

Case Study # 4: Linear 1D Transport Equation

John Karasinski

Graduate Student Researcher

Center for Human/Robotics/Vehicle Integration and Performance

Department of Mechanical and Aerospace Engineering

University of California

Davis, California 95616

Email: karasinski@ucdavis.edu

1 Problem Description

The transport of various scalar quantities (e.g species mass fraction, temperature) in flows can be modeled using a linear convection-diffusion equation (presented here in a 1-D form),

$$\frac{\partial \phi}{\partial t} + u \frac{\partial \phi}{\partial x} = D \frac{\partial^2 \phi}{\partial x^2}, \quad (1)$$

ϕ is the transported scalar, u and D are known parameters (flow velocity and diffusion coefficient respectively). The purpose of this case study is to investigate the behavior of various numerical solutions of this equation.

This case study focuses on the solution of the 1-D linear transport equation (above) for $x \in [0, L]$ and $t \in [0, \tau]$ (where $\tau = 1/k^2 D$) subject to periodic boundary conditions and the following initial condition

$$\phi(x, 0) = \sin(kx), \quad (2)$$

with $k = 2\pi/L$ and $L = 1$ m. The convection velocity is $u = 0.2$ m/s, and the diffusion coefficient is $D = 0.005$ m²/s.

This problem has an analytical solution [1],

$$\Phi(x, t) = \exp(-k^2 D t) \sin[k(x - ut)]. \quad (3)$$

Numerical solutions of this problem were created using the following schemes:

1. FTCS (Explicit) Forward-Time and central differencing for both the convective flux and the diffusive flux.
2. Upwind Finite Volume method: Explicit (forward Euler), with the convective flux treated using the basic upwind method and the diffusive flux treated using central differencing.
3. Trapezoidal (AKA Crank-Nicholson).
4. QUICK Finite Volume method: Explicit, with the convective flux treated using the QUICK method and the diffusive flux treated using central differencing.

The following cases are considered:

$$(C, s) \in \{(0.1, 0.25), (0.5, 0.25), (2, 0.25), (0.5, 0.5), (0.5, 1)\}$$

where $C = u\Delta t/\Delta x$ and $s = D\Delta t/\Delta x^2$. A uniform mesh for all solvers and cases. The stability and accuracy of these schemes was investigated.

2 Numerical Solution Approach

Four schemes were developed to investigate the five considered cases. These are an explicit FTCS scheme, an upwind finite volume scheme, an implicit trapezoidal scheme, and a QUICK finite volume scheme. Each case makes use of C and s to compute the spatial ($\Delta x = CD/us$) and temporal ($\Delta t = C\Delta x/u$) discretizations.

2.1 FTCS Scheme

The first scheme involves using forward-time and central differencing (FTCS) for both the convective flux and the diffusive flux and yields second-order convergence in space and first-order convergence in time. In order to implement this method, the domain of the problem must be discretized. This method calculates the state of the system at a later time from the state of the system at the current time, and is thus an explicit method. For the 1-D transport equation on a uniform grid, the state ϕ at grid point i and timestep f can be calculated by the following equation,

$$\begin{aligned} \phi_i^f = & \left(\frac{D\Delta t}{\Delta x^2} + \frac{u\Delta t}{2\Delta x} \right) \phi_{i-1}^{f-1} + \\ & \left(1 - \frac{2D\Delta t}{\Delta x^2} \right) \phi_i^{f-1} + \\ & \left(\frac{D\Delta t}{\Delta x^2} - \frac{u\Delta t}{2\Delta x} \right) \phi_{i+1}^{f-1}. \end{aligned} \quad (4)$$

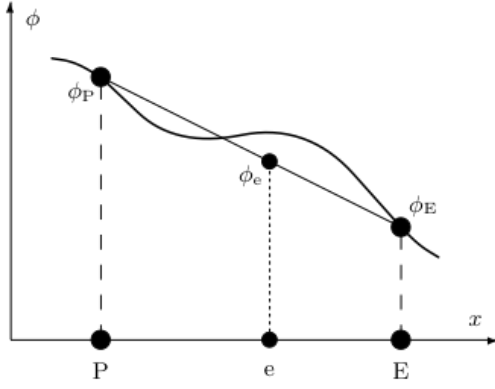


Fig. 1: The 1-D FTCS scheme interpolates between the two nearby grid points [1]

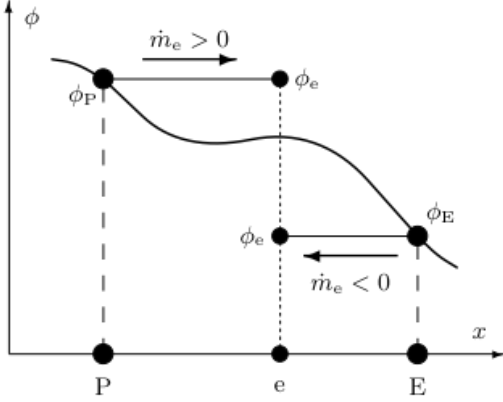


Fig. 2: Upwind scheme's interpolation for the diffusive flux [1]

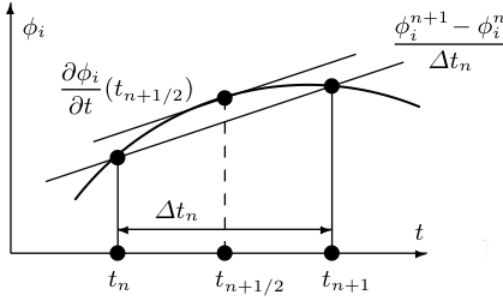


Fig. 3: Trapezoidal scheme's interpolation for the time derivative [1]

To impose the periodic boundary condition, the last node in the domain reaches around to the second node, while the first node is set equivalent to the last node. This scheme is numerically stable as long as the following conditions are satisfied:

$$C \leq \sqrt{2su} \text{ and } s \leq \frac{1}{2}. \quad (5)$$

2.2 Upwind Scheme

The second scheme is an explicit upwind finite volume method. For this method the convective flux is treated using the basic upwind method and the diffusive flux treated using central differencing. This is a second-order scheme which uses a three point backward difference, as described below

$$\begin{aligned} \phi_i^f = & \left(\frac{D\Delta t}{\Delta x^2} \right) \left[\phi_{i+1}^{f-1} - 2\phi_i^{f-1} + \phi_{i-1}^{f-1} \right] - \\ & \left(\frac{u\Delta t}{2\Delta x} \right) \left[3\phi_i^{f-1} - 4\phi_{i-1}^{f-1} + \phi_{i-2}^{f-1} \right] + \\ & \phi_i^{f-1}. \end{aligned} \quad (6)$$

The upwind method is more stable than the FTCS scheme, and unlike the FTCS scheme, the stability of the Upwind scheme does not depend on u . To impose the periodic boundary condition, the last node and the second node in the domain reach across the edge of the domain, while the first node is set equivalent to the last node. This scheme is numerically stable as long as the following condition is satisfied:

$$C + 2s \leq 1. \quad (7)$$

2.3 Trapezoidal (Crank-Nicholson) Scheme

The Trapezoidal scheme is a finite difference method which is implicit and unconditionally stable. This method is an equally weighted average of the explicit and implicit central difference solutions. As this is an implicit method, a system of algebraic equations must be solved to find values of the transported scalar for the next timestep. This problem requires the solution of a nearly tridiagonal matrix, with the exception of the top right and bottom left corners, which are set to impose the periodic boundary condition [2].

The following set of equations must be solved to advance the solution to the next timestep:

$$\begin{bmatrix} b & c & & a \\ a & b & c & \\ & a & b & \ddots \\ & & \ddots & \ddots & c \\ c & & a & b \end{bmatrix} \begin{bmatrix} \phi_1^f \\ \phi_2^f \\ \phi_3^f \\ \vdots \\ \phi_i^f \end{bmatrix} = \begin{bmatrix} RHS_1^f \\ RHS_2^f \\ RHS_3^f \\ \vdots \\ RHS_i^f \end{bmatrix}, \quad (8)$$

where $a = -A - B$, $b = 1 + 2A$, and $c = -A + B$, and where

$$A = \frac{D\Delta t}{2\Delta x^2} \text{ and } B = \frac{u\Delta t}{4\Delta x}, \quad (9)$$

The right hand side of the equation is a linear combination of the solutions from the previous timestep,

$$\begin{aligned} RHS_i^f = & A(\phi_{i+1}^{f-1} - 2\phi_i^{f-1} + \phi_{i-1}^{f-1}) - \\ & B(\phi_{i+1}^{f-1} - \phi_{i-1}^{f-1}) + \\ & \phi_i^{f-1}. \end{aligned} \quad (10)$$

2.4 QUICK Scheme

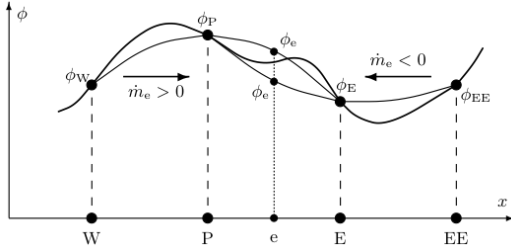


Fig. 4: The QUICK scheme interpolates between two quadratic equations [1]

The Quadratic Upstream Interpolation for Convective Kinematics (QUICK) method is an explicit method which uses three point upstream weighted quadratic interpolation for cell phase values (see Figure 4). Here the convective flux is treated using the QUICK method, while the diffusive flux treated using central differencing. This scheme is second-order accurate for the finite difference model [3]. This can be implemented with the following equation:

$$\begin{aligned} \phi_i^f = & \left(\frac{D\Delta t}{\Delta x^2} \right) \left[\phi_{i+1}^{f-1} - 2\phi_i^{f-1} + \phi_{i-1}^{f-1} \right] - \\ & \left(\frac{u\Delta t}{8\Delta x} \right) \left[3\phi_{i+1}^{f-1} + 3\phi_i^{f-1} + \phi_{i-2}^{f-1} - 7\phi_{i-1}^{f-1} \right] + \\ & \phi_i^{f-1} \end{aligned} \quad (11)$$

To impose the periodic boundary condition, the last node and the second node in the domain reach across the edge of the domain, while the first node is set equivalent to the last node. This scheme is numerically stable under the following condition:

$$C \leq \min(2 - 4s, \sqrt{2s}). \quad (12)$$

3 Results Discussion

3.1 Stability

For the results below, cases 1, 2, 3, 4, 5 refer to $(C, s) = (0.1, 0.25), (0.5, 0.25), (2, 0.25), (0.5, 0.5), (0.5, 1)$, respectively. The stability for each scheme was investigated for each case. The stability criteria for the FTCS, Upwind, and QUICK schemes can be found as Equations 5, 7, and 12 [4]. For the cases considered, FTCS was least stable, the Upwind and QUICK schemes were effectively equally stable, and the Trapezoidal scheme is always stable. For the full results, see Table 1.

3.2 NRMS

The computational result for the 1-D linear convection-diffusion equation can be compared to the analytical result

Case	FTCS	Upwind	Trap	QUICK
1	True	True	True	True
2	False	True	True	True
3	False	False	True	False
4	False	False	True	False
5	False	False	True	False

Table 1: Stability results for each case and method

Case	FTCS	Upwind	Trap	QUICK
1	7.23E-03	9.68E-03	2.42E-02	7.67E-03
2	2.23E-01	2.89E-01	1.30E-01	2.32E-01
3	8.26E+00	3.64E+01	7.72E-01	1.24E+01
4	1.06E-01	6.99E+22	4.56E-02	1.10E-01
5	1.10E+61	1.28E+97	2.14E-02	7.37E+71

Table 2: NRMS results for each case and method

above, Equation 3. The Root Mean Square error,

$$RMSE = \sqrt{\frac{1}{N} \sum_{i=1}^N [\phi_i - \phi_i^*]^2}, \quad (13)$$

and the Normalized Root Mean Square error,

$$NRMS = \frac{RMSE}{\max(\phi^*) - \min(\phi^*)}, \quad (14)$$

can be calculated. Here ϕ_i is the computational result for the the transported scalar for each point on the 1-D domain, ϕ_i^* is the analytical solution, and N is the number of points on the 1-D domain. The NRMS for each case is expressed as a percentage, where lower values indicate a result closer to the analytic solution. For the complete NRMS results for each case and scheme, see Table 2.

The lowest NRMS error is found by using the FTCS scheme under Case 1. Despite this, the FTCS case quickly blows up for Cases 3 and 5 due to numerical instability. The QUICK method performs similarly, with approximately the same error for all cases. The Trapezoidal method's unconditional stability leads to it performing best for all cases aside from Case 1.

The cases of large NRMS arise from the loss of stability in the scheme. The effect of instability can be seen quite clearly in Figure 8. Making use of Equation 12 with values $C = 0.5$ and $s = 1$, for instance, one can see that

$$\begin{aligned} C &\leq \min(2 - 4s, \sqrt{2s}) \\ 0.5 &\leq \min(-2, \sqrt{2}) \\ 0.5 &\leq -2 \end{aligned} \quad (15)$$

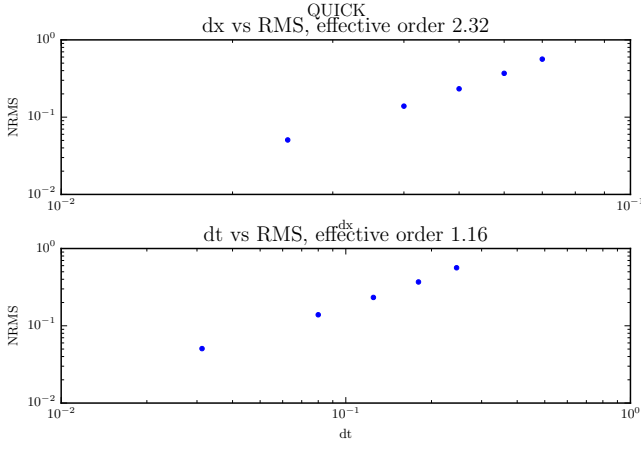


Fig. 5: Effective order of the QUICK method

Method	Δx	Δt
FTCS	2.06	1.06
Upwind	1.08	0.55
Trap	1.97	0.99
QUICK	2.32	1.16

Table 3: Effective order of each method for Δx and Δt

is false. This leads to the instability and resultant large NRMS of $4.70\text{E}+85$.

3.3 Effective Order

The effective order of each method was calculated by fitting the NRMS for cases within the stability region of each method. The effective order was found by fitting with a linear function against a log log plot of the NRMS versus the Δx and Δt , see Figure 5 for an example. The slope of the fit estimates the order of accuracy of the method. The effective orders of accuracy for the FTCS, Trapezoidal, and QUICK methods are approximately 2 for the spatial dimension and approximately 1 for the temporal dimension. The Upwind method is approximately first order accurate in the spatial dimension. For full results, see Table 3.

4 Conclusion

Only the first case led to stable solutions for each scheme. The results from this case can be seen in Figure 6. The error between each scheme's result and the analytic solution can be seen in Figure 7. This error shows that the three explicit methods can perform better than the implicit method for low CFL numbers, though for larger CFL numbers the opposite is true.

While the FTCS and QUICK schemes produced the lowest error in the majority of the considered cases, the unconditional stability of the Trapezoidal method makes it the more reliable method if the C and s values cannot be chosen freely.

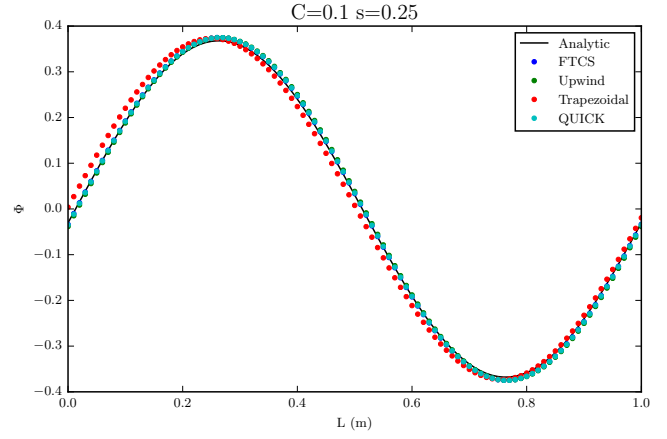


Fig. 6: Results of each scheme for Case 1

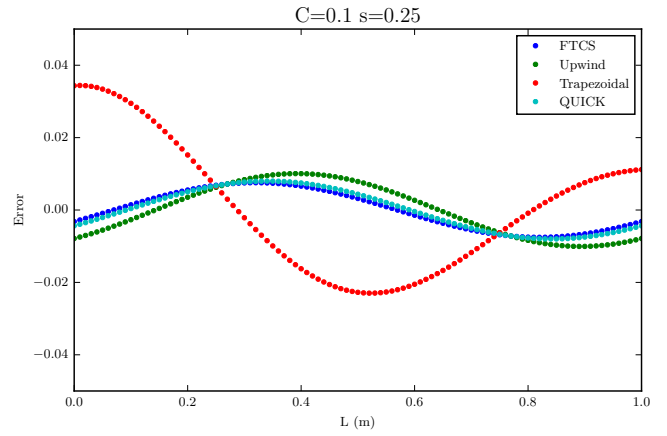


Fig. 7: Error for each scheme for Case 1

References

- [1] Tannehill, J. C., Anderson, D. A., and Pletcher, R. C., 1997. *Computational Fluid Mechanics and Heat Transfer*, 2nd ed. Taylor & Francis.
- [2] Hogarth, W., Noye, B., Stagnitti, J., Parlange, J., and Bolt, G., 1990. "A comparative study of finite difference methods for solving the one-dimensional transport equation with an initial-boundary value discontinuity". *Computers & Mathematics with Applications*, **20**(11), pp. 67–82.
- [3] Chen, Y., and Falconer, R. A., 1992. "Advection-diffusion modelling using the modified quick scheme". *International journal for numerical methods in fluids*, **15**(10), pp. 1171–1196.
- [4] Tryggvason, G., 2013. The advection-diffusion equation. <http://www3.nd.edu/~gtryggva/CFD-Course/>.

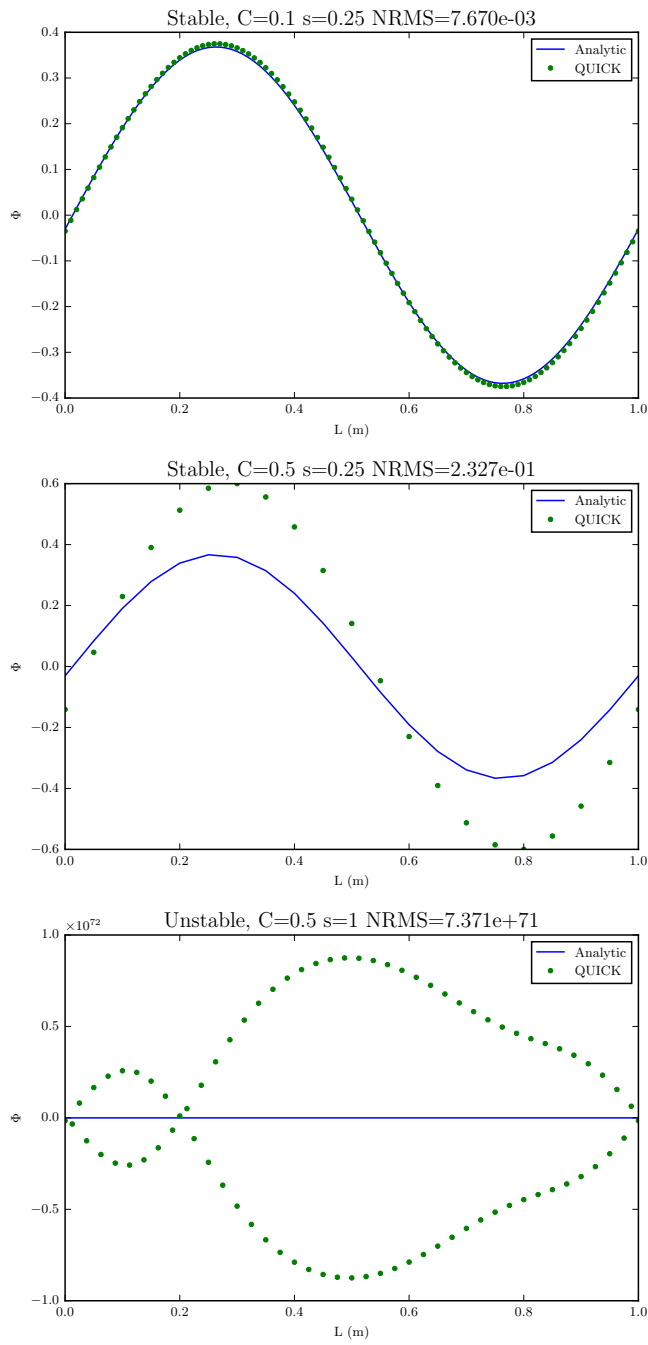


Fig. 8: QUICK method's transition into instability

Appendix A: Python Code

```
1 from PrettyPlots import *
2 import numpy as np
3 import scipy.sparse as sparse
4
5
6 class Config(object):
7     def __init__(self, C, s):
8         # Import parameters
9         self.C = C
10        self.s = s
11
12        # Problem constants
13        self.L = 1.          # m
14        self.D = 0.005       # m^2/s
15        self.u = 0.2         # m/s
16        self.k = 2 * np.pi / self.L # m^-1
17        self.tau = 1 / (self.k ** 2 * self.D)
18
19        # Set-up Mesh and Calculate time-step
20        self.dx = self.C * self.D / (self.u * self.s)
21        self.dt = self.C * self.dx / self.u
22        self.x = np.append(np.arange(0, self.L, self.dx), self.L)
23
24
25 def Analytic(c):
26     k, D, u, tau, x = c.k, c.D, c.u, c.tau, c.x
27
28     N = len(x)
29     Phi = np.array(x)
30
31     for i in range(0, N):
32         Phi[i] = np.exp(-k ** 2 * D * tau) * np.sin(k * (x[i] - u * tau))
33
34     return np.array(Phi)
35
36
37 def FTCS(Phi, c):
38     """
39     FTCS (Explicit) - Forward-Time and central differencing for both the
40     convective flux and the diffusive flux.
41     """
42
43     D, dt, dx, u, tau = c.D, c.dt, c.dx, c.u, c.tau
44
45     N = len(Phi)
46     Phi = np.array(Phi)
47     Phi_old = np.array(Phi)
48
49     A = D * dt / dx ** 2
50     B = u * dt / (2 * dx)
51
52     t = 0
53     while t < tau:
54         for i in range(1, N - 1):
55             Phi[i] = ((A + B) * Phi_old[i - 1] +
56                      (1 - 2 * A) * Phi_old[i] +
57                      (A - B) * Phi_old[i + 1])
58
59         # Enforce our periodic boundary condition
60         Phi[-1] = ((A + B) * Phi_old[-2] +
61                   (1 - 2 * A) * Phi_old[-1] +
62                   (A - B) * Phi_old[1])
63         Phi[0] = Phi[-1]
64
65         Phi_old = np.array(Phi)
66         t += dt
67
```

```

68     return np.array(Phi_old)
69
70
71 def Upwind(Phi, c):
72     '''
73     Upwind-Finite Volume method: Explicit (forward Euler), with the convective
74     flux treated using the basic upwind method and the diffusive flux treated
75     using central differencing.
76     '''
77
78     D, dt, dx, u, tau = c.D, c.dt, c.dx, c.u, c.tau
79
80     N = len(Phi)
81     Phi = np.array(Phi)
82     Phi_old = np.array(Phi)
83
84     A = D * dt / dx ** 2
85     B = u * dt / (2 * dx)
86
87     t = 0
88     while t <= tau:
89         for i in range(2, N - 1):
90             Phi[i] = (A * (Phi_old[i + 1] - 2 * Phi_old[i] + Phi_old[i - 1]) -
91                      B * (3 * Phi_old[i] - 4 * Phi_old[i - 1] + Phi_old[i - 2]) +
92                      Phi_old[i])
93
94             Phi[-1] = (A * (Phi_old[1] - 2 * Phi_old[-1] + Phi_old[-2]) -
95                       B * (3 * Phi_old[-1] - 4 * Phi_old[-2] + Phi_old[-3]) +
96                       Phi_old[-1])
97             Phi[0] = Phi[-1]
98             Phi[1] = (A * (Phi_old[2] - 2 * Phi_old[1] + Phi_old[-1]) -
99                      B * (3 * Phi_old[1] - 4 * Phi_old[-1] + Phi_old[-2]) +
100                      Phi_old[1])
101
102             Phi_old = np.array(Phi)
103             t += dt
104
105     return np.array(Phi_old)
106
107
108 def Trapezoidal(Phi, c):
109     D, dt, dx, u, tau = c.D, c.dt, c.dx, c.u, c.tau
110
111     N = len(Phi)
112     Phi = np.array(Phi)
113     Phi_old = np.array(Phi)
114
115     A = dt * D / (2 * dx**2)
116     B = dt * u / (4 * dx)
117
118     # Create Coefficient Matrix
119     lower = [-A - B for _ in range(0, N)]
120     main = [1 + 2 * A for _ in range(0, N)]
121     upper = [-A + B for _ in range(0, N)]
122
123     data = lower, main, upper
124     diags = np.array([-1, 0, 1])
125     matrix = sparse.spdiags(data, diags, N, N).todense()
126
127     # Set values for periodic boundary conditions
128     matrix[0, N - 1] = -A - B
129     matrix[N - 1, 0] = -A + B
130
131     # Initialize RHS
132     RHS = np.array(Phi_old)
133
134     t = 0
135     while t <= tau:
136         # Enforce our periodic boundary condition

```

```

137     RHS[0] = (A * (Phi_old[1] - 2 * Phi_old[0] + Phi_old[-1]) -
138              B * (Phi_old[1] - Phi_old[-1]) +
139              Phi_old[0])
140
141     for i in range(1, N - 1):
142         RHS[i] = (A * (Phi_old[i + 1] - 2 * Phi_old[i] + Phi_old[i - 1]) -
143                  B * (Phi_old[i + 1] - Phi_old[i - 1]) +
144                  Phi_old[i])
145
146     # Enforce our periodic boundary condition
147     RHS[-1] = (A * (Phi_old[0] - 2 * Phi_old[-1] + Phi_old[-2]) -
148               B * (Phi_old[0] - Phi_old[-2]) +
149               Phi_old[-1])
150
151     # Solve matrix
152     Phi = np.linalg.solve(matrix, RHS)
153
154     Phi_old = np.array(Phi)
155     t += dt
156
157     return np.array(Phi_old)
158
159
160 def QUICK(Phi, c):
161     D, dt, dx, u, tau = c.D, c.dt, c.dx, c.u, c.tau
162
163     N = len(Phi)
164     Phi = np.array(Phi)
165     Phi_old = np.array(Phi)
166
167     A = D * dt / dx**2
168     B = u * dt / (8 * dx)
169
170     t = 0
171     while t <= tau:
172         for i in range(2, N - 1):
173             Phi[i] = (A * (Phi_old[i + 1] - 2 * Phi_old[i] + Phi_old[i - 1]) -
174                      B * (3 * Phi_old[i + 1] + Phi_old[i - 2] - 7 * Phi_old[i - 1] + 3 * Phi_old[i]) +
175                      Phi_old[i])
176
177             Phi[-1] = (A * (Phi_old[1] - 2 * Phi_old[-1] + Phi_old[-2]) -
178                        B * (3 * Phi_old[1] + Phi_old[-3] - 7 * Phi_old[-2] + 3 * Phi_old[-1]) +
179                        Phi_old[-1])
180             Phi[0] = Phi[-1]
181             Phi[1] = (A * (Phi_old[2] - 2 * Phi_old[1] + Phi_old[0]) -
182                       B * (3 * Phi_old[2] + Phi_old[-2] - 7 * Phi_old[0] + 3 * Phi_old[1]) +
183                       Phi_old[1])
184
185         # Increment
186         Phi_old = np.array(Phi)
187         t += dt
188
189     return np.array(Phi_old)
190
191
192 def stability(c):
193     C, s, u = c.C, c.s, c.u
194
195     FTCS = C <= np.sqrt(2 * s * u) and s <= 0.5
196     Upwind = C + 2*s <= 1
197     Trapezoidal = True
198     QUICK = C <= min(2 - 4 * s, np.sqrt(2 * s))
199
200     # print('C = ', C, ' s = ', s)
201     # print('FTCS: ' + str(FTCS))
202     # print('Upwind: ' + str(Upwind))
203     # print('Trapezoidal: ' + str(Trapezoidal))
204     # print('QUICK: ' + str(QUICK))
205

```



```

206     return [FTCS, Upwind, Trapezoidal, QUICK]
207
208
209 def calc_stability(C, s, solver):
210     results = []
211     for C_i, s_i in zip(C, s):
212         out = generate_solutions(C_i, s_i, find_order=True)
213         results.append(out)
214
215     # Sort and convert
216     results.sort(key=lambda x: x[0])
217     results = np.array(results)
218
219     # Pull out data
220     x = results[:, 0]
221     t = results[:, 1]
222     RMS_FTCS = results[:, 2]
223     RMS_Upwind = results[:, 3]
224     RMS_Trapezoidal = results[:, 4]
225     RMS_QUICK = results[:, 5]
226
227     # Plot effective orders
228     rms_list = [(RMS_FTCS, 'FTCS'),
229                (RMS_Upwind, 'Upwind'),
230                (RMS_Trapezoidal, 'Trapezoidal'),
231                (RMS_QUICK, 'QUICK')]
232
233     for rms in rms_list:
234         if rms[1] == solver:
235             plot_order(x, t, rms)
236
237
238 def generate_solutions(C, s, find_order=False):
239     c = Config(C, s)
240
241     # Spit out some stability information
242     stable = stability(c)
243
244     # Initial Condition with boundary conditions
245     Phi_initial = np.sin(c.k * c.x)
246
247     # Analytic Solution
248     Phi_analytic = Analytic(c)
249
250     # Explicit Solution
251     Phi_ftcs = FTCS(Phi_initial, c)
252
253     # Upwind Solution
254     Phi_upwind = Upwind(Phi_initial, c)
255
256     # Trapezoidal Solution
257     Phi_trapezoidal = Trapezoidal(Phi_initial, c)
258
259     # QUICK Solution
260     Phi_quick = QUICK(Phi_initial, c)
261
262     # Save group comparison
263     solutions = [(Phi_ftcs, 'FTCS'),
264                 (Phi_upwind, 'Upwind'),
265                 (Phi_trapezoidal, 'Trapezoidal'),
266                 (Phi_quick, 'QUICK')]
267
268     if not find_order:
269         # Save individual comparisons
270         save_figure(c.x, Phi_analytic, Phi_ftcs,
271                   'FTCS ' + str(C) + ' ' + str(s), stable[0])
272         save_figure(c.x, Phi_analytic, Phi_upwind,
273                   'Upwind ' + str(C) + ' ' + str(s), stable[1])
274         save_figure(c.x, Phi_analytic, Phi_trapezoidal,

```

```

275         'Trapezoidal ' + str(C) + ' ' + str(s), stable[2])
276     save_figure(c.x, Phi_analytic, Phi_quick,
277               'QUICK ' + str(C) + ' ' + str(s), stable[3])
278
279     # and group comparisons
280     save_state(c.x, Phi_analytic, solutions, str(C) + ' ' + str(s))
281     save_state_error(c.x, Phi_analytic, solutions, str(C) + ' ' + str(s))
282
283     NRMS = []
284     for solution in solutions:
285         err = solution[0] - Phi_analytic
286         NRMS.append(np.sqrt(np.mean(np.square(err)))/(max(Phi_analytic) - min(Phi_analytic)))
287
288     return [c.dx, c.dt, NRMS[0], NRMS[1], NRMS[2], NRMS[3]]
289
290
291 def main():
292     # Cases
293     C = [0.1, 0.5, 2, 0.5, 0.5]
294     s = [0.25, 0.25, .25, 0.5, 1]
295     for C_i, s_i in zip(C, s):
296         generate_solutions(C_i, s_i)
297
298     # Stable values for each case to find effective order of methods
299     C = [0.10, 0.50, 0.40, 0.35, 0.5]
300     s = [0.25, 0.25, 0.25, 0.40, 0.5]
301     calc_stability(C, s, 'FTCS')
302
303     C = [0.1, 0.2, 0.3, 0.05, 0.1]
304     s = [0.4, 0.3, 0.2, 0.1, 0.1]
305     calc_stability(C, s, 'Upwind')
306
307     C = [0.5, 0.6, 0.7, 0.8, 0.9]
308     s = [0.25, 0.25, 0.25, 0.25, 0.25]
309     calc_stability(C, s, 'Trapezoidal')
310
311     C = [0.25, 0.4, 0.5, 0.6, 0.7]
312     s = [0.25, 0.25, 0.25, 0.25, 0.25]
313     calc_stability(C, s, 'QUICK')
314
315
316 if __name__ == "__main__":
317     main()

```

Listing 1: Code to create solutions

```

1 import numpy as np
2 import matplotlib
3 matplotlib.use('TkAgg')
4 import matplotlib.pyplot as plt
5 import os
6 from scipy import log10
7 from scipy.optimize import curve_fit
8
9 # Configure figures for production
10 WIDTH = 495.0 # the number latex spits out
11 FACTOR = 1.0 # the fraction of the width the figure should occupy
12 fig_width_pt = WIDTH * FACTOR
13
14 inches_per_pt = 1.0 / 72.27
15 golden_ratio = (np.sqrt(5) - 1.0) / 2.0 # because it looks good
16 fig_width_in = fig_width_pt * inches_per_pt # figure width in inches
17 fig_height_in = fig_width_in * golden_ratio # figure height in inches
18 fig_dims = [fig_width_in, fig_height_in] # fig dims as a list
19
20
21 def linear_fit(x, a, b):
22     '''Define our (line) fitting function'''
23     return a + b * x

```

```

24
25
26 def effective_order(x, y):
27     '''Find slope of log log plot to find our effective order of accuracy'''
28
29     logx = log10(x)
30     logy = log10(y)
31     out = curve_fit(linear_fit, logx, logy)
32
33     return out[0][1]
34
35
36 def save_figure(x, analytic, solution, title, stable):
37     plt.figure(figsize=fig_dims)
38     plt.plot(x, analytic, label='Analytic')
39     plt.plot(x, solution, '.', label=title.split(' ')[0])
40
41     # Calculate NRMS for this solution
42     err = solution - analytic
43     NRMS = np.sqrt(np.mean(np.square(err)))/(max(analytic) - min(analytic))
44
45     plt.ylabel('$\Phi$')
46     plt.xlabel('L (m)')
47
48     if stable:
49         stability = 'Stable, '
50     else:
51         stability = 'Unstable, '
52
53     plt.title(stability + 'C=' + title.split(' ')[1] +
54              ' s=' + title.split(' ')[2] +
55              ' NRMS={0:.3e}'.format(NRMS))
56     plt.legend(loc='best')
57
58     # Save plots
59     save_name = title + '.pdf'
60     try:
61         os.mkdir('figures')
62     except Exception:
63         pass
64
65     plt.savefig('figures/' + save_name, bbox_inches='tight')
66     plt.close()
67
68
69 def save_state(x, analytic, solutions, state):
70     plt.figure(figsize=fig_dims)
71
72     plt.plot(x, analytic, 'k', label='Analytic')
73     for solution in solutions:
74         plt.plot(x, solution[0], '.', label=solution[1])
75
76     plt.ylabel('$\Phi$')
77     plt.xlabel('L (m)')
78
79     title = 'C=' + state.split(' ')[0] + ' s=' + state.split(' ')[1]
80     plt.title(title)
81     plt.legend(loc='best')
82
83     # Save plots
84     save_name = title + '.pdf'
85     try:
86         os.mkdir('figures')
87     except Exception:
88         pass
89
90     plt.savefig('figures/' + save_name, bbox_inches='tight')
91     plt.close()
92

```

```

93
94 def save_state_error(x, analytic, solutions, state):
95     plt.figure(figsize=fig_dims)
96
97     for solution in solutions:
98         Error = solution[0] - analytic
99         plt.plot(x, Error, '.', label=solution[1])
100
101     plt.ylabel('Error')
102     plt.xlabel('L (m)')
103     plt.ylim([-0.05, 0.05])
104
105     title = 'C=' + state.split(' ')[0] + ' s=' + state.split(' ')[1]
106     plt.title(title)
107     plt.legend(loc='best')
108
109     # Save plots
110     save_name = 'Error ' + title + '.pdf'
111     try:
112         os.mkdir('figures')
113     except Exception:
114         pass
115
116     plt.savefig('figures/' + save_name, bbox_inches='tight')
117     plt.close()
118
119
120 def plot_order(x, t, RMS):
121     fig = plt.figure(figsize=fig_dims)
122     RMS, title = RMS[0], RMS[1]
123
124     # Find effective order of accuracy
125     order_accuracy_x = effective_order(x, RMS)
126     order_accuracy_t = effective_order(t, RMS)
127     # print(title, 'x order: ', order_accuracy_x, 't order: ', order_accuracy_t)
128
129     # Show effect of dx on RMS
130     fig.add_subplot(2, 1, 1)
131     plt.plot(x, RMS, '.')
132     plt.title('dx vs RMS, effective order {0:1.2f}'.format(order_accuracy_x))
133     plt.xscale('log')
134     plt.yscale('log')
135     plt.xlabel('dx')
136     plt.ylabel('NRMS')
137     fig.subplots_adjust(hspace=.35)
138
139     # Show effect of dt on RMS
140     fig.add_subplot(2, 1, 2)
141     plt.plot(t, RMS, '.')
142     plt.title('dt vs RMS, effective order {0:1.2f}'.format(order_accuracy_t))
143     plt.xscale('log')
144     plt.yscale('log')
145     plt.xlabel('dt')
146     plt.ylabel('NRMS')
147
148     # Slap the method name on
149     plt.suptitle(title)
150
151     # Save plots
152     save_name = 'Order ' + title + '.pdf'
153     try:
154         os.mkdir('figures')
155     except Exception:
156         pass
157
158     plt.savefig('figures/' + save_name, bbox_inches='tight')
159     plt.close()

```

Listing 2: Code to generate pretty plots