Homework 3

Varun Nair

February 14, 2019

```
In [1]: %precision %g
        %matplotlib inline
        %config InlineBackend.figure_format = 'retina'
```

```
In [2]: from math import sqrt, pi, sin, cos, exp, inf, factorial, tan
        from cmath import exp as cexp
        import numpy as np
        from scipy import constants as C
        from scipy import integrate
        import matplotlib.pyplot as plt

        #from IPython.display import set_matplotlib_formats
        #set_matplotlib_formats('png', 'pdf')
```

1 CP 5.3 Gaussian error function

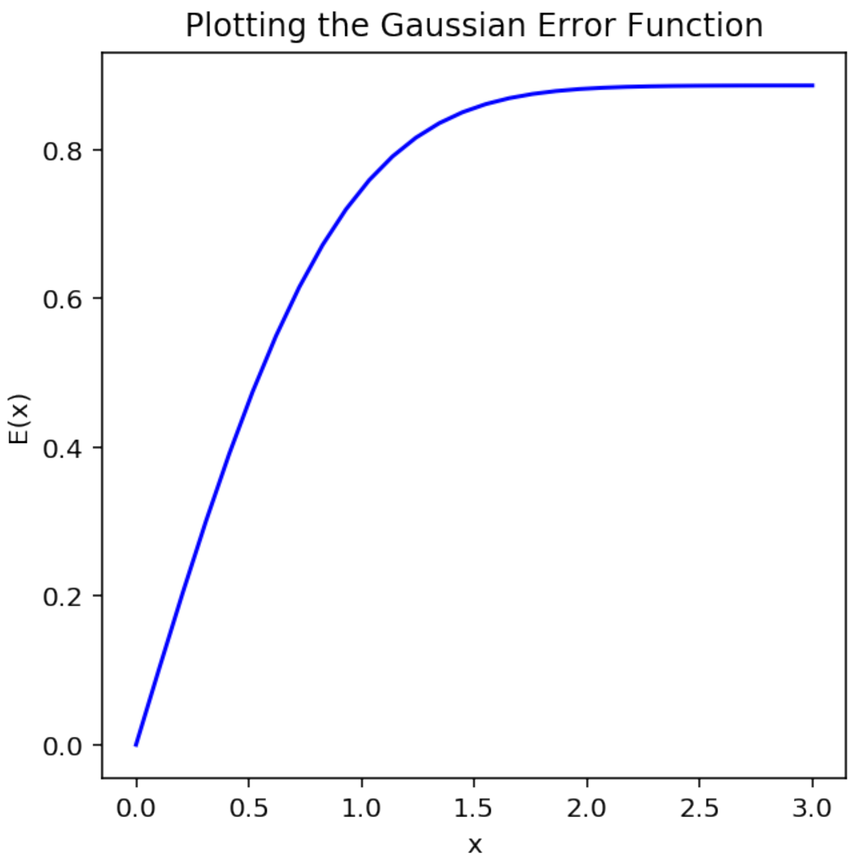
The function $E(x) = \int_0^x e^{-t^2}\, dt$ must be solved numerically. In this problem, I'll employ the trapezoidal rule

$$I(a,b) = h\left(\frac{1}{2}(f(a)+f(b)) + \sum_k^{N-1} f(a+kh)\right)$$

to find the values for $E(x)$ given $x \in [0,3]$ with increments of 0.1.

```
In [3]: #defining integrand function
        def f(t):
            return exp(-(t**2))

        #defines a trapezoidal rule based on the one listed above
        def E(x, N):
            '''x is the upper limit for the integral
               N is the number of slices used for integration'''

            h = x / N
            s1 = f(0)
            s2 = f(x)
            s3 = 0
            for k in range(N):
                s3 += f(k*h)

            return ((0.5 * (s1 + s2)) + s3) * h

        #calculating E(x) for 0-3 by 0.1
        sols = np.zeros([30,2], dtype=float)
        sols[:,0] = np.linspace(0,3,30)

        for i in range(np.shape(sols)[0]):
            sols[i,1] = E(sols[i,0], 100000)
```

```
In [4]: #created a new cell for plotting, so
        #for loop doesn't run every time

        fig, ax = plt.subplots(1, 1, figsize = (5, 5))
        plt.plot(sols[:,0],sols[:,1], 'k-', color='blue')
        plt.title("Plotting the Gaussian Error Function")
        plt.xlabel("x")
        plt.ylabel("E(x)")
        plt.show()
```



2 CP 5.4 The diffraction limit of a telescope

The circular diffraction pattern for a star in a telescope is given by

$$I(r) = \left(\frac{J_1(kr)}{kr}\right)^2.$$

This value makes use of Bessel functions $J_m(x)$. The Bessel functions are given by

$$J_m(x) = \frac{1}{\pi}\int_0^\pi \cos(m\theta - x\sin\theta)\, d\theta.$$

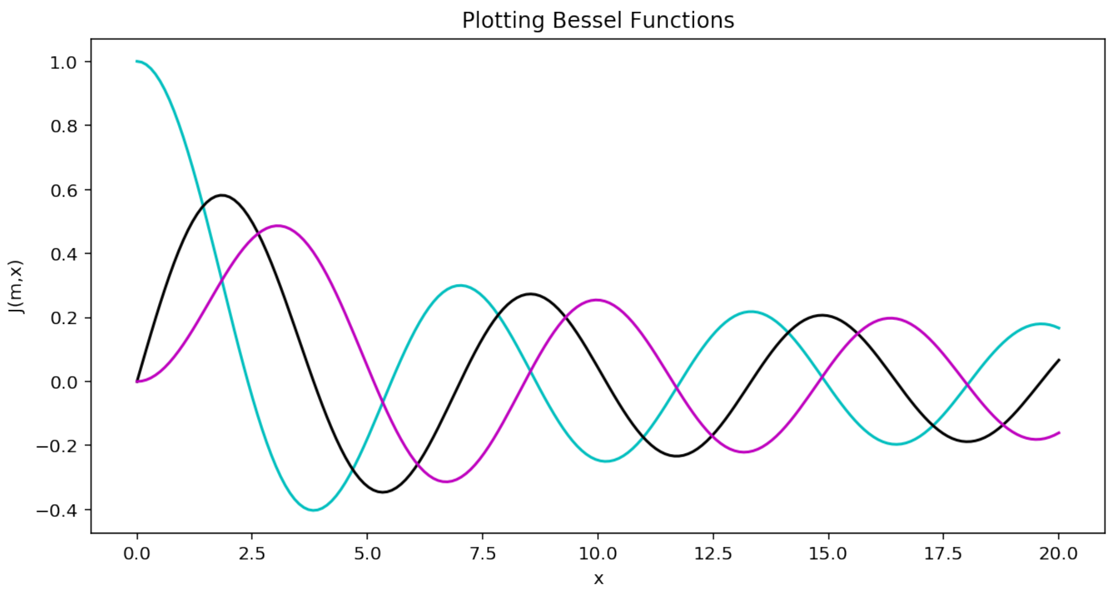
We can evaluate this integral numerically and thus find the value of any given Bessel function via Simpson's rule. The simplified form of Simpson's rule is

$$I = \frac{1}{3}h\left(f(a) + f(b) + 4\sum_{k=\text{odd}} f(a+kh) + 2\sum_{k=\text{even}} f(a+kh)\right).$$

```
In [5]: def J(m,x):
            '''Evaluates the Bessel function for order m and x
            with N = 1000 slices for Simpsons Rule'''

        #defines integrand
        def f(m, x, theta):
            return cos(m * theta - x * sin(theta))

        N = 1000

        #always this b/c bounds and slices are set
        h = pi / N

        s1 = f(m, x, 0)
        s2 = f(m, x, pi)
        s3 = 0
        s4 = 0

        for k in range(1,N,2):
            s3 += f(m, x, k*h)
        for k in range(2,N,2):
            s4 += f(m, x, k*h)

        s = s1 + s2 + 4*s3 + 2*s4
        I = (1/3) * h * s

        return I / pi


    small = 0.000001
    J(1, small) / small
```

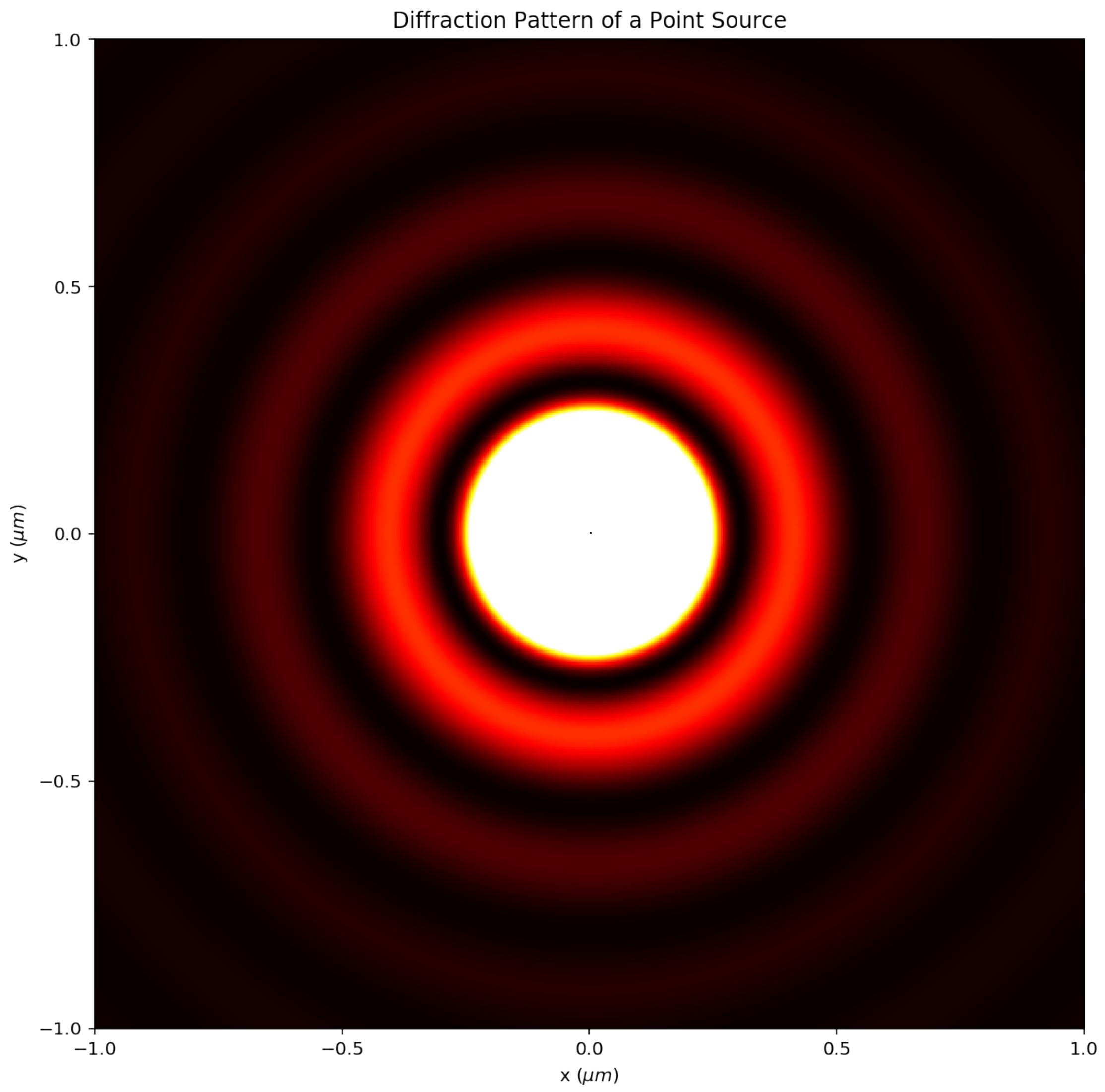
```
Out[5]: 0.5
```

```
In [6]: x = np.linspace(0, 20, 200)
        #initializes plot
        fig, ax = plt.subplots(1, 1, figsize = (10, 5))

        j0 = np.empty(200)
        j1 = np.empty(200)
        j2 = np.empty(200)

        for i in range(200):
            j0[i] = J(0,x[i])
            j1[i] = J(1,x[i])
            j2[i] = J(2,x[i])

        plt.plot(x, j0, 'c-')
        plt.plot(x, j1, 'k-')
        plt.plot(x, j2, 'm-')
        plt.title("Plotting Bessel Functions")
        plt.xlabel("x")
        plt.ylabel("J(m,x)")
        plt.show()
```



```
In [7]: %%time
        def diffraction():
            """plots the diffraction pattern for a point
                source of light through small circular
                aperture"""

            l = 500e-9 #wavelength (m)

            m = 1 #order of the bessel function used for intensity
            k = 2 * pi / l

            side = 2e-6 #side length of the square in m
            points = 500 #number of grid points along each side
            spacing = side / points #spacing of points in nm

            I = np.empty([points,points],float)
            #calculate the values in the array
            for i in range(points):
                y = spacing * i
                for j in range(points):
                    x = spacing * j
                    r = sqrt((x-side/2)**2 + (y-side/2)**2)
                    if k* r != 0:
                        I[i,j] = (J(m, k*r) / (k*r))**2

            fig, ax = plt.subplots(1, 1, figsize = (10, 10))

            #increases readability of plot
            ax.set_title("Diffraction Pattern of a Point Source")
            ax.set_xlabel("x ($\mu m$)")
            ax.set_ylabel("y ($\mu m$)")
            ax.set_xticks([-1, -0.5, 0, 0.5, 1])
            ax.set_yticks([-1, -0.5, 0, 0.5, 1])

            ax.imshow(I,origin="lower",extent=\
                      [-1,1,-1,1],\
                      cmap="hot", vmax=0.01)

        diffraction()
```

```
CPU times: user 1min 29s, sys: 636 ms, total: 1min 30s
Wall time: 1min 32s
```



3 CP 5.7 Romberg integration

This problem will evaluate the integral

$$I = \int_0^1 \sin^2 \sqrt{100x}\, dx$$

in two different ways. First, through adaptive trapezoidal integration and second, through Romberg integration. The basic trapezoidal integration method is given by

$$I(a,b) = h\, \left(\frac{1}{2}(f(a) + f(b)) + \sum_{k}^{N-1} f(a+kh) \right).$$

The adaptive method is given by

$$I_i = \tfrac{1}{2}I_{i-1} + h_i \sum_{k \text{ odd}} f(a + kh_i).$$

Thus, because the adaptive method uses the result of the previou iteration, it takes marginally more computing power to do this. The error term for this can be written as

$$\epsilon_i = \tfrac{1}{3}(I_i - I_{i-1}).$$

```
In [43]: epsilon = 1e-6

         def f(x):
             return sin(sqrt(100*x))**2

In [44]: %%time

         def adaptive_trap():
             '''performs adaptive trapezoidal integration,
                 starting with N slices and stopping after
                 set error has been reached'''

             a = 0
             b = 1
             N = 1
             error = 1 #initialization value to start while loop

             h = (b-a) / N
             I1 = h * 0.5 * (f(a) + f(b))

             while error > epsilon: #condition to keep running integration
                 h = (b-a) / N
                 I2 = 0.5 * I1

                 for k in range(1,N,2):
                     I2 += h * f(a + k*h)

                 error = abs((I2 - I1)/3)

                 print("For N = {:4}, I = {:4.6f} and the error is {:4.3e} \n"\
                           .format(N, I2, error))

                 I1 = I2
                 N *= 2

             return I2

         adaptive_trap()
```

```
For N =    1, I = 0.073990 and the error is 2.466e-02
For N =    2, I = 0.288237 and the error is 7.142e-02

For N =    4, I = 0.493785 and the error is 6.852e-02

For N =    8, I = 0.393749 and the error is 3.335e-02

For N =   16, I = 0.425479 and the error is 1.058e-02

For N =   32, I = 0.446102 and the error is 6.874e-03

For N =   64, I = 0.452757 and the error is 2.218e-03

For N =  128, I = 0.454770 and the error is 6.712e-04

For N =  256, I = 0.455422 and the error is 2.173e-04

For N =  512, I = 0.455658 and the error is 7.848e-05

For N = 1024, I = 0.455753 and the error is 3.167e-05

For N = 2048, I = 0.455795 and the error is 1.394e-05

For N = 4096, I = 0.455814 and the error is 6.495e-06

For N = 8192, I = 0.455823 and the error is 3.129e-06

For N = 16384, I = 0.455828 and the error is 1.535e-06

For N = 32768, I = 0.455830 and the error is 7.601e-07

CPU times: user 18.3 ms, sys: 3.11 ms, total: 21.4 ms
Wall time: 19.6 ms
```

So, on the 16th iteration of the adaptive trapezoidal integration, we reached the desired error set out by the problem.

However, Romberg integration will likely give us this result faster. The idea behind Romberg integration is to cancel out higher and higher order error terms by using the values we've already calculated from trapezoidal integration.

The value of the integral is given by

$$I = R_{i,m+1} + O(h_i^{2m+2})$$

with errors given by

$$c_m h_i^{2m} = \frac{1}{4^m - 1}(R_{i,m} - R_{i-1,m}) + O(h_i^{2m+2}).$$

The actual terms in the 'series' can be calculated from the preceding terms, which is what makes this such an efficient method of integration.

$$R_{i,m+1} = R_{i,m} + \frac{1}{4^m - 1}(R_{i,m} - R_{i-1,m})$$

```
In [45]: def simple_trap(N):
             '''defines a simple trapezoidal integration
                for use in the Romberg integration'''
             a = 0
             b = 1

             h = (b - a) / N

             I = f(a) * 0.5
             I += f(b) * 0.5
             for k in range(N):
                 I += f(a + k*h)

             I *= h
             return I
```

```
In [66]: %%time
         #romberg integration

         def romberg(m):
             '''Romberg integrates a function up to m rows'''

             a = 0
             b = 1

             R = np.zeros((m, m))
             for i in range(0, m):
                 R[i, 0] = simple_trap(2**i)
                 #R[i,0] = adaptive_trap()

                 for m in range(0, i):
                     R[i, m+1] = R[i,m] + ((R[i,m] - R[i-1, m])/(4**(m+1) - 1))

                     error = (R[i,m] - R[i - 1, m]) / (4**m - 1)

                     if error < epsilon:
                         break

                 print(R[i, 0:i+1])



         romberg(7)
```

```
[ 0.14797948]
[ 0.32523191  0.38431605]
[ 0.51228285  0.57463317  0.58732097]
[ 0.40299745  0.36656898  0.          0.        ]
[ 0.43010337  0.43913868  0.44397666  0.4510239   0.45279263]
[ 0.44841467  0.45451843  0.45554375  0.45572735  0.4557458   0.45574868]
[ 0.45391293  0.45574569  0.4558275   0.45583201  0.45583242  0.4558325   0.        ]
CPU times: user 6.89 ms, sys: 2.49 ms, total: 9.38 ms
Wall time: 7.89 ms


/Users/Varun/anaconda/lib/python3.6/site-packages/ipykernel_launcher.py:17: RuntimeWarning: divi
```

As can be seen from the results of Romberg integration, a precise and accurate numerical value is reached for the integral in less time (about a quarter) than with the adaptive trapezoidal method. We see that the our desired error level is reached in the 7th row

```
In [12]: %%time

         #testing romberg integration
         integrate.romberg(f, 0, 1, show=1)
```

```
Romberg integration of <function vectorize1.<locals>.vfunc at 0x1176069d8> from [0, 1]

 Steps  StepSize   Results
     1  1.000000  0.147979
     2  0.500000  0.325232  0.384316
     4  0.250000  0.512283  0.574633  0.587321
     8  0.125000  0.402997  0.366569  0.352698  0.348974
    16  0.062500  0.430103  0.439139  0.443977  0.445426  0.445804
    32  0.031250  0.448415  0.454518  0.455544  0.455727  0.455768  0.455777
    64  0.015625  0.453913  0.455746  0.455828  0.455832  0.455832  0.455832  0.455832
   128  0.007812  0.455349  0.455827  0.455832  0.455833  0.455833  0.455833  0.455833  0.455833
   256  0.003906  0.455711  0.455832  0.455833  0.455833  0.455833  0.455833  0.455833  0.455833

The final result is 0.455832532309 after 257 function evaluations.
CPU times: user 8.9 ms, sys: 4.52 ms, total: 13.4 ms
Wall time: 11.9 ms


Out[12]: 0.455833
```

3.0.1 Gaussian Quadrature Problems

For the following questions, the method of Gaussian Quadrature will be used to estimate the value of integrals. Gaussian Quadrature attempts to pick points with varied spacing along a curve in the most efficient way, so as to best calculate the area under the curve. In practice this gives a sum that is very close to the actual value of an integral.

$$\sum_i w_i\, f(x_i) \approx \int_a^b f(x)\, dx$$

```
In [13]: """Gaussian Quadrature Functions as defined by Mark Newman

            used for the following problems in the homework.

            I edited the numpy functions, adding the prefix 'np.'''"""

         # x,w = gaussxw(N) returns integration points x and integration
         #           weights w such that sum_i w[i]*f(x[i]) is the Nth-order
         #           Gaussian approximation to the integral int_{-1}^1 f(x) dx
         #
         # x,w = gaussxwab(N,a,b) returns integration points and weights
         #           mapped to the interval [a,b], so that sum_i w[i]*f(x[i])
         #           is the Nth-order Gaussian approximation to the integral
         #           int_a^b f(x) dx

         def gaussxw(N):

             # Initial approximation to roots of the Legendre polynomial
             a = np.linspace(3,4*N-1,N)/(4*N+2)
             x = np.cos(np.pi*a+1/(8*N*N*np.tan(a)))

             # Find roots using Newton's method
             epsilon = 1e-15
             delta = 1.0
             while delta>epsilon:
                 p0 = np.ones(N,float)
                 p1 = np.copy(x)
                 for k in range(1,N):
                     p0,p1 = p1,((2*k+1)*x*p1-k*p0)/(k+1)
                 dp = (N+1)*(p0-x*p1)/(1-x*x)
                 dx = p1/dp
                 x -= dx
                 delta = max(abs(dx))

             # Calculate the weights
             w = 2*(N+1)*(N+1)/(N*N*(1-x*x)*dp*dp)

             return x,w

         def gaussxwab(N,a,b):
             x,w = gaussxw(N)
             return 0.5*(b-a)*x+0.5*(b+a),0.5*(b-a)*w
```

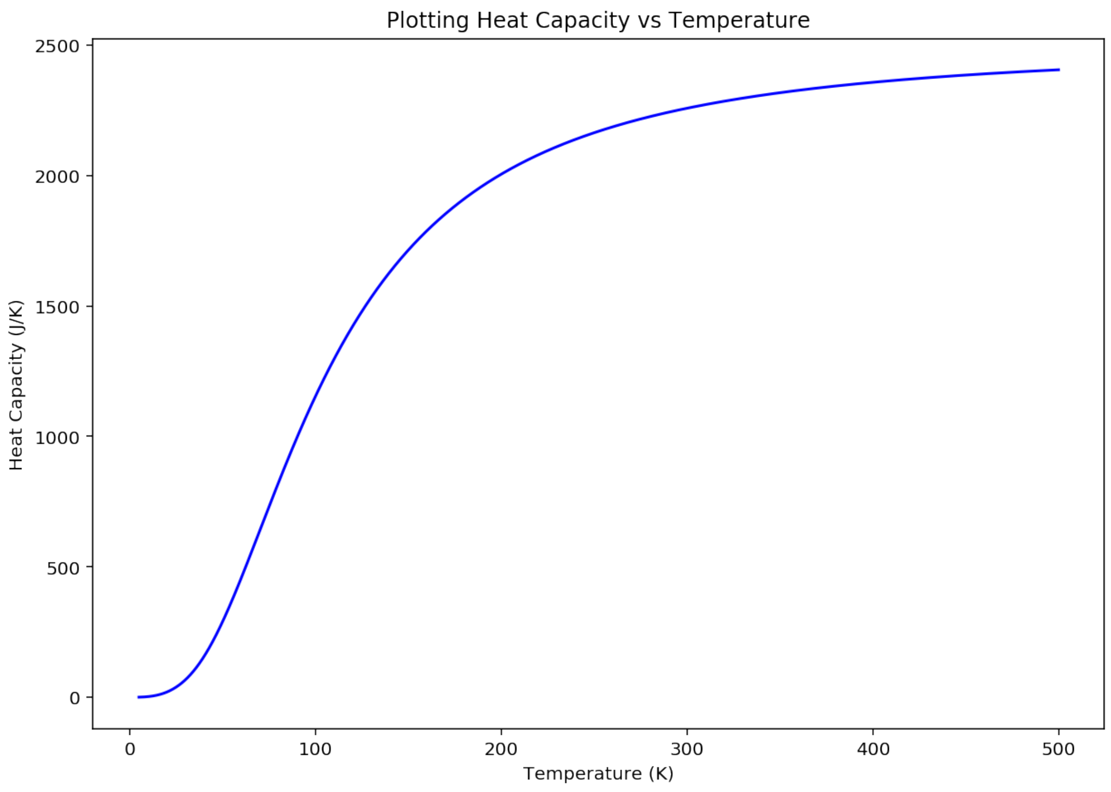
4 CP 5.9 Heat apacity using Gaussian quadrature

Debye gives the relationship between heat capacity of solids and temperature as

$$C_V(T) = 9V\rho k_B \left(\frac{T}{\theta_D}\right)^3 \int_0^{\theta_D/T} \frac{x^4\, e^x}{(\, e^x - 1)^2}\, dx.$$

Thus we can numerically evaluate this integral in order to find the heat capacity of a given solid at a given temperature. I'll look at the case specific to Aluminum.

```
In [14]: #definining constants for this problem in SI units
         V = 0.001 #volume (m^3)
         rho = 6.022e28 #number density (m^-3)
         thetaD = 428.0 #Debye temp (K)
         kB = C.k #Boltzmann's constant (J/K)

         #defining function
         def g(x):
             num = (x**4)*exp(x)
             den = (exp(x) - 1)**2
             return num / den

In [15]: def cv(T):
             """This will calculate the heat capacity of a solid
                 for a given temperature (K)"""

             #part of function outside of integral
             out = 9 * V * rho * kB * ((T/thetaD)**3)

             x,w = gaussxwab(50, 0, thetaD/T)

             I = 0
             for i in range(len(x)):
                 I += w[i]*g(x[i])

             return out*I

In [16]: #plots heat capacity from 5 to 500 K

         T = np.linspace(5, 500, 5000)
         CV = np.zeros(5000)
         for i in range(5000):
             CV[i] = cv(T[i])

         fig, ax = plt.subplots(1, 1, figsize = (10, 7))

         plt.plot(T, CV, 'k-', color='blue')

         plt.title("Plotting Heat Capacity vs Temperature")
         plt.xlabel("Temperature (K)")
         plt.ylabel("Heat Capacity (J/K)")
         plt.show()
```



5 CP 5.10 Period of an anharmonic oscillator with Gaussian quadrature

Anharmonic oscillators are any oscillators not of the form $V(x) \propto x^2$. So by treating the total energy as constant given mass m and position x, we have

$$E = \tfrac{1}{2}m\left(\frac{dx}{dt}\right)^2 + V(x)$$

which is a nonlinear differential equation in the variables x and t. We can then derive a relation for the period of the oscillator for a particle starting at rest a distance a from the center.

In this scenario, the particle's total energy $E = V(a)$, so we can rewrite the entire equation as

$$V(a) = \tfrac{1}{2}m\left(\frac{dx}{dt}\right)^2 + V(x).$$

We can use algebra to rearrange it to

$$dt = \sqrt{\frac{m}{2}}\frac{dx}{\sqrt{V(a) - V(x)}}.$$

This can be integrated on either side to find the time it takes the particle to travel any distance. For this, because we know it travels a quarter of a period in a distance a (the distance to the center of the potential), we'll use these as bounds for the integration.

$$\int_0^{\frac{1}{4}T} dt = \tfrac{1}{4}T = \int_0^a \sqrt{\frac{m}{2}} \frac{dx}{\sqrt{V(a) - V(x)}}$$

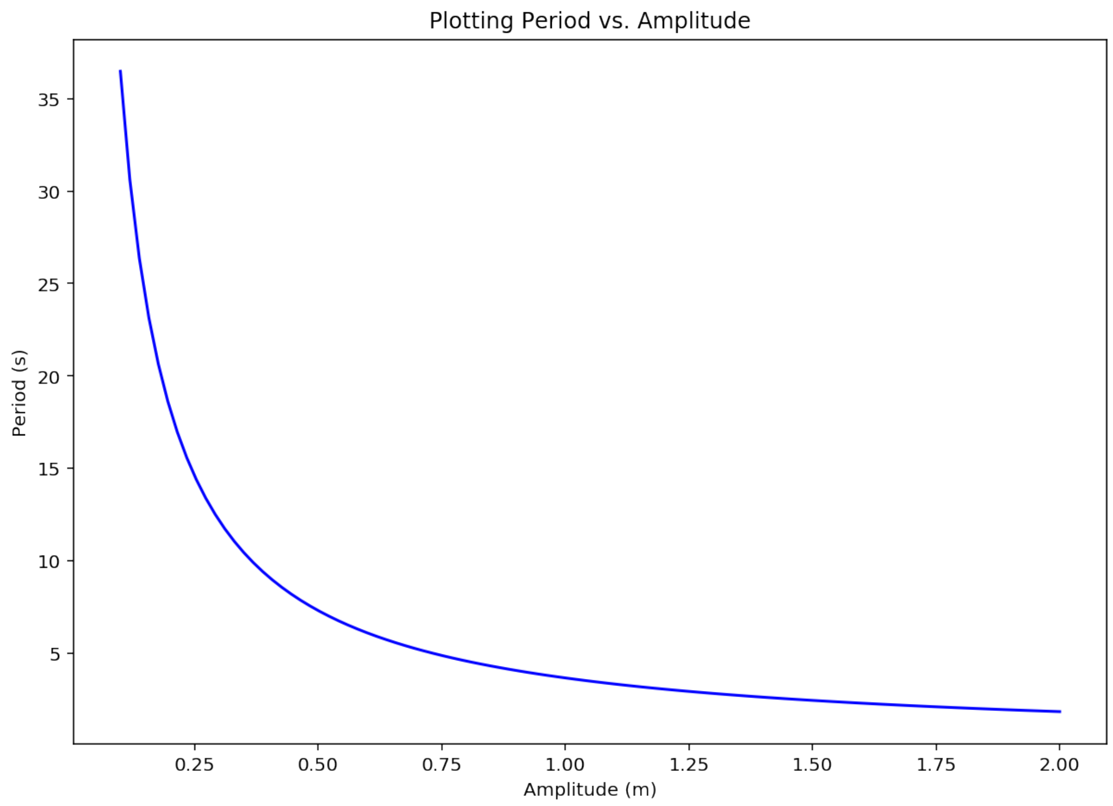
Finally, we are left with the expression for the period of the particle that we were looking for:

$$T = \sqrt{8m} \int_0^a \frac{dx}{\sqrt{V(a) - V(x)}}.$$

```
In [17]: #definining constants for this problem in SI units
         m = 1 #mass (kg)

         #defining function
         def h(a, x):
             return 1 / (sqrt(a**4 - x**4))
```

```
In [18]: def period(a, N):
             """This will calculate the period of an anharmonic
                 oscillator's period given the amplitude"""

             #part of function outside of integral
             p1 = sqrt(8*m)

             x,w = gaussxwab(N, 0, a)

             p2 = 0
             for i in range(N):
                 p2 += w[i]*h(a, x[i])

             return p1*p2
```

```
In [19]: #plots period as amplitude ranges from a=0 to a=2

         a = np.linspace(1e-1, 2, 100)
         T = np.zeros(100)
         for i in range(100):
             T[i] = period(a[i], 20)

         fig, ax = plt.subplots(1, 1, figsize = (10, 7))

         plt.plot(a, T, 'k-', color='blue')

         plt.title("Plotting Period vs. Amplitude")
         plt.xlabel("Amplitude (m)")
         plt.ylabel("Period (s)")
         plt.show()
```



The oscillator's increased speed, and thus decreasing period as a result of increasing amplitude is explained by the fact that the potential is quartic. Because this problem was done under the assumptions of energy conservation, the (quadratic in velocity) kinetic energy

$$K = \tfrac{1}{2}mv^2 = \tfrac{1}{2}m\left(\frac{dx}{dt}\right)^2$$

must make up for any lost potential energy. So as the potential drops off as a quartic function while it approaches zero, the speed must increase in greater proportion because it is only quadratic.

This same trade-off is responsible for the period going to infinity as amplitude goes to zero. Because the maximum potential energy is incredibly small in this scenario, at $x = 0$ the speed must also be incredibly small (and this is where it is at its maximum). Therefore, the period will diverge as amplitude goes to zero.

6 CP 5.11 Diffraction around edges with Gaussian quadrature

When a plane wave diffracts aroud an edge, the intensity at a point (x, z) is given by

$$I = \frac{I_0}{8}\left(\left[2C(u)+1\right]^2 + \left[2S(u)+1\right]^2\right)$$

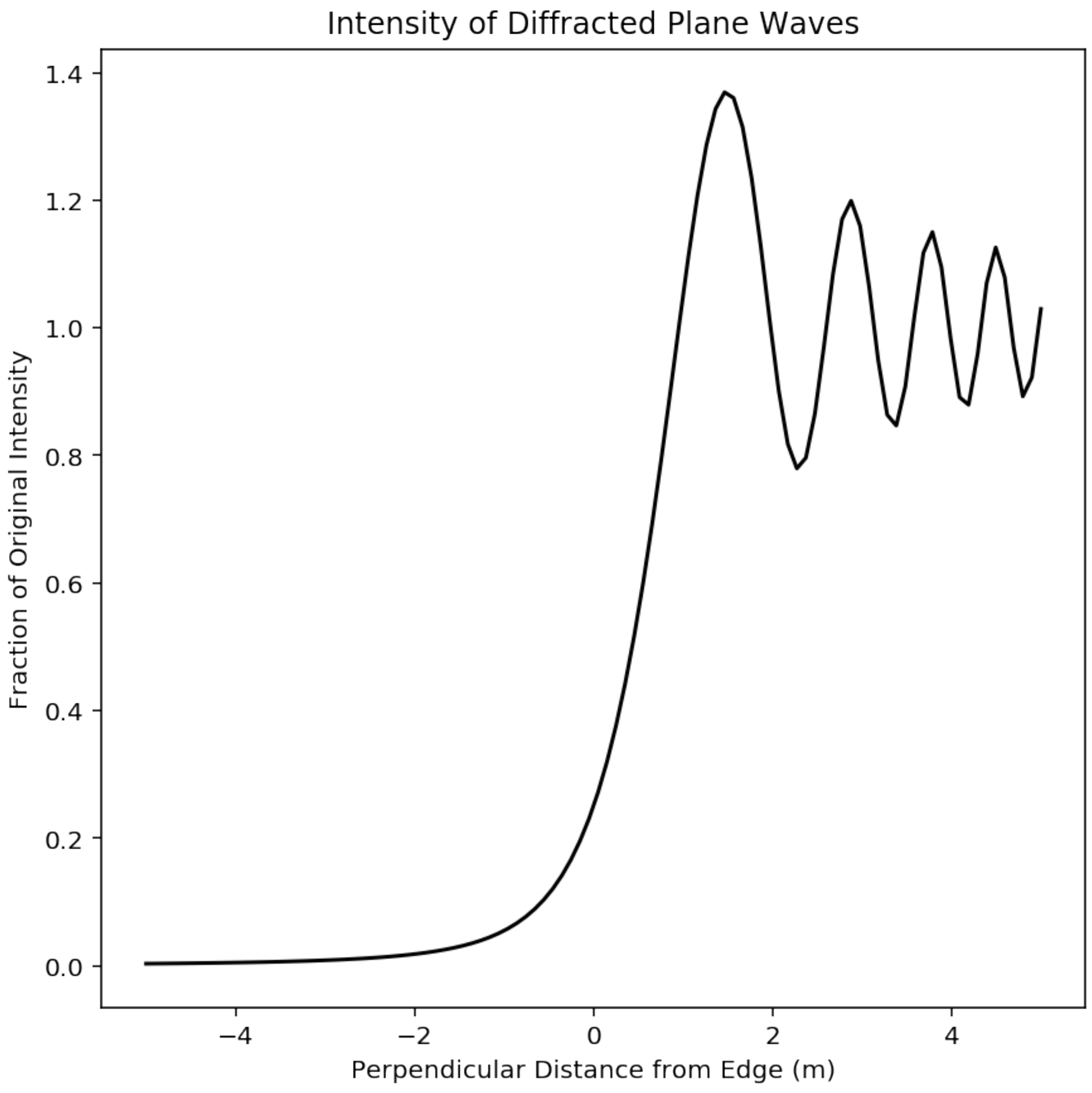
where

$$u = x\sqrt{\frac{2}{\lambda z}}, \qquad C(u) = \int_0^u \cos\tfrac{1}{2}\pi t^2\,dt, \qquad S(u) = \int_0^u \sin\tfrac{1}{2}\pi t^2\,dt.$$

So, given this, a function can be defined to calculate the intensity I of a sound wave relative to its pre diffracted intensity I_0.

```
In [20]: #definining constants for this problem in SI units
         l = 1 #wavelength (m)
         N = 50 #sampling points for Gaussian quadrature

         #defining position function
         def u(x,z):
             return x*sqrt(2 / (l*z))

         #Main functions are defined below

         def Cu(u):
             def c(t):
                 return cos(0.5*pi*(t**2))

             x,w = gaussxwab(N, 0, u)

             pp1 = 0
             for i in range(N):
                 pp1 += w[i]*c(x[i])

             return pp1

         def Su(u):
             def s(t):
                 return sin(0.5*pi*(t**2))

             x,w = gaussxwab(N, 0, u)

             pp1 = 0
             for i in range(N):
                 pp1 += w[i]*s(x[i])

             return pp1

         def ratio(u):
             """Takes position as an argument and calculates the fraction
                 of original intensity of a diffracted wave after edge"""

             II0 = ((2*Cu(u) + 1)**2 + (2*Su(u) + 1)**2) / 8
             return II0

In [21]: #plots diffraction pattern

         density = 100
         z = 3 #distance past edge (m)
```

```
x = np.linspace(-5,5,density)
position = np.zeros(density)
intensity = np.zeros(density)

for i in range(density):
    position[i] = u(x[i],z)
    intensity[i] = ratio(position[i])

fig, ax = plt.subplots(1, 1, figsize = (7, 7))

plt.plot(x, intensity, 'k-')

plt.title("Intensity of Diffracted Plane Waves")
plt.xlabel("Perpendicular Distance from Edge (m)")
plt.ylabel("Fraction of Original Intensity")
plt.show()
```



As per the problem, there is noticeable variation in the intensity of sound diffracted around the edge, from almost nonexistent if you're well 'covered' by the screen to greater than the original intensity a little under 2 m up when 3 m away.

7 CP 5.12 The Stefan-Boltzmann constant with Gaussian quadrature

In some time interval $d\omega$ a black body electromagnetically radiates thermal energy $I(\omega)d\omega$, where

$$I(\omega) = \frac{\hbar}{4\pi^2 c^2} \frac{\omega^3}{(e^{\hbar\omega/k_B T} - 1)}.$$

Therefore the total energy per unit area radiated by that same black body is

$$W = \frac{k_B^4 T^4}{4\pi^2 c^2 \hbar^3} \int_0^\infty \frac{x^3}{e^x - 1}\, dx.$$

```
In [22]: #defining constants
         kB = C.k #Boltzmann's constant (J/K)
         hbar = C.hbar #reduced Planck's constant (kg m^2 s^-1)
         c = C.c #speed of light (m/s)

         #defining the integrand
         def intgrnd(z):
             num = (z/(1-z))**3 #numerator of f(x)
             den = (exp(z/(1-z)) - 1) #denominator of f(x)
             return (num / den) * (1/ ((1-z)**2)) #chain rule


         def radiated_energy(T):
             """Gives the total energy per unit area radiated
                by a black body."""

             N = 50
             num = (kB**4) * T**4
             den = 4 * (pi*c)**2 * hbar**3
             out = num / den #constants outside of integral

             x,w = gaussxwab(N, 0.0001, 0.999)

             I = 0
             for i in range(N):
                 I += w[i]*intgrnd(x[i])

             return out*I
```

To evaluate this integral I used Gaussian quadrature over the range $(0, 1)$. To change the range, and calculate this integral from zero to infinity, a substitution was required to change it to a finite integral. In general,

$$\int_0^\infty f(x)\,dx = \int_0^1 \frac{1}{(1-z)^2} f\left(\frac{z}{1-z}\right) dz.$$

My answer should be accurate the same order as all other integrals calculated through Gaussian quadrature, i.e., about 50 points would correspond to the limit of the computer's precision.

```
In [23]: #calcualtes Stefan-Boltzmann constant and fractional error


         def find_sigma(T):
             """This function ends up always taking the same value
                because the T^4 from the energy and the Stefan-Boltzmann
                relation cancel. It also finds the error on calculating
                sigma"""
             sigma = radiated_energy(T) / T**4
             error = (abs(sigma - C.sigma)) / C.sigma * 100 #in percent
```

```
    return sigma, error


print("Stefan-Boltzmann constant is {:2.3e} and error is {:2.2e}%."\
      .format(find_sigma(1000)[0], find_sigma(1000)[1]))
```

```
Stefan-Boltzmann constant is 5.670e-08 and error is 3.28e-06%.
```

8 CP 5.13 Quantum uncertainty in the harmonic oscillator with Gaussian quadrature

In a one-dimensional quantum harmonic oscillator, a spinless point particle's wavefunction is

$$\psi_n(x) = \frac{1}{\sqrt{2^n n! \sqrt{\pi}}} e^{-x^2/2} H_n(x)$$

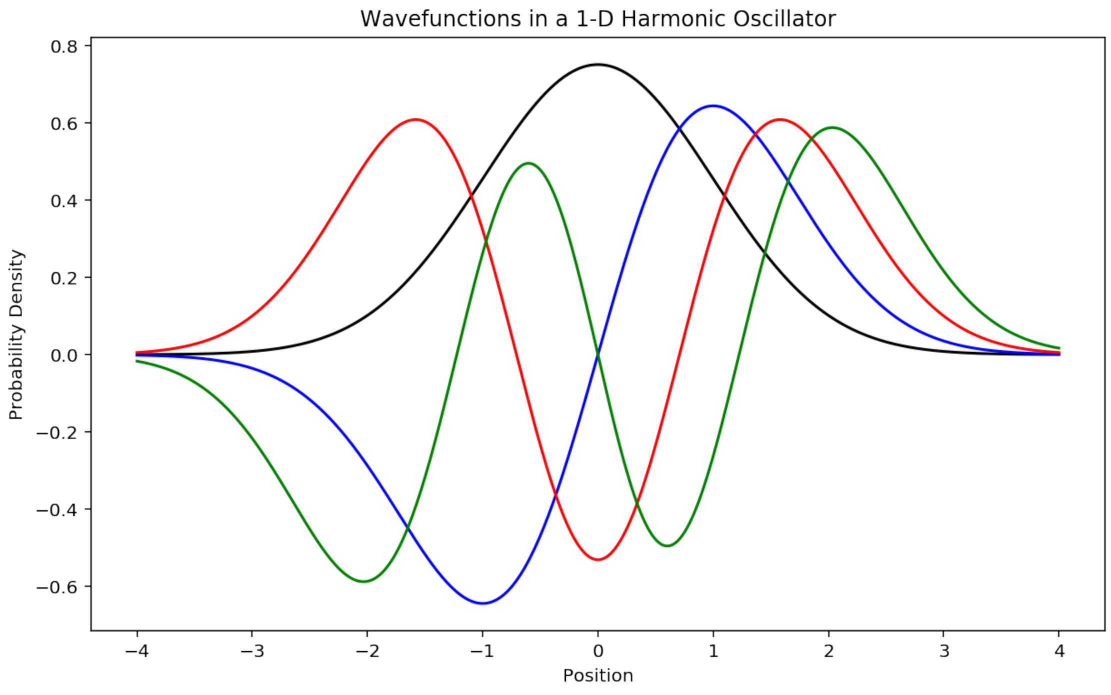
if that particle is in the nth energy level. (Assuming use of units that make constants 1.) The Hermite polynomials $H_n(x)$ are given by

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x),$$

where $H_0(x) = 1$ and $H_1(x) = 2x$.

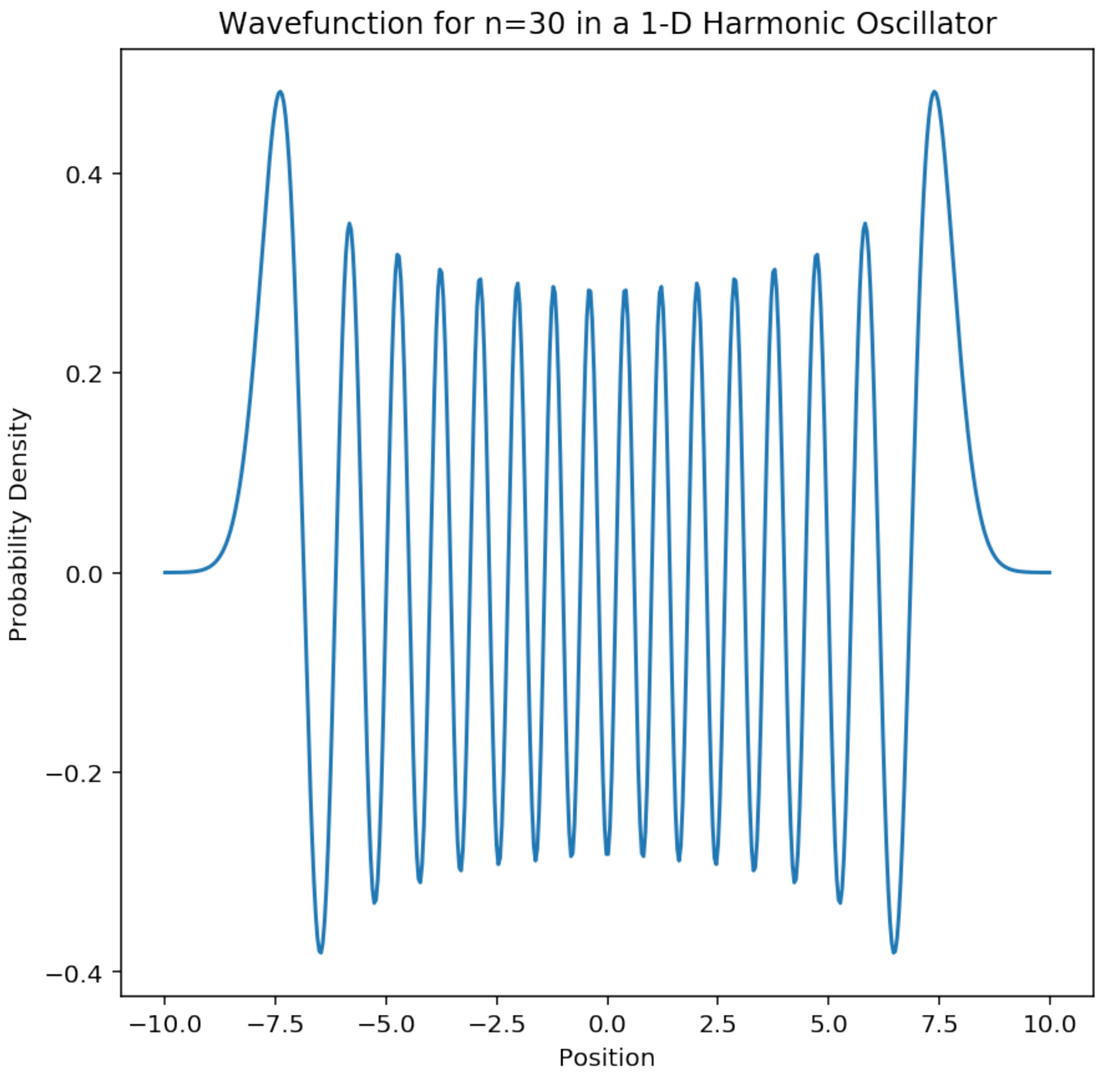
```
In [71]: def Hermite(n,x):
             """Gives nth Hermite polynomial for given n and x"""

             if n == 0:
                 return 1
             elif n == 1:
                 return 2*x
             else:
                 return 2*x*Hermite(n-1,x) - 2*(n-1)*Hermite(n-2,x)


         def psi(n,x):
             """Gives the wavefunction for a particle in the nth energy
                 level over position x"""

             frac = 1 / (sqrt((2**n)*factorial(n)*sqrt(pi)))
             expo = exp((-x**2)/2)

             return frac * expo * Hermite(n,x)

In [73]: #plots on same set of axes


         x = np.linspace(-4,4,400)

         fig, ax = plt.subplots(1, 1, figsize = (10, 6))
```

```
#jet= plt.get_cmap('jet')
#colors = iter(jet(np.linspace(0,1,4)))

psi0 = np.zeros(400)
psi1 = np.zeros(400)
psi2 = np.zeros(400)
psi3 = np.zeros(400)

for i in range(400):
    psi0[i] = psi(0,x[i])
    psi1[i] = psi(1,x[i])
    psi2[i] = psi(2,x[i])
    psi3[i] = psi(3,x[i])

plt.plot(x, psi0, 'k-')
plt.plot(x, psi1, 'b-')
plt.plot(x, psi2, 'r-')
plt.plot(x, psi3, 'g-')

plt.title("Wavefunctions in a 1-D Harmonic Oscillator")
plt.xlabel("Position")
plt.ylabel("Probability Density")
plt.show()
```



```
In [79]: #plot for 30th energy level from x=-10 to x=10
         count = 500
```

```
x2 = np.linspace(-10,10,count)

fig, ax = plt.subplots(1, 1, figsize = (7, 7))

psi30 = np.zeros(count)
for i in range(count):
    psi30[i] = psi(30,x2[i])

plt.plot(x2,psi30, '-')

plt.title("Wavefunction for n=30 in a 1-D Harmonic Oscillator")
plt.xlabel("Position")
plt.ylabel("Probability Density")
plt.show()
```



A different method of substitution can be used to evaluate the integral

$$\langle x^2 \rangle = \int_{-\infty}^{\infty} x^2 |\psi_n(x)|^2 \, dx.$$

In this case, we make the substitutions $x = \tan z$ and $dx = \frac{dz}{\cos^2 z}$. This will give us the equivalency

$$\int_{\infty}^{\infty} f(x) \, dx = \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} \frac{f(\tan z)}{\cos^2 z} \, dz.$$

So, we can make these substitutions in the wavefunctions to find the RMS position.
The wavefunctions in x are

$$\psi_n(x) = \frac{1}{\sqrt{2^n n! \sqrt{\pi}}} e^{-x^2/2} H_n(x).$$

We should be evaluating the integral

$$\int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} \frac{\tan^2 z}{\cos^2 z} \left| \psi_n(\tan z) \right|^2 dz$$

For $n = 5$, we should find that the root-mean-square position $\sqrt{\langle x^2 \rangle} = 2.3$.

```
In [77]: #defines integrand in change of variables
         def change(z):
             n = 5
             front = ((tan(z))**2 / (cos(z))**2)
             return front * (abs(psi(n, tan(z))))**2

         def RMS():
             a = -pi/2
             b = pi/2
             N = 100

             x,w = gaussxwab(N, a, b)

             s = 0
             for i in range(N):
                 s += w[i]*change(x[i])

             return sqrt(s)

         error = 100 * (RMS() - 2.3)/2.3

         print("The RMS = {:2.3f}, which has an error of {:2.2f}%"\
              .format(RMS(), error))
```

```
The RMS = 2.345, which has an error of 1.97%
```

This value of the RMS $\sqrt{\langle x^2 \rangle}$ is in the neighborhood of the known value.

9 CP 5.19 Diffraction gratings

Light diffracted through a grating then put through a lens has intensity given by the function

$$I(x) = \left| \int_{-w/2}^{w/2} \sqrt{q(u)}\, e^{2\pi i x u/\lambda f}\, du \right|^2$$

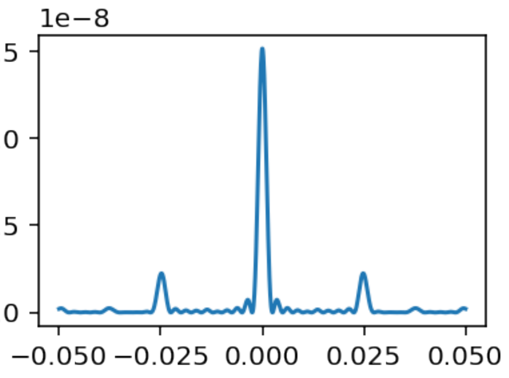
for grating of width w, light of wavelength λ, lens of focal length f, and distance from the central axis u.

For a grating with the transmission function $q(u) = \sin^2 \alpha u$, the separation of slits $d = \frac{\pi}{\alpha}$.

```
In [31]: #constants
         alpha = pi / 20e-6 #(m)
         l = 500e-9 #(m)
         f = 1 #(m)
         w = 10 * (pi/alpha)

         slow = -0.05 #screen lower bound (m)
         supp = 0.05 #screen upper bound (m)

         def q(u):
             return (sin(alpha*u))**2
```

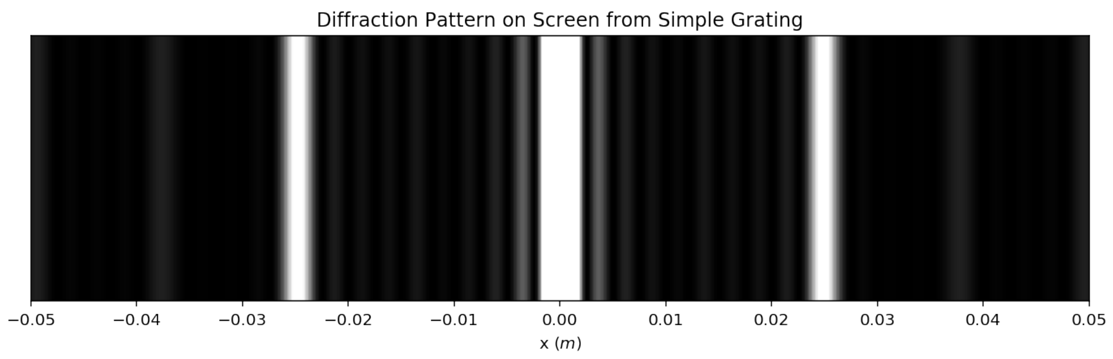
```
In [32]: #defines integrand
         def intgrd(u,x):
             """Gives value of the integrand for given u and x"""
             return sqrt(q(u)) * (cexp(2*pi*x*u*1j/(l*f)))


         def intensity(x):
             """Uses Simpson's rule to integrate the intensity function"""
             a = -w/2
             b = w/2
             N = 50
             h = (b-a) / N

             sum1 = 0
             sum2 = 0
             for k in range(N):
                 if k % 2 == 1:
                     sum1 += intgrd(a+k*h, x)
                 elif k%1 == 0:
                     sum2 += intgrd(a+k*h, x)

             intensity = h*(intgrd(a,x) + intgrd(b,x) + 4*sum1 + 2*sum2) / 3

             I = (abs(intensity))**2

             return I
```

In [33]:
```
#plotting intensity profile to find ideal umax

points = 1000
xx = np.linspace(slow, supp, points)
density = np.empty(points)

for i in range(points):
    density[i] = intensity(xx[i])

fig, ax = plt.subplots(1, 1, figsize = (3, 2))

plt.plot(xx, density)
plt.show()
```



In [34]:
```
#code creating the screen
screen = 0.1 #(m)
points = 1000 #number of grid points on long side
spacing = screen / points #spacing of points in m
center = screen/2

pattern = np.zeros([400,points],float)
#calculate the values in the array

for i in range(points):
    x = spacing * i
    pattern[:,i] = intensity(x-center)

fig, ax = plt.subplots(1, 1, figsize = (12,3))

#increases readability of plot
ax.set_title("Diffraction Pattern on Screen from Simple Grating")
ax.set_xlabel("x ($m$)")
ax.set_xticks([-0.05, -0.04, -0.03, -0.02, -0.01, 0,\
               0.01, 0.02, 0.03, 0.04, 0.05])
ax.set_yticks([])

#vmax deduced from plot above
ax.imshow(pattern,origin="lower",\
          cmap="gray", vmax=0.2e-8,\
          extent=[-0.05,0.05, 0,.025])
```

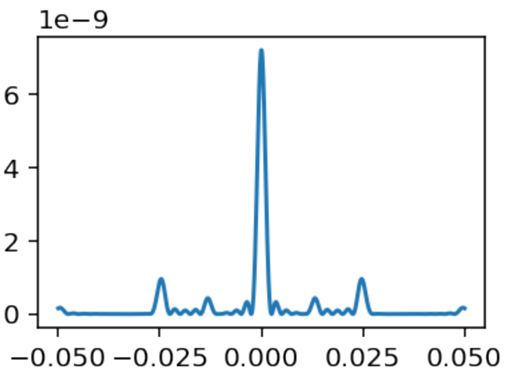
```
Out[34]: <matplotlib.image.AxesImage at 0x11744bcc0>
```



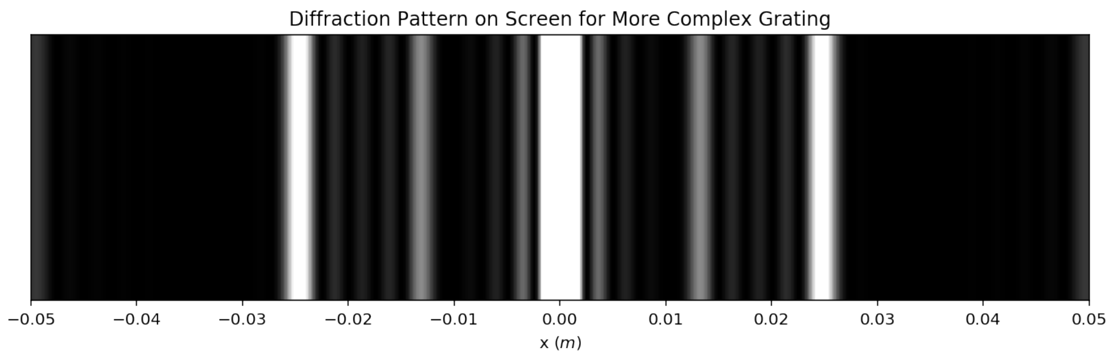
The next few cells redefine the transmission function $q(u)$ and create a density plot for this physical phenomenon. The code for plotting is the same, except the value for vmax on the density plot will likely be different and now determined from a new intensity profile.

```
In [35]: #new cell for defining things
         beta = 0.5 * alpha

         def q(u):
             return ((sin(alpha*u))**2) * ((sin(beta*u))**2)
```

```
In [36]: #plotting intensity profile to find ideal vmax


         points = 1000
         xx = np.linspace(slow, supp, points)
         density = np.empty(points)

         for i in range(points):
             density[i] = intensity(xx[i])

         fig, ax = plt.subplots(1, 1, figsize = (3, 2))

         plt.plot(xx, density)
         plt.show()
```



```
In [37]: #code creating the screen
         screen = 0.1 #(m)
         points = 1000 #number of grid points on long side
         spacing = screen / points #spacing of points in m
         center = screen/2

         pattern = np.zeros([400,points],float)
         #calculate the values in the array

         for i in range(points):
             x = spacing * i
             pattern[:,i] = intensity(x-center)

         fig, ax = plt.subplots(1, 1, figsize = (12,3))

         #increases readability of plot
         ax.set_title("Diffraction Pattern on Screen for More Complex Grating")
         ax.set_xlabel("x ($m$)")
         ax.set_xticks([-0.05, -0.04, -0.03, -0.02, -0.01, 0,\
                        0.01, 0.02, 0.03, 0.04, 0.05])
         ax.set_yticks([])

         #vmax deduced from plot above
         ax.imshow(pattern,origin="lower",\
                   cmap="gray",vmax=0.8e-9,\
                   extent=[-0.05,0.05, 0,.025])

Out[37]: <matplotlib.image.AxesImage at 0x116fbce48>
```



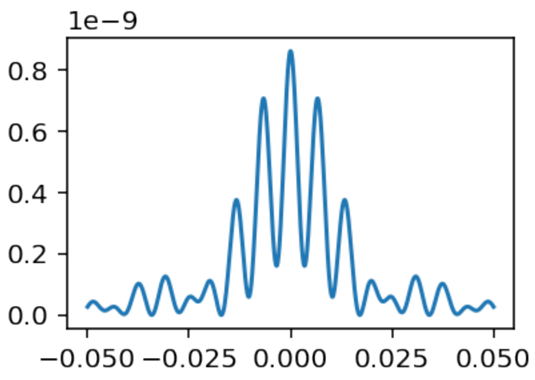
The next few cells redefine the transmission function $q(u)$ again and create a density plot for the two "square" slits scenario. The transmission function for this system should be defined piece-wise for the 4 regions that exist. The regions can be combined into ones of transmission and ones without.

1. Beyond the far edges of the slits
2. In front of the 10 μm slit
3. Behind the 60 μm gap
4. In front of the 20 μm slit

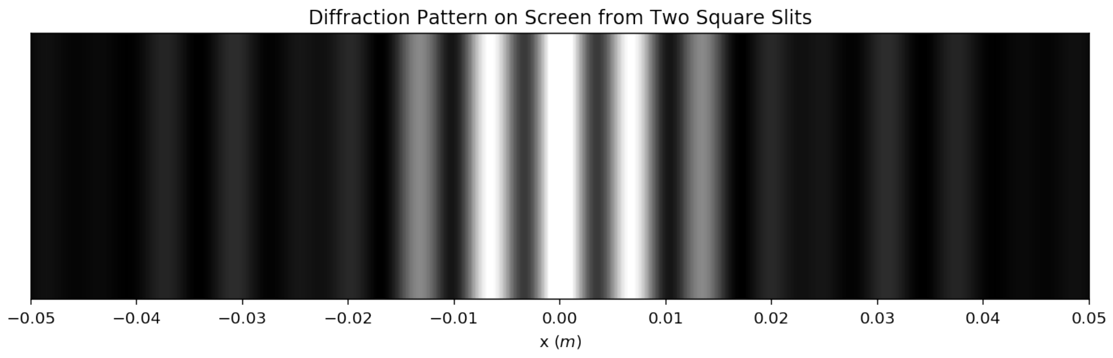
The code for plotting is the same, except the value for vmax on the density plot will likely be different and now determined from a new intensity profile.

```
In [38]: def q(u):
             #in front of slits
             if (u > 0 and u < 10e-6) or (u > 70e-6 and u < 90e-6):
                 return 1

             #outside
             else:
                 return 0
```

```
In [39]: #plotting intensity profile to find ideal vmax

         points = 1000
         xx = np.linspace(slow, supp, points)
         density = np.empty(points)

         for i in range(points):
             density[i] = intensity(xx[i])

         fig, ax = plt.subplots(1, 1, figsize = (3, 2))

         plt.plot(xx, density)
         plt.show()
```



```
In [40]: #code creating the screen
         screen = 0.1 #(m)
         points = 1000 #number of grid points on long side
         spacing = screen / points #spacing of points in m
         center = screen/2

         pattern = np.zeros([400,points],float)
         #calculate the values in the array

         for i in range(points):
             x = spacing * i
             pattern[:,i] = intensity(x-center)

         fig, ax = plt.subplots(1, 1, figsize = (12,3))

         #increases readability of plot
         ax.set_title("Diffraction Pattern on Screen from Two Square Slits")
         ax.set_xlabel("x ($m$)")
         ax.set_xticks([-0.05, -0.04, -0.03, -0.02, -0.01, 0,\
                       0.01, 0.02, 0.03, 0.04, 0.05])
         ax.set_yticks([])

         #vmax deduced from plot above
         ax.imshow(pattern,origin="lower",\
                   cmap="gray",vmax=0.7e-9,\
                   extent=[-0.05,0.05, 0,.025])

Out[40]: <matplotlib.image.AxesImage at 0x116f5e550>
```


Diffraction Pattern on Screen from Two Square Slits
−0.05
−0.04
−0.03
−0.02
−0.01
0.00
0.01
0.02
0.03
0.04
0.05
x (m)

```
In [ ]:
```