Solutions to Homework 3

Question 1: 10 points.

Problem Statement: Figure 1 plots the function

$$f(x) = \left[\frac{x}{2} - \frac{1}{x}\right]^3 + \left[\frac{x}{2} - \frac{1}{x}\right] - 30. \tag{1}$$

over the range $[-5, 8]$.

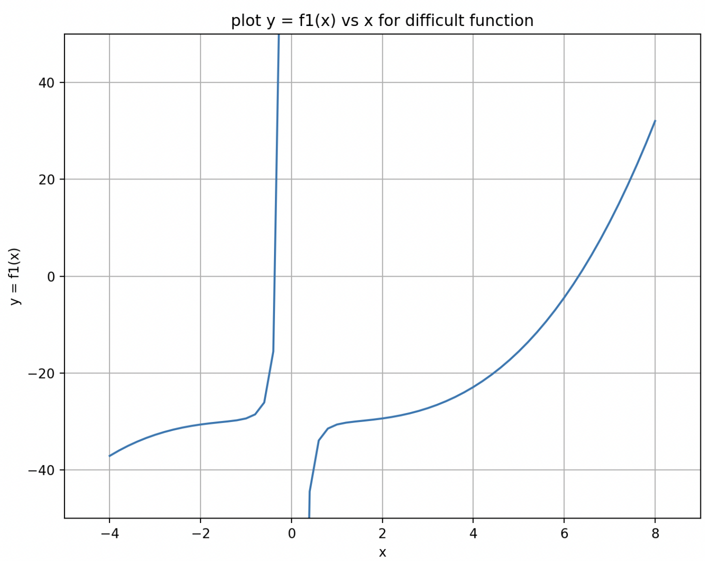

Figure 1: Plot y = f(x) vs x.

Theoretical considerations indicate that equation 1 has two real roots:

$$[r_1, r_2] = \frac{6}{2} \pm \frac{\sqrt{44}}{2} \tag{2}$$

Now suppose that we have Figure 1, and can see that the two roots lie in the intervals $[-1, 0]$ and $[6, 8]$, but for some reason don't know about equation 2.

Write a Python program that:

1. Uses the **method of bisection** to compute the lower and upper roots to equation
2. Computes the roots with the **method of Newton Raphson** iteration.
3. Investigates behavior of **Newton Raphson iteration** when we seek solutions to the lower root and the starting value x_o = -2.

Briefly discuss the strengths and weaknesses of bisection and newton raphson algorithms in terms of reliability and efficiency of computations.

Python Source Code:

```
# ===============================================================================
# TestNewtonRaphson05.py: Use methods of bisection and newton raphson to
# compute roots to:
#
#  --- f(x) = (x/2 - 1/x)**3 + (x/2 - 1/x) - 30 = 0.0
#
# Written By: Mark Austin                                             April 2025
# ===============================================================================

import math;
import Solutions;
import numpy as np
import matplotlib.pyplot as plt

# Function: f(x) = (x/2 - 1/x)**3 + (x/2 - 1/x) - 30

def f1(x):
    u = x/2 - 1/x;
    return u**3 + u - 30;

def df1(x):
    u  = x/2 - 1/x;
    du = 1/2 + 1/x**2;
    return 3*u*u*du + du;

# main method ...

def main():
    print("--- Enter TestNewtonRaphson05.main()          ... ");
    print("--- ========================================= ... ");

    print("--- ");
    print("--- Part 1: Generate plot f1(x) ... ");
    print("--- =========================== ... ");
    print("--- ");
```

```
print("==============================");
print("          Coord         Value");
print("            (x)           (y)");
print("==============================");

# Define problem parameters.

xcoord = np.linspace( -4, 8, 61)

# Create list for y coordinates ...

ycoord = [];

# Traverse xcoord array and compute y values ...

for x in xcoord:
    result = f1(x)
    print("        {:7.2f}   {:12.3f}".format(x,result) );
    ycoord.append(result)

print("==============================");
print("");

# Plot y=f(x) vs x for test function ...

plt.plot(xcoord,ycoord)
plt.title('plot y = f1(x) vs x for difficult function')
plt.ylabel('y = f1(x)')
plt.xlabel('x')
plt.xlim(  -5,  9)
plt.ylim( -50, 50)
plt.grid(True)
plt.show()

print("--- ");
print("--- Part 2: Method of Bisection to solve f1(x) = 0 for upper root ... ");
print("--- ============================================================ ... ");

# Initialize problem setup ...

a =  6.0; b = 9.0;
tolerance     = 0.00000001
maxiterations = 100

print("--- Inputs:")
print("---   a = {:5.2f} ...".format(a) )
print("---   b = {:5.2f} ...".format(b) )
print("---   tolerance      = {:16.8f} ...".format(tolerance) )
print("---   max iterations = {:16.8f} ...".format(maxiterations) )

# Compute roots to equation ...

print("--- Execution:")
root, i, converged = Solutions.bisection(f1, a, b, tolerance, maxiterations )
```

```
# Summary of computations ...

print("--- Output:")
print("---   root = {:10.5f} ...".format(root) )
print("---   f(root) --> {:12.5e} ...".format( f1(root)) )
print("---   no iterations = {:d} ...".format(i) )
print("---   converged: {:s} ...".format( str(converged) ) )

print("--- ");
print("--- Part 3: Method of Bisection to solve f1(x) = 0 for lower root ... ");
print("--- ============================================================== ... ");

# Initialize problem setup ...

a =  -2.0; b =  -0.1;
tolerance     = 0.00000001
maxiterations = 100

print("--- Inputs:")
print("---   a = {:5.2f} ...".format(a) )
print("---   b = {:5.2f} ...".format(b) )
print("---   tolerance      = {:16.8f} ...".format(tolerance) )
print("---   max iterations = {:16.8f} ...".format(maxiterations) )

# Compute roots to equation ...

print("--- Execution:")
root, i, converged = Solutions.bisection(f1, a, b, tolerance, maxiterations )

# Summary of computations ...

print("--- Output:")
print("---   root = {:10.5f} ...".format(root) )
print("---   f(root) --> {:12.5e} ...".format( f1(root)) )
print("---   no iterations = {:d} ...".format(i) )
print("---   converged: {:s} ...".format( str(converged) ) )

print("--- ");
print("--- Part 4: Use Newton Raphson to solve f1(x) = 0 for upper root: x0 = 8  ... ");
print("--- ====================================================================== ... ");

# Initialize problem setup ...

x0 = 8.0;
tolerance     = 0.00000001
maxiterations = 100

print("--- Inputs:")
print("---   x0 = {:5.2f} ...".format(x0) )
print("---   tolerance      = {:16.8f} ...".format(tolerance) )
print("---   max iterations = {:16.8f} ...".format(maxiterations) )

# Compute roots to equation ...
```

```
print("--- Execution:")
root, i, converged = Solutions.newtonraphson(f1, df1, x0, tolerance, maxiterations )

# Summary of computations ...

print("--- Output:")
print("---   root = {:16.8f} ...".format(root) )
print("---   f(root)   --> {:16.8e} ...".format(   f1(root)) )
print("---   df(root)  --> {:16.8e} ...".format(  df1(root)) )
print("---   no iterations = {:d} ...".format(i) )
print("---   converged: {:s} ...".format( str(converged) ) )

print("--- ");
print("--- Part 5: Use Newton Raphson to solve f1(x) = 0 for lower root: x0 = -0.1 ... ");
print("--- ====================================================================== ... ");

x0 = -0.10;
print("--- Inputs:")
print("---   x0 = {:5.2f} ...".format(x0) )
print("---   tolerance      = {:8.5f} ...".format(tolerance) )
print("---   max iterations = {:8.2f} ...".format(maxiterations) )

# Compute roots to equation ...

print("--- Execution:")
root, i, converged = Solutions.newtonraphson(f1, df1, x0, tolerance, maxiterations )

# Summary of computations ...

print("--- Output:")
print("---   root = {:16.8f} ...".format(root) )
print("---   f(root)   --> {:16.8e} ...".format(   f1(root)) )
print("---   df(root)  --> {:16.8e} ...".format(  df1(root)) )
print("---   no iterations = {:d} ...".format(i) )
print("---   converged: {:s} ...".format( str(converged) ) )

print("--- ");
print("--- Part 6: Solve f1(x) = 0 for lower root: x0 = -2.0 ... ");
print("--- ================================================= ... ");

x0 = -2.00; # <-- This could converge to the wrong root ...

print("--- Inputs:")
print("---   x0 = {:5.2f} ...".format(x0) )
print("---   tolerance      = {:8.5f} ...".format(tolerance) )
print("---   max iterations = {:8.2f} ...".format(maxiterations) )

# Compute roots to equation ...

print("--- Execution:")
root, i, converged = Solutions.newtonraphson(f1, df1, x0, tolerance, maxiterations )

# Summary of computations ...

print("--- Output:")
```

```
    print("---   root = {:16.8f} ...".format(root) )
    print("---   f(root)   --> {:16.8e} ...".format(   f1(root)) )
    print("---   df(root)  --> {:16.8e} ...".format(  df1(root)) )
    print("---   no iterations = {:d} ...".format(i) )
    print("---   converged: {:s} ...".format( str(converged) ) )

    print("--- ========================================= ... ");
    print("--- Leave TestNewtonRaphson05.main()          ... ");

# call the main method ...

main()
```

Abbreviated Output:

```
--- Enter TestNewtonRaphson05.main()          ...
--- ========================================= ...
---
--- Part 1: Generate plot f1(x) ...
--- =========================== ...
---
==============================
          Coord          Value
            (x)            (y)
==============================
          -4.00        -37.109
          -3.80        -36.022
          -3.60        -35.049

... lines of output removed ...

          -0.20         92.549
           0.00           -inf
           0.20       -152.549

... lines of output removed ...

           7.60         23.036
           7.80         27.431
           8.00         32.061
==============================

---
--- Part 2: Method of Bisection to solve f1(x) = 0 for upper root ...
--- ============================================================= ...
--- Inputs:
---   a =  6.00 ...
---   b =  9.00 ...
---   tolerance      =       0.00000001 ...
---   max iterations =     100.00000000 ...
--- Execution:
---   Initial Conditions:
---   f(a) --> -4.42130e+00 ...
```

```
---   f(b) -->  5.89292e+01 ...
---   Main Loop for Root Computation:
---   Iteration 000: dx = 1.50000e+00, x =  7.5000000e+00, f(x) -->  2.0923671e+01 ...
---   Iteration 001: dx = 7.50000e-01, x =  6.7500000e+00, f(x) -->  6.8266819e+00 ...
---   Iteration 002: dx = 3.75000e-01, x =  6.3750000e+00, f(x) -->  8.6631967e-01 ...
---   Iteration 003: dx = 1.87500e-01, x =  6.1875000e+00, f(x) --> -1.8591128e+00 ...

... lines of output removed ...

---   Iteration 027: dx = 1.11759e-08, x =  6.3166248e+00, f(x) -->  5.5866085e-08 ...
---   Iteration 028: dx = 5.58794e-09, x =  6.3166248e+00, f(x) --> -2.6286394e-08 ...
---   Iteration 029: dx = 2.79397e-09, x =  6.3166248e+00, f(x) -->  1.4789855e-08 ...
---   Iteration 030: dx = 1.39698e-09, x =  6.3166248e+00, f(x) --> -5.7482694e-09 ...
--- Output:
---   root =    6.31662 ...
---   f(root) --> -5.74827e-09 ...
---   no iterations = 30 ...
---   converged: True ...
---
--- Part 3: Method of Bisection to solve f1(x) = 0 for lower root ...
--- ============================================================= ...
--- Inputs:
---   a = -2.00 ...
---   b = -0.10 ...
---   tolerance      =       0.00000001 ...
---   max iterations =     100.00000000 ...
--- Execution:
---   Initial Conditions:
---   f(a) --> -3.06250e+01 ...
---   f(b) -->  9.65025e+02 ...
---   Main Loop for Root Computation:
---   Iteration 000: dx = 9.50000e-01, x = -1.0500000e+00, f(x) --> -2.9494556e+01 ...
---   Iteration 001: dx = 4.75000e-01, x = -5.7500000e-01, f(x) --> -2.5489449e+01 ...
---   Iteration 002: dx = 2.37500e-01, x = -3.3750000e-01, f(x) --> -5.3896170e+00 ...
---   Iteration 003: dx = 1.18750e-01, x = -2.1875000e-01, f(x) -->  6.3301192e+01 ...

... lines of output removed ...

---   Iteration 029: dx = 1.76951e-09, x = -3.1662479e-01, f(x) -->  3.3320152e-07 ...
---   Iteration 030: dx = 8.84756e-10, x = -3.1662479e-01, f(x) -->  7.3704019e-08 ...
---   Iteration 031: dx = 4.42378e-10, x = -3.1662479e-01, f(x) --> -5.6044733e-08 ...
---   Iteration 032: dx = 2.21189e-10, x = -3.1662479e-01, f(x) -->  8.8296375e-09 ...
--- Output:
---   root =   -0.31662 ...
---   f(root) -->  8.82964e-09 ...
---   no iterations = 32 ...
---   converged: True ...
---
--- Part 4: Use Newton Raphson to solve f1(x) = 0 for upper root: x0 = 8  ...
--- ===================================================================== ...
--- Inputs:
---   x0 =  8.00 ...
---   tolerance      =       0.00000001 ...
---   max iterations =     100.00000000 ...
--- Execution:
```

```
---   Initial Conditions:
---     x0     -->  8.00000e+00 ...
---     f(x0)  -->  3.20605e+01 ...
---     df(x0) -->  2.37429e+01 ...
---   Main Loop for Newton Raphson Iteration:
---   Iteration 001: dx = -1.35032e+00, x =  6.64968e+00, f(x) -->  5.16401e+00 ...
---   Iteration 002: dx = -3.16382e-01, x =  6.33330e+00, f(x) -->  2.45777e-01 ...
---   Iteration 003: dx = -1.66280e-02, x =  6.31667e+00, f(x) -->  6.56468e-04 ...
---   Iteration 004: dx = -4.46517e-05, x =  6.31662e+00, f(x) -->  4.72551e-09 ...
---   Iteration 005: dx = -3.21425e-10, x =  6.31662e+00, f(x) -->  0.00000e+00 ...
--- Output:
---   root =       6.31662479 ...
---   f(root)   -->   0.00000000e+00 ...
---   df(root)  -->   1.47017588e+01 ...
---   no iterations = 5 ...
---   converged: True ...
---
--- Part 5: Use Newton Raphson to solve f1(x) = 0 for lower root: x0 = -0.1 ...
--- ======================================================================= ...
--- Inputs:
---   x0 = -0.10 ...
---   tolerance      =  0.00000 ...
---   max iterations =   100.00 ...
--- Execution:
---   Initial Conditions:
---     x0     --> -1.00000e-01 ...
---     f(x0)  -->  9.65025e+02 ...
---     df(x0) -->  2.99498e+04 ...
---   Main Loop for Newton Raphson Iteration:
---   Iteration 001: dx = -3.22215e-02, x = -1.32221e-01, f(x) -->  3.98859e+02 ...
---   Iteration 002: dx = -4.07553e-02, x = -1.72977e-01, f(x) -->  1.60365e+02 ...
---   Iteration 003: dx = -4.80997e-02, x = -2.21076e-01, f(x) -->  6.03415e+01 ...
---   Iteration 004: dx = -4.84504e-02, x = -2.69527e-01, f(x) -->  1.92832e+01 ...
---   Iteration 005: dx = -3.43500e-02, x = -3.03877e-01, f(x) -->  4.06455e+00 ...
---   Iteration 006: dx = -1.17405e-02, x = -3.15617e-01, f(x) -->  2.97379e-01 ...
---   Iteration 007: dx = -1.00095e-03, x = -3.16618e-01, f(x) -->  1.89334e-03 ...
---   Iteration 008: dx = -6.45480e-06, x = -3.16625e-01, f(x) -->  7.79024e-08 ...
---   Iteration 009: dx = -2.65608e-10, x = -3.16625e-01, f(x) -->  1.06581e-14 ...
--- Output:
---   root =      -0.31662479 ...
---   f(root)   -->   1.06581410e-14 ...
---   df(root)  -->   2.93298241e+02 ...
---   no iterations = 9 ...
---   converged: True ...
---
--- Part 6: Solve f1(x) = 0 for lower root: x0 = -2.0 ...
--- ================================================= ...
--- Inputs:
---   x0 = -2.00 ...
---   tolerance      =  0.00000 ...
---   max iterations =   100.00 ...
--- Execution:
---   Initial Conditions:
---     x0     --> -2.00000e+00 ...
---     f(x0)  --> -3.06250e+01 ...
```

```
---      df(x0) -->  1.31250e+00 ...
---    Main Loop for Newton Raphson Iteration:
---    Iteration 001: dx =  2.33333e+01, x =  2.13333e+01, f(x) -->  1.17832e+03 ...
---    Iteration 002: dx = -6.91439e+00, x =  1.44189e+01, f(x) -->  3.41153e+02 ...
---    Iteration 003: dx = -4.38994e+00, x =  1.00290e+01, f(x) -->  9.36325e+01 ...
---    Iteration 004: dx = -2.49932e+00, x =  7.52969e+00, f(x) -->  2.15447e+01 ...
---    Iteration 005: dx = -1.02578e+00, x =  6.50390e+00, f(x) -->  2.83728e+00 ...
---    Iteration 006: dx = -1.81846e-01, x =  6.32206e+00, f(x) -->  7.99439e-02 ...
---    Iteration 007: dx = -5.42820e-03, x =  6.31663e+00, f(x) -->  6.98764e-05 ...
---    Iteration 008: dx = -4.75292e-06, x =  6.31662e+00, f(x) -->  5.35429e-11 ...
---    Iteration 009: dx = -3.64194e-12, x =  6.31662e+00, f(x) -->  1.06581e-14 ...
--- Output:
---    root =       6.31662479 ...                <--- Converges to wrong root !!!!
---    f(root)   -->   1.06581410e-14 ...
---    df(root)  -->   1.47017588e+01 ...
---    no iterations = 9 ...
---    converged: True ...

--- ========================================= ...
--- Leave TestNewtonRaphson05.main()          ...
```

Points to note:

1. The method of bisection is a reliable workhorse for computing the roots to an equation. But it is not terribly fast (30 odd iterations).

2. Convergence rate of Newton Raphson is much higher than for Bisection (5 iterations vs 30 iterations).

3. Newton Raphson requires that the the initial estimate for the root be carefully chosen – otherwise, the algorithm can converge to the wrong root.

Question 2: 10 points.

Problem Statement: This question covers numerical solutions to the root of the equation:

$$f(x) = 1 - e^{-(x-1)^2} = 0. \tag{3}$$

at x = 1. Figure 2 plots f(x) over the range $[-2, 4]$.

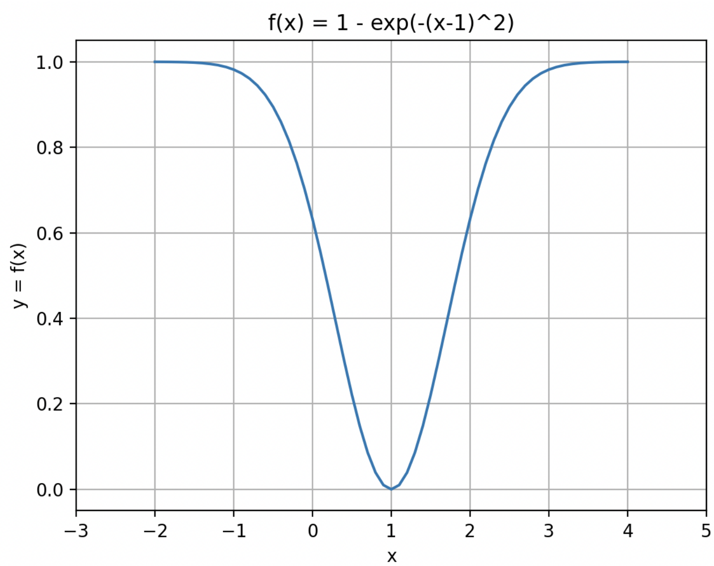

Figure 2: Plot y = f(x) vs x.

Write a Python program to plot equation 3 over the range [-2, 4], and then demonstrate that while the **method of Newton Raphson** struggles to find value(s) of x for which f(x) = 0 (why?), the **method of Modified Newton Raphson** computes the same roots with ease.

Python Source Code:

```
# ===========================================================================
# TestNewtonRaphson06.py: Use newton raphson/modified newton raphson algorithms
# to compute roots of equations:
#
#            f(x)  = 1 - exp(-(x-1)^2)  = 0.0
#
# Written By: Mark Austin                                             April 2025
# ===========================================================================

import math;
import Solutions;
```

```
import numpy as np
import matplotlib.pyplot as plt

# Functions: f(x) = 1 - exp(-(x-1)^2) = 0.0.

def f1(x):
    return 1 - np.exp(-(x-1)**2);

def df1(x):
    return -(2 - 2*x)*np.exp(-(x-1)**2);

def ddf1(x):
    return -(2 - 8*x + 4*x*x)*np.exp(-(x-1)**2);

# main method ...

def main():
    print("--- Enter TestNewtonRaphson06.main()          ... ");
    print("--- ========================================== ... ");

    print("--- ");
    print("--- Plot f1(x) over range [ -2, 4 ] ... ");
    print("--- ========================================== ... ");

    # Generate x and y coordinate arrays ....

    xcoords = np.linspace(-2, 4, 61, endpoint=True);

    ycoords = [];
    for i in range( len(xcoords) ):
        ycoords.append( f1( xcoords[i] ) );

    # Plot f(x) ...

    plt.plot( xcoords, ycoords )
    plt.title('f(x) = 1 - exp(-(x-1)^2)');
    plt.xlabel('x');
    plt.ylabel('y = f(x)');
    plt.xlim( [-3,5] );
    plt.grid(True)
    plt.show()

    print("--- ");
    print("--- Case Study 1: Use Newton Raphson with initial guess: x0 = 2 ... ");
    print("--- ============================================================ ... ");

    # Initialize problem setup ...

    x0 = 2.0;
    tolerance     = 0.00000000001
    maxiterations = 100

    print("--- Inputs:")
    print("---   x0 = {:5.2f} ...".format(x0) )
    print("---   tolerance      = {:16.8f} ...".format(tolerance) )
```

```
    print("---   max iterations = {:16.8f} ...".format(maxiterations) )

    # Compute roots to equation ...

    print("--- Execution:")
    root, i, converged = Solutions.newtonraphson(f1, df1, x0, tolerance, maxiterations )

    # Summary of computations ...

    print("--- Output:")
    print("---   root = {:12.7f} ...".format(root) )
    print("---   f(root)   --> {:16.8e} ...".format(   f1(root)) )
    print("---   df(root)  --> {:16.8e} ...".format(  df1(root)) )
    print("---   no iterations = {:d} ...".format(i) )
    print("---   converged: {:s} ...".format( str(converged) ) )

    print("--- ");
    print("--- Case Study 2: Use Modified Newton Raphson with initial guess: x0 = 2 ... ");
    print("--- ==================================================================== ... ");

    print("--- Inputs:")
    print("---   x0 = {:5.2f} ...".format(x0) )
    print("---   tolerance      = {:16.8f} ...".format(tolerance) )
    print("---   max iterations = {:16.8f} ...".format(maxiterations) )

    # Compute roots to equation ...

    print("--- Execution:")
    root, i, converged = Solutions.modifiednewtonraphson(f1, df1, ddf1, x0, tolerance, maxiterations

    # Summary of computations ...

    print("--- Output:")
    print("---   root = {:12.7f} ...".format(root) )
    print("---   f(root)   --> {:16.8e} ...".format(   f1(root)) )
    print("---   df(root)  --> {:16.8e} ...".format(  df1(root)) )
    print("---   ddf(root) --> {:16.8e} ...".format( ddf1(root)) )
    print("---   no iterations = {:d} ...".format(i) )
    print("---   converged: {:s} ...".format( str(converged) ) )

    print("--- ========================================= ... ");
    print("--- Leave TestNewtonRaphson06.main()          ... ");

# call the main method ...

main()
```

Abbreviated Output:

```
--- Enter TestNewtonRaphson06.main()          ...
--- ========================================= ...
---
--- Plot f1(x) over range [ -2, 4 ] ...
```

```
--- ========================================== ...
---
--- Case Study 1: Use Newton Raphson with initial guess: x0 = 2 ...
--- ============================================================== ...
--- Inputs:
---   x0 =  2.00 ...
---   tolerance      =       0.00000000 ...
---   max iterations =     100.00000000 ...
--- Execution:
---   Initial Conditions:
---     x0     -->  2.00000e+00 ...
---     f(x0)  -->  6.32121e-01 ...
---     df(x0) -->  7.35759e-01 ...
---   Main Loop for Newton Raphson Iteration:
---   Iteration 001: dx = -8.59141e-01, x =  1.14086e+00, f(x) -->  1.96457e-02 ...
---   Iteration 002: dx = -7.11329e-02, x =  1.06973e+00, f(x) -->  4.84994e-03 ...
---   Iteration 003: dx = -3.49480e-02, x =  1.03478e+00, f(x) -->  1.20879e-03 ...
---   Iteration 004: dx = -1.73996e-02, x =  1.01738e+00, f(x) -->  3.01970e-04 ...
---   Iteration 005: dx = -8.69060e-03, x =  1.00869e+00, f(x) -->  7.54782e-05 ...
---   Iteration 006: dx = -4.34415e-03, x =  1.00434e+00, f(x) -->  1.88686e-05 ...
---   Iteration 007: dx = -2.17193e-03, x =  1.00217e+00, f(x) -->  4.71711e-06 ...
---   Iteration 008: dx = -1.08595e-03, x =  1.00109e+00, f(x) -->  1.17927e-06 ...
---   Iteration 009: dx = -5.42972e-04, x =  1.00054e+00, f(x) -->  2.94818e-07 ...
---   Iteration 010: dx = -2.71486e-04, x =  1.00027e+00, f(x) -->  7.37045e-08 ...
---   Iteration 011: dx = -1.35743e-04, x =  1.00014e+00, f(x) -->  1.84261e-08 ...
---   Iteration 012: dx = -6.78714e-05, x =  1.00007e+00, f(x) -->  4.60653e-09 ...
---   Iteration 013: dx = -3.39357e-05, x =  1.00003e+00, f(x) -->  1.15163e-09 ...
---   Iteration 014: dx = -1.69679e-05, x =  1.00002e+00, f(x) -->  2.87908e-10 ...
---   Iteration 015: dx = -8.48393e-06, x =  1.00001e+00, f(x) -->  7.19771e-11 ...
---   Iteration 016: dx = -4.24197e-06, x =  1.00000e+00, f(x) -->  1.79943e-11 ...
---   Iteration 017: dx = -2.12098e-06, x =  1.00000e+00, f(x) -->  4.49851e-12 ...
---   Iteration 018: dx = -1.06048e-06, x =  1.00000e+00, f(x) -->  1.12466e-12 ...
---   Iteration 019: dx = -5.30248e-07, x =  1.00000e+00, f(x) -->  2.81219e-13 ...
---   Iteration 020: dx = -2.65176e-07, x =  1.00000e+00, f(x) -->  7.02771e-14 ...
---   Iteration 021: dx = -1.32561e-07, x =  1.00000e+00, f(x) -->  1.75415e-14 ...
---   Iteration 022: dx = -6.61870e-08, x =  1.00000e+00, f(x) -->  4.44089e-15 ...
---   Iteration 023: dx = -3.34768e-08, x =  1.00000e+00, f(x) -->  1.11022e-15 ...
---   Iteration 024: dx = -1.68978e-08, x =  1.00000e+00, f(x) -->  2.22045e-16 ...
---   Iteration 025: dx = -6.95916e-09, x =  1.00000e+00, f(x) -->  1.11022e-16 ...
---   Iteration 026: dx = -6.17186e-09, x =  1.00000e+00, f(x) -->  0.00000e+00 ...
---   Iteration 027: dx = -0.00000e+00, x =  1.00000e+00, f(x) -->  0.00000e+00 ...
--- Output:
---   root =    1.0000000 ...
---   f(root)   -->   0.00000000e+00 ...
---   df(root)  -->   5.64473934e-09 ...
---   no iterations = 27 ...
---   converged: True ...
---
--- Case Study 2: Use Modified Newton Raphson with initial guess: x0 = 2 ...
--- ======================================================================= ...
--- Inputs:
---   x0 =  2.00 ...
---   tolerance      =       0.00000000 ...
---   max iterations =     100.00000000 ...
--- Execution:
```

```
---   Initial Conditions:
---     x0     -->  2.00000e+00 ...
---     f(x0)  -->  6.32121e-01 ...
---   Main Loop for Modified Newton Raphson Iteration:
---   Iteration 001: dx = -4.62117e-01, x =  1.53788e+00, f(x) -->  2.51226e-01 ...
---   Iteration 002: dx = -4.12724e-01, x =  1.12516e+00, f(x) -->  1.55426e-02 ...
---   Iteration 003: dx = -1.23223e-01, x =  1.00194e+00, f(x) -->  3.74532e-06 ...
---   Iteration 004: dx = -1.93528e-03, x =  1.00000e+00, f(x) -->  0.00000e+00 ...
---   Iteration 005: dx = -0.00000e+00, x =  1.00000e+00, f(x) -->  0.00000e+00 ...
--- Output:
---   root =    1.0000000 ...
---   f(root)   -->   0.00000000e+00 ...
---   df(root)  -->   1.44964574e-08 ...
---   ddf(root) -->   2.00000000e+00 ...
---   no iterations = 5 ...
---   converged: True ...
--- ========================================= ...
--- Leave TestNewtonRaphson06.main()          ...
```

Question 3: 10 points.

Problem Statement: Figure 4 shows an elastic column fixed at its base and pinned at the top. The critical buckling load, P_{cr}, corresponds to the first positive solution to

$$\lambda = \tan[\lambda] \quad (4)$$

where $\lambda = L \cdot \sqrt{\frac{P_{cr}}{EI}}$.

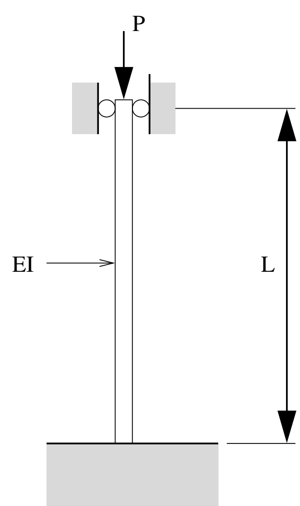

Figure 3: Elastic column carrying an axial load.

Use Python to create a single plot of y_1 = x and y_2 = tan(x) – the intersection of contours on this plot should give you an approximate range within which an accurate solution exists. Use the **method of bisection** to compute the numerical solution to equation 4. Print the buckling load corresponding to your solution?

Python Source Code:

```
# ===================================================================
# TestElasticColumnBuckling01.py: Use bisection algorithm to compute
# buckling load of an elastic column.
#
# Written by: Mark Austin                                 April 2025
# ===================================================================

import math
import matplotlib.pyplot as plt
import numpy as np

import Solutions;
```

```
# Compute y = tan(x) - x ...

def f1(x):
    return math.tan(x) - x;

# Main function ...

def main():
    print("--- Part 1: Create plots for y = x, y = tan(x) ... ");

    print("---   Create (x,y) coords for y1 = tan(x) ... ");

    x  = np.arange( -2.0, 6.0, 0.1)
    y1 = np.tan(x)

    print("---   Create (x,y) coords for y2 = x ... ");

    y2 = x

    plt.plot(x,y1, 'b', label='y1 = tan(x)')
    plt.plot(x,y2, 'r', label='y2 = x')
    plt.title('Plot: y = x and y = tan(x)')
    plt.ylabel('y')
    plt.xlabel('x')
    plt.ylim( -10, 10)
    plt.xlim(  -2,  7)
    plt.grid(True)
    plt.legend()
    plt.show()

    print("--- Part 2: Use Bisection algorithm to compute positive root tan(x) - x = 0 ... ");

    # Initialize problem setup ...

    a =  4.0;
    b =  4.7
    tolerance     = 0.01
    maxiterations = 100

    print("--- Inputs:")
    print("---   a = {:5.2f} ...".format(a) )
    print("---   b = {:5.2f} ...".format(b) )
    print("---   tolerance      = {:8.5f} ...".format(tolerance) )
    print("---   max iterations = {:8.2f} ...".format(maxiterations) )

    # Compute roots to equation ...

    print("--- Execution:")
    root, i, converged = Solutions.bisection(f1, a, b, tolerance, maxiterations )

    # Summary of computations ...

    print("--- Output:")
    print("---   root = {:10.5f} ...".format(root) )
```

```
    print("---   f(root) --> {:12.5e} ...".format( f1(root)) )
    print("---   no iterations = {:d} ...".format(i) )
    print("---   converged: {:s} ...".format( str(converged) ) )

    print("--- Part 3: Compute column buckling load ... ");

    print("")
    print("---   Buckling load = {:5.2f} * EI/L^2 ... ".format(root*root));

# call the main method ...

main()
```

Program Output:

```
--- Part 1: Create plots for y = x, y = tan(x) ...

---   Create (x,y) coords for y1 = tan(x) ...
---   Create (x,y) coords for y2 = x ...

--- Part 2: Use Bisection algorithm to compute positive root tan(x) - x = 0 ...

--- Inputs:
---   a =  4.00 ...
---   b =  4.70 ...
---   tolerance      =  0.01000 ...
---   max iterations =   100.00 ...
--- Execution:
---   Initial Conditions:
---   f(a) --> -2.84218e+00 ...
---   f(b) -->  7.60128e+01 ...
---   Main Loop for Root Computation:
---   Iteration 000: dx = 3.50000e-01, x =  4.3500000e+00, f(x) --> -1.7124015e+00 ...
---   Iteration 001: dx = 1.75000e-01, x =  4.5250000e+00, f(x) -->  7.4888339e-01 ...
---   Iteration 002: dx = 8.75000e-02, x =  4.4375000e+00, f(x) --> -8.9176234e-01 ...
---   Iteration 003: dx = 4.37500e-02, x =  4.4812500e+00, f(x) --> -2.3217078e-01 ...
---   Iteration 004: dx = 2.18750e-02, x =  4.5031250e+00, f(x) -->  2.0556909e-01 ...
---   Iteration 005: dx = 1.09375e-02, x =  4.4921875e+00, f(x) --> -2.4530831e-02 ...
---   Iteration 006: dx = 5.46875e-03, x =  4.4976563e+00, f(x) -->  8.7497087e-02 ...
---   Iteration 007: dx = 2.73438e-03, x =  4.4949219e+00, f(x) -->  3.0756122e-02 ...
---   Iteration 008: dx = 1.36719e-03, x =  4.4935547e+00, f(x) -->  2.9343009e-03 ...
--- Output:
---   root =    4.49355 ...
---   f(root) -->  2.93430e-03 ...
---   no iterations = 8 ...
---   converged: True ...

--- Part 3: Compute column buckling load ...

---   Buckling load = 20.19 * EI/L^2 ...
```

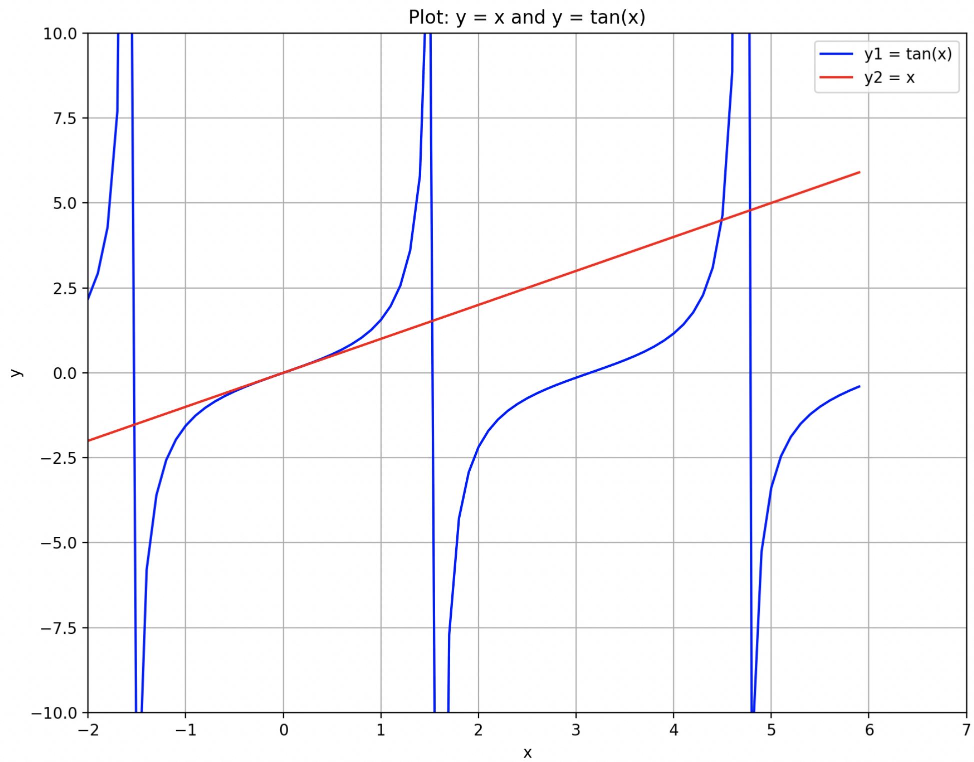

Figure 4: Plots: y_1 = x and y_2 = tan(x).

Question 4: 10 points.

Problem Statement: Consider the family of matrix equations AX = B defined by

$$\begin{bmatrix} 1 & 2 & -3 \\ 3 & -1 & 5 \\ 4 & 1 & a^2-14 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \\ a+2 \end{bmatrix}. \tag{5}$$

Determine the values of 'a' for which matrix A will be singular. Then,

1. Develop a program that uses NumPy to store matrices A and B, and then systematically evaluates the determinant and rank of A, and the rank of augmented matrix [A | B] for the values of 'a' that make A singular.

1. Develop a second program that uses SymPy to store matrices A and B symbolically, and then computes symbolic solution to the matrix equations 5. For the values of 'a' that make A singular, evaluate the determinant and rank of A, and the rank of augmented matrix [A | B].

You should find that the Python module SymPy is considerably more powerful than its numerical counterpart NumPy.

Hand Calculation: We begin with:

$$det(A) = 1 \cdot det \begin{bmatrix} -1 & 5 \\ 1 & a^2-14 \end{bmatrix} - 2 \cdot det \begin{bmatrix} 3 & 5 \\ 4 & a^2-14 \end{bmatrix} - 3 \cdot det \begin{bmatrix} 3 & -1 \\ 4 & 1 \end{bmatrix} = 112 - 7a^2 \tag{6}$$

Setting det(A) = $112 - 7a^2$ = 0 gives a = -4 or a = 4.

Program 1: Python Source Code: Implementation with NumPy.

```
# ==============================================================
# TestMatrixOperationsNumPy01.py: Compute symbolic solutions to
# linear matrix equations modeled in NumPy.
#
# Written by: Mark Austin                              April 2025
# ==============================================================

import numpy as np
from numpy.linalg import matrix_rank

# ================================================================
```

```
# Function to print two-dimensional matrices ...
# ================================================================

def PrintMatrix(name, a):
    print("Matrix: {:s} ".format(name) );
    for row in a:
        for col in row:
            print("{:8.2f}".format(col), end=" ")
        print("")

# ================================================================
# main method ...
# ================================================================

def main():
    print("--- Case 1: Set a = 4 in matrices A and B ...");

    a = 4
    A = np.array( [ [ 1,  2, -3],
                    [ 3, -1,  5],
                    [ 4,  1, a*a - 14 ] ])
    B = np.array( [ [4], [2], [a+2] ])

    PrintMatrix("A", A)
    PrintMatrix("B", B)

    print("");
    print("--- determinant(A)  --> {:f} ...".format( np.linalg.det(A) ))
    print("--- rank [A]        --> {:f} ...".format( matrix_rank(A) ))

    print("");
    print("--- Create augmented matrix [ A | B ] ...\n");

    C = np.concatenate((A, B), axis=1)
    PrintMatrix("Matrix [A|B]", C)

    print("--- rank [A|B]      --> {:f} ...".format( matrix_rank(C) ))

    print("");
    print("--- Case 2: Set a = -4 in matrices A and B ...");

    a = -4
    A = np.array( [ [ 1,  2, -3],
                    [ 3, -1,  5],
                    [ 4,  1, a*a - 14 ] ])
    B = np.array( [ [4], [2], [a+2] ])

    PrintMatrix("A", A)
    PrintMatrix("B", B)

    print("");
    print("--- determinant(A)  --> {:f} ...".format( np.linalg.det(A) ))
    print("--- rank [A]        --> {:f} ...".format( matrix_rank(A) ))

    print("");
```

```
    print("--- Create augmented matrix [ A | B ] ...\n");

    C = np.concatenate((A, B), axis=1)
    PrintMatrix("Matrix [A|B]", C)

    print("");
    print("--- rank [A|B]      --> {:f} ...".format( matrix_rank(C) ))

# call the main method ...

main()
```

Program 1 Output:

```
--- Case 1: Set a = 4 in matrices A and B ...

Matrix: A
    1.00     2.00    -3.00
    3.00    -1.00     5.00
    4.00     1.00     2.00

Matrix: B
    4.00
    2.00
    6.00

--- determinant(A)  --> 0.000000 ...
--- rank [A]        --> 2.000000 ...

--- Create augmented matrix [ A | B ] ...

Matrix: Matrix [A|B]
    1.00     2.00    -3.00     4.00
    3.00    -1.00     5.00     2.00
    4.00     1.00     2.00     6.00

--- rank [A|B]      --> 2.000000 ...

--- Case 2: Set a = -4 in matrices A and B ...

Matrix: A
    1.00     2.00    -3.00
    3.00    -1.00     5.00
    4.00     1.00     2.00
Matrix: B
    4.00
    2.00
   -2.00

--- determinant(A)  --> 0.000000 ...
--- rank [A]        --> 2.000000 ...

--- Create augmented matrix [ A | B ] ...
```

```
Matrix: Matrix [A|B]
    1.00     2.00    -3.00     4.00
    3.00    -1.00     5.00     2.00
    4.00     1.00     2.00    -2.00

--- rank [A|B]      --> 3.000000 ...
```

Note: When a = 4 (Case 1), rank $[A]$ = rank $[A|B]$ = 2 – the equations overlap and there are an infinite number of solutions. Conversely, a = -4 (Case 2), rank $[A]$ = 2 and rank $[A|B]$ = 3 – the equations are inconsistent and there are zero solutions.

Program 2: Python Source Code: Implementation with SymPy.

```
# ===============================================================
# TestMatrixOperationsSympy01.py: Compute symbolic solutions to
# linear matrix equations with sympy ...
#
# Written by: Mark Austin                              April 2025
# ===============================================================

import sympy as sp
from sympy import Integral, Matrix, pi, pprint

# main method ...

def main():
    # Define symbolic representation of matrices ...

    print("--- Create matrices A and B ...");

    a = sp.symbols('a')
    A = sp.Matrix(( [ 1,  2, -3],
                    [ 3, -1,  5],
                    [ 4,  1, a*a - 14 ] ))
    B = sp.Matrix(( [4], [2], [a+2] ))

    pprint(A)
    pprint(B)

    print("--- Compute and print matrix determinant ...\n");

    print("--- A.det() --> {:s} ...".format( str(A.det() ) ))

    print("--- General symbolic solution to matrix system ...\n");

    solution = A.solve(B)
    print(solution)

    print("--- Expressions for individual solution elements ...\n");
    print("--- x1 = {:s} ...".format( str( solution[0] ) ))
    print("--- x2 = {:s} ...".format( str( solution[1] ) ))
```

```
    print("--- x2 = {:s} ...".format( str( solution[2] ) ))

    print("");
    print("--- Create augmented matrix [ A | B ] ...\n");

    C = A.row_join(B)
    pprint(C)

    print("--- Case 1: Set a = 4 in matrix [A|B] ...");

    A1 = A.subs( {a:4} )
    C1 = C.subs( {a:4} )
    pprint(C1)

    print("");
    print("--- A1.det()  --> {:s} ...".format( str( A1.det() ) ))
    print("--- A1.rank() --> {:s} ...".format( str( A1.rank() ) ))
    print("--- C1.rank() --> {:s} ...".format( str( C1.rank() ) ))

    print("--- Case 2: Set a = -4 in matrix [A|B] ...");

    A2 = A.subs( {a:-4} )
    C2 = C.subs( {a:-4} )
    pprint(C2)

    print("--- A2.det()  --> {:s} ...".format( str( A2.det() ) ))
    print("--- A2.rank() --> {:s} ...".format( str( A2.rank() ) ))
    print("--- C2.rank() --> {:s} ...".format( str( C2.rank() ) ))

# call the main method ...

main()
```

Program 2 Output:

```
--- Create matrices A and B ...

| 1  2     -3    |
|                |
| 3  -1     5    |
|                |
|        2       |
| 4  1   a  - 14 |

|   4   |
|       |
|   2   |
|       |
| a + 2 |

--- Compute and print matrix determinant ...

--- A.det() --> 112 - 7*a**2 ...
```

```
--- General symbolic solution to matrix system ...

Matrix([[(8*a + 25)/(7*a + 28)], [(10*a + 54)/(7*a + 28)], [1/(a + 4)]])

--- Expressions for individual solution elements ...

--- x1 = (8*a + 25)/(7*a + 28) ...
--- x2 = (10*a + 54)/(7*a + 28) ...
--- x2 = 1/(a + 4) ...

--- Create augmented matrix [ A | B ] ...

| 1  2     -3       4    |
|                        |
| 3  -1     5       2    |
|                        |
|       2                |
| 4  1   a  - 14  a + 2 |

--- Case 1: Set a = 4 in matrix [A|B] ...

| 1  2   -3  4 |
|              |
| 3  -1  5   2 |
|              |
| 4  1   2   6 |

--- A1.det()  --> 0 ...
--- A1.rank() --> 2 ...
--- C1.rank() --> 2 ...

--- Case 2: Set a = -4 in matrix [A|B] ...

| 1  2   -3  4 |
|              |
| 3  -1  5   2 |
|              |
| 4  1   2  -2 |

--- A2.det()  --> 0 ...
--- A2.rank() --> 2 ...
--- C2.rank() --> 3 ...
```

Note: Using SymPy leads to results consistent with NumPy.

Question 5: 10 points.

Problem Statement: In the design of highway bridge structures and crane structures, engineers are often required to compute the maximum and minimum member forces and support reactions due to a variety of loading conditions.

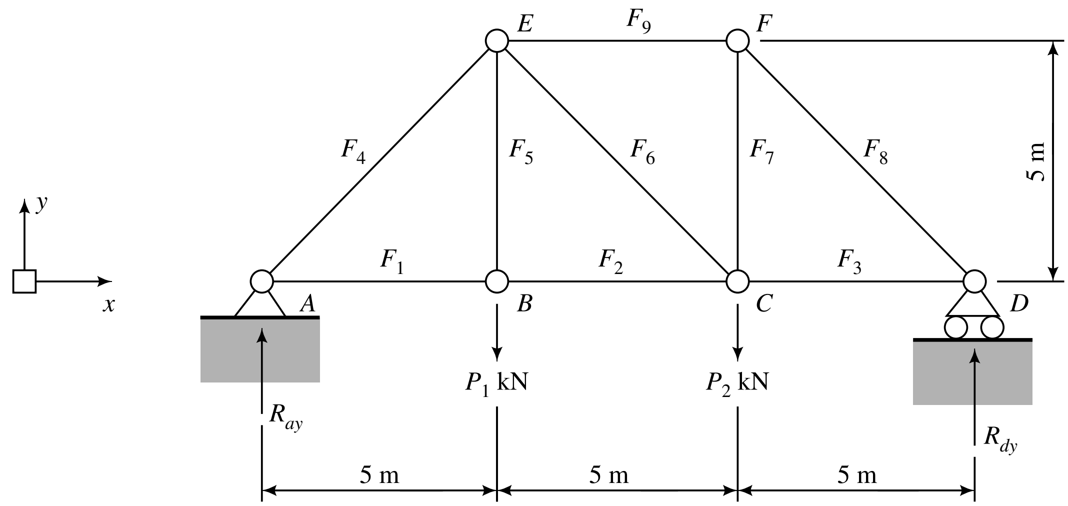

Figure 5: Front elevation of pin-jointed bridge truss.

Figure 5 shows a nine bar pin-jointed bridge truss carrying vertical loads P_1 kN and P_2 kN at joints `B` and `C`. The symbols F_1, F_2, $\cdots$ F_9 represent the axial forces in truss members 1 through 9, and R_{ay} and R_{dy} are the support reactions at joints `A` and `D`. You can also include a horizontal reaction force at A, R_{ax}, but its value will be zero.

Write down the equations of equilibrium for joints `A` through `F` and put the equations in matrix form. Now suppose that a heavy load moves across the bridge and that, for engineering purposes, it can be represented by the sequence of external load vectors

$$\begin{bmatrix} P_1 \\ P_2 \end{bmatrix} = \begin{bmatrix} 100 \\ 0 \end{bmatrix}, \begin{bmatrix} P_1 \\ P_2 \end{bmatrix} = \begin{bmatrix} 100 \\ 100 \end{bmatrix}, \begin{bmatrix} P_1 \\ P_2 \end{bmatrix} = \begin{bmatrix} 50 \\ 100 \end{bmatrix} \tag{7}$$

Develop a Python program that will solve the matrix equations for each of the external load conditions, and compute and print the minimum and maximum support reactions at nodes A and D, and axial forces in each of the truss members.

Hint: See the folder python-code.d/applications/structures/ for examples of similar problems and their solutions.

Python Source Code:

```
# ===============================================================
# TestTrussAnalysis06.py: Compute distribution of element forces
# and support reactions in a nine-bar highway bridge truss.
#
# Written by: Mark Austin                               April 2025
# ===============================================================

import math
import numpy as np
from numpy.linalg import matrix_rank

# ===============================================================
# Function to print one- and two-dimensional matrices ...
# ===============================================================

def PrintMatrix(name, matrix):
    NoColumns = 6;

    # Compute no of blocks of rows to be printed .....

    if matrix.ndim == 1:
       noMatrixRows = matrix.shape[0]
       noMatrixCols = 1

    if matrix.ndim == 2:
       noMatrixRows = matrix.shape[0]
       noMatrixCols = matrix.shape[1]

    #  Compute number of blocks to be printed ...

    if noMatrixCols % NoColumns == 0:
       iNoBlocks = noMatrixCols/NoColumns;
    else:
       iNoBlocks = noMatrixCols/NoColumns + 1;

    #  Loop over the number of blocks ...

    for ib in range( int(iNoBlocks) ):
        iFirstColumn = ib*NoColumns + 1
        iLastColumn  = min ( (ib+1)*NoColumns, noMatrixCols )

        # Print title of matrix at top of each block ....

        print("Matrix: {:s} ".format(name) );

        # Label row and column nos */

        print("row/col     ", end="")
        colList = range(iFirstColumn, iLastColumn + 1)
        for col in [ *colList ]:
            print("       {:3d}    ".format(col),end="")
        print("")
```

```
        # Loop over rows and print matrix elements ....

        ii = 1
        for row in matrix:
            print(" {:3d}     ".format(ii),end="")
            colList = range( iFirstColumn, iLastColumn + 1)
            for col in [ *colList ]:
                if matrix.ndim == 1:
                   print(" {:12.5e} ".format( matrix[ii-1] ), end="")
                else:
                   print(" {:12.5e} ".format(matrix[ii-1][col-1]), end="")
            print("")
            ii = ii + 1
        print("")

# ================================================================
# Compute maximum and minumun of three numbers ...
# ================================================================

def maximum(a, b, c):
    list = [a, b, c]
    return max(list)

def minimum(a, b, c):
    list = [a, b, c]
    return min(list)

# ================================================================
# Print element forces ...
# ================================================================

def printElementForces(name, minF, maxF):
    if( minF < 0):
        print("---     Minimum {:s} = {:7.2f} (C) ... ".format( name, minF ) )
    else:
        print("---     Minimum {:s} = {:7.2f} (T) ... ".format( name, minF ) )

    if( maxF < 0):
        print("---     Maximum {:s} = {:7.2f} (C) ... ".format( name, maxF ) )
    else:
        print("---     Maximum {:s} = {:7.2f} (T) ... ".format( name, maxF ) )

# ================================================================
# main method ...
# ================================================================

def main():
    print("--- Enter TestTrussAnalysis06.main()         ... ");
    print("--- ======================================== ... ");

    print("--- ");
    print("--- Part 1: Initialize coefficients for matrix equations ... ");

    # Node A ...
```

```
a11  = 1               # < --- equilibrium in x direction ...
a14  = 1/math.sqrt(2)
a110 = 1
a24  = 1/math.sqrt(2)  # < --- equilibrium in y direction ...
a211 = 1

# Node B ...

a31  =  1              # < --- equilibrium in x direction ...
a32  = -1
a45  = -1              # < --- equilibrium in y direction ...

# Node C ...

a52  =  1              # < --- equilibrium in x direction ...
a53  = -1
a56  =  1/math.sqrt(2)

a66  = -1/math.sqrt(2) # < --- equilibrium in y direction ...
a67  = -1

# Node D ...

a73  = 1               # < --- equilibrium in x direction ...
a78  = 1/math.sqrt(2)
a88  = 1/math.sqrt(2)  # < --- equilibrium in y direction ...
a812 = 1

# Node E ...

a99  =  1              # < --- equilibrium in x direction ...
a96  =  1/math.sqrt(2)
a94  = -1/math.sqrt(2)

a104 =  1/math.sqrt(2) # < --- equilibrium in y direction ...
a105 =  1
a106 =  1/math.sqrt(2)

# Node F ...

a118 = -1/math.sqrt(2)  # < --- equilibrium in x direction ...
a119 =  1
a127 =  1               # < --- equilibrium in y direction ...
a128 =  1/math.sqrt(2)

print("--- ");
print("--- Part 2: Create test matrix A ... ");
print("--- ");

A = np.array([ [ a11,    0,    0,  a14,    0,    0,    0,    0,    0, a110,    0,    0 ],
               [   0,    0,    0,  a24,    0,    0,    0,    0,    0,    0, a211,    0 ],
               [ a31,  a32,    0,    0,    0,    0,    0,    0,    0,    0,    0,    0 ],
               [   0,    0,    0,    0,  a45,    0,    0,    0,    0,    0,    0,    0 ],
               [   0,  a52,  a53,    0,    0,  a56,    0,    0,    0,    0,    0,    0 ],
               [   0,    0,    0,    0,    0,  a66,  a67,    0,    0,    0,    0,    0 ],
```

```
               [   0,    0,  a73,    0,    0,    0,    0,  a78,    0,    0,    0,    0 ],
               [   0,    0,    0,    0,    0,    0,    0,  a88,    0,    0,    0, a812 ],
               [   0,    0,    0,  a94,    0,  a96,    0,    0,  a99,    0,    0,    0 ],
               [   0,    0,    0, a104, a105, a106,    0,    0,    0,    0,    0,    0 ],
               [   0,    0,    0,    0,    0,    0,    0, a118, a119,    0,    0,    0 ],
               [   0,    0,    0,    0,    0,    0, a127, a128,    0,    0,    0,    0 ] ]);
PrintMatrix("A", A);

print("--- ");
print("--- Part 2: Initialize load vectors ... ");
print("--- ");

print("--- Load Case 1: b4 = -100, b6 = 0 ...");

b4 = -100.0; b6 = 0.0

B1 = np.array([ [0], [0], [0], [b4],  [0], [b6], [0], [0], [0], [0], [0], [0] ]);
PrintMatrix("Load Case 1: B", B1);

print("--- Load Case 2: b4 = -100, b6 = -100 ...");

b4 = -100.0; b6 = -100.0

B2 = np.array([ [0], [0], [0], [b4],  [0], [b6], [0], [0], [0], [0], [0], [0] ]);
PrintMatrix("Load Case 2: B", B2);

print("--- Load Case 3: b4 = -50, b6 = -100 ...");

b4 = -50.0; b6 = -100.0

B3 = np.array([ [0], [0], [0], [b4],  [0], [b6], [0], [0], [0], [0], [0], [0] ]);
PrintMatrix("Load Case 3: B", B3);

print("--- ");
print("--- Part 4: Check properties of matrix A ... ");
print("--- ");

rank = matrix_rank(A)
det  = np.linalg.det(A)

print("--- Matrix A: rank = {:f}, det = {:f}  ...".format(rank, det) );

print("--- ");
print("--- Part 5: Solve A.F = B for three load cases ... ");
print("--- ");

F1 = np.linalg.solve(A, B1)
PrintMatrix("Load Case 1: Forces ...", F1);

F2 = np.linalg.solve(A, B2)
PrintMatrix("Load Case 2: Forces ...", F2);

F3 = np.linalg.solve(A, B3)
PrintMatrix("Load Case 3: Forces ...", F3);
```

```
print("--- ");
print("--- Part 6: Print support reactions and element-level forces ... ");

print("--- ");
print("--- Support Reactions and Element-Level Forces: Load Case 1:");
print("--- ");
print("---   Reaction A: R_ax = {:7.2f} ... ".format( F1[9][0] ) );
print("---              : R_ay = {:7.2f} ... ".format( F1[10][0] ) );
print("---   Reaction D: R_dy = {:7.2f} ... ".format( F1[11][0] ) );
print("--- ");
print("--- Element Level Forces:");
print("--- ");
print("---   Element A-B: F1 = {:7.2f} ... ".format( F1[0][0] ) );
print("---   Element B-C: F2 = {:7.2f} ... ".format( F1[1][0] ) );
print("---   Element C-D: F3 = {:7.2f} ... ".format( F1[2][0] ) );
print("---   Element A-E: F4 = {:7.2f} ... ".format( F1[3][0] ) );
print("---   Element B-E: F5 = {:7.2f} ... ".format( F1[4][0] ) );
print("---   Element C-E: F6 = {:7.2f} ... ".format( F1[5][0] ) );
print("---   Element C-F: F7 = {:7.2f} ... ".format( F1[6][0] ) );
print("---   Element D-F: F8 = {:7.2f} ... ".format( F1[7][0] ) );
print("---   Element E-F: F9 = {:7.2f} ... ".format( F1[8][0] ) );

print("--- ");
print("--- Support Reactions and Element-Level Forces: Load Case 2:");
print("--- ");
print("---   Reaction A: R_ax = {:7.2f} ... ".format( F2[9][0] ) );
print("---              : R_ay = {:7.2f} ... ".format( F2[10][0] ) );
print("---   Reaction D: R_dy = {:7.2f} ... ".format( F2[11][0] ) );
print("--- ");
print("--- Element Level Forces:");
print("--- ");
print("---   Element A-B: F1 = {:7.2f} ... ".format( F2[0][0] ) );
print("---   Element B-C: F2 = {:7.2f} ... ".format( F2[1][0] ) );
print("---   Element C-D: F3 = {:7.2f} ... ".format( F2[2][0] ) );
print("---   Element A-E: F4 = {:7.2f} ... ".format( F2[3][0] ) );
print("---   Element B-E: F5 = {:7.2f} ... ".format( F2[4][0] ) );
print("---   Element C-E: F6 = {:7.2f} ... ".format( F2[5][0] ) );
print("---   Element C-F: F7 = {:7.2f} ... ".format( F2[6][0] ) );
print("---   Element D-F: F8 = {:7.2f} ... ".format( F2[7][0] ) );
print("---   Element E-F: F9 = {:7.2f} ... ".format( F2[8][0] ) );

print("--- ");
print("--- Support Reactions and Element-Level Forces: Load Case 3:");
print("--- ");
print("---   Reaction A: R_ax = {:7.2f} ... ".format( F3[9][0] ) );
print("---              : R_ay = {:7.2f} ... ".format( F3[10][0] ) );
print("---   Reaction D: R_dy = {:7.2f} ... ".format( F3[11][0] ) );
print("--- ");
print("--- Element Level Forces:");
print("--- ");
print("---   Element A-B: F1 = {:7.2f} ... ".format( F3[0][0] ) );
print("---   Element B-C: F2 = {:7.2f} ... ".format( F3[1][0] ) );
print("---   Element C-D: F3 = {:7.2f} ... ".format( F3[2][0] ) );
print("---   Element A-E: F4 = {:7.2f} ... ".format( F3[3][0] ) );
print("---   Element B-E: F5 = {:7.2f} ... ".format( F3[4][0] ) );
```

```
print("---   Element C-E: F6 = {:7.2f} ... ".format( F3[5][0] ) );
print("---   Element C-F: F7 = {:7.2f} ... ".format( F3[6][0] ) );
print("---   Element D-F: F8 = {:7.2f} ... ".format( F3[7][0] ) );
print("---   Element E-F: F9 = {:7.2f} ... ".format( F3[8][0] ) );

print("--- ");
print("--- Summary of Max/Min Reactions and Element-Level Forces ...");
print("--- ----------------------------------------------------- ...");

MinRax = minimum( F1[9][0], F2[9][0], F3[9][0] )
MaxRax = maximum( F1[9][0], F2[9][0], F3[9][0] )

MinRay = minimum( F1[10][0], F2[10][0], F3[10][0] )
MaxRay = maximum( F1[10][0], F2[10][0], F3[10][0] )

MinRdy = minimum( F1[11][0], F2[11][0], F3[11][0] )
MaxRdy = maximum( F1[11][0], F2[11][0], F3[11][0] )

print("--- ");
print("---   Reaction A: Minimum R_ax = {:7.2f}, Maximum R_ax = {:7.2f} ... ".format( MinRax, Max
print("---             : Minimum R_ay = {:7.2f}, Maximum R_ay = {:7.2f} ... ".format( MinRay, Max
print("--- ");
print("---   Reaction D: Minimum R_dy = {:7.2f}, Maximum R_dy = {:7.2f} ... ".format( MinRdy, Max

MinF1 = minimum( F1[0][0], F2[0][0], F3[0][0] )
MaxF1 = maximum( F1[0][0], F2[0][0], F3[0][0] )
MinF2 = minimum( F1[1][0], F2[1][0], F3[1][0] )
MaxF2 = maximum( F1[1][0], F2[1][0], F3[1][0] )
MinF3 = minimum( F1[2][0], F2[2][0], F3[2][0] )
MaxF3 = maximum( F1[2][0], F2[2][0], F3[2][0] )
MinF4 = minimum( F1[3][0], F2[3][0], F3[3][0] )
MaxF4 = maximum( F1[3][0], F2[3][0], F3[3][0] )
MinF5 = minimum( F1[4][0], F2[4][0], F3[4][0] )
MaxF5 = maximum( F1[4][0], F2[4][0], F3[4][0] )
MinF6 = minimum( F1[5][0], F2[5][0], F3[5][0] )
MaxF6 = maximum( F1[5][0], F2[5][0], F3[5][0] )
MinF7 = minimum( F1[6][0], F2[6][0], F3[6][0] )
MaxF7 = maximum( F1[6][0], F2[6][0], F3[6][0] )
MinF8 = minimum( F1[7][0], F2[7][0], F3[7][0] )
MaxF8 = maximum( F1[7][0], F2[7][0], F3[7][0] )
MinF9 = minimum( F1[8][0], F2[8][0], F3[8][0] )
MaxF9 = maximum( F1[8][0], F2[8][0], F3[8][0] )

print("--- ");
print("---   Element A-B: ")
printElementForces( "F1", MinF1, MaxF1)

print("---   Element B-C: ")
printElementForces( "F2", MinF2, MaxF2)

print("---   Element C-D: ")
printElementForces( "F3", MinF3, MaxF3)

print("---   Element A-E: ")
printElementForces( "F4", MinF4, MaxF4)
```

```
    print("---   Element B-E: ")
    printElementForces( "F5", MinF5, MaxF5)

    print("---   Element C-E: ")
    printElementForces( "F6", MinF6, MaxF6)

    print("---   Element C-F: ")
    printElementForces( "F7", MinF7, MaxF7)

    print("---   Element D-F: ")
    printElementForces( "F8", MinF8, MaxF8)

    print("---   Element E-F: ")
    printElementForces( "F9", MinF9, MaxF9)

    print("--- ======================================== ... ");
    print("--- Leave TestTrussAnalysis06.main()         ... ");

# call the main method ...

main()
```

Abbreviated Output:

```
--- Enter TestTrussAnalysis06.main()         ...
--- ======================================== ...
---
--- Part 1: Initialize coefficients for matrix equations ...
--- Part 2: Create test matrix A ...

Matrix: A
row/col             1             2             3             4             5             6
   1      1.00000e+00   0.00000e+00   0.00000e+00   7.07107e-01   0.00000e+00   0.00000e+00
   2      0.00000e+00   0.00000e+00   0.00000e+00   7.07107e-01   0.00000e+00   0.00000e+00
   3      1.00000e+00  -1.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00
   4      0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00  -1.00000e+00   0.00000e+00
   5      0.00000e+00   1.00000e+00  -1.00000e+00   0.00000e+00   0.00000e+00   7.07107e-01
   6      0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00  -7.07107e-01
   7      0.00000e+00   0.00000e+00   1.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00
   8      0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00
   9      0.00000e+00   0.00000e+00   0.00000e+00  -7.07107e-01   0.00000e+00   7.07107e-01
  10      0.00000e+00   0.00000e+00   0.00000e+00   7.07107e-01   1.00000e+00   7.07107e-01
  11      0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00
  12      0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00

Matrix: A
row/col             7             8             9            10            11            12
   1      0.00000e+00   0.00000e+00   0.00000e+00   1.00000e+00   0.00000e+00   0.00000e+00
   2      0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   1.00000e+00   0.00000e+00
   3      0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00
   4      0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00
   5      0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00
```

```
   6      -1.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00
   7       0.00000e+00   7.07107e-01   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00
   8       0.00000e+00   7.07107e-01   0.00000e+00   0.00000e+00   0.00000e+00   1.00000e+00
   9       0.00000e+00   0.00000e+00   1.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00
  10       0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00
  11       0.00000e+00  -7.07107e-01   1.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00
  12       1.00000e+00   7.07107e-01   0.00000e+00   0.00000e+00   0.00000e+00   0.00000e+00

--- Part 2: Initialize load vectors ...
---
--- Load Case 1: b4 = -100, b6 = 0 ...

Matrix: Load Case 1: B
row/col             1
   1       0.00000e+00
   2       0.00000e+00
   3       0.00000e+00
   4      -1.00000e+02
   5       0.00000e+00
   6       0.00000e+00
   7       0.00000e+00
   8       0.00000e+00
   9       0.00000e+00
  10       0.00000e+00
  11       0.00000e+00
  12       0.00000e+00

--- Load Case 2: b4 = -100, b6 = -100 ...

Matrix: Load Case 2: B
row/col             1
   1       0.00000e+00
   2       0.00000e+00
   3       0.00000e+00
   4      -1.00000e+02
   5       0.00000e+00
   6      -1.00000e+02
   7       0.00000e+00
   8       0.00000e+00
   9       0.00000e+00
  10       0.00000e+00
  11       0.00000e+00
  12       0.00000e+00

--- Load Case 3: b4 = -50, b6 = -100 ...

Matrix: Load Case 3: B
row/col             1
   1       0.00000e+00
   2       0.00000e+00
   3       0.00000e+00
   4      -5.00000e+01
   5       0.00000e+00
   6      -1.00000e+02
   7       0.00000e+00
```

```
   8       0.00000e+00
   9       0.00000e+00
  10       0.00000e+00
  11       0.00000e+00
  12       0.00000e+00

--- Part 4: Check properties of matrix A ...
---
--- Matrix A: rank = 12.000000, det = 1.060660  ...
---
--- Part 5: Solve A.F = B for three load cases ...
---

Matrix: Load Case 1: Forces ...
row/col              1
   1       6.66667e+01
   2       6.66667e+01
   3       3.33333e+01
   4      -9.42809e+01
   5       1.00000e+02
   6      -4.71405e+01
   7       3.33333e+01
   8      -4.71405e+01
   9      -3.33333e+01
  10      -0.00000e+00
  11       6.66667e+01
  12       3.33333e+01

Matrix: Load Case 2: Forces ...
row/col              1
   1       1.00000e+02
   2       1.00000e+02
   3       1.00000e+02
   4      -1.41421e+02
   5       1.00000e+02
   6      -0.00000e+00
   7       1.00000e+02
   8      -1.41421e+02
   9      -1.00000e+02
  10      -0.00000e+00
  11       1.00000e+02
  12       1.00000e+02

Matrix: Load Case 3: Forces ...
row/col              1
   1       6.66667e+01
   2       6.66667e+01
   3       8.33333e+01
   4      -9.42809e+01
   5       5.00000e+01
   6       2.35702e+01
   7       8.33333e+01
   8      -1.17851e+02
   9      -8.33333e+01
  10      -0.00000e+00
```

```
  11       6.66667e+01
  12       8.33333e+01

---
--- Part 6: Print support reactions and element-level forces ...
---
--- Support Reactions and Element-Level Forces: Load Case 1:
---
---   Reaction A: R_ax =   -0.00 ...
---             : R_ay =   66.67 ...
---   Reaction D: R_dy =   33.33 ...
---
--- Element Level Forces:
---
---   Element A-B: F1 =   66.67 ...
---   Element B-C: F2 =   66.67 ...
---   Element C-D: F3 =   33.33 ...
---   Element A-E: F4 =  -94.28 ...
---   Element B-E: F5 =  100.00 ...
---   Element C-E: F6 =  -47.14 ...
---   Element C-F: F7 =   33.33 ...
---   Element D-F: F8 =  -47.14 ...
---   Element E-F: F9 =  -33.33 ...
---
--- Support Reactions and Element-Level Forces: Load Case 2:
---
---   Reaction A: R_ax =   -0.00 ...
---             : R_ay =  100.00 ...
---   Reaction D: R_dy =  100.00 ...
---
--- Element Level Forces:
---
---   Element A-B: F1 =  100.00 ...
---   Element B-C: F2 =  100.00 ...
---   Element C-D: F3 =  100.00 ...
---   Element A-E: F4 = -141.42 ...
---   Element B-E: F5 =  100.00 ...
---   Element C-E: F6 =   -0.00 ...
---   Element C-F: F7 =  100.00 ...
---   Element D-F: F8 = -141.42 ...
---   Element E-F: F9 = -100.00 ...
---
--- Support Reactions and Element-Level Forces: Load Case 3:
---
---   Reaction A: R_ax =   -0.00 ...
---             : R_ay =   66.67 ...
---   Reaction D: R_dy =   83.33 ...
---
--- Element Level Forces:
---
---   Element A-B: F1 =   66.67 ...
---   Element B-C: F2 =   66.67 ...
---   Element C-D: F3 =   83.33 ...
---   Element A-E: F4 =  -94.28 ...
---   Element B-E: F5 =   50.00 ...
```

```
---   Element C-E: F6 =   23.57 ...
---   Element C-F: F7 =   83.33 ...
---   Element D-F: F8 = -117.85 ...
---   Element E-F: F9 =  -83.33 ...
---
--- Summary of Max/Min Reactions and Element-Level Forces ...
--- ----------------------------------------------------- ...
---
---   Reaction A: Minimum R_ax =   -0.00, Maximum R_ax =   -0.00 ...
---             : Minimum R_ay =   66.67, Maximum R_ay =  100.00 ...
---
---   Reaction D: Minimum R_dy =   33.33, Maximum R_dy =  100.00 ...
---
---   Element A-B:
---     Minimum F1 =   66.67 (T) ...
---     Maximum F1 =  100.00 (T) ...
---   Element B-C:
---     Minimum F2 =   66.67 (T) ...
---     Maximum F2 =  100.00 (T) ...
---   Element C-D:
---     Minimum F3 =   33.33 (T) ...
---     Maximum F3 =  100.00 (T) ...
---   Element A-E:
---     Minimum F4 = -141.42 (C) ...
---     Maximum F4 =  -94.28 (C) ...
---   Element B-E:
---     Minimum F5 =   50.00 (T) ...
---     Maximum F5 =  100.00 (T) ...
---   Element C-E:
---     Minimum F6 =  -47.14 (C) ...
---     Maximum F6 =   23.57 (T) ...
---   Element C-F:
---     Minimum F7 =   33.33 (T) ...
---     Maximum F7 =  100.00 (T) ...
---   Element D-F:
---     Minimum F8 = -141.42 (C) ...
---     Maximum F8 =  -47.14 (C) ...
---   Element E-F:
---     Minimum F9 = -100.00 (C) ...
---     Maximum F9 =  -33.33 (C) ...

--- ======================================== ...
--- Leave TestTrussAnalysis06.main()          ...
```