Gabriel Rios
AOS 575
October 13, 2022

Problem set 3

1. **Constant wind speed advection equation with 4th-order centered difference and 3rd-order Adams-Bashforth**

To compute solutions for constant wind speed advection, whose equation is

$$\frac{\partial \psi}{\partial t} = -c\frac{\partial \psi}{\partial x}, \tag{1}$$

the 3rd-order Adams-Bashforth scheme was used for time-stepping, while a 4th-order centered difference scheme was used to discretize in space. The schemes are shown as follows:

$$\psi_{n+1} = \psi_n + \frac{\Delta t}{12}\left[23F(\psi_n) - 16F(\psi_{n-1}) + 5F(\psi_{n-2})\right] \tag{2}$$

$$\frac{\partial \psi}{\partial x} = F(\psi_n) = \frac{4}{6\Delta x}(\psi_{j+1} - \psi_{j-1}) - \frac{1}{12\Delta x}(\psi_{j+2} - \psi_{j-2} \tag{3}$$

where (2) is the Adams-Bashforth scheme (AB3) in time and (3) is the centered difference scheme (CD) for space. Theses schemes were implemented with a *top-down* approach, which is my way of saying that stepping through time was defined first, with subroutines (such as the computation of $\partial\psi/\partial x$) being defined afterwards. So, AB3 was implemented first as it sets a structure within which CD is used and is called first in the sequence of program flow. Then, CD was implemented, as it is called as a subroutine of AB3. However, because AB3 requires two previous time steps to compute the next time step, a starter scheme was required. For this, an explicit 3rd-order Runge-Kutta (RK3) scheme was implemented, as shown below:

$$\psi_1 = \psi_n + \frac{q_1}{3} \Rightarrow q_1 = \Delta x F(\psi_n) \tag{4}$$

$$\psi_2 = \psi_1 + \frac{15q_2}{16} \Rightarrow q_2 = \Delta x F(\psi_n) - \frac{5q_1}{9} \tag{5}$$

$$\psi_{n+1} = \psi_2 + \frac{8q_1}{15} \tag{6}$$

Note: the CD scheme shown in (3) was used for the RK3 starter, as well. Implementation of these schemes to solve (1) is provided under the heading *Problem 1* in the appendix. A copy of the Jupyter Notebook containing this implementation has also been e-mailed to Bob Hallberg and Sam Ditkovsky. The schemes were executed over a periodic domain ($0 \leq x \leq 1$) with an initial condition of $\psi(x, 0) = \sin^6(2\pi x)$.

(a) **Fixed Courant number of 0.1, variable grid spacing**

The solutions for each set of parameters (grid spacing Δx of 1/20, 1/40, 1/80, 1/160) at time $t = 50$ with wave speed $c = 0.1$ are provided in Figure 1. For reference, the exact solution is shown in red. The approximations at a Courant number of 0.1 resulted in a modest approximation at a larger grid spacing ($\Delta x = 1/20$) to excellent approximations at smaller grid spacings ($\Delta x = 1/80, 1/160$). There is a visible amplitude and phase error for larger grid spacing values, which shows a weakness in the joint space/time scheme to properly advect the wave when spatial resolution is not sufficiently fine. The solution with $\Delta x = 1/20$ demonstrates some diffusion in addition to the errors mentioned, as shown by the broader area under the curve and a slight breakdown of the waveform. To a lesser degree, the solution with $\Delta x = 1/40$ shown a small spurious secondary wave, demonstrating a drawback to using smaller grid spacing with this scheme to resolve advection. To quantify the degree of error associated with each solution, the root-mean-squared error (RMSE) was derived, and is shown in Table 1 and graphically in Figure 1. This confirms the high accuracy ($5.43\text{e}-4$ at $\Delta x = 1/160$!) of this scheme at small grid spacings.

Table 1: Root-mean-squared error for solutions to Equation 1 at $t = 50$ at multiple grid spacings, relative to the exact solution.

Δx	RMSE
$1/20$	$2.61\text{e}-1$
$1/40$	$8.81\text{e}-2$
$1/80$	$8.52\text{e}-3$
$1/160$	$5.43\text{e}-4$

(b) **Fixed grid spacing of $\Delta x = 1/20$, variable Courant number**

The solutions for each set of parameters ($c\Delta t/\Delta x$ of 0.1, 0.2, 0.4, and 0.8) at time $t = 50$ are provided in Figure 2. For reference, the exact solution is shown in red. The solutions at a grid spacing of $\Delta x = 1/20$ with increasing Courant numbers resulted in increasingly poor approximations of the exact solution. At lower Courant numbers, the amplitude of the wave was better preserved (although still damped) with a larger phase error, while the amplitude error increased (and phase error decreased) with increasing Courant number. However, the solution became unstable at $c\Delta t/\Delta x = 0.8$, as shown in Figure 2. This occurred due to the increase in time step, since the Courant number is the constraint that determines Δt, given that c, Δx, and the Courant number are prescribed. At a larger Courant number, the time step must increase, which amplifies instability in the solution. For reference, the instability occurs very soon after the full AB3/CD scheme begins (post-RK3), with unstable growth occurring at a couple of timesteps into the full scheme. To quantify the degree of error associated with each solution, the RMSE is shown in Table 2 and graphically in Figure 2.

Table 2: Root-mean-squared error for solutions to Equation 1 at $t = 50$ at multiple Courant numbers, relative to the exact solution.

$c\Delta t/\Delta x$	RMSE
0.1	$2.61\text{e}-1$
0.2	$2.19\text{e}-1$
0.4	$0.17\text{e}-1$
0.8	1.69e32

(c) **Fixed grid spacing of $\Delta x = 1/80$, variable Courant number**

The solutions at a grid spacing of $\Delta x = 1/80$ with increasing Courant numbers resulted in excellent approximations of the exact solution, with the exception of the run with $c\Delta t/\Delta x = 0.8$, as shown in Figure 3. The approximations by the runs at lower Courant numbers improved due to the decreased grid spacing, allowing for improved spatial (and by extension, temporal) resolution. Similar to the example with larger grid spacing $\Delta x = 0.2$, this occurred due to the increase in time step. To quantify the degree of error associated with each solution, the RMSE is shown in Table 3 and graphically in Figure 3.

Table 3: Root-mean-squared error for solutions to Equation 1 at $t = 50$ at multiple Courant numbers, relative to the exact solution.

$c\Delta t/\Delta x$	RMSE
0.1	$8.52\text{e}-3$
0.2	$8.39\text{e}-3$
0.4	$1.36\text{e}-2$
0.8	1.78e121

2. **Constant wind speed advection equation with 4th-order compact difference and leapfrog**

with Asselin filtering

For this exercise, different schemes were used to solve (1). In space, Lele's 4th-order compact difference scheme is used, whereas a leapfrog scheme with an Asselin filter ($\gamma = 0.1$) is used to step in time. The schemes are shown as follows:

$$\frac{1}{24}\left[5\frac{\partial\psi_{j+1}}{\partial x} + 14\frac{\partial\psi_j}{\partial x} + 5\frac{\partial\psi_{j-1}}{\partial x}\right] = \frac{1}{12}\left[\frac{11}{2\Delta x}\left(\psi_{j+1} - \psi_{j-1}\right) + \frac{1}{4\Delta x}\left(\psi_{j+2} - \psi_{j-2}\right)\right] \quad (7)$$

$$\psi_{n+1} = \overline{\psi_{n-1}} + 2\Delta t F(\psi_n) \qquad \text{,where} \quad (8)$$

$$\overline{\psi_n} = \psi_n + \gamma\left(\psi_{n+1} - 2\psi_n + \psi_{n-1}\right), \quad (9)$$

where (7) is the 4th-order Lele scheme and (8) is the leapfrog scheme with an Asselin filter (LF). For the Lele scheme, a tridiagonal solver was required to solve for $\partial\psi_j/\partial x$. The solver used is included in the attached/sent code, which is the solver provided by Durran and translated to Python by Sam (thanks, Sam!). In addition to the leapfrog scheme presented, an additional filtering step was taken at each timestep n, where a filtering operation [shown in (9)], was applied. The program flow for these schemes followed the same approach as the approach taken in Exercise 1, with an RK3 starter.

(a) **Fixed Courant number of 0.1, variable grid spacing**

The solutions for each set of parameters (grid spacing Δx of 1/20, 1/40, 1/80, 1/160) at time $t = 50$ with wave speed $c = 0.1$ are provided in Figure 4. For reference, the exact solution is shown in red. The approximations at a Courant number of 0.1 resulted in a modest approximation at a larger grid spacing ($\Delta x = 1/20$) to good approximations at smaller grid spacings ($\Delta x = 1/80, 1/160$). There is a visible amplitude error for all grid spacing values, with small phase error for larger grid spacing values. Compared to the AB3/CD scheme presented in Exercise 1, the LF/Lele scheme minimized phase error at the cost of dissipation and moderate diffusion for large Δx. This scheme also demonstrates less spurious behavior (more intact wave profile) than the AB3/CD scheme. To quantify the degree of error associated with each solution, the root-mean-squared error (RMSE) was derived, and is shown in Table 4 and graphically in Figure 4.

Table 4: Root-mean-squared error for solutions to Equation 1 at $t = 50$ at multiple grid spacings, relative to the exact solution.

Δx	RMSE
1/20	1.63e − 1
1/40	9.93e − 2
1/80	5.69e − 2
1/160	3.09e − 2

(b) **Fixed grid spacing of $\Delta x = 1/20$, variable Courant number**

The solutions for each set of parameters ($c\Delta t/\Delta x$ of 0.1, 0.2, 0.4, and 0.8) at time $t = 50$ are provided in Figure 5. For reference, the exact solution is shown in red. The solutions at a grid spacing of $\Delta x = 1/20$ with increasing Courant numbers resulted in increasingly poor approximations of the exact solution. At lower Courant numbers, the amplitude of the wave was better preserved (although still damped) with a larger phase error, while the amplitude error increased (and phase error decreased) with increasing Courant number. This is suggestive of strong diffusion and dissipation, given that the peaks and troughs are lower and higher, respectively. This is a similar phenomenon to the behavior for Exercise 1b. Again, the solution became unstable at $c\Delta t/\Delta x = 0.8$, as shown in Figure 5 for the same reasons as previously stated. To quantify the degree of error associated with each solution, the RMSE is shown in Table 5 and graphically in Figure 5.

Table 5: Root-mean-squared error for solutions to Equation 1 at $t = 50$ at multiple Courant numbers, relative to the exact solution.

Δx	RMSE
1/20	1.63e − 1
1/40	2.38e − 1
1/80	3.35e − 1
1/160	3.83e52

(c) **Fixed grid spacing of $\Delta x = 1/80$, variable Courant number**

The solutions at a grid spacing of $\Delta x = 1/80$ with increasing Courant numbers resulted in better approximations of the exact solution compared to the solutions with $\Delta x = 1/80$ and variable Courant number, with the exception of the run with $c\Delta t/\Delta x = 0.8$, as shown in Figure 6. The approximations by the runs at lower Courant numbers improved due to the decreased grid spacing, allowing for improved spatial (and by extension, temporal) resolution. However, amplitude errors increase, with solutions at larger Courant numbers exhibiting smaller peaks. Similar to the example with larger grid spacing $\Delta x = 0.2$, this occurred due to the increase in time step. To quantify the degree of error associated with each solution, the RMSE is shown in Table 6 and graphically in Figure 6.

Table 6: Root-mean-squared error for solutions to Equation 1 at $t = 50$ at multiple Courant numbers, relative to the exact solution.

$c\Delta t/\Delta x$	RMSE
0.1	5.69e − 2
0.2	9.90e − 2
0.4	7.16e − 1
0.8	4.25e88

3. **Derivation of a 3rd-order method in space and time**

(a) To derive a method to Equation (1), a scheme with the form

$$\frac{1}{\Delta t}(\psi_j^{n+1} - \psi_j^n) = \frac{-c}{\Delta x}(\alpha\psi_{j-2}^n + \beta\psi_{j-1}^n + \gamma\psi_j^n + \delta\psi_{j+1}^n), \tag{10}$$

I needed to expand a lot of terms and didn't see the point in writing it out *and* typesetting it on LaTeX. So, I'll compromise by explaining by derivation strategy with the final result, and I'll append the handwritten derivation to the appendix (as I typed this out, I realized where the word 'appendix' comes from).

To derive the method, I began by equating (10) to Equation (1). Then, I started by deriving a 3rd-order accurate method in space by finding an expression for $\frac{\partial\psi}{\partial x}$ as a function of the points $\psi_{j-2}...\psi_{j+1}$. This was done using a Taylor table (expansion of each ψ term and construct a system of equations) and using a solver (code provided in the appendix) to calculate coefficient values for each ψ term. Then, $\frac{\partial\psi}{\partial t}$ was expanded to 3rd-order accuracy and commutation of partials was used to allow for the expression of $\frac{\partial\psi}{\partial t}$ in terms of $\frac{\partial\psi}{\partial x}$ with the help of (1). Once all terms were expressed as functions of $\frac{\partial\psi}{\partial x}$, like ψ terms were grouped and coefficients as functions of Courant number (μ) were obtained. These are listed below:

$$\alpha = \frac{1}{6}(1+\mu^2) \tag{11}$$

$$\beta = \frac{1}{2}(-2+\mu-\mu^2) \tag{12}$$

$$\gamma = \frac{1}{2}(1-2\mu+\mu^2) \tag{13}$$

$$\delta = \frac{1}{6}(2+3\mu-\mu^2) \tag{14}$$

(b) To derive expressions for the discrete flux terms listed in:

$$\frac{1}{\Delta t}(\psi_j^{n+1}-\psi_j^n) = \frac{-1}{\Delta x}(F_{j+1/2}-F_{j-1/2}), \tag{15}$$

the expression provided for a general flux term F was formalized:

$$F_{j+1/2} = c\left[\psi_j + d_1(\psi_j-\psi_{j-1}) + d_0(\psi_{j+1}-\psi_j\right] \tag{16}$$

$$F_{j-1/2} = c\left[\psi_{j-1} + d_3(\psi_{j-1}-\psi_{j-2}) + d_0(\psi_j-\psi_{j-1}\right], \tag{17}$$

where d_m are functions of the Courant number, which is being expressed as $\frac{c\Delta t}{\Delta x} = \mu$. Similar to the process for (a), I will discuss the derivation strategy and provide the expressions for the flux terms. For the actual derivations, I have appended my handwritten derivation to the appendix. I plugged the expressions flux terms into (15) and set the right-hand side of this equation to equal (10). From here, I used the known coefficient values (α, etc.) to solve for the d terms. This led to the expressions for the flux terms:

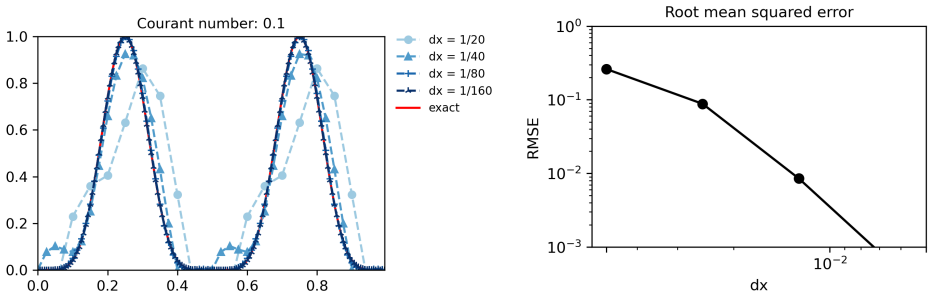
$$F_{j+1/2} = c\left[\psi_j + \frac{1}{12}\left(-1-6\mu+4\mu^2\right)(\psi_j-\psi_{j-1}) + \frac{1}{12}\left(4+6\mu-2\mu^2\right)(\psi_{j+1}-\psi_j)\right] \tag{18}$$

$$F_{j-1/2} = c\left[\psi_{j-1} + \frac{1}{12}(2+2\mu)(\psi_{j-1}-\psi_{j-2}) + \frac{-1}{12}(\psi_j-\psi_{j-1})\right] \tag{19}$$

(c) As the Courant number ($\mu \to 0$), (15) corresponds to the 3rd-order accurate upwind scheme. Using (18) and (19) in (15), the scheme takes the form (note: handwritten derivation provided in appendix):

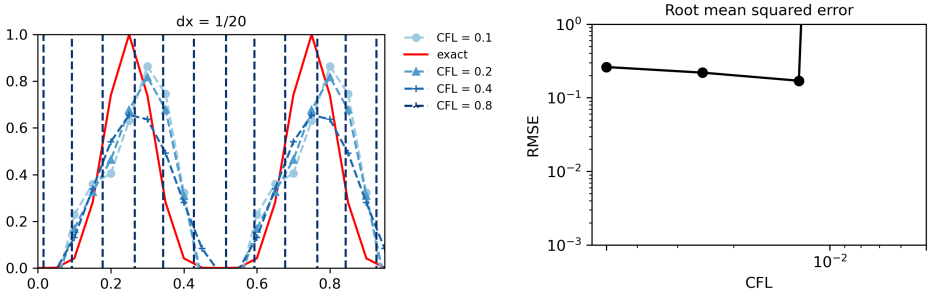
$$\frac{1}{\Delta t}(\psi_j^{n+1}-\psi_j^n) = \frac{-c}{6\Delta x}(\psi_{j-2}-6\psi_{j-1}+3\psi_j+2\psi_{j+1}) \tag{20}$$

Appendix



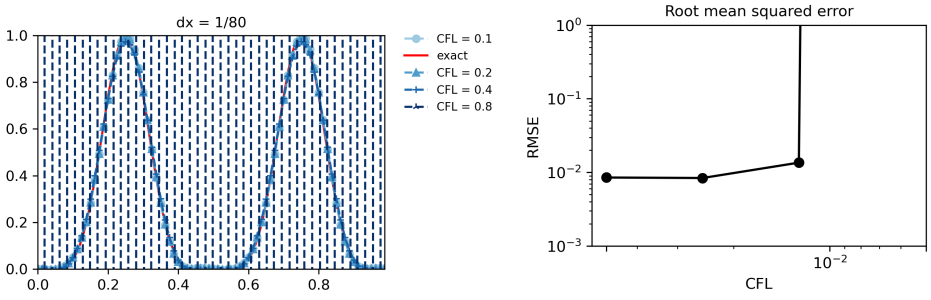
(a) (b)

Figure 1: *(left)* Numerical solutions for Equation 1 at $t = 50$ at multiple grid spacings. The exact solution is shown in red, for comparison. *(right)* Root-mean-squared error for solutions to Equation 1 at $t = 50$ at multiple grid spacings, relative to the exact solution.



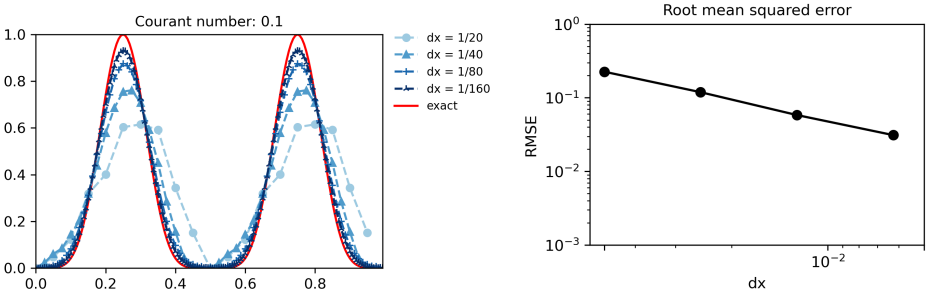
(a) (b)

Figure 2: *(left)* Numerical solutions for Equation 1 at $t = 50$ at multiple Courant numbers. The exact solution is shown in red, for comparison. *(right)* Root-mean-squared error for solutions to Equation 1 at $t = 50$ at multiple Courant numbers, relative to the exact solution.



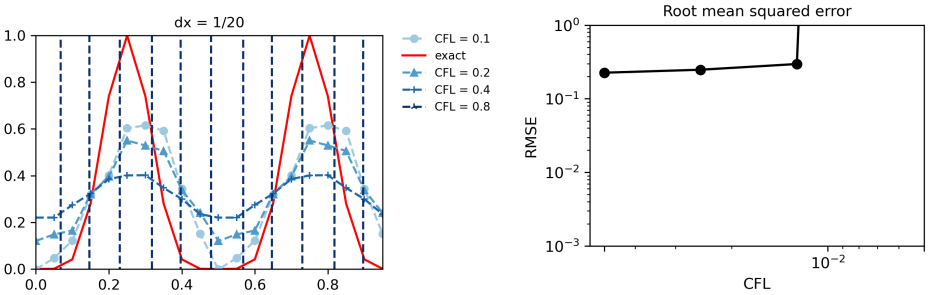
(a) (b)

Figure 3: *(left)* Numerical solutions for Equation 1 at $t = 50$ at multiple Courant numbers. The exact solution is shown in red, for comparison. *(right)* Root-mean-squared error for solutions to Equation 1 at $t = 50$ at multiple Courant numbers, relative to the exact solution.



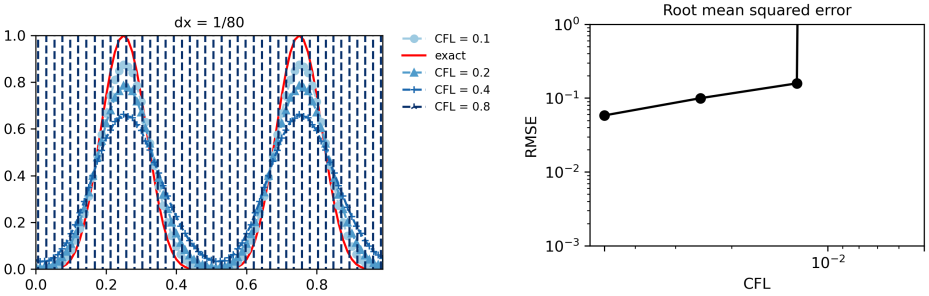
(a) (b)

Figure 4: *(left)* Numerical solutions for Equation 1 at $t = 50$ at multiple grid spacings. The exact solution is shown in red, for comparison. *(right)* Root-mean-squared error for solutions to Equation 1 at $t = 50$ at multiple grid spacings, relative to the exact solution.



(a) (b)

Figure 5: *(left)* Numerical solutions for Equation 1 at $t = 50$ at multiple Courant numbers. The exact solution is shown in red, for comparison. *(right)* Root-mean-squared error for solutions to Equation 1 at $t = 50$ at multiple Courant numbers, relative to the exact solution.



(a) (b)

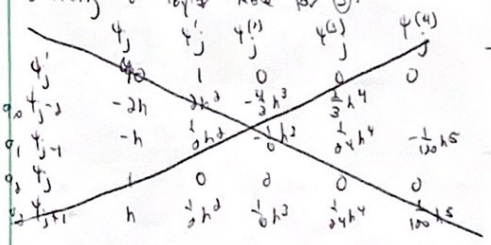
Figure 6: *(left)* Numerical solutions for Equation 1 at $t = 50$ at multiple Courant numbers. The exact solution is shown in red, for comparison. *(right)* Root-mean-squared error for solutions to Equation 1 at $t = 50$ at multiple Courant numbers, relative to the exact solution.

3a

(x) $\frac{\partial \psi}{\partial t} + c\frac{\partial \psi}{\partial x} = 0 \rightarrow \frac{\partial \psi}{\partial t} = -c\frac{\partial \psi}{\partial x}$ (1). Rewriting: $\frac{1}{\Delta t}(\psi_j^{n+1} - \psi_j) + \frac{c}{\Delta x}(\alpha\psi_{j-2} + \beta\psi_{j-1} + \gamma\psi_j + \delta\psi_{j+1}) = \frac{\partial \psi}{\partial t} + c\frac{\partial \psi}{\partial x} + O(\Delta x^3, \Delta t^2)$ (2)

→ Build a 3rd order-accurate scheme for $\frac{\partial \psi}{\partial x}$. Equation: $\psi_j' + a_0\psi_{j-2} + a_1\psi_{j-1} + a_2\psi_j + a_3\psi_{j+1} = 0$. (3)

- Building a Taylor table for (3):



	ψ_j	ψ_j'	ψ_j''	ψ_j'''	$\psi_j^{(4)}$
ψ_j'	0	1	0	0	0
$a_0\psi_{j-2}$	1	$-2h$	$2h^2$	$-\frac{4}{3}h^3$	$\frac{2}{3}h^4$
$a_1\psi_{j-1}$	1	$-h$	$\frac{1}{2}h^2$	$-\frac{1}{6}h^3$	$\frac{1}{24}h^4$
$a_2\psi_j$	1	0	0	0	0
$a_3\psi_{j+1}$	1	h	$\frac{1}{2}h^2$	$\frac{1}{6}h^3$	$\frac{1}{24}h^4$

→ Using a solver:
$a_0 = -\frac{1}{6h}$
$a_1 = \frac{1}{h}$
$a_2 = -\frac{1}{2h}$
$a_3 = -\frac{1}{3h}$

- Rewriting (3) using a_m values: $\psi_j' - \frac{1}{6h}\psi_{j-2} + \frac{1}{h}\psi_{j-1} - \frac{1}{2h}\psi_j - \frac{1}{3h}\psi_{j+1} = O(\Delta x^3)$ (4)
- (4) → (3): $\frac{1}{\Delta t}\left[\psi_j^{n+1} - \psi_j\right] + \frac{c}{\Delta x}\left[\alpha\psi_{j-2} + \beta\psi_{j-1} + \gamma\psi_j + \delta\psi_{j+1}\right] = \frac{\partial \psi}{\partial t} + c\frac{\partial \psi}{\partial x} + O(\Delta x^3, \Delta t^2)$. (4) represents this. (5) Expand $\frac{\partial \psi}{\partial t}$.
- From (5): RHS: $= \frac{\partial \psi}{\partial t} + c\frac{\partial \psi}{\partial x} + O(\Delta x^3, \Delta t^2)$. Try again:

$$= c\frac{\partial \psi}{\partial x} + \psi_j' + \underbrace{\frac{1}{2}\Delta t\psi_j''}_{(6a)} + \underbrace{\frac{1}{6}\Delta t^2\psi_j^{(3)}}_{(6b)} + \frac{1}{24}\Delta t^3\psi_j^{(4)} + O(\Delta t^4) + O(\Delta t^3) \quad (6)$$

- (6a) and (6b) must be discretized: writing a Taylor table equation for each:
- (6a): $\psi_j'' + b_0\psi_{j-2} + b_1\psi_{j-1} + b_2\psi_j + b_3\psi_{j+1} = O(\Delta t^3)$ (6b): $\psi_j^{(3)} + c_0\psi_{j-2} + c_1\psi_{j-1} + c_2\psi_j + c_3\psi_{j+1} = O(\Delta t^3)$

However, conversion from time to space must be made using (1).

- (6a): $\frac{\partial^2 \psi}{\partial t^2} = \frac{\partial}{\partial t}\left(\frac{\partial \psi}{\partial t}\right) = \frac{\partial}{\partial t}\left(-c\frac{\partial \psi}{\partial x}\right) = -c\frac{\partial}{\partial x}\left(\frac{\partial \psi}{\partial t}\right) = +c^2\frac{\partial^2 \psi}{\partial x^2}$ (6a1)
- (6b): $\frac{\partial^3 \psi}{\partial t^3} = \frac{\partial}{\partial t}\left(\frac{\partial}{\partial t}\left(\frac{\partial \psi}{\partial t}\right)\right) = \frac{\partial}{\partial t}\left(\frac{\partial}{\partial t}\left(-c\frac{\partial \psi}{\partial x}\right)\right) = \frac{\partial}{\partial t}\left(-c\frac{\partial}{\partial x}\left(\frac{\partial \psi}{\partial t}\right)\right) = \frac{\partial}{\partial t}\left(c^2\frac{\partial^2 \psi}{\partial x^2}\right) = c^2\frac{\partial}{\partial t}\left(\frac{\partial}{\partial x}\left(\frac{\partial \psi}{\partial x}\right)\right) = c^3\left(\frac{\partial}{\partial x}\left(\frac{\partial}{\partial x}\left(\frac{\partial \psi}{\partial t}\right)\right)\right) = -c^3\frac{\partial^3 \psi}{\partial x^3}$ (6b1)
- Rewrite (6) with (6a1) and (6b1): RHS: $= c\frac{\partial \psi}{\partial x} + \frac{\partial \psi}{\partial t} + \frac{1}{2}c^2\Delta t^2\frac{\partial^2 \psi_j}{\partial x^2} - \frac{1}{6}c^3\Delta t^3\frac{\partial^3 \psi_j}{\partial x^3} + O(\Delta t^3, \Delta x^3)$ (7)
- Revisiting Taylor tables: (6a): $b_0 = 0$, $b_1 = -\frac{1}{\Delta x^2}$, $b_2 = \frac{2}{\Delta x^2}$, $b_3 = -\frac{1}{\Delta x^2}$ | (6b): $c_0 = \frac{1}{\Delta x^3}$, $c_1 = -\frac{3}{\Delta x^3}$, $c_2 = \frac{3}{\Delta x^3}$, $c_3 = -\frac{1}{\Delta x^3}$
- Applying these to (6a) and (6b) results in: $\psi_j'' = \frac{1}{\Delta x^2}\psi_{j-1} - \frac{2}{\Delta x^2}\psi_j + \frac{1}{\Delta x^2}\psi_{j+1} = \frac{\partial^2 \psi_j}{\partial x^2}$ (8)
$\psi_j''' = -\frac{1}{\Delta x^3}\psi_{j-2} + \frac{3}{\Delta x^3}\psi_{j-1} - \frac{3}{\Delta x^3}\psi_j + \frac{1}{\Delta x^3}\psi_{j+1} = \partial^3\psi_j/\partial x^3$ (9)
- From (7), ~~[illegible]~~ rewrite (7) using (4), (8), (9):

$$= \left[a\frac{c}{\Delta x}\psi_{j-2} - \frac{c}{\Delta x}\psi_{j-1} + \frac{c}{\Delta x}\psi_j + \frac{c}{\Delta x}\psi_{j+1}\right] + \left[\frac{c^2\Delta t^2}{2\Delta x^2}\psi_{j-1} - \frac{2c^2\Delta t^2}{2\Delta x^2}\psi_j + \frac{c^2\Delta t^2}{2\Delta x^2}\psi_{j+1}\right] + \left[\frac{c^3\Delta t^3}{6\Delta x^3}\psi_{j-2} - \frac{3c^3\Delta t^3}{6\Delta x^3}\psi_{j-1} + \frac{3c^3\Delta t^3}{6\Delta x^3}\psi_j - \frac{c^3\Delta t^3}{6\Delta x^3}\psi_{j+1}\right] + \frac{\partial \psi}{\partial t} + O(\Delta x^3, \Delta t^3) \quad (10)$$

- Group ψ terms: $\left[\frac{c}{\Delta x} + \frac{c^3\Delta t^2}{6\Delta x^3}\right]\psi_{j-2} + \left[-\frac{c}{\Delta x} + \frac{c^2\Delta t^2}{2\Delta x^2} - \frac{c^3\Delta t^2}{2\Delta x^3}\right]\psi_{j-1} + \left[\frac{c}{\Delta x} - \frac{c^2\Delta t^2}{2\Delta x^2} + \frac{c^3\Delta t^2}{2\Delta x^3}\right]\psi_j + \left[\frac{c}{\Delta x} + \frac{c^2\Delta t^2}{2\Delta x^2} - \frac{c^3\Delta t^2}{6\Delta x^3}\right]\psi_{j+1} + \frac{\partial \psi}{\partial t} + O(\Delta x^3, \Delta t^3)$ (11)
- Factor out $\frac{c}{\Delta x}$ & rewrite $\frac{c\Delta t}{\Delta x} = \mu$: $\frac{c}{\Delta x}\left\{\left[\frac{1}{6}(1+\mu^3)\right]\psi_{j-2} + \left[-1 - \frac{1}{2}\mu^2 + \frac{1}{2}\mu^3\right]\psi_{j-1} + \left[1 + \frac{1}{2}\mu^2 - \mu^3\right]\psi_j + \left[\frac{1}{6} - \frac{1}{2}\mu + \frac{1}{2}\mu^2 - \frac{1}{6}\mu^3\right]\psi_{j+1}\right\} + \frac{\partial \psi}{\partial t} + O(\Delta x^3, \Delta t^3)$ (12)
- By this,

$$\boxed{\begin{aligned} \alpha &= \tfrac{1}{6}(1+\mu^3) \\ \beta &= \tfrac{1}{2}(-2+\mu-\mu^2) \\ \gamma &= \tfrac{1}{2}(1-2\mu+\mu^2) \\ \delta &= \tfrac{1}{6}(2+3\mu-\mu^3) \end{aligned}} \quad \text{ANS} \quad \bigstar$$

[3b] • $\frac{1}{\delta t}(\psi_j^{n+1} - \psi_j) = -\frac{1}{\delta x}\underbrace{\left[\tilde{F}_{j+1/2} - \tilde{F}_{j-1/2}\right]}_{(10)}$ (1), where $\tilde{F}_{j+1/2} = c\left[\psi_j + d_1(\psi_j - \psi_{j-1}) + d_0(\psi_{j+1} - \psi_j)\right]$ (2a)

$\tilde{F}_{j-1/2} = c\left[\psi_{j-1} + d_3(\psi_{j+1} - \psi_j) + d_2(\psi_j - \psi_{j-1})\right]$ (2b) → assume d_0, d_1 for $\tilde{F}_{j+1/2}$ ≠ d_0, d_1 for $\tilde{F}_{j-1/2}$

• Note: (10) from [3a] must match $-\frac{c}{\delta x}\left[\tilde{F}_{j+1/2} - \tilde{F}_{j-1/2}\right]$. Write (10) using (2a) and (2b).

• (10) = $-\frac{c}{\delta x}\left[\psi_j + d_1\psi_j - d_1\psi_{j-1} + d_0\psi_{j+1} - d_0\psi_j - \psi_{j-1} - d_2\psi_{j+1} + d_2\psi_j - d_3\psi_{j+2} + d_3\psi_{j+1}\right]$ Regrouping by like ψ terms: (3)

$= -\frac{c}{\delta x}\left[\psi_j + d_1\psi_j - d_1\psi_{j-1} + d_0\psi_{j+1} - d_0\psi_j - \psi_{j-1} - d_2\psi_{j+1} + d_2\psi_{j-1} - d_3\psi_{j+2} + d_3\psi_{j+1}\right]$ Regrouping by like ψ terms:

$= -\frac{c}{\delta x}\left[d_3\psi_{j-2} + (-d_1 - 1 - d_2 - d_3)\psi_{j-1} + (1 + d_1 - d_0 - d_3)\psi_j + d_0\psi_{j+1}\right]$ (4)

• Matching (4) to (11) from [3a]: $d_3 = \alpha = \frac{1}{6}(1+\mu^2)$ (8a) Try (8b) + (8c) to solve for d_2.

$-d_1 - 1 - d_2 - d_3 = \beta = \frac{1}{2}(-2+\mu-\mu^2)$ (8b) ↓

$-d_0 + d_1 + 1 - d_2 = \gamma = \frac{1}{2}(1-2\mu+\mu^2)$ (8c) $-d_0 - d_2 - 2d_3 = \beta + \gamma$ (6)

$d_0 = \delta = \frac{1}{6}(2+3\mu-\mu^2)$ (8d)

• Expand (6): $-d_0 - d_2 - 2d_3 = -\frac{1}{6}(2+3\mu-\mu^2) - \frac{1}{2}(1-2\mu+\mu^2)$

$= -\frac{1}{2}\left[\frac{1}{3} + \frac{1}{2}\mu - \frac{1}{6}\mu^2 + \frac{1}{6} + \frac{1}{6}\mu^2 - 1 + \frac{1}{2}\mu - \frac{1}{2}\mu^2\right]$ (7)

• Simplify (7): $d_3 = -\frac{1}{2}\left[\frac{1}{6}\right] = -\frac{1}{12}$. (8) Use (8) in (8b) to get d_1. → $d_1 = -1 - d_2 - d_3 - \frac{1}{2}(-2+\mu-\mu^2)$ (9)

• Expand (9): $d_1 = -1 - (-\frac{1}{12}) - \frac{1}{6}(1+\mu^2) - \frac{1}{2}(-2+\mu-\mu^2) = -1 + \frac{1}{12} - \frac{1}{6} - \frac{1}{6}\mu^2 + 1 - \frac{1}{2}\mu + \frac{1}{2}\mu^2$

$= -\frac{1}{12} - \frac{1}{2}\mu + \frac{1}{3}\mu^2 = \frac{1}{12}\left[-1 - 6\mu + 4\mu^2\right]$ (10)

• Therefore,

[ANS]

$d_0 = \frac{1}{12}\left[4 + 6\mu - 2\mu^2\right]$ (11a)

$d_1 = \frac{1}{12}\left[-1 - 6\mu + 4\mu^2\right]$ (11b)

$d_2 = -\frac{1}{12}$ (11c)

$d_3 = \frac{1}{12}\left[2 + 2\mu^2\right]$ (11d)

★

• Therefore, $\tilde{F}_{j+1/2} = c\left[\psi_j + \frac{1}{12}(-1-6\mu+4\mu^2)(\psi_j - \psi_{j-1}) + \frac{1}{12}(4+6\mu-2\mu^2)(\psi_{j+1} - \psi_j)\right]$ (12)

$\tilde{F}_{j-1/2} = c\left[\psi_{j-1} + \frac{1}{12}(2+2\mu^2)(\psi_{j-1} - \psi_{j-2}) + \frac{1}{12}(\psi_j - \psi_{j-1})\right]$ (13)

[3c] • If $\mu \to 0$: $\tilde{F}_{j+1/2} = c\left[\psi_j - \frac{1}{12}(\psi_j - \psi_{j-1}) + \frac{4}{12}(\psi_{j+1} - \psi_j)\right] = c\left[\frac{7}{12}\psi_j + \frac{1}{12}\psi_{j-1} + \frac{4}{12}\psi_{j+1}\right]$ (14)

(from (11)) $\tilde{F}_{j-1/2} = c\left[\psi_{j-1} + \frac{2}{12}(\psi_{j-1} - \psi_{j-2}) + \frac{1}{12}(\psi_j - \psi_{j-1})\right] = c\left[\frac{10}{12}\psi_{j-1} - \frac{1}{6}\psi_{j-2} + \frac{1}{12}\psi_j\right]$ (15)

• Combining (14) and (15) → (1): $\frac{1}{\delta t}\left[\psi_j^{n+1} - \psi_j\right] = -\frac{c}{\delta x}\left[\left(\frac{7}{12}\psi_j + \frac{1}{12}\psi_{j-1} + \frac{4}{12}\psi_{j+1}\right) - \left(\frac{10}{12}\psi_{j-1} - \frac{1}{6}\psi_{j-2} + \frac{1}{12}\psi_j\right)\right]$

$= -\frac{c}{\delta x}\left[\frac{1}{2}\psi_j - \frac{5}{6}\psi_{j-1} + \frac{1}{3}\psi_{j+1} + \frac{1}{6}\psi_{j-2}\right)$.

[ANS] → 3rd order upwind: $= \boxed{-\frac{c}{\delta x}\left[\frac{1}{6}\psi_{j-2} - \frac{5}{6}\psi_{j-1} + \frac{3}{6}\psi_j + \frac{2}{6}\psi_{j+1}\right]}$

notebook

October 13, 2022

[948]:
```
import numpy as np
import matplotlib.pyplot as plt
import cycler
```

Define initial condition $\psi(x, 0)$.

[949]:
```
def f(x):
    return np.sin(2*np.pi*x)**6
```

Define $\partial\psi/\partial x$.

[950]:
```
def f_x(c, x):
    return -12*c*np.pi*((np.sin(2*np.pi*x))**5)*np.cos(2*np.pi*x)
```

0.0.1 Problem 1

Compute solutions to the wind speed advection equation using Adams-Bashforth (3rd-order) in time and 4th-order central differencing in space.

Function F for AB3.

[951]:
```
def f_x_(c, dx, dt, f_):
    return c*cdf(dt, dx, f_)
```

Define discretization schemes: - Starter: Runge-Kutta, 3rd order - Temporal: Adams-Bashforth, 3rd-order - Spatial: 4th-order compact difference

Runge-Kutta, 3rd order

[952]:
```
def rk3(h, c, dt, dx, x, psi):
    # q_1 = h*f_x(c, x)
    q_1 = h*f_x_(c, dx, dt, psi)
    psi_1 = psi + (1/3)*q_1
    # q_2 = h*f_x(c, psi_1) - (5/9)*q_1
    q_2 = h*f_x_(c, dx, dt, psi) - (5/9)*q_1
    psi_2 = psi_1 + (15/16)*q_2
    # q_3 = h*f_x(c, psi_2) - (153/128)*q_2
    q_3 = h*f_x_(c, dx, dt, psi) - (153/128)*q_2
    psi_n_1 = psi_2 + (8/15)*q_3
    return psi_n_1
```

4th-order centered difference

[953]:
```
def cdf(dt, dx, f_):
    f_prime = np.zeros((len(f_),))
    for i in range(0, len(f_)):
        # Using modulo for indices to handle boundary conditions, since BCs are␣
↪periodic
        # f_prime[i] = (f_[(i-2) % len(f_)] - 8*f_[(i-1) % len(f_)] +␣
↪8*f_[(i+1) % len(f_)] - f_[(i+2) % len(f_)])/(12*dx)
        f_prime[i] = (4/(6*dx))*(f_[(i+1) % len(f_)] - f_[(i-1) % len(f_)]) -␣
↪(1/(12*dx))*(f_[(i+2) % len(f_)] - f_[(i-2) % len(f_)])
    return f_prime
```

Adams-Bashforth, 3rd order

[954]:
```
def ab3(dx, dt, c, x, psis):
    # Get previous psi values [psi(n), psi(n-1), psi(n-2)]
    psi_n, psi_n_1, psi_n_2 = psis[0], psis[1], psis[2]
    # Generate new psi value psi(n+1)
    psi_np1 = psi_n + (dt/12)*(23*f_x_(c, dx, dt, psi_n) - 16*f_x_(c, dx, dt,␣
↪psi_n_1) + 5*f_x_(c, dx, dt, psi_n_2))

    return psi_np1
```

Define function to carry out the discretization and step through time.

[955]:
```
def advection_p1(dx, c, cfl, t_max, plot=False, plot_step=40, printout=False):
    # Compute timestep to meet Courant number
    dt = dx*cfl/c
    # Spatial domain
    x = np.arange(0, 1, dx)
    # Temporal domain
    t = np.arange(0, t_max, dt)

    # Initialize array for values and apply initial condition
    y = np.full((len(t)+1, len(x)), np.nan)
    y[0, :] = f(x)

    # Get exact solution
    exact = np.sin(2*np.pi*(x-c*t_max))**6

    # Iterate through timesteps
    for i, t_ in enumerate(t):
        if printout and i % 10 == 0:
            print('Step: {0} | Timestep: {1:.2f}'.format(i, t_))

        # Starter function
        if i < 2:
```

```
            print('Using starter scheme...') if printout else None
            y[i+1, :] = rk3(dt, c, dt, dx, x, y[i, :])
        else:
            print('Using full scheme...') if printout else None
            # Get function values from previous timesteps
            psis = [y[i, :], y[i-1, :], y[i-2, :]]
            y[i+1, :] = ab3(dx, dt, c, x, psis)

    ''' Plotting. '''
    if plot:
        plt.rcParams["axes.prop_cycle"] = plt.cycler('color', plt.cm.viridis(np.
↪linspace(0, 1, len(plot_step))))
        fig, ax = plt.subplots(figsize=(4, 3))
        for step in plot_step:
            im = ax.plot(x, y[step], marker='o', markersize=4, label='Step {0}'.
↪format(step))

        ax.set_title('Time = {0:.2f} | dx = {1:.2f} | dt = {2:.2f}'.
↪format(t[step], dx, dt), fontsize=10)
        ax.set_xlim([min(x), max(x)])
        ax.set_ylim([0, 1])
        fig.tight_layout()
        fig.legend(loc='upper right', bbox_to_anchor=(1.25, 0.925),␣
↪frameon=False, fontsize=8)

    # Return the values at the maximum time
    return x, y[-1, :], exact
```

Execute and plot the results

[ ]:
```
# Wave speed (c)
c = 0.1
# Spatial increments
dxs = [1/20, 1/40, 1/80, 1/160]
# Strings to represent spatial increments (for plotting purposes only)
dxs_str = ['1/20', '1/40', '1/80', '1/160']
# Courant number and Courant number list
cfls = [0.1, 0.2, 0.4, 0.8]
# Maximum time
t_max = 50
# Boolean to control prints to console
printout = False
# Computation mode: variable dx or CFL values
dx_calc = True

# Initialize array to hold each run's output array
values = {}
```

```
# Iterate over dx values
if dx_calc:
    cfl = 0.1
    for i, dx in enumerate(dxs):
        # End step
        end_step = int(t_max/(cfl*dx/c))
        # Perform calculation, plot individual results
        arr = advection_p1(dx, c, cfl, t_max, plot=False, plot_step=[end_step],␣
 ↪printout=printout)
        # Store array
        values['dx = {0}'.format(dxs_str[i])] = {'x': arr[0], 'y': arr[1],␣
 ↪'exact': arr[2]}

# Iterate over Courant number values
else:
    dx_index = 2
    dx = dxs[dx_index]
    for i, cfl in enumerate(cfls):
        # End step
        end_step = int(t_max/(cfl*dx/c))
        # Perform calculation, plot individual results
        arr = advection_p1(dx, c, cfl, t_max, plot=False, plot_step=[0,␣
 ↪end_step // 16, end_step // 2, 3*end_step // 4], printout=printout)
        # Store array
        values['CFL = {0}'.format(cfl)] = {'x': arr[0], 'y': arr[1], 'exact':␣
 ↪arr[2]}

''' Plotting.'''
# Initialize figure
fig, ax = plt.subplots(figsize=(4, 3))
# Define formatting index, color list, marker list
f_index, markers = 0, ['o', '^', '+', '2']
# Set color cycling
plt.rcParams["axes.prop_cycle"] = plt.cycler('color', plt.cm.Blues(np.
 ↪linspace(0.375, 1, len(dxs))))
# Plot the actual values
for key, value in values.items():
    im = ax.plot(value['x'], value['y'], marker=markers[f_index], markersize=5,␣
 ↪ls='--', label=key)
    if key == 'dx = 1/160' or key == 'CFL = 0.1':
        ax.plot(value['x'], value['exact'], c='r', label='exact', zorder=0)
    f_index += 1
# Metadata
if dx_calc:
    ax.set_title('Courant number: {0:.1f}'.format(cfl), fontsize=10)
```

```
else:
    ax.set_title('dx = {0}'.format(dxs_str[dx_index]), fontsize=10)
ax.set_xlim([min(value['x']), max(value['x'])])
ax.set_ylim([0, 1])
fig.tight_layout()
fig.legend(loc='upper right', bbox_to_anchor=(1.25, 0.925), frameon=False,␣
 ↪fontsize=8)
# plt.savefig('figs/p1c1.png', dpi=300, bbox_inches='tight')

# Initialize figure
fig, ax = plt.subplots(figsize=(4, 3))
# Collect RMS values
rmse = [rms(value['y'], value['exact']) for _, value in values.items()]
print(rmse)
# Plot the actual values
im = ax.loglog(dxs, rmse, marker='o', color='k')
ax.set_xticks([0.005, 0.01, 0.05])
ax.set_ylim([1e-3, 1e0])
# Metadata
ax.set_ylabel('RMSE')
ax.set_xlabel('dx') if dx_calc else ax.set_xlabel('CFL')
ax.set_title('Root mean squared error', fontsize=10)
plt.gca().invert_xaxis()
fig.tight_layout()
# plt.savefig('figs/p1c2.png', dpi=300, bbox_inches='tight')
```

0.0.2 Problem 2

Compute solutions to the wind speed advection equation using leapfrog with an Asselin filter in time and Lele's 4th-order compact differencing in space.

Runge-Kutta, 3rd order

[956]:
```
def rk3_2(h, c, dt, dx, x, psi):
    # q_1 = h*f_x(c, x)
    q_1 = h*f_x_2(c, dx, dt, psi)
    psi_1 = psi + (1/3)*q_1
    # q_2 = h*f_x(c, psi_1) - (5/9)*q_1
    q_2 = h*f_x_2(c, dx, dt, psi) - (5/9)*q_1
    psi_2 = psi_1 + (15/16)*q_2
    # q_3 = h*f_x(c, psi_2) - (153/128)*q_2
    q_3 = h*f_x_2(c, dx, dt, psi) - (153/128)*q_2
    psi_n_1 = psi_2 + (8/15)*q_3
    return psi_n_1
```

4th-order Lele compact difference

[957]:
```
def cdf_compact(dt, dx, f_):
    # f_ is an array of function values at a given time
    # dt is the timestep

    # Create upper, central, and bottom diagonals
    a = np.full(len(f_), 5/24)
    b = np.full(len(f_), 14/24)
    c = np.full(len(f_), 5/24)

    # Create RHS
    rhs = np.zeros((len(f_), ))
    for i in range(0, len(f_)):
        # Using modulo for indices to handle boundary conditions, since BCs are␣
 ↪periodic
        rhs[i] = (11*(f_[(i+1) % len(f_)] - f_[(i-1) % len(f_)])/2 + (f_[(i+2)␣
 ↪% len(f_)] - f_[(i-2) % len(f_)])/4)/(12*dx)

    f_prime = cyc_tridiag(len(f_), a, b, c, rhs)

    return f_prime
```

Leapfrog with Asselin filter

[958]:
```
def leapfrog(dx, dt, c, x, gamma, psis):
    # Get previous psi values [psi(n), psi(n-1)]
    psi_n, psi_n_1, psi_n_2 = psis
    # Asselin filter - previous step
    psi_n_1_ass = psi_n_1 + gamma*(psi_n - 2*psi_n_1 + psi_n_2)
    # Compute values at next time step
    psi_np1 = psi_n_1_ass + 2*dt*f_x_2(c, dx, dt, psi_n)
    # Compute filtered values at current time step
    psi_n_ass = psi_n + gamma*(psi_n_1_ass - 2*psi_n + psi_np1)

    return psi_n_ass, psi_np1
```

Define $\partial\psi/\partial x$.

[959]:
```
def f_x_2(c, dx, dt, f_):
    return c*cdf_compact(dt, dx, f_)
```

Define function to carry out the discretization and step through time

[960]:
```
def advection_p2(dx, c, cfl, t_max, plot=False, plot_step=40, printout=False):
    # Compute timestep to meet Courant number
    dt = dx*cfl/c
    # Spatial domain
    x = np.arange(0, 1, dx)
    # Temporal domain
```

```
    t = np.arange(0, t_max, dt)
    # Asselin filter value
    gamma = 0.1

    # Initialize array for values and apply initial condition
    y = np.full((len(t)+1, len(x)), np.nan)
    y[0, :] = f(x)

    # Get exact solution
    exact = np.sin(2*np.pi*(x-c*t_max))**6

    # Iterate through timesteps
    for i, t_ in enumerate(t):
        if printout and i % 10 == 0:
            print('Step: {0} | Timestep: {1:.2f}'.format(i, t_))

        # Starter function
        if i < 3:
            print('Using starter scheme...') if printout else None
            y[i+1, :] = rk3_2(dt, c, dt, dx, x, y[i, :])
        else:
            print('Using full scheme...') if printout else None
            # Get function values from previous timesteps
            psis = [y[i, :], y[i-1, :], y[i-2, :]]
            y[i, :], y[i+1, :] = leapfrog(dx, dt, c, x, gamma, psis)

    ''' Plotting. '''
    if plot:
        plt.rcParams["axes.prop_cycle"] = plt.cycler('color', plt.cm.viridis(np.
↪linspace(0, 1, len(plot_step))))
        fig, ax = plt.subplots(figsize=(4, 3))
        for step in plot_step:
            im = ax.plot(x, y[step], marker='o', markersize=4, label='Step {0}'.
↪format(step))

        ax.set_title('Time = {0:.2f} | dx = {1:.2f} | dt = {2:.2f}'.
↪format(t[step], dx, dt), fontsize=10)
        ax.set_xlim([min(x), max(x)])
        ax.set_ylim([0, 1])
        fig.tight_layout()
        fig.legend(loc='upper right', bbox_to_anchor=(1.25, 0.925),␣
↪frameon=False, fontsize=8)
        plt.show()

    # Return the values at the maximum time
    return x, y[-1, :], exact
```

Execute and plot the runs

[ ]:
```
# Wave speed (c)
c = 0.1
# Spatial increments
dxs = [1/20, 1/40, 1/80, 1/160]
# Strings to represent spatial increments (for plotting purposes only)
dxs_str = ['1/20', '1/40', '1/80', '1/160']
# Courant number and Courant number list
cfls = [0.1, 0.2, 0.4, 0.8]
# Maximum time
t_max = 50
# Boolean to control prints to console
printout = False
# Computation mode: variable dx or CFL values
dx_calc = False

# Initialize array to hold each run's output array
values = {}

# Iterate over dx values
if dx_calc:
    cfl = 0.1
    for i, dx in enumerate(dxs):
        # End step
        end_step = int(t_max/(cfl*dx/c))
        # Perform calculation, plot individual results
        arr = advection_p2(dx, c, cfl, t_max, plot=False, plot_step=[0],␣
 ↪printout=printout)
        # Store array
        values['dx = {0}'.format(dxs_str[i])] = {'x': arr[0], 'y': arr[1],␣
 ↪'exact': arr[2]}

# Iterate over Courant number values
else:
    dx_index = 2
    dx = dxs[dx_index]
    for i, cfl in enumerate(cfls):
        # End step
        end_step = int(t_max/(cfl*dx/c))
        # Perform calculation, plot individual results
        arr = advection_p2(dx, c, cfl, t_max, plot=False, plot_step=[end_step],␣
 ↪printout=printout)
        # Store array
        values['CFL = {0}'.format(cfl)] = {'x': arr[0], 'y': arr[1], 'exact':␣
 ↪arr[2]}
```

```
''' Plotting. '''
# Initialize figure
fig, ax = plt.subplots(figsize=(4, 3))
# Define formatting index, color list, marker list
f_index, markers = 0, ['o', '^', '+', '2']
# Set color cycling
plt.rcParams["axes.prop_cycle"] = plt.cycler('color', plt.cm.Blues(np.
 ↪linspace(0.375, 1, len(dxs))))
# Plot the actual values
for key, value in values.items():
    im = ax.plot(value['x'], value['y'], marker=markers[f_index], markersize=5,␣
 ↪ls='--', label=key)
    if key == 'dx = 1/160' or key == 'CFL = 0.1':
        ax.plot(value['x'], value['exact'], c='r', label='exact', zorder=0)
    f_index += 1
# Metadata
if dx_calc:
    ax.set_title('Courant number: {0:.1f}'.format(cfl), fontsize=10)
else:
    ax.set_title('dx = {0}'.format(dxs_str[dx_index]), fontsize=10)
ax.set_xlim([min(value['x']), max(value['x'])])
ax.set_ylim([0, 1])
fig.tight_layout()
fig.legend(loc='upper right', bbox_to_anchor=(1.25, 0.925), frameon=False,␣
 ↪fontsize=8)
# plt.savefig('figs/p2c1.png', dpi=300, bbox_inches='tight')

# Initialize figure
fig, ax = plt.subplots(figsize=(4, 3))
# Collect RMS values
rmse = [rms(value['y'], value['exact']) for _, value in values.items()]
print(rmse)
# Plot the actual values
im = ax.loglog(dxs, rmse, marker='o', color='k')
ax.set_xticks([0.005, 0.01, 0.05])
ax.set_ylim([1e-3, 1e0])
# Metadata
ax.set_ylabel('RMSE')
ax.set_xlabel('dx') if dx_calc else ax.set_xlabel('CFL')
ax.set_title('Root mean squared error', fontsize=10)
plt.gca().invert_xaxis()
fig.tight_layout()
# plt.savefig('figs/p2c2.png', dpi=300, bbox_inches='tight')\
```

0.0.3 Problem 3

(a) Derive coefficient values that give a 3rd-order accurate scheme in space and time.

(a) (i). Derive a 3rd-order accurate scheme for space.

[ ]:
```
from sympy import *

# Initialize symbolic variable
h = Symbol('h')
# Define the matrix system
A = Matrix([[1, 1, 1, 1],
            [-2*h, -h, 0, h],
            [2*(h**2), (1/2)*(h**2), 0, (1/2)*(h**2)],
            [(-4/3)*(h**3), (-1/6)*(h**3), 0, (1/6)*(h**3)]])
B = Matrix([0, -1, 0, 0])

# Solve the linear system
system = A, B
linsolve(system)
```

Test the coefficients to ensure 3rd-order accuracy.

[806]:
```
a, b, c, d, e = -1/6, 1, -1/2, -1/3, 1

#
print('0th-order: {0:.3f}'.format(a + b + c + d + 0*e))
# '
print('1st-order: {0:.3f}'.format(-2*a - b + 0*c + d + e))
# ''
print('2nd-order: {0:.3f}'.format(2*a + (1/2)*b + 0*c + (1/2)*d + 0*e))
# '''
print('3rd-order: {0:.3f}'.format((-4/3)*a - (1/6)*b + 0*c + (1/6)*d + 0*e))
# '''
print('4th-order: {0:.3f}'.format((16/24)*a + (1/24)*b + 0*c + (1/24)*d + 0*e))
```

```
0th-order: 0.000
1st-order: 0.000
2nd-order: 0.000
3rd-order: 0.000
4th-order: -0.083
```

(a) (ii) Do this for the spatial representation of time of the $\partial^2\psi/\partial x^2$ term.

- Trial 1: do regular Taylor table
- Trial 2: shift by a derivative up (instead of starting at ψ, start at $\partial\psi$).

[967]:
```
from sympy import *

# Initialize symbolic variable
h = Symbol('h')
# Define the matrix system
```

```
''' Trial 1. '''
A_1 = Matrix([[1, 1, 1, 1],
              [-2*h, -h, 0, h],
              [2*(h**2), (1/2)*(h**2), 0, (1/2)*(h**2)],
              [(-4/3)*(h**3), (-1/6)*(h**3), 0, (1/6)*(h**3)]])
B_1 = Matrix([0, 0, -1, 0])

''' Trial 2. '''
# A_1 = Matrix([[-2*h, -h, h],
#               [2*(h**2), (1/2)*(h**2), (1/2)*(h**2)],
#               [(-4/3)*(h**3), (-1/6)*(h**3), (1/6)*(h**3)],
#               [(2/3)*(h**4), (1/24)*(h**4), (1/24)*(h**4)]])
# B_1 = Matrix([0, -1, 0])

# Solve the linear system
system_1 = A_1, B_1
linsolve(system_1)
```

[967]:
$$\left\{\left(\frac{2.22044604925031 \cdot 10^{-16}}{h^2}, -\frac{1.0}{h^2}, \frac{2.0}{h^2}, -\frac{1.0}{h^2}\right)\right\}$$

Test the coefficients to make sure they're 3rd-order accurate

[865]:
```
a, b, c, d, e = 0, -1, 2, -1, 1

#
print('0th-order: {0:.3f}'.format(a + b + c + d + 0*e))
# '
print('1st-order: {0:.3f}'.format(-2*a - b + 0*c + d + 0*e))
# ''
print('2nd-order: {0:.3f}'.format(2*a + (1/2)*b + 0*c + (1/2)*d + e))
# '''
print('3rd-order: {0:.3f}'.format((-4/3)*a - (1/6)*b + 0*c + (1/6)*d + 0*e))
# '''
print('4th-order: {0:.3f}'.format((16/24)*a + (1/24)*b + 0*c + (1/24)*d + 0*e))
```

```
0th-order: 0.000
1st-order: 0.000
2nd-order: 0.000
3rd-order: 0.000
4th-order: -0.083
```

(b) (iii) Do this for the spatial representation of time of the $\partial^3\psi/\partial x^3$ term.

[884]:
```
from sympy import *

# Initialize symbolic variable
h = Symbol('h')
# Define the matrix system
```

```
''' Trial 1. '''
A_2 = Matrix([[1, 1, 1, 1],
              [-2*h, -h, 0, h],
              [2*(h**2), (1/2)*(h**2), 0, (1/2)*(h**2)],
              [(-4/3)*(h**3), (-1/6)*(h**3), 0, (1/6)*(h**3)]])
B_2 = Matrix([0, 0, 0, -1])

''' Trial 2. '''
# A_1 = Matrix([[-2*h, -h, h],
#               [2*(h**2), (1/2)*(h**2), (1/2)*(h**2)],
#               [(-4/3)*(h**3), (-1/6)*(h**3), (1/6)*(h**3)],
#               [(2/3)*(h**4), (1/24)*(h**4), (1/24)*(h**4)]])
# B_1 = Matrix([0, -1, 0])

# Solve the linear system
system_2 = A_2, B_2
linsolve(system_2)
```

[884]:
$$\left\{\left(\frac{1.0}{h^3}, -\frac{3.0}{h^3}, \frac{3.0}{h^3}, -\frac{1.0}{h^3}\right)\right\}$$

Test the coefficients to make sure they're 3rd-order accurate

[887]:
```
a, b, c, d, e = 1, -3, 3, -1, 1

#
print('0th-order: {0:.3f}'.format(a + b + c + d + 0*e))
# '
print('1st-order: {0:.3f}'.format(-2*a - b + 0*c + d + 0*e))
# ''
print('2nd-order: {0:.3f}'.format(2*a + (1/2)*b + 0*c + (1/2)*d + 0*e))
# '''
print('3rd-order: {0:.3f}'.format((-4/3)*a - (1/6)*b + 0*c + (1/6)*d + e))
# '''
print('4th-order: {0:.3f}'.format((16/24)*a + (1/24)*b + 0*c + (1/24)*d + 0*e))
```

```
0th-order: 0.000
1st-order: 0.000
2nd-order: 0.000
3rd-order: 0.000
4th-order: 0.500
```

0.0.4 Auxiliary functions

Root mean square (RMS) calculation

[947]:
```
```

```
def rms(x, y):
    return np.sqrt(np.nansum(np.array([(x[i] - y[i])**2 for i in range(0,␣
 ↪len(x))])/len(x)))
```

Cyclic tridiagonal solver

[946]:
```
def cyc_tridiag(jmx, a, b, c, f):

    ''' Written by D. Durran, translated by S. Ditkovsky. '''

    # jmx = dimension of all arrays
    # a = sub (lower) diagonal
    # b = center diagonal
    # c = super (upper) diagonal
    # f = right hand side

    fmx = f[-1]

    # Create work arrays
    q = np.empty(jmx)
    s = np.empty(jmx)

    #forward elimination sweep
    q[0] = -c[0]/b[0]
    f[0] = f[0]/b[0]
    s[0] = -a[0]/b[0]

    for j in range(jmx-1):
        p = 1./(b[j+1]+ a[j+1]*q[j])
        q[j+1] = -c[j+1]*p
        f[j+1] = (f[j+1] - a[j+1]*f[j])*p
        s[j+1] = -a[j+1]*s[j]*p

    #Backward pass

    q[-1] = 0.0
    s[-1] = 1.0

    for j in reversed(range(jmx-1)):
        s[j] = s[j] + q[j]*s[j+1]
        q[j] = f[j] + q[j]*q[j+1]

    #final pass
    f[-1] = (fmx-c[-1]*q[0] - a[-1]*q[-2])/(c[-1]*s[0] + a[-1]*s[-2] + b[-1])

    for j in range(jmx-1):
        f[j] = f[-1]*s[j] + q[j]
```

```
    return f
```