

# 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),

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

$\phi$  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  $D = 0.005$  m<sup>2</sup>/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

## 3 Results Discussion

## 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.

## 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.plot(x, analytic, label='Analytic')
192     plt.plot(x, solution, '.', label=title.split(' ')[0])
193
194     # Calculate NRMS for this solution
195     err = solution - analytic
196     NRMS = np.sqrt(np.mean(np.square(err))) / (max(analytic) - min(analytic))
197
198     plt.ylabel('$\Phi$')
199     plt.xlabel('L (m)')
200
201     if stable:
202         stability = 'Stable, '
203     else:
204         stability = 'Unstable, '
205

```

```

206
207 plt.title(stability +
208           'C=' + title.split(' ')[1] +
209           ' s=' + title.split(' ')[2] +
210           ' NRMS={0:.3e}'.format(NRMS))
211 plt.legend(loc='best')
212
213 # Save plots
214 save_name = title + '.pdf'
215 try:
216     os.mkdir('figures')
217 except Exception:
218     pass
219
220 plt.savefig('figures/' + save_name, bbox_inches='tight')
221 plt.clf()
222
223
224 def save_state(x, analytic, solutions, state):
225     plt.plot(x, analytic, 'k', label='Analytic')
226     for solution in solutions:
227         plt.plot(x, solution[0], '.', label=solution[1])
228
229     plt.ylabel('$\Phi$')
230     plt.xlabel('L (m)')
231
232     title = 'C=' + state.split(' ')[0] + ' s=' + state.split(' ')[1]
233     plt.title(title)
234     plt.legend(loc='best')
235
236     # Save plots
237     save_name = title + '.pdf'
238     try:
239         os.mkdir('figures')
240     except Exception:
241         pass
242
243     plt.savefig('figures/' + save_name, bbox_inches='tight')
244     plt.clf()
245
246
247 def save_state_error(x, analytic, solutions, state):
248     for solution in solutions:
249         Error = solution[0] - analytic
250         plt.plot(x, Error, '.', label=solution[1])
251
252     plt.ylabel('Error')
253     plt.xlabel('L (m)')
254     plt.ylim([-0.05, 0.05])
255
256     title = 'C=' + state.split(' ')[0] + ' s=' + state.split(' ')[1]
257     plt.title(title)
258     plt.legend(loc='best')
259
260     # Save plots
261     save_name = 'Error ' + title + '.pdf'
262     try:
263         os.mkdir('figures')
264     except Exception:
265         pass
266
267     plt.savefig('figures/' + save_name, bbox_inches='tight')
268     plt.clf()
269
270
271 def plot_order(x, t, RMS):
272     fig = plt.figure()
273     RMS, title = RMS[0], RMS[1]
274

```

```

275 # Find effective order of accuracy
276 order_accuracy_x = effective_order(x, RMS)
277 order_accuracy_t = effective_order(t, RMS)
278 # print(title, 'x order: ', order_accuracy_x, 't order: ', order_accuracy_t)
279
280 # Show effect of dx on RMS
281 fig.add_subplot(2, 1, 1)
282 plt.plot(x, RMS, '.')
283 plt.title('dx vs RMS, effective order {0:1.2f}'.format(order_accuracy_x))
284 plt.xscale('log')
285 plt.yscale('log')
286 plt.xlabel('dx')
287 plt.ylabel('NRMS')
288 fig.subplots_adjust(hspace=.35)
289
290 # Show effect of dt on RMS
291 fig.add_subplot(2, 1, 2)
292 plt.plot(t, RMS, '.')
293 plt.title('dt vs RMS, effective order {0:1.2f}'.format(order_accuracy_t))
294 plt.xscale('log')
295 plt.yscale('log')
296 plt.xlabel('dt')
297 plt.ylabel('NRMS')
298
299 # Slap the method name on
300 plt.suptitle(title)
301
302 # Save plots
303 save_name = 'Order ' + title + '.pdf'
304 try:
305     os.mkdir('figures')
306 except Exception:
307     pass
308
309 plt.savefig('figures/' + save_name, bbox_inches='tight')
310 plt.clf()
311
312
313 def stability(c):
314     C, s, D, u, dx, dt = c.C, c.s, c.D, c.u, c.dx, c.dt
315
316     FTCS = dx < (2 * D) / u and dt < dx ** 2 / (2 * D)
317     FTCS = C <= np.sqrt(2 * s * u) and s <= 0.5
318     Upwind = C + 2*s < 1
319     Trapezoidal = True
320     QUICK = C < min(2-4*s, np.sqrt(2*s))
321
322     # print('C = ', C, ' s = ', s)
323     # print('FTCS: ' + str(FTCS))
324     # print('Upwind: ' + str(Upwind))
325     # print('Trapezoidal: ' + str(Trapezoidal))
326     # print('QUICK: ' + str(QUICK))
327
328     return [FTCS, Upwind, Trapezoidal, QUICK]
329
330
331 def linear_fit(x, a, b):
332     '''Define our (line) fitting function'''
333     return a + b * x
334
335
336 def effective_order(x, y):
337     '''Find slope of log log plot to find our effective order of accuracy'''
338
339     logx = log10(x)
340     logy = log10(y)
341     out = curve_fit(linear_fit, logx, logy)
342
343     return out[0][1]

```

```

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

```

```

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