PHYS 501: Mathematical Physics I

Homework #2

Prakash Gautam

April 18, 2018

1. An electrical network consists of N interconnected nodes. Each pair of nodes (i, j) is connected by a resistor of resistance $R_{ij} = \min(i, j) + 2\max(i, j)$, for $i, j = 1, ..., N$. Let V_i be the electrical potential of node i, and choose the zero level of potential to set $V_1 = 0$. Then Kirchhoffs laws for the other nodes in the network can be conveniently written as

$$\sum_{j=1}^{N} \frac{V_j - V_i}{R_{ij}} = I_i$$

, for $i = 2, ..., N$ where I_i is the current flowing from node i to some external circuit. Suppose N = 100 and the external connection is such that current flows out of node 2 and back into node 1, so $I_1 = 1$, $I_2 = 1$, and $I_i = 0$ for $i > 2$. By solving the above $(N1)$-dimensional matrix equation, calculate the total resistance between nodes 1 and 2.

Solution:
Expanding the i^{th} current value

$$\begin{aligned} I_1 &= \frac{V_2 - V_1}{R_{12}} + \frac{V_3 - V_1}{R_{13}} + \cdots + \frac{V_N - V_1}{R_{1N}} \\ &= -\left(\frac{1}{R_{12}} + \frac{1}{R_{12}} + \cdots + \frac{1}{R_{1N}}\right) V_1 + \frac{V_2}{R_{12}} + \frac{V_3}{R_{13}} + \cdots + \frac{V_N}{R_{1N}} \end{aligned}$$

Smililarly expanding all others we get the pattern. So the equivalent matrix would be.

$$\begin{pmatrix} I_1 \\ I_2 \\ I_3 \\ \vdots \\ I_N \end{pmatrix} = \begin{pmatrix} -\left(\frac{1}{R_{12}} + \frac{1}{R_{12}} + \cdots + \frac{1}{R_{1N}}\right) & \frac{1}{R_{12}} & \frac{1}{R_{13}} & \cdots & \frac{1}{R_{1N}} \\ \frac{1}{R_{21}} & -\left(\frac{1}{R_{21}} + \frac{1}{R_{23}} + \cdots + \frac{1}{R_{2N}}\right) & \frac{1}{R_{23}} & \cdots & \frac{1}{R_{2N}} \\ \vdots & \cdots & \ddots & & \vdots \\ \frac{1}{R_{N1}} & \frac{1}{R_{N2}} & \frac{1}{R_{N3}} & \cdots & -\left(\frac{1}{R_{N1}} + \frac{1}{R_{N2}} + \cdots + \frac{1}{R_{NN}}\right) \end{pmatrix} \begin{pmatrix} V_1 \\ V_2 \\ V_3 \\ \vdots \\ V_N \end{pmatrix}$$

This is the Matrix equation relating the Ohm's law where $I = \frac{V}{R}$. The $1/R$ Matrix is $N \times N$ matrix. But since V_1 is zero and I_1 is known we can eleminate the first row and column of $1/R$ Matrix to get $(N-1)$ Dimensional Matrix

```python
import numpy as np
from numpy import linalg as LA

num_nodes = 100
#Construct the matrix
N = num_nodes
def Res(i,j):
    return min(i,j) + 2*max(i,j)

# Construct the R matrix with every elements except the diagonals
# int(i!=j) returns 1 for non diagonal places and 0 for diagonal
# places so all the diagonal elements
R=np.array([(int(i!=j)*(1/Res(i,j))) for i in range(1,N+1) for j in range(1,N+1)]).reshape
    (N,N)

# Since  in the matrix we see that the  diagonal
# elements are  simply negative sum of all other
# elements in the matrix, we sum them to get the
# diagonal elements and  put them back to matrix
Diag = [-sum(R[i]) for i in range(N)]
np.fill_diagonal(R,Diag)

#Deleting the first row and colums
R = R[1:,1:]

#Initializing the Current column matrix to zero
I = np.zeros(N-1).reshape(N-1,1)

# The first value of current I_2 is indexed at zero
# so I[0] means the I_2 which is 1
I[0] = 1

# Solving the matrix equation to get the potentials at all nodes
V =LA.solve(R,I).reshape(N-1)

# Since our zero index in program is index 2 for current and voltages
# the voltage at node 2 is V[0]
V2 = V[0]
print('The  electric potential  at  node 2 is {:.4}V'.format(V[0]))

# The current in the whole circuit is 1A so the equivalent resistance
# can be calculated by Ohm's law. R_eq = (V1 - V2) / I, since V1 = 0
# The total resistance of circuit is just R_eq  = -V2/1 = -V2
print('The equivalent Resistance of circuit is {:.4}'.format(-V[0]))
```

```
The  electric potential at   node 2 is -0.9442V
The equivalent Resistance of circuit is 0.9442Ω
```

□

2. The data file hw2.2.dat on the course Web page contains (hypothetical) experimental data on the measurement of a function y(x). The N data points are arranged, one measurement per line, in the format

```
xi          yi (measured)          σi
```

where σ_i is an estimate of the uncertainty in the i-th measurement. It is desired to find the least-square fit to the data by polynomials of the form

$$y(x) = \sum_{j=1}^{N} a_j x^{j-1}$$

for specified values of m, by minimizing the quantity

$$\chi^2 = \sum_{i=1}^{N} \left[\frac{y_i - \sum_{j=1}^{m} a_j x^{j-1}}{\sigma_i} \right]^2$$

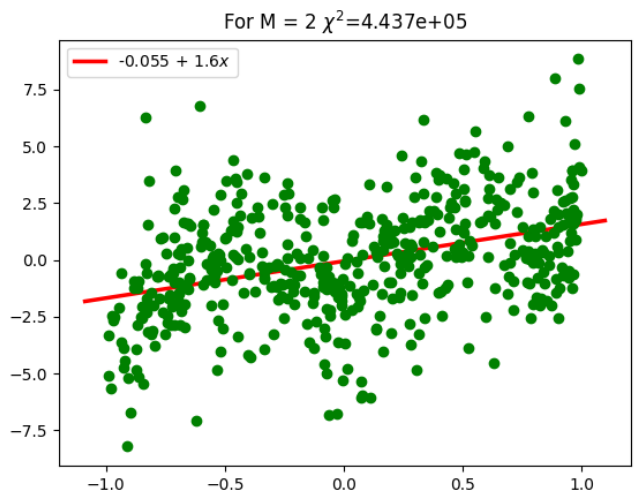
As discussed in class (and in Numerical Recipes, pp 671676), write down the overdetermined design matrix equation that results from writing $y(x_i) = y_i$,

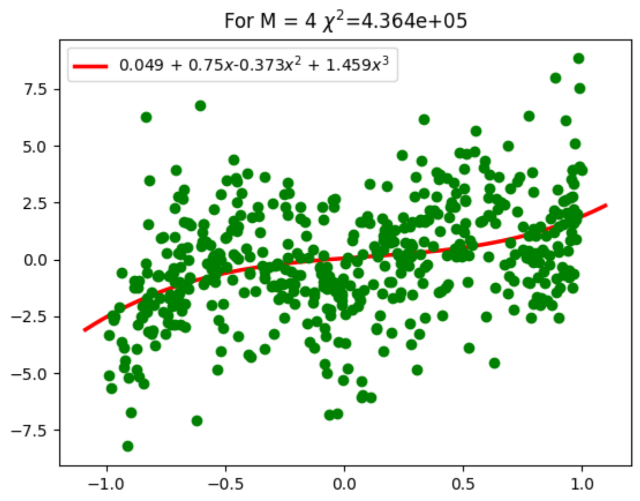
$$Aa = b$$

, where $A_{ij} = \frac{x_i^{j1}}{\sigma_i}$, $b_i = \frac{y_i}{\sigma_i}$ (so the measurement undertainties are included in each row), and a is the vector of unknown coefficients. Solve this system using singular value decomposition (svdcmp in Numerical Recipes, svd in Python or Matlab) to obtain the best fitting polynomial for each of the cases m = 2, 4, 7, and 13. For each m, give the values of a_j and χ^2, and plot the data and the best fit on a single graph.
Solution:

```
#!/usr/bin/env python3

import numpy as np
import numpy.linalg as LA
import matplotlib.pyplot as plt
from matplotlib2tikz import save as tikz_save

class LeastSquare():
    mlist = [2,4,7,13]#6,7,8,9,13]
    pltcnt = len(mlist)
    pltprm = 221
    clr = 0.1 #clearence

    datafile = './data/hw2.2.dat'
    slc = 500  # slice length to test for fewer data points
    epsilon = 1e-3 # zero threshold for svd inverting

    def __init__(self):
        self.readfile()

    def readfile(self):
        read = np.genfromtxt(self.datafile)
        self.x   = read[:,0]; self.x = self.x[:self.slc]
        self.y   = read[:,1]; self.y  = self.y[:self.slc]
        self.sd = read[:,2]; self.sd = self.sd[:self.slc]
        self.err  = self.sd.reshape(self.slc,1)
        return read

    def construct_A(self,M):
        #Reform x shape to column shape
        xcol = self.x.reshape(self.slc,1)
        # Same for error values
        err  = self.sd.reshape(self.slc,1)

        # initialize the first column of the A matrix
        A = (np.zeros(self.slc) + 1).reshape(self.slc,1)/err
        for m in range(1,M):
            A = np.append(A,xcol**m/err,1)

        return np.matrix(A)

    def construct_b(self):
        coly = self.y.reshape(len(self.y),1)
        return np.matrix(coly/self.err)

    def get_svd_inverse(self,M):
        U,W,V = LA.svd(M)
        WI_star = []
        for wi in W:
            if wi < self.epsilon:
                WI_star.append(0)
            else:
                WI_star.append(1/wi)
        WI = np.diag(WI_star)
        r,c = M.shape
        inc_prm = r-c
        cm = np.matrix(np.zeros(inc_prm*c).reshape(c,inc_prm))
        WI = np.hstack([WI,cm])
        return V.T*WI*U.T

    def get_polynomial_values(self,aj,x):
        y = np.zeros(len(x))
        for j in range(len(aj)):
            y += aj[j]*x**j #since our index starts at 0, it works
        return y

    def get_coefficient(self,m):
        A = self.construct_A(m)
        b = self.construct_b()
        AI = self.get_svd_inverse(A)
        a = AI*b
        #print('a shape is {}'.format(a.shape))
        ar = np.squeeze(np.asarray(a))
        return ar

    def get_chi_sq(self,true,predicted,error):
        return np.sum(((true-predicted)/error)**2)

    def get_legend(self,aj):
        j = 2
        lg = r'{:.2} + {:.2}$x$'.format(aj[0],aj[1])
        for a in aj[j:]:
            sgn = ' + '
            if a < 0: sgn = ''
            t = sgn + r'{:.4}$x^'.format(a)+'{'+'{}'.format(j)+'}$'
            lg += t
            j += 1

        return lg



    def draw_graph(self):
        cnt = 0
        for m in self.mlist:
            #plt.subplot(self.pltprm+cnt); cnt+=1

            aj = self.get_coefficient(m)
            #print('ar was ',ar)
            x = np.linspace(1.1*min(self.x),1.1*max(self.x),500)
            y = self.get_polynomial_values(aj,x)

            prediction = self.get_polynomial_values(aj,self.x)

            chisq = self.get_chi_sq(self.y,prediction,self.err)

            plt.plot(x,y,'r',linewidth='2.5',label=self.get_legend(aj))
            plt.legend()
            plt.plot(self.x,self.y,'go')
            plt.title('For M = {} $\chi^2$={:.3e}'.format(m,chisq))
            plt.savefig('M{}.png'.format(m))
            plt.show()

        plt.suptitle('M degree polynomial least square fit by SVD')
        #tikz_save('Least square plots.tex',figurewidth="14cm",figureheight="9cm")
        plt.show()
```

```
if __name__ == '__main__':
    LS = LeastSquare()
    LS.draw_graph()
```



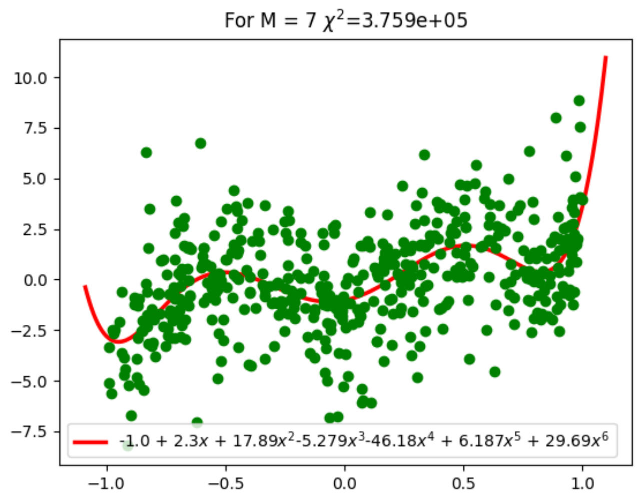


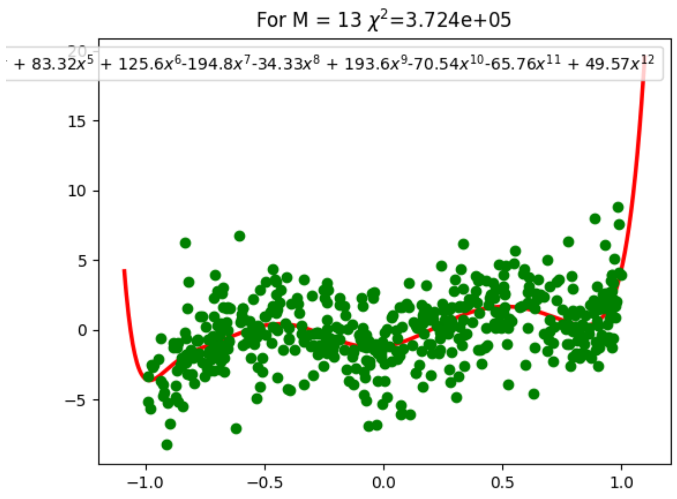
The a_j values are shown as polynomial coefficient in the legend of each plot and the χ^2 values are given at the top of each graph. In each case the continuous line is the fitted polynomial and the scattered dots are the values read from file. □

3.

Solution:
This Question was mostly solved by use of Sympy package in python.

```
import numpy as np
import numpy.linalg as LA

from scipy.integrate import quad

import sympy as smp
import sympy.functions as smf
import sympy.physics.quantum.constants as qc
```





The function $\phi_n(x)$ was defined to get any order function

```
def fai(self,n):
    x = self.x; b = self.b
    of = smp.Rational(1,4)
    oh = smp.Rational(1,2)
    A = (b**2/smp.pi)**of*1/(smp.sqrt(2**n*smf.factorial(n)))
    return (smp.exp(-oh*b**2*x**2)*smf.hermite(n,b*x)) * A
```

So for example this function gives for $fai(n)$ as.

$$\frac{2^{-\frac{p}{2}}\sqrt{|\beta|}}{\sqrt[4]{\pi}\sqrt{p!}}e^{-\frac{\beta^2x^2}{2}}H_p\left(\beta x\right)$$

Then the hamiltonial operator function is defined as.

```
def H(self,psi):
   hf = smp.Rational(1,2)
   x = self.x; m = self.m; hcut = self.hcut
   hctm = -hf*(hcut**2)/(m)
   return hctm * smp.diff(psi,x,2) + self.V(x)*psi
```

We wish to approximate the energy eigenfunctions of a one-dimensional square well by expanding them in terms of a *finite* (N-dimensional) subset of harmonic oscillator wavefunctions. The square well is defined by the potential

$$V(x) = \begin{cases} 0 & (|x| < a)\,, \\ V_0 & (|x| > a)\,. \end{cases}$$

The harmonic oscillator potential is $V_{ho}(x) = \frac{1}{2}kx^2$, where we will take $k = 2V_0/a^2$ here. As discussed in class, solving the problem entails diagonalization (e.g. using the Python function `eigvals` or the *Numerical Recipes* functions `tred2` and `tqli`) of the Hamiltonian matrix $H = (h_{nm})$, where

$$h_{nm} = \langle n|H|m\rangle = \int dx\, \phi_n^*(x) \left[-\frac{\hbar^2}{2m}\frac{d^2}{dx^2} + V(x) \right] \phi_m(x)$$

and $\phi_n(x)$ is the n-th harmonic oscillator wavefunction:

$$\phi_n(x) = \left(\frac{\beta^2}{\pi}\right)^{1/4} \frac{1}{\sqrt{2^n n!}}\, e^{-\frac{1}{2}\beta^2 x^2} H_n(\beta x)\,,$$

with $\beta^4 = mk/\hbar^2$.

Use the recurrence relations given in Riley & Hobson, p. 373, to generate the H_n, and the differential relations (same page) along with the trapezoidal rule, where needed, to compute the matrix elements h_{nm}.

Hence, by diagonalizing the matrix H, determine the first (and only) two energy levels E_0 and E_1 of a square well with $V_0 a^2 = 2\hbar^2/m$, for three different values of N: (a) use $\phi_0, \ldots, \phi_4$ as a basis ($N = 5$); (b) use $\phi_0, \ldots, \phi_9$ ($N = 10$); and (c) use $\phi_0, \ldots, \phi_{19}$ ($N = 20$). In each case, compare your answers with the exact values

$$E_0 = 0.53\frac{\hbar^2}{ma^2}\,, \quad E_1 = 1.80\frac{\hbar^2}{ma^2}\,.$$

Writing $A_p = \frac{2^{-\frac{p}{2}}\sqrt{|\beta|}}{\sqrt[4]{\pi}\sqrt{p!}}$ and after operating $\phi_p(x)$ by Hamiltonian operator and using the recurrence relation we get.

$$\hat{H}\phi_p(x) = \frac{A_p e^{-\frac{\beta^2 x^2}{2}}}{2m} \left(-\beta^2\hbar^2 \left(\beta^2 x^2 H_p(\beta x) - 4\beta p x H_{p-1}(\beta x) + 4p(p-1) H_{p-2}(\beta x) - H_p(\beta x) \right) + kmx^2 H_p(\beta x) \right)$$

If we evaluate the functions operated by at numeric value 0.

$$\hat{H}\phi_0(x) = \frac{A_p e^{-\frac{\beta^2 x^2}{2}}}{2m} \left(-\beta^2\hbar^2 \left(\beta^2 x^2 - 1 \right) + kmx^2 \right)$$

Evaluating function $\phi_0(x)$ we get.

$$\phi_0(x) = A_q e^{-\frac{\beta^2 x^2}{2}}$$

Now to get the hamiltonian matrix element we do.

$$h_{00} = \int_{-a}^{a} \phi_0^*(x) \hat{H} \phi_0(x) dx$$

Putting $a = 1, m = 1$ and $\hbar = 1$ to work in the Energy units of $\frac{\hbar^2}{ma^2}$ we get $k = 4 \quad \beta = 4^{1/4}$. Evaluating the integrals at these values we get. $h_{00} = 0.9544$ We can construct the matrix similarly for every value of p and q for the dimension given. Getting eigenvalues from the constructed matrix gives the Energy level in the units of $\frac{\hbar^2}{ma^2}$

For N = 5

$$\begin{bmatrix} 0.954500 & 0.000000 & -0.763548 & 0.000000 & -0.396751 \\ 0.000000 & 2.215608 & 0.000000 & -2.468673 & 0.000000 \\ -0.152710 & 0.000000 & 2.072950 & 0.000000 & -3.085998 \\ 0.000000 & -1.058003 & 0.000000 & 2.146257 & 0.000000 \\ -0.044083 & 0.000000 & -1.714443 & 0.000000 & 3.164405 \end{bmatrix}$$

The eigen values for this matrix is:

$$[0.109385071685, \quad 0.564434747324, \quad 1.09686038844, \quad 3.79742982845, \quad 4.98561012119]$$

Which means the first two energy level are

$$E_0 = 0.109\frac{\hbar^2}{ma^2} \quad E_1 = 0.564\frac{\hbar^2}{ma^2}$$

For N = 10

$$E_0 = 7.24 \times 10^{-7}\frac{\hbar^2}{ma^2} \quad E_1 = 1.102 \times 10^{-5}\frac{\hbar^2}{ma^2}$$

The complete program is □

```
#!/usr/bin/env python3

import numpy as np
import numpy.linalg as LA

from scipy.integrate import quad

import sympy as smp
import sympy.functions as smf

class EnergyLevels():

    #define constants
    ca = 1 # potetial well half width = 1
    cm = 1 # mass
    chc = 1 #hbar = 1
    cv0 = 2*chc**2/(cm*ca**2) # constanv V0 inferred from conditions
    ck = 2*cv0/(ca**2) # harmonic oscillator constant
    cb = (cm*ck/(chc**2))**(1/4.) # beta parameter.

    dimlist = [5,10,20] # dimension list
    #define vars and consts
    b,fi,k,m,p,q,x = smp.symbols('beta,phi,k,m,p,q,x',real=True)

    #special variables
    hcut = smp.symbols('hbar',real=True)
    ndim = 5
    lim = 1

    # constants substution dictionary. This should only affect the
    # scale of the output value.
    subd = {b:cb,hcut:chc,k:ck,m:cm}

    def __init__(self):
        pass

    def V(self,x): #potential function
```

```
        k = self.k
        return smp.Rational(1,2)*k*x**2

    def fai(self,n):
        x = self.x; b = self.b
        of = smp.Rational(1,4)
        oh = smp.Rational(1,2)
        A = (b**2/smp.pi)**of*1/(smp.sqrt(2**n*smf.factorial(n)))
        return (smp.exp(-oh*b**2*x**2)*smf.hermite(n,b*x)) * A

    def H(self,psi):
        hf = smp.Rational(1,2)
        x = self.x; m = self.m; hcut = self.hcut
        hctm = -hf*(hcut**2)/(m)
        return hctm * smp.diff(psi,x,2) + self.V(x)*psi

    def getf(self,p,q):
        php = self.fai(p)
        phq = self.fai(q)
        fx = php * self.H(phq)
        return fx


    def geth(self,p,q):
        fx = self.getf(p,q)
        fx = fx.subs(self.subd)

        hpq = self.integrate(fx,-self.ca,self.ca)
        return  hpq

    def integrate(self,fx,a,b):
        flx = smp.utilities.lambdify(self.x,fx)
        v,e = quad(flx,-self.lim,self.lim)
        return v

    def construct_H(self,n):
        H = np.zeros(n*n).reshape(n,n)

        for v in [(p,q) for p in range(n) for q in range(n)]:
            p,q = v
            H[p][q] = self.geth(p,q)

        return np.matrix(H)

    def get_energy(self):
        for m in self.dimlist:
            hmn = self.construct_H(m)
            evl,evc = LA.eig(hmn)
            evl = np.squeeze(evl)
            evl = sorted(evl)
            print(evl)
            #print('For N = {} E0 = {:.3e} and E1 = {:.2e}'.format(m,evl[0],evl[1]))


if __name__ == '__main__':
    EL = EnergyLevels()
    EL.get_energy()
```