

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,

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

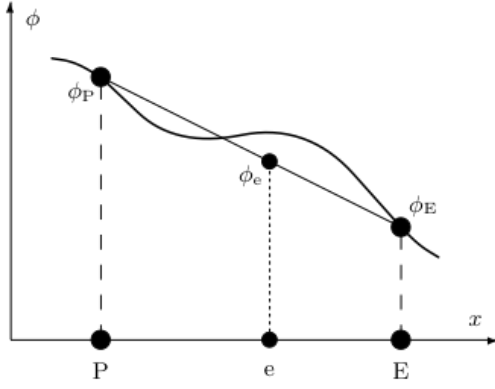


Fig. 1: The 1-D FTCS scheme interpolates between the two nearby grid points [1]

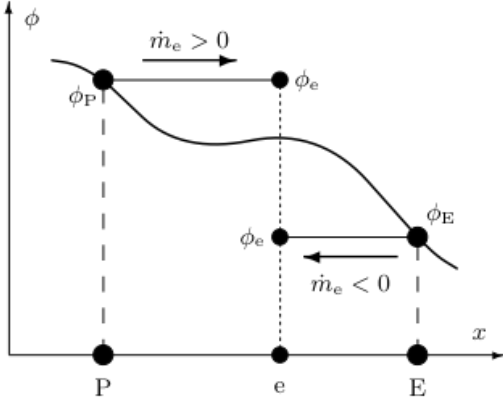


Fig. 2: Upwind scheme's interpolation for the diffusive flux [1]

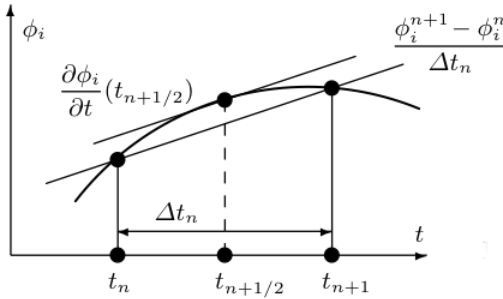


Fig. 3: Trapezoidal scheme's interpolation for the time derivative [1]

To impose the periodic boundary condition, the last node in the domain reaches around to the first 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 first two nodes in domain reach back to the other side of the domain. 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

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:

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

To impose the periodic boundary condition, the first two nodes in domain reach back to the other side of the domain. This scheme is numerically stable under the following condition:

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

3 Results Discussion

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 equally stable, and the Trapezoidal scheme is always stable. For the full results, see Table 1.

The computational result for the 1-D linear convection-diffusion equation can be compared to the analytical result above, Equation 3. The Root Mean Square error,

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

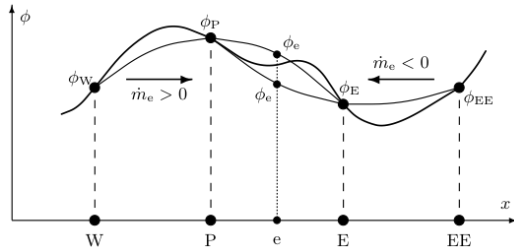


Fig. 4: The QUICK scheme interpolates between two quadratic equations [1]

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

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 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 analytical solution.

Case	FTCS	Upwind	Trap	QUICK
1	7.23E-3	2.20E-2	2.42E-2	2.45E-2
2	2.23E-1	2.34E+10	1.30E-1	2.59E-1
3	8.26E+0	1.18E+2	7.72E-1	9.40E+0
4	1.06E-1	4.22E+36	4.56E-2	8.73E+11
5	1.10E+61	8.22E+110	2.14E-2	4.70E+85

Table 2: NRMS results for each case and method

4 Conclusion

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/>.

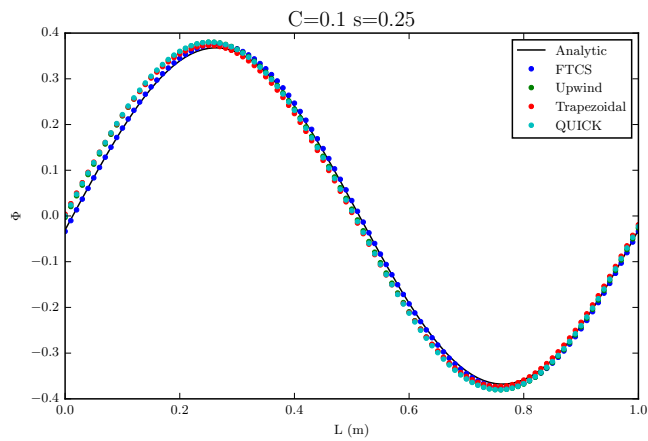


Fig. 5: Results of first case

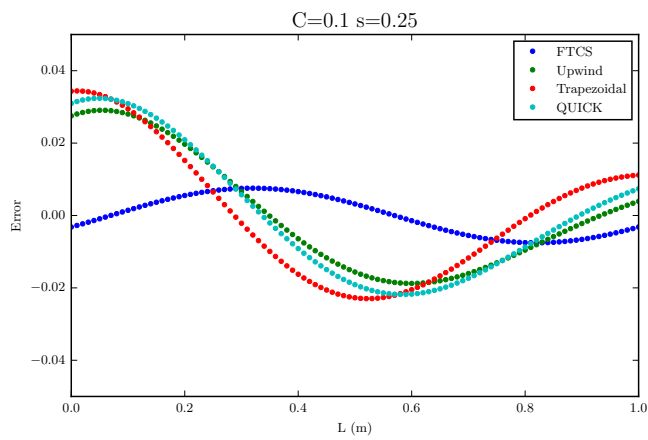
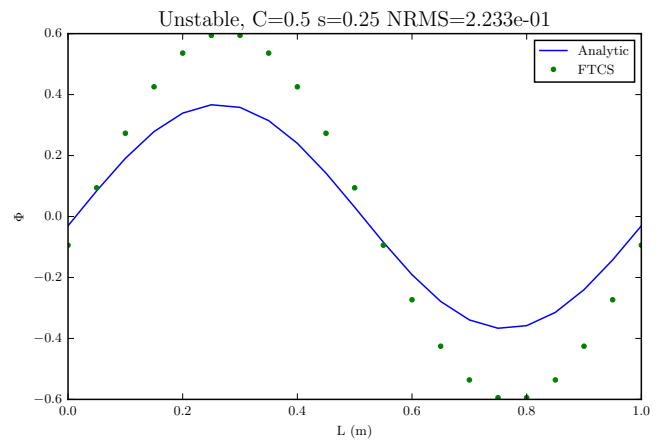
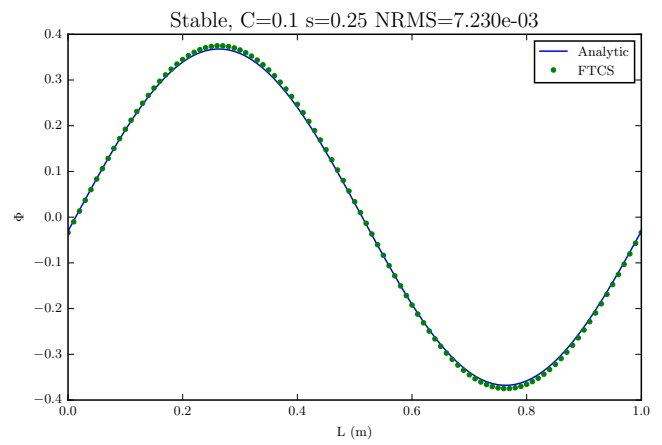


Fig. 6: Error for first case

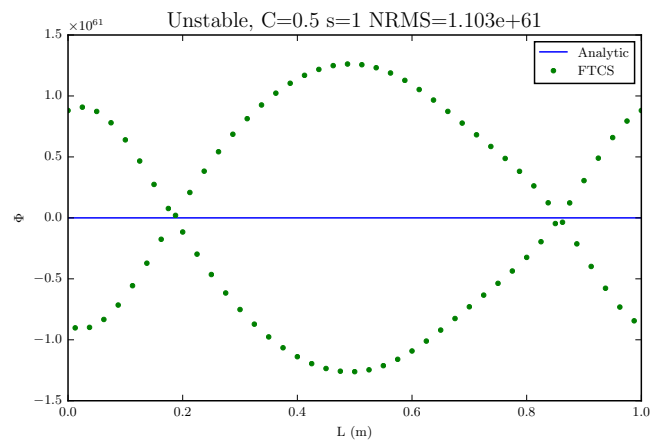


Fig. 7: FTCS method's transition into instability

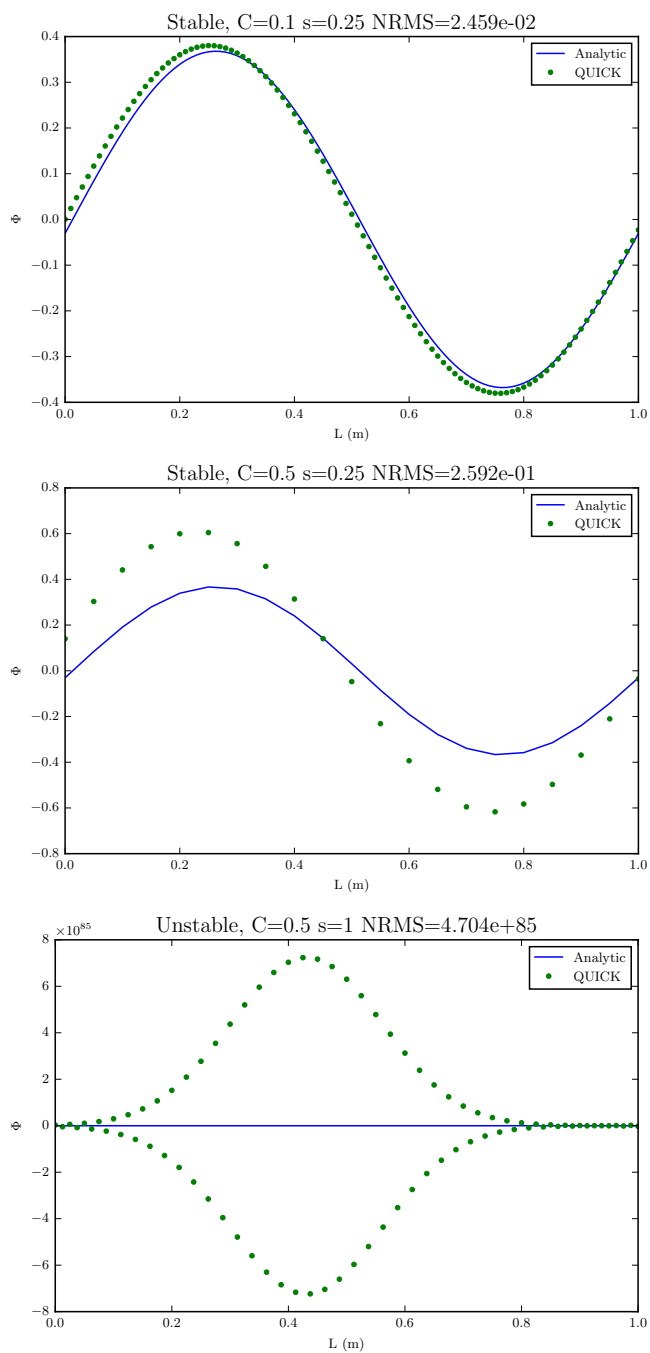


Fig. 8: QUICK method's transition into instability

Appendix A: Python Code

```
1 from PrettyPlots import *
2 import numpy as np
3 import matplotlib.pyplot as plt
4 from scipy import log10
5 from scipy.optimize import curve_fit
6 import scipy.sparse as sparse
7 import os
8
9
10 class Config(object):
11     def __init__(self, C, s):
12         # Import parameters
13         self.C = C
14         self.s = s
15
16         # Problem constants
17         self.L = 1. # m
18         self.D = 0.005 # m^2/s
19         self.u = 0.2 # m/s
20         self.k = 2 * np.pi / self.L # m^-1
21         self.tau = 1 / (self.k ** 2 * self.D)
22
23         # Set-up Mesh and Calculate time-step
24         self.dx = self.C * self.D / (self.u * self.s)
25         self.dt = self.C * self.dx / self.u
26         self.x = np.append(np.arange(0, self.L, self.dx), self.L)
27
28
29 def Analytic(c):
30     k, D, u, tau, x = c.k, c.D, c.u, c.tau, c.x
31
32     N = len(x)
33     Phi = np.array(x)
34
35     for i in range(0, N):
36         Phi[i] = np.exp(-k ** 2 * D * tau) * np.sin(k * (x[i] - u * tau))
37
38     return np.array(Phi)
39
40
41 def FTCS(Phi, c):
42     """
43     FTCS (Explicit) - Forward-Time and central differencing for both the
44     convective flux and the diffusive flux.
45     """
46
47     D, dt, dx, u, tau = c.D, c.dt, c.dx, c.u, c.tau
48
49     N = len(Phi)
50     Phi = np.array(Phi)
51     Phi_old = np.array(Phi)
52
53     t = 0
54     while t < tau:
55         for i in range(1, N - 1):
56             Phi[i] = ((1 - 2 * D * dt / dx ** 2) * Phi_old[i] +
57                     (D * dt / dx ** 2 + u * dt / (2 * dx)) * Phi_old[i - 1] +
58                     (D * dt / dx ** 2 - u * dt / (2 * dx)) * Phi_old[i + 1])
59
60         # Enforce our periodic boundary condition
61         Phi[-1] = ((1 - 2 * D * dt / dx ** 2) * Phi_old[-1] +
62                  (D * dt / dx ** 2 + u * dt / (2 * dx)) * Phi_old[-2] +
63                  (D * dt / dx ** 2 - u * dt / (2 * dx)) * Phi_old[1])
64         Phi[0] = Phi[-1]
65
66         Phi_old = np.array(Phi)
67         t += dt
```

```

68
69     return np.array(Phi_old)
70
71
72 def Upwind(Phi, c):
73     """
74     Upwind-Finite Volume method: Explicit (forward Euler), with the convective
75     flux treated using the basic upwind method and the diffusive flux treated
76     using central differencing.
77     """
78
79     D, dt, dx, u, tau = c.D, c.dt, c.dx, c.u, c.tau
80
81     N = len(Phi)
82     Phi = np.array(Phi)
83     Phi_old = np.array(Phi)
84
85     t = 0
86     while t <= tau:
87         Phi[0] = (D * dt / dx ** 2 * (Phi_old[1] - 2 * Phi_old[0] + Phi_old[-1]) -
88                 u * dt / (2 * dx) * (3 * Phi_old[0] - 4 * Phi_old[-1] + Phi_old[-2])) +
89                 Phi_old[0]
90
91         Phi[1] = (D * dt / dx ** 2 * (Phi_old[2] - 2 * Phi_old[1] + Phi_old[0]) -
92                 u * dt / (2 * dx) * (3 * Phi_old[1] - 4 * Phi_old[0] + Phi_old[-1])) +
93                 Phi_old[1]
94
95         for i in range(2, N - 1):
96             Phi[i] = (D * dt / dx ** 2 * (Phi_old[i + 1] - 2 * Phi_old[i] + Phi_old[i - 1]) -
97                     u * dt / (2 * dx) * (3 * Phi_old[i] - 4 * Phi_old[i - 1] + Phi_old[i - 2])) +
98                     Phi_old[i]
99
100        Phi[-1] = (D * dt / dx ** 2 * (Phi_old[0] - 2 * Phi_old[-1] + Phi_old[-2]) -
101                u * dt / (2 * dx) * (3 * Phi_old[-1] - 4 * Phi_old[-2] + Phi_old[-3])) +
102                Phi_old[-1]
103
104        Phi_old = np.array(Phi)
105        t += dt
106
107    return np.array(Phi_old)
108
109
110 def Trapezoidal(Phi, c):
111     D, dt, dx, u, tau = c.D, c.dt, c.dx, c.u, c.tau
112
113     N = len(Phi)
114     Phi = np.array(Phi)
115     Phi_old = np.array(Phi)
116
117     # Create Coefficient Matrix
118     upper = [-(dt * D) / (2 * dx ** 2) + dt * u / (4 * dx) for _ in range(0, N)]
119     main = [1 + (dt * D) / (dx ** 2) for _ in range(0, N)]
120     lower = [-(dt * D) / (2 * dx ** 2) - dt * u / (4 * dx) for _ in range(0, N)]
121
122     data = lower, main, upper
123     diags = np.array([-1, 0, 1])
124     matrix = sparse.spdiags(data, diags, N, N).todense()
125
126     # Set values for cyclic boundary conditions
127     matrix[0, N - 1] = -(dt * D) / (2 * dx ** 2) - dt * u / (4 * dx)
128     matrix[N - 1, 0] = -(dt * D) / (2 * dx ** 2) + dt * u / (4 * dx)
129
130     # create blank b array
131     b = np.array(Phi_old)
132
133     t = 0
134     while t <= tau:
135         # Enforce our periodic boundary condition
136         b[0] = ((dt * D) / (2 * dx ** 2)) * (Phi_old[1] - 2 * Phi_old[0] + Phi_old[-1]) -

```

```

137         (u * dt / (4 * dx)) * (Phi_old[1] - Phi_old[-1]) +
138         Phi_old[0])
139
140     for i in range(1, N - 1):
141         b[i] = ((dt * D / (2 * dx ** 2)) * (Phi_old[i + 1] - 2 * Phi_old[i] + Phi_old[i - 1]) -
142                (u * dt / (4 * dx)) * (Phi_old[i + 1] - Phi_old[i - 1]) +
143                Phi_old[i])
144
145     # Enforce our periodic boundary condition
146     b[-1] = ((dt * D / (2 * dx ** 2)) * (Phi_old[0] - 2 * Phi_old[-1] + Phi_old[-2]) -
147              (u * dt / (4 * dx)) * (Phi_old[0] - Phi_old[-2]) +
148              Phi_old[-1])
149
150     # Solve matrix
151     Phi = np.linalg.solve(matrix, b)
152
153     Phi_old = np.array(Phi)
154     t += dt
155
156     return np.array(Phi_old)
157
158 def QUICK(Phi, c):
159     D, dt, dx, u, tau = c.D, c.dt, c.dx, c.u, c.tau
160
161     N = len(Phi)
162     Phi = np.array(Phi)
163     Phi_old = np.array(Phi)
164
165     t = 0
166     while t <= tau:
167         Phi[0] = (dt * D / dx ** 2 * (Phi_old[1] - 2 * Phi_old[0] + Phi_old[N - 1]) -
168                  dt * u / (8 * dx) * (3 * Phi_old[1] + Phi_old[-2] - 7 * Phi_old[N - 1] + 3 * Phi_old[0]) +
169                  Phi_old[0])
170         Phi[1] = (dt * D / dx ** 2 * (Phi_old[2] - 2 * Phi_old[1] + Phi_old[0]) -
171                  dt * u / (8 * dx) * (3 * Phi_old[2] + Phi_old[N - 1] - 7 * Phi_old[0] + 3 * Phi_old[1]) +
172                  Phi_old[1])
173
174         for i in range(2, N - 1):
175             Phi[i] = (dt * D / dx ** 2 * (Phi_old[i + 1] - 2 * Phi_old[i] + Phi_old[i - 1]) -
176                      dt * u / (8 * dx) * (3 * Phi_old[i + 1] + Phi_old[i - 2] - 7 * Phi_old[i - 1] + 3 * Phi_old[i]) +
177                      Phi_old[i])
178
179         Phi[-1] = (dt * D / dx ** 2 * (Phi_old[0] - 2 * Phi_old[-1] + Phi_old[-2]) -
180                   dt * u / (8 * dx) * (3 * Phi_old[0] + Phi_old[-3] - 7 * Phi_old[-2] + 3 * Phi_old[-1]) +
181                   Phi_old[-1])
182
183         # Increment
184         Phi_old = np.array(Phi)
185         t += dt
186
187     return np.array(Phi_old)
188
189
190 def save_figure(x, analytic, solution, title, stable):
191     plt.figure(figsize=fig_dims)
192
193     plt.plot(x, analytic, label='Analytic')
194     plt.plot(x, solution, '.', label=title.split(' ')[0])
195
196     # Calculate NRMS for this solution
197     err = solution - analytic
198     NRMS = np.sqrt(np.mean(np.square(err))) / (max(analytic) - min(analytic))
199
200     plt.ylabel('$\Phi$')
201     plt.xlabel('L (m)')
202
203     if stable:
204         stability = 'Stable, '
205

```



```

206     else:
207         stability = 'Unstable, '
208
209     plt.title(stability +
210              'C=' + title.split(' ')[1] +
211              ' s=' + title.split(' ')[2] +
212              ' NRMS={0:.3e}'.format(NRMS))
213     plt.legend(loc='best')
214
215     # Save plots
216     save_name = title + '.pdf'
217     try:
218         os.mkdir('figures')
219     except Exception:
220         pass
221
222     plt.savefig('figures/' + save_name, bbox_inches='tight')
223     plt.close()
224
225
226 def save_state(x, analytic, solutions, state):
227     plt.figure(figsize=fig_dims)
228
229     plt.plot(x, analytic, 'k', label='Analytic')
230     for solution in solutions:
231         plt.plot(x, solution[0], '.', label=solution[1])
232
233     plt.ylabel('$\Phi$')
234     plt.xlabel('L (m)')
235
236     title = 'C=' + state.split(' ')[0] + ' s=' + state.split(' ')[1]
237     plt.title(title)
238     plt.legend(loc='best')
239
240     # Save plots
241     save_name = title + '.pdf'
242     try:
243         os.mkdir('figures')
244     except Exception:
245         pass
246
247     plt.savefig('figures/' + save_name, bbox_inches='tight')
248     plt.close()
249
250
251 def save_state_error(x, analytic, solutions, state):
252     plt.figure(figsize=fig_dims)
253
254     for solution in solutions:
255         Error = solution[0] - analytic
256         plt.plot(x, Error, '.', label=solution[1])
257
258     plt.ylabel('Error')
259     plt.xlabel('L (m)')
260     plt.ylim([-0.05, 0.05])
261
262     title = 'C=' + state.split(' ')[0] + ' s=' + state.split(' ')[1]
263     plt.title(title)
264     plt.legend(loc='best')
265
266     # Save plots
267     save_name = 'Error ' + title + '.pdf'
268     try:
269         os.mkdir('figures')
270     except Exception:
271         pass
272
273     plt.savefig('figures/' + save_name, bbox_inches='tight')
274     plt.close()

```

```

275
276
277 def plot_order(x, t, RMS):
278     fig = plt.figure(figsize=fig_dims)
279
280     RMS, title = RMS[0], RMS[1]
281
282     # Find effective order of accuracy
283     order_accuracy_x = effective_order(x, RMS)
284     order_accuracy_t = effective_order(t, RMS)
285     # print(title, 'x order: ', order_accuracy_x, 't order: ', order_accuracy_t)
286
287     # Show effect of dx on RMS
288     fig.add_subplot(2, 1, 1)
289     plt.plot(x, RMS, '.')
290     plt.title('dx vs RMS, effective order {0:1.2f}'.format(order_accuracy_x))
291     plt.xscale('log')
292     plt.yscale('log')
293     plt.xlabel('dx')
294     plt.ylabel('NRMS')
295     fig.subplots_adjust(hspace=.35)
296
297     # Show effect of dt on RMS
298     fig.add_subplot(2, 1, 2)
299     plt.plot(t, RMS, '.')
300     plt.title('dt vs RMS, effective order {0:1.2f}'.format(order_accuracy_t))
301     plt.xscale('log')
302     plt.yscale('log')
303     plt.xlabel('dt')
304     plt.ylabel('NRMS')
305
306     # Slap the method name on
307     plt.suptitle(title)
308
309     # Save plots
310     save_name = 'Order ' + title + '.pdf'
311     try:
312         os.mkdir('figures')
313     except Exception:
314         pass
315
316     plt.savefig('figures/' + save_name, bbox_inches='tight')
317     plt.close()
318
319
320 def stability(c):
321     C, s, u = c.C, c.s, c.u
322
323     FTCS = C <= np.sqrt(2 * s * u) and s <= 0.5
324     Upwind = C + 2*s <= 1
325     Trapezoidal = True
326     QUICK = C <= min(2 - 4 * s, np.sqrt(2 * s))
327
328     # print('C = ', C, ' s = ', s)
329     # print('FTCS: ' + str(FTCS))
330     # print('Upwind: ' + str(Upwind))
331     # print('Trapezoidal: ' + str(Trapezoidal))
332     # print('QUICK: ' + str(QUICK))
333
334     return [FTCS, Upwind, Trapezoidal, QUICK]
335
336
337 def linear_fit(x, a, b):
338     '''Define our (line) fitting function'''
339     return a + b * x
340
341
342 def effective_order(x, y):
343     '''Find slope of log log plot to find our effective order of accuracy'''

```

```

344
345     logx = log10(x)
346     logy = log10(y)
347     out = curve_fit(linear_fit, logx, logy)
348
349     return out[0][1]
350
351
352 def calc_stability(C, s, solver):
353     results = []
354     for C_i, s_i in zip(C, s):
355         out = generate_solutions(C_i, s_i, find_order=True)
356         results.append(out)
357
358     # Sort and convert
359     results.sort(key=lambda x: x[0])
360     results = np.array(results)
361
362     # Pull out data
363     x = results[:, 0]
364     t = results[:, 1]
365     RMS_FTCS = results[:, 2]
366     RMS_Upwind = results[:, 3]
367     RMS_Trapezoidal = results[:, 4]
368     RMS_QUICK = results[:, 5]
369
370     # Plot effective orders
371     rms_list = [(RMS_FTCS, 'FTCS'),
372                 (RMS_Upwind, 'Upwind'),
373                 (RMS_Trapezoidal, 'Trapezoidal'),
374                 (RMS_QUICK, 'QUICK')]
375
376     for rms in rms_list:
377         if rms[1] == solver:
378             plot_order(x, t, rms)
379
380
381 def generate_solutions(C, s, find_order=False):
382     c = Config(C, s)
383
384     # Spit out some stability information
385     stable = stability(c)
386
387     # Initial Condition with boundary conditions
388     Phi_initial = np.sin(c.k * c.x)
389
390     # Analytic Solution
391     Phi_analytic = Analytic(c)
392
393     # Explicit Solution
394     Phi_ftcs = FTCS(Phi_initial, c)
395
396     # Upwind Solution
397     Phi_upwind = Upwind(Phi_initial, c)
398
399     # Trapezoidal Solution
400     Phi_trapezoidal = Trapezoidal(Phi_initial, c)
401
402     # QUICK Solution
403     Phi_quick = QUICK(Phi_initial, c)
404
405     # Save group comparison
406     solutions = [(Phi_ftcs, 'FTCS'),
407                  (Phi_upwind, 'Upwind'),
408                  (Phi_trapezoidal, 'Trapezoidal'),
409                  (Phi_quick, 'QUICK')]
410
411     if not find_order:
412         # Save individual comparisons

```

```

413     save_figure(c.x, Phi_analytic, Phi_ftcs,
414                 'FTCS ' + str(C) + ' ' + str(s), stable[0])
415     save_figure(c.x, Phi_analytic, Phi_upwind,
416                 'Upwind ' + str(C) + ' ' + str(s), stable[1])
417     save_figure(c.x, Phi_analytic, Phi_trapezoidal,
418                 'Trapezoidal ' + str(C) + ' ' + str(s), stable[2])
419     save_figure(c.x, Phi_analytic, Phi_quick,
420                 'QUICK ' + str(C) + ' ' + str(s), stable[3])
421
422     # and group comparisons
423     save_state(c.x, Phi_analytic, solutions, str(C) + ' ' + str(s))
424     save_state_error(c.x, Phi_analytic, solutions, str(C) + ' ' + str(s))
425
426     NRMS = []
427     for solution in solutions:
428         err = solution[0] - Phi_analytic
429         NRMS.append(np.sqrt(np.mean(np.square(err)))/(max(Phi_analytic) - min(Phi_analytic)))
430
431     return [c.dx, c.dt, NRMS[0], NRMS[1], NRMS[2], NRMS[3]]
432
433
434 def main():
435     # Cases
436     C = [0.1, 0.5, 2, 0.5, 0.5]
437     s = [0.25, 0.25, .25, 0.5, 1]
438     for C_i, s_i in zip(C, s):
439         generate_solutions(C_i, s_i)
440
441     # Stable values for each case to find effective order of methods
442     C = [0.10, 0.50, 0.40, 0.35, 0.5]
443     s = [0.25, 0.25, 0.25, 0.40, 0.5]
444     calc_stability(C, s, 'FTCS')
445
446     C = [0.1, 0.2, 0.3, 0.05, 0.1]
447     s = [0.4, 0.3, 0.2, 0.15, 0.1]
448     calc_stability(C, s, 'Upwind')
449
450     C = [0.5, 0.6, 0.7, 0.8, 0.9]
451     s = [0.25, 0.25, 0.25, 0.25, 0.25]
452     calc_stability(C, s, 'Trapezoidal')
453
454     C = [0.25, 0.4, 0.5, 0.6, 0.7]
455     s = [0.25, 0.25, 0.25, 0.25, 0.25]
456     calc_stability(C, s, 'QUICK')
457
458
459 if __name__ == "__main__":
460     main()

```

Listing 1: Code to create plots and solutions

```

1 import numpy as np
2 import matplotlib
3 matplotlib.use('TkAgg')
4
5 # Configure figures for production
6 WIDTH = 495.0 # the number latex spits out
7 FACTOR = 1.0 # the fraction of the width the figure should occupy
8 fig_width_pt = WIDTH * FACTOR
9
10 inches_per_pt = 1.0 / 72.27
11 golden_ratio = (np.sqrt(5) - 1.0) / 2.0 # because it looks good
12 fig_width_in = fig_width_pt * inches_per_pt # figure width in inches
13 fig_height_in = fig_width_in * golden_ratio # figure height in inches
14 fig_dims = [fig_width_in, fig_height_in] # fig dims as a list

```

Listing 2: Code to generate pretty plots