Chapter 6

Roots: Open Methods

Chapter Objectives

- Recognize the difference between bracketing and open methods for root location.

- Solve a roots problem with the Newton-Raphson method and recognize the concept of quadratic convergence.

- Use the secant method.

- Use root_scalar and fsolve functions of scipy.optimize modul, to estimate roots.

Open Methods

- *Open methods* differ from bracketing methods, in that open methods require only a single starting value or two starting values that do not necessarily bracket a root.

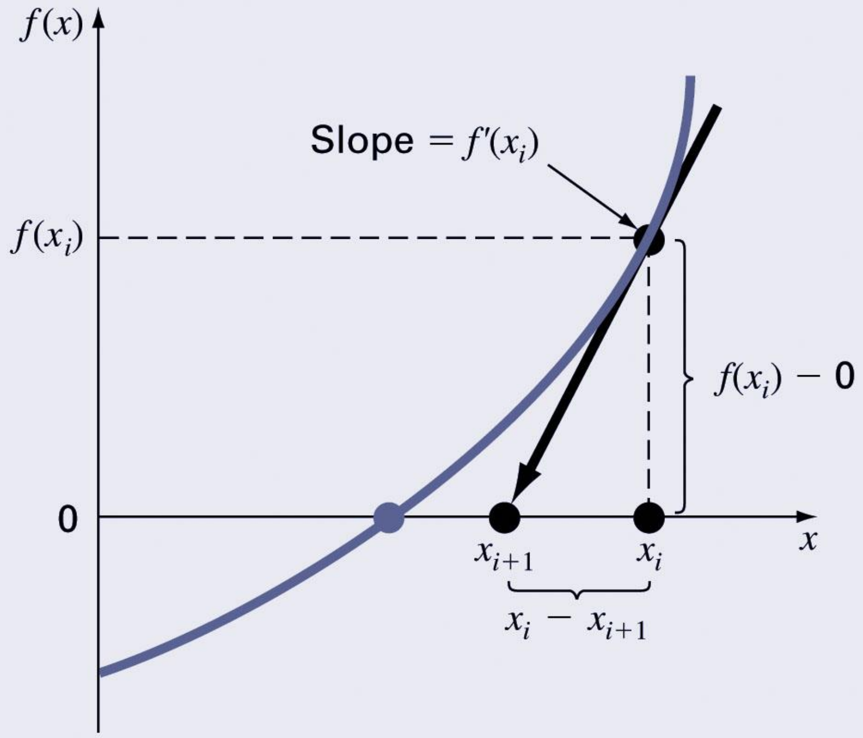
- Open methods may diverge as the computation progresses, but when they do converge, they usually do so much faster than bracketing methods.

Newton-Raphson Method

Based on forming the tangent line to the $f(x)$ curve at some guess x, then following the tangent line to where it crosses the x-axis.

$$f'(x_i) = \frac{f(x_i) - 0}{x_i - x_{i+1}}$$

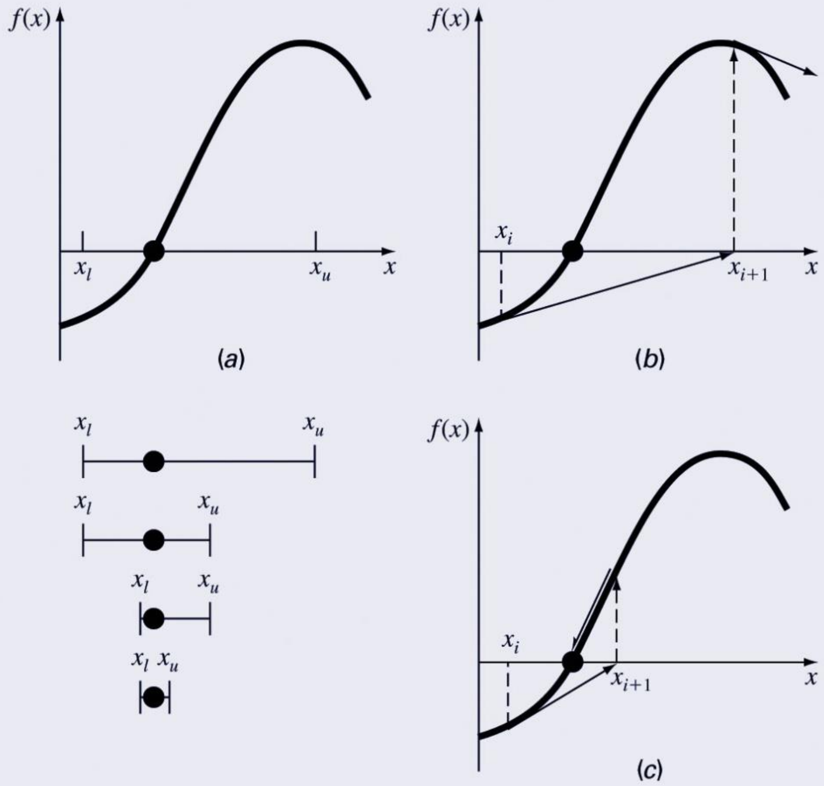
$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$



Alternatively

- Recall Taylor expansion
- $f(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1} - x_i)+..$
- If $f(x_{i+1}) = 0$, then
- $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$

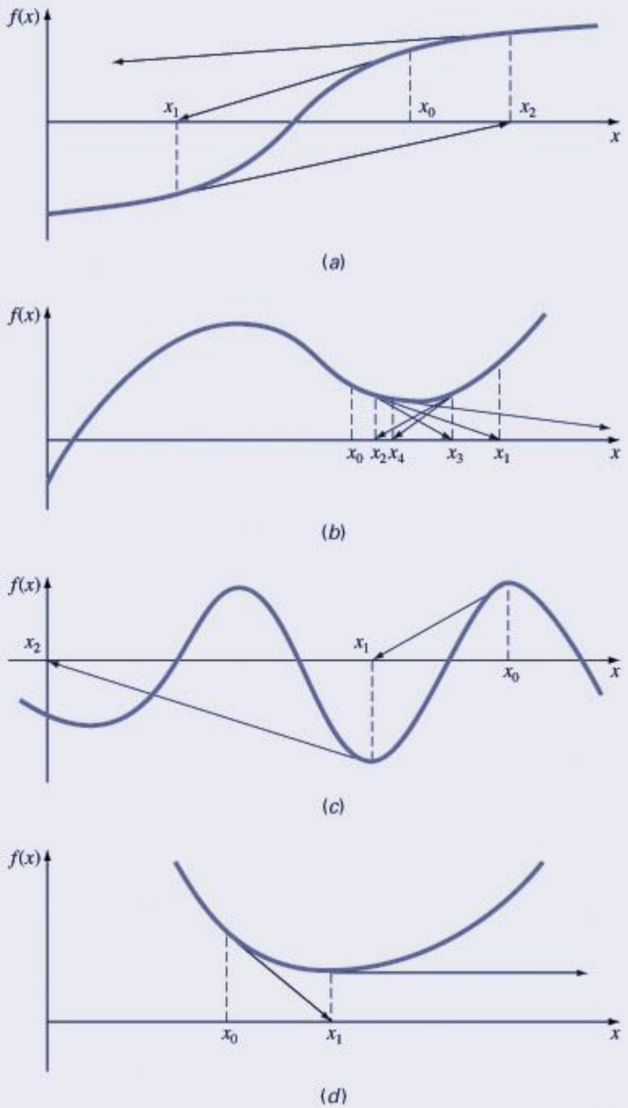
Graphical Comparison of Methods



a) Bracketing method
b) Diverging open method
c) Converging open method - note speed!

Pros and Cons

- Pro: The error of the (i+1)th iteration is roughly proportional to the square of the error of the ith iteration - this is called *quadratic convergence*
- Con: Some functions show slow or poor convergence
- Con: Solution may converge to unexpected root.



Errors

- Taylor expansion to second order
- $f(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1} - x_i) + \frac{1}{2}f''(x_i)(x_{i+1} - x_i)^2 + ..$
- Setting $f(x_{i+1}) = 0$ gives
- $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} - \frac{f''(x_i)}{2f'(x_i)}(x_{i+1} - x_i)^2$
- Error is $R = \left| -\frac{f''(x_i)}{2f'(x_i)}(x_{i+1} - x_i)^2 \right|$
- *R* decreases in quadratic form as solution is approached

Example

Find zero of $f(x) = x^2 - x$ given initial point $x_i = 0.6$;

$\boldsymbol{x_i}$	$\boldsymbol{f(x_i)}$	$\boldsymbol{f'(x_i)}$	$\boldsymbol{x_{i+1}}$	$\boldsymbol{R}$
0.600	-0.240	0.200	1.800	7.200
1.800	1.440	2.600	1.246	0.118
1.246	0.307	1.492	1.041	0.028
1.041	0.043	1.082	1.002	0.001
1.002	0.002	1.004	1.000	0.000

Note $x_i = 0.5$ diverge, $x_i = 0.4$ leads to x = 0

Example

Find root of cos(x) with x_initial = 0.01

x_i	$f(x_i)$	$f'(x_i)$	x_{i+1}	R
0.01	1.00	-0.01	100.00	5*10^5
100.00	0.87	0.50	98.28	3.59
98.28	-0.63	0.78	99.09	0.73
99.09	0.13	0.99	98.96	0.001
98.96	-7*10^(-4)	1.00	98.96	0.000

- Expected answer is the closest root at x =pi/2= 1.57

```python
def newtraph(f,fp,x0,Ea=1.e-7,maxit=30):
    """
    This function solves f(x)=0 using the Newton-Raphson method.
    The method is repeated until either the relative error
    falls below Ea (default 1.e-7) or reaches maxit (default 30).
    Input:
    f = name of the function for f(x)
    fp = name of the function for f'(x)
    x0 = initial guess for x
    Ea = relative error threshold
    maxit = maximum number of iterations
    Output:
    x1 = solution estimate
    f(x1) = equation error at solution estimate
    ea = relative error
    i+1 = number of iterations
    """
    for i in range(maxit):
        x1 = x0 - f(x0)/fp(x0)
        ea = abs((x1-x0)/x1)
        if ea < Ea: break
        x0 = x1
    return x1,f(x1),ea,i+1
```

Example

- Download newton in your working directory
- Open a file and name it **example_newton**
- Dthe function

$$f(z) = \sqrt{z_0^2 - z^2}\,\sin(z) + z\cos(z)$$

and its derivative

- Set $z_0 = 10$ before you define the function.

- Use newton to find root for initial guess 5.5

```python
import numpy as np

def example_newton():
    z0 = 10
    # Define the function f and its derivative fd
    f = lambda z: np.sqrt(z0**2 - z**2) * np.sin(z) + z * 
np.cos(z)
    fd = lambda z: -z / np.sqrt(z0**2 - z**2) * np.sin(z) 
+ np.sqrt(z0**2 - z**2) * np.cos(z) + np.cos(z) - z * 
np.sin(z)

    ze = 5.5  # Initial guess

    # Call the newton function (defined previously)
    zr, fz, er, n, = newtraph(f, fd, ze)

    # Print results
    print(f'zr = {zr:.4f}, f(z) = {fz:.4f},   er = 
{er:e},  n = {n}')

# Ensure the newton function is present when you run this 
code
```

zr =5.6792,   er=4.082964e-007, n=3
Recall bisect method needed 18 iterations

Secant Methods

- A potential problem in implementing the Newton-Raphson method is the evaluation of the derivative - there are certain functions whose derivatives may be difficult or inconvenient to evaluate.

- For these cases, the derivative can be approximated by a backward finite divided difference:

$$f'(x_i) \cong \frac{f(x_{i-1}) - f(x_i)}{x_{i-1} - x_i}$$

Secant Methods (cont)

- Substitution of this approximation for the derivative to the Newton-Raphson method equation gives:

$$x_{i+1} = x_i - \frac{f(x_i)\left(x_{i-1} - x_i\right)}{f(x_{i-1}) - f(x_i)}$$

- Note - this method requires *two* initial estimates of *x* but does *not* require an analytical expression of the derivative.

```python
def secant(f, x0, x1, Ea=1.e-7, maxit=30):
    """
    This function solves f(x)=0 using the Secant method.
    The method is repeated until either the relative error
    falls below Ea (default 1.e-7) or reaches maxit (default 30).
    Input:
    f = name of the function for f(x)
    x0 = initial guess for x (first point)
    x1 = initial guess for x (second point)
    Ea = relative error threshold
    maxit = maximum number of iterations
    Output:
    x2 = solution estimate
    f(x2) = equation error at solution estimate
    ea = relative error
    i+1 = number of iterations
    """
    for i in range(maxit):
        if f(x1) - f(x0) == 0:  # Prevent division by zero
            raise ValueError("Denominator in secant method became zero.")

        x2 = x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0))  # Secant formula
        ea = abs((x2 - x1) / x2) if x2 != 0 else float('inf')  # Avoid division by
zero

        if ea < Ea:
            break

        x0, x1 = x1, x2  # Update guesses for next iteration

    return x2,  ea, i + 1
```

Example

- Download **secant** in your working directory
- Open a file and name it **example_secant**
- Define the function

$$f(z) = \sqrt{z_0^2 - z^2}\sin(z) + z\cos(z)$$

- Set $z_0 = 10$ before you define the function.
- Use secant to find root for initial guess 5.0, 5.5

```python
# Define the example function using Secant method
def example_secant():
    z0 = 10

    # Define the function f
    f = lambda z: np.sqrt(z0**2 - z**2) * np.sin(z) + z * np.cos(z)

    zrold = 5.0  # First initial guess
    zr = 5.5     # Second initial guess

    # Call the Secant method
    zr, ea, n = secant(f, zrold, zr)

    # Print the results
    print(f'zr = {zr:.4f},  n = {n}')
```

```
zr = 5.6792, n = 4
```

Python Functions for Root Finding

- scipy.optimize.newton: Implements the Newton-Raphson method. It requires a function and its derivative.
- scipy.optimize.bisect: Implements the bisection method, which is useful for continuous functions and requires two initial guesses that bracket the root.
- scipy.optimize.root: A more general function that can handle multiple root-finding algorithms.
- scipy.optimize.fsolve: Uses a numerical method to find roots of a system of equations.

```python
from scipy.optimize import newton

def f(x):
    return x**2 - 2  # Example function

root = newton(f, x0=1)
```

```python
from scipy.optimize import fsolve

def f(x):
    return x**2 - 2

root = fsolve(f, x0=1)
```

```python
from scipy.optimize import bisect

def f(x):
    return x**2 - 2

root = bisect(f, 0, 2)
```

```python
from scipy.optimize import root

def f(x):
    return x**2 - 2

sol = root(f, x0=1)
root = sol.x
```

Scipy’s `fsolve` Function

- To solve systems of nonlinear equations in Python, particularly multiple equations, you can use the scipy.optimize.fsolve function from the SciPy library.

- This function is well-suited for finding roots of a set of equations simultaneously.

- You can still use the fsolve when dealing with a single nonlinear equation that may have multiple zeros

- Important Consideration:The choice of initial guesses is critical; they should be chosen based on the function's behavior, potentially using graphical analysis

Example: Several(Multiple) roots

Consider the equation

$$f(x) = x^3 - x \ = 0$$

```
from scipy.optimize import fsolve
import numpy as np

# Define the function
def f(x):
    return x**3 - x  # Example function with multiple roots

# Initial guesses for different roots
initial_guesses = [-2, 0, 2]

# Find roots
roots = []
for guess in initial_guesses:
    root = fsolve(f, guess)
    # Avoid duplicates by checking if the root is already found (considering a small tolerance)
    if not any(np.isclose(root, r, atol=1e-5) for r in roots):
        roots.append(root[0])

# Output the results
print("Found roots:", roots)
```

Example: Multiple Equations

- Consider the equations

$$2x - y - e^{-x} = 0$$
$$-x + 2y - e^{-y} = 0$$

```
from scipy.optimize import fsolve
import numpy as np

# Define the system of equations
def equations(vars):
    x, y = vars
    eq1 = 2*x - y - np.exp(-x)  # 2x - y - e^(-x) = 0
    eq2 = -x + 2*y - np.exp(-y)  # -x + 2y - e^(-y) = 0
    return [eq1, eq2]

# Initial guess for (x, y)
initial_guess = (1, 1)

# Solve the system of equations
solution = fsolve(equations, initial_guess)

# Output the results
x_solution, y_solution = solution
print(f"Solution: x = {x_solution}, y = {y_solution}")
```

Example: Finite Well

- Consider a finite well of width a of potential
- $V(x) = \begin{cases} \infty & x < 0 \\ 0 & 0 \le x \le a \\ V_0 & x > a \end{cases}$
- The condition for obtaining the bound state eigenvalues in this case is
- $\tan(z) = -\frac{z}{\sqrt{z_0^2 - z^2}}$,
- where $z = ka$, $z_0 = \sqrt{\frac{2mV_0}{\hbar^2}}a$, $k = \sqrt{\frac{2mE}{\hbar^2}}$

Example: Finite Well

- Since $\tan(z)$ can be infinite for $z = \frac{(2n+1)\pi}{2}$, we convert the condition to
- $\sqrt{z_0^2 - z^2}\sin(z) = -z\cos(z)$
- The solutions are the roots of the function
- $f(z) = \sqrt{z_0^2 - z^2}\sin(z) + z\cos(z)$

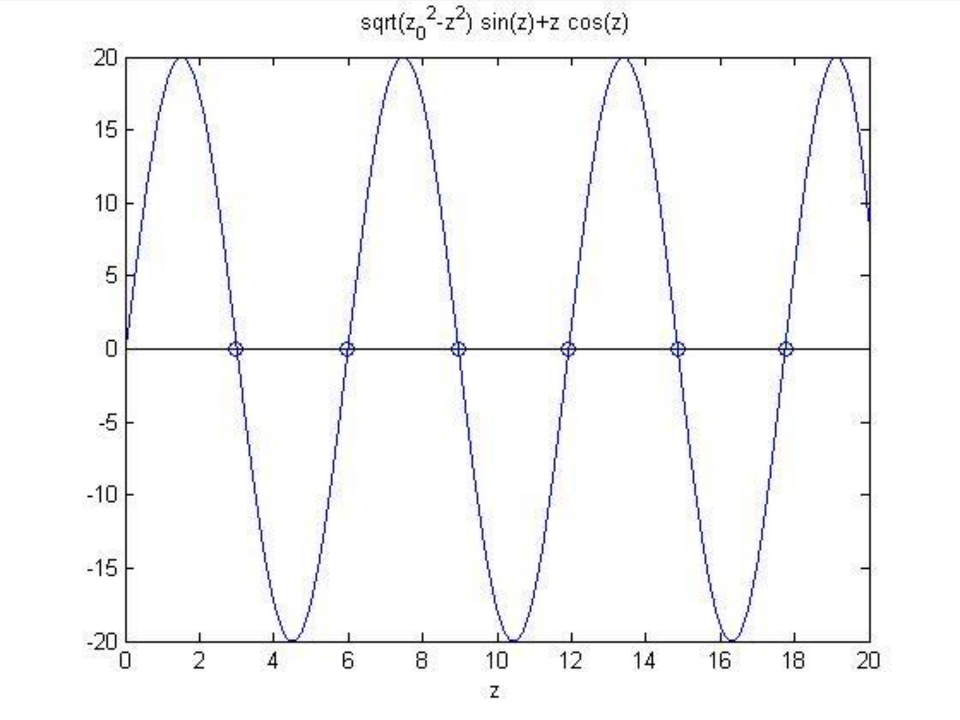
Example: Finite Well

- Set z0=20
- Define the function
- Use incsearch to estimate initial guesses
- Use fsolve to find roots

Example: Finite Well

- First draw the graph of $f(z)$ for a given z_0

- Recall the number of bound states is given by $N = \text{ceil}\left(\frac{z_0}{\pi} - \frac{1}{2}\right) = 6$, for $z_0 = 20$. This gives



Example: Finite Well

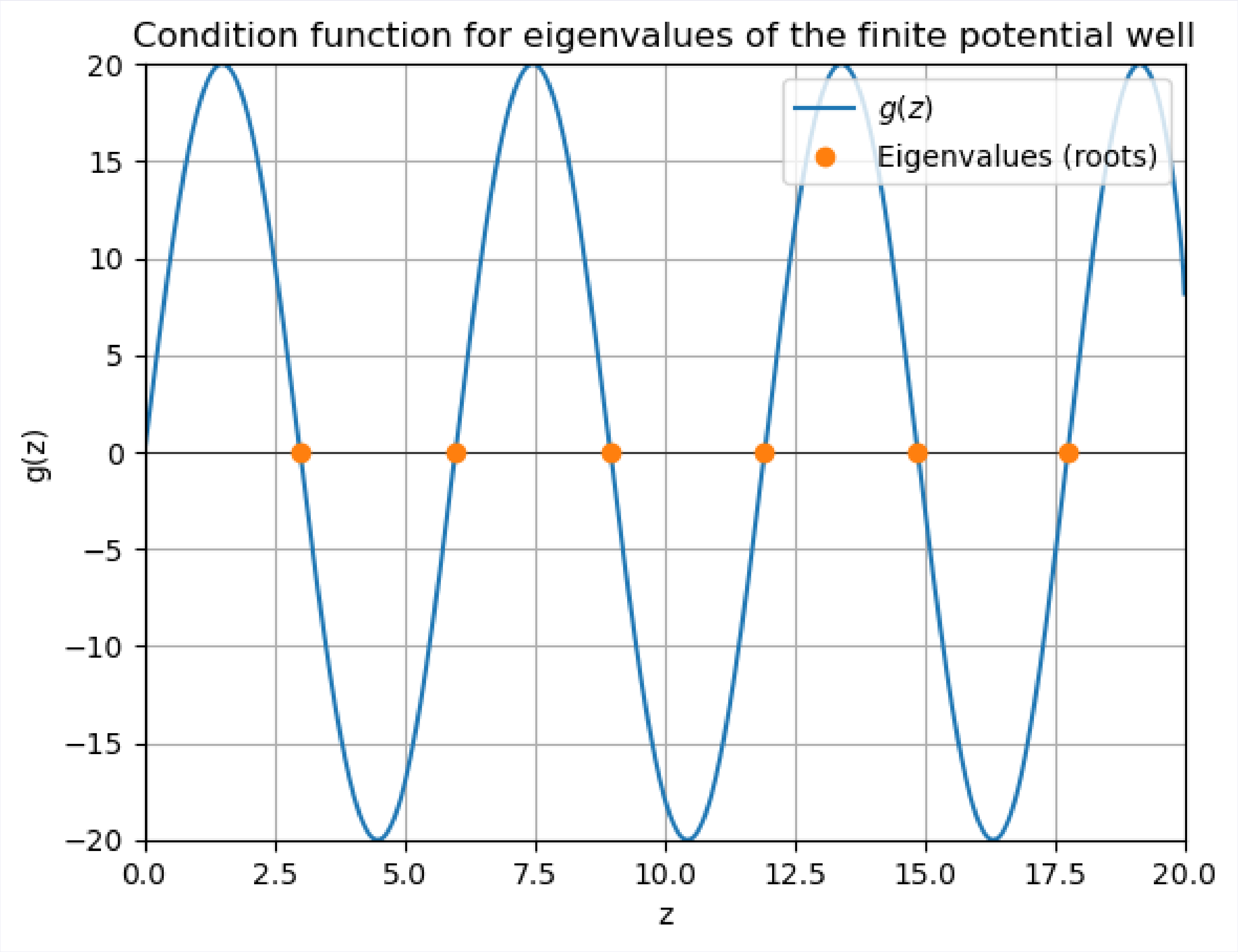
- The numerical solutions for z_n are

n	z_n
1	2.9915
2	5.9796
3	8.9602
4	11.9274
5	14.8697
6	17.7569

Note the eigenvalues are given by $E_n = \frac{\hbar^2 z_n^2}{2ma^2}$

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import fsolve

def finite_well():
    # Define constants
    a = 1  # Well width
    z0 = 20  # Measure of the strength of the potential V0

    # Define the condition function for eigenvalues
    def g(z):
        return np.sqrt(z0**2 - z**2) * np.sin(z) + z * np.cos(z)

    # Function to search for initial guesses for zeros
    def incsearch(func, xmin, xmax, ns=50):
        # Create equally spaced points between xmin and xmax
        x = np.linspace(xmin, xmax, ns)
        f = []
        for k in range(ns - 1):
            f.append(func(x[k]))

        xb = []  # Empty list if no sign change detected

        # Loop through points to find sign changes
        for k in range(ns - 1):
            if func(x[k]) * func(x[k + 1]) < 0:  # Check for sign change
                xb.append((x[k], x[k + 1]))  # save the bracketing pair

        # Output the result
        if not xb:
            print('No brackets found')
        return xb  # Return only the list of intervals

    # Estimate array of initial guesses
    xb = incsearch(g, 0.1, z0)  # Gives bounds where the zeros lie
    x0 = [(x[0] + x[1]) / 2 for x in xb]  # Midpoints of the intervals as initial guesses

    # Use fsolve to find the eigenvalues (roots)
    z = fsolve(g, x0)

    # Compute the number of bound eigenvalues
    ne = len(z)
    nee = int(np.ceil(z0 / np.pi - 0.5))

    # Print results
    print(f'ne (from length) = {ne}, nee (from ceil) = {nee}')
```

```
    # Plot the function g(z) and the
eigenvalues (roots)
    plt.figure()
    z_vals = np.linspace(0, z0, 1000)
    g_vals = g(z_vals)

    plt.plot(z_vals, g_vals,
label=r'$g(z)$')
    plt.axhline(0, color='black',
linewidth=0.5)
    plt.plot(z, np.zeros_like(z), 'o',
label='Eigenvalues (roots)')

    plt.xlim(0, z0)
    plt.ylim(-z0, z0)
    plt.title('Condition function for
eigenvalues of the finite potential well')
    plt.xlabel('z')
    plt.ylabel('g(z)')
    plt.legend(loc='upper right')
    plt.grid(True)
    plt.show()

# Run the finite_well function
finite_well()
```


Condition function for eigenvalues of the finite potential well
g(z)
Eigenvalues (roots)
20
15
10
5
0
−5
−10
−15
−20
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
z
g(z)

Homework (problem 6.21 Chapra)

- Write the function $f(\theta_0)$ that determines the initial angles
- Plot $f(\theta_0)$ for $0.5 \le \theta_0 \le 1.0$ in figure1. Add grid
- Use incsearch to find the possible bounds of the roots of $f(\theta_0)$
- Use fsolve to find the values θ_0
- Plot $y\ vs\ x$ for all possible values of θ_0 in figure2 for $0 \le x \le 90$. Add grid