Overview

Eigenvalue problems show up all over in physics and this week we will give some examples and learn how to solve them numerically (using python). Two notable and familiar examples that we will

1. Standing waves on a string
2. Quantum Bound states (time-independent Schrodinger equation)

Standing waves on a string

In PH123 you studied standing waves, both acoustical waves and waves on a string. The differential equation for standing waves on a string is given by:

$$Tg''(x) + \mu\omega^2 g(x) = 0$$

where T is the tension in the string, and μ is the mass per unit length of the string. Let's rearrange it a little:

$$g''(x) = -\frac{\mu\omega^2}{T} g(x) \tag{3}$$

Differential equation of this form, where a function is acted on by some differential operator and is then set equal to itself times some constant, is called an eigenvalue problem. Notice that there are two unknowns in this equation: $\lambda = -\frac{\mu\omega^2}{T}$, which we'll call the eigenvalues, and $g(x)$, which we'll call the eigenfunction. To solve this differential equations, we'll need two boundary conditions. Let's choose simple ones for now:

$$g(0) = 0 \qquad g(L) = 0$$

The key to solving this equation numerically is to first recall the finite-difference version of the second derivative that we have used so many times before. Using it, we can re-write this equation in index notation:

$$\frac{g_{i+1} - 2g_i + g_{i-1}}{\Delta x^2} = -\frac{\mu\omega^2}{T} g_i \tag{4}$$

With this equation, we can now frame the problem as a linear algebra problem (with matrices..):

$$\mathbf{A}\mathbf{g} = \lambda \mathbf{g} \tag{5}$$

$$\begin{bmatrix} ? & ? & ? & ? & \dots & ? & ? & ? \\ \frac{1}{\Delta x^2} & -\frac{2}{\Delta x^2} & \frac{1}{\Delta x^2} & 0 & \dots & 0 & 0 & 0 \\ 0 & \frac{1}{\Delta x^2} & -\frac{2}{\Delta x^2} & \frac{1}{\Delta x^2} & \dots & 0 & 0 & 0 \\ . & . & . & . & \dots & . & . & . \\ . & . & . & . & \dots & . & . & . \\ . & . & . & . & \dots & . & . & . \\ 0 & 0 & 0 & 0 & \dots & \frac{1}{\Delta x^2} & -\frac{2}{\Delta x^2} & \frac{1}{\Delta x^2} \\ ? & ? & ? & ? & \dots & ? & ? & ? \end{bmatrix} \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ . \\ . \\ . \\ g_{N-1} \\ g_N \end{bmatrix} = \lambda \begin{bmatrix} ? \\ g_2 \\ g_3 \\ . \\ . \\ . \\ g_{N-1} \\ ? \end{bmatrix} \tag{6}$$

Each row in the matrix above are equation (4) evaluated for different values of i. (Read that last sentence again!) Notice that only the middle rows are filled with numbers and the top and bottom rows are blank. (question marks actually) The top and bottom row of the matrix are the boundary conditions equations and the question marks are to remind us that we need to invent something to put in these rows that will implement the correct boundary conditions. Note that having question marks in the g-vector on the right is a real problem because without g_1 and g_N in the top and bottom positions, we don't have an eigenvalue problem (i.e. the vector g on left side of the equation is not the same as the vector g on the right side).

The simplest way to deal with this question-mark problem and to also handle the boundary conditions is to change the form of Eq. (5) to the slightly more complicated form of a *generalized eigenvalue problem*, like this:

$$\mathbf{A}\mathbf{g} = \lambda \mathbf{B}\mathbf{g} \tag{1}$$

In matrix form, the equation looks like this:

$$\mathbf{A} \qquad \mathbf{g} \quad = \lambda \qquad \mathbf{B} \qquad \mathbf{g} \tag{2}$$

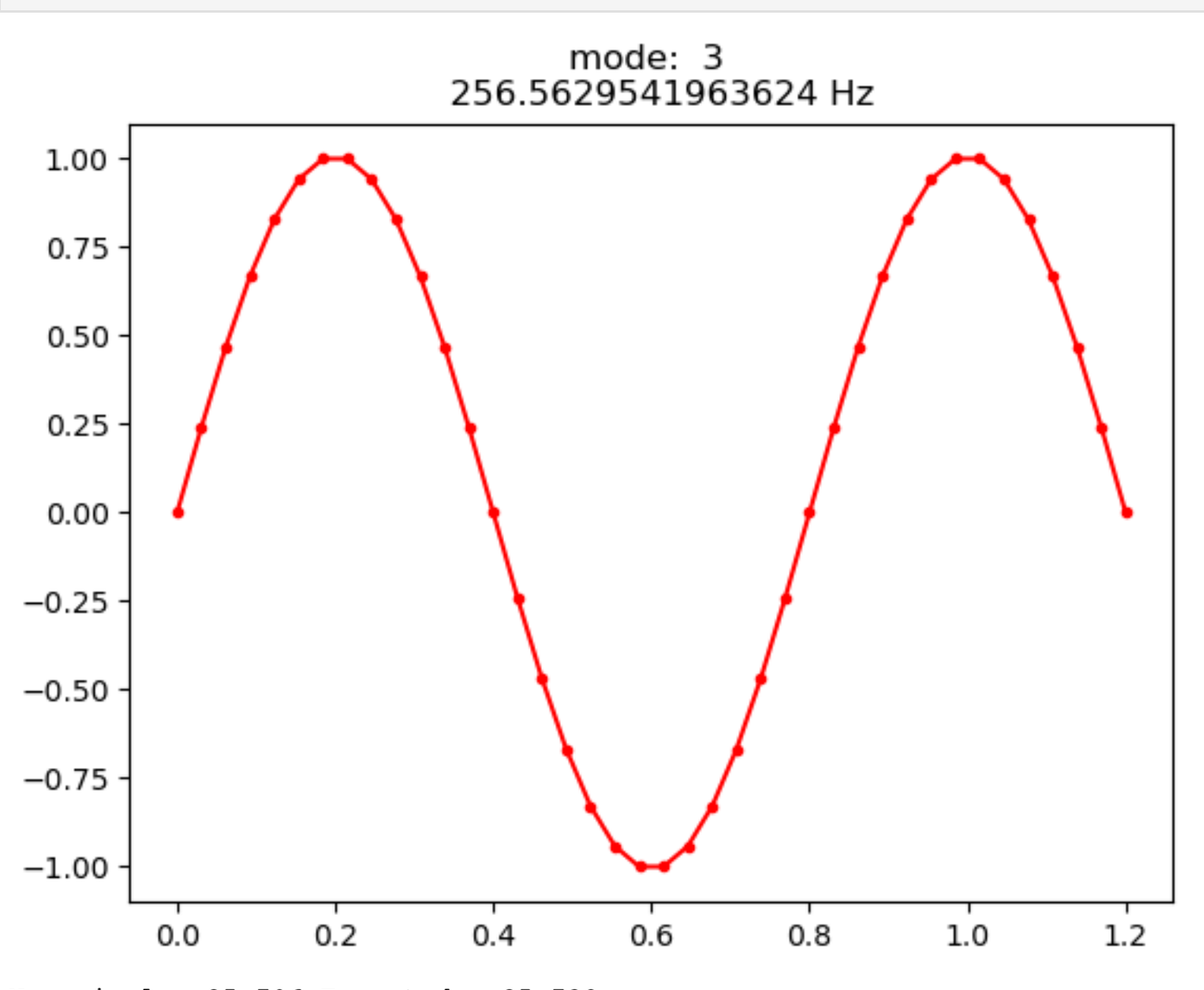
$$\begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ \frac{1}{\Delta x^2} & -\frac{2}{\Delta x^2} & \frac{1}{\Delta x^2} & \dots & 0 & 0 \\ 0 & \frac{1}{\Delta x^2} & -\frac{2}{\Delta x^2} & \dots & 0 & 0 \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ 0 & 0 & 0 & \dots & -\frac{2}{\Delta x^2} & \frac{1}{\Delta x^2} \\ 0 & 0 & 0 & \dots & 0 & 1 \end{bmatrix} \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ . \\ . \\ . \\ g_{N-1} \\ g_N \end{bmatrix} = \lambda \begin{bmatrix} 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ . & . & . & \dots & . & . \\ 0 & 0 & 0 & \dots & 1 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 \end{bmatrix} \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ . \\ . \\ . \\ g_{N-1} \\ g_N \end{bmatrix} \tag{7}$$

1. Take the time to convince yourself that equation (3) is equivalent to equation (4)
2. By performing the matrix multiplation by hand (or on a whiteboard), convince yourself that equations (7) and (6) are equivalent to equation (4).
3. Below you will find a code that solves the eigenvalue problem discussed above. Study the code (asking questions where needed) and add comments next to each line.
4. You may remember from PH123 that the expected frequencies for this situation are: $f_n = \sqrt{\frac{T}{\mu}}\frac{n}{2L}$ (for $n = 1, 2, 3 \ldots$). Check that your numerical results agree with the analytical result.
5. A "free-end" boundary condition can be enforced by making the last row of the matrix $\mathbf{A}$ produce the equation given below. Modify the code to implement this boundary condition and verify that the resulting plots of the standing waves seem correct.

$$\frac{1}{2\Delta x} g_{N-2} - \frac{2}{\Delta x} g_{N-1} + \frac{3}{2\Delta x} g_N = 0$$

6. The analytic solution for the situation where one end is free says that the expected family of freqencies is: $f_n = \sqrt{\frac{T}{\mu}}\frac{2n-1}{4L}$ (for $n = 1, 2, 3 \ldots$). Once again, verify that your numerical results are agreeing with the analytic one.

In [19]:
```python
import numpy as np
class eig:
    def __init__(self,a,b,N,T,mu):
        from numpy import linspace
        self.L = b # Second boundary
        self.N = N # number of points on the wave
        self.x,self.dx = linspace(a,b,N,retstep=True) # Set x array from a to b, step size dx
        self.T, self.mu = T,mu # tension and mass density of string

    def loadMatrices(self, V : str = ''):
        from numpy import zeros,sin,cos,eye
        # Make a potential energy term, depending on the type of potential desired
        if V == 'harmonic oscillator':
            V = np.zeros((N, N), float)
            for i in range(N):
                V[i, i] = 0.5 * self.x[i] ** 2
        else:
            V = np.zeros((N, N), float)

        self.A = zeros((self.N, self.N)) # set A to zero
        self.A[0, 0] = 1 # update the boundaries at the corners
        self.A[-1, -1] = 1

        for i in range(1,self.N-1):
            # Change the middle components of the matrix to match the differential equation above
            self.A[i,i] = -2./self.dx**2 + V[i, i]
            self.A[i,i+1] = 1./self.dx**2
            self.A[i,i-1] = 1./self.dx**2

        self.B = eye(self.N) # Make an NxN identity matrix
        self.B[0] = 0 # And enforce boundary conditions
        self.B[-1] = 0

    def solveProblem(self):
        from scipy.linalg import eig
        from numpy import sqrt,pi,argsort,isinf

        eVals,eVecs = eig(self.A,self.B) # solve for eigenvalues (E)
        eVecs = eVecs[:,~isinf(eVals)] # Remove the infinities from the arrays if any
        eVals = eVals[~isinf(eVals)]
        freq = sqrt(-self.T/self.mu * eVals)/(2 * pi) # get the harmonic frequencies in Hz
        self.eVecs = eVecs[:,argsort(freq)]
        self.freq = sorted(freq)

    def plot(self,mode):
        from numpy import real
        from matplotlib import pyplot

        fig,ax = pyplot.subplots(1,1)
        prefactor = 1/max(abs(self.eVecs[:,mode-1]))
        ax.plot(self.x,prefactor * self.eVecs[:,mode-1],'r.-')
        ax.set_title(f"mode:  {mode} \n  {str(real(self.freq[mode-1]))} Hz")
        pyplot.show()

a = 0
b = 1.2
N = 40
T = 127
mu = 0.003
n = 3

def ClassicalFrequencies(T : float, mu : float, L : float, N : int):
    return [(T / mu) ** 0.5 * n / (2 * L) for n in range(1, N + 1)]

myEig = eig(a,b,N,T,mu)
myEig.loadMatrices(V = None)
myEig.solveProblem()
myEig.plot(n)

for numerical, expected in zip(myEig.freq, ClassicalFrequencies(T, mu, (b - a), N)):
    print(f"Numerical : {numerical.real:.3f} Expected : {expected:.3f}")
```




```
Numerical : 85.706 Expected : 85.729
Numerical : 171.274 Expected : 171.459
Numerical : 256.563 Expected : 257.188
Numerical : 341.436 Expected : 342.918
Numerical : 425.756 Expected : 428.647
Numerical : 509.385 Expected : 514.377
Numerical : 592.187 Expected : 600.106
Numerical : 674.029 Expected : 685.836
Numerical : 754.778 Expected : 771.565
Numerical : 834.303 Expected : 857.294
Numerical : 912.474 Expected : 943.024
Numerical : 989.166 Expected : 1028.753
Numerical : 1064.253 Expected : 1114.483
Numerical : 1137.613 Expected : 1200.212
Numerical : 1209.129 Expected : 1285.942
Numerical : 1278.683 Expected : 1371.671
Numerical : 1346.163 Expected : 1457.400
Numerical : 1411.460 Expected : 1543.130
Numerical : 1474.467 Expected : 1628.859
Numerical : 1535.083 Expected : 1714.589
Numerical : 1593.209 Expected : 1800.318
Numerical : 1648.751 Expected : 1886.048
Numerical : 1701.618 Expected : 1971.777
Numerical : 1751.725 Expected : 2057.507
Numerical : 1798.992 Expected : 2143.236
Numerical : 1843.340 Expected : 2228.965
Numerical : 1884.698 Expected : 2314.695
Numerical : 1922.999 Expected : 2400.424
Numerical : 1958.181 Expected : 2486.154
Numerical : 1990.187 Expected : 2571.883
Numerical : 2018.965 Expected : 2657.613
Numerical : 2044.468 Expected : 2743.342
Numerical : 2066.655 Expected : 2829.072
Numerical : 2085.490 Expected : 2914.801
Numerical : 2100.942 Expected : 3000.530
Numerical : 2112.986 Expected : 3086.260
Numerical : 2121.603 Expected : 3171.989
Numerical : 2126.779 Expected : 3257.719
```

Results for Problem 4

Above, the results are given for this state with $N = 40$. It is clear to see that the results closely match the expected values up to around the 10th or 11th term.

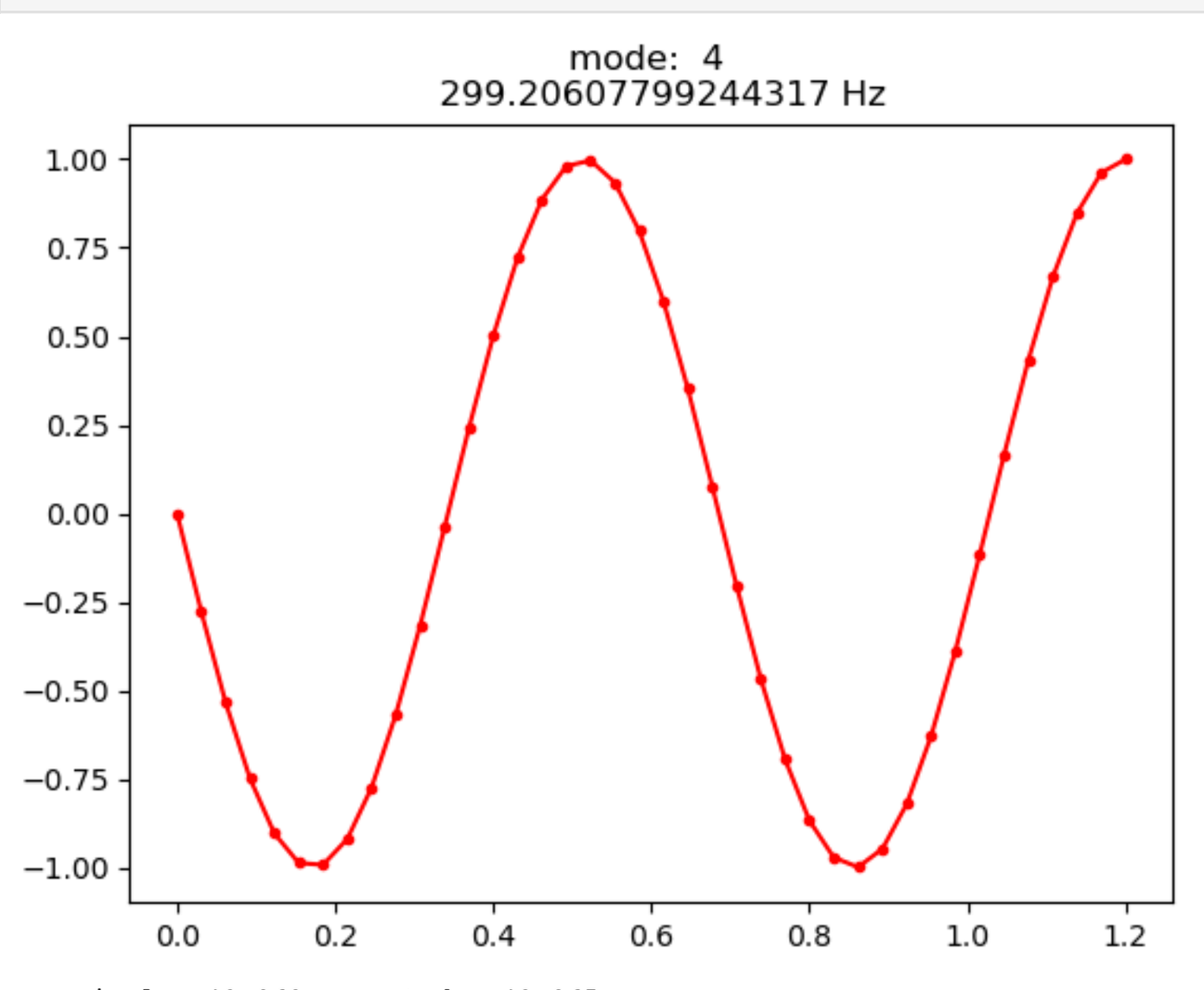
Results for Problem 5

Below, it is shown that the modification to 'loosen' an end of the wave function is straightforward, by simply changing 3 values in the A matrix.

Results for Problem 6

And lastly, the expected frequencies vs the numerical approximations are printed below. As it is shown, they are also very accurate to the expected values for smaller frequencies. Higher accuracies may be achieved by increasing N.

In [46]:
```python
import numpy as np

class eig2:
    def __init__(self,a,b,N,T,mu):
        from numpy import linspace
        self.L = b # Second boundary
        self.N = N # number of points on the wave
        self.x, self.dx = linspace(a,b,N,retstep=True) # Set x array from a to b, step size dx
        self.T, self.mu = T, mu # tension and mass density of string

    def loadMatrices(self, V : str = ''):
        from numpy import zeros,sin,cos,eye
        # Make a potential energy term, depending on the type of potential desired
        if V == 'harmonic oscillator':
            V = np.zeros((N, N), float)
            for i in range(N):
                V[i, i] = 0.5 * self.x[i] ** 2
        else:
            V = np.zeros((N, N), float)

        self.A = zeros((self.N, self.N)) # set A to zero
        self.A[0, 0] = 1 # update the boundaries at the corners
        self.A[-1, -1] = 1

        for i in range(1, self.N - 1):
            # Change the middle components of the matrix to match the differential equation above
            self.A[i, i] = -2. / self.dx ** 2 + V[i, i]
            self.A[i, i + 1] = 1. / self.dx ** 2
            self.A[i, i - 1] = 1. / self.dx ** 2

        #####################################
        self.A[-1, -3] = 1. / (2. * self.dx) # Boundary modification
        self.A[-1, -2] = -2. / self.dx
        self.A[-1, -1] = 3. / (2. * self.dx)
        #####################################

        self.B = eye(self.N) # Make an NxN identity matrix
        self.B[0] = 0 # And enforce boundary conditions
        self.B[-1] = 0 # Boundary conditions

    def solveProblem(self):
        from scipy.linalg import eig
        from numpy import sqrt,pi,argsort,isinf

        eVals,eVecs = eig(self.A,self.B) # solve for eigenvalues (E)
        eVecs = eVecs[:,~isinf(eVals)] # Remove the infinities from the arrays if any
        eVals = eVals[~isinf(eVals)]
        freq = sqrt(-self.T/self.mu * eVals)/(2 * pi) # get the harmonic frequencies in Hz
        self.eVecs = eVecs[:,argsort(freq)]
        self.freq = sorted(freq)

    def plot(self,mode):
        from numpy import real
        from matplotlib import pyplot

        fig, ax = pyplot.subplots(1, 1)
        prefactor = 1 / max(abs(self.eVecs[:,mode-1]))
        ax.plot(self.x, prefactor * self.eVecs[:, mode - 1], 'r.-')
        ax.set_title(f"mode:  {mode} \n  {str(real(self.freq[mode-1]))} Hz")
        pyplot.show()

a = 0
b = 1.2
N = 40
T = 127
mu = 0.003
n = 4

# Modified function for the new frequencies expected with free end
def ClassicalFrequencies(T : float, mu : float, L : float, N : int):
    return [(T / mu) ** 0.5 * (2 * n - 1) / (4 * L) for n in range(1, N + 1)]

myEig = eig2(a, b, N, T, mu)
myEig.loadMatrices(V = None)
myEig.solveProblem()
myEig.plot(n)

for numerical, expected in zip(myEig.freq, ClassicalFrequencies(T, mu, (b - a), N)):
    print(f"Numerical : {numerical.real:.3f} Expected : {expected:.3f}")
```




```
Numerical : 42.862 Expected : 42.865
Numerical : 128.528 Expected : 128.594
Numerical : 214.016 Expected : 214.324
Numerical : 299.206 Expected : 300.053
Numerical : 383.975 Expected : 385.782
Numerical : 468.195 Expected : 471.512
Numerical : 551.735 Expected : 557.241
Numerical : 634.463 Expected : 642.971
Numerical : 716.242 Expected : 728.700
Numerical : 796.934 Expected : 814.430
Numerical : 876.399 Expected : 900.159
Numerical : 954.501 Expected : 985.889
Numerical : 1031.099 Expected : 1071.618
Numerical : 1106.059 Expected : 1157.347
Numerical : 1179.247 Expected : 1243.077
Numerical : 1250.532 Expected : 1328.806
Numerical : 1319.786 Expected : 1414.536
Numerical : 1386.884 Expected : 1500.265
Numerical : 1451.708 Expected : 1585.995
Numerical : 1514.140 Expected : 1671.724
Numerical : 1574.069 Expected : 1757.454
Numerical : 1631.388 Expected : 1843.183
Numerical : 1685.996 Expected : 1928.912
Numerical : 1737.795 Expected : 2014.642
Numerical : 1786.694 Expected : 2100.371
Numerical : 1832.605 Expected : 2186.101
Numerical : 1875.448 Expected : 2271.830
Numerical : 1915.147 Expected : 2357.560
Numerical : 1951.631 Expected : 2443.289
Numerical : 1984.838 Expected : 2529.019
Numerical : 2014.707 Expected : 2614.748
Numerical : 2041.187 Expected : 2700.477
Numerical : 2064.231 Expected : 2786.207
Numerical : 2083.798 Expected : 2871.936
Numerical : 2099.855 Expected : 2957.666
Numerical : 2112.373 Expected : 3043.395
Numerical : 2121.330 Expected : 3129.125
Numerical : 2126.711 Expected : 3214.854
```

Quantum Bound States (time-independent Schrodinger Equation)

Eigenvalue problems show up in quantum mechanics routinely. One interesting problem is that of the particle confined to a quadratic potential. The time-independent Schrodinger equation for this situation is:

$$-\frac{1}{2}\frac{d^2\psi}{d\xi^2} + \frac{1}{2}\xi^2\psi = \epsilon\psi \tag{8}$$

This equation might look a little strange to you because it has been put in dimensionless form so that the length scale of our problem is not so small. ($-5 < \xi < 5$ instead of $-5 \times 10^{-10} < x < 5 \times 10^{-10}$). The unitless energy ϵ is related to the true energy E via:

$$E = \epsilon\hbar\sqrt{\frac{k}{m}} \tag{9}$$

and the unitless variable ξ is related to length via:

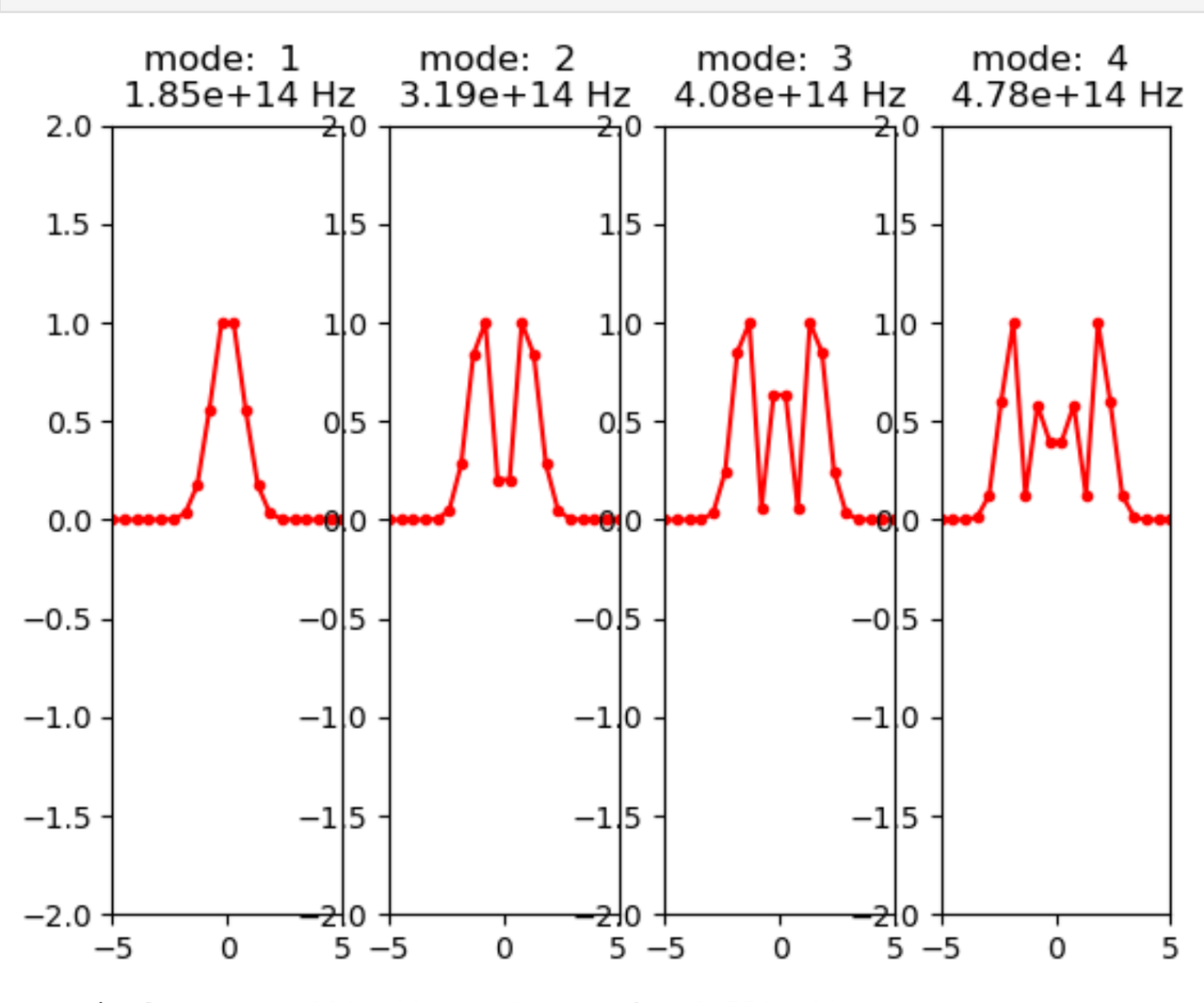
$$x = \sqrt{\frac{\hbar}{m\omega}}\xi$$

1. Following the same process that we followed above for standing waves on a string, find the bound states for this quantum system. Follow the steps below if you need help:
   A. Convert (8) into index notation, just like we did in equation (4) above. Note that the dimensionless ξ is now our independent variable instead of x
   B. Form the matrix $\mathbf{A}$ for this problem, similar to how we did in equation (7). The boundary conditions are $\psi(-5) = \psi(5) = 0$
   C. Copy the code from above and modify the function "loadMatrices" so that it constructs the correct matrix $\mathbf{A}$.
   D. The domain of your problem is $-5 < x < 5$. Hence modify your definition of "self.x" in the initialization routine.
   E. Instead of plotting the wavefunction: $\psi(x)$, better to plot $\psi \cdot \psi^*$ where ψ^* indicates complex conjugate. There is a function called "conjugate" in numpy that will find the complex conjugate of an array of numbers.
   F. You may want to normalize your wavefunction; in other words, you should plot $\frac{\psi \cdot \psi^*}{\int \psi \cdot \psi^* dx}$. The "dot" function from numpy will be your friend.
2. Once you get solutions that start making sense calculate the true energy of the first few eigenstates using equation (9). Verify that they agree with the analytical result:

$$E_n = (n + \frac{1}{2})\hbar\sqrt{\frac{k}{m}}$$

-{1\over 2} {\psi_{i + 1} - 2 \psi_i + \psi_{i - 1} \over (\Delta \xi)^2} + {1\over 2} \xi^2 \psi_i = \epsilon \psi_i \label{eq:Schrodinger}

In [104…
```python
class SchrodingerEquationSolver:
    def __init__(self, a, b, N, k, m):
        from numpy import linspace
        self.L = abs(b - a) # width of region
        self.N = N # number of points on the wavefunction
        self.ξ, self.dξ = linspace(a, b, N, retstep = True) # Set x array from a to b, step size dx
        self.k, self.m = k, m # wavenumber and mass of particle

    def loadMatrices(self):
        from numpy import zeros, sin, cos, eye

        # Make a potential energy term for the oscillator
        V = np.zeros((N, N), float)
        for i in range(N):
            V[i, i] = 0.5 * self.ξ[i] ** 2

        self.A = zeros((self.N, self.N)) # set A to zero
        self.A[0, 0] = 1 # update the boundaries at the 2 diagonal corners
        self.A[-1, -1] = 1

        for i in range(1, self.N - 1):
            # Change the middle components of the matrix to match the differential equation above
            self.A[i, i] = 1. / (self.dξ ** 2) + V[i, i]
            self.A[i, i + 1] = - 1. / (2. * self.dξ ** 2)
            self.A[i, i - 1] = - 1. / (2. * self.dξ ** 2)

        self.B = eye(self.N) # Make an NxN identity matrix
        self.B[0] = 0 # And enforce boundary conditions
        self.B[-1] = 0 # Boundary conditions

    def solveProblem(self):
        from scipy.linalg import eig
        from numpy import sqrt, pi, argsort, isinf

        eVals, eVecs = eig(self.A, self.B) # solve for eigenvalues (E) and eigenvectors (Ψ)
        eVecs = eVecs[:, ~isinf(eVals)] # Remove the infinities from the arrays if any
        eVals = eVals[~isinf(eVals)]
        freq = sqrt(self.k / self.m * eVals) / (2 * pi) # get the harmonic frequencies in Hz
        self.eVecs = eVecs[:, argsort(freq)]

        ########################## NORMALIZE ##########################
        # To normalize, we need the absolute square of the wavefunction by doing recursive dot products
        absoluteSquare = np.array([ψt * ψ for ψt, ψ in zip(np.conjugate(eVecs), self.eVecs)])

        # and the area is given to A2
        A2 = 0.0
        for i in range(1, self.N - 1):
            # Simpsons rule to integrate the probability ditribution
            A2 += self.dξ / 3 * (absoluteSquare[i - 1] + 4 * absoluteSquare[i] + absoluteSquare[i + 1])

        # Solving A^2 <ψ|ψ> = A^2 * <A2> = 1 for A gives
        A = np.array([1 / abs(a2) ** 0.5 for a2 in A2])

        # Save the probability distribution according to the energy
        self.probabilityDistribution = A * A * np.conjugate(self.eVecs) * self.eVecs
        ############################# DONE ############################

        self.freq = sorted(freq)

    def plot(self, modeList):
        from numpy import real
        from matplotlib import pyplot

        fig, ax = pyplot.subplots(1, len(modeList))
        for i, mode in enumerate(modeList):
            prefactor = 1 / max(abs(self.probabilityDistribution[:, mode - 1]))
            ax[i].plot(self.ξ, prefactor * self.probabilityDistribution[:, mode - 1], 'r.-')
            ax[i].set_title(f"mode:  {mode} \n  {real(self.freq[mode - 1]):.2e} Hz")
            ax[i].set_xlim(-5, 5)
            ax[i].set_ylim(-2, 2)

        pyplot.show()

ħ = 6.626e-34 / (2 * 3.14159265)

# Modified function for the new frequencies expected with free end
def QuantizedFrequencies(k : float, m : float, L : float, N : int):
    return [(k / m) ** 0.5 * ħ * (n + 1 / 2) for n in range(N)]

a = - 5.
b = 5.
N = 20
k = 4 / 10 * 2 * 3.14159265
m = 9.11e-31
n = list(range(1, 5))

solver = SchrodingerEquationSolver(a, b, N, k, m)
solver.loadMatrices()
solver.solveProblem()
solver.plot(n)

for numerical, expected in zip(solver.freq, QuantizedFrequencies(k, m, (b - a), N)):
    print(f"Numerical : \
    {numerical.real * 2 * 3.14159265 * ħ:.3e} \
    Expected : {expected:.3e}")
```




```
Numerical :      1.228e-19      Expected : 8.758e-20
Numerical :      2.113e-19      Expected : 2.627e-19
Numerical :      2.703e-19      Expected : 4.379e-19
Numerical :      3.166e-19      Expected : 6.131e-19
Numerical :      3.550e-19      Expected : 7.882e-19
Numerical :      3.878e-19      Expected : 9.634e-19
Numerical :      4.160e-19      Expected : 1.139e-18
Numerical :      4.395e-19      Expected : 1.314e-18
Numerical :      4.617e-19      Expected : 1.489e-18
Numerical :      4.713e-19      Expected : 1.664e-18
Numerical :      5.022e-19      Expected : 1.839e-18
Numerical :      5.028e-19      Expected : 2.014e-18
Numerical :      5.482e-19      Expected : 2.189e-18
Numerical :      5.482e-19      Expected : 2.365e-18
Numerical :      6.029e-19      Expected : 2.540e-18
Numerical :      6.029e-19      Expected : 2.715e-18
Numerical :      6.740e-19      Expected : 2.890e-18
Numerical :      6.740e-19      Expected : 3.065e-18
```

Conclusion

As shown above, the wavefunctions $\psi_n(\xi)$ acummulated a lot of error in the energies due to small deviations, most likely. Making use of Simpsons method to integrate the absolute square of the wavefunctions allowed me to find the normalizing amplitude factors to apply to each solution, but there was certainly some error accumulated in that process as well. To improve accuracy, I will rerun the program with a larger N.

In [105…
```python
a = - 5.
b = 5.
N = 200
k = 4 / 10 * 2 * 3.14159265
m = 9.11e-31
n = list(range(1, 5))

solver = SchrodingerEquationSolver(a, b, N, k, m)
solver.loadMatrices()
solver.solveProblem()
solver.plot(n)
```