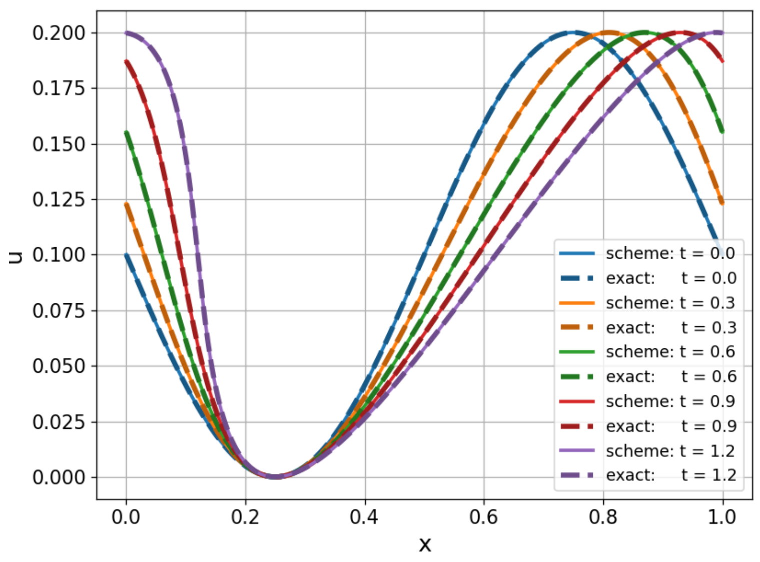
Accompanying Report for the Development and Testing of a Hyperbolic Conservation Law PDE Solver Software Package in Python

maxrob27 on `github.com`



1 Introduction

Hyperbolic conservation laws arise frequently in the fields of mathematics, science and engineering. These partial differential equations (PDEs) involve the conservation of quantities, such as mass or energy. In addition, their hyperbolic nature with characteristic information propagation invokes rich mathematical content.

In this report, we implement selected numerical solvers for hyperbolic conservation laws. We develop, explain, test and evaluate a Python package in order to achieve this goal. The functionality of this package is tailored for the implementation of further features in the future.

First, the mathematical background is presented in section 2. The appropriate numerical schemes with their implementation are explained in section 3. Thereafter, the package code is explained in section 4, along with the corresponding flows and structure. Importantly, verification of the package is detailed in section 5. Finally, an example to highlight the uses of the package is presented in section 6.

2 Mathematical Background

The main focus of this report is on the implementation of numerical schemes for hyperbolic conservation laws of the form:

$$\frac{\partial}{\partial t}u(x,t) + \frac{\partial}{\partial x}f(u(x,t)) = 0, \quad (x,t) \in \Omega_x \times \Omega_t. \tag{2.1}$$

We define the domain: $\Omega_x \in [x_0, x_1]$ and $\Omega_t \in [0, T]$. The variable u represents the density function where the conserved quantity, $\int_{-\infty}^{+\infty} u(x,t)\,\mathrm{d}x$, is constant with respect to time (t). The function f is known as the flux function [4]. The initial conditions are given by $u(x,0) = u_0(x)$ and the boundary conditions must be specified depending on the problem at hand. For simplicity, we consider only the one-dimensional case. Hyperbolic conservation laws often lead to the formation of shocks, a discontinuity in the solution within the domain. For this reason, extreme care is required when shocks form.

We employ the linear advection equation to demonstrate the case in which the flux function is linear, $f(u) = c\,u$, where c is the wavespeed:

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0, \;\; (x,t) \in \Omega_x \times \Omega_t. \tag{2.2}$$

For the nonlinear case, we examine Burgers' equation, where $f(u) = u^2/2$:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0, \;\; (x,t) \in \Omega_x \times \Omega_t \tag{2.3}$$

2.1 Linear Advection Equation

The linear advection equation (equation 2.2) is a simple form of a conservation law. This will therefore provide a suitable starting point for the mathematics and coding. It will allow the basis upon which further equations may be analysed and implemented.

Through the use of the method of characteristics, one finds that the characteristics of the PDE are linear and therefore do not naturally form shocks, given a smooth initial condition. These characteristics travel at a speed c (in positive x-direction). Hence, the initial condition, $u_0(x)$, is simply propagated to the right at a speed c. The analytical solution is therefore $u(x,t) = u_0(x - c\,t)$.

2.2 Burgers' Equation

A similar analysis may be performed on the inviscid Burgers' equation (equation 2.3). The key difference now is the formation of shocks, given non-zero initial conditions. The method of characteristics yields $u(x,t) = u_0(x - u(x,t)\,t)$ which is an implicit equation that may be solved for $u(x,t)$ using Newton's method or another nonlinear solver. This implicit equation implies that the speed and direction at which u is transported depends on the sign and magnitude of u itself. The characteristics may cross forming shocks.

3 Numerical Implementation

To solve these problems numerically, we employ the method of finite differences. We remark that studying how to solve hyperbolic PDEs numerically was not part of the core MMSC courses and that this was learned independently. This section builds upon aspects from [4] and serves to provide the mathematical background required for the package. We first clarify the mesh discretisation (section 3.1) after which the notion of conservative schemes and the Lax-Wendroff theorem are discussed (section 3.2). The selected finite difference schemes are then presented in section 3.3. This is followed by the mathematical concept of total variation diminishing schemes and the application of flux limiters (sections 3.4 and 3.5). A concluding overview is presented in section 3.6.

3.1 Discretisation

In order to remain consistent throughout the report, let us discretise the spatial domain into N_x nodes with spacing Δx between x_0 and x_1. Hence $\Delta x = (x_1 - x_0)/(N_x - 1)$. Furthermore, we discretise in time with a spacing of Δt and a (user-specified) final time $\widetilde{T}$. Due to $\widetilde{T}$ possibly not being divisible by Δt with no remainder, we often just almost reach $\widetilde{T}$, stopping at T. Given that Δt is typically very small, the difference $\widetilde{T} - T$ should be negligible. We then have N_t nodes in time where $N_t := T/\Delta t + 1$. The time at time step n is then at $t = n\,\Delta t$ where $n \in \mathbb{N}$ and $n \in [0, N_t]$.

3.2 Conservative Schemes and Lax-Wendroff Theorem

A numerical scheme is said to be conservative if it may be written in the form:

$$U_j^{n+1} = U_j^n - \frac{\Delta t}{\Delta x}\left(F_{j+1/2} - F_{j-1/2}\right), \tag{3.1}$$

where, with $p, q \in \mathbb{N}$,

$$F_{j+1/2} = F\left(U_{j-p}^n, \ldots, U_{j+q}^n\right), \tag{3.2}$$

$$F_{j-1/2} = F\left(U_{j-p-1}^n, \ldots, U_{j+q-1}^n\right), \tag{3.3}$$

and the numerical flux function (F) is consistent with the analytical flux function (f), namely $F(u, \ldots, u) = f(u)$. The Lax-Wendroff theorem then tells us that, given a conservative finite difference scheme for a conservation law, if the finite difference solution converges to a total variation diminishing function (section 3.4), then the limit function is a weak solution of the conservation law. A weak solution of a conservation law must satisfy

$$\int_0^\infty \int_{-\infty}^{+\infty} \left[\phi_t u + \phi_x f(u)\right] \mathrm{d}x\,\mathrm{d}t = -\int_{-\infty}^{\infty} \phi(x,0)u(x,0)\,\mathrm{d}x, \tag{3.4}$$

for all test functions $\phi \in C_0^1\left(\mathbb{R} \times \mathbb{R}^+\right)$. This is important because this explains that, under these conditions, a found solution will not be a non-solution and have discontinuities in the correct location moving with the correct speed. This latter statement is known as shock capturing.

3.3 Finite Difference Schemes

The schemes we consider are one-step explicit methods. This has two advantages. Firstly, the computational cost is less than for implicit methods. In addition, according to [4], implicit methods are rarely used for hyperbolic conservation laws. Secondly, no 'warm start' is required at the initial condition as we consider only one-step methods.

Warm starts refer to the calculation of intermediate time step values through alternative methods when the initial condition data does not suffice to start the specified scheme. The implemented schemes are be presented with the flux function ($f(u)$) which have been defined for each equation in section 2. The linear case allows for the schemes to be separately written in matrix form, thus increasing overall scheme efficiency.

3.3.1 Upwind Scheme

The upwind methods are implemented consistently using equation 3.1 and either $F(v,w) = f(v)$ (left upwind) or $F(v,w) = f(w)$ (right upwind). This choice is dependent on the characteristic direction. Characteristics travelling from left to right will require the left upwind scheme and vice versa in order to satisfy the CFL condition. For simplicity, the user must input the method dependent on the problem at hand. This scheme is therefore conservative and is first-order accurate in space and time.

3.3.2 Lax-Friedrichs Scheme

We may increase the spatial accuracy to second-order by using the Lax-Friedrichs scheme in conservative form. The consistent numerical flux function is defined by

$$F\left(u,v\right) = \frac{f(u)+f(v)}{2} - \frac{\Delta x}{2\Delta t}(v-u). \tag{3.5}$$

3.3.3 Lax-Wendroff Scheme

The Lax-Wendroff scheme is a second-order accurate scheme in time and space. The scheme may be defined as equation 3.1 with the consistent numerical flux function

$$F\left(u,v\right) = \frac{f(u)+f(v)}{2} - \frac{\Delta t}{2\Delta x} f'(w)(f(v)-f(u)), \;\; w = \frac{u+v}{2}. \tag{3.6}$$

Further details and a derivation may be found in [1]. In the case of linear problems, the scheme may be formulated as

$$U_j^{n+1} = U_j^n - c\,\frac{\Delta t}{2\Delta x}\left(U_{j+1}^n - U_{j-1}^n\right) + c^2\,\frac{\Delta t^2}{2\Delta x^2}\left(U_{j+1}^n - 2U_j^n + U_{j-1}^n\right). \tag{3.7}$$

For simplicity, the Richtmyer two-step Lax-Wendroff scheme, which does not require the evaluation of $f'(w)$, is implemented for nonlinear problems:

$$\begin{aligned} U_{j+1/2}^{n+1/2} &= \frac{1}{2}\left(U_j^n + U_{j+1}^n\right) - \frac{\Delta t}{2\Delta x}\left[f\left(U_{j+1}^n\right) - f\left(U_j^n\right)\right]. \\ U_j^{n+1} &= U_j^n - \frac{\Delta t}{\Delta x}\left[f\left(U_{j+1/2}^{n+1/2}\right) - f\left(U_{j-1/2}^{n+1/2}\right)\right]. \end{aligned} \tag{3.8}$$

3.4 Total Variation Diminishing Schemes

It is important to be able to distinguish which schemes are total variation diminishing (TVD) for two main reasons. The first is that the analytical and weak solutions of conservation laws follow this property and this is a feature we wish our schemes to mimic. The second reason is such that the Lax-Wendroff theorem (section 3.2) may be applied. From [4], the definition of a TVD scheme requires $TV(U^{n+1}) \leq TV(U^n)$ where

$$TV(U) = \sum_j |U_j - U_{j-1}|. \tag{3.9}$$

3.5 Flux Limiters

First-order accurate methods are TVD and typically behave well near discontinuities. Conversely, second-order methods work well in smooth regions but do not necessarily satisfy the TVD property [4]. We therefore wish to consider a scheme form in which we combine these methods to achieve a more accurate, TVD, scheme. The following explanation is adapted from [7, 4] and focuses on the application to linear problems.

We take the left upwind scheme as the lower accuracy scheme and the Lax-Wendroff scheme as the higher accuracy scheme. We rewrite equation 3.7 as

$$U_j^{n+1} = \underbrace{U_j^n - c\,\frac{\Delta t}{\Delta x}\left(U_j^n - U_{j-1}^n\right)}_{\text{Left upwind}} \underbrace{- c\,\frac{\Delta t}{2\Delta x}\left(1 - c\,\frac{\Delta t}{\Delta x}\right)\left(U_{j+1}^n - 2U_j^n + U_{j-1}^n\right)}_{\text{Anti-diffusive flux}}. \tag{3.10}$$

Given a positive wavespeed, an amended numerical flux equation reads

$$F(u, v) = cu + \phi(\theta)\,\frac{c}{2}\left(1 - c\,\frac{\Delta t}{\Delta x}\right)(v - u), \tag{3.11}$$

where we have just introduced the function $\phi(\theta)$ in front of the correction term with θ defined as

$$\theta_j = \frac{U_j^n - U_{j-1}^n}{U_{j+1}^n - U_j^n}. \tag{3.12}$$

The overarching concept is that we define the function $\phi(\theta)$ such that for regions of low smoothness (large $|\theta|$), this function increases the diffusive correctional term in order to reduce the oscillatory behaviour, namely Gibbs phenomenon, thus creating a TVD scheme. Substituting the numerical flux function (equation 3.11) into equation 3.1 gives

$$\begin{aligned} U_j^{n+1} = {} & U_j^n - c\,\frac{\Delta t}{\Delta x}\left(U_j^n - U_{j-1}^n\right) \\ & - c\,\frac{\Delta t}{2\Delta x}\left(1 - c\,\frac{\Delta t}{\Delta x}\right)\left[\phi_j(\theta_j)\left(U_{j+1}^n - U_j^n\right) - \phi_{j-1}(\theta_{j-1})\left(U_j^n + U_{j-1}^n\right)\right]. \end{aligned} \tag{3.13}$$

With $\phi = 0$ we recover the lower accuracy, first-order, (left) upwind scheme. As expected, with $\phi = 1$ we recover equation 3.10 because no change is made to the numerical flux function (equation 3.11). A wide range of literature exists on the choice of the function ϕ such that the aforementioned scheme is TVD. This is desired because we are then presented with a second-order accurate scheme which mimics the TVD property of the solution of conservation laws. Table 1 presents examples of flux limiters which are implemented into the software package.

Table 1: Flux limiter functions [1, 4].

Flux limiter	**Function**	**Restrictions**
Sweby	$\max(0, \min(\beta r, 1), \min(r, \beta))$	$0 \leq \beta \leq 1$
Chakravarthy-Osher	$\max(0, \min(r, \beta))$	$1 \leq \beta \leq 2$
Van Leer	$(r + \lvert r\rvert)/(1 + \lvert r\rvert))$	
Superbee	$\max(0, \min(1, 2r), \min(2, r))$	
Ospre	$3r(r + 1)/(2(r^2 + r + 1))$	
Centered Limiter	$\max(0, \min(2r, (2 + r)/2, 2))$	
Minmod	$\min(1, r)$	

3.6 Scheme Overview

Table 2 presents a brief overview of this section where the solver input column is added for future reference.

Table 2: Finite difference methods.

Method	**Solver Input**	**Order** Δx	Δt	**Comments**
Left Upwind	"luw"	1	1	Characteristics must travel to the right
Right Upwind	"ruw"	1	1	Characteristics must travel to the left
Lax-Friedrichs	"lfm"	2	1	
Lax-Wendroff	"lwm"	2	2	Ensure TVD with flux limiter

4 Explanation of Code

It is important to provide a clear documentation of our software package. This not only enables the developer to have a structured overview, but also allows external contributors and users to understand the process 'behind the scenes'. The structure of the software package is designed such that each process is separated and no/as little as possible code is duplicated. This former ideology is known as 'separation of concerns'.

The first key aspect is the inputs that are expected from the user (section 4.1). Then, the package structure must be clearly defined (section 4.2). Section 4.3 then details the process from user input to output. This is achieved by breaking up the code into smaller, comprehensible flows with the aid of diagrams for each section. Further details on the package's structure are documented in section 4.4 and the implementation for the user is explained in section 4.5. This section also includes an explained example script for the linear advection equation. Installation and dependencies of the package may be found in Appendix A.

4.1 User Inputs

The developed Python code is designed to take user inputs and present the output of the PDE under these input conditions. The aim is for the user to do as little as possible with any errors clearly displayed. We still stick to the linear advection equation and Burgers' equation for examples of linear and nonlinear cases. Table 3 presents the expected inputs.

Table 3: User inputs for the PDE solver package.

User Input	**Symbol**	**Python Variable**	**Description**	**Comments**
Domain	x_0	x0	Start of the domain	
	x_1	x1	End of the domain	$x_1 > x_0$
	Δt	dt	Time step spacing	$\Delta t > 0$
	T	T	Final user-specified time	$T \geq \Delta t$
	Δx	dx	Spatial spacing	Specify Δx or N_x
	N_x	Nx	Number of spatial nodes	Specify Δx or N_x
Conditions	$f(x)$	f	Initial condition	Example notation
	$g(x,t)$/pbc	g/"pbc"	Boundary conditions	Example notation
	`TOL`	tol	Consistency tolerance	
Solution	N/A	method	Solution method	See Table 2
	N/A	fluxlim	Selected flux limiter	See Table 1
	c_{wave}	c_wave	Wavespeed	Advection equation

4.2 Package Structure

The package is constructed to follow conventions of a standard Python package. The structure of the directory tree may be found below. The ST_Python file is the GitHub repository.

```
ST_Python
└── py_scicomp_package
    ├── pde_solver
    │   ├── __init__.py
    │   ├── solver
    │   │   ├── __init__.py
    │   │   ├── conditions.py
    │   │   ├── domain.py
    │   │   ├── fluxlimiters.py
    │   │   └── solvers.py
    │   ├── examples
    │   │   ├── example_advection.py
    │   │   ├── example_burgers.py
    │   │   └── example_burgers_trafficflow.py
    │   └── tests
    │       ├── unittests_conditions.py
    │       ├── unittests_discretise.py
    │       ├── unittests_solver.py
    │       ├── systemtests_advection_mms.py
    │       ├── systemtests_burgers_mms.py
    │       ├── systemtests_convergence_dx.py
    │       └── systemtests_convergence_dt.py
    └── setup.py
```

4.3 Code Flow

Now that the inputs and the package directory structure have been explained, the flow of the solution method may be detailed. However, first, the flow chart shapes must be clarified. This is presented in Figure 1.

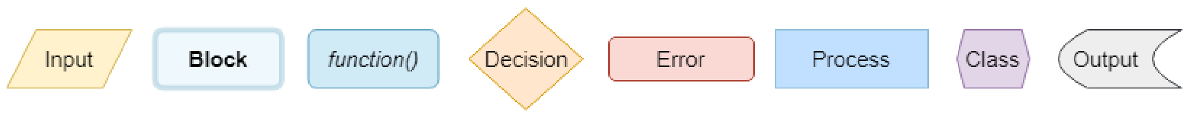


Figure 1: Flow chart diagram legend.

The overarching goal is to present the solution of the PDE on the (user-specified) domain with the (user-specified) initial and boundary conditions using the (user-specified) solution method. The main structure consists of four steps as can be seen in Figure 2.

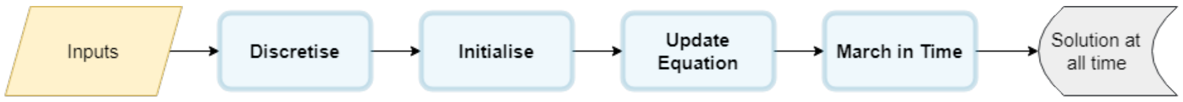


Figure 2: General overview of the solution method.

We now examine each step in closer detail.

4.3.1 Discretisation

The discretisation flow is presented in Figure 3. User inputs are required to define the domain. Careful attention is paid to the specification of Δx and N_x. Only one of these may be specified to avoid contradiction unless they are consistent. In the case of a contradiction or insufficient information, a `ValueError` is produced. Further details on this, including the relevant tests, can be found in section 5. This flow forms the class: Domain. Domain contains all the necessary information about the domain and is used frequently in other functions.

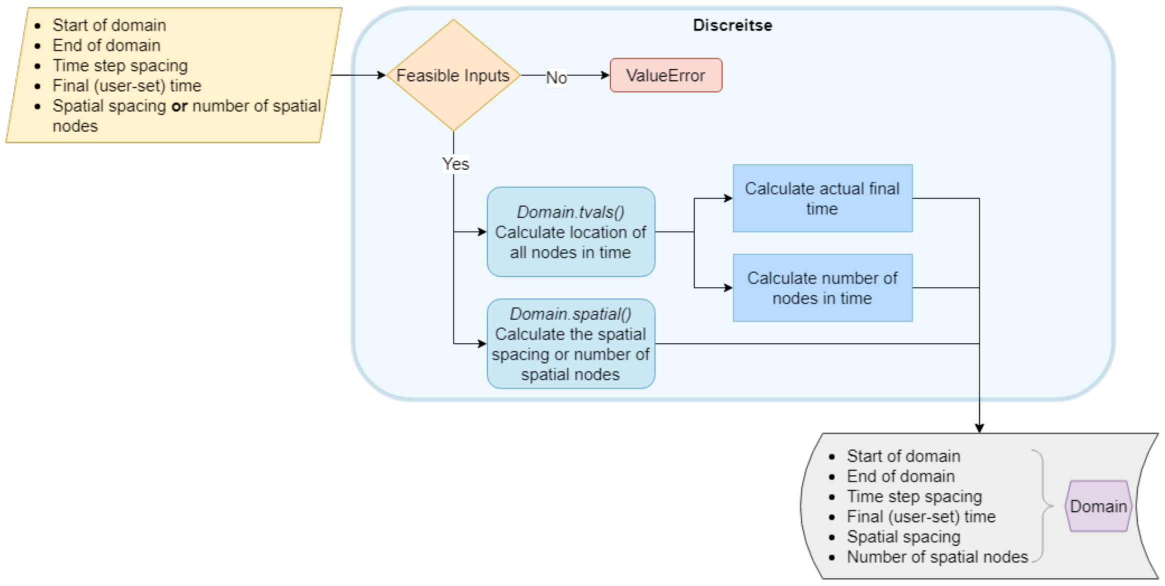
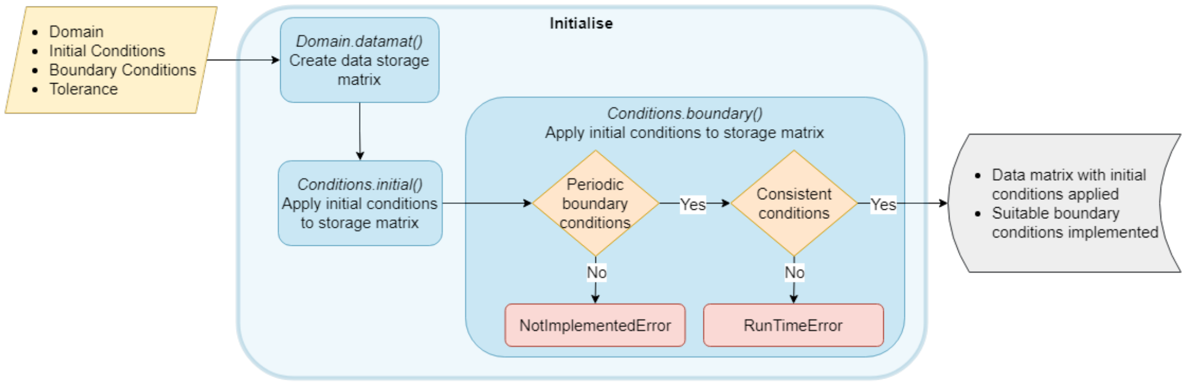

Start of domain
End of domain
Time step spacing
Final (user-set) time
Spatial spacing **or** number of spatial nodes
Discreitse
Feasible Inputs
No
ValueError
Yes
Domain.tvals() Calculate location of all nodes in time
Calculate actual final time
Calculate number of nodes in time
Domain.spatial() Calculate the spatial spacing or number of spatial nodes
Start of domain
End of domain
Time step spacing
Final (user-set) time
Spatial spacing
Number of spatial nodes
Domain

Figure 3: Discretisation flow.

4.3.2 Initialisation

The initialisation flow refers to the creation of a storage matrix with the appropriate initial and boundary conditions. This matrix will eventually contain the solution at every time step (each column). This is created using the *datamat()* function from the Domain class. The initial conditions are then implemented for the 0th time step (t = 0). This corresponds to the first column of the data matrix. Thereafter, the boundary conditions are applied. Two checks are made. First, a `NotImplementedError` is raised if non-periodic boundary conditions are input[1]. A `RunTimeError` is produced if the input conditions are not consistent with periodicity at the boundaries. This occurs if $|f(x_0) - f(x_1)| \geq$ `TOL`. This flow is presented in Figure 4.



Domain
Initial Conditions
Boundary Conditions
Tolerance
Initialise
Domain.datamat() Create data storage matrix
Conditions.initial() Apply initial conditions to storage matrix
Conditions.boundary() Apply initial conditions to storage matrix
Periodic boundary conditions
Yes
Consistent conditions
Yes
No
No
NotImplementedError
RunTimeError
Data matrix with initial conditions applied
Suitable boundary conditions implemented

Figure 4: Initialisation flow.

4.3.3 Update Equation

The update equation is a function that is created to provide the solution at time step $n + 1$, given the solution to time step n. This is the sole required input, however, to create the function further information is required (see Figure 5).

First, the class: Scheme is created. This class contains the methods that may be used. An overview is presented in Table 2. If the selected method is not found then, then a `NotImplementedError` is produced. Furthermore, if the selected method is unstable given the input parameters, a `RuntimeError` occurs. The appropriate stability requirements are taken from [1]. A successful method input creates a function which requires just the index of the previous time step in the data matrix. The method also incorporates a flux limiter (if selected) which is implemented from the Fluxlimiter class using *Fluxlimiter.fluxfunc()*. The user must input the flux limiter name as a string with the value of β (flux limiter parameter, see Table 1) appended after an underscore, e.g. `fluxlimiter="sweby_0.5"`. The *Fluxlimiter.extractbeta()* function performs input

[1]Currently the software package only supports periodic boundary conditions. Implementing other boundary conditions is possible but left for further work.

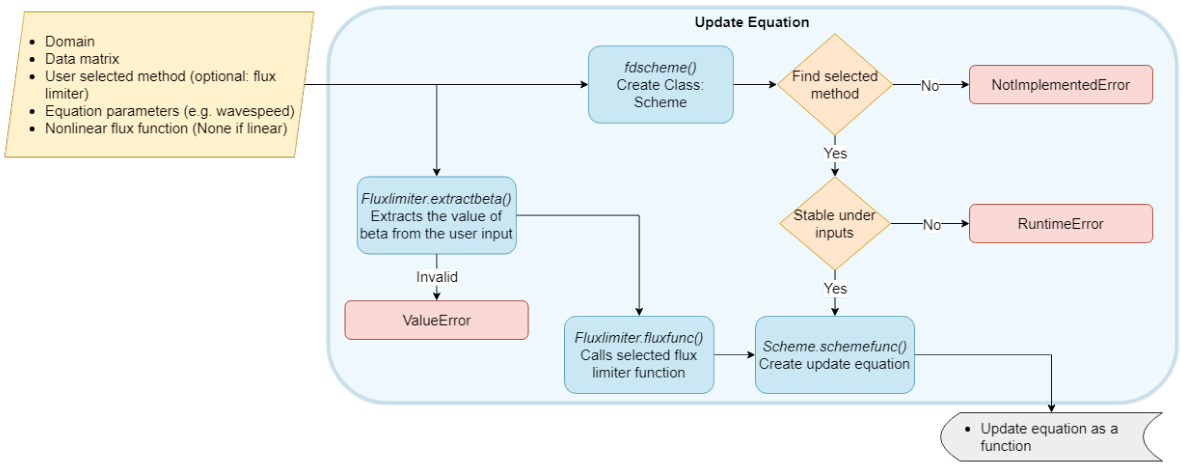

Domain
Data matrix
User selected method (optional: flux limiter)
Equation parameters (e.g. wavespeed)
Nonlinear flux function (None if linear)
Update Equation
fdscheme()
Create Class: Scheme
Find selected method
No
NotImplementedError
Yes
Fluxlimiter.extractbeta()
Extracts the value of beta from the user input
Stable under inputs
No
RuntimeError
Invalid
ValueError
Yes
Fluxlimiter.fluxfunc()
Calls selected flux limiter function
Scheme.schemefunc()
Create update equation
Update equation as a function

Figure 5: Update equation flow.

checks and extracts the value of β. An input for β is neglected if appended to a flux limiter for which the variable is not required.

4.3.4 Time Marching

Finally, the time marching flow is presented in Figure 6. Given the inputs, the function *marcher()* progresses through time, calculating and storing the function value at each time step until the final time step is reached. The full storage data matrix may then be used for analysis and plotting. The method of manufactured solutions (MMS) may be implemented here by inputting an MMS source term. Further details on this is presented in section 5.3.

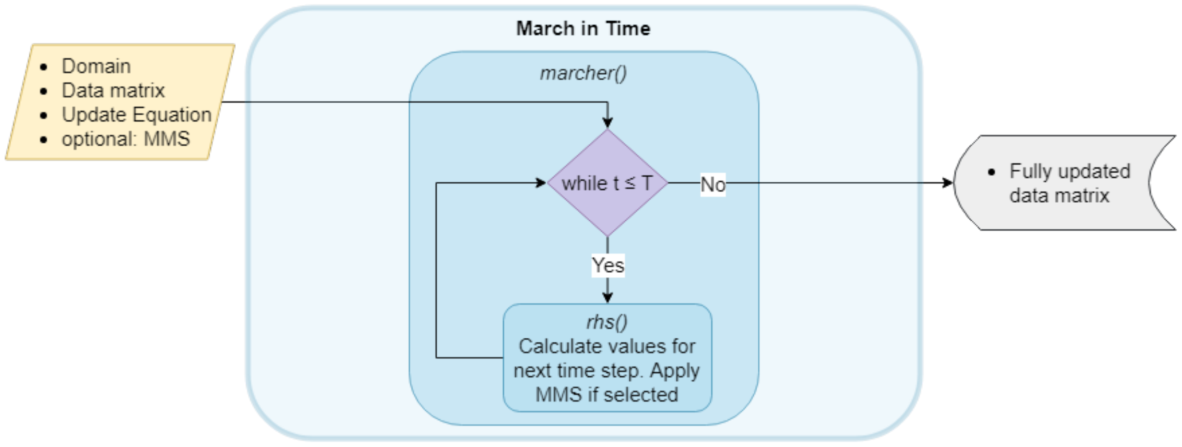


Figure 6: March flow.

The final data storage matrix is input into the class: Soln which allows for direct plotting against the exact solution using *Soln.plot()* if the solution is known. An example is given in the section 4.5.

4.4 Further details

The importance of clear structure and documentation is not to be underestimated. All functions within the package include a docstring description. For high-level functions, their inputs and outputs are listed. Where relevant, the raised errors are also documented. Furthermore, print statements are occasionally used to indicate to the user which scheme has been selected. This serves to confirm the user has correctly input the intended method. Additionally, the progress of the solver is shown in the console. From the user perspective, this gives an indication of waiting time.

A few notes on the code style are now made. Firstly, inheritance was considered but not implemented due to the very modular structure of the code flow (see Figure 2). Secondly, the code was developed in such a way as to allow for a high degree of flexibility and further development avoiding 'hard coding' where possible. An example to highlight this can be found in the *marcher()* function in which the function *rhs()* is called to update the solution passing each time step. Any suitable scheme may be added by changing the *rhs()* function. Not only does this increase simplicity of the code, but it allows for errors to be found more easily. While not required for this report, this function does also support an input for time. Thirdly, rather than calling a function, which calls another function (and so on), where possible, a function object is returned. This requires one 'initialisation run' rather than creating the scheme every time the function is required, improving code efficiency. Lastly, the use of `__str__` within classes has been implemented, where relevant, for user use and debugging efforts.

4.5 Implementation and Example Script

In order to solve the selected PDE, the required functions (initial, boundary and/or exact) must be defined. First, the domain must be initialised. This is achieved through the use of the Domain class as follows: `Domain`$(x_0, x_1, \Delta t, T, \Delta x)$. Note that N_x may be specified instead of Δx. The reader is referred to Table 3 for notation. Secondly, the initial and boundary condition functions must be specified for the solver. This can be achieved using `Conditions`(initial, boundary[2], `TOL`) where the tolerance parameter is optional and set to `TOL` $= 1\text{e}{-8}$ when not specified. The equation may now be solved by applying the function *{PDE}_eqn_solve(*Domain, Conditions, method, optional: fluxlim, equation parameters*)* where *{PDE}* must be replaced by *advection* or *burgers*. The solution may now be plotted using `Sol.plot`(number of curves, exact

[2]This is "pbc" for periodic boundary conditions

function) where `Sol` is the class returned from the solving function. The curves are shown at evenly spaced time intervals. The corresponding exact function may be optionally input as the second argument if it is known. The advantage of the solution as a class (Sol) is the ease at which the aforementioned plot may be constructed.

A very brief example script is presented for the linear advection equation. Due to space limitations, spacings and function docstrings have been shortened. This script and more may be found under the 'examples' directory within the package.

```
1  from pde_solver.solver import Domain, Conditions, advection_eqn_solve
2  import numpy as np
3  
4  def f(x):
5      """Square initial condition."""
6      IC = 0 * x
7      IC[int(len(IC)*0.35):int(len(IC)*0.55)] = 1
8      return IC
9  
10 def exactfunc(x, t):
11     """Exact solution is shifted initial conditions: f(x - c t)."""
12     n = int((c_wave*t*Grid.Nx)%Grid.Nx)
13     return np.hstack((f(x)[-n:], f(x)[:-n]))
14 
15 c_wave = 1  # Wavespeed
16 
17 Grid = Domain(0, 1, 0.001, 1, dx=0.01)  # Domain specifications
18 Conds = Conditions(f, "pbc")  # Initial and boundary conditions
19 Sol = advection_eqn_solve(Grid, Conds, method="lwm", c=c_wave)  # Solve
   ↪  with specified method
20 Sol.plot(5, exactfunc)  # Plots (at 5 evenly spaced time intervals with
   ↪  corresponding exact solution)
```

The initial condition is represented by $f(x)$ and so the exact solution is known as $f(x - c_{\text{wave}}\, t)$ which is the shifted initial condition (see section 2.1). The Domain class is initialised as `Grid` in line 17, followed by the initial and (periodic) boundary conditions in line 18 with `Conds`. The solution is then returned as a class (`Sol`) from the function *advection_eqn_solve()* (line 19). The required inputs are input within the parentheses. Finally, a plot results from line 20. The result obtained is presented by Figure 7. With a flux limiter added in parenthesis in line 19, in this case, using `fluxlim="vanleer"`, it can be seen in Figure 8 how the oscillations are clearly dampened.

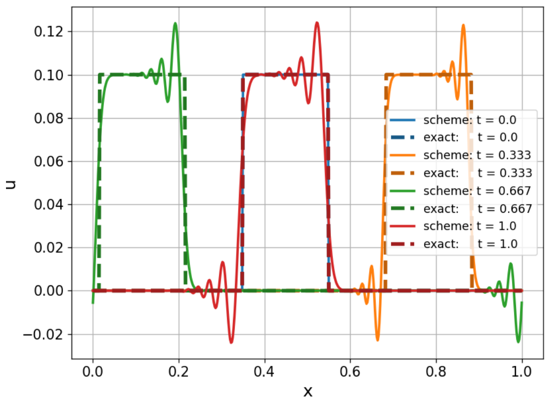


Figure 7: Output from the example script in section 4.5.

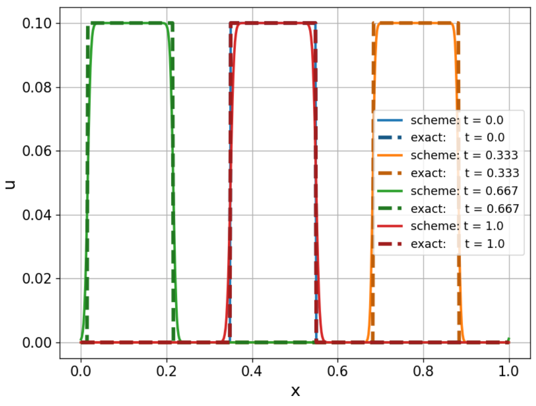


Figure 8: Output from example script with the Van Leer flux limiter.

5 Verification

A result may look pretty but this has no relation as to how correct the solution is. How can we ensure we have the correct solution? Unfortunately, this is quite difficult without automated procedures. Testing shows the incorrectness of a program, rather than the correctness of a program. More meaningful tests, however, help minimise the likelihood of existence and influence of errors. This specific testing is known as verification and is defined as:

> The process of determining that a computational model accurately represents the underlying mathematics and its solution [5].

Given a larger and more complicated system which specifically simulates a real-world problem, another process of testing would involve validation. Since the aim of this report is focused on the mathematics rather than the reflection of mathematics on a real-world problem, validation specifically is not be performed.

This verification is implemented through the frequent use of debugging, unit tests and system tests (sections 5.1, 5.2 and 5.3 respectively). Each section explains the tests in further detail and highlight key examples. Finally, further tests which do not fall into the aforementioned categories are highlighted in section 5.4. The reader is directed to Appendix B for a complete overview of all tests and testing coverage.

5.1 Debugging

During the implementation of the code, debugging was required. It was important to proceed systematically. In this case, the visual studio code debugger was used. First, the rough location of the bug would be traced using the debugger. Thereafter a line-by-line analysis of the code at that point is performed, examining the variables at each relevant step. This is known as static and dynamic analysis respectively [5]. It was occasionally found that incorrect inputs, such as x_1 being smaller than x_0 were being erroneously entered. A result of this was the implementation of automatic test functions, e.g. *checks()* in Domain.py, in order to raise `ValueError` in such instances.

5.2 Unit Tests

An appropriate starting point to performing tests on the code is to split it up into smaller units. These units may be tested separately and are widely known as 'unit tests'. The approach to unit testing for this package was to, where possible, test each function individually and occasionally parts of the function if the function was sufficiently complex. The pytest package for Python offers an efficient, comprehensible and well-documented method to perform these tests. Scripts starting with 'unittests_' within the 'tests' directory contain specific unit tests which must be passed.

These tests are designed to catch any unexpected results. For example, upon entering the attributes for the Domain class, if Δx and N_x are specified, then an inconsistent discretisation may occur. It is therefore expected to obtain a `ValueError` if both are entered. However, if they are consistent, e.g. $x_0 = 0, x_1 = 1, \Delta x = 0.01, N_x = 101$, then no error is expected. These correspond respectively to the tests in unittests_discretise.py: *test_discretise_inconsistentinputs()* and *test_discretise_consistentinputs()*. The pytest feature `with pytest.raises(Error)` was frequently used to ensure the correct error was captured. On one occasion, an incorrect error was reported which led to a small, but crucial, reconfiguration of the code structure. Smaller errors were occasionally found such as indexing. A failed test from *test_solver_incorrect()* in unittests_solver.py meant that the incorrect indexing error `[1]` was corrected to `[-1]` in the *extendvec()* function.

While there are no systematic rules to devise unit tests, a creative approach can be beneficial. Two examples used highlight this. Firstly, in certain cases, a random number would be generated to really ensure that the results were in fact correct and not so by chance. Secondly, matrices could be reconstructed from the functions and then tests performed on this reconstructed matrix. For example, for most schemes, a tridiagonal matrix is expected. This matrix can be reconstructed and then the sum of all non-tridiagonal[3] elements are expected to be zero. These examples can be found in *test_solver_correctbeta()* and *test_solver_tridiagonal()* in unittests_solver.py respectively.

5.3 System Tests

The system tests focus on the package as a whole and evaluate its response to specific inputs. Two major system tests are performed. The first is the 'method of manufactured solutions' (MMS) in which the original PDE is amended to include a source term such that the exact solution is known [2]. This then allows for the output to be directly compared to the (known) analytical solution. The second system test is to evaluate the rate of convergence. The selected schemes are expected to follow the order given by Table 2.

5.3.1 Method of Manufactured Solutions

We may first demonstrate a simple case using MMS for the linear advection equation. By implementing a source term in equation 2.2 we know that

$$u(x,t) = \sin(2\pi(x-t))\cos(a\,t), \tag{5.1}$$

is the exact solution for

$$\frac{\partial u}{\partial t} + c\,\frac{\partial u}{\partial x} = \underbrace{-a\sin(2\pi(x-t))\,\sin(a\,t)}_{\text{S(x, t): source term}}, \tag{5.2}$$

where the initial condition is given by $u(x,0)$ in equation 5.1. We have periodic boundary conditions using a domain from $x=0$ to $x=1$ and a is an arbitrary[4] constant. In the *marcher()* function, if an MMS source term is specified, it is added in the following way:

$$U^{n+1} = rhs(x, n\Delta t) + \Delta t\, S(x, n\Delta t). \tag{5.3}$$

This corresponds to Figure 6 where *rhs()* is the update equation function. Equivalently we may undertake the same treatment for Burgers' equaiton (equation 2.3). This time a more complicated function is used to demonstrate more intricate MMS functions. We find that

$$u(x,t) = \sin(\pi x)\cos(a\,t) + e^{-t}\cos(2\pi(x-0.25)), \tag{5.4}$$

[3]Care is taken for periodic boundary implementation where two other entries are non-zero.

[4]With e.g. small T, one may wish to increase a to obtain more visually changing effects.

is the exact solution for

$$\frac{\partial u}{\partial t} + u\,\frac{\partial u}{\partial x} = -e^{-t}\cos(2\pi(x-0.25)) - a\sin(a\,t)\sin(\pi x) + [e^{-t}\cos(2\pi(x-0.25)) + \cos(a\,t)\sin(\pi x)]\cdot[(\pi\cos(a\,t)\cos(\pi x) - 2e^{-t}\pi\sin(2\pi(x-0.25))], \quad (5.5)$$

over the same domain and initial conditions $u(x,0)$ (equation 5.4) with periodic boundary conditions. The procedure outlined in equation 5.3 is followed.

Given this information we obtain the plots in Figures 9 and 10 using the files systemtest_advection_mms.py and systemtest_burgers_mms.py in the 'tests' directory. The results show overlapping curves as expected. This is an indication that the numerical scheme is working as intended and the MMS implementation also functions correctly.

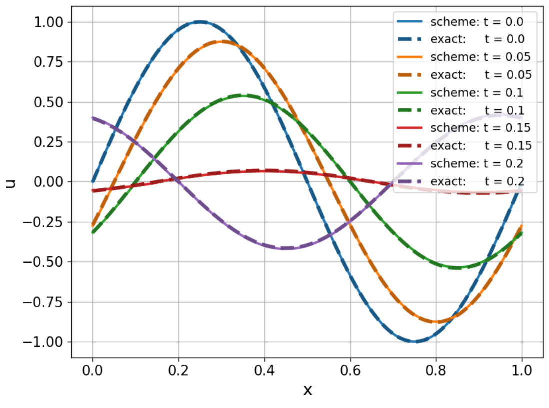


Figure 9: MMS of the linear advection equation (equation 5.2) from $t=0$ *to* $t=1$ *with* $\Delta x = \Delta t = 0.001, a = 10$.

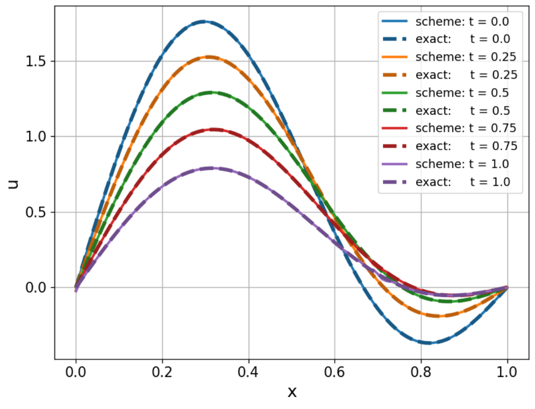


Figure 10: MMS of Burgers equation (equation 5.5) from $t=0$ *to* $t=1$ *with* $\Delta x = \Delta t = 0.001, a = 1$.

5.3.2 Convergence Rates

Our second major system test involves the inspection of convergence rates to ensure the scheme has been correctly implemented. These rates are expected to follow from the information provided in Table 2. For this, we require smooth initial conditions otherwise the order of accuracy may be decreased. This is understood by the reduction of the order of accuracy of the solution at non-smooth regions of the solution, such as shocks [4]. We apply the infinity norm for error measurement. For spatial refinement we obtain Figure 11 (using systemtest_convergence_dx.py). For temporal and spatial refinement we obtain Figure 12 (using systemtest_convergence_dt.py). For the latter figure, the minimum spatial or temporal order of convergence is expected. As the temporal order of convergence for the selected schemes is always less than or equal to the spatial order of convergence, we may therefore use this to investigate the temporal convergence. Simply refining Δt often leads to instabilities and diffusive effects and is therefore not considered.

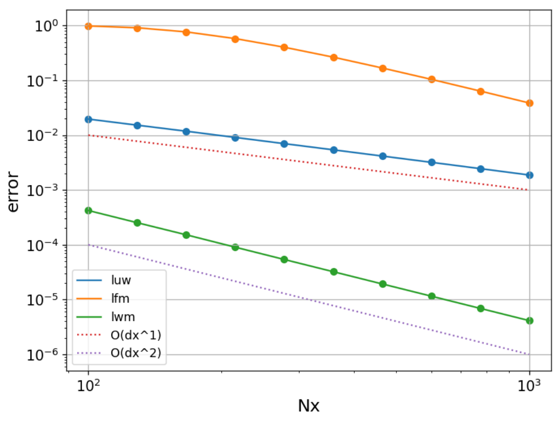


Figure 11: Spatial mesh refinement for the methods from Table 2.

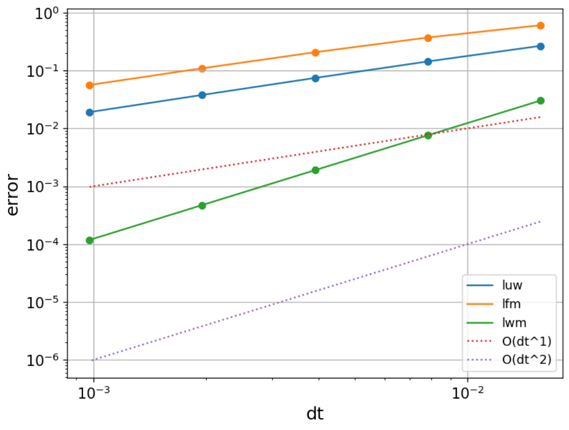


Figure 12: Spatial and temporal mesh refinement for the methods from Table 2.

It can be seen that the schemes follow their respective order. The Lax-Friedrichs scheme converges to the expected convergence rater slower due to the scheme's diffusive effects for a small time step. The correct convergence rate is, however, still achieved.

5.4 Further Tests

In addition, 'sense tests' were continuously performed during the development. This involved logically reasoning the feasibility of the solution. To highlight one example, for the linear advection equation, the solution is expected to move in the direction of sign(c). Three further test instances highlight the verification procedure. As mentioned in section 4.4, the console would print the progress of the solver. During the code development, this was also used as an indication for unexpected changes. For example, on one occasion the time to solve the PDE was significantly increased. Following the debugging steps outlined in section 5.1, it was found that an unnecessary function evaluation was performed. This was subsequently changed. In addition, manually changing the ϕ function in equation 3.13 to output zeros should yield the same result as the left upwind scheme. This was found to be the case. Finally, an extra check was implemented after an invalid input once gave unexpected results. This additional check prints any inputs that were not used or needed for the solution. While this would not change the result, it indicated inconsistencies with user inputs and in some cases, this led to further input errors being found.

6 Software Package Example Application

In order to demonstrate the functionality of the Python package, an example is given in the context of traffic flow. Whilst there exist many books on this subject (see e.g. [8]), the main focus of this report is on the development and verification of a Python package, and so, only an introductory example is given here. The Lighthill-Whitham-Richards model for traffic flow is given by

$$\rho_t + (\rho v(\rho))_x = 0, \quad v(\rho) = v_{\max}\left(1 - \frac{\rho}{\rho_{\max}}\right), \quad 0 \le \rho \le \rho_{\max}, \tag{6.1}$$

where ρ represents the vehicle density (number of vehicles per unit distance) and v represents the vehicle velocity (unit distance per unit time) [3]. The variables $\rho_{\max}$ and $v_{\max}$ denote the vehicle density at which there is essentially no gap between cars and maximum driving speed respectively. Through the transformation $\rho = \rho_{\max}(1-u)/2$ and a suitable nondimensionalisation, we may arrive at Burgers' equation (equation 2.3).

We now turn to an example in which a 10-mile stretch of a one way, two lane, road has a closed lane between miles 4 and 7. The initial conditions for this closure are represented by $\rho = \rho_1 \, \forall \, x \in [4, 7]$ and everywhere else $\rho = \rho_2$, noting that $\rho_1 > \rho_2$. By using the aforementioned transformation, and given real-world inputs collected in 2019 from [6], the PDE solver produces the outputs presented by Figures 13 and 14. The value of $\rho_{\max}$ is increased for the latter figure.

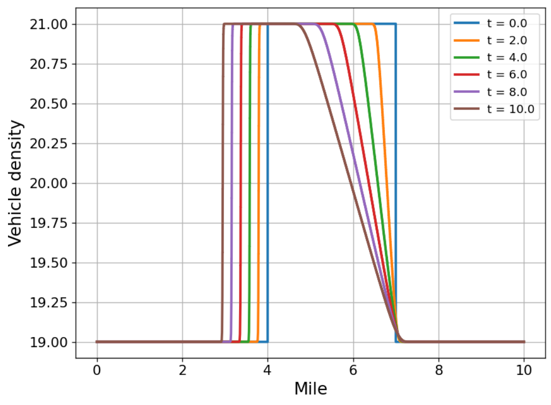


Figure 13: Vehicle density from $t = 0$ to $t = 10$ with $\Delta x = \Delta t = 0.001, \rho_{max} = 38, \rho_1 = 19, \rho_2 = 21$.

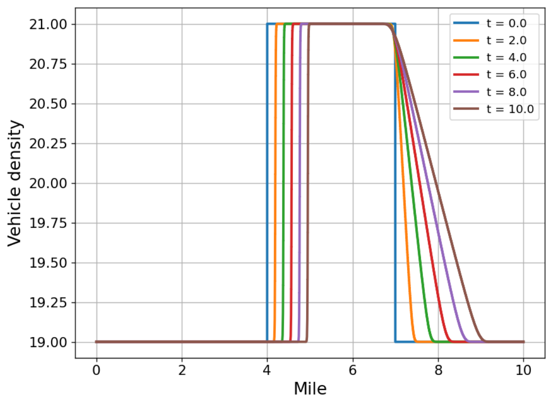


Figure 14: Vehicle density from $t = 0$ to $t = 10$ with $\Delta x = \Delta t = 0.001, \rho_{max} = 42, \rho_1 = 19, \rho_2 = 21$.

In Figure 13, it can be seen that the tail end of the shock moves backwards as the maximum density ($\rho_{\max}$) is not sufficient to reduce the backlog. Figure 14 shows that an increased maximum density allows for the tail shock to move forwards. This increased maximum density corresponds to a situation in which vehicles drive closer to each other. Intuitively this makes sense and can be seen as an initial form of validation[5]. Hence, the critical maximum density lies in $\rho_{\max} \in (38, 42)$. The plots for the quantity u and vehicle speed (separately) may be found in Appendix C. A final note is made on the boundary conditions. While periodic boundary conditions are implemented, they do not affect the outcome of the results as the changes occur away from the boundaries. This example may be found as example_burgers_trafficflow.py in the 'examples' directory.

7 Conclusion

This report has described the implementation of selected schemes to solve hyperbolic conservation laws through the development of a Python package. Tests have been described to verify the code as far as possible. These include unit and system tests. These have been passed and an applicable example has shown an instance of the use of the Python package.

This package will be further used and adapted hereafter. Hence, there are aspects which are still to be implemented, such as non-periodic boundary conditions. Furthermore, the nonlinear scheme structure was designed such that the conservation flux function may be easily changed. Therefore, further equations may be implemented aside from the linear advection equation and Burgers' equation. Stability requirements must, however, be implemented accordingly.

[5]Validation is not the aim of this report and is hence not discussed further.

References

[1] Hesthaven, J. S. (2017). *Numerical Methods for Conservation Laws: From Analysis to Algorithms.* SIAM, Philadelphia PA. ISBN: 978-1-61197-509-3.

[2] Hulshoff, S. J. (2019). *Computational modelling lecture notes.* Delft University of Technology, Delft.

[3] Jüngel, A. (2002). *Modeling and numerical approximation of traffic flow problems lecture notes.* Universität Mainz, Mainz.

[4] LeVeque, R. J. (1992). *Numerical Methods for Conservation Laws.* Springer, Basel. ISBN: 978-3-0348-5116-9.

[5] Mooij, E., Papp, Z. & van der Wal, W. (2021). *Simulation, verification and validation lecture notes.* Delft University of Technology, Delft.

[6] O'Rourke, C., Gradel, V., Robertson, M. & Alqattan, H. (2019). *Speed collection and analysis lab report.* Embry-Riddle Aeronautical University, Daytona FL.

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

[8] Treiber, M. & Kesting, A. (2013). *Traffic Flow Dynamics: Data, Models and Simulation.* Springer, Berlin. ISBN: 978-91-637-4473-0.

A Package Installation and Dependencies

Please note that in order to run the examples and tests in their respective directories, the package will need to be installed (using the below) or be activated in the virtual environment. We present the command lines on the Windows OS. Commands may need to be altered for Linux or Mac OS.

A.1 Package Installation

The package may be installed from the command line (or virtual environment setting) using `pip install {path/to/package}/py_scicomp_package` or otherwise.

A.2 Example and Test File Execution

The simplest way is to run the example and test files is from the command line using `python {path/to/file}/{filename}.py`. This includes the files containing the pytest tests (where `python` is replaced by `pytest`). It is again stressed that the package should be installed.

A.3 Dependencies

This package has the following dependencies:

- numpy==1.23.4 (`https://numpy.org/`)
- matplotlib==3.7.1 (`https://matplotlib.org/`)
- pytest==6.2.4 (`https://docs.pytest.org/en/6.2.x/`)

B Package Functions and Testing

The following tables describe the functions within the package and also how they were tested. This is in conjunction with section 5. The file which contains the unit test(s) for the specific function is indicated. The unit test description 'Line-by-line' means that the function was inspected examining each line using the debugger (similar to section 5.1) where each variable could be inspected. This was generally undertaken for small and/or simple functions. The functions used to generate the system tests (section 5.3) are labelled as 'covered'. Where the system test reads 'separate', this indicates that further tests were undertaken that may not be presented in this report. For example, the left upwind scheme was used for the order convergence plot in Figures 11 and 12 and the same result was found for the right upwind scheme. Where not applicable, the function will have been covered within the unit test.

Table 4: Package functions and testing in solver.py.

Function	**Description**	**Unit test**	**System test**
plot()	Plots solution between ‘t=0’ and ‘t=T’	Line-by-line	covered
schemefunc()	Creates the update function (func) for the finite difference scheme	unittest_solver.py	covered
lwmpbc()	Lax-Wendroff method (periodic)	Line-by-line	covered
lfmpbc()	Lax-Friedrichs Method (periodic)	Line-by-line	covered
luwpbc()	Left upwind method (periodic)	Line-by-line	covered
ruwpbc()	Right upwind method (periodic)	Line-by-line	separate
requiredinputs()	Performs a check that all required inputs have been provided	unittest_solver.py	N/A
initialise()	Initialises storage matrix and applies PDE conditions	unittest_solver.py	covered
marcher()	Marches forward in time from ‘t=0’ to ‘t=T’	Line-by-line	covered
extendvec()	Extends input vector by one either end peridodically	unittest_solver.py	covered
fluxlimiter()	Implements a flux limiter in the solution	unittest_solver.py	covered
fdscheme()	Creates the update function (func) for the finite difference scheme	Line-by-line	covered

Table 5: Package functions and testing in domain.py.

Function	**Description**	**Unit test**	**System test**
checks()	Performs user input checks. Raises errors for infeasible inputs	unittest_discretise.py	N/A
spatial()	Returns 'dx' and 'Nx' under given parameters	unittest_discretise.py	covered
xvals()	Returns array of x values with Nx points from x0 to x1	unittest_discretise.py	covered
tvals()	Returns array of t values with spacing dt from 0 to T	unittest_discretise.py	covered
datamat()	Returns empty data matrix of dimension: Nx by len(tvals)	Line-by-line	covered

Table 6: Package functions and testing in conditions.py.

Function	**Description**	**Unit test**	**System test**
initial()	Applies initial condition to data storage matrix	unittest_conditions.py	covered
boundary()	Applies boundary condition to data storage matrix	unittest_conditions.py	covered

Table 7: Package functions and testing in fluxlimiter.py.

Function	**Description**	**Unit test**	**System test**
extractbeta()	Extracts the value of β from the user input	unittest_solver.py	separate
fluxfunc()	Implements a flux limiter in the solution	Line-by-line	covered
vanleer()	Van Leer flux limiter	Line-by-line	covered
superbee()	Superbee flux limiter	Line-by-line	covered
ospre()	Ospre flux limiter	Line-by-line	covered
centredlimiter()	Monotised central flux limiter	Line-by-line	covered
minmod()	Minmod flux limiter	Line-by-line	covered
sweby()	Sweby flux limiter ($0 \leq \beta \leq 1$)	Line-by-line	covered
chakosher()	Chakravarthy-Osher flux limiter ($1 \leq \beta \leq 2$)	Line-by-line	covered

C Further Results on Traffic Flow Example

These results accompany section 6. The variable u is the transformed variable to achieve the form of Burgers' equation. Figures 15 and 16 show these intermediary results. Finally, Figures 17 and 18 show the corresponding vehicle speeds which may be calculated using equation 6.1. As expected, the speeds are lower for the higher density section and gradually increase at the this front.

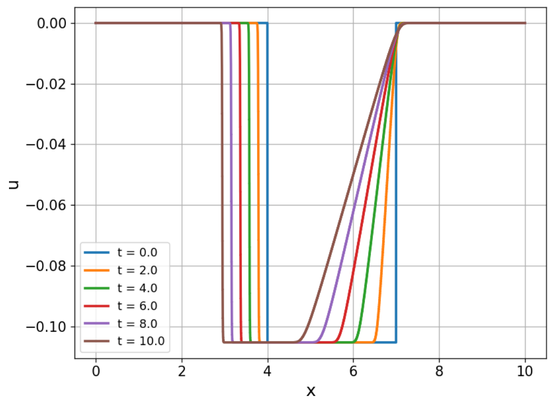


Figure 15: Variable u for $t = 0$ to $t = 10$ with $\Delta x = \Delta t = 0.001, \rho_{max} = 38, \rho_1 = 19, \rho_2 = 21$.

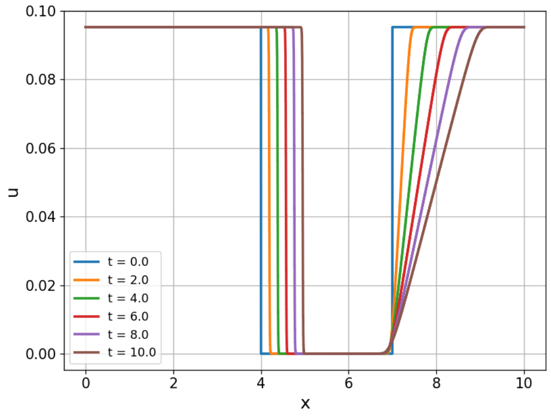


Figure 16: Variable u for $t = 0$ to $t = 10$ with $\Delta x = \Delta t = 0.001, \rho_{max} = 42, \rho_1 = 19, \rho_2 = 21$.

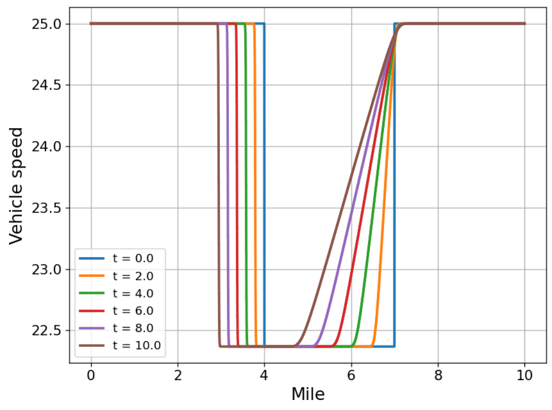


Figure 17: Vehicle speeds for $t = 0$ to $t = 10$ with $\Delta x = \Delta t = 0.001, \rho_{max} = 38, \rho_1 = 19, \rho_2 = 21$.

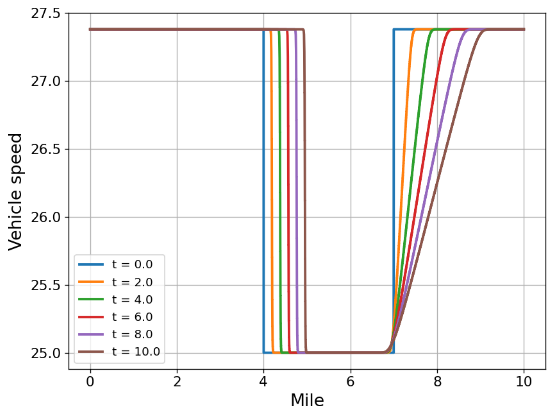


Figure 18: Vehicle speeds for $t = 0$ to $t = 10$ with $\Delta x = \Delta t = 0.001, \rho_{max} = 42, \rho_1 = 19, \rho_2 = 21$.