Comp Lab Book

Sam Coverly (sdrc500)

March 2019

1 Landau Damping

Wherever ':/' is referred to in a file path, it refers to the root directory 'Home/Documents/Comp_Lab'.

Original version of code downloaded from 'Fusion Laboratory MSc and Fusion CDT / Computational Lab / Lab 1' VLE page, saved in :/

Amplitude of first harmonic should damp over time. In realistic scenario, Landau damping will continue to damp the system until a constant amplitude is reached. In a perfect simulation, the amplitude should also damp exponentially forever. When the change in amplitude becomes constant, it is due to the noise level of the simulation.

Using initial values ($L = 4\pi, ncells = 20, npart = 1000, t = 20s, steps = 50$), the values originally given in the code. Output saved as 'Amplitude_1.png' in :/

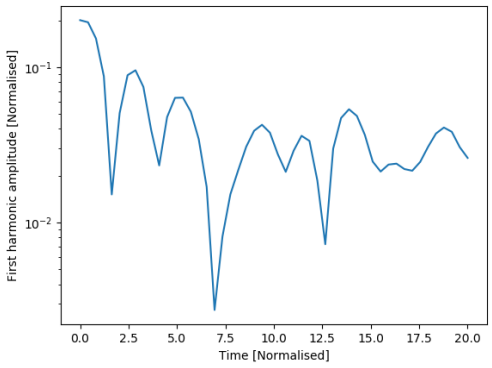


Figure 1: Amplitude of first harmonic over time with original initial values

Amplitude definitely seems to be decreasing over time. Consider running for longer time with same step size.

using t=100s,steps= 250:

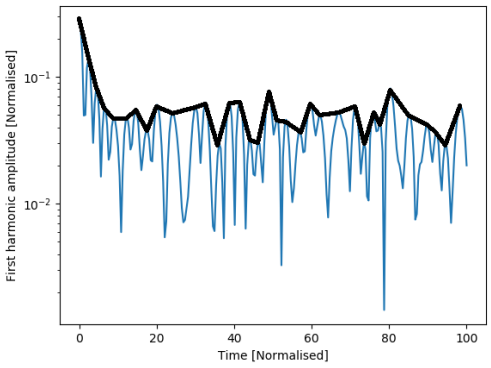


Figure 2: Amplitude of first harmonic over longer time, with line drawn between peaks to distinguish damping rate

Noise level seems to be reached after ∼10 seconds.

Need to find location of each peak and compare to determine rate at which amplitude is decreasing. Can find peak by taking the derivative (gradient) at each point and multiplying by each neighbour. Only at points where the product is negative has the gradient of the line shifted from positive to negative, or visa versa. However, need to find some way to eliminate troughs from this analysis.

Since each peak must necessarily be followed by a trough, can place every odd numbered value in one array, with even numbered values in another array. Array with consistently larger values is therefore the one that contains peaks. Therefore, after finding all peaks and troughs, split array into odd and even elements.

Code Modification: Currently a part of 'main', will move to its own function after bugtesting. Compares value 'i' of first harmonic against value 'i+1' to find the gradient between those two points. If this gradient multiplied by the 'i-1' gradient is negative, appends value 'i' of first harmonic to the list of peak values

```
gradient = [] #stores gradient of first harmonic
    gradient.append(1)
```

```
turn_t = [] #stores time values of turning points
turn_amp = [] #stores first harmonic amplitudes of turning points
for i in range(1,len(s.firstharmonic)-1,1): #for each first harmonic value
    gradient.append((s.firstharmonic[i+1]-s.firstharmonic[i])) #find gradient
    if (gradient[i]*gradient[i-1] < 0):
    #if gradient changes sign compared to previous gradient
        turn_t.append(s.t[i]) #append values to turning point arrays
        turn_amp.append(s.firstharmonic[i])
```

Version saved as 'epc1d_grad.py' in :/

Using t=20s, steps = 50: output saved as 'Peak_test.png' in :/

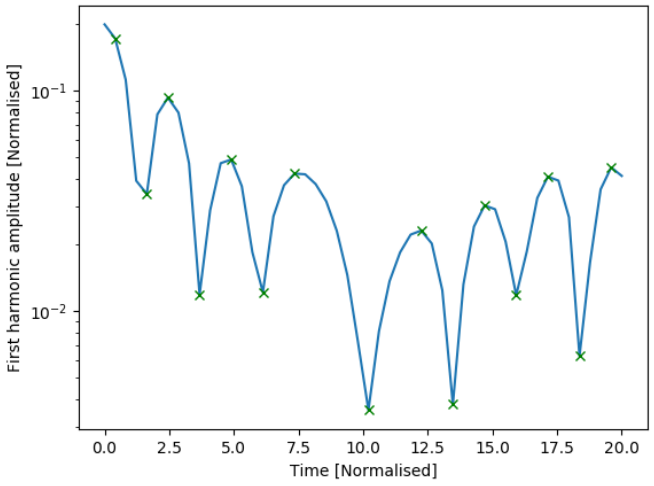


Figure 3: Amplitude of first harmonic over time, with peaks and troughs indicated

Code Modification: Simple 'if' condition to remove troughs

```
if (turn_amp[0]>turn_amp[1]):
        del turn_amp[1::2]
        del turn_t[1::2]
    else:
        del turn_amp[0::2]
        del turn_t[0::2]
```

Output saved as 'Peak_test2.png' in :/

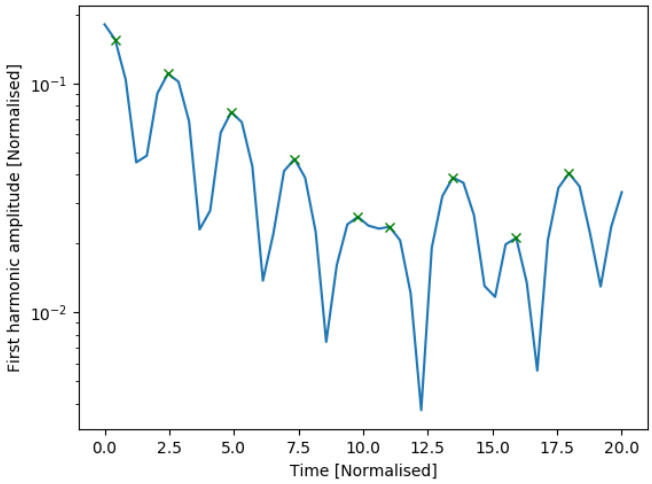


Figure 4: Amplitude of first harmonic over time, with only peaks indicated

Now have arrays storing amplitudes and times of all peaks. Since system should always be damped, noise level begins with first peak that has a larger amplitude than its predecessor. Can find estimate of noise level by averaging over all peak amplitudes that come after this point. Furthermore, can find damping rate by fitting a gradient to the amplitudes of all peaks before this point.

Code Modification: Using scipy.optimize.curvefit to fit a straight line to all peaks before the noise becomes dominant.

```
def linefit(x,*params):
    m = params[0] #gradient
    y0 = params[1] #calculated background noise level,
    return A*(-m*x)-y0

guess = [0.001,noise_avg]
    popt, pcov = curve_fit(linefit, damp_t, log(damp_amp), p0=guess,maxfev = 10000000)
    yfit = linefit(array(damp_t),*popt)
```

Now have code that can calculate the frequency, damping rate and noise level, along with appropriate errors, as well as the time taken. By retrieving these calculated values for varying starting parameters(box length, particle

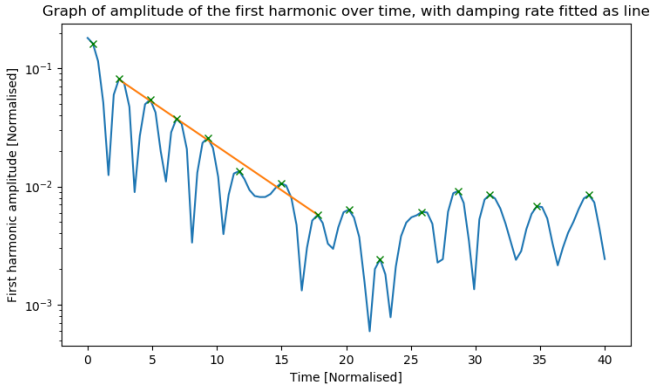


Figure 5: Graph of amplitude of first harmonic over time, with damping rate fitted as a line

number, cell number) it will be possible to determine how these parameters affect the damping of the system.

Since each execution can return vastly different results, should average over multiple executions of program, removing any possible outliers. Furthermore, to reduce the time taken for calculation, the code should be optimized before proceeding.

1.1 Optimization

Code Modification: The first step of optimization should be to remove the visual phase space display, as well as to stop the time and first harmonic being printed to the terminal after every timestep. In order to reduce uncertainty in the time comparisons, should also modify code to allow for a constant seed to be used.

```
# Added to start of Landau function:

random.seed()
# In main function:

pos, vel = run(pos, vel, L, ncells,
                  out=[s],                        # These are called each output
                  output_times=linspace(0.,40,100)) # The times to output

# In run function:
#print "Time: ", time, dt
```

Using L = 4π,ncells=20,npart=1000,t=40,step=100:

	Time taken (s)	
Seed	Original	Optimized
1	13.95569	2.92984
2	14.18689	2.97359
3	13.82384	2.96635

Average speedup = 4.73

Optimized version now runs 4.73* faster than the original
Version saved as 'epc1d_optimized.py' in ': /'

Profiling code using 'python -m profile' linux command reveals that almost 95% of operation time is spent on the 'calc_density' function.

The calc_density function is inefficient since it loops over each element in an array to generate values, while it could instead simply generate a new array by performing the calculations on the old array as a whole. I attempted to modify this code to remove as much of the calculation from the loop as possible.

```
density = zeros([ncells])
    nparticles = len(position)
    position = array(position) #convert position into array
    dx = L/ncells
    p = position/dx #create p array
    plower = position.astype(int) #creates array where values are rounded down to left cell
    offset = p-plower #array of offsets from left
    offsetmin = 1-offset
    pupper = (plower+1)%ncells

    for i in range(ncells): #distributes particles amongst ncells
        density[i]+=sum(offsetmin[plower==i])
        density[i]+=sum(offset[pupper==i])
    density *= float(ncells)/float(nparticles) #ensures that average density = 1

    return density
```

While this new code successfully generates values almost twice as quickly as the original, the following error occurs when attempting to fit a line to the damping rate:

```
Traceback (most recent call last):
  File "epc1d_Optimized2.py", line 371, in <module>
    popt, pcov = curve_fit(linefit, damp_t, log(damp_amp), p0=guess,maxfev = 10000000)
  File "/usr/lib/python2.7/dist-packages/scipy/optimize/minpack.py", line 736, in curve_fit
    res = leastsq(func, p0, Dfun=jac, full_output=1, **kwargs)
  File "/usr/lib/python2.7/dist-packages/scipy/optimize/minpack.py", line 380, in leastsq
    raise TypeError('Improper input: N=%s must not exceed M=%s' % (n, m))
TypeError: Improper input: N=2 must not exceed M=0
```

Figure 6: Error Message

Since I was unable to successfully resolve this error, this optimization was discarded.

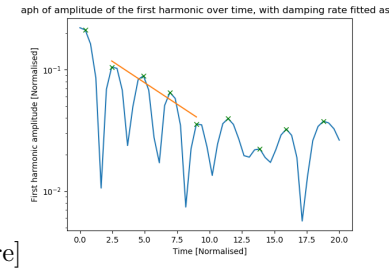
In function 'pic', there is a line which ensures that the particle positions are always within the length of the box:

```
# Ensure that pos is between 0 and L
    pos = ((pos % L) + L) % L
```

Since adding L to a value and then taking the modulo of L will return the original value, this line can be optimized by simply removing the '+L%L' term:

```
# Ensure that pos is between 0 and L
    pos = pos % L
```

To check the validity of this modification, a simple test case was run before and afterwards using the same seed.
Using seed = 1, L = $4 * \pi$,c = 20, n = 2000:

[Before] 

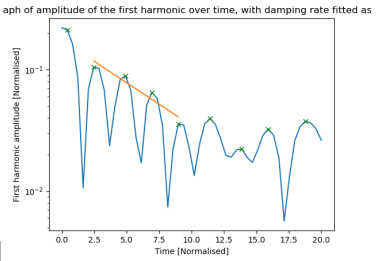
[After] 

Figure 7: Before and after modification comparison

Initial Time: 2.8669s Improved Time: 2.71967s

While there is only a small improvement to the computation time, there is also no noticeable change to the output data as a result of this modification, as seen in figure 7

1.2 Data Analysis

In order to efficiently iterate over multiple values for box length, number of particles and number of cells, as well as including multiple tests and results for each value, a python script was written to automate the data collection process, which will be included in the appendix.

All outputted data files and graphs saved in ' :/' using naming convention 'xy.dat' or 'xy.png' where x = [length,parts,cells] and y = [freq,noise,damp,time],

depending on which parameters are being changed and which values are being output.

1.2.1 Length of Box

Frequency

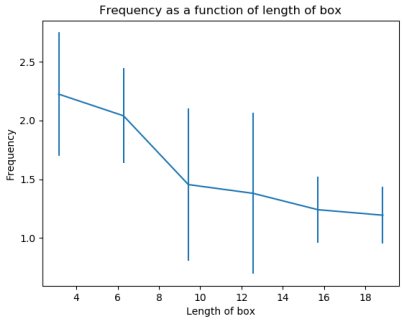


Figure 8: Graph of frequency as a function of length of box

Although the errors in calculated frequencies are large enough to cast some doubt on the accuracy of this data, there definitely seems to be a linear decrease in frequency as a function of the length of the box seen in figure 8. Furthermore, the calculated frequency at $L = 4\pi$ closely matches the analytical result of $\sigma = 1.41566$, though the error is still very large, indicating that the program is calculating the frequency properly.

As the length of the box increases, we would also expect the length of the first harmonic to increase, assuming that it represents a single complete sinusoidal oscillation of particles along the length of the box. As this distance increases, the period (time for particles to complete one oscillation) would also increase, meaning that a decrease in frequency should be expected. However, I would expect the frequency to halve every time the box length doubles, which certainly does not seem to be the case.

Noise Amplitude

Based on figure 9, the amplitude of the noise seems to follow an x^2 relation, with a trough at $L \approx 3 - 4\pi$. However, since the errorbars are very large and there is no decent physical explanation for this behaviour, it is safe to assume that this is merely due to the large uncertainty.

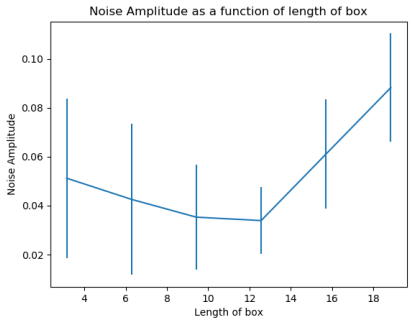


Figure 9: Graph of noise Amplitude as a function of length of box

Damping Rate

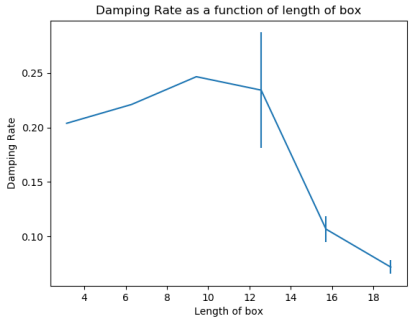


Figure 10: Graph of Damping Rate as a function of length of box

Curiously, for $L = 1, 2, 3\pi$ the damping uncertainty returned by the 'scipy.optimize.curvefit' function was always 0, for reasons unknown. Furthermore, the damping rate increases gradually up until this point before steeply declining in figure 10. Assuming that this is due to some anomaly in the code, it seems reasonable to rerun this test starting at $L = 4\pi$.

From figure 11, we see that the damping rate does indeed decrease with the length of the box. This makes sense since as the length increases, particles become more spread out and so are less likely to inelastically interact with each other, thus reducing the main source of damping seen in the system.

Curiously, 'scipy.optimize.curvefit' once again returned zero uncertainties, but this time only for $L8\pi$. Again, the reason for this is unknown.

Computation Time

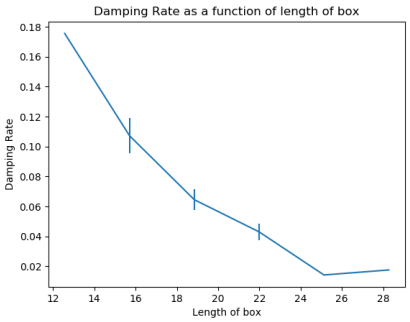


Figure 11: Graph of Damping Rate as a function of length of box, with larger values of L

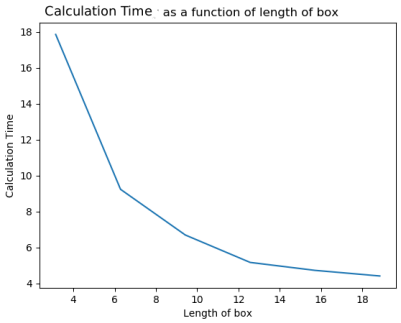


Figure 12: Graph of computation time as a function of length of box

Curiously enough, the computation time seems to decrease asymptotically as a function of the length of the box, as seen in figure 12. This may be due to the 'calc_density' function, which determines a uniform spacing 'dx' dependent on the length and the number of cells, which is then used to assign each position to a cell where 'p = position / dx '. As 'L' increases, so does 'dx', meaning that the already inefficient for loop iterating over all 'p' values increases its number of iterations. When L becomes larger, the number of iterations is decreased. If the optimization of the 'calc_density' function had worked correctly, this trend may not have occurred.

1.2.2 Number of Particles

Frequency

Although there is once again an incredibly large uncertainty in the frequencies, there seems to be no relationship between frequency and number of particles

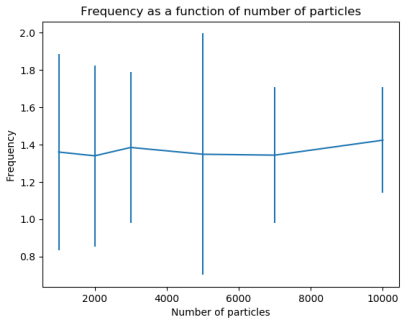


Figure 13: Graph of frequency as a function of number of particles

seen in figure 13. Assuming that the length of the first harmonic (the length over which particles oscillate sinusoidally) does not increase with number of particles, we should not expect the frequency to change.

Noise Amplitude

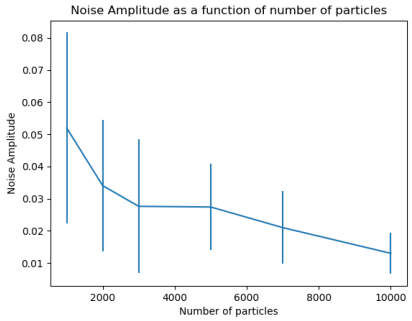


Figure 14: Graph of noise Amplitude as a function of number of particles

We see in figure 14 that the amplitude of the background noise decreases as the number of particles increases, although the errorbars are to large to determine whether this relationship is linear or slightly curved. As the number of particles increases, the number of calculations and thus the signal strength will also increase, improving the signal:noise ratio.

Damping Rate

From figure 15, there is no discernible relationship between the number of particles and the damping rate. I would expect the damping rate to increase with the number of particles, since the likelihood of particles inelastically in-

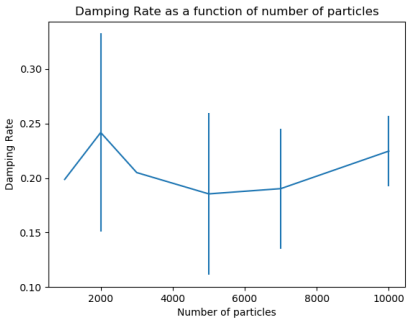


Figure 15: Graph of Damping Rate as a function of number of particles

teracting would be increased. While there seems to be a slight upward trend towards the end of the graph, the errors are once again far too large to confirm this relation.

Computation Time

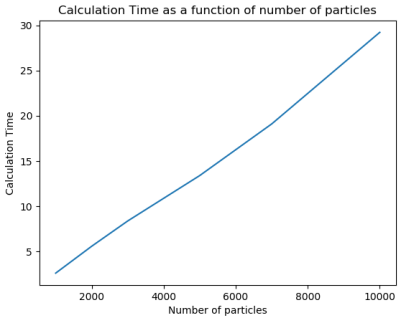


Figure 16: Graph of Computation Time as a function of number of particles

From figure 16, we see that there is an almost perfectly linear relationship between the number of particles and the calculation time. This is to be expected, since the larger the number of particles, the more calculations that must be completed.

1.2.3 Number of Cells

Frequency

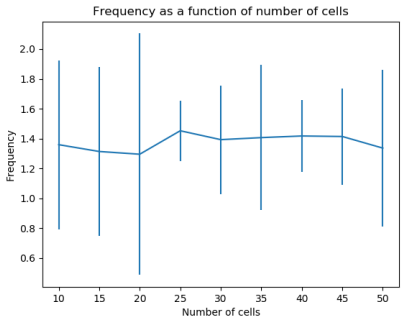


Figure 17: Graph of Frequency as a function of number of cells

Once again, we see from figure 17 no discernable relationship between the frequency and the number of cells.Instead, the frequency stays constant at approximately the analytical result, $\sigma = 1.41566$. The reason for this lack of relationship is the same as the one given in the number of particles analysis.

Noise Amplitude

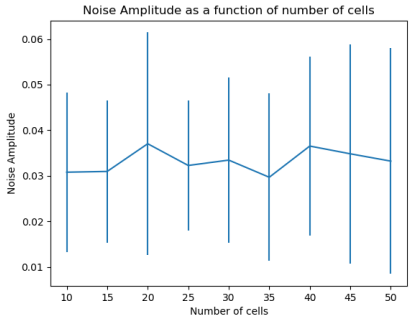


Figure 18: Graph of Noise Amplitude as a function of number of cells

From figure 18 we can see no discernable relationship between the number of cells and the noise amplitude. While the noise amplitude should decrease with an increase in particle number, the number of cells does not necessarily affect the distribution of particles, just the precision of their position. If the uncertainty in particle position was the leading cause of uncertainty, then it would be affected by the number of cells. However, this uncertainty is negligible compared to the noise level generated.

Damping Rate

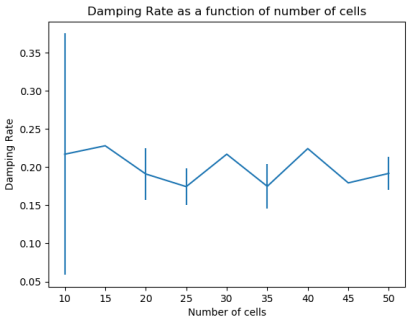


Figure 19: Graph of Damping Rate as a function of number of cells

From figure 19 we see no discernable relationship between the number of cells and the damping rate. While the damping rate is predominantly determined by the number of particles (since more particles leads to more inelastic collisions), the number of cells does not affect the number, only the precision of their positions.

Computation Time

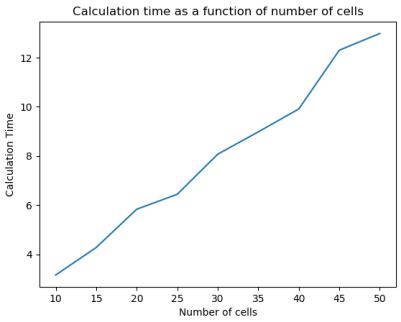


Figure 20: Graph of Computation time as a function of number of cells

From figure 20 we see a linear relationship between the number of cells and the computation time. In the 'calc_density' function, the value '$dx = L/ncells$' meaning that as 'ncells' increases, 'dx' decreases. The inefficient for loop that runs in this function iterates over each p in 'p = position / dx', meaning that as 'dx' decreases, the number of positions that this loop iterates over will increase, thus massively affecting the computation time. If the modification to this function had been successful, the effect from the number of cells may well have been minimized.

1.3 Further Analysis

While many of the relations obtained in the data analysis section were clear and consistent with the expected results, a large number of them presented seemingly erroneous data, with errorbars that were too large to draw any meaningful conclusions. One way of reducing the size of these errors may be to decrease the time step between iterations. While I literally don't have time to do this for the number of particles, which already takes a remarkably long time to compute with larger values, it can be done very quickly with the box length or the number of cells.

In an attempt to increase the accuracy of results, the number of steps between t=0 and t=40 was increased from 100 to 300 and the box length analysis was rerun.

1.3.1 Results

Frequency

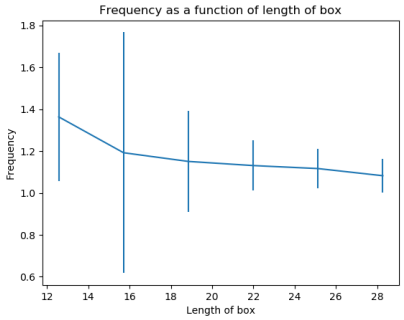


Figure 21: Graph of Frequency as a function of length of box

Strangely, there the uncertainty in the frequency seems to decrease proportionally with the length of the box, but there is no discernable difference due to changing the timestep. Since the locations of the peaks are determined based on the change in gradient between neighbouring datapoints, I had expected that a decreased step size would proportionally increase the precision in the locations, and thus the uncertainty in the time between peaks should also decrease.

While the errorbars towards the end of the graph are smaller, the first three are just as large as they had been in the initial computation, indicating that there may actually be a larger source of frequency uncertainty directly as a result of box length.

Noise amplitude

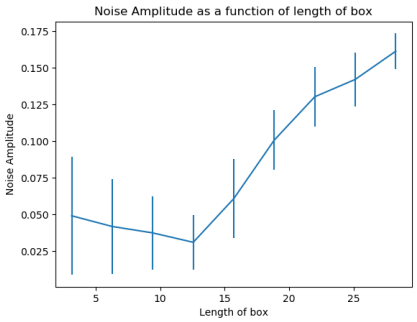


Figure 22: Graph of Noise Amplitude as a function of length of box

Just like in the initial computation, the noise amplitude seems to decrease slightly for lengths $L \leq 4\pi$, and then gradually increase beyond that point. While I had initially considered this to be an erroneous result, it now seems as though there is an optimum box length at which the noise amplitude is minimized, at $L = 4\pi$.

Damping Rate

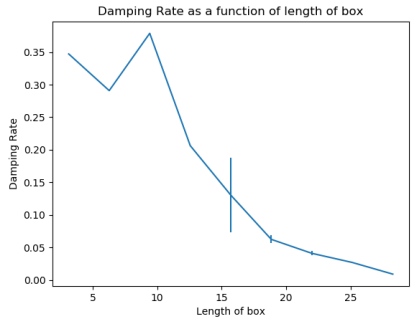


Figure 23: Graph of Damping Rate as a function of length of box

Just like in the original analysis, most of the errors returned for the damping rate are 0 except at the middle values of box length. While this anomaly is undoubtedly affected by the box length, there may be a more consistent method of calculating the uncertainty in the damping rate than by using the covariant returned by 'scipy.optimize.curvefit'.

2 Appendix

2.1 epc1d.py

```
from numpy import arange, concatenate, zeros, linspace, floor, array, pi
from numpy import sin, cos, sqrt, random, histogram, log, exp
from scipy.optimize import curve_fit
import time

import matplotlib.pyplot as plt # Matplotlib plotting library

try:
    import matplotlib.gridspec as gridspec  # For plot layout grid
    got_gridspec = True
except:
    got_gridspec = False

# Need an FFT routine, either from SciPy or NumPy
try:
    from scipy.fftpack import fft, ifft
except:
    # No SciPy FFT routine. Import NumPy routine instead
    from numpy.fft import fft, ifft

def linefit(x,*params):
    m = params[0] #gradient
    y0 = params[1] #background noise level
    return (-m*x)-y0 #return line equation

def rk4step(f, y0, dt, args=()):
    """ Takes a single step using RK4 method """
    k1 = f(y0, *args)
    k2 = f(y0 + 0.5*dt*k1, *args)
    k3 = f(y0 + 0.5*dt*k2, *args)
    k4 = f(y0 + dt*k3, *args)

    return y0 + (k1 + 2.*k2 + 2.*k3 + k4)*dt / 6.

def calc_density(position, ncells, L):
    """ Calculate charge density given particle positions

    Input
      position  - Array of positions, one for each particle
                  assumed to be between 0 and L
      ncells    - Number of cells
      L         - Length of the domain
```

```
    Output
      density   - contains 1 if evenly distributed
    """
    # This is a crude method and could be made more efficient

    density = zeros([ncells])
    nparticles = len(position)
    '''
    position = array(position) #convert position into array
    dx = L/ncells
    p = position/dx #create p array
    plower = position.astype(int) #creates array where values are rounded down to left cell
    offset = p-plower #array of offsets from left
    offsetmin = 1-offset
    pupper = (plower+1)%ncells

    for i in range(ncells): #distributes particles amongst ncells
        density[i]+=sum(offsetmin[plower==i])
        density[i]+=sum(offset[pupper==i])
    density *= float(ncells)/float(nparticles) #ensures that average density = 1

    return density
'''
    dx = L / ncells       # Uniform cell spacing
    for p in position / dx:    # Loop over all the particles, converting position into a cel
        plower = int(p)        # Cell to the left (rounding down)
        offset = p - plower    # Offset from the left
        density[plower] += 1. - offset
        density[(plower + 1) % ncells] += offset
    # nparticles now distributed amongst ncells
    density *= float(ncells) / float(nparticles)  # Make average density equal to 1
    return density

def periodic_interp(y, x):
    """
    Linear interpolation of a periodic array y at index x

    Input

    y - Array of values to be interpolated
    x - Index where result required. Can be an array of values

    Output

    y[x] with non-integer x
```

```
    """
    ny = len(y)
    if len(x) > 1:
        y = array(y) # Make sure it's a NumPy array for array indexing
    xl = floor(x).astype(int) # Left index
    dx = x - xl
    xl = ((xl % ny) + ny) % ny  # Ensures between 0 and ny-1 inclusive
    return y[xl]*(1. - dx) + y[(xl+1)%ny]*dx

def fft_integrate(y):
    """ Integrate a periodic function using FFTs
    """
    n = len(y) # Get the length of y

    f = fft(y) # Take FFT
    # Result is in standard layout with positive perioduencies first then negative
    # n even: [ f(0), f(1), ... f(n/2), f(1-n/2) ... f(-1) ]
    # n odd:  [ f(0), f(1), ... f((n-1)/2), f(-(n-1)/2) ... f(-1) ]

    if n % 2 == 0: # If an even number of points
        k = concatenate( (arange(0, n/2+1), arange(1-n/2, 0)) )
    else:
        k = concatenate( (arange(0, (n-1)/2+1), arange( -(n-1)/2, 0)) )
    k = 2.*pi*k/n

    # Modify perioduencies by dividing by ik
    f[1:] /= (1j * k[1:])
    f[0] = 0. # Set the arbitrary zero-perioduency term to zero

    return ifft(f).real # Reverse Fourier Transform


def pic(f, ncells, L):
    """ f contains the position and velocity of all particles
    """
    nparticles = len(f)/2     # Two values for each particle
    pos = f[0:nparticles] # Position of each particle
    vel = f[nparticles:]      # Velocity of each particle

    dx = L / float(ncells)    # Cell spacing

    # Ensure that pos is between 0 and L
    pos = (pos % L)

    # Calculate number density, normalised so 1 when uniform
    density = calc_density(pos, ncells, L)
```

```
    # Subtract ion density to get total charge density
    rho = density - 1.

    # Calculate electric field
    E = -fft_integrate(rho)*dx

    # Interpolate E field at particle locations
    accel = -periodic_interp(E, pos/dx)

    # Put back into a single array
    return concatenate( (vel, accel) )

#########################################################################

def run(pos, vel, L, ncells=None, out=[], output_times=linspace(0,20,100), cfl=0.5):

    if ncells == None:
        ncells = int(sqrt(len(pos))) # A sensible default

    dx = L / float(ncells)

    f = concatenate( (pos, vel) )   # Starting state
    nparticles = len(pos)

    time = 0.0
    for tnext in output_times:
        # Advance to tnext
        stepping = True
        while stepping:
            # Maximum distance a particle can move is one cell
            dt = cfl * dx / max(abs(vel))
            if time + dt >= tnext:
                # Next time will hit or exceed required output time
                stepping = False
                dt = tnext - time
            #print "Time: ", time, dt
            f = rk4step(pic, f, dt, args=(ncells, L))
            time += dt

        # Extract position and velocities
        pos = ((f[0:nparticles] % L) + L) % L
        vel = f[nparticles:]

        # Send to output functions
        for func in out:
```

```
            func(pos, vel, ncells, L, time)

    return pos, vel

##########################################################################
#
# Output functions and classes
#

class Plot:
    """
    Displays three plots: phase space, charge density, and velocity distribution
    """
    def __init__(self, pos, vel, ncells, L):

        d = calc_density(pos, ncells, L)
        vhist, bins  = histogram(vel, int(sqrt(len(vel))))
        vbins = 0.5*(bins[1:]+bins[:-1])

        # Plot initial positions
        if got_gridspec:
            self.fig = plt.figure()
            self.gs = gridspec.GridSpec(4, 4)
            ax = self.fig.add_subplot(self.gs[0:3,0:3])
            self.phase_plot = ax.plot(pos, vel, '.')[0]
            ax.set_title("Phase space")

            ax = self.fig.add_subplot(self.gs[3,0:3])
            self.density_plot = ax.plot(linspace(0, L, ncells), d)[0]

            ax = self.fig.add_subplot(self.gs[0:3,3])
            self.vel_plot = ax.plot(vhist, vbins)[0]
        else:
            self.fig = plt.figure()
            self.phase_plot = plt.plot(pos, vel, '.')[0]

            self.fig = plt.figure()
            self.density_plot = plt.plot(linspace(0, L, ncells), d)[0]

            self.fig = plt.figure()
            self.vel_plot = plt.plot(vhist, vbins)[0]
        plt.ion()
        plt.show()

    def __call__(self, pos, vel, ncells, L, t):
        d = calc_density(pos, ncells, L)
```

```
        vhist, bins  = histogram(vel, int(sqrt(len(vel))))
        vbins = 0.5*(bins[1:]+bins[:-1])

        self.phase_plot.set_data(pos, vel) # Update the plot
        self.density_plot.set_data(linspace(0, L, ncells), d)
        self.vel_plot.set_data(vhist, vbins)
        plt.draw()
        plt.pause(0.05)


class Summary:
    def __init__(self):
        self.t = []
        self.firstharmonic = []

    def __call__(self, pos, vel, ncells, L, t):
        # Calculate the charge density
        d = calc_density(pos, ncells, L)

        # Amplitude of the first harmonic
        fh = 2.*abs(fft(d)[1]) / float(ncells)

        #print "Time:", t, "First:", fh

        self.t.append(t)
        self.firstharmonic.append(fh)

######################################################################
#
# Functions to create the initial conditions
#

def landau(npart, L, alpha=0.2):
    """
    Creates the initial conditions for Landau damping

    """
    # Start with a uniform distribution of positions
    random.seed(1)
    pos = random.uniform(0., L, npart)
    pos0 = pos.copy()
    k = 2.*pi / L
    for i in range(10): # Adjust distribution using Newton iterations
        pos -= ( pos + alpha*sin(k*pos)/k - pos0 ) / ( 1. + alpha*cos(k*pos) )

    # Normal velocity distribution
```

```
    vel = random.normal(0.0, 1.0, npart)

    return pos, vel

def twostream(npart, L, vbeam=2):
    # Start with a uniform distribution of positions
    pos = random.uniform(0., L, npart)
    # Normal velocity distribution
    vel = random.normal(0.0, 1.0, npart)

    np2 = int(npart / 2)
    vel[:np2] += vbeam  # Half the particles moving one way
    vel[np2:] -= vbeam  # and half the other

    return pos,vel

######################################################################

if __name__ == "__main__":
    # Generate initial condition
    #
    if False:
        # 2-stream instability
        L = 100
        ncells = 20
        pos, vel = twostream(10000, L, 3.)
    else:
        # Landau damping
        L = 4.*pi
        ncells = 20
        npart = 2000
        pos, vel = landau(npart, L)
    t0 = time.time()
    # Create some output classes
    #p = Plot(pos, vel, ncells, L) # This displays an animated figure
    s = Summary()                 # Calculates, stores and prints summary info

    # Run the simulation
    pos, vel = run(pos, vel, L, ncells,
                   out=[s],                       # These are called each output
                   output_times=linspace(0.,20,50)) # The times to output

    # Summary stores an array of the first-harmonic amplitude
    # Make a semilog plot to see exponential damping
    gradient = [] #stores gradient of first harmonic
    gradient.append(1)
```

```
turn_t = []
turn_amp = []

for i in range(1,len(s.firstharmonic)-1,1): #for every value of first harmonic excluding
    gradient.append((s.firstharmonic[i+1]-s.firstharmonic[i])) #calculate gradient betwe
    if (gradient[i]*gradient[i-1] < 0): #if gradient changes sign
        turn_t.append(s.t[i]) #add peak time to list
        turn_amp.append(s.firstharmonic[i]) #add peak amplitude to list

if (turn_amp[0]>turn_amp[1]): #if first peak amp is larger than next
    del turn_amp[1::2] #delete troughs from lists
    del turn_t[1::2]
else:
    del turn_amp[0::2] #delete troughs from lists
    del turn_t[0::2]

period = []
period_max =0
period_avg=0
period_min = 100
damp_amp = []
damp_t = []
noise_amp = []
noise_t = []

for i in range(1,len(turn_amp),1):
    period.append(turn_t[i]-turn_t[i-1]) #calculates time between peaks
    if turn_amp[i]<turn_amp[i-1]:
        damp_amp.append(turn_amp[i]) #if peaks are decreasing in amplitude
        damp_t.append(turn_t[i])
    else: #once next peak is larger amp than previous, noise has started
        noisetime = turn_t[i] #add remaining peaks to noise list
        for j in range(i,len(turn_amp),1):
            noise_amp.append(turn_amp[j])
            noise_t.append(turn_t[j])
        break

period_max = max(period)
period_min = min(period)
period_avg = 2*sum(period)/(len(period)) #calculate average period value
period_error = (period_max-period_min)/2 #calculate average period error
freq = 2*pi*(1/period_avg) #calculate frequency from period

noise_max = max(noise_amp)
noise_min = min(noise_amp)
noise_avg = sum(noise_amp)/len(noise_amp) #calculate average noise amplitude
```

```
    noise_error = (noise_max-noise_min)/2 #calculate noise error

    guess = [0.001,noise_avg]
    popt, pcov = curve_fit(linefit, damp_t, log(damp_amp), p0=guess,maxfev = 10000000) #atte
    yfit = linefit(array(damp_t),*popt)

    plt.figure()
    plt.plot(s.t, s.firstharmonic)
    plt.plot(damp_t,exp(yfit))
    plt.plot(turn_t,turn_amp, 'gx')
    plt.xlabel("Time [Normalised]")
    plt.ylabel("First harmonic amplitude [Normalised]")
    plt.yscale('log')
    plt.title("Graph of amplitude of the first harmonic over time, with damping rate fitted
    print "Damping rate = ",popt[0], "+- ",pcov[1][1]
    print "Frequency = ",freq,"+- ",period_error," , noisetime = ",noisetime
    print "Average noise level = ",noise_avg, "+- ",noise_error
    t1= time.time()
    ttot = t1-t0
    print "Time taken = ", ttot
    #plt.ioff() # This so that the windows stay open
    plt.show()


def auto(npart,L,ncells): #Version of main that can be used as a function in automated data
    s = Summary()                  # Calculates, stores and prints summary info
    pos, vel = landau(npart, L)
    # Run the simulation
    pos, vel = run(pos, vel, L, ncells,
                   out=[s],                        # These are called each output
                   output_times=linspace(0.,40,300)) # The times to output

    # Summary stores an array of the first-harmonic amplitude
    # Make a semilog plot to see exponential damping
    gradient = [] #stores gradient of first harmonic
    gradient.append(1)
    turn_t = []
    turn_amp = []

    for i in range(1,len(s.firstharmonic)-1,1): #for every value of first harmonic excludin
        gradient.append((s.firstharmonic[i+1]-s.firstharmonic[i])) #calculate gradient betwe
        if (gradient[i]*gradient[i-1] < 0): #if gradient changes sign
            turn_t.append(s.t[i]) #add peak time to list
            turn_amp.append(s.firstharmonic[i]) #add peak amplitude to list

    if (turn_amp[0]>turn_amp[1]): #if first peak amp is larger than next
```

```
    del turn_amp[1::2] #delete troughs from lists
    del turn_t[1::2]
else:
    del turn_amp[0::2] #delete troughs from lists
    del turn_t[0::2]

period = []
period_max =0
period_avg=0
period_min = 100
damp_amp = []
damp_t = []
noise_amp = []
noise_t = []

for i in range(1,len(turn_amp),1):
    period.append(turn_t[i]-turn_t[i-1]) #calculates time between peaks
    if turn_amp[i]<turn_amp[i-1]:
        damp_amp.append(turn_amp[i]) #if peaks are decreasing in amplitude
        damp_t.append(turn_t[i])
    else: #once next peak is larger amp than previous, noise has started
        noisetime = turn_t[i] #add remaining peaks to noise list
        for j in range(i,len(turn_amp),1):
            noise_amp.append(turn_amp[j])
            noise_t.append(turn_t[j])
        break

period_max = max(period)
period_min = min(period)
period_avg = 2*sum(period)/(len(period)) #calculate average period value
period_error = (period_max-period_min)/2 #calculate average period error
freq = 2*pi*(1/period_avg) #calculate frequency from period

noise_max = max(noise_amp)
noise_min = min(noise_amp)
noise_avg = sum(noise_amp)/len(noise_amp) #calculate average noise amplitude
noise_error = (noise_max-noise_min)/2 #calculate noise error

guess = [0.001,noise_avg]
popt, pcov = curve_fit(linefit, damp_t, log(damp_amp), p0=guess,maxfev = 10000000) #att
yfit = linefit(array(damp_t),*popt)

return freq,period_error,noise_avg,noise_error,popt[0],pcov[1][1] #return values for us
```

2.1.1 DataCollection.py

```
import epc1d
import numpy as np
import time
import matplotlib.pyplot as plt

freq = []
freq_avg = []
freqerror = []
freqerror_avg = []
noise = []
noise_avg = []
noiserror = []
noiserror_avg = []
damp = []
damp_avg = []
damperror = []
damperror_avg = []
timer = []
timer_avg = []

def analysis(P,L,C,filename,n):
    i = 0
    while i < 5: #iterates five times with different seeds
        try: #if an error occurs, discard and start this iteration over
            start = time.time()
            data = epc1d.auto(P,L,C) #runs epc1d func with P,L,C passed in
            stop = time.time()
            if np.isinf(data[5]): #if damping error returned = infinity, raise exception
                assert "damping error"
            freq.append(data[0]) #retrieve useful values from data
            freqerror.append(data[1])
            noise.append(data[2])
            noiserror.append(data[3])
            damp.append(data[4])
            damperror.append(data[5])
            timer.append(stop-start)
            print data
            i=i+1
        except:
            print "an error has occured" #print when an error occurs
    freq_avg.append(sum(freq)/len(freq)) #calculate average values over all iterations
    freqerror_avg.append(sum(freqerror)/len(freqerror))
    f = open("%sfreq.txt" % (filename),"w+") #write average values to file
    f.write("%f %f %f" %(l[n],freq_avg[n],freqerror_avg[n])) #writes varying parameter, ave
```

```
    f.close()
    noise_avg.append(sum(noise)/len(noise))
    noiserror_avg.append(sum(noiserror)/len(noiserror))
    f = open("%snoise.txt" % (filename),"w+")
    f.write("%f %f %f" %(l[n],noise_avg[n],noiserror_avg[n]))
    f.close()
    damp_avg.append(sum(damp)/len(damp))
    damperror_avg.append(sum(damperror)/len(damperror))
    f = open("%sfreq.txt" % (filename),"w+")
    f.write("%f %f %f" %(l[n],damp_avg[n],damperror_avg[n]))
    f.close()
    timer_avg.append(sum(timer)/len(timer))
    f = open("%sfreq.txt" % (filename),"w+")
    f.write("%f %f" %(l[n],timer_avg[n]))
    f.close()
    del freq[:] #clear arrays
    del freqerror[:]
    del noise[:]
    del noiserror[:]
    del damp[:]
    del damperror[:]
    del timer[:]

p = [1000,2000,3000,5000,7000,10000] #Number of particles
l = [1*np.pi,2*np.pi,3*np.pi,4*np.pi,5*np.pi,6*np.pi,7*np.pi,8*np.pi,9*np.pi,10*np.pi] #Box
c = [10,15,20,25,30,35,40,45,50] #Number of cells


for n in range(0,9): #iterate over all values of box length, get averages from each
    analysis(2000,l[n],20,'length',n)
print type(l)
print l
print type(freq_avg)
print freq_avg
print type(freqerror_avg)
print freqerror_avg
plt.figure()
plt.errorbar(l,freq_avg,yerr=freqerror_avg) #plot average values against box length
plt.xlabel("Length of box")
plt.ylabel("Frequency")
plt.title("Frequency as a function of length of box")
plt.figure()
plt.errorbar(l,noise_avg,yerr=noiserror_avg)
plt.xlabel("Length of box")
plt.ylabel("Noise Amplitude")
plt.title("Noise Amplitude as a function of length of box")
```

```
plt.figure()
plt.errorbar(l,damp_avg,yerr=damperror_avg)
plt.xlabel("Length of box")
plt.ylabel("Damping Rate")
plt.title("Damping Rate as a function of length of box")
plt.figure()
plt.plot(l,timer_avg)
plt.xlabel("Length of box")
plt.ylabel("Calculation Time")
plt.title("Frequency as a function of length of box")
plt.show()
del freq_avg[:] #clear arrays
del freqerror_avg[:]
del noise_avg[:]
del noiserror_avg[:]
del damp_avg[:]
del damperror_avg[:]
del timer_avg[:]
'''
for n in range(0,6):
    analysis(p[n],4*np.pi,20,'parts',n)
plt.figure()
plt.errorbar(p,freq_avg,yerr=freqerror_avg)
plt.xlabel("Number of particles")
plt.ylabel("Frequency")
plt.title("Frequency as a function of number of particles")
plt.figure()
plt.errorbar(p,noise_avg,yerr=noiserror_avg)
plt.xlabel("Number of particles")
plt.ylabel("Noise Amplitude")
plt.title("Noise Amplitude as a function of number of particles")
plt.figure()
plt.errorbar(p,damp_avg,yerr=damperror_avg)
plt.xlabel("Number of particles")
plt.ylabel("Damping Rate")
plt.title("Damping Rate as a function of number of particles")
plt.figure()
plt.plot(p,timer_avg)
plt.xlabel("Number of particles")
plt.ylabel("Calculation Time")
plt.title("Calculation Time as a function of number of particles")
plt.show()
del freq_avg[:]
del freqerror_avg[:]
del noise_avg[:]
del noiserror_avg[:]
```

```
del damp_avg[:]
del damperror_avg[:]
del timer_avg[:]

'''''''
for n in range(0,9):
    analysis(2000,4*np.pi,c[n],'cells',n)
plt.figure()
plt.errorbar(c,freq_avg,yerr=freqerror_avg)
plt.xlabel("Number of cells")
plt.ylabel("Frequency")
plt.title("Frequency as a function of number of cells")
plt.figure()
plt.errorbar(c,noise_avg,yerr=noiserror_avg)
plt.xlabel("Number of cells")
plt.ylabel("Noise Amplitude")
plt.title("Noise Amplitude as a function of number of cells")
plt.figure()
plt.errorbar(c,damp_avg,yerr=damperror_avg)
plt.xlabel("Number of cells")
plt.ylabel("Damping Rate")
plt.title("Damping Rate as a function of number of cells")
plt.figure()
plt.plot(c,timer_avg)
plt.xlabel("Number of cells")
plt.ylabel("Calculation Time")
plt.title("Calculation time as a function of number of cells")
plt.show()
'''
```