An Implicit-Explicit Discontinuous Galerkin Scheme using a Newton-Free Picard Iteration for a Thin-Film Model

by

Caleb Logemann

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Applied Mathematics

Program of Study Committee:
James Rossmanith, Major Professor
Alric Rothmayer
Jue Yan

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation/thesis. The Graduate College will ensure this dissertation/thesis is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University

Ames, Iowa

2019

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This paper provides a high-order numerical scheme for solving thin-film models of the form $q_t + \left(q^2 - q^3\right)_x = -\left(q^3 q_{xxx}\right)_x$. The second term in this equation is of nonlinear hyperbolic type, while the right-hand side is of nonlinear parabolic type. The nonlinear hyperbolic term is discretized with the standard modal discontinuous Galerkin method, and the nonlinear parabolic term is discretized with the local discontinuous Galerkin method. Propagation in time is done with an implicit-explicit Runge-Kutta scheme so as to allow for larger time steps. The timestep restriction for these methods is determined by the hyperbolic wavespeed restriction, and is not limited by the nonlinear parabolic term. A novel aspect of this method is that the resulting nonlinear algebraic equations are solved with a Newton-free iteration with a Picard iteration. The number of iterations required to converge is less than or equal to the estimated order of the method. We have demonstrated with this method up to third order convergence.

CHAPTER 1. Introduction

In this paper we look at the model equation,

$$q_t + \left(q^2 - q^3\right)_x = -\left(q^3 q_{xxx}\right)_x \quad (x,t) \in [a,b] \times [0,T]. \tag{1.1}$$

This equation describes the motion of a thin film of liquid flowing over a one-dimensional domain, $[a,b]$, where $q(x,t) \geq 0$ is the height of the liquid. This fluid is acted upon by gravity, by forces on the surface, and by surface tension. The surface forces can have many different causes, including wind shear, thermocapillary forces, or molecular forces. In all cases an equivalent model can be derived. This model is useful in many different applications including airplane de-icing [19, 20] and industrial coating. Some experimental study [5, 14, 18] has been done and numerical results have shown good agreement with those experiments in [3].

Previous numerical methods for this type of equation have focused on finite difference approaches. Bertozzi and Brenner [2] used a fully implicit centered finite difference scheme to explore instabilities. Ha et al. [13] explored several different finite difference schemes, some fully implicit and some using the Crank-Nicolson method. In their analysis, they considered several different methods for the hyperbolic terms including WENO, Godunov, and an adapted upwind method. All of these methods were limited to just first or second order, and they required solving a Newton iteration. Finite difference methods also lack provable stability.

We chose to use discontinuous Galerkin methods as they allow for high order convergence. The discontinuous Galerkin methods were first introduced by Reed and Hill [25], and then were formalized by Cockburn and Shu in a series of papers [9, 8, 7, 6, 11]. We use the original modal discontinuous Galerkin method as well as the local discontinuous Galerkin method. The local discontinuous Galerkin method was also formulated by Cockburn and Shu [10] to handle convection-diffusion equations with the discontinuous Galerkin method. We use the modal discontinuous

Galerkin method to discretize the convection term, $\left(q^2 - q^3\right)_x$, and the local discontinuous Galerkin method to discretize the diffusion term, $-\left(q^3 q_{xxx}\right)_x$.

The nonlinear diffusion is much stiffer, as it has an infinite wavespeed, than the hyperbolic convection, which only has a finite wavespeed. If the diffusion is handled explicitly in time, then a very strict time step restriction is needed to insure stability. Therefore the nonlinear diffusion should be solved implicitly. The whole system could be solved implicitly, however we chose to use an implicit-explicit (IMEX) Runge-Kutta scheme for propagating in time. These schemes were first introduced by Ascher et al. [1] and have been expanded on by Kennedy and Carpenter [15] and Pareschi and Russo [23]. The IMEX Runge-Kutta schemes allows us not only to solve the nonlinear diffusion implicitly, but it also allows for the nonlinear convection to be propagated explicitly. Solving the convection explicitly fully captures the nonlinear behavior without a nonlinear solve.

The diffusion still requires a nonlinear solve, but this is simpler than solving both convection and diffusion nonlinearly. We solve the nonlinear system using a Picard iteration, first introduced by Émile Picard and then formalized by Lindelöf [16]. We find that very minimal number of iterations is required for the iteration to converge. The number of iterations is less than or equal to the overall order of accuracy of the method. With these methods we have demonstrated up to third order accuracy with a manufactured solution example. We also demonstrate the nonlinear traveling wave behavior through several numerical examples introduced by Bertozzi [4]. These examples demonstrate that the steady state traveling wave may not be unique. They also show a double shock structure with an undercompressive shock.

CHAPTER 2. The Thin-Film Model

In this model we consider a thin film of liquid on a flat surface with a free interface. This liquid is driven by gravity, shear and normal forces on the surface, and surface tension (see Figure 2.1). We begin by considering the two dimensional incompressible Navier-Stokes equations, which have the form,

$$u_x + w_z = 0, \tag{2.1}$$

$$\rho(u_t + uu_x + wu_z) = -p_x + \mu\Delta u - \phi_x, \tag{2.2}$$

$$\rho(w_t + uw_x + ww_z) = -p_z + \mu\Delta w - \phi_z, \tag{2.3}$$

where ρ is the density, u is the horizontal velocity, w is the vertical velocity, p is the pressure, and ϕ is the force of gravity. Equation (2.1) is the incompressibility condition and also represents conservation of mass. Equations (2.2) and (2.3) represent the conservation of momentum in the x and z directions respectively. We take a no penetration and no slip boundary condition at the lower boundary and the kinematic boundary condition at the upper boundary. These boundary conditions can be expressed as follows,

$$w = 0, u = 0, \qquad \text{at } z = 0, \tag{2.4}$$

$$w = h_t + uh_x, \qquad \text{at } z = h, \tag{2.5}$$

where h is the height of the liquid. We can also describe the stress tensor, $\mathbf{T}$, at the free surface, $z = h$, as

$$\mathbf{T} \cdot \mathbf{n} = (-\kappa\sigma + \Pi_0)\mathbf{n} + \left(\frac{\partial\sigma}{\partial s} + \tau_0\right)\mathbf{t}, \qquad \text{at } z = h,$$

where κ is the mean curvature, σ is the surface tension, and Π_0 and τ_0 are the normal and tangential components of the forcing respectively.

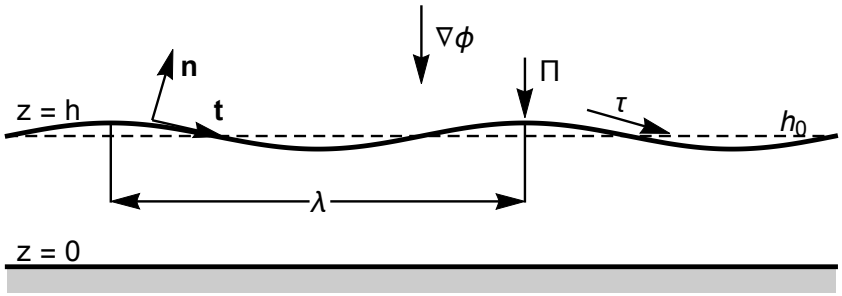

Figure 2.1 A diagram of the Thin-Film Model

2.1 Derivation

These equations completely describe the fluid, but we are now going to make a lubrication approximation, through a scaling argument similar to that given by Oron et al. [22]. For the lubrication approximation we are going to assume that the average height of the liquid, h_0, is much smaller then the characteristic wavelength of the liquid, λ. We will denote the ratio of these two lengths as ε, that is

$$\varepsilon = \frac{h_0}{\lambda} \ll 1. \tag{2.6}$$

Now we nondimensionalize the rest of the variables with respect to this ratio. We denote the nondimensional variables as the uppercase variables. As we stated earlier the characteristic height is h_0 and the characteristic length is $\lambda = h_0/\varepsilon$, so the nondimensional length variables are

$$Z = \frac{z}{h_0}, \quad X = \frac{\varepsilon x}{h_0}, \quad H = \frac{h}{h_0}. \tag{2.7}$$

Let U_0 be the characteristic horizontal velocity, then

$$U = \frac{u}{U_0}, \quad W = \frac{w}{\varepsilon U_0}, \tag{2.8}$$

where the nondimensional vertical velocity, W, follows from the continuity equation, equation (2.1). It follows that time will be scaled by λ/U_0, so nondimensional time is

$$T = \frac{\varepsilon U_0 t}{h_0}. \tag{2.9}$$

Finally we assume that the flow is locally parallel or equivalently $p_x \sim \mu u_{zz}$. This gives us the proper scaling for the pressure, gravity, and surface stresses,

$$P = \frac{\varepsilon h_0}{\mu U_0} p, \quad \Phi = \frac{\varepsilon h_0}{\mu U_0}\phi, \quad \Pi = \frac{\varepsilon h_0}{\mu U_0}\Pi_0, \quad \tau = \frac{h_0}{\mu U_0}\tau_0. \tag{2.10}$$

Lastly we can nondimensionalize the surface tension as

$$\Sigma = \frac{\varepsilon\sigma}{\mu U_0}. \tag{2.11}$$

Substituting these nondimensional variables into the continuity equation (2.1), gives

$$\begin{aligned} u_x + w_z &= 0, \\ \frac{\varepsilon U_0}{h_0} U_X + \frac{\varepsilon U_0}{h_0} W_Z &= 0, \\ U_X + W_Z &= 0. \end{aligned}$$

This computation also justifies the scaling of w as the nondimensional variables should also conserve mass.

We can nondimensionalize the conservation of momentum equations as follows,

$$\begin{aligned} \rho(u_t + uu_x + wu_z) &= -p_x + \mu\Delta u - \phi_x, \\ \rho\left(\frac{\varepsilon U_0^2}{h_0} U_T + \frac{\varepsilon U_0^2}{h_0} UU_X + \frac{\varepsilon U_0^2}{h_0} WU_Z\right) &= -\frac{\mu U_0}{h_0^2} P_X + \mu\left(\frac{\varepsilon^2 U_0}{h_0^2} U_{XX} + \frac{U_0}{h_0^2} U_{ZZ}\right) - \frac{\mu U_0}{h_0^2}\Phi_X, \\ \frac{\varepsilon U_0^2 \rho}{h_0}(U_T + UU_X + WU_Z) &= \frac{\mu U_0}{h_0^2}\big(-P_X + \big(\varepsilon^2 U_{XX} + U_{ZZ}\big) - \Phi_X\big), \\ \frac{\varepsilon U_0 \rho h_0}{\mu}(U_T + UU_X + WU_Z) &= \big(-P_X + \big(\varepsilon^2 U_{XX} + U_{ZZ}\big) - \Phi_X\big), \end{aligned}$$

and

$$\begin{aligned} \rho(w_t + uw_x + ww_z) &= -p_z + \mu\Delta w - \phi_z, \\ \rho\left(\frac{\varepsilon^2 U_0^2}{h_0} W_T + \frac{\varepsilon^2 U_0^2}{h_0} UW_X + \frac{\varepsilon^2 U_0^2}{h_0} WW_Z\right) &= -\frac{\mu U_0}{\varepsilon h_0^2} P_Z + \mu\left(\frac{\varepsilon^3 U_0}{h_0^2} W_{XX} + \frac{\varepsilon U_0}{h_0^2} W_{ZZ}\right) - \frac{\mu U_0}{\varepsilon h_0^2}\Phi_Z, \\ \frac{\varepsilon^2 \rho U_0^2}{h_0}(W_T + UW_X + WW_Z) &= \frac{\mu U_0}{\varepsilon h_0^2}\big(-P_Z + \big(\varepsilon^4 W_{XX} + \varepsilon^2 W_{ZZ}\big) - \Phi_Z\big), \\ \varepsilon^3 \frac{\rho U_0 h_0}{\mu}(W_T + UW_X + WW_Z) &= \big(-P_Z + \varepsilon^2\big(\varepsilon^2 W_{XX} + W_{ZZ}\big) - \Phi_Z\big). \end{aligned}$$

The boundary conditions are nondimensionalized as below,

$$\begin{aligned} w = 0, \quad u = 0, &\qquad \text{at } z = 0, \\ \varepsilon U_0 W = 0, \quad U_0 U = 0, &\qquad \text{at } Z = 0, \\ W = 0, \quad U = 0, &\qquad \text{at } Z = 0, \end{aligned}$$

with

$$\begin{aligned} w = h_t + u h_x, &\qquad \text{at } z = h, \\ \varepsilon U_0 W = \varepsilon U_0 H_T + \varepsilon U_0 U H_x, &\qquad \text{at } Z = H, \\ W = H_T + U H_x, &\qquad \text{at } Z = H. \end{aligned}$$

In order to nondimensionalize the stress tensor at the free surface, we consider the normal and tangential components of $\mathbf{T} \cdot \mathbf{n}$ separately. The normal and tangential vector at the surface can be expressed in terms of the free surface as

$$\mathbf{n} = \frac{\langle -h_x, 1 \rangle}{(1 + h_x^2)^{1/2}}, \quad \mathbf{t} = \frac{\langle 1, h_x \rangle}{(1 + h_x^2)^{1/2}}, \tag{2.12}$$

and the mean curvature, κ, can be expressed in terms of h as

$$\kappa = -\frac{h_{xx}}{(1 + h_x^2)^{3/2}}. \tag{2.13}$$

We would now like to write the following vector equation in terms of its normal and tangential components.

$$\mathbf{T} \cdot \mathbf{n} = (-\kappa\sigma + \Pi_0)\mathbf{n} + \left(\frac{\partial \sigma}{\partial s} + \tau_0\right)\mathbf{t}, \qquad \text{at } z = h.$$

Note that we are using a Newtonian stress tensor which is given by

$$\mathbf{T} = \begin{bmatrix} -p & 0 \\ 0 & -p \end{bmatrix} + \mu \begin{bmatrix} 2u_x & u_z + w_x \\ u_z + w_x & 2w_z \end{bmatrix} \tag{2.14}$$

The normal component of this vector equation is given by

$$\langle \mathbf{T} \cdot \mathbf{n}, \mathbf{n} \rangle = (-\kappa\sigma + \Pi_0). \tag{2.15}$$

First we will simplify the left hand side.

$$
\begin{aligned}
a &= \left(1+h_x^2\right)^{1/2},\\
\langle \mathbf{T}\cdot\mathbf{n},\mathbf{n}\rangle &= \frac{1}{a^2}\begin{bmatrix}-h_x & 1\end{bmatrix}\begin{bmatrix}-p+2\mu u_x & \mu(u_z+w_x)\\ \mu(u_z+w_x) & -p+2\mu w_z\end{bmatrix}\begin{bmatrix}-h_x\\ 1\end{bmatrix},\\
&= \frac{1}{a^2}\begin{bmatrix}-h_x & 1\end{bmatrix}\begin{bmatrix}-h_x(-p+2\mu u_x)+\mu(u_z+w_x)\\ -h_x\mu(u_z+w_x)+-p+2\mu w_z\end{bmatrix},\\
&= \frac{1}{a^2}\big(h_x^2(-p+2\mu u_x)-\mu h_x(u_z+w_x)-\mu h_x(u_z+w_x)+-p+2\mu w_z\big),\\
&= \frac{1}{a^2}\big(h_x^2(-p+2\mu u_x)-2\mu h_x(u_z+w_x)-p+2\mu w_z\big),\\
&= \frac{1}{a^2}\big(\big(1+h_x^2\big)(-p)+2\mu\big(h_x^2u_x+w_z\big)-2\mu h_x(u_z+w_x)\big),\\
&= -p+\frac{2\mu}{a^2}\big(\big(h_x^2u_x+w_z\big)-h_x(u_z+w_x)\big).
\end{aligned}
$$

Next simplify the right hand side,

$$(-\kappa\sigma+\Pi_0) = \frac{h_{xx}}{(1+h_x^2)^{3/2}}\sigma+\Pi_0.$$

This gives the following scalar equation,

$$-p-\Pi_0+\frac{2\mu}{1+h_x^2}\big(\big(h_x^2u_x+w_z\big)-h_x(u_z+w_x)\big) = \frac{h_{xx}}{(1+h_x^2)^{3/2}}\sigma. \tag{2.16}$$

The tangential component is given by

$$\langle \mathbf{T}\cdot\mathbf{n},\mathbf{t}\rangle = \left(\frac{\partial\sigma}{\partial s}+\tau_0\right). \tag{2.17}$$

First simplify the left hand side,

$$\begin{aligned}
\langle \mathbf{T}\cdot\mathbf{n},\mathbf{n}\rangle &= \frac{1}{a^2}\begin{bmatrix}1 & h_x\end{bmatrix}\begin{bmatrix}-p+2\mu u_x & \mu(u_z+w_x)\\ \mu(u_z+w_x) & -p+2\mu w_z\end{bmatrix}\begin{bmatrix}-h_x\\ 1\end{bmatrix},\\
&= \frac{1}{a^2}\begin{bmatrix}1 & h_x\end{bmatrix}\begin{bmatrix}-h_x(-p+2\mu u_x)+\mu(u_z+w_x)\\ -h_x\mu(u_z+w_x)+-p+2\mu w_z\end{bmatrix},\\
&= \frac{1}{a^2}\big(-h_x(-p+2\mu u_x)+\mu(u_z+w_x)+-h_x^2\mu(u_z+w_x)+h_x(-p+2\mu w_z)\big),\\
&= \frac{1}{a^2}\big(-h_x(2\mu u_x)+\mu(u_z+w_x)+-h_x^2\mu(u_z+w_x)+h_x(2\mu w_z)\big),\\
&= \frac{\mu}{a^2}\big(2h_x(w_z-u_x)+\big(1-h_x^2\big)(u_z+w_x)\big).
\end{aligned}$$

Next simplify the right hand side

$$\begin{aligned}
\left(\frac{\partial\sigma}{\partial s}+\tau_0\right) &= \frac{\partial\sigma}{\partial x}\frac{\partial x}{\partial s}+\tau_0,\\
&= \frac{\partial\sigma}{\partial x}\frac{1}{(1+h_x^2)^{1/2}}+\tau_0.
\end{aligned}$$

This gives the following scalar equation,

$$\mu\big(2h_x(w_z-u_x)+\big(1-h_x^2\big)(u_z+w_x)\big)=\frac{\partial\sigma}{\partial x}\big(1+h_x^2\big)^{1/2}+\tau_0\big(1+h_x^2\big). \tag{2.18}$$

Lastly we will nondimensionalize these two equations,

$$-p-\pi+\frac{2\mu}{1+h_x^2}\big(\big(h_x^2u_x+w_z\big)-h_x(u_z+w_x)\big)=\frac{h_{xx}}{(1+h_x^2)^{3/2}}\sigma,$$

$$-\frac{\mu U_0}{\varepsilon h_0}P-\frac{\mu U_0}{\varepsilon h_0}\Pi+\frac{2\mu}{1+\varepsilon^2H_X^2}\left(\left(\varepsilon^2H_X^2\frac{\varepsilon U_0}{h_0}U_X+\frac{\varepsilon U_0}{h_0}W_Z\right)-\varepsilon H_X\left(\frac{U_0}{h_0}U_Z+\frac{\varepsilon^2U_0}{h_0}W_X\right)\right)$$
$$=\frac{\varepsilon^2}{h_0}\frac{H_{XX}}{\big(1+\varepsilon^2H_X^2\big)^{3/2}}\frac{\mu U_0}{\varepsilon}\Sigma,$$

$$\left(-\frac{\mu U_0}{\varepsilon h_0}P-\frac{\mu U_0}{\varepsilon h_0}\Pi+\frac{\varepsilon\mu U_0}{h_0}\frac{2}{1+\varepsilon^2H_X^2}\big(\big(\varepsilon^2H_X^2U_X+W_Z\big)-H_X\big(U_Z+\varepsilon^2W_X\big)\big)\right)$$
$$=\frac{\varepsilon\mu U_0}{h_0}\frac{H_{XX}}{\big(1+\varepsilon^2H_X^2\big)^{3/2}}\Sigma,$$

$$\left(-\frac{\mu U_0}{\varepsilon h_0}P-\frac{\mu U_0}{\varepsilon h_0}\Pi+\frac{\varepsilon\mu U_0}{h_0}\frac{2}{1+\varepsilon^2H_X^2}\big(\big(\varepsilon^2H_X^2U_X+W_Z\big)-H_X\big(U_Z+\varepsilon^2W_X\big)\big)\right)$$
$$=\frac{\varepsilon\mu U_0}{h_0}\frac{H_{XX}}{\big(1+\varepsilon^2H_X^2\big)^{3/2}}\Sigma,$$

$$\frac{\mu U_0}{\varepsilon h_0}\left(-P-\Pi+\frac{2\varepsilon^2}{1+\varepsilon^2H_X^2}\big(\big(\varepsilon^2H_X^2U_X+W_Z\big)-H_X\big(U_Z+\varepsilon^2W_X\big)\big)\right)=\frac{\mu U_0}{\varepsilon h_0}\frac{\varepsilon^2H_{XX}}{\big(1+\varepsilon^2H_X^2\big)^{3/2}}\Sigma,$$

$$\left(-P-\Pi+\frac{2\varepsilon^2}{1+\varepsilon^2H_X^2}\big(\big(\varepsilon^2H_X^2U_X+W_Z\big)-H_X\big(U_Z+\varepsilon^2W_X\big)\big)\right)=\frac{\varepsilon^2H_{XX}}{\big(1+\varepsilon^2H_X^2\big)^{3/2}}\Sigma,$$

and

$$\mu\big(2h_x(w_z-u_x)+\big(1-h_x^2\big)(u_z+w_x)\big)=\frac{\partial\sigma}{\partial x}\big(1+h_x^2\big)^{1/2}+\tau_0\big(1+h_x^2\big),$$

$$\mu\left(2\varepsilon H_X\left(\frac{\varepsilon U_0}{h_0}W_Z-\frac{\varepsilon U_0}{h_0}U_X\right)+\big(1-\varepsilon^2H_X^2\big)\left(\frac{U_0}{h_0}U_Z+\frac{\varepsilon^2U_0}{h_0}W_X\right)\right)$$
$$=\frac{\mu U_0}{h_0}\Sigma_X\big(1+\varepsilon^2H_X^2\big)^{1/2}+\frac{\mu U_0}{h_0}\tau\big(1+\varepsilon^2H_X^2\big),$$

$$\frac{\mu U_0}{h_0}\big(2\varepsilon^2H_X(W_Z-U_X)+\big(1-\varepsilon^2H_X^2\big)\big(U_Z+\varepsilon^2W_X\big)\big)$$
$$=\frac{\mu U_0}{h_0}\Big(\Sigma_X\big(1+\varepsilon^2H_X^2\big)^{1/2}+\tau\big(1+\varepsilon^2H_X^2\big)\Big),$$

$$2\varepsilon^2H_X(W_Z-U_X)+\big(1-\varepsilon^2H_X^2\big)\big(U_Z+\varepsilon^2W_X\big)$$
$$=\Sigma_X\big(1+\varepsilon^2H_X^2\big)^{1/2}+\tau\big(1+\varepsilon^2H_X^2\big).$$

The full nondimensional equations are thus

$$U_X + W_Z = 0, \tag{2.19}$$

$$\frac{\varepsilon U_0 \rho h_0}{\mu}(U_T + UU_X + WU_Z) = -P_X + \left(\varepsilon^2 U_{XX} + U_{ZZ}\right) - \Phi_X, \tag{2.20}$$

$$\varepsilon^3 \frac{\rho U_0 h_0}{\mu}(W_T + UW_X + WW_Z) = -P_Z + \varepsilon^2\left(\varepsilon^2 W_{XX} + W_{ZZ}\right) - \Phi_Z, \tag{2.21}$$

at $Z = 0$

$$W = 0, \quad U = 0, \tag{2.22}$$

and at $Z = H$,

$$W = H_T + UH_x, \tag{2.23}$$

$$-P - \Pi + \frac{2\varepsilon^2}{1+\varepsilon^2 H_X^2}\left(\left(\varepsilon^2 H_X^2 U_X + W_Z\right) - H_X\left(U_Z + \varepsilon^2 W_X\right)\right) = \frac{\varepsilon^2 H_{XX}}{\left(1+\varepsilon^2 H_X^2\right)^{3/2}}\Sigma, \tag{2.24}$$

$$2\varepsilon^2 H_X(W_Z - U_X) + \left(1 - \varepsilon^2 H_X^2\right)\left(U_Z + \varepsilon^2 W_X\right) = \Sigma_X\left(1+\varepsilon^2 H_X^2\right)^{1/2} + \tau\left(1+\varepsilon^2 H_X^2\right). \tag{2.25}$$

We can now let $\varepsilon \to 0$ which results in

$$U_X + W_Z = 0, \tag{2.26}$$

$$P_X + \Phi_X = U_{ZZ}, \tag{2.27}$$

$$P_Z + \Phi_Z = 0, \tag{2.28}$$

at $Z = 0$,

$$W = 0, \quad U = 0, \tag{2.29}$$

$$\tag{2.30}$$

and at $Z = H$

$$W = H_T + UH_x, \tag{2.31}$$

$$-P - \Pi = \bar{\Sigma} H_{XX}, \tag{2.32}$$

$$U_Z = \Sigma_X + \tau. \tag{2.33}$$

Note that we assume that the surface tension is large, so that $\bar{\Sigma} = \varepsilon^2\Sigma = O(1)$. This is important in order to keep surface tension effects in the final equation.

Next we integrate the continuity equation over Z,

$$\begin{aligned}
\int_0^H U_X + W_Z \,\mathrm{d}Z &= 0,\\
\int_0^H U_X \,\mathrm{d}Z + W|_{Z=0}^H &= 0,\\
\int_0^H U_X \,\mathrm{d}Z + H_T + UH_X &= 0,\\
H_T + \int_0^H U_X \,\mathrm{d}Z + UH_X &= 0,\\
H_T + \frac{\partial}{\partial X}\left(\int_0^H U \,\mathrm{d}Z\right) &= 0.
\end{aligned}$$

Using the boundary conditions we can solve for an expression of U as follows

$$\begin{aligned}
\int_Z^H U_{ZZ} \,\mathrm{d}Z &= \int_Z^H P_X + \Phi_X \,\mathrm{d}Z,\\
U_Z|_Z^H &= (P_X + \Phi_X)(H - Z),\\
(\tau + \Sigma_X) - U_Z &= (P_X + \Phi_X)(H - Z),\\
\int_0^Z (\tau + \Sigma_X) - U_Z \,\mathrm{d}Z &= \int_0^Z (P_X + \Phi_X)(H - Z)\,\mathrm{d}Z,\\
(\tau + \Sigma_X)Z - U|_{Z=0}^Z &= (P_X + \Phi_X)\left(HZ - \frac{1}{2}Z^2\right),\\
(\tau + \Sigma_X)Z - U &= (P_X + \Phi_X)\left(HZ - \frac{1}{2}Z^2\right),\\
U &= (\tau + \Sigma_X)Z + (P_X + \Phi_X)\left(\frac{1}{2}Z^2 - HZ\right).
\end{aligned}$$

The boundary conditions also give an expression for $P + \Phi$,

$$\begin{aligned}
P_Z + \Phi_Z &= 0,\\
\int_Z^H P_Z + \Phi_Z \,\mathrm{d}Z &= 0,\\
P|_{Z=H} - P + \Phi|_{Z=H} - \Phi &= 0,\\
-\Pi - \bar{\Sigma}H_{XX} - P + \Phi|_{Z=H} - \Phi &= 0,\\
P + \Phi &= \Phi|_{Z=H} - \Pi - \bar{\Sigma}H_{XX}.
\end{aligned}$$

Plugging both of these into the integrated continuity equation gives,

$$\begin{aligned}
H_T + \frac{\partial}{\partial X}\left(\int_0^H U \,\mathrm{d}Z\right) &= 0,\\
H_T + \frac{\partial}{\partial X}\left(\int_0^H (\tau+\Sigma_X)Z + (P_X+\Phi_X)\left(\frac{1}{2}Z^2 - HZ\right)\mathrm{d}Z\right) &= 0,\\
H_T + \left(\frac{1}{2}(\tau+\Sigma_X)H^2 + (P_X+\Phi_X)\left(\frac{1}{6}H^3 - \frac{1}{2}H^3\right)\right)_X &= 0,\\
H_T + \left(\frac{1}{2}(\tau+\Sigma_X)H^2 - \frac{1}{3}(P+\Phi)_X H^3\right)_X &= 0,\\
H_T + \left(\frac{1}{2}(\tau+\Sigma_X)H^2 - \frac{1}{3}\big(\Phi|_{Z=H} - \Pi - \bar{\Sigma}H_{XX}\big)_X H^3\right)_X &= 0,\\
H_T + \left(\frac{1}{2}(\tau+\Sigma_X)H^2 - \frac{1}{3}(\Phi|_{Z=H} - \Pi)_X H^3\right)_X &= -\big(\bar{\Sigma}H^3 H_{XXX}\big)_X.
\end{aligned}$$

This is our final thin-film equation and taking all of the constants to be one gives

$$H_T + \big(H^2 - H^3\big)_X = -\big(H^3 H_{XXX}\big)_X. \tag{2.34}$$

For the rest of the paper we will relabel the variables and use variables originally shown in equation 1.1,

$$q_t + \big(q^2 - q^3\big)_x = -\big(q^3 q_{xxx}\big)_x. \tag{2.35}$$

2.2 Equation Properties

CHAPTER 3. Numerical Methods

3.1 Implicit-Explicit Runge-Kutta Scheme

The thin-film model we are trying to solve is a very stiff equation due to the high order derivatives. In order to take reasonable timesteps despite this stiffness we use implicit-explicit (IMEX) Runge-Kutta schemes to propagate our solution through time. The IMEX Runge-Kutta scheme propagates time for ordinary differential equations or systems of equations of the form

$$q_t = F(t,q) + G(t,q), \tag{3.1}$$

where F is solely handled explicitly, but G needs to be solved implicitly. A single step of the Runge-Kutta IMEX scheme is computed as follows,

$$q^{n+1} = q^n + \Delta t \sum_{i=1}^{s} \left(b'_i F(t_i, u_i)\right) + \Delta t \sum_{i=1}^{s} \left(b_i G(t_i, u_i)\right) \tag{3.2}$$

$$u_i = q^n + \Delta t \sum_{j=1}^{i-1} \left(a'_{ij} F(t_j, u_j)\right) + \Delta t \sum_{j=1}^{i} \left(a_{ij} G(t_j, u_j)\right) \tag{3.3}$$

$$t_i = t^n + c_i \Delta t \tag{3.4}$$

where the coefficients a_{ij}, b_i, c_i, a'_{ij}, b'_i, and c'_i are set in the double Butcher tableau

$$\begin{array}{c|c} c' & a' \\ \hline & (b')^T \end{array} \qquad \begin{array}{c|c} c & a \\ \hline & b^T \end{array}.$$

Specifically we used the IMEX schemes first introduced by Pareschi and Russo [23, 24]. These schemes are strong stability preserving (SSP) in the sense of Gottlieb et al. [12] and they are designed for stiff systems. The double Butcher tableaus are shown below for the one stage, L-stable, 1st order method:

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array} \qquad \begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array},$$

the three stage, 2nd order method:

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ \hline & 0 & \frac{1}{2} & \frac{1}{2} \end{array} \qquad \begin{array}{c|ccc} \frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & -\frac{1}{2} & \frac{1}{2} & 0 \\ 1 & 0 & \frac{1}{2} & \frac{1}{2} \\ \hline & 0 & \frac{1}{2} & \frac{1}{2} \end{array},$$

and the four stage, L-stable, third order method:

$$\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{4} & \frac{1}{4} & 0 \\ \hline & 0 & \frac{1}{6} & \frac{1}{6} & \frac{2}{3} \end{array} \qquad \begin{array}{c|cccc} \alpha & \alpha & 0 & 0 & 0 \\ 0 & -\alpha & \alpha & 0 & 0 \\ 1 & 0 & 1-\alpha & \alpha & 0 \\ \frac{1}{2} & \beta & \eta & \zeta & \alpha \\ \hline & 0 & \frac{1}{6} & \frac{1}{6} & \frac{2}{3} \end{array},$$

where

$$\begin{aligned} \alpha &= 0.24169426078821, \\ \beta &= 0.06042356519705, \\ \eta &= 0.1291528696059, \\ \zeta &= \frac{1}{2} - \beta - \eta - \alpha. \end{aligned}$$

For our thin-film model F and G will be the spatial discretizations of the hyperbolic convection and parabolic terms respectively, that is

$$F(t,q) = -\left(q^2 - q^3\right)_x \tag{3.5}$$

$$G(t,q) = -\left(q^3 q_{xxx}\right)_x. \tag{3.6}$$

Since G is the stiffest part of our model, we would like to handle this term implicitly so that our time step is less restricted. Also handling F explicitly allows us to capture the nonlinear behavior without a nonlinear solve.

3.2 Space Discretization

We chose to use the discontinuous Galerkin Method to discretize our equation in space. First let $\{x_{j+1/2}\}_0^N$ partition the domain, $[a,b]$, and denote each interval as $I_j = [x_{j-1/2}, x_{j+1/2}]$ with $\Delta x_j = x_{j+1/2} - x_{j-1/2}$. The discontinuous Galerkin solution then exists in the following finite dimensional space,

$$V_h^k = \Big\{ v \in L^1([a,b]) : v|_{I_j} \in P^k(I_j), j = 1, \ldots, N \Big\}, \tag{3.7}$$

where $P^k(I_j)$ denotes the set of polynomials of degree k or less on I_j.

3.2.1 Modal Discontinuous Galerkin for Hyperbolic Convection

First we consider the continuous operator $F(t,q) = -\left(q^2 - q^3\right)_x$. Note that $F : [0,T] \times C^1(a,b) \to C^0(a,b)$, and we would like to form an approximation, F_h to this continuous operator, such that $F_h : [0,T] \times V_h^k \to V_h^k$. In order to do this, consider the weak formulation of the continuous operator, F. The weak form requires finding $F(t,q) \in C^0(a,b)$ such that

$$\int_a^b F(t,q)v(x)\,\mathrm{d}x = -\int_a^b \left(q^2 - q^3\right)_x v(x)\,\mathrm{d}x \tag{3.8}$$

for all smooth functions, $v \in C^\infty(a,b)$. The discontinuous Galerkin approximation is formed by replacing these function spaces with the DG space, V_h^k. This gives that the weak formulation of the approximation F_h is to find $F_h(t,q_h) \in V_h^k$ such that

$$\int_a^b F_h(t,q_h)v_h(x)\,\mathrm{d}x = -\int_a^b \left(q_h^2 - q_h^3\right)_x v_h(x)\,\mathrm{d}x \tag{3.9}$$

for all $v_h \in V_h^k$. Using integration by parts this is equivalent to finding $F_h(t,q_h) \in V_h^k$ such that

$$\begin{aligned}\int_{I_j} F_h(t,q_h)v_h(x)\,\mathrm{d}x = &\int_{I_j} \left(q_h^2 - q_h^3\right)v_h'(x)\,\mathrm{d}x \\ &- \hat{f}(q_h)_{j+1/2}v_h(x_{j+1/2}) + \hat{f}(q_h)_{j-1/2}v_h(x_{j-1/2})\end{aligned} \tag{3.10}$$

for $j = 1, \ldots, N$ and for all $v_h \in V_h^k$. Since q_h is discontinuous at the cell interfaces $x_{j\pm1/2}$, some numerical flux $\hat{f}$ must be chosen.

We chose to use the Local Lax-Friedrich's numerical flux, that is

$$\hat{f}(q_l, q_r) = \frac{1}{2}(f(q_l) + f(q_r) - \lambda_{\max}(q_r - q_l)) \tag{3.11}$$

where f is the flux function, q_l and q_r and the left and right states at the interface, and $\lambda_{\max}$ is the locally maximum wavespeed, that is

$$\lambda_{\max} = \max_{q\in[\nu,\mu]} \{f'(q)\}, \tag{3.12}$$

where $\nu = \min\{q_l, q_r\}$ and $\mu = \max\{q_l, q_r\}$. In our case $f(q) = q^2 - q^3$. At the interface $x_{j+1/2}$ we have $q_l = q_h^-(x_{j+1/2})$ and $q_r = q_h^+(x_{j+1/2})$.

3.2.2 Local Discontinuous Galerkin Scheme for Parabolic Term

Next we look at the continuous operator $G(t, q) = -\left(q^3 q_{xxx}\right)_x$. In discretizing this operator we follow the local discontinuous Galerkin (LDG) method first introduced by Cockburn and Shu [10] for convection-diffusion systems. We first introduce three auxilliary variables, r, s, u, and rewrite the equation as the following system,

$$r = q_x, \tag{3.13}$$

$$s = r_x, \tag{3.14}$$

$$u = s_x, \tag{3.15}$$

$$G(t, q) = -\left(q^3 u\right)_x. \tag{3.16}$$

The weak form of this system is solved by finding functions r, s, u, G such that

$$\int_a^b r(x)w(x)\,\mathrm{d}x = \int_a^b q_x(x)w(x)\,\mathrm{d}x, \tag{3.17}$$

$$\int_a^b s(x)y(x)\,\mathrm{d}x = \int_a^b r_x(x)y(x)\,\mathrm{d}x, \tag{3.18}$$

$$\int_a^b u(x)z(x)\,\mathrm{d}x = \int_a^b s_x(x)z(x)\,\mathrm{d}x, \tag{3.19}$$

$$\int_a^b G(t, q)v(x)\,\mathrm{d}x = -\int_a^b \left(q^3(x)u(x)\right)_x v(x)\,\mathrm{d}x, \tag{3.20}$$

for all smooth functions $w, y, z, v \in C^\infty(a,b)$. The LDG method arrives from applying the standard DG method to each of these equations. That is we replace the continuous function spaces with the discontinuous finite dimensional DG space, V_h^k. The approximate operator G_h becomes the process of finding $r_h, s_h, u_h, G_h(t,q_h) \in V_h$ such that for all test functions $v_h, w_h, y_h, z_h \in V_h$ the following equations are satisfied

$$\int_a^b r_h w_h \,\mathrm{d}x = \int_a^b (q_h)_x w_h \,\mathrm{d}x, \tag{3.21}$$

$$\int_a^b s_h y_h \,\mathrm{d}x = \int_a^b (r_h)_x y_h \,\mathrm{d}x, \tag{3.22}$$

$$\int_a^b u_h z_h \,\mathrm{d}x = \int_a^b (s_h)_x z_h \,\mathrm{d}x, \tag{3.23}$$

$$\int_a^b G_h(t,q_h) v_h \,\mathrm{d}x = -\int_a^b \left(q_h^3 u_h\right)_x v_h \,\mathrm{d}x, \tag{3.24}$$

given t and $q_h \in V_h^k$. This is equivalent to the following equations for all j, if we use integration by parts,

$$\int_{I_j} r_h w_h \,\mathrm{d}x = \left(\left(\hat{q}_h w_h^-\right)_{j+1/2} - \left(\hat{q}_h w_h^+\right)_{j-1/2}\right) - \int_{I_j} q_h (w_h)_x \,\mathrm{d}x, \tag{3.25}$$

$$\int_{I_j} s_h y_h \,\mathrm{d}x = \left(\left(\hat{r}_h y_h^-\right)_{j+1/2} - \left(\hat{r}_h y_h^+\right)_{j-1/2}\right) - \int_{I_j} r_h (y_h)_x \,\mathrm{d}x, \tag{3.26}$$

$$\int_{I_j} u_h z_h \,\mathrm{d}x = \left(\left(\hat{s}_h z_h^-\right)_{j+1/2} - \left(\hat{s}_h z_h^+\right)_{j-1/2}\right) - \int_{I_j} s_h (z_h)_x \,\mathrm{d}x, \tag{3.27}$$

$$\int_{I_j} G_h(t,q_h) v_h \,\mathrm{d}x = -\left(\left(\widehat{q^3 u_h} v_h^-\right)_{j+1/2} - \left(\widehat{q^3 u_h} v_h^+\right)_{j-1/2}\right) + \int_{I_j} q_h^3 u_h (v_h)_x \,\mathrm{d}x, \tag{3.28}$$

where $\hat{q}, \hat{r}, \hat{s}, \widehat{q^3 u}$ are suitably chosen numerical fluxes. A common choice of numerical fluxes are the so-called alternating fluxes, shown below

$$\hat{q}_h = q_h^-, \tag{3.29}$$

$$\hat{r}_h = r_h^+, \tag{3.30}$$

$$\hat{s}_h = s_h^-, \tag{3.31}$$

$$\widehat{q^3 u_h} = \left(q^3 u\right)_h^+. \tag{3.32}$$

These numerical fluxes are one-sided fluxes that alternate sides for each derivative. They are chosen to make this method stable, and they also allow for the auxilliary variables to be locally solved in terms of q_h, hence where the local discontinuous Galerkin method gets its name.

3.3 Nonlinear Solver

Using these discretizations in the Runge-Kutta IMEX scheme, requires solving the nonlinear system,

$$u_i - \Delta t a_{ii} G_h(t_i, u_i) = q^n + \Delta t \sum_{j=1}^{i-1} \left(a'_{ij} F_h(t_j, u_j)\right) + \Delta t \sum_{j=1}^{i-1} \left(a_{ij} G_h(t_j, u_j)\right), \tag{3.33}$$

for u_i. The standard approach to solving this system would be to use a Newton iteration, however that would require the Jacobian of the operator $I - \Delta t a_{ii} G_h$, which would be relatively intractable. Therefore we chose to linearize this operator and use a Picard iteration instead, which does not require the Jacobian.

Suppose we are trying to solve the nonlinear equation $L(u) = b$, where $L'(v, u)$ is the nonlinear operator linearized about v acting on u. The Picard iteration for solving this nonlinear equation starts with some initial guess u_0. The next solution is found by solving the following linear problem,

$$L'(u_i, u_{i+1}) = b. \tag{3.34}$$

In other words the next iteration is found by solving the problem linearized about the the previous iteration.

We linearize the operator $G_h(t, q_h)$ by first linearizing $G(t, q)$. The continuous operator linearized about v is given by

$$G'(v, t, q) = -\left(v^3 q_{xxx}\right)_x. \tag{3.35}$$

The linearized discrete operator is now just the LDG method applied to this linearized continuous operator.

We find that the Picard iteration approach provides good results with relatively few iterations. In fact in Section 4.1 we show that we can achieve first, second, and third order accuracy with

only one, two, and three iterations respectively per stage. This approach is more tractable than a Newton iteration and it converges quickly to the nonlinear solution.

These methods were implemented in a python code called PyDoGPack [17] developed by the authors. PyDoGPack uses the packages NumPy [21] and SciPy [26] extensively. In particular they are used to solve linear systems of equations and to integrate numerically. All of the examples and tests shown in Chapter 4 were done using this code. Key parts of the code are shown in the Appendix.

CHAPTER 4. Results

4.1 Manufactured Solution

In order to confirm the order of accuracy of our method, we use the method of manufactured solutions. The method of manufactured solutions picks an exact solution $\hat{q}$ and then adds a source term to the initial partial differential equation to make $\hat{q}$ the true solution to the new differential equation. So we actually solve

$$q_t + \left(q^2 - q^3\right)_x = -\left(q^3 q_{xxx}\right)_x + s, \tag{4.1}$$

where

$$s = \hat{q}_t + \left(\hat{q}^2 - \hat{q}^3\right)_x + \left(\hat{q}^3 \hat{q}_{xxx}\right)_x. \tag{4.2}$$

This new PDE's exact solution is now $\hat{q}$. We chose

$$\hat{q} = 0.1 \times \sin(2\pi/20.0 \times (x - t)) + 0.15, \tag{4.3}$$

on $x \in [0, 40]$ with periodic boundary conditions to be our manufactured solution. This manufactued solution is chosen so that our flux $q^2 - q^3$ is in it's convex region.

We solve this problem until $t = 5.0$ with constant, linear, and quadratic basis polynomials. The time step size for these simulations is determined by the Courant–Friedrichs–Lewy (CFL) condition. The CFL condition states that

$$\Delta t = \nu \frac{\Delta x}{\lambda}, \tag{4.4}$$

where λ is the wavespeed and ν is the CFL number. For hyperbolic problems solved with explicit Runge-Kutta time-stepping, ν, typically follows the pattern $\frac{1}{2n-1}$ where n is the order of the method. For this example the wavespeed is 1, and the timestep is chosen just smaller then the typical CFL number. Specifically the timesteps were $\Delta t = 0.9\Delta x$, $\Delta t = 0.2\Delta x$, and $\Delta t = 0.1\Delta x$ for first, second and third order respectively.

Table 4.1 shows the error for each simulation as the mesh is refined. The table also shows the rate of convergence to the true solution, and in each case the rate of convergence approaches the expected order. Note that the first, second, and third order methods only require one, two, and three Picard iterations respectively.

The relative L^2 error is defined as

$$E = \|e(x)\|_{L^2(a,b)} = \sqrt{\frac{\int_a^b (\hat{q} - q_h)^2 \, \mathrm{d}x}{\int_a^b \hat{q}^2 \, \mathrm{d}x}} \tag{4.5}$$

Since we are using an orthonormal basis of V_h^k, there is an easier way to compute this error numerically. The basis consists of functions ϕ_j^l, that are polynomials of degree l on I_j and are zero everywhere else for $j = 1, \ldots, N$ and $l = 0, \ldots, k$. By definition of orthonormality we have that

$$\int_a^b \phi_i^k \phi_j^l \, \mathrm{d}x = \delta_{kl}\delta_{ij} \tag{4.6}$$

where δ is the Kronecker delta. In other words on cell I_j, ϕ_j^l is a Legendre polynomial. Given this basis we can express our numerical solution, $q_h \in V_h^k$, as

$$q_h|_{I_j} = \sum_{l=0}^{k} \left(Q_j^l \phi_j^l \right), \tag{4.7}$$

for all j, where Q_j^l are constant coefficients.

In order to maintain leading order accuracy we project our exact solution, $\hat{q}$, onto the space V_h^{k+1}. Let $\hat{q}_h$ denote this projection. We use the same orthonormal basis with the addition of ϕ_j^{k+1} on each cell. In this case we have the following representation

$$\hat{q}_h|_{I_j} = \sum_{l=0}^{k+1} \left(\hat{Q}_j^l \phi_j^l(x) \right) \text{ and } \; q_h|_{I_j} = \sum_{l=0}^{k+1} \left(Q_j^l \phi_j^l(x) \right). \tag{4.8}$$

where Q_j^{k+1} will be zero. Substituting these representations into the L^2 error, equation (4.5), and using the orthonormality property gives the following expression for the error,

$$E_N = \sqrt{\frac{\sum_{j=1}^{N} \sum_{l=0}^{k+1} \left(\hat{Q}_i - Q_i \right)^2}{\sum_{j=1}^{N} \left(\sum_{l=0}^{k+1} \left(\hat{Q}_i^2 \right) \right)}}. \tag{4.9}$$

	1st Order		2nd Order		3rd Order	
n	error	order	error	order	error	order
20	0.136	—	7.34×10^{-3}	—	5.29×10^{-4}	—
40	0.0719	0.91	1.99×10^{-3}	1.89	5.38×10^{-5}	3.30
80	0.0378	0.93	5.60×10^{-4}	1.83	7.47×10^{-6}	2.85
160	0.0191	0.99	1.56×10^{-4}	1.85	9.97×10^{-7}	2.91
320	0.00961	0.99	3.98×10^{-5}	1.97	1.26×10^{-7}	2.98
640	0.00483	0.99	1.00×10^{-5}	1.99	1.58×10^{-8}	3.00
1280	0.00242	1.00	2.50×10^{-6}	2.00	1.98×10^{-9}	3.00

Table 4.1 Convergence table with a constant, linear, quadratic polynomial bases. Nonlinear solve uses one, two, or three Picard iterations respectively. CFL = 0.9, 0.2, 0.1 respectively.

This expression is equivalent to equation (4.5) to leading order. The order of convergence is then estimated from these errors as

$$\text{order} \approx \log_2\left(\frac{E_N}{E_{2N}}\right). \tag{4.10}$$

4.2 Traveling Waves

In this section we showcase several numerical examples that demonstrate the traveling wave profiles of equation (1.1). The traveling wave profiles differ from the standard hyperbolic wave profile in several ways. They differ in that they may not be unique and may include undercompressive shocks. These examples were first shown in [4] with first order accuracy.

In these examples, we use a moving reference frame to keep the mesh size reasonable. This is done by actually simulating the following equation,

$$q_t + \left(q^2 - q^3 - sq\right)_x = -\left(q^3 q_{xxx}\right)_x \tag{4.11}$$

where s is the Rankine-Hugoniot wavespeed of the original numerical flux, f

$$s = \frac{f(q_l) - f(q_r)}{q_l - q_r} = q_l + q_r - (q_l^2 + q_l q_r + q_r^2). \tag{4.12}$$

This modified equation will have zero wavespeed in most cases and simulates the original PDE in a moving reference frame.

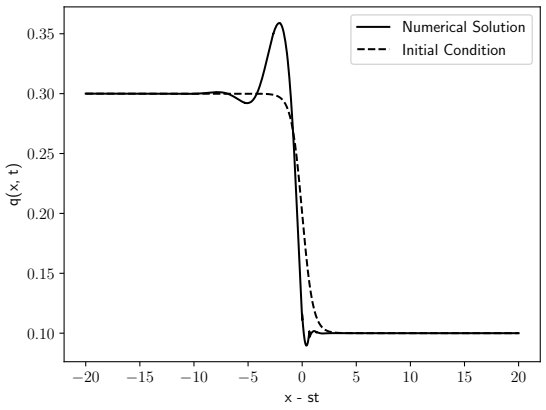

Figure 4.1 Case 1: Unique Lax Shock.

These examples consider Riemann Problems with different left and right states. Depending on the left and right states they are several different wave profiles. For these examples we will fix the right state and vary the left state. The wave profiles would be qualitatively equivalent for different values for the right state, however the values of the left state would also need to change accordingly.

4.2.1 Case 1: Unique Weak Lax Shock

Consider a Riemann Problem with left state, $q_l = 0.3$, and right state, $q_r = 0.1$. We will use the following smoothed out profile as an initial condition for this problem

$$q_0(x) = (\tanh(-x) + 1)\frac{q_l - q_r}{2} + q_r. \tag{4.13}$$

For this initial condition the numerical solution approaches a steady wave profile with a unique Lax type shock. Figure 4.1 shows a plot of the solution with this initial condition after enough time for the solution to hit its steady state. Note that this behavior persists for all q_l up to some bound which depends on q_r.

4.2.2 Case 2: Multiple Lax Shocks

If the left state is increased to $q_l = 0.3323$, then the wave profile is no longer unique. In this case the steady traveling wave depends on the initial conditions.

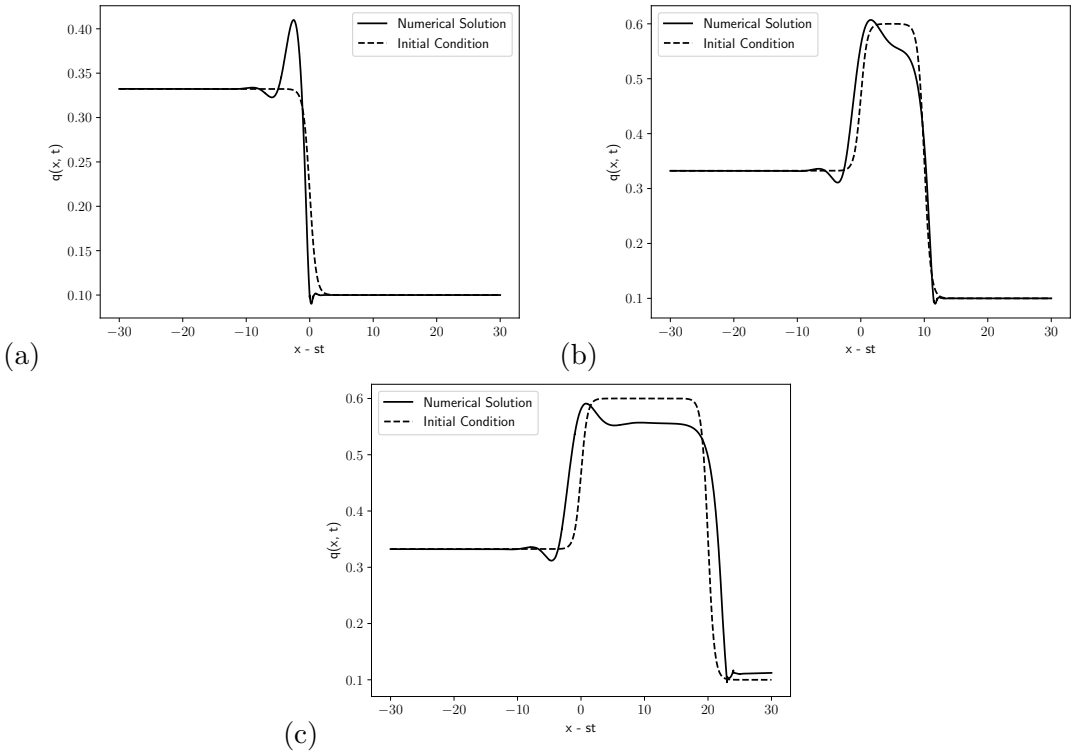

Figure 4.2 Case 2: Distinct Lax Shocks. Panels (a) and (b) show two distinct steady travelling waves formed from different initial conditions. Panel (c) shows an unsteady wave with a double shock structure from a third initial condition.

For the following initial conditions,

$$q_0(x) = (\tanh(-x) + 1)\frac{q_l - q_r}{2} + q_r, \tag{4.14}$$

the behavior is the same as in case 1. The numerical solution for this initial condition is shown in Figure 4.2

Now consider the following initial conditions,

$$q_0(x) = \begin{cases} ((0.6 - q_l)/2)\tanh(x) + ((0.6 + q_l)/2) & x < 5 \\ -((0.6 - q_r)/2)\tanh(x - 10) + ((0.6 + q_r)/2) & x > 5 \end{cases}. \tag{4.15}$$

The reader might expect this initial condition to approach the traveling wave shown earlier as it has the same far field boundary values, however this is not the case. The traveling wave profile that is approaches is shown in Figure 4.2.

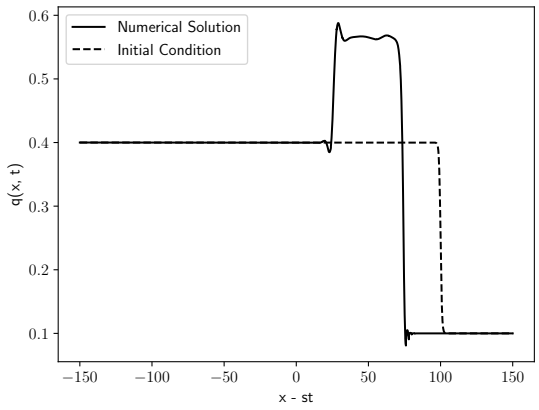

Figure 4.3 Case 3: Undercompressive Double Shock.

These are not the only possible wave profiles possible with these left and right states. Figure 4.2 part (c) shows the result for the following initial condition,

$$q_0(x) = \begin{cases} ((0.6 - q_l)/2)\tanh(x) + ((0.6 + q_l)/2) & x < 10 \\ -((0.6 - q_r)/2)\tanh(x - 20) + ((0.6 + q_r)/2) & x > 10 \end{cases}. \tag{4.16}$$

This initial condition has a larger hump then equation (4.15), and the traveling wave reflects this aspect. In this case the traveling wave is not a steady wave, and there are in fact two shocks. The right shock is an undercompressive shock and the left shock is a traditional compressive shock. Both shocks travel slower than the moving reference frame. They also travel at different speeds from one another so the wave profile changes over time.

4.2.3 Case 3: Undercompressive Double Shock

For the left state in the next regime, there is again a unique shock profile for all initial conditions. However this shock profile is not the single Lax shock seen in Case 1. With the following initial conditions,

$$q_0(x) = (\tanh(-x) + 1)\frac{q_l - q_r}{2} + q_r, \tag{4.17}$$

where $q_l = 0.4$ and $q_r = 0.1$, Figure 4.3 shows a double shock structure. Similar to Figure 4.2 part (c), we see a undercompressive shock on the right and a Lax shock on the left.

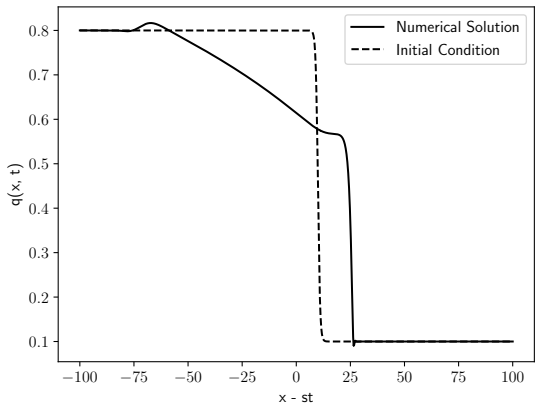

Figure 4.4 Case 4: Rarefaction-undercompressive shock.

4.2.4 Case 4: Rarefaction-Undercompressive Shock

The final traveling wave structure appears when the left state is greater than the undercompressive shock height. In this case we see a rarefaction wave along with the undercompressive shock. Figure 4.4 shows the numerical solution for initial condition

$$q_0(x) = (\tanh(-x + 110) + 1)\frac{q_l - q_r}{2} + q_r, \tag{4.18}$$

where $q_l = 0.8$ and $q_r = 0.1$. Note that the rarefaction wave and undercompressive shock are traveling at different speeds so they separate from each other.

CHAPTER 5. Conclusion

This paper has given a discontinuous Galerkin method for solving a Thin-Film model. The nonlinear convection term, $\left(q^2 - q^3\right)_x$, is discretized with the modal discontinuous Galerkin method and the nonlinear diffusion term, $-\left(q^3 q_{xxx}\right)_x$, is discretized with the local discontinuous Galerkin method. The nonlinear diffusion is very stiff and would force a very tight time restriction, therefore implicit-explicit Runge-Kutta methods are used for solving the semi-discretized system. The implicit-explicit time-stepping allows for reasonable time steps to be taken given the stiffness of the problem. As the diffusion is nonlinear, this still requires solving a nonlinear system, however we are able to solve this nonlinear problem without a Newton iteration. In fact a Picard Iteration was used, and it was demonstrated that only a minimal number of iterations were required to achieve high accuracy. The number of Picard Iterations needed is less than or equal to the order of the LDG discretization. Using this method we have demonstrated up to third order accuracy with the method of manufactured solutions. Also we have showcased several numerical examples first given by Bertozzi. These examples show that our method preserves the nonlinear behavior of the Thin-Film model that was first shown in those examples.

Bibliography

[1] U. M. Ascher, S. J. Ruuth, and R. J. Spiteri. Implicit-explicit runge-kutta methods for time-dependent partial differential equations. *Applied Numerical Mathematics*, 25(2-3):151–167, 1997.

[2] A. L. Bertozzi and M. P. Brenner. Linear stability and transient growth in driven contact lines. *Physics of Fluids*, 9(3):530–539, 1997.

[3] A. L. Bertozzi, A. Münch, X. Fanton, and A. M. Cazabat. Contact line stability and "undercompressive shocks" in driven thin film flow. *Physical review letters*, 81(23):5169, 1998.

[4] A. L. Bertozzi, A. Münch, and M. Shearer. Undercompressive shocks in thin film flows. *Physica D: Nonlinear Phenomena*, 134(4):431–464, 1999.

[5] A. Cazabat, F. Heslot, S. Troian, and P. Carles. Fingering instability of thin spreading films driven by temperature gradients. *Nature*, 346(6287):824, 1990.

[6] B. Cockburn, S. Hou, and C.-W. Shu. The runge-kutta local projection discontinuous galerkin finite element method for conservation laws. iv. the multidimensional case. *Mathematics of Computation*, 54(190):545–581, 1990.

[7] B. Cockburn, S.-Y. Lin, and C.-W. Shu. Tvb runge-kutta local projection discontinuous galerkin finite element method for conservation laws iii: one-dimensional systems. *Journal of Computational Physics*, 84(1):90–113, 1989.

[8] B. Cockburn and C.-W. Shu. Tvb runge-kutta local projection discontinuous galerkin finite element method for conservation laws. ii. general framework. *Mathematics of computation*, 52(186):411–435, 1989.

[9] B. Cockburn and C.-W. Shu. The runge-kutta local projection-discontinuous-galerkin finite element method for scalar conservation laws. *ESAIM: Mathematical Modelling and Numerical Analysis*, 25(3):337–361, 1991.

[10] B. Cockburn and C.-W. Shu. The local discontinuous galerkin method for time-dependent convection-diffusion systems. *SIAM Journal on Numerical Analysis*, 35(6):2440–2463, 1998.

[11] B. Cockburn and C.-W. Shu. The runge–kutta discontinuous galerkin method for conservation laws v: multidimensional systems. *Journal of Computational Physics*, 141(2):199–224, 1998.

[12] S. Gottlieb, C.-W. Shu, and E. Tadmor. Strong stability-preserving high-order time discretization methods. *SIAM review*, 43(1):89–112, 2001.

[13] Y. Ha, Y.-J. Kim, and T. G. Myers. On the numerical solution of a driven thin film equation. *Journal of Computational Physics*, 227(15):7246–7263, 2008.

[14] D. E. Kataoka and S. M. Troian. A theoretical study of instabilities at the advancing front of thermally driven coating films. *Journal of colloid and interface science*, 192(2):350–362, 1997.

[15] C. A. Kennedy and M. H. Carpenter. Additive runge–kutta schemes for convection–diffusion–reaction equations. *Applied Numerical Mathematics*, 44(1-2):139–181, 2003.

[16] E. Lindelöf. Sur l'application de la méthode des approximations successives aux équations différentielles ordinaires du premier ordre. *Comptes rendus hebdomadaires des séances de l'Académie des sciences*, 116(3):454–457, 1894.

[17] C. Logemann. PyDoGPack. Available from `http://www.github.com/caleblogemann/pydogpack`.

[18] V. Ludviksson and E. Lightfoot. The dynamics of thin liquid films in the presence of surface-tension gradients. *AIChE Journal*, 17(5):1166–1173, 1971.

[19] T. Myers, J. Charpin, and C. Thompson. Slowly accreting ice due to supercooled water impacting on a cold surface. *Physics of Fluids*, 14(1):240–256, 2002.

[20] T. G. Myers, J. P. Charpin, and S. J. Chapman. The flow and solidification of a thin fluid film on an arbitrary three-dimensional surface. *Physics of Fluids*, 14(8):2788–2803, 2002.

[21] T. E. Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[22] A. Oron, S. H. Davis, and S. G. Bankoff. Long-scale evolution of thin liquid films. *Reviews of modern physics*, 69(3):931, 1997.

[23] L. Pareschi and G. Russo. Implicit-explicit runge-kutta schemes for stiff systems of differential equations. *Recent trends in numerical analysis*, 3:269–289, 2000.

[24] L. Pareschi and G. Russo. Implicit–explicit runge–kutta schemes and applications to hyperbolic systems with relaxation. *Journal of Scientific computing*, 25(1):129–155, 2005.

[25] W. H. Reed and T. Hill. Triangular mesh methods for the neutron transport equation. Technical report, Los Alamos Scientific Lab., N. Mex.(USA), 1973.

[26] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. Jarrod Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. Carey,

İ. Polat, Y. Feng, E. W. Moore, J. Vand erPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and S. . . Contributors. SciPy 1.0–Fundamental Algorithms for Scientific Computing in Python. *arXiv e-prints*, page arXiv:1907.10121, Jul 2019.

APPENDIX. PyDogPack Code

```
class IMEXRungeKutta:
    # IMEX Runge Kutta scheme for solving q_t = F(t, q) + G(t, q)
    # where F is solved explicitly and G is solved implicitly
    # these schemes are represented by two Butcher Tableaus
    # a, b, c for operator G
    # ap, bp, cp for operator F
    # y^{n+1} = y^n + delta_t \sum{i = 1}{s}{bp_i F(t^n + c_i delta_t, u_i)}
    # + delta_t \sum{i = 1}{s}{b_i G(t^n + c_i delta_t, u_i)}
    # u_i = y^n + delta_t \sum{j = 1}{i-1}{ap_ij F(t^n + c_j delta_t, u_j)}
    # + delta_t \sum{j = 1}{i}{a_ij G(t^n + c_j delta_t, u_j)}
    # means solving
    # u_i - delta_t a_ii G(t^n + c_i delta_t, u_i) = rhs
    # rhs = y^n + delta_t \sum{j = 1}{i-1}{ap_ij F(t^n + c_j delta_t, u_j)}
    # + delta_t \sum{j = 1}{i-1}{a_ij G(t^n + c_j delta_t, u_j)}
    def __init__(self, a, b, c, ap, bp, cp):
        self.a = a
        self.b = b
        self.c = c
        self.ap = ap
        self.bp = bp
        self.cp = cp

        self.num_stages = self.c.size

    def time_step(
        self,
        q_old,
        t_old,
        delta_t,
        explicit_operator,
        implicit_operator,
        solve_operator,
    ):
        stages = []
        for i in range(self.num_stages):
            stages.append(
                self.stage(
                    q_old,
                    t_old,
                    delta_t,
                    explicit_operator,
                    implicit_operator,
                    solve_operator,
                    stages,
                    i,
                )
            )
```

```
        q_new = q_old.copy()
        for i in range(self.num_stages):
            if self.b[i] != 0.0:
                time = t_old + self.c[i] * delta_t
                q_new += delta_t * self.b[i] * implicit_operator(time, stages[i])
            if self.bp[i] != 0.0:
                time = t_old + self.cp[i] * delta_t
                q_new += delta_t * self.bp[i] * explicit_operator(time, stages[i])
        return q_new

    def stage(
        self,
        q_old,
        t_old,
        delta_t,
        explicit_operator,
        implicit_operator,
        solve_operator,
        stages,
        stage_num,
    ):
        # rhs = y^n + delta_t \sum{j = 1}{i-1}{ap_ij F(t^n + c_j delta_t, u_j)}
        # + delta_t \sum{j = 1}{i-1}{a_ij G(t^n + c_j delta_t, u_j)}
        stage_rhs = q_old.copy()
        for j in range(stage_num):
            if self.a[stage_num, j] != 0.0:
                time = t_old + self.c[j] * delta_t
                stage_rhs += (
                    delta_t * self.a[stage_num, j] * implicit_operator(time, stages[
                        j])
                )
            if self.ap[stage_num, j] != 0.0:
                time = t_old + self.cp[j] * delta_t
                stage_rhs += (
                    delta_t * self.ap[stage_num, j] * explicit_operator(time, stages
                        [j])
                )

        # d q + e G(t, f q) = rhs
        # solve_function(q, e, f, t, rhs)
        # u_i - delta_t a_ii G(t^n + c_i delta_t, u_i) = rhs
        time = t_old + self.c[stage_num] * delta_t
        e = -1.0 * delta_t * self.a[stage_num, stage_num]
        return solve_operator(
            1.0,
            e,
            time,
            stage_rhs,
            q_old,
            t_old,
            delta_t,
            implicit_operator,
            stages,
            stage_num,
```

```
        )


# TODO: add more description of these methods and where they come from
class IMEX1(IMEXRungeKutta):
    def __init__(self):
        ap = np.array([[0.0]])
        bp = np.array([1.0])
        cp = np.array([0.0])

        a = np.array([[1.0]])
        b = np.array([1.0])
        c = np.array([1.0])

        IMEXRungeKutta.__init__(self, a, b, c, ap, bp, cp)


# TODO: Could add other IMEX schemes of 2 and 3 order
class IMEX2(IMEXRungeKutta):
    def __init__(self):
        ap = np.array([[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
        bp = np.array([0.0, 0.5, 0.5])
        cp = np.array([0.0, 0.0, 1.0])

        a = np.array([[0.5, 0.0, 0.0], [-0.5, 0.5, 0], [0.0, 0.5, 0.5]])
        b = np.array([0.0, 0.5, 0.5])
        c = np.array([0.5, 0.0, 1.0])

        IMEXRungeKutta.__init__(self, a, b, c, ap, bp, cp)


class IMEX3(IMEXRungeKutta):
    def __init__(self):
        alpha = 0.24169426078821
        beta = 0.06042356519705
        eta = 0.1291528696059
        zeta = 0.5 - beta - eta - alpha

        ap = np.array(
            [
                [0.0, 0.0, 0.0, 0.0],
                [0.0, 0.0, 0.0, 0.0],
                [0.0, 1.0, 0.0, 0.0],
                [0.0, 0.25, 0.25, 0.0],
            ]
        )
        bp = np.array([0.0, 1.0 / 6.0, 1.0 / 6.0, 2.0 / 3.0])
        cp = np.array([0.0, 0.0, 1.0, 0.5])

        a = np.array(
            [
                [alpha, 0.0, 0.0, 0.0],
                [-alpha, alpha, 0.0, 0.0],
                [0.0, 1.0 - alpha, alpha, 0.0],
                [beta, eta, zeta, alpha],
```

```
            ]
        )
        b = np.array([0.0, 1.0 / 6.0, 1.0 / 6.0, 2.0 / 3.0])
        c = np.array([alpha, 0.0, 1.0, 0.5])

        IMEXRungeKutta.__init__(self, a, b, c, ap, bp, cp)
```

```
def evaluate_weak_form(
    dg_solution,
    numerical_fluxes,
    quadrature_function,
    vector_left,
    vector_right,
    source_quadrature_function=None,
):
    mesh_ = dg_solution.mesh
    basis_ = dg_solution.basis

    num_elems = mesh_.num_elems
    num_basis_cpts = basis_.num_basis_cpts

    transformed_solution = solution.DGSolution(
        np.zeros((num_elems, num_basis_cpts)), basis_, mesh_
    )
    for i in range(num_elems):
        left_face_index = mesh_.elems_to_faces[i, 0]
        right_face_index = mesh_.elems_to_faces[i, 1]
        transformed_solution[i, :] = (
            1.0
            / mesh_.elem_metrics[i]
            * (
                quadrature_function(i)
                - (
                    numerical_fluxes[right_face_index] * vector_right
                    - numerical_fluxes[left_face_index] * vector_left
                )
            )
        )

        if source_quadrature_function is not None:
            transformed_solution[i, :] += (
                1.0 / mesh_.elem_metrics[i] * source_quadrature_function(i)
            )

    return transformed_solution
def dg_weak_form_matrix(
    basis_,
    mesh_,
    t,
    boundary_condition,
    numerical_flux,
    quadrature_matrix_function,
    source_quadrature_function=None,
):
    num_basis_cpts = basis_.num_basis_cpts
```

```
num_elems = mesh_.num_elems
# matrix size
n = num_basis_cpts * num_elems

L = np.zeros((n, n))
S = np.zeros(n)

phi1 = basis_.evaluate(1.0)
phim1 = basis_.evaluate(-1.0)

C11 = np.matmul(basis_.mass_matrix_inverse, np.outer(phi1, phi1))
C1m1 = np.matmul(basis_.mass_matrix_inverse, np.outer(phi1, phim1))
Cm11 = np.matmul(basis_.mass_matrix_inverse, np.outer(phim1, phi1))
Cm1m1 = np.matmul(basis_.mass_matrix_inverse, np.outer(phim1, phim1))

# iterate over all the rows of the matrix
for i in range(num_elems):
    left_elem_index = mesh_.get_left_elem_index(i)
    right_elem_index = mesh_.get_right_elem_index(i)
    left_face_index = mesh_.elems_to_faces[i, 0]
    right_face_index = mesh_.elems_to_faces[i, 1]

    m_i = mesh_.elem_metrics[i]

    x_left = mesh_.get_face_position(left_face_index)
    tuple_ = numerical_flux.linear_constants(x_left, t)
    c_l_imh = tuple_[0]
    c_r_imh = tuple_[1]

    x_right = mesh_.get_face_position(right_face_index)
    tuple_ = numerical_flux.linear_constants(x_right, t)
    c_l_iph = tuple_[0]
    c_r_iph = tuple_[1]

    indices_i = solution.vector_indices(i, num_basis_cpts)

    # 1/m_i(B_i - C11_i + Cm1m1_i) Q_i
    L[indices_i, indices_i] = (1.0 / m_i) * (
        quadrature_matrix_function(i) - c_l_iph * C11 + c_r_imh * Cm1m1
    )

    # S = S_i
    if source_quadrature_function is not None:
        S[indices_i] = (1.0 / m_i) * source_quadrature_function(i)

    # check boundary
    if left_elem_index != -1:
        indices_l = solution.vector_indices(left_elem_index, num_basis_cpts)
        # 1/m_i Cm11_i Q_{i-1}
        L[indices_i, indices_l] = (1.0 / m_i) * c_l_imh * Cm11

    if right_elem_index != -1:
        indices_r = solution.vector_indices(right_elem_index, num_basis_cpts)
        # -1/m_i C1m1_i Q_{i+1}
        L[indices_i, indices_r] = (-1.0 / m_i) * c_r_iph * C1m1
```

```
    # Do boundary conditions
    for i in mesh_.boundary_faces:
        tuple_ = boundary_condition.evaluate_boundary_matrix(
            mesh_, basis_, i, numerical_flux, t, L, S
        )
        L = tuple_[0]
        S = tuple_[1]

    return (L, S)
def compute_quadrature_weak(dg_solution, t, flux_function, i):
    basis_ = dg_solution.basis
    result = np.zeros(basis_.num_basis_cpts)

    # if first order then will be zero
    # phi_xi(xi) = 0 for 1st basis_cpt
    if basis_.num_basis_cpts == 1:
        return 0.0

    for l in range(basis_.num_basis_cpts):

        def quadrature_function(xi):
            x = dg_solution.mesh.transform_to_mesh(xi, i)
            q = dg_solution.evaluate_canonical(xi, i)
            phi_xi = basis_.evaluate_gradient_canonical(xi, l)
            return flux_function(q, x, t) * phi_xi

        result[l] = math_utils.quadrature(quadrature_function, -1.0, 1.0)

    result = np.matmul(basis_.mass_matrix_inverse, result)

    return result
def compute_quadrature_matrix_weak(basis_, mesh_, t, flux_function, i):
    num_basis_cpts = basis_.num_basis_cpts

    B = np.zeros((num_basis_cpts, num_basis_cpts))
    for j in range(num_basis_cpts):
        for k in range(num_basis_cpts):

            def quadrature_function(xi):
                x = mesh_.transform_to_mesh(xi, i)
                phi_xi_j = basis_.evaluate_gradient_canonical(xi, j)
                phi_k = basis_.evaluate_canonical(xi, k)
                a = flux_function.q_derivative(0.0, x, t)
                return a * phi_xi_j * phi_k

            B[j, k] = math_utils.quadrature(quadrature_function, -1.0, 1.0)

    B = np.matmul(basis_.mass_matrix_inverse, B)
    return B
```

```
def operator(
    dg_solution,
    t,
    diffusion_function=None,
```

```
    source_function=None,
    q_boundary_condition=None,
    r_boundary_condition=None,
    s_boundary_condition=None,
    u_boundary_condition=None,
    q_numerical_flux=None,
    r_numerical_flux=None,
    s_numerical_flux=None,
    u_numerical_flux=None,
    quadrature_matrix_function=None,
):
    (
        diffusion_function,
        source_function,
        q_boundary_condition,
        r_boundary_condition,
        s_boundary_condition,
        u_boundary_condition,
        q_numerical_flux,
        r_numerical_flux,
        s_numerical_flux,
        u_numerical_flux,
        quadrature_matrix_function,
    ) = get_defaults(
        dg_solution,
        t,
        diffusion_function,
        source_function,
        q_boundary_condition,
        r_boundary_condition,
        s_boundary_condition,
        u_boundary_condition,
        q_numerical_flux,
        r_numerical_flux,
        s_numerical_flux,
        u_numerical_flux,
        quadrature_matrix_function,
    )

    basis_ = dg_solution.basis
    mesh_ = dg_solution.mesh
    Q = dg_solution

    # Frequently used constants
    # M^{-1} S^T
    quadrature_matrix = -1.0 * basis_.mass_inverse_stiffness_transpose
    # M^{-1} \Phi(1.0)
    vector_right = np.matmul(basis_.mass_matrix_inverse, basis_.evaluate(1.0))
    # M^{-1} \Phi(-1.0)
    vector_left = np.matmul(basis_.mass_matrix_inverse, basis_.evaluate(-1.0))

    quadrature_function = dg_utils.get_quadrature_function_matrix(Q,
        quadrature_matrix)
    FQ = dg_utils.evaluate_fluxes(Q, t, q_boundary_condition, q_numerical_flux)
    R = dg_utils.evaluate_weak_form(
```

```
        Q, FQ, quadrature_function, vector_left, vector_right
    )

    quadrature_function = dg_utils.get_quadrature_function_matrix(R,
        quadrature_matrix)
    FR = dg_utils.evaluate_fluxes(R, t, r_boundary_condition, r_numerical_flux)
    S = dg_utils.evaluate_weak_form(
        R, FR, quadrature_function, vector_left, vector_right
    )

    quadrature_function = dg_utils.get_quadrature_function_matrix(S,
        quadrature_matrix)
    FS = dg_utils.evaluate_fluxes(S, t, s_boundary_condition, s_numerical_flux)
    U = dg_utils.evaluate_weak_form(
        S, FS, quadrature_function, vector_left, vector_right
    )

    # quadrature_function(i) = B_i * U_i
    quadrature_function = ldg_utils.get_quadrature_function(
        U, quadrature_matrix_function
    )

    # source_quadrature_function
    if isinstance(source_function, flux_functions.Zero):
        source_quadrature_function = None
    else:
        source_quadrature_function = dg_utils.get_source_quadrature_function(
            source_function, basis_, mesh_, t
        )

    FU = dg_utils.evaluate_fluxes(U, t, u_boundary_condition, u_numerical_flux)
    L = dg_utils.evaluate_weak_form(
        U,
        FU,
        quadrature_function,
        vector_left,
        vector_right,
        source_quadrature_function,
    )

    return L


def matrix(
    dg_solution,
    t,
    diffusion_function=None,
    source_function=None,
    q_boundary_condition=None,
    r_boundary_condition=None,
    s_boundary_condition=None,
    u_boundary_condition=None,
    q_numerical_flux=None,
    r_numerical_flux=None,
    s_numerical_flux=None,
```

```
    u_numerical_flux=None,
    quadrature_matrix_function=None,
):

    (
        diffusion_function,
        source_function,
        q_boundary_condition,
        r_boundary_condition,
        s_boundary_condition,
        u_boundary_condition,
        q_numerical_flux,
        r_numerical_flux,
        s_numerical_flux,
        u_numerical_flux,
        quadrature_matrix_function,
    ) = get_defaults(
        dg_solution,
        t,
        diffusion_function,
        source_function,
        q_boundary_condition,
        r_boundary_condition,
        s_boundary_condition,
        u_boundary_condition,
        q_numerical_flux,
        r_numerical_flux,
        s_numerical_flux,
        u_numerical_flux,
        quadrature_matrix_function,
    )

    basis_ = dg_solution.basis
    mesh_ = dg_solution.mesh

    # quadrature_matrix_function, B_i = M^{-1}\dintt{-1}{1}{a(xi) \Phi_xi \Phi^T}{xi
        }
    # for these problems a(xi) = -1.0 for r, s, and u equations
    quadrature_matrix = -1.0 * basis_.mass_inverse_stiffness_transpose
    const_quadrature_matrix_function = dg_utils.
        get_quadrature_matrix_function_matrix(
        quadrature_matrix
    )

    # r - q_x = 0
    # R = A_r Q + V_r
    tuple_ = dg_utils.dg_weak_form_matrix(
        basis_,
        mesh_,
        t,
        q_boundary_condition,
        q_numerical_flux,
        const_quadrature_matrix_function,
    )
    r_matrix = tuple_[0]
```

```
    r_vector = tuple_[1]

    # s - r_x = 0
    # S = A_s R + V_s
    tuple_ = dg_utils.dg_weak_form_matrix(
        basis_,
        mesh_,
        t,
        r_boundary_condition,
        r_numerical_flux,
        const_quadrature_matrix_function,
    )
    s_matrix = tuple_[0]
    s_vector = tuple_[1]

    # u - s_x = 0
    # U = A_u S + V_u
    tuple_ = dg_utils.dg_weak_form_matrix(
        basis_,
        mesh_,
        t,
        s_boundary_condition,
        s_numerical_flux,
        const_quadrature_matrix_function,
    )
    u_matrix = tuple_[0]
    u_vector = tuple_[1]

    # source_quadrature_function
    if isinstance(source_function, flux_functions.Zero):
        source_quadrature_function = None
    else:
        source_quadrature_function = dg_utils.get_source_quadrature_function(
            source_function, basis_, mesh_, t
        )

    # l + (q^3 u)_x = 0
    # L = A_l U + V_l
    tuple_ = dg_utils.dg_weak_form_matrix(
        basis_,
        mesh_,
        t,
        u_boundary_condition,
        u_numerical_flux,
        quadrature_matrix_function,
        source_quadrature_function
    )
    l_matrix = tuple_[0]
    l_vector = tuple_[1]

    # R = A_r Q + V_r
    # S = A_s R + V_s = A_s(A_r Q + V_r) + V_S = A_s A_r Q + A_s V_r + V_s
    # U = A_u S + V_u = A_u(A_s A_r Q + A_s V_r + V_s) + V_u
    #   = A_u A_s A_r Q + A_u (A_s V_r + V_s) + V_u
    # L = A_l U + V_l = A_l (A_u A_s A_r Q + A_u (A_s V_r + V_s) + V_u) + V_l
```

```
    #    = A_l A_u A_s A_r Q + A_l(A_u (A_s V_r + V_s) + V_u)) + V_l
    matrix = np.matmul(l_matrix, np.matmul(u_matrix, np.matmul(s_matrix, r_matrix)))
    vector = (
        np.matmul(
            l_matrix,
            np.matmul(u_matrix, np.matmul(s_matrix, r_vector) + s_vector) + u_vector
                ,
        )
        + l_vector
    )
    return (matrix, vector)
```