Gabriel Rios
AOS 575
September 29, 2022

Problem set 2

1. **Chemostat equation**
(note: the list below corresponds to the given problems as follows: 1-a1 → a, 1-a2 → b, 1-a3 → c, 1-b1 → d, 1-b2 → e, 1-b3 → f, 1-b4 → g, 1-b5 → h)

   (a) An explicit **two-stage** equation was used to model the chemostat equation. The time-stepping scheme chosen was the Heun variation of the 2nd-order Runge-Kutta method, which is shown below:

$$\psi_{n+1} = \psi_n + \Delta t \left[b_1 F\left(\xi_1, t_n + c_1 \Delta t\right) + b_2 F\left(\xi_2, t_n + c_2 \Delta t\right)\right] \tag{1}$$
$$\xi_1 = \psi_n \tag{2}$$
$$\xi_2 = \psi_n + \Delta t a_{2,1} F\left(\xi_1, t_n\right) = \psi_n + \Delta t a_{2,1} F\left(\psi_n, t_n\right) \tag{3}$$

   Using the Heun-specific values of $a_{2,1} = 1, b_1 = b_2 = 1/2, c_1 = 0, c_2 = 1$ and assuming that $F(\psi_n, t_n) = \frac{\partial \psi}{\partial t}$, (1) simplifies to:

$$\psi_{n+1} = \psi_n + \Delta t \psi_n^{(1)} + \frac{1}{2}\Delta t^2 \psi_n^{(2)} \tag{4}$$

   (b) To determine the leading order error term for this scheme using the simple equation given ($\frac{\partial \psi}{\partial t} = (\gamma - S)\psi$), rewrite the 2nd-order equation derived above:

$$\psi_{n+1} = \psi_n + \Delta t(\gamma - S)\psi_n + \frac{1}{2}\Delta t^2(\gamma - S)^2\psi_n \tag{5}$$

   Because this equation has the first 3 terms of the Taylor series expansion about ψ_{n+1}, the next term in the series can be considered as the leading order error term. This is:

$$= \frac{1}{6}\Delta t^3(\gamma - S)^3\psi_n \;\Rightarrow (\text{assume } \gamma/S = 10) \Rightarrow\; \frac{1}{6}\Delta t^3(9S)^3\psi_n \tag{6}$$
$$= \frac{729}{6} S \Delta t^3 \psi_n \tag{7}$$

   (c) Using Von Neumann stability analysis to evaluate the range of stable timesteps with Equation 5:

$$\psi_{n+1} = \psi_n \left[1 + \Delta t(\gamma - S) + \frac{1}{2}\Delta t^2(\gamma - S)^2\right] \tag{8}$$
$$A = \frac{\psi_{n+1}}{\psi_n} = 1 + \Delta t(\gamma - S) + \frac{1}{2}\Delta t^2(\gamma - S)^2 \tag{9}$$
$$\|A\|^2 = \frac{1}{4}(\gamma - S)^4\Delta t^4 + (\gamma - S)^3\Delta t^3 + 2(\gamma - S)^2\Delta t^2 + 2(\gamma - S)\Delta t + 1 \leq 1 \tag{10}$$

   Solving for Δt: (11)

$$0 \leq \Delta t \leq \frac{2}{9S} \tag{12}$$

   (d) The figures showing the approximation of the chemostat equations using the discretization provided can be seen in Figure 1.

   (e) See Figure 2a for a plot of the approximation of the chemostat equation solution using the scheme devised in (a). There is some overshooting of the approximation relative to the convergence value, although the time to convergence is very similar to that of the given scheme and slightly better for the nutrient concentration approximation.

(f) See Figure 2b for a plot of the approximation of the chemostat equation solution using the scheme devised in (a). The scheme is unstable at this timestep, and diverges. This is expected, since the timestep is much larger than the maximum timestep allowable for a stable solution, as shown in a previous section.

(g) See Figure 3 for a plot comparing the error of scheme approximation relative to the convergence value (value at maximum time for scheme run with timestep $\Delta t = 0.001/\gamma$) for nutrient concentration, N, at multiple timesteps. For reference, this error is calculated as:

$$\text{error} = \frac{\text{scheme value - convergence value}}{\text{convergence value}}$$

It can be seen that as the timestep increases, the performance between the schemes differs increasingly, with the custom 2nd-order scheme reaching convergence ($y = 1$) before the provided 1st-order scheme. It is worth noting that after a certain timestep ($\Delta t > 2/9S$), the 2nd-order scheme becomes unstable, whereas the 1st-order scheme remains stable despite its slower convergence.

(h) The results shown here demonstrate tradeoffs in choosing a time-stepping scheme for ecosystem models. The 1st-order scheme is stable over a wider range of timesteps than the derived 2nd-order scheme, although it is slower to converge. This leads to selection of a scheme based on the desired outcome and the resources available. If there are extensive computational resources available to accommodate a small timestep, the 2nd-order scheme is preferable because a steady-state will be obtained with fewer iterations and greater accuracy at any given timestep. On the other hand, if fewer resources are available and/or a less precise estimate is sufficient, the 1st-order approximation may be preferred.

2. **Compact difference derivation**

To derive the simplest spatially staggered 4th-order accurate approximation to $\frac{\partial \psi}{\partial x}$ with the given points, each term was expanded as a Taylor series as follows:

$$\psi_{j-1}^{(1)} = 0 + \psi_j^{(1)} - \Delta x \psi_j^{(2)} + \frac{1}{2}\Delta x^2 \psi_j^{(3)} - \frac{1}{6}\Delta x^3 \psi_j^{(4)} + \frac{1}{24}\Delta x^4 \psi_j^{(5)} + O(\Delta x^5) \tag{13}$$

$$\psi_{j-\frac{1}{2}} = \psi_j - \frac{\Delta x}{2}\psi_j^{(1)} + \frac{1}{2}\frac{\Delta x^2}{4}\psi_j^{(2)} - \frac{1}{6}\frac{\Delta x^3}{8}\psi_j^{(3)} + \frac{1}{24}\frac{\Delta x^4}{16}\psi_j^{(4)} - \frac{1}{120}\frac{\Delta x^5}{32}\psi_j^{(1)} + O(\Delta x^6) \tag{14}$$

$$\psi_{j+\frac{1}{2}} = \psi_j + \frac{\Delta x}{2}\psi_j^{(1)} + \frac{1}{2}\frac{\Delta x^2}{4}\psi_j^{(2)} + \frac{1}{6}\frac{\Delta x^3}{8}\psi_j^{(3)} + \frac{1}{24}\frac{\Delta x^4}{16}\psi_j^{(4)} + \frac{1}{120}\frac{\Delta x^5}{32}\psi_j^{(1)} + O(\Delta x^6) \tag{15}$$

$$\psi_{j+1}^{(1)} = 0 + \psi_j^{(1)} + \Delta x \psi_j^{(2)} + \frac{1}{2}\Delta x^2 \psi_j^{(3)} + \frac{1}{6}\Delta x^3 \psi_j^{(4)} + \frac{1}{24}\Delta x^4 \psi_j^{(5)} + O(\Delta x^5) \tag{16}$$

To find the combination of these points that produce a 4th-order accurate scheme for $\frac{\partial \psi}{\partial x}$, the following equation was devised:

$$a_0\psi_{j-1}^{(1)} + a_1\psi_{j-\frac{1}{2}} + a_2\psi_{j+\frac{1}{2}} + a_3\psi_{j+1}^{(1)} = -\psi_j^{(1)} \tag{17}$$

From this equation, a table can be constructed to allow for the construction of a system of equations to solve for the relevant coefficients.

	ψ_j	$\psi_j^{(1)}$	$\psi_j^{(2)}$	$\psi_j^{(3)}$
$\psi_j^{(1)}$	0	-1	0	0
$a_0\psi_{j-1}^{(1)}$	0	1	$-\Delta x$	$\frac{1}{2}\Delta x^2$
$a_1\psi_{j-\frac{1}{2}}$	0	$-\frac{1}{2}\Delta x$	$\frac{1}{8}\Delta x^2$	$-\frac{1}{48}\Delta x^3$
$a_2\psi_{j+\frac{1}{2}}$	0	$\frac{1}{2}\Delta x$	$\frac{1}{8}\Delta x^2$	$\frac{1}{48}\Delta x^3$
$a_3\psi_{j+1}^{(1)}$	0	1	Δx	$\frac{1}{2}\Delta x^2$

Solving for the coefficients a_n and renaming to denote an approximate solution, (18) becomes:

$$-\frac{1}{22}\psi^{(1)}_{j-1} - \frac{12}{11\Delta x}\psi_{j-\frac{1}{2}} + \frac{12}{11\Delta x}\psi_{j+\frac{1}{2}} - \frac{1}{22}\psi^{(1)}_{j+1} = \psi^{(1)}_{j,approx} \tag{18}$$

This is our 4th-order approximation for $\frac{\partial \psi}{\partial x}$. This equation will be used to approximate the derivative of the exact solution for the equation provided by Durran, Equation 3.27:

$$\psi_{j,exact}(x,t) = e^{i(kx-\omega t)} \tag{19}$$

Taking the derivative with respect to x:

$$\psi^{(1)}_{j,exact} = ike^{i(kx-\omega t)} \tag{20}$$

Now, (19) and (20) can be plugged into (18) to obtain an approximate value for $\psi^{(1)}_{j,approx}$. Once this is done, the spatial domain is defined with N grid points on the domain, resulting in $\frac{N}{2}$ wavenumbers. For each of these wavenumbers, the ratio $\psi^{(1)}_{j,approx}/\psi^{(1)}_{j,exact}$ is calculated and plotted in Figure 4. Additionally, the scaled frequency is plotted for both the exact and approximate solutions, as done in Durran, Figure 3.2. Another figure is attached to show the exact and approximate solutions over the spatial domain for a given wavenumber (see Figure 5). More details for this process can be seen in the code, which has been appended to this document and emailed to Bob Hallberg and Sam Ditkovsky.

Appendix

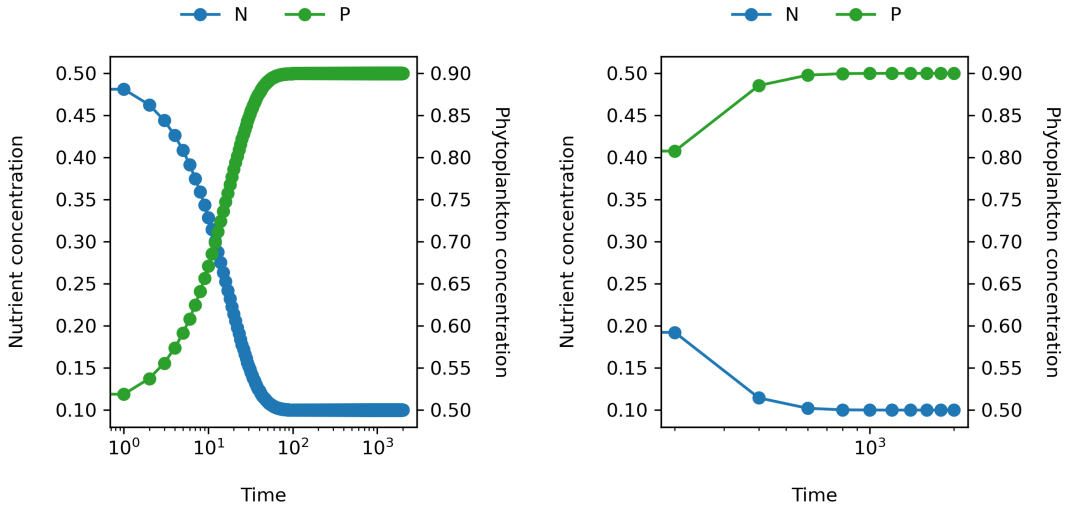

Figure 1: Plots demonstrating the discretization of the chemostat equations using the provided scheme at $\Delta t = 0.1/\gamma$ and $20/\gamma$, respectively.

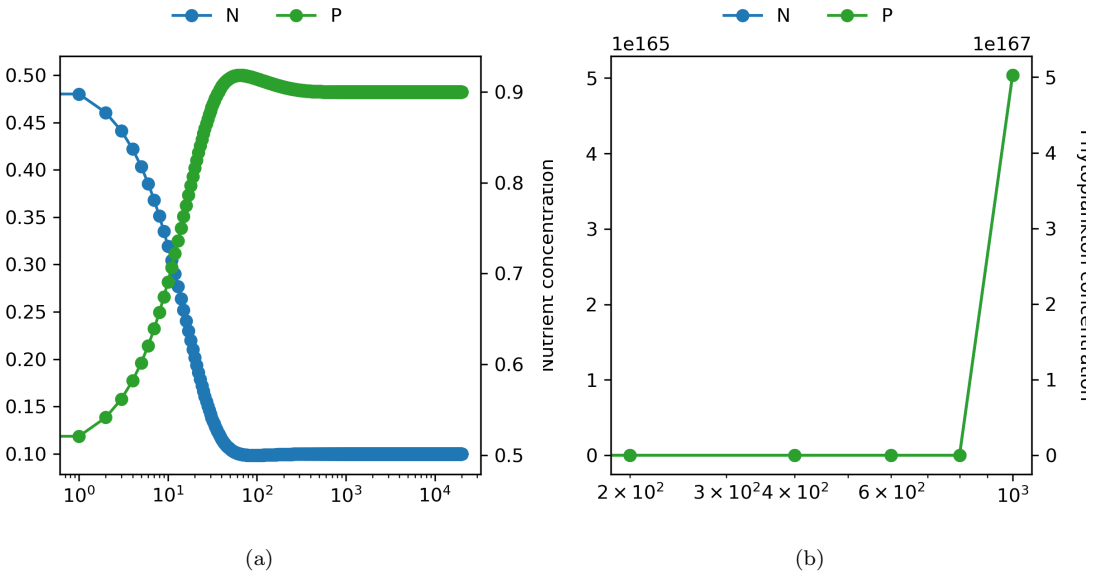

Figure 2: Plots demonstrating the discretization of the chemostat equations using the custom scheme at $\Delta t = 0.1/\gamma$ and $20/\gamma$, respectively.

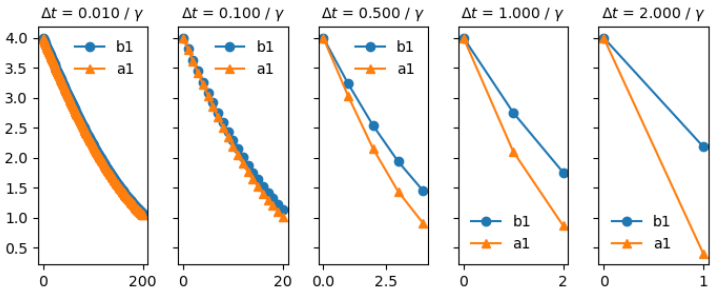

Figure 3: Plot demonstrating the ratio of the approximation to the convergence value for nutrient concentration, N, over time for the provided scheme (b1, blue) and the custom scheme (a1, orange) at multiple runs with increasing timesteps in each subplot.

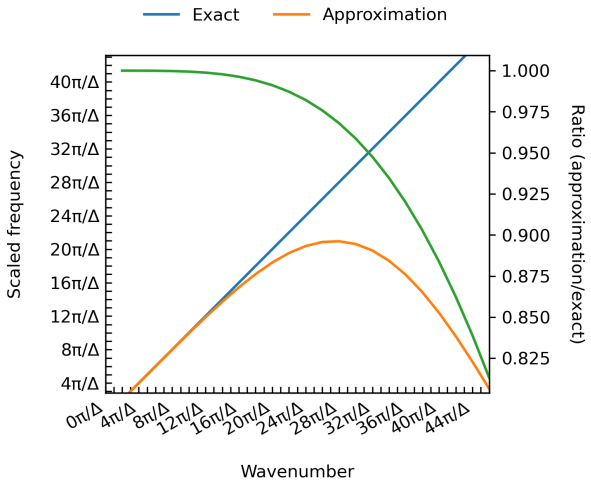

Figure 4: *Left y-axis*: Scaled frequency as a function of the wavenumber for the exact (blue) and approximate (orange) solutions. *Right y-axis*: Ratio of approximation to the exact solution over the range of resolvable wavenumbers.

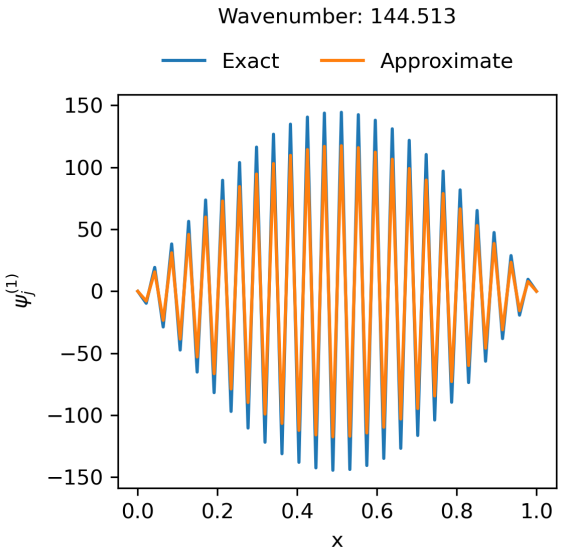

Figure 5: Approximate and exact solutions for a given wavenumber.

aos_575-pset2-code-rios

September 29, 2022

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

0.0.1 Problem 1-b1

Provided scheme, first-order

[ ]:
```
''' Assumed values and initial conditions. '''
# Phytoplankton growth rate
gamma = 0.1
# Water inflow rate
S = gamma/10
# Initial nutrient concentration
N_0 = 1

''' Loop controls. '''
# Maximum time
t_max = 200/gamma
# Time step
dt = 0.1/gamma
# Loop step number
n = 0

''' Initialize arrays. '''
# Initialize nutrient concentration array
N = [0.5*N_0]
# Initialize phytoplankton concentration array
P = [0.5*N_0]
# Initialize time array
t = np.linspace(0, t_max, int(t_max/dt)+1)

''' Run loop. '''
# While (step number * time step) is leq the maximum time
while n*dt < t_max:
    N_n1 = (N_0*N[n] + S*dt*N_0**2)/(N_0 + S*dt*N_0 + gamma*P[n]*dt)
    P_n1 = (N_0*P[n] + gamma*P[n]*N_n1*dt)/(N_0 + S*dt*N_0)
```

```
    # print('Step num: {0} | Time: {1:7.2f} | N(n): {2:7.3f} | P(n)$: {3:7.3f}
 ↪| N(n+1): {4:7.3f} | P(n+1)$: {5:7.3f}'.format(n, n*dt, N[n], P[n], N_n1,
 ↪P_n1))
    N.append(N_n1)
    P.append(P_n1)
    n += 1

''' Plotting. '''
fig, ax = plt.subplots(figsize=(4, 4))

ax.plot(t, N, marker='o', label='N')
ax_ = ax.twinx()
ax_.plot(t, P, marker='o', c='tab:green', label='P')

labelpad = 15
ax.set_xlabel('Time', labelpad=labelpad)
ax.set_ylabel('Nutrient concentration', labelpad=labelpad)
ax_.set_ylabel('Phytoplankton concentration', labelpad=labelpad+5, rotation=270)
ax.set_xscale('log')

fig.legend(frameon=False, ncol=2, bbox_to_anchor=(0.7, 1))
fig.suptitle('$\Delta$t = {0:.2f} / $\gamma$'.format(dt*gamma), y=1.05);
fig.tight_layout()
plt.savefig('figs/p1b1a.png', dpi=300)

# Remove arrays from memory
del N, P, n, N_n1, P_n1
```

0.0.2 Problem 1-b2

[ ]:
```
def given(gamma, N_0, t_max, dt):
    ''' Method for the scheme in 1-b1. '''

    ' Assumed values and initial conditions. '
    # Water inflow rate
    S = gamma/10
    # Initial nutrient concentration
    N_0 = 1

    ' Loop controls. '
    # Maximum time
    t_max = t_max/gamma
    # Time step
    dt = dt/gamma
    # Loop step number
    n = 0
```

```
    ' Initialize arrays. '
    # Initialize nutrient concentration array
    N = [0.5*N_0]
    # Initialize phytoplankton concentration array
    P = [0.5*N_0]
    # Initialize time array
    t = np.linspace(0, t_max, int(t_max/dt)+1)

    ' Run loop. '
    # While (step number * time step) is leq the maximum time
    while n*dt < t_max:
        N_n1 = (N_0*N[n] + S*dt*N_0**2)/(N_0 + S*dt*N_0 + gamma*P[n]*dt)
        P_n1 = (N_0*P[n] + gamma*P[n]*N_n1*dt)/(N_0 + S*dt*N_0)
        # print('Step num: {0} | Time: {1:7.2f} | N(n): {2:7.3f} | P(n)$: {3:7.
↪3f} | N(n+1): {4:7.3f} | P(n+1)$: {5:7.3f}'.format(n, n*dt, N[n], P[n],␣
↪N_n1, P_n1))
        N.append(N_n1)
        P.append(P_n1)
        n += 1

    return N, P
```

[ ]:
```
def custom(gamma, N_0, t_max, dt, plot=False):
    ''' Second-order Heun. '''

    ' Assumed values and initial conditions. '
    # Water inflow rate
    S = gamma/10
    # Initial nutrient concentration
    N_0 = 1

    ' Loop controls. '
    # Maximum time
    t_max = t_max/gamma
    # Time step
    dt = dt/gamma
    # Loop step number
    n = 0

    ' Initialize arrays. '
    # Initialize nutrient concentration array
    N = [0.5*N_0]
    # Initialize phytoplankton concentration array
    P = [0.5*N_0]
    # Initialize time array
    t = np.linspace(0, t_max, int(t_max/dt)+1)
```

```
    ' Derivative functions. '
    def N_(gamma, N_0, p, n, S):
        return S*N_0 - (S + gamma*p/N_0)*n
    def N__(gamma, N_0, p, n, S):
        # return -S*N_(gamma, N_0, p, n, S) - (gamma/N_0)*(P_(gamma, N_0, p, n,␣
↪S)*n + p*N_(gamma, N_0, p, n, S))
        return -S*N_(gamma, N_0, p, n, S) - (gamma/N_0)*(P_(gamma, N_0, p, n,␣
↪S)*n + p*N_(gamma, N_0, p, n, S))
    def P_(gamma, N_0, p, n, S):
        return (gamma/N_0)*p*n - S*p
    def P__(gamma, N_0, p, n, S):
        return (gamma/N_0)*(P_(gamma, N_0, p, n, S)*n - p*N_(gamma, N_0, p, n,␣
↪S)) - S*P_(gamma, N_0, p, n, S)

    ' Run loop. '
    # While (step number * time step) is leq the maximum time
    while n*dt < t_max:
        S = gamma/10
        N_n1 = N[n] + dt*N_(gamma, N_0, P[n], N[n], S) + ((dt**2)/2)*N__(gamma,␣
↪N_0, P[n], N[n], S)
        P_n1 = P[n] + dt*P_(gamma, N_0, P[n], N[n], S) + ((dt**2)/2)*P__(gamma,␣
↪N_0, P[n], N[n], S)
        # N_n1 = N[n]*(1 + dt*(gamma-S) + (1/2)*(dt**2/2)*(gamma-S)**2)
        # P_n1 = P[n]*(1 + dt*(gamma-S) + (1/2)*(dt**2/2)*(gamma-S)**2)

        # print('Step num: {0} | Time: {1:7.2f} | N(n): {2:7.3f} | P(n)$: {3:7.
↪3f} | N(n+1): {4:7.3f} | P(n+1)$: {5:7.3f}'.format(n, n*dt, N[n], P[n],␣
↪N_n1, P_n1))
        N.append(N_n1)
        P.append(P_n1)
        n += 1

    ''' Plotting. '''
    if plot:
        fig, ax = plt.subplots(figsize=(4, 4))
        ax.plot(t, N, marker='o', label='N')
        ax_ = ax.twinx()
        ax_.plot(t, P, marker='o', c='tab:green', label='P')
        ax.set_xscale('log')

        labelpad = 15
        ax.set_xlabel('Time', labelpad=labelpad)
        ax.set_ylabel('Nutrient concentration', labelpad=labelpad)
        ax_.set_ylabel('Phytoplankton concentration', labelpad=labelpad+5,␣
↪rotation=270)
```

```
        fig.legend(frameon=False, ncol=2, bbox_to_anchor=(0.675, 1))
        fig.suptitle('$\Delta$t = {0:.2f} / $\gamma$'.format(dt * gamma), y=1.
↪05);
        plt.savefig('figs/p1ca.png', dpi=300)

    return N, P

    # Remove arrays from memory
    del N, P, n, N_n1, P_n1
```

Run the custom scheme and generate plots.

[ ]:
```
''' Assumed values and initial condit'ions. '''
# Phytoplankton growth rate
gamma = 0.1
# Initial nutrient concentration
N_0 = 1
# Time step
dt = 0.1
# Maximum time
t_max = 200/gamma

_, _ = custom(gamma, N_0, t_max, dt, plot=True)
```

0.0.3 Problem 1-b4

Compare scheme performance over a range of timesteps.

[ ]:
```
''' Assumed values and initial conditions. '''
# Phytoplankton growth rate
gamma = 0.1
# Initial nutrient concentration
N_0 = 1

''' Exercise-specific parameters. '''
# Define solutions
N_final, P_final = 0.1, 0.9
# Define timestep coefficients
dts = [0.01, 0.1, 0.5, 1, 2, 2]
# Define maximum runtime
t_max = 2

# Initialize arrays to hold errors for given (g) and scheme (s)
g = {'N': {}, 'P': {}}
s = {'N': {}, 'P': {}}

# Iterate over timesteps:
```

```
for dt_ in dts:
    # Calculate results from each scheme
    N_given, P_given = given(gamma, N_0, t_max, dt_)
    N_scheme, P_scheme = custom(gamma, N_0, t_max, dt_)

    # Uncomment to check in on results
    # print('\t N(given): {0:8.3f} | P(given): {1:8.3f} | N(scheme): {2:8.3f} |␣
 ↪P(scheme): {3:8.3f} '.format(N_given[-1], P_given[-1], N_scheme[-1],␣
 ↪P_scheme[-1]))

    # Calculate relative errors for future plotting
    g['N']['{0}'.format(dt_)] = (np.abs(np.array(N_given) - N_final)/N_final)
    g['P']['{0}'.format(dt_)] = (np.abs(np.array(P_given) - P_final)/P_final)
    s['N']['{0}'.format(dt_)] = (np.abs(np.array(N_scheme) - N_final)/N_final)
    s['P']['{0}'.format(dt_)] = (np.abs(np.array(P_scheme) - P_final)/P_final)

''' Plot. '''
fig, axs = plt.subplots(figsize=(7, 3), ncols=len(dts)-1, nrows=1, sharey=True)
for i, ax in enumerate(fig.axes):
    ax.plot(g['N']['{0}'.format(dts[i])], label='b1', marker='o')
    ax.plot(s['N']['{0}'.format(dts[i])], label='a1', marker='^')
    ax.legend(frameon=False)
    ax.set_title('$\Delta t$ = {0:.3f} / $\gamma$'.format(dts[i]), fontsize=10)
fig.tight_layout()

plt.savefig('figs/p1b4a.png', bbox_inches='tight')
```

0.0.4 Problem 2a

Use solver to identify the coefficients that generate a 4th-order accurate approximation.

[ ]:
```
from sympy import *
h = Symbol('h')
# Define the matrix system
A = Matrix([[0, 1, 1, 0],
            [1, -(1/2)*h, (1/2)*h, 1],
            [-h, (1/8)*h**2, (1/8)*h**2, h],
            [(1/2)*h**2, -(1/48)*h**3, (1/48)*h**3, (1/2)*h**2]])
B = Matrix([0, 1, 0, 0])
# Solve the linear system
system = A, B
linsolve(system)
```

Check the coefficient values to ensure they satisfy the system of equations.

[ ]:
```
a, b, c, d = 1/22, 12/11, -12/11, 1/22
```

```
# '
print('1st-order: {0:.3f}'.format(a - (1/2)*b + (1/2)*c + d))
# ''
print('2nd-order: {0:.3f}'.format(-a + (1/8)*b + (1/8)*c + d))
# '''
print('3rd-order: {0:.3f}'.format((1/2)*a - (1/48)*b + (1/48)*c + (1/2)*d))
# ''''
print('4th-order: {0:.3f}'.format(-(1/6)*a + (1/384)*b + (1/384)*c + (1/6)*d))
# '''''
print('5th-order: {0:.3f}'.format((1/24)*a - (1/3840)*b + (1/3840)*c + (1/
↪24)*d))
```

0.0.5 Problem 2b

Calculate the exact and approximate solutions, and plot them.

[ ]:
```
def advection(c, x, dx, t, k):
    w = c*k
    return np.exp(1j*(k*x - w*t))
```

[ ]:
```
def psi_2(k, x, w, t):
    ''' Psi term for Problem 2.'''
    return np.exp(1j*(k*x - w*t))

def psi_2p(k, x, w, t):
    ''' Psi' term for Problem 2.'''
    return 1j*k*np.exp(1j*(k*x - w*t))

def approx(c, dx, k, t, x):
    # 4th-order dispersion relation (Durran, page 102 under Equation 3.33)
    w_4c = (c/dx)*((4/3)*np.sin(k*dx) - (1/6)*np.sin(2*k*dx))
    # Result (psi'_j)
    res = -psi_2p(k, x-dx, w_4c, t)/22 - (12/(11*dx))*psi_2(k, x-dx/2, w_4c, t)␣
↪+ (12/(11*dx))*psi_2(k, x+dx/2, w_4c, t) - psi_2p(k, x+dx, w_4c, t)/22
    return res, w_4c
```

[ ]:
```
' Define domain parameters. '
N = 48 # Number of grid points
L = 1 # Length of domain
x = np.linspace(0, L, N) # Generate stencil
dx = L/N # Derive the grid spacing
dt = 1/dx # Define a time step to render a CFL of 1
c = dx*dt # Calculate CFL
ns = np.arange(0, N//2, 1, dtype=int) # Define wavenumber integers
t = 0 # Sample time of evaluation

# Store wavenumbers and frequencies (exact and approximate)
```

```
ks, w_exact, w_approx, ratio = [], [], [], []
# Iterate over wavenumber integers and extract frequencies
for _, n in enumerate(ns):
    # Define wavenumber
    k = 2*np.pi*n/L
    # Exact solution
    psi = psi_2p(k, x, c*k, t)
    # Approximation
    psi_approx, w_ = approx(c, dx, k, t, x)
    # Storage
    ks.append(k)
    w_exact.append(c*k)
    w_approx.append(w_)

    ratio.append(np.abs(np.real(psi_approx)[N//2])/np.abs(np.real(psi)[N//2]))

' Plotting. '
fig, ax = plt.subplots(figsize=(4, 4))
labelpad = 15
ax.plot(ks, np.array(w_exact)/c, label='Exact')
ax.plot(ks, np.array(w_approx)/c, label='Approximation')
ax_ = ax.twinx()
ax_.plot(ks, ratio, color='tab:green')
ax.set_xlabel('Wavenumber', labelpad=labelpad)
ax.set_ylabel('Scaled frequency', labelpad=labelpad)
ax_.set_ylabel('Ratio (approximation/exact)', rotation=270, labelpad=labelpad+5)

ax.set_xticks([i*np.pi for i in range(0, N-1)])
ax.set_yticks([i*np.pi for i in range(0, N-1)])

ax.set_xticklabels(['{0}/Δ'.format(i) if i % 4 == 0 else '' for i in range(0,␣
 ↪len(ax.get_xticks()))])
ax.set_yticklabels(['{0}/Δ'.format(i) if i % 4 == 0 else '' for i in range(0,␣
 ↪len(ax.get_yticks()))])

ax.tick_params(axis='both', direction='in')
ax.set_xlim([0, max(ks)])
ax.set_ylim([0, max(np.array(w_exact)/c)])
ax.set_aspect('equal')

fig.legend(frameon=False, ncol=2, loc='upper center')
fig.autofmt_xdate()

plt.savefig('figs/p2.png', dpi=300, bbox_inches='tight')

' Plot wavenumbers over domain. '
'''
```

```
fig, ax = plt.subplots(figsize=(4, 4))
markers = ['o', '^', 'x', '1', 's', '+', 'D', 'P']
colors = [plt.cm.get_cmap('Paired')(i) for i in np.linspace(0, 1, len(markers))]
for j, n in enumerate(ns):
    k = 2*np.pi*n/L
    # Scatter plot
    psi = advection(c, x, dx, t, k)
    ax.scatter(x, psi, label='Wavenumber {0}'.format(j), c=colors[j], s=20,␣
 ↪marker=markers[j])
    if j != 0:
        ks.append(k)
        w_exact.append(c*k)
    # Line plot with higher resolution
    x_ = np.linspace(0, L, N*10)
    psi = advection(c, x_, dx, t, k)
    ax.plot(x_, psi, c=colors[j], lw=0.5)
    ax.set_xlim([0, L])

fig.suptitle('{0} points, {1} unique nonzero modes'.format(N, N//2), y=0.95)

fig.legend(frameon=False, bbox_to_anchor=(0.925, 0.925), loc='upper left')
'''
```

Sample plot for a given wavenumber.

[ ]:
```
fig, ax = plt.subplots(figsize=(4, 4))
ax.plot(x, psi, label='Exact')
ax.plot(x, psi_approx, label='Approximate')
ax.set_ylabel('$\psi_j^{(1)}$')
ax.set_xlabel('x')
fig.legend(frameon=False, ncol=2, loc='upper center', bbox_to_anchor=(0.6, 0.
 ↪9), fontsize=10)
ax.set_title('Wavenumber: {0:.3f}'.format(k), fontsize=10, y=1.15)
fig.tight_layout()
plt.savefig('figs/p2_psi.png', dpi=300)
```