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),

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

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 $\tau = 1/k^2D$) subject to periodic boundary conditions and the following initial condition

$$\phi(x,0) = sin(kx), \tag{2}$$

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],

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

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 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,

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

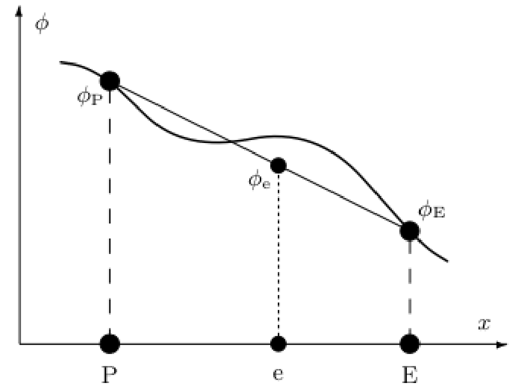

Fig. 1: The 1-D FTCS scheme interpolates between the two nearby grid points [1]

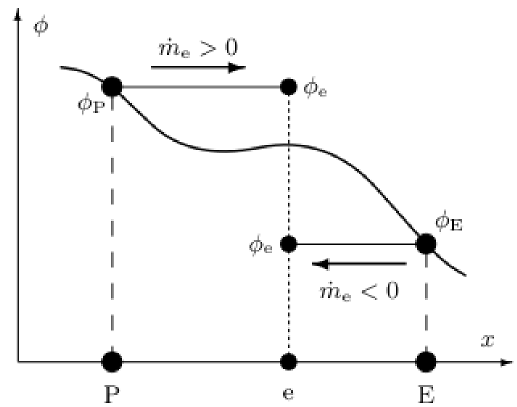

Fig. 2: Upwind scheme's interpolation for the diffusive flux [1]

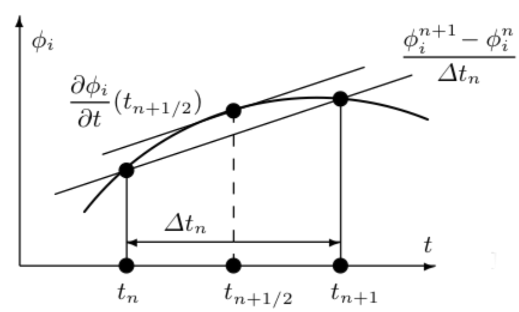

Fig. 3: Trapezoidal scheme's interpolation for the time derivative [1]

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:

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

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

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

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:

$$C + 2s \leq 1. \tag{7}$$

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:

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

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

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

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

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

2.4 QUICK Scheme

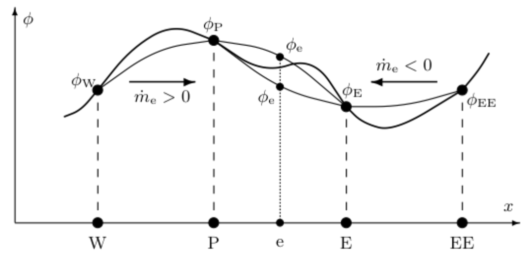

Fig. 4: The QUICK scheme interpolates between two quadratic equations [1]

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:

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

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:

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

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 above, Equation 3. The Root Mean Square error,

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

and the Normalized Root Mean Square error,

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

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.

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-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

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 Trapezoidal method's unconditionable 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

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

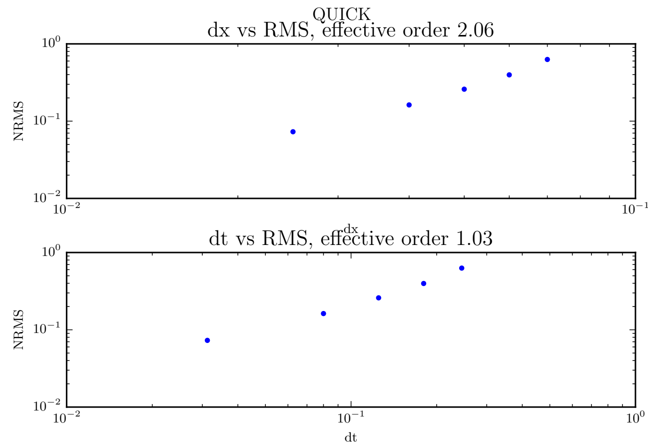

Fig. 5: Effective order of the QUICK method

is false. This leads to the instability and resultant large NRMS of 4.70E+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.

Method	Δx	Δt
FTCS	2.06	1.06
Upwind	1.00	0.53
Trap	1.97	0.99
QUICK	2.06	1.03

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

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. The error suggests thats the primary error from the FTCS scheme is in the ampltitude of the solution, while the primary error in the Upwind, Trapezoidal, and QUICK schemes is in the wave speed.

While the FTCS scheme 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.

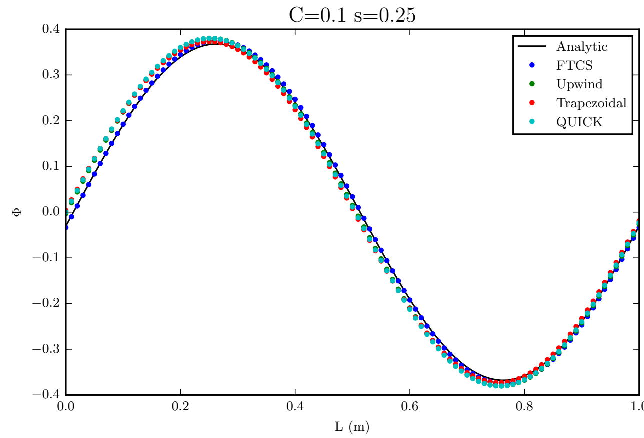

Fig. 6: Results of each scheme for Case 1

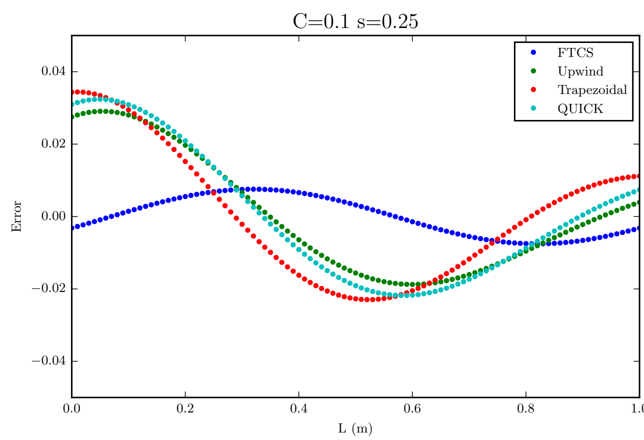

Fig. 7: Error for each scheme for Case 1

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/.

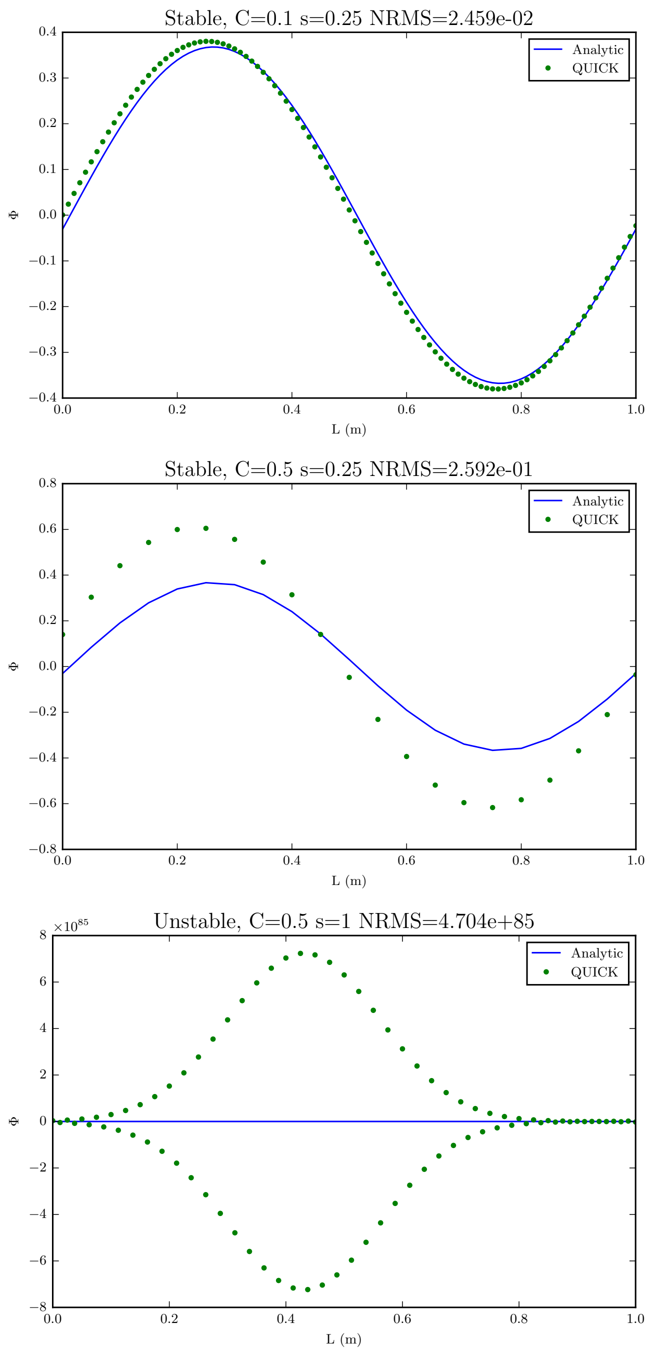

Fig. 8: QUICK method's transition into instability

Appendix A: Python Code

```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
```

```
    return np.array(Phi_old)


def Upwind(Phi, c):
    '''
    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.
    '''

    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] = (D * dt / dx ** 2 * (Phi_old[1] - 2 * Phi_old[0] + Phi_old[-1]) -
                  u * dt / (2 * dx) * (3 * Phi_old[0] - 4 * Phi_old[-1] + Phi_old[-2]) +
                  Phi_old[0])

        Phi[1] = (D * dt / dx ** 2 * (Phi_old[2] - 2 * Phi_old[1] + Phi_old[0]) -
                  u * dt / (2 * dx) * (3 * Phi_old[1] - 4 * Phi_old[0] + Phi_old[-1]) +
                  Phi_old[1])

        for i in range(2, N - 1):
            Phi[i] = (D * dt / dx ** 2 * (Phi_old[i + 1] - 2 * Phi_old[i] + Phi_old[i - 1]) -
                      u * dt / (2 * dx) * (3 * Phi_old[i] - 4 * Phi_old[i - 1] + Phi_old[i - 2]) +
                      Phi_old[i])

        Phi[-1] = (D * dt / dx ** 2 * (Phi_old[0] - 2 * Phi_old[-1] + Phi_old[-2]) -
                   u * dt / (2 * dx) * (3 * Phi_old[-1] - 4 * Phi_old[-2] + Phi_old[-3]) +
                   Phi_old[-1])

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

    return np.array(Phi_old)


def Trapezoidal(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)

    # Create Coefficient Matrix
    upper = [-(dt * D) / (2 * dx ** 2) + dt * u / (4 * dx) for _ in range(0, N)]
    main = [1 + (dt * D / (dx ** 2)) for _ in range(0, N)]
    lower = [-(dt * D) / (2 * dx ** 2) - dt * u / (4 * dx) for _ in range(0, N)]

    data = lower, main, upper
    diags = np.array([-1, 0, 1])
    matrix = sparse.spdiags(data, diags, N, N).todense()

    # Set values for cyclic boundary conditions
    matrix[0, N - 1] = -(dt * D) / (2 * dx ** 2) - dt * u / (4 * dx)
    matrix[N - 1, 0] = -(dt * D) / (2 * dx ** 2) + dt * u / (4 * dx)

    # create blank b array
    b = np.array(Phi_old)

    t = 0
    while t <= tau:
        # Enforce our periodic boundary condition
        b[0] = ((dt * D / (2 * dx ** 2)) * (Phi_old[1] - 2 * Phi_old[0] + Phi_old[-1]) -
```

```
                (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
    else:
        stability = 'Unstable, '

    plt.title(stability +
              'C=' + title.split(' ')[1] +
              ' s=' + title.split(' ')[2] +
              ' NRMS={0:.3e}'.format(NRMS))
    plt.legend(loc='best')

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

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


def save_state(x, analytic, solutions, state):
    plt.figure(figsize=fig_dims)

    plt.plot(x, analytic, 'k', label='Analytic')
    for solution in solutions:
        plt.plot(x, solution[0], '.', label=solution[1])

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

    title = 'C=' + state.split(' ')[0] + ' s=' + state.split(' ')[1]
    plt.title(title)
    plt.legend(loc='best')

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

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


def save_state_error(x, analytic, solutions, state):
    plt.figure(figsize=fig_dims)

    for solution in solutions:
        Error = solution[0] - analytic
        plt.plot(x, Error, '.', label=solution[1])

    plt.ylabel('Error')
    plt.xlabel('L (m)')
    plt.ylim([-0.05, 0.05])

    title = 'C=' + state.split(' ')[0] + ' s=' + state.split(' ')[1]
    plt.title(title)
    plt.legend(loc='best')

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

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

```
        save_figure(c.x, Phi_analytic, Phi_ftcs,
                    'FTCS ' + str(C) + ' ' + str(s), stable[0])
        save_figure(c.x, Phi_analytic, Phi_upwind,
                    'Upwind ' + str(C) + ' ' + str(s), stable[1])
        save_figure(c.x, Phi_analytic, Phi_trapezoidal,
                    'Trapezoidal ' + str(C) + ' ' + str(s), stable[2])
        save_figure(c.x, Phi_analytic, Phi_quick,
                    'QUICK ' + str(C) + ' ' + str(s), stable[3])

        # and group comparisons
        save_state(c.x, Phi_analytic, solutions, str(C) + ' ' + str(s))
        save_state_error(c.x, Phi_analytic, solutions, str(C) + ' ' + str(s))

    NRMS = []
    for solution in solutions:
        err = solution[0] - Phi_analytic
        NRMS.append(np.sqrt(np.mean(np.square(err)))/(max(Phi_analytic) - min(Phi_analytic)))

    return [c.dx, c.dt, NRMS[0], NRMS[1], NRMS[2], NRMS[3]]


def main():
    # Cases
    C = [0.1,   0.5,   2, 0.5, 0.5]
    s = [0.25, 0.25, .25, 0.5,   1]
    for C_i, s_i in zip(C, s):
        generate_solutions(C_i, s_i)

    # Stable values for each case to find effective order of methods
    C = [0.10, 0.50, 0.40, 0.35, 0.5]
    s = [0.25, 0.25, 0.25, 0.40, 0.5]
    calc_stability(C, s, 'FTCS')

    C = [0.1, 0.2, 0.3, 0.05, 0.1]
    s = [0.4, 0.3, 0.2, 0.15, 0.1]
    calc_stability(C, s, 'Upwind')

    C = [0.5, 0.6, 0.7, 0.8, 0.9]
    s = [0.25, 0.25, 0.25, 0.25, 0.25]
    calc_stability(C, s, 'Trapezoidal')

    C = [0.25, 0.4, 0.5, 0.6,  0.7]
    s = [0.25, 0.25, 0.25, 0.25, 0.25]
    calc_stability(C, s, 'QUICK')


if __name__ == "__main__":
    main()
```

Listing 1: Code to create plots and solutions

```
import numpy as np
import matplotlib
matplotlib.use('TkAgg')

# Configure figures for production
WIDTH = 495.0  # the number latex spits out
FACTOR = 1.0   # the fraction of the width the figure should occupy
fig_width_pt = WIDTH * FACTOR

inches_per_pt = 1.0 / 72.27
golden_ratio = (np.sqrt(5) - 1.0) / 2.0      # because it looks good
fig_width_in = fig_width_pt * inches_per_pt  # figure width in inches
fig_height_in = fig_width_in * golden_ratio   # figure height in inches
fig_dims = [fig_width_in, fig_height_in]  # fig dims as a list
```

Listing 2: Code to generate pretty plots