Theoretical & Computational Seismology

Contents

This is a solution companion to the book “Theoretical and Computational Seismology” by Jeroen Tromp, Blair Professor of Geology and Professor of Applied & Computational Mathematics.

Theoretical and Computational Seismology is available for purchase at Princeton University Press.

Solutions to problems in Chapters 1-5 were prepared by Jeroen Tromp and are static PDFs available for download.

Solutions to Chapters 6 and 7 were prepared by Will Eaton. Each solution is contained within an individual Jupyter notebook and rendered on this Jupyter Book. A static PDF version of all Chapter 6 and 7 solutions can also be downloaded `here`.

This website was created by Lucas Sawade and Will Eaton. Please report or mistakes any issues via Github or email.

Errata

A list of errata for the textbook can be found here. Please report any mistakes you find via email.

Solutions

1 Kinematics

This chapter introduces topics in Kinematics. A PDF version of solutions is available below:

Chapter 1 solutions

2 Dynamics

This chapter introduces topics in Dynamics. A PDF version of solutions is available below:

Chapter 2 solutions

4 Linearized Equations of Motion

This chapter explores the linearized equations of motion. A PDF version of solutions is available below:

Chapter 4 solutions

5 Anelasticity and Attenuation

This chapter discusses anelasticity and attenuation. A PDF version of solutions is available below:

Chapter 5 solutions

6 Strong Methods

This chapter is about strong methods which directly solve the the PDE in its strong form.

Chapter Solutions

Problem 6.1

We begin with the following Taylor expansions

$$
\begin{aligned}
f(x+\Delta x) =& f(x) + \Delta x \, \frac{df(x)}{dx} + \frac{1}{2!} \left(\Delta x\right)^2 \frac{d^2 f(x)}{dx^2} + \frac{1}{3!} \left(\Delta x\right)^3 \frac{d^3 f(x)}{dx^3} + \frac{1}{4!} \left(\Delta x\right)^4 \frac{d^4}{d} \\
f(x-\Delta x) =& f(x) - \Delta x \, \frac{df(x)}{dx} + \frac{1}{2!} \left(\Delta x\right)^2 \frac{d^2 f(x)}{dx^2} - \frac{1}{3!} \left(\Delta x\right)^3 \frac{d^3 f(x)}{dx^3} + \frac{1}{4!} \left(\Delta x\right)^4 \frac{d^4}{d} \\
f(x+2\Delta x) =& f(x) + 2\Delta x \, \frac{df(x)}{dx} + \frac{1}{2!} \left(2\Delta x\right)^2 \frac{d^2 f(x)}{dx^2} + \frac{1}{3!} \left(2\Delta x\right)^3 \frac{d^3 f(x)}{dx^3} + \frac{1}{4!} \left(2\Delta x\right) \\
f(x-2\Delta x) =& f(x) - 2\Delta x \, \frac{df(x)}{dx} + \frac{1}{2!} \left(2\Delta x\right)^2 \frac{d^2 f(x)}{dx^2} - \frac{1}{3!} \left(2\Delta x\right)^3 \frac{d^3 f(x)}{dx^3} + \frac{1}{4!} \left(2\Delta x\right)
\end{aligned}
$$

where each of these expansions has been truncated at $\mathcal{O}(\Delta x)^5$. Now let us combine these expansions together as follows

$$
\begin{aligned}
f(x-2\Delta x) + 8f(x+\Delta x) - 8f(x-\Delta x) - f(x+2\Delta x) = & \\
f(x) - 2\,\Delta x \, \frac{df(x)}{dx} + 2\left(\Delta x\right)^2 \frac{d^2 f(x)}{dx^2} - \frac{8}{6} \left(\Delta x\right)^3 \frac{d^3 f(x)}{dx^3} + \frac{2}{3} \left(\Delta x\right)^4 \frac{d^4}{d} & \\
+8f(x) + 8\Delta x \, \frac{df(x)}{dx} + 4\left(\Delta x\right)^2 \frac{d^2 f(x)}{dx^2} + \frac{8}{6} \left(\Delta x\right)^3 \frac{d^3 f(x)}{dx^3} + \frac{1}{3} \left(\Delta x\right)^4 \frac{d^4}{d} & \\
-8f(x) + 8\Delta x \, \frac{df(x)}{dx} - 4\left(\Delta x\right)^2 \frac{d^2 f(x)}{dx^2} + \frac{8}{6} \left(\Delta x\right)^3 \frac{d^3 f(x)}{dx^3} - \frac{1}{3} \left(\Delta x\right)^4 \frac{d^4}{d} & \\
-f(x) - 2\Delta x \, \frac{df(x)}{dx} - 2\left(\Delta x\right)^2 \frac{d^2 f(x)}{dx^2} - \frac{8}{6} \left(\Delta x\right)^3 \frac{d^3 f(x)}{dx^3} - \frac{2}{3} \left(\Delta x\right)^4 \frac{d^4}{d} & \\
+\mathcal{O}(\Delta x)^5 \quad . &
\end{aligned}
$$

Evidently all terms on the right-hand side cancel out, except for the $\frac{df(x)}{dx}$ terms, yielding

$$12\,\Delta x\,\frac{df(x)}{dx} + \mathcal{O}(\Delta x)^5 = f(x-2\Delta x) + 8f(x+\Delta x) - 8f(x-\Delta x) - f(x+2\Delta x)$$

and therefore

$$\frac{df(x)}{dx} = \frac{f(x-2\Delta x) + 8f(x+\Delta x) - 8f(x-\Delta x) - f(x+2\Delta x)}{12\,\Delta x} - \mathcal{O}(\Delta x)^4$$

approximates the first derivative of $f(x)$ to 4th-order accuracy, noting that the division by Δx reduces the error from order $\mathcal{O}(\Delta x)^5$ to $\mathcal{O}(\Delta x)^4$.

Similarly, we may consider the Taylor expansions:

$$\begin{aligned} f(x+\frac{3}{2}\Delta x) \;=\; f(x) + \frac{3}{2}\,\Delta x\,\frac{df(x)}{dx} + \frac{1}{2!}\,(\frac{3}{2}\,\Delta x)^2\,\frac{d^2f(x)}{dx^2} + \frac{1}{3!}\,(\frac{3}{2}\,\Delta x)^3\,\frac{d^3f(x)}{dx^3} \\ + \frac{1}{4!}\,(\frac{3}{2}\,\Delta x)^4\,\frac{d^4f(x)}{dx^4} + \mathcal{O}(\Delta x)^5 \end{aligned}$$

$$\begin{aligned} f(x-\frac{3}{2}\Delta x) \;=\; f(x) - \frac{3}{2}\,\Delta x\,\frac{df(x)}{dx} + \frac{1}{2!}\,(\frac{3}{2}\,\Delta x)^2\,\frac{d^2f(x)}{dx^2} - \frac{1}{3!}\,(\frac{3}{2}\,\Delta x)^3\,\frac{d^3f(x)}{dx^3} \\ + \frac{1}{4!}\,(\frac{3}{2}\,\Delta x)^4\,\frac{d^4f(x)}{dx^4} - \mathcal{O}(\Delta x)^5 \end{aligned}$$

$$\begin{aligned} f(x+\frac{1}{2}\,\Delta x) \;=\; f(x) + \frac{1}{2}\Delta x\,\frac{df(x)}{dx} + \frac{1}{2!}\,(\frac{1}{2}\,\Delta x)^2\,\frac{d^2f(x)}{dx^2} + \frac{1}{3!}\,(\frac{1}{2}\,\Delta x)^3\,\frac{d^3f(x)}{dx^3} \\ + \frac{1}{4!}\,(\frac{1}{2}\,\Delta x)^4\,\frac{d^4f(x)}{dx^4} + \mathcal{O}(\Delta x)^5 \end{aligned}$$

$$\begin{aligned} f(x-\frac{1}{2}\,\Delta x) \;=\; f(x) - \frac{1}{2}\,\Delta x\,\frac{df(x)}{dx} + \frac{1}{2!}\,(\frac{1}{2}\,\Delta x)^2\,\frac{d^2f(x)}{dx^2} - \frac{1}{3!}\,(\frac{1}{2}\,\Delta x)^3\,\frac{d^3f(x)}{dx^3} \\ + \frac{1}{4!}\,(\frac{1}{2}\,\Delta x)^4\,\frac{d^4f(x)}{dx^4} - \mathcal{O}(\Delta x)^5 \quad . \end{aligned}$$

Let us combine multiples of these expansions, each up to order $\mathcal{O}(\Delta x)^5$, as follows:

$$
\begin{aligned}
-f(x+\frac{3}{2}\,\Delta x)\,+27f(x+\frac{1}{2}\,\Delta x)\,-\,27f(x-\frac{1}{2}\,\Delta x)\,+\,f(x-\frac{3}{2}\,\Delta x) = \\
-f(x)-\frac{3}{2}\Delta x\,\frac{df(x)}{dx}-\frac{9}{8}\,(\Delta x)^2\,\frac{d^2f(x)}{dx^2}-\frac{9}{16}\,(\Delta x)^3\,\frac{d^3f(x)}{dx^3}-\frac{27}{128}\,(\Delta \\
+27\,f(x)+\frac{27}{2}\,\Delta x\,\frac{df(x)}{dx}+\frac{27}{8}\,(\Delta x)^2\,\frac{d^2f(x)}{dx^2}+\frac{9}{16}\,(\Delta x)^3\,\frac{d^3f(x)}{dx^3}+\frac{9}{128} \\
-27\,f(x)+\frac{27}{2}\,\Delta x\,\frac{df(x)}{dx}-\frac{27}{8}\,(\Delta x)^2\,\frac{d^2f(x)}{dx^2}+\frac{9}{16}\,(\Delta x)^3\,\frac{d^3f(x)}{dx^3}-\frac{9}{128} \\
+f(x)-\frac{3}{2}\,\Delta x\,\frac{df(x)}{dx}+\frac{9}{8}\,(\Delta x)^2\,\frac{d^2f(x)}{dx^2}-\frac{9}{16}\,(\Delta x)^3\,\frac{d^3f(x)}{dx^3}+\frac{27}{128}\,(\Delta \\
+\mathcal{O}(\Delta x)^5
\end{aligned}
$$

Again, we observe that all right-hand-side terms of $f(x)$, $\frac{d^2f(x)}{dx^2}$ and $\frac{d^3f(x)}{dx^3}$ cancel out, leaving only terms of $\frac{df(x)}{dx}$ such that

$$
-f(x+\frac{3}{2}\,\Delta x)+27f(x+\frac{1}{2}\,\Delta x)-27f(x-\frac{1}{2}\,\Delta x)+f(x-\frac{3}{2}\,\Delta x) = 24\Delta x\,\frac{df(x)}{dx}+\mathcal{O}
$$

and so dividing by $24\,\Delta x$ yields, as required

$$
\frac{df(x)}{dx}+\mathcal{O}(\Delta x)^4 = \frac{1}{\Delta x}\left[-\frac{1}{24}f(x+\frac{3}{2}\,\Delta x)+\frac{9}{8}f(x+\frac{1}{2}\,\Delta x)-\frac{9}{8}f(x-\frac{1}{2}\,\Delta x)+\frac{1}{24}f
$$

Problem 6.2

Here we will use the Taylor expansions listed in Problem 6.1. As in Problem 6.1 we combine multiples of these Taylor series, such that

$$
\begin{aligned}
-f(x+2\Delta x) \,+16\, f(x+\Delta x) \,-\, 30\, f(x) \,+\, 16\, f(x-\Delta x) \,-\, f(x-2\Delta x) = \\
&-f(x) - 2\Delta x\, \frac{df(x)}{dx} - 2\,(\Delta x)^2\, \frac{d^2f(x)}{dx^2} - \frac{8}{6}\,(\Delta x)^3\, \frac{d^3f(x)}{dx^3} - \mathcal{O}(\Delta x)^4 \\
&+16f(x) + 16\,\Delta x\, \frac{df(x)}{dx} + 8\,(\Delta x)^2\, \frac{d^2f(x)}{dx^2} + \frac{8}{3}\,(\Delta x)^3\, \frac{d^3f(x)}{dx^3} + \mathcal{O}(\Delta x)^4 \\
&-30f(x) \\
&+16f(x) - 16\,\Delta x\, \frac{df(x)}{dx} + 8\,(\Delta x)^2\, \frac{d^2f(x)}{dx^2} - \frac{8}{3}\,(\Delta x)^3\, \frac{d^3f(x)}{dx^3} + \mathcal{O}(\Delta x)^4 \\
&-f(x) + 2\Delta x\, \frac{df(x)}{dx} - 2\,(\Delta x)^2\, \frac{d^2f(x)}{dx^2} + \frac{8}{6}\,(\Delta x)^3\, \frac{d^3f(x)}{dx^3} - \mathcal{O}(\Delta x)^4 \quad ,
\end{aligned}
$$

which simplifies to

$$
-f(x+2\Delta x) + 16f(x+\Delta x) - 30f(x) + 16f(x-\Delta x) - f(x-2\Delta x) = 12(\Delta x)^2\, \frac{d^2f(x)}{dx^2}
$$

and therefore dividing by $12\,(\Delta x)^2$ yields

$$
\begin{aligned}
\frac{d^2f(x)}{dx^2} = \frac{1}{(\Delta x)^2}\Big[&-\frac{1}{12}\, f(x+2\Delta x) + \frac{4}{3}\, f(x+\Delta x) - \frac{5}{2}\, f(x) + \frac{4}{3}\, f(x-\Delta x) - \frac{1}{12}\, f(x \\
&+ \frac{\mathcal{O}(\Delta x)^4}{(\Delta x)^2} \quad .
\end{aligned}
$$

It is important to note here that due to the division by $(\Delta x)^2$, the error is of order $\frac{\mathcal{O}(\Delta x)^4}{(\Delta x)^2} = \mathcal{O}(\Delta x)^2$.

Problem 6.3

The discrete version of the homogeneous wave equation

$$
\partial_t^2 s = \beta^2\, \partial_x^2 s
$$

is given in Eqn. 6.20 as

$$s_j^{n+1} = 2s_j^n - s_j^{n-1} + \frac{\beta^2(\Delta t)^2}{(\Delta x)^2}\left(s_{j+1}^n - 2s_j^n + s_{j-1}^n\right)$$

First let us define some parameters:

```
import numpy as np
import matplotlib.pyplot as plt
from copy import copy

L     = 100     # length of domain
dx    = 0.1     # spatial discretiation
beta  = 1       # constant wavespeed
```

Next lets define a function that sets the displacement initial condition

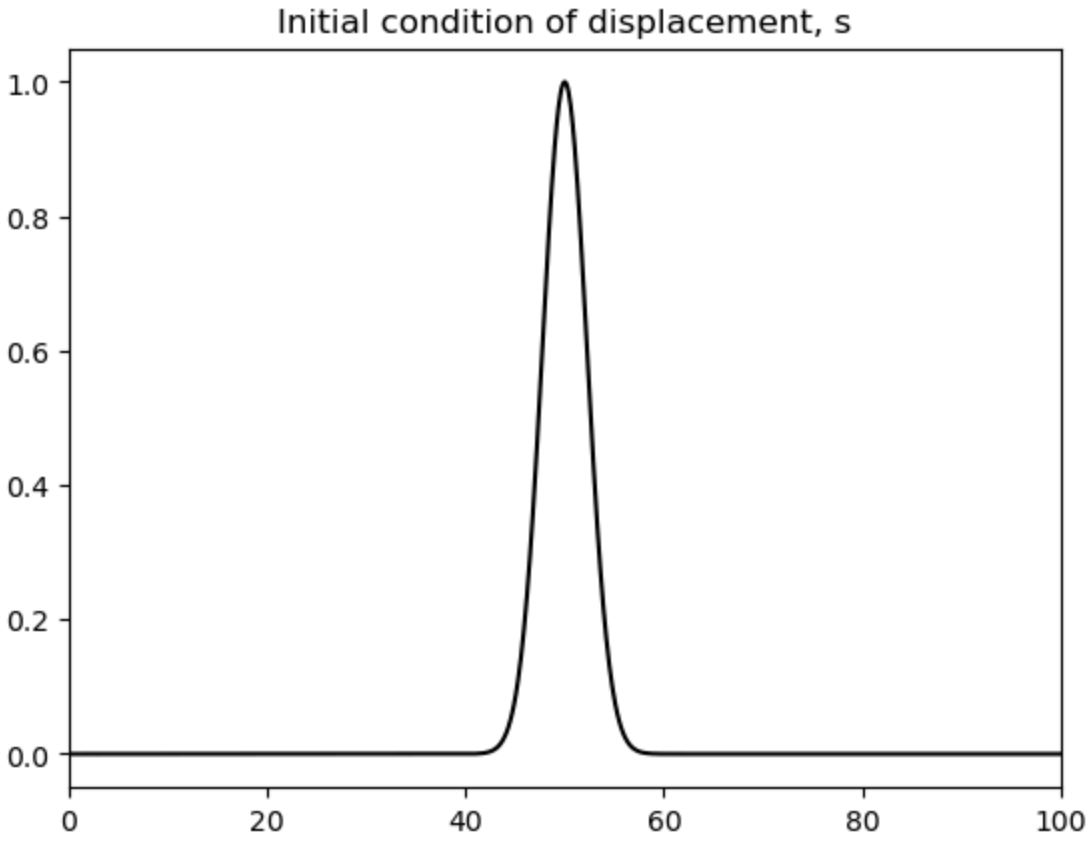
```
# Initial condition:
def apply_initial_condition(x, s):
    """
    Applies initial condition defined in Eqn 6.21
    x   1D numpy array of spatial positions (array x)
    s   1D numpy array of displacement variable (array s)
    """
    s = np.exp(-0.1 * (x-50)**2)
    return s
```

We can create our grid of points, x, using a numpy array. Note that using L as our upper bound in the arange function produces an array of values up to L - dx, hence we use L+dx so that the last element of the array has value L:

```
# Create array x
x = np.arange(0, L+dx, dx)

# Length of array x
N = len(x)
```

Next we create an empty numpy array to hold the displacement values, with the same length as x:

```
s      = np.zeros(N)
s_prev = np.zeros(N)

# Apply initial condition to s:
s = apply_initial_condition(x,s)

# Plot initial condition
fig, ax = plt.subplots()
ax.set_title('Initial condition of displacement, s')
ax.plot(x,s, 'k');
ax.set_xlim([0,100]);
```



Timestepping

We can define a function that computes the displacement at timestep $n+1$ based on equation 6.20:

$$s_j^{n+1} = 2s_j^n - s_j^{n-1} + \frac{\beta^2(\Delta t)^2}{(\Delta x)^2}\left(s_{j+1}^n - 2s_j^n + s_{j-1}^n\right)$$

Note here that problem 6.3 requires us to investigate the effect of varying a, where

$$\Delta t = \alpha \Delta x / \beta$$

which we can rearrange to

$$\alpha = \beta \Delta t / \Delta x$$

such that we can write the timestepping condition equation in terms of α as

$$s_j^{n+1} = 2s_j^n - s_j^{n-1} + \alpha^2 \left(s_{j+1}^n - 2s_j^n + s_{j-1}^n \right)$$

```
def step_in_time(s, s_prev, alpha, use_Dirichlet_bc):
    '''
    Function computes one timestep of s and applies the boundary
    condition.
    Inputs:
    s
       1D np array of displacement variable at timestep n
    s_prev
       1D np array of displacement at timestep n-1
    alpha
       float value defined in problem
    use_Dirichlet_bc
       boolean value, if True then applies Dirichlet bc
                      if False then applies Neumann bc

    Outputs:
    s_new
       updated displacement at n+1
    s
       displacement at time n which will be used as 's_prev in next timestep'
    '''
    # Compute the first two terms of the equation
    # this applies to all elements in the array
    s_new = 2*s - s_prev

    # Now we add the 3rd term, which affects all elements of the
    # array except for the first and last elements (boundaries)
    s_new[1:-1] += (alpha**2)*( s[2:] - 2*s[1:-1] + s[:-2])


    # Apply boundary condition
    if use_Dirichlet_bc:
        s_new[0]  = 0
        s_new[-1] = 0
    else:
        # Neumann boundary condition
        s_new[0]  = s_new[1]
        s_new[-1] = s_new[-2]
    return s_new, s
```

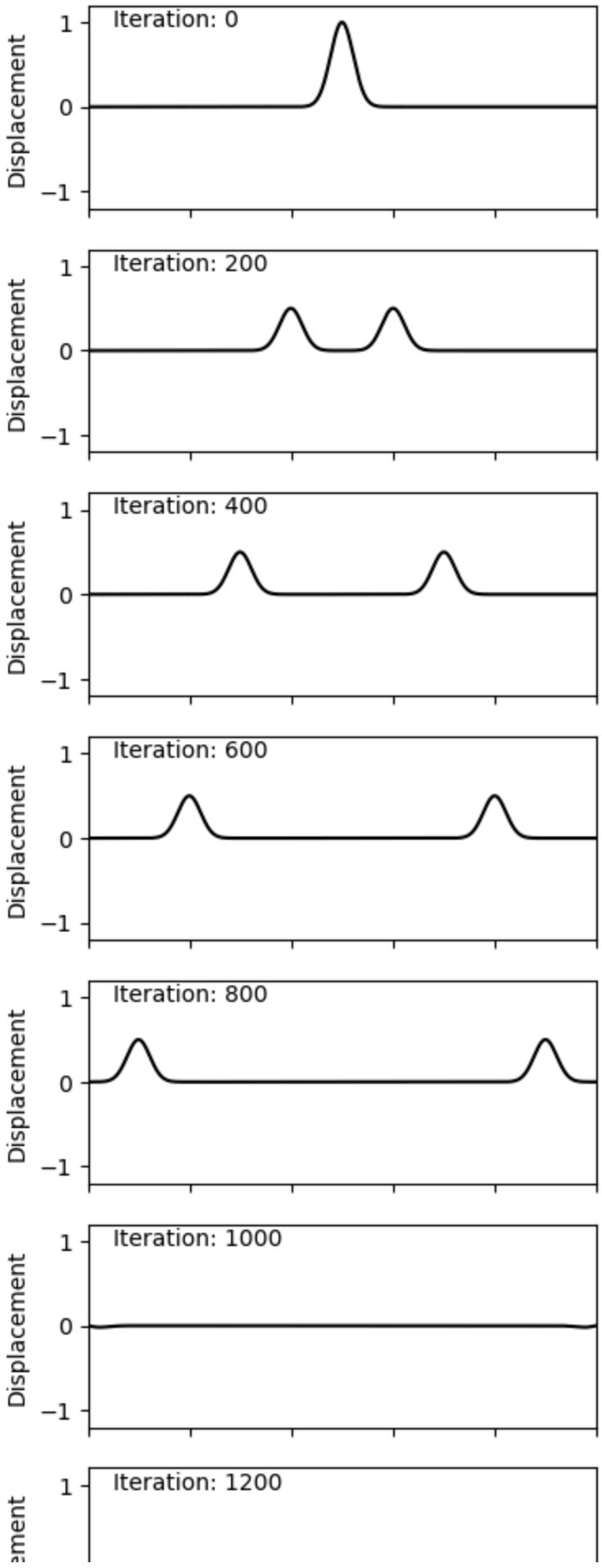
Dealing with previous timestep?

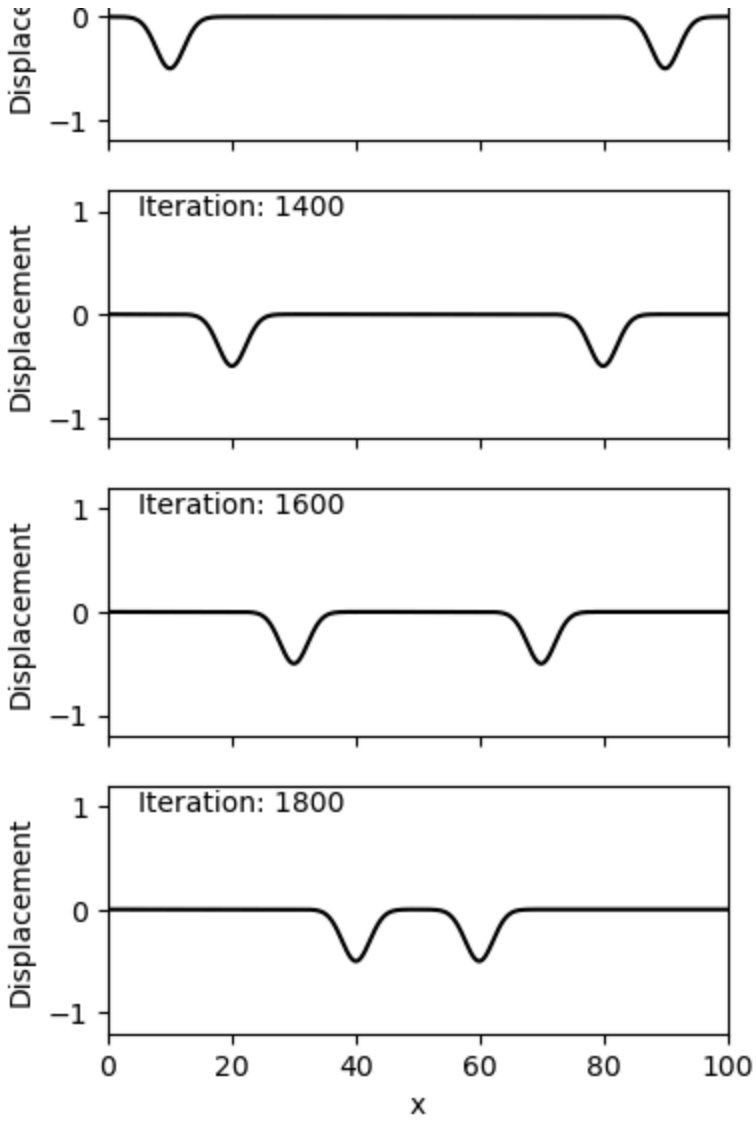
At time = 0, what should be in our 'previous timestep' array of displacement? Lets just assume that the system was in the same state as at the initial condition, so `s_prev = s`.

Dirichlet BC - $\alpha = 0.5$

Lets start with $\alpha = 0.5$ and see what happens. To see how the system evolves, we will loop over 2000 timesteps, and plot the system at every 200th iteration

```
# To begin with let us assume that the initial condition holds for any time before (
# is the same as s
# Lets also ensure our boundary condition is applied
s = apply_initial_condition(x,s)
s_prev = s

# Simulation parameters:
alpha         = 0.5
use_dirichlet = True
ntimesteps    = 2000

# Plotting
nplots = 10
fig, ax = plt.subplots(nplots, figsize=(4,19),
                       sharex=True, sharey=True)
iax = 0

# Loop over the timesteps
for i in range(ntimesteps):
    s, s_prev = step_in_time(s, s_prev, alpha,
                             use_Dirichlet_bc=use_dirichlet)

    # Plot every 200th timestep
    if i%200 == 0:
        # Plot the wave
        ax[iax].plot(x, s, 'k')

        # Add some annotations
        ax[iax].text(x=5, y = 0.95, s=f"Iteration: {i}")
        # Set plot limits for each axis:
        ax[iax].set_xlim([0,100])
        ax[iax].set_ylim([-1.2,1.2])
        ax[iax].set_ylabel("Displacement")
        iax +=1

ax[-1].set_xlabel("x");
```


Iteration: 0
Iteration: 200
Iteration: 400
Iteration: 600
Iteration: 800
Iteration: 1000
Iteration: 1200
Displacement
1
0
−1



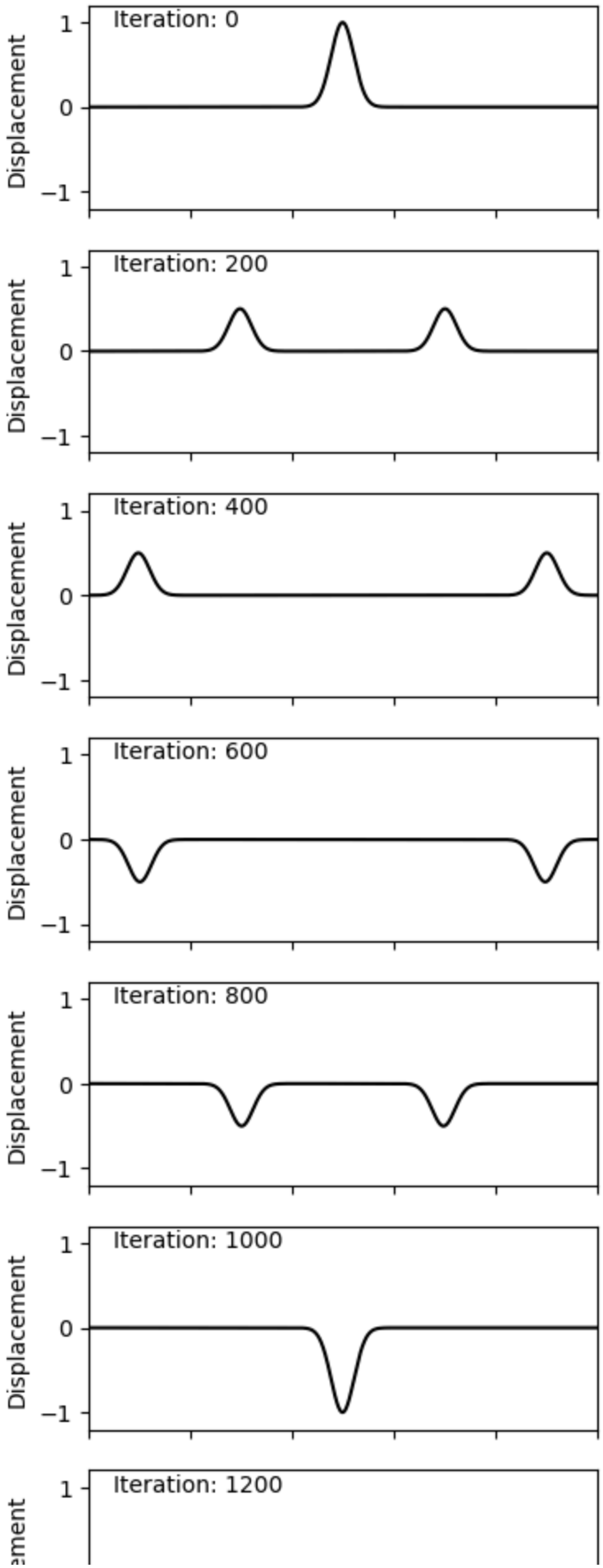
We notice a few key things:

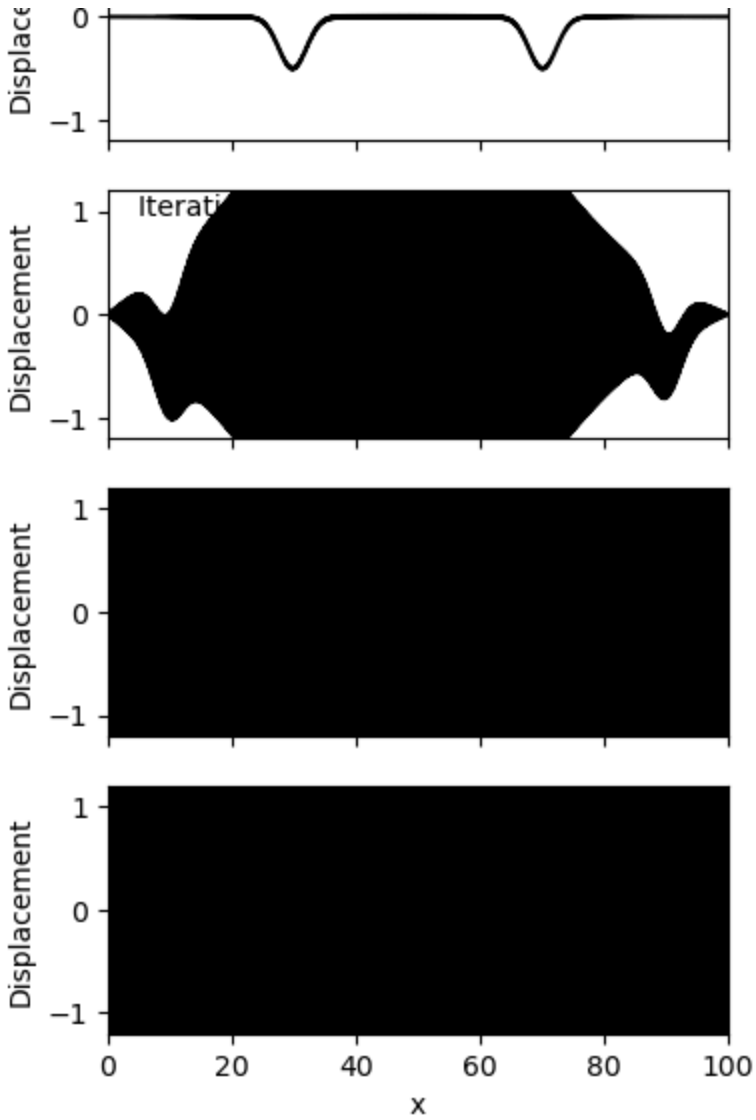
(1) The reflection at the boundary causes a change in the polarity of the waves
(2) The wavefield is smooth and well resolved
(3) Energy is well-conserved within the system

If you try to run this code with $\alpha = 1.5$ you will get a runtime error due to numerical overflow. This is because the system is unstable and the values are growing through time.

You should find that $\alpha = 1$ is the cutoff for this behaviour. In the code below, you can try experimenting with the precision of the alpha being used. For example, what if alpha is 1.0001 or 1.00001? How many timesteps occur before the system blows up?

```
# To begin with let us assume that the initial condition holds for any time before (
# is the same as s
# Lets also ensure our boundary condition is applied
s = apply_initial_condition(x,s)
s_prev = s

# Simulation parameters:
alpha         = 1.0001
use_dirichlet = True
ntimesteps    = 2000

# Plotting
nplots = 10
fig, ax = plt.subplots(nplots, figsize=(4,19),
                       sharex=True, sharey=True)
iax = 0

# Loop over the timesteps
for i in range(ntimesteps):
    s, s_prev = step_in_time(s, s_prev, alpha,
                             use_Dirichlet_bc=use_dirichlet)

    # Plot every 200th timestep
    if i%200 == 0:
        # Plot the wave
        ax[iax].plot(x, s, 'k')

        # Add some annotations
        ax[iax].text(x=5, y = 0.95, s=f"Iteration: {i}")
        # Set plot limits for each axis:
        ax[iax].set_xlim([0,100])
        ax[iax].set_ylim([-1.2,1.2])
        ax[iax].set_ylabel("Displacement")

        iax +=1


ax[-1].set_xlabel("x");
```


Iteration: 0
Iteration: 200
Iteration: 400
Iteration: 600
Iteration: 800
Iteration: 1000
Iteration: 1200
Displacement
1
0
−1

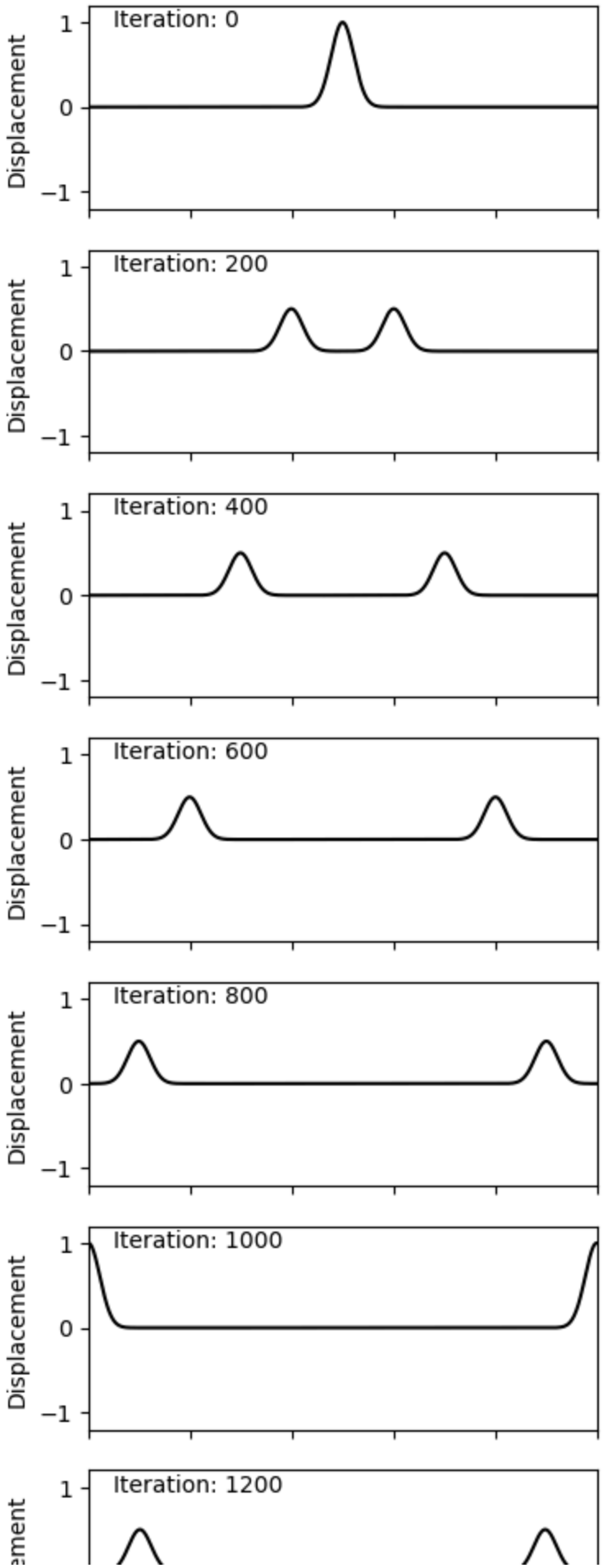


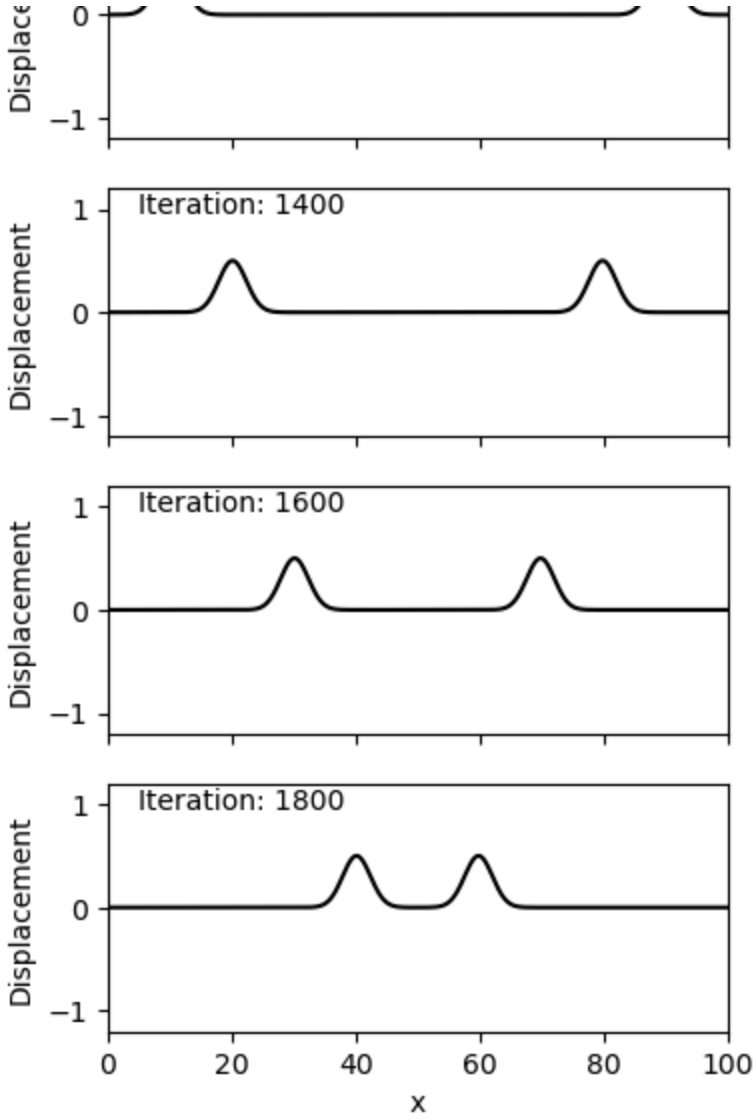
Neumann BC - $\alpha = 0.5$

Lets now look at with $\alpha = 0.5$ for Neumann boundary condition.

The major observed difference is the change in polarity observed above with the Dirichlet condition is no longer observed. Instead of the end point being ‘fixed’ as above, the end point is ‘free’ to move around, as observed at iteration 1000. Note that in the Dirichlet condition the end points never have a non-zero displacement (as is the nature of their boundary condition).

```
# To begin with let us assume that the initial condition holds for any time before (
# is the same as s
# Lets also ensure our boundary condition is applied
s = apply_initial_condition(x,s)
s_prev = s

# Simulation parameters:
alpha         = 0.5
use_dirichlet = False
ntimesteps    = 2000

# Plotting
nplots = 10
fig, ax = plt.subplots(nplots, figsize=(4,19),
                       sharex=True, sharey=True)
iax = 0

# Loop over the timesteps
for i in range(ntimesteps):
    s, s_prev = step_in_time(s, s_prev, alpha, use_Dirichlet_bc=use_dirichlet)

    # Plot every 200th timestep
    if i%200 == 0:

        # Plot the wave
        ax[iax].plot(x, s, 'k')

        # Add some annotations
        ax[iax].text(x=5, y = 0.95, s=f"Iteration: {i}")
        # Set plot limits for each axis:
        ax[iax].set_xlim([0,100])
        ax[iax].set_ylim([-1.2,1.2])
        ax[iax].set_ylabel("Displacement")
        iax +=1

ax[-1].set_xlabel("x");
```


Iteration: 0
Iteration: 200
Iteration: 400
Iteration: 600
Iteration: 800
Iteration: 1000
Iteration: 1200
Displacement
1
0
−1



Problem 6.4

Let us define our domain:

```
import numpy as np
import matplotlib.pyplot as plt
from copy import copy

# Domain parameters
L  = 100
dx = 0.1

# 1D Array holding grid points, x
x = np.arange(0, L+dx, dx)

# Number of grid points in x
N = len(x)

# Velocity (v) and stress (sigma)
v     = np.zeros(N)
sigma = np.zeros(N)

v_old = np.zeros(N)
s_old = np.zeros(N)

v_new = np.zeros(N)
s_new = np.zeros(N)

# Material properties - this is an array in so
# properties can be defined at each point in the grid
rho = np.zeros(N) + 1
mu  = np.zeros(N) + 1
```

CFL Condition

We can use the CFL condition to determine our timestep. If we choose a Courant number of 0.5, this will be stable and set our timestep:

```
beta = np.sqrt(np.max(mu)/np.min(rho))
C = 0.5
dt = dx * C / beta
```

Define the initial condition as per the problem:

```
def apply_initial_condition(x, v, sigma):
    # Equation 6.44
    sigma[:] = 0
    v = -0.2 * (x-50) * np.exp(-0.1*(x-50)**2)
    return v, sigma
```

Solver:

The point of these methods is that they are flexible to whatever medium the wavefield is being simulated in. That is, the scheme is independent of whether the domain is homogeneous (part A of the problem) or heterogeneous (part B). Hence, we want to define a function that solves for the next timestep, which is relatively generic.

At each grid point, the solver computes the new velocity and stress based on values from adjacent grid points. This can be written efficiently using arrays. Let us store the stress and velocity at each grid point in 1D arrays. The stress in the first grid point ($j = 1$) will be stored in σ[0], and the stress in the second grid point in $\sigma[1]$ and so on…. Note here at Python indexing starts at 0 (like most programming languages, e.g. C++, Rust etc, while some others such as MATLAB start at 1!). If there are N grid points then the final stress value will be stored in $\sigma[N-1]$.

Let us now consider the first point (away from the boundary) that we could apply our solver to, at the second ($j = 2$) grid point. To update the velocity we know this would be:

$$v_{j=2}^{n+1} = v_{j=2}^{n-1} + \frac{\Delta t}{\Delta x\, \rho_{j=2}} \left(\sigma_{j=3}^{n} - \sigma_{j=1}^{n}\right)$$

For now let us assume that we have access to the term $v_{j=2}^{n-1}$ which will be stored in a separate array, as well as the simulation parameters Δt, Δx. The important part to note is that we need access to the term $\sigma_{j=3}^{n} - \sigma_{j=1}^{n}$. Similarly, for the following j values we need access to:

$$\begin{aligned}
j = 2: &\quad \sigma_{j=3}^{n} - \sigma_{j=1}^{n} \\
j = 3: &\quad \sigma_{j=4}^{n} - \sigma_{j=2}^{n} \\
j = 4: &\quad \sigma_{j=5}^{n} - \sigma_{j=3}^{n} \\
j = 5: &\quad \sigma_{j=6}^{n} - \sigma_{j=4}^{n}
\end{aligned}$$

We may compute all of these at the same time, if we take two sub-arrays of σ and subtract one from the other. The first array starts at the 3rd grid point ($j = 3$) and represents the σ_{j+1}^{n} values, while the second array starts at j=1 and represnts the σ_{j+1}^{n} values.

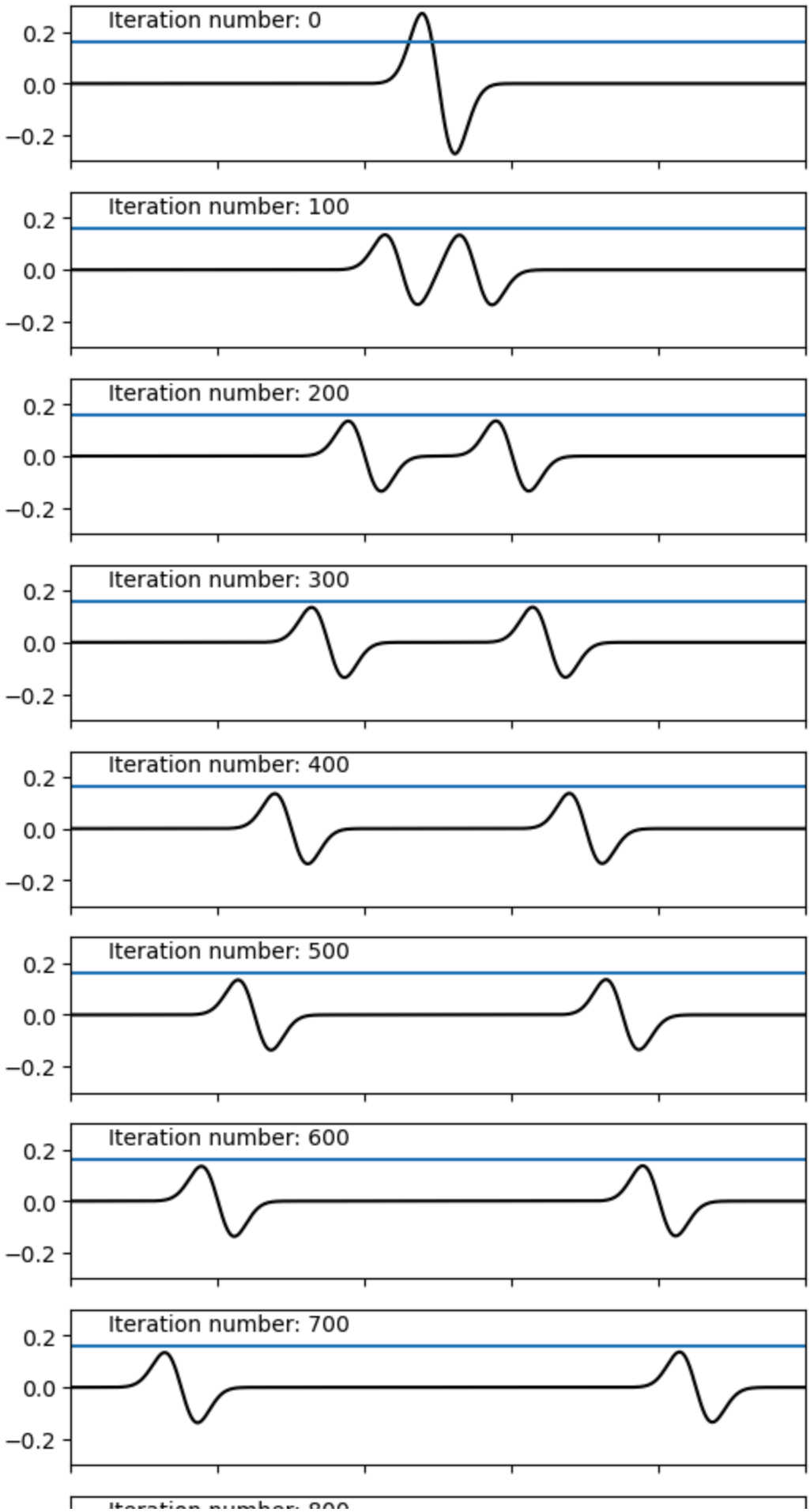
The point here is that for EACH point (excluding the boundary terms) we need to compute the difference of the terms either side of it, which can be done as an array subtraction, rather than in a loop where the computation must be done for each individual grid point in serial.

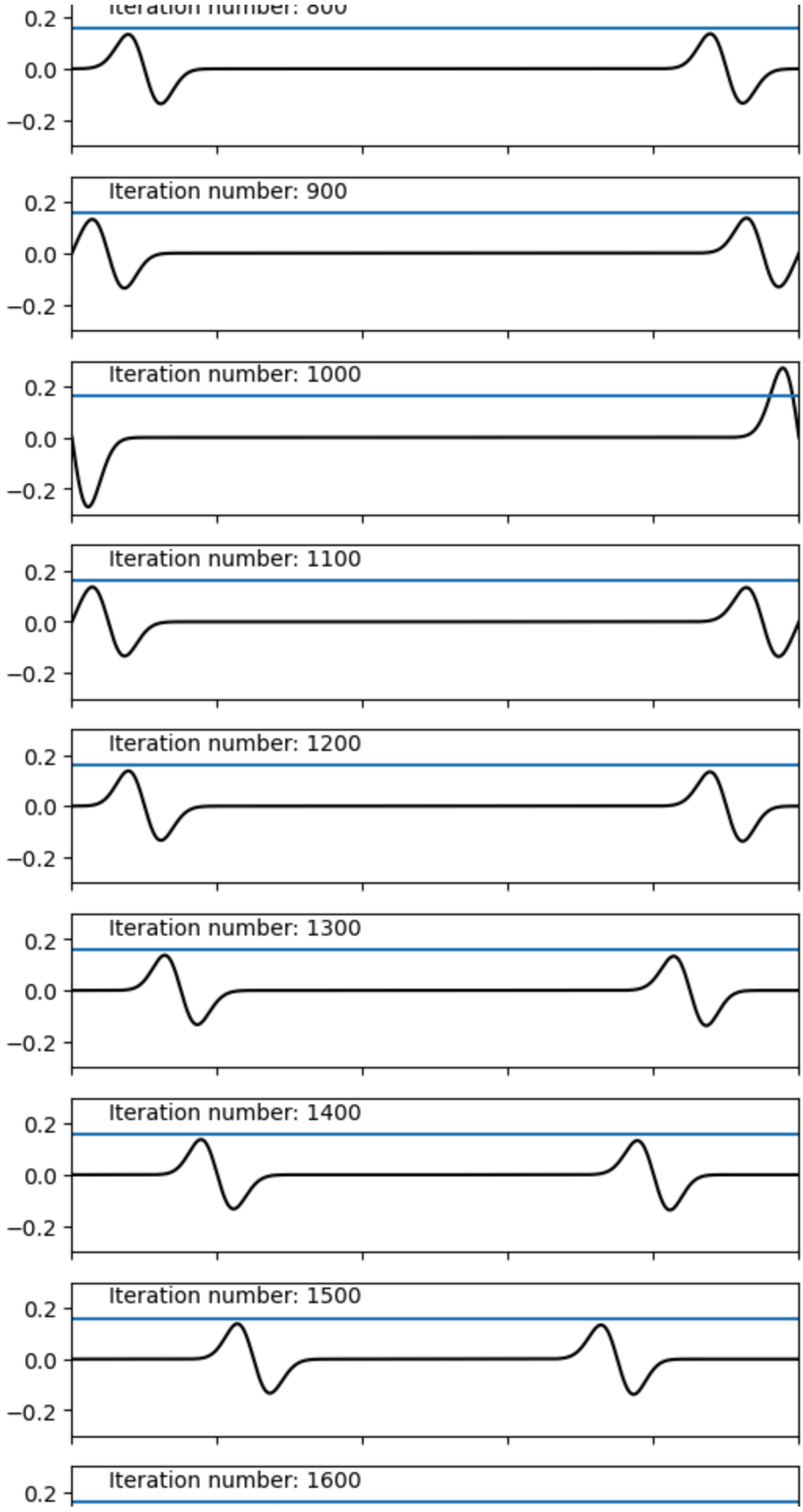
To do this we use slicing, including negative slicing. Read more about this here.

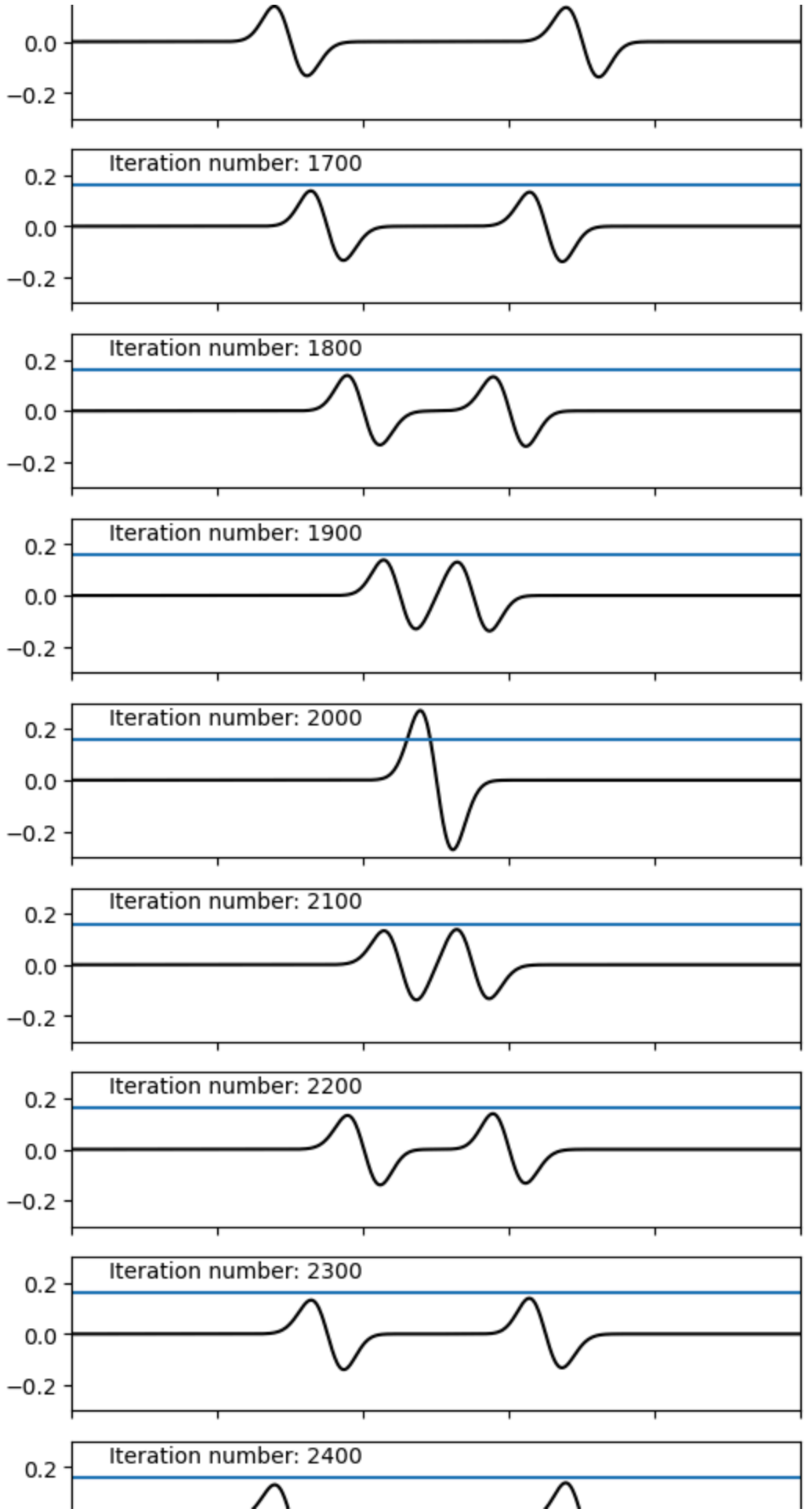
```
def step_in_time(s_old, s, s_new, v_old, v, v_new, dx, dt, mu, rho, use_Dirichlet_BC
    # Notes on slicing:
    # [1:-1] includes all points except boundaries
    # [2:  ] is from 3rd to end
    # [:-2 ] is from start to 3rd-last (missing last two points)

    # [1:-1] represents lower index = j
    # [2:  ] represents lower index = j + 1
    # [:-2 ] represents lower index = j - 1

    # Variable names (s and v):
    # s for sigma, v for v (velocity)
    # ? below means s or v
    # ?_new  represents superscript = n + 1
    # ?      represents superscript = n
    # ?_old  represents superscript = n - 1

    # Compute new v and s
    v_new[1:-1] = v_old[1:-1] + (dt / (rho[1:-1] * dx)) * (s[2:] - s[:-2])
    s_new[1:-1] = s_old[1:-1] + (dt * mu[1:-1] / dx) * (v[2:] - v[:-2])

    # Apply boundary conditions:
    if use_Dirichlet_BC:
        # Dirichlet:
        v_new[0] = 0
        v_new[-1] = 0

        # Use a 1-sided gradient approximation for the LHS edge:
        # sigma^(n+1)_j = sig^n_j + 2 dt mu_j  (v_1 - v_0) /dx
        # . where we know that v_0 is going to be 0
        # sigma^(n+1)_j = sig^n_j + 2 dt mu_j  (v_1 ) /dx
        s_new[0]  = s_old[0]  + 2 * dt * mu[0]  * v[1] / dx
        s_new[-1] = s_old[-1] + 2 * dt * mu[-1] * -v[-2] / dx
    else:
        s_new[0] = 0
        s_new[-1] = 0

        v_new[0] = v_old[0] + 2 * dt * s[1] / (dx * rho[0])
        v_new[-1] = v_old[-1] - 2 * dt * s[-2] / (dx * rho[-1])

    return v, v_new, s, s_new
```

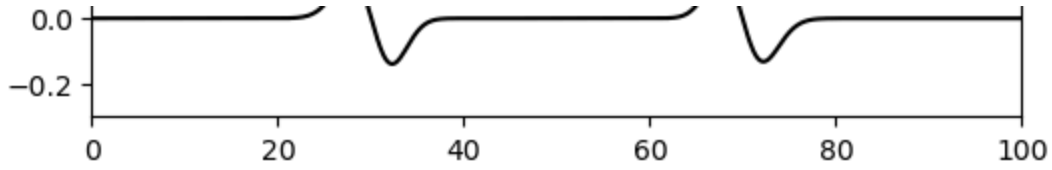
Now we have defined a function to compute each timestep, let us finally run our simulation:

```
# Apply initial conditions
v, sigma     = apply_initial_condition(x, v, sigma)
v_old, s_old = apply_initial_condition(x, v_old, s_old)


# Number of timesteps and steps to plot
nsteps     = 2500
plot_every = 100
nplots     = nsteps//plot_every


# Create Figure
fig, ax = plt.subplots(nplots, figsize=(6,38), sharex=True, sharey=True)
iax = 0

# Iterate through timesteps
for i in range(nsteps):
    v, v_new, sigma, s_new = step_in_time(s_old=s_old, s=sigma, s_new=s_new,
                                          v_old=v_old, v=v, v_new=v_new,
                                          dx=dx, dt=dt, mu=mu, rho=rho,
                                          use_Dirichlet_BC=True)
    # Update timestep
    v_old = copy(v)
    s_old = copy(sigma)
    v     = copy(v_new)
    sigma = copy(s_new)

    if i%plot_every==0:
        ax[iax].plot(x, v, 'k')

        ax[iax].text(x=5, y = 0.22, s=f"Iteration number: {i}")
        ax[iax].set_xlim([0,100])
        ax[iax].set_ylim([-0.3,0.3])

        ax[iax].axhline(0.16)

        iax += 1
```


Iteration number: 0
Iteration number: 100
Iteration number: 200
Iteration number: 300
Iteration number: 400
Iteration number: 500
Iteration number: 600
Iteration number: 700
0.2
0.0
−0.2


Iteration number: 800
0.2
0.0
−0.2
Iteration number: 900
0.2
0.0
−0.2
Iteration number: 1000
0.2
0.0
−0.2
Iteration number: 1100
0.2
0.0
−0.2
Iteration number: 1200
0.2
0.0
−0.2
Iteration number: 1300
0.2
0.0
−0.2
Iteration number: 1400
0.2
0.0
−0.2
Iteration number: 1500
0.2
0.0
−0.2
Iteration number: 1600
0.2


0.0
−0.2
Iteration number: 1700
0.2
0.0
−0.2
Iteration number: 1800
0.2
0.0
−0.2
Iteration number: 1900
0.2
0.0
−0.2
Iteration number: 2000
0.2
0.0
−0.2
Iteration number: 2100
0.2
0.0
−0.2
Iteration number: 2200
0.2
0.0
−0.2
Iteration number: 2300
0.2
0.0
−0.2
Iteration number: 2400
0.2



Inhomogeneous Domain

Let us now consider the case of an inhomogeneous domain in which the shear modulus changes from $\mu = 1$ for $x \in [0,\ 65]$ to $\mu = 4$ for $x \in [65,\ 100]$.

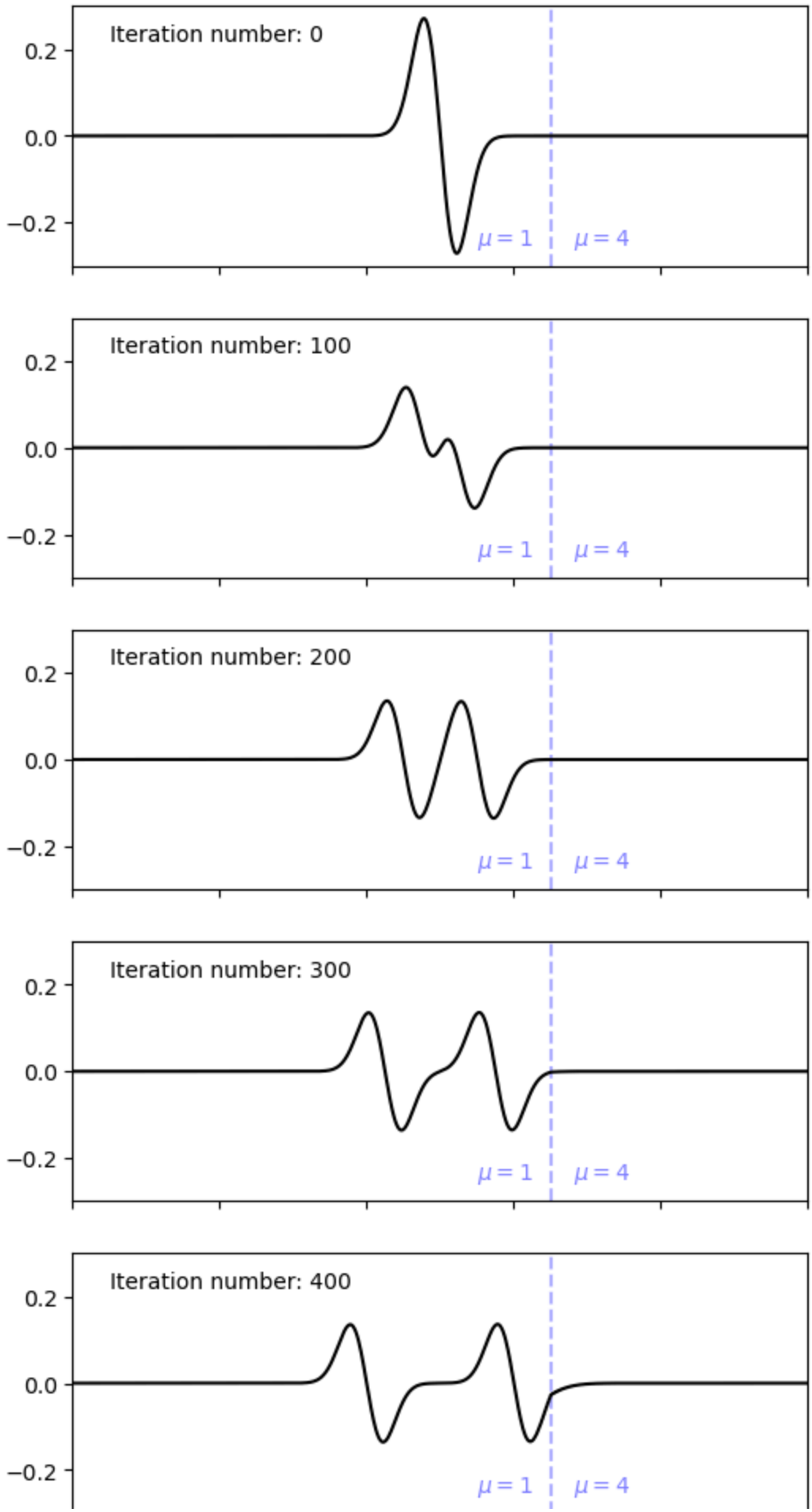
```
# Velocity (v) and stress (sigma)
v     = np.zeros(N)
sigma = np.zeros(N)

v_old = np.zeros(N)
s_old = np.zeros(N)

v_new = np.zeros(N)
s_new = np.zeros(N)

# Material properties - this is an array in so
# properties can be defined at each point in the grid
rho = np.zeros(N) + 1
mu  = np.zeros(N) + 1

# For the region where x is greater than 65 we will now edit the shear values:
mu[x > 65] = 4
```

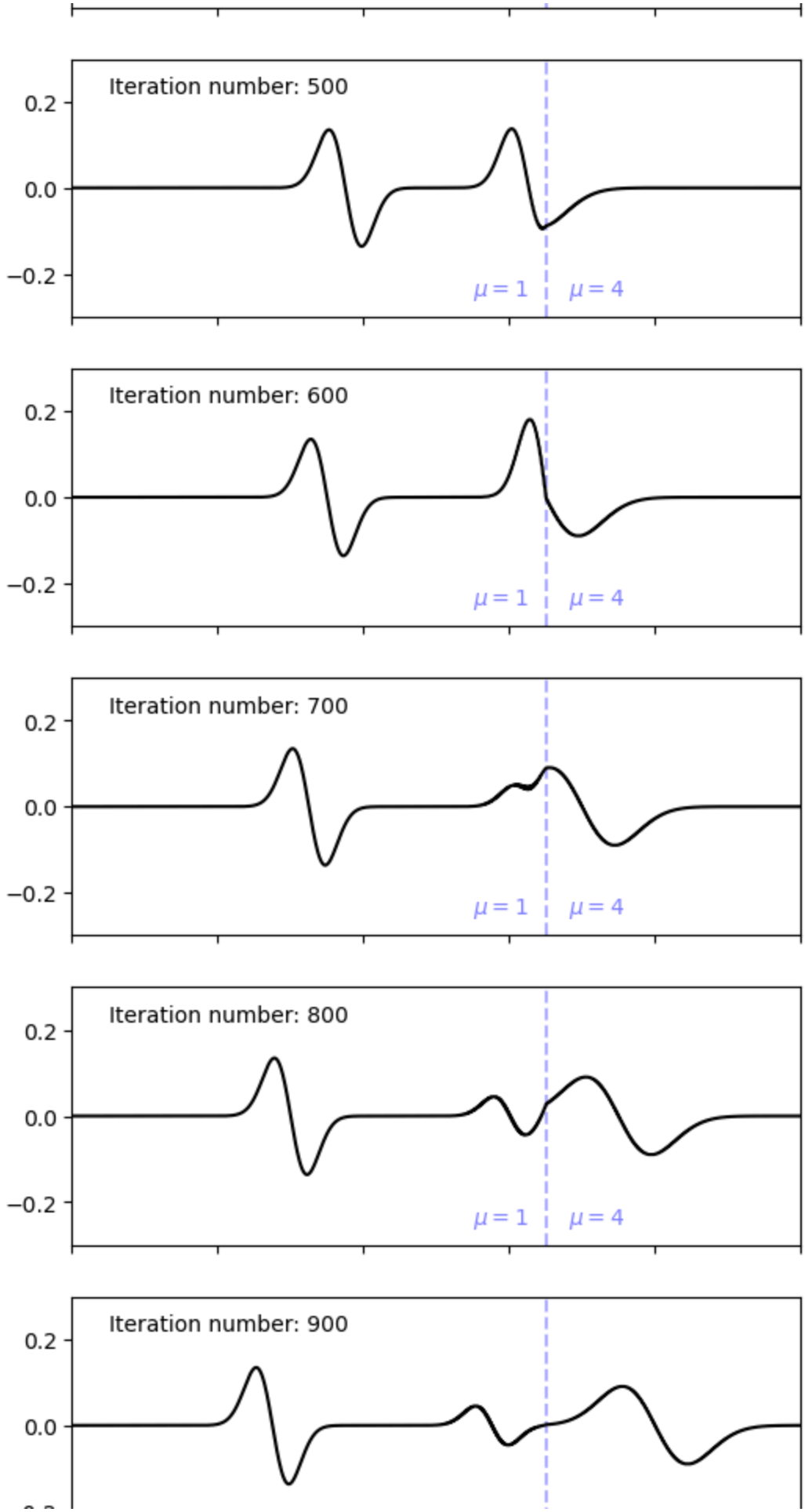
Since we have changed our domain, we need to update our CFL condition and timestep

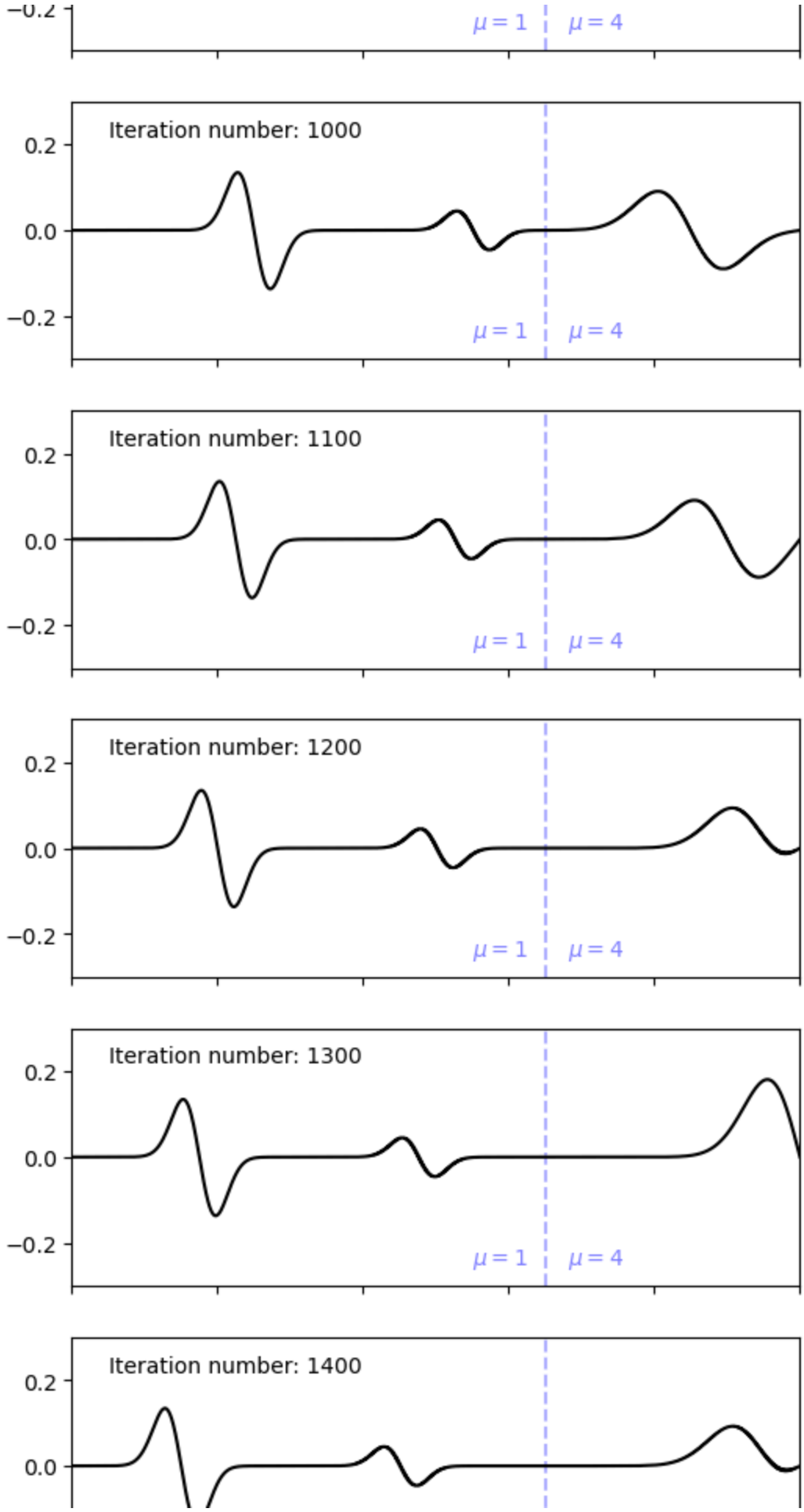
```
# We also need to update our CFL condition:
beta = np.sqrt(np.max(mu)/np.min(rho))
C = 0.5
dt = dx * C / beta
```

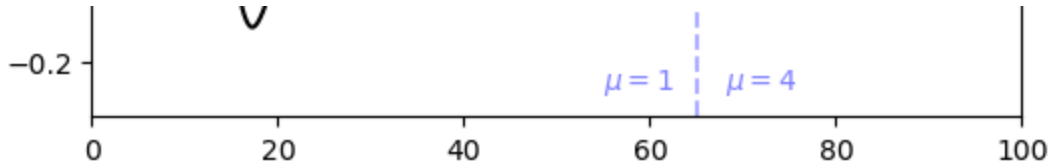
Now let us re-run the simulation and see how the waves propagate. In this case, we will observe a reflected phase at the material interface ($x = 65$) every time a wave interacts with it.

```
# Apply initial conditions
v, sigma     = apply_initial_condition(x, v, sigma)
v_old, s_old = apply_initial_condition(x, v_old, s_old)


# Number of timesteps and steps to plot
nsteps     = 1500
plot_every = 100
nplots     = nsteps//plot_every

# Create Figure
fig, ax = plt.subplots(nplots, figsize=(6,38), sharex=True, sharey=True)
iax = 0

# Iterate through timesteps
for i in range(nsteps):
    v, v_new, sigma, s_new = step_in_time(s_old=s_old, s=sigma, s_new=s_new,
                                          v_old=v_old, v=v, v_new=v_new,
                                          dx=dx, dt=dt, mu=mu, rho=rho,
                                          use_Dirichlet_BC=True)
    # Update timestep
    v_old = copy(v)
    s_old = copy(sigma)
    v     = copy(v_new)
    sigma = copy(s_new)

    if i%plot_every==0:

        # Plot a line at x=65 and add annotation
        ax[iax].axvline(65, alpha=0.3, linestyle='--', color='blue')
        ax[iax].text(x=55, y = -0.25, s=r"$\mu = 1$", color='blue', alpha=0.5)
        ax[iax].text(x=68, y = -0.25, s=r"$\mu = 4$", color='blue', alpha=0.5)

        ax[iax].plot(x, v, 'k')

        ax[iax].text(x=5, y = 0.22, s=f"Iteration number: {i}")
        ax[iax].set_xlim([0,100])
        ax[iax].set_ylim([-0.3,0.3])


        iax += 1
```


Iteration number: 0
Iteration number: 100
Iteration number: 200
Iteration number: 300
Iteration number: 400
0.2
0.0
−0.2
$\mu = 1$
$\mu = 4$


Iteration number: 500
Iteration number: 600
Iteration number: 700
Iteration number: 800
Iteration number: 900
0.2
0.0
−0.2
$\mu = 1$
$\mu = 4$


Iteration number: 1000
Iteration number: 1100
Iteration number: 1200
Iteration number: 1300
Iteration number: 1400
$\mu = 1$
$\mu = 4$
0.2
0.0
−0.2



Problem 6.5

Let us start with the scheme given in Eqns 6.46-6.47:

$$v_j^{n+1} = \frac{1}{2}(v_{j+1}^n + v_{j-1}^n) + \frac{\Delta t}{\rho_j}\frac{\sigma_{j+1}^n - \sigma_{j-1}^n}{2\Delta x}$$
$$\sigma_j^{n+1} = \frac{1}{2}(\sigma_{j+1}^n + \sigma_{j-1}^n) + \Delta t\,\mu_j\frac{v_{j+1}^n - v_{j-1}^n}{2\Delta x}$$

Now we assume a periodic solution as in 6.34:

$$v_j^n = v_0 A^n e^{ikj\Delta x}$$
$$\sigma_j^n = \sigma_0 A^n e^{ikj\Delta x}$$

We can write these conditions as

$$\begin{aligned} v_0 A^{n+1} e^{ikj\Delta x} &= \frac{1}{2}(v_0 A^n e^{ik(j+1)\Delta x} + v_0 A^n e^{ik(j-1)\Delta x}) \\ &\quad + \frac{\Delta t}{\rho_j}\frac{\sigma_0 A^n e^{ik(j+1)\Delta x} - \sigma_0 A^n e^{ik(j-1)\Delta x}}{2\Delta x} \\ \sigma_0 A^{n+1} e^{ikj\Delta x} &= \frac{1}{2}(\sigma_0 A^n e^{ik(j+1)\Delta x} + \sigma_0 A^n e^{ik(j-1)\Delta x}) \\ &\quad + \Delta t\,\mu_j\frac{v_0 A^n e^{ik(j+1)\Delta x} - v_0 A^n e^{ik(j-1)\Delta x}}{2\Delta x} \end{aligned}$$

Let us divide by $A^n\, e^{ikj\Delta x}$

$$v_0 A = \frac{1}{2}(v_0 e^{ik\Delta x} + v_0 e^{-ik\Delta x}) + \frac{\Delta t}{\rho_j}\frac{\sigma_0 e^{ik\Delta x} - \sigma_0 e^{-ik\Delta x}}{2\Delta x}$$
$$\sigma_0 A = \frac{1}{2}(\sigma_0 e^{ik\Delta x} + \sigma_0 e^{-ik\Delta x}) + \Delta t\,\mu_j\frac{v_0 e^{ik\Delta x} - v_0 e^{-ik\Delta x}}{2\Delta x}$$

We can then use the identity that

$$\frac{1}{2}\left(e^{i\phi}+e^{-i\phi}\right)=\cos(\phi)$$
$$\frac{1}{2}\left(e^{i\phi}-e^{-i\phi}\right)=i\,\sin(\phi)$$

to simplify this to

$$v_0 A = v_0\cos(k\Delta x)+\frac{i\Delta t}{\rho_j\,\Delta x}\sigma_0\sin(k\Delta x)$$
$$\sigma_0 A = \sigma_0\cos(k\Delta x)+\frac{i\Delta t\mu_j}{\Delta x}v_0\sin(k\Delta x)$$

This set of linear equations may be written as, for a given $\mu_j=\mu$ and $\rho_j=\rho$:

$$\begin{bmatrix}\cos(k\Delta x)-A & \frac{i\Delta t}{\rho\,\Delta x}\sin(k\Delta x)\\ \frac{i\Delta t\mu}{\Delta x}\sin(k\Delta x) & \cos(k\Delta x)-A\end{bmatrix}\begin{bmatrix}v_0\\ \sigma_0\end{bmatrix}=0$$

For this system of equations to have solutions, the determinant of the 2×2 matrix must be zero:

$$\cos^2(k\Delta x)-2A\cos(k\Delta x)+A^2-\frac{i^2(\Delta t)^2\,\mu}{\rho(\Delta x)^2}\sin^2(k\Delta x)=0$$

Substituting the Courant number squared

$$C^2=\frac{\mu(\Delta t)^2}{\rho(\Delta x)^2}$$

and the fact that $i^2=-1$ we get

$$\cos^2(k\Delta x)-2A\cos(k\Delta x)+A^2+C^2\sin^2(k\Delta x)=0$$

This is a quadratic equation in A such that solutions can be found with a quadratic formula:

$$A^2-2A\cos(k\Delta x)+\cos^2(k\Delta x)+C^2\sin^2(k\Delta x)=0$$

such that the solution for A is

$$A = \frac{2\,\cos(k\Delta x) \pm \sqrt{4\cos^2(k\Delta x) - 4(\cos^2(k\Delta x) + C^2\sin^2(k\Delta x))}}{2}$$

which reduces to

$$A = \frac{2\,\cos(k\Delta x) \pm \sqrt{-4\,C^2\sin^2(k\Delta x)}}{2}$$

and therefore to

$$A = \cos(k\Delta x) \pm i\,C\sin(k\Delta x)$$

We require A to be less than or equal to 1, so that the A^n component of the periodic velocity or stress does not grow exponentially with the timestep. Evidently the real component of A, $\cos(k\Delta x)$, is bounded between [-1, 1] due to the range of the cosine function. Since the sine function has a similar range, it is clear that $C \leq 1$ such that the imaginary component of A remains bounded for any positive A^n.

Problem 6.6

In this method we will use the Lax-Friedrichs method for solving the velocity-stress formulation:

$$\begin{aligned} v_j^{n+1} &= \frac{1}{2}\left(v_{j+1}^n + v_{j-1}^n\right) + \frac{\Delta t}{\rho_j}\frac{\sigma_{j+1}^n - \sigma_{j-1}^n}{2\,\Delta x} \\ \sigma_j^{n+1} &= \frac{1}{2}\left(\sigma_{j+1}^n + \sigma_{j-1}^n\right) + \Delta t\,\mu_j\frac{v_{j+1}^n - v_{j-1}^n}{2\,\Delta x} \quad . \end{aligned}$$

We will compare our solution against the method from Problem 6.4. To do this, we will simultaneously solve for each system at each time step. There are therefore a lot of arrays to track! Anything with `LF_` is related to the Lax-Freidrichs method.

First let us define our domain:

```
import numpy as np
import matplotlib.pyplot as plt
from copy import copy

# Domain parameters
L  = 100
dx = 0.1

# Switch for dirichlet bc
# set to false for Neumann
DBC = True

# 1D Array holding grid points, x
x = np.arange(0, L+dx, dx)

# Number of grid points in x
N = len(x)

# CENTERED FD arrays
v_old       = np.zeros(N)
s_old       = np.zeros(N)

v           = np.zeros(N)
sigma       = np.zeros(N)

v_new       = np.zeros(N)
s_new       = np.zeros(N)

# Lax Freidrichs arrays
LF_v        = np.zeros(N)
LF_sigma    = np.zeros(N)

LF_v_new    = np.zeros(N)
LF_s_new    = np.zeros(N)


# Material properties - this is an array in so
# properties can be defined at each point in the grid
rho = np.zeros(N) + 1
mu  = np.zeros(N) + 1
```

CFL condition

Let us compute our timestep based on a Courant number of 0.5

```
beta = np.sqrt(np.max(mu)/np.min(rho))
C    = 0.5
dt   = dx * C / beta
```

Initial conditions

Next let us define a function that provides updates the velocity, v, with the initial condition

```
def apply_initial_condition(x, v, sigma):
    # Equation 6.50
    sigma[:] = 0
    v = -0.2 * (x-50) * np.exp(-0.1*(x-50)**2)
    return v, sigma
```

Solver

We now define the solver that iterates in time. Unlike in Problem 6.4, this formalisation only relies on the current timestep. The function below iterates the system one timestep. For comparison, we also copy over the centered finite difference method function from Problem 6.4.

```
def Lex_step_in_time(s, s_new, v, v_new, dx, dt, mu, rho, use_Dirichlet_BC):
    # Notes on slicing:
    # [1:-1] includes all points except boundaries
    # [2:  ] is from 3rd to end
    # [:-2 ] is from start to 3rd-last (missing last two points)

    # [1:-1] represents lower index = j
    # [2:  ] represents lower index = j + 1
    # [:-2 ] represents lower index = j - 1

    # Variable names (s and v):
    # s for sigma, v for v (velocity)
    # ? below means s or v
    # ?_new  represents superscript = n + 1
    # ?      represents superscript = n

    # Compute new v and s
    v_new[1:-1] = 0.5*(v[2:]+v[:-2]) + (dt / (2* rho[1:-1] * dx)) * (s[2:] - s[:-2])

    s_new[1:-1] = 0.5*(s[2:]+s[:-2]) + (dt * mu[1:-1] / (2*dx))   * (v[2:] - v[:-2])

    # Apply boundary conditions:
    if use_Dirichlet_BC:
        # Dirichlet:
        v_new[0]  = 0
        v_new[-1] = 0

        # Use a 1-sided gradient approximation for the edges:
        s_new[0]  = s[0]  +  dt * mu[0]  *  v[1] / dx
        s_new[-1] = s[-1] -  dt * mu[-1] * v[-2] / dx
    else:
        s_new[0]  = 0
        s_new[-1] = 0

        v_new[0]  = v[0]  + dt *   s[1] / (dx * rho[0])
        v_new[-1] = v[-1] - dt *  s[-2] / (dx * rho[-1])

    return  v_new, s_new


# Centered finite difference from Problem 6.4
def step_in_time(s_old, s, s_new, v_old, v, v_new, dx, dt, mu, rho, use_Dirichlet_BC
    v_new[1:-1] = v_old[1:-1] + (dt / (rho[1:-1] * dx)) * (s[2:] - s[:-2])
    s_new[1:-1] = s_old[1:-1] + (dt * mu[1:-1] / dx) * (v[2:] - v[:-2])

    # Apply boundary conditions:
    if use_Dirichlet_BC:
        # Dirichlet:
        v_new[0] = 0
        v_new[-1] = 0

        s_new[0]  = s_old[0]  + 2 * dt * mu[0]  * v[1] / dx
        s_new[-1] = s_old[-1] + 2 * dt * mu[-1] * -v[-2] / dx
```

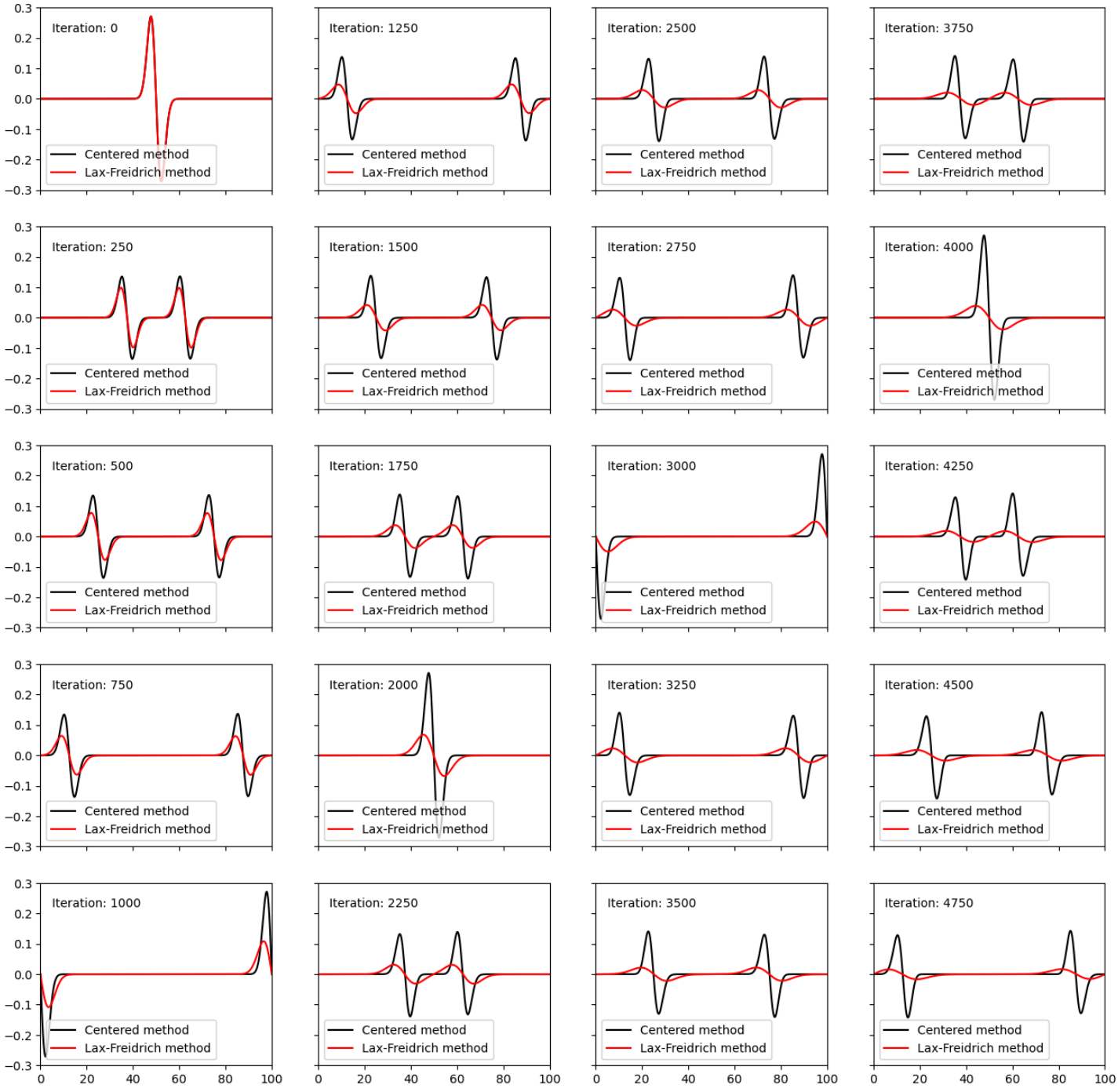
```
    else:
        s_new[0] = 0
        s_new[-1] = 0

        v_new[0] = v_old[0] + 2 * dt * s[1] / (dx * rho[0])
        v_new[-1] = v_old[-1] - 2 * dt * s[-2] / (dx * rho[-1])

    return v, v_new, s, s_new
```

Running the solver

Finally, let us run the solver in time and plot the timesteps

```
# Number of timesteps and steps to plot
nsteps        = 5000
plot_every    = 250
nplotcols     = 4
DBC           = True
plots_percol  = (nsteps//plot_every)//nplotcols

# Create Figure
fig, ax = plt.subplots(plots_percol, nplotcols,
                       figsize=(16, 16),
                       sharex=True,
                       sharey=True)

# Apply initial conditions for FD method
v, sigma     = apply_initial_condition(x, v, sigma)
v_old, s_old = apply_initial_condition(x, v_old, s_old)


# Apply initial conditions for LF method
LF_v, LF_sigma = apply_initial_condition(x, LF_v, LF_sigma)


# Iterate through timesteps
iax  = 0
icol = 0
for i in range(nsteps):

    # CENTERED SCHEME FROM PROBLEM 6.4:
    v, v_new, sigma, s_new = step_in_time(s_old=s_old,
                                          s=sigma,
                                          s_new=s_new,
                                          v_old=v_old,
                                          v=v,
                                          v_new=v_new,
                                          dx=dx, dt=dt,
                                          mu=mu, rho=rho,
                                          use_Dirichlet_BC=DBC)
    # Update timestep for FD method
    v_old[:] = v[:]
    s_old[:] = sigma[:]
    v[:]     = v_new[:]
    sigma[:] = s_new[:]


    # ----------- LEX-FRIEDRICH SCHEME --------------------
    LF_v_new, LF_s_new = Lex_step_in_time(s=LF_sigma,
                                          s_new=LF_s_new,
                                          v=LF_v,
                                          v_new=LF_v_new,
                                          dx=dx,
                                          dt=dt,
                                          mu=mu,
                                          rho=rho,
```

```
                                                use_Dirichlet_BC=DBC)
    # Update timestep for LF method
    LF_sigma[:] = LF_s_new[:]
    LF_v[:]     = LF_v_new[:]


    # Plot
    if i%plot_every==0:
        ax[iax, icol].plot(x, v, 'k')
        ax[iax, icol].plot(x, LF_v, 'r')
        ax[iax,icol].text(x=5, y = 0.22, s=f"Iteration: {i}")
        ax[iax,icol].set_xlim([0,100])
        ax[iax, icol].set_ylim([-0.3,0.3])
        ax[iax, icol].legend(['Centered method', 'Lax-Freidrich method'],
                             loc='lower left')
        iax += 1
        if iax == plots_percol:
            iax = 0
            icol += 1
```



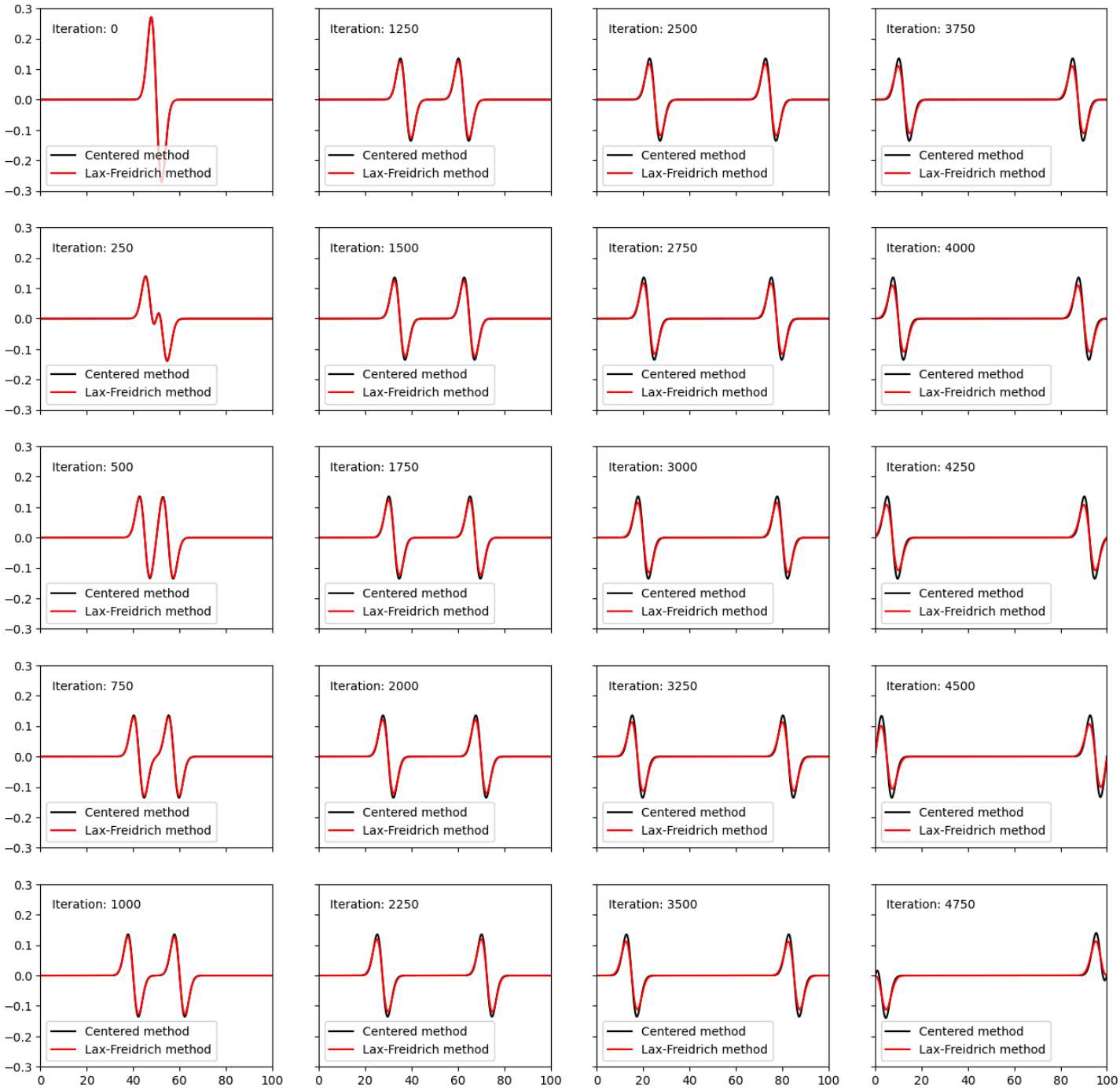
We observe the Lax-Freidrich method is doing a very poor job at reproducing the results of wave propagation. Why is this? This method is only 1st order accurate, but the main difference is due to the averaging that is employed. At each timestep the velocity and stress are being represented as

$$v_j^n \rightarrow \frac{1}{2}\left(v_{j+1}^n + v_{j-1}^n\right)$$
$$\sigma_j^n \rightarrow \frac{1}{2}\left(\sigma_{j+1}^n + \sigma_{j-1}^n\right)$$

as in Eqn 6.45. This means that we are averaging the values of these scalar fields spatially at each timestep, essentially smoothing the system and reducing high-frequency components. See, for example, iteration 1750 in which the true velocity field is being smoothed. This also results in disspiation of the amplitude through time.

In the cell below, we re-run the entire system described above but using a smaller grid discretisation (5x smaller), to improve the accuracy. Note this also affects the timestep.

▶ Show code cell source



In this case, we observe far better agreement between the two numerical methods, since the grid spacing is sufficiently small to prevent averaging-out of high-frequency wavefield components. However, we still observe dissipation in the amplitude due to this averaging.

Inhomogeneous Domain

Let us now consider the case of an inhomogeneous domain in which the shear modulus changes from $\mu = 1$ for $x \in [0,\, 65]$ to $\mu = 4$ for $x \in [65,\, 100]$. Once again, we will compare with the method used in Problem 6.4.

```
# Material properties - this is an array in so
# properties can be defined at each point in the grid
rho = np.zeros(N) + 1
mu  = np.zeros(N) + 1

# For the region where x is greater than 65 we will now edit the shear values:
mu[x > 65] = 4
```

Since we have changed our domain, we need to update timestep based on the CFL condition. Note here we are still using the finer grid spacing of dx = 0.02.

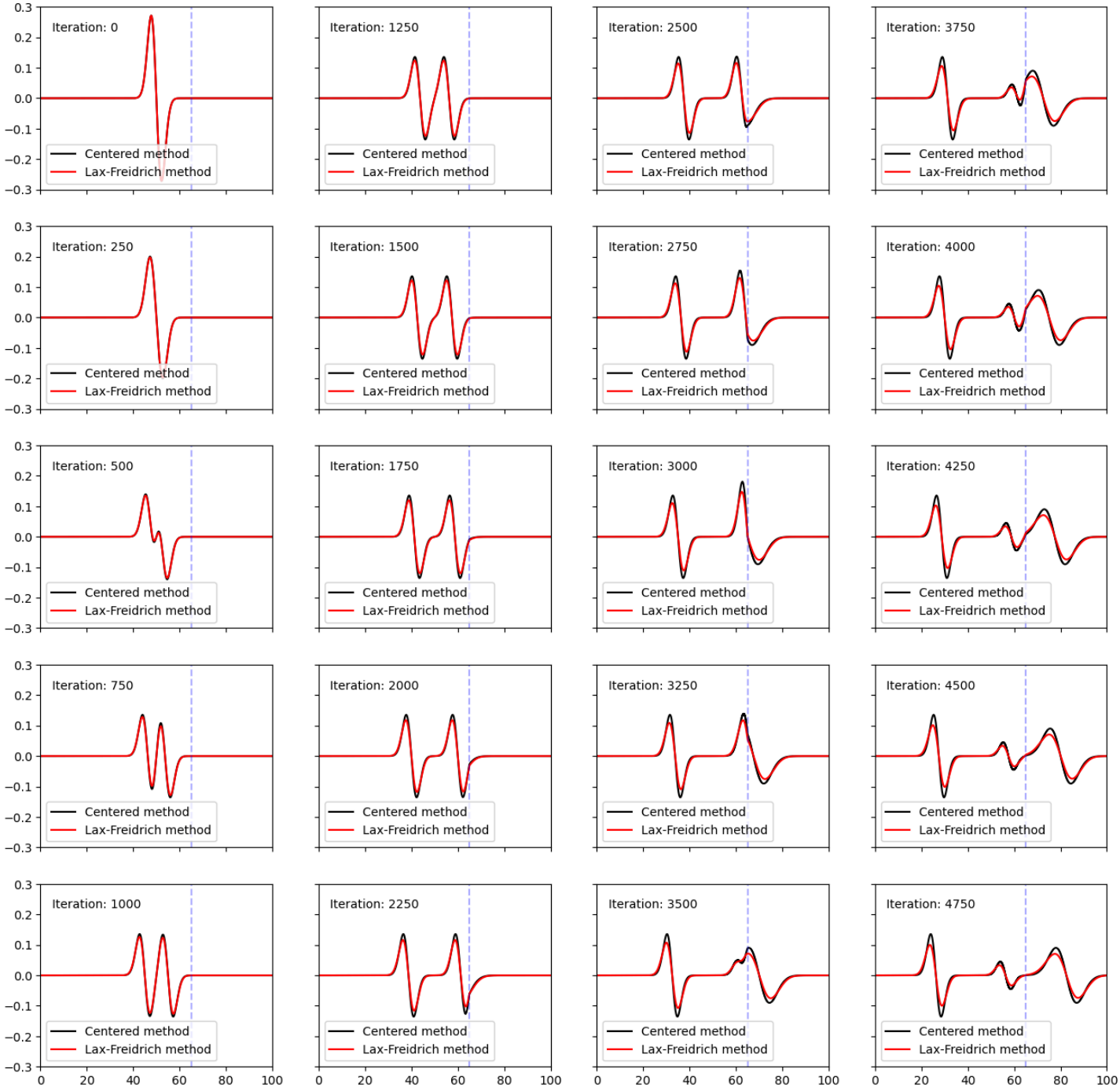
```
# We also need to update our CFL condition:
beta = np.sqrt(np.max(mu)/np.min(rho))
C = 0.5
dt = dx * C / beta
```

Once again, we run the solver. This time, we observe the expected reflection at the boundary in shear modulus properties ($x = 65$). Boundaries can also be a position of discrepency when using the Lax-Freidrich method since wavefield properties are being averaged across the boundary. Consider, for example, the case in which the shear modulus was equal to 0 in one section, i.e. it is a fluid. No wave should propagate within the fluid, so the velocity is 0. However the averaging scheme

$$v_j^n \to \frac{1}{2}\left(v_{j+1}^n + v_{j-1}^n\right)$$
$$\sigma_j^n \to \frac{1}{2}\left(\sigma_{j+1}^n + \sigma_{j-1}^n\right)$$

may allow for non-zero velocity values if, for example, one of v_{j+1}^n, or v_{j-1}^n lay within the solid region.

▶ Show code cell source



Problem 6.7

Starting with equation 6.54

$$\sin^2(\omega\,\Delta t) = C^2\,\sin^2(k\,\Delta x)$$

where

$$C^2 = \frac{\Delta t^2 \, \mu}{\rho \, \Delta x^2}$$

which can be seen by setting the determinant of the matrix in equation 6.53 to equal 0. Taking the derivative of this with respect to k, and remembering that ω is a function of k, we use the chain rule to get

$$2 \, \Delta t \, \cos(\omega \, \Delta t) \, \sin(\omega \, \Delta t) \frac{\partial \omega}{\partial k} = 2 \, C^2 \, \Delta x \, \cos(k \, \Delta x) \, \sin(k \, \Delta x)$$

Using the trigonometric identity that

$$\sin(2\theta) = 2 \sin(\theta) \cos(\theta)$$

this may be simplified to

$$\Delta t \, \sin(2\omega \, \Delta t) \frac{\partial \omega}{\partial k} = C^2 \, \Delta x \, \sin(2k \, \Delta x)$$

followed by division

$$\frac{\partial \omega}{\partial k} = \frac{C^2 \, \Delta x}{\Delta t} \frac{\sin(2k \, \Delta x)}{\sin(2\omega \, \Delta t)}$$

and finally substituting the definition of C^2

$$\frac{\partial \omega}{\partial k} = \frac{\Delta t^2 \, \mu}{\rho \, \Delta x^2} \frac{\Delta x}{\Delta t} \frac{\sin(2k \, \Delta x)}{\sin(2\omega \, \Delta t)} = \frac{c^2 \, \Delta t}{\Delta x} \frac{\sin(2k \, \Delta x)}{\sin(2\omega \, \Delta t)}$$

Problem 6.8

Starting with our equation (6.75) for the grid phase speed

$$c^{\text{grid}} = \frac{2}{k\,\Delta t}\arcsin\left\{c\,\Delta t\left[\left(\frac{\sin\left(\frac{1}{2}k_x\,\Delta x\right)}{\Delta x}\right)^2+\left(\frac{\sin\left(\frac{1}{2}k_y\,\Delta y\right)}{\Delta y}\right)^2\right]^{1/2}\right\}$$

This is written slightly differently to the textbook, collecting the powers of 2 on the sine functions and Δx/Δy outside of a bracket. Hopefully this makes it clearer for the next step of our solution.

The fact that the question tells us to use $k_x^2 + k_y^2 = k^2$ hints we should aim to extract the k_x and k_y from the sine functions. In fact, when Δx and Δy are small, and the argument of the sine functions tends to 0. The small-angle approximation states that as $\theta \to 0$ the sine function

$$\sin(\theta) \to \theta$$

Therefore since $\Delta x \to 0$ and $\Delta y \to 0$ we may approximate

$$\sin\left(\frac{1}{2}k_x\,\Delta x\right) \to \frac{1}{2}k_x\,\Delta x$$
$$\sin\left(\frac{1}{2}k_y\,\Delta y\right) \to \frac{1}{2}k_y\,\Delta y$$

The equation for the grid speed then becomes

$$c^{\text{grid}} = \frac{2}{k\,\Delta t}\arcsin\left\{c\,\Delta t\left[\left(\frac{\frac{1}{2}k_x\,\Delta x}{\Delta x}\right)^2+\left(\frac{\frac{1}{2}k_y\,\Delta y}{\Delta y}\right)^2\right]^{1/2}\right\}$$

which may be reduced to

$$c^{\text{grid}} = \frac{2}{k\,\Delta t}\arcsin\left\{c\,\Delta t\left[\frac{1}{4}\left(k_x^2+k_y^2\right)\right]^{1/2}\right\}$$

Next, we utilise the aforementioned information that $k_x^2 + k_y^2 = k^2$ such that this simplifies to

$$c^{\text{grid}} = \frac{2}{k\,\Delta t}\arcsin\left\{\frac{1}{2}c\,\Delta t\,k\right\}$$

Finally, dealing we must navigate the limit of the $\arcsin$ as $\Delta t \to 0$. A Taylor expansion of $\arcsin(\Delta t)$ about 0 is

$$\Delta t + \frac{1}{6}(\Delta t)^3 + \mathcal{O}(\Delta t^5) \quad .$$

Hence to first order we may approximate the arcsin function in this limit as simply its argument, making the speed as

$$c^{\text{grid}} = \frac{1}{2} c\,\Delta t\, k \frac{2}{k\,\Delta t} = c$$

as required.

How wrong is the grid speed?

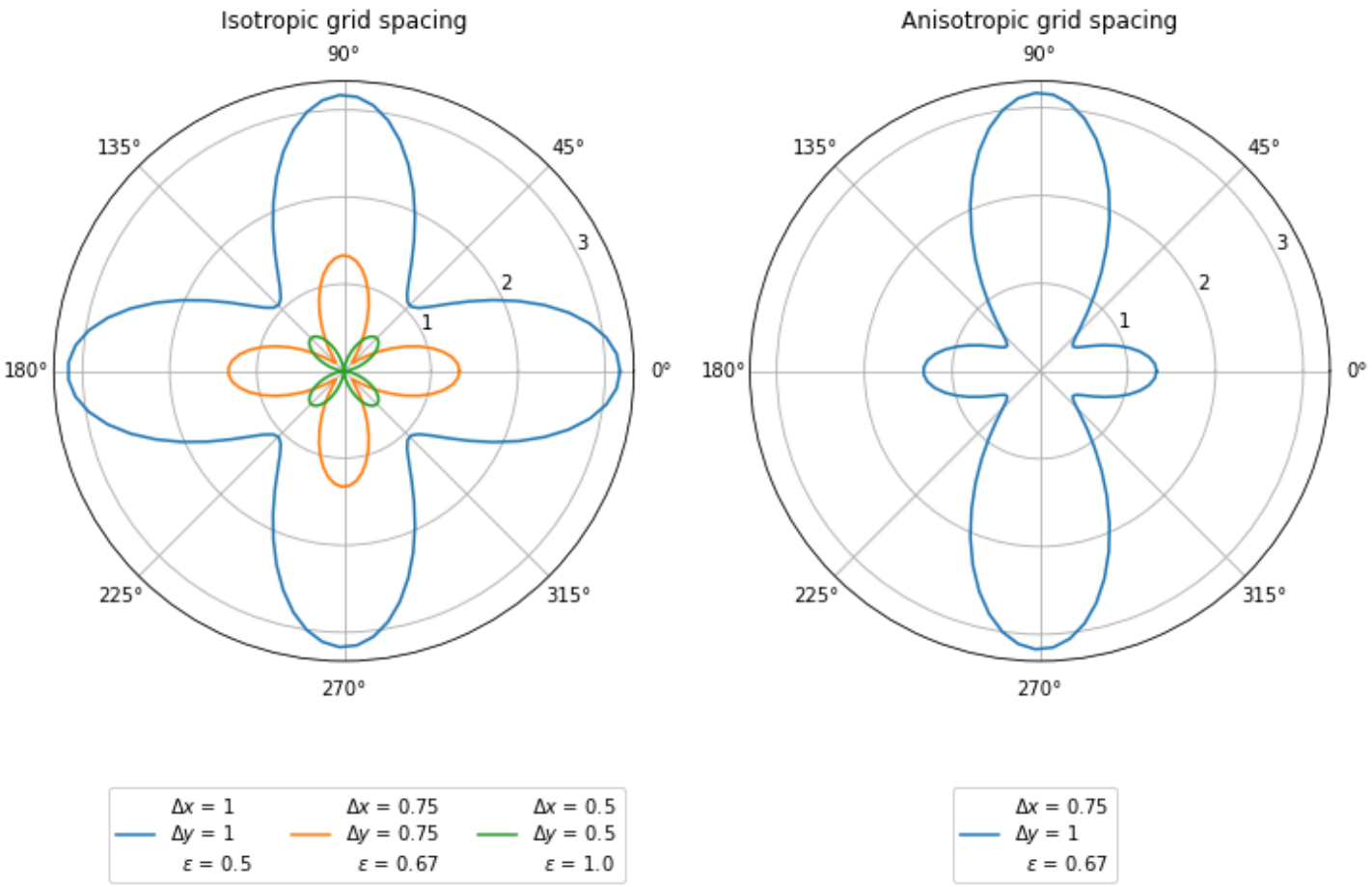
The phase speed on the grid is, as above,

$$c^{\text{grid}} = \frac{2}{k\,\Delta t} \arcsin\left\{ c\,\Delta t \left[\left(\frac{\sin\left(\frac{1}{2} k_x\,\Delta x\right)}{\Delta x} \right)^2 + \left(\frac{\sin\left(\frac{1}{2} k_y\,\Delta y\right)}{\Delta y} \right)^2 \right]^{1/2} \right\}.$$

Below, we plot the error between the true speed and the grid speed. These plots are inspired by Fig 4.14 of Heiner Igel's Computational Seismology book.

```python
import numpy as np
import matplotlib.pyplot as plt


k  = 1                          # Wavenumber
DX = [1, 0.75, 0.5,  0.75]      # Grid spacing in x
DY = [1, 0.75, 0.5,  1]         # Grid spacing in y
dt = 0.5                        # Timestep
c  = 1                          # True wavespeed

# courant number for each case
courant_x = c*dt/np.array(DX);
courant_y = c*dt/np.array(DY);

# number of scenarios
nscenarios = len(DX)

# create angle over which to evaluate
# x and y given by cos and sin of theta
theta = np.linspace(0, 2*np.pi, 100)

fig, axes = plt.subplots(1, 2, figsize=(12,6), subplot_kw={'projection': 'polar'})

leglabels = []

for i in range(nscenarios):

    if i < nscenarios-1:
        ax = axes[0]
        ax.set_title("Isotropic grid spacing", va='bottom')
        bbox = (1, -0.2)

    else:
        ax = axes[1]
        ax.set_title("Anisotropic grid spacing", va='bottom')
        bbox = (0.65, -0.2)

    dx = DX[i]
    dy = DY[i]

    # Compute grid speed at each theta
    cgrid = (2/(k*dt)) * np.arcsin( c*dt * ((np.sin(k*np.cos(theta) * dx/2)/dx)**2 +
                                            (np.sin(k*np.sin(theta) * dy/2)/dy)**2)

    # Compute percentage error from 'true' c
    deltac  = 100*np.abs((cgrid - c))/c

    ax.plot(theta, deltac)
    ax.set_rlabel_position(27)  # Move radial labels away from plotted line
    ax.set_rticks([1, 2, 3])    # Less radial ticks

    leglabels.append(r'$\Delta x$ = ' +f'{dx}\n' + r'$\Delta y$ = '+f'{dy}\n'
                     + r'  $\epsilon$ = '+f'{np.around(courant_x[i],2)}')
```

```
    if i >= nscenarios-2:
        ax.legend(leglabels, ncols=3, bbox_to_anchor=bbox)
        leglabels = []
```



Problem 6.9

Starting with Equation 6.74:

$$\sin\left(\frac{1}{2}\,\omega\,\Delta t\right) = c\,\Delta t\left[\left(\frac{\sin\left(\frac{1}{2}k_x\,\Delta x\right)}{\Delta x}\right)^2 + \left(\frac{\sin\left(\frac{1}{2}k_y\,\Delta y\right)}{\Delta y}\right)^2\right]^{1/2}$$

Taking the derivative of this with respect to the wavevector in the x direction, k_x, remembering that $\omega \sim \omega(k_x,\, k_y)$ we get

$$\frac{1}{2}\,\Delta t\cos\left(\frac{1}{2}\,\omega\,\Delta t\right)\frac{\partial\omega}{\partial k_x}=\frac{1}{2}\,c\,\Delta t\left[\left(\frac{\sin\left(\frac{1}{2}k_x\,\Delta x\right)}{\Delta x}\right)^2+\left(\frac{\sin\left(\frac{1}{2}k_y\,\Delta y\right)}{\Delta y}\right)^2\right]^{-1/2}\cos$$

Noting here that we can rearrange Eqn. 6.74 to

$$\left[\left(\frac{\sin\left(\frac{1}{2}k_x\,\Delta x\right)}{\Delta x}\right)^2+\left(\frac{\sin\left(\frac{1}{2}k_y\,\Delta y\right)}{\Delta y}\right)^2\right]^{-1/2}=\frac{c\,\Delta t}{\sin\left(\frac{1}{2}\,\omega\,\Delta t\right)}$$

and substitute it in to yield

$$\frac{1}{2}\,\Delta t\cos\left(\frac{1}{2}\,\omega\,\Delta t\right)\frac{\partial\omega}{\partial k_x}=\frac{1}{2}\,\frac{c^2\,(\Delta t)^2}{\sin\left(\frac{1}{2}\,\omega\,\Delta t\right)}\,\frac{\cos\left(\frac{1}{2}k_x\,\Delta x\right)\sin\left(\frac{1}{2}k_x\,\Delta x\right)}{\Delta x}$$

Rearranging this slightly to get

$$\frac{\partial\omega}{\partial k_x}=\frac{c^2\Delta t}{\Delta x}\,\frac{\cos\left(\frac{1}{2}k_x\,\Delta x\right)\sin\left(\frac{1}{2}k_x\,\Delta x\right)}{\cos\left(\frac{1}{2}\,\omega\,\Delta t\right)\sin\left(\frac{1}{2}\,\omega\,\Delta t\right)}$$

Finally the double angle formula $\sin(2\theta)=2\sin(\theta)\cos(\theta)$ allows both the numerator and denomenator to be converted:

$$\cos\left(\frac{1}{2}k_x\,\Delta x\right)\sin\left(\frac{1}{2}k_x\,\Delta x\right)=\frac{1}{2}\sin\left(k_x\,\Delta x\right)$$
$$\cos\left(\frac{1}{2}\,\omega\,\Delta t\right)\sin\left(\frac{1}{2}\,\omega\,\Delta t\right)=\frac{1}{2}\sin\left(\omega\,\Delta t\right)$$

such that derivative can be written as

$$\frac{\partial\omega}{\partial k_x}=\frac{c^2\Delta t}{\Delta x}\,\frac{\sin\left(k_x\,\Delta x\right)}{\sin\left(\omega\,\Delta t\right)}$$

which is the desired solution for Eqn 6.76. There is is no difference in deriving 6.77 other than to take the derivative with respect to k_y and so this is left to the reader.

Problem 6.10

In this problem we will compare an explicit and implicit timescheme for the heat equation,

$$\partial_t \theta = \alpha \, \partial_x^2 \theta \quad .$$

First let us look at the explicit scheme.

Explicit scheme

Using the forward difference approximation in time and a centred finite difference for the spatial gradient temperature, θ, at the next timestep $n+1$ may be approximated as

$$\theta_j^{n+1} = \theta_j^n + \frac{\alpha \, \Delta t}{(\Delta x)^2} \left(\theta_{j+1}^n - 2\theta_j^n + \theta_{j-1}^n \right)$$

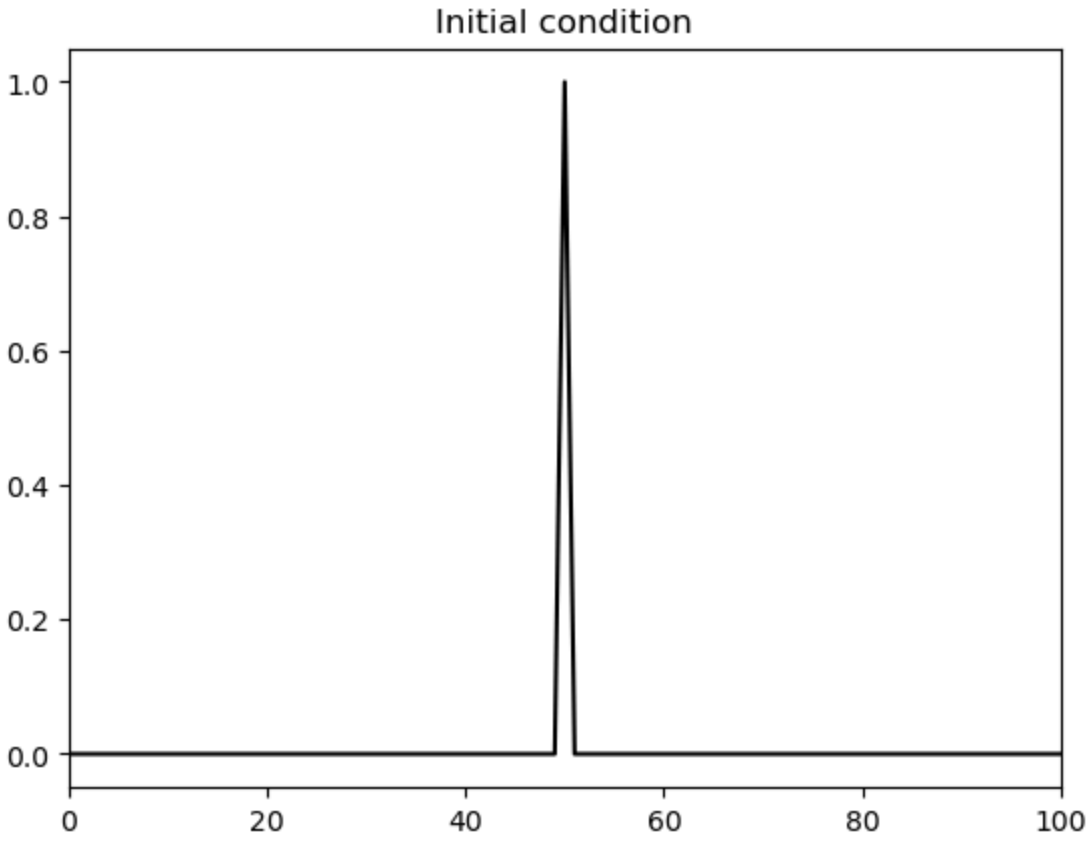
Let us first set up the domain and define a function for the initial condition:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors


# Since the dx is 1, lets initialise x using arange
x  = np.arange(0, 101)
dx = 1
# Initialise theta array of the same length
N = len(x)
theta = np.zeros(N)


# Set alpha value
alpha = 1.0
```

Initial condition:

Below we define a function to set the initial condition. This is a delta-function-like spike.

```
def set_init_condition(x, theta):
    theta[:]     = 0
    theta[x==50] = 1
    return theta

# Set the initial condition and plot it
theta   = set_init_condition(x, theta)

fig, ax = plt.subplots()
ax.plot(x, theta, 'k')
ax.set_xlim([0,100])
ax.set_title('Initial condition');
```



Timestepping:

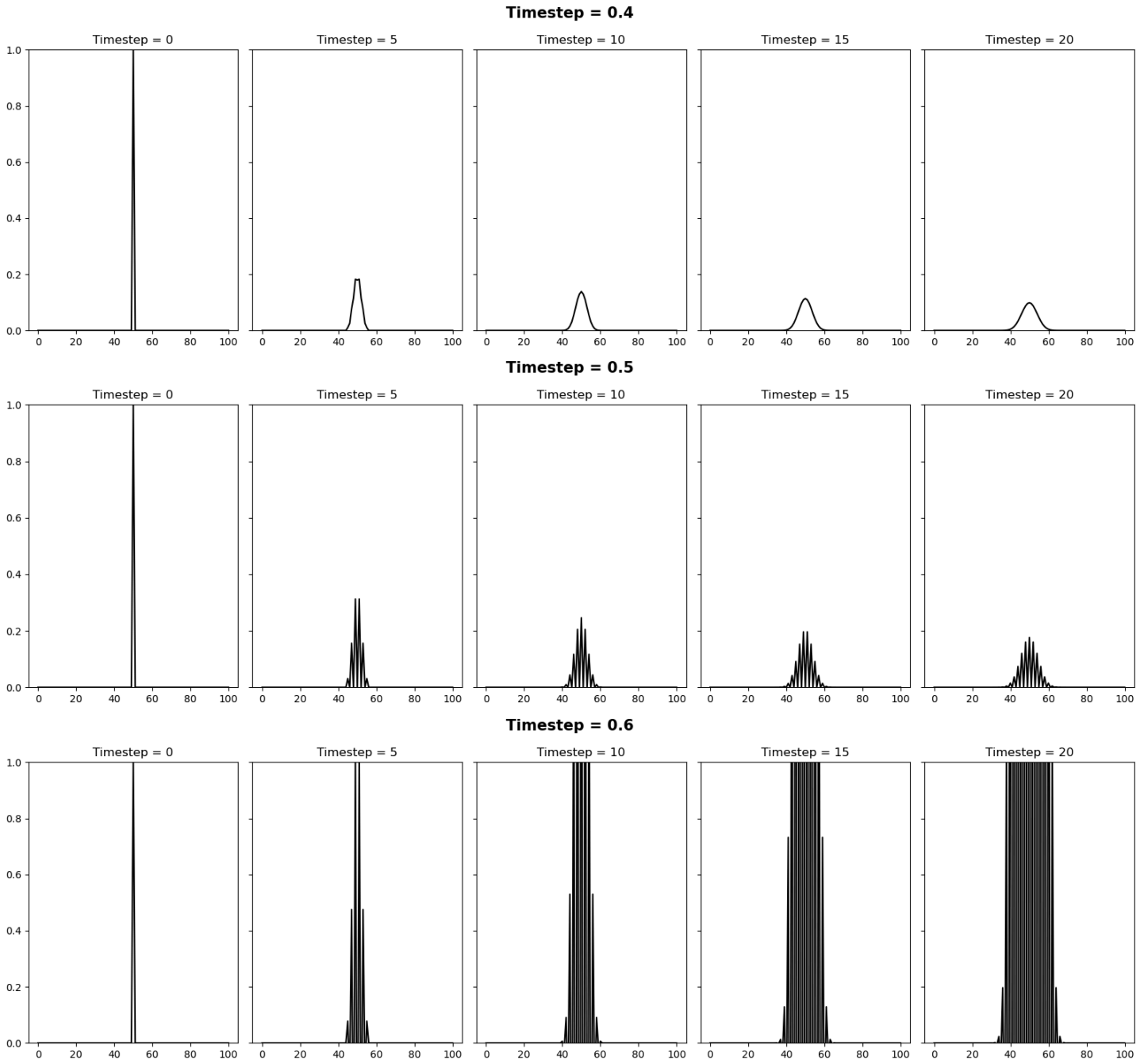
Below we define the function that solves for θ^{n+1} based on θ^n. We will impose boundary conditions that $\theta^n = 0$ for $x = 0, L$.

```python
def explicit_step_in_time(theta, alpha, dt, dx):
    # theta = 1D array, current timestep n
    # alpha = float, diffusivity
    # dt    = float, timestep
    # dx    = float, grid spacing

    # Impose boundary conditions
    theta[0]  = 0
    theta[-1] = 0

    # Update values not at boundary
    theta[1:-1] = theta[1:-1] + (alpha*dt/(dx*dx)) * (theta[2:] - 2*theta[1:-1] + tl

    return theta
```

Investigating the stability

We can vary the stability of this scheme by varying the value of k where $\Delta t = \alpha(\Delta x)^2/k$. It appears there is a slight typo in the question. It is supposed to be asking the reader to test how the stability changes when the timestep is between 0.4 and 0.6. We can relate these to k values of 10/4 and 10/6. Below we will plot 3 figures for different timesteps of 0.4, 0.5 and 0.6. In each plot we show the initial condition in grey, and the temperature at the current timestep in blue. We observe a system that is stable ($\Delta t = 0.4$), on the edge of stability ($\Delta t = 0.5$) and completely unstable ($\Delta t = 0.6$).

```
for k in [10/4, 10/5, 10/6]:
    dt = alpha * (dx**2) / k


    # Lets reimpose the boundary conidition to ensure it is
    # imposed every time we run this notebook cell
    theta   = set_init_condition(x, theta)

    # Setup 5 subplots
    nplots = 5
    fig, ax = plt.subplots(1,nplots, figsize=(16,5),
                           sharey=True, sharex=True)
    fig.set_tight_layout(True)
    fig.suptitle(f'Timestep = {dt}', fontsize=15, weight='bold')


    # Loop through each timestep (i) and plot every
    # 'plot_every'th timestep so that
    # there are 5 plots
    ntimesteps = 25
    plot_every = int(ntimesteps/nplots)

    ictr = 0
    for i in range(ntimesteps):


        if i%plot_every==0:
            ax[ictr].plot(x, theta, 'k')
            ax[ictr].set_title(f'Timestep = {i}')
            ax[ictr].set_ylim([0,1])
            ictr += 1

        theta = explicit_step_in_time(theta, alpha, dt, dx)
```



Implicit scheme

Using the forward difference approximation in time and a centred finite difference for the spatial gradient temperature, θ, at the next timestep $n+1$ may be approximated as

$$-\beta\,\theta_{j-1}^{n} + \gamma\,\theta_{j}^{n} - \beta\,\theta_{j+1}^{n} = \theta_{j}^{n-1}$$

where I am denoting $\beta = \frac{\alpha\,\Delta t}{(\Delta x)^2}$ and $\gamma = [1 + 2\beta]$. This system of linear equations for each spatial point, $j = 1, N$ may then be written as a matrix:

$$\begin{bmatrix} 1 & & & & & & \\ -\beta & \gamma & -\beta & & & & \\ & -\beta & \gamma & -\beta & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & -\beta & \gamma & -\beta & \\ & & & & -\beta & \gamma & -\beta \\ & & & & & & 1 \end{bmatrix} \begin{bmatrix} \theta_0^n \\ \theta_1^n \\ \theta_2^n \\ \vdots \\ \theta_{N-2}^n \\ \theta_{N-1}^n \\ \theta_N^n \end{bmatrix} = \begin{bmatrix} \theta_0^{n-1} \\ \theta_1^{n-1} \\ \theta_2^{n-1} \\ \vdots \\ \theta_{N-2}^{n-1} \\ \theta_{N-1}^{n-1} \\ \theta_N^{n-1} \end{bmatrix},$$

or in matrix form as

$$\mathbf{K}\boldsymbol{\Theta}^n = \boldsymbol{\Theta}^{n-1}.$$

The structure of the matrix $\mathbf{K}$ is sparse (mostly zeros) and tri-diagonal. Note that the values in the first and last rows ensure that

$$\theta_0^n = \theta_0^{n-1}$$
$$\theta_N^n = \theta_N^{n-1}$$

as per the Dirichlet boundary condition. As long as $\mathbf{K}$ can be inverted, this solution may then be time-marched as

or in matrix form as

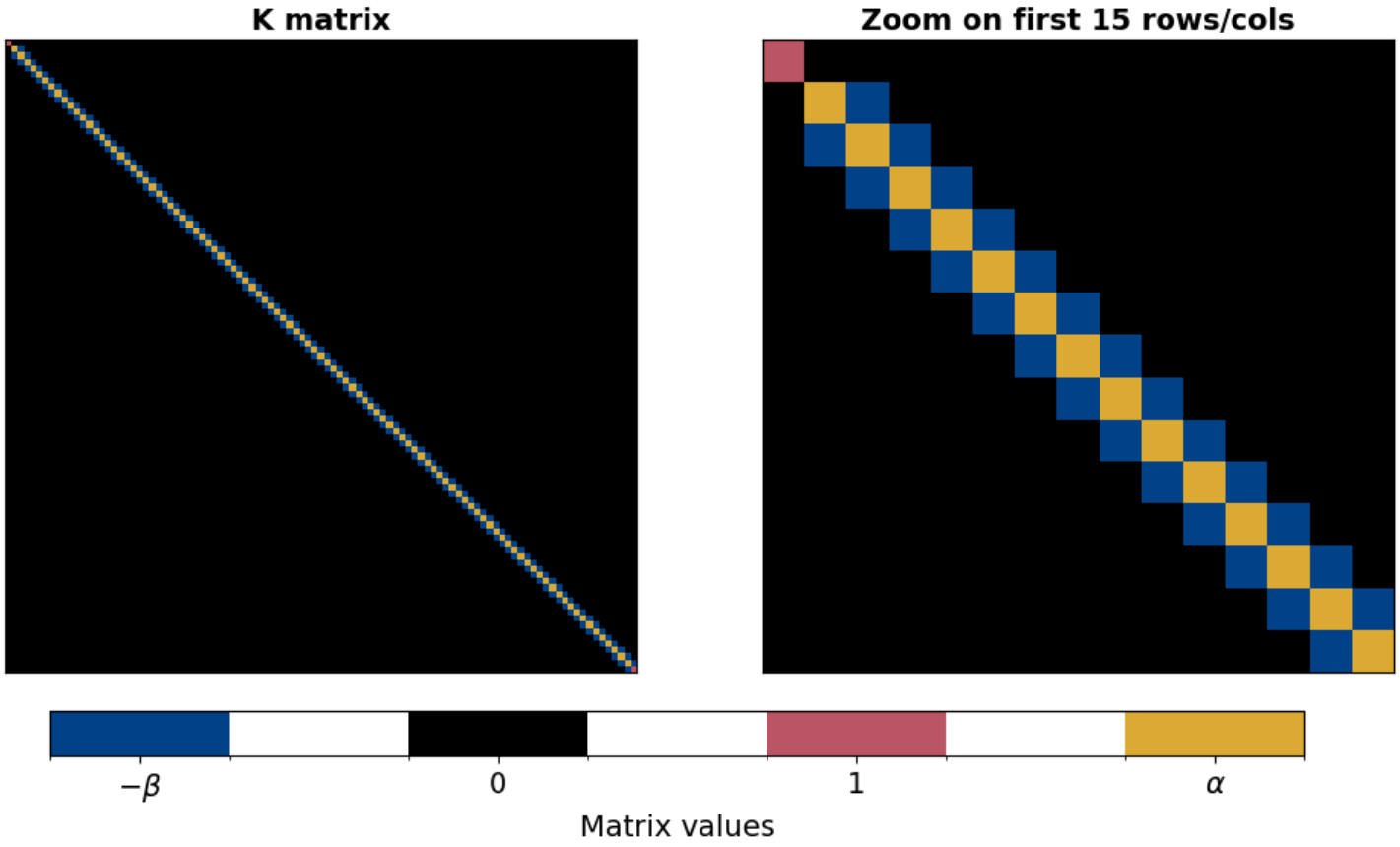
$$\boldsymbol{\Theta}^n = \mathbf{K}^{-1}\boldsymbol{\Theta}^{n-1},$$

and since $\mathbf{K}$ is time-independent, the inversion must be completed only once. In the code cell below, a function is written to build the $\mathbf{K}$ matrix. Plotting this matrix, we then see the tridiagonal structure.

```python
def build_K_mat(alpha, dt, dx, return_beta_gamma=False):

    beta =  alpha*dt/(dx**2)
    gamma = 1 + 2*beta


    # Build our matrix
    K = np.zeros((N,N))

    # Avoiding the first and last elements:
    for i in range(1, N-1):
        K[i,i] = gamma
        K[i+1,i] = -beta
        K[i,i+1] = -beta
    K[0,0]   = 1
    K[-1,-1] = 1

    if return_beta_gamma:
        return -np.around(beta,3), np.around(gamma,3), K
    else:
        return K
```

Let us briefly investigate the structure of the stiffness matrix. We observe its tri-diagonal structure with variations at the first and last element imposing the boundary conditions.

▶ Show code cell source



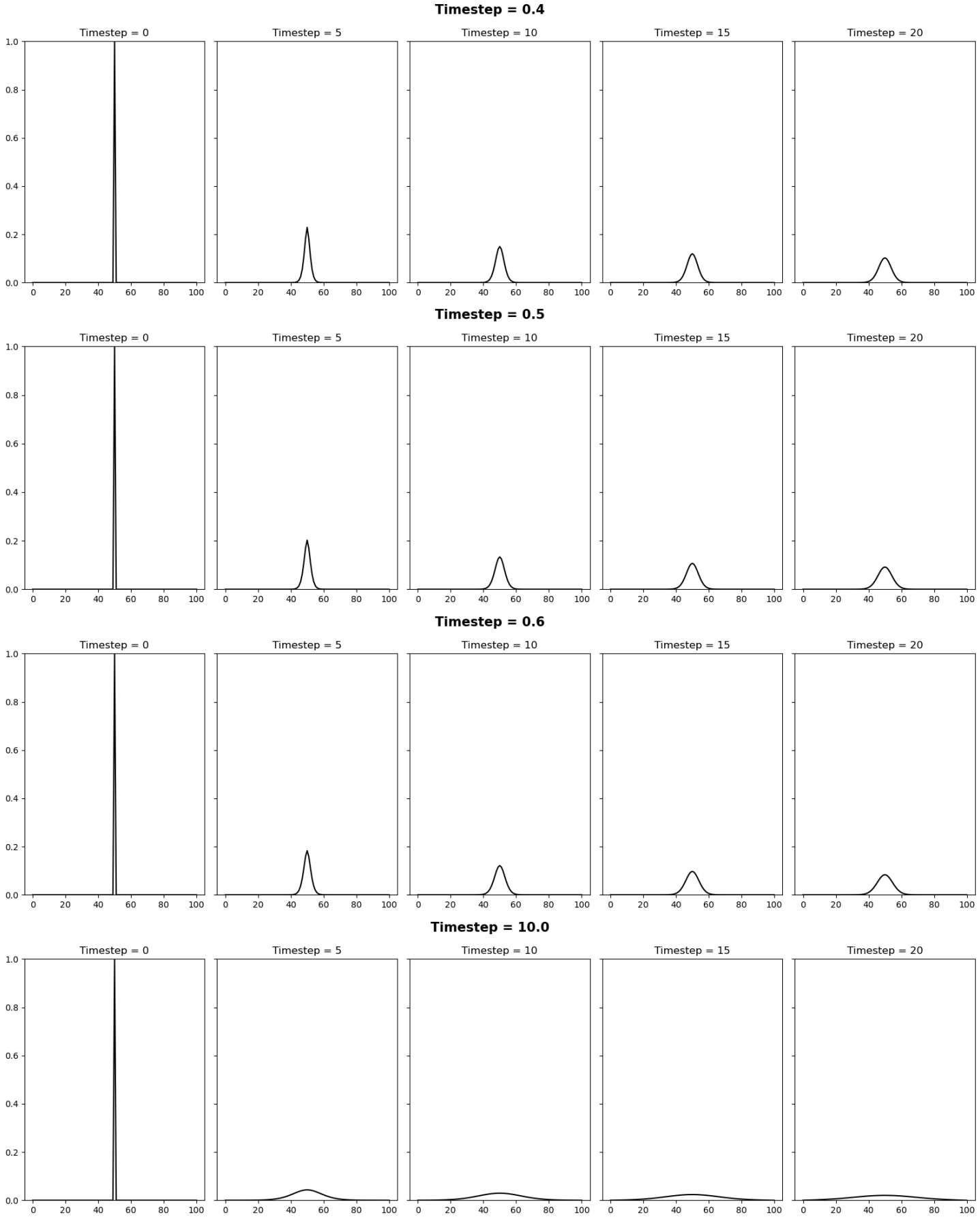
Solving with the Implicit method

The code snippet below is identical to the code above apart from the following

- It computes K for the chosen timestep and inverts it
- The `explicit_step_in_time` is replaced with a line of code that simply computes the matrix calculation $\mathbf{\Theta}^n = \mathbf{K}^{-1}\mathbf{\Theta}^{n-1}$

In this case we will try the same timesteps as above, as well as $\Delta t = 10$. Since this setup is unconditionally stable, all these timesteps provide a stable solution.

```
for k in [10/4, 10/5, 10/6, 0.1]:
    dt = alpha * (dx**2) / k

    # Compute the K matrix for this timestep:
    K = build_K_mat(alpha, dt, dx)
    Kinv = np.linalg.inv(K)

    # Lets reimpose the boundary conidition to ensure it is
    # imposed every time we run this notebook cell
    theta   = set_init_condition(x, theta)

    # Setup 5 subplots
    nplots = 5
    fig, ax = plt.subplots(1,nplots, figsize=(16,5), sharey=True, sharex=True)
    fig.set_tight_layout(True)
    fig.suptitle(f'Timestep = {dt}', fontsize=15, weight='bold')

    # Loop through each timestep (i) and plot every 'plot_every'th timestep so that
    # there are 5 plots
    ntimesteps = 25
    plot_every = int(ntimesteps/nplots)

    ictr = 0
    for i in range(ntimesteps):


        if i%plot_every==0:
            ax[ictr].plot(x, theta, 'k')
            ax[ictr].set_title(f'Timestep = {i}')
            ax[ictr].set_ylim([0,1])
            ictr += 1

        theta = np.matmul(Kinv, theta)
```


Timestep = 0.4
Timestep = 0
Timestep = 5
Timestep = 10
Timestep = 15
Timestep = 20
Timestep = 0.5
Timestep = 0
Timestep = 5
Timestep = 10
Timestep = 15
Timestep = 20
Timestep = 0.6
Timestep = 0
Timestep = 5
Timestep = 10
Timestep = 15
Timestep = 20
Timestep = 10.0
Timestep = 0
Timestep = 5
Timestep = 10
Timestep = 15
Timestep = 20

Problem 6.11

Let us take Eqn. 6.92

$$-\frac{\gamma}{2}\theta_{j-1}^{n}+\left[1+\gamma\right]\theta_{j}^{n}-\frac{\gamma}{2}\theta_{j+1}^{n}=\frac{\gamma}{2}\theta_{j-1}^{n-1}+[1-\gamma]\theta_{j}^{n-1}+\frac{\gamma}{2}\theta_{j+1}^{n-1}$$

where $\gamma=\frac{\alpha\Delta t}{(\Delta x)^2}$. Next as instructed we will sub in our plane wave approximations from 6.83 for the current timestep

$$\begin{aligned}\theta_{j-1}^{\ n}&=A^n\,e^{ik(j-1)\Delta x}\\\theta_{j}^{\ n}&=A^n\,e^{ikj\Delta x}\\\theta_{j+1}^{\ n}&=A^n\,e^{ik(j+1)\Delta x}\end{aligned}$$

and previous timestep

$$\begin{aligned}\theta_{j-1}^{\ n-1}&=A^{n-1}\,e^{ik(j-1)\Delta x}\\\theta_{j}^{\ n-1}&=A^{n-1}\,e^{ikj\Delta x}\\\theta_{j+1}^{\ n-1}&=A^{n-1}\,e^{ik(j+1)\Delta x}\end{aligned}$$

which yields

$$\begin{aligned}&-\frac{\gamma}{2}A^n\,e^{ik(j-1)\Delta x}+\left[1+\gamma\right]A^n\,e^{ikj\Delta x}-\frac{\gamma}{2}A^n\,e^{ik(j+1)\Delta x}\\=&\frac{\gamma}{2}A^{n-1}\,e^{ik(j-1)\Delta x}+[1-\gamma]A^{n-1}\,e^{ikj\Delta x}+\frac{\gamma}{2}A^{n-1}\,e^{ik(j+1)\Delta x}\end{aligned}$$

Let us now divide by $A^{n-1}e^{ikj\Delta x}$,

$$\begin{aligned}&A\left\{-\frac{\gamma}{2}\,e^{-ik\Delta x}+\left[1+\gamma\right]\,-\frac{\gamma}{2}\,e^{ik\Delta x}\right\}\\=&\frac{\gamma}{2}\,e^{-ik\Delta x}+\left[1-\gamma\right]\,+\frac{\gamma}{2}\,e^{ik\Delta x}\end{aligned}$$

and collect the terms

$$
\begin{aligned}
&A\left\{1-\gamma\left(-1+\frac{e^{-ik\Delta x}+e^{ik\Delta x}}{2}\right)\right\}\\
=&1+\gamma\left(-1+\frac{e^{-ik\Delta x}+e^{ik\Delta x}}{2}\right)
\end{aligned}
$$

where we can use the identity that

$$
\frac{e^{-i\theta}+e^{i\theta}}{2}=\cos\theta
$$

to reduce this to

$$
A\left\{1-\gamma\left(-1+\cos\left(k\Delta x\right)\right)\right\}=1+\gamma\left(-1+\cos\left(k\Delta x\right)\right)
$$

Finally we may use the identity that

$$
2\sin^2\theta=1-\cos 2\theta
$$

where in our case, $\theta=\frac{1}{2}k\Delta x$ such that

$$
-1+\cos(k\Delta x)=-2\sin^2\left(\frac{1}{2}k\Delta x\right)
$$

and so we may write

$$
A\left\{1+2\gamma\sin^2\left(\frac{1}{2}k\Delta x\right)\right\}=1-2\,\gamma\sin^2\left(\frac{1}{2}k\Delta x\right)
$$

and so

$$
A=\frac{1-2\,\gamma\sin^2\left(\frac{1}{2}k\Delta x\right)}{1+2\gamma\sin^2\left(\frac{1}{2}k\Delta x\right)}
$$

as required.

Unconditional stability

Why is this method unconditionally stable for any value of k? Let us consider our expression

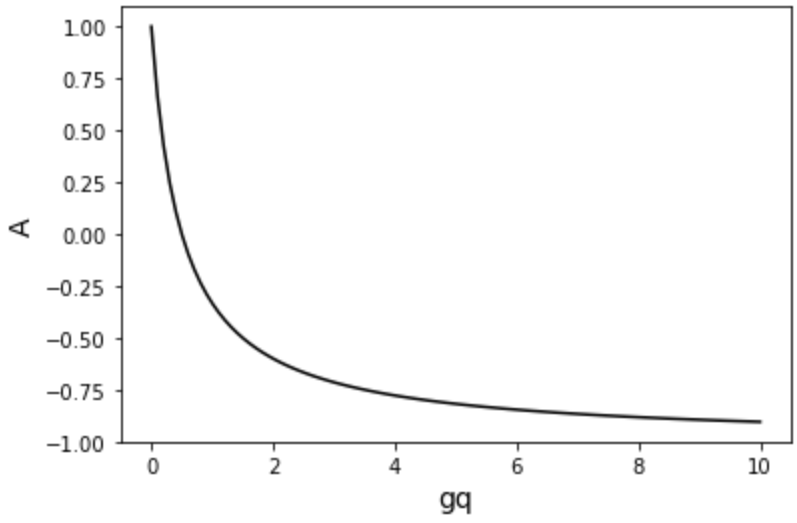
$$A = \frac{1 - 2\,\gamma \sin^2\left(\frac{1}{2}k\Delta x\right)}{1 + 2\gamma \sin^2\left(\frac{1}{2}k\Delta x\right)}.$$

Regardless of k the $\sin^2$ terms are bounded between $[0, 1]$ such that the maximum value of A is controlled by

$$A = \frac{1 - 2\,\gamma q}{1 + 2\gamma q}.$$

where q ranges from $[0, 1]$. γq is therefore always positive and bounded between $0 < \gamma q \leq \gamma$. By plotting this we can clearly see that A will never be larger than 1. At $\gamma q = 0$, $A = 1$ and for $\gamma q > 0$ then $A < 1$.

```
import numpy as np
import matplotlib.pyplot as plt

gq = np.linspace(0, 10, 100)
A = (1 - 2*gq)/(1 + 2*gq)

fig,  ax = plt.subplots()

ax.plot(gq, A, 'k');
ax.set_xlabel('gq', fontsize=14);
ax.set_ylabel('A', fontsize=14);
```



Problem 6.12

In this problem we study the velocity-stress formulation of the 1D wave equation using a pseudospectral method. First let us define a function to produce our initial condition, setting the velocity and stress:

```
import numpy as np
import matplotlib.pyplot as plt
from copy import copy


def init_condition(x, v, sigma):
    sigma[:] = 0
    v = -0.2*(x-50) * np.exp(-0.1 * (x-50)**2)
    return v, sigma
```

Next we can define the domain and properties of the system. Note that by the nature of a pseudospectral method, the first and last points of the domain are identical. Here, we create an array of $N + 1$ length for plotting purposes, with the understanding that the first and last nodes will be the same.

```
L            = 100       # domain size
dx           = 0.10      # grid spacing
dirichlet_bc = True     # switch for BC

# Here x goes from 0 (j=0) to 99.9 (j=N-1)
# and therefore has length of N
x =  np.arange(0, L+dx, dx)
N = len(x)

# Homogeneous domain
# rho = mu = 1
rho = np.zeros(N) + 1
mu  = np.zeros(N) + 1

# Initialise some arrays of
# velocity and stress at different
# timesteps
s = np.zeros(N, dtype='complex')
v = np.zeros(N, dtype='complex')

sgrad = np.zeros(N, dtype='complex')
vgrad = np.zeros(N, dtype='complex')

sold = np.zeros(N, dtype='complex')
vold = np.zeros(N, dtype='complex')

snew = np.zeros(N, dtype='complex')
vnew = np.zeros(N, dtype='complex')
```

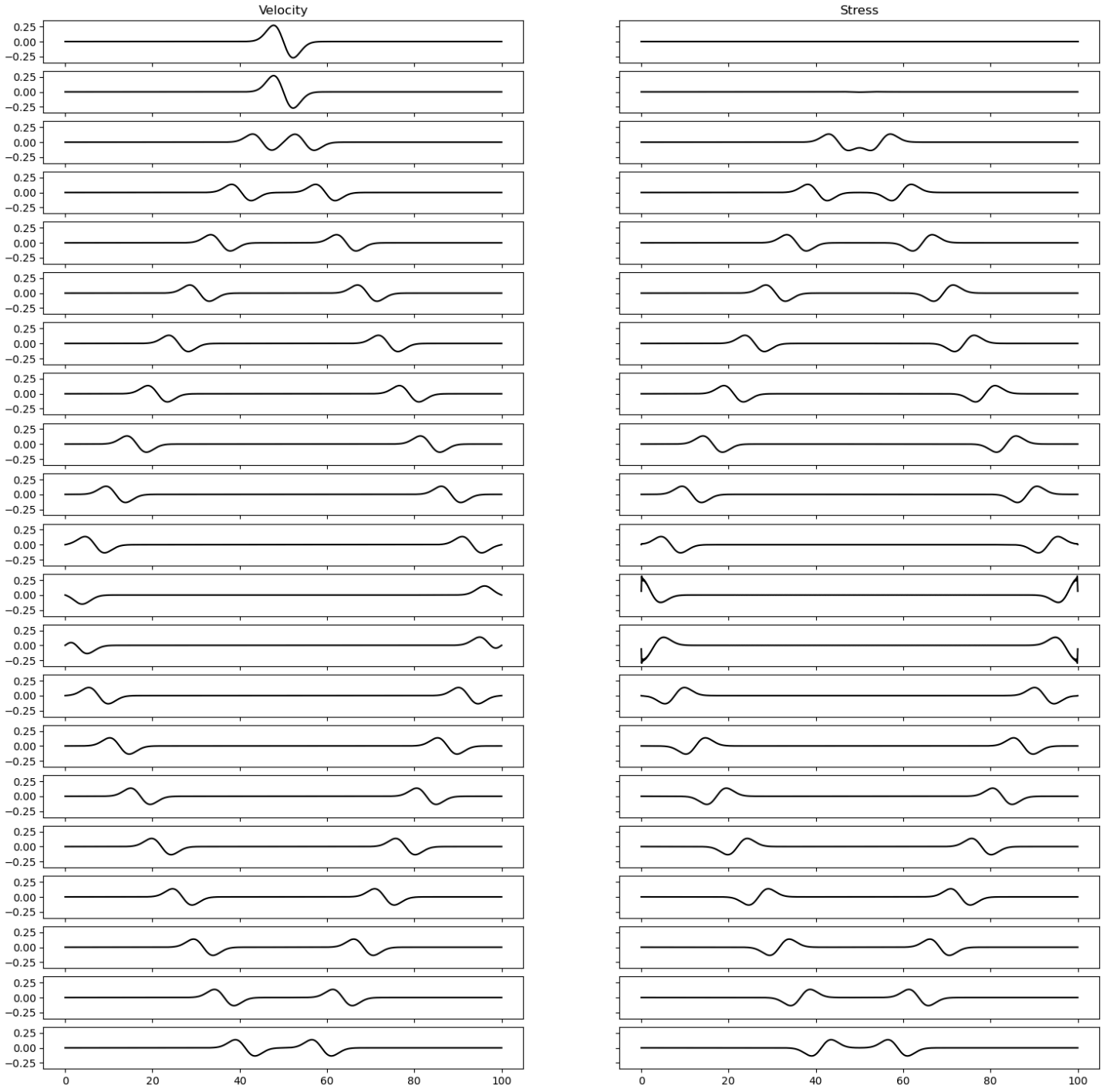
Next let us define the timescheme. For a pseudospectral method, the Courant number should be below $1/(2\pi) \approx 0.16$ to ensure stability. Let us compute the timestep based on this condition, using a Courant number of 0.12.

```
nsteps = 8000
dt     = 0.0315

# Compute dt
cfl           = 0.12
max_wavespeed = np.max((mu/rho)**0.5)

dt = cfl*dx/max_wavespeed
```

Next, we can initialise the boundary conditions and run our simulation. In this case we have set `dirichlet_bc = True` such that the velocity is being fixed at the boundaries.

```
nplots  = 20
fig, ax = plt.subplots(nplots+1, 2, figsize=(18, 18),
                       sharey=True, sharex=True)

# Set the initial condition
v, s       = init_condition(x, v, s)
vold, sold = init_condition(x, vold, sold)

# Plot the initial condition
ax[0,0].plot(x, v.real, 'k')
ax[0,1].plot(x, s.real, 'k')

# Add some labels
ax[0,0].set_ylim([-0.35, 0.35])
ax[0,0].set_title('Velocity')
ax[0,1].set_title('Stress')


# Compute the wavenumbers for FFT:
kfft = np.fft.fftfreq(N, d=dx) * 2 * np.pi

iax    = 1
for istep in range(nsteps):

    # compute gradient of velocity using FFT
    vk    = np.fft.fft(v)
    vgrad = np.fft.ifft(1j * kfft * vk)

    # compute gradient of stress using FFT
    sk    = np.fft.fft(s)
    sgrad = np.fft.ifft(1j * kfft * sk)

    # Compute v-s at new timestep
    vnew = vold + sgrad*(2 * dt / rho)
    snew = sold + vgrad*(2 * dt * mu)

    if dirichlet_bc:
        # Dirichlet
        v[0]     = 0
        v[-1]    = 0
        vnew[0]  = 0
        vnew[-1] = 0
    else:
        # Neumann
        s[0]     = 0
        s[-1]    = 0
        snew[0]  = 0
        snew[-1] = 0

    # Update arrays:
    vold[:] = v[:]
    sold[:] = s[:]
    v[:]    = vnew[:].real
```

```
    s[:]    = snew[:].real

    # Plot some timesteps
    if istep % (nsteps/nplots) == 0:
        ax[iax,0].plot(x, v.real, 'k')
        ax[iax,1].plot(x, s.real, 'k')
        iax +=1
```



We observe the expected behaviour in the velocity, where the wave reflects and changes polarity. Note, however, the non-physical nature of the stress at the boundary introduced by the periodic

boundary condition.

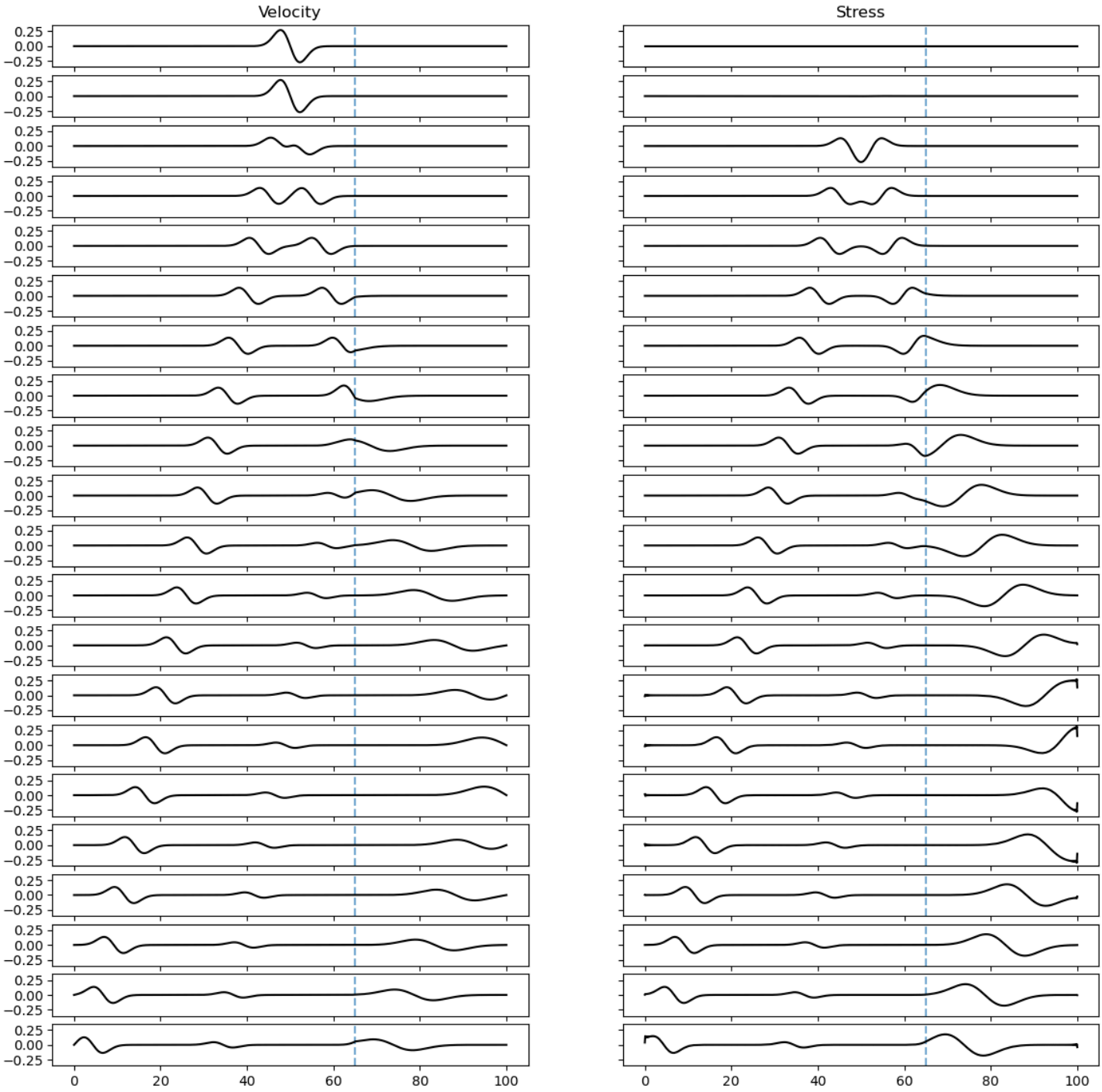
Heterogeneous model:

Next let us turn to the heterogeneous case. The code here is identical, with the only difference being in the definition of the physical parameters. The shear modulus is now equal to 4 above $x = 65$. This affects our CFL condition and therefore the timestep - the maximum wavespeed is now twice as large and therefore the timestep must reduce by 2 to maintain the same CFL condition.

```
rho = np.zeros(N) + 1
mu  = np.zeros(N) + 1
mu[x>65]= 4

max_wavespeed = np.max((mu/rho)**0.5)
dt = cfl*dx/max_wavespeed


# Re-initialise the arrays
s[:]     = 0
v[:]     = 0
sgrad[:] = 0
vgrad[:] = 0
sold[:]  = 0
vold[:]  = 0
snew[:]  = 0
vnew[:]  = 0
vk[:]    = 0
sk[:]    = 0
```

Next, we can re-run the system for the heterogeneous model. The blue dashed line represents the location of the boundary between $\mu = 1$ and $\mu = 4$. At this boundary, we observe a reflection as well as stretching of the waveform to the right-hand-side of the boundary as the speed of propagation increases.

▶ Show code cell source



Problem 6.13

Derivation for 6.109

We start with Equation 6.106

$$\frac{v_j^{n+1} - v_j^{n-1}}{2\Delta t} = \frac{1}{\rho_j \, N\Delta x} \sum_{\ell=0}^{N-1} i\ell\Delta k \, \tilde{\sigma}_\ell^n \, \exp\left(2\pi i \ell j / N\right) \quad .$$

Let us now now consider the relationship of v_j^{n+1} and v_j^{n-1} with their counterparts in the wavenumber domain. From 6.104 we have

$$v_j^{n+1} = \frac{1}{N\Delta x} \sum_{\ell=0}^{N-1} \tilde{v}_\ell^{n+1} \, \exp\left(2\pi i \ell j / N\right),$$
$$v_j^{n-1} = \frac{1}{N\Delta x} \sum_{\ell=0}^{N-1} \tilde{v}_\ell^{n-1} \, \exp\left(2\pi i \ell j / N\right),$$

such that Equation 6.106 may be written as

$$\frac{\frac{1}{N\Delta x} \sum_{\ell=0}^{N-1} \tilde{v}_\ell^{n+1} \, \exp\left(2\pi i \ell j / N\right) - \frac{1}{N\Delta x} \sum_{\ell=0}^{N-1} \tilde{v}_\ell^{n-1} \, \exp\left(2\pi i \ell j / N\right)}{2\Delta t} = \frac{1}{\rho_j \, N\Delta x} \sum_{\ell=0}^{N-1} i\ell\Delta k$$

This can be simplified to

$$\frac{1}{2\Delta t} \sum_{\ell=0}^{N-1} \left(\tilde{v}_\ell^{n+1} - \tilde{v}_\ell^{n-1}\right) \exp\left(2\pi i \ell j / N\right) = \frac{1}{\rho_j} \sum_{\ell=0}^{N-1} i\ell\Delta k \, \tilde{\sigma}_\ell^n \, \exp\left(2\pi i \ell j / N\right) \quad ,$$

where the $\frac{1}{N\Delta x}$ cancels from both sides. Ultimately this is saying that the two discrete Fourier transforms are equal to one another. Due to the fact that the $\exp\left(2\pi i \ell j / N\right)$ form an orthogonal basis for the different values of ℓ, we know that the Fourier coefficient (amplitude) of each term must be equal. That is to say, for each individual value of ℓ the equation

$$\frac{1}{2\Delta t} \left(\tilde{v}_\ell^{n+1} - \tilde{v}_\ell^{n-1}\right) \exp\left(2\pi i \ell j / N\right) = \frac{1}{\rho_j} i\ell\Delta k \, \tilde{\sigma}_\ell^n \, \exp\left(2\pi i \ell j / N\right)$$

must hold. This can be simplified to

$$\frac{1}{2\Delta t} \left(\tilde{v}_\ell^{n+1} - \tilde{v}_\ell^{n-1}\right) = \frac{1}{\rho} i\ell\Delta k \, \tilde{\sigma}_\ell^n \quad ,$$

which is the equation we seek to obtain. Note here that, as the question suggests, we have assumed a homogeneous medium such that $\rho_j = \rho$ everywhere.

Derivation for 6.110

The derivation for the second relationship is essentially identical. We start with Equation 6.107

$$\frac{\sigma_j^{n+1} - \sigma_j^{n-1}}{2\Delta t} = \frac{1}{\mu_j \, N\Delta x} \sum_{\ell=0}^{N-1} i\ell\Delta k \, \tilde{v}_\ell^n \, \exp\left(2\pi i\ell j/N\right) \quad .$$

Next we take the wavenumber versions of the stress,

$$\sigma_j^{n+1} = \frac{1}{N\Delta x} \sum_{\ell=0}^{N-1} \tilde{\sigma}_\ell^{n+1} \, \exp\left(2\pi i\ell j/N\right),$$
$$\sigma_j^{n-1} = \frac{1}{N\Delta x} \sum_{\ell=0}^{N-1} \tilde{\sigma}_\ell^{n-1} \, \exp\left(2\pi i\ell j/N\right),$$

such that Equation 6.107 may be written as

$$\frac{\frac{1}{N\Delta x} \sum_{\ell=0}^{N-1} \tilde{\sigma}_\ell^{n+1} \, \exp\left(2\pi i\ell j/N\right) - \frac{1}{N\Delta x} \sum_{\ell=0}^{N-1} \tilde{\sigma}_\ell^{n-1} \, \exp\left(2\pi i\ell j/N\right)}{2\Delta t} = \frac{\mu_j}{N\Delta x} \sum_{\ell=0}^{N-1} i\ell\Delta k \, \tilde{v}_\ell^n \, \exp\left(2\pi i\ell j/N\right) \quad ,$$

This can be simplified to

$$\frac{1}{2\Delta t} \sum_{\ell=0}^{N-1} \left(\tilde{\sigma}_\ell^{n+1} - \tilde{\sigma}_\ell^{n-1}\right) \exp\left(2\pi i\ell j/N\right) = \mu_j \sum_{\ell=0}^{N-1} i\ell\Delta k \, \tilde{v}_\ell^n \, \exp\left(2\pi i\ell j/N\right) \quad ,$$

where, under the same argument as before, we can reduce this to

$$\frac{1}{2\Delta t} \left(\tilde{\sigma}_\ell^{n+1} - \tilde{\sigma}_\ell^{n-1}\right) = \mu i\ell\Delta k \, \tilde{v}_\ell^n \quad .$$

Problem 6.14

We start with relationship for the standing wave in 6.114,

$$\sin(\omega\Delta t) = \beta\ell\Delta k\Delta t = \beta\, k_\ell\, \Delta t \quad .$$

Next, take the derivative with respect to the wavenumber, k, yielding

$$\frac{\partial\omega}{\partial k}\Delta t\, \cos(\omega\Delta t) = \beta\ell\Delta k\Delta t \quad ,$$

which reduces to

$$\frac{\partial\omega}{\partial k} = \frac{\beta\, k_\ell}{\cos(\omega\Delta t)} \quad .$$

7 Weak Methods

This chapter is about weak methods. The weak form of a PDE is formed by contracting with a test function integrating over the domain.

Chapter Solutions

Problem 7.1

First let us demonstrate that our integral form satisfies the heat equation. To do this, we need to take the second partial derivative of θ with respect to x. Since x is in the limit of the integral, we need to use Leibniz rule, which states:

$$\frac{d}{dx}\left(\int_{a(x)}^{b(x)} f(x,\, y)\,\mathrm{d}y\right) = f(x,\, b(x))\cdot\frac{d}{dx}b(x) - f(x,\, a(x))\cdot\frac{d}{dx}a(x) + \int_{a(x)}^{b(x)}\frac{\partial}{\partial x}f(x,\, y)$$

In our case we will only consider situations in which the integrand function is not a function of the variable, x, with which we are taking the derivative, i.e., $f(x,t) \to f(t)$ and $\frac{d}{dx}f(x,t) = 0$, so the

rule simplifies to

$$\frac{d}{dx}\left(\int_{a(x)}^{b(x)} f(y)\,\mathrm{d}y\right) = f(b(x))\cdot\frac{d}{dx}b(x) - f(a(x))\cdot\frac{d}{dx}a(x)$$

To show that our solution

$$\theta(x) = \theta_1 + (1-x)\,H_0 + \int_x^1\int_0^y h(z)\,dz\,dy$$

satisfies the steady-state equation

$$\partial_x^2\theta + h = 0$$

let us take the first derivative of the solution with respect to x:

$$\frac{d\theta}{dx} = -H_0 + \frac{d}{dx}\left[\int_x^1\int_0^y h(z)\,dz\,dy\right]$$

Using the simplified Leibniz rule above we identify $a(x) = x$, $b(x) = 1$ and $f(y) = \int_0^y h(z)\,dz$ such that the result is

$$\frac{d\theta}{dx} = -H_0 - \int_0^x h(z)\,\mathrm{d}z$$

Employing this a second time, in this case where $b = x$, $a = 0$ and $f(y) = h(z)$ we get

$$\frac{d^2\theta}{dx^2} = -h(x)$$

Hence, we see this satisfies our steady-state equation.

Problem 7.2

Here we compare the solutions in Eqn 7.19 and 7.36 for the case that $h = 1$ across the domain. Starting with Eqn. 7.19

$$\theta(x) = \theta_1 + (1 - x)H_0 + \int_x^1 \int_0^y h(z)\,\mathrm{d}z\,\mathrm{d}y$$

which, for $h(z) = 1$ becomes

$$\theta(x) = \theta_1 + (1 - x)H_0 + \frac{1}{2}(1 - x^2)$$

For Eqn 7.36, the solution

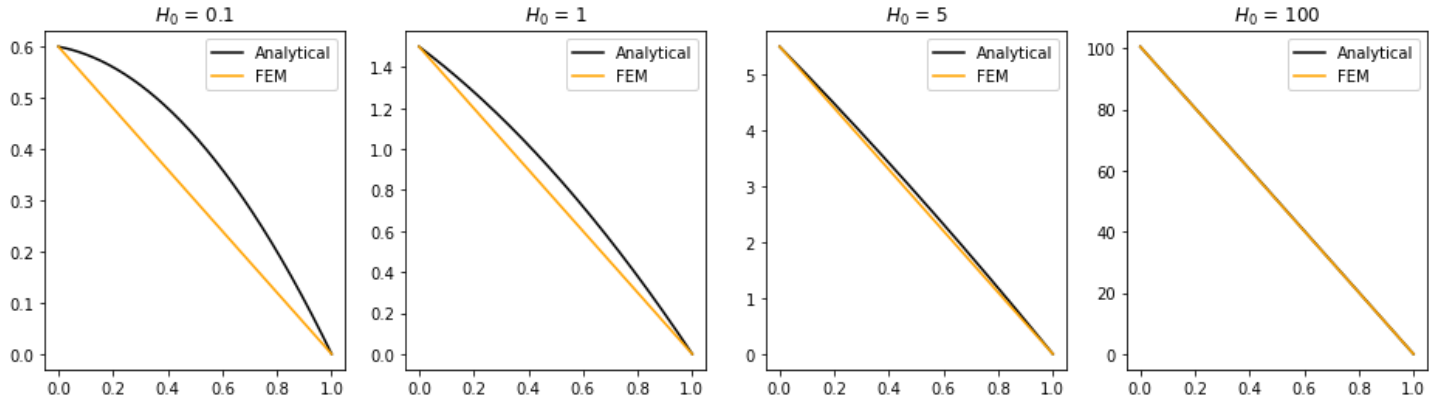
$$\theta(x) = \theta_1 + (1 - x)H_0 + \left[\int_0^1 (1 - y)\,\mathrm{d}y\right](1 - x)$$

becomes

$$\theta(x) = \theta_1 + (1 - x)H_0 + \frac{1}{2}(1 - x)$$

As shown below, the FEM solution is therefore acceptable when H_0 is sufficiently large that the quadratic term in the analytical solution is relatively negligible, but a poor approximation otherwise.

▶ Show code cell source



Problem 7.3

To compute the elements of the diffusivity matrix, let us use the formula of Eqn. 7.31:

$$K_{AB} = a(N_A,\, N_B) = \int_0^1 \left(\partial_x N_A\right)\left(\partial_x N_B\right) \mathrm{d}x$$

Computing the kernel requires taking the integral over the domain, which we can break up into an integral over each of the elements. Let us determine the partial derivative of each of the shape functions in each element:

$$\partial_x N_1(x) = \begin{cases} -2, & 0 \leq x \leq \frac{1}{2} \\ 0, & \frac{1}{2} \leq x \leq 1 \end{cases}$$
$$\partial_x N_2(x) = \begin{cases} 2, & 0 \leq x \leq \frac{1}{2} \\ -2, & \frac{1}{2} \leq x \leq 1 \end{cases}$$
$$\partial_x N_3(x) = \begin{cases} 0, & 0 \leq x \leq \frac{1}{2} \\ 2, & \frac{1}{2} \leq x \leq 1 \end{cases}$$

Computing K_{11}

$$K_{11} = \int_0^{\frac{1}{2}} \left(\partial_x N_1\right)\left(\partial_x N_1\right) \mathrm{d}x + \int_{\frac{1}{2}}^1 \left(\partial_x N_1\right)\left(\partial_x N_1\right) \mathrm{d}x$$

$$K_{11} = \int_0^{\frac{1}{2}} (-2)\,(-2)\, \mathrm{d}x$$

$$K_{11} = \left[4x\right]_0^{\frac{1}{2}} = 2$$

Computing K_{12}

$$K_{12} = \int_0^{\frac{1}{2}} \left(\partial_x N_1\right)\left(\partial_x N_2\right) \mathrm{d}x + \int_{\frac{1}{2}}^1 \left(\partial_x N_1\right)\left(\partial_x N_2\right) \mathrm{d}x$$

$$K_{12} = \int_0^{\frac{1}{2}} (-2)\,(2)\,\mathrm{d}x$$

$$K_{12} = \left[-4x\right]_0^{\frac{1}{2}} = -2$$

Computing K_{22}

$$K_{22} = \int_0^{\frac{1}{2}} (\partial_x N_2)\,(\partial_x N_2)\,\mathrm{d}x + \int_{\frac{1}{2}}^1 (\partial_x N_2)\,(\partial_x N_2)\,\mathrm{d}x$$

$$K_{22} = \int_0^{\frac{1}{2}} (2)\,(2)\,\mathrm{d}x + \int_{\frac{1}{2}}^1 (-2)\,(-2)\,\mathrm{d}x$$

$$K_{22} = \left[4x\right]_0^{\frac{1}{2}} + \left[4x\right]_{\frac{1}{2}}^1 = 2 + 2 = 4$$

Computing Global Heat Supply Vetor (F_A)

F_A is computed using Eqn. 7.30:

$$F_A = \int_0^1 N_A\,h\,\mathrm{d}x + N_A(0)\,H_0 - a(N_A,\,N_{n+1})\theta_1$$

For F_1 therefore this is

$$F_1 = \int_0^1 N_1\,h\,\mathrm{d}x + N_1(0)\,H_0 - a(N_1,\,N_3)\theta_1$$

where $a(N_1,\,N_3)$ is the diffusivity matrix element K_{13}. Note that this could be computed using the Eqn. 7.31 however, since $\partial_x N_1$ is 0 from 0.5 to 1, while $\partial_x N_3$ is 0 from 0 to 0.5 it is clear that this will be zero. So F_1 reduces to

$$F_1 = \int_0^1 N_1 \, h \, \mathrm{d}x + N_1(0) \, H_0$$

where $N_1(0)$ is defined to be 1, such that this further reduces to

$$F_1 = \int_0^1 N_1 \, h \, \mathrm{d}x + \, H_0 \quad .$$

Writing the shape function in the integral explicity for the region it is non-zero (0 to 0.5), and using y as the dummy variable of integration, we can write this as

$$F_1 = \int_0^{\frac{1}{2}} (1 - 2x) \, h(y) \, \mathrm{d}y + \, H_0 \quad .$$

Similarly for computing F_2 let us write the equation from 7.30:

$$F_2 = \int_0^1 N_2 \, h \, \mathrm{d}x + N_2(0) \, H_0 - a(N_2, \, N_3)\theta_1$$

First, let us note that $N_2(0) = 0$ to remove that term:

$$F_2 = \int_0^1 N_2 \, h \, \mathrm{d}x - a(N_2, \, N_3)\theta_1$$

Next we must evaluate $a(N_2, \, N_3)$:

$$a(N_2, \, N_3) = \int_0^1 (\partial_x N_2) \, (\partial_x N_3) \, \mathrm{d}x = \int_{\frac{1}{2}}^1 (\partial_x N_2) \, (\partial_x N_3) \, \mathrm{d}x$$

since $\partial_x N_3$ is 0 between 0 and 0.5. Substituting in the correct values, this yields

$$a(N_2, \, N_3) = \int_{\frac{1}{2}}^1 (-2) \, (2) \, \mathrm{d}x = -2$$

such that the force is

$$F_2 = \int_0^1 N_2 \, h \, \mathrm{d}x + 2\theta_1$$

Finally, substituting in the N_2 shape function for the two elements this yields:

$$F_2 = 2\int_0^{\frac{1}{2}} x \, h \, \mathrm{d}x + 2\int_{\frac{1}{2}}^1 (1-x) \, h \, \mathrm{d}x + 2\theta_1$$

Problem 7.4

Starting with Eqn. 7.28 and taking our answers from Problem 7.3:

$$\Sigma_{B=1}^{n} \mathrm{K}_{AB} \, \mathrm{d}_b = \mathrm{F}_A$$

where the matrix is

$$\mathrm{K} = \begin{bmatrix} 2 & -2 \\ -2 & 4 \end{bmatrix} \quad .$$

The force vector can be written explicitly from Eqns 7.42-7.43, with the assumption of $h = 0$, as

$$\mathrm{F} = \begin{bmatrix} H_0 \\ 2\theta_1 \end{bmatrix}$$

such that the overall linear system of equations is

$$\begin{bmatrix} 2 & -2 \\ -2 & 4 \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} H_0 \\ 2\theta_1 \end{bmatrix} \quad .$$

To determine an expression for $\theta(x)$, we seek the coefficients d_B such that we can write $\theta(x)$ in terms of the shape functions in Eqn. 7.26:

$$\theta(x) = \Sigma_{B=1}^{n} d_B N_B(x) + \theta_1 N_{n+1}(x)$$

where $n = 2$ for this case.

Solving for the coefficients d_B

In this case it is straightforward to solve this linear system of equations by inverting the matrix K. The inverse of this 2 x 2 matrix, K^{-1}, is

$$\begin{bmatrix} 1 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

and so the coefficients may be determined by

$$\begin{bmatrix} d_1 \\ d_2 \end{bmatrix} = \begin{bmatrix} 1 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} H_0 \\ 2\theta_1 \end{bmatrix} \quad .$$

such that

$$\begin{aligned} d_1 &= H_0 + \theta_1 \\ d_2 &= \frac{1}{2} H_0 + \theta_1 \end{aligned}$$

Applying the coefficients d_B

Substituting the coefficients d_1 and d_2 into

$$\theta(x) = d_1 N_1 + d_2 N_2 + \theta_1 N_{n+1}(x)$$

will provide the final solution for θ. Because the form of the shape functions is not the same at each part of the domain (there are two elements) we need to consider each element separately to be certain. We will see, however, that each element provides the same solution for θ.

Element 1: $0 \leq x \leq \frac{1}{2}$

Applying the linear shape functions in 7.39-7.41 as well as the coefficients for d_1 and d_2, while noting d_{n+1} = d_3 = θ_1:

$$\theta(x) = (H_0 + \theta_1)(1 - 2x) + 2x\,(\frac{1}{2}H_0 + \theta_1)$$

since N_3 is 0 here. Hence, this reduces to

$$\theta(x) = \theta_1 + (1 - x)H_0$$

Element 2: $\frac{1}{2} \leq x \leq 1$

In this case N_1 = 0 and so

$$\theta(x) = 2(\frac{1}{2}H_0 + \theta_1)(1 - x) + \theta_1(2x - 1)$$

which also reduces to

$$\theta(x) = \theta_1 + (1 - x)H_0$$

Problem 7.5

The domain contains $n + 1$ nodes. However, final node ($n + 1$) has its temperature fixed at θ_1 due to the boundary condition. As such, we do not need to ever solve for the temperature at this point.

Diffusivity matrix

The diffusivity matrix, $\mathbf{K}$, is therefore $(n \times n)$ and for constant grid spacing Δx has the following components:

$$
\begin{aligned}
K_{1\,1} &= 1/\Delta x \\
K_{A\,A} &= 1/\Delta x_{A-1} + 1/\Delta x_A \\
&= 2/\Delta x \\
K_{A-1\,A} = K_{A\,A-1} &= -1/\Delta x_{A-1} \\
&= -1/\Delta x
\end{aligned}
$$

or in matrix format

$$
\mathbf{K} = \frac{1}{\Delta x}
\begin{bmatrix}
1 & -1 & & & & & \\
-1 & 2 & -1 & & & & \\
 & -1 & 2 & -1 & & & \\
 & & \ddots & \ddots & \ddots & & \\
 & & & -1 & 2 & -1 & \\
 & & & & -1 & 2 & -1 \\
 & & & & & -1 & 2
\end{bmatrix}
$$

Forcing matrix

The forcing matrix components are defined as follows:

Point x_1

$$
F_1 = \frac{1}{\Delta x_1} \int_{x_1}^{x_2} (x_2 - x)\, h(x)\, \mathrm{d}x + H_0
$$

in the case that $h = 0$ this is simply

$$
F_1 = H_0 \quad .
$$

In the case that $h = 1$ this is

$$
\begin{aligned}
F_1 &= \frac{1}{\Delta x_1}\int_{x_1}^{x_2}(x_2 - x)\,\mathrm{d}x + H_0 \\
&= \frac{1}{2\Delta x_1}(x_2 - x_1)^2 + H_0 \\
&= \frac{1}{2}(x_2 - x_1) + H_0 \\
&= \frac{1}{2}\Delta x_1 + H_0
\end{aligned}
$$

Internal points (A = 2, … n)

For all points except the last ($2, \ldots n-1$), the forcing is given by

$$
F_A = \frac{1}{\Delta x_{A-1}}\int_{x_{A-1}}^{x_A}(x - x_{A-1})\,h(x)\,\mathrm{d}x \;+\; \frac{1}{\Delta x_A}\int_{x_A}^{x_{A+1}}(x_{A+1} - x)\,h(x)\,\mathrm{d}x
$$

When $h = 0$ this produces $F_A = 0$. When $h = 1$ this produces

$$
\begin{aligned}
F_1 &= \frac{1}{2\Delta x_{A-1}}(x_A - x_{A-1})^2 + \frac{1}{2\Delta x_A}(x_{A+1} - x_A)^2 \\
&= \frac{1}{2}\Delta x_{A-1} + \frac{1}{2}\Delta x_A
\end{aligned}
$$

which for a constant Δx is simply equal to Δx .

The n'th node contains contributions from the last element and θ_1, given in 7.54 as

$$
F_n = \frac{1}{\Delta x_{n-1}}\int_{x_{n-1}}^{x_n}(x - x_{n-1})\,h(x)\,\mathrm{d}x + \frac{1}{\Delta x_n}\int_{x_n}^{x_{n+1}}(x_{n+1} - x)\,h(x)\,\mathrm{d}x \;+\; \frac{\theta_1}{\Delta x_n} \quad .
$$

For the case that $h = 0$ this reduces to

$$
F_n = \frac{\theta_1}{\Delta x_n},
$$

where as for $h(x) = 1$ it is

$$F_n = \frac{\Delta x_{n-1}}{2} + \frac{\Delta x_n}{2} + \frac{\theta_1}{\Delta x_n},$$

Case 1: h = 0

For 10 elements ($n = 10$) the equations then becomes

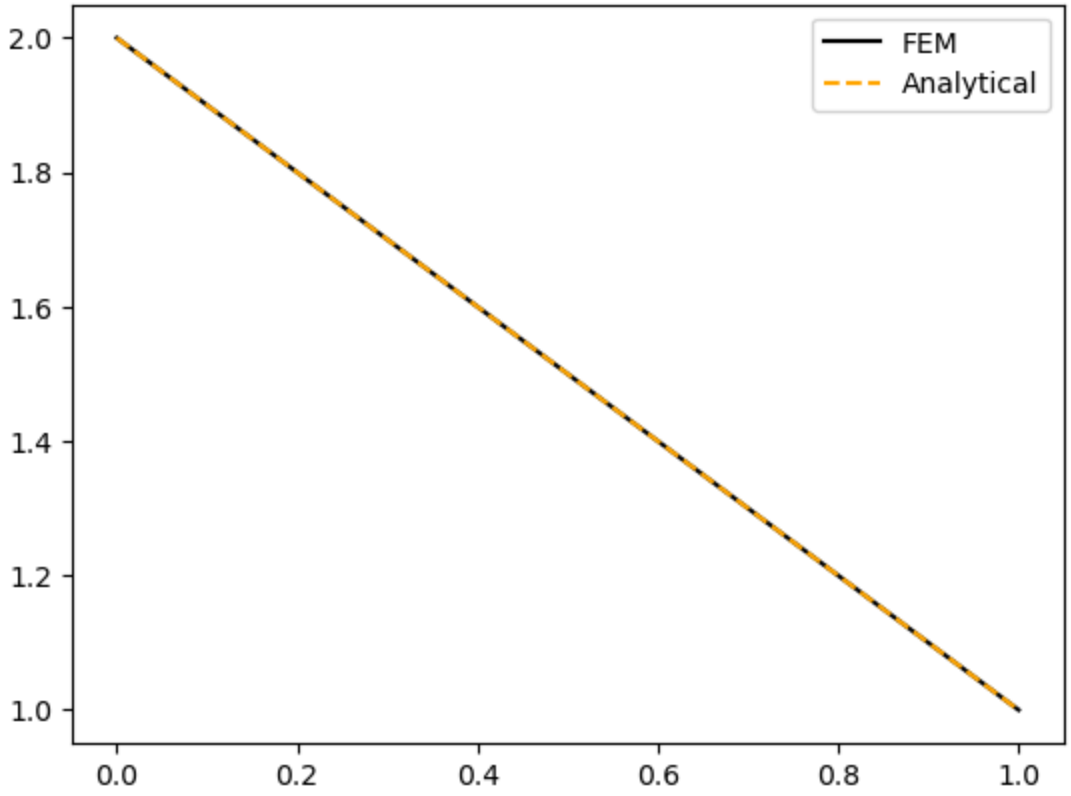
$$\frac{1}{\Delta x}\begin{bmatrix} 1 & -1 & & & & & & & & \\ -1 & 2 & -1 & & & & & & & \\ & -1 & 2 & -1 & & & & & & \\ & & -1 & 2 & -1 & & & & & \\ & & & -1 & 2 & -1 & & & & \\ & & & & -1 & 2 & -1 & & & \\ & & & & & -1 & 2 & -1 & & \\ & & & & & & -1 & 2 & -1 & \\ & & & & & & & -1 & 2 & -1 \end{bmatrix} \mathbf{\Theta} = \begin{bmatrix} H_0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ \frac{\theta_1}{\Delta x} \end{bmatrix}$$

since H_0 = 1

The solution is then:

```
import numpy as np
import matplotlib.pyplot as plt

L = 1     # Domain 0 to L
n = 10    # Number of elements

# Boundary conditions
theta_1 = 1
H0      = 1

x = np.linspace(0, L, n+1)

dx = x[1:] - x[:-1]  # Delta x in each element

# Create matrix K
diag     =  2 * np.ones(n)     # Main diagonal
off_diag = -1 * np.ones(n-1)  # Off-diagonals
K = np.diag(diag) + np.diag(off_diag, k=1) + np.diag(off_diag, k=-1)

# Edit the first diagonal element
K[0,0]   = 1

# Remember division by dx for K matrix
K = K/dx

# Create vector f:

F = np.zeros(n)
F[0] = H0
F[-1] = theta_1 / dx[-1]


# Compute inverse matrix K^-1
Kinv = np.linalg.inv(K)

# Compute theta
theta = np.matmul(Kinv, F)

# Theta array for whole domain contains the n+1 point also:
total_theta = np.zeros(n+1)
total_theta[:n] = theta
total_theta[-1] = theta_1



# Plot for comparison
fig, ax = plt.subplots()

# Plot FEM
ax.plot(x, total_theta, 'k')

# Plot analytical
analytical = 1 + (1-x)*H0
```

```
ax.plot(x, analytical, '--', color='orange')
ax.legend(['FEM', 'Analytical']);
```



Case 2: h = 0

Now lets look at the case when $h = 1$. Remember that h doesn't affect the diffusivity matrix so all we need to do is compute the new $\mathbf{F}$ vector

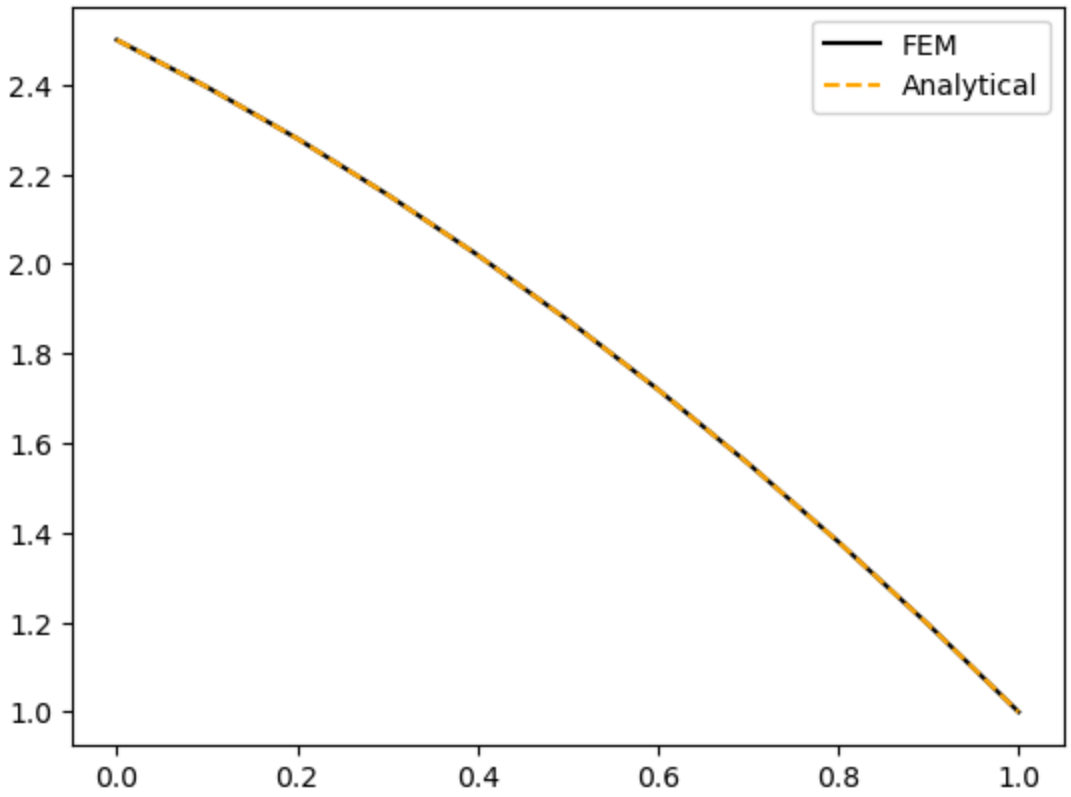
```
# Create a new F vector
F = np.zeros(n)

for inode in range(n):
    if inode == 0:
        F[inode] = H0 + dx[0]/2
    else:
        F[inode] = 0.5 * (dx[inode-1] + dx[inode])


    # The last node (n) also gets an extra contribution
    # note the python indexing from 0 here means its 'n-1'
    if inode == n-1:
        F[inode] += theta_1/dx[inode]


# Now we can compute the temperature field:
# Compute theta
theta = np.matmul(Kinv, F)

# Theta array for whole domain contains the n+1 point also:
total_theta = np.zeros(n+1)
total_theta[:n] = theta
total_theta[-1] = theta_1
```

Let us compare against an analytical solution (7.19) for $h(x) = 1$ which ends up being

$$\theta(x) = \theta_1 + (1-x)H_0 + \frac{1}{2}(1-x^2) \quad .$$

Note how the internal heating contributes an extra quadratic term compared to the case when $h = 0$.

```
# Plot for comparison
fig, ax = plt.subplots()

# Plot FEM
ax.plot(x, total_theta, 'k')

# Plot analytical
analytical = theta_1 + (1-x)*H0 + 0.5*(1 - x**2)
ax.plot(x, analytical, '--', color='orange')
ax.legend(['FEM', 'Analytical']);
```



Problem 7.6

The local element solution has the spatial mapping defined as

$$x(\xi) = \sum_{a=1}^{2} x_{A+a-1} N_a(\xi)$$

for which the linear shape functions are

$$N_a(\xi) = \frac{1}{2}[1 + (-1)^a \xi]$$

Let us expand the mapping with these shape functions by writing the summation explicitly

$$x(\xi) = x_A N_1(\xi) + x_{A+1} N_2(\xi)$$

and now substituting in the shape functions:

$$N_1(\xi) = \frac{1}{2}[1-\xi]$$
$$N_2(\xi) = \frac{1}{2}[1+\xi]$$

the spatial mapping is then

$$x(\xi) = \frac{x_A}{2}[1-\xi] + \frac{x_{A+1}}{2}[1+\xi]$$

Verifying the values at $\xi = \pm 1$:

At $\xi = -1$ the mapping becomes

$$x(\xi) = \frac{2x_A}{2} = x_A$$

as required. Similarly at $\xi = 1$ the mapping becomes

$$x(\xi) = \frac{2x_{A+1}}{2} = x_{A+1}$$

Problem 7.7

We seek the jacobian ($\frac{\partial x}{\partial \xi}$) of the the spatial mapping

$$x(\xi) = \sum_{a=1}^{2} x_{A+a-1} N_a(\xi)$$

with linear shape functions

$$N_a(\xi) = \frac{1}{2}[1 + (-1)^a \xi].$$

First, let us take the derivative of the mapping with respect to ξ.

$$\frac{dx}{d\xi} = \sum_{a=1}^{2} x_{A+a-1} \frac{d}{d\xi} N_a(\xi) \quad .$$

We therefore need the derivatives of the shape functions:

$$\frac{d}{d\xi} N_a(\xi) = \frac{1}{2}(-1)^a$$

such that the derivatives of the two shape functions are

$$\begin{aligned} \frac{dN_1}{d\xi} &= -\frac{1}{2} \\ \frac{dN_2}{d\xi} &= +\frac{1}{2} \end{aligned}$$

We may then write the summation in the jacobian explicitly as

$$\begin{aligned} \frac{dx}{d\xi} &= x_A \frac{dN_1}{d\xi} + x_{A+1} \frac{dN_2}{d\xi} \\ &= \frac{1}{2}[x_{A+1} - x_A] \\ &= \frac{1}{2}\Delta x_A \end{aligned}$$

as required.

Problem 7.8

We seek the inverse of the mapping defined in 7.55,

$$x(\xi) = \sum_{a=1}^{2} x_{A+a-1} N_a(\xi)$$

where

$$N_a(\xi) = \frac{1}{2}[1 + (-1)^a \xi].$$

To do this, let us write out the mapping explicitly:

$$\begin{aligned} x(\xi) &= x_A N_1(\xi) + x_{A+1} N_2(\xi) \quad , \\ x(\xi) &= \frac{1}{2}\Big(x_A\,[1-\xi] + x_{A+1}\,[1+\xi]\Big). \end{aligned}$$

Multiplying by 2 and collecting terms this rearranges to

$$2x = x_A + x_{A+1} + \xi\,(x_{A+1} - x_A)$$

and therefore

$$\xi = \frac{2x - x_A - x_{A+1}}{x_{A+1} - x_A} = \frac{2x - x_A - x_{A+1}}{\Delta x_A}$$

as required. We can then see that at $x = x_A$ and $x = x_{A+1}$ this becomes

$$\begin{aligned} \xi(x_A) &= \frac{x_A - x_{A+1}}{x_{A+1} - x_A} = -1 \\ \xi(x_{A+1}) &= \frac{x_{A+1} - x_A}{x_{A+1} - x_A} = +1 \quad . \end{aligned}$$

Problem 7.9

The local linear shape functions are defined in 7.56 as

$$N_a(\xi) = \frac{1}{2}[1 + (-1)^a\,\xi]$$

and the local diffusivity matrix is defined in 7.63 as

$$k_{ab}^{A} = \int_{x_A}^{x_{A+1}} \left(\partial_x N_a\right)\left(\partial_x N_b\right) \mathrm{d}x \quad .$$

To get the derivatives of the shape functions, we need to use the chain rule

$$\partial_x N_a = \frac{\mathrm{d}N_a}{\mathrm{d}\xi}\frac{\mathrm{d}\xi}{\mathrm{d}x}.$$

yielding

$$k_{ab}^{A} = \int_{x_A}^{x_{A+1}} \frac{\mathrm{d}N_a}{\mathrm{d}\xi}\frac{\mathrm{d}N_b}{\mathrm{d}\xi}\left[\frac{\mathrm{d}\xi}{\mathrm{d}x}\right]^2 \mathrm{d}x \quad .$$

Next we can change the variable of integration as

$$\mathrm{d}x = \frac{\mathrm{d}x}{\mathrm{d}\xi}\,\mathrm{d}\xi$$

and the limits

$$\begin{aligned} x = x_A &\rightarrow \xi = -1 \\ x = x_{A+1} &\rightarrow \xi = +1 \end{aligned}$$

to produce

$$k_{ab}^{A} = \int_{-1}^{1} \frac{\mathrm{d}N_a}{\mathrm{d}\xi}\frac{\mathrm{d}N_b}{\mathrm{d}\xi}\left[\frac{\mathrm{d}\xi}{\mathrm{d}x}\right]^2 \frac{\mathrm{d}x}{\mathrm{d}\xi}\,\mathrm{d}\xi = \int_{-1}^{1} \frac{\mathrm{d}N_a}{\mathrm{d}\xi}\frac{\mathrm{d}N_b}{\mathrm{d}\xi}\frac{\mathrm{d}\xi}{\mathrm{d}x}\;\mathrm{d}\xi$$

as in 7.64. Now let us evaluate this integral using

$$\frac{\mathrm{d}N_a}{\mathrm{d}\xi} = \frac{1}{2}(-1)^a \quad ,$$

$$\frac{\mathrm{d}N_b}{\mathrm{d}\xi} = \frac{1}{2}(-1)^b \quad ,$$

$$\frac{\mathrm{d}\xi}{\mathrm{d}x} = \frac{2}{\Delta x_A} \quad ,$$

which gives

$$\begin{aligned} k_{ab}^A &= \frac{1}{4}\int_{-1}^{1}(-1)^a(-1)^b\frac{2}{\Delta x_A}\ \mathrm{d}\xi \\ &= \frac{1}{4}\int_{-1}^{1}(-1)^{a+b}\frac{2}{\Delta x_A}\ \mathrm{d}\xi\ = \frac{(-1)^{a+b}}{\Delta x_A} \end{aligned}$$

Problem 7.10

Let us consider the case with n elements (nodes from x_1 to x_{n+1}). Because the final node ($n+1$) has its temperature is fixed, we do not need to solve for this node. The elements of the local diffusivity matrix for nodes $1, \ldots n$ are given in 7.60-7.62.

$$\begin{aligned} K_{11} &= k_{11}^1 \\ K_{A-1\,A} &= k_{12}^{A-1} \\ K_{A\,A} &= k_{22}^{A-1} + k_{11}^A \end{aligned}$$

In Problem 7.9, it is shown that

$$k_{ab}^A = \frac{(-1)^{a+b}}{\Delta x_A}$$

which holds for $A = 1,\ n$ inclusive.

Similarly the heat supply vector may be written as

$$
\begin{aligned}
F_1 &= f_1^1 + H_0 \\
F_A &= f_2^{A-1} + f_1^A \\
F_{n+1} &= f_2^n + \frac{\theta_1}{\Delta x_n}
\end{aligned}
$$

where

$$
f_a^A = \int_{x_A}^{x_{A+1}} N_a \, h \, \mathrm{d}x
$$

For example, the local forcing vectors from the first 3 elements would be:

$$
\begin{bmatrix} F_1 \\ F_2 \end{bmatrix} \leftarrow \underbrace{\begin{bmatrix} f_1^1 + H_0 \\ f_2^1 \end{bmatrix}}_{\text{Element 1 contribution}} \quad , \quad \begin{bmatrix} F_2 \\ F_3 \end{bmatrix} \leftarrow \underbrace{\begin{bmatrix} f_1^2 \\ f_2^2 \end{bmatrix}}_{\text{Element 2 contribution}} \quad , \quad \begin{bmatrix} F_3 \\ F_4 \end{bmatrix} \leftarrow \underbrace{\begin{bmatrix} f_1^3 \\ f_2^3 \end{bmatrix}}_{\text{Element 3 contribution}}
$$

So now let us determine the values of f_a^A for the cases of $h = 0$ and $h = 1$ by solving Eqn 7.69,

$$
f_a^A = \int_{x_A}^{x_{A+1}} N_a \, h(x) \, \mathrm{d}x \quad .
$$

Evidently $f_a^A = 0$ when $h = 0$. When $h(x) = 1$, we get

$$
\begin{aligned}
f_a^A &= \int_{x_A}^{x_{A+1}} \frac{1}{\Delta x_A} (x_{A+1} - x) \, \mathrm{d}x \\
&= \frac{\Delta x_A}{2}
\end{aligned}
$$

Case 1: h = 0

Note here that the textbook has indices going from 1 to n. Python indexes starting at 0 so our indices will be from 0 to n and the first element of a matrix will be K_{00}.

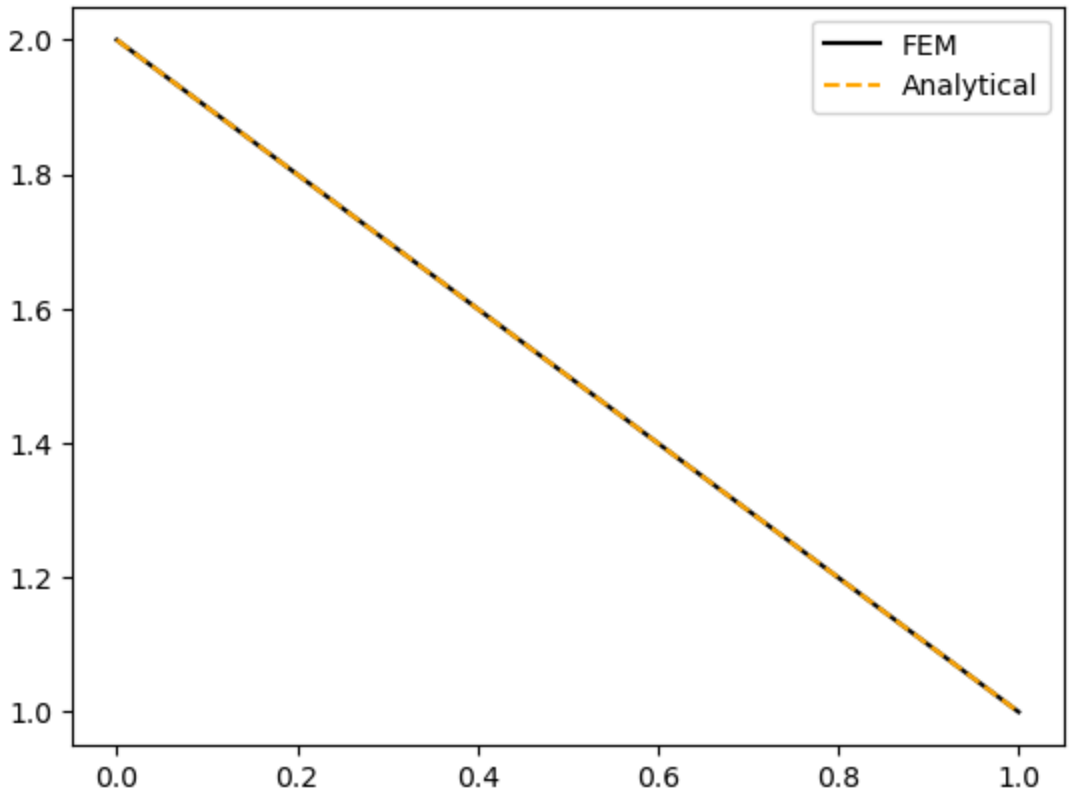
```
import numpy as np
import scipy.linalg as sl
import matplotlib.pyplot as plt
# Boundary conditions:
theta_1 = 1
H0      = 1

nelem   = 10        # number of elements, n
np1     = nelem + 1 # n+1

# Global domain node locations
x = np.linspace(0, 1, np1)

# Compute Delta x for each element:
dx = x[1:] - x[:-1]

# Create our diffusivity matrix:
K = np.zeros((nelem, nelem))

# Create global heating matrix:
F = np.zeros(nelem)

for ielem in range(nelem):
    # Compute local diffusivity matrix for this element:
    # using 7.65
    kloc = np.zeros((2,2))
    kloc[0,0] = 1
    kloc[1,1] = 1
    kloc[0,1] = -1
    kloc[1,0] = -1

    # Multiply by grid spacing
    kloc *= (1/dx[ielem])

    # Add to the global matrix:
    if ielem == nelem -1:
        # For the last element we are ignoring the
        #contributions from the n + 1 node
        K[ielem, ielem] += kloc[0,0]
    else:
        K[ielem:ielem+2, ielem:ielem+2] += kloc


    # Create local forcing vector for element:
    # In the case that h = 0, we know floc is also
    # 0 except at the boundaries
    floc = np.zeros(2)
    if ielem == 0:
        floc[0] += H0
    elif ielem == nelem-1:
        floc[0] += theta_1/dx[ielem]
```

```
    # Add to the global vector:
    # For the nth element we need to edit the
    # indexing since only the nth (not n+1th) node
    # is added
    if ielem == nelem-1:
        F[ielem] += floc[0]
    else:
        F[ielem:ielem+2] += floc


# Compute d as K^-1 * F
Kinv = np.linalg.inv(K)
d = np.matmul(Kinv, F)

# The total temperature field includes the n+1 node
# which is equal to theta_1
temp = np.zeros(np1)
temp[:-1] = d
temp[-1]  = theta_1
```

We can now compare against the analytical solution

$$\theta(x) = \theta_1 + (1-x)H_0$$

```
# Plot against analytical
# When h is 0:
analytical = theta_1 + (1-x)*H0

fig, ax = plt.subplots()

ax.plot(x, temp, 'k')
ax.plot(x, analytical, '--', color='orange')
ax.legend(['FEM', 'Analytical']);
```



Case 2: h = 1

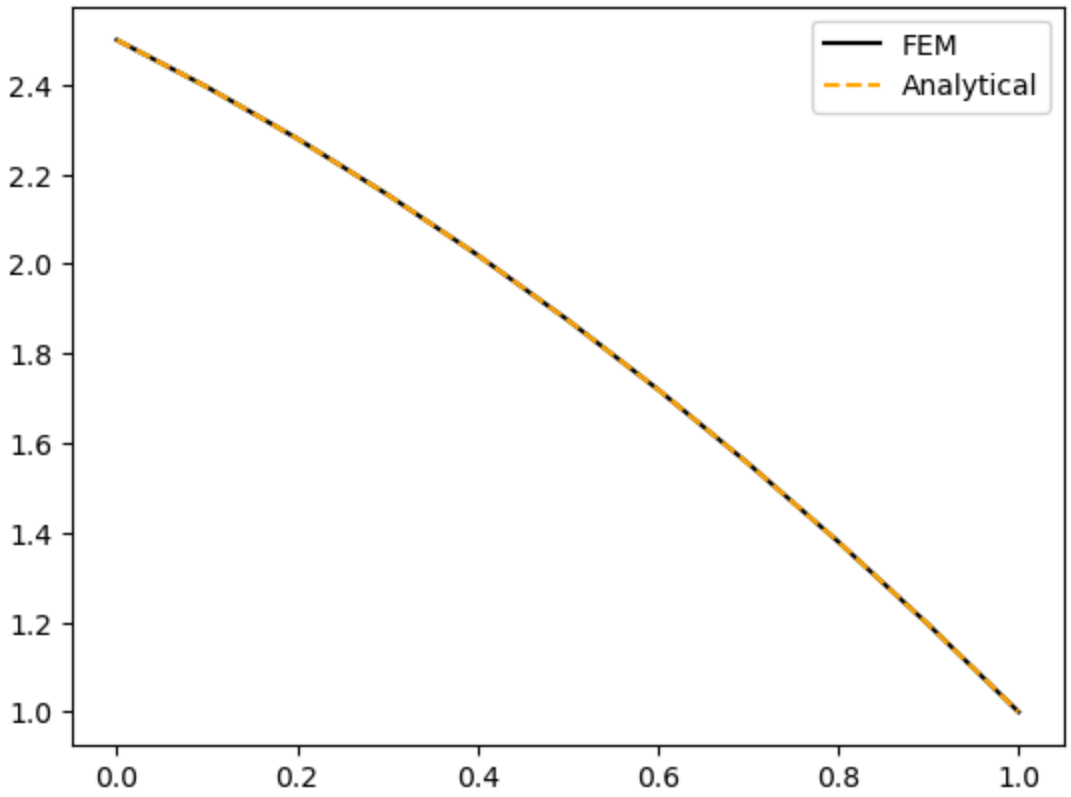
Now lets consider the case when $h = 1$. This does not affect the diffusivity matrix, only the force matrix, so lets update that and rerun the code:

```
# Reinitialise global heating matrix:
F = np.zeros(nelem)

for ielem in range(nelem):
    # Create local forcing vector for element:
    # In the case that h = 0, we know floc is also 0
    # except at the boundaries
    floc = np.zeros(2)
    if ielem == 0:
        floc[0] += H0 + dx[0]/2
        floc[1] += dx[0]/2
    elif ielem == nelem-1:
        floc[0] += (theta_1/dx[ielem]) + dx[-1]/2
    else:
        floc[0] +=  dx[ielem-1]/2
        floc[1] +=  dx[ielem]/2

    # Add to the global vector:
    # For the nth element we need to edit the indexing
    # since only the nth (not n+1th) node is added
    if ielem == nelem-1:
        F[ielem] += floc[0]
    else:
        F[ielem:ielem+2] += floc

d = np.matmul(Kinv, F)


# The total temperature field includes the n+1 node
# which is equal to theta_1
temp = np.zeros(np1)
temp[:-1] = d
temp[-1]  = theta_1
```

Finally, let us compare against the analytical solution (7.19) for $h(x) = 1$ which ends up being

$$\theta(x) = \theta_1 + (1-x)H_0 + \frac{1}{2}(1-x^2) \quad .$$

```
fig, ax = plt.subplots()

# Analytical solution in this case

theta = theta_1 + (1-x)*H0 +  0.5*(1 - x**2)

ax.plot(x, temp, 'k');
ax.plot(x, theta, '--', color='orange');
ax.legend(['FEM', 'Analytical']);
```



Problem 7.11

The global weak form of the dynamic heat equation is given in Eqn. 7.80 as

$$\int_0^1 \tilde{\theta}\,\partial_t\theta\,\mathrm{d}x = -\int_0^1 \alpha\,(\partial_x\tilde{\theta})\,(\partial_x\theta)\,\mathrm{d}x \;+\; \int_0^1 \tilde{\theta}\,h\,\mathrm{d}x \;+\alpha\,\tilde{\theta}\,\partial_x\theta\,|_0^1 \quad .$$

Writing the temperature field, $\theta(x,\,t)$, and test function, $\tilde{\theta}(x,\,t)$, as in Eqns 7.26 and 7.23,

$$\theta(x,\,t) = \sum_{B=1}^{n} d_B\,N_B(x) \;+\; \theta_1 N_{n+1}(x)$$

$$\tilde{\theta}(x,\,t) = \sum_{A=1}^{n} c_A\,N_A(x)$$

we may then express the temporal gradients of these functions:

$$
\begin{aligned}
\partial_t \theta(x,\, t) &= \sum_{B=1}^{n} \dot{d}_B \, N_B(x) \quad , \\
\partial_x \theta(x,\, t) &= \sum_{B=1}^{n} d_B \, \partial_x N_B(x) \, + \, \theta_1 \partial_x N_{n+1} \quad , \\
\partial_x \tilde{\theta}(x,\, t) &= \sum_{A=1}^{n} c_A \, \partial_x N_A \quad ,
\end{aligned}
$$

where a dot above a variable indicates a derivative with respect to time. Note here that only the weights c_A and d_B are time dependent, while only the shape functions N_i are spatially-varying. Finally, θ_1 has no time dependence due to the boundary condition 7.81,

$$
\theta(1,\, t) = \theta_1 \quad .
$$

Substituting in these summations to the heat equation produces

$$
\begin{aligned}
\int_0^1 \left(\sum_{A=1}^{n} c_A \, N_A \right) \left(\sum_{B=1}^{n} \dot{d}_B \, N_B \right) \mathrm{d}x = & \\
& - \int_0^1 \alpha \left(\sum_{A=1}^{n} c_A \, \partial_x N_A \right) \left(\sum_{B=1}^{n} d_B \, \partial_x N_B \, + \, \theta_1 \partial_x N_{n+1} \right) \mathrm{d}x \\
& + \int_0^1 \sum_{A=1}^{n} c_A \, N_A \, h \, \mathrm{d}x \\
& + \alpha \left[\tilde{\theta} \, \partial_x \theta \right]_0^1 \quad .
\end{aligned}
$$

Let us consider the different terms separately:

Left-hand side

The left-hand side can be re-ordered by removing the weights, which aren't spatially dependent, from the integral:

$$
\begin{aligned}
&\int_0^1 \left(\sum_{A=1}^{n} c_A\, N_A\right)\, \left(\sum_{B=1}^{n} \dot{d}_B\, N_B(x)\right) \mathrm{d}x \\
&= \sum_{A=1}^{n}\sum_{B=1}^{n} c_A\, \dot{d}_B \int_0^1 N_A(x)\, N_B(x)\, \mathrm{d}x \\
&= \sum_{A=1}^{n}\sum_{B=1}^{n} c_A\, M_{AB}\, \dot{d}_B
\end{aligned}
$$

Right-hand side - Term 1:

We can similarly rearrange

$$
\begin{aligned}
&-\int_0^1 \alpha\, \left(\sum_{A=1}^{n} c_A\, \partial_x N_A\right)\, \left(\sum_{B=1}^{n} d_B\, \partial_x N_B \;+\; \theta_1 \partial_x N_{n+1}\right) \mathrm{d}x \\
&= -\int_0^1 \alpha\, \left(\sum_{A=1}^{n} c_A\, \partial_x N_A\right)\, \left(\sum_{B=1}^{n} d_B\, \partial_x N_B\right) \;+\; \alpha\, \left(\sum_{A=1}^{n} c_A\, \partial_x N_A\right) \theta_1 \partial_x N_{n+1}\, \mathrm{d}x \\
&= -\sum_{A=1}^{n}\sum_{B=1}^{n} c_A d_B \int_0^1 \alpha\, \left(\partial_x N_A\right)\, \left(\partial_x N_B\right) \mathrm{d}x \;-\; \sum_{A=1}^{n} c_A \int_0^1 \alpha\, \left(\partial_x N_A\right) \theta_1\, \left(\partial_x N_{n+1}\right) \mathrm{d}x
\end{aligned}
$$

With Eqn. 7.86 this can then be written as

$$
= -\sum_{A=1}^{n}\sum_{B=1}^{n} c_A K_{AB}\, d_B \;-\; \sum_{A=1}^{n} c_A\, \theta_1 \int_0^1 \alpha\, \left(\partial_x N_A\right) \left(\partial_x N_{n+1}\right) \mathrm{d}x
$$

Right-hand side - Term 2:

$$
\begin{aligned}
&\int_0^1 \sum_{A=1}^{n} c_A\, N_A\, h\, \mathrm{d}x \\
&= \sum_{A=1}^{n} c_A \int_0^1 N_A\, h\, \mathrm{d}x
\end{aligned}
$$

Right-hand side - Term 3:

Upon imposing the constraint of 7.21 that

$$\tilde{\theta}(1) = 0 \quad ,$$

this term reduces to

$$\begin{aligned} \alpha \left[\tilde{\theta}\, \partial_x \theta \right]_0^1 &= \alpha(1)\, \tilde{\theta}(1)\, \partial_x \theta(1) - \alpha(0)\, \tilde{\theta}(0)\, \partial_x \theta(0) \\ &= -\alpha(0)\, \tilde{\theta}(0)\, \partial_x \theta(0) \end{aligned}$$

and using the boundary condition in Eqn 7.82 that

$$\partial_x \theta(0,\, t) = -H_0 \quad ,$$

this becomes

$$\begin{aligned} &= \alpha(0)\, H_0\, \tilde{\theta}(0) \\ &= \sum_{A=1}^{n} c_A\, N_A(0)\, \alpha(0)\, H_0 \end{aligned}$$

Re-assembling the equation

We can now write our global heat equation as

$$\begin{aligned} \sum_{A=1}^{n} \sum_{B=1}^{n} c_A\, M_{AB}\, \dot{d}_B = &- \sum_{A=1}^{n} \sum_{B=1}^{n} c_A K_{AB}\, d_B \; - \; \sum_{A=1}^{n} c_A\, \theta_1 \int_0^1 \alpha \; (\, \partial_x N_A)\, (\partial_x N_{n+1})\, \mathrm{d}x \\ &+ \sum_{A=1}^{n} c_A \int_0^1 N_A\, h\, \mathrm{d}x + \sum_{A=1}^{n} c_A\, N_A(0)\, \alpha(0)\, H_0 \quad . \end{aligned}$$

By moving the term containing the diffusivity matrix to the LHS, we may write this as

$$\sum_{A=1}^{n} c_A \left\{ \sum_{B=1}^{n} M_{AB}\, \dot{d}_B + \sum_{B=1}^{n} K_{AB}\, d_B \right\} = \sum_{A=1}^{n} c_A \Big\{ -\theta_1 \int_0^1 \alpha\, (\partial_x N_A)\, (\partial_x N_{n+1})\, \mathrm{d}x + \int_0^1 N_A\, h\, \mathrm{d}x + N_A(0)\, \alpha(0)\, H_0 \Big\} \quad .$$

which can be simplified to

$$\sum_{A=1}^{n} c_A \left\{ \sum_{B=1}^{n} M_{AB}\, \dot{d}_B + \sum_{B=1}^{n} K_{AB}\, d_B \right\} = \sum_{A=1}^{n} c_A F_A \quad .$$

in which

$$F_A = -\theta_1 \int_0^1 \alpha\, (\partial_x N_A)\, (\partial_x N_{n+1})\, \mathrm{d}x + \int_0^1 N_A\, h\, \mathrm{d}x + N_A(0)\, \alpha(0)\, H_0$$

Note here the minor differences between the definition above and Eqn 7.87

Probing the test function

For the weak form to hold, the equations must hold for any test function $\tilde{\theta}$. Hence, we can probe the system of equations for different weak functions. This means that for any C_A

$$\sum_{A=1}^{n} c_A \left\{ \sum_{B=1}^{n} M_{AB}\, \dot{d}_B + \sum_{B=1}^{n} K_{AB}\, d_B \right\} = \sum_{A=1}^{n} c_A F_A \quad .$$

must hold. To probe this, let us consider each case $i = 1, n$ for which

$$C_A = \begin{cases} 1 & A = i \\ 0 & A \neq i \end{cases} \quad .$$

For this to hold, it requires that

$$\sum_{B=1}^{n} M_{AB}\,\dot{d}_B + \sum_{B=1}^{n} K_{AB}\,d_B = F_A$$

for every A.

Problem 7.12

Eqn 7.85 tells us that

$$M_{AB} = \int_0^1 N_A\,N_B\,\mathrm{d}x$$

M_{11}

This is the first node in the global domain, meaning it will only have contributions from the first element. Given that, from Eqns 7.56 and 7.57,

$$\begin{aligned} N_1 &= \frac{1}{2}[1-\xi] \\ \frac{\mathrm{d}x}{\mathrm{d}\xi} &= \frac{1}{2}\Delta x_1 \end{aligned}$$

we may write the integral as

$$\begin{aligned} M_{11} &= \int_{-1}^{1} N_1\,N_1 \frac{\mathrm{d}x}{\mathrm{d}\xi}\,\mathrm{d}\xi \\ &= \int_{-1}^{1} \frac{\Delta x_1}{8}[1-\xi]^2\,\mathrm{d}\xi \\ &= \frac{\Delta x_1}{8}\left[\frac{\xi^3}{3} - \xi^2 + \xi\right]_{-1}^{1} \\ &= \frac{\Delta x_1}{3} \quad . \end{aligned}$$

Note here the important change in the integration limits from $x = [0,1]$ to $\xi = [-1,1]$.

$M_{A-1,\,A}$

This part in the global matrix is only contributed to by a single element, $A-1$. Given that, from Eqns 7.56 and 7.57,

$$
\begin{aligned}
N_1 &= \frac{1}{2}[1-\xi] \\
N_2 &= \frac{1}{2}[1+\xi] \\
\frac{\mathrm{d}x}{\mathrm{d}\xi} &= \frac{1}{2}\Delta x_{A-1}
\end{aligned}
$$

we may write the integral as

$$
\begin{aligned}
M_{A-1,\,A} &= \int_{-1}^{1} N_1\, N_2 \frac{\mathrm{d}x}{\mathrm{d}\xi}\,\mathrm{d}\xi \\
&= \int_{-1}^{1} \frac{\Delta x_{A-1}}{8}(1-\xi)\,(1+\xi)\,\mathrm{d}\xi \\
&= \frac{\Delta x_{A-1}}{8}\left[\xi - \frac{\xi^3}{3}\right]_{-1}^{1} \\
&= \frac{\Delta x_{A-1}}{6} \quad .
\end{aligned}
$$

$M_{A,\,A}$

In this case a global node may have contributions from two different elements (the last node of the $A-1$ element) and the first node of the Ath element. Hence we consider the integral for both elements and assemble them. For the $A-1$'th element, $a=2$ for

$$
N_a = \frac{1}{2}[1+(-1)^a\,\xi]
$$

meaning the integral for this element is

$$
\begin{aligned}
M_{AA} &= \int_{-1}^{1} N_2 \, N_2 \frac{\mathrm{d}x}{\mathrm{d}\xi} \, \mathrm{d}\xi \\
M_{AA} &= \int_{-1}^{1} \frac{1}{4}(1+\xi)^2 \frac{\Delta x_{A-1}}{2} \, \mathrm{d}\xi \quad = \frac{1}{3}\Delta x_{A-1}
\end{aligned}
$$

Now let us consider the A'th element. In this case we are focused on the first shape function

$$
\begin{aligned}
M_{AA} &= \int_{-1}^{1} N_2 \, N_2 \frac{\mathrm{d}x}{\mathrm{d}\xi} \, \mathrm{d}\xi \\
M_{AA} &= \int_{-1}^{1} \frac{1}{4}(1-\xi)^2 \frac{\Delta x_{A}}{2} \, \mathrm{d}\xi \quad = \frac{1}{3}\Delta x_{A}
\end{aligned}
$$

hence the overall contribution to this global (assembled) matrix is

$$
M_{AA} = \frac{1}{3}\Delta x_{A-1} + \frac{1}{3}\Delta x_{A} \quad .
$$

Problem 7.13

The initial and boundary conditions stated in the question do not produce the analytical solution stated. Instead we will consider a slightly different system. For the domain $x \in [0, 1]$ and boundary conditions

$$
\begin{aligned}
\partial_x \theta(x=0,\, t) = H_0 = 0 \\
\theta(x=1,\, t) = \theta_1 = 1
\end{aligned}
$$

let us consider a different initial condition.

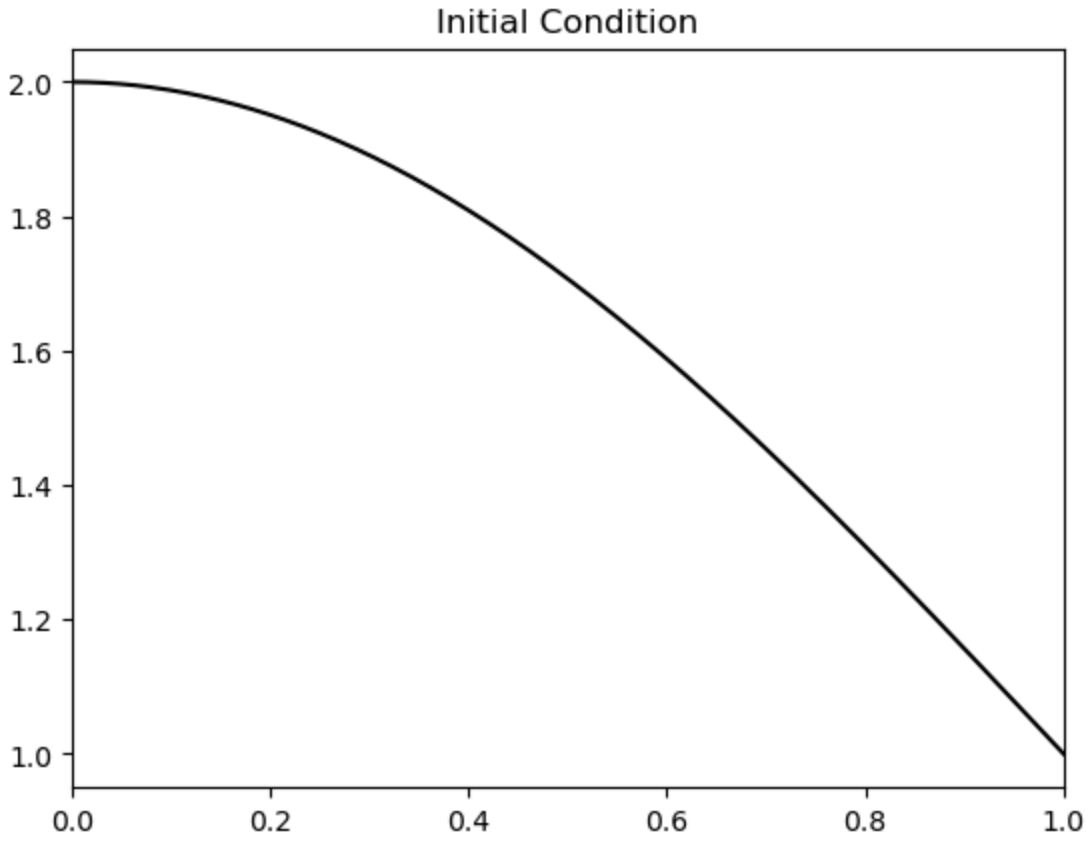
$$
\theta(x,\, t=0) = \theta_0 = 1 + \cos\left(\frac{\pi x}{2}\right) \quad .
$$

This system has the analytical solution

$$
\theta(x,\, t) = 1 + \exp\left(-\left(\frac{\pi}{2}\right)^2 t\right) \cos\left(\frac{\pi x}{2}\right)
$$

which we will model instead. Let us first plot this new initial condition

▶ Show code cell source



Physically, this system allows for no heat flux at the left boundary, while the RHS boundary must maintain a constant temperature. The domain has no input of heat. To facilitate these conditions, the heat must exit though the right boundary of the system, for which the heat flux is not fixed. Hence, we expect to see a decay in temperature through time until the system is at steady state.

Domain setup

Let us define our domain numerically with 10 elements, leading to 11 global nodes.

```
L       = 1         # domain size
nelem   = 10        # number of elements, n
npts    = nelem + 1 # n+1

# Global domain node locations
x = np.linspace(0, L, npts)

# Compute Delta x for each element:
dx = x[1:] - x[:-1]

# Boundary conditions:
theta_1 = 1
H0      = 0
```

K matrix

Next, we compute our global diffusivity matrix based on Eqns. 7.49 - 7.51

```
# Generate K
K = np.zeros((nelem, nelem))

for ielem in range(nelem):

    if ielem == 0:
        K[ielem, ielem] = 1 / dx[0]
    else:
        K[ielem, ielem] = (1 / dx[ielem - 1]) + (1 / dx[ielem])
        K[ielem - 1, ielem] = -1 / dx[ielem - 1]
        K[ielem, ielem - 1] = -1 / dx[ielem - 1]
```

Mass matrix

The mass matrix can be similarly computed based on Eqns 7.88 - 7.90

```
M = np.zeros((nelem, nelem))

for ielem in range(nelem):
    if ielem == 0:
        # 7.88
        M[ielem, ielem] = dx[0] / 3
    else:
        # 7.90
        M[ielem, ielem] = (dx[ielem - 1] + dx[ielem]) / 3
        # 7.89
        M[ielem - 1, ielem] = dx[ielem - 1] / 6
        M[ielem, ielem - 1] = dx[ielem - 1] / 6
```

Forcing

The force vector is given in Eqn. 7.87 as

$$F_A = \int_0^1 N_A \, h \, \mathrm{d}x + N_A(0) H_0 - \theta_1 \int_0^1 \left(\partial_x N_A\right) \left(\partial_x N_{n+1}\right) \mathrm{d}x \quad .$$

For our case, $h = 0$ and $H_0 = 0$ such that the force is only

$$\begin{aligned} F_A &= \theta_1 \int_0^1 \left(\partial_x N_A\right) \left(\partial_x N_{n+1}\right) \mathrm{d}x \quad , \\ &= \theta_1 \int_{x_x}^{x_{n+1}} \left(\partial_x N_n\right) \left(\partial_x N_{n+1}\right) \mathrm{d}x \quad , \\ &= \frac{\theta_1}{\Delta_x} \end{aligned}$$

at the nth node only.

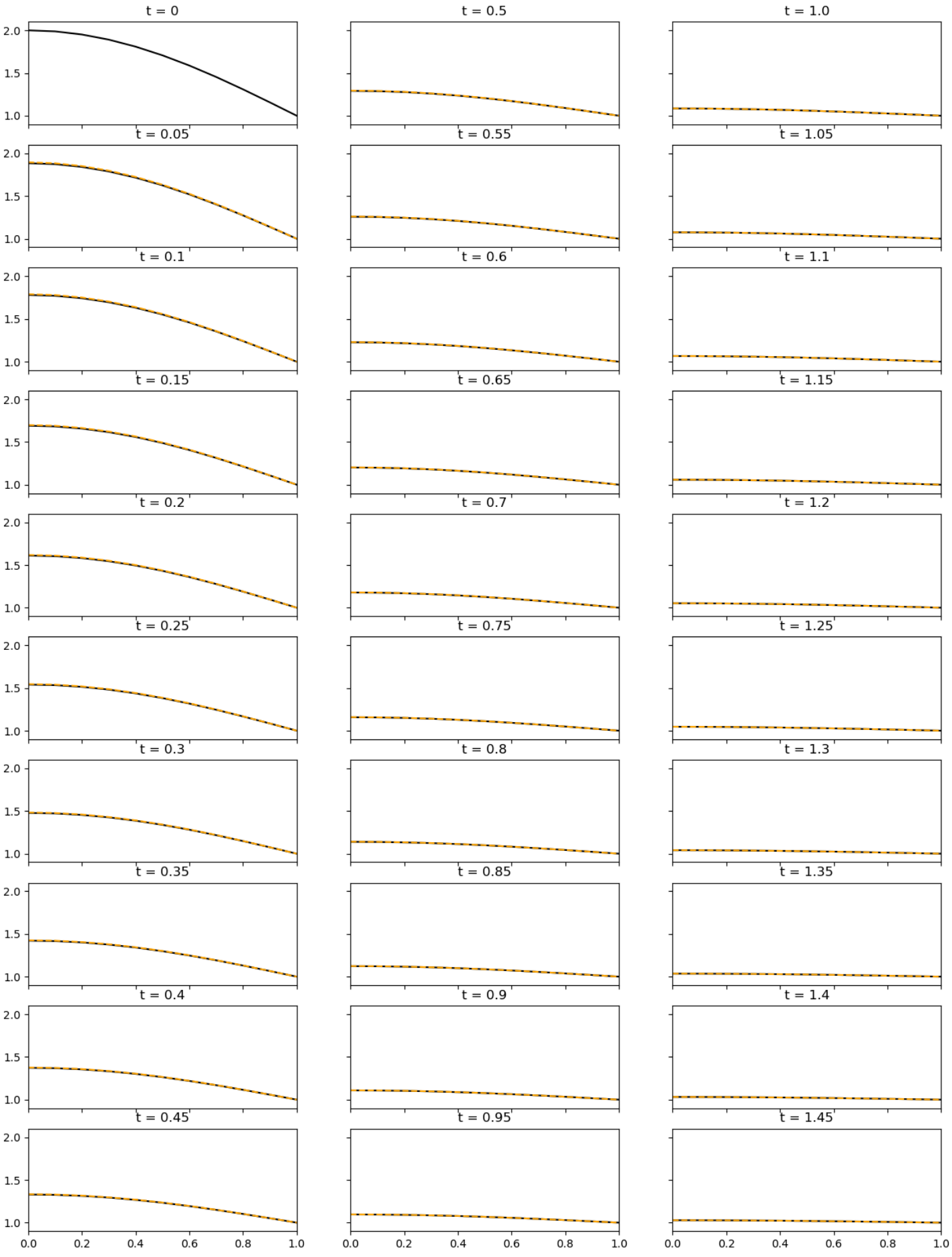
```
F = np.zeros(nelem)
F[-1] = theta_1/(dx[-1])
```

Timestepping

Finally, we can step our system of equations in time using the Generalised Trapezoidal Time Scheme discussed in Section 7.3.3. For $\eta = 0.5$ this is a centered finite difference method in time.

```
eta    = 0.5    # time-scheme parameter
dt     = 0.01   # timestep
nsteps = 150    # number of timesteps

# Initialise temperature and derivative
temp       = np.zeros(npts)
dtemp_dt   = np.zeros(npts)

# Set initial condition
temp       = 1 + np.cos(0.5* np.pi * x)

# Inverse of [M + eta \delta t K]
# rearranged version of Eqn 7.92
MKinv = np.linalg.inv(M + eta*dt*K)

# Create plots of time evolution
fig, ax = plt.subplots(10, 3,
                       sharey=True,
                       figsize=(15,20),
                       sharex=True)

# Step in time
time = 0

# Plot initial condition
ax[0,0].set_title(f"t = 0");
ax[0,0].plot(x, temp, 'k')

iax = 1
icol = 0
for istep in range(1,nsteps):
    # Compute time
    time += dt

    # --- FEM calculation ---
    # Predictor:
    dtilde = temp[:-1] + (1-eta)* dt * dtemp_dt[:-1]

    # Solve for dtemp_dt at next timestep
    dtemp_dt[:-1] = np.matmul(MKinv, F - np.matmul(K, dtilde))

    # Corrector:
    temp[:-1] = dtilde + (eta * dt * dtemp_dt[:-1])

    # Plot every 5 timesteps
    if istep%5 == 0:

        # Compute analytical solution and plot
        analytical =  1 + (np.exp(- 0.25 * time * np.pi**2 ) *
                           np.cos(0.5*np.pi *x))
        ax[iax, icol].plot(x, analytical, 'k')

        # Plot FEM
```

```
        ax[iax, icol].set_title(f"t = {np.around(time,2)}");
        ax[iax, icol].plot(x, temp, 'orange', linestyle='--')

        iax += 1
        if iax==10:
            icol += 1
            iax = 0

# Some plot formatting
ax[0,0].set_xlim([0,1]);
ax[0,0].set_ylim([0.9,2.1]);
```


t = 0
t = 0.5
t = 1.0
t = 0.05
t = 0.55
t = 1.05
t = 0.1
t = 0.6
t = 1.1
t = 0.15
t = 0.65
t = 1.15
t = 0.2
t = 0.7
t = 1.2
t = 0.25
t = 0.75
t = 1.25
t = 0.3
t = 0.8
t = 1.3
t = 0.35
t = 0.85
t = 1.35
t = 0.4
t = 0.9
t = 1.4
t = 0.45
t = 0.95
t = 1.45
2.0
1.5
1.0
0.0
0.2
0.4
0.6
0.8
1.0

Problem 7.14

This problem is similar to Problem 7.13, with an alteration to the initial conditions and the forcing. First, we note observe that there is no internal heating ($h = 0$) – is this a valid assumption for oceanic lithosphere?

The domain boundary at $x = L$ is defined by the Neumann boundary condition that

$$\partial_x \theta(L, t) = 0 \quad .$$

By analogy with Problem 7.13 this would be the left-hand-side boundary, i.e. the first element. The other boundary (the surface) is defined by the Dirichlet condition that

$$\theta(0, t) = \theta_1 = 0 \quad .$$

Considering the heat supply vector defined in Eqn 7.87, \begin{equation*} F_A = - ,\theta_1 \int_0^1 \alpha, \left(,\partial_x N_A \right) \left(\partial_x N_{n+1} \right) : \mathrm{d}x + \int_0^1 ,N_A , h, \mathrm{d}x - , N_A(0), \alpha(0) , \partial_x \theta(0) \end{equation*} we notice that the forcing term is always 0.

Domain setup

Let us define the parameters for this simulation:

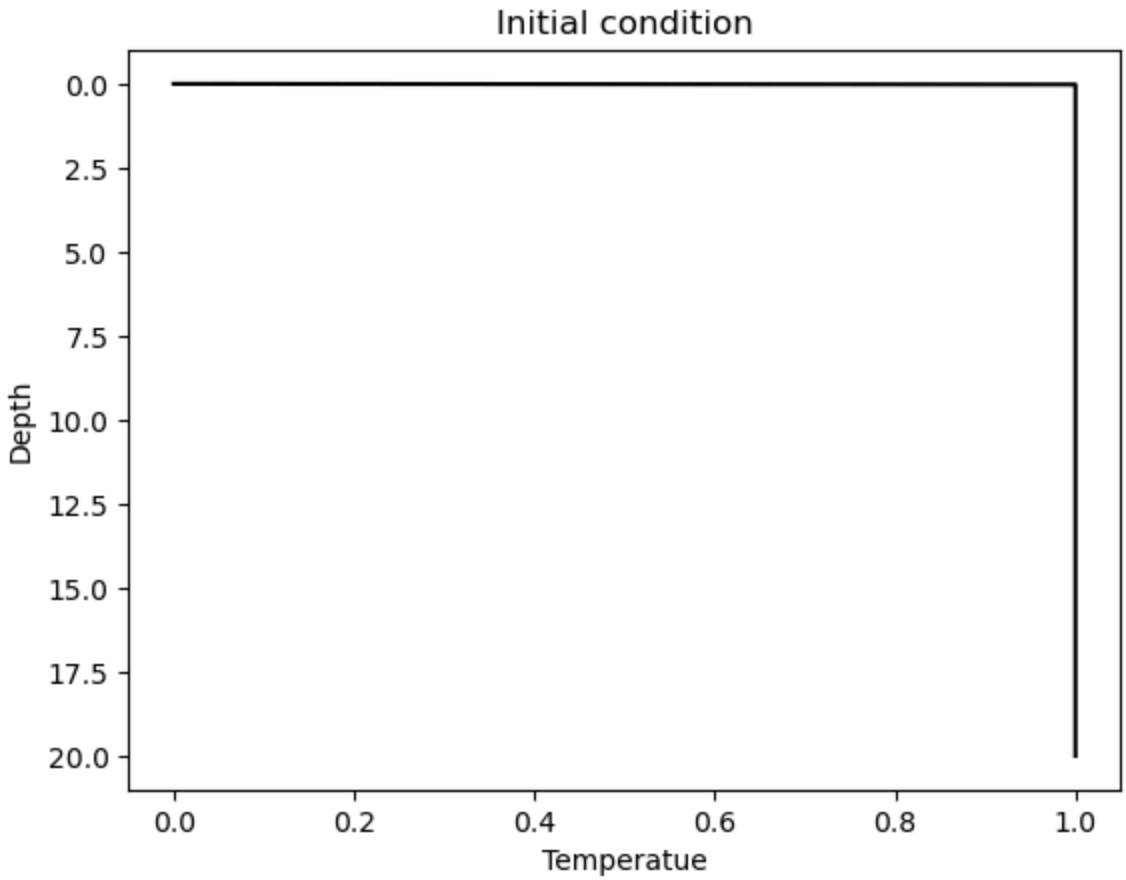
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.special as ss

L       = 20        # Domain size
alpha   = 1         # Diffusivity

nelem   = 1000      # Elements
npts    = nelem + 1 # Control points

eta     = 0.5       # Time scheme param.
dt      = 0.05      # Time step

# Define grid points of domain
# and grid spacing
# Note LHS is defined as L where
# Neumann condition exists
# and RHS (z=0) for Dirichlet BC
z = np.linspace(L, 0, npts)
dz =  z[:-1] - z[1:]
```

Initial conditions

Note also that the Dirichlet boundary condition and initial condition are not compatible. It is sufficient to setup the problem with the initial condition being $\theta(x) = 1$ everywhere except $x = 0$ where $\theta(x) = 0$, as plotted below.

Since the heat equation is a 1st order PDE in time, we do not need the initial condition for $\partial_t \theta(x,\, t = 0)$. However, it is required by the time-marching scheme we are using. We can assume that it is 0.

```
# Initial condition:
analytical = np.zeros(npts)
temp       = np.zeros(npts)

# assume initial dtemp/dt = 0
dtemp_dt = np.zeros(npts)

# zero only at surface
analytical[:-1] = 1
temp[:-1] = 1

# Plot initial condition
fig, ax = plt.subplots()
ax.plot(analytical, z, 'k')
ax.set_ylim([21, -1]);
ax.set_title("Initial condition");
ax.set_ylabel("Depth");
ax.set_xlabel("Temperatue");
```



Capacity & Diffusivity matrices

The capacity and diffusivity matrices are identical to those in Problem 7.13. Let us build them once and time below

```
# Generate K & M matrices
M = np.zeros((nelem, nelem))
K = np.zeros((nelem, nelem))

for ielem in range(nelem):

    if ielem == 0:
        K[ielem, ielem] = 1 / dz[0]

        M[ielem, ielem] = dz[0] / 3
    else:
        K[ielem, ielem] = (1 / dz[ielem - 1]) + (1 / dz[ielem])
        K[ielem - 1, ielem] = -1 / dz[ielem - 1]
        K[ielem, ielem - 1] = -1 / dz[ielem - 1]

        M[ielem, ielem] = (dz[ielem - 1] + dz[ielem]) / 3
        M[ielem - 1, ielem] = dz[ielem - 1] / 6
        M[ielem, ielem - 1] = dz[ielem - 1] / 6
```

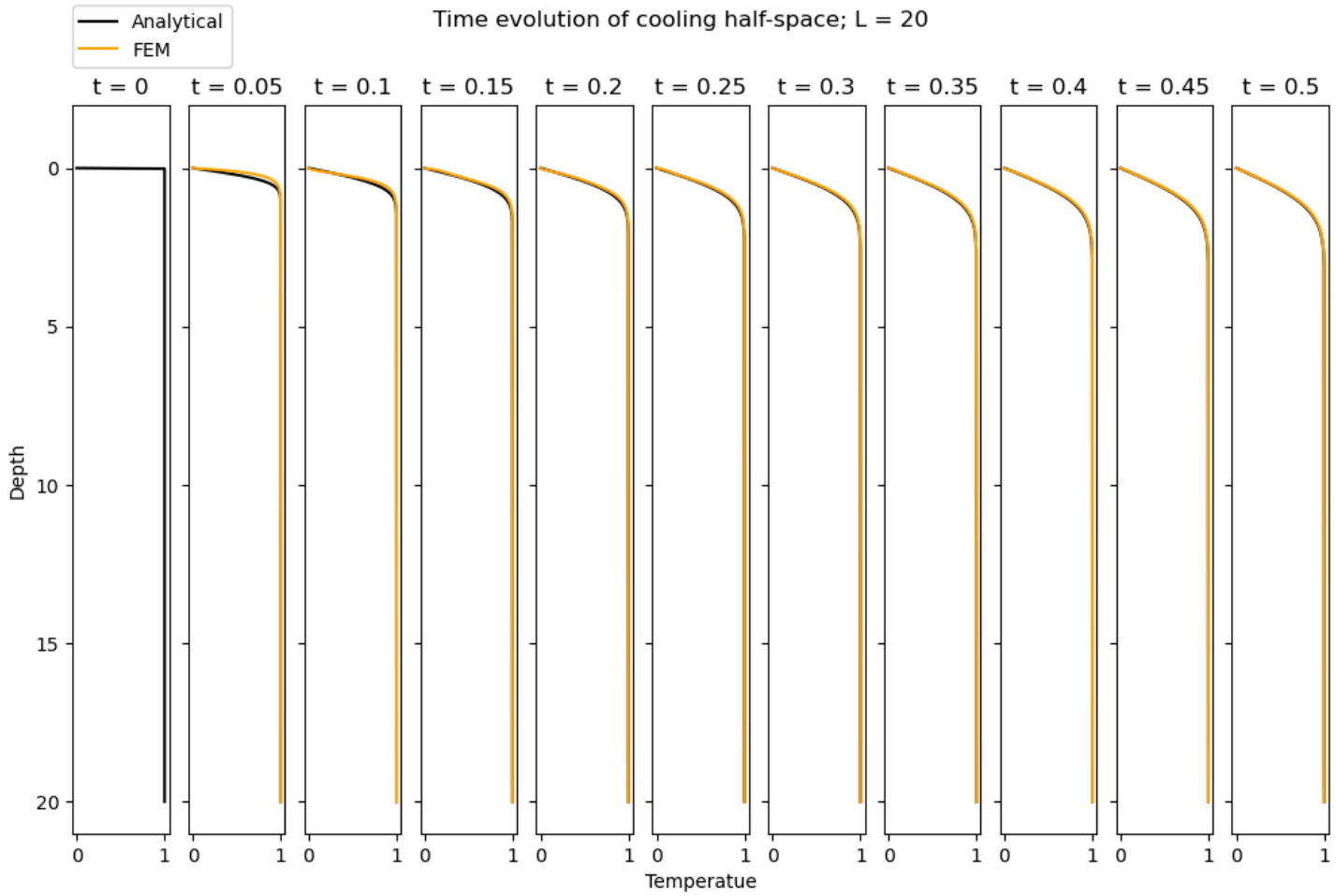
Time-marching

We can now time-march the system. To do this, we will use the predictor-corrector method in Eqns. 7.95 - 7.97. For Eqn. 7.96, we need to compute

$$\dot{\mathsf{d}}_{n+1} = (\mathsf{M} + \eta \Delta t \mathsf{K})^{-1} \left[-\mathsf{K}\, \tilde{\mathsf{d}}_{n+1} \right]$$

since $\mathsf{F} = 0$. Note that this inverse matrix is not time dependent and so we may compute it only once.

```
# Inverse of [M + eta \delta t K]
MKinv = np.linalg.inv(M + eta*dt*K)

# Create plots of time evolution
fig, ax = plt.subplots(1, 11,
                       sharey=True,
                       figsize=(12,7))

# Step in time
time = 0

# Plot initial condition
ax[0].plot(temp, z, 'k')

for istep in range(10):
    # Compute time
    time += dt

    # Compute analytical solution
    argument = z / (2 * np.sqrt(alpha * time))
    analytical = ss.erf(argument)
    ax[1+istep].plot(analytical, z, 'k', label='Analytical')

    # --- FEM calculation ---
    # Predictor:
    dtilde = temp[:-1] + (1-eta)* dt * dtemp_dt[:-1]

    # Solve for dtemp_dt at next timestep
    dtemp_dt[:-1] = np.matmul(MKinv,  - np.matmul(K, dtilde))

    # Corrector:
    temp[:-1] = dtilde + (eta * dt * dtemp_dt[:-1])

    ax[istep+1].set_title(f"t = {np.around(time,2)}");

    ax[istep+1].plot(temp, z, 'orange', label='FEM')


# Some plot formatting
ax[0].set_ylim([21, -0.1*L]);
ax[0].set_title(f"t = 0");
fig.suptitle(f"Time evolution of cooling half-space; L = {L}");
ax[0].set_ylabel("Depth");
ax[5].set_xlabel("Temperatue");
ax[1].legend(bbox_to_anchor=(0.55, 1.15));
```



Varying the domain size, L

When $L = 20$, we observe relatively good agreement between the analytical solution and FEM solution. How does this vary with L? In the hidden cell below, we re-run the entire code for $L = 1$ and $L = 100$. We observe that the FEM simulation does a rather bad job at matching the analytical solution when $L = 1$ but a good job for $L = 100$. Why is this?

The analytical solution is for a half-space, $x \in [0, \infty)$, but evidently our domain is finite. So now the question is how good is our finite domain at approximating the half-space? This depends on the properties of the medium. In particular, consider the characteristic length scale that is defined by the analytical solution,

$$\theta(x,\, t) = \mathrm{erf}\left(\frac{x}{2\sqrt{\alpha\, t}}\right).$$

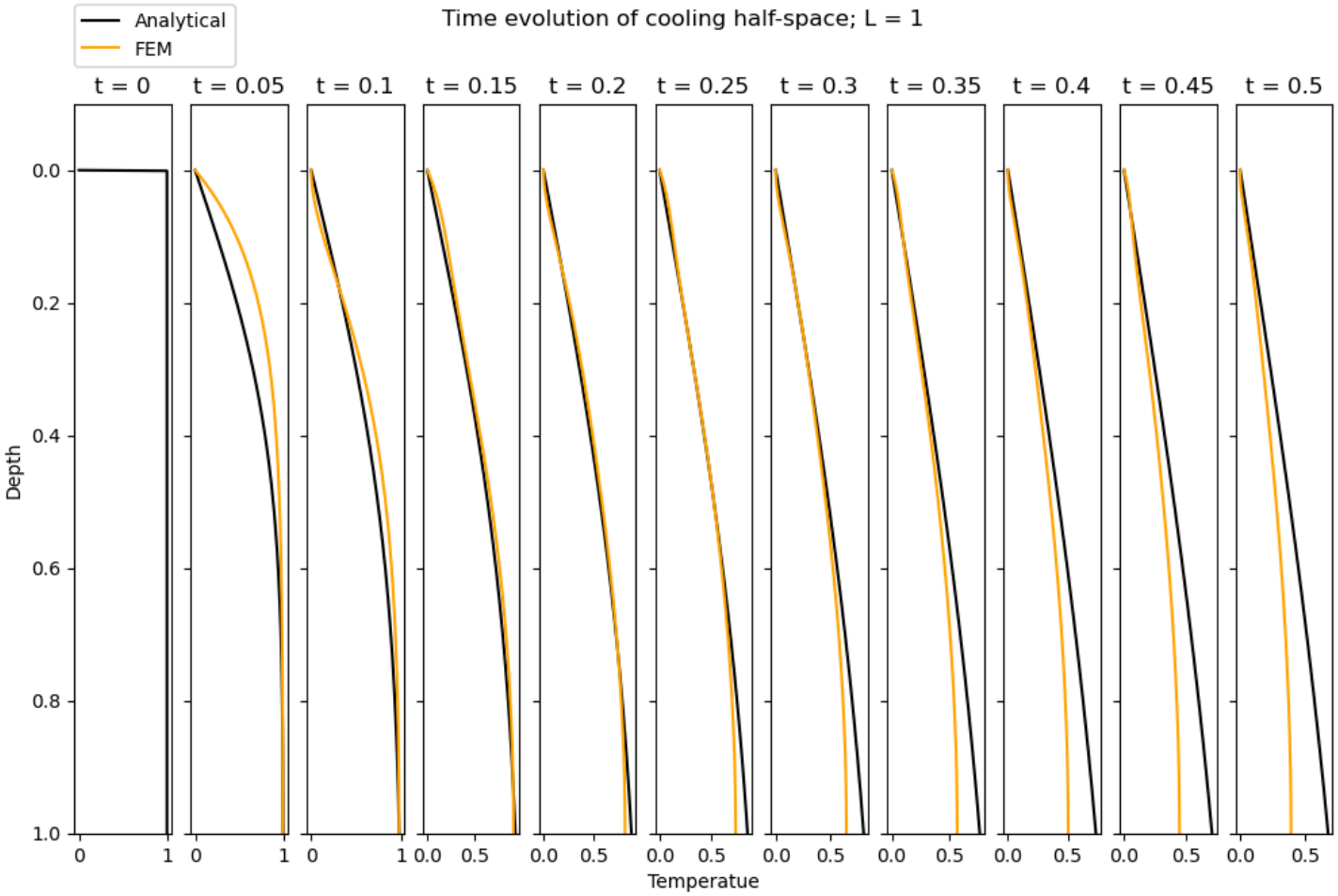
The length scale is then $\mathcal{L} \sim 2\sqrt{\alpha\, t}$. When this value is large, the error function tends to 0. When this value is small, the argument of the error function is large and theta tends towards 1. So we know

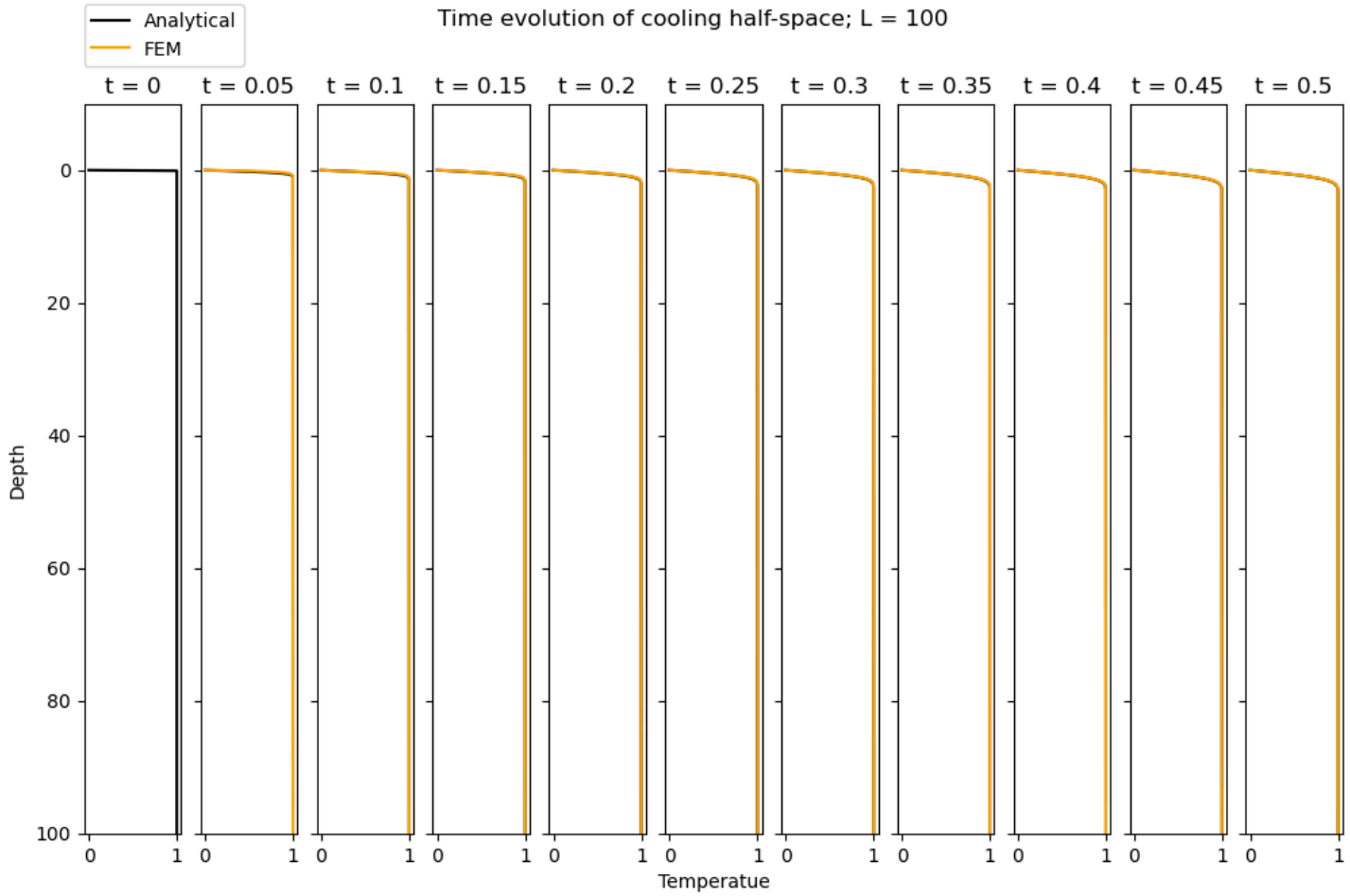
that when $\mathcal{L} \sim 2\sqrt{\alpha t}$ is small, the temperature profile approximates the initial condition (where its 1 everywhere), while then the characteristic length scale is large, the temperature profile is essentially 0 everywhere.

This length scale essentially represents the length over which heat can diffuse within a given time. So when this value is large (relatively to the domain we have defined) the heat will diffuse out of it quickly. This is where we observe disagreement with the half-space solution, which is supposed to be infinitely large such that the heat does not diffuse out of the system.

So, when our domain is too small (relative to the characteristic length scale defined by the time and the diffusivity, α), our simulation does not replicate a half-space well. Note that with $\alpha = 1$ the characteristic length scale for $t = 0.5$ is 0.5. Hence, a domain of $L = 1$ is not sufficiently large to represent infinite space.

Finally, note there are always some small differences for the first timestep(s) as the system adjusts from the inconsistent boundary and initial conditions.


Analytical
FEM
Time evolution of cooling half-space; L = 1
t = 0
t = 0.05
t = 0.1
t = 0.15
t = 0.2
t = 0.25
t = 0.3
t = 0.35
t = 0.4
t = 0.45
t = 0.5
Depth
0.0
0.2
0.4
0.6
0.8
1.0
0
1
0.0
0.5
Temperatue



Problem 7.15

Starting with the definition for the local capacity matrix in Eqn 7.104

$$m_{ab}^{A} = \int_{x_A}^{x_{A+1}} N_a(x)\, N_b(x)\, \mathrm{d}x \quad ,$$

for which the shape functions in the reference coordinates, ξ, are defined in 7.56 as

$$N_a = \frac{1}{2}[1 + (-1)^a \xi] \quad .$$

We may pull back the integration to a reference element using the mapping in 7.55 such that the bounds of integration go from $[x_A,\, x_{A+1}] \rightarrow [-1,\, 1]$:

$$
\begin{aligned}
m_{ab}^{A} &= \int_{x_A}^{x_{A+1}} N_a(x)\, N_b(x)\, \mathrm{d}x \quad , \\
&= \int_{-1}^{1} N_a(\xi)\, N_b(\xi)\, \frac{\mathrm{d}x}{\mathrm{d}\xi} \mathrm{d}\xi \quad , \\
&= \frac{1}{4} \int_{-1}^{1} \left[1 + (-1)^a \xi\right] \left[1 + (-1)^b \xi\right] \frac{\Delta x_A}{2} \mathrm{d}\xi \quad ,
\end{aligned}
$$

where we have used $\frac{\mathrm{d}x}{\mathrm{d}\xi} = \frac{\Delta x_A}{2}$ defined in Eqn 7.57. Evaluating this integral yields

$$
\begin{aligned}
m_{ab}^{A} &= \frac{\Delta x_A}{8} \left[\xi + \frac{(-1)^a + (-1)^b}{2} \xi^2 + \frac{(-1)^{a+b}}{3} \xi^3 \right]_{-1}^{1} \\
&= \frac{\Delta x_A}{4} \left[1 + \frac{(-1)^{a+b}}{3} \right]
\end{aligned}
$$

as required.

Problem 7.16

This problem reproduces the global-element version of Problem 7.14 for a cooling half-space, using a local element method. We introduced the local-element method for the diffusivity matrix in Problem 7.10. Similarly, the timestepping component was introduced in Problem 7.14. The details are therefore not re-capitulated here.

The new component introduced in this problem is the assembly of the capacity matrix from local elements.

In the hidden cell below, we will set up the domain and initial conditions in an identical manner to Problem 7.14.

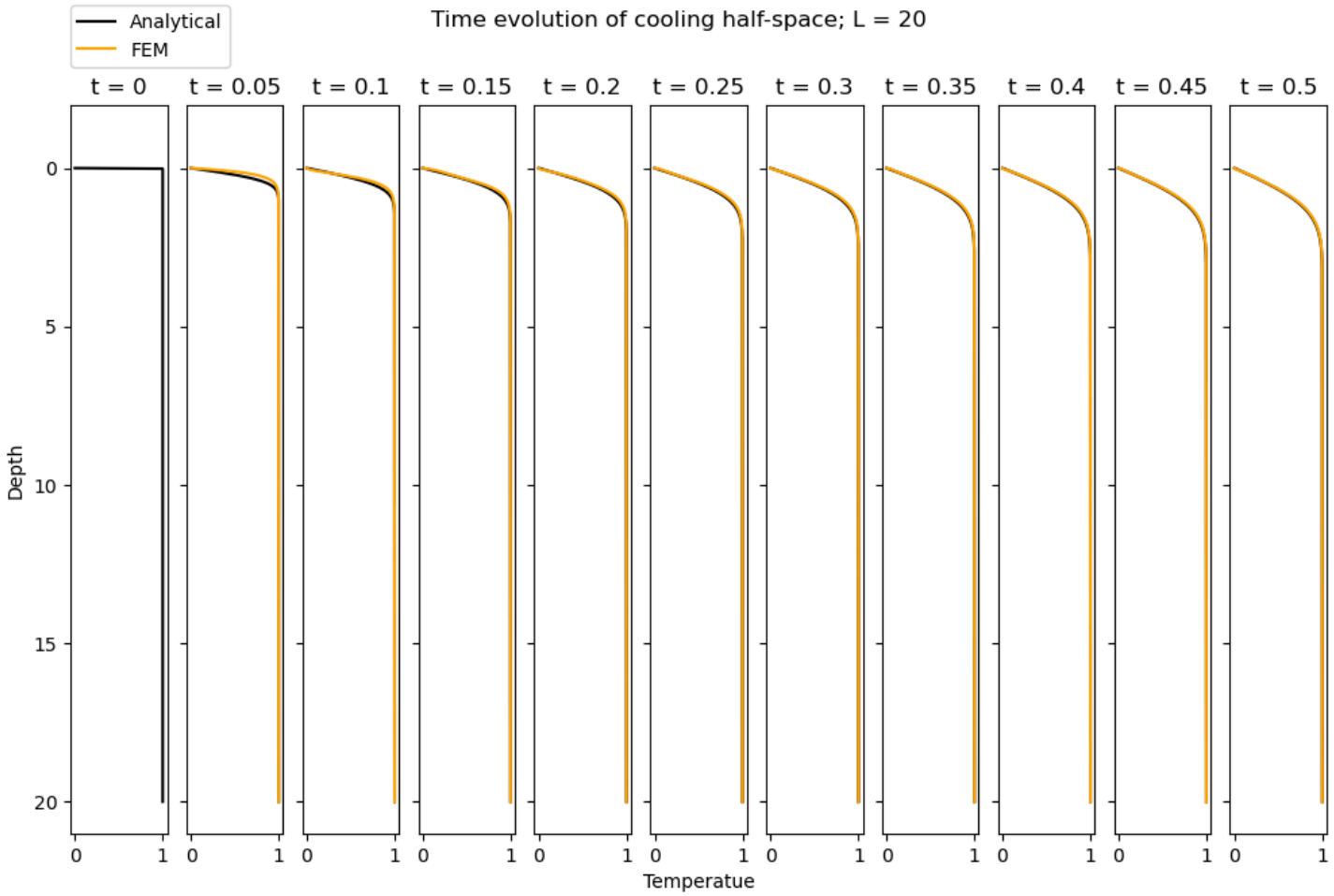
▶ Show code cell source

Next, the global capacity and diffusivity matrices are assembled from local elements:

```
# Generate Global K & M matrices
M = np.zeros((nelem, nelem))
K = np.zeros((nelem, nelem))

# loop over elements
for ielem in range(nelem):

    # Compute local diffusivity matrix
    # for this element using 7.65
    kloc = np.zeros((2,2))
    kloc[0,0] = 1
    kloc[1,1] = 1
    kloc[0,1] = -1
    kloc[1,0] = -1
    kloc *= (1/dz[ielem])

    # Compute local capacity matrix:
    # Eqn. 7.106
    mloc = np.zeros((2,2))
    mloc[0,0] = 2
    mloc[1,1] = 2
    mloc[0,1] = 1
    mloc[1,0] = 1
    mloc *= dz[ielem]/6


    # Assemble the K and M matrices
    if ielem == nelem -1:
        # For the last element we are ignoring the
        #contributions from the n + 1 node
        K[ielem, ielem] += kloc[0,0]
        M[ielem, ielem] += mloc[0,0]
    else:
        K[ielem:ielem+2, ielem:ielem+2] += kloc
        M[ielem:ielem+2, ielem:ielem+2] += mloc
```

Finally, we can conduct timestepping in a similar manner to Problem 7.14 and reproduce the results:

```
# Inverse of [M + eta \delta t K]
MKinv = np.linalg.inv(M + eta*dt*K)

# Create plots of time evolution
fig, ax = plt.subplots(1, 11,
                       sharey=True,
                       figsize=(12,7))


# Step in time
time = 0

# Plot initial condition
ax[0].plot(temp, z, 'k')

for istep in range(10):
    # Compute time
    time += dt

    # Compute analytical solution
    argument = z / (2 * np.sqrt(alpha * time))
    analytical = ss.erf(argument)
    ax[1+istep].plot(analytical, z, 'k', label='Analytical')

    # --- FEM calculation ---
    # Predictor:
    dtilde = temp[:-1] + (1-eta)* dt * dtemp_dt[:-1]

    # Solve for dtemp_dt at next timestep
    dtemp_dt[:-1] = np.matmul(MKinv,  - np.matmul(K, dtilde))

    # Corrector:
    temp[:-1] = dtilde + (eta * dt * dtemp_dt[:-1])

    ax[istep+1].set_title(f"t = {np.around(time,2)}");

    ax[istep+1].plot(temp, z, 'orange', label='FEM')


# Some plot formatting
ax[0].set_ylim([21, -0.1*L]);
ax[0].set_title(f"t = 0");
fig.suptitle(f"Time evolution of cooling half-space; L = {L}");
ax[0].set_ylabel("Depth");
ax[5].set_xlabel("Temperatue");
ax[1].legend(bbox_to_anchor=(0.55, 1.15));
```



Problem 7.17

There are a number of ways one can obtain the solutions of the equation

$$(1-\xi^2)\frac{\mathrm{d}}{\mathrm{d}\xi}P_n(\xi) = 0$$

for which P_n is the Legendre polynomial of degree n.

Numerous algorithms have been developed for finding the roots of functions. In this case, we will use a simple Bisection method. This requires us to be able to evaluate

$$f(\xi) = (1-\xi^2)\frac{\mathrm{d}}{\mathrm{d}\xi}P_n(\xi)$$

within the domain $\xi \in [-1, +1]$. To do this, we can utilise one of the many identities of the Legendre polynomials,

$$(1-\xi^2)\frac{\mathrm{d}}{\mathrm{d}\xi}P_n(\xi) = n\left[P_{n-1}(\xi) - xP_n(\xi)\right];$$

see Relation 14.10.5 of DLFM. This allows us to write our function $f(\xi)$ in terms of Legendre polynomials, which are available in SciPy.

Setting up the code:

First let us define a few parameters. n is the order of the Legendre polynomial in our function. There will then be $n+1$ roots including the end points $\xi = \pm 1$. In the case that n is even, one of the roots will be $\xi = 0$. Note that the roots are always symmetric about $\xi = 0$. We will seek out roots by splitting up the domain into a number of segments and testing if a root lies within that segment. The number of segments is given by `Nsegs`. When this number is high, computation time is longer. When this number is too low, you risk missing roots or not converging.

```
import scipy.special as ss
import numpy as np
from copy import copy
import matplotlib.pyplot as plt


n      = 11                # Degree of Legendre polynomial
Nsegs  = 100               # Number of segments
nroots = n + 1             # Number of roots
roots  = np.zeros(nroots)  # Array to hold all the roots

# Compute relevant Legendre functions
# leg_n and leg_n_1 are orthopoly1d objects that can be called to
# return a value of the polynomial at (a) specific xi value(s)
leg_n   = ss.legendre(n)    # P_n
leg_n_1 = ss.legendre(n-1)  # P_{n-1}


# Let us also define a function that returns our function f
# at point(s) xi
# Note here that we parse in leg_n and leg_n_1
# as arguments so they are only generated once above,
# instead of being initialised every time the function is computed
def compute_functional(xi, n, lnm1, ln):
    return n * (lnm1(xi) - xi * ln(xi))
```

Since we know two of the roots are $\xi = \pm 1$, lets set those now:

```
# Two roots will always be +=1
roots[0]   = -1
roots[-1]  =  1
# Count the number of roots found so far
nrts_found =  2
```

Note here that if $n \leq 2$ then we are done, since the roots are at $\xi = \pm 1$ or $\xi = 0$. Hence we only need to execute the next cells if there are more roots to be found. To do this, we will use the Bisection method. Since the roots are symmetric about 0, we only need to seek roots in the range $\xi = [0, 1]$

```
# For n <= 2 we are done already...
if nroots > 3:

    # End point to search for root
    # -- avoiding xi = 1
    xi_end = 0.99999

    if n%2==0:
        # Even power n
        # Odd number of roots including xi = 0
        # can start hunting only from > 0 and < 1
        xi_start = 0.00001
        nrts_found += 1
    else:
        # Look for the positive roots (mirror around xi = 0)
        xi_start = 0.0

    # Create array of segments
    segments = np.linspace(xi_start, xi_end, Nsegs)

    # Evaluate function at each segment boundary
    fseg = compute_functional(segments, n, leg_n_1, leg_n)

    # Index in the 'roots' array to store the root
    idx = int(nroots / 2) + nroots%2

    # Loop through each segment and test if root lies within
    for iseg in range(Nsegs-1):
        # Deal with the unlikely cases we hit
        # the root position exactly
        if fseg[iseg] == 0:
            # Root found exactly here
            error = 0
            nrts_found += 1
        elif fseg[iseg+1] == 0:
            # Root found exactly here
            error = 0
            nrts_found += 1
        # More likely - check if a root is within the segment
        # indicated by change in sign of the function values at
        # each end of the segment
        elif fseg[iseg] * fseg[iseg+1] < 0:
            # There is a change in sign between the two values:
            # Start with the current edge xi positions of the 'segment'
            a = segments[iseg]
            b = segments[iseg+1]

            # initialise with a high error
            error = 999

            # We iterate until the function value at the midpoint ('error')
            # is close to 0
            while np.abs(error) > 1e-9:
```

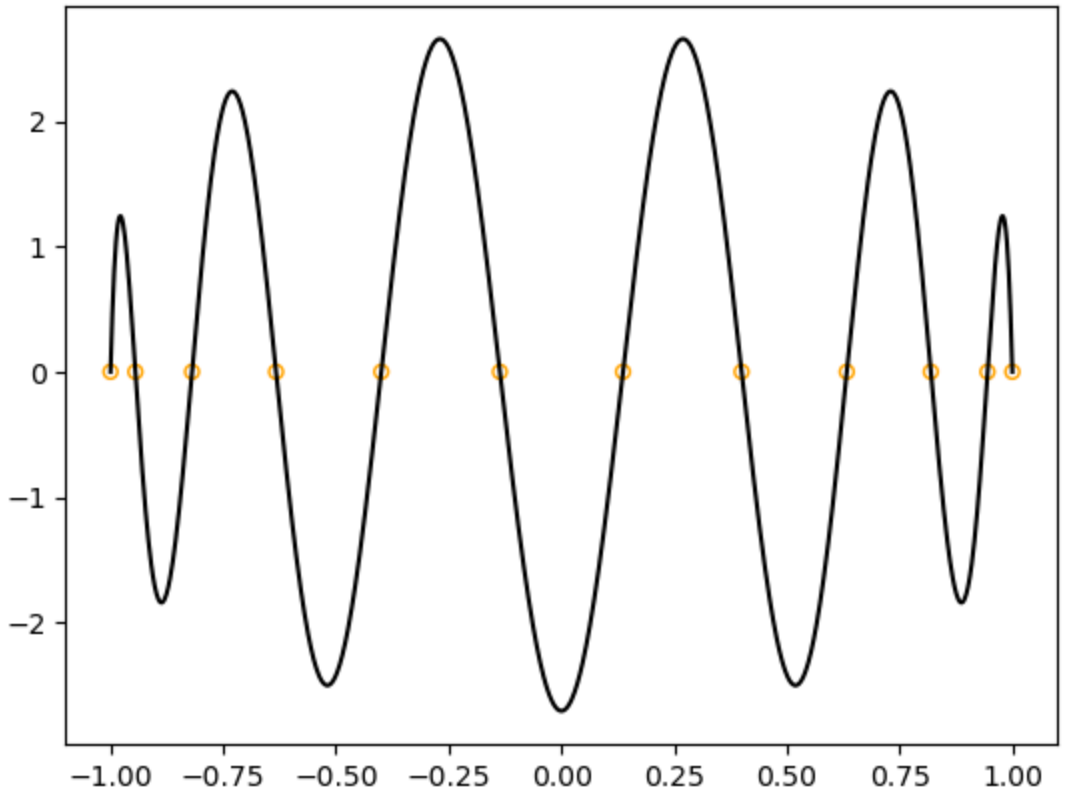
```
                # See if function has local positive or negative gradient
                grad =  compute_functional(b, n, leg_n_1, leg_n) \
                      - compute_functional(a, n, leg_n_1, leg_n)

                # Get midpoint and function value at midpoint
                this_root = a + (b-a)/2
                error    =  compute_functional(this_root, n, leg_n_1, leg_n)

                if error !=0:
                    # if sign of gradient * sign of error
                    # (functional value at midpoint) is > 0
                    # then we replace
                    # upper bound else replace the lower bound
                    if grad*error > 0:
                        b = copy(this_root)
                    else:
                        a = copy(this_root)

            # Add newest root found and update index
            roots[idx] = this_root
            idx += 1

            # Adding 2 to account for symmetry
            nrts_found+=2


    # Now we have scanned for all the roots within [0, 1]
    # Mirror roots around 0 since symmetric:
    # The n%2 accounts for the difference in mirroring (due to the 0 root) for n be
    # odd or even
    for irt in range(1, int(n/2) + n%2):
        i1 = int(nroots/2) -irt
        i2 = int(nroots/2) +irt  - n%2
        roots[i1] = -roots[i2]
```

Checking our roots

We should now have all the roots, but lets check that:

```python
if nrts_found != nroots:
    raise ValueError(f"Error: found {nrts_found} but there are {nroots}. Try larger


print(f"Number of roots found: {nrts_found}/{nroots}")
print(f"Roots                :\n {roots}")


fig, ax = plt.subplots()
xi = np.linspace(-1, 1, 1000)
ax.plot(xi, compute_functional(xi, n, leg_n_1, leg_n), 'k')

ax.scatter(roots, np.zeros(nroots), s=25, marker='o', color='None', edgecolor='orang
```

```
Number of roots found: 12/12
Roots                :
 [-1.         -0.94489927 -0.81927932 -0.63287615 -0.39953094 -0.13655293
  0.13655293  0.39953094  0.63287615  0.81927932  0.94489927  1.        ]
```



Problem 7.18

To find the weights for the $n+1$ GLL points we need to evaluate

$$\omega_\alpha = \int_{-1}^{1} \ell_\alpha^n(\xi)\,\mathrm{d}\xi$$

for which ℓ_α^n is a Lagrange polynomial. There is no good python function for computing Lagrange polynomials, so we will need to make our own. To compute the lagrange polynomials of degree n, we require the control points (GLL points) that define it. This means we need a function to compute the relevant GLL points also. In fact, this is what Problem 7.17 asks you to do. To avoid importing functions, we will copy the function that generates gll points below. Please refer to Problem 7.17 for details

▶ Show code cell source

Next, lets write a function to compute the Lagrange polynomials:

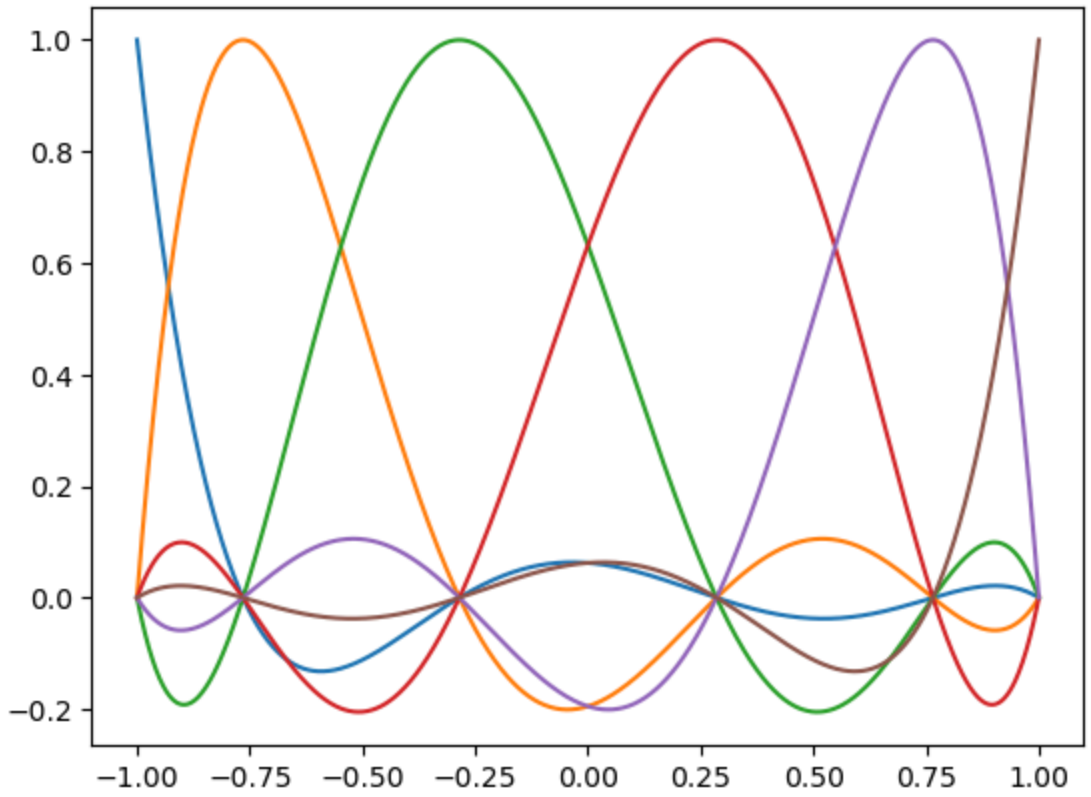
```
def lagrange(N, a, x, GLL):
    # Computes a'th Lagrange polynomial of degree N at points x
    # using control points specified in GLL array
    poly = 1
    for j in range(0, N+1):
        if j != a:
            poly = poly * ((x - GLL[j]) / (GLL[a] - GLL[j]))
    return poly
```

Now this is all set up, its straightforward to compute the weights. Let us define a value for n, and compute the lagrange polynomials over the interval $[-1, 1]$. From this, we can integrate to compute the weights:

```
# N value defining the lagrange polynomials
n = 5

# Compute the n+1 GLL points
gllpts = gll(n)

# Define the domain interval [-1, +1] on 1000 points:
x = np.linspace(-1, 1, 1000)

# We will plot the Lagrange polynomials to check they are correct
fig, ax = plt.subplots()

print(f'Weights for degree {n}: ')
# Loop over the n+1 polynomials to compute the weights
for alpha in range(n+1):
    # Integrand is simply the alpha'th lagrange polynomial of degree n
    integrand = lagrange(n, alpha, x, gllpts)

    # Plot lagrange polynomials as sanity check
    ax.plot(x, integrand)

    # Trapezoid rule to integrate
    weight = np.trapz(y=integrand, x=x)

    print(alpha, weight)
```

```
Weights for degree 5:
0 0.06666900466679278
1 0.37847201994856156
2 0.5548589753846457
3 0.5548589753846458
4 0.37847201994856156
5 0.0666690046667928
```

```
/tmp/ipykernel_4748/3267321456.py:23: DeprecationWarning: `trapz` is deprecated. Use
  weight = np.trapz(y=integrand, x=x)
```



Problem 7.19

The lagrange polynomials of degree n are described by their control points ξ_i , in this case GLL points, as

$$\ell_i^n(\xi) = \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{\xi - \xi_j}{\xi_i - \xi_j} \quad .$$

Using the product rule the derivative is then

$$\ell'^{n}_{i}(\xi) = \sum_{\substack{j=0 \\ j\neq i}}^{n} \left[\frac{1}{\xi_i - \xi_j} \prod_{\substack{m=0 \\ m\neq(i,j)}}^{n} \frac{\xi - \xi_m}{\xi_i - \xi_m} \right] .$$

An alternative formula for the derivative at the GLL points is given in Canuto et al 2006, Eqn. 2.3.28 in terms of Legendre polynomials of degree n, P_n:

$$\ell'_i(\xi_j) = \begin{cases} \frac{P_n(\xi_i)}{P_n(\xi_j)(\xi_i - \xi_j)} , & i \neq j \\ -\frac{n}{4}(n+1) , & i = j = 0 \\ \frac{n}{4}(n+1) , & i = j = n \\ 0 , & \text{otherwise} \end{cases}$$

To evaluate the function at GLL points specifically, we will need a function to compute the GLL points for a specific n value. This function, copied from Problem 7.17 is in the cell below.

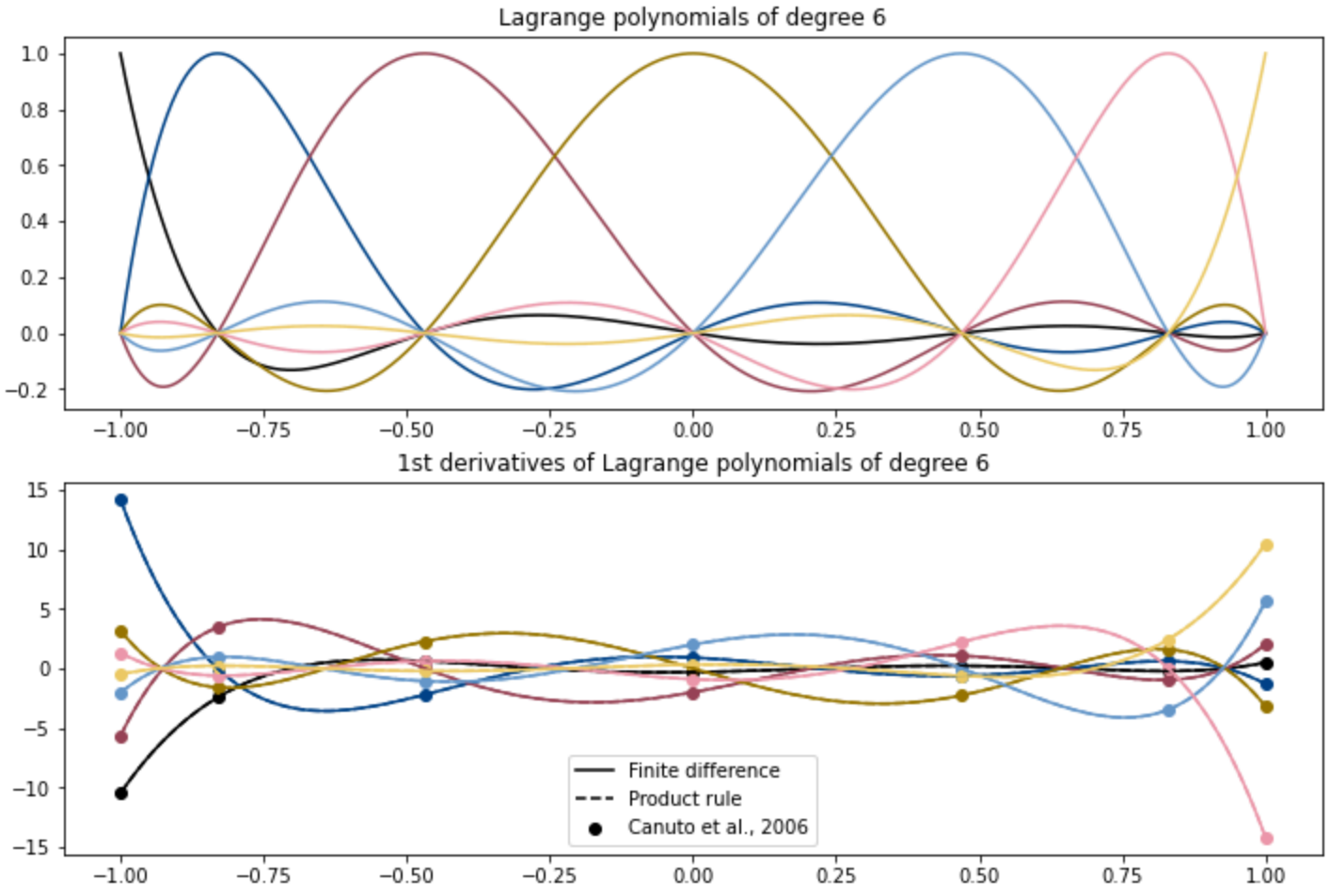
▶ Show code cell source

```
/usr/local/Caskroom/miniconda/base/lib/python3.8/site-packages/scipy/__init__.py:146
  warnings.warn(f"A NumPy version >={np_minversion} and <{np_maxversion}"
```

Now let us compute the derivatives using this product rule, and compare it against a finite-difference approach and the formula by Canuto et al 2006, to ensure consistency.

```
# Compute the Lagrange polynomials for n
n = 6

# Compute the n+1 GLL points
gllpts = gll(n)

# We will plot the Lagrange polynomials and their derivatives
fig, ax = plt.subplots(2, figsize=(12,8))

# Define domain
x = np.linspace(-1, 1, 1000)

# collect objects for legend
leglines = []

# plot colours: Paul Tol's medium contrast scheme
clrs = ['k', '#004488',  "#994455", '#997700',
        '#6699CC', '#EE99AA',  '#EECC66' ]


# Loop over the n+1 polynomials
for i in range(n+1):
    lag = lagrange(n, i, x, gllpts)

    # Plot lagrange polynomials as sanity check
    ax[0].plot(x, lag, color=clrs[i])


    # Compute the derivative as a finite difference
    fd = (lag[2:] - lag[:-2])/(x[2]-x[0])
    fd, = ax[1].plot(x[1:-1], fd, color=clrs[i])
    if i ==0:
        leglines.append(fd)

    # Compute the derivative with Product rule formula
    sum = x*0
    for j in range(n+1):
        if j!=i:
            prod = 1 + np.zeros(len(x))
            for m in range(n+1):
                if m != i and m != j:
                    prod *= (x - gllpts[m])/(gllpts[i] - gllpts[m])
            sum += (1/(gllpts[i] - gllpts[j])) * prod
    # Plot product rule version
    prodr, = ax[1].plot(x, sum, '--', color=clrs[i])
    if i ==0:
        leglines.append(prodr)


# Computing specifically at GLL points:
# Using method in Canuto et al 2006
# Loop over polynomials
for i in range(n+1):
```

```
    # Loop over GLL points
    derivs = np.zeros(n+1)

    leg_n = ss.legendre(n)
    for j in range(n+1):
        if i == 0 and j == 0:
            derivs[j] = -(n+1)*n/4
        elif i == n and j == n:
            derivs[j] = (n+1)*n/4
        elif i != j:
            derivs[j] = leg_n(gllpts[j]) / ( leg_n(gllpts[i]) * (gllpts[j]-gllpts[i]
        else:
            derivs[j] = 0

    # Plot
    canuto = ax[1].scatter(gllpts, derivs, marker='o', c=clrs[i])
    if i ==0:
        leglines.append(canuto)



# Cosmetics
ax[0].set_title(f'Lagrange polynomials of degree {n}');
ax[1].set_title(f'1st derivatives of Lagrange polynomials of degree {n}');
ax[1].legend(leglines, ['Finite difference', 'Product rule', 'Canuto et al., 2006']
```



Problem 7.20

To develop a functioning SEM solver, we need a number of ingredients that have been developed in previous problems. In the cell below, we copy a number of functions that we will use:

Function	Developed in Problem	Functionality
`gll`	Problem 7.17	This finds the GLL points given for the degree n Lagrange polynomials.
`weights`	Problem 7.18	Developed in Problem 7.18 - this finds weights at GLL points and also returns the gll points (so we don't have to explicitly call `gll` again!
`dlagrange1`	Problem 7.19	Finds first derivative of lagrange polynomials at GLL points.

▶ Show code cell source

Setting up the domain

First things first need to set up the domain. In this case we are using 12 elements. Lets use 7 GLL points in each element. For the timestepping, we will use $\eta = 0.5$ in the Generalised Trapezoidal Time Scheme (Section 7.3.3), equivalent to a centered difference method in time.

```
L      = 1            # Domain size
n      = 6            # Lagrange degree
ngll   = n + 1        # number of gll per element
nelmts = 12           # Number of elements
nbound = nelmts + 1   # Number of element boundaries

# Timestepping
eta    = 0.5          # Predictor corrector eta
dt     = 0.00001      # Timestep
nsteps = 20000        # Number of timesteps


# Physical parameter:
# diffusivity, alpha = 1 everywhere
alpha = np.zeros((ngll, nelmts)) + 1
```

The spatial domain is defined between $x \in [0, 1]$. The elements will be evenly spaced (though they don't have to be!). First, let us find where the element boundaries are:

```
# Global coordinates of element boundaries
elembnds = np.linspace(0, L, nbound)
elemsize = elembnds[1:] - elembnds[:-1]
```

Local to global mapping

Next, we can get the location of the GLL points in the natural domain, $\xi \in [-1, 1]$. Given the number of GLL points in each element, we can determine the global number of GLL points (since elements share GLL points on their boundaries.

Next, we wish to determine the coordinates of each global GLL point in the spatial domain. As we loop through each element to calculate this, we also compute the array called `ibool`. This array holds the mapping of node ID's between each element and the global ID of the nodes.

Let us take the second and third elements in our domain, for example. Both elements have 7 GLL nodes. Within each element, these nodes simply have local IDs of $a = 1, \ldots 7$ (or $a = 0, \ldots 6$ in python indexing).

However these nodes have global IDs of $A = 7 \ldots 19$. Note that the node with global ID $A = 13$ is the last node of element 2, and also the first node of element 3. See Fig 7.9 for an example with 5 GLL nodes in each element.

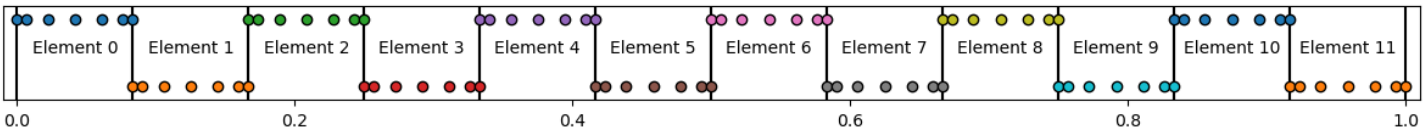
The ibool array stores the global IDs for each local node in each element, that is:

```
globalID = ibool(local_id, element_id)
```

Below, we plot the domain setup for clarity, with vertical lines representing element boundaries

```
# Get GLL and weights:
xigll, wgll = weights(n)


# Create an array of the GLOBAL node coordinates:
nglob = nelmts*(ngll-1) + 1


# Array of global node x coordinates
# and ibool (lobal --> global mappping)
xcoord = np.zeros(nglob)
ibool = np.zeros((ngll, nelmts)).astype(int)
for ielem in range(nelmts):
    xcoord[n*ielem : n*ielem+ngll] = elembnds[ielem] + elemsize[ielem]*(xigll+1)/2
    ibool[:, ielem] = np.arange(n*ielem, n*ielem+ngll)



# Plot domain setup for reference
fig, ax = plt.subplots(figsize=(15,1))
for ielem in range(nelmts):
    # Add element boindaries
    ax.axvline(xcoord[ibool[0, ielem]], color='k', zorder=0)
    ax.axvline(xcoord[ibool[-1, ielem]], color='k', zorder=0)
    # Plot GLL points
    ax.scatter(xcoord[n*ielem : n*ielem+ngll],
               (-1)**ielem + np.zeros(len(xcoord[n * ielem: n * ielem + ngll])),
               marker='o', edgecolor='k' )
    # Add some text for each element
    ax.text(x=xcoord[n*ielem+ ngll//2] , y=0, s=f'Element {ielem}',horizontalalignme

ax.set_xlim([-0.01, 1.01])
ax.set_ylim([-1.4, 1.4])
ax.set_yticks([]);
```

```
/tmp/ipykernel_4771/3114727472.py:103: DeprecationWarning: `trapz` is deprecated. Us
  weights.append(np.trapz(y=integrand, x=x))
```



Jacobian of the mapping

Next we need the jacobian. This requires computing $\frac{\mathrm{d}x}{\mathrm{d}\xi}$ at the GLL points. We know that the mapping between the real spatial domain $x \in [0, 1]$ and the natural domain, $\xi \in [-1, 1]$ is defined by

$$x(\xi) = \sum_{a=0}^{n} x_a \, \ell_a^n(\xi)$$

then the derivative is

$$\frac{\mathrm{d}x}{\mathrm{d}\xi} = \sum_{a=0}^{n} x_a \, \partial_\xi \ell_a^n(\xi)$$

and so to sample this at a specific GLL point, say ξ_γ, we have

$$J_\gamma = \frac{\mathrm{d}x}{\mathrm{d}\xi}(\xi_\gamma) = \sum_{a=0}^{n} x_a \, \partial_\xi \ell_a^n(\xi_\gamma)$$

Note that for this very simple mapping the jacobian is analytical and is constant across the domain. If the elements were not evenly spaced, it would not be constant. For each element we will calculate this Jacobian using the general method described above:

```
# Derivative of lagrange polynomial
# from Problem 7.17
# dlag1[i,j] is l^n'_i(xi_j)
dlag1 = dlagrange1(n)


# Compute jacobian:
# jac[i, ielem] is jacobian at
# gll point i within element ielem
jac = np.zeros((ngll, nelmts))
for ielem in range(nelmts):
    for gamma in range(ngll):
        # Evaluate sum over a
        for igll in range(ngll):
            jac[gamma, ielem] += xcoord[ibool[igll, ielem]] * dlag1[igll, gamma]
```

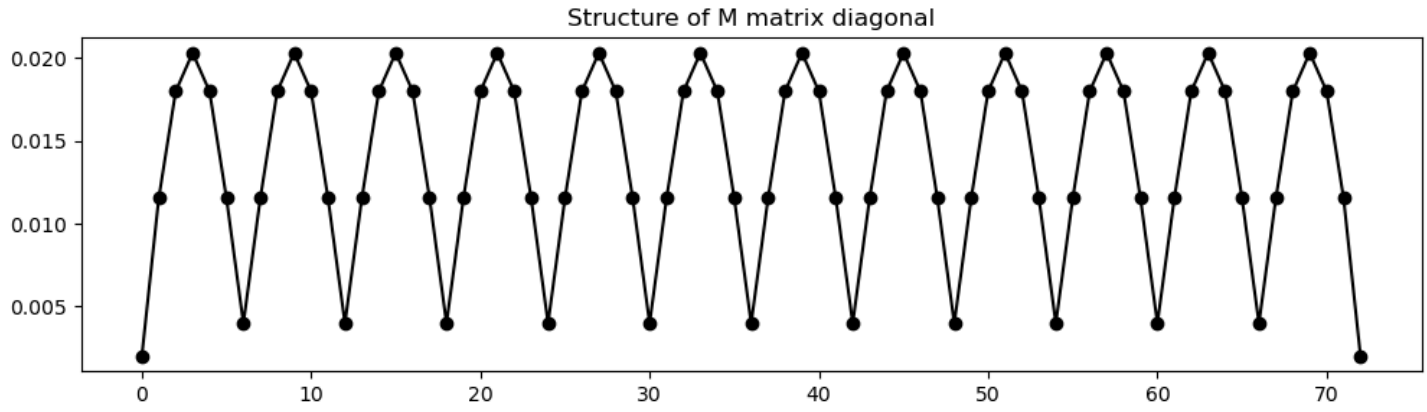
Matrix computation and assembly

Mass Matrix

The mass matrix is diagonal, so we will store it as a vector. It must be assembled globally. Lets do this below and look at the structure of it. For a homogeneous medium the matrix will be the same for each

element, except at the boundaries

```
# Mass matrix is diagonal so can be 1 x nglob
M = np.zeros(nglob)

# For each element we can construct
# the local (diagonal) matrix and assemble:
for ielem in range(nelmts):
    for igll in range(ngll):
        # global node ID
        iglob = ibool[igll, ielem]

        # local mass matrix is scalar
        mass_local = wgll[igll]*jac[igll,ielem]

        # add to global mass matrix
        M[iglob] +=  mass_local


# Plot the matrix diagonal for reference below
fig, ax = plt.subplots(figsize=(12,3))
ax.plot(np.arange(nglob), M, '-ok')
ax.set_title('Structure of M matrix diagonal');
```



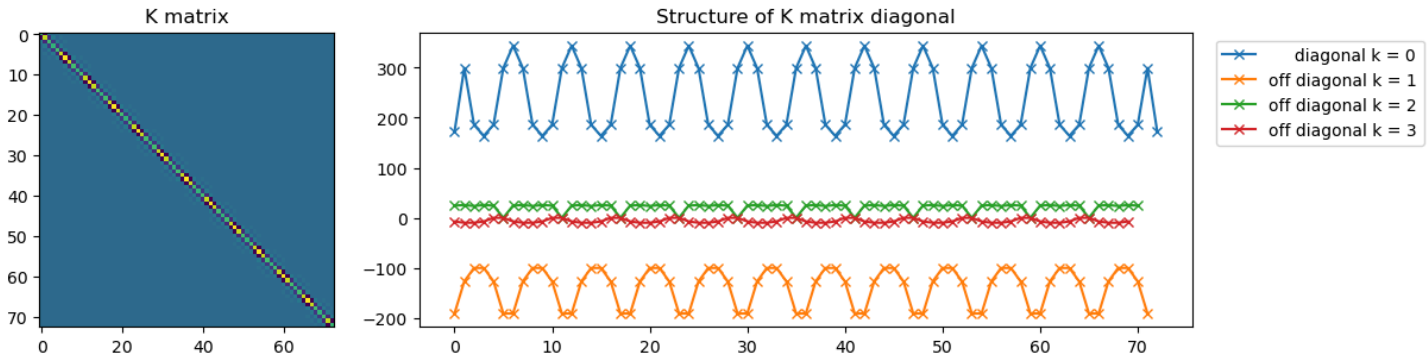
Stiffness matrix

The stiffness matrix is not diagonal, but does not necessarily need to be assembled. Consider the global system \begin{equation*} \mathsf{M}\dot{\mathsf{\theta}} = \mathsf{F} - \mathsf{K}\mathsf{\theta} \quad. \end{equation*} We observe that only the matrix-vector product $K\theta$ needs to be available globally. Hence, we could compute this vector locally for each element, and then assemble that vector globally. This avoids assembly of the global stiffness matrix. For our case when the degrees of freedom is small, we can afford to assemble the matrix. Below however, we include code that shows how to run the same solver without this global assembly.

You may find that the method avoiding global assembly is much slower, since we are effectively recomputing the local stiffness matrix at each timestep. There is therefore a tradeoff between performance and storage availability; if you can afford to store the whole K matrix in memory, it might be beneficial!

Below we see the stiffness matrix is a banded, block diagonal matrix `ngll` diagonals that are non-zero

```
AVOID_GLOBAL_ASSEMBLY = False


if not AVOID_GLOBAL_ASSEMBLY:
    # Initialise global stiffness matrix
    K = np.zeros((nglob, nglob))

    # Assemble K matrix : using Eqn 7.125
    for ielem in range(nelmts):

        # Loop over elements a,b of the local matrix
        for a in range(ngll):
            glob_id_row = ibool[a, ielem]

            for b in range(ngll):
                glob_id_col = ibool[b, ielem]

                kab = 0
                for g in range(ngll):
                    kab += wgll[g] * alpha[g,ielem] * dlag1[a,g] * \
                           dlag1[b,g] / jac[g, ielem]

                # Now lets add this kab to the global matrix:
                K[glob_id_row, glob_id_col] += kab


    # Plot K matrix:
    fig    = plt.figure(figsize=(12,3), constrained_layout=True)
    gs     = fig.add_gridspec(3, 10)
    axKmat = fig.add_subplot(gs[:, :3])
    axstrc = fig.add_subplot(gs[:, 3:])

    axKmat.imshow(K)

    for k in range(4):
        if k==0:
            label = f'     diagonal k = {k}'
        else:
            label = f'off diagonal k = {k}'
        axstrc.plot(np.arange(nglob-k), np.diag(K, k=k), '-x', label=label)

    axstrc.legend(bbox_to_anchor=(1.02, 1.))
    axstrc.set_title('Structure of K matrix diagonal');
    axKmat.set_title('K matrix');
```



Boundary coniditions

Next, we need to compute the force vector. There is no internal heating so Eqn 7.127 reduces to

$$f_\alpha = \alpha_n \sum_{\beta=0}^{n} \theta_\beta \ell'^n_\beta(\xi_n)\, \partial_x \xi(\xi_n) \; - \alpha_0 \sum_{\beta=0}^{n} \theta_\beta \ell'^n_\beta(\xi_0)\, \partial_x \xi(\xi_0)$$

where the two terms are contributed by the last and first elements respectively.

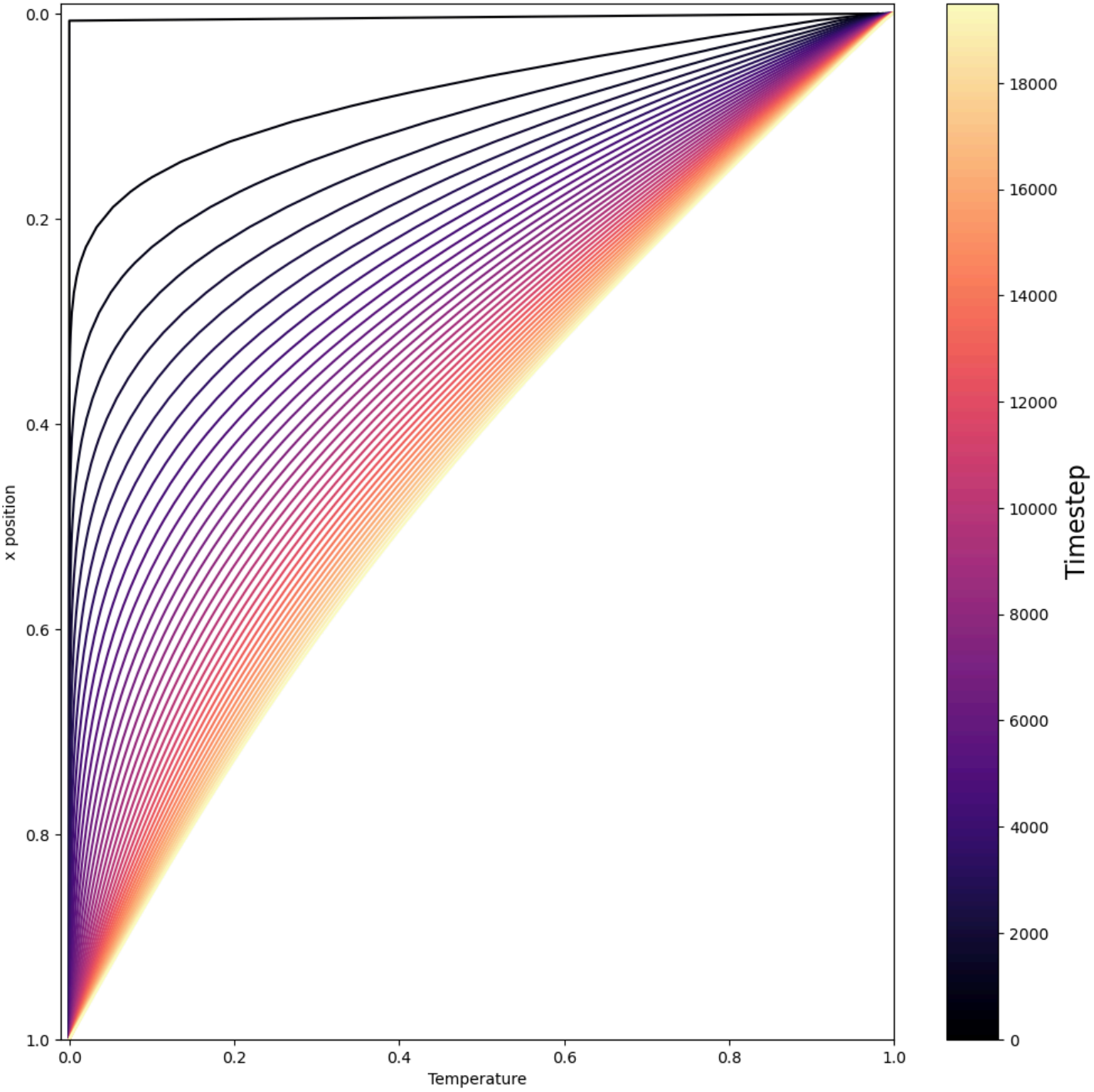
The initial condition sets $\theta(x, 0) = 0$ however the boundary condition also necessitates that $\theta(1,\, t) = 1$. Similar to Problem 7.14, this introduces a discrepency. In computing the force vector, we enforce that $\theta(1,\, t) = 1$.

Solver:

Finally, we are ready to run our solver! Below, we plot the temperature profile at different timesteps as the system evolves. We observe the system go from its heaviside-like temperature profile to a smooth, linear gradient in temperature with depth.

```
# Initialise the temperature
# and time-derivative of temperature
# Initial condition is 0 everywhere
theta           = np.zeros(nglob)
dtemperature_dt = np.zeros(nglob)

# The matrix-vector RHS product
rhs_global = np.zeros(nglob)

# Boundary temperatures we will enforce
temp0 = 1
tempN = 0


# Setup plotting
fig, ax = plt.subplots(figsize=(12,12))
plot_every = 500

# Colorbar for plotting through time
cmap = cm.magma
norm = mcolors.Normalize(vmin=0,
                         vmax=nsteps//plot_every-1)
sm = cm.ScalarMappable(cmap=cmap, norm=norm)
sm.set_array([])
cbar = fig.colorbar(sm, ax=ax)
cbar.set_label('Timestep', fontsize=16)
cbar.set_ticks(np.arange(0, nsteps//plot_every, 4))
cbar.set_ticklabels(np.arange(0, nsteps, 4*plot_every))


# Step in time
for istep in range(nsteps):

    # ------------ STEP 1: predictor step -----------------------
    theta += (1-eta) * dt * dtemperature_dt


    # ------------ STEP 2: solver step --------------------------
    if AVOID_GLOBAL_ASSEMBLY:
        # Avoid global assembly of K:
        rhs_global[:] = 0

        # Loop global elements
        for ispec in range(nelmts):
            for i in range(ngll):
                iglob = ibool[i, ispec]

                for j in range(ngll):
                    for k in range(ngll):
                        # Local K matrix:
                        diffusivity = wgll[k] * dlag1[i,k] *         \
                                      dlag1[j,k] / jac[k,ispec]
```

```
                        # Assembly global RHS (mat-vec product)
                        rhs_global[iglob] -= diffusivity *           \
                                             theta[ibool[j,ispec]]
    else:
        # Use globally assembled matrix:
        rhs_global =  -np.matmul(K, theta)

    # Add boundary condition forcing:
    theta[0]  = temp0
    theta[-1] = tempN

    # --- RHS boundary
    bNsum = 0
    for bb in range(ngll):
        bNsum += theta[ibool[bb,-1]] * dlag1[bb,-1] / jac[-1, -1]
    rhs_global[-1] += alpha[-1,-1]*bNsum

    # --- LHS boundary
    b0sum = 0
    for bb in range(ngll):
        b0sum += theta[ibool[bb,0]] * dlag1[bb,0] / jac[0,0]
    rhs_global[0] += -alpha[0,0]*bNsum


    # inverse of Mass matrix * RHS
    dtemperature_dt = rhs_global / M


    # ------------ STEP 3: corrector step ----------------------
    theta = theta + (eta * dt * dtemperature_dt)



    # Plot some of the timesteps
    if istep % plot_every ==0:
        ax.plot(theta, xcoord, color=cmap(norm(istep//plot_every)))



# Cosmetics for the plot:
ax.set_xlim([-0.01, 1]);
ax.set_ylim([1, -0.01]);
ax.set_ylabel("x position");
ax.set_xlabel("Temperature");
```



Problem 7.21

Here we produce a 1D SEM solver to the wave equation. As with Problem 7.20, this relies on a number of functions we developed in previous problems 7.17 - 7.19, which are copied in the hidden cell below. Similarly, a lot of the domain setup is identical to Problem 7.20, so is only commented on briefly.

▶ Show code cell source

Setting up the domain

Let us define our domain as the problem prescribes. Here we are using 7 GLL points and 12 elements for the domain $x \in [0,\ 1]$. Note that the timestep needs to be quite small here to satisfy the CFL condition since our wavespeed is 1, and the whole domain size is also 1.

```
# Domain discretisation
L      = 1           # Domain size
n      = 6           # Lagrange degree
ngll   = n + 1       # number of gll per element
nelmts = 12          # Number of elements
nbound = nelmts + 1  # Number of element boundaries

# Timestepping
dt        = 0.0005   # Timestep
nsteps    = 4000     # Number of timesteps
plotevery = 100      # Timestep to plot

# Physical properties
rho    = 1
mu     = 1


# Global coordinates of element boundaries:
elembnds = np.linspace(0, L, nbound)
# Length of each element
elemsize = elembnds[1:] - elembnds[:-1]

# Get GLL and weights:
xigll, wgll = weights(n)

# Create an array of the GLOBAL node coordinates:
nglob = nelmts*(ngll-1) + 1

# Array of global node x coordinates
# and ibool (lobal --> global mappping)
xcoord = np.zeros(nglob)
ibool = np.zeros((ngll, nelmts)).astype(int)
for ielem in range(nelmts):
    xcoord[n*ielem : n*ielem+ngll] \
       = elembnds[ielem] + elemsize[ielem]*(xigll+1)/2
    ibool[:, ielem] = np.arange(n*ielem, n*ielem+ngll)

# Derivative of lagrange polynomials
dlag1 = dlagrange1(n)


# Compute jacobian
jac = np.zeros((ngll, nelmts))
for ielem in range(nelmts):
    for gamma in range(ngll):

        # Evaluate sum over a
        for igll in range(ngll):
            jac[gamma, ielem] +=             \
                  xcoord[ibool[igll, ielem]] \
                  * dlag1[igll, gamma]
```

```
/tmp/ipykernel_4794/2521511565.py:100: DeprecationWarning: `trapz` is deprecated. Us
  weights.append(np.trapz(y=integrand, x=x))
```

Assembly of the matrices

Two major benefits of the Spectral Element Method are:

- The diagonal mass matrix
- No requirement to assemble the global stiffness matrix

Note here that the global stiffness matrix is a square matrix with dimensions equal to the global degrees of freedom. This becomes prohibative to assemble for realistic simulations. As we will see below, we only ever require access to the matrix-vector product $\mathsf{K}\mathbf{s}$, which is a vector. Hence, in real SEM software the matrix-vector product is computed at the elemental level, and only the resultant vector is assembled. Here, however, we will assemble the stiffness matrix for clarity.

The two elemental matrices are described in Eqns. 7.132 and 7.136 for the mass matrix and stiffness matrix respectively.

```
# Initialise mass 'matrix'
# only a vector since diagonal
M = np.zeros(nglob)

# For each element we can construct the local
# matrix and assemble:
for ielem in range(nelmts):
    for igll in range(ngll):
        iglob = ibool[igll, ielem]
        mass_local = wgll[igll] * jac[igll,ielem] * rho
        M[iglob] +=  mass_local

# Create the K matrix even though we dont
# need to assemble it technically
K = np.zeros((nglob, nglob))

# Eqn. 7.136
for ielem in range(nelmts):
    # Loop over GLL pts (a,b)
    for a in range(ngll):
        iglob_a = ibool[a, ielem]

        for b in range(ngll):
            iglob_b = ibool[b, ielem]

            # Sum over gamma:
            gsum = 0
            for g in range(ngll):
                gsum += (wgll[g] * mu * dlag1[a,g] *
                         dlag1[b,g] / jac[g,ielem])
            # Assemble to global matrix
            K[iglob_a, iglob_b] += gsum
```

A comment on the initial condition

The initial condition within the problem statement,

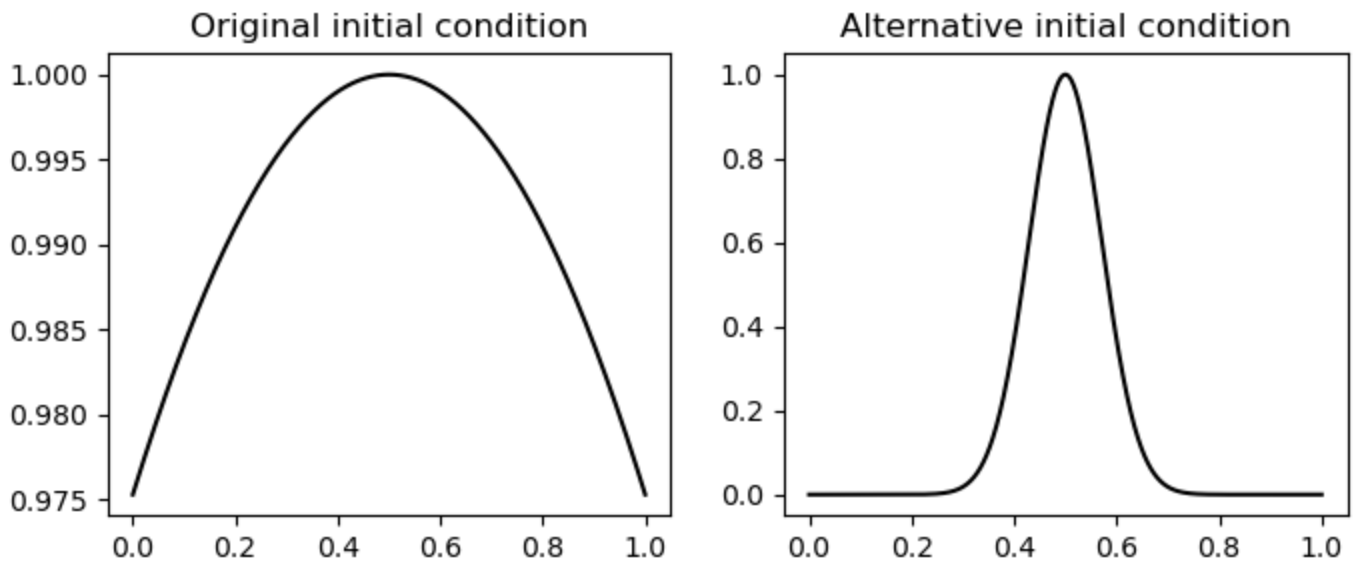
$$s(x, 0) = \exp\left[-0.1(x - 0.5)^2\right]$$

is plotted below. Note however that this is a very long-wavelength displacement relative to the size of the domain. As such, the resultant wave propagation resembles that of a plucked guitar string. We will also simulate a second, shorter-wavelength initial condition to observe clearer propagation of the wave and interaction with the boundary. This initial condition will be

$$s(x,\,0) = \exp\left[-100 * (x - 0.5)^2\right]$$

which is also plotted below:

▶ Show code cell source



Neumann boundary condition

Let us start with the Neumann boundary condition since it is easier! Note that the boundary conditions

$$\partial_x s(0, t) = 0$$
$$\partial_x s(L, t) = 0$$

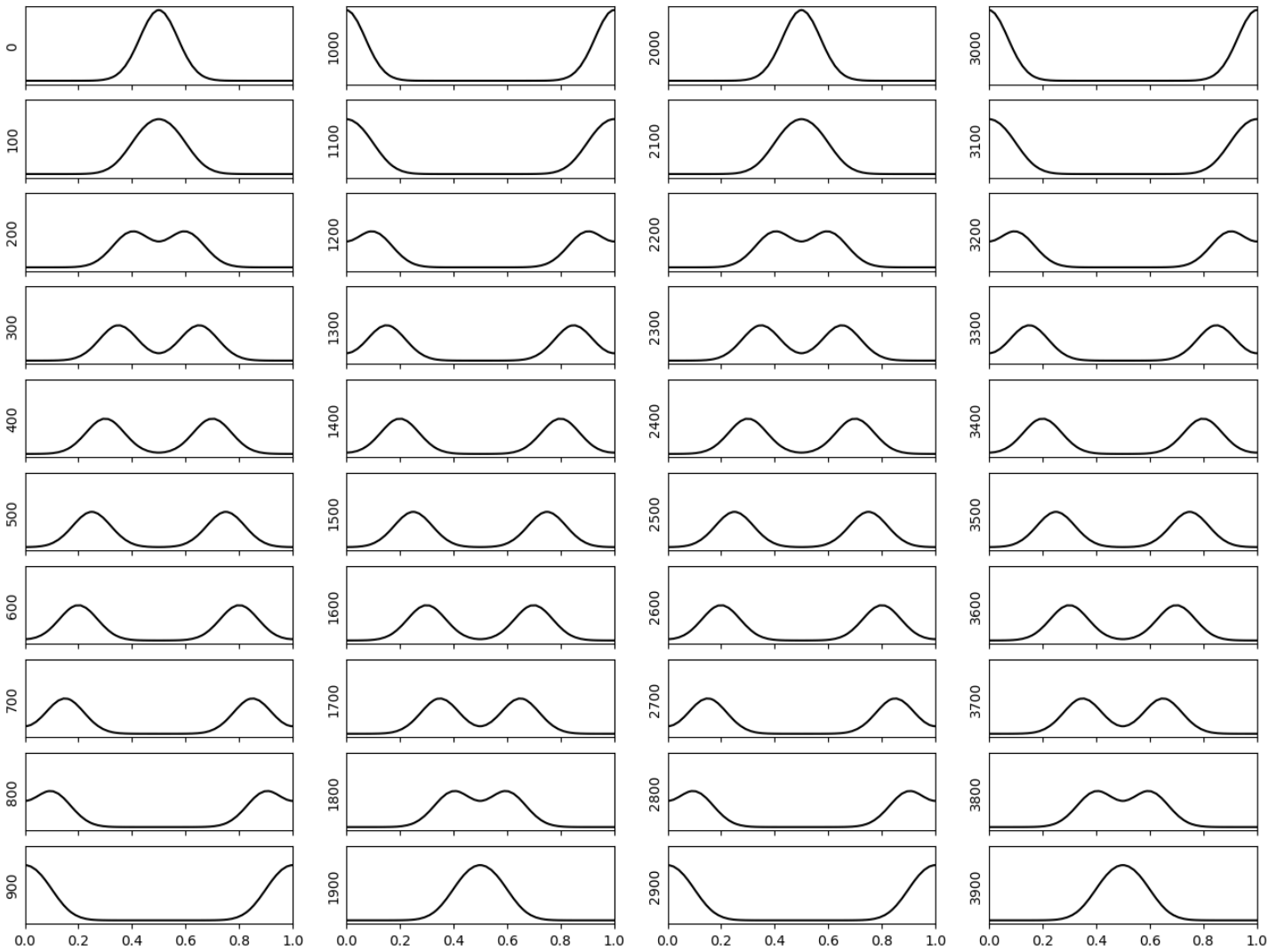
are naturally satisfied by setting the forcing term, f_α, (Eqn. 7.138) equal to zero. That is, we only need to solve a modified version of the system of equations in Eqn. 7.137 as

$$\sum_{\beta=0}^{n} (m_{\alpha\beta}\ddot{s}_\beta + k_{\alpha\beta}s_\beta) = 0$$

Alternative initial condition

Below, we simulate first for our alternative initial condition. We observe the propagation of the wave away from the centre. At the boundaries, the stress-free surface allows for non-zero displacement.

```
# Initialise the disp, vel, acc:
disp = np.zeros(nglob)
vel  = np.zeros(nglob)
acc  = np.zeros(nglob)

disp = np.exp(- ((xcoord-0.5)**2) /0.01  )

# Plot the initial condition
fig, ax = plt.subplots(10,4, figsize=(16,12), sharex=True, sharey=True)

# Loop in time
iax  = 0
icol = 0
for istep in range(nsteps):

    # Predictor:
    newdisp = disp + dt*vel + 0.5*dt*dt*acc
    newvel  = vel  + 0.5*dt*acc
    newacc  = acc*0

    # Solver:
    Da = - np.matmul(K, newdisp)/ M

    # Corrector:
    newacc  += Da
    newvel  += 0.5*dt*newacc

    # Now need to copy to old arrays:
    disp[:] = copy(newdisp[:])
    vel[:] = copy(newvel[:])
    acc[:] = copy(newacc[:])


    if istep%plotevery == 0:
        ax[iax, icol].plot(xcoord, disp, 'k')
        ax[iax, icol].set_ylabel(istep)
        ax[iax, icol].set_yticks([])

        iax += 1
        if iax==10:
            icol +=1
            iax  = 0

ax[0,0].set_xlim([0,1]);
```

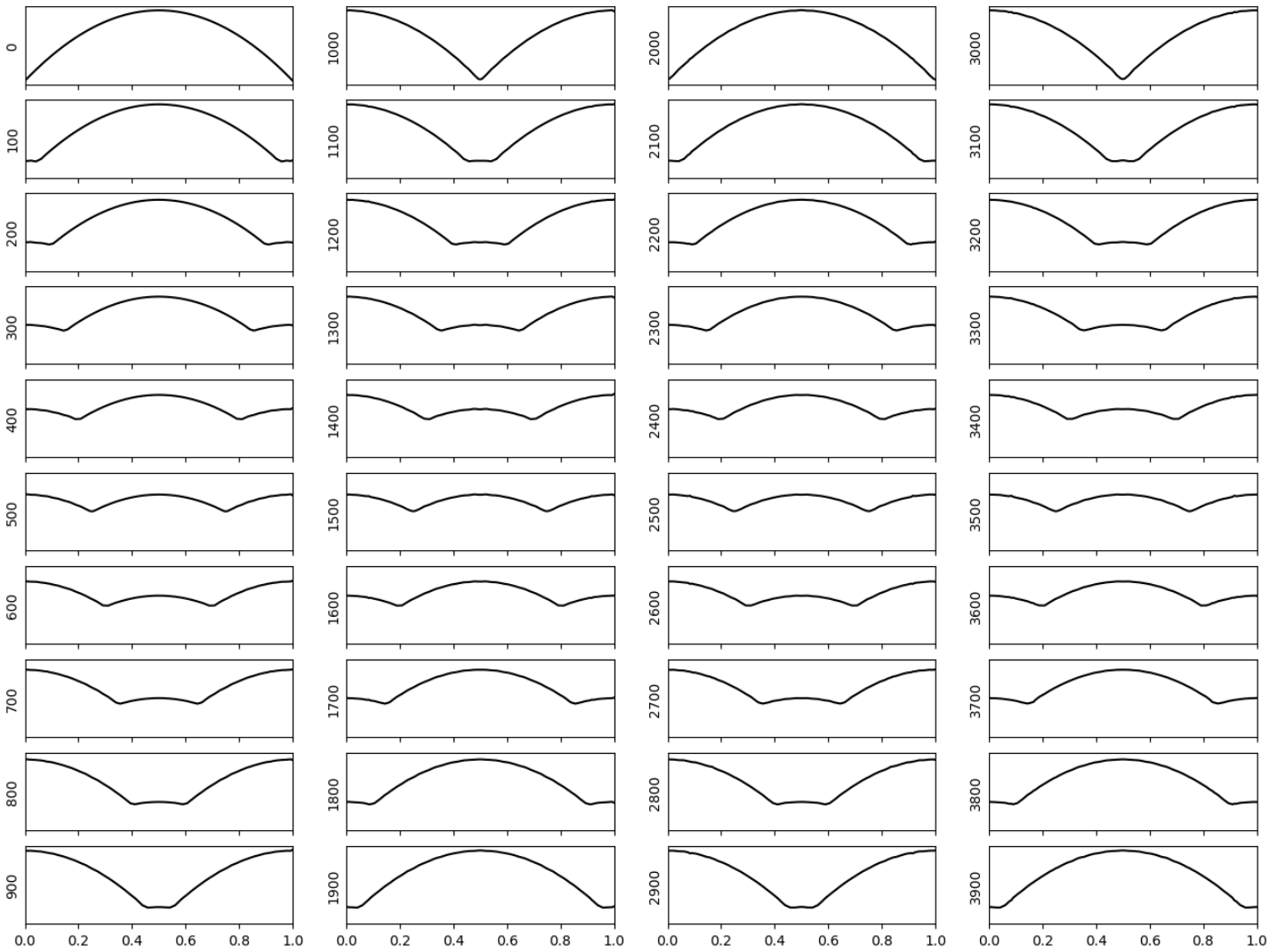


We observe the wave propagating to the boundary and being reflected.

Original initial condition

Next, let us re-run the code with the original initial condition. This is the type of propagation one would see when plucking a (broken!) guitar string, if the ends of the string were not fixed, perhaps more similar to wobbling a sheet of plastic.

```
# Initialise the disp, vel, acc:
disp = np.zeros(nglob)
vel  = np.zeros(nglob)
acc  = np.zeros(nglob)

nsteps     = 4000
disp = np.exp(- 0.1*((xcoord-0.5)**2))


fig, ax = plt.subplots(10,4, figsize=(16,12), sharex=True, sharey=True)

# Loop in time
iax  = 0
icol = 0
for istep in range(nsteps):

    # Predictor:
    newdisp = disp + dt*vel + 0.5*dt*dt*acc
    newvel  = vel  + 0.5*dt*acc
    newacc  = acc*0

    # Solver:
    Da = - np.matmul(K, newdisp)/ M

    # Corrector:
    newacc  += Da
    newvel  += 0.5*dt*newacc

    # Now need to copy to old arrays:
    disp[:] = copy(newdisp[:])
    vel[:]  = copy(newvel[:])
    acc[:]  = copy(newacc[:])

    if istep%plotevery == 0:
        ax[iax, icol].plot(xcoord, disp, 'k')
        ax[iax, icol].set_ylabel(istep)
        ax[iax, icol].set_yticks([])

        iax += 1
        if iax==10:
            icol +=1
            iax  = 0

ax[0,0].set_xlim([0,1]);
```



Dirichlet boundary condition

Finally, let us implement a Dirichlet boundary condition. The code is very similar, except we must now implement the forcing term as a contribution to the RHS. This acts only on the GLL points that represent the boundary of the domain.

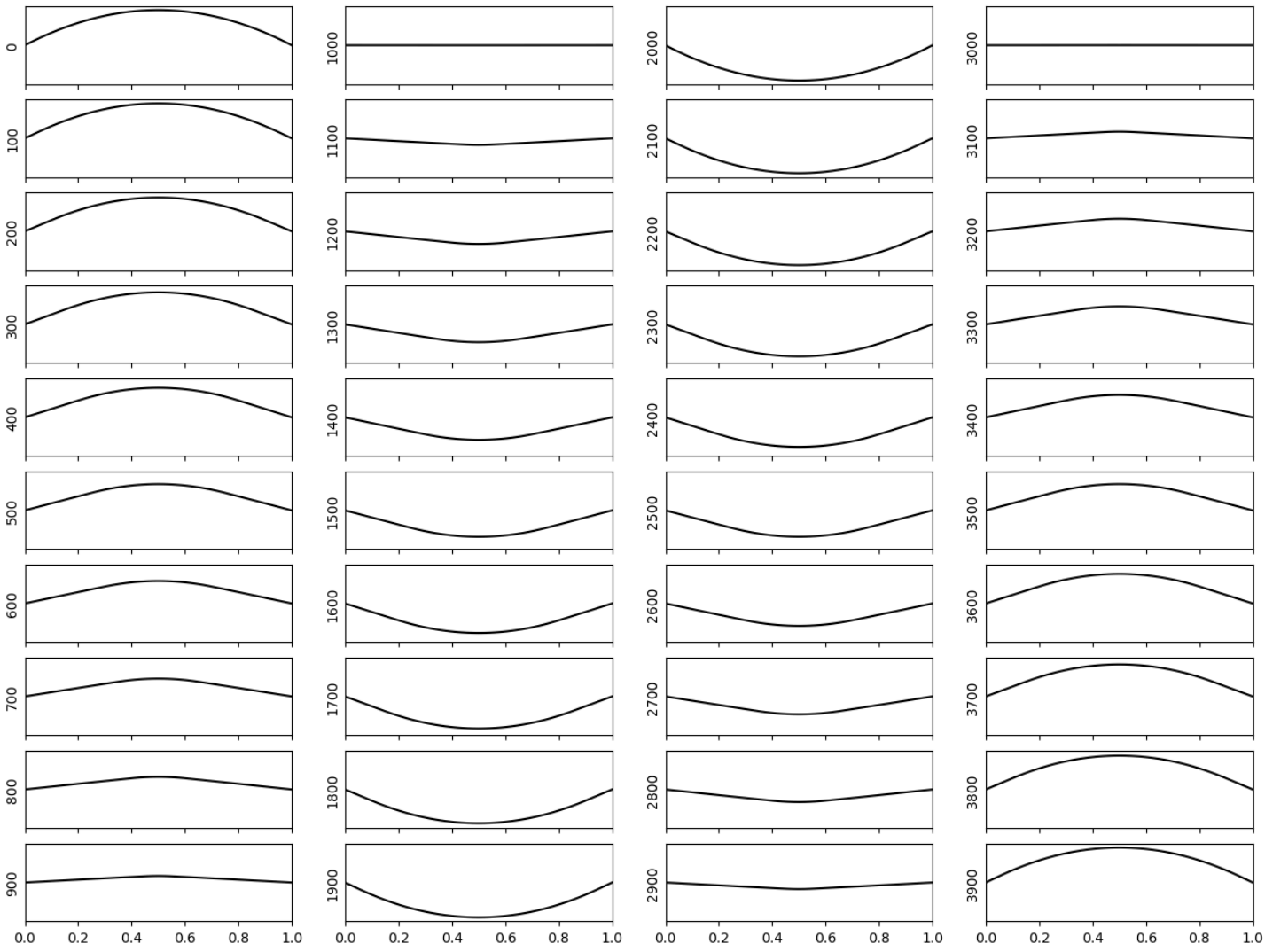
Original initial condition

Note: the original initial condition specified in the question has displacement values of ~ 0.97 at the edges of the domain, yet the dirichlet condition says the boundaries should have a displacement of 0. Hence, let us modify the initial condition by a constant so that the edges of the domain are initially at 0, i.e.

$$s(x,\,0) = \exp\left[-0.1(x-0.5)^2\right] - 0.97\ldots$$

```
# Initialise the disp, vel, acc:
disp = np.zeros(nglob)
vel  = np.zeros(nglob)
acc  = np.zeros(nglob)

# Apply initial condition:
disp = np.exp(- 0.1*((xcoord-0.5)**2))
# Shift by constant to get boundaries at y=0
disp -= disp[0]

# Plot the initial condition
fig, ax = plt.subplots(10,4, figsize=(16,12), sharex=True, sharey=True)

# Loop in time
iax  = 0
icol = 0
for istep in range(nsteps):

    # Predictor:
    newdisp = disp + dt*vel + 0.5*dt*dt*acc

    # (1) Force boundary condition on displacement
    newdisp[0]  = 0
    newdisp[-1] = 0

    # (2) finish predictor stage
    newvel  = vel+ 0.5*dt*acc
    newacc  = acc*0

    # Solver:
    Da = -np.matmul(K, newdisp)

    # Add boundary terms for first/ last gll points
    bNsum = 0
    for bb in range(ngll):
        bNsum += disp[ibool[bb,-1]] * dlag1[bb,-1] / jac[-1, -1]

    Da[-1] += wgll[-1]*mu*bNsum

    b0sum = 0
    for bb in range(ngll):
        b0sum += disp[ibool[bb,0]] * dlag1[bb,0] / jac[0, 0]
    Da[0] -= wgll[0]*mu*b0sum

    # Divide by mass matrix
    DaM = Da/M

    # Corrector:
    newacc  += DaM
    newvel  += 0.5*dt*newacc


    # Now need to copy to old arrays:
```

```
    disp[:] = copy(newdisp[:])
    vel[:]  = copy(newvel[:])
    acc[:]  = copy(newacc[:])

    # Once again force boundary conidition just in case
    disp[0] = 0
    disp[-1] = 0


    if istep%plotevery == 0:
        ax[iax, icol].plot(xcoord, disp, 'k')
        ax[iax, icol].set_ylabel(istep)
        ax[iax, icol].set_yticks([])

        iax += 1
        if iax==10:
            icol +=1
            iax  = 0

ax[0,0].set_xlim([0,1]);
```

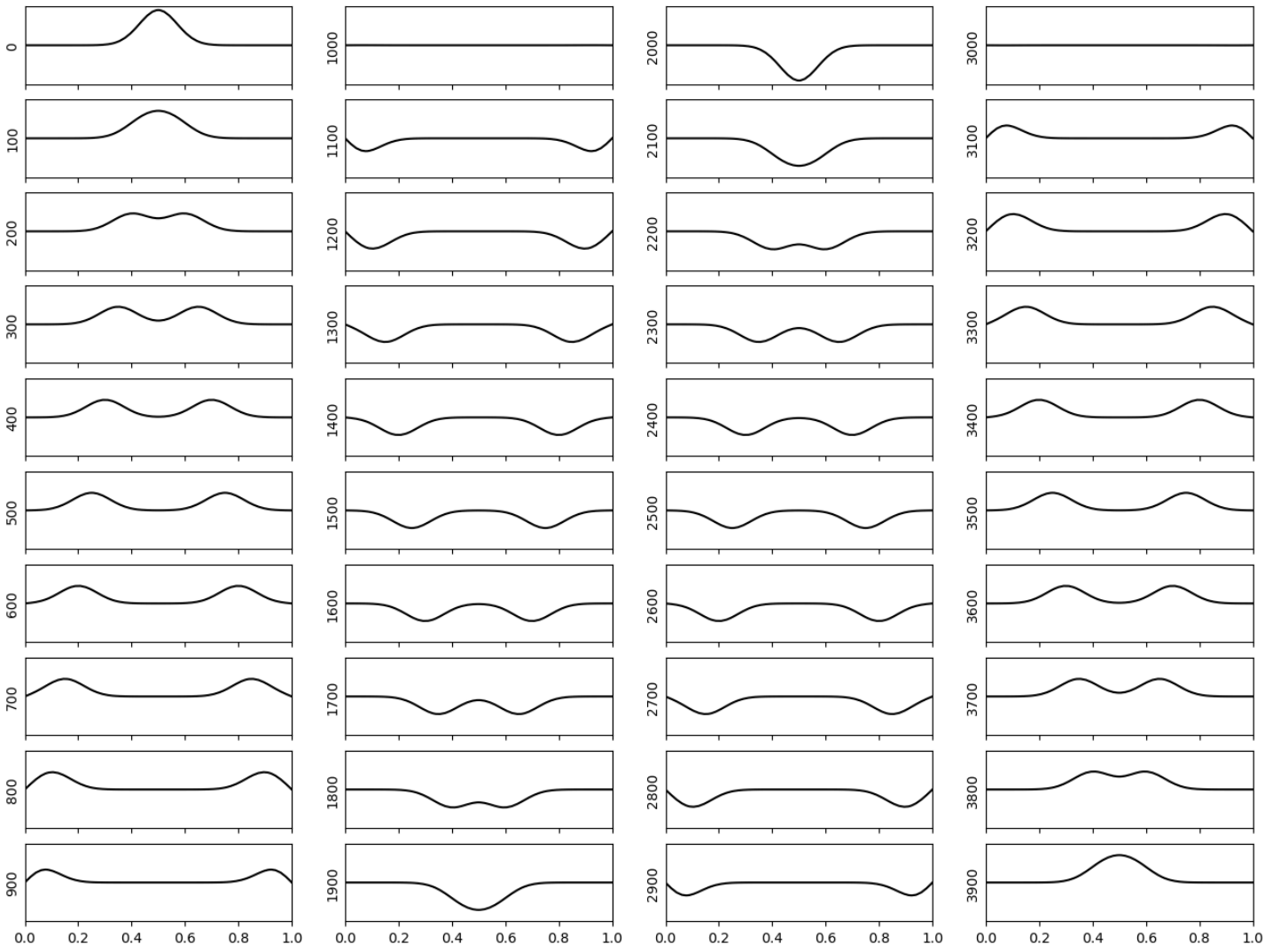


With the Dirichlet condition fixing the boundaries, this type of propagation is analgous to plucking a guitar string.

Alternative initial condition

Finally, let us consider the alternative initial condition to observe spatial propagation of a small wave. The code is identical to above, but using the alternative initial condition:

▶ Show code cell source



Due to the fixed boundaries, we see a reflection of the wave where the polarity is flipped.