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),

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

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

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

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^2/s.

This problem has an analytical solution [1],

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

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.
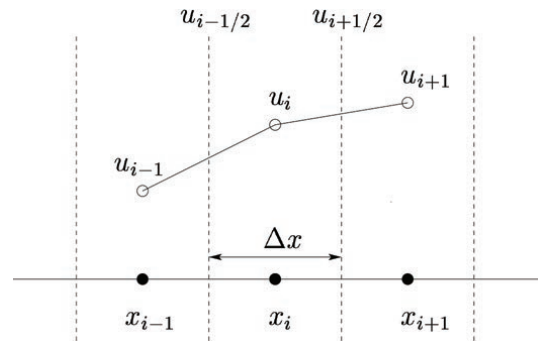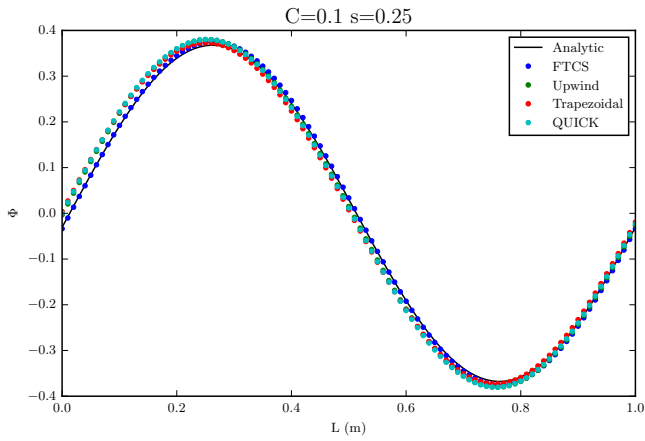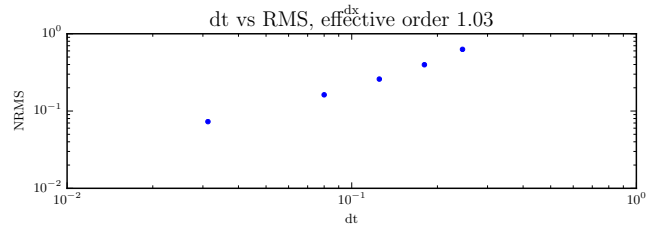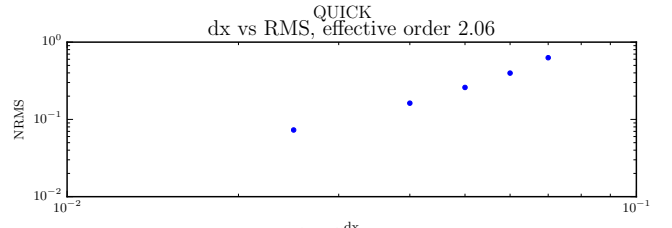
Fig. 1: FTCS scheme

Fig. 2: Upwind scheme

Fig. 5: Error for first case

Fig. 3: QUICK scheme

Fig. 4: Results of first case
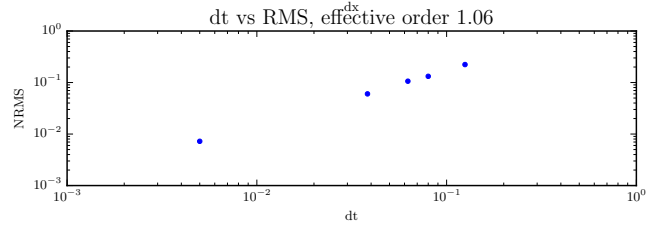
Fig. 6: Effective order of accuracy
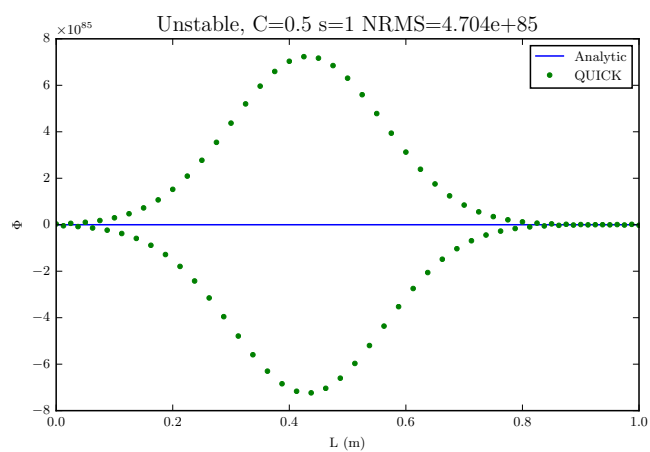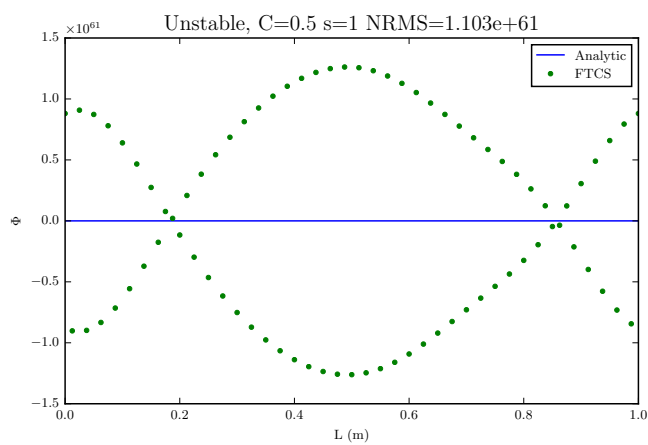
Fig. 7: FTCS method's transition into instability

Fig. 8: QUICK method's transition into instability

Appendix A: Python Code

```python
from PrettyPlots import *
import numpy as np
import matplotlib.pyplot as plt
from scipy import log10
from scipy.optimize import curve_fit
import scipy.sparse as sparse
import os


class Config(object):
    def __init__(self, C, s):
        # Import parameters
        self.C = C
        self.s = s

        # Problem constants
        self.L = 1.                     # m
        self.D = 0.005                  # m^2/s
        self.u = 0.2                    # m/s
        self.k = 2 * np.pi / self.L     # m^-1
        self.tau = 1 / (self.k ** 2 * self.D)

        # Set-up Mesh and Calculate time-step
        self.dx = self.C * self.D / (self.u * self.s)
        self.dt = self.C * self.dx / self.u
        self.x = np.append(np.arange(0, self.L, self.dx), self.L)


def Analytic(c):
    k, D, u, tau, x = c.k, c.D, c.u, c.tau, c.x

    N = len(x)
    Phi = np.array(x)

    for i in range(0, N):
        Phi[i] = np.exp(-k ** 2 * D * tau) * np.sin(k * (x[i] - u * tau))

    return np.array(Phi)


def FTCS(Phi, c):
    """
    FTCS (Explicit) - Forward-Time and central differencing for both the
    convective flux and the diffusive flux.
    """

    D, dt, dx, u, tau = c.D, c.dt, c.dx, c.u, c.tau

    N = len(Phi)
    Phi = np.array(Phi)
    Phi_old = np.array(Phi)

    t = 0
    while t < tau:
        for i in range(1, N - 1):
            Phi[i] = ((1 - 2 * D * dt / dx ** 2) * Phi_old[i] +
                      (D * dt / dx ** 2 + u * dt / (2 * dx)) * Phi_old[i - 1] +
                      (D * dt / dx ** 2 - u * dt / (2 * dx)) * Phi_old[i + 1])

        # Enforce our periodic boundary condition
        Phi[-1] = ((1 - 2 * D * dt / dx ** 2) * Phi_old[-1] +
                   (D * dt / dx ** 2 + u * dt / (2 * dx)) * Phi_old[-2] +
                   (D * dt / dx ** 2 - u * dt / (2 * dx)) * Phi_old[1])
        Phi[0] = Phi[-1]

        Phi_old = np.array(Phi)
        t += dt
```
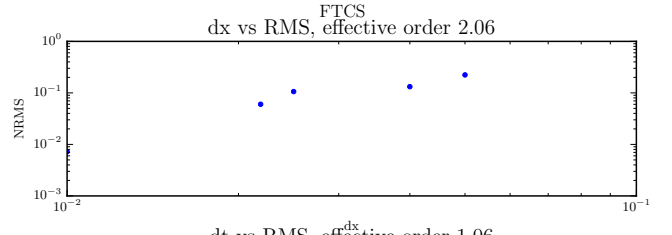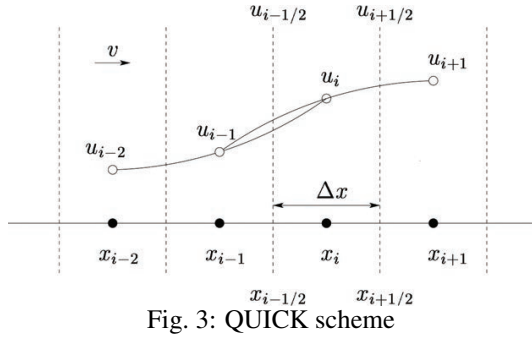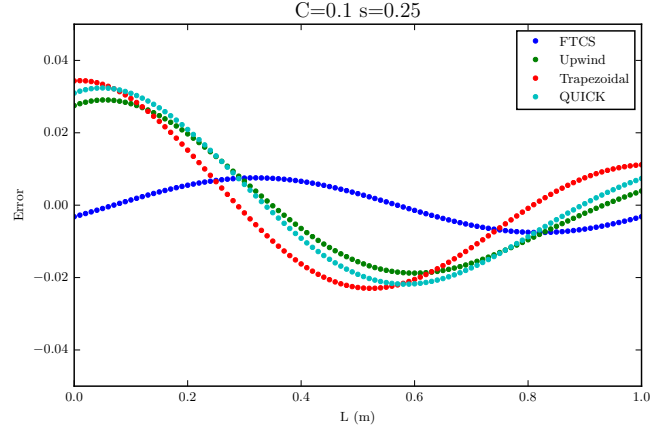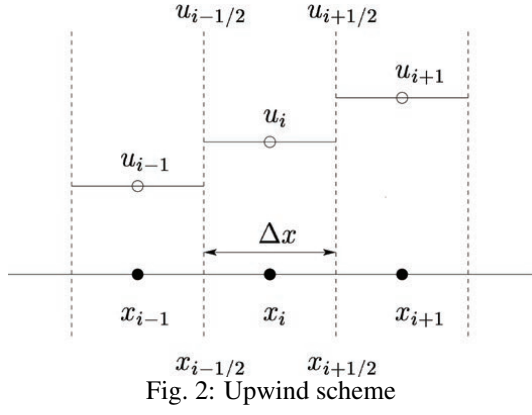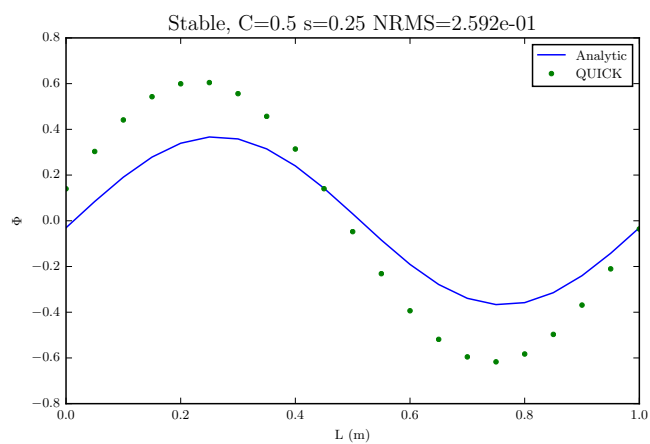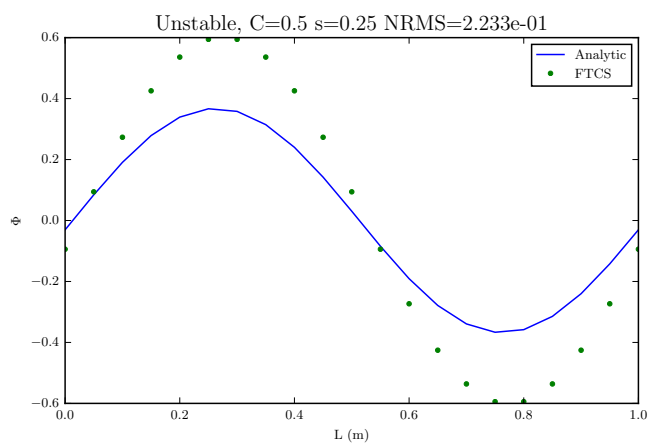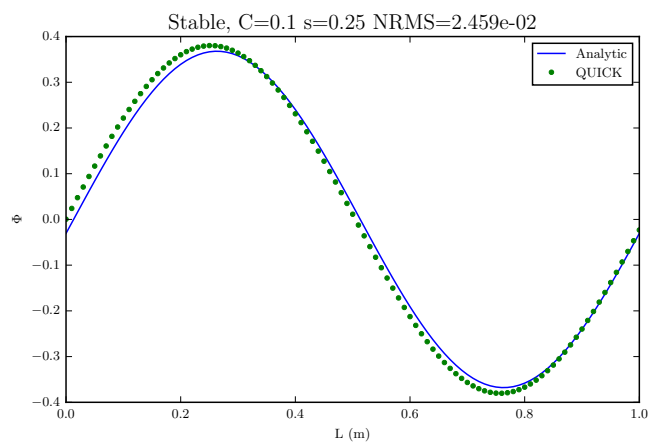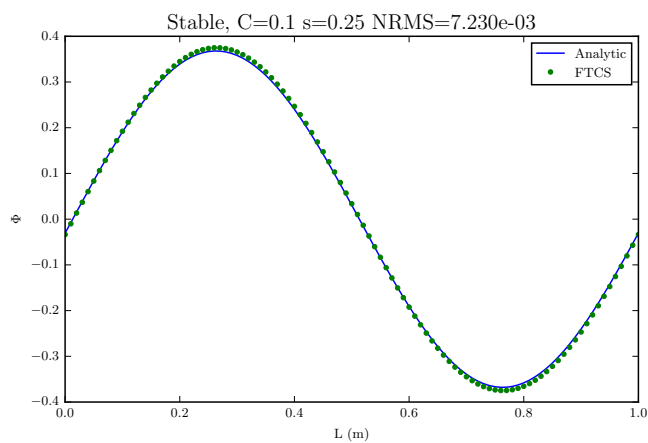
```python
68
69       return np.array(Phi_old)
70
71
72  def Upwind(Phi, c):
73      '''
74      Upwind-Finite Volume method: Explicit (forward Euler), with the convective
75      flux treated using the basic upwind method and the diffusive flux treated
76      using central differencing.
77      '''
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]) -
```

```python
                        (u * dt / (4 * dx)) * (Phi_old[1] - Phi_old[-1]) +
                        Phi_old[0])

            for i in range(1, N - 1):
                b[i] = ((dt * D / (2 * dx ** 2)) * (Phi_old[i + 1] - 2 * Phi_old[i] + Phi_old[i - 1]) -
                        (u * dt / (4 * dx)) * (Phi_old[i + 1] - Phi_old[i - 1]) +
                        Phi_old[i])

            # Enforce our periodic boundary condition
            b[-1] = ((dt * D / (2 * dx ** 2)) * (Phi_old[0] - 2 * Phi_old[-1] + Phi_old[-2]) -
                        (u * dt / (4 * dx)) * (Phi_old[0] - Phi_old[-2]) +
                        Phi_old[-1])

            # Solve matrix
            Phi = np.linalg.solve(matrix, b)

            Phi_old = np.array(Phi)
            t += dt

        return np.array(Phi_old)


def QUICK(Phi, c):
    D, dt, dx, u, tau = c.D, c.dt, c.dx, c.u, c.tau

    N = len(Phi)
    Phi = np.array(Phi)
    Phi_old = np.array(Phi)

    t = 0
    while t <= tau:
        Phi[0] = (dt * D / dx ** 2 * (Phi_old[1] - 2 * Phi_old[0] + Phi_old[N - 1]) -
                    dt * u / (8 * dx) * (3 * Phi_old[1] + Phi_old[-2] - 7 * Phi_old[N - 1] + 3 * Phi_old[0]) +
                    Phi_old[0])
        Phi[1] = (dt * D / dx ** 2 * (Phi_old[2] - 2 * Phi_old[1] + Phi_old[0]) -
                    dt * u / (8 * dx) * (3 * Phi_old[2] + Phi_old[N - 1] - 7 * Phi_old[0] + 3 * Phi_old[1]) +
                    Phi_old[1])

        for i in range(2, N - 1):
            Phi[i] = (dt * D / dx ** 2 * (Phi_old[i + 1] - 2 * Phi_old[i] + Phi_old[i - 1]) -
                        dt * u / (8 * dx) * (3 * Phi_old[i + 1] + Phi_old[i - 2] - 7 * Phi_old[i - 1] + 3 * Phi_old[i]) +
                        Phi_old[i])

        Phi[-1] = (dt * D / dx ** 2 * (Phi_old[0] - 2 * Phi_old[-1] + Phi_old[-2]) -
                    dt * u / (8 * dx) * (3 * Phi_old[0] + Phi_old[-3] - 7 * Phi_old[-2] + 3 * Phi_old[-1]) +
                    Phi_old[-1])

        # Increment
        Phi_old = np.array(Phi)
        t += dt

    return np.array(Phi_old)


def save_figure(x, analytic, solution, title, stable):
    plt.figure(figsize=fig_dims)

    plt.plot(x, analytic, label='Analytic')
    plt.plot(x, solution, '.', label=title.split(' ')[0])

    # Calculate NRMS for this solution
    err = solution - analytic
    NRMS = np.sqrt(np.mean(np.square(err)))/(max(analytic) - min(analytic))

    plt.ylabel('$\Phi$')
    plt.xlabel('L (m)')

    if stable:
        stability = 'Stable, '
```

```python
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()
```

```python
def plot_order(x, t, RMS):
    fig = plt.figure(figsize=fig_dims)

    RMS, title = RMS[0], RMS[1]

    # Find effective order of accuracy
    order_accuracy_x = effective_order(x, RMS)
    order_accuracy_t = effective_order(t, RMS)
    # print(title, 'x order: ', order_accuracy_x, 't order: ', order_accuracy_t)

    # Show effect of dx on RMS
    fig.add_subplot(2, 1, 1)
    plt.plot(x, RMS, '.')
    plt.title('dx vs RMS, effective order {0:1.2f}'.format(order_accuracy_x))
    plt.xscale('log')
    plt.yscale('log')
    plt.xlabel('dx')
    plt.ylabel('NRMS')
    fig.subplots_adjust(hspace=.35)

    # Show effect of dt on RMS
    fig.add_subplot(2, 1, 2)
    plt.plot(t, RMS, '.')
    plt.title('dt vs RMS, effective order {0:1.2f}'.format(order_accuracy_t))
    plt.xscale('log')
    plt.yscale('log')
    plt.xlabel('dt')
    plt.ylabel('NRMS')

    # Slap the method name on
    plt.suptitle(title)

    # Save plots
    save_name = 'Order ' + title + '.pdf'
    try:
        os.mkdir('figures')
    except Exception:
        pass

    plt.savefig('figures/' + save_name, bbox_inches='tight')
    plt.close()


def stability(c):
    C, s, u = c.C, c.s, c.u

    FTCS = C <= np.sqrt(2 * s * u) and s <= 0.5
    Upwind = C + 2*s < 1
    Trapezoidal = True
    QUICK = C < min(2-4*s, np.sqrt(2*s))

    # print('C = ', C, ' s = ', s)
    # print('FTCS: ' + str(FTCS))
    # print('Upwind: ' + str(Upwind))
    # print('Trapezoidal: ' + str(Trapezoidal))
    # print('QUICK: ' + str(QUICK))

    return [FTCS, Upwind, Trapezoidal, QUICK]


def linear_fit(x, a, b):
    '''Define our (line) fitting function'''
    return a + b * x


def effective_order(x, y):
    '''Find slope of log log plot to find our effective order of accuracy'''
```

```python
        logx = log10(x)
        logy = log10(y)
        out = curve_fit(linear_fit, logx, logy)

        return out[0][1]


def calc_stability(C, s, solver):
        results = []
        for C_i, s_i in zip(C, s):
                out = generate_solutions(C_i, s_i, find_order=True)
                results.append(out)

        # Sort and convert
        results.sort(key=lambda x: x[0])
        results = np.array(results)

        # Pull out data
        x = results[:, 0]
        t = results[:, 1]
        RMS_FTCS = results[:, 2]
        RMS_Upwind = results[:, 3]
        RMS_Trapezoidal = results[:, 4]
        RMS_QUICK = results[:, 5]

        # Plot effective orders
        rms_list = [(RMS_FTCS, 'FTCS'),
                        (RMS_Upwind, 'Upwind'),
                        (RMS_Trapezoidal, 'Trapezoidal'),
                        (RMS_QUICK, 'QUICK')]

        for rms in rms_list:
                if rms[1] == solver:
                        plot_order(x, t, rms)


def generate_solutions(C, s, find_order=False):
        c = Config(C, s)

        # Spit out some stability information
        stable = stability(c)

        # Initial Condition with boundary conditions
        Phi_initial = np.sin(c.k * c.x)

        # Analytic Solution
        Phi_analytic = Analytic(c)

        # Explicit Solution
        Phi_ftcs = FTCS(Phi_initial, c)

        # Upwind Solution
        Phi_upwind = Upwind(Phi_initial, c)

        # Trapezoidal Solution
        Phi_trapezoidal = Trapezoidal(Phi_initial, c)

        # QUICK Solution
        Phi_quick = QUICK(Phi_initial, c)

        # Save group comparison
        solutions = [(Phi_ftcs, 'FTCS'),
                        (Phi_upwind, 'Upwind'),
                        (Phi_trapezoidal, 'Trapezoidal'),
                        (Phi_quick, 'QUICK')]

        if not find_order:
                # 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