# Gotta Catch 'em all

Arun Suresh

## 1 Introduction

Root finding algorithms have existed since the 1600s and have sparked many an interest among mathematicians and scientists alike. Almost any scientific problem can be rewritten in some fashion as a root finding problem, with the most common rewriting $f(x) = g(x) \implies h(x) \equiv f(x) - g(x) = 0$. This could however get very complicated depending on the function(s) themselves. Thus most root finding for complicated functions are performed numerically. Currently there is a large number of root finding algorithms that exist, but all of them stem from the same basic 10-12 unique parent algorithms.

### 1.1 Root Finding - A general idea

A root finding algorithm works by first bracketing a root of the given continuous function. This is achieved by identifying two points at which the function takes opposite signs - the existence of a root in that interval is then guaranteed by the intermediate value theorem. After bracketing a root, an approximation of the root is made (for example, a linear interpolation between the two end points, and finding the $x$-intercept). This approximation is most probably a bad one, and so - the end points are then shuffled and the process is repeated to iteratively converge to the true root. Root finding algorithms usually have fast convergence rate - the more sophisticated ones having a cubic convergence.

### 1.2 Issues with root finding

While root finding methods are very efficient, and can be customized for a particular problem, physical or non-physical; they come with an intrinsic downside that can not be avoided - they force the user to have some idea about the distribution of zeroes of the function the user is trying to analyze. The following exemplifies the issue in various dimensions.

- Let $f(x) = x^4 - 102632x^3 - 115571937x^2 - 114853500x + 692815788$. The roots of $f$ are $103746, 2, -3$ and $-1113$. As someone who doesn't know what the roots are - the user would set the initial bracket to something like $[-10000, 10000]$ to let the code run. Since there are indeed 3 zeroes in the interval, the code doesn't have problem converging to one of them. But unless the initial interval spans the 100000s, the code will never be able to converge to that root. One may argue that polynomial long/synthetic division would yield the fourth root, which is true in this case - but it only gets harder and harder to compute as the degree of the polynomial increases or, even worse - the given function isn't a polynomial to begin with.

- A more serious problem than the previous issue is that - Since root finding methods are often built on approximating a root, and then iteratively improving the approximation - the root finding algorithm converges to exactly one root per run. If the function is very oscillatory then the root finding method would oscillate between two roots until it gets sufficiently close to one. In either case, the output of a successful root finding run is always a single number.

- A root finding algorithm will not identify complex roots. If a system has no roots, with a large enough tolerance, the algorithm converges to the minimizer of $f$.

## 1.3 Eigenvalue problems: A detour

### 1.3.1 Eigenvalue problem

Let $A = n \times n$ matrix. An eigenvalue problem refers to the following question "Find all $\lambda$ such that $A\mathbf{x} = \lambda\mathbf{x}$". That is to say, if we think of $A$ as a linear transformation from $\mathbb{R}^n \to \mathbb{R}^n$, the problem asks us to find the scale factor of all the eigenvectors, where eigenvectors refer to $\mathbf{x}$, the vectors in $\mathbb{R}^n$ that only get re-scaled by a factor of $\lambda$ upon the transformation. Often times an eigenvalue problem is solved by seeing that

$$A\mathbf{x} = \lambda\mathbf{x} \implies (A - \lambda I)\mathbf{x} = 0 \implies \det(A - \lambda I) = 0$$

However, from elementary linear algebra, we know that the determinant of a $n \times n$ matrix with an indeterminate yields an $n^{th}$ degree polynomial in the indeterminate. This polynomial is known as the characteristic equation of the matrix. In our case, the indeterminate is $\lambda$. With the determinant being equal to zero, we have this polynomial equal to zero. So, any given eigenvalue problem can be reduced to a root finding problem.

### 1.3.2 QR Algorithm

The practical QR algorithm proceeds in the following fashion. It is a generally known result in Linear algebra that given a matrix $A$ we can decompose $A$ in as $QR$ where $Q$ is an orthogonal matrix (A matrix that satisfies the property that $Q^T = Q^{-1}$) and $R$ is an upper triangular matrix.
So, we initialize $A_0 = K = Q_0R_0$ and at the $k^{th}$ step, we compute $A_k = A_kR_k$ and then form $A_{k+1} = R_kQ_k$. This yields,

$$A_{k+1} = R_kQ_k = Q_k^{-1}Q_kR_kQ_k = q_k^{-1}A_kQ_k = Q_k^TA_kQ_k$$

This means every all $A_k$s are similar to each other ($A, B$ are similar if $\exists Q$ non-singular such that $A = Q^{-1}BQ$). If the matrix $A$ is hessenberg or quasi-hessenberg, then the sequence $A_k$ converges to an upper triangular matrix whose eigenvalues are listed along the diagonal. Since similarity relations preserve eigenvalues, our original eigenvalue problem is solved. This algorithm comes built in with the numpy package in python and was used extensively in this project

### 1.3.3 The most natural question

A very interesting question is, can we go the other direction? The implication of the answer being "yes" is that, we can turn root finding problem to an eigenvalue problem - and upon doing so, we can use the rich literature for matrix theory and eigenvalue problems to obtain the eigenvalues in a different way! This seems promising because of the large number of eigenvalue solvers that exist in the literature, and also that some eigenvalue solvers such as QR Transformations and Householder methods do not rely on iterative approximation of a single root.

# 2 Companion Matrix and Root finding

## 2.1 Finding the $A$ matrix

Given a polynomial $f(x) = x^n + c_{n-1}x^{n-1} + c_{n-2}x^{n-2} + \cdots + c_1x + c_0$[1] and this particular direction of pursuit, the most natural question we can ask is - "can we find a matrix whose characteristic equation is $f$". Before identifying the answer, I would like to work through a few examples to intuitively understand the process behind building such a matrix.

Consider a Quadratic polynomial $f = x^2 + 3x + 4$. Notice that $f$ can be factorized in the following way $= x(x+3)+4 = (0-x)(-3-x)-(-4)(1)$. The last bit of the equality looks like the determinant of a $2 \times 2$ matrix. In particular, it is this matrix $A' = \begin{bmatrix} 0-x & -4 \\ 1 & -3-x \end{bmatrix}$. So from here, it is straight forward to find the matrix $A$ for this case it is simply

$$A_2 = \begin{bmatrix} 0 & -4 \\ 1 & -3 \end{bmatrix}$$

.

Increasing the complexity a little bit more, we consider a cubic polynomial and do a similar simplification $g(x) = x^3 + 3x^2 + 4x + 3 = x(x^2 + 3x + 4) - (-3)(1)$. Notice that we already know $A_2$ for $x^2 + 3x + 4$, so the natural thing to do is to make $A_2$ the lower principle matrix of our $A_3$. After some manipulation, we can see that

$$A_3 = \begin{bmatrix} 0 & 0 & -3 \\ 1 & \mathbf{0} & \mathbf{-4} \\ 0 & \mathbf{1} & \mathbf{-3} \end{bmatrix}$$

This gives us an idea of how we can obtain $A_n$ iteratively.

$A_n$ is called a companion matrix and for $f(x) = x^n + c_{n-1}x^{n-1} + c_{n-2}x^{n-2} + \cdots + c_1x + c_0$,

[1] If $f$ is not monic, we can divide out by the leading coefficient to make it monic. This does not change the position of the root

it is given by

$$A_n = \begin{bmatrix} 0 & 0 & \dots & 0 & -c_0 \\ 1 & 0 & \dots & 0 & -c_1 \\ 0 & 1 & \dots & 0 & -c_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & -c_{n-1} \end{bmatrix}$$

The companion matrix is a member of the Hessenberg matrices which is a family of lower triangular matrices that are often used in numerical analysis. In fact, as we will see in the later sections, $A_n$ being Hessenberg proves to be vital to our eigenvalue solver.

### 2.2 Benefits of this method

This method of computation is very robust, and is as accurate as the eigenvalue solver. Since we make use of sparse matrices, this is not computationally costly either. The main benefits of this method is that with a good eigenvalue solver, we can obtain

- Roots that are arbitrarily large
- All roots at the same time!
- Identify even complex roots!

The eigenvalue solvers that are widely used today are **Divide and Conquer** which has a quadratic convergence and **QR algorithm** which has a cubic convergence.

## 3 Results

For my project, I implemented this method for three examples, and provided the proof of concept for an additional example case. The cases that I considered are:

- Polynomial function
- Non-polynomial function, non-algebraic roots
- A physics application

## 3.1 polynomial function

For this application, I considered $n \in \mathbb{Z}^+$. Following that, I generated an $n^{th}$ degree polynomial with random integer coefficients. Presented below is an example plot of a $4^{th}$ degree polynomial and the output given by the code. A fourth degree polynomial was chosen be-

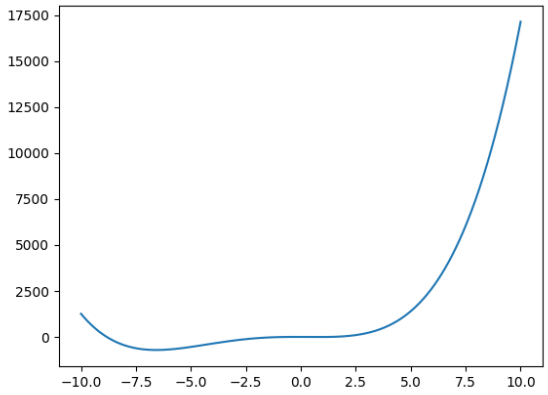


Figure 1: Random biquadratic Polynomial

cause it was the least order polynomial whose zeroes would be a nightmare to calculate by hand (There is also no said formula for finding the roots of a bi-quadratic polynomial), but more importantly - this was the degree in which the plot displayed significant oscillations near its minima to be visually appealing/convincing. The following is the result that was observed as output This checks out as the actual roots of the polynomial, which was verified

```
[1, 8, -8, -6, 4]
[ 1.11505411  0.50838569 -0.79968166 -8.82375813]
```

Figure 2: Output

using wolframalpha, that returned the following figures

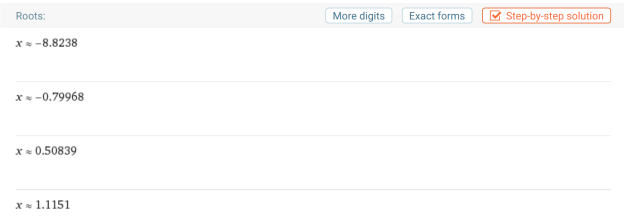


Figure 3: Wolfram Alpha numerical solution

## 3.2 Non-polynomial Function, Non-algebraic roots

After testing the code with a polynomial function, the next most obvious question is to see if the code can work for non-polynomial functions. So I considered a simple function $f(x) = \cos(x)$ in the domain $[-2\pi, 2\pi]$ to see if the code can identify the roots of this function. I also chose this function because we know analytically that the function becomes zero at $\frac{-3\pi}{2}, \frac{-\pi}{2}, \frac{\pi}{2}, \frac{3\pi}{2}$. However, the method demands that we use polynomials - so we consider the Taylor series of the function. Plotted below is $f$ along with its taylor series approximation up to the first twelve terms (a 24th degree polynomial). Presented below was the output

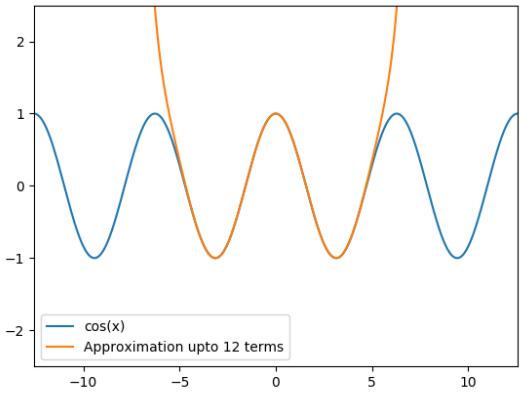


Figure 4: $\cos(x)$ + Taylor Series

that was obtained We can compute $\frac{\pi}{2} = 1.57079632679$ and $\frac{3\pi}{2} = 4.71238898038$ Comparing

```
Zero (-4.686517663795762+0j)
Zero (-1.5707963331163652+0j)
Zero (1.5707963331163661+0j)
Zero (4.68651766379572+0j)
```

Figure 5: Output

this to the output the absolute errors are:

- $|1.57079632679 - 1.57079633311| = 6.32000008 \times 10^{-9}$
- $|4.71238898038 - 4.68651766379| = 0.02587131659$

So, absolute error is always less than 0.03 for $|x| < \frac{3\pi}{2}$. The significant jump in absolute error is observed because the Taylor approximation is deemed accurate only until a little after $\frac{3\pi}{2}$. We notice that the polynomial diverges just before 5. For more accurate estimates of zeroes all we have to do is increase number of terms!

But the above example more than just an arbitrary example that was chosen. It alludes to a particular Sturm-Liouville BVP.

**Definition 1.** ***Sturm Liouville Eigen Value problem***

$$\mathcal{L}[y] \equiv \frac{d}{dx}\left[p(x)\frac{dy}{dx}\right] + q(x)y$$

*with a corresponding equation*

$$\mathcal{L}[y] + \lambda r(x)y = 0$$

*where* $p > 0$, $r \geq 0$ *and* $p, q, r$ *continuous functions in the interval* $[a, b]$ *with boundary conditions*

$$a_1 y(a) + a_2 p(a) y'(a) = 0, \quad b_1 y(b) + b_2 p(b) y'(b) = 0$$

*where* $a_1^2 + a_2^2 \neq 0$ *and* $b_1^2 + b_2^2 \neq 0$

*The problem of finding a complex number* $\mu$ *such that the above BVP with* $\lambda = \mu$ *has a non-trivial solution is called a Sturm-Liouville Eigenvalue problem*

.

Consider the following Sturm-Liouville problem: $y'' + \lambda y = 0$ with $y(0) = y'(\pi) = 0$.
From elementary differential equations class we know that the general solution to this equation is of the form $A\cos(\mu x) + B\sin(\mu x)$. The boundary conditions are satisfied $\iff$ $A = 0$ and $B\cos(\mu x) = 0$. $B\cos(\mu\pi) = 0$ non trivially $\iff$ $\cos(\mu\pi) = 0$ $\iff$ $\mu = \frac{2n-1}{2}$ with $n \in \mathbb{Z}$. So, the eigenvalues are simply just $\frac{2n-1}{2}$ with $n \in \mathbb{Z}$ which is exactly the values that we get if we divide our output by $\pi$! and the reason this is better is because, we get all eigenvalues at the same time - as opposed to a root finding routine which would converge to each eigenvalue one by one!

This is obviously just a proof of concept, and can be easily applied to more complicated problems. Since our matrix is upper and lower Hessenberg by construction - it is fairly easy to increase the dimensionality of the problem without giving up a lot of computational power. I computed the taylor approximation upto 250 terms in my laptop, and was able to get results with percentage error of 21.632%. I could not run matrices with higher dimension because the taylor coefficients were very large and the python compiler was not able to handle big integers. The error computation was done manually by looking at the raw output which is presented below. Upon running the code, the console returned

$-17.49097221241405, -16.4999711223405, -15.498767212232405, -13.490971421241405,$
$-12.49344097221225, -11.06723626485919, -10.478147322942231, -9.497354545648344,$
$-8.500440775361506, -7.499953437871345, -6.50000410029158, -5.499999720198007,$
$-4.50000001301965, -3.4999999996546, -2.5000000000103864, -1.4999999999977824,$
$-0.49999999999992467, 0.4999999999994516, 1.5000000000030957, 2.4999999999695053,$
$3.5000000000758895, 4.500000006028772, 5.499999853254162, 6.500001375671566,$
$7.500006856158792, 8.499618868451943, 9.504351367661023, 10.519878582384374,$

11.06723626485919, 12.49344097221225, 13.490971421241405, 15.498767212232405, 16.87867431087975, 17.192646169383544

as eigenvalues. It is clear that all of these numbers are sufficiently close to $\frac{2n-1}{2}$ for $n \in \mathbb{Z}$. It is noticable that the approximation gets worse as we move towards the tails of the approximation, but it is still good enough, given that we maintain a maximum absolute error of 0.30735383061 towards the very end. The high % error can be attributed to us missing two eigenvalues in the list 14.5 and $-14.5$. This is because the outputs corresponding to these terms had small non-zero imaginary parts and thus were not able to pass through the filter. It is important to notice that these errors stem purely from the eigenvalue solver (especially for systems with very large coefficients) and optimizing the eigenvalue solver ought to rectify a lot of these errors that were observed. Perhaps an eigenvalue solver customized to these types of problems would perform better in comparison to a pre-pubilshed universal eigenvalue solver.

## 3.3 A more direct physics application

Now to consider an actual physical application. The quantum harmonic oscillator has solutions of the form

$$\Psi = \left(\frac{\alpha}{\pi}\right)^{1/4} \frac{1}{\sqrt{2^n n!}} H_n(y) e^{-y^2/2}$$

Where $H_n$ is the $n^{th}$ Hermite polynomial. It is given by

$$H_n = (-1)^n e^{x^2} \frac{d^n}{dx^n} e^{-x^2}$$

So, we naturally have $\Psi = 0 \iff H_n = 0$. This now becomes a root finding problem for polynomials! My motivation for considering this problem is that a lot of quantum systems are best approximated by harmonic oscillators, and the zeroes of the wave-function are places where the probability of observing the particle is zero. Plotted below is the fourth eigenstate solution to the quantum harmonic oscillator wavefunction.

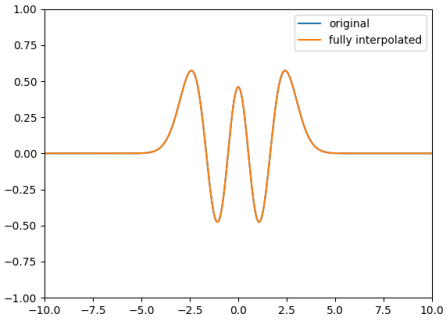


Figure 6: $\Psi_4$

It can be seen that $H_4(y) = 16y^4 - 48y^2 + 12$ which has nice closed form zeroes

$$\psi_i = \pm\sqrt{\frac{3}{2} \pm \sqrt{\frac{3}{2}}}$$

Presented below is the output that was obtained using this method

```
[-1.65068012 -0.52464762  0.52464762  1.65068012]
```

Figure 7: Output

We note that $\sqrt{\frac{3}{2} + \sqrt{\frac{3}{2}}} = 1.65068012388578$ and $\sqrt{\frac{3}{2} - \sqrt{\frac{3}{2}}} = 0.524647623275290$. So the absolute errors are:

- $|1.65068012388578 - 1.65068012000000| = 3.88 \times 10^{-9}$
- $|0.524647623275290 - 0.52464762000000| = 3.27 \times 10^{-9}$

So, absolute error is always less than $3.88 \times 10^{-9}$.

## 3.4 A variation of application

Suppose that we do not have an explicit closed form function but rather just data points. In this case, the process becomes more complicated. One can follow the algorithm presented below to deal with this case

- With only data points, one can
  - interpolate through the points
  - obtain a set of polynomials
    - $m \times n$ coefficients for $m^{th}$ order interpolation with $n$ data points
  - For each polynomial check if it changes sign in its respective domain
    - if it does, run the algorithm to find the root in that domain
    - if not, move to the next polynomial
- This is a painful process, and perhaps root-finding can do better here, if the function is well-behaved in the interested domain

Presented below is a proof of concept that this would indeed work. Consider $\Psi_4$ again and the cubic interpolation at $x = 1.1$. The interpolated polynomial returned 0.524647, with an absolute error of $|0.524647623275290 - 0.524647| = 6.23 \times 10^{-7}$ which is still a very good approximation!

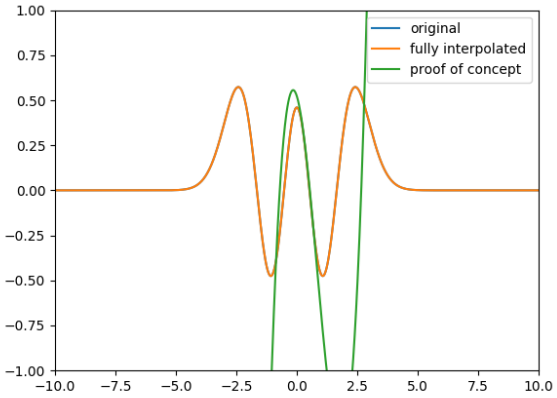


Figure 8: Output

# 4 Discussion of applications

It is important to note that while this method could be potentially stable for higher order matrices, it is simply a method to compute all roots in parallel. This is useful in the sense that the user does not have to bracket roots, or know the distribution of the zeroes of a function. It is also useful in the sense that a user need not run the code $n$-times in order to obtain the $n$ roots. The computational cost of the QR algorithm is $\frac{10}{3}n^3$. Whereas in NewtonRhapson method on average, assuming it takes $\log(n)$ steps to converge to a root with every step, we have $(n-1)n^2$ computations for computing $n-1$ derivatives, and $n$ multiplications. So total computational cost is $\log(n)(n-1)n$ every run. For $n$ runs, we have computational cost of $\sim \log(n)n^3$. So, QR is slightly better than NewtonRhapson root finding for $n$ roots.

Moreover, It can also produce complex roots that are often times crucial in physics applications, especially in areas such as electrical engineering. Scaling up this model to higher order matrices requires optimization of the eigenvalue solver. In a real-world scenario, this could be important in many aspects of quantum computing and technology in the nano-scales. As is noted in [3] "The relationship between the energy eigen-states of a simple harmonic oscillator and quantum computation comes by taking a finite subset of these states to represent quantum bits".

References

J. Elizondo, R. Geijn, F. Van Zee, G. Orti, Algorithms for reducing matrix to condensed form, web url: http://www.cs.utexas.edu/users/flame/pubs/flawn53.pdf

Horn, Roger A.; Charles R. Johnson (1985). Matrix Analysis. Cambridge, UK: Cambridge University Press. pp. 146–147. ISBN 0-521-30586-1. Retrieved 2010-02-10

Lu, Jun; Physical Realization of Harmonic Oscillator Quantum Computer, *Future Communication, Computing, Control and Management: Volume 1*, 2012, Springer Berlin Heidelberg.

Appendix 1

```
import numpy as np
import matplotlib.pyplot as plt
import random
from scipy import interpolate

#
    ------------------------------------------------------------------------------

# Finding zeroes for randomly generated polynomial
n = 15  # degree of polynomial
coeffs = [1]
f = 0
x = np.linspace(-10, 10, num=1000)
A = np.zeros((n, n))
# assigning random integer coefficients.
for i in range(n):
    coeffs.append(random.randrange(-10, 10))
print(coeffs)

# creating upper hessenberg matrix
for i in range(n):
    for j in range(n):
        if(j == n-1):
            A[i][j] = -coeffs[n-i]
        if(i == j+1):
            A[i][j] = 1

print(np.linalg.eig(A)[0])

# plotting the polynomial
for i in range(len(coeffs)):
    f += coeffs[i]*(x**(n-i))

plt.plot(x, f)
plt.show()

#
    ------------------------------------------------------------------------------


# Using taylor approximation of non-polynomial function to identify zeroes
#
    ------------------------------------------------------------------------------

coeffs = []
tempsum = 0

# Defining a factorial function


def fact(n):
    temp = 1
```

```
    for i in range(1, n+1):
        temp *= i
    return temp

# Getting the taylor polynomial


def p(x, nmax):
    tempsum = 0
    for n in range(nmax+1):
        nfact = fact(2*n)
        tempsum += (-1)**n * ((x**(2*n))/nfact)
    return tempsum

# Getting coefficients of the taylor polynomial (This couldve been avoided
    )


def get_coeffs(nmax):
    coeffs = []
    for n in range(nmax):
        if(n % 2 == 0):
            if(n % 4 == 0):
                coeffs.append(1/fact(n))
            else:
                coeffs.append(-1/fact(n))
        else:
            coeffs.append(0)
    return np.array(coeffs)


x = np.arange(-4*np.pi, 4*np.pi, 0.01)  # initalizing x values
y = np.cos(x)
t = p(x, 10)  # obtaining taylor polynomial for first 6 terms
degree = 150 # and obviously, with degree 12
coeffs = fact(degree)*get_coeffs(degree)  # making the polynomial monic
dim = len(coeffs)
C = np.zeros((dim, dim))
# creating the upper hessenberg matrix
for i in range(dim):
    for j in range(dim):
        if(j == dim-1):
            C[i][j] = -coeffs[i]
        if(i == j+1):
            C[i][j] = 1

ev = []
# solving for real eigenvalues
for i in np.linalg.eig(C)[0]:
    if(np.abs(i.imag) == 0):
        ev.append(i.real/np.pi)
        print("Zero", i.real, "Eigenvalue:", i.real/np.pi)

print(sorted(ev))
```

```
per_error = []
for i in ev:
    temp = int(i)+0.5
    tempcalc = 100*(np.abs(temp - i)/np.abs(temp))
    per_error.append(tempcalc)
print(np.average(per_error))

# plotting
plt.plot(x, y, label="cos(x)")
plt.plot(x, t, label="Approximation upto 12 terms")
plt.axis([-4*np.pi, 4*np.pi, -2.5, 2.5])
plt.legend()
plt.show()
#
    ------------------------------------------------------------------------------


# Rootfinding for the nth energy state of the Quantum Harmonic Oscillator
#
    ------------------------------------------------------------------------------


# Funtion to define all hermite polynomials


def Hermite(n, x):
    if(n == 0):
        return [1, [1]]
    if(n == 1):
        return [2*x, [1, 0]]
    if(n == 2):
        return [4*(x**2) - 2, [1, 0, (-1/2)]]
    if(n == 3):
        return [8*(x**3) - 12*x, [1, 0, (-12/8), 0]]
    if(n == 4):
        return [16*(x**4)-48*(x**2)+12, [1, 0, -48/16, 0, 12/16]]
    if(n == 5):
        return [32*(x**5) - 160*(x**3) + 120*x, [1, 0, -160/32, 0, 120/32,
    0]]
    if(n == 6):
        return [64*(x**6) - 480*(x**4) + 720*(x**2) - 120, [1, 0, -480/64,
    0, 720/64, 0 - 120/64]]
    if(n == 7):
        return [128*(x**7)-1344*(x**5) + 3360*(x**3) - 1680*x, [1, 0,
   -1344/128, 0, 3360/128, 0, -1680/128]]
    if(n == 8):
        return [256*(x**8) - 3584*(x**6) + 13440*(x**4) - 13440*(x**2) +
   1680, [1, 0, -3584/256, 0, 13440/256, 0, -13440/256, 0, 1680]]
    if(n == 9):
        return [512*(x**9)-9216*(x**7)+48384*(x**5) - 80640*(x**3) +
   30240*x, [1, 0, -9216/512, 0, 48384/512, 0, 80640/512, 0, 30240/512]]
    if(n == 10):
        return [1024*(y**10)-23040*(y**8)+161280*(y**6)-403200*(y**4)
   +302400*(y**2)-30240, [1, 0, -23040/1024, 0, 161280/1024, 0,
```

```
    -403200/1024, 0, 302400/1024, 0, -30240]]
144
145
146 # psi_n = 0 \implies H_n = 0. Solving for H_n = 0
147 coeffs = []
148 n = 10
149 y = []
150 x = np.linspace(-10, 10, num=1000000)
151 B = np.zeros((n, n))
152 hbar = 1  # because theoretical physics
153 m = 1  # normalized mass
154 omega = np.sqrt(1/m)  # k = 1 in the oscillator
155
156 # set up for psi_n
157 alpha = (m*omega)/hbar
158 y = np.sqrt(alpha)*x
159
160 # defining psi_n
161 psi_n = ((alpha/np.pi)**(1/4))*(1/np.sqrt((2**n)*np.math.factorial(n))) *
    \
162     (Hermite(n, y)[0])*(np.e**(-0.5*y**2))
163
164 # obtaining the coefficients for the Hermite polynomials
165 coeffs = Hermite(n, y)[1]
166 # creating upper hessenberg matrix
167 for i in range(n):
168     for j in range(n):
169         if(j == n-1):
170             B[i][j] = -coeffs[n-i]
171         if(i == j+1):
172             B[i][j] = 1
173 # solving for eigenvalues.
174 print(np.linalg.eig(B)[0])
175
176 # proof of concept! Very tedious to actually perform.
177 # interpolate through n data points to obtain n cubic functions.
178 # In each domain see if the resp. cubic function changes sign
179 # if it does, do rootfinding with the restricted domain.
180 # Repeat process for all cubic functions to obtain all roots.
181 # Can be optimized with further work.
182
183 #fitted = 0
184 #spl1 = interpolate.CubicSpline(y, psi_n)
185 #for i in range(4):
186     #fitted += spl1.c[i, 500]*(y**(3-i))
187
188 # plotting
189 plt.plot(y, psi_n, label="original")
190 #plt.plot(y, fitted, label="proof of concept")
191 plt.axis([-10, 10, -1, 1])
192 plt.legend()
193 plt.show()
```