notebook

October 13, 2022

[948]:
```
import numpy as np
import matplotlib.pyplot as plt
import cycler
```

Define initial condition $\psi(x, 0)$.

[949]:
```
def f(x):
    return np.sin(2*np.pi*x)**6
```

Define $\partial\psi/\partial x$.

[950]:
```
def f_x(c, x):
    return -12*c*np.pi*((np.sin(2*np.pi*x))**5)*np.cos(2*np.pi*x)
```

0.0.1 Problem 1

Compute solutions to the wind speed advection equation using Adams-Bashforth (3rd-order) in time and 4th-order central differencing in space.

Function F for AB3.

[951]:
```
def f_x_(c, dx, dt, f_):
    return c*cdf(dt, dx, f_)
```

Define discretization schemes: - Starter: Runge-Kutta, 3rd order - Temporal: Adams-Bashforth, 3rd-order - Spatial: 4th-order compact difference

Runge-Kutta, 3rd order

[952]:
```
def rk3(h, c, dt, dx, x, psi):
    # q_1 = h*f_x(c, x)
    q_1 = h*f_x_(c, dx, dt, psi)
    psi_1 = psi + (1/3)*q_1
    # q_2 = h*f_x(c, psi_1) - (5/9)*q_1
    q_2 = h*f_x_(c, dx, dt, psi) - (5/9)*q_1
    psi_2 = psi_1 + (15/16)*q_2
    # q_3 = h*f_x(c, psi_2) - (153/128)*q_2
    q_3 = h*f_x_(c, dx, dt, psi) - (153/128)*q_2
    psi_n_1 = psi_2 + (8/15)*q_3
    return psi_n_1
```

4th-order centered difference

[953]:
```
def cdf(dt, dx, f_):
    f_prime = np.zeros((len(f_),))
    for i in range(0, len(f_)):
        # Using modulo for indices to handle boundary conditions, since BCs are␣
↪periodic
        # f_prime[i] = (f_[(i-2) % len(f_)] - 8*f_[(i-1) % len(f_)] +␣
↪8*f_[(i+1) % len(f_)] - f_[(i+2) % len(f_)])/(12*dx)
        f_prime[i] = (4/(6*dx))*(f_[(i+1) % len(f_)] - f_[(i-1) % len(f_)]) -␣
↪(1/(12*dx))*(f_[(i+2) % len(f_)] - f_[(i-2) % len(f_)])
    return f_prime
```

Adams-Bashforth, 3rd order

[954]:
```
def ab3(dx, dt, c, x, psis):
    # Get previous psi values [psi(n), psi(n-1), psi(n-2)]
    psi_n, psi_n_1, psi_n_2 = psis[0], psis[1], psis[2]
    # Generate new psi value psi(n+1)
    psi_np1 = psi_n + (dt/12)*(23*f_x_(c, dx, dt, psi_n) - 16*f_x_(c, dx, dt,␣
↪psi_n_1) + 5*f_x_(c, dx, dt, psi_n_2))

    return psi_np1
```

Define function to carry out the discretization and step through time.

[955]:
```
def advection_p1(dx, c, cfl, t_max, plot=False, plot_step=40, printout=False):
    # Compute timestep to meet Courant number
    dt = dx*cfl/c
    # Spatial domain
    x = np.arange(0, 1, dx)
    # Temporal domain
    t = np.arange(0, t_max, dt)

    # Initialize array for values and apply initial condition
    y = np.full((len(t)+1, len(x)), np.nan)
    y[0, :] = f(x)

    # Get exact solution
    exact = np.sin(2*np.pi*(x-c*t_max))**6

    # Iterate through timesteps
    for i, t_ in enumerate(t):
        if printout and i % 10 == 0:
            print('Step: {0} | Timestep: {1:.2f}'.format(i, t_))

        # Starter function
        if i < 2:
```

```
            print('Using starter scheme...') if printout else None
            y[i+1, :] = rk3(dt, c, dt, dx, x, y[i, :])
        else:
            print('Using full scheme...') if printout else None
            # Get function values from previous timesteps
            psis = [y[i, :], y[i-1, :], y[i-2, :]]
            y[i+1, :] = ab3(dx, dt, c, x, psis)

    ''' Plotting. '''
    if plot:
        plt.rcParams["axes.prop_cycle"] = plt.cycler('color', plt.cm.viridis(np.
↪linspace(0, 1, len(plot_step))))
        fig, ax = plt.subplots(figsize=(4, 3))
        for step in plot_step:
            im = ax.plot(x, y[step], marker='o', markersize=4, label='Step {0}'.
↪format(step))

        ax.set_title('Time = {0:.2f} | dx = {1:.2f} | dt = {2:.2f}'.
↪format(t[step], dx, dt), fontsize=10)
        ax.set_xlim([min(x), max(x)])
        ax.set_ylim([0, 1])
        fig.tight_layout()
        fig.legend(loc='upper right', bbox_to_anchor=(1.25, 0.925),␣
↪frameon=False, fontsize=8)

    # Return the values at the maximum time
    return x, y[-1, :], exact
```

Execute and plot the results

[ ]:
```
# Wave speed (c)
c = 0.1
# Spatial increments
dxs = [1/20, 1/40, 1/80, 1/160]
# Strings to represent spatial increments (for plotting purposes only)
dxs_str = ['1/20', '1/40', '1/80', '1/160']
# Courant number and Courant number list
cfls = [0.1, 0.2, 0.4, 0.8]
# Maximum time
t_max = 50
# Boolean to control prints to console
printout = False
# Computation mode: variable dx or CFL values
dx_calc = True

# Initialize array to hold each run's output array
values = {}
```

```
# Iterate over dx values
if dx_calc:
    cfl = 0.1
    for i, dx in enumerate(dxs):
        # End step
        end_step = int(t_max/(cfl*dx/c))
        # Perform calculation, plot individual results
        arr = advection_p1(dx, c, cfl, t_max, plot=False, plot_step=[end_step],␣
 ↪printout=printout)
        # Store array
        values['dx = {0}'.format(dxs_str[i])] = {'x': arr[0], 'y': arr[1],␣
 ↪'exact': arr[2]}

# Iterate over Courant number values
else:
    dx_index = 2
    dx = dxs[dx_index]
    for i, cfl in enumerate(cfls):
        # End step
        end_step = int(t_max/(cfl*dx/c))
        # Perform calculation, plot individual results
        arr = advection_p1(dx, c, cfl, t_max, plot=False, plot_step=[0,␣
 ↪end_step // 16, end_step // 2, 3*end_step // 4], printout=printout)
        # Store array
        values['CFL = {0}'.format(cfl)] = {'x': arr[0], 'y': arr[1], 'exact':␣
 ↪arr[2]}

''' Plotting.'''
# Initialize figure
fig, ax = plt.subplots(figsize=(4, 3))
# Define formatting index, color list, marker list
f_index, markers = 0, ['o', '^', '+', '2']
# Set color cycling
plt.rcParams["axes.prop_cycle"] = plt.cycler('color', plt.cm.Blues(np.
 ↪linspace(0.375, 1, len(dxs))))
# Plot the actual values
for key, value in values.items():
    im = ax.plot(value['x'], value['y'], marker=markers[f_index], markersize=5,␣
 ↪ls='--', label=key)
    if key == 'dx = 1/160' or key == 'CFL = 0.1':
        ax.plot(value['x'], value['exact'], c='r', label='exact', zorder=0)
    f_index += 1
# Metadata
if dx_calc:
    ax.set_title('Courant number: {0:.1f}'.format(cfl), fontsize=10)
```

```
else:
    ax.set_title('dx = {0}'.format(dxs_str[dx_index]), fontsize=10)
ax.set_xlim([min(value['x']), max(value['x'])])
ax.set_ylim([0, 1])
fig.tight_layout()
fig.legend(loc='upper right', bbox_to_anchor=(1.25, 0.925), frameon=False,␣
 ↪fontsize=8)
# plt.savefig('figs/p1c1.png', dpi=300, bbox_inches='tight')

# Initialize figure
fig, ax = plt.subplots(figsize=(4, 3))
# Collect RMS values
rmse = [rms(value['y'], value['exact']) for _, value in values.items()]
print(rmse)
# Plot the actual values
im = ax.loglog(dxs, rmse, marker='o', color='k')
ax.set_xticks([0.005, 0.01, 0.05])
ax.set_ylim([1e-3, 1e0])
# Metadata
ax.set_ylabel('RMSE')
ax.set_xlabel('dx') if dx_calc else ax.set_xlabel('CFL')
ax.set_title('Root mean squared error', fontsize=10)
plt.gca().invert_xaxis()
fig.tight_layout()
# plt.savefig('figs/p1c2.png', dpi=300, bbox_inches='tight')
```

0.0.2 Problem 2

Compute solutions to the wind speed advection equation using leapfrog with an Asselin filter in time and Lele's 4th-order compact differencing in space.

Runge-Kutta, 3rd order

[956]:
```
def rk3_2(h, c, dt, dx, x, psi):
    # q_1 = h*f_x(c, x)
    q_1 = h*f_x_2(c, dx, dt, psi)
    psi_1 = psi + (1/3)*q_1
    # q_2 = h*f_x(c, psi_1) - (5/9)*q_1
    q_2 = h*f_x_2(c, dx, dt, psi) - (5/9)*q_1
    psi_2 = psi_1 + (15/16)*q_2
    # q_3 = h*f_x(c, psi_2) - (153/128)*q_2
    q_3 = h*f_x_2(c, dx, dt, psi) - (153/128)*q_2
    psi_n_1 = psi_2 + (8/15)*q_3
    return psi_n_1
```

4th-order Lele compact difference

[957]:
```python
def cdf_compact(dt, dx, f_):
    # f_ is an array of function values at a given time
    # dt is the timestep

    # Create upper, central, and bottom diagonals
    a = np.full(len(f_), 5/24)
    b = np.full(len(f_), 14/24)
    c = np.full(len(f_), 5/24)

    # Create RHS
    rhs = np.zeros((len(f_), ))
    for i in range(0, len(f_)):
        # Using modulo for indices to handle boundary conditions, since BCs are␣
 ↪periodic
        rhs[i] = (11*(f_[(i+1) % len(f_)] - f_[(i-1) % len(f_)])/2 + (f_[(i+2)␣
 ↪% len(f_)] - f_[(i-2) % len(f_)])/4)/(12*dx)

    f_prime = cyc_tridiag(len(f_), a, b, c, rhs)

    return f_prime
```

Leapfrog with Asselin filter

[958]:
```python
def leapfrog(dx, dt, c, x, gamma, psis):
    # Get previous psi values [psi(n), psi(n-1)]
    psi_n, psi_n_1, psi_n_2 = psis
    # Asselin filter - previous step
    psi_n_1_ass = psi_n_1 + gamma*(psi_n - 2*psi_n_1 + psi_n_2)
    # Compute values at next time step
    psi_np1 = psi_n_1_ass + 2*dt*f_x_2(c, dx, dt, psi_n)
    # Compute filtered values at current time step
    psi_n_ass = psi_n + gamma*(psi_n_1_ass - 2*psi_n + psi_np1)

    return psi_n_ass, psi_np1
```

Define $\partial\psi/\partial x$.

[959]:
```python
def f_x_2(c, dx, dt, f_):
    return c*cdf_compact(dt, dx, f_)
```

Define function to carry out the discretization and step through time

[960]:
```python
def advection_p2(dx, c, cfl, t_max, plot=False, plot_step=40, printout=False):
    # Compute timestep to meet Courant number
    dt = dx*cfl/c
    # Spatial domain
    x = np.arange(0, 1, dx)
    # Temporal domain
```

```
    t = np.arange(0, t_max, dt)
    # Asselin filter value
    gamma = 0.1

    # Initialize array for values and apply initial condition
    y = np.full((len(t)+1, len(x)), np.nan)
    y[0, :] = f(x)

    # Get exact solution
    exact = np.sin(2*np.pi*(x-c*t_max))**6

    # Iterate through timesteps
    for i, t_ in enumerate(t):
        if printout and i % 10 == 0:
            print('Step: {0} | Timestep: {1:.2f}'.format(i, t_))

        # Starter function
        if i < 3:
            print('Using starter scheme...') if printout else None
            y[i+1, :] = rk3_2(dt, c, dt, dx, x, y[i, :])
        else:
            print('Using full scheme...') if printout else None
            # Get function values from previous timesteps
            psis = [y[i, :], y[i-1, :], y[i-2, :]]
            y[i, :], y[i+1, :] = leapfrog(dx, dt, c, x, gamma, psis)

    ''' Plotting. '''
    if plot:
        plt.rcParams["axes.prop_cycle"] = plt.cycler('color', plt.cm.viridis(np.
↪linspace(0, 1, len(plot_step))))
        fig, ax = plt.subplots(figsize=(4, 3))
        for step in plot_step:
            im = ax.plot(x, y[step], marker='o', markersize=4, label='Step {0}'.
↪format(step))

        ax.set_title('Time = {0:.2f} | dx = {1:.2f} | dt = {2:.2f}'.
↪format(t[step], dx, dt), fontsize=10)
        ax.set_xlim([min(x), max(x)])
        ax.set_ylim([0, 1])
        fig.tight_layout()
        fig.legend(loc='upper right', bbox_to_anchor=(1.25, 0.925),␣
↪frameon=False, fontsize=8)
        plt.show()

    # Return the values at the maximum time
    return x, y[-1, :], exact
```

Execute and plot the runs

[ ]:
```
# Wave speed (c)
c = 0.1
# Spatial increments
dxs = [1/20, 1/40, 1/80, 1/160]
# Strings to represent spatial increments (for plotting purposes only)
dxs_str = ['1/20', '1/40', '1/80', '1/160']
# Courant number and Courant number list
cfls = [0.1, 0.2, 0.4, 0.8]
# Maximum time
t_max = 50
# Boolean to control prints to console
printout = False
# Computation mode: variable dx or CFL values
dx_calc = False

# Initialize array to hold each run's output array
values = {}

# Iterate over dx values
if dx_calc:
    cfl = 0.1
    for i, dx in enumerate(dxs):
        # End step
        end_step = int(t_max/(cfl*dx/c))
        # Perform calculation, plot individual results
        arr = advection_p2(dx, c, cfl, t_max, plot=False, plot_step=[0],
 ↪printout=printout)
        # Store array
        values['dx = {0}'.format(dxs_str[i])] = {'x': arr[0], 'y': arr[1],
 ↪'exact': arr[2]}

# Iterate over Courant number values
else:
    dx_index = 2
    dx = dxs[dx_index]
    for i, cfl in enumerate(cfls):
        # End step
        end_step = int(t_max/(cfl*dx/c))
        # Perform calculation, plot individual results
        arr = advection_p2(dx, c, cfl, t_max, plot=False, plot_step=[end_step],
 ↪printout=printout)
        # Store array
        values['CFL = {0}'.format(cfl)] = {'x': arr[0], 'y': arr[1], 'exact':
 ↪arr[2]}
```

```python
''' Plotting. '''
# Initialize figure
fig, ax = plt.subplots(figsize=(4, 3))
# Define formatting index, color list, marker list
f_index, markers = 0, ['o', '^', '+', '2']
# Set color cycling
plt.rcParams["axes.prop_cycle"] = plt.cycler('color', plt.cm.Blues(np.linspace(0.375, 1, len(dxs))))
# Plot the actual values
for key, value in values.items():
    im = ax.plot(value['x'], value['y'], marker=markers[f_index], markersize=5, ls='--', label=key)
    if key == 'dx = 1/160' or key == 'CFL = 0.1':
        ax.plot(value['x'], value['exact'], c='r', label='exact', zorder=0)
    f_index += 1
# Metadata
if dx_calc:
    ax.set_title('Courant number: {0:.1f}'.format(cfl), fontsize=10)
else:
    ax.set_title('dx = {0}'.format(dxs_str[dx_index]), fontsize=10)
ax.set_xlim([min(value['x']), max(value['x'])])
ax.set_ylim([0, 1])
fig.tight_layout()
fig.legend(loc='upper right', bbox_to_anchor=(1.25, 0.925), frameon=False, fontsize=8)
# plt.savefig('figs/p2c1.png', dpi=300, bbox_inches='tight')

# Initialize figure
fig, ax = plt.subplots(figsize=(4, 3))
# Collect RMS values
rmse = [rms(value['y'], value['exact']) for _, value in values.items()]
print(rmse)
# Plot the actual values
im = ax.loglog(dxs, rmse, marker='o', color='k')
ax.set_xticks([0.005, 0.01, 0.05])
ax.set_ylim([1e-3, 1e0])
# Metadata
ax.set_ylabel('RMSE')
ax.set_xlabel('dx') if dx_calc else ax.set_xlabel('CFL')
ax.set_title('Root mean squared error', fontsize=10)
plt.gca().invert_xaxis()
fig.tight_layout()
# plt.savefig('figs/p2c2.png', dpi=300, bbox_inches='tight')\
```

0.0.3 Problem 3

(a) Derive coefficient values that give a 3rd-order accurate scheme in space and time.

(a) (i). Derive a 3rd-order accurate scheme for space.

[ ]:
```
from sympy import *

# Initialize symbolic variable
h = Symbol('h')
# Define the matrix system
A = Matrix([[1, 1, 1, 1],
            [-2*h, -h, 0, h],
            [2*(h**2), (1/2)*(h**2), 0, (1/2)*(h**2)],
            [(-4/3)*(h**3), (-1/6)*(h**3), 0, (1/6)*(h**3)]])
B = Matrix([0, -1, 0, 0])

# Solve the linear system
system = A, B
linsolve(system)
```

Test the coefficients to ensure 3rd-order accuracy.

[806]:
```
a, b, c, d, e = -1/6, 1, -1/2, -1/3, 1

#
print('0th-order: {0:.3f}'.format(a + b + c + d + 0*e))
# '
print('1st-order: {0:.3f}'.format(-2*a - b + 0*c + d + e))
# ''
print('2nd-order: {0:.3f}'.format(2*a + (1/2)*b + 0*c + (1/2)*d + 0*e))
# '''
print('3rd-order: {0:.3f}'.format((-4/3)*a - (1/6)*b + 0*c + (1/6)*d + 0*e))
# '''
print('4th-order: {0:.3f}'.format((16/24)*a + (1/24)*b + 0*c + (1/24)*d + 0*e))
```

```
0th-order: 0.000
1st-order: 0.000
2nd-order: 0.000
3rd-order: 0.000
4th-order: -0.083
```

(a) (ii) Do this for the spatial representation of time of the $\partial^2\psi/\partial x^2$ term.

- Trial 1: do regular Taylor table
- Trial 2: shift by a derivative up (instead of starting at ψ, start at $\partial\psi$).

[967]:
```
from sympy import *

# Initialize symbolic variable
h = Symbol('h')
# Define the matrix system
```

```python
''' Trial 1. '''
A_1 = Matrix([[1, 1, 1, 1],
              [-2*h, -h, 0, h],
              [2*(h**2), (1/2)*(h**2), 0, (1/2)*(h**2)],
              [(-4/3)*(h**3), (-1/6)*(h**3), 0, (1/6)*(h**3)]])
B_1 = Matrix([0, 0, -1, 0])

''' Trial 2. '''
# A_1 = Matrix([[-2*h, -h, h],
#                [2*(h**2), (1/2)*(h**2), (1/2)*(h**2)],
#                [(-4/3)*(h**3), (-1/6)*(h**3), (1/6)*(h**3)],
#                [(2/3)*(h**4), (1/24)*(h**4), (1/24)*(h**4)]])
# B_1 = Matrix([0, -1, 0])

# Solve the linear system
system_1 = A_1, B_1
linsolve(system_1)
```

[967]: $$\left\{\left(\frac{2.22044604925031 \cdot 10^{-16}}{h^2},\ -\frac{1.0}{h^2},\ \frac{2.0}{h^2},\ -\frac{1.0}{h^2}\right)\right\}$$

Test the coefficients to make sure they're 3rd-order accurate

[865]:
```python
a, b, c, d, e = 0, -1, 2, -1, 1

#
print('0th-order: {0:.3f}'.format(a + b + c + d + 0*e))
# '
print('1st-order: {0:.3f}'.format(-2*a - b + 0*c + d + 0*e))
# ''
print('2nd-order: {0:.3f}'.format(2*a + (1/2)*b + 0*c + (1/2)*d + e))
# '''
print('3rd-order: {0:.3f}'.format((-4/3)*a - (1/6)*b + 0*c + (1/6)*d + 0*e))
# '''
print('4th-order: {0:.3f}'.format((16/24)*a + (1/24)*b + 0*c + (1/24)*d + 0*e))
```

```
0th-order: 0.000
1st-order: 0.000
2nd-order: 0.000
3rd-order: 0.000
4th-order: -0.083
```

(b) (iii) Do this for the spatial representation of time of the $\partial^3\psi/\partial x^3$ term.

[884]:
```python
from sympy import *

# Initialize symbolic variable
h = Symbol('h')
# Define the matrix system
```

```python
''' Trial 1. '''
A_2 = Matrix([[1, 1, 1, 1],
              [-2*h, -h, 0, h],
              [2*(h**2), (1/2)*(h**2), 0, (1/2)*(h**2)],
              [(-4/3)*(h**3), (-1/6)*(h**3), 0, (1/6)*(h**3)]])
B_2 = Matrix([0, 0, 0, -1])

''' Trial 2. '''
# A_1 = Matrix([[-2*h, -h, h],
#               [2*(h**2), (1/2)*(h**2), (1/2)*(h**2)],
#               [(-4/3)*(h**3), (-1/6)*(h**3), (1/6)*(h**3)],
#               [(2/3)*(h**4), (1/24)*(h**4), (1/24)*(h**4)]])
# B_1 = Matrix([0, -1, 0])

# Solve the linear system
system_2 = A_2, B_2
linsolve(system_2)
```

[884]: $\left\{\left(\frac{1.0}{h^3}, -\frac{3.0}{h^3}, \frac{3.0}{h^3}, -\frac{1.0}{h^3}\right)\right\}$

Test the coefficients to make sure they're 3rd-order accurate

[887]:
```python
a, b, c, d, e = 1, -3, 3, -1, 1

#
print('0th-order: {0:.3f}'.format(a + b + c + d + 0*e))
# '
print('1st-order: {0:.3f}'.format(-2*a - b + 0*c + d + 0*e))
# ''
print('2nd-order: {0:.3f}'.format(2*a + (1/2)*b + 0*c + (1/2)*d + 0*e))
# '''
print('3rd-order: {0:.3f}'.format((-4/3)*a - (1/6)*b + 0*c + (1/6)*d + e))
# '''
print('4th-order: {0:.3f}'.format((16/24)*a + (1/24)*b + 0*c + (1/24)*d + 0*e))
```

```
0th-order: 0.000
1st-order: 0.000
2nd-order: 0.000
3rd-order: 0.000
4th-order: 0.500
```

0.0.4 Auxiliary functions

Root mean square (RMS) calculation

[947]:

```python
def rms(x, y):
    return np.sqrt(np.nansum(np.array([(x[i] - y[i])**2 for i in range(0,␣
 ↪len(x))])/len(x)))
```

Cyclic tridiagonal solver

[946]:
```python
def cyc_tridiag(jmx, a, b, c, f):

    ''' Written by D. Durran, translated by S. Ditkovsky. '''

    # jmx = dimension of all arrays
    # a = sub (lower) diagonal
    # b = center diagonal
    # c = super (upper) diagonal
    # f = right hand side

    fmx = f[-1]

    # Create work arrays
    q = np.empty(jmx)
    s = np.empty(jmx)

    #forward elimination sweep
    q[0] = -c[0]/b[0]
    f[0] = f[0]/b[0]
    s[0] = -a[0]/b[0]

    for j in range(jmx-1):
        p = 1./(b[j+1]+ a[j+1]*q[j])
        q[j+1] = -c[j+1]*p
        f[j+1] = (f[j+1] - a[j+1]*f[j])*p
        s[j+1] = -a[j+1]*s[j]*p

    #Backward pass

    q[-1] = 0.0
    s[-1] = 1.0

    for j in reversed(range(jmx-1)):
        s[j] = s[j] + q[j]*s[j+1]
        q[j] = f[j] + q[j]*q[j+1]

    #final pass
    f[-1] = (fmx-c[-1]*q[0] - a[-1]*q[-2])/(c[-1]*s[0] + a[-1]*s[-2] + b[-1])

    for j in range(jmx-1):
        f[j] = f[-1]*s[j] + q[j]
```

```
    return f
```