Department of Mechanical Engineering, IIT Kharagpur
Data analysis and visualization for mechanical sciences (ME40229)

23-09-2024 **Midsemester Examination** **FN** **60 Marks**

Name of student:

- **This question paper contains 6 questions.**
- **Attempt question 1 and any 4 out of the remaining questions.**
- **Hints are provided in the appendix to this question paper.**
- **This question paper contains a total of 6 pages.**
- **DO NOT scribble anything on the question paper.**
- **Mention all assumptions explicitly, if any.**

Q1. Give an explanation of the output produced by the code snippets below.

(a) Code:

```
import numpy as np
arr = np.arange(12)
reshaped_arr = arr.reshape((3, 4))
print(reshaped_arr)
```

(b) Code:

```
array = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
mask = (array > 5) & (array % 2 == 0)
filtered_array = array[mask]
print(filtered_array)
```

(c) Code:

```
matrix = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12],[13, 14, 15, 16]])
diagonal_matrix = np.diag(np.diag(matrix))
print("Diagonal Matrix:\n", diagonal_matrix)
```

(d) Code:

```
matrix = np.array([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
rows = np.array([0, 1, 2])
cols = np.array([2, 1, 0])
selected_elements = matrix[rows, cols]
print("Selected Elements:", selected_elements)
```

(e) Code:

```
import numpy as np
matrix = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]])
submatrix = matrix[-2:, -2:]
submatrix_sum = np.sum(submatrix)
submatrix_mean = np.mean(submatrix)
print("Submatrix:\n", submatrix)
print("Sum of Submatrix:", submatrix_sum)
print("Mean of Submatrix:", submatrix_mean)
```

Q2(a). Solve a system of linear equations Ax = b using the Conjugate Gradient Method (**Show 3 iterations by hand**). The matrix A is symmetric and positive definite, which makes the conjugate gradient method applicable.

$$A = \begin{bmatrix} 4 & 1 & 1 \\ 1 & 3 & -1 \\ 1 & -1 & 2 \end{bmatrix} \text{ and } b = \begin{bmatrix} 1 \\ 2 \\ 0 \end{bmatrix}$$

The code for solving a system of linear equations using the conjugate gradient method with numpy has been provided below. **Complete the code** by implementing the iteration loop and therefore, returning the solution. Then, use the `conjugate_gradient` function to find the solution vector x. The incomplete code is given below:

```
def conjugate_gradient(A, b, x0=None, tol=1e-10, max_iter=None):
    n = len(b)
    if x0 is None:
        x0 = np.zeros(n)
    if max_iter is None:
        max_iter = n

    x = x0
    r = b - np.dot(A, x)  # Initial residual
    p = r.copy()
    rs_old = np.dot(r, r)

    for i in range(max_iter):
      <Your code goes here>
```

HINTS: Steps for CG method:
1. Choose an initial guess x_0 for the solution vector x.
2. Implement the Conjugate Gradient Algorithm:
 Initialize the residual $r_0 = b - Ax_0$ and the direction vector $p_0 = r_0$. (syntax: p0 = r0.copy())

 For each iteration:

 1. Compute the step size $\alpha_k = (r_k^T r_k) / (p_k^T A p_k)$. (for dot product syntax: np.dot())
 2. Update the solution vector $x_{k+1} = x_k + \alpha_k p_k$.
 3. Update the residual $r_{k+1} = r_k - \alpha_k A p_k$.
 4. Check for convergence: If $||r_{k+1}||$ is less than a specified tolerance stop the iteration.
 5. Update the direction vector $p_{k+1} = r_{k+1} + \beta_k p_k$, where $\beta_k = (r_{k+1}^T r_{k+1}) / (r_k^T r_k)$.
 6. Return the solution vector x.

Q2(b) LU decomposition is essential in engineering for efficiently solving linear systems, calculating matrix inverses, and determining determinants. It simplifies complex systems by breaking them into lower and upper triangular matrices, allowing faster and more stable computations, especially in structural analysis and simulations.

$A = \begin{bmatrix} \mathbf{2} & \mathbf{-1} & \mathbf{1} \\ \mathbf{-3} & \mathbf{3} & \mathbf{9} \\ \mathbf{3} & \mathbf{3} & \mathbf{5} \end{bmatrix}$ and $b = \begin{bmatrix} \mathbf{2} \\ \mathbf{15} \\ \mathbf{5} \end{bmatrix}$. As Ax = b **write a code** for finding the solution vector x using LU decomposition method.

Example solution of LU decomposition is given for sanity check of code and algorithm.

$P = \begin{bmatrix} 2 & 3 & 1 \\ 4 & 7 & 2 \\ 6 & 18 & 5 \end{bmatrix}$; P = LU then the $L = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 3 & 9 & 1 \end{bmatrix}$ and $U = \begin{bmatrix} 2 & 3 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 2 \end{bmatrix}$

Algorithm for Solving Ax = b Using LU Decomposition

1. Input:
 - A square matrix A of size n×n.
 - A vector b of size n .
2. Step 1: LU Decomposition

- Decompose matrix A (A = LU) into a lower triangular matrix L (with ones on the diagonal) and an upper triangular matrix U.

The decomposition is done by:

- Initializing L as an identity matrix.
- Initializing U as a zero matrix.

-Loop over columns of matrix A:
- For each column k (1 to n):
1. Upper triangular matrix U:
   - For each row i from k to n:
     - Compute the elements kth row of matrix U:
       $U[k,i] = A[k,i] - \sum_{j=0}^{k-1} L[k,j]U[j,i]$

2. Lower triangular matrix L:
   - For each row i from k+1 to n:
   - Compute the elements k-th column of matrix L : $L[i,k] = \frac{A[i,k]-\sum_{j=0}^{k-1} L[i,j]U[j,k]}{U[k,k]}$
   - set L[k,k] = 1 (diagonal element of L are 1)

3. Step 2: solve L × y = b using forward substitution.

Use the forward substitution method where: $y[i] = b[i] - \sum_{j=0}^{i-1} L[i,j] \times y[j]$

4. Step 3: Solve U× $x = y$ using backward Substitution.

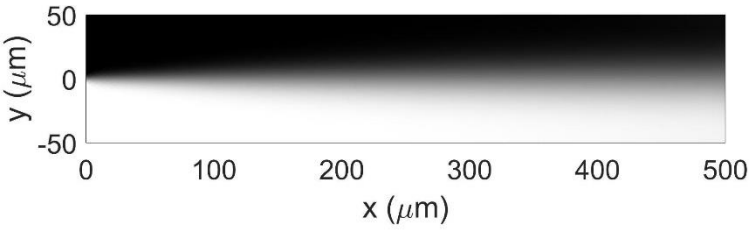
Compute the solution vector x by solving the system U × x = y

Use the backward substitution method where $x[i] = \frac{y[i]-\sum_{j=i+1}^{n-1} U[i,j] \times x[j]}{U[i,i]}$

Output: The solution vector x to the system Ax = b.

5. Output: The solution vector x to the system Ax = b

Q3. The given figure shows the mixing of two dyes – red (shown in black) and blue (shown in white) in a microchannel; inlet is at the left wall and the right wall is the exit. The length and width of the microchannel are 500 μm and 100 μm respectively. The dyes enter from the inlets (x = 0 μm) and their mixture is expelled from the outlet (x = 500 μm). The normalized dye concentration denoted by 'c', ranges from 0 to 1, with c^* = 0.5 denoting homogeneous mixing.



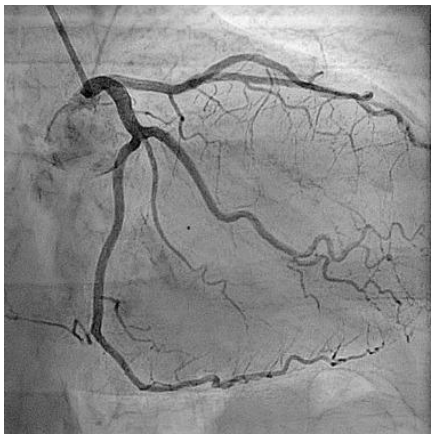
The mixing index (η) at the channel outlet (or any section) is given by the following expression:

$$\eta = 1 - \sqrt{\frac{\sum_N (c - c^*)^2}{\sum_N c^{*2}}}$$

where 'c' is the concentration value at a pixel point, N is the total number of pixels along the width of any section under investigation and $\sum_N$ presents the summation over N pixels.

Write a program to find the mixing index (η) at - (a) mid-section (x = 250 μm) and (b) microchannel outlet (x = 500 μm) using image processing of the image provided. Note that the figure given here may or may not be in grayscale format and appropriate conversions may be necessary. Also, the axes given in the figure are your reference only and are not part of the image to be processed.

Q4. A coronary angiogram is a diagnostic procedure that uses X-rays to examine blood vessels – arteries and veins and the blood flow. In this process, a special contrast dye is injected into the blood vessels, which shows up in the X-ray. The given figure is a coronary angiogram, which shows the arterial network of the heart. A high contrast angiogram is very important for the examination of the coronary arterial network and subsequent calculation of arterial volume and blood flow rate.



Perform a thresholding operation on the given image to make the arteries more prominent. Thereafter, calculate the total projected area of the arterial network. Assume circular cross section for the arteries, with a mean area 'A' (choose an appropriate value for A in your code) and calculate the total volume of the arterial network. Explain the flowchart first and then write down the program. Note that the figure given may or may not be in grayscale format and appropriate conversion may be necessary. The image is of size 374 x 374 (in pixels). Take 1 pixel = 0.14 mm to evaluate the area of the coronary network.

Q5. The equation y'' + 2εy' - y = 0 originates from the 1D damped harmonic oscillator model. Starting with the basic harmonic oscillator equation $\frac{d^2y}{dt^2} + ky = 0$, which describes the balance between restoring and inertial forces, we introduce damping as $-c\frac{dy}{dt}$, leading to $m\frac{d^2y}{dt^2} + c\frac{dy}{dt} + ky = 0$. Normalizing time by defining $\tau = \omega t$, where $\omega = \sqrt{\frac{k}{m}}$, simplifies the equation to $m\frac{d^2y}{d\tau^2} + 2\varepsilon\frac{dy}{d\tau} + y = 0$ with $\varepsilon = \frac{c}{2m\omega}$. The given equation y'' + 2εy' - y = 0 resembles this form but with a negative sign on the y-term, which might indicate a system with an inverted potential or instability, rather than simple oscillation. Boundary conditions y(0) = 0 and y(1) = 1 specify constraints at fixed points.

Complete the code given below for solving the differential equation: y"+ 2εy'- y = 0, y(0) = 0 and y(1) = 1 using the regular perturbation method. Plot the exact solution and solution from regular perturbation. Give the solution steps as comments.

Code:

```
import sympy as sp

# Define symbols
tau, epsilon = sp.symbols('tau epsilon')
y = sp.Function('y')(tau)

# Define the differential equation
diffeq = y.diff(tau, tau) + 2 *epsilon* y.diff(tau) - y

# Assume the perturbation expansion for y(tau)
y0 = sp.Function('y0')(tau)
y1 = sp.Function('y1')(tau)
y_perturbation = y0 + epsilon * y1

# Substitute the perturbation expansion into the original differential equation and expand
```

```
diffeq_sub_expanded = sp.simplify(diffeq.subs(y, y_perturbation).expand())
```

<Your code goes here>

Q6. The Sobel filter is used for edge detection in images. It works by detecting changes in intensity in both horizontal and vertical directions and works by finding out gradients in x and y directions. To achieve this, there are two kernels as mentioned below:

Horizontal Gradient Kernel G_x: Detects vertical edges by highlighting horizontal changes.
Vertical Gradient Kernel G_y: Detects horizontal edges by highlighting vertical changes.

Horizontal Gradient Kernel $G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$ Vertical Gradient Kernel $G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$

Magnitude of Gradient G = $\sqrt{{G_x}^2 + {G_y}^2}$. This gives the strength of the edge. Direction of Gradient θ: $\theta = \tan^{-1}\left(\frac{G_y}{G_x}\right)$. This gives the angle of the edge.

Convolution Process: Convolution is the process of applying a kernel to an image to extract features. For each pixel, the kernel is applied to a 3×3 neighborhood around the pixel, and the result is computed by summing the product of the kernel values and the corresponding image pixel values.

Create your own Sobel filter for an image of arbitrary size and subplot the grayscale image of image, G_x, G_y and edge magnitude. Detailed step by step implementation for a sample 5x5 grayscale image is given below for your perusal:

Step 1: Consider a simple grayscale image represented as a 5×5 matrix: Image = $\begin{bmatrix} 200 & 200 & 200 & 200 & 200 \\ 200 & 100 & 100 & 100 & 200 \\ 200 & 100 & 50 & 100 & 200 \\ 200 & 100 & 100 & 100 & 200 \\ 200 & 200 & 200 & 200 & 200 \end{bmatrix}$

Step 02: Apply G_x kernel: $G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$

for a pixel at position (2,2) in the image (considering 0-based indexing), the 3×3 neighborhood of the image corresponding to the stencil, i.e. the kernel, is: Neighborhood of kernel = $\begin{bmatrix} 100 & 100 & 100 \\ 100 & 50 & 100 \\ 100 & 100 & 100 \end{bmatrix}$

Perform element-wise multiplication and sum:

G_x = (−1 × 100) + (0 × 100) + (1 × 100) + (−2 × 100) + (0 × 50) + (2 × 100) + (−1 × 100) + (0 × 100) + (1×100) = 0

Syntax: `convolve(image, Gx)`

Step 03: Apply G_y kernel: $G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$ Then perform element-wise multiplication and sum:

G_y = (−1 × 100) + (−2 × 100) + (−1 × 100) + (0 × 50) + (0 × 100) + (0 × 100) + (1 × 100) + (2 × 100) + (1 × 100) = 0

Syntax: `convolve(image, Gy)`

Step 04: Compute the gradient magnitude Magnitude of Gradient G = $\sqrt{{G_x}^2 + {G_y}^2}$

Step 05: Apply the **Sobel kernels to each pixel in the image**. The final matrix should reveal the edge. For border pixels, use padding (e.g., zero-padding).

Step 06: For the display figure use matplotlib

```
plt.subplot(1, 4, 1)
plt.imshow(image, cmap='gray', vmin=0, vmax=255)
plt.title('Original Image')
plt.axis('off')
```

APPENDIX

General numpy functions:

Assigning array to the matrix: `np.array([1, 2, 3]); Array shape: A.shape()`
Array of zeros: `np.zeros((2, 3))`
Identity matrix: `np.eye(3)`
Create an array with evenly spaced values between start and stop: `np.linspace(0, 1, 5)`
Array with random numbers: `np.linspace(0, 1, 5)`
Dot product of vectors: `np.dot(A, B)`
Matrix multiplication: `np.matmul(A, B)`
`Converting a 2D array to 1D array: array.flatten()`

General sympy functions:

For defining symbols: `sp.symbols('x y')`
For creating a function: `sp.Function('y')(x)`
For creating equation: `sp.Eq()`
For solving differential equation: sp.dsolve(diff_eq, ics = bcs)
Expand the equation: `diffeq.subs(y, y_perturbation).expand()`
Collet the coefficient of the equation: `sp.collect(diffeq_sub_expanded, epsilon).coeff(epsilon, 0)`

General functions for opencv:

Syntax for convolution: `convolve(image, $G_x$)`
To show the matrix as image: `plt.imshow(image, cmap='gray', vmin=0, vmax=255)`
`Syntax for reading image: im1 = cv2.imread('image.jpg', cv2.IMREAD_COLOR); cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)`
`Syntax for converting BGR image to RGB: img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)`
`Syntax for converting BGR image to grayscale: img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)`
`Converting image array to NumPy array: img_array = np.array(img)`
`Syntax for getting the shape of the image array: height, width, _ = img.shape`
`Syntax for splitting colour channels: im1_b, im1_g, im1_r = cv2.split(im1)`
`Syntax for generating histogram: hist = cv2.calcHist([image], [0], None, [256], [0, 256])`
`Syntax for slicing the last column of the image array: last_column = img[:, -1, :] (for BGR/RGB); last_column = img[:, - 1] (for GRAYSCALE)`
`Syntax for slicing any other column of the image array: any_column = img[:, column_number-1, :] (for BGR/RGB); last_column = img[:,column_number-1] (for GRAYSCALE)`
`Syntax for binary threshold: _, thresholded_image_array = cv2.threshold (original_image, threshold value, max value, cv2.THRESH_BINARY)`