = w_j^{(n)}(x_l)$ and $u_j(x_r) = w_j^{(n)}(x_r)$, the solution requires no limiting and the high order accuracy is retained, $v_j = w_j^{(n)}$.
- Otherwise it is assumed a spurious oscillation has been detected. The numerical solution requires limiting, $v_j = \Lambda\Pi_h(w_j^1)$, using the MUSCL limiter. w_j^1 is the projection of the polynomial function $w_j^{(n)}$ onto a piecewise linear function, achieved by constructing a linear function, w_j^1, by constructing a gradient from the edge values, $w_j^{(n)}(x_r)$ and $w_j^{(n)}(x_l)$, of element j.

Looping over all the elements, and over each of the conserved fields, gives a slope limited solution after each explicit Euler timestep.

2.7.4 TVDM-property

As argued in (2.28), since the explicit Euler steps of the RK-TVD scheme are bounded in the TV semi-norm, the solution at step n also remains bounded in the TV semi-norm. Hesthaven *et al.* [3, p.160] summarise this important result in the theorem below:

Theorem 2.7.1. *If the limiter $\Lambda\Pi_h$ ensures the TVDM property,*

$$v_h = \Lambda\Pi_h(w_h) \implies |v_h|_{TV} \leq |w_h|_{TV},$$

then the DG method with the RK-TVD timestepper is TVDM,

$$|w_h^n|_{TV} \leq |w_h^0|_{TV} \quad \forall n. \tag{2.38}$$

2.7.5 TVBM-property

Although the arguments above imply that the approximate solution diminishes in the TV semi-norm, the slope limiter still has one main failing point. Consider a solution with a smooth maximum or minimum. The solution gradient in elements near this region will change sign and the minmod slope limiter will return 0, resulting in piecewise constant approximation. Thus, high order accuracy is lost near regions containing smooth local extrema. One way to recover the high order accuracy is to alter the minmod function.

Shu [14] considered a modified minmod function below, where the parameter M is an approximation to the second derivative near the smooth extrema,

$$\bar{m}(a_1, a_2, a_3) = \begin{cases} a_1 & \text{if } |a_1| \leq Mh_j^2 \\ m(a_1, a_2, a_3) & \text{otherwise.} \end{cases} \tag{2.39}$$

This modified minmod function has the effect of retaining the high order accuracy near regions with smooth extrema. The slope limiter is changed by replacing the occurrence of the minmod function in (2.35), by the modified minmod function (2.39). *A-priori*, determining the value of M is difficult, since this would assume prior knowledge of smooth extrema, however, for now, this parameter is incorporated into the slope limiters via an extra label M: $\Lambda\Pi_h \to \Lambda\Pi_{h,M}$.

The modified MUSCL limiter, however, does not now ensure the TVDM property from theorem 2.7.1 between successive Euler steps. A weaker property of total variation bounded in the means (TVBM), can be found. If we consider an explicit Euler step $w_h \to v_h$ in the RK-TVD scheme (2.22), and $v_h = \Lambda\Pi_{h,M} w_h$, then Cockburn *et al.* [7, p.197] show that,

$$|v_h|_{TV} \leq |w_h|_{TV} + CM\Delta x, \tag{2.40}$$

where C is a constant dependent only on the approximation order and M is the modified minmod function constant. Through a similar argument to (2.28), the TVBM property ensures that the approximate solution remains bounded after T timesteps in the TV semi-norm and again spurious oscillations of the solution are bounded. Hesthaven *et al.* [3, p.160] summarise this as a new theorem for the RK-TVD DG method:

Theorem 2.7.2. *If the limiter $\Lambda\Pi_{h,M}$ ensures the TVBM property,*

$$v_h = \Lambda\Pi_{h,M}(w_h) \implies |v_h|_{TV} \leq |w_h|_{TV} + CM\Delta x,$$

then the DG method with the RK-TVD timestepper is TVBM:

$$|w_h^n|_{TV} \leq |w_h^0|_{TV} + CMQ \quad n = 1 : T, \tag{2.41}$$

where $T\Delta x \leq Q$.

The minmod slope limiters are powerful objects, as they not only detect and perform limiting where required in an element, but also retain high order accuracy in regions of smooth extrema. When the slope limiter is used in conjunction with an RK-TVD timestepper, spurious oscillations that can occur in the numerical solution of a discontinuous problem are bounded.

2.7.6 CFL timestep condition

An extra condition on the size of the timestep must also be satisfied, a Courant-Friedrichs-Levy (CFL) condition, to ensure the explicit Euler timesteps have the TVDM or TVBM property,

$$|c|\frac{\Delta t}{\Delta x} \leq CFL,$$

where $|c|$ is the largest wave speed, Δx is the smallest element width, and Δt is a stable timestep. Physically this condition bounds the size of the timestep to ensure the physical features of the solution are resolved over the mesh. Cockburn *et al.* [7] argue in practice that the CFL number is given by

$$CFL = \frac{1}{2k+1},$$

where k is the degree of the approximating polynomial. Unless otherwise stated, any computational results presented will have timesteps chosen small enough to satisfy the CFL condition. The following algorithm describes the complete DG RK-TVD method that will be used for one dimensional problems in this report:

1. Set $\mathbf{w}_h^{(0)} = \Lambda\Pi_{h,M}\mathbf{w}_h^0$
2. For $n = 0 : T-1$ - Perform T timesteps
   (a) Set $\mathbf{w}_h^n = \mathbf{w}_h^{(0)}$
   (b) For $i = 1 : k$ - Perform explicit TVBM steps
   $$\mathbf{w}_h^{(i)} = \Lambda\Pi_{h,M}\Big(\sum_{l=0}^{i-1}\alpha_{il}\mathbf{v}_h^{il}\Big) \text{ where } \mathbf{v}_h^{il} = \mathbf{w}_h^{(l)} + \frac{\beta_{il}}{\alpha_{il}}\Delta t\mathbf{L}_h(\mathbf{w}_h^{(l)})$$
   (c) Set $\mathbf{w}_h^{n+1} = \mathbf{w}_h^{(k)}$ (2.42)

A similar algorithm will be used for two dimensional problems but with an alternative slope limiter applied at each explicit Euler step. This will be discussed in more detail in section 5.2.1.

2.8 Norms and convergence

2.8.1 Broken L^2 and L^1 norms

In this report, problems are considered on a computational domain $\Omega_h \in \mathbb{R}^n$ that is a good approximation to the physical domain $\Omega \in \mathbb{R}^n$. The domain, Ω_h, is broken into K geometry conforming elements, D^m. In each of these elements, the approximation is piecewise continuous, and norms can be defined in each of these. Summing contributions over all the elements defines broken norms over the whole computational domain, Ω_h, whereas global norms are considered over the physical domain Ω. Consider a conserved field, $\mathbf{w}(\mathbf{x}, t)$, then global and broken norms can be defined on the individual components of the conserved field, w_i. The global and broken L^2-norms are defined in (2.43) and (2.44) respectively,

$$||w_i||^2_{\Omega,L^2} = \int_\Omega w_i^2 \mathbf{dx}, \tag{2.43}$$

$$||w_i||^2_{\Omega,h,L^2} = \sum_{m=1}^{K} ||w_i||^2_{D^m,L^2} \quad \text{where} \quad ||w_i||^2_{D^m,L^2} = \int_{D^m} w_i^2 d\mathbf{x}, \tag{2.44}$$

and the space of functions, $w_i \in L^2(\Omega)$, is defined for which $||w_i||_{\Omega,L^2}$ and $||w_i||_{\Omega,h,L^2}$ are bounded. The global and broken L^1-norms are defined in (2.45) and (2.46) respectively:

$$||w_i||_{\Omega,L^1} = \int_\Omega |w_i| \mathbf{dx}, \tag{2.45}$$

$$||w_i||_{\Omega,h,L^1} = \sum_{m=1}^{K} ||w_i||_{D^m,L^1} \quad \text{where} \quad ||w_i||_{D^m,L^1} = \int_{D^m} |w_i| d\mathbf{x}. \tag{2.46}$$

The literature on the DGFEM appears to suggests that concrete *a-priori* error analysis for problems with genuine discontinuities is unknown, but some qualitative analysis is possible. Assuming a consistent conservative numerical flux, such as the Lax-Friedrichs flux, and a limiter ensuring the TVBM property of the solution, it seems reasonable to expect that performing h-refinement (increasing the number of elements), should result in a convergent numerical solution, as any discontinuity present in the exact solution will be resolved in fewer elements. Errors will be measured in the broken L^2 and L^1 norms when performing h-refinement.

In a conventional finite element method with smooth solutions, p-refinement, where the number of elements is fixed and the degree of the approximating polynomial space is increased, is generally known to provide convergence [4, 15]. However, for discontinuous solutions, errors are created when approximating a discontinuous function by members of a continuous function space, the Gibbs phenomenon. These errors manifest themselves as oscillations near the discontinuity and the oscillations are not removed by increasing the order of the polynomial approximation. Removing these oscillations numerically, through a slope limiter, has made convergence results very difficult to construct for p-refinement in the DGFEM.

2.8.2 Linear Problems

Although the convergence of the DGFEM is an open problem for genuine discontinuous problems with non linear fluxes, error bounds are possible for problems involving a linear flux, $f(u) = cu$. Consider a problem, in one spatial dimension, with a linear flux and a smooth initial condition at t=0. If an orthogonal polynomial basis is used, the initial condition, w_0, is smooth, timestepping errors are negligible, and an upwind

numerical flux is used, then Hesthaven *et. al* [3, p.88] show that at time T,

$$||w(T) - w_h(T)||_{\Omega,h,L^2} \leq C(N_p)h^{N_p}(1 + C_1(N_p)T),$$

where h is the smallest element width, N_p is the number of nodes per element and C, C_1 are constants independent of h. Although linear fluxes have limited application, the error bound does provide useful validation to code that has been written.

Chapter 3

Continuous Problems in 1D

3.1 Flux transport equation

Consider the 1D flux transport equation (3.1), consisting of a single field $w(x,t) \in \mathbb{R}$, with a scalar flux $f(w(x,t)) \in \mathbb{R}$:

$$\frac{\partial w}{\partial t} + \frac{\partial f(w)}{\partial x} = 0 \quad x \in [A,B]. \tag{3.1}$$

For a sufficiently smooth flux it is possible to write this in the form (3.2), the 1D kinematic wave equation, where $c(w) = \frac{df(w)}{dw}$:

$$\frac{\partial w}{\partial t} + c(w)\frac{\partial w}{\partial x} = 0 \quad x \in [A,B]. \tag{3.2}$$

To give insight into the behaviour of solutions to the 1D kinematic wave equation, assume an initial distribution $w(x,0) = w_0(x)$. A characteristic curve of the differential equation is defined as

$$\frac{dx}{dt} = c(w), \tag{3.3}$$

and along these characteristics curves, the 1D advection equation becomes:

$$\frac{\partial w}{\partial t} + c(w)\frac{\partial w}{\partial x} = \frac{\partial w}{\partial t} + \frac{dx}{dt}\frac{\partial w}{\partial x} = \frac{dw}{dt} = 0.$$

So w is a constant on the characteristics curves defined by (3.3) and these curves represent straight lines in spacetime (x,t). Consider a point (x,t) in spacetime, and project back along the characteristic to a point $(\eta,0)$. The equation of the

characteristic, from (3.3), is

$$x - \eta = c(w_0(\eta))t, \tag{3.4}$$

and since w is a constant on the characteristic, the following is true:

$$w(x,t) = w(\eta,0) = w_0(\eta). \tag{3.5}$$

If a solution exists to the 1D kinematic wave equation, then it is given by (3.5), where η is found implicitly through (3.4). Logan [1, p.70] writes this result as a uniqueness theorem:

Theorem 3.1.1. *If c, $w_0 \in C^1(\mathbb{R})$ and are both either non-decreasing or non-increasing on $\mathbb{R}$, then a unique solution for the initial value problem* (3.2) *exists for all $t > 0$, given implicitly through* (3.4) *and* (3.5).

3.2 Advection equation

If the function $c(w)$ is taken to be constant, $c(w) = C > 0$, this is known as the 1D advection equation. From the characteristic equation (3.4), $x - \eta = Ct$, and η can be directly computed giving the solution at time t as

$$w(x,t) = w_0(x - Ct).$$

The effect of 1D advection is thus to move an initial distribution, with speed C, in the positive x direction. To solve the advection equation numerically, the Lax-Friedrichs flux was chosen as the numerical flux. The Jacobian of the flux for a single field in one dimension is a scalar, and specifically for 1D advection

$$\frac{df}{du} = \frac{d(Cu)}{du} = C. \tag{3.6}$$

Using the definition of the Lax-Friedrichs flux (2.7), where a and b are the internal and external fields at an element edge

$$\begin{aligned} f^*(a,b) &= \frac{1}{2}(f(a) + f(b)) - \frac{C_{LF}}{2}(b-a) \\ &= \frac{1}{2}(Ca + Cb) - \frac{C}{2}(b-a) = Ca. \end{aligned} \tag{3.7}$$

Thus, for 1D advection, the Lax-Friedrichs flux is an upwind flux that passes information solely from the direction in which it is coming. Setting $C = 1$, the domain $x \in [0, 1]$ and a sinusoidal initial distribution, an exact solution can be found through the method of characteristics,

$$w(x, 0) = \sin(2\pi x) \quad \Longrightarrow \quad w(x, t) = \sin(2\pi(x - t)).$$

To test the validity of the DGFEM code, the initial distribution above was implemented, along with periodic boundary conditions, $w(0, t) = w(1, t)$. A grid of N equally spaced elements was set up, with linear approximation, and figure 3.1 illustrates the computed and exact solution at $T = 0.4$. Visually, it appears that with only 40 linear elements, the computed solution is a good approximation to the exact solution.

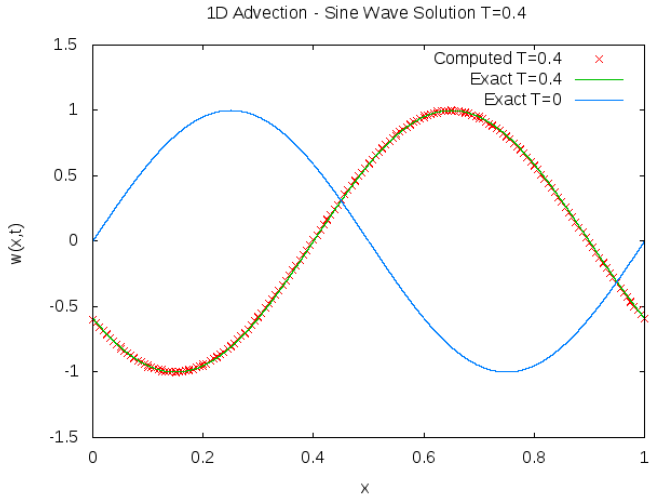

Figure 3.1: *1D advection - Computed and exact solution at T=0.4s with $N = 40$ linear elements. An RK-4 timestep of $dt = 10^{-3}$ with a Lax-Friedrichs flux was used. The initial sine wave distribution is also illustrated.*

It was discussed in section 2.8.2 for the 1D advection equation, solved numerically with an upwind flux, that the error measured in the L^2-norm should be of the form,

$$||u(T) - u_h(T)|| \leq C(T)h^{N_p} \approx (C_1 + C_2 T)h^{N_p}. \tag{3.8}$$

To test the validity of the code, the broken L^2-error (2.44) of the difference between the numerical and exact solution across the domain was computed, using the N_p GLL quadrature rule (2.19) over each element. Table 3.1 displays the broken L^2-errors,

N	$d=1$	$d=2$	$d=3$	$d=4$
20	3.24×10^{-2}	4.16×10^{-4}	6.98×10^{-6}	9.90×10^{-8}
40	8.40×10^{-3}	5.21×10^{-5}	4.37×10^{-7}	3.15×10^{-9}
80	2.12×10^{-3}	6.51×10^{-6}	2.73×10^{-8}	9.85×10^{-11}
160	5.31×10^{-4}	8.14×10^{-7}	1.71×10^{-9}	3.08×10^{-12}
Rate	1.98	3.00	4.00	4.99

Table 3.1: *1D Advection equation - Broken L^2-errors at $T=0.4$ for a sinusoidal initial condition, with an RK-4 timestep of $dt=10^{-4}$ and a Lax-Friedrichs (upwind) flux. The data represents different numbers of elements, N, and different polynomial orders, $d=N_p-1$. The columns and rows represent h- and p-refinement respectively.*

at $T=0.4$, as a function of the number of elements, N, and polynomial degree, $d=N_p-1$, in one spatial dimension.

Table 3.1 demonstrates that greater accuracy can be achieved by either increasing the polynomial order or increasing the number of elements in the domain. The convergence rates were estimated by fixing the polynomial degree, and measuring the broken L^2-error as a function of the number of elements. Assuming a relationship of the form

$$||w(T)-w_h(T)||_{\Omega,h,L^2}=C(T)h^{Rate},$$

the convergence rate is estimated as the best fit gradient of $\ln||w(T)-w_h(T)||_{\Omega,h,L^2}$ against $\ln N$ for each polynomial degree, as demonstrated in figure 3.2. The graphs indicate $rate\approx N_p$, in agreement with the estimate (3.8).

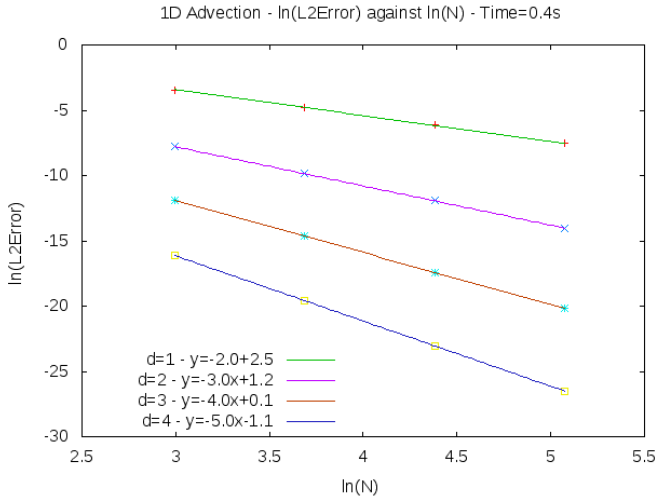

Figure 3.2: *1D advection - Broken L^2-errors at $T=0.4$ for a sinusoidal initial condition, RK-4 timestep of $dt=10^{-4}$ and Lax-Friedrichs flux. The individual lines represent h-refinement for a fixed polynomial degree, $d=N_p-1$. The convergence rate $\approx N_p=d+1$.*

3.3 1D shallow water equations

The shallow water equations in one spatial dimension are a special case of the two dimensional equations derived in section 5.1. No variation is assumed in the y-direction, so that the partial derivative, $\frac{\partial}{\partial y}$, and the y-velocity, $v(x,t)$, can be set to 0. For a domain $x \in [A,B]$, the governing equations are (3.9), where $u(x,t)$ and $h(x,t)$ are the water velocity and height respectively and G is the constant of gravity,

$$\frac{\partial h}{\partial t} + \frac{\partial hu}{\partial x} = 0, \qquad \frac{\partial hu}{\partial t} + \frac{1}{2}\frac{\partial Gh^2}{\partial x} + \frac{\partial u^2 h}{\partial x} = 0. \tag{3.9}$$

The system of PDEs can be written in conservative form (2.1), with the corresponding field and flux,

$$\mathbf{w} = \begin{pmatrix} h \\ hu \end{pmatrix}, \quad \mathbf{f}(\mathbf{w}) = \begin{pmatrix} hu \\ \frac{1}{2}Gh^2 + u^2 h \end{pmatrix} = \begin{pmatrix} hu \\ \frac{1}{2}Gh^2 + \frac{(hu)^2}{h} \end{pmatrix}. \tag{3.10}$$

Consider the equation for conserved field hu in (3.9), and using the product rule

$$h\frac{\partial u}{\partial t} + u\frac{\partial h}{\partial t} + h\frac{\partial Gh}{\partial x} + hu\frac{\partial u}{\partial x} + u\frac{\partial uh}{\partial x} = 0,$$

and substituting the equation for the conserved field h from (3.9) into the above

$$h\frac{\partial u}{\partial t} - u\frac{\partial hu}{\partial x} + h\frac{\partial Gh}{\partial x} + hu\frac{\partial u}{\partial x} + u\frac{\partial uh}{\partial x} = 0.$$

Cancelling terms, and absorbing the velocity term into the x partial derivative,

$$h\frac{\partial u}{\partial t} + h\frac{\partial Gh}{\partial x} + \frac{1}{2}h\frac{\partial u^2}{\partial x} = 0.$$

Assuming $h \neq 0$, we can divide by h above, yields an alternative form for the shallow water equations,

$$\frac{\partial h}{\partial t} + \frac{\partial hu}{\partial x} = 0, \qquad \frac{\partial u}{\partial t} + \frac{\partial Gh}{\partial x} + u\frac{\partial u}{\partial x} = 0,$$

written in conservative form, with the corresponding field and flux below,

$$\mathbf{w} = \begin{pmatrix} h \\ u \end{pmatrix}, \quad \mathbf{f}(\mathbf{w}) = \begin{pmatrix} hu \\ Gh + \frac{1}{2}u^2 \end{pmatrix}. \tag{3.11}$$

Although this is seemingly just an algebraic trick, the two different forms will become relevant when analysing a discontinuous problem in section 4.2.4.

3.3.1 Numerical Solution

The conservative forms (3.10) and (3.11) of the equations will be used to solve a continuous problem of the shallow water equations with the Lax-Friedrichs flux prescribed at element edges (2.7).

The definition of the Lax-Friedrichs flux constant is the largest magnitude eigenvalue of the Jacobian of the flux vector over all possibilities of the field across the boundary. To save solving an optimization problem at each element edge, the maximum is taken only over the internal or external field, $\mathbf{w}_i$ or $\mathbf{w}_e$. Unless otherwise stated, this assumption will be used for all future numerical calculations of the Lax-Friedrichs constant.

For the conservative form (3.10), the Jacobian, corresponding eigenvalues and Lax-Friedrichs constant are calculated as

$$\frac{\partial \mathbf{f}}{\partial \mathbf{w}} = \begin{pmatrix} 0 & 1 \\ -u^2 + Gh & 2u \end{pmatrix}, \quad \lambda = u \pm \sqrt{Gh}, \quad C_{LF} = \max_{\mathbf{w}_i, \mathbf{w}_e} |u \pm \sqrt{Gh}|, \tag{3.12}$$

and for the conservative form (3.11), the Jacobian, the corresponding eigenvalues and the Lax-Friedrichs constant are

$$\frac{\partial \mathbf{f}}{\partial \mathbf{w}} = \begin{pmatrix} u & h \\ G & u \end{pmatrix}, \quad \lambda = u \pm \sqrt{Gh}, \quad C_{LF} = \max_{\mathbf{w}_i, \mathbf{w}_e} |u \pm \sqrt{Gh}|. \tag{3.13}$$

Thus the optimization problem to find the Lax-Friedrichs constant is identical for both conserved forms of the shallow water equations. The algorithm to compute the Lax-Friedrichs constant at an element edge can be seen below and further details of the implementation can be seen in Appendix B.

```
1 Find w_i and w_e
2 Compute C_i^+ = u_i + sqrt(Gh_i), C_i^- = u_i - sqrt(Gh_i), C_e^+ = u_e + sqrt(Gh_e), C_e^- = u_e - sqrt(Gh_e)
3 Take C_LF = max(C_i^+, C_i^-, C_e^+, C_e^-)
```

A continuous exact solution (3.14) is known to the shallow water equations [3, p.165], where H represents the steady state water height, that can be validated by direct substitution into the governing PDEs (3.9):

$$h(x,t) = \epsilon(x,t)^2, \quad u(x,t) = 2\sqrt{G}(\epsilon(x,t) - \sqrt{H}), \quad \epsilon(x,t) = \frac{x + 2\sqrt{GH}t}{1 + 3\sqrt{G}t}. \tag{3.14}$$

The shallow water equations were modelled numerically by setting the domain to $x \in [0, 1]$, and parameters to $G = 10$, $H = 1$. The initial height and velocity distribution were set at $T = 0.1$ in (3.14), and Dirichlet boundary conditions used at the domain edge. These boundary conditions are set, after each explicit timestep, by setting the pointer to the neighbouring face of a boundary face as itself. This overloads the numerical flux at the boundary by the exact flux, as described in section 2.3.1.

A complete, original C++ code to solve this problem, using linear approximation for the equations in conservative form (3.11), is described in Appendix E. Figure 3.3 illustrates the exact and computed solution for the height and velocity at $T = 0.5$ for both the conservative form (3.10) and (3.11). The figure appears to show the numerical solutions for both conservative forms are, within errors from the numerical integration schemes, equivalent.

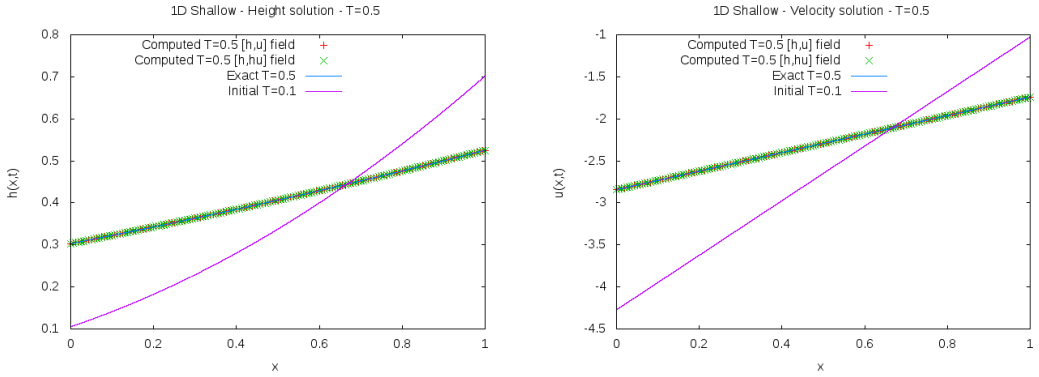

Figure 3.3: *1D shallow water - Continuous problem height and velocity solution at $T = 0.5$ with $N = 40$ linear elements, the Lax-Friedrichs flux and an RK-4 timestep of $dt = 10^{-4}$. The figure shows the solution with both the $[h, u]^T$ and $[h, hu]^T$ form of the equations.*

Using the conservative form of the equations (3.11), errors were measured in the broken L^2-norm, for each of the field components $h_h(x, t)$ and $u_h(x, t)$. A single measure for the broken L^2-error of the field $\mathbf{w} = [h, u]^T$ is obtained by summing the squares of the L^2-errors of the components

$$||\mathbf{w} - \mathbf{w}_h||^2_{\Omega,h,L^2} = \sum_{i=1}^{m} ||w_i - w_{i,h}||^2_{\Omega,h,L^2}, \tag{3.15}$$

where the index m represents the number of fields. Table 3.2 illustrates the L^2-errors as a function of the number of elements, N, and approximating polynomial degree, $d = N_p - 1$. Convergence rates were estimated by fixing the polynomial degree and finding best fit gradients of $\ln ||\mathbf{w} - \mathbf{w}_h||_{\Omega,h,L^2}$ against $\ln N$. The rates for linear and

N	$d=1$	$d=2$	$d=3$
10	2.23×10^{-3}	9.70×10^{-4}	4.91×10^{-14}
20	5.24×10^{-4}	2.18×10^{-4}	4.93×10^{-14}
40	8.97×10^{-5}	4.62×10^{-5}	4.86×10^{-14}
80	1.88×10^{-5}	9.69×10^{-6}	3.25×10^{-13}
Rate	2.33	2.22	N/A

Table 3.2: *1D shallow water - Broken L^2-errors for continuous problem at $T = 0.5$, with an RK-4 timestep of $dt = 10^{-4}$ and a Lax-Friedrichs flux. The conserved field $[h, u]^T$ is used and the errors are shown as a function of the number of elements, N, and the polynomial order, $d = N_p - 1$. The columns and rows represent h- and p-refinement respectively.*

quadratic approximation were found to be 2.33 and 2.22 respectively, so increasing the polynomial order did not increase the convergence rate. As discussed in section 2.8.1, error analysis for non-linear problems is difficult to achieve, but the solutions are converging when performing h-refinement.

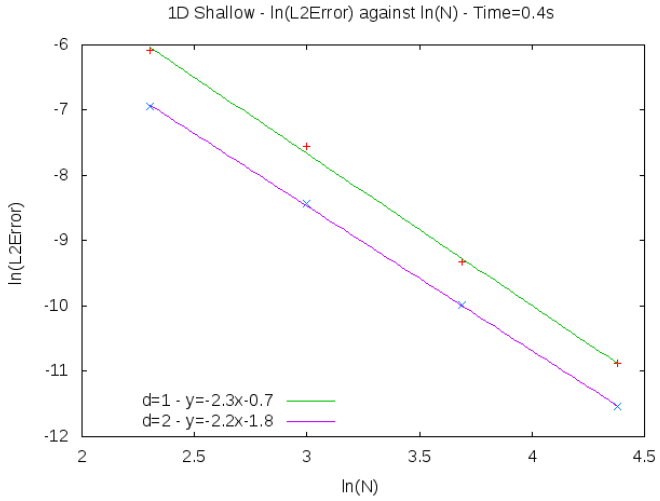

Figure 3.4: *1D shallow water - Continuous problem broken L^2-errors at $T = 0.5$, with an RK-4 timestep of $dt = 10^{-4}$ and a Lax-Friedrichs flux with the conserved field $[h, u]^T$. The lines represent h-refinement for fixed polynomial degrees 1 and 2 respectively.*

Using a cubic approximation, the errors have dramatically decreased to $O(10^{-13})$. This is explained by considering the numerical flux matrix (2.17) and flux matrix (2.16), evaluated at each explicit Euler timestep, in turn:

- Numerical Flux Matrix - For a continuous problem the field jumps between boundaries are 0 and so the coefficients of the numerical flux matrix are an evaluation of the flux at an element's edge. The exact height and velocity are quadratic and linear, so at cubic approximation the flux is evaluated exactly

and hence the numerical flux matrix coefficients are evaluated exactly.

- Flux Matrix - The flux matrix has a highest order term, $uh\frac{d\psi}{dx}$, that must be integrated across each element. At cubic approximation, $\frac{\partial\psi}{\partial x}$ is a degree 2 polynomial and the exact solutions of u and h are degree 1 and 2 polynomials. Thus the highest order flux term represents a polynomial of degree 5. At cubic approximation, the GLL quadrature scheme has $N_k = 4$ knot points and integrates degree $2N_k - 3 = 5$ polynomials exactly.

The flux and numerical flux matrices are calculated exactly at each timestep, and the errors of magnitude 10^{-13} are due to timestepping errors of the RK-4 scheme and the finite machine precision.

It was also worth noting how a different flux affects the accuracy of the numerical solution. Figure 3.5 displays the numerical solutions at $T = 0.4$ with linear approximation obtained with the central (2.6) and Lax-Friedrichs (2.7) flux. At linear approximation, the central flux is clearly inferior to the Lax-Friedrichs flux, with large discontinuities in the approximate solution present.

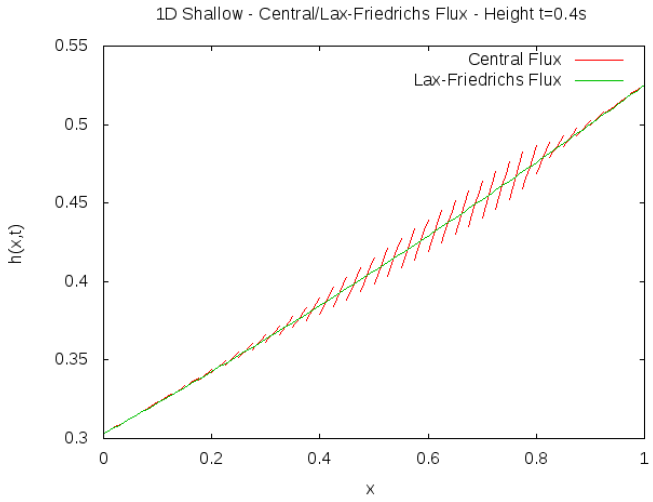

Figure 3.5: *1D shallow water - Central and Lax-Friedrichs flux comparison at $T = 0.4s$ with an RK-4 timestep of $dt = 10^{-4}$ and 40 linear elements.*

Chapter 4

Discontinuous Problems in 1D

4.1 Inviscid Burgers' equation

The 1D inviscid Burgers' equation is an example of a 1D flux transport equation, with non-linear flux $f(w) = \frac{1}{2}w^2$, leading to the PDE

$$\frac{\partial w}{\partial t} + \frac{1}{2}\frac{\partial w^2}{\partial x} = \frac{\partial w}{\partial t} + w\frac{\partial w}{\partial x} = 0 \qquad x \in [A, B]. \tag{4.1}$$

The PDE is useful to understand some of the basic theory behind shocks. In particular, *shock formation*, where a solution becomes multivalued at a single spatial point, from continuous initial conditions, and *shock propagation*, where a discontinuous initial condition propagates through time.

4.1.1 Shock Formation

We revisit the the 1D kinematic wave equation

$$\frac{\partial w}{\partial t} + c(w)\frac{\partial w}{\partial x} = 0 \quad x \in [A, B]. \tag{4.2}$$

As described in section 3.1, exact solutions can be found using the method of characteristics. The solution at a point (x, t) is given by $w(x, t) = w(\eta, 0)$ where η is found implicitly through $x - \eta = c(w_0(\eta))t$. However, from theorem 3.1.1, for some initial conditions unique solutions to the kinematic wave equation can not be guaranteed for all time and a shock can develop in finite time.

Consider the kinematic wave equation (4.2), and find the total time derivative of $w_x(x(t), t)$, the gradient of w along the characteristic $x = x(t)$. Using the chain rule

and $\frac{dx}{dt} = c(w)$ along a characteristic (3.3),

$$\frac{d}{dt}w_x(x(t),t) = w_{tx} + \frac{dx}{dt}w_{xx} = w_{tx} + c(w)w_{xx}.$$

Take the partial derivative with respect to x of the kinematic wave equation (4.2)

$$w_{tx} + c(w)w_{xx} + \frac{dc(w)}{dw}w_x^2 = 0,$$

and combining the above equations, leads to a first order differential equation,

$$\frac{d}{dt}w_x = -c'(w)(w_x)^2.$$

Using the boundary condition $x = \eta$ at $t = 0$, and that w is constant on a characteristic, the solution is

$$w_x = \frac{w_0'(\eta)}{1 + w_0'(\eta)c'(w_0(\eta))t}. \tag{4.3}$$

For the inviscid Burgers' equation (4.1), since $c(w) = w$ is a strictly increasing function, then if the initial condition is not, the denominator of (4.3) will be equal to zero in a finite time, the *breaking time*. The breaking point of the wave will thus be on the characteristic in which the denominator first vanishes. Let

$$G(\eta) = c(w_0(\eta)) \quad \Longrightarrow \quad G'(\eta) = w_0'(\eta)c'(w_0(\eta)), \tag{4.4}$$

the breaking point will be on the characteristic $\eta = \eta_B$, with the conditions that (i) $G'(\eta_B) < 0$ and (ii) $|G'(\eta_B)|$ is a maximum, with the time

$$T_B = -\frac{1}{G'(\eta_B)}. \tag{4.5}$$

As an example, consider the 1D inviscid Burgers' equation with initial condition (4.6), for $x \in [0, 2\pi]$, with Dirichlet boundary conditions, $w(0,t) = w(2\pi,t) = 0$,

$$w(x,0) = \sin(x). \tag{4.6}$$

For this initial condition, $G(\eta) = \sin(\eta)$, $G'(\eta) = \cos(\eta)$ and so $|G'(\eta)|$ is a maximum at $\eta = 0, \pi, 2\pi$. Using the further condition that $G'(\eta_B) < 0$, then the breaking point is given by $\eta_B = \pi$ and using (4.5), the breaking time is given by $T_B = 1$.

To model the inviscid Burgers' equation numerically, the Lax-Friedrichs flux was used. The Jacobian of the flux is w, and from the definition of the Lax-Friedrichs flux (2.7)

$$C_{LF} = \max_{w_i, w_e} w, \tag{4.7}$$

where w_i and w_e are the internal and external fields at an element boundary. The algorithm to compute the Lax-Friedrichs constant at an element edge can be seen below and further details of the implementation can be seen in Appendix B.

```
1 Find w_i and w_e
2 Compute C_i = w_i , C_e = w_e
3 Take C_LF = max(C_i, C_e)
```

Figure 4.1 represents the evolution of the numerical solution from $T = 0 - 1.5$, at regular intervals of 0.5, with the sine wave initial condition and fixed boundary conditions. As $t \to 1$, the gradient of the solution at $x = \pi$ becomes unbounded,

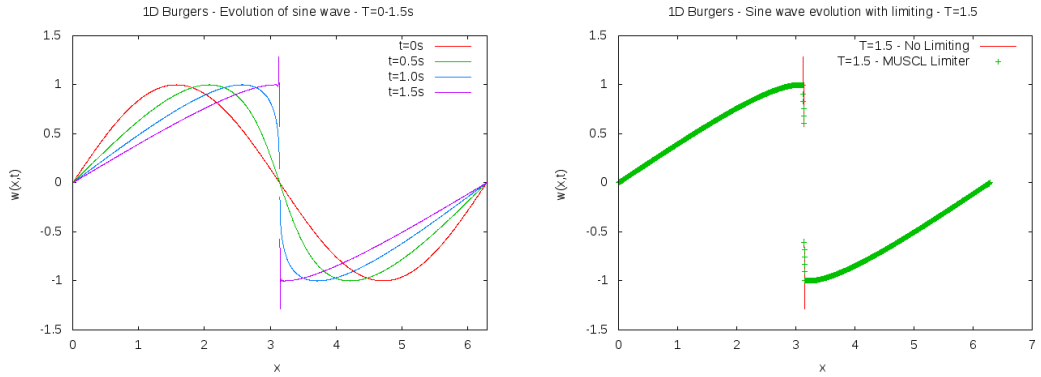

Figure 4.1: *Burgers' equation - Shock formation from smooth sinusoidal initial condition. An RK-4 timestep of* $dt = 5 \times 10^{-4}$ *is used with* $N = 400$ *linear elements and a Lax-Friedrichs flux. The left graphic illustrates the solution at regular time intervals of* 0.5 *and the right graphic indicates that oscillations are removed with the MUSCL limiter.*

and a shock develops. The approximation space, which is the space of linear functions, is not adequate to deal with this discontinuity in the solution, resulting in the oscillations of the solution near $x = \pi$ at $T = 1.5$. In section 2.7, slope limiters were constructed to remove such oscillations from a numerical solution. The MUSCL slope limiter (2.35) was implemented, using the modified minmod function (2.39) with smoothness parameter set to $M = 5$. The right hand graphic of figure 4.1 shows the computed solution at $T = 1.5$ and the oscillatory behaviour has been removed.

4.1.2 Shock Propagation

As mentioned previously, another effect common in conservation laws is the effect of shock propagation, in which a discontinuous initial condition propagates through time. To understand this, we re-examine the integral form of a scalar conservation law for a field, w, and flux, f, in one dimension (1.1),

$$\frac{d}{dt}\int_L^R w(x,t)\,dx = f(L,t) - f(R,t). \tag{4.8}$$

It was shown in Section 1.1 that if w and f are continuously differentiable the familiar PDE form for the conservation law can be recovered:

$$\frac{\partial w}{\partial t} + \frac{\partial f}{\partial x} = 0.$$

However, for a discontinuous initial condition, w_0, the assumption that w is continuously differentiable is not true, and the method of characteristics to determine solutions can not be directly used. Assume there is a smooth curve, $x = s(t)$, in which w has a *simple* discontinuity, so that w, and its derivatives, have well defined limits as $x \to s(t)^-$ and $x \to s(t)^+$ [1]. Consider the integral form (4.8)

$$\frac{d}{dt}\int_L^{s(t)} w(x,t)\,dx + \frac{d}{dt}\int_{s(t)}^R w(x,t)\,dx = f(L,t) - f(R,t),$$

with $L < s(t)$ and $R > s(t)$. Applying Leibniz' rule [12, p.125], to evaluate the derivative of an integral whose integrand and limits depend on the differentiation parameter, on the first two terms

$$\int_L^{s(t)} \frac{\partial}{\partial t} w(x,t)\,dx + \int_{s(t)}^R \frac{\partial}{\partial t} w(x,t)\,dx + w^-(t)s' - w^+(t)s' = f(L,t) - f(R,t), \tag{4.9}$$

where the following notation change has been used,

$$w^{\pm}(t) = \lim_{x \to s(t)^{\pm}} w(x,t) \quad \text{and} \quad s' = \frac{ds}{dt}.$$

Taking the limits $L \to s(t)^-$, $R \to s(t)^+$, the integral terms disappear, since w and its derivatives are continuous away from the shock and (4.9) simplifies to the Rankine Hugoniot condition

$$-s'[w] + [f(w)] = 0, \tag{4.10}$$

where the curve $s(t)$ is the shock path, and the propagating discontinuity in w is the shock wave. The square bracket notation is the difference in the field across the discontinuity, $[w] = w^+(t) - w^-(t)$. We write this shock condition as the correspondence

$$(...)_t \leftrightarrow -s'[(...)], \qquad (...)_x \leftrightarrow [(...)],$$

between a conservative PDE and its shock condition. Thus, for simple shocks, the Rankine Hugoniot conditions determine not only the shock velocity, through (4.10), but also the shock path, since $s = x(t)$.

Consider the 1D inviscid Burgers' equation, the shock velocity is given by

$$s' = \frac{1}{2}\frac{[w^2]}{[w]} = \frac{(w^+(t)^2 - w^-(t)^2)}{2(w^+(t) - w^-(t))} = \frac{w^+(t) + w^-(t)}{2}, \tag{4.11}$$

the average value of w before and after the shock. For a general initial condition this can be difficult to determine analytically, but consider the discontinuous initial condition below, for $x \in [-1, 1] = \Omega$, and Dirichlet boundary conditions $w(-1, t) = 3$ and $w(1, t) = 1$:

$$w(x, 0) = \begin{cases} 3 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \end{cases} \tag{4.12}$$

The initial condition (4.12) is piecewise constant and from (4.11) the shock speed is

$$s' = \frac{3 + 1}{2} = 2. \tag{4.13}$$

The shock moves with speed 2 in the positive x-direction. The characteristics are straight lines either side of the shock, because the initial condition is piecewise constant, and so the exact solution, for $0 < t < 0.5$, is

$$w(x, t) = w(x - 2t, 0). \tag{4.14}$$

Burgers' equation was modelled with the above initial condition and Dirichlet boundary conditions, $w(-1, t) = 3$ and $w(1, t) = 1$, consistent with the exact solution (4.14). Figure 4.2 displays the solution at $T = 0.25$ without using a slope limiter. There is oscillatory behaviour of the solution near the discontinuity and the behaviour for h- and p-refinement is investigated. The left hand graphic of figure 4.3 displays the solution with linear, quadratic and quartic approximation. Although using higher order approximation appears to localise the oscillatory behaviour nearer to the shock

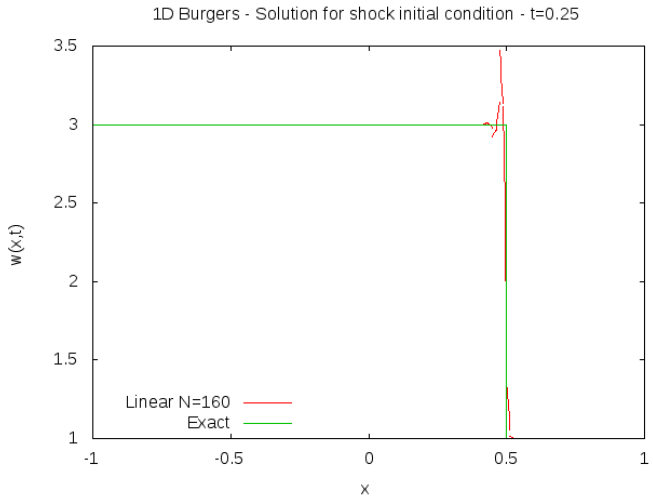

Figure 4.2: *Burgers' equation - Discontinuous solution with* 160 *linear elements at* $T = 0.25$ *with an RK-4 timestep of* $dt = 10^{-4}$ *and Lax-Friedrichs flux. The solution is exhibiting oscillatory behaviour near the shock.*

region at $x = 0.5$, the oscillations are still clearly present. The number of elements in the domain was also increased, with linear approximation. The right hand graphic of figure 4.3 demonstrates that oscillatory behaviour is localised nearer to the shock region, but again has not been removed.

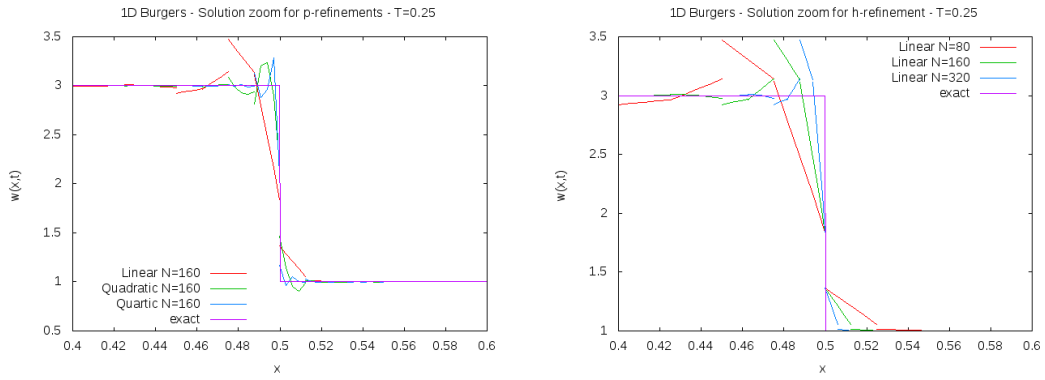

Figure 4.3: *Burgers' equation - Discontinuous solutions with h- and p-refinement at* $T = 0.25$ *without slope limiting. An RK-4 timestep of* $dt = 10^{-4}$ *and Lax-Friedrichs flux were used. The left hand figure represents* $N = 160$ *elements with linear, quadratic and quartic approximation and the right hand figure is linear approximation with* $N = 80$, 160 *and* 320 *elements for a domain* $x \in [-1, 1]$ *and the figures show a zoom for* $x \in [0.4, 0.6]$.

Table 4.1 shows the broken L^2-errors, at $T = 0.25$, as a function of the number of elements, N, and polynomial degree, $N_p - 1$. For this particular problem, the errors are decreasing when performing h- and p-refinement. However, convergence is slow, and using high order approximation has no effect on the convergence rate. It

remains a priority to remove the oscillatory behaviour present in figure 4.3 as this could represent non-physical fluctuations in the field. For example, if the physical process was modelling temperature or pressure, it is fundamental that these fields remain positive.

N	$d=1$	$d=2$	$d=3$	$d=4$
20	2.33×10^{-1}	1.81×10^{-1}	1.26×10^{-1}	8.73×10^{-2}
40	1.66×10^{-1}	1.27×10^{-1}	8.89×10^{-2}	6.17×10^{-2}
80	1.19×10^{-1}	8.98×10^{-2}	6.28×10^{-2}	4.36×10^{-2}
160	8.38×10^{-2}	6.35×10^{-2}	4.44×10^{-2}	3.08×10^{-2}
Rate	0.49	0.50	0.50	0.50

Table 4.1: *Burgers' equation - Discontinuous problem broken L^2-errors as a function of the number of elements, N, and polynomial degree, d, at $T = 0.25$. An RK-4 timestep of $dt = 10^{-4}$ and the Lax-Friedrichs flux were used.*

To remove the oscillations observed in figure 4.3, the MUSCL limiter, using the modified minmod function, was applied to the numerical solution after each explicit timestep of the RK-4 algorithm. Figure 4.4 represents the solution with the parameter in the modified minmod function set to $M = 0$ and $M = 5$. Firstly, it can be seen that the limiter does indeed do its job by removing the oscillations near the shock. The errors are not sensitive to the parameter choice M, because this parameter was designed to retain high order accuracy near regions of smooth extrema, as discussed in section 2.7.5, which are not present in the exact solution (4.14).

Figure 4.5 displays the solution with linear, quadratic and quartic approximation. The RK-4 timestepper and the RK-TVD3 timestepper were used to progress the solution through time (see Appendix D for RK-TVD3 algorithm description). In both cases the slope limiter is indeed removing the oscillatory behaviour for the different polynomial orders. In section 2.7, the MUSCL limiter was shown to enforce each explicit Euler timestep of the RK-TVD3 algorithm to be bounded in the TV semi-norm (2.40). As a consequence, this gave us a bound on the growth of the numerical solution in the total variation semi-norm in (2.28) leading to theorem 2.7.2. A similar bound is not possible for the RK-4 timestepper, but the numerical results for both timesteppers appear similar.

Table 4.2 shows the L^2-errors with the MUSCL limiter, with $M = 5$, for the RK-4 and RK-TVD3 timestepper. It can be seen that the L^2 errors are converging with

h-refinement and the RK-TVD3 and RK-4 method have almost identical measures of the error in the L^2 norm. However, since the use of the RK-TVD timestepper *ensures* the solution remains bounded in the TV semi-norm, and hence bounds the growth of spurious oscillations in the numerical solution, then this timestepper will be used throughout the rest of the analysis.

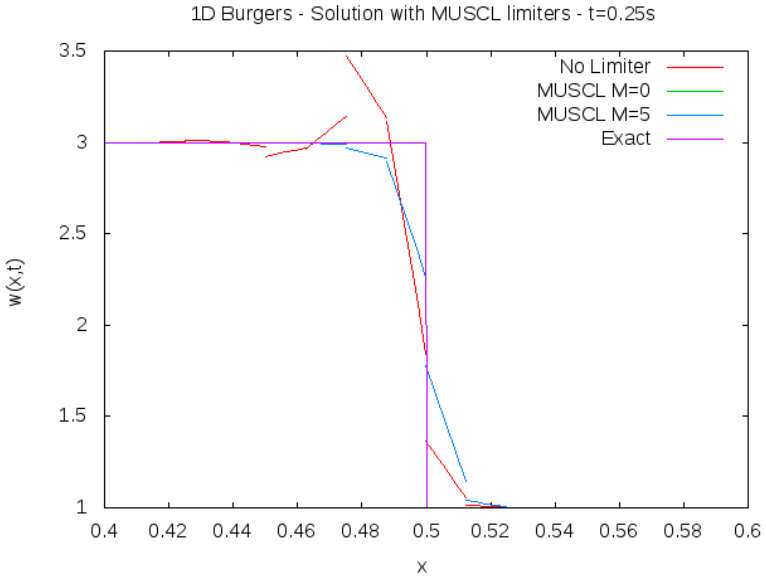

Figure 4.4: *Burgers' equation - Linear solutions with MUSCL slope limiting at* $T = 0.25$ *with* 160 *linear elements, an RK-4 timestep of of* $dt = 10^{-4}$ *and a Lax-Friedrichs flux for a domain* $x \in [-1, 1]$*. The figures show a zoom* $x \in [0.4 : 0.6]$*.*

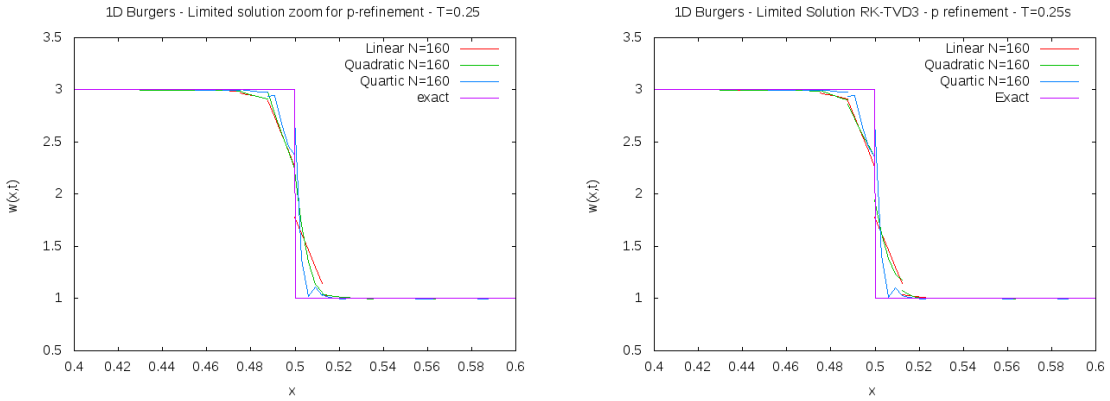

Figure 4.5: *Burgers' equation - Limited solutions with p-refinement at* $T = 0.25$ *with* 160 *elements and a Lax-Friedrichs flux for a domain* $x \in [-1, 1]$*. The left and right figures illustrate the RK-4 and RK-TVD3 timestepper solutions respectively, both with a timestep of* $dt = 10^{-4}$ *and a zoom* $x \in [0.4 : 0.6]$*.*

N	RKTVD3	RK4
20	3.32×10^{-1}	3.32×10^{-1}
40	2.35×10^{-1}	2.35×10^{-1}
80	1.67×10^{-1}	1.67×10^{-1}
160	1.18×10^{-1}	1.18×10^{-1}

Table 4.2: *Burgers' equation - Broken L^2-errors with h-refinement and linear approximation, with a Lax-Friedrichs flux. The RK-4 and RK-TVD3 timestepper are used to march the solution through time with MUSCL (M=5) slope limiting with a timestep of $dt = 10^{-4}$.*

4.2 Shallow water equations

We move onto a one dimensional discontinuous problem for the shallow water equations. In section 3.3 it was shown that these equations can be written in conservative form, with field and flux

$$\mathbf{w} = \begin{pmatrix} h \\ hu \end{pmatrix}, \quad \mathbf{f}(\mathbf{w}) = \begin{pmatrix} hu \\ \frac{1}{2}Gh^2 + u^2h \end{pmatrix}, \tag{4.15}$$

from which an alternative conservative form below was derived,

$$\mathbf{w} = \begin{pmatrix} h \\ u \end{pmatrix}, \quad \mathbf{f}(\mathbf{w}) = \begin{pmatrix} hu \\ Gh + \frac{1}{2}u^2 \end{pmatrix}, \tag{4.16}$$

which were both used successfully to solve a continuous test problem for the shallow water equations. In this chapter, we will discover that for a discontinuous problem different numerical solutions are obtained, using the same initial conditions, for the two different conservative forms. This result, which is not well documented in the literature, is explained in section 4.2.4. To understand how exact solutions are determined for discontinuous problems, it is essential to revisit the discussion on the Rankine Hugoniot conditions for shock propagation discussed in section 4.1.2.

4.2.1 Rankine Hugoniot conditions

In section 4.1.2, the integral form of a conserved PDE was considered and the velocity of a simple discontinuity was determined from the Rankine Hugoniot conditions. The conditions for a conserved vector field turn out to be very similar to that of a scalar field, but, this time there is a shock condition for *each* component of the field. To

be more precise, assuming a simple shock, and a conserved field $\mathbf{u} \in \mathbb{R}^m$, with corresponding flux $\mathbf{f}(\mathbf{u}) \in \mathbb{R}^m$, the Rankine Hugoniot conditions [1, p.180] are

$$-\frac{ds}{dt}[u_i] + [f(\mathbf{u})_i] = 0 \qquad i = 1 : m.$$

For the shallow water system in conserved form (4.15), with the field $[h, hu]^T$, the following Rankine Hugoniot conditions thus hold at a discontinuity:

$$\frac{ds}{dt} = \frac{[hu]}{[h]}, \qquad \frac{ds}{dt} = \frac{[hu^2 + Gh^2/2]}{[hu]}. \tag{4.17}$$

However, in the conserved form (4.16), with the field $[h, u]^T$, the Rankine Hugoniot conditions below hold at a discontinuity:

$$\frac{ds}{dt} = \frac{[hu]}{[h]}, \qquad \frac{ds}{dt} = \frac{[u^2/2 + Gh]}{[hu]}.$$

Physically, the conserved form of the equations with field, $[h, hu]^T$, represents the true conserved properties of the system, the conservation of fluid mass and so (4.17) are the correct Rankine Hugoniot conditions to use at a discontinuity.

4.2.2 Characteristic Curves

In section 3.1 characteristic curves were explored for a scalar field in 1 spatial dimension. Along these curves the PDE reduced to a simple ODE, along which the solution $w(x, t)$ remained constant. For an extension to a system of PDEs, we consider the conserved form (4.15) and write the shallow water PDEs as

$$h_t + uh_x + hu_x = 0,$$
$$u_t + Gh_x + uu_x = 0.$$

Taking a linear combination of the above equations

$$\alpha_1(h_t + uh_x + hu_x) + \alpha_2(u_t + Gh_x + uu_x) = 0,$$

and grouping the derivatives of h and u

$$\alpha_1\Big[h_t + \Big(u + \frac{G\alpha_2}{\alpha_1}\Big)h_x\Big] + \alpha_2\Big[u_t + \Big(u + \frac{\alpha_1}{\alpha_2}h\Big)u_x\Big] = 0. \tag{4.18}$$

Consider the linear combination below, with $\alpha_1 = 1$,

$$\frac{\alpha_2}{\alpha_1} = \pm\sqrt{\frac{h}{G}},$$

then (4.18) becomes,

$$h_t + (u \pm \sqrt{Gh})h_x \pm \sqrt{\frac{h}{G}}[u_t + (u \pm \sqrt{Gh})u_x] = 0. \tag{4.19}$$

If we define the characteristic curves, $C^{\pm}$, as

$$\frac{dx}{dt} = u \pm \sqrt{Gh}, \tag{4.20}$$

then the PDE (4.19) reduces to a set of ODEs, defined on the characteristic curves (4.20) giving the characteristic form:

$$\frac{dh}{dt} \pm \sqrt{\frac{h}{G}}\frac{du}{dt} = 0 \quad \text{along} \quad \frac{dx}{dt} = u \pm \sqrt{Gh}. \tag{4.21}$$

4.2.3 1D Dam-Break Setup

A 1D dam-break problem, where the domain is set to $\Omega = [-D, D]$, the constant $G = 10$ with Dirichlet boundary conditions $h(-D, t) = h_L$, $h(D, t) = h_R$, $h_L \geq h_R$, and the initial condition

$$h(x, 0) = \begin{cases} h_L & \text{if } x < 0 \\ h_R & \text{if } x > 0 \end{cases} \qquad u(x, 0) = 0.0, \tag{4.22}$$

is studied. We can imagine stationary water of height h_L separated by a thin sheet at $x = 0$ from stationary water of height h_R. At $t = 0$, the sheet is removed and, ignoring the time to lift the sheet and any friction, the higher fluid, in the region $x < 0$, travels into the region $x > 0$. The initial distribution is illustrated in figure 4.6 below.

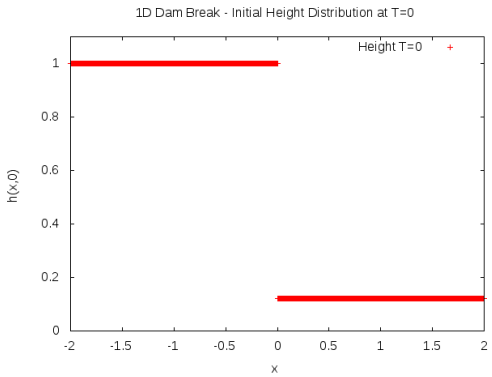

Figure 4.6: *Initial height distribution for 1D dam-break problem, $h = 1$ for $x < 0$ and $h = 0.12$ for $x > 0$.*

Huang *et al.* [16] use the Rankine Hugoniot conditions (4.17) at the shock and the characteristic curves (4.21) away from the shock, to find an exact solution that is determined from the solution of a non-linear problem. This makes the 1D dam-break problem ideal to understand how different numerical fluxes and the MUSCL slope limiter effect the accuracy of the computed solution.

For the specific initial condition $h_L = 1$, $h_R = 0.12$, there are a number of key features of the solution at time t as a function of the spatial coordinate x, as discussed below:

- $x < -\sqrt{G}t$ - The water remains undisturbed, and the left hand initial condition, $h = 1$, $u = 0$, is retained.
- $-\sqrt{G}t < x < 0.050\sqrt{G}t$ - At $x = -\sqrt{G}t$ there is a discontinuity in the gradient, the height decreases parabolically and the velocity increases linearly.
- $0.050\sqrt{G}t < x < 0.977\sqrt{G}t$ - At $x = 0.050\sqrt{G}t$ there is another discontinuity in the gradient. This is the shock region and the height and velocity are constant, $h_S = 0.423$, $u_S = 0.699\sqrt{G}$.
- $x > 0.977\sqrt{G}t$ - At $x = 0.977\sqrt{G}t$, there is a genuine discontinuity in the height and velocity. After this point, the water remains undisturbed and the right hand initial condition, $h = 0.12$, $u = 0$, is retained.

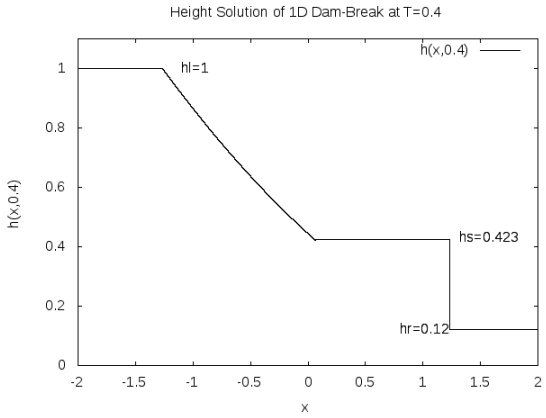

Figure 4.7: *1D dam-break - Exact height solution at $T = 0.4$ with $G = 10$ and the initial condition* (4.22) *with $h_L = 1$, $h_R = 0.12$. The discontinuities in the gradients, at $x \approx -1.27$ and $x \approx 0.06$, the true discontinuity, at $x \approx 1.24$, and the shock height, $h_s = 0.423$, are illustrated.*

The conservative form (4.15), with the field $[h, hu]^T$, is used to solve the 1D dam-break problem and the Lax-Friedrichs flux used as described in (3.12). Figure 4.8 demonstrates the height and momentum solution at $T = 0.4$, with linear approximation. There are two areas where oscillations have been introduced into the solution. There are large oscillations at $x \approx 0.1$, where the exact solution has a discontinuity in the gradient, and also at $x \approx 1.2$, where the exact solution has a shock.

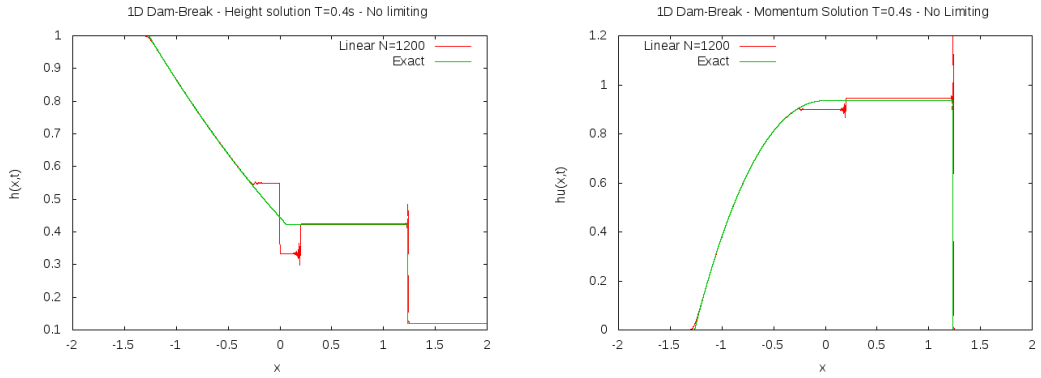

Figure 4.8: *1D dam-break - Height and momentum solution with at $T = 0.4$ with 1200 linear elements, a Lax-Friedrichs flux and an RK-TVD3 timestep of $dt = 10^{-4}$. If no limiting is applied oscillations appear in the solution near discontinuities.*

To remove these oscillations, a limiter is applied after each explicit timestep of the RK-TVD3 algorithm. The MUSCL limiter (2.35) was implemented, using the modified minmod function (2.39) with the smoothness parameter set to $M = 50$, and figure 4.9 demonstrates the height solution, h. At the shock, $x \approx 1.23$, the oscillations have been removed by the MUSCL slope limiter. The solution has been plotted with 5 plot points per element, and so the shock is resolved in approximately 4 elements. At the gradient discontinuity the MUSCL limiter is removing most of the oscillatory behaviour, but has not resolved the solution as well as at the shock.

4.2.4 Importance of shocks and conserved forms

The conserved field $[h, u]^T$ and its associated flux in equation (4.16), derived in section 3.3, were now used to model the dam-break problem with the Lax-Freidrichs flux. Figure 4.10 demonstrates the height solution, at $T = 0.4$, for both the conserved fields $[h, u]^T$ and $[h, hu]^T$. The figure clearly shows that if the conserved field $[h, u]^T$ is used, the numerical solution is not a good approximation to the analytic solution and is breaking early, at the same spatial resolution. The conserved PDEs with the

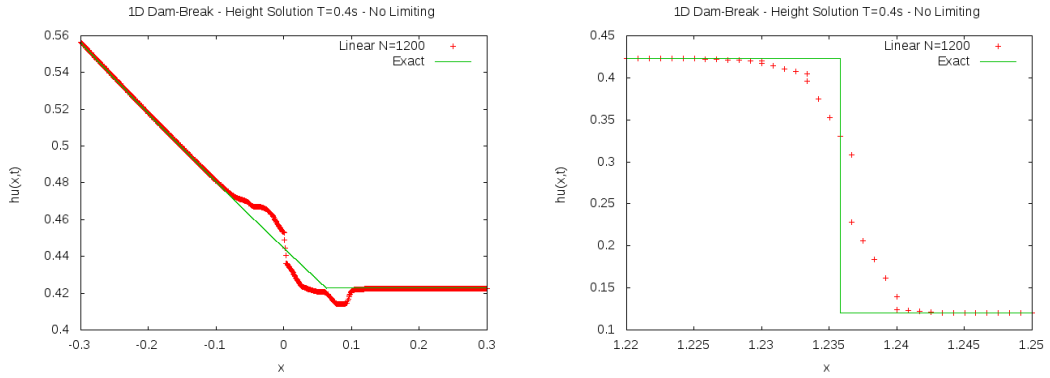

Figure 4.9: *1D dam-break - Height solution with* 1200 *elements, linear approximation, a Lax-Friedrichs flux and RK-TVD3 timestep of* $dt = 10^{-4}$ *with the domain* $x \in [-2, 2]$. *The left hand and right hand figures represent a zoom for* $x \in [-0.3 : 0.3]$ *and* $x \in [1.22 : 1.25]$ *respectively. The true discontinuity is being resolved in approximately* 4 *elements and the oscillations of the solution near the gradient discontinuity are nearly removed.*

field $[h, u]^T$ were derived directly from the conserved form (4.15). However, these two forms have different Rankine Hugoniot conditions at the shock, as discussed in section 4.2.1, yielding different shock velocities. The conserved field $[h, hu]^T$ represents the true conserved properties of the system, the conservation of fluid mass and linear momentum, and the conserved field $[h, u]^T$ yields alternative, but non-physical solutions. It is thus fundamental to enforce the conservation of the correct physical variables, height and momentum, for solutions with shocks. The correct conserved form (4.15), with field $[h, hu]^T$ will be used throughout the rest of this chapter.

The Roe average flux, described in section 2.2.3, can also be used to pass information between elements. This requires the evaluation of a new flux constant, C_{RA}, at each element edge. Consider the internal and external fields at a boundary: $\mathbf{w}_e = [h_e, (hu)_e]^T$ and $\mathbf{w}_e = [h_e, (hu)_e]^T$. The constant is evaluated as:

$$C_{RA} = max_{\bar{h},\bar{u}} |\bar{u} \pm \sqrt{g\bar{h}}|, \tag{4.23}$$

$$\bar{h} = \frac{h_i + h_e}{2}, \qquad \bar{u} = \frac{\sqrt{h_i}u_i + \sqrt{h_e}u_e}{\sqrt{h_i} + \sqrt{h_e}}. \tag{4.24}$$

The algorithm to compute the Roe average constant at an element edge can be seen below and further details of the implementation can be seen in Appendix B.

1 `Find` $\mathbf{w}_i$ `and` $\mathbf{w}_e$
2 `Compute` $\bar{h} = \frac{h_i+h_e}{2}$, $\bar{u} = \frac{\sqrt{u_i}h_i+\sqrt{u_e}h_e}{\sqrt{u_i}+\sqrt{u_e}}$, $C^- = |\bar{u} - \sqrt{G\bar{h}}|$, $C^+ = |\bar{u} + \sqrt{G\bar{h}}|$
3 `Take` $C_{RA} = \max(C^-, C^+)$

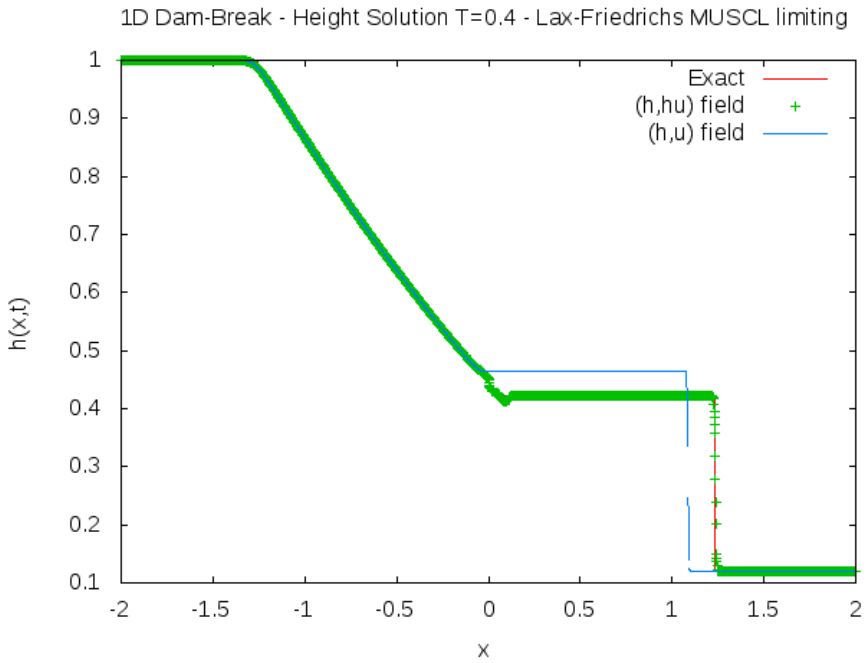

Figure 4.10: *1D dam-break - Height solution with* 600 *elements, linear approximation, Lax-Friedrichs flux and RK − TVD3 timestep of* $dt = 10^{-4}$ *at T=0.4s. The numerical solutions are using the conserved forms* $[h, hu]^T$ *and* $[h, u]^T$ *of the shallow water equations, given by* (4.15) *and* (4.16) *respectively.*

To measure errors, both the broken L^1 and L^2-norm were used to measure the height and momentum field respectively. To combine these in a single measure for the L^2 norm, equation (3.15) is used, and for the L^1 norm, equation (4.25) below is used,

$$||\mathbf{w} - \mathbf{w}_h||_{\Omega,h,L^1} = \sum_{i=1}^{m} ||w_i - w_{i,h}||_{\Omega,h,L^1}. \tag{4.25}$$

Table 4.3 demonstrates the errors, measured at $T = 0.4$, with linear approximation using the Roe average (RA) and Lax-Friedrichs (LF) flux. The rates were estimated through best fit plots of ln(Error) against $\ln(N)$, as seen in figure 4.11 for the L^1 and L^2 norm. The graphs also show the errors without limiting, using a Roe average flux, are large and are converging very slowly, if it all.

The errors are smaller in the L^1-norm than the L^2-norm and the convergence rate is also faster. There is little known about convergence rates for discontinuous problem using the DGFEM, but since the L^1 errors are consistently smaller than the L^2 errors, and are generally used in the literature to measure errors for discontinuous problems, then this norm will be chosen for future convergence analysis.

Table 4.3 and figure 4.11 indicate that the errors using a Lax-Friedrichs flux and

Roe average flux are almost identical, with the Lax-Friedrichs flux converging at a marginally faster rate. The computational times for the Roe average and Lax-Friedrichs flux can be seen in table 4.4, and the Roe average flux also takes slightly more time to compute. For these reasons, the Lax-Friedrichs flux will be used for future computations.

N	LF L^2	LF L^1	RA L^2	RA L^1
150	5.87×10^{-2}	4.00×10^{-2}	5.85×10^{-2}	3.93×10^{-2}
300	3.53×10^{-2}	1.94×10^{-2}	3.49×10^{-2}	1.92×10^{-2}
600	2.51×10^{-2}	1.02×10^{-2}	2.50×10^{-2}	1.06×10^{-2}
1200	2.62×10^{-2}	6.27×10^{-3}	2.70×10^{-2}	6.30×10^{-3}
Rate	0.40	0.90	0.38	0.88

Table 4.3: *1D dam-break - Broken L^1 and L^2 errors at $T = 0.4$ with linear approximation and MUSCL limiting. The results are with an RK-TVD3 timestep of $dt = 10^{-5}$ with the Lax-Friedrichs (LF) and Roe average (RA) fluxes.*

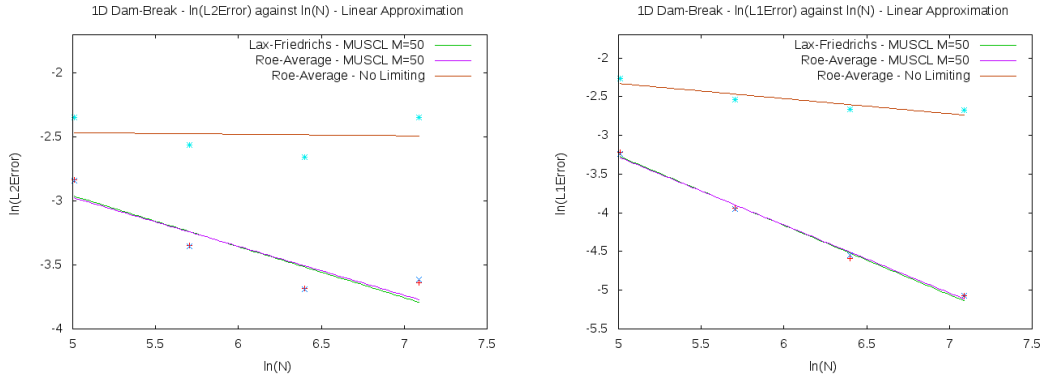

Figure 4.11: *1D dam-break - Broken L^1 and L^2 errors at $T = 0.4$ with linear approximation and a timestep of $dt = 10^{-5}$ as a function of the number of elements. The results are shown with Lax-Friedrichs and Roe average flux with MUSCL, $M = 50$, limiting. Errors with the Roe average flux, without limiting, are also shown.*

N	150	300	600	1200
LF Time/s	38.0	82.4	191.2	385.7
RA Time/s	38.5	86.0	199.8	402.8

Table 4.4: *1D dam-break - Computation times, in seconds, with linear approximation, with a Lax-Friedrichs and Roe average flux, at $T = 0.4$ using the MUSCL, $M = 50$, limiter and an RK-TVD3 timestep of $dt = 10^{-5}$.*

The effect of higher order approximation on the L^1-errors using the Lax-Friedrichs flux is shown in table 4.5, for linear, quadratic and cubic approximation, with the number of elements varied from $N = 150 - 1200$. The height and momentum errors

are combined using (4.25). Figure 4.12 represents h-refinement for the 1D dam-break problem for different polynomial orders. For a fixed element size, increasing the polynomial order does not reduce the error and similar convergence rates are obtained for the different polynomial orders. The computation times are displayed in table 4.6. Increasing the polynomial order increases the number of nodes in an element, the mass matrix is larger, and so the timestepping algorithm (2.42) takes longer to complete. These observations suggest that there is no advantage to be gained from using higher order bases for the dam-break problem in the DGFEM.

N	$d=1$	$d=2$	$d=3$
150	4.00×10^{-2}	3.94×10^{-2}	4.45×10^{-2}
300	1.94×10^{-2}	1.83×10^{-2}	2.57×10^{-2}
600	1.02×10^{-2}	1.04×10^{-2}	1.22×10^{-2}
1200	6.27×10^{-3}	7.11×10^{-3}	7.40×10^{-3}
Rate	0.90	0.82	0.88

Table 4.5: *1D dam-break - L^1 errors, with $N = 150 - 1200$ elements, at $T = 0.4$ with linear approximation and the MUSCL, $M = 50$, limiting. The results shown are using linear, quadratic and cubic approximation.*

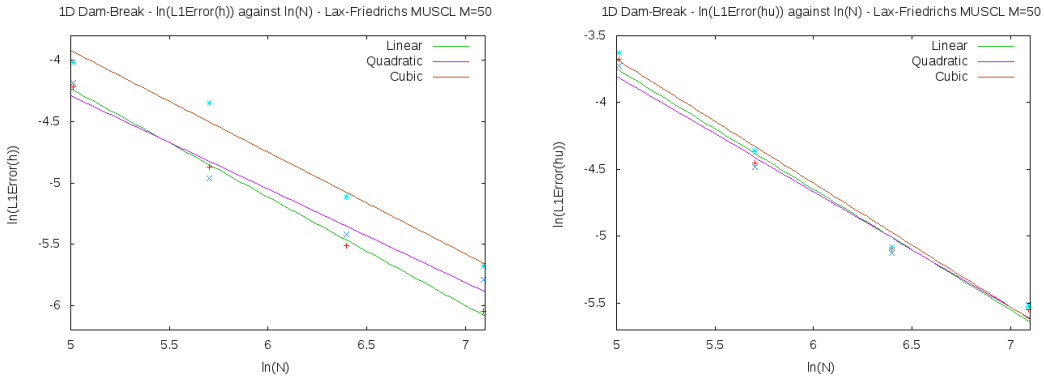

Figure 4.12: *1D dam-break - Broken L^1-errors for the Height and Momentum solution with MUSCL, $M = 50$, limiter, Lax-Friedrichs Flux and RK-TVD3 timestep of $dt = 10^{-5}$. The results shown are with linear, quadratic and cubic approximation.*

N	$d=1$	$d=2$	$d=3$
150	38.0	47.5	61.4
300	82.4	107.8	141.8
600	191.2	261.8	300.6
1200	385.7	690.0	814.2

Table 4.6: *1D dam-break - Computation times, in seconds, with a Lax-Friedrichs flux and RK-TVD3 timestep of $dt = 10^{-5}$, for $N = 150 - 1200$ elements, with linear, quadratic and cubic approximation.*

4.2.5 Field Conservation

Consider the PDEs (4.15) and integrate each of the conserved fields over the domain,

$$\int_{-2}^{2}\Big(\frac{\partial h}{\partial t}+\frac{\partial hu}{\partial x}\Big)dx=0 \implies \frac{d}{dt}\int_{-2}^{2}hdx+[hu]_{-2}^{2}=0,$$
$$\int_{-2}^{2}\Big(\frac{\partial hu}{\partial t}+\frac{\partial(\frac{1}{2}Gh^2+u^2h)}{\partial x}\Big)dx=0 \implies \frac{d}{dt}\int_{-2}^{2}hudx+[u^2h+\frac{1}{2}Gh^2]_{-2}^{2}=0,$$
$$\implies \frac{d}{dt}\int_{-2}^{2}hdx=0, \qquad \frac{d}{dt}\int_{-2}^{2}hudx-4.928=0.$$

For small times, the dam-break does not reach the boundary, and the field remains unchanged, so that $h(-2)=1$, $h(2)=0.12$, $u(-2)=u(2)=0$, which are also enforced by the Dirichlet boundary conditions. The integral of the height field, h, should stay constant, and the momentum field, hu, depend linearly on time. From the initial condition,

$$\int_{-2}^{2}h(x,0)dx=\int_{-2}^{0}1dx+\int_{0}^{2}0.12dx=2.24,$$
$$\int_{-2}^{2}hu(x,0)dx=\int_{-2}^{2}0dx=0.$$

So, for small enough times, the integrals should satisfy,

$$\int_{-2}^{2}h(x,t)dx=2.24, \qquad \int_{2}^{2}hu(x,t)dx=4.928t. \tag{4.26}$$

Table 4.7 illustrates the integral average of the fields, with linear approximation at times $T=0-0.4$. It can be seen that the height field, h, remains unchanged, and the momentum field, hu, is linearly increasing in time. The results are in excellent agreement with the expected values (4.26).

T	L^1 h field	L^1 hu field
0	2.24	0
0.1	2.24	0.4928
0.2	2.24	0.9856
0.3	2.24	1.4784
0.4	2.24	1.9712

Table 4.7: *1D dam-break - The integral of the height and momentum field across the domain, measured in the broken L^1-norm, as a function of the integration time, T. The Lax-Friedrichs flux, an RK-TVD3 timestep of $dt=10^{-4}$ with $N=1200$ equally spaced linear elements are used.*

Chapter 5

2D Dam-Break Problem

5.1 Shallow water equations

For a domain $\Omega \subset \mathbb{R}^2$, the shallow water equations model water waves under the condition that a horizontal length scale is much larger than a vertical length scale i.e. waves that have a wavelength much greater than the water depth. The equations can be derived by considering fluid mass and momentum conservation in a control volume of fluid [17].

The fluid is assumed to be of constant density, ρ, and any frictional, viscous or Coriolis forces are ignored. The topographic height is also set constant, so the fluid flows on a flat surface and hydrostatic equilibrium is assumed in the vertical direction. These assumptions imply the velocity of the fluid is constant at a given point $\mathbf{x}$ in the domain, so there is no variation with the fluid height. Defining the height, $h(\mathbf{x}, t)$, and the velocities $u(\mathbf{x}, t)$ and $v(\mathbf{x}, t)$ in the x and y direction respectively, the shallow water equations can be written,

$$\begin{aligned}
&\frac{\partial}{\partial t}(h) + \frac{\partial}{\partial x}(hu) + \frac{\partial}{\partial x}(hv) = 0, \\
&\frac{\partial}{\partial t}(hu) + \frac{\partial}{\partial x}(u^2 h + \frac{1}{2}Gh^2) + \frac{\partial}{\partial y}(uvh) = 0, \\
&\frac{\partial}{\partial t}(hv) + \frac{\partial}{\partial x}(uvh) + \frac{\partial}{\partial y}(v^2 h + \frac{1}{2}Gh^2) = 0,
\end{aligned}$$

and in conservative form (2.1), by defining the conserved field $\mathbf{w} = (h, hu, hv)^T \in \mathbb{R}^3$

and the flux matrix $F \in \mathbb{R}^{3\times 2}$

$$F = \begin{pmatrix} uh & vh \\ u^2h + \frac{1}{2}Gh^2 & uvh \\ uvh & v^2h + \frac{1}{2}Gh^2 \end{pmatrix}. \tag{5.1}$$

As discussed in a 1D specialisation of these equations in section 4.2, these equations pose a challenge numerically because solutions can have discontinuities present.

5.2 Discontinuous Galerkin Method in 2D

A discontinuous Galerkin method in 2D, with rectangular finite elements, is more complex than in a single spatial dimension, for the following reasons:

- Neighbouring Scheme - An element in the mesh now has four neighbours, as opposed to two neighbours in one spatial dimension and the elemental linear system (2.18) requires coupling between adjacent faces in the mesh, as discussed in section 2.3.1.

- Numerical Flux matrix- The contributions to this matrix are line integrals of the numerical flux along the edge of the 2D element, instead of function evaluations at the edge points of an element in one spatial dimension. The numerical integration scheme was discussed in section 2.4.

- Mass and Flux matrix - The contributions to these matrices are 2D surface integrals over the element, instead of line integrals across the element in one spatial dimension. The numerical integration scheme was also discussed in section 2.4.

- Slope Limiting - The process of slope limiting, to remove spurious oscillations appearing in numerical solutions, needs to be addressed. This will be discussed in the following section for rectangular finite elements aligned with the coordinate axes.

5.2.1 2D Slope Limiting

In section 2.7, slope limiting was introduced in the context of 1D problems. For the MUSCL slope limiter, a TVBM property for the solution, where the solution remains bounded in the TV semi-norm, was obtained by considering values of the field at element edges in 1D grids (see theorem 2.7.2).

From the literature, it appears that a rigorous proof of 2D stability remains largely unknown for general meshes. In this analysis a 2D limiter will be implemented that will operate on piecewise bilinear approximation, of the form $w = (a + bx)(c + dy)$, in each element.

A slope limiter essentially attempts to remove spurious oscillations from the approximate solution by suitably replacing the approximate solution gradient in each element. However, since a bilinear approximation has a twist term, Cxy, then a single measure for the x and y gradient across an element does not exist, making the limiting process less intuitive.

A rectangular domain will be considered, aligned with the coordinate axes. Consider a rectangular element $[x_i, x_{i+1}] \times [y_j, y_{j+1}]$, where i, j are indices for the elements in the x and y direction. After each explicit Euler timestep, before the limiter is implemented, the approximate solution in element (i, j) will be converted from bilinear to pure linear approximation of the form

$$w(\mathbf{x}, t) = \bar{w}_{i,j}(t) + w_x(t)\Big(\frac{x - x_{i+1/2}}{\Delta x_i/2}\Big) + w_y(t)\Big(\frac{y - y_{j+1/2}}{\Delta y_j/2}\Big), \tag{5.2}$$

where $w_x(t)$, $w_y(t)$ are the average gradients in the x and y direction and $\bar{w}_{i,j}(t)$ is the integral average of the field in element (i, j),

$$\bar{w}_{i,j}(t) = \frac{\int_{x_i}^{x_{i+1}} \int_{y_j}^{y_{j+1}} (a + bx)(c + dy) dx dy}{\Delta x_i \Delta y_j}.$$

This integral average over the element, is the field at the central point $(x_{i+1/2}, y_{j+1/2})$ of the element, as the argument below shows:

$$\begin{aligned}
\bar{w}_{i,j}(t) &= \frac{\int_{x_i}^{x_{i+1}} (a + bx) dx \int_{y_j}^{y_{j+1}} (c + dy) dy}{\Delta x_i \Delta y_j} \\
&= \frac{\Delta x_i (a + b/2(x_{i+1} + x_i)) \Delta y_j (c + d/2(y_{j+1} + y_j)}{\Delta x_i \Delta y_j} \\
&= \Big(a + b\frac{(x_{i+1} + x_i)}{2}\Big)\Big(c + d\frac{(y_{j+1} + y_j)}{2}\Big) = w_{i+1/2,j+1/2}.
\end{aligned}$$

The choice for the average gradients $w_x(t)$, $w_y(t)$ is not unique and in the implementation the conversion to pure linear approximation is achieved through the following algorithm, for each field component:

- Find the approximate solution at the four corner (nodal) points of the element: $w_{i,j}, w_{i+1,j}, w_{i,j+1}, w_{i+1,j+1}$.
- Find the gradients along bottom, top, left and right edge of element:

$$\begin{aligned} w_x(t)_{Bottom} &= \frac{w_{i+1,j} - w_{i,j}}{\Delta x_i} \qquad w_x(t)_{Top} = \frac{w_{i+1,j+1} - w_{i,j+1}}{\Delta x_i} \\ w_x(t)_{Left} &= \frac{w_{i,j+1} - w_{i,j}}{\Delta y_j} \qquad w_x(t)_{Right} = \frac{w_{i+1,j+1} - w_{i+1,j}}{\Delta y_j} \end{aligned} \tag{5.3}$$

- Take arithmetic averages of quantities to find the gradients:

$$w_x(t) = \frac{w_x(t)_{Bottom} + w_x(t)_{Top}}{2} \qquad w_y(t) = \frac{w_y(t)_{Left} + w_y(t)_{Right}}{2} \tag{5.4}$$

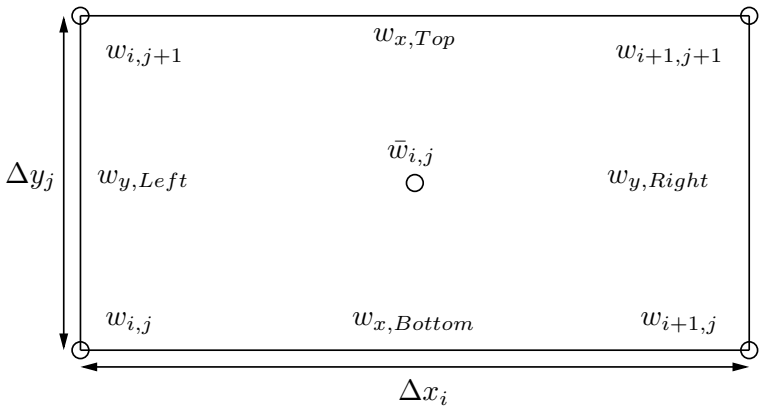

Figure 5.1: *A rectangular element illustrating the data required to perform 2D limiting on bilinear approximation. The figure illustrates the approximate solution at the four nodes, the element widths, the approximate gradients given by* (5.3) *and the element average (at the central point).*

After the true bilinear approximation has been converted to linear approximation of the form (5.2), without a twist term, the limiting process is similar to the 1D case. A variant of the original 1D modified minmod function (2.39) is used,

$$\bar{m}(a_1, a_2, a_3) = \begin{cases} a_1 & \text{if } |a_1| \leq M\Delta h^2 \\ m(a_1, a_2, a_3) & \text{otherwise.} \end{cases} \tag{5.5}$$

The x and y directions are treated independently, and in each element, for each field, $w_x(t)$ and $w_y(t)$ are limited separately using the 2D modified minmod function (5.5).

Consider the x direction, the gradient $w_x(t)$ is replaced, using the 2D modified minmod function (5.5) with $h = \Delta x_i$, to

$$w_x^{'}(t) = \bar{m}\Big(w_x(t), \frac{\bar{w}_{i+1,j}(t) - \bar{w}_{i,j}(t)}{\Delta x_i}, \frac{\bar{w}_{i,j}(t) - \bar{w}_{i-1,j}(t)}{\Delta x_i}\Big), \tag{5.6}$$

where $\bar{w}_{i,j}$ is the average in element (i, j) and $\bar{w}_{i+1,j}$ and $\bar{w}_{i-1,j}$ are the averages in elements $(i + 1, j)$ and $(i - 1, j)$, the west and east neighbours.

Similarly for the y direction, the gradient $w_y(t)$ is replaced using the modified minmod function (5.5), with $h = \Delta y_j$, to

$$w_y^{'}(t) = \bar{m}\Big(w_y(t), \frac{\bar{w}_{i,j+1}(t) - \bar{w}_{i,j}(t)}{\Delta y_j}, \frac{\bar{w}_{i,j}(t) - \bar{w}_{i,j-1}(t)}{\Delta y_j}\Big), \tag{5.7}$$

where $\bar{w}_{i,j+1}$ and $\bar{w}_{i,j-1}$ are the averages in elements $(i, j+1)$ and $(i, j-1)$, the north and south neighbours.

To ensure conservation of the field, the average value, $\bar{w}(t)$, is forced to remain invariant by the limiting process. This average value, for linear approximation, is the field at the central point of the element and so the new unknowns, at the four nodal positions, are reconstructed through the following formulae:

$$\begin{aligned} w_{i,j} &= \bar{w}_{i,j}(t) - \Delta x_i w_x^{'}/2 - \Delta y_j w_y^{'}/2, \\ w_{i+1,j} &= \bar{w}_{i,j}(t) + \Delta x_i w_x^{'}/2 - \Delta y_j w_y^{'}/2, \\ w_{i,j+1} &= \bar{w}_{i,j}(t) - \Delta x_i w_x^{'}/2 + \Delta y_j w_y^{'}/2, \\ w_{i+1,j+1} &= \bar{w}_{i,j}(t) + \Delta x_i w_x^{'}/2 + \Delta y_j w_y^{'}/2. \end{aligned} \tag{5.8}$$

The implementation of this 2D slope limiter into the `oomph-lib` is explained in more detail in Appendix C. This includes a discussion of the process to find an element's neighbour and the actual implementation of the algorithm above.

5.3 2D Dam-Break Problem

A discontinuous problem in 2D can now be implemented, that of a 2D dam-break. The Lax-Friedrichs flux will be used, using the definition (2.7), to define numerical fluxes at element edges. This requires maximising the eigenvalues of the Jacobian

of the flux matrix in the direction of the outer unit normal at an element edge. Letting the general outer unit normal be $(n_x, n_y)^T$, it is shown in Appendix A that the eigenvalues are,

$$\lambda_1 = \hat{u}_1 n_x + \hat{u}_2 n_y + \sqrt{G\hat{u}_0}, \qquad \lambda_2 = \hat{u}_1 n_x + \hat{u}_2 n_y, \qquad \lambda_3 = \hat{u}_1 n_x + \hat{u}_2 n_y - \sqrt{G\hat{u}_0},$$

where the variables $\hat{u}_0 = h$, $\hat{u}_1 = u$, $\hat{u}_2 = v$. For rectangular meshes, with outer unit normals aligned with the x and y axes, the eigenvalues simplify to:

- $\mathbf{n} = \pm(0,1)^T$ - Normal parallel to y-axis:

$$\lambda_1 = \pm\hat{u}_2 + \sqrt{G\hat{u}_0}, \qquad \lambda_2 = \pm\hat{u}_2, \qquad \lambda_3 = \pm\hat{u}_2 - \sqrt{G\hat{u}_0}.$$

- $\mathbf{n} = \pm(1,0)^T$ - Normal parallel to x-axis:

$$\lambda_1 = \pm\hat{u}_1 + \sqrt{G\hat{u}_0}, \qquad \lambda_2 = \pm\hat{u}_1, \qquad \lambda_3 = \pm\hat{u}_1 - \sqrt{G\hat{u}_0}.$$

In both cases the second eigenvalue can never be the largest in absolute value, and the Lax-Friedrichs constant will be the maximum in absolute value of the first and third eigenvalue across all possibilities of the field over the boundary. Assuming that this maximum is taken only over the internal or external field, the resulting optimization problem is almost identical to that of a 1D dam-break (3.13). The only difference is what velocity to use in the optimization algorithm. If the outer unit normal is parallel to y-axis, the y-velocity will be used in the optimization algorithm, and if the outer unit normal is parallel to the x-velocity, the x-velocity will be used in the optimization algorithm. See Appendix B for further details of the Numerical Flux implementation.

5.3.1 Smoothed 2D Dam-Break Problem

For the smoothed 2D dam-break problem, the domain is set to $(x, y) \in [-1, 1] \times [-1, 1]$, with the initial condition

$$\begin{aligned} h(x, y, 0) &= 0.75 - 0.25 \tanh(200((x^2 + y^2) - 0.25)), \\ u(x, y, 0) &= v(x, y, 0) = 0. \end{aligned} \tag{5.9}$$

This is clearly not a genuine discontinuity, since the tanh function is smooth. However, $\tanh(x)$ changes very rapidly about $x = 0$ from -1 to $+1$. If $x^2 + y^2 < 0.25$,

$h(x, y, 0) \approx 1$, and if $x^2 + y^2 > 0.25$, $h(x, y, 0) \approx 0.5$. The nearly discontinuous initial condition is illustrated in figure 5.2, where the height distribution is plotted as a function of the x-coordinate over the entire mesh.

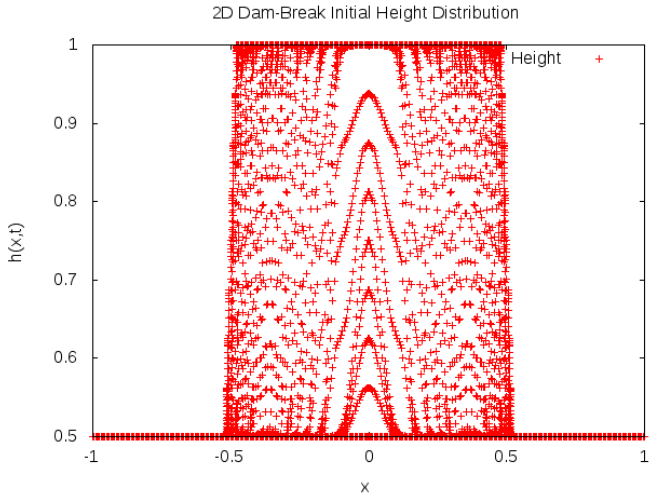

Figure 5.2: *2D dam-break - Initial height distribution plotted as a function of the x-coordinate. The initial condition* (5.9) *is strictly smooth, but the height changes rapidly near the region* $x^2 + y^2 = 0.25$*, making the initial distribution close to a true discontinuity.*

From the literature, it appears that no analytic exact solution exists to this initial value problem and validation of the results was performed by comparing the results with code written by Chris Johnson [18]. He uses a finite volume method, devised by Jiang *et al.* [5], which used a piecewise constant approximation within each element and a limiter, called MinMod1, to slope limit the solution after each explicit timestep of the Runge-Kutta algorithm.

The dam-break problem was modelled using the Lax-Friedrichs flux, periodic boundary conditions, and the slope limiter described in section 5.2.1, for a linear approximation. Figure 5.3 illustrates the average height solution in each element, for elements lying on the y-axis, for both a non-limited and MUSCL limited solution. The plot shows that different height solutions are obtained with and without a slope limiter, with the non-limited solution oscillating in the region $x \in \pm[0.35 : 0.70]$. For a 100×100 grid the oscillations were so severe that no computational results were obtainable without a slope limiter at $T = 0.4$.

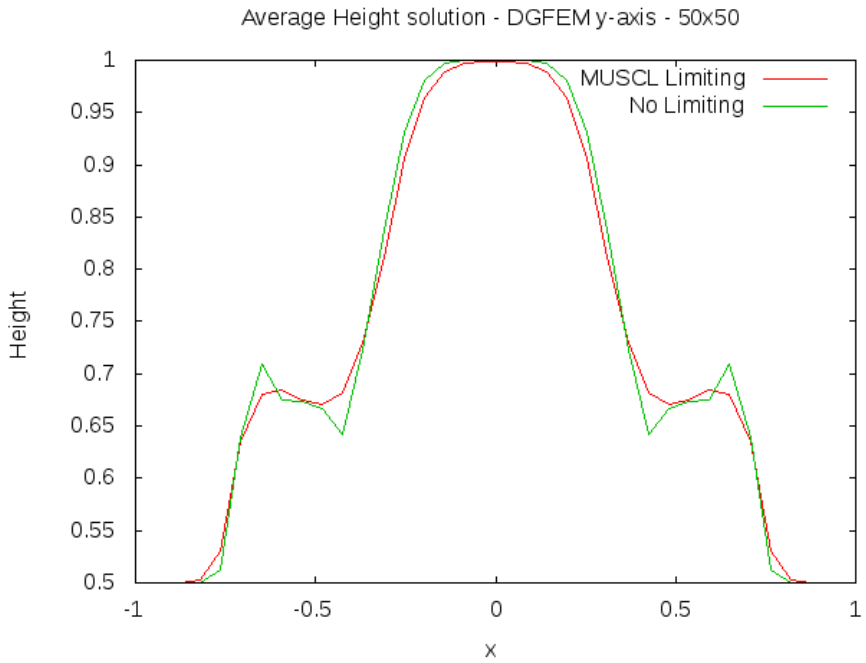

Figure 5.3: *2D dam-break - Illustration of the DGFEM average height solution for the dam-break problem on a* 50×50 *grid with an RK-TVD3 timestep of* $dt = 0.01$ *and Lax-Friedrichs flux for elements lying on the y-axis. Oscillatory behaviour can be observed in the numerical solution without slope limiting but disappear with the 2D slope limiter.*

Figure 5.4 is an illustration of the height density for different resolutions of the grid, by plotting the average height in each element over the domain. This illustration shows that as the resolution is increased, the shock, at $r = \sqrt{x^2 + y^2} \approx 0.73$, is better resolved and circular symmetry is generally preserved. However, in the 50×50 plot, the height density is slightly greater at a radial distance of $r \approx 0.5$ in the x and y direction than at the same radial distance in directions 45^o to the coordinate axes, indicating a potential asymmetry of the 2D slope limiter.

Figure 5.5 is a plot of the average heights along the line $y = x$ for different resolutions of the grid. It can be clearly seen that the shock at $r \approx 0.73$ is better resolved as the mesh resolution is increased.

The DGFEM 2D dam-break results and an axisymmetric dam-break problem, obtained by Chris Johnson [18], are compared. The axisymmetric dam-break problem implicitly assumes circular symmetry so only the radial direction is considered and solutions can be found at a high spatial resolution. Figure 5.6 illustrates the axisymmetric solution using a high resolution 12800 element 1D grid. The density plot is the axisymmetric solution mapped onto a 12800×12800 2D grid and a plot

of the axisymmetric solution as a function of the radial distance, r, is also shown.

Figure 5.7 illustrates a series of height density plots of the difference between the element average from the 2D dam-break DGFEM results and the high resolution axisymmetric FVM solution. As the DGFEM resolution is increased, the difference between the solutions appears to be diminishing in the smoother regions of the solution. The discontinuous Galerkin method is also resolving *where* the shock location is, to a high degree of accuracy, as the spatial resolution is increased. However, near the discontinuity, the differences in the solution are still large.

Results for the actual 2D dam-break problem using the finite volume method were obtained from Chris Johnson [18]. A better understanding of the DGFEM accuracy was obtained by plotting the average height solution at different cross sections through the domain for the FVM and DGFEM solutions, solved at the same spatial resolution. The average height solutions, along the x-axis, y-axis and the line $y = x$, are plotted in figures 5.8, 5.9 and 5.10 respectively. These graphs illustrate that the FVM and DGFEM are both resolving the shock in approximately 6 elements on a 400×400 grid.

A plot of average height solutions of the DGFEM along the x-axis, y-axis and the line $x = y$ is illustrated in figure 5.11 along with the high resolution axisymmetric dam-break solution. Along the x and y-axis the average height solutions are identical and are also very similar to the average height solution along the line $y = x$.

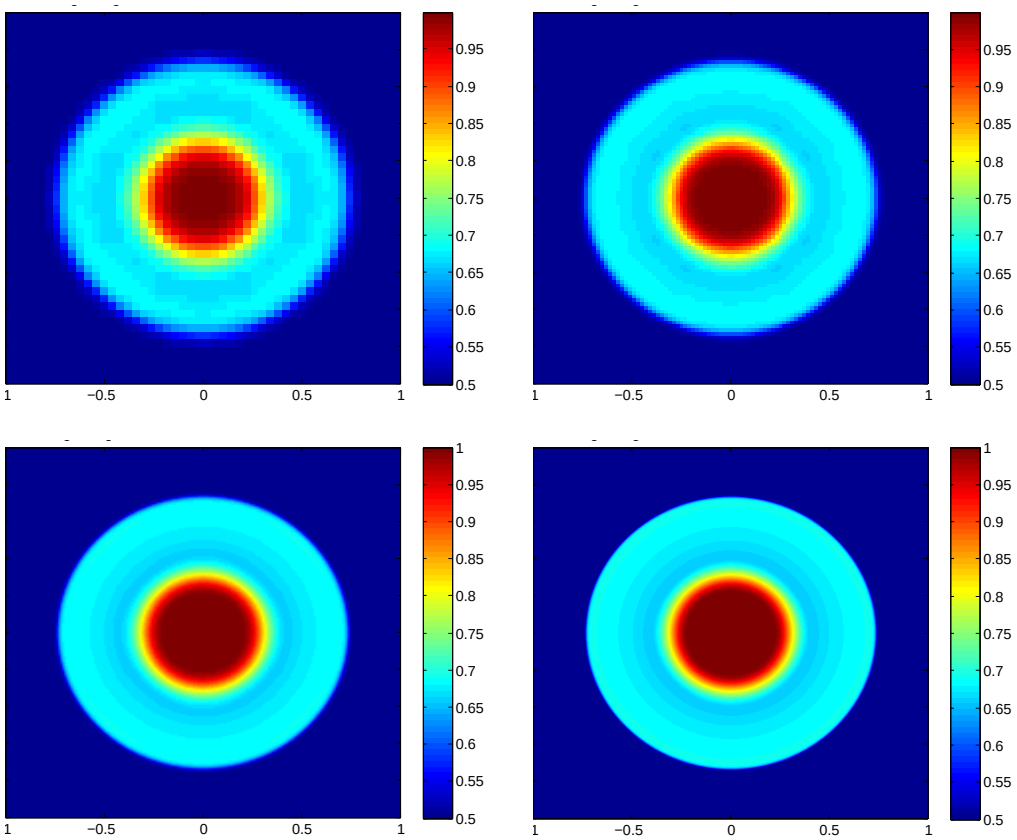

Figure 5.4: *2D dam-break - Density plots of the average height field in each element, at* $T = 0.5$*, with a Lax-Friedrichs flux and the 2D slope limiter. The figures represent a* 50×50*,* 100×100*,* 200×200 *and a* 400×400 *grid with RK-TVD3 timesteps of* $dt = 0.01, 0.005, 0.0025, 0.00125$ *respectively. A shock at* $r = \sqrt{x^2 + y^2} \approx 0.73$ *can be observed in the plots.*

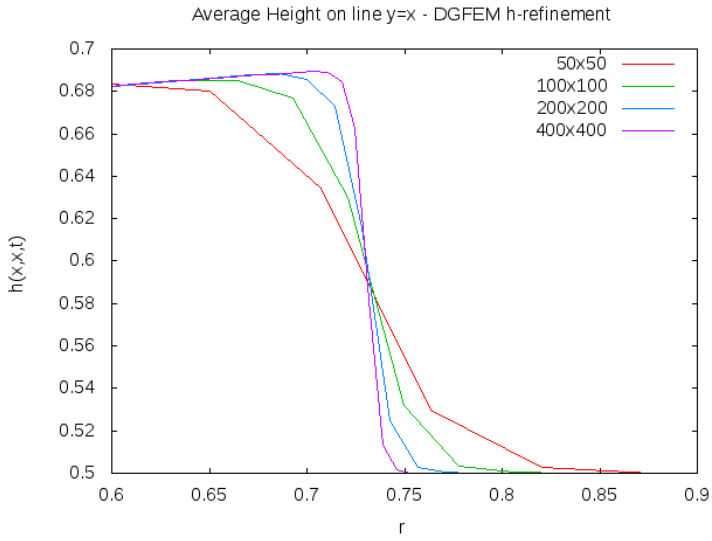

Figure 5.5: *2D dam-break - Average height solution along the line* $y = x$ *for h-refinement. The figures represent data from a* 50×50*,* 100×100*,* 200×200 *and a* 400×400 *grid with RK-TVD3 timesteps of* $dt = 0.01, 0.005, 0.0025, 0.00125$ *respectively. As the mesh resolution is increased the shock at* $r \approx 0.73$ *is better resolved.*

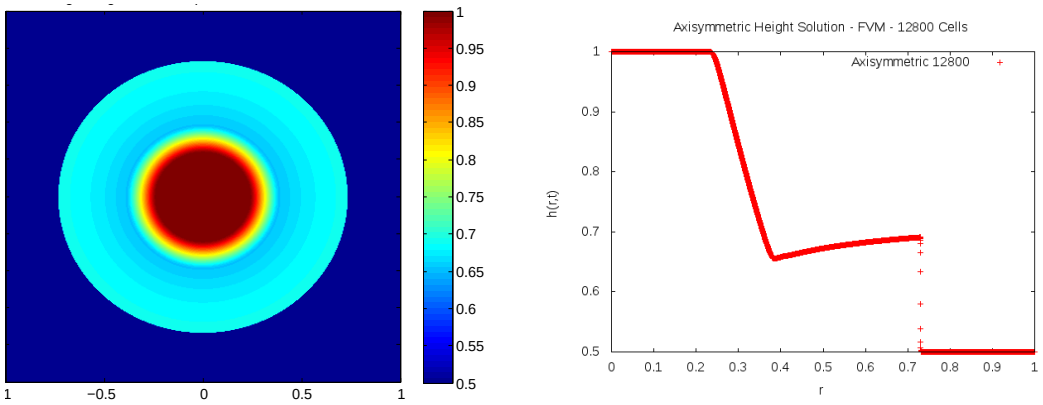

Figure 5.6: *Axisymmetric dam-break - The left hand figure illustrates an axisymmetric dam-break problem solution, $h(r,t)$, mapped into the (x,y) plane. The solution resolution is on a* 12800 *element grid and is deemed a good approximation to the exact 2D dam-break problem solution. The right hand figure illustrates the height as a function of the radius, illustrating a shock at $r \approx 0.73$.*

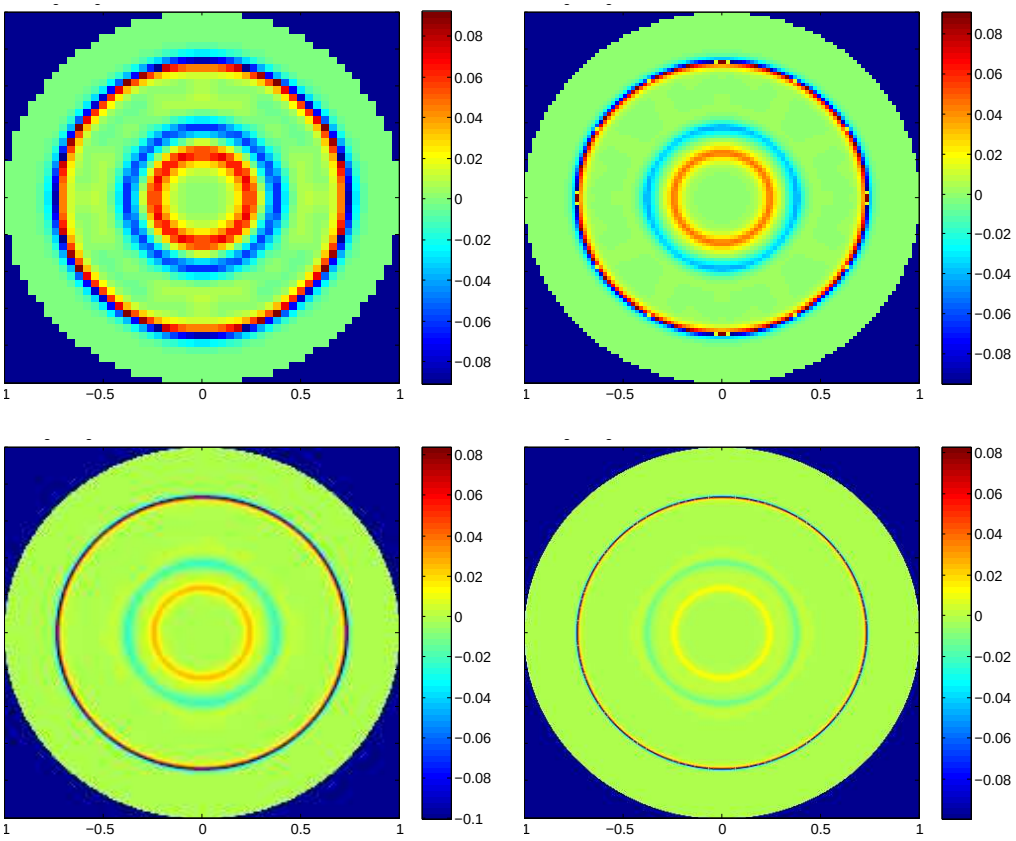

Figure 5.7: *2D dam-break - Density plots of the difference in the average height in each element, at $T = 0.5$, between a finite volume method and discontinuous Galerkin method. The figures represent a 50×50, 100×100, 200×200 and a 400×400 grid with RK-TVD3 timesteps of $dt = 0.01, 0.005, 0.0025, 0.00125$ respectively. The difference near the shock at $r = \sqrt{x^2 + y^2} \approx 0.73$ is large, but the solutions are approximately the same in smooth regions.*

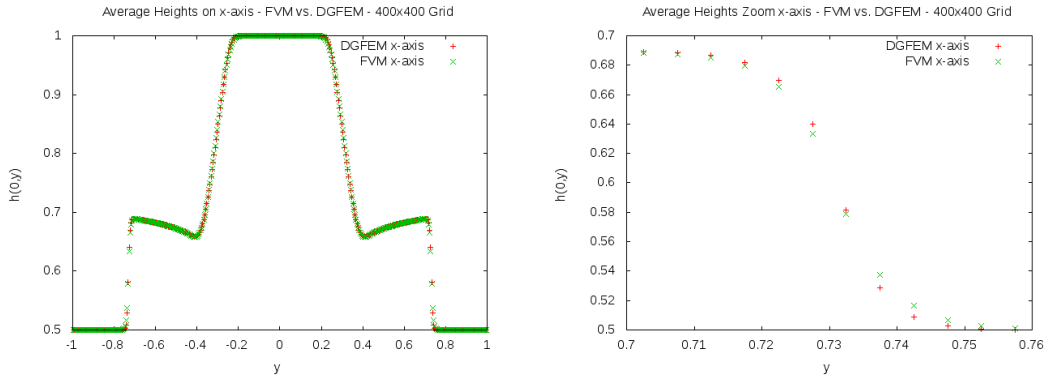

Figure 5.8: *2D dam-break - Illustration of the average DGFEM height solution for the 2D dam-break problem on a* 400×400 *grid with an RK-TVD3 timestep of* $dt = 0.00125$ *along the x-axis. The comparison with the finite volume method shows that the two methods have very similar shock resolutions, from the zoom of the solution* $x \in [0.70 : 0.76]$*, although the DGFEM appears marginally sharper.*

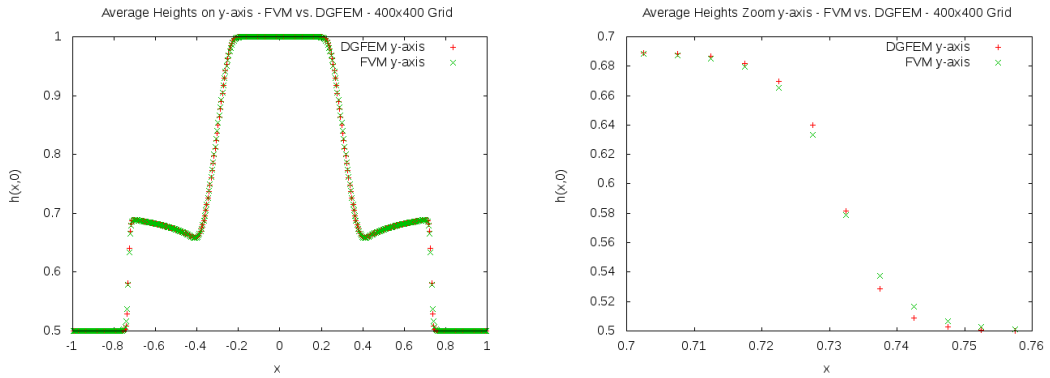

Figure 5.9: *2D Dam-Break - Illustration of the average DGFEM height solution for the 2D dam-break problem on a* 400×400 *grid with an RK-TVD3 timestep of* $dt = 0.00125$ *along the y-axis. The FVM and DGFEM again have very similar shock resolutions.*

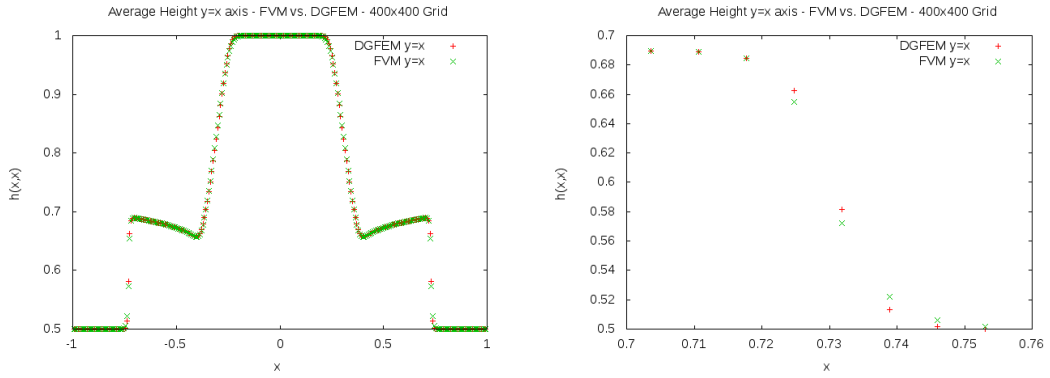

Figure 5.10: *2D Dam-Break - Illustration of the average DGFEM and FVM height solution for the 2D dam-break problem on a* 400×400 *grid with an RK-TVD3 timestep of* $dt = 0.00125$ *along the line* $y = x$*. Again, the DGFEM and FVM have very similar resolutions of the shock.*

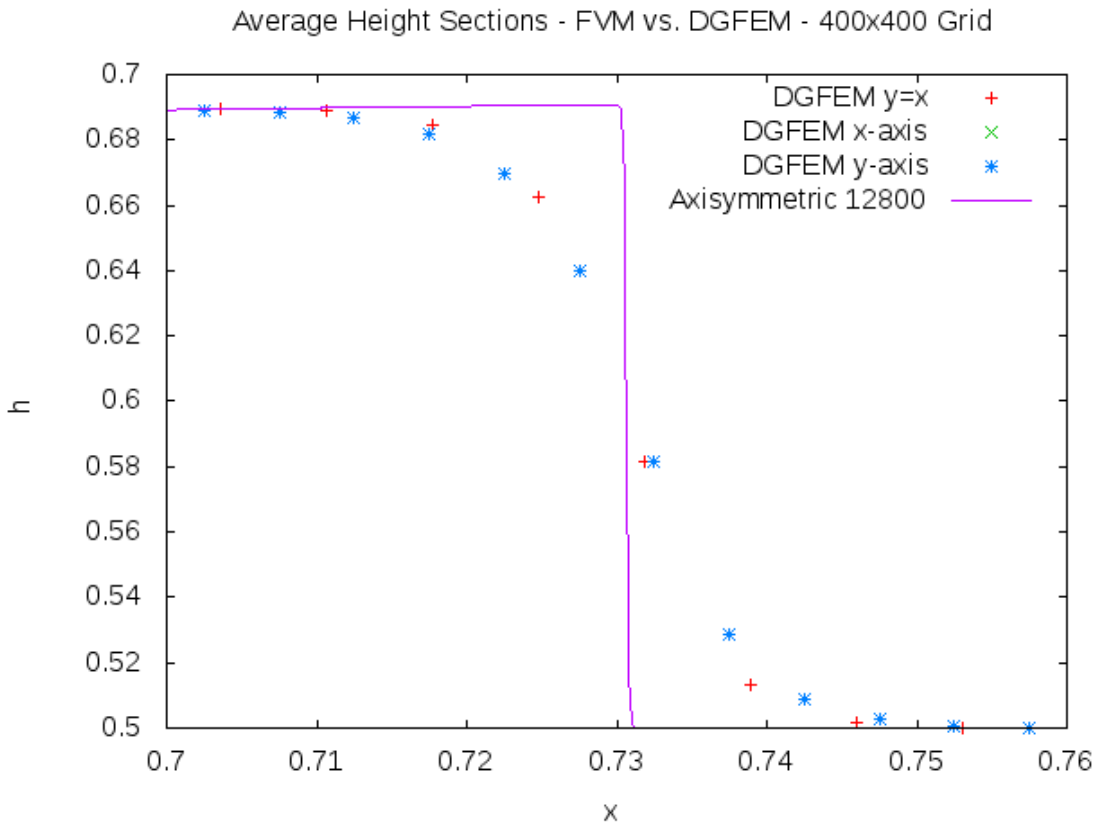

Figure 5.11: *2D Dam-Break - Illustration of the average DGFEM height solution for the 2D dam-break problem on a* 400×400 *grid with an RK-TVD3 timestep of* $dt = 0.00125$ *along the y-axis, x-axis and line* $y = x$*. The figure represents a zoom of the solution with* $r \in [0.70 : 0.76]$*. The FVM axisymmetric high resolution solution is also shown. The three directions show very similar solutions near the shock, as one would expect from a circularly symmetric dam-break problem.*

Chapter 6

Conclusion

A discontinuous Galerkin formulation for a system of conserved PDEs was derived, with a discussion of different types of numerical flux that can be assigned at the edges of adjacent elements. In 1D, an analysis of the total variation diminishing explicit Runge-Kutta class of timesteppers was undertaken to enforce numerical solutions to be bounded in the total variation semi-norm by using MUSCL slope limiting.

Exact solutions were found, using the method of characteristics, for the advection equation. Using the Lax-Friedrichs flux, L^2-errors were computed for a smooth problem and these agreed well with theoretical convergence bounds in this norm. A continuous problem for the 1D shallow water equations was implemented and the Lax-Friedrichs flux was a superior choice to the central flux.

A theoretical discussion of shock formation and propagation was carried out relating to the 1D inviscid Burgers' equation. Numerical simulations, using the Lax-Friedrichs flux at element edges, without slope limiting displayed oscillatory behaviour for discontinuous problems. These oscillations were completely removed when the MUSCL limiter was applied to the numerical solutions.

A discontinuous dam break problem of the shallow water equations was also solved numerically with MUSCL slope limiting. The numerical solutions converged when performing h-refinement at different polynomial orders. However, for a fixed element size, higher order approximation did not decrease the broken L^1-errors. This suggested, for this particular problem, that there there is no advantage in using high order bases in the DGFEM. However, convergence analysis in strictly smooth regions

of the solutions was not undertaken. It also appeared that the Lax-Friedrichs flux was marginally more accurate, and quicker, than the Roe average flux.

A different conserved form for the shallow water equations was derived and used to compute solutions to a continuous problem and the dam-break problem. For the continuous problem, the two forms of the equations led to equivalent results. However, for the dam-break problem, one of the forms was converging to a non physical solution. For problems with shocks, it is essential to enforce conservation of the correct physical variables. So, for the shallow water equations, it is necessary to use a form of the equations that describe the conservation of fluid mass and momentum.

In 2D, a slope limiter applicable to quadrilateral finite elements aligned with the coordinate axes was implemented into the `oomph-lib`. This limiter operates by replacing an average solution gradient in the x and y directions separately, using the modified minmod function. Numerical results for a 2D dam break problem were obtained using the DGFEM, with a Lax-Friedrichs flux, and shocks were being resolved when performing h-refinement. A comparison was made with results from a finite volume method [18] and suggested that at the same spatial resolution both methods were resolving the shock over the same length scale.

6.1 Further Work

In the current implementation, the method of finding neighbouring bulk elements through the face elements in 2D slope limiting is inefficient and could be modified by storing direct pointers to neighbouring bulk elements. The current execution time could also be reduced by using variable timesteps. At each Runge-Kutta timestep, the fastest wave speed in the system could be calculated over the mesh and this used to find a timestep satisfying the CFL condition within a desired tolerance.

For the shallow water equations, a slope limiter for general quadrilateral elements that are not aligned with the coordinate axes could also be devised so that problems over irregular domains could be computed. Also, a modification of the governing equations could be achieved to model flow other bumpy terrains so that more realistic problems for shallow water flow could be investigated.

The 2D slope limiter was devised by essentially performing 1D MUSCL slope

limiting in two orthogonal directions. The 1D MUSCL limiter ensured that numerical solutions were total variation bounded at each explicit timestep and it would be interesting to see if an analogous property could be found for the 2D slope limiter developed in this report. It would also be of interest to construct slope limiter in 2D that operate on pure bilinear and biquadratic approximation.

A further theoretical investigation could be carried out into the 1D shallow water equations with the conserved field $[h, u]^T$. An exact solution for this set of equations could be obtained by using the Rankine Hugoniot conditions at the discontinuity. It would be reassuring to see if the non physical solution obtained in the numerical experiments agreed with such an analytic exact solution.

Appendix A

Shallow Water numerical flux

Lax-Friedrichs constant

In section 2.2 it was stated that the eigenvalues of the matrix $B = \mathbf{n}.\frac{\partial F}{\partial \mathbf{w}}$ must be calculated to find the Lax-Friedrichs flux constant. In component form, this matrix is given by

$$B_{ik} = \frac{\partial F_{ij}}{\partial w_k} n_j. \tag{A.1}$$

Flux Jacobian

The Jacobian of the flux matrix is a rank-3 tensor

$$J_{ijk} = \frac{\partial F_{ij}}{\partial w_k} \quad \text{so} \quad J \in \mathbb{R}^{N_f \times N_p \times N_f}, \tag{A.2}$$

where the indices $j = 0 : N_p - 1$, $i, k = 0 : N_f - 1$, run over the spatial dimensions and fields repectively. The flux for the 2D shallow water equations (5.1) is

$$F = \begin{pmatrix} uh & vh \\ u^2h + \frac{1}{2}Gh^2 & uvh \\ uvh & v^2h + \frac{1}{2}Gh^2 \end{pmatrix},$$

and the unknown field is $\mathbf{w} = [h, hu, hv]^T = [u_0, u_1, u_2]^T$. It is convenient to introduce the notation, $\hat{u}_1 = u_1/u_0$ and $\hat{u}_2 = u_2/u_0$. From the definition of the Jacobian (A.2)

the 18 entries are computed as below:

$$\begin{aligned}
J_{000} = 0.0 \quad J_{001} = 1.0 \quad J_{002} = 0.0 \\
J_{010} = 0.0 \quad J_{011} = 0.0 \quad J_{012} = 1.0 \\
J_{100} = -\hat{u}_1^2 + Gu_0 \quad J_{101} = 2\hat{u}_1 \quad J_{102} = 0.0 \\
J_{110} = -\hat{u}_1\hat{u}_2 \quad J_{111} = \hat{u}_2 \quad J_{112} = \hat{u}_1 \\
J_{200} = -\hat{u}_1\hat{u}_2 \quad J_{201} = \hat{u}_2 \quad J_{202} = \hat{u}_1 \\
J_{210} = -\hat{u}_2^2 + Gu_0 \quad J_{211} = 0.0 \quad J_{212} = 2\hat{u}_2
\end{aligned}$$

Jacobian eigenvalues in normal direction

The definition of the matrix B (A.1) requires the contraction of the tensor above with the outer unit normal $\mathbf{n} = (n_x, n_y)^T$ at an element's edge,

$$B = \begin{pmatrix} 0.0 & n_x & n_y \\ (-\hat{u}_1^2 + Gu_0)n_x - (\hat{u}_1\hat{u}_2)n_y & (2\hat{u}_1)n_x + (\hat{u}_2)n_y & (\hat{u}_1)n_y \\ -(\hat{u}_1\hat{u}_2)n_x + (-\hat{u}_2^2 + Gu_0)n_x & \hat{u}_2 n_x & (\hat{u}_1)n_x + (2\hat{u}_2)n_y \end{pmatrix}.$$

From the solutions of the eigenvalue equation $det(B - \lambda I) = 0$, the eigenvalues are:

$$\lambda_1 = \hat{u}_1 n_x + \hat{u}_2 n_y + \sqrt{Gu_0}, \qquad \lambda_2 = \hat{u}_1 n_x + \hat{u}_2 n_y, \qquad \lambda_3 = \hat{u}_1 n_x + \hat{u}_2 n_y - \sqrt{Gu_0}.$$

The constant in the Lax-Friedrichs flux is taken as the maximum of the three eigenvalues above, over all possibilities of the field at an element boundary. For simplicity, as with the 1D problem, the maximum is taken to be either on the internal or external face, to save solving a potentially expensive optimisation problem.

Appendix B

Numerical Flux Implementation

B.1 Numerical flux at knot points

As was described in section 2.3.1, finite elements in the DGFEM communicate with their neighbours through numerical fluxes defined at the element edges or faces.

The class `DGShallowWaterFaceElement`, describing a face element contains a function that performs numerical integration of a numerical flux function along the edge of two neighbouring elements. To perform this integration, a numerical flux function must be defined at all the integration knot points on the 1D edge of an element.

A function `numerical_flux()` outputs the numerical flux at each of the integration knot points. This function takes as input the outer unit normal, `n_out`, the internal and external fields, `u_int` and `u_ext`, and outputs the numerical flux at a knot point, `flux`. The function `bulk_element_pt()` returns a pointer to a bulk element in the mesh.

```
1  void numerical_flux(const Vector<double> &n_out,
2                      const Vector<double> &u_int,
3                      const Vector<double> &u_ext,
4                      Vector<double> &flux)
5   {
6    //Pointer to a bulk element
7    ELEMENT* cast_bulk_element_pt =
8      dynamic_cast<ELEMENT*>(this->bulk_element_pt());
9
10   //Find spatial dimension and number of fluxes
11   const unsigned dim = this->dimension();
12   const unsigned n_flux = this->required_nflux();
```

```
13
14   //Storage for internal/external fluxes
15   DenseMatrix<double> flux_int(n_flux,dim);
16   DenseMatrix<double> flux_ext(n_flux,dim);
17
18   //Internal and external fluxes at knot point
19   cast_bulk_element_pt->flux(u_int,flux_int);
20   cast_bulk_element_pt->flux(u_ext,flux_ext);
21
22   //Find the flux average over boundary
23   DenseMatrix<double> flux_av(n_flux,dim);
24   for(unsigned i=0;i<n_flux;i++) {
25     for(unsigned j=0;j<dim;j++) {
26       flux_av(i,j) = 0.5*(flux_int(i,j) + flux_ext(i,j));
27     }
28   }
29
30  //Inner product of flux matrix with outer normal
31  flux.initialise(0.0);
32  for(unsigned i=0;i<n_flux;i++) {
33    for(unsigned j=0;j<dim;j++) {
34      flux[i] += flux_av(i,j)*n_out[j];
35    }
36  }
37
38  //Find the field jump over boundary
39  Vector<double> field_jump(n_flux);
40  for(unsigned i=0;i<n_flux;i++)
41    {
42    field_jump[i] = u_int[i] - u_ext[i];
43    }
44
45  //Initialise C_LF and find gravitational cosntant in bulk
46  double C_LF =0.0; const double G = cast_bulk_element_pt->g();
47
48  if(dim == 1) { //1D case - Numerical flux constant
49
50  //Return C_LF by extracting components of u_int and u_ext
51  //(i) Central Flux (C_LF=0)
52  //(ii) Lax-Friedrichs flux
53  //(iii) Roe-Average flux
54
55  } else if(dim==2) { //2D case - Numerical flux constant
56
57  //Return C_LF by extracting components of u_int and u_ext
58  //(i)Lax-Friedrichs flux
59
60  }
61
62  //Final Numerical Flux : flux_av + 0.5*C_LF*field_jump
```

```
63  for(unsigned i=0;i<n_flux;i++) {
64    flux[i] += 0.5*C_LF*field_jump[i];
65  }
66
67 }//End Numerical Flux function
```

B.2 Flux Constant

All the numerical fluxes in this report require a calculation of a flux constant and an if statement on line 48 distinguishes between a 1D and 2D problem. In 1D the constant is calculated through either a central, Lax-Friedrichs or Roe average flux and in 2D the constant is calculated through a Lax-Friedrichs flux. See section 2.2 for the defintions of the constant in each case.

Appendix C

Slope Limiting Implementation

C.1 Limiting over all elements

The `limit_slopes(slope_limiter_pt)` function is a mesh level function that limits the slopes of all the elements in the mesh for a given slope limiter.

This function takes as input a pointer to a slope limiter `slope_limiter_pt`. In the function, all the element averages of the field in the mesh are calculated by looping over the elements. The elements are looped over again and the element level function `slope_limit` is applied to each element (see section C.2.)

The following are used in the code extract:

- `nelement()` - returns the number of elements in the mesh.
- `element_pt` - vector of pointers to elements in the mesh.
- `calculate_average()` - computes the integral average of a field in an element.

```
1  //Inherit new class MYDGMesh from DGMesh
2  class MYDGMesh : public DGMesh
3  {
4  public:
5
6  //Constructor
7  MYDGMesh() : DGMesh() {}
8
9  //Redefine limit_slopes function
10 void limit_slopes(SlopeLimiter* const &slope_limiter_pt);
11 };
12
```

```
13 void MYDGMesh::limit_slopes(SlopeLimiter* const &
      slope_limiter_pt)
14  {
15   //Loop over elements and calculate the averages
16   const unsigned n_element = this->nelement();
17   for(unsigned e=0;e<n_element;e++)
18    {
19     dynamic_cast<DGElement*>(this->element_pt(e))
20      ->calculate_averages();
21    }
22
23   //Loop over again and limit the values
24   for(unsigned e=0;e<n_element;e++)
25    {
26     dynamic_cast<DGElement*>(this->element_pt(e))
27      ->slope_limit(slope_limiter_pt);
28    }
29  }
```

C.2 Limiting over single element

This function will limit the slope in an element by setting up the neighbouring element information. To perform limiting, an element in the mesh must know about its neighbours and a vector of pointers to neighbouring elements, `required_element_pt`, is created for every bulk element.

In each element, one of the pointers to a face is found, the neighbouring face of this face is found and the pointer to this face's corresponding bulk element is returned. This neighbour finding process is repeated for each of the faces in the element. It would be more efficient to store the pointers to neighbouring bulk elements directly, but due to time constraints this was not possible.

All the fluxes are looped over and the `limit` function is called, defined in section C.3. The following are used in the code extract:

- `face_element_pt` is a vector of pointers to each of the faces of an element. The numbering convention is to label the north, east, south and west faces 0, 1, 2 and 3 respectively.

- `required_element_pt` is a vector of pointers to the neighbouring elements in a given element. The numbering convention is to label 0 as the current element

and 1, 2, 3 and 4 as the north, east, south and west neighbours respectively.

- `neighbour_face_pt` is a vector of pointers to the integration knot points on the neighbouring face of a face.
- `bulk_element_pt()` returns a pointer to a bulk element in the mesh.
- `required_nflux()` returns the number of fields for the problem.

```
1  //Redefine slope_limit function
2  template<unsigned NNODE_1D>
3  void DGSpectralShallowWaterElement<2,NNODE_1D>::slope_limit(
       SlopeLimiter* const &slope_limiter_pt)
4  {
5   //Number of fluxes
6   const unsigned n_flux = this->required_nflux();
7
8   //Storage for the element and its neighbours (5 in total)
9   Vector<DGElement*> required_element_pt(5);
10  required_element_pt[0] = this;
11
12  //Get the pointer to NORTH element
13  required_element_pt[1] = dynamic_cast<DGElement*>(
14  dynamic_cast<DGFaceElement*>(this->face_element_pt(0))
15   ->neighbour_face_pt(1)->bulk_element_pt());
16
17  //Get the pointer to EAST element
18  required_element_pt[2] = dynamic_cast<DGElement*>(
19   dynamic_cast<DGFaceElement*>(this->face_element_pt(1))
20   ->neighbour_face_pt(1)->bulk_element_pt());
21
22  //Get the pointer to SOUTH element
23  required_element_pt[3] = dynamic_cast<DGElement*>(
24   dynamic_cast<DGFaceElement*>(this->face_element_pt(2))
25   ->neighbour_face_pt(1)->bulk_element_pt());
26
27  //Get the pointer to WEST element
28  required_element_pt[4] = dynamic_cast<DGElement*>(
29   dynamic_cast<DGFaceElement*>(this->face_element_pt(3))
30   ->neighbour_face_pt(1)->bulk_element_pt());
31
32  //Loop over the fluxes
33  for(unsigned i=0;i<n_flux;i++)
34   {
35    //Call limiter, takes the current element and required
36    //neighbours as the function arguments
37    slope_limiter_pt->limit(i,required_element_pt);
38   }
39 }
```

C.3 Limit function

This function contains the mathematical details of the limiting within a single element. This takes as input `i`, the component of the flux, and `required_element_pt` the vector of pointers to an element's neighbours.

The algorithm to limit the slopes is described in section 5.2.1. The following are used in the code extract:

- `node_pt` is a vector of pointers to the nodes of an element. For linear approximation the components 0, 1, 2, 3 correspond to the bottom left, bottom right, top left and top right nodes of the element.
- `x` is a vector of coordinates at a given node.
- `value` is a vector of field values at a given node.
- `set_value(i,val)` sets the i^{th} field the value val, at a given node.
- `minmodB(G,h)` the modified minmod function described in equation (2.39). G is a vector of the three required gradients and h is the element width.

```
1  class MYMinModLimiter : public MinModLimiter
2  {
3   //Minmod constant for properties near smooth extrema
4   double M;
5
6   ///Boolean flag to indicate a MUSCL limiter
7   bool MUSCL;
8
9   public:
10
11  //Constructor
12  MYMinModLimiter(const double &m=0.0,const bool &muscl=false):
13   MinModLimiter(), M(m), MUSCL(muscl) {}
14
15  ///Redefine limit function
16  void limit(const unsigned &i,
17             const Vector<DGElement*> &required_element_pt);
18 };
19
20 ///Overload limit function - Linear Only
21 void MYMinModLimiter::limit(const unsigned &i,
22            const Vector<DGElement*> &required_element_pt)
23 {
```

```
24 //Pointers to nodes - Bottom Left/Right, Top Left/Right
25 Node* nod_ptbl = required_element_pt[0]->node_pt(0);
26 Node* nod_ptbr = required_element_pt[0]->node_pt(1);
27 Node* nod_pttl = required_element_pt[0]->node_pt(2);
28 Node* nod_pttr = required_element_pt[0]->node_pt(3);
29
30 //Find nodal coordinates
31 Vector<double> x_bl(2), x_tl(2), x_br(2), x_tr(2);
32 x_bl[0] = nod_ptbl->x(0); x_bl[1] = nod_ptbl->x(1);
33 x_br[0] = nod_ptbr->x(0); x_br[1] = nod_ptbr->x(1);
34 x_tl[0] = nod_pttl->x(0); x_tl[1] = nod_pttl->x(1);
35 x_tr[0] = nod_pttr->x(0); x_tr[1] = nod_pttr->x(1);
36
37 //Widths of element
38 const double hx = x_tr[0]-x_tl[0]; //x direction
39 const double hy = x_tr[1]-x_br[1]; //y direction
40
41 //Find nodal unknowns (ith field component)
42 double u_bl, u_br, u_tl, u_tr;
43 u_bl = nod_ptbl->value(i); u_br = nod_ptbr->value(i);
44 u_tr = nod_pttr->value(i); u_tl = nod_pttl->value(i);
45
46 //Find gradients across top and bottom
47 double u_xB = (u_br-u_bl)/hx; double u_xT = (u_tr-u_tl)/hx;
48 //Find gradient along left and right
49 double u_yL = (u_tl-u_bl)/hy; double u_yR = (u_tr-u_br)/hy;
50
51 //Find average gradients in x/y direction
52 double u_xapp = 0.5*(u_xB + u_xT);
53 double u_yapp = 0.5*(u_yL + u_yR);
54
55 //Average values of elements (This,Top,Right,Bottom,Left)
56 const double u_av =required_element_pt[0]->average_value(i);
57 const double u_avT =required_element_pt[1]->average_value(i);
58 const double u_avR =required_element_pt[2]->average_value(i);
59 const double u_avB =required_element_pt[3]->average_value(i);
60 const double u_avL =required_element_pt[4]->average_value(i);
61
62 //Gradient adjustment for MUSCL or alternative limiter
63 double gradient_factor = 0.5;
64 if(MUSCL) {gradient_factor = 1.0;}
65
66 //Average gradients in y-direction
67 double u_yT = (u_avT - u_av)/(hy*gradient_factor);
68 double u_yB = (u_av - u_avB)/(hy*gradient_factor);
69
70 //Average gradients in x-direction
71 double u_xL = (u_av - u_avL)/(hx*gradient_factor);
72 double u_xR = (u_avR - u_av)/(hx*gradient_factor);
73
```

```
74  //Storage for gradients to minmod functions
75  Vector<double> argx(3), argy(3);
76
77  //Minmod function for the x and y direction
78  argx[0] = u_xapp; argx[1] = u_xL; argx[2] = u_xR;
79  argy[0] = u_yapp; argy[1] = u_yT; argy[2] = u_yB;
80
81  //Limited gradients through minmod function
82  double u_xN = this->minmodB(argx,hx);
83  double u_yN = this->minmodB(argy,hy);
84
85  //Reconstruct element - element has lost "twist" term
86  //and average (centre) point the same by conservation
87  u_bl = (u_av - u_xN*(hx/2.0) - u_yN*(hy/2.0));
88  u_br = (u_av + u_xN*(hx/2.0) - u_yN*(hy/2.0));
89  u_tl = (u_av - u_xN*(hx/2.0) + u_yN*(hy/2.0));
90  u_tr = (u_av + u_xN*(hx/2.0) + u_yN*(hy/2.0));
91
92  //Set values at the nodal positions
93  required_element_pt[0]->node_pt(0)->set_value(i,u_bl);
94  required_element_pt[0]->node_pt(1)->set_value(i,u_br);
95  required_element_pt[0]->node_pt(2)->set_value(i,u_tl);
96  required_element_pt[0]->node_pt(3)->set_value(i,u_tr);
97
98 }
```

Appendix D

RK-TVD3 Implementation

The annotated code describes the algorithm to perform the RK-TVD3 timestepping, where the degrees of freedom, the unknowns, $\mathbf{u}$, are stored in the pointer `object_pt` and `dt` is the timestep.

The algorithm for the RK-TVD3 timestepping can be seen in (2.29). In the code below the function `get_inverse_mass_matrix_times_residual(k)` is a function that performs the operation $\mathbf{L}_h$ (in the algorithm (2.29)) on the current unknowns $\mathbf{u}$ in `object_pt` and stores this on a new vector $\mathbf{k}$. The function `set_dofs(u)` sets the current unknowns in `object_pt` to $\mathbf{u}$ and `add_dofs(u)` adds $\mathbf{u}$ to the current unknowns in `object_pt`.

It is essential to call the function `actions_after_explicit_timestep()` on the unknowns after each explicit timestep of the Runge-Kutta algorithm. The slope limiter is applied to the unknowns at this point to ensure the numerical solution is total variation bounded in the means.

```
1  template<>
2  void RungeKutta<3>::timestep(ExplicitTimeSteppableObject*
       const &object_pt, const double &dt)
3  {
4   //Store initial values, u, and add to object
5   DoubleVector u;
6   object_pt->get_dofs(u);
7
8   //Storage for unknown k1 = M^{-1}*R(u0) = L_{h}(u0)
9   DoubleVector k1;
10  object_pt->get_inverse_mass_matrix_times_residuals(k1);
11
12  //u1 = u0+dt*M{-1}*R(u0)
```

```
13  object_pt->add_to_dofs(dt,k1);
14  //Slope Limit after explicit step
15  object_pt->actions_after_explicit_timestep();
16
17  //Storage for unknown k2 = M^{-1}*R(u1) = L_{h}(u1)
18  DoubleVector k2;
19  object_pt->get_inverse_mass_matrix_times_residuals(k2);
20
21  //Reset dofs
22  object_pt->set_dofs(u);
23  //u2 = u0 + 0.25*dt*M^{-1}*R(u0) + 0.25*dt*M{-1}*R(u1)
24  object_pt->add_to_dofs(0.25*dt,k1);
25  object_pt->add_to_dofs(0.25*dt,k2);
26  //Slope Limit after explicit step
27  object_pt->actions_after_explicit_timestep():
28
29  //Storage for unknown k3 = M{-1}*R(u2) = L_{h}(u2)
30  DoubleVector k3;
31  object_pt->get_inverse_mass_matrix_times_residuals(k3);
32
33  object_pt->time() += dt; //Increment time
34  //Reset dofs, construct final vectot and limit
35  //u_n+1 = u_n + dt*k1/6 + dt*k2/6 + 2*dt*k3/3
36  object_pt->set_dofs(u);
37  object_pt->add_to_dofs((dt/6.0),k1);
38  object_pt->add_to_dofs((dt/6.0),k2);
39  object_pt->add_to_dofs((2.0*dt/3.0),k3);
40  //Slope Limit after explicit step
41  object_pt->actions_after_explicit_timestep();
42 }
```

Appendix E

1D Shallow Water Code

An original C++ code to solve the continuous shallow water in one spatial dimension is described below. A new class is written that describes a 1D shallow water element, `WaveElement`.

The class has functions that return the coordinate and unknowns in an element at a given local coordinate, a function that takes as input the unknown field and outputs the flux, a function that uses a 2 point Gauss quadrature rule to numerically integrate the flux function along an element and a function to return the numerical flux by calculating the Lax-Friedrichs constant.

Linear approximation is used with the shape functions,

$$\psi_0(s) = \frac{1-s}{2}, \qquad \psi_1(s) = \frac{1+s}{2}, \tag{E.1}$$

where s is the local elemental coordinate. For this choice of basis, the mass matrix is not diagonal but is also not ill conditioned since the basis is only of first order. Assuming equally spaced elements of width dx, the mass matrix and its inverse are:

$$M = \begin{pmatrix} \frac{dx}{3} & \frac{dx}{6} \\ \\ \frac{dx}{6} & \frac{dx}{3} \end{pmatrix} \implies M^{-1} = \begin{pmatrix} \frac{4}{dx} & \frac{-2}{dx} \\ \\ \frac{-2}{dx} & \frac{4}{dx} \end{pmatrix}. \tag{E.2}$$

A function `timestep(dt)` performs Euler timestepping of the nodal values in the element. Further functions to output the numerical solution and calculate the L^2 error are also included. The code for the `WaveElement` class can be seen below:

```
1 #include<iostream>
2 #include<fstream>
3 #include<vector>
```

```
4  #include<cmath>
5
6  using namespace std;
7
8  //Number of fields, approximation order and nodes
9  const unsigned Nfield=2; const unsigned Napprox=2;
10 const unsigned Nnode=Nfield*Napprox; const unsigned NLaxFre=4;
11
12 //Set the constant G, H, 2*pi and DomainWidth
13 const double G=10.0; const double H=25.0;
14 const double two_pi=8.0*atan(1.0); const double DomWidth=10.0;
15
16 //Number elements, steps and step size
17 const unsigned N_element=160; const unsigned step_no=20000;
      const double dtime=0.0005;
18
19 //Number of plot points and plotting frequency
20 const unsigned inter=10; double stepplot=step_no;
21
22 //exact(x,t) - returns exact solution (u,h)' at a point (x,t)
23 vector<double> exact(double x, double t) {
24   vector<double> VecSize(Nfield);
25   double eta = (x + 2.0*sqrt(G*H)*t)/(1.0 + 3.0*sqrt(G)*t);
26   VecSize[0] = pow(eta,2); //Height
27   VecSize[1] = 2.0*sqrt(G)*(eta - sqrt(H)); //Velocity
28   return VecSize;
29 }
30
31 //exactderiv(x,t,dt) - approx derivative of exact solution
32 vector<double> exactderiv(double x, double t, double dt){
33   vector<double> VecSize(Nfield);
34   for(unsigned i=0; i<Nfield; i++) {
35     VecSize[i] = (exact(x,t+dt)[i] - exact(x,t)[i])/dt;}
36   return VecSize;
37 }
38
39 class WaveElement {
40
41 public:
42
43 //Pointer to neighbours of each WaveElement
44 WaveElement *Left_neigh_pt, *Right_neigh_pt;
45
46 //Storage for X (coords), U (unknowns) and Utemp
47 vector<double> X, U, Utemp;
48
49 //Initialise the coordinate and unknowns within element
50 WaveElement() {
51  X.resize(Napprox,0.0);
52  U.resize(Nnode,0.0);  Utemp.resize(Nnode,0.0);
```

```
 Left_neigh_pt = 0;  Right_neigh_pt = 0;
}

//shape(s, psi) function for linear interpolation
void shape(double s, vector<double> & psi) {
 psi.resize(Napprox,0.0);
 psi[0] = 0.5*(1.0-s);  psi[1] = 0.5*(1.0+s);
} //End shape function

//inter_x(s) - returns coordianate X at local coordinate s
double inter_x(double s) {
 vector<double> psi;
 this->shape(s,psi);
 double inter = X[0]*psi[0] + X[1]*psi[1];
 return inter;
} //End inter_x function

//inter_u(s) - returns unknowns U at local coordinate s
vector<double> inter_u(double s) {
 vector<double> VecSize(Nfield);
 vector<double> psi;
 this->shape(s,psi);
 VecSize[0] = U[0]*psi[0] + U[1]*psi[1]; //Height
 VecSize[1] = U[2]*psi[0] + U[3]*psi[1]; //Velocity
 return VecSize;
} //End inter_u function

//flux(u) - returns (u[0]*u[1] , G*u[0]+0.5*u[1]*u[1])
virtual vector<double> flux(vector<double> & u) {
 vector<double> VecSize(Nfield);
 VecSize[0] = u[0]*u[1]; //Height
 VecSize[1] = G*u[0] + 0.5*u[1]*u[1]; //Velocity
 return VecSize;
} //End flux function

//integ_flux() - (2 point Gauss ruler in each element)
vector<double> integ_flux() {
 vector<double> gauss1(Nfield),gauss2(Nfield);
 vector<double> VecSize(Nfield);
 gauss1=inter_u(sqrt(1.0/3.0));gauss2=inter_u(-sqrt(1.0/3.0));
 for(unsigned i=0; i<Nfield; i++) {
  VecSize[i] = flux(gauss1)[i] + flux(gauss2)[i];}
 return VecSize;
} //End integ_flux function

//maxvec(u) - returns absolute maximum entry of vector
double maxvec(vector<double> & u) {
 double Maximum = abs(u[0]);
  for(unsigned i=0; i<NLaxFre; i++) {
   if (Maximum < abs(u[i]) ){ Maximum = abs(u[i]); }
```

```
103  return Maximum;
104 } //End maxvec function
105
106 //num_flux(a,b) - numerical flux function
107 virtual vector<double> num_flux(vector<double> & a, vector<
       double> & b) {
108  vector<double> VecSize(Nfield), ua(Nfield), ub(Nfield);
109  vector<double> fluxa(Nfield), fluxb(Nfield);
110  ua[0] = a[1]; ua[1] = a[3]; //Left field
111  ub[0] = b[0]; ub[1] = b[2]; //Right field
112  fluxa = flux(ua); fluxb = flux(ub); //Left/Right flux
113
114  vector<double> LaxFre(NLaxFre); //Possible constants
115  LaxFre[0] = ua[1] + sqrt(abs(G*ua[0]));
116  LaxFre[1] = ua[1] - sqrt(abs(G*ua[0]));
117  LaxFre[2] = ub[1] + sqrt(abs(G*ub[0]));
118  LaxFre[3] = ub[1] - sqrt(abs(G*ub[0]));
119
120  double LaxFreCons = maxvec(LaxFre); //Lax-Friedrichs constant
121  for(unsigned i=0; i<Nfield; i++) {
122  VecSize[i]=0.5*(fluxa[i]+fluxb[i]-LaxFreCons*(ub[i]-ua[i]));}
123
124  return VecSize; //Numerical flux
125 } //End num_flux function
126
127
128 void timestep(double dt, double time) { //Euler timestep
129  double dx = X[1] - X[0]; //Element width dx
130  //Mass Matrix (Inverse only) as vector of vectors
131  vector<vector<double> > Minv(Nnode);
132  for(unsigned i=0; i<Nnode; i++) { Minv[i].resize(Nnode); }
133  Minv[0][0] = 4.0/dx; Minv[0][1] = -2.0/dx;
134  Minv[0][2] = 0.0; Minv[0][3] = 0.0;
135  Minv[1][0] = -2.0/dx; Minv[1][1] = 4.0/dx;
136  Minv[1][2] = 0.0; Minv[1][3] = 0.0;
137  Minv[2][0] = 0.0; Minv[2][1] = 0.0;
138  Minv[2][2] = 4.0/dx; Minv[2][3] = -2.0/dx;
139  Minv[3][0] = 0.0; Minv[3][1] = 0.0;
140  Minv[3][2] = -2.0/dx; Minv[3][3] = 4.0/dx;
141
142  //Form residual - integrated flux/numerical flux
143  vector<double> int_f = this->integ_flux();
144  vector<double> F(Nnode);
145  F[0] = -0.5*int_f[0];
146  F[1] = 0.5*int_f[0];
147  F[2] = -0.5*int_f[1];
148  F[3] = 0.5*int_f[1];
149
150  vector<double> H(Nnode), Deriv(Nnode);
151  if (Left_neigh_pt == 0) {//Left element
```

```
152   H[1] = -num_flux(U,Right_neigh_pt->U)[0];
153   H[3] = -num_flux(U,Right_neigh_pt->U)[1];
154   H[0] = H[2] = 0.0;
155   Deriv[1] = exactderiv(0.0,time,dt)[0];
156   Deriv[3] = exactderiv(0.0,time,dt)[1];
157   Deriv[0] = Deriv[2] = 0.0;
158   Utemp[1] = U[1] + dt*(F[1]+H[1]-dx/6.0*Deriv[1])*3.0/dx;
159   Utemp[3] = U[3] + dt*(F[3]+H[3]-dx/6.0*Deriv[3])*3.0/dx;
160   Utemp[0] = Utemp[2] = 0.0;
161  } else if (Right_neigh_pt == 0) {//Right element
162   H[0] = num_flux(Left_neigh_pt->U,U)[0];
163   H[2] = num_flux(Left_neigh_pt->U,U)[1]; H[1] = H[3] = 0.0;
164   Deriv[0] = exactderiv(DomWidth,time,dt)[0];
165   Deriv[2] = exactderiv(DomWidth,time,dt)[1];
166   Deriv[1] = Deriv[3] = 0.0;
167   Utemp[0] = U[0] + dt*(F[0]+H[0]-dx/6.0*Deriv[0])*3.0/dx;
168   Utemp[2] = U[2] + dt*(F[2]+H[2]-dx/6.0*Deriv[2])*3.0/dx;
169   Utemp[1] = Utemp[3] = 0.0;
170  } else { //Central elements - solve left/right nodes
171   H[0] = num_flux(Left_neigh_pt->U,U)[0];
172   H[1] = -num_flux(U,Right_neigh_pt->U)[0];
173   H[2] = num_flux(Left_neigh_pt->U,U)[1];
174   H[3] = -num_flux(U,Right_neigh_pt->U)[1];
175
176   //Compute product Minv*(F+H) and store as vector p
177   vector<double> p(Nnode,0.0);
178   for(unsigned k=0; k<Nnode; k++) {
179    for(unsigned j=0; j<Nnode; j++) {
180     p[k] = p[k] + Minv[k][j]*(F[j]+H[j]);
181    }
182   }
183
184   //Perform the Euler timestep
185   for(unsigned j=0; j<Nnode; j++) {
186    Utemp[j] = U[j] + dt*p[j];
187   }
188  } //End if statement for Numerical Flux
189 } //End timestep function
190
191
192 //update function to put temporary to true storage
193 void update() {
194   for(unsigned j=0; j<Nnode; j++) {
195     U[j] = Utemp[j];
196   }
197 } //End Update function
198
199
200 //Coords function to print solution across element
201 void coords(unsigned n_plot, double totaltime) {
```

```
202  for(unsigned n=0; n<n_plot; n++) {
203  double s = -1.0 + n*(2.0)/((double)n_plot-1);
204
205  //True field at local coordinate s
206  vector<double> TrueField = exact(inter_x(s),totaltime);
207
208  //Output, at coordinate x, the approximate and true fields
209  cout << inter_x(s) << " " << inter_u(s)[0] << " " <<
210  inter_u(s)[1]  << " " << TrueField[0] << " " << TrueField[1]
211  << " " << abs(pow((TrueField[0] - inter_u(s)[0]),1)) << " "
212  << abs(pow((TrueField[1] - inter_u(s)[1]),1)) << "\n";
213   }
214
215} //End Coords function
216
217//L2Error(totaltime) - returns L2error for each field
218vector<double> L2Error(double totaltime) {
219
220//Approximate/true solutions at Gauss points
221double GaussP = sqrt(3.0/5.0); //Symmetric local Gauss 3 point
222vector<double> TrueGauss1 = exact(inter_x(GaussP),totaltime);
223vector<double> ApprGauss1 = inter_u(GaussP);
224vector<double> TrueGauss2 = exact(inter_x(-GaussP),totaltime);
225vector<double> ApprGauss2 = inter_u(-GaussP);
226vector<double> TrueGauss3 = exact(inter_x(0.0),totaltime);
227vector<double> ApprGauss3 = inter_u(0.0);
228
229double dx = X[1] - X[0]; //Constant element width dx
230vector<double> VecSize(Nfield);
231for(unsigned i=0; i<Nfield; i++) {
232VecSize[i] =
233   dx/2.0*( 5.0/9.0*pow((TrueGauss1[i]-ApprGauss1[i]),2)
234   + 5.0/9.0*pow((TrueGauss2[i]-ApprGauss2[i]),2)
235   + 8.0/9.0*pow((TrueGauss3[i]-ApprGauss3[i]),2) );
236}
237
238return VecSize;
239} //End L2Error function
240
241}; //End WaveElement class
```

The main driver code sets up a vector of pointers, `Element_pt`, whose components are members of the `WaveElement` class. The neighbouring information is set up by assigning a pointer to the left element and a pointer to the right element for each component of `Element_pt` so that can numerical fluxes can be assigned at element boundaries.

After setting the initial condition for both the height and velocity field in each

element, the code performs explicit Euler timestepping for each of the elements, using the function `timestep(dt)` written in the `WaveElement` class.

The code outputs the broken L^2-errors of both the height and velocity solution at a given time and the approximate solution across the domain. The driver code can be seen below:

```
#include<iostream>
#include<fstream>
#include<vector>
#include<cmath>

#include"Wave1DClass.cc"

using namespace std;

main(){
 //Pointers to elements
 vector<WaveElement*> Element_pt(N_element);
 for(unsigned e=0; e<N_element; e++) {
  Element_pt[e] = new WaveElement;
 }

 //Set up neighbouring elements (ignore end elements)
 for(unsigned e=1; e<N_element-1; e++) {
  Element_pt[e]->Left_neigh_pt = Element_pt[e-1];
  Element_pt[e]->Right_neigh_pt = Element_pt[e+1];
 }

 //First/last element only have single neighbour
 Element_pt[0]->Right_neigh_pt = Element_pt[1];
 Element_pt[N_element-1]->Left_neigh_pt = Element_pt[N_element
    -2];

 //Loop over elements, set X coords and intialise U and H at t
    =0
 for(unsigned e=0;e<N_element;e++) {
  //Set left and right x coordinates 0 < x < DomWidth
  Element_pt[e]->X[0] = DomWidth*e/((double)N_element);
  Element_pt[e]->X[1] = DomWidth*(e+1)/((double)N_element);
  //Unknown U has order of fields (ho,h1,u0,u1)'
  Element_pt[e]->U[0] = exact(Element_pt[e]->X[0],0.0)[0]; //
     Height
  Element_pt[e]->U[1] = exact(Element_pt[e]->X[1],0.0)[0];
  Element_pt[e]->U[2] = exact(Element_pt[e]->X[0],0.0)[1]; //
     Velocity
  Element_pt[e]->U[3] = exact(Element_pt[e]->X[1],0.0)[1];
 }

 //Output solution across domain at t=0
```

```
40  for(unsigned e=0;e<N_element;e++) {
41   Element_pt[e]->coords(inter,0.0);
42  }
43  cout << "\n\n"; unsigned count=0; //Start counter
44
45  //Perform the timesteps
46  for(unsigned i=1; i<(step_no+1); i++) {
47   ++count; //Increment counter
48   //Loop elements - Perform timestep on unknowns
49   for(unsigned e=0;e<N_element;e++) {
50          Element_pt[e]->timestep(dtime, double(i)*dtime); }
51
52   //Loop elements - Update values of unknowns
53   for(unsigned e=0;e<N_element;e++) {
54         Element_pt[e]->update(); }
55
56   //Update BCs at end points for Field and Flux
57   Element_pt[0]->U[0] =
58   exact(Element_pt[0]->X[0],double(i)*dtime)[0];
59   Element_pt[0]->U[2] =
60   exact(Element_pt[0]->X[0],double(i)*dtime)[1];
61   Element_pt[N_element-1]->U[1] =
62   exact(Element_pt[N_element-1]->X[1],double(i)*dtime)[0];
63   Element_pt[N_element-1]->U[3] =
64   exact(Element_pt[N_element-1]->X[1],double(i)*dtime)[1];
65
66   //Output solution across domain and calulate L2 error
67   if(count==stepplot) {
68    count = 0; //Reset counter for next plot
69    double SumHL2err = 0.0; double SumUL2err = 0.0; //L2errors
70    for(unsigned e=0;e<N_element;e++) {
71     //Output the approximate solution with coords function
72           Element_pt[e]->coords(inter,double(i)*dtime);
73     SumHL2err = SumHL2err
74     + Element_pt[e]->L2Error(double(i)*dtime)[0]; //Height L2
75     SumUL2err = SumUL2err
76     + Element_pt[e]->L2Error(double(i)*dtime)[1]; //Vel L2
77     }
78     SumHL2err = sqrt(SumHL2err); SumUL2err = sqrt(SumUL2err);
79     cout << "\n\n" << "h L2-Err = " << SumHL2err <<
80      " " << "u L2-Err = " << SumUL2err << "\n\n";
81
82    } //End plotting step
83  } //End timestep
84 } //End Main
```

Bibliography

[1] J.D. Logan. *An Introduction to Nonlinear Partial Differential Equations (2nd Edition)*. John Wiley and Sons Inc., Hoboken, New Jersey, 2008.

[2] R.J. LeVeque. *Finite-Volume Methods for Hyperbolic Problems*. Cambridge University Press, Cambridge, 2002.

[3] J.S. Hesthaven and T. Warburton. *Nodal Discontinuous Galerkin Methods - Algorithms, Analysis and Applications*. Springer, New York, 2008.

[4] G.E. Karniadakis and S.J. Sherwin. *Spectral/hp Element Methods for Computational Fluid Dynamics*. Oxford University Press, Oxford, 2005.

[5] G-S Jiang and E. Tadmor. Nonoscillatory Central Schemes for Multidimensional Hyperbolic Conservation Laws. *SIAM J. Sci. Comput.*, 19(6):1892–1917, 1998.

[6] A. Hazel and M. Heil. OOMPH-LIB. `http://oomph-lib.maths.man.ac.uk/doc/html/index.html`.

[7] B. Cockburn and C-W. Shu. Runge-Kutta Discontinuous Galerkin Methods for Convection Dominated Problems. *J. Sci. Comput.*, 16(3):173–261, 2001.

[8] B.Q. Li. *Discontinuous Finite Element Methods in Fluid Dynamics and Heat Transfer*. Springer-Verlag, London, 2008.

[9] P.L. Roe. Approximate Riemann Solvers, Parameter Vectors and Difference Schemes. *J. Comput. Phys.*, 43:357–372, 1981.

[10] P. Glaister. A Weak Formulation of Roe's Approximate Riemann Solver to the St. Venant Equations. *J. Comput. Phys.*, 116:189–191, 1995.

[11] S. Gottlieb and C-W. Shu. Total Variation Diminishing Runge-Kutta Schemes. *Math. Comp.*, 67(221):73–85, 1998.

[12] S.J. Bence, M.P. Hobson, and K.F. Riley. *Mathematical Methods for Physics and Engineering*. Cambridge University Press, Cambridge, Second edition, 2008.

[13] B. Van Leer. Towards the ultimate conservation difference scheme V. *J. Comput. Phys.*, 32:1–136, 1979.

[14] C-W Shu. TVB uniformly high order schemes for conservation laws. *Math. Comp.*, 49:105–121, 1987.

[15] H. Elman, D. Silvester, and A. Wathen. *Finite Elements and Fast Iterative Solvers: with Applications in Incompressible Fluid Dynamics (2nd Edition)*. Oxford University Press, Oxford, 2006.

[16] G. Huang, C. Wu, and Y. Zheng. Theoretical Solution of Dam-Break Shock Wave. *J. Hydraul. Eng.*, 125(11):1210–1215, 1999.

[17] D. J. Acheson. *Elementary Fluid Dynamics*. Oxford University Press, Oxford, 1990.

[18] C. Johnson. Personal communication, 2010.