Quantum Mechanics Problems

Ryan J. Glusic

May 09, 2021

CONTENTS

This Jupyter book is an assortment of computational physics problems from within Griffith's Quantum Mechanics, 3rd edition.

CHAPTER

ONE

APPROXIMATING THE PROBABILITY DENSITY FOR A HARMONIC OSCILLATOR

As per the problem, we need 1.11(b) solution to begin.

$\rho(x) = \frac{1}{\pi\sqrt{a^2-x^2}}$, where a is our upper bound.

Integrating this, one can verify that it is indeed normalized and equal to one from some $-a$ to a.

This is our analytical solution, now our numerical solution is as follows:

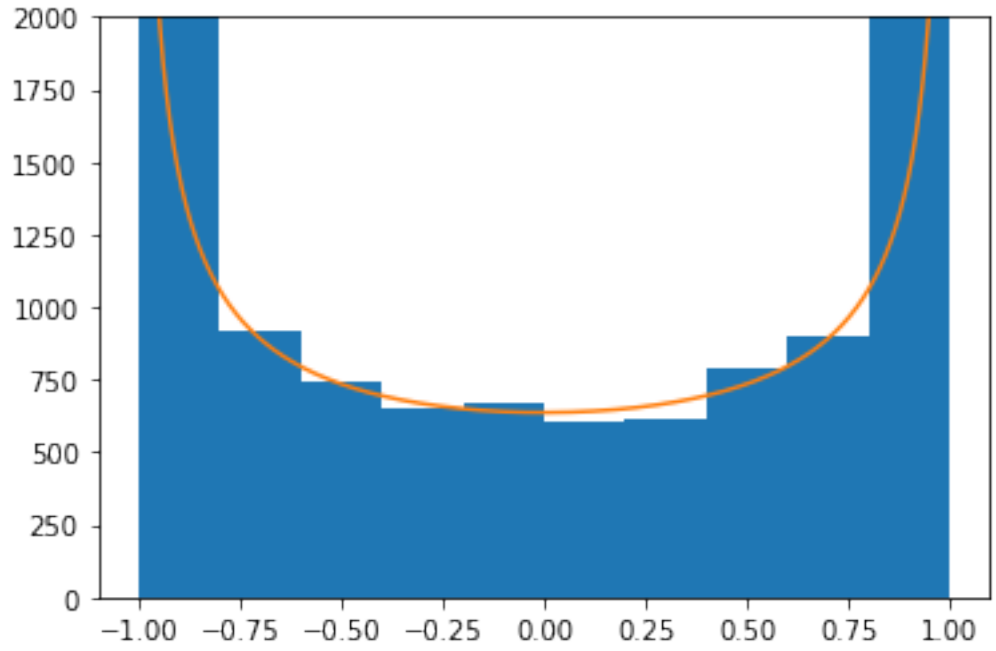
$\rho_{approx}(x)$ =a$\cdot \cos(\omega t)$, where we let a and ω both equal 1 as stated in the problem.

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

# Constants
a = 5
w = 1

# Point generation
x_inputs = np.linspace(-a, a, 10000)
analytical = 1/(np.pi*np.sqrt(a**2-x_inputs**2))
approx = a*np.cos(w*x_inputs)
```

```
<ipython-input-1-0294d58c52ee>:10: RuntimeWarning: divide by zero encountered in true_
 ↪divide
  analytical = 1/(np.pi*np.sqrt(a**2-x_inputs**2))
```

```
times = np.pi * np.random.rand(10000)
cosine = np.cos(times)
plt.hist(cosine)
plt.plot(np.linspace(-1, 1, 10000), 2000/(np.pi*np.sqrt(1**2 - np.linspace(-1, 1,␣
 ↪10000)**2)))
plt.ylim(0, 2000)
plt.show()
```

```
<ipython-input-2-4a4cbf29a3b7>:4: RuntimeWarning: divide by zero encountered in true_
 ↪divide
  plt.plot(np.linspace(-1, 1, 10000), 2000/(np.pi*np.sqrt(1**2 - np.linspace(-1, 1,␣
 ↪10000)**2)))
```


2000
1750
1500
1250
1000
750
500
250
0
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00

CHAPTER

TWO

FINDING THE EIGENSTATE ENERGIES

Problem 2.56 in Griffith's Quantum Mechanics 3rd Edition asks us to use the 'wag the dog' method for finding E_1, E_2 and E_3 energy states for a quantum oscillator.

CHAPTER

THREE

QUANTUM OSCILLATOR

$$\frac{\partial^2 \psi}{\partial \xi^2} = (\xi^2 - K)\psi$$

$$\xi = \sqrt{\frac{m\omega}{\hbar}} x, K = \frac{2E}{\hbar\omega}$$

CHAPTER

FOUR

FINDING EVEN AND ODD E_N

As hinted at in the problem description `For the first (and third) excited state you will need to set` $\psi(\xi) = 0, \psi'(\xi) = 1$. We need to flip our initial conditions to find the first and third energy states. That is for $n = 1$ and $n = 3$, odd n correspond to the odd eigenstates of the wave function; even n correspond to even eigenstates.

CHAPTER

FIVE

WAG THE DOG

The following code will numerically solve Eq. (1) using the wag the dog method. It's also listed in the common folder in this repository.

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

def wag_the_dog(x_range=(0,5), k_values=np.linspace(0.9, 1.1, 10), initial_values=(1.,
 ↪0.)):
    """
        This function numerically solves the wave equation for a harmonic oscillator,␣
 ↪based on a 'guess' for k and goes from there. This is for the problem 2.55 in␣
 ↪Griffth's quantum mechanics (3rd edition).
    """
    positions = np.linspace(x_range[0], x_range[1], 1000)
    plt.figure(figsize=(24,8))

    allowed_potentials = ['harmonic oscillator']
    for k_value in k_values:
        # Differential equation
        psi_prime = lambda xi, psi, k: [psi[1], (xi**2-k)*psi[0]]

        # Solve differential equation using scipy
        sol = solve_ivp(psi_prime, x_range, initial_values, t_eval=positions, args=[k_
 ↪value])

        # Plot solution
        plt.plot(positions, sol.y[0], label=fr'$k$: {k_value:.4f}')

    # Plotting configuration
    plt.legend()
    plt.axhline(c='black')
    plt.xlabel(r'$\xi$')
    plt.ylabel(r'$\psi_n(\xi)$')
    plt.ylim(-1,1)
    plt.show()
```

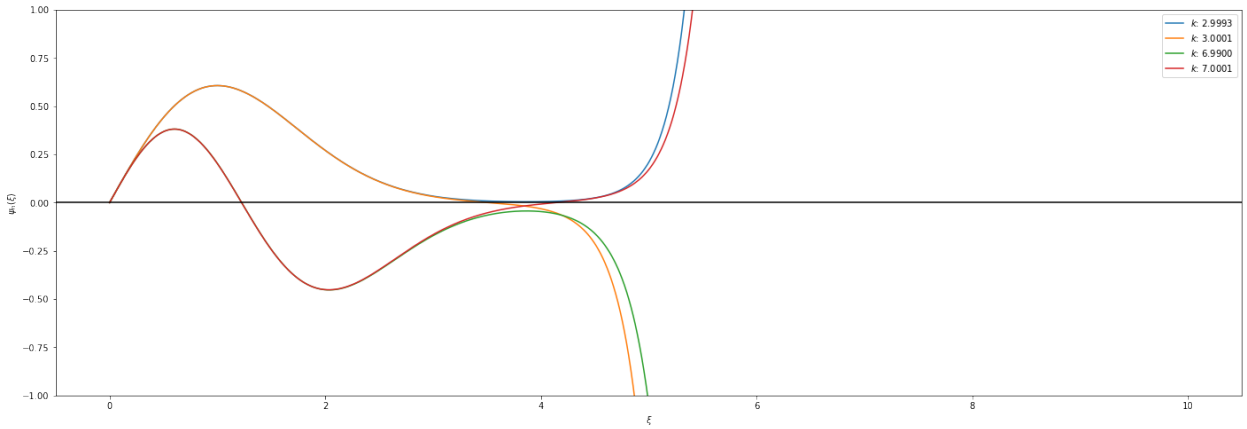
CHAPTER

SIX

USING THE ABOVE CODE

Using the above code, we can specify different K values to find the eigenstates through trial and error. Let's start by finding the even eigenstates.

```
wag_the_dog(x_range=(0,10), k_values=(2.9993, 3.0001, 6.9900, 7.0001), initial_
↪values=(0.,1.))
```



CHAPTER

SEVEN

CALCULATING THE ODD EIGENSTATE ENERGIES

As seen in the above code, it appears that around $K = 3$ and $K = 7$ the tail end of the wave function flips about the ξ axis. This indicates that the two lowest odd energy states E_1, E_3 are here as the graph did not flip anywhere below these points. We calculate the energy values as follows:

$$K = \frac{2E_1}{\hbar\omega} \approx 3 \implies E_1 = \frac{3}{2}\hbar\omega$$

$$K = \frac{2E_3}{\hbar\omega} \approx 7 \implies E_3 = \frac{7}{2}\hbar\omega$$

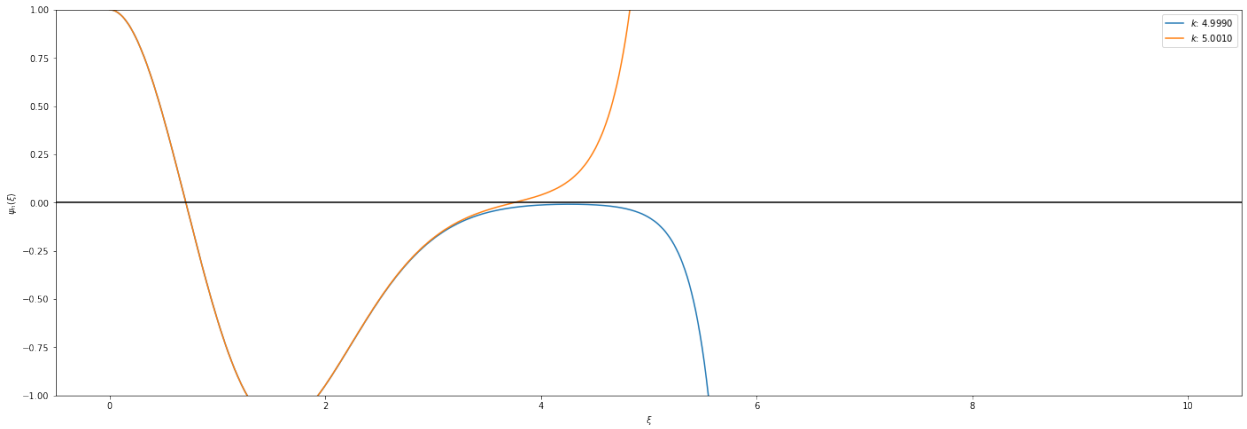
CHAPTER

EIGHT

CALCULATING THE EVEN EIGENSTATE ENERGIES

Using the new boundary conditions as stated above, we can calculate the energies for the even states (E_2)

```
wag_the_dog(x_range=(0,10), k_values=(4.999,5.001), initial_values=(1.,0.))
```



As we did above for the odd states,

$$K = \frac{2E_2}{\hbar\omega} \approx 5 \implies E_2 = \frac{5}{2}\hbar\omega$$

CHAPTER

NINE

PROBLEM 2.61

One way to obtain the allowed energies of a potential well numerically is to turn the Schrödinger equation into a matrix equation, by discretizing the variable x. Slice the relevant interval at evenly spaced points.

$$-\lambda\psi_{j-1} + (2\lambda + V_j)\psi - \lambda\psi_{j-1} = E\psi_j, \lambda = \frac{\hbar^2}{2m(\Delta x)^2}$$

```
# Create Hamiltonian Matrix
import numpy as np
def create_hamiltonian(dimension, end_point=0):
    x_naught = 0.
    x_end    = end_point
    delta_x  = end_point/(dimension+1)
    potential_well = 0. # Always zero inside the well

    # Let hbar = 1, m = 1.
    ham_mat = np.zeros((dimension,dimension), float)
    np.fill_diagonal(ham_mat[:,1:], -1)
    np.fill_diagonal(ham_mat, 2)
    np.fill_diagonal(ham_mat[1:], -1)
    w, v = np.linalg.eig(ham_mat)
    return (w,v)
create_hamiltonian(100)
```

```
(array([3.99903256e+00, 3.99613119e+00, 3.99129870e+00, 3.98453974e+00,
        3.97586088e+00, 3.96527050e+00, 3.95277884e+00, 9.67435416e-04,
        3.86880573e-03, 8.70130406e-03, 1.90671922e+00, 1.84463231e+00,
        1.96889638e+00, 2.03110362e+00, 2.09328078e+00, 3.93839800e+00,
        2.15536769e+00, 1.78269570e+00, 1.54602553e-02, 2.21730430e+00,
        2.40161415e+00, 2.34048711e+00, 1.65951289e+00, 1.59838585e+00,
        1.53764736e+00, 2.41391205e-02, 2.46235264e+00, 3.90402622e+00,
        3.88406853e+00, 1.47735615e+00, 2.52264385e+00, 3.86228812e+00,
        3.47295036e-02, 1.41757058e+00, 3.83870608e+00, 4.72211589e-02,
        2.58242942e+00, 1.35834846e+00, 6.16020016e-02, 1.29974710e+00,
        3.81334520e+00, 2.64165154e+00, 3.78623003e+00, 1.24182319e+00,
        2.70025290e+00, 1.15931473e-01, 9.59737849e-02, 1.37711876e-01,
        2.75817681e+00, 3.75738680e+00, 1.61293922e-01, 1.07267294e+00,
        9.64300750e-01, 1.01801184e+00, 9.11591634e-01, 1.86654798e-01,
        2.87176884e+00, 2.92732706e+00, 3.08840837e+00, 3.14006452e+00,
        3.19061773e+00, 3.03569925e+00, 8.59935484e-01, 2.13769968e-01,
        3.42516793e+00, 3.24001909e+00, 3.46811706e+00, 8.09382271e-01,
        3.38084004e+00, 2.42613200e-01, 3.50964588e+00, 7.59980905e-01,
        5.31882942e-01, 5.74832072e-01, 4.90354122e-01, 3.33517628e+00,
        2.98198816e+00, 6.19159959e-01, 4.50285786e-01, 2.73156590e-01,
```

(continues on next page)

(continued from previous page)

```
        3.72684341e+00, 3.54971421e+00, 4.11716698e-01, 3.58828330e+00,
        6.64823720e-01, 3.62531583e+00, 3.69462941e+00, 3.74684172e-01,
        1.12823116e+00, 3.92214188e+00, 3.05370590e-01, 3.66077597e+00,
        3.39224034e-01, 7.78581192e-02, 3.28822082e+00, 7.11779177e-01,
        1.18463277e+00, 2.81536723e+00, 1.72096932e+00, 2.27903068e+00]),
 array([[-0.00437636,  0.00874848, -0.01311214, ...,  0.12849426,
          0.13934326, -0.13934326],
        [ 0.00874848, -0.01746312,  0.02611019, ..., -0.10477001,
          0.03888104,  0.03888104],
        [-0.01311214,  0.02611019, -0.03888104, ..., -0.04306823,
         -0.12849426,  0.12849426],
        ...,
        [ 0.01311214,  0.02611019,  0.03888104, ...,  0.04306823,
          0.12849426,  0.12849426],
        [-0.00874848, -0.01746312, -0.02611019, ...,  0.10477001,
         -0.03888104,  0.03888104],
        [ 0.00437636,  0.00874848,  0.01311214, ..., -0.12849426,
         -0.13934326, -0.13934326]]))
```

CHAPTER

TEN

PROBLEM 2.62

Similar to problem 2.62, except now the potential well is $V(x) = \frac{1}{2}kx^2$ (harmonic oscillator). Lets make our life simple and do as the book said: ($\hbar = 1, m = 1, a = 1$).

CHAPTER

ELEVEN

GENERATING THE HAMILITONIAN

The following code will generate the Hamilitonian matrix for our problem.

```
# Create Hamiltonian Matrix
import numpy as np
import scipy.sparse.linalg as sp

def create_hamiltonian(dimension, end_point=0):
    # Let hbar = 1, m = 1.
    ham_mat = np.zeros((dimension,dimension), float)

    # Fill upper diagonal
    np.fill_diagonal(ham_mat[:,1:], -1)

    # Fill diagonal (potential changes the main diagonal)
    diag = [(2 + 1/2 * (j/dimension)**2) for j in range(1,dimension+1)]
    np.fill_diagonal(ham_mat, diag)

    # Fill lower diagonal
    np.fill_diagonal(ham_mat[1:], -1)

    # Retrieve vectors and values then return them
    w, v = sp.eigs(ham_mat, k=3, which='SR', return_eigenvectors=True)
    return (w,v)
```

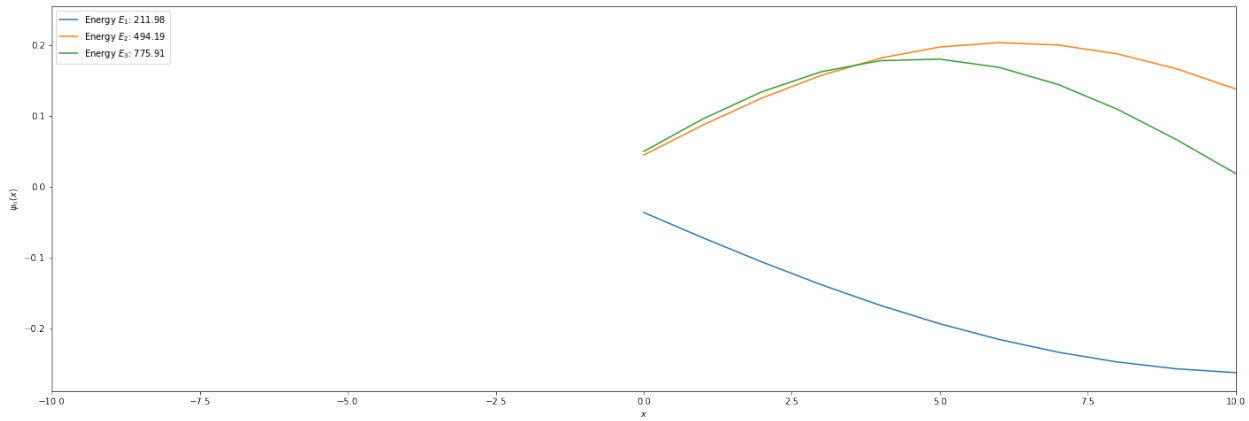
CHAPTER

TWELVE

USING THE ABOVE HAMILITONIAN

Lets use our function above to generate a 100x100 matrix and find the first three energy states along with the wave function values for these three energy states. We can then see the approximate wave function by plotting the results.

```
import matplotlib.pyplot as plt
evals, evecs = create_hamiltonian(100)

plt.figure(figsize=(24,8))

# Plot each eigenstate
for n in range(1,4):
    plt.plot(evecs.real[:, n-1], label=fr'Energy $E_{n}$: {100**2*evals.real[n-1]:.2f}
 ↪')

plt.xlabel(r'$x$')
plt.ylabel(r'$\psi_n(x)$')
plt.xlim(-10,10)
plt.legend()
plt.show()
```



CHAPTER

THIRTEEN

PROBLEM 2.62

Similar to problem 2.61, except now the diagonal is a function of x rather than a constant due to the varying well. Lets make our life simple and do as the book said: ($\hbar = 1, m = 1, a = 1$).

CHAPTER

FOURTEEN

GENERATING THE HAMILITONIAN

The following code will generate the Hamilitonian matrix for our problem.

```
# Create Hamiltonian Matrix
import numpy as np
import scipy.sparse.linalg as sp

def create_hamiltonian(dimension, end_point=0):
    # Let hbar = 1, m = 1.
    ham_mat = np.zeros((dimension,dimension), float)

    # Fill upper diagonal
    np.fill_diagonal(ham_mat[:,1:], -1)

    # Fill diagonal
    diag = [(2 + (500 / (dimension**2)) * np.sin(np.pi * j/dimension)) for j in␣
↪range(1,dimension+1)]
    np.fill_diagonal(ham_mat, diag)

    # Fill lower diagonal
    np.fill_diagonal(ham_mat[1:], -1)

    # Retrieve vectors and values then return them
    w, v = sp.eigs(ham_mat, k=3, which='SR', return_eigenvectors=True)
    return (w,v)
```

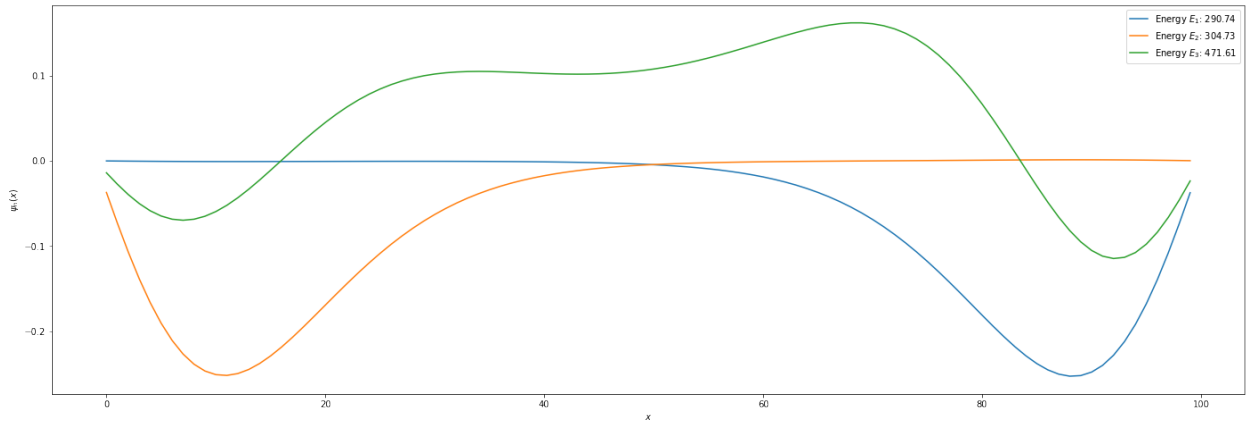
CHAPTER

FIFTEEN

USING THE ABOVE HAMILITONIAN

Lets use our function above to generate a 100x100 matrix and find the first three energy states along with the wave function values for these three energy states. We can then see the approximate wave function by plotting the results.

```
import matplotlib.pyplot as plt
evals, evecs = create_hamiltonian(100)

plt.figure(figsize=(24,8))

# Plot each eigenstate
for n in range(1,4):
    plt.plot(evecs.real[:, n-1], label=fr'Energy $E_{n}$: {100**2*evals.real[n-1]:.2f}
↪')

plt.xlabel(r'$x$')
plt.ylabel(r'$\psi_n(x)$')
plt.legend()
plt.show()
```



CHAPTER

SIXTEEN

GRAM-SCHMIDT ORTHONORMALIZATION

The following produces the Gram-Schmit orthonormalization on arbitrary basis vectors. In the case of this example, we will be using the following vectors:

$|e1\rangle = (1+i)i + (1)j + (i)k$, $|e2\rangle = (i)i + (3)j + (1)k$, $|e3\rangle = (0)i + (28)j + (8)k$

CHAPTER

SEVENTEEN

THE GRAM-SCHMIDT

We can create our Gram-Schmit Orthonormalization function:

```
import numpy as np
def gram_schmidt(vectors):
    bases = []
    for vec in vectors:
        w = vec - np.sum(np.dot(vec, q) * q for q in bases)
        bases.append(w / np.linalg.norm(w))
    return np.array(bases)
```

CHAPTER

EIGHTEEN

EXAMPLE BASES

Let's now use our `gram_schmidt` function to evaluate the original bases given at the top of this page:

$|e1\rangle = (1+i)i + (1)j + (i)k$, $|e2\rangle = (i)i + (3)j + (1)k$, $|e3\rangle = (0)i + (28)j + (8)k$

```
bases = np.array([
    [1+1.j, 1., 1.j],
    [1.j, 3., 1.],
    [0., 28., 8.]
]).T

print(gram_schmidt(bases))
```

```
[[0.57735027+0.57735027j 0.         +0.57735027j 0.         +0.j        ]
 [0.0702873 -0.05857275j 0.15228915-0.01171455j 0.98402221+0.j        ]
 [0.31547059+0.83647199j 0.32032088+0.30859935j 0.03395199-0.04243999j]]
```

```
<ipython-input-1-fb9ee8129d3d>:5: DeprecationWarning: Calling np.sum(generator) is␣
 ↪deprecated, and in the future will give a different result. Use np.sum(np.
 ↪fromiter(generator)) or the python sum builtin instead.
  w = vec - np.sum(np.dot(vec, q) * q for q in bases)
```

CHAPTER

NINETEEN

ANOTHER EXAMPLE

Below I test the following basis vectors as required by the Homework 3B assignment: $|e1\rangle = (1+i)i+(1)j+(i)k, |e2\rangle = (i)i + (3)j + (1)k, |e3\rangle = (0)i + (28)j + (8)k$

Note: I enter them as row vectors, then transpose them using .T as it's easier to type.

```
bases_2 = np.array([
    [1+1.j, 1, 1.j],
    [1.j, 3, 1],
    [0, 28, 8]
]).T

print(gram_schmidt(bases_2))
```

```
[[0.57735027+0.57735027j 0.         +0.57735027j 0.         +0.j        ]
 [0.0702873 -0.05857275j 0.15228915-0.01171455j 0.98402221+0.j        ]
 [0.31547059+0.83647199j 0.32032088+0.30859935j 0.03395199-0.04243999j]]
```

```
<ipython-input-1-fb9ee8129d3d>:5: DeprecationWarning: Calling np.sum(generator) is␣
 ↪deprecated, and in the future will give a different result. Use np.sum(np.
 ↪fromiter(generator)) or the python sum builtin instead.
  w = vec - np.sum(np.dot(vec, q) * q for q in bases)
```

CHAPTER

TWENTY

FINDING BOHR ENERGIES IN HYDROGEN

We can use the 'wag the dog' method to find Bohr energies in the Hydrogen atom. From the book, we know that

$$\frac{d^2u}{d\rho^2} = \left[1 - \frac{\rho_0}{\rho} + \frac{l(l+1)}{\rho^2}\right] u$$

Let's let

$$\rho_0 = 2n$$

Then we can find the discrete energy levels using the 'wag the dog' method. We'll do it for 3 discrete n for each $l \in \{0, 1, 2\}$

```
import numpy as np
from scipy.integrate import solve_ivp
import matplotlib.pyplot as plt

def wag_the_dog(x_range=(0.0000001,25), n_values=np.linspace(0.9, 1.1, 10), initial_
↪values=(1.,0.), l=0):
    positions = np.linspace(x_range[0], x_range[1], 1000)
    plt.figure(figsize=(24,8))

    for n_value in n_values:
        # Differential equation
        psi_prime = lambda rho, psi, n: [psi[1], (1-(2*n/(rho)) + (l*(l+1)/
↪(rho**2)))*psi[0]]

        # Solve differential equation using scipy
        sol = solve_ivp(psi_prime, x_range, initial_values, t_eval=positions, args=[n_
↪value])

        # Plot solution
        plt.plot(positions, sol.y[0], label=fr'$n$: {n_value:.4f}, $l$: {l}')

    # Plotting configuration
    plt.legend()
    plt.axhline(c='black')
    plt.xlabel(r'$\rho$')
    plt.ylabel(r'$u_n(\rho)$')
    plt.ylim(-5,5)
    plt.show()
```

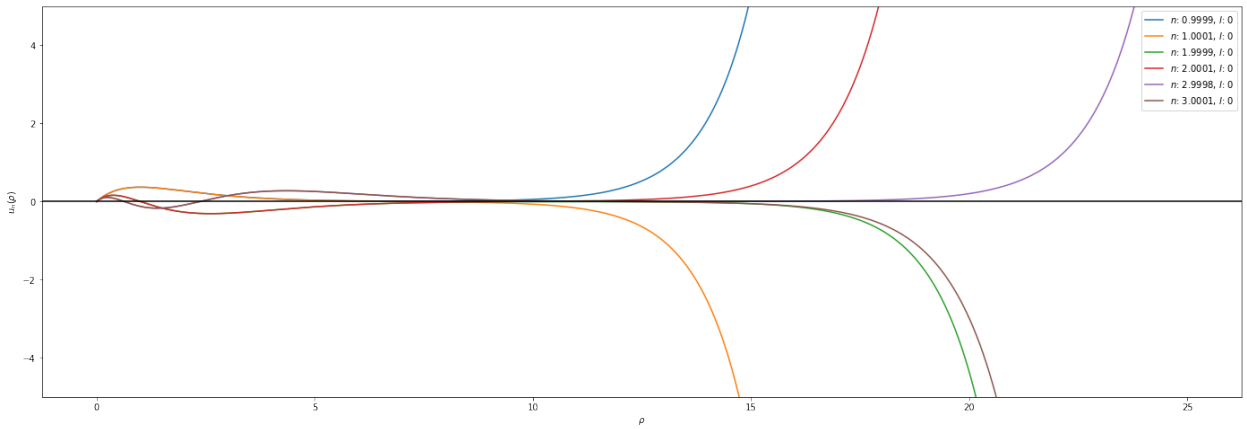
CHAPTER

TWENTYONE

VISUALIZATION

for $l = 0$:

```
wag_the_dog(initial_values=(0., 1.), n_values=[0.9999, 1.0001, 1.9999, 2.0001, 2.9998,
 ↪ 3.0001], l=0)
```



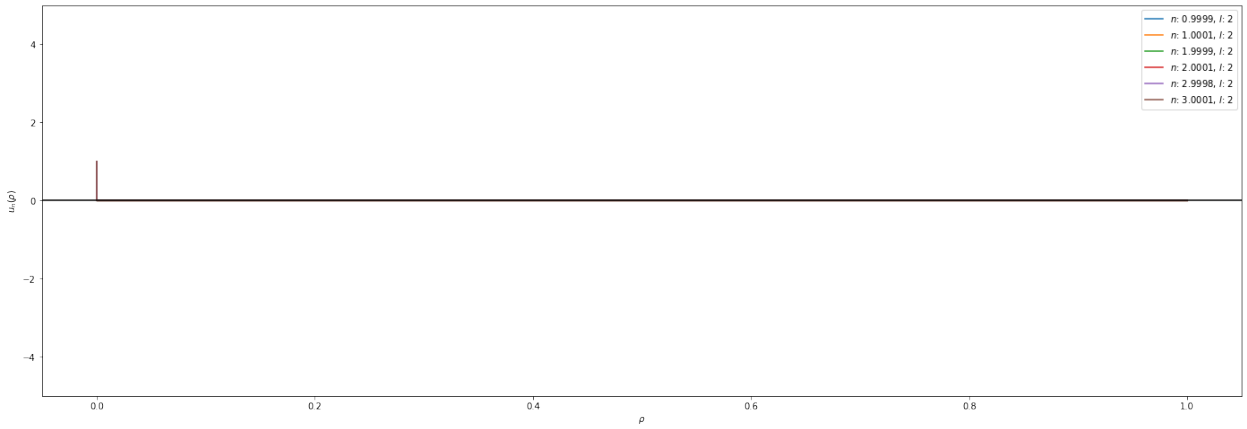
for $l = 1$, our initial value problem becomes a boundary value problem. This is due to $u(1) = 1, u'(0) = 0$

```
from scipy.integrate import solve_bvp
def wag_the_dog2(x_range=(0.0000001,25), n_values=np.linspace(0.9, 1.1, 10), initial_
 ↪u=1., initial_up=0., l=0):
    positions = np.linspace(x_range[0], x_range[1], 1000)
    plt.figure(figsize=(24,8))

    for n_value in n_values:
        # Differential equation
        u_pp = lambda rho, u: [u[1], (1 - (2*n_value/(rho)) + (l*(l+1)/
 ↪(rho**2)))*u[0]]
        bc   = lambda u1, u2: [u1[0]-initial_u, u2[0]-initial_up]

        sol = solve_bvp(u_pp, bc, [0.0000001, 1.], [[0.,0.],[1.,0.]])

        # Plot solution
        plt.plot(sol.x, sol.y[0], label=fr'$n$: {n_value:.4f}, $l$: {l}')

    # Plotting configuration
    plt.legend()
    plt.axhline(c='black')
    plt.xlabel(r'$\rho$')
```

(continues on next page)

(continued from previous page)

```
    plt.ylabel(r'$u_n(\rho)$')
    plt.ylim(-5,5)
    plt.show()

wag_the_dog2(x_range=(1,25), n_values=[0.9999, 1.0001, 1.9999, 2.0001, 2.9998, 3.
↪0001], l=2)
```



$l = 2$

```
wag_the_dog2(x_range=(1,25), n_values=[0.9999, 1.0001, 1.9999, 2.0001, 2.9998, 3.
↪0001], l=3)
```