Assignment for Part 2 in ME5701

The code is open-source and available on **github.com/0123YHYQ1129/YeQing_Mathematics_For_Engineering.** It is written in **Jupyter Notebook using Python 3.11.5 and MATLAB R2024a.** This report investigates the stability of the Mathieu equation, a linear second-order differential equation with periodic coefficients. Two distinct methods are employed: Floquet Theory and Fourier Series Approach.A parametric study is conducted by varying the parameters δ and ε , generating a **stability map** that visually depicts stable and unstable regions in the parameter space.The time-domain behavior of the Mathieu equation is simulated for both stable and unstable parameter combinations, confirming the predictions of the stability analysis.The results demonstrate the effectiveness of both Floquet theory and the Fourier series approach in analyzing the stability characteristics of the Mathieu equation.

Authors:

- $QingYe$, A0304458A, National University of Singapore (NUS), College of Design and Engineering.
- $RanranWang$, A0303612W, National University of Singapore (NUS), College of Design and Engineering.

Task (a)

The Mathieu equation is:

$$\frac{d^2y}{dx^2} + (\delta + 2\epsilon\cos(t))u = 0$$

To simplify the caculation, we use v to represent the velocity.:

$$\frac{du}{dt} = v$$

$$\frac{dv}{dt} = -(\delta + 2\epsilon\cos(t))u$$

Tranfer it to matrix form:

$$\frac{d}{dt}\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -(\delta + 2\epsilon\cos(t)) & 0 \end{pmatrix}\begin{pmatrix} u \\ v \end{pmatrix}$$

The stability of the system is determined by the real parts of its eigenvalues λ. Specifically, the system is unstable if at least one eigenvalue has a positive real part. Conversely, the system is stable if the real parts of all the eigenvalues are negative.

$$u(t) = e^{\lambda t} \sum_{n=-\infty}^{\infty} a_n e^{int}$$

$$v(t) = e^{\lambda t} \sum_{n=-\infty}^{\infty} b_n e^{int}$$

Substitute the formula, which has already been proven in the problem:

$$\lambda a_n + ina_n = b_n$$

Similarly, we can prove:

Given Equation (2):

$$\frac{dv}{dt} = -(\delta + 2\epsilon \cos t)u$$

We express v(t) and u(t) as:

$$v(t) = e^{\lambda t} \sum_{n=-\infty}^{\infty} b_n e^{int}$$

$$u(t) = e^{\lambda t} \sum_{n=-\infty}^{\infty} a_n e^{int}$$

Substitute these expansions into the right-hand side of Equation (2).

Since cos t can be represented as:

$$\cos t = \frac{e^{it} + e^{-it}}{2}$$

Equation (2) can be rewritten as:

$$\frac{dv}{dt} = -\left(\delta + \epsilon(e^{it} + e^{-it})\right) u$$

Substituting

$$u(t) = e^{\lambda t} \sum_{n=-\infty}^{\infty} a_n e^{int}$$

we get:

$$\frac{dv}{dt} = -\delta \left(e^{\lambda t} \sum_{n=-\infty}^{\infty} a_n e^{int} \right) - \epsilon \left(e^{\lambda t} \sum_{n=-\infty}^{\infty} a_n e^{int} \right) (e^{it} + e^{-it})$$

Expanding

$$\left(e^{\lambda t} \sum_{n=-\infty}^{\infty} a_n e^{int} \right) (e^{it} + e^{-it})$$

we obtain:

$$\frac{dv}{dt} = -\delta e^{\lambda t} \sum_{n=-\infty}^{\infty} a_n e^{int} - \epsilon e^{\lambda t} \sum_{n=-\infty}^{\infty} a_n (e^{i(n+1)t} + e^{i(n-1)t})$$

Combining these parts, we get:

$$\frac{dv}{dt} = e^{\lambda t} \left(-\delta a_n - \epsilon a_{n-1} - \epsilon a_{n+1}\right) e^{int}$$

The derivative on the left side is:

$$\frac{dv}{dt} = \frac{d}{dt}\left(e^{\lambda t} \sum_{n=-\infty}^{\infty} b_n e^{int}\right) = e^{\lambda t} \sum_{n=-\infty}^{\infty} (\lambda + in) b_n e^{int}$$

Setting the right side equal to the left side, we obtain:

$$\lambda b_n + in b_n = -(\delta a_n + \epsilon a_{n-1} + \epsilon a_{n+1})$$

This is the desired Equation (5).

Task(b)

$$\lambda a_n = -in a_n + b_n$$

$$\lambda b_n = -in b_n - (\delta a_n + \epsilon a_{n-1} + \epsilon a_{n+1})$$

As delta = 1 epsilon = 1,the fumula becomes

$$\lambda a_n = -in a_n + b_n$$

$$\lambda b_n = -in b_n - (a_n + a_{n-1} + a_{n+1})$$

By converting the formula into matrix form, we get matrix M

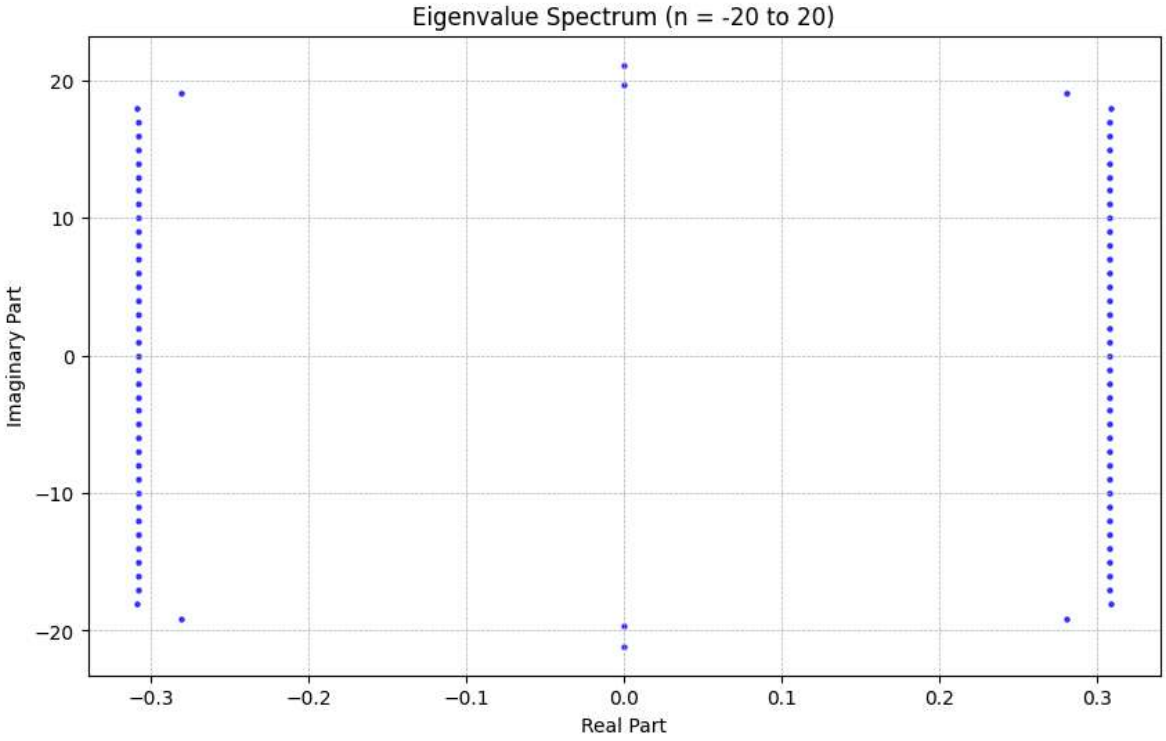
$$\lambda \begin{pmatrix} \cdots \\ a_{n-1} \\ a_n \\ a_{n+1} \\ \cdots \\ \cdots \\ b_{n-1} \\ b_n \\ b_{n+1} \\ \cdots \end{pmatrix} = \underbrace{\begin{pmatrix} & & & & \cdots & & & & \\ & ? & ? & 0 & & ? & ? & 0 & \\ & 0 & ? & 0 & & 0 & ? & 0 & \\ & 0 & ? & ? & & 0 & ? & ? & \\ \cdots & & & & \cdots & & & & \cdots \\ & & & & \cdots & & & & \\ & ? & ? & 0 & & ? & 0 & 0 & \\ & ? & ? & ? & & 0 & ? & 0 & \\ & 0 & ? & ? & & 0 & 0 & ? & \\ & & & & \cdots & & & & \end{pmatrix}}_{\mathbf{M}} \begin{pmatrix} \cdots \\ a_{n-1} \\ a_n \\ a_{n+1} \\ \cdots \\ \cdots \\ b_{n-1} \\ b_n \\ b_{n+1} \\ \cdots \end{pmatrix}$$

Construct the matrix M based on equations (6) and (7),

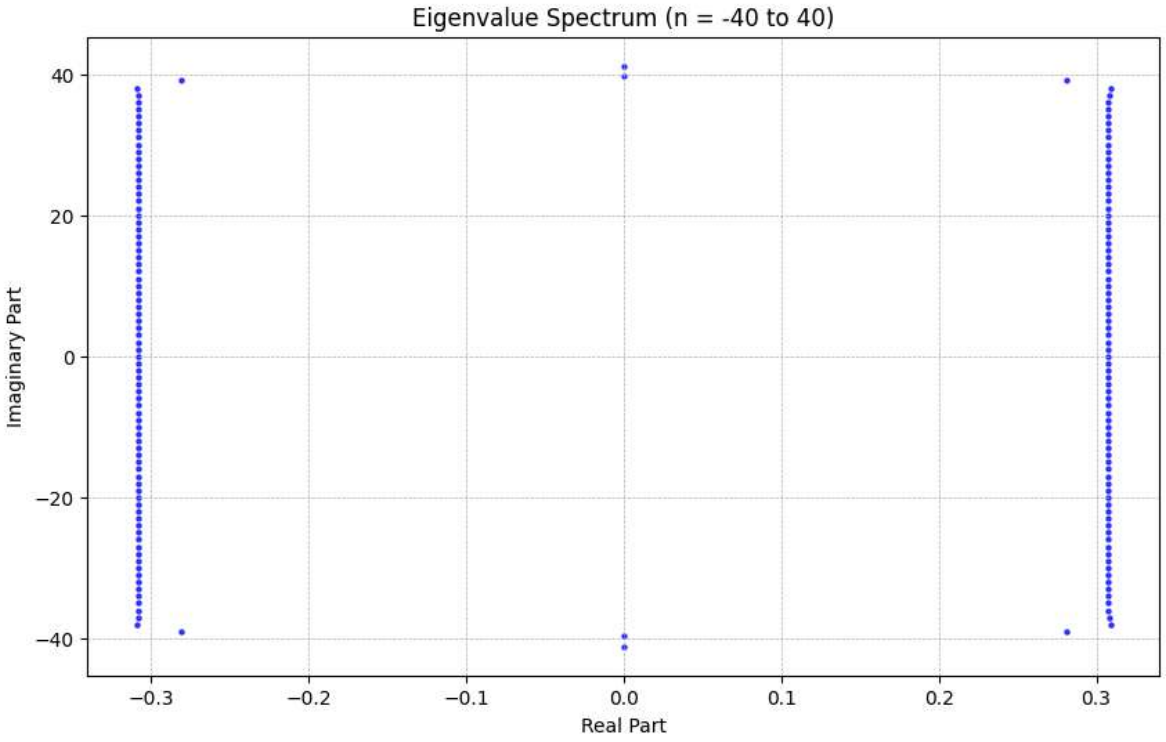
In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from numpy.linalg import eigvals

# Define parameters
delta = 1
```

```python
epsilon = 1

# Function to construct the M matrix
def construct_M(n_min, n_max):
    N = n_max - n_min + 1  # Matrix dimension
    M = np.zeros((2 * N, 2 * N), dtype=complex)  # Create a 2N x 2N complex matr

    for n in range(N):
        # Indices for a_n and b_n
        a_index = n
        b_index = N + n

        # Fill in parts of the M matrix for a_n and b_n
        M[a_index, b_index] = 1  # Corresponds to b_n term in λa_n = -ina_n + b_
        M[a_index, a_index] = -1j * (n + n_min)  # Corresponds to -ina_n term

        # Fill parts for b_n equation
        M[b_index, b_index] = -1j * (n + n_min)  # Corresponds to -inb_n term
        M[b_index, a_index] = -delta  # Corresponds to -δa_n term

        # Corresponds to -εa_{n-1} and -εa_{n+1} terms
        if n > 0:
            M[b_index, a_index - 1] = -epsilon  # Corresponds to -εa_{n-1}
        if n < N - 1:
            M[b_index, a_index + 1] = -epsilon  # Corresponds to -εa_{n+1}
    print('Created M Matrix Successfully! Shape:', M.shape)
    return M

# Function to solve the eigenvalue problem and plot the spectrum
def solve_and_plot(n_min, n_max):
    # Construct matrix M
    M = construct_M(n_min, n_max)

    # Solve for eigenvalues
    eigenvalues= eigvals(M)

    # Plot using Matplotlib
    plt.figure(figsize=(10, 6))
    plt.scatter(eigenvalues.real, eigenvalues.imag, color='blue', s=5, alpha=0.7

    # Set plot labels and title
    plt.title(f"Eigenvalue Spectrum (n = {n_min} to {n_max})")
    plt.xlabel("Real Part")
    plt.ylabel("Imaginary Part")
    plt.grid(True, which='both', linestyle='--', linewidth=0.5)

    # Show plot
    plt.show()

# Test with n in [-20, 20]
solve_and_plot(-20, 20)

# Test with n in [-40, 40] to check for convergence
solve_and_plot(-40, 40)
```

```
Created M Matrix Successfully! Shape: (82, 82)
```




```
Created M Matrix Successfully! Shape: (162, 162)
```



- In eigenvalue problems, the real part of the eigenvalue λ determines the growth or decay of the solution.
- Our plotted eigenvalue spectrum contains eigenvalues λ with positive real parts 0.307844, and any small perturbation (i.e., non-zero solutions for $u(t)$ or $v(t)$) will grow exponentially with time, implying that the fixed point $u = 0$ is unstable.
- It is worth mentioning that some points on the boundary do not conform to this pattern. When a_{n-1} and a_{n+1} cannot be represented in the matrix M, resulting in fluctuation points.

Task(c)

(I)

At time $t = 0$, the system is typically in its initial state. In this state, the function $Q(t)$ is defined as the identity matrix I. This indicates that the system is in a normalized state at the initial moment, which facilitates calculations. Any state can be represented as a linear combination of the identity matrix.

Assuming $Q(t) = I$, we can derive that $Q(t) = EI = E$. This formulation provides a clearer representation of the state transition matrix.

(II)

Use **Python** code to compute the eigenvalues of matrix E and determine the stability of the system.

In [ ]:
```python
from scipy.integrate import solve_ivp

# Define the period T
T = 2 * np.pi

#Define the solver options
options = {'rtol': 1e-7, 'atol': 1e-8}

# Define the matrix A(t)
def A(t, delta, epsilon):
    return np.array([[0, 1],
                     [-(delta + 2 * epsilon * np.cos(t)), 0]])

# Convert the matrix differential equation to a vector differential equation
def odefun(t, Q_flat, delta, epsilon):
    Q = Q_flat.reshape(2, 2)  # Reshape the flattened Q back to 2x2 matrix
    dQ_dt = A(t, delta, epsilon) @ Q  # Compute dQ/dt = A(t) * Q
    return dQ_dt.flatten()  # Flatten the result to a 1D array for solve_ivp

# Initial condition: flattened form of the identity matrix
Q0 = np.eye(2).flatten()

# Use solve_ivp for numerical integration
options = {'rtol': 1e-7, 'atol': 1e-8}
solution = solve_ivp(odefun, [0, T], Q0, args=(delta, epsilon), method='RK45', *

# Extract Q(T) and convert it back to 2x2 matrix form
Q_T = solution.y[:, -1].reshape(2, 2)

# Output the monodromy matrix E = Q(T)
print("Monodromy matrix E:")
print(Q_T)

# Compute eigenvalues of E
eigenvalues = eigvals(Q_T)
print("Eigenvalues of E: " + str(eigenvalues))

# Check stability
```

```python
if all(abs(eigenvalue) < 1 for eigenvalue in eigenvalues):
    print("The system is stable.")
else:
    print("The system is unstable.")
```

```
Monodromy matrix E:
[[ 3.5316624   0.79013798]
 [14.51979231  3.5316624 ]]
Eigenvalues of E: [0.14453392 6.91879089]
The system is unstable.
```

I completed this part in MATLAB as well, and the results were consistent with those from Python.

```matlab
function main
    % Define the ODE function
    odefun = @(t, y) [0, 1; -(1 + 2*cos(t)), 0] * y;

    % Set the time span and initial conditions
    tspan = [0, 2*pi];
    y0_1 = [1; 0];
    y0_2 = [0; 1];

    % Set options for ode45 with specified tolerances
    options = odeset('AbsTol', 1e-7, 'RelTol', 1e-8);

    % Call ode45 to solve the ODE with specified options
    [T1, Y1] = ode45(odefun, tspan, y0_1, options);
    [T2, Y2] = ode45(odefun, tspan, y0_2, options);

    % Construct the monodromy matrix
    monodromy_matrix = [Y1(end, :)', Y2(end, :)'];

    % Calculate eigenvalues
    eigenvalues = eig(monodromy_matrix);

    % Calculate stability
    magnitudes = abs(eigenvalues);
    disp('Magnitudes of the eigenvalues:');
    disp(magnitudes)
    if any(magnitudes > 1)
        disp('The system is unstable');
    else
        disp('The system is stable');
    end

    % Calculate λ
    lambda = (1 / (2 * pi)) * log(eigenvalues);
    disp('Eigenvalues λ:');
    disp(lambda);
end
```

Result:

命令行窗口

```
>> task_C
Magnitudes of the eigenvalues:
    0.1445
    6.9188

The system is unstable
Eigenvalues λ:
   -0.3078
    0.3078
```

Notably, I set **options = odeset('AbsTol', 1e-7, 'RelTol', 1e-8)** as the tolerance convergence criteria.

In [ ]:
```
transformed_eigenvalues = (1/(np.pi*2)) * np.log(eigenvalues)
print(transformed_eigenvalues)
```

```
[-0.30784403  0.30784402]
```

The real parts of the eigenvalues are the same as those in the results of task (b). The proof is as follows:

Based on the results shown in the image, we obtained the magnitudes of the eigenvalues $|\mu|$ = 0.1445 and $|\mu|$ = 6.9188 from the monodromy matrix E in Task (c). Using the transformation formula

$$\lambda = \frac{1}{2\pi}\log|\mu|,$$

we can calculate the corresponding eigenvalues λ as follows.

For the first eigenvalue $|\mu|$ = 0.1445 , we have

$$\lambda_1 = \frac{1}{2\pi}\log(0.1445).$$

Calculating

$$\log(0.1445) \approx -1.933,$$

we find

$$\lambda_1 = \frac{1}{2\pi} \times (-1.933) \approx -0.3078.$$

For the second eigenvalue |λ| = 6.9188 , we calculate

$$\lambda_2 = \frac{1}{2\pi}\log(6.9188).$$

With

$$\log(6.9188) \approx 1.933,$$

we obtain

$$\lambda_2 = \frac{1}{2\pi} \times 1.933 \approx 0.3078.$$

The final results are

$$\lambda_1 \approx -0.3078$$

and

$$\lambda_2 \approx 0.3078.$$

These values match the results from Task (b), confirming that the methods used in Tasks (b) and (c) yield consistent eigenvalue forms. This alignment validates both approaches for analyzing the system's stability.

Tsak(d)

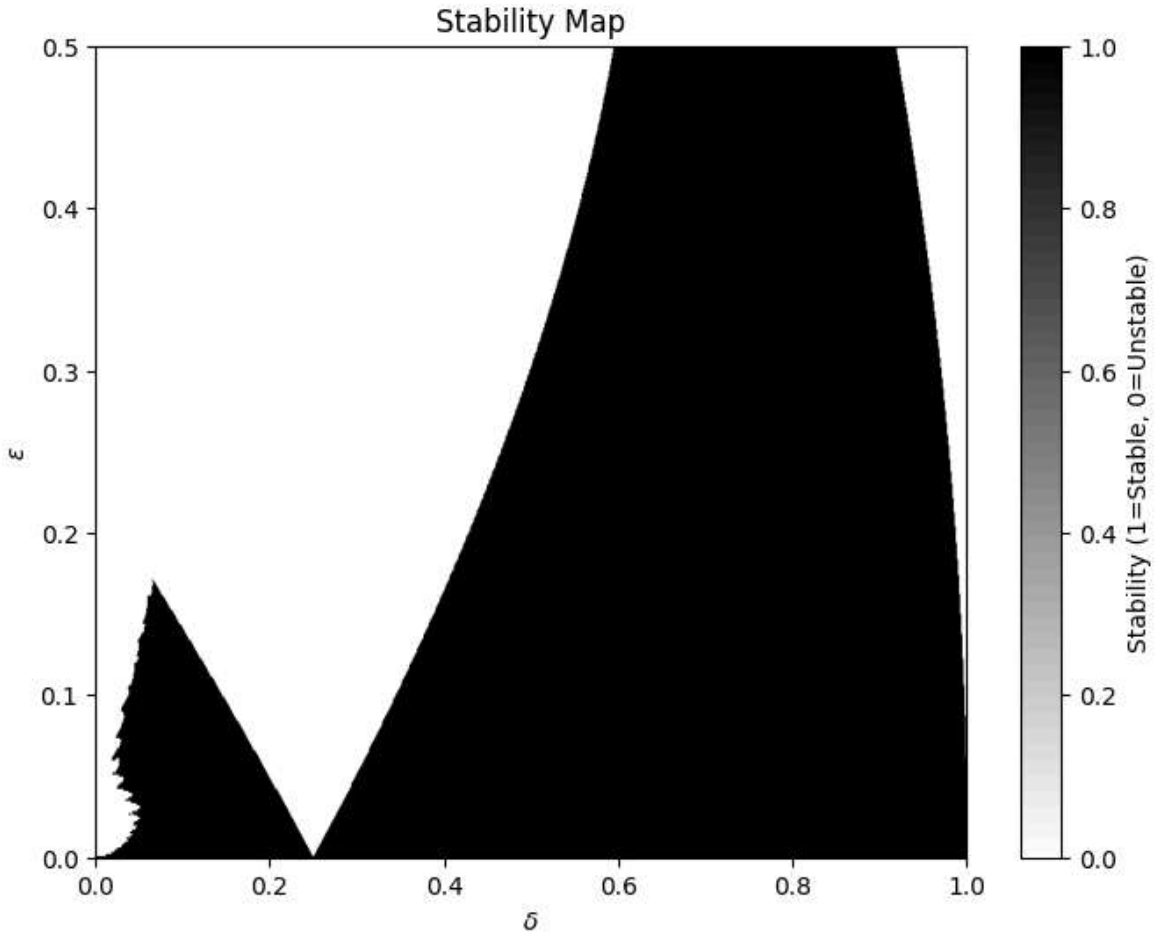
The code performs a parametric study of the stability of the system by varying δ and ϵ. The output is a visual representation of the stability regions in the δ-ϵ plane.

Regions where the system is stable are indicated by light color (corresponding to a gray value of 1 in the `stability_map`), while unstable regions are black.

The eigenvalue analysis is based on Floquet theory, where the system is stable if the magnitude of all eigenvalues of the monodromy matrix $Q(T)$ is less than 1.

In [ ]:
```python
# Define the range for delta and epsilon
delta_values = np.linspace(0, 1, 800)
epsilon_values = np.linspace(0, 0.5, 400)

# Initialize the stability matrix
stability_map = np.zeros((len(epsilon_values), len(delta_values)))

# Loop through the values of delta and epsilon
for i, epsilon in enumerate(epsilon_values):
    for j, delta in enumerate(delta_values):
        # Use solve_ivp for numerical integration
        solution = solve_ivp(
            odefun, [0, T], Q0, args=(delta, epsilon),
            method='RK45', **options
        )

        # Extract Q(T) and reshape it back to a 2x2 matrix
        Q_T = solution.y[:, -1].reshape(2, 2)

        # Calculate the eigenvalues of Q(T)
        eigenvalues = eigvals(Q_T)

        # If the maximum absolute value of the eigenvalues is less than 1,
        # consider the system stable
        if np.abs(eigenvalues).max() < 1:
            stability_map[i, j] = 1
```

```
# Plot the stability map
plt.figure(figsize=(8, 6))
plt.imshow(
    stability_map, extent=[delta_values[0], delta_values[-1],
                           epsilon_values[0], epsilon_values[-1]],
    origin='lower', aspect='auto', cmap='gray_r'
)

plt.colorbar(label='Stability (1=Stable, 0=Unstable)')
plt.xlabel(r'$\delta$')
plt.ylabel(r'$\epsilon$')
plt.title('Stability Map')
plt.show()
```

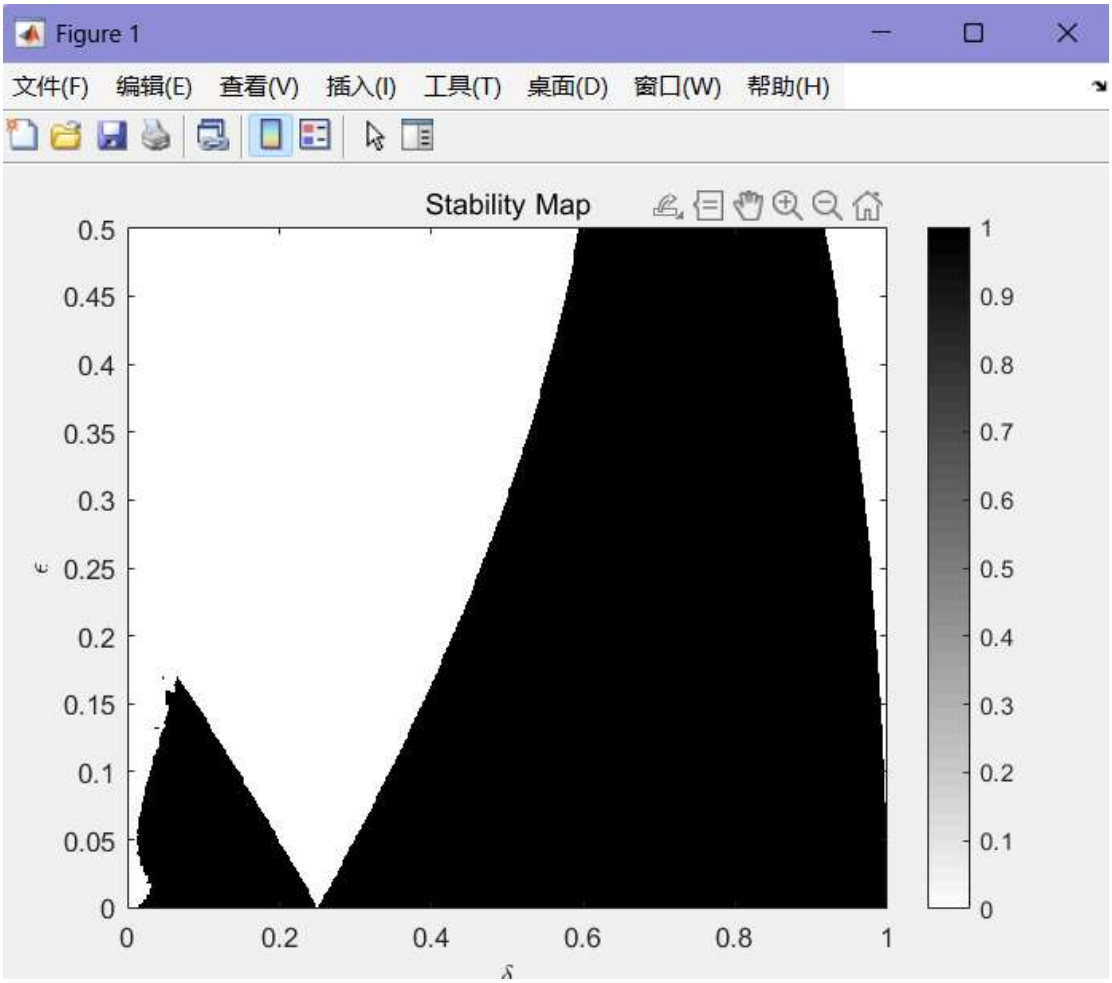


I completed this part in MATLAB as well, and the results were consistent with those from Python.

```
% Define parameter ranges
delta_values = linspace(0, 1, 200);     % delta from 0 to 1
epsilon_values = linspace(0, 0.5, 200); % epsilon from 0 to 0.5
T = 2 * pi;  % period T

% Set ODE solver options
options = odeset('AbsTol', 1e-7, 'RelTol', 1e-8);

% Initialize stability map matrix
stability_map = zeros(length(epsilon_values), length(delta_values));

% Define matrix A(t)
```

```
A = @(t, delta, epsilon) [0, 1; -(delta + 2 * epsilon * cos(t)), 0];

% Define differential equation function
odefun = @(t, Q_flat, delta, epsilon) ...
    reshape(A(t, delta, epsilon) * reshape(Q_flat, 2, 2), 4, 1);

% Initial condition: flattened identity matrix
Q0 = eye(2);
Q0_flat = Q0(:);

% Loop over epsilon and delta values
for i = 1:length(epsilon_values)
    epsilon = epsilon_values(i);
    for j = 1:length(delta_values)
        delta = delta_values(j);

        % Integrate using ode45 with specified options
        [~, Q_solution] = ode45(@(t, Q_flat) odefun(t, Q_flat, delta,
epsilon), [0 T], Q0_flat, options);

        % Extract Q(T) and reshape to 2x2 matrix
        Q_T = reshape(Q_solution(end, :)', 2, 2);

        % Compute eigenvalues of Q(T)
        eigenvalues = eig(Q_T);

        % Determine stability
        if max(abs(eigenvalues)) < 1
            stability_map(i, j) = 1; % Stable
        else
            stability_map(i, j) = 0; % Unstable
        end
    end
end

% Plot stability map with X and Y axes flipped
figure;
imagesc(delta_values, epsilon_values, stability_map);
set(gca, 'YDir', 'normal');
xlabel('\delta'); % X-axis is delta
ylabel('\epsilon'); % Y-axis is epsilon
title('Stability Map');
colorbar;
colormap(flipud(gray)); % Grayscale: black = unstable, white = stable
```

Results:

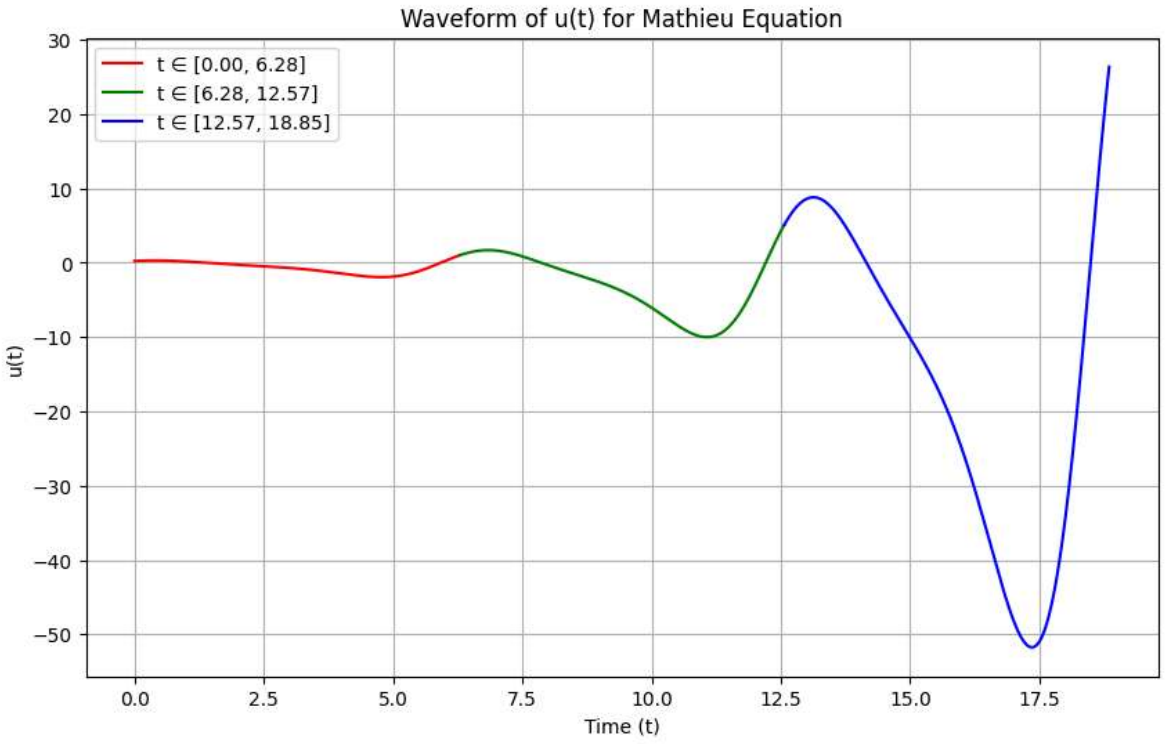


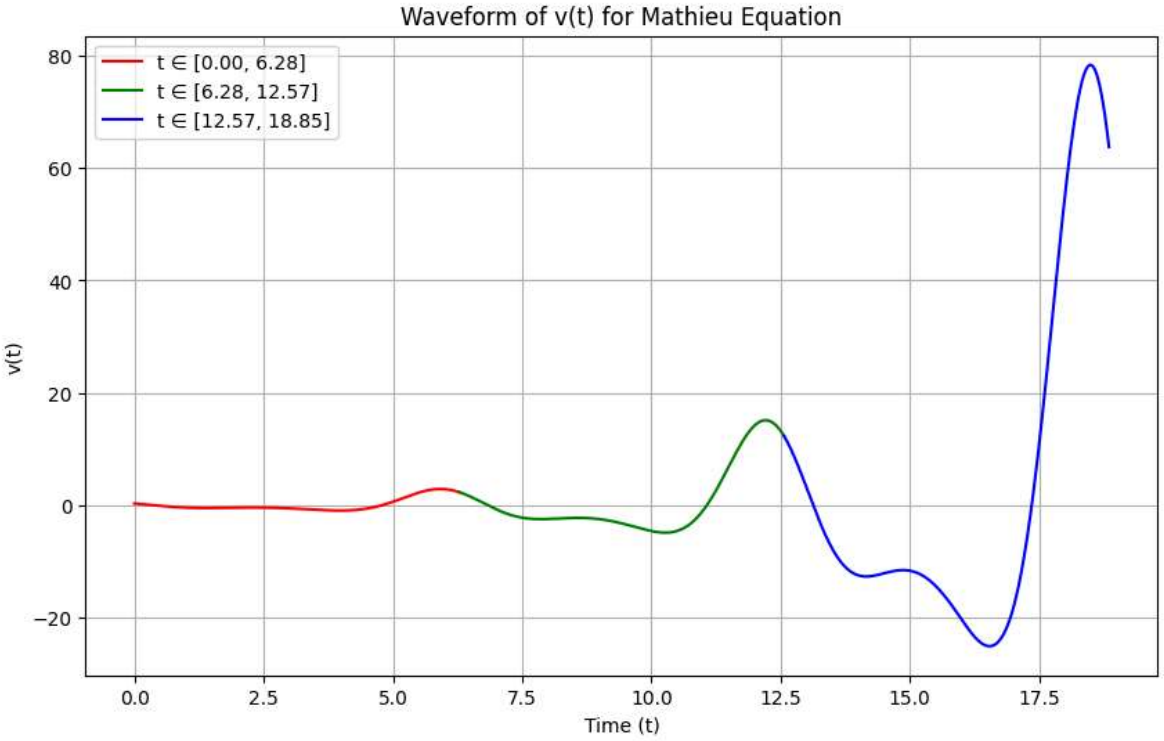
Tsak(e)

Based on the stability map, I select δ =1.2 and ϵ = 0.9 as the unstable point.

In [ ]:
```python
def plot_mathieu(delta=0.5, epsilon=0.5, t_start=0, t_end=6*np.pi, initial_q=[0.
    # Mathieu 方程
    def mathieu(t, q, delta=delta, epsilon=epsilon):
        dq_dt = A(t, delta, epsilon) @ q
        return dq_dt

    # 时间采样点
    t_eval = np.linspace(t_start, t_end, 10000)

    # 调用求解器
    t_span = (t_start, t_end)
    solution = solve_ivp(mathieu, t_span, initial_q, method='RK45', t_eval=t_eva

    # 提取解
    t_values = solution.t
    u_values = solution.y[0]
    v_values = solution.y[1]

    # 定义时间段
    time_intervals = [
        (0, 2 * np.pi),  # [0, 2π]
        (2 * np.pi, 4 * np.pi),  # [2π, 4π]
        (4 * np.pi, 6 * np.pi)   # [4π, 6π]
    ]
```

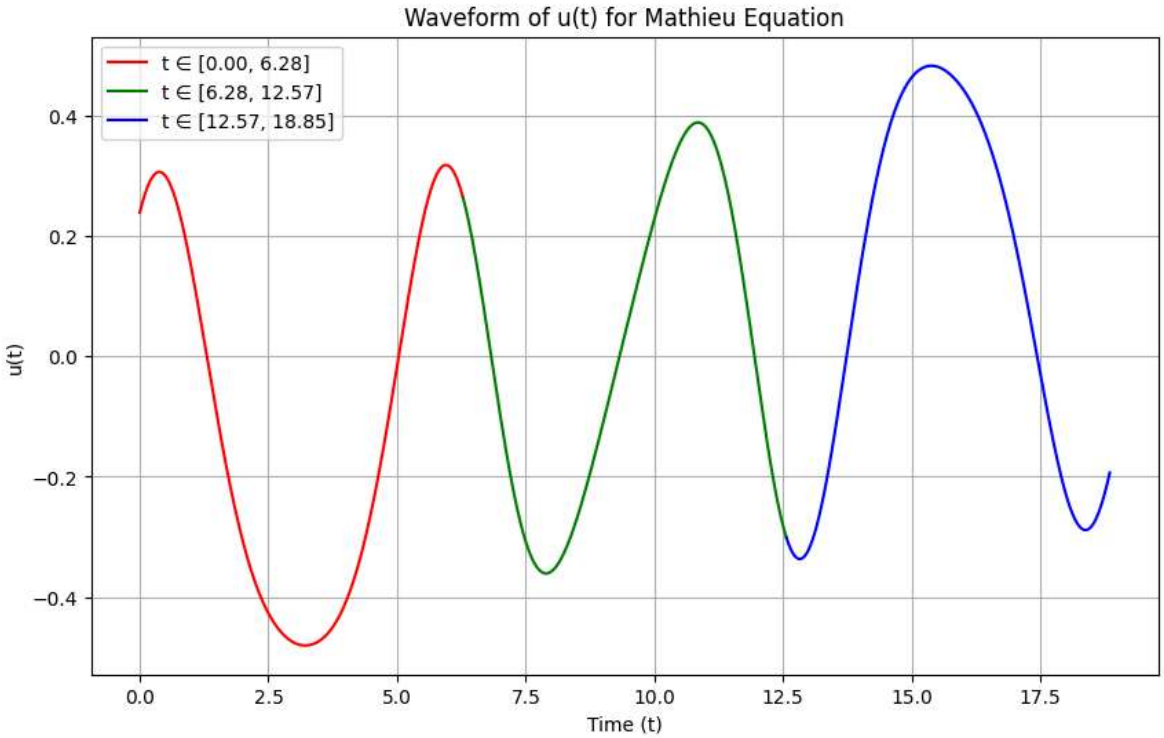
```
    # 颜色列表
    colors = ['red', 'green', 'blue']

    # 创建 u(t) 图形
    plt.figure(figsize=(10, 6))
    for interval, color in zip(time_intervals, colors):
        start, end = interval
        mask = (t_values >= start) & (t_values <= end)
        plt.plot(t_values[mask], u_values[mask], color=color, label=f't ∈ [{sta

    plt.title('Waveform of u(t) for Mathieu Equation')
    plt.xlabel('Time (t)')
    plt.ylabel('u(t)')
    plt.grid(True)
    plt.legend()
    plt.show()

    # 创建 v(t) 图形
    plt.figure(figsize=(10, 6))
    for interval, color in zip(time_intervals, colors):
        start, end = interval
        mask = (t_values >= start) & (t_values <= end)
        plt.plot(t_values[mask], v_values[mask], color=color, label=f't ∈ [{sta

    plt.title('Waveform of v(t) for Mathieu Equation')
    plt.xlabel('Time (t)')
    plt.ylabel('v(t)')
    plt.grid(True)
    plt.legend()
    plt.show()

# 调用函数
plot_mathieu(delta=1.2, epsilon=0.9, t_start=0, t_end=6*np.pi, initial_q=[0.2392
```

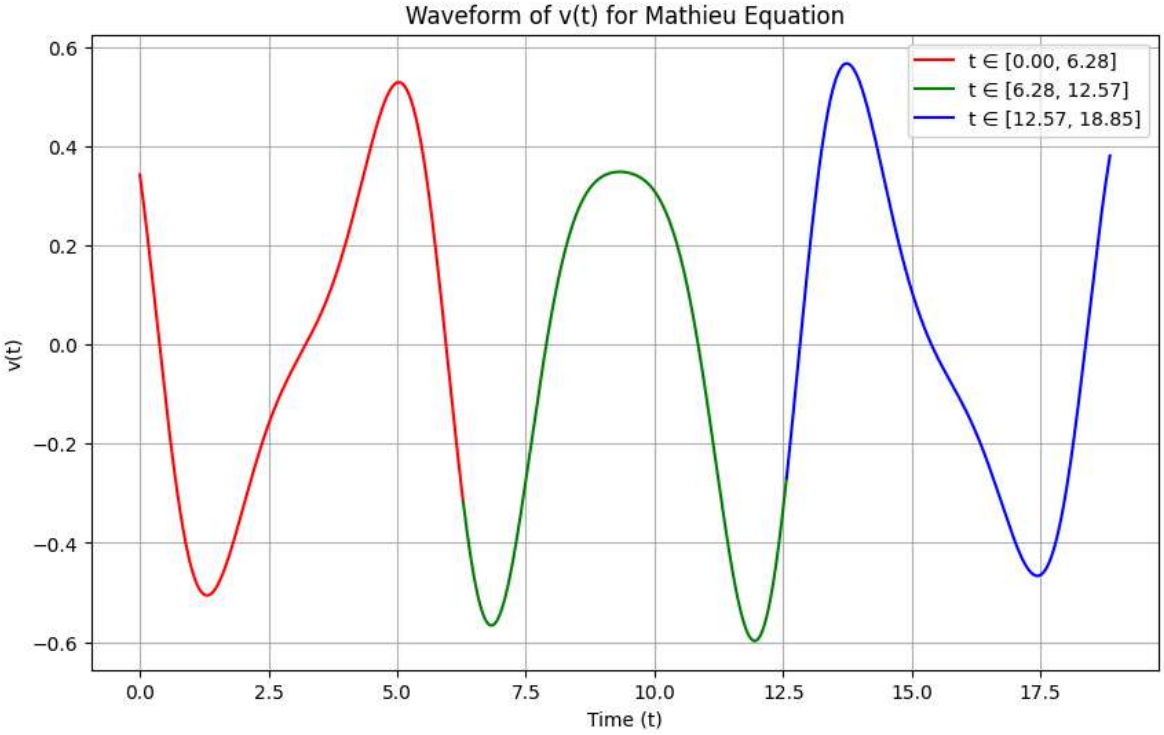




I select δ = 1.8 and ϵ = 0.7 as the stable point.

```
In [36]: plot_mathieu(delta=1.8, epsilon=0.7, t_start=0, t_end=6*np.pi, initial_q=[0.2392
```





Conclusion

For an unstable point, plotting U against time reveals oscillations that grow within each cycle. Stable systems, however, maintain constant amplitude oscillations of U over each period.