Sebastian Straszak, s1728659, Grid Hydrodynamics Work

Contents

1 Workshop 1 Practice Assignment

I didn't submit it at the time because I was fairly behind on a lot of things... but I caught up now! I know you don't have to mark it or anything but.. I mean I wouldn't complain :(

The three 1st-order finite-difference approximations yield (somewhat) wildly different results as the grid spacing is reduced when it comes to calculating the gradient of e^x about the origin. Pure values can be seen in Table 1, with (absolute) errors in table 2

In respect to the value, forward and centred differencing overestimate the true value, while backward underestimates. In the case of forward differencing,

$$\frac{d}{dx}\left(e^x\right) \approx \frac{e^{\Delta x} - e^0}{\Delta x} \tag{1}$$

$$e^{\Delta x} = 1 + \Delta x + \frac{1}{2!}\Delta x^2 + \frac{1}{3!}\Delta x^3 + \ldots \tag{2}$$

$$\therefore \frac{d}{dx}\left(e^x\right) \approx \frac{\Delta x + \frac{1}{2!}\Delta x^2 + \ldots}{\Delta x} = 1 + \frac{\frac{1}{2!}\Delta x^2 + \frac{1}{3!}\Delta x^3 + \ldots}{\Delta x} = 1 + \frac{1}{2!}\Delta x + \frac{1}{3!}\Delta x^2 + \ldots \tag{3}$$

which clearly shows that there'll be a whole sleuth of error terms in 1st order and above that cause overestimation. Backward differencing is the exact same scenario with a caveat: working the problem yields

$$\frac{d}{dx}\left(e^x\right) \approx \frac{e^0 - e^{-\Delta x}}{\Delta x} \tag{4}$$

$$e^{-\Delta x} = 1 - \Delta x + \frac{1}{2!}\Delta x^2 - \frac{1}{3!}\Delta x^3 + \ldots \tag{5}$$

$$\therefore \frac{d}{dx}\left(e^x\right) \approx= 1 - \frac{\frac{1}{2!}\Delta x^2 - \frac{1}{3!}\Delta x^3 + \ldots}{\Delta x} = 1 - \left(\frac{1}{2!}\Delta x - \frac{1}{3!}\Delta x^2 + \ldots\right) \tag{6}$$

which demonstrably shows that backward differencing underestimates, but the magnitude of the error is reduced: all odd-powers of our grid spacing have been signed negative, reducing error relative to forward differencing. Clearly backward differencing the gradient of e^x is expected to be more accurate.

Numerically, this is seen in the results: both forward and backward differencing yield fairly large errors (relative to centred differencing- more on that later) but backward differncing comes out as the victor. Note that the terms that make backward better than forward are quadratic and onward in Δx- as the timestep is reduced, the error terms will equalize: this is also seen, as by a timestep of 0.02s, these schemes are within a relative difference of 1% of eachother.

Centred differencing yields the finest result, and is guaranteed to overestimate for our situation. This can be seen via the same analysis as above,

$$\frac{d}{dx}\left(e^x\right) \approx \frac{e^{\Delta x} - e^{-\Delta x}}{\Delta x} \tag{7}$$

$$\frac{e^{\Delta x} - e^{-\Delta x}}{\Delta x} = \frac{\left(\Delta x + \frac{1}{2!}\Delta x^2 + \frac{1}{3!}\Delta x^3 + \ldots\right) - \left(-\Delta x + \frac{1}{2!}\Delta x^2 - \frac{1}{3!}\Delta x^3 + \ldots\right)}{2\Delta x} \tag{8}$$

$$\therefore \frac{d}{dx}\left(e^x\right) \approx 1 + \frac{2}{3!}\Delta x^2 + \ldots \tag{9}$$

where all overestimates are second order and beyond. This indicates that centred differencing approximation should be more accurate than either forward or backward (which error 1st order with Δx and this is indeed observed, with centred differencing yielding the finest results.

It's possible to verify (roughly) that the above actually holds somewhat well- compare the estimates for centred and forward as an example:

$$\frac{d}{dx}\left(e^x\right)_{\text{forwrd}} - \frac{d}{dx}\left(e^x\right)_{\text{centrd}} \approx \frac{1}{2!}\Delta x - \frac{1}{3!}\Delta x^2 + \ldots \tag{10}$$

This gets somewhat close to estimating the numerical differences once you get down to smaller timesteps (10^{-1} and below seem somewhat reasonable) though to reasonably estimate larger steps, more terms are required.

Δx	Forward	Backward	Centred
1.000000e+00	1.718281828	0.632120559	1.175201194
1.000000e-01	1.051709181	0.951625820	1.001667500
1.000000e-02	1.005016708	0.995016625	1.000016667
1.000000e-03	1.000500167	0.999500167	1.000000167
1.000000e-04	1.000050002	0.999950002	1.000000002
1.000000e-05	1.000005000	0.999995000	1.000000000
1.000000e-06	1.000000500	0.999999500	1.000000000
1.000000e-07	1.000000049	0.999999950	0.999999999
1.000000e-08	0.999999994	1.000000005	0.999999999

Table 1: Finite-difference approximations of $\frac{d}{dx}\left(e^x\right)$ about 0, repeated Table 2.1 from Bodenheimer

Δx	Forward	Backward	Centred
1.000000e+00	0.718281828	0.367879441	0.175201194
1.000000e-01	0.051709181	0.048374180	0.001667500
1.000000e-02	0.005016708	0.004983375	0.000016667
1.000000e-03	0.000500167	0.000499833	0.000000167
1.000000e-04	0.000050002	0.000049998	0.000000002
1.000000e-05	0.000005000	0.000005000	0.000000000
1.000000e-06	0.000000500	0.000000500	0.000000000
1.000000e-07	0.000000049	0.000000050	0.000000001
1.000000e-08	0.000000006	0.000000005	0.000000001

Table 2: Error in finite-difference approximations, absolute, for each step. True value should be $e^0 = 1$.

2 Practice Workshop 2 - t.b.marked. Part 1.

2.1 The Method and Theory

The equation to be modelled is given by

$$\partial_t u(x,t) = -u(x,t) \tag{11}$$

where for simplicity the x term has been ignored in seeking the simplest possible rendition of the problem:

$$\partial_t u(t) = -u(t) \tag{12}$$

The general solution to this is

$$u(t) = Ae^{-t} \tag{13}$$

where our model is concerned with the case where

$$\begin{align} u(0) &= 1 \tag{14} \\ \therefore A &= 1 \tag{15} \end{align}$$

Three different time-differencing routines are to be trialled in attacking Equation (12) numerically:

1. Forward
2. Backward
3. Centred (average)

These can be derived and are given, for this problem, with grid spacing Δt in time, as

$$\begin{align} u_j^{n+1} &= u_j^n - u_j^n \Delta t \tag{16} \\ u_j^{n+1} &= u_j^n - u_j^{n+1} \Delta t \tag{17} \\ u_j^{n+1} &= u_j^n - \frac{u_j^{n+1} + u_j^n}{2} \Delta t \tag{18} \end{align}$$

The forward scheme relies purely on the current state n to deduce the forward state $n+1$. Backward and Centred, however, satisfy

$$u_j^{n+1} = F\left(u_j^n, u_j^{n+1}\right) \tag{19}$$

and are thus implicit schemes: they can be rearranged into pseudo-explicit form, however. The integration methods for these three schemes can be given as

$$u_j^{n+1} = u_j^n(1 - \Delta t) \tag{20}$$

$$u_j^{n+1} = \frac{u_j^n}{1 + \Delta t} \tag{21}$$

$$u_j^{n+1} = u_j^n \frac{2 - \Delta t}{2 + \Delta t} \tag{22}$$

We can examine which of these schemes provide the tightest fit to our analytic solution via some error analysis. Considering an analysis somewhat similar to that of 1, and leveraging Equation (12),

$$u_j^{n+1} = u_j^n + \partial_t u \Delta t + \frac{1}{2!}\partial_t^2 u \Delta t^2 + \frac{1}{3!}\partial_t^3 u \Delta t^3 + \ldots \rightarrow u_j^{n+1} = u_j^n - u_j^n \Delta t - \frac{1}{2!}\partial_t u \Delta t^2 - \frac{1}{3!}\partial_t^2 u \Delta t^3 + \ldots \tag{23}$$

The forward approximation is right there in the writing- the first two terms on the far right. There are, however, a whole sleuth of error terms, all summed up nice and linearly. Clearly, using the forward approximation on our problem yields

$$u_j^{n+1} + \frac{1}{2!}\partial_t u \Delta t^2 + \frac{1}{3!}\partial_t^2 u \Delta t^3 + \ldots = \left(u_j^{n+1}\right)_{\text{forward}} = u_j^n(1 - \Delta t) \tag{24}$$

which can be re-written (ostensibly) using Equation (12) as

$$u_j^{n+1} + \frac{1}{2!}\partial_t u \Delta t^2 - \frac{1}{3!}\partial_t u \Delta t^3 + \ldots = u_j^n(1 - \Delta t) \tag{25}$$

Now consider finding the error for the backward estimation. The error on this can be found via

$$\left(u_j^{n+1}\right)_{\text{backward}} - u_j^{n+1} = u_j^n(1 - \Delta t + \Delta t^2 - \Delta t^3 + \ldots) - \left(u_j^n - u_j^n \Delta t - \frac{1}{2!}\partial_t u \Delta t^2 - \frac{1}{3!}\partial_t^2 u \Delta t^3 + \ldots\right) \tag{26}$$

$$\left(u_j^{n+1}\right)_{\text{backward}} - u_j^{n+1} = -\partial_t u \Delta t^2 + \partial_t \Delta t^3 + \ldots - \left(-\frac{1}{2!}\partial_t \Delta t^2 + \frac{1}{3!}\partial_t u \Delta t^3 + \ldots\right) \tag{27}$$

$$\therefore u_j^{n+1} - \frac{1}{2!}\partial_t \Delta t^2 + \left(1 - \frac{1}{3!}\right)\partial_t u \Delta t^3 + \ldots = \left(u_j^{n+1}\right)_{\text{backward}} \tag{28}$$

The backward estimation has a stronger error tolerance![1] The cubic term in Δt^3 is more positive than that for the forward estimation, and acts to reduce the error contribution from the quadratic term. Consequently, the backward estimator, *for this given equation*, provides a more robust scheme, or it should. We'll see (it does actually.)

The same analysis can be carried out for the centred estimator by combining these results (since the centred estimator is just the average of the forward and backward estimators, implicitly) and obviously provides the best estimator for the problem. Given that the backward estimate is closer to the real deal than the forward, the forward contributes more error to the average, and thus we can tentatively posit that the centred estimator will be biased with an error with the same sign as the forward estimator.

[1] Holy mother of Christ I am never doing LaTeXagain in my life

2.2 The Results

Starting from the boundary conditions afforded, these three differencing routines have been leveraged. For each routine, the fractional error

$$\frac{\epsilon(t)}{w(t)} = \frac{u(t) - w(t)}{w(t)} \tag{29}$$

has been found over the simulation period, also. These have been plotted, alongside for the analytic solution, in Figure 1.

As can be seen, forward differencing underestimates the solution. Backward, however, overestimates it (though by a slightly lower amount- see perpendicular distance to the analytic solution in the inset axis on the upper axis of the graph in Figure 12.) Centred differencing underestimates, also, however by a significantly lower magnitude than the forward differencing scheme.

This all matches expectation: we anticipated forward differencing to be the worst, backward the second best, and centred to be the best. We also anticipated centred and forward errors to share the same sign: in our $\frac{\epsilon(t)}{w(t)}$ plot, they clearly do. At all points, backward error is less than forward error.

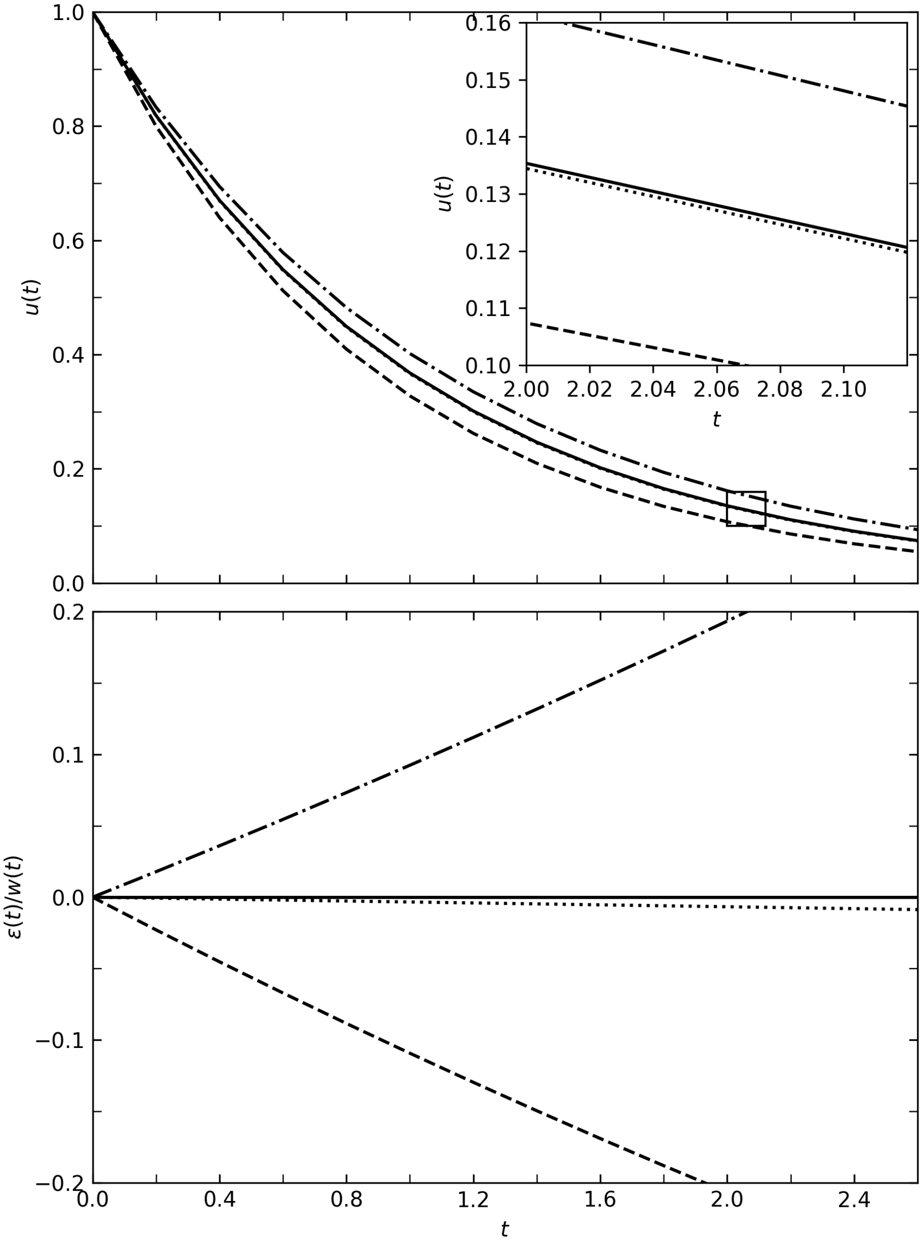

Figure 1: Solutions to the differential $\frac{du}{dt} = -u(t)$ with $u(0) = 1$. Forward, Backward and Centred differencing in time are represented by dashed, dashed-dotted, and dotted lines. The analytic solution $w(t)$ is represented by the solid line. The upper axis displays the solution, with an inset to a certain region displaying finely how the estimates differ from the analytic solution. The lower axis displays the fractional error relative to the analytic solution as a function of time: $\frac{u-w}{w}$.

3 Practice Workshop 2 - t.b.marked. Part 2

3.1 Method and Theory

The mass-continuity equation satisfies

$$\partial_t \rho(\vec{x}, t) + \nabla \cdot (\rho \vec{v}) = 0 \tag{30}$$

where for our one-dimensional case, considering an incompressible flow, we obtain

$$\partial_t \rho(x, t) + v \partial_x \rho = 0 \tag{31}$$

We aim to get a numerical model for this simplified physical model. We use the scheme[2]

$$\rho^{n+1}_{j+0.5} = -d\left(\rho^n_{j+0.5} - \rho^n_{j-0.5}\right) + \rho^n_{j+0.5} \tag{32}$$

which operates on a staggered grid[3] with timesteps Δt from initial condition of $t = 0$, such that

$$\rho^n_{j+0.5} = \rho\left((j + 0.5)\Delta x, n\Delta t\right) \tag{33}$$

This scheme is accurate to first order Δt. d is a constant satisfying $d = v\frac{\Delta t}{\Delta x}$. Carrying out some error analysis (and for simplification ignoring the staggered nature of the problem) yields

$$\rho^{n+1}_{j+1} = -d\left(\rho^n_{j+1} - \rho^n_j\right) + \rho^n_{j+1} \tag{34}$$

$$\rho^n_{j+1} + \partial_t \rho_{j+1} \Delta t + \frac{1}{2}\partial_t^2 \rho_{j+1} \Delta t^2 + \ldots = -d(\Delta x \partial_x \rho^n_j + \Delta x^2 \partial_x^2 \rho^n_j + \ldots) + \rho^n_{j+1} \tag{35}$$

$$\rho^n_{j+1} + \partial_t \rho_{j+1} \Delta t + \frac{1}{2}\partial_t^2 \rho_{j+1} \Delta t^2 + \ldots = -v\frac{\Delta t}{\Delta x}(\Delta x \partial_x \rho^n_j + \Delta x^2 \partial_x^2 \rho^n_j + \ldots) + \rho^n_{j+1} \tag{36}$$

$$\partial_t \rho + v \partial_x \rho = \frac{1}{2} v \Delta x \partial_x^2 \rho - \frac{1}{2}\Delta t \partial_t^2 \rho + \ldots \tag{37}$$

where all the second order terms have been dumped and we've just jumped straight to ρ for ease. Our original equation was Equation (31): this equation, the result of our discretisation, runs amok with a swath of terms that we did not wish to include in our physics. In realistic simulations, the diffusion term

$$\partial_t \rho \ \propto \ \nabla^2 \rho \tag{38}$$

usually dominates these terms. It "diffuses" our solution through the coordinate system of choice, and is attributed to a term called "Numerical Viscosity," in homage (I assume) to viscous drag diffusing momentum from a projectile through its medium via shear. Other terms can skew the numerical result from the consistent physical result, too, though diffusion is generally the strongest (mind you there is a wave term, though, $\partial_t^2 \propto \nabla^2$.)

[2] I don't have a bibtex set up for this, but it's from Bodenheimer.

[3] I mention in the caption of Figure 2- debugging reasons. I ended up sticking with a staggered grid despite using a regular grid initially- read there to know why. I know you recommended not to but... time is short and I had no time to change it back...

In the context of an incompressible flow simulation, the diffusion term can effectively smooth out a travelling density profile that should otherwise remain constant.[4]

3.2 Results and Analysis

The results of this model are shown in Figure 2. The model has a timestep of $\Delta t = 0.002$ and grid spacing $\Delta x = 1.0$, starting at $t = 0$, with

$$\rho^{0}_{j+0.5} = \left\{ \begin{array}{lr} 1 & 0 \leqslant j < 5 \\ 0.5 & 6 \leqslant j < 50 \\ 0 & 51 \leqslant j \end{array} \right\}$$

A source term, constant, satisfying $\rho^{n}_{0-0.5} = 1$, has been placed on the grid as a boundary condition, and is held constant for $\forall n$. Realistically this was the only way the model could be done such that it replicated the results in Bodenheimer. See caption on Figure 2 for more information.

Our physical model is equivalent to a flow being loosed at $t = 0$, the source producing a density wave with a rigid profile that propagates along $+x$ over t. The source is staggered, with a lower density region being released further afield ($\rho = 0.5$), with discontinuities in its density where these regions meet initially. The model attributes a constant velocity to all the matter in the system: the flow is incompressible (hence why we were permitted to assume $\nabla \cdot \vec{v} = 0$.) An incompressible flow should have a density profile that remains constant over time (see footnote.)

The results do not show this physical behaviour: at $t = 0$ the wave is loosed, and initially has its rigid density profile over x, discontinuities and all. As time goes by and as the wave propagates however, it gradually spreads out and disperses- the density profile smoothing with the discontinuity vanishing. The gradient of the wave over the simulation loses its unique inflected steepness (the remnant of the discontinuity.) By 3 seconds, the discontinuity remnant is virtually absent.

Physically, given that the flow is incompressible and the flow rate is constant, we would not expect the discontinuity to vanish.[5] The discontinuity does, however, vanish: the density wave smooths over time, with components of density diffusing through the simulation- this implies that some of the fluid must be compressed to cause a change in the density profile over x as time goes by (i.e. not remaining the same shape as at $t = 0$.) While the simulation does not match the physical reality of what we were modelling, it *does* match the expectations of the diffusion term that we introduced through our discretisation. [6]

4 Code

4.1 Workshop 1

```
import numpy as np
import pandas as pd
import os
np.set_printoptions(precision=9) # for that 1.000000167
```

[4] I think...

[5] or at least I wouldn't- my understanding is fairly lacking in this department but from a physical point of view I can't see why it would physically vanish if the flow is incompressible- if it is meant to physically vanish too, please elaborate :(

[6] Thanks for reading!!! Sorry if there are errors or typos... I did this all in one go and I'm fairly poofed out now...

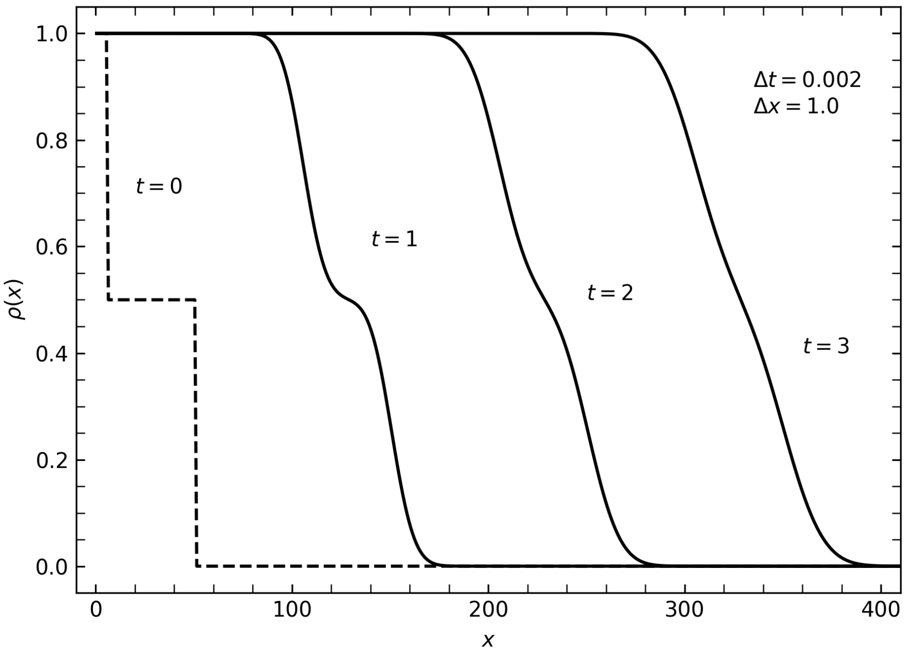

Figure 2: Solution of the continuity equation for $v = 100$ in the x-axis equivalent to Bodenheimer Figure 2.1, with grid parameters displayed. Snapshots at times of 0,1,2 and 3 are displayed. The sim was ran with a source boundary term on the leftmost edge of the grid, i.e. $\rho_{-0.5} = 1$. The first 5 elements were set to $\rho = 1$ with the successive 50 elements set to $\rho = 0.5$, the remainder set to $\rho = 0$.

I have to concede that without the source term, I wasn't able to replicate the results in the book. If possible I'd like to know how it's meant to be done correctly. The reason I used a staggered grid is because I assumed my sim wasn't working without a staggered grid (i.e. without that source term) and so I added the staggered grid to copy the book exactly- that didn't fix my problem, I added the source term after hours of lamentations, and I never removed the staggered grid.

```
# Define grid spacings and a placeholder for calculated values
spacings = [1, 1e-1, 1e-2, 1e-3, 1e-4, 1e-5, 1e-6, 1e-7, 1e-8]
spacingvalues = []

# Calculate for each spacing the three different approximations (forward, backward, centered/centred)
for spacing in spacings:
    # Set up the grid (for ease I′m using odd number of cells- centred on 0).
    # Note: Minor Rounding Error introduced.
    n_cells = 9
    n_middle = np.rint(n_cells/2 - 1e-5, out=np.zeros(1, int), casting=′unsafe′)[0] # middle element inde
    max_cell = n_cells - 1 # pythonic maximum cell index
    n_right = np.rint(n_cells/2 - 1e-5, out=np.zeros(1, int), casting=′unsafe′)[0] # number of spacings t
    grid = np.linspace(-1*n_right*spacing, +1*n_right*spacing, n_cells)
    grid = np.exp(grid) # set the values on the grid

    # Forward Method (periodic boundary conditions)
    forwrd_gradient = np.zeros(shape=np.shape(grid))
    for num in range(n_cells):
        try:
            gradient = (grid[num+1] - grid[num])/spacing
            forwrd_gradient[num] = gradient
        # Hit a boundary (the upper, since this is a forward method.)
        except:
            gradient = (grid[0] - grid[num]) / spacing
            forwrd_gradient[num] = gradient

    # Backward Method (periodic boundary conditions)
    backwd_gradient = np.zeros(shape=np.shape(grid))
    for num in range(n_cells):
        try:
            gradient = (grid[num] - grid[num-1])/spacing
            backwd_gradient[num] = gradient
        # Hit a boundary (the lower, since this is a backward method.)
        except:
            gradient = (grid[num] - grid[max_cell]) / spacing
            backwd_gradient[num] = gradient

    # Centred Method (periodic boundary conditions)
    """
    Two options for periodicity: numpy rolling, or numpy tiling.
    - roll is better for memory/etc- try/except loops would be involved though- no lengthy arrays
    - tiling is what I′ll use, since there′s no need for loops at all + faster to code
    """
    centred_gradient = np.zeros(shape=np.shape(grid))
    tiled_grid = np.tile(grid, 3)
    for num in range(n_cells):
        gradient = (tiled_grid[num+n_cells+1] - tiled_grid[num+n_cells-1])/(2*spacing)
        centred_gradient[num] = gradient

    # Select the middle element (i.e.
    forwrd_val = forwrd_gradient[n_middle]
    backwd_val = backwd_gradient[n_middle]
    centre_val = centred_gradient[n_middle]
    vals = [forwrd_val, backwd_val, centre_val]

    # Append
    spacingvalues.append(vals)

# Formatting stuff for export to a table
spacingvalues = np.array(spacingvalues)
df = pd.DataFrame(data=spacingvalues,
                  index=spacings,
                  columns=["Forward","Backward","Centred"])
dir = os.getcwd()
df.to_latex(buf=dir + "\\boden2.1.tex",
            label="Table 2.1, Bodenheimer, Repeated",
            float_format='%11.9f')

# Create and format out a table for the "distance" to the true value
distancevalues = spacingvalues
for i in range(np.shape(spacingvalues)[0]):
    for j in range(np.shape(spacingvalues)[1]):
        distancevalues[i,j] = np.abs(distancevalues[i,j] - 1)
df = pd.DataFrame(data=distancevalues,
                  index=spacings,
                  columns=["Forward","Backward","Centred"])
df.to_latex(buf=dir + "\\boden2.1_Difference.tex",
            label="Table 2.1, Bodenheimer, Repeated, Error Size",
            float_format='%11.9f')
```

4.2 Workshop 2: Part A

```
import numpy as np
import math
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
import os
from matplotlib.pyplot import *
import matplotlib.colors as colors
from matplotlib import rc, ticker, cm, patches
import pdb
import netCDF4 as netCDF
from pylab import *
from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
from mpl_toolkits.axes_grid1.inset_locator import mark_inset

# First part
"""
Equation to solve is
du/dt = -u
General solution:
u(t) = Ae^(-t)
"""
# Define left-side boundary condition and analytic solution
boundt, boundu = 0,1
```

```
A = boundu/np.exp(-1*boundt)
u = lambda t: A*np.exp(-1*t)

# Define range and step and get the analytic solution for this range
dt = 0.2
tlims = [0,2.6]
n_steps = np.rint((tlims[1]-tlims[0])/dt,out=np.zeros(1,int),casting='unsafe')[0]
tspread = np.linspace(*tlims, n_steps + 1)
analytic_solution = u(tspread)

# Set up various integrators
solutions = [[boundu] for d in range(3)] # forward, backward, centered

# All integration schemes
for step in range(n_steps):
    forward = solutions[0][step]*(1-dt)
    backwrd = solutions[1][step]/(1+dt)
    centerd = solutions[2][step]*(2-dt)/(2+dt)
    solutions[0].append(forward)
    solutions[1].append(backwrd)
    solutions[2].append(centerd)

# Get fractional error to true analytic value
solutions = [np.array(d) for d in solutions]
errors = [(solution - analytic_solution)/analytic_solution for solution in solutions]
solerrs = [solutions,errors]

# Get the graphs
fig, axs = plt.subplots(nrows=2,ncols=1, sharex='all',figsize=(7,10))
plt.subplots_adjust(wspace=0, hspace=0.05)
for ax in axs:
    ax.tick_params(axis="x", which="both", direction="in", length=4, bottom=True, left=True, right=True,
                   top=True)
    ax.tick_params(axis="y", which="both", direction="in", length=4, bottom=True, left=True, right=True,
                   top=True)

# Set up the ticker for xaxis
xticks = ticker.MultipleLocator(0.4)
xticksminor = ticker.MultipleLocator(0.2)
for ax in axs:
    ax.xaxis.set_major_locator(xticks)
    ax.xaxis.set_minor_locator(xticksminor)

# Now do the yaxis
yticks = ticker.MultipleLocator(0.2)
yticksminor = ticker.MultipleLocator(0.1)
axs[0].yaxis.set_major_locator(yticks)
axs[0].yaxis.set_minor_locator(yticksminor)
yticks1 = ticker.MultipleLocator(0.1)
yticksminor1 = ticker.MultipleLocator(0.05)
axs[1].yaxis.set_major_locator(yticks1)
axs[1].yaxis.set_minor_locator(yticksminor1)

# For the solutions
```

```
linestyles = ['dashed','dashdot','dotted']

# Plot solutions and set xlims
for axnum,ax in enumerate(axs):
    for num,linestyle in enumerate(linestyles):
        ax.plot(tspread, solerrs[axnum][num], linestyle=linestyle, color='black')
        ax.set(xlim=tlims)

# Set the ylims
axs[0].set(ylim=[0,1])
axs[1].set(ylim=[-0.2,0.2])

# Do the analytics, too
axs[0].plot(tspread, analytic_solution, color='black')
axs[1].axhline(0, color='black')

# Add some labels
axs[0].set(ylabel=r'$u(t)$')
axs[1].set(ylabel=r'$\epsilon(t)/w(t)$',
           xlabel=r'$t$')

# Create the inset
axins = zoomed_inset_axes(axs[0], 10, loc=1)  # zoom=6
for num, linestyle in enumerate(linestyles):
    axins.plot(tspread, solerrs[0][num], linestyle=linestyle, color='black')
axins.plot(tspread, analytic_solution, color='black')
axins.set(ylabel=r'$u(t)$',
          xlabel=r'$t$')
insxlocator = ticker.MultipleLocator(0.02)
axins.xaxis.set_major_locator(insxlocator)
# Region to zoom
xx,yy = [2, 2.12-1e-6],[0.1, 0.16]
axins.set_xlim(*xx)
axins.set_ylim(*yy)

# Add a rectangular patch to finish it up :)
rectangle = patches.Rectangle((xx[0],yy[0]),0.12, 0.06, edgecolor='black',facecolor='none')
axs[0].add_patch(rectangle)

plt.savefig("figure2.2bodenheimer.png", dpi=300)
plt.show()
```

4.3 Workshop 2: Part B

```
    # v
v = 100

# Steps and number of 'em
dx = 1.0
nx = 600
tlims=[0,4]
dt = 0.002
stability_dt = dx/v
```

```
nt = np.rint(max(tlims)/dt, out=np.zeros(1,int), casting='unsafe')[0]
d = v*dt/dx

# Set up domain
x_domain = np.arange(0, dx*nx, dx) # nx elements
rhox_domain = np.arange(dx/2,(dx*(nx-1))-(dx/2) + 1e-10, dx)
lenrho = nx-1
rho_domain = np.zeros(lenrho) # nx-1 elements (staggered to halves)

# Boundary rho
rho_domain[0] = 0
rho_domain[0:6] = 1
rho_domain[6:51] = 0.5

# Set up list of lists for time series
rho_over_time = [rho_domain]

# Step forward rho_over_time. Constant flow in at x=0
# (for some reason this is the only way I can get this to work.)
# (if that isn't what the guy in the book did, then I have no idea how to make my code work)
# (spent a few hours trying but... yeah... it's 6 AM and I need sleep :_:)
for step in range(1,nt):
    # Set up empty rho_over_time
    rho_step = np.zeros(lenrho)
    # Tile it: same periodic method as previously
    rho_tiled = np.tile(rho_over_time[step-1],3)
    rho_tiled[lenrho-1] = 1
    rho_tiled[2*lenrho-10:3*lenrho] = 0
    # Get all new rho elements
    for j in range(lenrho):
        rho_step[j] = -d*(rho_tiled[j] - rho_tiled[(j-1)+lenrho]) + rho_tiled[j]
    # Append
    rho_over_time.append(rho_step)

# Convert a time to a timestep
timestep = lambda t: np.rint(t/dt, out=np.zeros(1,int), casting='unsafe')[0]

# Set up graphing environent
fig, ax = plt.subplots(nrows=1,ncols=1,figsize=(7,5))
xlocators = [MultipleLocator(100),MultipleLocator(20)]
ylocators = [MultipleLocator(0.2),MultipleLocator(0.05)]
ax.xaxis.set_major_locator(xlocators[0])
ax.xaxis.set_minor_locator(xlocators[1])
ax.yaxis.set_major_locator(ylocators[0])
ax.yaxis.set_minor_locator(ylocators[1])
ax.tick_params(axis="x", which="both", direction="in", length=4, bottom=True, left=True, right=True,
               top=True)
ax.tick_params(axis="y", which="both", direction="in", length=4, bottom=True, left=True, right=True,
               top=True)
ax.set(xlim=[-10,410],
       ylim=[-0.05,1.05])

yticks = ax.yaxis.get_major_ticks()
yticks[-1].set_visible(False)
```

```
ax.plot(rhox_domain, rho_over_time[0],linestyle='dashed', color='black')
ax.text(20, 0.7, r'$t=0$')
ax.plot(rhox_domain, rho_over_time[timestep(1)], color='black')
ax.text(140, 0.6, r'$t=1$')
ax.plot(rhox_domain, rho_over_time[timestep(2)], color='black')
ax.text(250, 0.5, r'$t=2$')
ax.plot(rhox_domain, rho_over_time[timestep(3)], color='black')
ax.text(360, 0.4, r'$t=3$')
ax.margins(x=0.05, y=0.1)
ax.set(ylabel=r'$\rho(x)$',
       xlabel=r'$x$')
ax.text(335, 0.9, r'$\Delta{t} = 0.002$')
ax.text(335,0.85,r'$\Delta{x} = 1.0$')
plt.savefig("secondplot.png",dpi=300)
plt.show(dpi=300)
```