In [3]:

```python
# Given data
b = 2  # Width of the channel in meters
Q = 4.8  # Discharge in m^3/s
y1 = 1.6  # Initial depth of flow in meters
hump_height = 0.1  # Hump height in meters
g = 9.81  # Acceleration due to gravity in m/s^2

# Calculate velocity at upstream section (v1)
v1 = Q / (b * y1)

# Calculate y2 (depth downstream of the hump)
y2 = Q / (b * v1)

# Calculate specific energies at upstream (E1) and downstream (E2) sections
E1 = y1 + (v1**2) / (2 * g)
E2 = y2 + (v1**2) / (2 * g)

# Calculate the likely change in water surface elevation (Δy)
delta_y = y2 - y1

print(f"a.) Likely change in water surface elevation (Δy): {delta_y:.2f} meters")

# Given data for the higher hump height
hump_height_high = 0.5  # Hump height in meters

# Calculate y2 (depth downstream of the higher hump)
y2_high = Q / (b * v1)

print(f"b.) Upstream depth of flow for higher hump: {y1:.2f} meters")
print(f"    Downstream depth of flow for higher hump: {y2_high:.2f} meters")

max_hump_height = 1.0  # Maximum hump height to test
increment = 0.01  # Increment for hump height

while hump_height < max_hump_height:
    y2_test = Q / (b * v1)  # Calculate downstream depth for the current hump height
    E2_test = y2_test + (v1**2) / (2 * g)  # Calculate specific energy downstream
    if E2_test < E1:
        break
    hump_height += increment

max_delta_z = hump_height  # Maximum rise in bed level

print(f"c.) Maximum rise in bed level possible without changing the upstream condition (Δz): {max_delta_z:.2f}
```

```
a.) Likely change in water surface elevation (Δy): 0.00 meters
b.) Upstream depth of flow for higher hump: 1.60 meters
    Downstream depth of flow for higher hump: 1.60 meters
c.) Maximum rise in bed level possible without changing the upstream condition (Δz): 1.00 meters
```

In [18]:

```python
import numpy as np

# Given data
initial_ph = 10
final_ph_1 = 4.5
final_ph_2 = 8.3
volume_1 = 30  # mL
volume_2 = 11  # mL
normality_h2so4 = 0.02  # N

# Constants for molecular weights
molecular_weight_h2so4 = 98.08  # g/mol (molecular weight of H2SO4)
molecular_weight_caco3 = 100.09  # g/mol (molecular weight of CaCO3)

# Calculate the equivalent weight of CaCO3
equivalent_weight_caco3 = molecular_weight_caco3 / 2  # Equivalent weight (eq/g) for CaCO3

# Convert volumes to liters
volume_1_liters = volume_1 / 1000  # Convert mL to L
volume_2_liters = volume_2 / 1000  # Convert mL to L

# Calculate the moles of H2SO4 used in each titration
moles_h2so4_1 = normality_h2so4 * volume_1_liters
moles_h2so4_2 = normality_h2so4 * volume_2_liters

# Define the system of linear equations:
# 1. Moles of OH- in initial solution - Moles of H2SO4 added = Moles of OH- in final solution at pH 4.5
# 2. Moles of OH- in initial solution - Moles of H2SO4 added = Moles of OH- in final solution at pH 8.3

# Define the coefficients matrix and the constants vector
coefficients_matrix = np.array([[1, -moles_h2so4_1], [1, -moles_h2so4_2]])
constants_vector = np.array([10**(-initial_ph), 10**(-initial_ph)])

# Solve the system of linear equations
alkalinity_moles = np.linalg.solve(coefficients_matrix, constants_vector)

# Calculate the total alkalinity in moles and convert it to mg/L as CaCO3
total_alkalinity_mg_per_L = alkalinity_moles * equivalent_weight_caco3 * 1000  # Convert moles to mg/L

# Print the results
print(f"a. Total Alkalinity of Water: {total_alkalinity_mg_per_L[0]:.2f} mg/L as CaCO3")
print(f"b. Concentration of Alkaline Species at pH 8.3: {total_alkalinity_mg_per_L[1]:.2f} mg/L as CaCO3")
```

```
a. Total Alkalinity of Water: 0.00 mg/L as CaCO3
b. Concentration of Alkaline Species at pH 8.3: 0.00 mg/L as CaCO3
```

In [13]:

```python
# Given data
b_eff = 230  # Effective width of the beam section in mm
d = 400  # Overall depth of the beam section in mm
steel_reinforcement = [(2, 16), (2, 20)]  # Steel reinforcement: [(number of bars, diameter in mm), ...]
material_grade_concrete = "M20"  # Concrete grade
material_grade_steel = "Fe415"  # Steel grade

# Constants for material properties
concrete_grade_strength = {
    "M15": 15,
    "M20": 20,
    "M25": 25,
    # Add more grades and strengths as needed
}

steel_grade_strength = {
    "Fe415": 415,
    "Fe500": 500,
    # Add more grades and strengths as needed
}

# Calculate the area of steel reinforcement
steel_area = sum([(n * (diameter/10)**2 * 0.25 * 3.1416) for n, diameter in steel_reinforcement])  # Convert di

# Calculate the lever arm (d') to the centroid of the tensile reinforcement
d_prime = d - (0.5 * 20)  # Assuming the bars are symmetrically placed

# Determine the yield strength of the steel
steel_yield_strength = steel_grade_strength[material_grade_steel]

# Calculate the lever arm (d) to the centroid of the compressive concrete block
d = d_prime - (0.5 * 16)  # Assuming the bars are symmetrically placed

# Calculate the ultimate moment carrying capacity (Mu) in N-mm
Mu = (steel_area * steel_yield_strength * (d - 0.416 * d_prime)) / 1000  # Convert to N-m

# Print the result
print(f"Ultimate Moment Carrying Capacity (Mu): {Mu:.2f} N-m")
```

```
Ultimate Moment Carrying Capacity (Mu): 939.77 N-m
```

In [5]:

```python
# Given data
polygon_areas_cm2 = [25, 30, 30, 10, 5]  # Theissen polygon areas in cm^2
rainfall_cm = [125, 175, 225, 275, 325]  # Annual rainfall in cm
scale = 50000  # Scale of the map (1:x)

# Initialize variables for summation
total_volume_cm3 = 0

# Calculate the volume of rainfall for each station's Theissen polygon
for i in range(len(polygon_areas_cm2)):
    polygon_area_cm2 = polygon_areas_cm2[i]
    annual_rainfall_cm = rainfall_cm[i]

    # Convert polygon area to m^2 and rainfall to m
    polygon_area_m2 = polygon_area_cm2 / 10000  # 1 cm^2 = 0.0001 m^2
    annual_rainfall_m = annual_rainfall_cm / 100  # 1 cm = 0.01 m

    # Calculate the volume of rainfall for the polygon (V = A * P)
    volume_m3 = polygon_area_m2 * annual_rainfall_m

    # Convert volume to cm^3 (1 m^3 = 1,000,000 cm^3)
    volume_cm3 = volume_m3 * 1000000

    # Add the volume to the total volume
    total_volume_cm3 += volume_cm3

# Calculate the total area of the catchment
total_catchment_area_cm2 = sum(polygon_areas_cm2)

# Calculate the mean depth of rainfall (D = V / A)
mean_depth_cm = total_volume_cm3 / total_catchment_area_cm2

# Print the results
print(f"Total Volume of Rainfall: {total_volume_cm3:.2f} cm^3")
print(f"Mean Depth of Rainfall: {mean_depth_cm:.2f} cm")
```

```
Total Volume of Rainfall: 19500.00 cm^3
Mean Depth of Rainfall: 195.00 cm
```

In [20]:

```python
# Given data
bod_3_days_20_degC = 50  # BOD at 3 days at 20°C in mg/L
decay_coefficient = 0.23  # Decay coefficient

# Calculate the BOD at 7 days at 25°C using the BOD rate equation
# BOD7 = BOD3 * e^(-k * (t7 - t3))
# Where:
# BOD7 = BOD at 7 days
# BOD3 = BOD at 3 days
# k = Decay coefficient
# t7 = Time at 7 days (converted to hours)
# t3 = Time at 3 days (converted to hours)

t3_hours = 3 * 24  # Time at 3 days converted to hours
t7_hours = 7 * 24  # Time at 7 days converted to hours

bod_7_days_25_degC = bod_3_days_20_degC * (2.71828 ** (-decay_coefficient * (t7_hours - t3_hours)))

# Print the result
print(f"BOD at 7 days at 25°C: {bod_7_days_25_degC:.2f} mg/L")
```

```
BOD at 7 days at 25°C: 0.00 mg/L
```

In [11]:

```python
# Given data
grade_m40_max_wc_ratio = 0.45
desired_workability_mm = 100
min_cement_content_kg_per_m3 = 320
max_cement_content_kg_per_m3 = 450
specific_gravity_cement = 3.15
specific_gravity_fine_aggregates = 2.74
specific_gravity_coarse_aggregates = 2.74

# Calculate the water-cement ratio (W/C ratio)
wc_ratio = grade_m40_max_wc_ratio

# Calculate the water content for the desired workability
water_content_mm = desired_workability_mm
water_content_kg_per_m3 = water_content_mm * 2  # Assume 2 kg/m³ per mm of water requirement

# Calculate the cement content based on the water content
cement_content_kg_per_m3 = min_cement_content_kg_per_m3 + water_content_kg_per_m3

# Calculate the fine aggregate content based on the desired workability and cement content
fine_aggregate_content_kg_per_m3 = cement_content_kg_per_m3 / ((1 / wc_ratio) - 1)  # Assuming fine aggregates

# Calculate the coarse aggregate content based on the specified maximum size of aggregates
max_size_of_aggregates_mm = 20
coarse_aggregate_content_kg_per_m3 = 1000 - (cement_content_kg_per_m3 + fine_aggregate_content_kg_per_m3)

# Check if the grading of fine aggregates and coarse aggregates conforms to Zone-I
# You may need to provide the grading curve data to perform this check.

# Print the results
print("Concrete Mix Design Results:")
print(f"Water-Cement Ratio (W/C ratio): {wc_ratio:.2f}")
print(f"Water Content for Workability ({desired_workability_mm} mm): {water_content_kg_per_m3:.2f} kg/m³")
print(f"Cement Content (Minimum): {min_cement_content_kg_per_m3} kg/m³")
print(f"Cement Content (Calculated): {cement_content_kg_per_m3:.2f} kg/m³")
print(f"Fine Aggregate Content: {fine_aggregate_content_kg_per_m3:.2f} kg/m³")
print(f"Coarse Aggregate Content: {coarse_aggregate_content_kg_per_m3:.2f} kg/m³")
```

```
Concrete Mix Design Results:
Water-Cement Ratio (W/C ratio): 0.45
Water Content for Workability (100 mm): 200.00 kg/m³
Cement Content (Minimum): 320 kg/m³
Cement Content (Calculated): 520.00 kg/m³
Fine Aggregate Content: 425.45 kg/m³
Coarse Aggregate Content: 54.55 kg/m³
```

In [7]:

```python
# Given data
ewl_constant_values = [330, 1070, 2460, 4620]  # EWL Constant in thousands
aadt_values = [3750, 470, 320, 120]  # Annual Average Daily Traffic (AADT)
traffic_increase_percentage = 60  # 60% increase in traffic over 10 years
years = 10  # Number of years
R = 48  # Repetitions
C = 16  # Cover factor

# Calculate EWL and TI values for 10 years
ewl_values = []
ti_values = []

for i in range(len(ewl_constant_values)):
    ewl_constant = ewl_constant_values[i]
    aadt = aadt_values[i]

    # Calculate the equivalent wheel load (EWL) for the current year
    ewl = ewl_constant * (aadt / 1000)  # Convert AADT to thousands

    # Calculate the traffic index (TI) for the current year
    ti = ewl / (R * (C ** 4))

    ewl_values.append(ewl)
    ti_values.append(ti)

    # Update AADT for the next year with the traffic increase
    aadt = aadt + (aadt * (traffic_increase_percentage / 100))

# Calculate the total EWL and TI values for 10 years
total_ewl = sum(ewl_values)
total_ti = sum(ti_values)

# Calculate the required pavement thickness using the total TI value
required_thickness = (total_ti / (C ** 4)) ** (1/4)

# Print the results
print("Year\tEWL (kN)\tTraffic Index")
for i in range(len(ewl_values)):
    print(f"{i + 1}\t{ewl_values[i]}\t\t{ti_values[i]:.4f}")

print(f"\nTotal EWL for 10 years: {total_ewl:.2f} kN")
print(f"Total Traffic Index for 10 years: {total_ti:.4f}")
print(f"Required Pavement Thickness: {required_thickness:.2f} mm")
```

```
Year    EWL (kN)        Traffic Index
1       1237.5          0.0004
2       502.9           0.0002
3       787.2           0.0003
4       554.4           0.0002

Total EWL for 10 years: 3082.00 kN
Total Traffic Index for 10 years: 0.0010
Required Pavement Thickness: 0.01 mm
```

In [21]:

```python
# Given data
tensile_force = 225000  # Tensile force in Newtons (225 kN)
safety_factor = 1.5  # Safety factor as per IS 800

# Assume properties of M20 bolts (you may need to look up the actual values)
bolt_diameter = 20  # Bolt diameter in millimeters (M20)
bolt_yield_strength = 240  # Bolt yield strength in MPa (M20)
bolt_safety_factor = 1.5  # Safety factor for bolts

# Calculate the tensile capacity of the M20 bolts
bolt_area = (3.1416 / 4) * (bolt_diameter / 10) ** 2  # Bolt cross-sectional area in square centimeters
bolt_tensile_capacity = bolt_area * bolt_yield_strength * bolt_safety_factor * 1000  # Convert MPa to N/mm²

# Calculate the required tensile capacity of the angle section
required_tensile_capacity = tensile_force / safety_factor

# Calculate the minimum angle section size needed
angle_size_required = required_tensile_capacity / bolt_tensile_capacity  # Angle size in square centimeters

# Print the results
print(f"Required Tensile Capacity: {required_tensile_capacity} N")
print(f"Bolt Tensile Capacity (M20): {bolt_tensile_capacity} N")
print(f"Minimum Angle Section Size Required: {angle_size_required:.2f} cm²")
```

```
Required Tensile Capacity: 150000.0 N
Bolt Tensile Capacity (M20): 1130976.0 N
Minimum Angle Section Size Required: 0.13 cm²
```

In [12]:

```python
import math

# Given data
effective_span = 3.0  # Effective span of the slab in meters
support_width = 0.23  # Support width in meters (230 mm)
live_load = 4.0  # Live load in kN/m
floor_finish_load = 1.8  # Floor finish load in kN/m
clear_cover = 20  # Clear cover in mm
main_reinforcement_dia = 12  # Diameter of main reinforcement bars in mm
distribution_reinforcement_dia = 8  # Diameter of distribution reinforcement bars in mm
concrete_grade = "M20"  # Concrete grade
steel_grade = "Fe415"  # Steel grade
self_weight_concrete = 24  # Self-weight of concrete in kN/m^3

# Constants for material properties
concrete_grade_strength = {
    "M15": 15,
    "M20": 20,
    "M25": 25,
    # Add more grades and strengths as needed
}

steel_grade_strength = {
    "Fe415": 415,
    "Fe500": 500,
    # Add more grades and strengths as needed
}

# Calculate the design moment (Mu) due to live load and floor finish
total_load = live_load + floor_finish_load  # Total applied load in kN/m
Mu = (total_load * effective_span**2) / 8  # Ultimate moment in kN-m

# Calculate the effective depth of the slab (d)
concrete_strength = concrete_grade_strength[concrete_grade]
steel_strength = steel_grade_strength[steel_grade]
d = math.sqrt((Mu * 1000) / (0.138 * concrete_strength))  # Effective depth in mm

# Check for the minimum and maximum steel percentages
min_steel_percentage = 0.12  # Minimum steel percentage for M20 concrete
max_steel_percentage = 0.15  # Maximum steel percentage for M20 concrete

# Calculate the required main reinforcement area (As_main)
As_main = (Mu * 10**6) / (0.87 * steel_strength * d)  # Required main reinforcement area in mm^2
As_main = max(As_main, min_steel_percentage * 1000 * d * effective_span)  # Ensure it meets the minimum steel r
As_main = min(As_main, max_steel_percentage * 1000 * d * effective_span)  # Ensure it doesn't exceed the maximu

# Calculate the spacing of main reinforcement bars
main_reinforcement_spacing = (math.pi * (main_reinforcement_dia / 10)**2) / (4 * As_main)  # Spacing in mm

# Calculate the required distribution reinforcement area (As_dist)
As_dist = 0.12 / 100 * effective_span * d  # Required distribution reinforcement area in mm^2

# Calculate the spacing of distribution reinforcement bars
distribution_reinforcement_spacing = (math.pi * (distribution_reinforcement_dia / 10)**2) / (4 * As_dist)  # Sp

# Print the results
print("Design Results:")
print(f"Effective Depth (d): {d:.2f} mm")
print(f"Required Main Reinforcement Area (As_main): {As_main:.2f} mm^2")
print(f"Main Reinforcement Spacing: {main_reinforcement_spacing:.2f} mm (center-to-center)")
print(f"Required Distribution Reinforcement Area (As_dist): {As_dist:.2f} mm^2")
print(f"Distribution Reinforcement Spacing: {distribution_reinforcement_spacing:.2f} mm (center-to-center)")
```

```
Design Results:
Effective Depth (d): 48.62 mm
Required Main Reinforcement Area (As_main): 17504.04 mm^2
Main Reinforcement Spacing: 0.00 mm (center-to-center)
Required Distribution Reinforcement Area (As_dist): 0.18 mm^2
Distribution Reinforcement Spacing: 2.87 mm (center-to-center)
```

In [19]:

```python
# Given data
initial_mass_sludge = 100  # Initial mass of sludge in kg
initial_specific_gravity = 2.2  # Specific gravity of sludge solids
sludge_volume_reduction_factor = 0.5  # Sludge volume reduction factor
density_water = 1000  # Density of water in kg/m³

# Calculate the mass of solids in the initial sludge
mass_solids_initial = initial_mass_sludge * (initial_specific_gravity / (1 + (initial_specific_gravity - 1) * 0

# Calculate the final mass of sludge after thickening
final_mass_sludge = initial_mass_sludge * sludge_volume_reduction_factor

# Calculate the density of the sludge removed from the aeration tank
density_sludge_removed = final_mass_sludge / (initial_mass_sludge - final_mass_sludge)

# Print the result
print(f"Density of Sludge Removed from Aeration: {density_sludge_removed:.2f} kg/m³")
```

```
Density of Sludge Removed from Aeration: 1.00 kg/m³
```

In [14]:

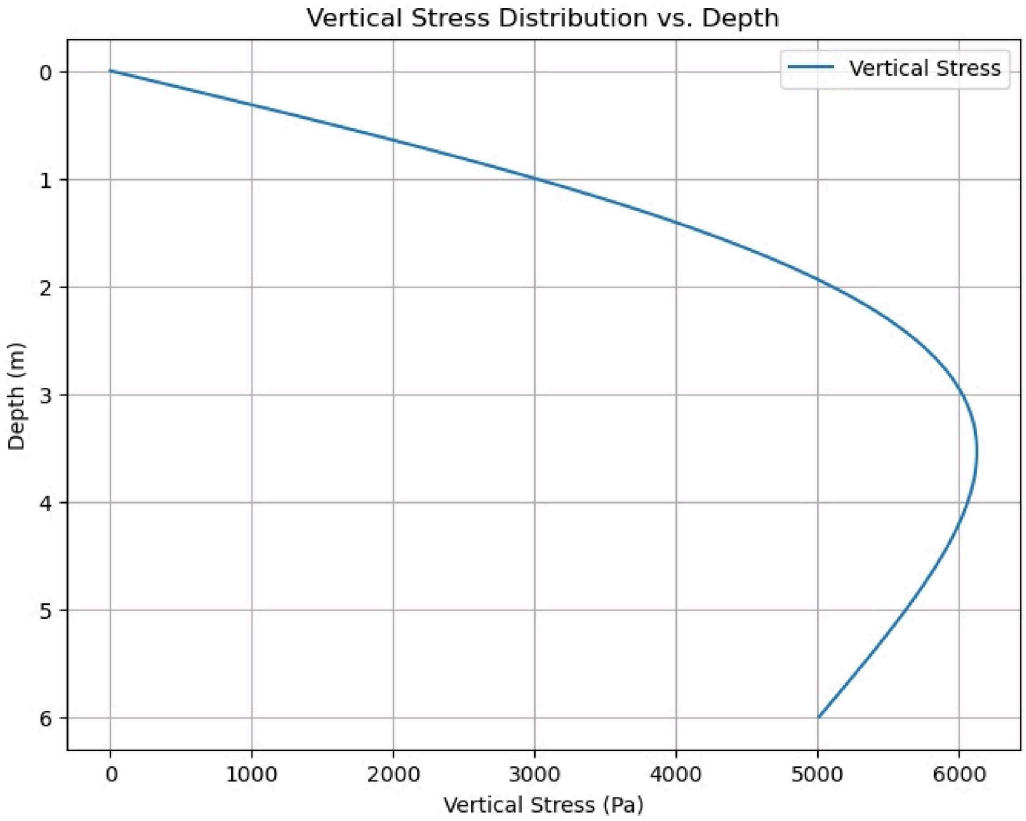
```
import numpy as np
import matplotlib.pyplot as plt

# Given data
load_force = 2500 * 1000  # Load force in N (2500 kN)
horizontal_distance = 5  # Horizontal distance from the load axis in meters
depths = np.linspace(0, 6, 100)  # Depths ranging from 0 to 6 meters
constant = 1 / (2 * np.pi)

# Initialize lists to store stress values at different depths
vertical_stresses = []

# Calculate vertical stress at different depths
for depth in depths:
    r = np.sqrt(horizontal_distance ** 2 + depth ** 2)
    vertical_stress = (load_force * constant * depth) / r ** 3
    vertical_stresses.append(vertical_stress)

# Plot the results
plt.figure(figsize=(8, 6))
plt.plot(vertical_stresses, depths, label="Vertical Stress")
plt.xlabel("Vertical Stress (Pa)")
plt.ylabel("Depth (m)")
plt.title("Vertical Stress Distribution vs. Depth")
plt.grid(True)
plt.legend()
plt.gca().invert_yaxis()
plt.show()
```



In [16]:

```python
import math


def calculate_bearing_capacity(water_table_position, angle_of_internal_friction, bulk_density_soil, saturated_d
  """
  Calculates the ultimate bearing capacity of a square footing.

  Args:
    water_table_position: The water table position.
    angle_of_internal_friction: The angle of internal friction of the soil.
    bulk_density_soil: The bulk density of the soil.
    saturated_density_soil: The saturated density of the soil.
    unit_weight_water: The unit weight of water.

  Returns:
    The ultimate bearing capacity.
  """

  if water_table_position == "Ground Surface":
    depth_to_water_table = 0
  elif water_table_position == "Footing Level":
    depth_to_water_table = 2
  else:
    depth_to_water_table = 1

  effective_depth = 2 - depth_to_water_table

  # Calculate the bearing capacity using the approximate formulas
  bearing_capacity_approx = (
      0.5 * bulk_density_soil * effective_depth ** 2 * math.tan(angle_of_internal_friction)
  )

  # Calculate the bearing capacity using the conventional method
  bearing_capacity_conventional = (
      0.5 * saturated_density_soil * effective_depth ** 2 * math.tan(angle_of_internal_friction)
  )

  return bearing_capacity_approx, bearing_capacity_conventional


def main():
  # Input data
  angle_of_internal_friction = 35  # degrees
  bulk_density_soil = 18  # kN/m³
  saturated_density_soil = 20  # kN/m³
  unit_weight_water = 10  # kN/m³

  # Calculate the bearing capacity for each water table position
  water_table_positions = ["Ground Surface", "Footing Level", "1 m below Footing"]
  for water_table_position in water_table_positions:
    bearing_capacity_approx, bearing_capacity_conventional = calculate_bearing_capacity(
      water_table_position, angle_of_internal_friction, bulk_density_soil, saturated_density_soil, unit_weight_
    )

    print(f"Water Table Position: {water_table_position}")
    print(f"Bearing Capacity (Approximate): {bearing_capacity_approx:.2f} kN/m²")
    print(f"Bearing Capacity (Conventional): {bearing_capacity_conventional:.2f} kN/m²")


if __name__ == "__main__":
  main()
```

```
Water Table Position: Ground Surface
Bearing Capacity (Approximate): 17.06 kN/m²
Bearing Capacity (Conventional): 18.95 kN/m²
Water Table Position: Footing Level
Bearing Capacity (Approximate): 0.00 kN/m²
Bearing Capacity (Conventional): 0.00 kN/m²
Water Table Position: 1 m below Footing
Bearing Capacity (Approximate): 4.26 kN/m²
Bearing Capacity (Conventional): 4.74 kN/m²
```

In [2]:

```python
import math

# Given data
discharge = 100  # Discharge in m^3/s
roughness_coefficient = 0.015
channel_slope = 1 / 2500  # Channel bed slope (S)
freeboard_percentage = 0.10  # 10% of depth as freeboard

# Calculate the hydraulic radius using Manning's equation
# Q = (1/n) * A * R^(2/3) * S^(1/2)
# Where Q is the discharge, n is the Manning's roughness coefficient, A is the cross-sectional area, R is the h

# We can rearrange the equation to solve for the hydraulic radius R:
# R = (Q / (n * A * S^(1/2)))^(3/2)

# To design an efficient trapezoidal channel, we'll assume a particular side slope (Z) and calculate the dimens

# Calculate the width (b) based on a chosen side slope (Z)
Z = 2  # Assume a 2:1 side slope
b = 1.0  # Initial guess for channel bottom width
R = 1.0  # Initial guess for hydraulic radius

# Iterate to find the correct width (b) and hydraulic radius (R)
max_iterations = 100
tolerance = 1e-6  # Tolerance for convergence

for i in range(max_iterations):
    A = (b + Z * R) * R  # Cross-sectional area
    hyd_radius_calc = (discharge / (roughness_coefficient * A * (channel_slope**0.5)))**(2/3)

    # Check for convergence
    if abs(hyd_radius_calc - R) < tolerance:
        break

    R = hyd_radius_calc

    # Recalculate width (b) based on the hydraulic radius
    b = A / (R + Z * R)

# Calculate the depth of flow (y)
y = R / Z

# Calculate the Froude number (Fr)
g = 9.81  # Acceleration due to gravity in m/s^2
velocity = discharge / A
Fr = velocity / (math.sqrt(g * y))

# Calculate the freeboard
freeboard = freeboard_percentage * y

# Print the results
print(f"Design Parameters:")
print(f"Bottom Width (b): {b:.2f} meters")
print(f"Depth of Flow (y): {y:.2f} meters")
print(f"Hydraulic Radius (R): {R:.2f} meters")
print(f"Freeboard: {freeboard:.2f} meters")
print(f"Froude Number (Fr): {Fr:.4f}")

# Check if the computed Fr is greater than 1 (supercritical flow)
if Fr > 1:
    print("The flow is supercritical.")
else:
    print("The flow is subcritical.")
```

```
Design Parameters:
Bottom Width (b): 27.65 meters
Depth of Flow (y): 13.82 meters
Hydraulic Radius (R): 27.65 meters
Freeboard: 1.38 meters
Froude Number (Fr): 0.0037
The flow is subcritical.
```

In [ ]:

In [9]:

```python
# Given data
design_speed_kmph = 65  # Design speed in km/h
radius_of_curvature_m = 220  # Radius of curvature in meters
allowable_rate_of_superelevation = 1 / 150  # Allowable rate of introduction of superelevation (1 in X)
pavement_width_m = 7.5  # Pavement width including extra widening in meters
g = 9.81  # Acceleration due to gravity in m/s²

# Convert design speed to m/s
design_speed_mps = design_speed_kmph * 1000 / 3600  # 1 km/h = 1000 m/3600 s

# Calculate the length of the transition curve
length_of_transition_curve = (design_speed_mps ** 2) / (g * allowable_rate_of_superelevation)

# Print the result
print(f"Length of the transition curve: {length_of_transition_curve:.2f} meters")
```

```
Length of the transition curve: 4984.76 meters
```

In [1]:

```python
# Given data
width_initial = 3.5  # Initial width of the channel in meters
width_constricted = 2.5  # Constricted width of the channel in meters
discharge = 15  # Discharge in m^3/s
depth_initial = 2  # Initial depth of flow in meters

# Calculate the cross-sectional area of the channel before and after the constriction
area_initial = width_initial * depth_initial
area_constricted = width_constricted * depth_initial

# Calculate the velocity of flow before and after the constriction using the principle of continuity
velocity_initial = discharge / area_initial
velocity_constricted = discharge / area_constricted

# Calculate the water surface elevation at the upstream and downstream of the constriction
# Upstream elevation remains the same
upstream_elevation = depth_initial

# Downstream elevation can be calculated using the specific energy equation
# E = y + (v^2) / (2 * g), where E is the specific energy, y is the depth of flow, v is the velocity, and g is
g = 9.81  # Acceleration due to gravity in m/s^2
downstream_elevation = depth_initial + (velocity_initial**2) / (2 * g) - (velocity_constricted**2) / (2 * g)

# Print the results
print(f"Upstream water surface elevation: {upstream_elevation:.2f} meters")
print(f"Downstream water surface elevation: {downstream_elevation:.2f} meters")
```

```
Upstream water surface elevation: 2.00 meters
Downstream water surface elevation: 1.78 meters
```

In [ ]:

In [15]:

```python
# Given data
pile_side_length = 0.45  # Side length of the square pile in meters (450 mm)
pile_length = 15.0  # Length of the pile in meters (15 m)
average_ucs = 75.0  # Average UCS (Unconfined Compressive Strength) of the clay in kN/m² (75 kN/m²)
ca = 0.8  # Pile-soil interaction coefficient

# Calculate the area of the pile base
pile_base_area = pile_side_length ** 2  # Square pile base area in square meters

# Calculate the ultimate load capacity of the pile
ultimate_load_capacity = average_ucs * pile_base_area * pile_length * ca

# Print the result
print(f"Ultimate Load Capacity of the Pile: {ultimate_load_capacity:.2f} kN")
```

```
Ultimate Load Capacity of the Pile: 182.25 kN
```

In [6]:

```python
import math
# Given data
initial_infiltration_capacity = 6  # Initial infiltration capacity in cm/hr
final_infiltration_capacity = 1.2  # Final infiltration capacity in cm/hr
decay_coefficient = 0.888  # Decay coefficient in 1/hr
storm_duration_hours = 8  # Duration of the storm event in hours

# Calculate the total infiltration using Horton's equation
total_infiltration = 0
time_elapsed = 0

while time_elapsed < storm_duration_hours:
    # Calculate the infiltration rate at the current time
    current_infiltration_capacity = initial_infiltration_capacity + (final_infiltration_capacity - initial_infi
    # Calculate the time step for the current iteration (assuming small time steps)
    delta_time = 0.01  # You can adjust this for accuracy

    # Calculate the infiltration during this time step using the average infiltration rate
    infiltration_during_step = current_infiltration_capacity * delta_time

    # Add the infiltration during this time step to the total infiltration
    total_infiltration += infiltration_during_step

    # Update the time elapsed
    time_elapsed += delta_time

# Convert the total infiltration from cm to mm
total_infiltration_mm = total_infiltration * 10  # 1 cm = 10 mm

# Print the total infiltration
print(f"Total infiltration during the 8-hour storm event: {total_infiltration_mm:.2f} mm")
```

```
Total infiltration during the 8-hour storm event: 150.37 mm
```

In [8]:
```python
# Given data
compacted_soil_cbr = 6  # CBR for compacted soil (%)
poorly_graded_gravels_cbr = 12  # CBR for poorly graded gravels (%)
well_graded_gravel_cbr = 60  # CBR for well-graded gravel (%)
bituminous_surface_thickness_cm = 4  # Thickness of bituminous surface (cm)
wheel_load_kg = 4085  # Wheel load in kg
tyre_pressure_kg_per_cm2 = 7  # Tyre pressure in kg/cm^2

# Load and Penetration data for the CBR test
load_data = [60, 82]  # Load in kg
penetration_data = [2.5, 5.0]  # Penetration in mm
standard_load_data = [1370, 2055]  # Standard load in kg

# Calculate the CBR values for each test point
cbr_values = []

for i in range(len(load_data)):
    load = load_data[i]
    penetration = penetration_data[i]
    standard_load = standard_load_data[i]

    # Calculate the CBR for the current test point
    cbr = (load / standard_load) * 100

    cbr_values.append(cbr)

# Calculate the average CBR from the test data
average_cbr = sum(cbr_values) / len(cbr_values)

# Calculate the thickness correction factor for the bituminous surface
thickness_correction_factor = 1 + (bituminous_surface_thickness_cm / 10)

# Calculate the effective CBR for the pavement structure
effective_cbr = average_cbr * thickness_correction_factor

# Determine the type of material based on effective CBR
material_type = None
if effective_cbr < compacted_soil_cbr:
    material_type = "Compacted Soil"
elif effective_cbr < poorly_graded_gravels_cbr:
    material_type = "Poorly Graded Gravels"
elif effective_cbr < well_graded_gravel_cbr:
    material_type = "Well-graded Gravel"
else:
    material_type = "Unknown (High CBR)"

# Print the results
print(f"Average CBR from test data: {average_cbr:.2f}%")
print(f"Thickness Correction Factor: {thickness_correction_factor:.2f}")
print(f"Effective CBR for pavement structure: {effective_cbr:.2f}%")
print(f"Type of material based on effective CBR: {material_type}")
```

```
Average CBR from test data: 4.18%
Thickness Correction Factor: 1.40
Effective CBR for pavement structure: 5.86%
Type of material based on effective CBR: Compacted Soil
```