Chance's Homework 2

Problem 1

Issues: Had problems running root_scalar function of scipy.optimize, so I changed it to root function instead and addacted it accordingly.

For the cubic perturbation, the energy levels are lower than without a perturbation. For the eigenfunctions of the cubic perturbation, the spacing between values (lines) of the eigenfunctions for different energy levels is greater than without a perturbation.

For the quartic perturbation, the energy levels are higher than without a perturbation. For the eigenfunctions of the quartic perturbation, the spacing between values (lines) of the eigenfunctions for different energy levels is less than without a perturbation.

Problem 2

For part a, changed the variables accordingly and hit run.

For part b, comapred to expected exact solution. Got amplitudes to match for this specific case.

schroedingerCubic

March 2, 2021

1 Problem 1 Part a

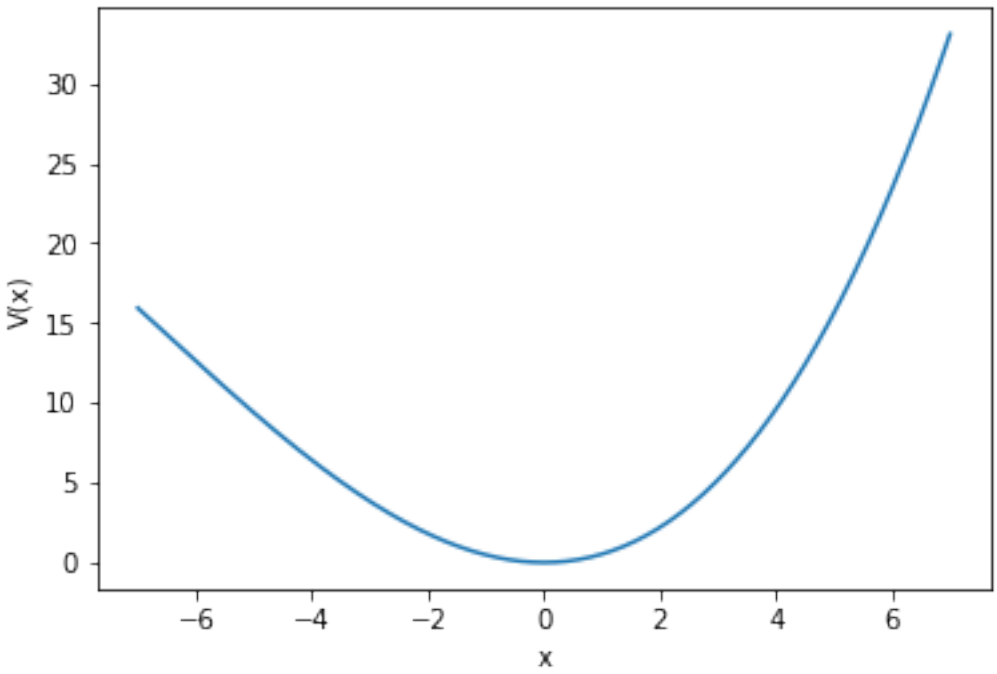
[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate
import scipy.optimize
from scipy.optimize import root as rootFinder
```

[2]:
```
class Schroedinger :

    def __init__(self) :
        self.hbar = 1.0                 # Planck's constant / 2pi
        self.m = 1.0                    # particle mass
        self.omega = 1.0                # oscillator frequency
        self.E = 0.0                    # current energy in search
        self.N = 500                    # number of lattice points = N+1
        self.x_left = -7.0              # left boundary
        self.x_right = 7.0              # right boundary
        self.h = (self.x_right - self.x_left) / self.N  # grid spacing
        self.xvals = np.linspace(self.x_left,self.x_right,self.N+1) # x value␣
↪for phi
        self.phi_left = np.zeros(self.N+1)  # wave function integrating from␣
↪left
        self.phi_right = np.zeros(self.N+1) # wave function integrating from␣
↪right
        self.phi = np.zeros(self.N+1)       # whole wave function
        self.i_match = 0                # Index of turning point
        self.x_match = 0.               # Turning point x value
        self.sign = 1                   # current sign used to make F(E)␣
↪continuous
        self.nodes = 0                  # current number of nodes in␣
↪wavefunction


    def V(self, x):                   # harmonic oscillator potential
        return 0.5 * self.m * self.omega**2 * (x**2 + (1/20)*x**3)
```

```
    def q(self, x):                    # Sturm-Liouville q function
        return 2 * self.m / self.hbar**2 * (self.E - self.V(x))


    def F(self, energy):               # eigenvalue at F(E) = 0

        # set energy needed by the q(x) function
        self.E = energy

        # find the right turning point
        self.i_match = self.N


        x = self.x_right               # start at right boundary
        while self.V(x) > self.E:          # in forbidden region
            self.i_match -= 1
            x -= self.h
            if self.i_match < 0:
                raise Exception("can't find right turning point")
        self.x_match = self.xvals[ self.i_match ]

        # integrate self.phi_left using Numerov algorithm
        self.phi_left[0] = 0.0
        self.phi_left[1] = 1.0e-10
        c = self.h**2 / 12.0           # constant in Numerov formula
        for i in range(1, self.i_match+1):
            x = self.x_left + i * self.h
            self.phi_left[i+1]  = 2 * (1 - 5 * c * self.q(x)) * self.phi_left[i]
            self.phi_left[i+1] -= (1 + c * self.q(x - self.h)) * self.
↪phi_left[i-1]
            self.phi_left[i+1] /= 1 + c * self.q(x + self.h)

        # integrate self.phi_right
        self.phi[self.N] = self.phi_right[self.N] = 0.0
        self.phi[self.N-1] = self.phi_right[self.N-1] = 1.0e-10
        for i in range(self.N - 1, self.i_match - 1, -1):
            x = self.x_right - i * self.h
            self.phi_right[i-1]  = 2 * (1 - 5 * c * self.q(x)) * self.
↪phi_right[i]
            self.phi_right[i-1] -= (1 + c * self.q(x + self.h)) * self.
↪phi_right[i+1]
            self.phi_right[i-1] /= 1 + c * self.q(x - self.h)
            self.phi[i-1] = self.phi_right[i-1]

        # rescale self.phi_left
        scale = self.phi_right[self.i_match] / self.phi_left[self.i_match]
```

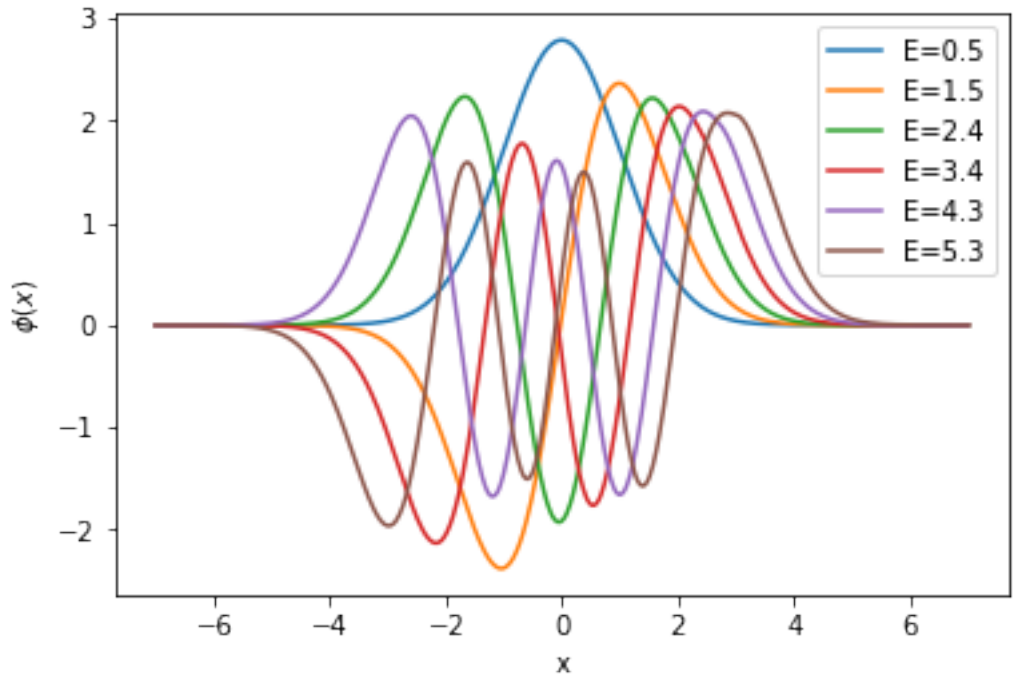
```
        for i in range(self.i_match + 2):
            self.phi_left[i] *= scale
            self.phi[i] = self.phi_left[i]

        # make F(E) continuous
        # count number of nodes in self.phi_left
        n = 0
        for i in range(1, self.i_match+1):
            if self.phi_left[i-1] * self.phi_left[i] < 0.0:
                n += 1

        # flip its sign when a new node develops

        if n != self.nodes:
            self.nodes = n
            self.sign = -self.sign

        return ( self.sign *
         ( self.phi_right[self.i_match-1] - self.phi_right[self.i_match+1] -
           self.phi_left [self.i_match-1] + self.phi_left[self.i_match+1] ) /
        (2 * self.h * self.phi_right[self.i_match]) )

    def normalize(self):
        norm = 0.0
        norm = np.sqrt( np.sum( self.phi**2 ) / self.N )
        self.phi = self.phi / norm
```

[3]:
```
schroedinger = Schroedinger()


x = schroedinger.x_left + schroedinger.h * np.linspace(0, schroedinger.N,␣
 ↪schroedinger.N+1)
vx = schroedinger.V(x)
plt.plot(x,vx)
plt.xlabel("x")
plt.ylabel("V(x)")
plt.show()
```



[4]:
```
# find the energy levels
E_max = 5.0
schroedinger.E = 0.1        # guess an E below the ground state
level = 0                   # level number
E_old = 0.0                 # previous energy eigenvalue


while True:                 # loop over levels

    # estimate next E and dE
    dE = 0.5 * (schroedinger.E - E_old)
    E_old = schroedinger.E
    schroedinger.E += dE

    res = rootFinder( schroedinger.F,
                                    x0=schroedinger.E)
      #                              x1=schroedinger.E+dE )
    schroedinger.E = res.x

    level += 1

    res = rootFinder( schroedinger.q,
                                    x0=schroedinger.x_left)
     #                              x1=schroedinger.x_match )
```

```
    x = res.x
    #swrite = '-: {0:4.2f} {1:4.2f}'.format( x, schroedinger.E )
    #print (swrite)

    res = rootFinder( schroedinger.q,
                                      x0=schroedinger.x_match)
      #                                x1=schroedinger.x_right )
    x = res.x
    #swrite = '+: {0:4.2f} {1:4.2f}'.format( x, schroedinger.E )
    #print (swrite)


    schroedinger.normalize()
    xvals = schroedinger.x_left + schroedinger.h * np.arange(schroedinger.N+1)

    plt.plot(xvals, schroedinger.phi, label="E=%2.1f" % (schroedinger.E) )

    if schroedinger.E >= E_max:           # we are done
        break
plt.legend()
plt.xlabel("x")
plt.ylabel(r"$\phi(x)$")
plt.show()
```



[5]:
```
%matplotlib notebook
# print the search function
schroedinger.E = 0.1
dE = 0.01
E_data = []
F_data = []
while schroedinger.E < E_max:
    E_data.append( schroedinger.E )
    F_data.append(  schroedinger.F(schroedinger.E) )
    schroedinger.E += dE

plt.plot(E_data,F_data)
plt.ylim(-10,10)
plt.xlabel("E")
plt.ylabel("F(E)")
plt.axhline(linewidth=1, color='k')
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[ ]:

schroedingerQuartic

March 2, 2021

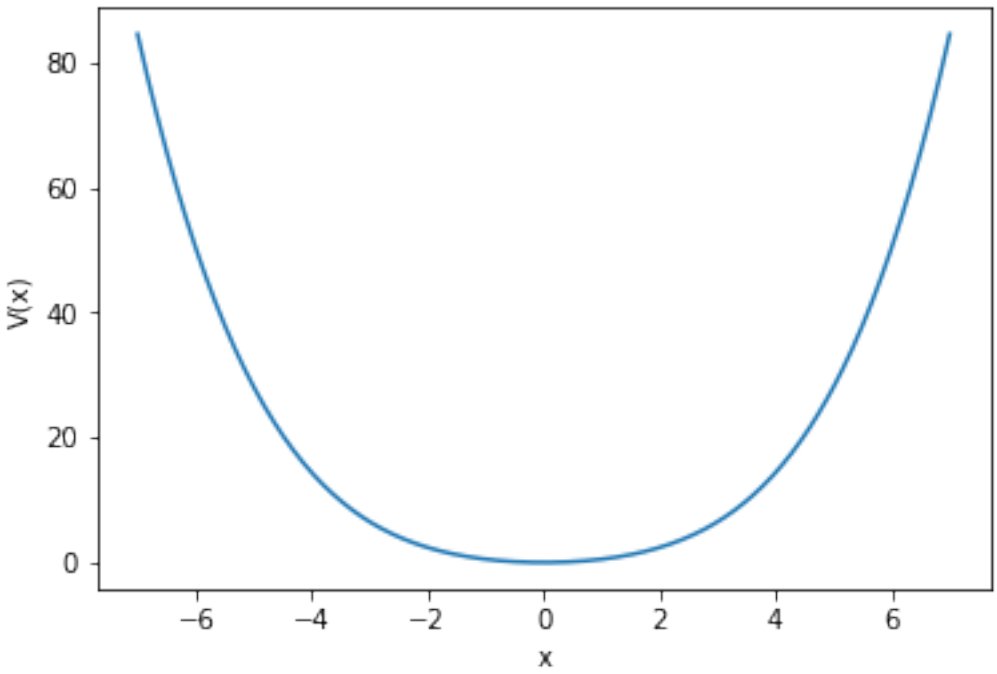
[1]:
```
import numpy as np
import matplotlib.pyplot as plt
import scipy.integrate
import scipy.optimize
from scipy.optimize import root as rootFinder
```

[2]:
```
class Schroedinger :

    def __init__(self) :
        self.hbar = 1.0                  # Planck's constant / 2pi
        self.m = 1.0                     # particle mass
        self.omega = 1.0                 # oscillator frequency
        self.E = 0.0                     # current energy in search
        self.N = 500                     # number of lattice points = N+1
        self.x_left = -7.0               # left boundary
        self.x_right = 7.0               # right boundary
        self.h = (self.x_right - self.x_left) / self.N  # grid spacing
        self.xvals = np.linspace(self.x_left,self.x_right,self.N+1) # x value␣
↪for phi
        self.phi_left = np.zeros(self.N+1)  # wave function integrating from␣
↪left
        self.phi_right = np.zeros(self.N+1) # wave function integrating from␣
↪right
        self.phi = np.zeros(self.N+1)       # whole wave function
        self.i_match = 0                 # Index of turning point
        self.x_match = 0.                # Turning point x value
        self.sign = 1                    # current sign used to make F(E)␣
↪continuous
        self.nodes = 0                   # current number of nodes in␣
↪wavefunction


    def V(self, x):                    # harmonic oscillator potential
        return 0.5 * self.m * self.omega**2 * (x**2 + (1/20)*x**4)


    def q(self, x):                    # Sturm-Liouville q function
```

```
        return 2 * self.m / self.hbar**2 * (self.E - self.V(x))


    def F(self, energy):              # eigenvalue at F(E) = 0

        # set energy needed by the q(x) function
        self.E = energy

        # find the right turning point
        self.i_match = self.N


        x = self.x_right              # start at right boundary
        while self.V(x) > self.E:          # in forbidden region
            self.i_match -= 1
            x -= self.h
            if self.i_match < 0:
                raise Exception("can't find right turning point")
        self.x_match = self.xvals[ self.i_match ]

        # integrate self.phi_left using Numerov algorithm
        self.phi_left[0] = 0.0
        self.phi_left[1] = 1.0e-10
        c = self.h**2 / 12.0          # constant in Numerov formula
        for i in range(1, self.i_match+1):
            x = self.x_left + i * self.h
            self.phi_left[i+1]  = 2 * (1 - 5 * c * self.q(x)) * self.phi_left[i]
            self.phi_left[i+1] -= (1 + c * self.q(x - self.h)) * self.
↪phi_left[i-1]
            self.phi_left[i+1] /= 1 + c * self.q(x + self.h)

        # integrate self.phi_right
        self.phi[self.N] = self.phi_right[self.N] = 0.0
        self.phi[self.N-1] = self.phi_right[self.N-1] = 1.0e-10
        for i in range(self.N - 1, self.i_match - 1, -1):
            x = self.x_right - i * self.h
            self.phi_right[i-1]  = 2 * (1 - 5 * c * self.q(x)) * self.
↪phi_right[i]
            self.phi_right[i-1] -= (1 + c * self.q(x + self.h)) * self.
↪phi_right[i+1]
            self.phi_right[i-1] /= 1 + c * self.q(x - self.h)
            self.phi[i-1] = self.phi_right[i-1]

        # rescale self.phi_left
        scale = self.phi_right[self.i_match] / self.phi_left[self.i_match]
        for i in range(self.i_match + 2):
            self.phi_left[i] *= scale
```

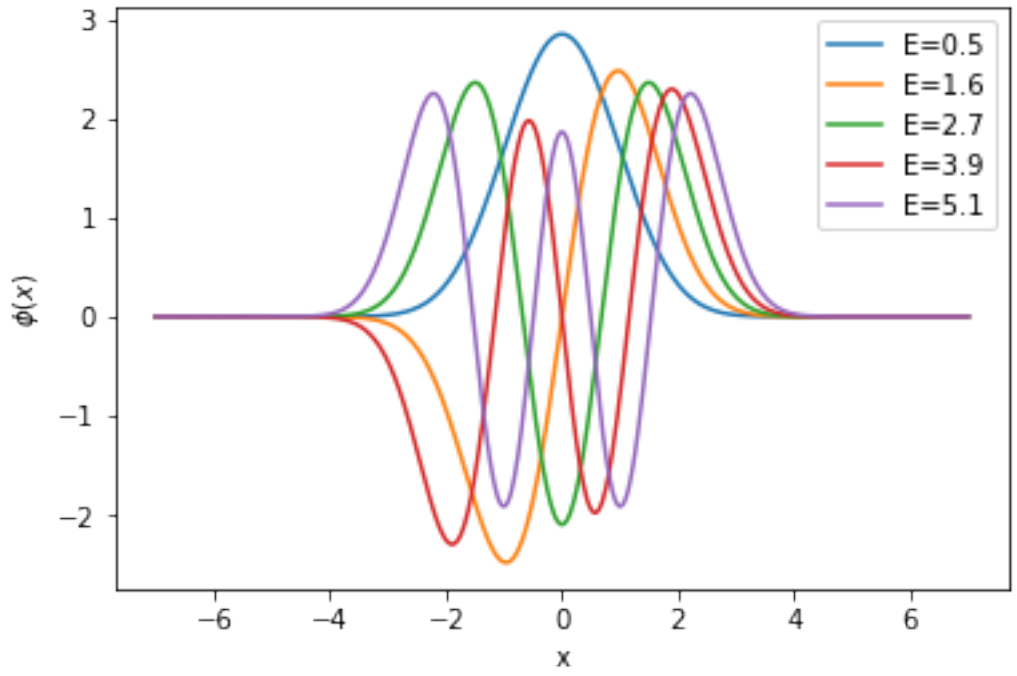
```
            self.phi[i] = self.phi_left[i]

        # make F(E) continuous
        # count number of nodes in self.phi_left
        n = 0
        for i in range(1, self.i_match+1):
            if self.phi_left[i-1] * self.phi_left[i] < 0.0:
                n += 1

        # flip its sign when a new node develops

        if n != self.nodes:
            self.nodes = n
            self.sign = -self.sign

        return ( self.sign *
         ( self.phi_right[self.i_match-1] - self.phi_right[self.i_match+1] -
           self.phi_left [self.i_match-1] + self.phi_left[self.i_match+1] ) /
        (2 * self.h * self.phi_right[self.i_match]) )

    def normalize(self):
        norm = 0.0
        norm = np.sqrt( np.sum( self.phi**2 ) / self.N )
        self.phi = self.phi / norm
```

[3]:
```
schroedinger = Schroedinger()


x = schroedinger.x_left + schroedinger.h * np.linspace(0, schroedinger.N,␣
 ↪schroedinger.N+1)
vx = schroedinger.V(x)
plt.plot(x,vx)
plt.xlabel("x")
plt.ylabel("V(x)")
plt.show()
```



[4]:
```
# find the energy levels
E_max = 5.0
schroedinger.E = 0.1        # guess an E below the ground state
level = 0                   # level number
E_old = 0.0                 # previous energy eigenvalue


while True:                 # loop over levels

    # estimate next E and dE
    dE = 0.5 * (schroedinger.E - E_old)
    E_old = schroedinger.E
    schroedinger.E += dE

    res = rootFinder( schroedinger.F,
                                    x0=schroedinger.E)
      #                              x1=schroedinger.E+dE )
    schroedinger.E = res.x

    level += 1

    res = rootFinder( schroedinger.q,
                                    x0=schroedinger.x_left)
     #                              x1=schroedinger.x_match )
```

```
    x = res.x
    #swrite = '-: {0:4.2f} {1:4.2f}'.format( x, schroedinger.E )
    #print (swrite)

    res = rootFinder( schroedinger.q,
                                     x0=schroedinger.x_match)
      #                               x1=schroedinger.x_right )
    x = res.x
    #swrite = '+: {0:4.2f} {1:4.2f}'.format( x, schroedinger.E )
    #print (swrite)


    schroedinger.normalize()
    xvals = schroedinger.x_left + schroedinger.h * np.arange(schroedinger.N+1)

    plt.plot(xvals, schroedinger.phi, label="E=%2.1f" % (schroedinger.E) )

    if schroedinger.E >= E_max:           # we are done
        break
plt.legend()
plt.xlabel("x")
plt.ylabel(r"$\phi(x)$")
plt.show()
```



[5]:
```
%matplotlib notebook
# print the search function
schroedinger.E = 0.1
dE = 0.01
E_data = []
F_data = []
while schroedinger.E < E_max:
    E_data.append( schroedinger.E )
    F_data.append(  schroedinger.F(schroedinger.E) )
    schroedinger.E += dE

plt.plot(E_data,F_data)
plt.ylim(-10,10)
plt.xlabel("E")
plt.ylabel("F(E)")
plt.axhline(linewidth=1, color='k')
plt.show()
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[ ]:

poisson

March 2, 2021

1 Poisson equation

We will now investigate solutions to Poisson's equation:

$$\nabla^2 V(\mathbf{r}) = \sum_i \frac{\partial^2 V}{\partial r_i^2} = -\alpha\rho(\mathbf{r})$$

for potential V and source density ρ. In the case of Coulomb's law, ρ is the charge density and V is the electric potential, with $\alpha = 1/\epsilon_0$.

We will investigate both **relaxation** and **Fourier** solutions

1.1 Relaxation solutions

As with BVPs, there are relaxation-based solutions in multiple dimensions for PDEs. For simplicity we will consider Dirichlet boundary conditions such that the potential vanishes along the boundary

$$V(\mathbf{r}_{\text{boundary}}) = 0$$

There are three methods we will investigate

- Jacobi iteration
$$V_{i,j}^{n+1} = \frac{1}{4}\left(V_{i+1,j}^n + V_{i-1,j}^n + V_{i,j+1}^n + V_{i,j-1}^n + h^2\rho_{i,j}\right)$$
- Gauss-Seidel iteration (G-S)
$$V_{i,j}^{n+1} = \frac{1}{4}\left(V_{i+1,j}^{n+1} + V_{i-1,j}^n + V_{i,j+1}^n + V_{i,j-1}^{n+1} + h^2\rho_{i,j}\right)$$
- Successive Over-Relaxation (SOR)
$$V_{i,j}^{n+1} = (1-\omega)V_{i,j}^n + \frac{\omega}{4}\left(V_{i+1,j}^{n+1} + V_{i-1,j}^n + V_{i,j+1}^n + V_{i,j-1}^{n+1} + h^2\rho_{i,j}\right)$$

where ω is the "over-relaxation" parameter and can be tuned for performance.

We will investigate the overall rates of convergence to a steady solution, as well as the overall computation time to a steady solution. We will see that the Jacobi solution has the worst performance per iteration, however, it can be vectorized, whereas the other two cannot! As such, despite it converging the slowest, it can achieve a target accuracy faster than the other two due to vectorization.

(There is a caveat here, the G-S and SOR implementations are in pure `python`, so cleverer tricks could be done with `C++` loops, however I will not go over a `swig` solution here).

[1]:
```
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
import time
```

[2]:
```
import numpy as np
import time
import cmath

class Poisson :
    def __init__ (self, L, rho_init, method, calcError = True):
        self.L = L                    # number of interior points in x and y
        self.omega = 1.88177          # over-relaxation parameter for L = 50
        self.N = L + 2                # interior plus two boundary points
        N=self.N
        self.V = np.zeros( (N, N) )   # potential to be found
        self.VNew = np.zeros( (N, N) )# new potential after each step
        self.h = 1.0 / (L + 1)        # lattice spacing assuming size in x and␣
 ↪y = 1
        self.nsmooth = 3              # smoothing steps for multigrid method
        self.rho = np.array( rho_init )
        self.method=method
        self.calcError = calcError
        self.error  = 0. if self.calcError else None
        self.errors = [] if self.calcError else None


    def __call__(self):
        return getattr(self,self.method)()

    def update(self):
        if self.calcError :
            self.error = np.sum( np.abs(self.V - self.VNew) ) / self.V.size
        self.V[:,:] = self.VNew[:,:]
        if self.calcError :
            self.errors.append(self.error)
            return self.error
        else :
            return None


    def Jacobi(self) :
        '''
```

```
        Jacobi algorithm for a single iterative step
        '''
        self.VNew = self.JacStep(self.VNew,self.V,self.h,self.rho)
        return self.update()

    def JacStep(self, u, v, h, rho):
        '''
        Jacobi algorithm on arbitrary inputs. Called once by "Jacobi".
        Called many times by "MultiGrid"
        '''
        u[1:-1,1:-1] = np.add.reduce([
            0.25*v[1:-1, 2:] ,
            0.25*v[1:-1, 0:-2],
            0.25*v[2:,   1:-1] ,
            0.25*v[0:-2, 1:-1] ,
            h**2 * rho[1:-1,1:-1]])
        return u


    def GaussSeidel(self):
        '''
        Gauss-Seidel algorithm for one iterative step
        '''
        self.VNew[:,:] = self.V[:,:]
        # perform Gauss-Seidel update
        for i in range(1, self.L+1):
            for j in range(1, self.L+1):
                self.VNew[i][j] = 0.25 * (self.VNew[i-1][j] + self.VNew[i+1][j]␣
↪+
                                          self.VNew[i][j-1] + self.
↪VNew[i][j+1] +
                                          self.h**2 * self.rho[i][j])
        return self.update()


    def SuccessiveOverRelaxation(self):
        '''
        Averages between G-S and J methods.
        '''

        # update even sites in red-black scheme
        for i in range(1, self.L+1):
            for j in range(1, self.L+1):
                if (i + j) % 2 == 0:
                    self.VNew[i][j] = (1 - self.omega) * self.V[i][j] + self.
↪omega / 4 * (
```

```
                                self.V[i-1][j] + self.V[i+1][j] + self.
↪V[i][j-1] +
                                self.V[i][j+1] + self.h**2 * self.rho[i][j] )

        # update odd sites in red-black scheme
        for i in range(1, self.L+1):
            for j in range(1, self.L+1):
                if (i + j) % 2 != 0:
                    self.VNew[i][j] = (1 - self.omega) * self.V[i][j] + self.
↪omega / 4 * (
                                self.VNew[i-1][j] + self.VNew[i+1][j] + self.
↪VNew[i][j-1] +
                                self.VNew[i][j+1] + self.h**2 * self.rho[i][j]␣
↪)
        return self.update()


    def FFT(self):
        '''
        Explicitly solves differential equation in Fourier domain
        '''

        # Make a grid to get indices
        x = np.arange(self.N)
        y = np.arange(self.N)
        xx,yy = np.meshgrid(x,y)
        # Construct root of unity
        i = 0. + 1.j
        W = cmath.exp( 2 * i * cmath.pi / float(self.N) )
        # Get the denominator using the grid
        d = 4.0 - np.power(W, xx ) - np.power(W, -xx) - np.power(W, yy) - np.
↪power(W, -yy)
        valid_d = (d > 0.0)
        # FFT of rho columns
        rhof = np.fft.fft(self.rho, axis=0)
        # FFT of rho rows
        rhof = np.fft.fft(rhof,axis=1)
        # Solve for V in Fourier domain
        Vf = np.divide( rhof * self.h**2, d, out=np.zeros_like(d,␣
↪dtype=complex), where=np.absolute(d)>0)
        # IFFT of V rows
        self.V = np.fft.ifft( Vf, axis=0)
        # IFFT of V columns
        self.V = np.fft.ifft( self.V, axis=1)
        return
```

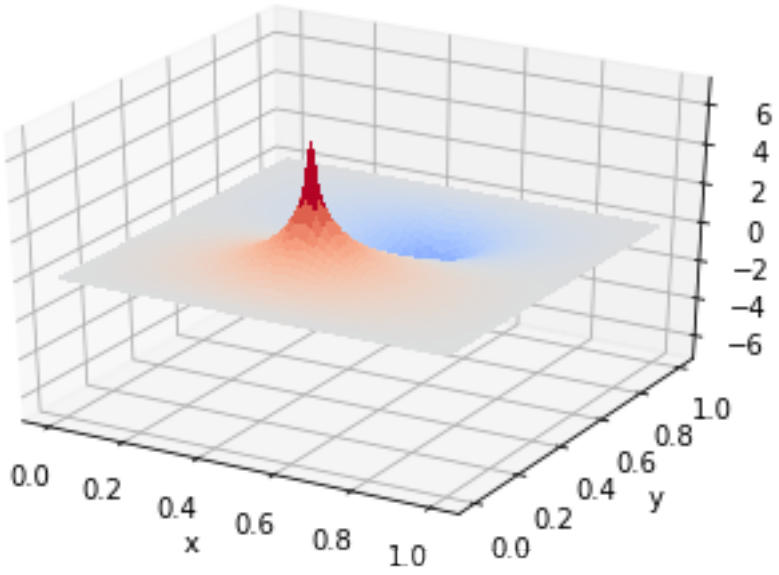
For the relaxation simulations, we will put a point charge in the center of a "grounded box" (Dirich-

let boundary conditions) and allow the solution to converge. We will first plot the wavefunction as determined by the Jacobi method (the others look almost identical).

[3]:
```
L = 50
N = L+2
q = 10.0  # point charge
i = N // 4 # center of lattice
j = (3*N)//4
k = N//2
h = 1/(L+1)
rho = np.zeros( (N,N))
rho[i,k] = q / h**2    # charge density
rho[j,k] = -q / h**2

nsteps = 100000
steps = np.arange(nsteps)
p = Poisson(L,rho, 'Jacobi')
for i in steps :
    p()
```

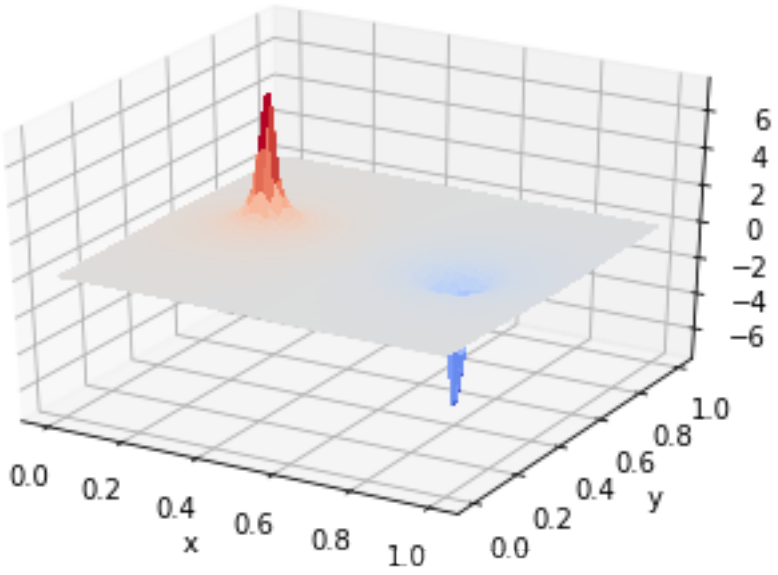
[7]:
```
# Define the axes
x = np.arange(0, p.h*(p.L+2), p.h)
y = np.arange(0, p.h*(p.L+2), p.h)
# Get the grid
X, Y = np.meshgrid(x, y)
# Set Z to the poisson V[i][j]
Z = np.array( p.V )

fig = plt.figure(1)
ax = fig.gca(projection='3d')
sur = ax.plot_surface( X, Y, Z, rstride=1, cstride=1, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False )
plt.xlabel("x")
plt.ylabel("y")
```

[7]:
```
Text(0.5, 0, 'y')
```



[5]:
```
k = 0.008314463
Z1 = np.zeros((N,N))
Z1 = k*q/(np.sqrt((X-0.25)**2+(Y-0.5)**2))-k*q/(np.sqrt((X-0.75)**2+(Y-0.5)**2))
fig1 = plt.figure(2)
ax1 = fig1.gca(projection='3d')
sur1 = ax1.plot_surface( X, Y, Z1, rstride=1, cstride=1, cmap=cm.coolwarm,
                         linewidth=0, antialiased=False )
plt.xlabel("x")
plt.ylabel("y")
```

[5]:
```
Text(0.5, 0, 'y')
```



[ ]:

1.1.1 Convergence versus iteration

Here we will compute the convergence (the difference between successive evaluations of the potential) as a function of the number of iterations.

We see that Jacobi performs worst, G-S is second, and SOR is the best.

[ ]:
```
methods = ["Jacobi", "GaussSeidel", "SuccessiveOverRelaxation"]
solvers = []
times = np.zeros( len(methods))
for i,method in enumerate(methods) :
    p = Poisson(L,rho, method)
    err = []
    t1 = time.perf_counter()
    for j in steps :
        ierr = p()
    t2 = time.perf_counter()
    solvers.append(p)
    times[i] = (t2-t1)
    print("Method %40s processed %d steps in %6.5f  s" % (method, len(steps),␣
 ↪times[i]) )
```

[ ]:
```
fig = plt.figure(2)
for i,method in enumerate(methods) :
    plt.plot(steps, solvers[i].errors, label=method )
plt.title("Convergence of algorithms")
```

```
plt.xlabel("Iteration")
plt.ylabel("Error")
plt.legend()
plt.show()
```

1.1.2 Performance in time

We now investigate how quickly it takes each algorithm to reach a given precision (in this case, one part per million). Due to the vectorization of the Jacobi algorithm, it is able to acheive the target precision faster than both G-S and SOR, despite requiring more iterations.

[ ]:
```
solvers = []
acc = 1e-6
times = np.zeros( len(methods))
for i,method in enumerate(methods) :
    p = Poisson(L,rho, method)
    err = []
    ierr = 9999.
    t1 = time.perf_counter()
    while ierr > acc :
        ierr = p()
    t2 = time.perf_counter()
    solvers.append(p)
    times[i] = (t2-t1)
    print("Accuracy %6.2e reached in %6.5f  s" % (acc, times[i]) )
```

1.1.3 Take-home message

The message here is that there is now truly a trade-off between complex algorithms with deep loops, and simple algorithms that can be easily vectorized. Depending on the application, one or another approach may be favored. You should be aware of this fact as you go forward in your career path!

1.2 FFT Solution

Similarly to your courses, you can also numerically solve PDEs in the frequency domain instead of the time domain using Fourier analysis. In this case, we can also look at FFTs. For simplicity we will impose boundary conditions such that the limit of the potential tends to zero at infinite radius.

Since the FFT is a linear operation, we can perform the FFT and inverse FFT separately for the rows and columns of the 2d distribution. The Fourier components are

$$\tilde{V}_{mn} = \frac{h^2 \tilde{\rho}_{mn}}{4 - W^m - W^{-m} - W^n - W^{-n}}$$

where $W = \exp(2i\pi/N)$ is a root of unity.

In the end, it is often much faster to compute FFT-based solutions, especially for large values of N. In an apples-to-apples comparison, the fastest relaxation technique takes around 300 times longer for N=14 interior points (N=16 total points).

[ ]:
```
L = 14
N = L+2
q = 10.0                # point charge
h = 1/(L+1)
rho = np.zeros( (N,N))
rho[N // 2,N // 2] = q / h**2    # charge density

p = Poisson(L,rho, method="FFT")
t1 = time.perf_counter()
p()
t2 = time.perf_counter()

print("Solution computation took %6.5f s"% (t2-t1))
```

[ ]:
```
# Define the axes
x = np.arange(0, p.h*(p.L+2), p.h)
y = np.arange(0, p.h*(p.L+2), p.h)
# Get the grid
X, Y = np.meshgrid(x, y)
# Set Z to the poisson V[i][j]
Z = np.array( p.V.real )

fig = plt.figure(1)
ax = fig.gca(projection='3d')
sur = ax.plot_surface( X, Y, Z, rstride=1, cstride=1, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False )
plt.xlabel("x")
plt.ylabel("y")
```