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

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

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

```python

    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.clf()


def save_state(x, analytic, solutions, state):
    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.clf()


def save_state_error(x, analytic, solutions, state):
    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.clf()


def plot_order(x, t, RMS):
    fig = plt.figure()
    RMS, title = RMS[0], RMS[1]

```

```python
    # 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.clf()


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

    FTCS = dx < (2 * D) / u and dt < dx ** 2 / (2 * D)
    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'''

    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