Indian Institute of Technology, Madras

Computational Tools: Algorithms, Data Structures and Programs - ID6105

Kanak Agarwal

November 7, 2025

1 Problem 1

1.1 Number Representation and Conversion

1.2 Binary to Decimal

$$\begin{aligned}1011001_2 &= 1\times 2^6 + 0\times 2^5 + 1\times 2^4 + 1\times 2^3 + 0\times 2^2 + 0\times 2^1 + 1\times 2^0\\ &= 64+16+8+1 = \boxed{89_{10}}\end{aligned}$$

$$\begin{aligned}110.00101_2 &= (1\times 2^2 + 1\times 2^1 + 0\times 2^0) + (0\times 2^{-1} + 0\times 2^{-2} + 1\times 2^{-3} + 0\times 2^{-4} + 1\times 2^{-5})\\ &= (4+2+0) + \left(\frac{1}{8}+\frac{1}{32}\right) = 6 + \frac{5}{32} = \boxed{6.15625_{10}}\end{aligned}$$

$$\begin{aligned}0.01011_2 &= 0\times 2^{-1} + 1\times 2^{-2} + 0\times 2^{-3} + 1\times 2^{-4} + 1\times 2^{-5}\\ &= \frac{1}{4}+\frac{1}{16}+\frac{1}{32} = \frac{11}{32} = \boxed{0.34375_{10}}\end{aligned}$$

1.3 Octal to Decimal

$$\begin{aligned}71563_8 &= 7\times 8^4 + 1\times 8^3 + 5\times 8^2 + 6\times 8^1 + 3\times 8^0\\ &= 7(4096) + 1(512) + 5(64) + 6(8) + 3(1)\\ &= 28672 + 512 + 320 + 48 + 3 = \boxed{29555_{10}}\end{aligned}$$

$$\begin{aligned}3.14_8 &= 3\times 8^0 + 1\times 8^{-1} + 4\times 8^{-2}\\ = 3 + \frac{1}{8} + \frac{4}{64} &= 3 + 0.125 + 0.0625 = \boxed{3.1875_{10}}\end{aligned}$$

1.4 Infinite Series

The infinite series,

$$f(n) = \sum_{i=1}^{n} i^{-4}$$

converges to $f(n) = \pi^4/90$ as $n \to \infty$.

Code:

```python
import numpy as np

n = 10000
f_true = (np.pi**4)/90

# Forward sum
f_forward = np.float32(0.0)
for i in range(1, n+1):
    f_forward += np.float32(1.0 / (i**4))

# Reverse sum
f_reverse = np.float32(0.0)
for i in range(n, 0, -1):
    f_reverse += np.float32(1.0 / (i**4))

err_forward = abs((f_forward - f_true)/f_true) * 100
err_reverse = abs((f_reverse - f_true)/f_true) * 100

print(f"Forward sum = {f_forward}, Error = {err_forward:.6f}%")
print(f"Reverse sum = {f_reverse}, Error = {err_reverse:.6f}%")
```

Output:

```
Forward sum = 1.082322120666504, Error = 0.000099%
Reverse sum = 1.0823231935501099, Error = 0.000000%
```

The reverse sum is more accurate since the small terms are added first (the terms in the series i^{-4} decrease as i increases). In single precision, small terms are lost when added to a larger running total due to roundoff errors as opposed to vice versa.

1.5 Random access Memory (RAM)

1.5.1 Memory Required for Array

The given array has dimensions $200 \times 400 \times 520$.

$$\text{Number of elements} = 200 \times 400 \times 520 = 41{,}600{,}000$$

Each element is of double precision, requiring 64 bits per value:

$$64 \text{ bits} = 8 \text{ bytes}$$

$$\text{Total bytes} = 41{,}600{,}000 \times 8 = 332{,}800{,}000 \text{ bytes}$$

Convert to gigabytes (GB),

$$1 \text{ GB} = 2^{30} \text{ bytes} = 1{,}073{,}741{,}824 \text{ bytes}$$

$$\text{Memory required} = \frac{332{,}800{,}000}{1{,}073{,}741{,}824} = \boxed{0.31 \text{ GB}}$$

Code:

```python
import numpy as np
import sys
```

```
M, N, P = 200, 400, 520
array = np.zeros((M, N, P), dtype=np.float64) # float64 - double precision

num_elements = array.size
memory_bytes = array.nbytes
memory_GB = memory_bytes / (2**30)

print(f"Array dimensions: {M} x {N} x {P}")
print(f"Number of elements = {num_elements:,}")
print(f"Each element size = {array.itemsize} bytes")
print(f"Total size = {memory_bytes:,} bytes ({memory_GB:.2f} GB)")

print(f"System-reported size (including Python object overhead):
 ↪  {sys.getsizeof(array):,} bytes")
```

Output:

```
Array dimensions: 200 x 400 x 520
Number of elements = 41,600,000
Each element size = 8 bytes
Total size = 332,800,000 bytes (0.31 GB)
System-reported size (including Python object overhead): 332,800,144 bytes
```

1.6 Maximum Array Size

My laptop has 32 GB of RAM. Accounting for the operating system and overhead, approximately 28 GB is available for array storage. Each element (double precision) requires 8 bytes. Therefore, the maximum number of storable elements is:

$$\text{Max elements} = \frac{28 \times 2^{30}}{8} = 3.76 \times 10^9$$

If the array is roughly cubic, we have:

$$M = N = P = \sqrt[3]{3.76 \times 10^9} \approx 1550$$

Hence, the maximum array size is approximately:

$$\boxed{M \times N \times P = 1550^3 \approx 3.7 \times 10^9 \text{ elements}}$$

2 Problem 2

The equations are,

$$y = -x^2 + x + 0.75$$
$$y + 5xy = x^2$$

2.1 Fixed Point

Rearranging,

$$x = \frac{5 \pm \sqrt{25y^2 + 4y}}{2}$$

Code:

```python
import math

def fixed_point(x0, y0, tol=1e-10, max_iter=200):
    x, y = x0, y0
    history = [(0, x, y)]
    for k in range(1, max_iter+1):
        y_new = -x**2 + x + 0.75
        disc = 25*y_new*y_new + 4*y_new
        if disc < 0:
            raise ValueError("Negative discriminant in fixed-point step")
        r1 = (5*y_new + math.sqrt(disc)) / 2.0
        r2 = (5*y_new - math.sqrt(disc)) / 2.0
        x_new = r1 if abs(r1 - x) < abs(r2 - x) else r2
        history.append((k, x_new, y_new))

        if abs(x_new - x) < tol and abs(y_new - y) < tol:
            return x_new, y_new, k, history
        x, y = x_new, y_new

    return x, y, max_iter, history

x0 = 1.2; y0 = 1.2
xf, yf, iters_fp, hist_fp = fixed_point(x0, y0)
for row in hist_fp:
    print("Iter {:3d}: x = {:.10f}, y = {:.10f}".format(row[0], row[1], row[2]))
```

Output:

```
Iter   0: x = 1.2000000000, y = 1.2000000000
Iter   1: x = -0.1863777746, y = 0.5100000000
Iter   2: x = -0.1868040445, y = 0.5288855506
Iter   3: x = -0.1867912353, y = 0.5283002045
Iter   4: x = -0.1867916206, y = 0.5283177992
Iter   5: x = -0.1867916091, y = 0.5283172698
Iter   6: x = -0.1867916094, y = 0.5283172857
Iter   7: x = -0.1867916094, y = 0.5283172853
Iter   8: x = -0.1867916094, y = 0.5283172853
```

2.2 Newton Raphson

In multivariate problems the Newton Raphson can be modified as,

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^{n+1} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^{n} - \left(J^{-1}\right)^n \begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{bmatrix}^n$$

where J^{-1} is,

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix}$$

Code:

```python
import numpy as np

def f_J(x, y):
    f1 = y + x*x - x - 0.75
    f2 = y + 5*x*y - x*x
    J = np.array([[2*x - 1, 1.0],
                  [-2*x + 5*y, 1.0 + 5*x]], dtype=float)
    F = np.array([f1, f2], dtype=float)
    return F, J

def newton_raphson_multi(x0, y0, tol=1e-10, max_iter=50):
    x, y = x0, y0
    history = [(0, x, y)]
    for k in range(1, max_iter+1):
        F, J = f_J(x, y)
        delta = np.linalg.solve(J, -F)
        x_new = x + delta[0]
        y_new = y + delta[1]
        history.append((k, x_new, y_new, np.linalg.norm(delta)))
        if np.linalg.norm(delta) < tol:
            return x_new, y_new, k, history
        x, y = x_new, y_new
    return x, y, max_iter, history

x0 = 1.2; y0 = 1.2
xn, yn, iters_nr, hist_nr = newton_raphson_multi(x0, y0)
for row in hist_nr:
    print("Iter {:3d}: x = {:.10f}, y = {:.10f}".format(row[0], row[1], row[2]))
```

Output:

```
Iter   0: x = 1.2000000000, y = 1.2000000000
Iter   1: x = 1.5435483871, y = 0.0290322581
Iter   2: x = 1.3941229465, y = 0.2228721189
Iter   3: x = 1.3724545855, y = 0.2392925142
Iter   4: x = 1.3720655204, y = 0.2395018794
Iter   5: x = 1.3720654058, y = 0.2395019280
Iter   6: x = 1.3720654058, y = 0.2395019280
```

2.3 Analytical Approach

The given nonlinear equations are,

$$\begin{cases} y = -x^2 + x + 0.75, \\ y + 5xy = x^2 \end{cases}$$

Substituting $y = -x^2 + x + 0.75$ from the first equation into the second,

$$(-x^2 + x + 0.75) + 5x(-x^2 + x + 0.75) = x^2$$

Expanding and simplifying,

$$-x^2 + x + 0.75 - 5x^3 + 5x^2 + 3.75x = x^2$$

$$-5x^3 + 4x^2 + 4.75x + 0.75 = 0$$

Multiplying through by -1,

$$\boxed{5x^3 - 4x^2 - 4.75x - 0.75 = 0}$$

Solving the above cubic gives three roots, two of which are real:

$$x_1 \approx -0.1867916094, \quad x_2 \approx 1.3720654058, \quad x_3 \approx 0.0157262036 + 0.6553394114i$$

The corresponding y-values (using $y = -x^2 + x + 0.75$) are,

$$y_1 = -(-0.1868)^2 + (-0.1868) + 0.75 = 0.5283,$$
$$y_2 = -(1.3721)^2 + (1.3721) + 0.75 = 0.2395$$

Thus the real roots are,

$$\boxed{(x_1, y_1) = (-0.1868,\ 0.5283), \qquad (x_2, y_2) = (1.3721,\ 0.2395)}$$

These match with the results of the numerical approach.

3 Problem 3

Prove: Bisection (midpoint) applied to the square root of any positive number a is equivalent to the Neqton-Raphson algorithm, i.e.,

$$x = \frac{x + a/x}{2}$$

Proof:

Newton–Raphson is given by,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Substituting $f(x) = x^2 - a$,

$$x_{n+1} = x_n - \frac{x_n^2 - a}{2x_n}.$$

Simplifying,

$$x_{n+1} = \frac{2x_n^2 - (x_n^2 - a)}{2x_n} = \frac{x_n^2 + a}{2x_n}.$$

Rewriting,

$$x_{n+1} = \frac{x_n}{2} + \frac{a}{2x_n} = \frac{x_n + a/x_n}{2}.$$

Hence,

$$\boxed{x_{n+1} = \frac{x_n + a/x_n}{2}}$$

In conclusion, the divide and average method is mathematically equivalent to the Newton Raphson method applied to $f(x) = x^2 - a = 0$.

4 Problem 4

Code:

```python
import math

R = 3.0
V_target = 30.0
pi = math.pi

def V(h):
    return (1/3) * pi * h**2 * (3*R - h)

def f(h):
    return V(h) - V_target

def df(h):
    return (1/3) * pi * (6*R*h - 3*h**2)

h = 3.0 # Initial guess
iter = []

for i in range(3):
    fh = f(h)
    dfh = df(h)
    delta = fh / dfh
    h_new = h - delta
    # approximate relative error after iteration: |(h_new - h)/h_new|
    approx_rel_err = abs((h_new - h)/h_new)
    iter.append({
        'iteration': i,
        'h_old': h,
        'f(h)': fh,
        "f'(h)": dfh,
        'delta': delta,
        'h_new': h_new,
        'approx_rel_err': approx_rel_err
    })
    h = h_new

# Results
print(f"{'Iter':<5} {'h_old':<12} {'f(h)':<15} {'f\'(h)':<15} {'delta':<15}
↪  {'h_new':<12} {'Approx Rel Err':<15}")
for entry in iter:
    print(f"{entry['iteration']:<5} {entry['h_old']:<12.6f} {entry['f(h)']:<15.6f}
    ↪  {entry['f\'(h)']:<15.6f} {entry['delta']:<15.6f} {entry['h_new']:<12.6f}
    ↪  {entry['approx_rel_err']:<15.6e}")
```

Output:

```
Iter  h_old      f(h)         f'(h)        delta       h_new      Approx Rel Err
0     3.000000   26.548668    28.274334    0.938967    2.061033   4.555808e-01
1     2.061033   0.866921     25.504520    0.033991    2.027042   1.676871e-02
2     2.027042   0.003449     25.300354    0.000136    2.026906   6.726269e-05
```

The tank must be filled to approximately 2.0269 m so that it hold 30 m^3.

5 Problem 5

Code:

```
def f(x):
    return 0.0074*x**4 - 0.284*x**3 + 3.355*x**2 - 12.183*x + 5.0

def df(x):
    return 4*0.0074*x**3 - 3*0.284*x**2 + 2*3.355*x - 12.183

x = 16.15 # Initial guess
tol = 1e-6
max_iter = 100
```

```
results = []
for i in range(1, max_iter+1):
    fx = f(x)
    dfx = df(x)
    if dfx == 0:
        results.append((i, x, fx, dfx, None, "Zero derivative"))
        break
    delta = fx/dfx
    x_new = x - delta
    approx_rel_err = abs((x_new - x)/x_new) if x_new != 0 else abs(x_new - x)
    results.append((i, x, fx, dfx, delta, x_new, approx_rel_err))
    x = x_new
    if abs(delta) < tol:
        break

print(f"{'Iter':<5} {'x_old':<12} {'f(x)':<15} {'f\'(x)':<15} {'delta':<15}
↪  {'x_new':<12} {'Approx Rel Err':<15}")
for res in results:
    if len(res) == 6:
        iter_num, x_old, fx, dfx, delta, message = res
        print(f"{iter_num:<5} {x_old:<12.6f} {fx:<15.6f} {dfx:<15.6f} {'N/A':<15}
         ↪  {'N/A':<12} {message:<15}")
    else:
        iter_num, x_old, fx, dfx, delta, x_new, approx_rel_err = res
        print(f"{iter_num:<5} {x_old:<12.6f} {fx:<15.6f} {dfx:<15.6f} {delta:<15.6f}
         ↪  {x_new:<12.6f} {approx_rel_err:<15.6e}")

f15 = f(15)
f20 = f(20)
print()
print(f"f(15) = {f15:.6f}")
print(f"f(20) = {f20:.6f}")
```

Output:

```
Iter  x_old         f(x)           f'(x)          delta          x_new          Approx Rel Err
1     16.150000    -9.574455      -1.353682       7.072898       9.077102       7.792021e-01
2     9.077102     8.678763       0.662596        13.098112      -4.021010      3.257419e+00
3     -4.021010    128.631764     -54.863959      -2.344559      -1.676451      1.398525e+00
4     -1.676451    36.249950      -25.965987      -1.396055      -0.280396      4.978871e+00
5     -0.280396    8.686147       -14.132095      -0.614640      0.334244       1.838897e+00
6     0.334244     1.292213       -10.034304      -0.128779      0.463023       2.781275e-01
7     0.463023     0.050416       -9.255836       -0.005447      0.468470       1.162708e-02
8     0.468470     0.000088       -9.223505       -0.000010      0.468480       2.038275e-05
9     0.468480     0.000000       -9.223449       -0.000000      0.468480       6.258006e-11

f(15) = -6.745000
f(20) = 15.340000
```

The final value of the root from the initial guess of 16.15 is 0.4684 which is not in the interval [15,20]. This happens because the Newton Raphson method is not globally convergent. It is locally convergent and depends heavily on the initial guess and its deriviatives. The delta in the initial iterations is relatively large moving the solution away from the initial interval, the algorithm eventually converged, however, to a different real root. Changing the initial guess to 17, we obtain a root in this interval as demonstrated below.

```
Iter  x_old        f(x)          f'(x)         delta        x_new        Approx Rel Err
1     17.000000    -9.752600     1.083800      -8.998524    25.998524     3.461167e-01
```

```
2     25.998524   346.103090   106.541483   3.248529   22.749995    1.427925e-01
3     22.749995   102.532341   48.032765    2.134633   20.615361    1.035458e-01
4     20.615361   28.042877    23.388984    1.198978   19.416383    6.175084e-02
5     19.416383   6.152592     13.569363    0.453418   18.962966    2.391071e-02
6     18.962966   0.703226     10.525864    0.066809   18.896156    3.535604e-03
7     18.896156   0.014042     10.106728    0.001389   18.894767    7.353361e-05
8     18.894767   0.000006     10.098090    0.000001   18.894766    3.145300e-08
```

6 Problem 6

The equation of the circle is given by,

$$(x+1)^2 + (y-2)^2 = 16$$

Rewriting and treating y as a function of x,

$$y(x) = 2 \pm \sqrt{16 - (x+1)^2}$$

The secant method is given by,

$$x_{n+1} = x_n - g(x_n)\frac{x_n - x_{n-1}}{g(x_n) - g(x_{n-1})}$$

$g(x)$ is split into its positive and negative components.

Code:

```python
import math

def y_plus(x):
    arg = 16 - (x+1)**2
    if arg < 0: return None  # outside domain
    return 2.0 + math.sqrt(arg)

def y_minus(x):
    arg = 16 - (x+1)**2
    if arg < 0: return None
    return 2.0 - math.sqrt(arg)

def secant_method(func, x0, x1, tol=1e-6, max_iter=100):
    results = []
    for i in range(1, max_iter+1):
        f_x0 = func(x0)
        f_x1 = func(x1)
        if f_x0 is None or f_x1 is None:
            results.append((i, x0, f_x0, x1, f_x1, None, "Out of domain"))
            break
        if f_x1 - f_x0 == 0:
            results.append((i, x0, f_x0, x1, f_x1, None, "Zero denominator"))
            break
        x2 = x1 - f_x1 * (x1 - x0) / (f_x1 - f_x0)
        approx_rel_err = abs((x2 - x1)/x2) if x2 != 0 else abs(x2 - x1)
        results.append((i, x0, f_x0, x1, f_x1, x2, approx_rel_err))
        if abs(x2 - x1) < tol:
            break
        x0, x1 = x1, x2
    return results

# Initial guesses
```

```
x0, x1 = 0.5, 3.0

results_plus = secant_method(y_plus, x0, x1)
results_minus = secant_method(y_minus, x0, x1)

print("Secant Method for y_plus:")
print(f"{'Iter':<5} {'x0':<12} {'f(x0)':<22} {'x1':<12} {'f(x1)':<25} {'x2':<20}
↪  {'Approx Rel Err':<15}")
for res in results_plus:
    print(f"{res[0]:<5} {res[1]:<12.6f} {res[2] if res[2] is not None else 'N/A':<22}
    ↪  {res[3]:<12.6f} {res[4] if res[4] is not None else 'N/A':<25} {res[5] if
    ↪  res[5] is not None else 'N/A':<20} {res[6] if res[6] is not None else
    ↪  'N/A':<20}")

print("\nSecant Method for y_minus:")
print(f"{'Iter':<5} {'x0':<12} {'f(x0)':<22} {'x1':<12} {'f(x1)':<25} {'x2':<20}
↪  {'Approx Rel Err':<15}")
for res in results_minus:
    print(f"{res[0]:<5} {res[1]:<12.6f} {res[2] if res[2] is not None else 'N/A':<22}
    ↪  {res[3]:<12.6f} {res[4] if res[4] is not None else 'N/A':<25} {res[5] if
    ↪  res[5] is not None else 'N/A':<20} {res[6] if res[6] is not None else
    ↪  'N/A':<20}")
```

Output:

```
Secant Method for y_plus:
Iter x0        f(x0)               x1        f(x1)  x2                  Approx Rel Err
1    0.500000  5.7080992435478315  3.000000  2.0    4.348399724926484
↪  0.3100910243363794
2    3.000000  2.0                 4.348400  N/A    N/A                 Out of domain

Secant Method for y_minus:
Iter  x0          f(x0)                   x1            f(x1)                      x2
↪  Approx Rel Err
1     0.500000    -1.7080992435478315    3.000000     2.0
↪  1.651600275073516    0.81642013826043
2     3.000000    2.0                    1.651600     -0.9948315447166731
↪  2.099515478721826    0.21334217736798897
3     1.651600    -0.9948315447166731    2.099515     -0.5284390040425753
↪  2.607019015912479    0.19466813785898776
4     2.099515    -0.5284390040425753    2.607019     0.2710078603863548
↪  2.4349782538903613   0.07065392134293132
5     2.607019    0.2710078603863548     2.434978     -0.04961567014411861
↪  2.4616011211693456   0.01081526452439116
6     2.434978    -0.04961567014411861   2.461601     -0.004324743627920569
↪  2.4641432876578273   0.0010316634187689815
7     2.461601    -0.004324743627920569  2.464143     7.218065870695511e-05
↪  2.4641015550193464   1.6936249399275942e-05
8     2.464143    7.218065870695511e-05  2.464102     -1.0412813411520006e-07
↪  2.4641016151363084   2.4397111590610957e-08
```

In the positive quadrant the solution goes away from the actual solution (2.4641) and effectively blows up (the solution diverges). This isn't the case in the negative quadrant where it converges.

7 Problem 7

Code:

```
import numpy as np

def poly_eval(coeffs, x): # Evaluate the polynomial and its derivative at x using
↪   Horner's method
    n = len(coeffs) - 1
    p, dp = coeffs[0], 0
    for i in range(1, n + 1):
        dp = dp * x + p
        p = p * x + coeffs[i]
    return p, dp

def newton_root(coeffs, x0, tol=1e-8, max_iter=100): # Find the root using
↪   Newton-Raphson
    x = x0
    for _ in range(max_iter):
        fx, dfx = poly_eval(coeffs, x)
        if abs(dfx) < 1e-12:
            break
        x_new = x - fx / dfx
        if abs(x_new - x) < tol:
            return x_new
        x = x_new
    return x

def simplify(coeffs, root):  # divide the polynomial by (x - root)
    n = len(coeffs)
    new_coeffs = [coeffs[0]]
    for i in range(1, n - 1):
        new_coeffs.append(coeffs[i] + root * new_coeffs[-1])
    remainder = coeffs[-1] + root * new_coeffs[-1]
    return np.array(new_coeffs, dtype=float), remainder

def recursive_roots(coeffs, guesses):
    roots = []
    c = coeffs.copy()
    for g in guesses:
        root = newton_root(c, g)
        roots.append(root)
        c, _ = simplify(c, root)
    return roots

num = [1, 12.5, 50.5, 66]
den = [1, 19, 122, 296, 192]

guesses = [-3, -4, -5]
roots_num = recursive_roots(num, guesses)

guesses = [-1, -3, -5, -7]
roots_den = recursive_roots(den, guesses)

print("Numerator Roots:")
for r in roots_num:
    print(f"{r:.6f}")

print("\nDenominator Roots:")
for r in roots_den:
    print(f"{r:.6f}")
```

Output:

```
Numerator Roots:
-3.000000
-4.000000
-5.500000

Denominator Roots:
-1.000000
-4.000000
-6.000000
-8.000000
```

Therefore,

$$G(s) = \frac{C(s)}{N(s)} = \frac{s^3 + 12.5s^2 + 50.5s + 66}{s^4 + 19s^3 + 122s^2 + 296s + 192} = \frac{(s+3)(s+5.5)}{(s+1)(s+6)(s+8)}$$

8 Problem 8

The three-dimensional stress field is represented as

$$\sigma = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{xy} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{xz} & \sigma_{yz} & \sigma_{zz} \end{bmatrix} = \begin{bmatrix} 10 & 14 & 25 \\ 14 & 7 & 15 \\ 25 & 15 & 16 \end{bmatrix} \text{ MPa.}$$

To obtain the principal stresses, we form the determinant

$$\begin{vmatrix} \sigma_{xx} - \sigma & \sigma_{xy} & \sigma_{xz} \\ \sigma_{xy} & \sigma_{yy} - \sigma & \sigma_{yz} \\ \sigma_{xz} & \sigma_{yz} & \sigma_{zz} - \sigma \end{vmatrix} = 0,$$

or equivalently,

$$\begin{vmatrix} 10 - \sigma & 14 & 25 \\ 14 & 7 - \sigma & 15 \\ 25 & 15 & 16 - \sigma \end{vmatrix} = 0.$$

Expanding this determinant leads to the characteristic cubic equation,

$$\sigma^3 - I_1\sigma^2 + I_2\sigma - I_3 = 0,$$

where I_1, I_2, and I_3 are the stress invariants defined as,

$$I_1 = \sigma_{xx} + \sigma_{yy} + \sigma_{zz}$$
$$I_1 = 10 + 7 + 16 = \boxed{33 \text{ MPa}}$$

$$I_2 = \sigma_{xx}\sigma_{yy} + \sigma_{yy}\sigma_{zz} + \sigma_{zz}\sigma_{xx} - (\sigma_{xy}^2 + \sigma_{yz}^2 + \sigma_{zx}^2)$$
$$I_2 = (10)(7) + (7)(16) + (16)(10) - (14^2 + 15^2 + 25^2)$$
$$I_2 = (70 + 112 + 160) - (196 + 225 + 625) = 342 - 1046 = \boxed{-704 \text{ MPa}^2}$$

$$I_3 = \sigma_{xx}\sigma_{yy}\sigma_{zz} + 2\sigma_{xy}\sigma_{yz}\sigma_{zx} - (\sigma_{xx}\sigma_{yz}^2 + \sigma_{yy}\sigma_{zx}^2 + \sigma_{zz}\sigma_{xy}^2)$$
$$I_3 = (10)(7)(16) + 2(14)(15)(25) - [\,10(15^2) + 7(25^2) + 16(14^2)\,]$$
$$I_3 = 1120 + 10500 - (2250 + 4375 + 3136) = 11620 - 9756 = \boxed{1859 \text{ MPa}^3.}$$

Substituting,

$$\sigma^3 - 33\sigma^2 - 704\sigma - 1859 = 0,$$

<u>**Code:**</u>

```python
import numpy as np

def poly_eval(coeffs, x): # Evaluate the polynomial and its derivative at x using
↪   Horner's method
    n = len(coeffs) - 1
    p, dp = coeffs[0], 0
    for i in range(1, n + 1):
        dp = dp * x + p
        p = p * x + coeffs[i]
    return p, dp

def newton_root(coeffs, x0, tol=1e-8, max_iter=100): # Find the root using
↪   Newton-Raphson
    x = x0
    for _ in range(max_iter):
        fx, dfx = poly_eval(coeffs, x)
        if abs(dfx) < 1e-12:
            break
        x_new = x - fx / dfx
        if abs(x_new - x) < tol:
            return x_new
        x = x_new
    return x

def simplify(coeffs, root):  # divide the polynomial by (x - root)
    n = len(coeffs)
    new_coeffs = [coeffs[0]]
    for i in range(1, n - 1):
        new_coeffs.append(coeffs[i] + root * new_coeffs[-1])
    remainder = coeffs[-1] + root * new_coeffs[-1]
    return np.array(new_coeffs, dtype=float), remainder

def recursive_roots(coeffs, guesses):
    roots = []
    c = coeffs.copy()
    for g in guesses:
        root = newton_root(c, g)
        roots.append(root)
        c, _ = simplify(c, root)
    return roots

poly = [1, -33, -704, -1859]
guesses = [50, -5, -10]
roots_num = recursive_roots(poly, guesses)

print("Roots:")
for r in roots_num:
    print(f"{r:.6f}")
```

Output:

```
Roots:
48.354284
-3.150211
-12.204073
```

Therefore the three roots correspond to the principal stresses,

$$\boxed{\sigma_1 = 48.354284 \text{ MPa}, \quad \sigma_2 = -3.150211 \text{ MPa}, \quad \sigma_3 = -12.204073 \text{ MPa}.}$$

9 Problem 9

In multivariate problems the Newton Raphson can be modified as,

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^{n+1} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^{n} - \left(J^{-1}\right)^n \begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{bmatrix}^{n}$$

where J^{-1} is,

$$J = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} \end{bmatrix}$$

Code:

```python
import math, numpy as np
# Newton's method for system of equations - multivariate problem

T0 = 450.0
T3 = 25.0

def q1(T1):
    return 1e-9 * ( (T0 + 273.0)**4 - (T1 + 273.0)**4 )

def dq1_dT1(T1):
    return -1e-9 * 4.0 * (T1 + 273.0)**3

def q2(T1, T2):
    return 4.0 * (T1 - T2)

def dq2_dT1(T1, T2):
    return 4.0
def dq2_dT2(T1, T2):
    return -4.0

def q3(T2):
    return 1.3 * (T2 - T3)**(4.0/3.0)

def dq3_dT2(T2):
    return 1.3 * (4.0/3.0) * (T2 - T3)**(1.0/3.0)

def residuals(T1, T2):
    r1 = q1(T1) - q2(T1, T2)
    r2 = q2(T1, T2) - q3(T2)
    return np.array([r1, r2], dtype=float)

def jacobian(T1, T2):
    # dr1/dT1, dr1/dT2; dr2/dT1, dr2/dT2
    dr1_dT1 = dq1_dT1(T1) - dq2_dT1(T1, T2)
    dr1_dT2 = - dq2_dT2(T1, T2)  # dq1/dT2 = 0
    dr2_dT1 = dq2_dT1(T1, T2)
    dr2_dT2 = dq2_dT2(T1, T2) - dq3_dT2(T2)
    return np.array([[dr1_dT1, dr1_dT2],[dr2_dT1, dr2_dT2]], dtype=float)

# Newton iterations
T1 = 200.0
T2 = 100.0
tol = 1e-10
max_iter = 50
history = []
```

```
for k in range(1, max_iter+1):
    F = residuals(T1, T2)
    J = jacobian(T1, T2)
    delta = np.linalg.solve(J, -F)
    T1_new = T1 + delta[0]
    T2_new = T2 + delta[1]
    approx_rel_err = np.max(np.abs(delta) / np.maximum([abs(T1_new), abs(T2_new)],
    ↪  1e-12))
    history.append((k, T1, T2, F[0], F[1], delta[0], delta[1], T1_new, T2_new,
    ↪  approx_rel_err))
    T1, T2 = T1_new, T2_new
    if approx_rel_err < tol:
        break

# Results
print("iter     T1_old     T2_old     r1          r2          dT1          dT2     T1_new
↪  T2_new    rel_err")
for row in history:
    print("{:3d} {:10.6f} {:10.6f} {:10.6e} {:10.6e} {:10.6e} {:10.6e} {:10.6f}
    ↪  {:10.6f} {:10.6e}".format(*row))

print("\nFinal T1, T2:", T1, T2)
print("Final fluxes q1, q2, q3:", q1(T1), q2(T1,T2), q3(T2))
```

Output:

```
iter T1_old      T2_old      r1              r2              dT1             dT2
↪   T1_new      T2_new      rel_err
  1  200.000000  100.000000  -1.768091e+02  -1.117342e+01  -6.008165e+01
   ↪  -2.223745e+01  139.918353  77.762549  4.294050e-01
  2  139.918353  77.762549   -4.448395e+00  -8.640004e+00  -2.806216e+00
   ↪  -1.891684e+00  137.112137  75.870865  2.493294e-02
  3  137.112137  75.870865   -8.019605e-03  -7.408143e-02  -1.329950e-02
   ↪  -1.221197e-02  137.098838  75.858653  1.609832e-04
  4  137.098838  75.858653   -1.784917e-07  -3.138192e-06  -5.045995e-07
   ↪  -4.947792e-07  137.098837  75.858652  6.522383e-09
  5  137.098837  75.858652   2.842171e-14   0.000000e+00   1.037035e-14    3.980177e-15
   ↪  137.098837  75.858652  7.564143e-17

Final T1, T2: 137.09883742088195 75.85865249812439
Final fluxes q1, q2, q3: 244.96073969103028 244.96073969103026 244.96073969103026
```

10 Problem 10

The three sets in the matrix form are given by,

$$\begin{bmatrix} 9 & 3 & 1 \\ -6 & 0 & 8 \\ 2 & 5 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 13 \\ 2 \\ 6 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 6 \\ 1 & 5 & -1 \\ 4 & 2 & -2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 8 \\ 5 \\ 4 \end{bmatrix}$$

$$\begin{bmatrix} -3 & 4 & 5 \\ -2 & 2 & -4 \\ 0 & 2 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 6 \\ -3 \\ 1 \end{bmatrix}$$

None of the sets satsifies the diagonally dominant condition, i.e.,

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|$$

Since none of them satisfy this check, they will not converge.

11 Problem 11

From Newton's second law for each mass ($\sum F_x = m\ddot{x}$),

$$\ddot{x}_1 + \frac{k_1 + k_2}{m_1} x_1 - \frac{k_2}{m_1} x_2 = 0,$$

$$\ddot{x}_2 - \frac{k_2}{m_2} x_1 + \frac{k_2 + k_3}{m_2} x_2 - \frac{k_3}{m_2} x_3 = 0,$$

$$\ddot{x}_3 - \frac{k_3}{m_3} x_2 + \frac{k_3 + k_4}{m_3} x_3 = 0$$

In matrix form,

$$\mathbf{0} = \ddot{\mathbf{x}} + \mathbf{K}_m \, \mathbf{x},$$

where

$$\ddot{\mathbf{x}} = \begin{bmatrix} \ddot{x}_1 \\ \ddot{x}_2 \\ \ddot{x}_3 \end{bmatrix}, \qquad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

and

$$\mathbf{K}_m = \begin{bmatrix} \frac{k_1 + k_2}{m_1} & -\frac{k_2}{m_1} & 0 \\ -\frac{k_2}{m_2} & \frac{k_2 + k_3}{m_2} & -\frac{k_3}{m_2} \\ 0 & -\frac{k_3}{m_3} & \frac{k_3 + k_4}{m_3} \end{bmatrix}$$

Thus,

$$\ddot{\mathbf{x}} = -\,\mathbf{K}_m \, \mathbf{x}.$$

Given,

$$k_1 = k_4 = 10 \text{ N/m}, \quad k_2 = k_3 = 30 \text{ N/m}, \quad m_1 = m_2 = m_3 = 2 \text{ kg}$$

Substituting,

$$\mathbf{K}_m = \begin{bmatrix} \frac{10 + 30}{2} & -\frac{30}{2} & 0 \\ -\frac{30}{2} & \frac{30 + 30}{2} & -\frac{30}{2} \\ 0 & -\frac{30}{2} & \frac{30 + 10}{2} \end{bmatrix} = \begin{bmatrix} 20 & -15 & 0 \\ -15 & 30 & -15 \\ 0 & -15 & 20 \end{bmatrix}$$

At a specific time given that,

$$x_1 = 0.05 \text{ m}, \qquad x_2 = 0.04 \text{ m}, \qquad x_3 = 0.03 \text{ m}$$

Then,

$$\ddot{\mathbf{x}} = -\mathbf{K}_m \begin{bmatrix} 0.05 \\ 0.04 \\ 0.03 \end{bmatrix} = -\begin{bmatrix} 20 & -15 & 0 \\ -15 & 30 & -15 \\ 0 & -15 & 20 \end{bmatrix} \begin{bmatrix} 0.05 \\ 0.04 \\ 0.03 \end{bmatrix}$$

from matrix multiplication,

$$\begin{aligned} \ddot{x}_1 &= -\big[20(0.05) - 15(0.04)\big] = -0.40, \\ \ddot{x}_2 &= -\big[-15(0.05) + 30(0.04) - 15(0.03)\big] = 0 \\ \ddot{x}_3 &= -\big[-15(0.04) + 20(0.03)\big] = 0 \end{aligned}$$

Therefore,

$$\ddot{\mathbf{x}} = \begin{bmatrix} -0.40 \\ 0 \\ 0 \end{bmatrix} \text{ m/s}^2$$

Hence, at this instant only the first mass experiences a nonzero acceleration, while the second and third masses are in equilibrium.

12 Problem 12

Code:

```
# Initialization
nx = 4
ny = 4
T = np.zeros((nx, ny))

# Boundary conditions
T[0, :] = 25  # Top boundary
T[-1, :] = 75  # Bottom boundary
T[:, 0] = 200   # Left boundary
T[:, -1] = 0   # Right boundary

print("Initial temperature distribution with Boundary conditions:")
print(T)
print()

# Iterative solution
tolerance = 1e-4
max_iterations = 10000
for iteration in range(max_iterations):
    T_old = T.copy()
    for i in range(1, nx - 1):
        for j in range(1, ny - 1):
            T[i, j] = 0.25 * (T_old[i + 1, j] + T_old[i - 1, j] + T_old[i, j + 1] +
             ↪  T_old[i, j - 1])
    # Check for convergence
    if np.max(np.abs(T - T_old)) < tolerance:
        print(f"Converged after {iteration} iterations.")
        break
print()
print("Steady-state temperature distribution:")
print(T)
```

Output:

```
Initial temperature distribution with Boundary conditions:
[[200.  25.  25.   0.]
 [200.   0.   0.   0.]
 [200.   0.   0.   0.]
 [200.  75.  75.   0.]]

Converged after 19 iterations.

Steady-state temperature distribution:
[[200.          25.          25.           0.        ]
 [200.          93.74992847  43.74992847   0.        ]
 [200.         106.24992847  56.24992847   0.        ]
 [200.          75.          75.           0.        ]]
```