OPERATION OCEAN STATE: Zombie Outbreak Simulator

Phase-Based MATLAB Protocol: Introduction to Matlab

Mission: Contain the Spread
Location: Rhode Island
Authors: Chloe Griffin and Additi Pandey
Clearance: Top Secret

Mission Protocol

1. Phase 1: Initialize RI Data (Arrays and Built-in Functions)
2. Phase 2: Migration (Matrix Multiplication)
3. Phase 3: Medical Response (Loops and Functions)
4. Phase 4: Satellite Scan (Vectorization)
5. Phase 5: The 30-Day Forecast (ODEs)
6. Classified Phase: Advanced Mapping

Phase 1: Initialize RI Data

Incoming Intel:
Field agents have reported infection counts across all 5 counties in Rhode Island. We have raw numbers, but the data is scattered.

The Objectives:

- Initialize infection map of Rhode Island and track infections.
- We need to calculate the **Total Threat Level** (Sum).
- We need to identify the **Hotspot** (Max).

Raw Data Feed

Providence: 1000
Kent: 50
Washington: 10
Bristol: 0
Newport: 5

Phase 1: Initialize RI Data

Incoming Intel:
Field agents have reported infection counts across all 5 counties in Rhode Island. We have raw numbers, but the data is scattered.

The Objectives:

- Initialize infection map of Rhode Island and track infections.
- We need to calculate the **Total Threat Level** (Sum).
- We need to identify the **Hotspot** (Max).

Raw Data Feed

Providence: 1000
Kent: 50
Washington: 10
Bristol: 0
Newport: 5

Solution: Arrays (Vectors) & Built-in Functions

Training Module: Variables & Vectors

1. Defining Variables

Assign values using the = sign.
End with ; to hide output.

- **Scalars (Single Numbers)**

```
x = 10;          % Hides output

y = 20           % Shows output
```

- **Vectors (Lists of Numbers)**

```
% Row Vector (Spaces or Commas)
r = [1, 2, 3, 4];

% Column Vector (Semicolons)
c = [1; 2; 3; 4];
```

2. Indexing (Modify Data)

Access specific items using parenthesis ().

```
v = [10; 20; 30];

val = v(2);          % Gets 20

v(3) = 100;          % Changes 30 to 100
```

3. Analysis Functions

```
data = [10; 5; 20];

total = sum(data);   % Result: 35

[highest, index] = max(data); %Result: 20, 3
```

4. Saving Data

```
data = [12; 1; 2];

save("file_name.mat", 'data')
```

Mission Execution: Initialize Map

Task: Open your script `phase1.m`.

We need to store the zombie population for Rhode Island in a **Column Vector** (5x1) and analyze the threat.

The Order: [Providence; Kent; Washington; Bristol; Newport]

```
%% PHASE 1: THE SITUATION ROOM (Vectors & Indexing)
fprintf('--- PHASE 1: INITIALIZING RI DATA ---\n');

% 1. Define County Populations (Zombies)
%    TODO: Create a COLUMN vector with values: 1000, 50, 10, 0, 5
zombie_pop = [___];
% 2. Indexing: Update specific counties
%    TODO: Double the infection count in Washington County (3rd element)
zombie_pop(3) = ___ * 2;
% 3. Basic Stats
%    TODO: Calculate sum and find the maximum value
total_threat = sum(___);
[max_zombies, worst_county_idx] = max(___);
fprintf('Total Zombies in RI: %d\n', total_threat);
fprintf('Worst County Index: %d (Count: %d)\n\n', worst_county_idx, max_zombies);
%4. Save zombie population matrix
%   TODO: Save edited zombie_pop for next phase
save("____.mat", '__')
```

Mission Execution: Solution

Correct Output:
Total Zombies in RI: 1075
Worst County Index: 1 (Count: 1000)

```
%% PHASE 1: THE SITUATION ROOM (Vectors & Indexing)
%  Goal: Initialize county data and learn Column Vectors vs Row Vectors

fprintf('--- PHASE 1: INITIALIZING RI DATA ---\n');
% 1. Define County Populations (Zombies)
%    Order: [Providence; Kent; Washington; Bristol; Newport]
zombie_pop = [1000; 50; 10; 0; 5];
% 2. Indexing: Update specific counties
zombie_pop(3) = zombie_pop(3) * 2; % Outbreak doubles in Washington Cty
% 3. Basic Stats
total_threat = sum(zombie_pop);
[max_zombies, worst_county_idx] = max(zombie_pop);
fprintf('Total Zombies in RI: %d\n', total_threat);
fprintf('Worst County Index: %d (Count: %d)\n\n', worst_county_idx, max_zombies);
%4. Save zombie population matrix
save("zombiepop.mat", 'zombie_pop')
```

Phase 2: Migration (Matrix Multiplication)

The Situation:
Zombies do not stay put. They migrate between counties based on highway access. We need to predict the population for **tomorrow**.

The Math: Linear Algebra
We use a **Migration Matrix (M)** to transform our population vector.

$$\vec{x}_{tomorrow} = \mathbf{M} \times \vec{x}_{today}$$

Matrix Logic

Rows = Destination (To)
Cols = Origin (From)

Example:
Value $M_{1,2}$ is movement **FROM** County 2 (Kent) **TO** County 1 (Providence).

Phase 2: Migration (Matrix Multiplication)

The Situation:
Zombies do not stay put. They migrate between counties based on highway access. We need to predict the population for **tomorrow**.

The Math: Linear Algebra
We use a **Migration Matrix (M)** to transform our population vector.

$$\vec{x}_{tomorrow} = \mathbf{M} \times \vec{x}_{today}$$

Matrix Logic

Rows = Destination (To)
Cols = Origin (From)

Example:
Value $M_{1,2}$ is movement **FROM** County 2 (Kent) **TO** County 1 (Providence).

Solution: Matrix Multiplication (*)

Training Module: Matrices in MATLAB

1. Loading Data

```
load("file_name.mat")
```

2. Defining Matrices (2D)

Use ; to start a new row.

```
% A 2x2 Matrix
A = [1, 2;
     3, 4];

% A 3x3 Identity Matrix
I = eye(3);
```

3. Checking Dimensions

```
sz = size(A); % Returns [2 2]
```

4. The Golden Rule

MATLAB is sensitive to operators!

Warning: * vs .*

- `A * B` → **Matrix Math** (Linear Algebra / Dot Product).
- `A .* B` → **Element-wise** (Mult matching spots).

Micro-Task (1 min)

1. Define `M = [1 0; 0 1]`
2. Define `v = [10; 20]`
3. Calculate `v1 = M * v`
4. Calculate `v2 = M .* v`
5. Compare `v1` and `v2`

Mission Execution: Predict Movement

Task: Open your script `phase2.m`.

1. Review the provided Migration Matrix **M**.
2. Calculate `next_day_pop` using Matrix Multiplication.

```
%% PHASE 2: MIGRATION (Matrix Math: A*x)
%  Goal: Predict movement using Linear Algebra (Matrix Multiplication)
% 1. Load zombie population matrix
%   TODO: Load edited zombie_pop for next phase
load("_______.mat")
fprintf('--- PHASE 2: TRACKING MIGRATION ---\n');
%Check to see if loaded data is edited from phase1
% 2. Create Migration Matrix (5x5) provided by Intel
M = [0.90, 0.05, 0.00, 0.00, 0.00;
     0.10, 0.90, 0.10, 0.00, 0.00;
     0.00, 0.05, 0.80, 0.10, 0.00;
     0.00, 0.00, 0.05, 0.90, 0.05;
     0.00, 0.00, 0.05, 0.00, 0.95];
% 3. Predict Tomorrow's location
%    TODO: Multiply Matrix M by Vector zombie_pop (Hint: Use * not .*)
next_day_pop = ___;
disp('Projected Zombie Count for Tomorrow (By County):');
disp(next_day_pop);
```

Mission Execution: Solution

Analysis: The population has shifted. Providence (Row 1) received overflow from Kent (Row 2).

```
%% PHASE 2: MIGRATION (Matrix Math: A*x)
%  Goal: Predict movement using Linear Algebra (Matrix Multiplication)

% 1. Load zombie population matrix
%   TODO: Load edited zombie_pop for next phase
load("zombiepop.mat")
fprintf('--- PHASE 2: TRACKING MIGRATION ---\n');
%Check to see if loaded data is edited from phase1

% 2. Create Migration Matrix (5x5)
%     Rows = Destination, Columns = Origin
M = [0.90, 0.05, 0.00, 0.00, 0.00;  % To Providence
     0.10, 0.90, 0.10, 0.00, 0.00;  % To Kent
     0.00, 0.05, 0.80, 0.10, 0.00;  % To Washington
     0.00, 0.00, 0.05, 0.90, 0.05;  % To Bristol
     0.00, 0.00, 0.05, 0.00, 0.95]; % To Newport

% 3. Predict Tomorrow's location
next_day_pop = M * zombie_pop;

disp('Projected Zombie Count for Tomorrow (By County):');
disp(next_day_pop);
```

Console Output: 902.50, 147.00, 18.50, 1.25, 5.75

Phase 3: Medical Response

The Situation:
The zombie infection is overwhelming the medical infrastructure. We have two distinct operational orders:

1. **Operation Vax:** Deliver supplies to a specific list of 3 clinics.
2. **Operation Clear:** Treat ER patients until the waiting room is empty OR supplies run out.

Choosing the Tool

FOR Loop:
Use when you know the *exact number* of iterations (e.g., ”For every clinic...”).

WHILE Loop:
Use when the end depends on a *condition* (e.g., ”While patients ¿ 0”).

Phase 3: Medical Response

The Situation:
The zombie infection is overwhelming the medical infrastructure. We have two distinct operational orders:

1. **Operation Vax:** Deliver supplies to a specific list of 3 clinics.
2. **Operation Clear:** Treat ER patients until the waiting room is empty OR supplies run out.

Choosing the Tool

FOR Loop:
Use when you know the *exact number* of iterations (e.g., ”For every clinic...”).

WHILE Loop:
Use when the end depends on a *condition* (e.g., ”While patients ¿ 0”).

Solution: Control Flow (For/While) & Scope

Training Module: Looping Logic

1. The FOR Loop

Iterate through a set range.

```
% Count 1 to 3
for k = 1:3
    disp(k);
end
```

2. The WHILE Loop

Run as long as condition is TRUE.

```
x = 10;
while x > 0
    x = x - 2;
end
```

3. Logical Operators

Essential for complex conditions.

- `&&` (AND): Both must be true.
- `||` (OR): One must be true.

Micro-Task (1 min)

Write a loop that runs while `A > 0` **AND** `B > 0`.

```
while A > 0 && B > 0
    % do something
end
```

Mission Execution: Hospital Triage

Task: Open your script `phase3.m`.
Complete the loops. Pay attention to the **function inputs**!

```
% --- SCENARIO A: Vax Drive ---
clinics = [1, 2, 3];
risk_level = 0.3;
fprintf('Inspecting Clinics...\n');

% TODO: Loop k from 1 to length(clinics)
for k = ___:___

    % TODO: Call 'inspect_clinic_safety'
    % MUST pass 'k' and 'risk_level'
    is_safe = inspect_clinic_safety(__, __);

    if is_safe
        fprintf('Clinic %d: SECURE.\n', k);
    else
        fprintf('Clinic %d: UNSAFE.\n', k);
    end
end
```

```
% --- SCENARIO B: ER Overflow ---
patients = 200;
supplies = 500;
hour = 0;

% TODO: Loop while patients > 0
% AND supplies > 0
while ___ > 0 && ___ > 0
    hour = hour + 1;

    % TODO: Call 'treat_batch'
    % Pass 'supplies' as input
    [cured, used] = treat_batch_of_patients(
        ___);

    % Update State
    patients = patients - cured;
    supplies = supplies - used;

    fprintf('Hour %d: Pats: %d\n', hour,
        patients);
end
```

Phase 3: The Backend Logic (Local Functions)

Analysis: These functions live at the bottom of the script. They handle the "dirty work" so the main loops remain clean and readable.

1. Resource Management Logic

```
function [cured, cost] = treat_batch_of_patients(
    medicine_available)
    % Calculates cures based on supplies
    max_capacity = 40;
    cost_per_person = 2;

    people_waiting = floor(rand() * max_capacity)
        ;
    cost = people_waiting * cost_per_person;

    % Check affordability
    if cost > medicine_available
        cured = floor(medicine_available /
            cost_per_person);
        cost = medicine_available;
    else
        cured = people_waiting;
    end
end
```

2. Safety Check

```
function is_secure =
    inspect_clinic_safety(id, threshold
    )
    val = rand();
    if val > threshold
        is_secure = true;
    else
        fprintf('  [ALERT]: Breach at
            Site #%d.\n', id);
        is_secure = false;
    end
end
```

3. Mission Status

```
function check_mission_status(
    patients_remaining)
    if patients_remaining <= 0
        disp('  Result: SUCCESS.');
    else
        disp('  Result: FAILURE.');
    end
end
```

Mission Execution: Solution

Analysis: Notice how we must pass variables like `risk_level` inside the parentheses so the function can "see" them.

```matlab
% --- SCENARIO A: Vax Drive ---
clinics = [1, 2, 3];
risk_level = 0.3;
fprintf('Inspecting Clinics...\n');

for k = 1:length(clinics)
    % Pass 'k' and 'risk_level'
    is_safe = inspect_clinic_safety(k,
        risk_level);

    if is_safe
        fprintf('Clinic %d: SECURE.\n', k);
    else
        fprintf('Clinic %d: UNSAFE.\n', k);
    end
end
```

```matlab
% --- SCENARIO B: ER Overflow ---
patients = 200;
supplies = 500;
hour = 0;

% Keep going while BOTH are valid
while patients > 0 && supplies > 0
    hour = hour + 1;

    % Pass 'supplies'
    [cured, used] = treat_batch_of_patients(
        supplies);

    patients = patients - cured;
    supplies = supplies - used;

    fprintf('Hour %d: Pats: %d\n', hour,
        patients);
end
```

Phase 4: Satellite Scan (Vectorization)

The Situation:
We have a thermal satellite image of the entire state (10,000 pixels).

- Living Humans = Warm ($> 90°$F).
- Zombies = Cold ($< 10°$F).

The Problem:
Checking 10,000 pixels one-by-one with a loop is inefficient code. We need to process the whole map **instantly**.

The Tool: Logical Masks

Instead of asking "Is pixel 1 cold?", we ask:
"Return a map of ALL cold pixels."

This creates a **Logical Array** (Mask) of True/False values.

Phase 4: Satellite Scan (Vectorization)

The Situation:
We have a thermal satellite image of the entire state (10,000 pixels).

- Living Humans = Warm ($> 90^{\circ}$F).
- Zombies = Cold ($< 10^{\circ}$F).

The Problem:
Checking 10,000 pixels one-by-one with a loop is inefficient code. We need to process the whole map **instantly**.

The Tool: Logical Masks

Instead of asking "Is pixel 1 cold?", we ask:
"Return a map of ALL cold pixels."

This creates a **Logical Array** (Mask) of True/False values.

Solution: Vectorization (Logical Indexing)

Training Module: Logical Indexing

1. Creating a Mask

Compare a whole array to a number.

```
data = [10, 2, 50, 5];

% Ask: Which are less than 10?
mask = data < 10;
```

Result (Logical Array):

`[0, 1, 0, 1]`

(0=False, 1=True)

2. counting "True" Hits

Use `sum`.

```
% Summing a logical array counts
% the number of "True" items.

count = sum(mask);
% Result: 2
```

2D Matrices

If `data` is a matrix (2D), use:

`sum(mask, 'all')`

Micro-Task (1 min)

1. `temps = [98, 5, 102]`
2. Find zombies: `is_zom = temps < 10`
3. Count them.

Mission Execution: Thermal Scan

Task: Open your script `phase4.m`.
Find the zombies in the 100x100 grid without using a loop.

1. Create the mask (`zombie_mask`).
2. Count the total threats using `sum(..., 'all')`.

```
%% PHASE 4: SATELLITE SCAN (Vectorization)
%  Goal: Process large arrays instantly (No Loops)
fprintf('--- PHASE 4: SATELLITE THERMAL SCAN ---\n');

% 1. Simulate 100x100 thermal grid
thermal_map = rand(100, 100) * 100;

% 2. Create Mask: Zombies are Cold (< 10 deg)
%    TODO: Create a Logical Array (True/False) where thermal_map < 10
zombie_mask = ___;

% 3. Count instantly
%    TODO: Sum all the 'True' values in zombie_mask
total_detected = sum(___, 'all');

fprintf('Satellite Scan Complete. Cold Signatures: %d\n\n', total_detected);
```

Mission Execution: Solution

Analysis: This operation happens in nanoseconds, regardless of grid size.

```matlab
%% PHASE 4: SATELLITE SCAN (Vectorization)
%  Goal: Process large arrays instantly (No Loops)
fprintf('--- PHASE 4: SATELLITE THERMAL SCAN ---\n');

% 1. Simulate 100x100 thermal grid (0 to 100 degrees)
thermal_map = rand(100, 100) * 100;

% 2. Create Mask: Zombies are Cold (< 10 deg)
zombie_mask = thermal_map < 10;

% 3. Count instantly
total_detected = sum(zombie_mask, 'all');

fprintf('Satellite Scan Complete. Cold Signatures: %d\n\n', total_detected);
```

Satellite Recon: Visual Confirmation

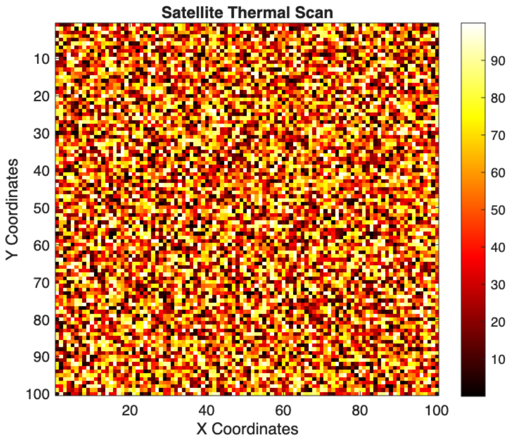


Figure 1: Raw Thermal Scan

(Red/Yellow = Warm, Dark = Cold)

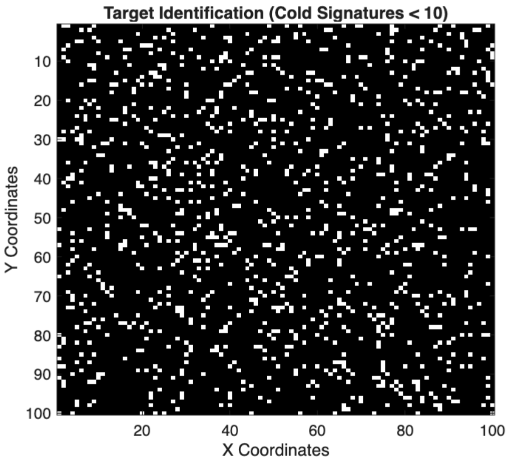


Figure 2: Target Acquisition

(White = Zombies, Black = Empty)

Phase 5: The 30-Day Forecast (ODEs)

The Situation:
We need to predict the long-term infection curve. We use the standard epidemiological **SIR Model**:

- **S**usceptible (Healthy)
- **I**nfected (Zombies)
- **R**ecovered (Immune/Dead)

The Math (Differential Equations):

$$\frac{dS}{dt} = -\beta SI$$

$$\frac{dI}{dt} = \beta SI - \gamma I$$

$$\frac{dR}{dt} = \gamma I$$

Vector Mapping

MATLAB uses a vector `y` to store variables:

- $S \rightarrow$ `y(1)`
- $I \rightarrow$ `y(2)`
- $R \rightarrow$ `y(3)`

Phase 5: The 30-Day Forecast (ODEs)

The Situation:
We need to predict the long-term infection curve. We use the standard epidemiological **SIR Model**:

- **S**usceptible (Healthy)
- **I**nfected (Zombies)
- **R**ecovered (Immune/Dead)

The Math (Differential Equations):

$$\frac{dS}{dt} = -\beta SI$$

$$\frac{dI}{dt} = \beta SI - \gamma I$$

$$\frac{dR}{dt} = \gamma I$$

Vector Mapping

MATLAB uses a vector `y` to store variables:

- $S \rightarrow$ `y(1)`
- $I \rightarrow$ `y(2)`
- $R \rightarrow$ `y(3)`

Solution: ODE Solver (ode45) & Anonymous Functions

Training Module: Solving ODEs

1. Anonymous Functions

Creating a quick function without a separate file. Syntax: `@(inputs) formula`

```
% A simple square function
sq = @(x) x.^2;

ans = sq(5); % Returns 25
```

2. The Solver: ode45

The workhorse of engineering integration.

```
[t, y] = ode45(func, time, initial_state);
```

3. Setup Checklist

1. **Function:** Define the equations (dydt).
2. **Time:** How long to run? `[0 30]`
3. **Initial State:** Where do we start? `y0`

Micro-Task (1 min)

Define a function that doubles a number:

```
f = @(x) x * 2;
f(10) % Should be 20
```

Mission Execution: SIR Prediction

Task: Open your script `phase5.m`.
Fill in the missing physics for `dI/dt` and call the solver.

1. Complete the differential equation for Infected (Line 2 of vector).
2. Execute `ode45` with the correct arguments.

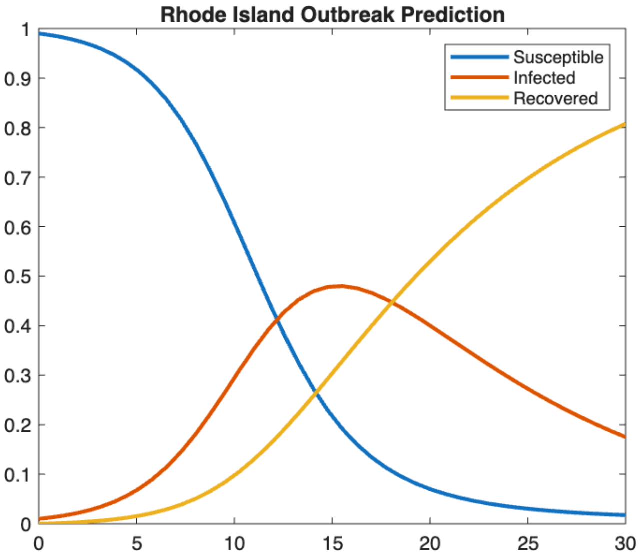
```
fprintf('--- PHASE 5: SIR MODEL PREDICTION ---\n');

% 1. Setup Time and Initial Conditions
tspan = [0 30];
y0 = [0.99; 0.01; 0]; % 99% Susceptible (S), 1% Infected (I), 0% Recovered (R)
beta = 0.5; gamma = 0.1;

% 3. Define the System (Anonymous Function)
%    TODO: Complete the dI/dt equation (Line 2)
%    dS/dt = -beta * S * I
%    dI/dt =  beta * S * I - gamma * I
%    dR/dt =                 gamma * I
dydt = @(t, y) [ -beta * y(1) * y(2);
                  ___ * y(1) * y(2) - ___*y(2);
                  gamma * y(2) ];

% 4. Solve using ODE45
%    TODO: Call ode45(function, time, initial_state)
[t, y] = ode45(___, ___, ___);

% 5. Visualize
```

Mission Execution: Solution

Analysis: The Blue line (Susceptible) drops as the Red line (Infected) spikes. Eventually, the Yellow line (Recovered) dominates.

```
%% PHASE 5: THE 30-DAY FORECAST (ODEs)
%  Goal: Solve the SIR Model
fprintf('--- PHASE 5: SIR MODEL PREDICTION ---\n');

% 1. Setup
tspan = [0 30];
y0 = [0.99; 0.01; 0]; % S, I, R

% 2. Parameters & System
beta = 0.5; gamma = 0.1;
dydt = @(t, y) [ -beta * y(1) * y(2);                  % dS/dt
                  beta * y(1) * y(2) - gamma*y(2);   % dI/dt
                  gamma * y(2) ];                      % dR/dt

% 3. Solve
[t, y] = ode45(dydt, tspan, y0);

% 4. Visualize
figure;
plot(t, y, 'LineWidth', 2);
legend('Susceptible', 'Infected', 'Recovered');
title('Rhode Island Outbreak Prediction');
grid on; xlabel('Days');
```

Phase 5 Analysis: The 30-Day Forecast



Strategic Forecast

This model predicts the course of the outbreak over 30 days:

- **Susceptible (Blue):**
- **Infected (Red):**
- **Recovered (Yellow):**

Advance Phase: Topographical Vision

The Situation:
We need to identify safe extraction zones in high-altitude terrain. Can we identify safe landing zones through the infection fog?

Raw Intel

1. Grid: 100×100
2. Source: Thermal Satellite

Two questions we need to answer:

- What is the terrain like? Spreadsheet information would not make the cut. We need to see it!
- How would the pilot know where the fog is less dense? We need a map with boundaries.

Advance Phase: Topographical Vision

The Situation:
We need to identify safe extraction zones in high-altitude terrain. Can we identify safe landing zones through the infection fog?

Raw Intel

1. Grid: 100×100
2. Source: Thermal Satellite

Two questions we need to answer:

- What is the terrain like? Spreadsheet information would not make the cut. We need to see it!
- How would the pilot know where the fog is less dense? We need a map with boundaries.

Solution: Surface Plots and Contour Maps

1. Building Grid using Meshgrid

Given two vectors **x** and **y**, we can create a grid with elements (x, y).

```
x = 1:4;
y = 5:7;
[X,Y] = meshgrid(x,y)

ans
X =
     1     2     3     4
     1     2     3     4
     1     2     3     4
Y =
     5     5     5     5
     6     6     6     6
     7     7     7     7
```

2. Surface Plot

The workhorse of engineering integration.
Syntax: `surf(X, Y, Z);`

3. Level Curves or Contour Maps

1. **Array Definition:** Define three arrays.
2. **Syntax:** `contour(X, Y, Z, 10);`
3. What do you observe when you change 10 to 5 or 20?

Mission Execution

Task: Open your script `phase_advanced.m`.
Build the maps needed by the air support to locate extraction zones: The height of the contours show the magnitude of the peaks in the surface plot.

```
% 1. Define grid vectors
x = linspace(-5, 5, 100);
y = linspace(-5, 5, 100);

% 2. Build the coordinate grid
[X, Y] = meshgrid(x, y);

% 3. Compute Infection Surface (two overlapping hotspots)
Z = 40 * exp(-((X).^2 + (Y).^2)) + 10 * exp(-((X-4).^2 + (Y-4).^2));
% 4. Plot the terrain
figure;
surf(X, Y, Z);
colorbar;
title('Infection Density Surface'); xlabel('East-West'); ylabel('North-South');
zlabel('Threat Level');
% 5. Locate safe zones
figure;
contourf(X, Y, Z, 15);
colorbar;
title('Extraction Zone Map'); xlabel('East-West');  ylabel('North-South');
fprintf('Prediction Complete. Check Figures.\n');
```

Classified Clearance: 3D Terrain Analysis

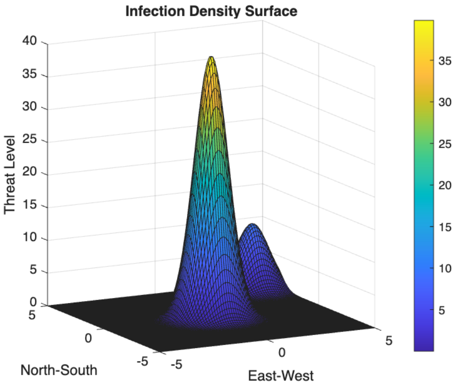


Figure 1: Threat Topology (3D)

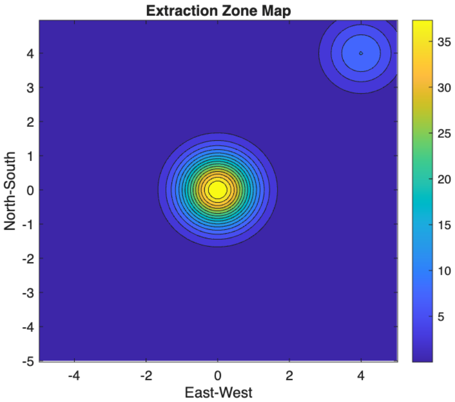


Figure 2: Aerial Extraction Zones

Analysis Complete.

Save all variables and report to the Governor.