Week 5: Vectors, Matrices, and Logical Operations 🧮

Objectives:

- Perform vector and matrix operations
- Use built-in functions for matrix analysis
- Understand and use logical vectors and operations

Topics Covered:

1. Vector Operations
2. Matrix Operations
3. Logical Vectors and Operations

1. Vector Operations

What is a Vector?

A **vector** is a one-dimensional array of numbers. In engineering and mathematics, vectors represent quantities that have both magnitude and direction.

Creating Vectors in MATLAB:

```
% Row vector
rowVector = [1, 2, 3, 4];

% Column vector
colVector = [1; 2; 3; 4];
```

💡 **Tip:** Use semicolons `;` to create column vectors.

Basic Vector Operations

Vectors can be manipulated using arithmetic operations. Operations can be element-wise or involve linear algebra concepts.

Element-wise Operations:

```
a = [1, 2, 3, 4];
b = [5, 6, 7, 8];

% Element-wise addition
c = a + b; % c = [6, 8, 10, 12]

% Element-wise multiplication
d = a .* b; % d = [5, 12, 21, 32]
```

💡 **Tip:** The `.*` operator performs element-wise multiplication.

Dot Product

The **dot product** is an algebraic operation that takes two equal-length sequences of numbers and returns a single number.

Mathematically, for vectors *a* and *b*:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i$$

```matlab
% MATLAB code
dotProduct = dot(a, b);
```

👩‍💻 **Exercise:** Compute the dot product of `a = [2, 4, 6]` and `b = [1, 3, 5]`.

Common Vector Functions

Sum and Mean:

```
total = sum(a); % Sum of elements in a
avg = mean(a); % Mean value of elements in a
```

Length:

```
len = length(a); % Number of elements in a
```

Maximum and Minimum:

```
maxValue = max(a); % Largest element in a
minValue = min(a); % Smallest element in a
```

💡 **Tip:** Utilize these functions to quickly analyze data in vectors.

2. Matrix Operations

What is a Matrix?

A **matrix** is a two-dimensional array of numbers arranged in rows and columns. Matrices are fundamental in linear algebra and are used to represent linear transformations.

Creating Matrices in MATLAB:

```
A = [1 2 3; 4 5 6; 7 8 9];
```

💡 **Tip:** Use semicolons `;` to separate rows in a matrix.

Matrix Indexing

Elements in a matrix are accessed using indices:

```
element = A(row, column);
```

Example:

```
element = A(2, 3); % Accesses the element in the 2nd row, 3rd column
```

Result: For matrix `A` above, `element = 6`.

Matrix Addition and Subtraction

Matrices of the **same dimensions** can be added or subtracted element-wise:

```
B = [9 8 7; 6 5 4; 3 2 1];
C = A + B; % Adds corresponding elements
D = A - B; % Subtracts corresponding elements
```

⚠️ **Gotcha:** Matrices must have the same number of rows and columns for addition or subtraction.

Understanding Errors in MATLAB

- Errors can occur when performing operations with incompatible dimensions or invalid inputs.
- Common error example:

```
A = [1, 2; 3, 4];    % 2x2 matrix
B = [5, 6, 7];       % 1x3 vector
C = A * B;           % Attempting matrix multiplication
disp('awww, dang!');
```

Understanding Errors in MATLAB

- Errors can occur when performing operations with incompatible dimensions or invalid inputs.
- Common error example:

```
A = [1, 2; 3, 4];     % 2x2 matrix
B = [5, 6, 7];        % 1x3 vector
C = A * B;            % Attempting matrix multiplication
disp('awww, dang!');
```

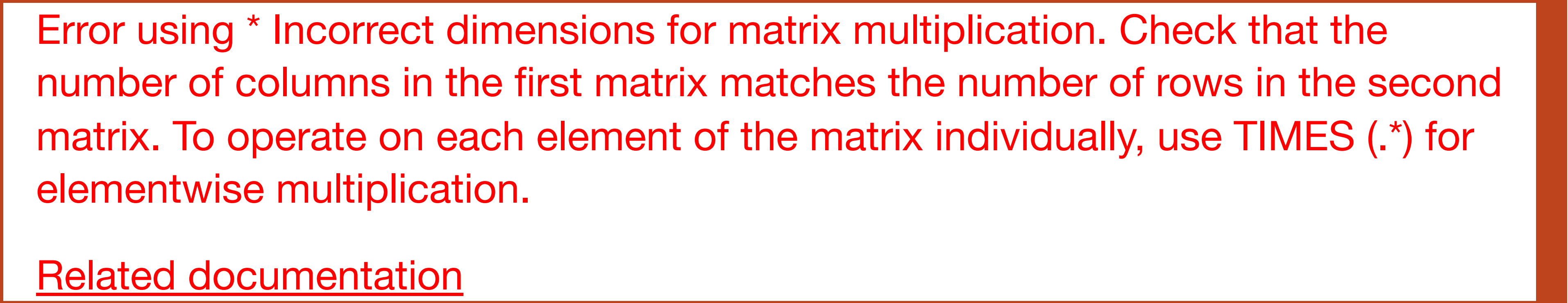
- Error using * Incorrect dimensions for matrix multiplication. Check that the number of columns in the first matrix matches the number of rows in the second matrix. To operate on each element of the matrix individually, use TIMES (.*) for elementwise multiplication.

  Related documentation

Understanding Errors in MATLAB

- Errors can occur when performing operations with incompatible dimensions or invalid inputs.
- Common error example:

```
A = [1, 2; 3, 4];    % 2x2 matrix
B = [5, 6, 7];       % 1x3 vector
C = A * B;           % Attempting matrix multiplication
disp('awww, dang!');
```

- 

  Error using * Incorrect dimensions for matrix multiplication. Check that the number of columns in the first matrix matches the number of rows in the second matrix. To operate on each element of the matrix individually, use TIMES (.*) for elementwise multiplication.

  Related documentation

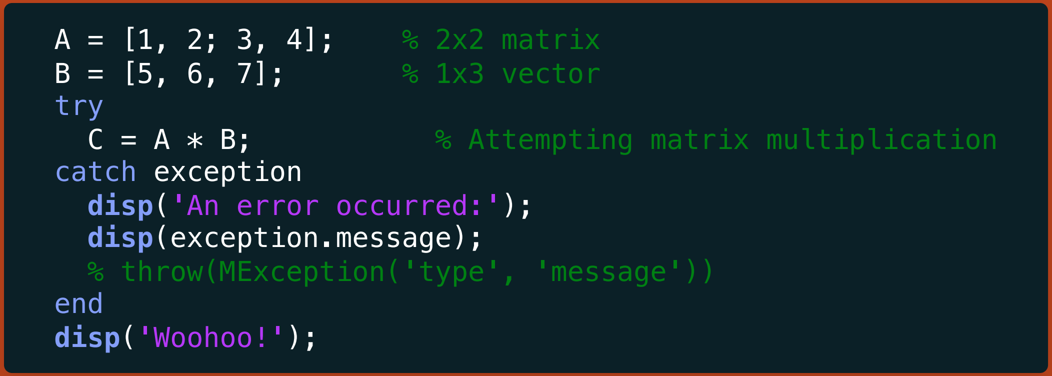
- **Note:** The program exits at this point

Introducing try/catch

- **Purpose:** Allows your program to handle errors without stopping execution abruptly.
- **Structure:**

```
try
    % Code that may produce an error
catch exception
    % Code to handle the error
end
```

Example: Handling Dimension Mismatch



```
A = [1, 2; 3, 4];    % 2x2 matrix
B = [5, 6, 7];       % 1x3 vector
try
  C = A * B;           % Attempting matrix multiplication
catch exception
  disp('An error occurred:');
  disp(exception.message);
  % throw(MException('type', 'message'))
end
disp('Woohoo!');
```

Output:

```
An error occurred:

Incorrect dimensions for matrix multiplication. Check that the number of columns in
the first matrix matches the number of rows in the second matrix. To operate on each
element of the matrix individually, use TIMES (.*) for elementwise multiplication.

Woohoo!
```

Explanation:

```
try Block:
        Attempts to execute the code that may cause an error.
catch Block:
        Executes if an error occurs in the try block.
        The exception object contains information about the error.
Benefit:
        Prevents the program from crashing.
        Allows for custom error messages or alternative actions.
```

Matrix vs. Element-wise Operations

Operation	Matrix	Element-wise
Addition (+)	Same as element-wise	Same as matrix
Subtraction (-)	Same as element-wise	Same as matrix
Multiplication	`*` (Matrix multiplication)	`.*` (Element-wise multiplication)
Division	`/` or `\` (Matrix division)	`./` or `.\` (Element-wise division)
Exponentiation	`^` (Matrix power)	`.^` (Element-wise power)

💡 **Tip:** Remember to use the dot (`.`) for element-wise operations when needed.

Matrix Multiplication

Matrix multiplication involves the dot product of rows and columns.

Condition: The number of **columns** in the first matrix must equal the number of **rows** in the second matrix.

For matrices A of size $m \times n$ and B of size $n \times p$, the product $C = AB$ will be of size $m \times p$.

Example:

```
A = [1 2; 3 4]; % 2x2 matrix
B = [5; 6];   % 2x1 matrix
C = A * B;    % Resulting in a 2x1 matrix
```

Mathematically:

$$C = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \times \begin{bmatrix} 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 1 * 5 + 2 * 6 \\ 3 * 5 + 4 * 6 \end{bmatrix} = \begin{bmatrix} 17 \\ 39 \end{bmatrix}$$

⚠ **Gotcha:** Always verify that matrix dimensions are compatible before multiplying.

Element-wise Multiplication

Use the `.*` operator to perform element-wise multiplication of matrices of the same size.

```
E = A .* B; % Multiplies corresponding elements
```

Example: If `A` and `B` are both 2x2 matrices:

```
A = [1 2; 3 4];
B = [5 6; 7 8];
E = A .* B; % E = [1*5, 2*6; 3*7, 4*8]
```

💡 **Tip:** Element-wise operations require matrices to be the same size.

Matrix Exponentiation

Matrix exponentiation involves multiplying a matrix by itself a certain number of times.

Example:

```
A_squared = A^2; % Equivalent to A * A
```

⚠️ **Gotcha:** This is not the same as squaring each element. For element-wise exponentiation, use `.^`.

Special Matrices

MATLAB provides functions to create special matrices:

- `zeros(n, m)`: Creates an n-by-m matrix of zeros.
- `ones(n, m)`: Creates an n-by-m matrix of ones.
- `eye(n)`: Creates an n-by-n identity matrix.
- `rand(n, m)`: Creates an n-by-m matrix of random numbers between 0 and 1.
- `magic(n)`: Creates an n-by-n magic square matrix.

What is a Magic Square?

A **magic square** is a square matrix where the sums of every row, column, and both main diagonals are the same.

Example:

```
magicMatrix = magic(3);
disp(magicMatrix);
% Output:
%     8     1     6
%     3     5     7
%     4     9     2
```

💡 **Fun Fact:** Magic squares have been studied for centuries and appear in various cultures and artworks.

3. Logical Vectors and Operations

What is a Logical Vector?

A **logical vector** is an array of logical values: `true` (1) or `false` (0). They are used for conditional indexing and control flow.

Creating Logical Vectors:

```
% Using conditions
vec = [1, 2, 3, 4, 5];
isEven = mod(vec, 2) == 0; % Checks if elements are even
% isEven = [0, 1, 0, 1, 0]

% Using logical() function
logicalVec = logical([1 0 1]);
% logicalVec = [1, 0, 1] (as logical values)
```

💡 **Tip:** `logical()` can create logical arrays from numeric arrays.

⚠️ **Gotcha:** A logical vector is different from a numeric vector. Operations on logical vectors may yield different results.

Logical Operations

Logical operators are used to perform element-wise comparisons and combine logical statements.

- `&`: Logical AND
- `|`: Logical OR
- `~`: Logical NOT
- `==`: Equal to
- `~=`: Not equal to
- `>`, `<`, `>=`, `<=`: Relational operators

Example:

```
A = [1, 2, 3, 4, 5];
B = [5, 4, 3, 2, 1];

% Element-wise comparison
isEqual = A == B; % [0, 0, 1, 0, 0]

% Combined conditions
condition = (A > 2) & (B < 4); % [0, 0, 1, 0, 0]
```

🧑‍💻 **Exercise:** Create a logical vector that identifies elements in `A` that are greater than 2 or less than 5.

Logical Indexing

Logical indexing allows you to select elements of an array that meet a certain condition.

Example:

```
vec = [10, 15, 20, 25, 30];
condition = vec > 20;
selectedElements = vec(condition); % Returns [25, 30]
```

💡 **Tip:** Logical indexing is a powerful tool for data manipulation and can simplify your code.

The `find()` Function

The `find()` function returns the indices of elements that meet a condition.

Usage:

```
indices = find(condition);
```

Example:

```
vec = [0, 3, 0, 7, 0, 5, 0, 9];
indices = find(vec); % Returns [2, 4, 6, 8]

% Using a condition
vec = [10, 15, 20, 25, 30, 35, 40];
indices = find(vec > 20); % Returns [4, 5, 6, 7]
```

💡 **Tip:** Use `find()` with conditions to locate positions of interest in your data.

🧑‍💻 **Exercise:** Given an array, use `find()` to locate all positions where the elements are divisible by 3.

The `find()` Function

The `find()` function can also return indices for 2D arrays/matrices.

Usage:

```
[r, c] = find(condition);
```

Example:

```
vec = [0, 3, 0, 15, 0, 5, 0;
    10, 9, 20, 40, 18, 35, 8];

[rows, cols] = find(vec > 10);
% rows contains [ 2; 1; 2; 2; 2 ]
% cols contains [ 3; 4; 4; 5; 6 ]

indices = find(vec > 10);
% indices contains [ 6; 7; 8; 10; 12 ]
```

Common Logical Functions

- `all(A)`: Returns `true` if all elements of `A` are non-zero or `true`.
- `any(A)`: Returns `true` if any element of `A` is non-zero or `true`.
- `logical(A)`: Converts numeric array `A` to a logical array.

Examples:

```
A = [0, 1, 2, 0, 4];

% Check if any elements are non-zero
hasNonZero = any(A); % Returns true

% Check if all elements are non-zero
allNonZero = all(A); % Returns false

% Convert to logical array
logicalA = logical(A); % [0, 1, 1, 0, 1]
```

👩‍💻 **Exercise:** Use `any()` and `all()` to determine if a dataset meets certain criteria.

Practical Application: Filtering Data

Suppose you have a dataset and you want to extract elements that meet certain criteria.

Example:

```
data = [12, 5, 8, 15, 7, 20];
% Extract elements greater than 10
condition = data > 10;
filteredData = data(condition); % [12, 15, 20]
```

💡 **Tip:** Logical indexing allows for efficient data filtering without the need for loops.

MATLAB Tables

What is a Table?

- A **table** is a data type suitable for storing column-oriented or tabular data.
- Tables can hold different types of data in each column, including numbers, strings, or even other tables.

Creating a Table

From Arrays

You can create a table by combining arrays of the same length.

```
% Example data
Name = ["Alice"; "Bob"; "Charlie"];
Age = [25; 30; 35];
Height = [5.5; 6.1; 5.8];

% Create the table
myTable = table(Name, Age, Height);
```

From Existing Data

Tables can also be created from existing arrays or data already in the workspace.

```
% Convert an array into a table
randArray = rand(5, 3); % Random 5x3 array
myTable = array2table(randArray, 'VariableNames', ["Var1", "Var2", "Var3"]);
```

⚠️**Gotcha**: Be sure to use double quotes `"` around strings

Accessing Data in Tables

By Variable Name

You can access the data in a table by referring to the variable names (columns).

```
% Access the 'age' column
ages = myTable.Age;

% Access multiple columns
subset = myTable(:, {'Name', 'Height'});
```

By Row and Column Index

Tables also support matrix-style indexing.

```
% Access the data in the 2nd row, 3rd column
value = myTable{2, 3};

% Access the entire 1st row
row1 = myTable(1, :);
```

Logical Indexing

Logical conditions can be used to filter rows.

```
% Find rows where Age is greater than 30
olderThan30 = myTable(myTable.Age > 30, :);
```

Updating Data in Tables

Update Specific Elements

You can directly assign new values to elements in the table.

```
% Update the Age of the 2nd row
myTable.Age(2) = 32;

% Update a specific element
myTable{3, 'Height'} = 6.0;
```

Adding New Variables (Columns)

Adding new columns is straightforward:

```
% Add a new column 'Weight'
myTable.Weight = [150; 180; 165];
```

Adding New Rows

New rows can be appended using the `end+1` indexing.

```
% Add a new row
newRow = {"Diana", 28, 5.7, 140};
myTable(end+1, :) = newRow;
```

Summary

- **Tables** are versatile for handling mixed data types.
- Access data by **variable names** or **indices.**
- **Update** tables by directly assigning values to elements, variables, or rows.
- Use `[]` when you want the value, and `{}` when you want to work with the table
  - Also, use `{}` when you have mixed data (numbers and strings)

Conclusion

We've covered the basics of vectors, matrices, and logical operations in MATLAB, providing you with foundational tools for engineering computations.

Remember to:

- Ensure matrix dimensions are compatible for operations.
- Utilize logical indexing to simplify data manipulation.
- Explore MATLAB's built-in functions to enhance your programs.

Feel free to experiment with these concepts to deepen your understanding.