Numerical Analysis Project 2

Group 5

Problem Setup

Modeling a Car Suspension System

A car suspension acts as a spring-mass-dashpot system on each wheel. This allows the car to stay on the road and enable good handling of the car under normal driving conditions. In this problem, we examine a "quarter car" model that involves a second order differential equation for one wheel.

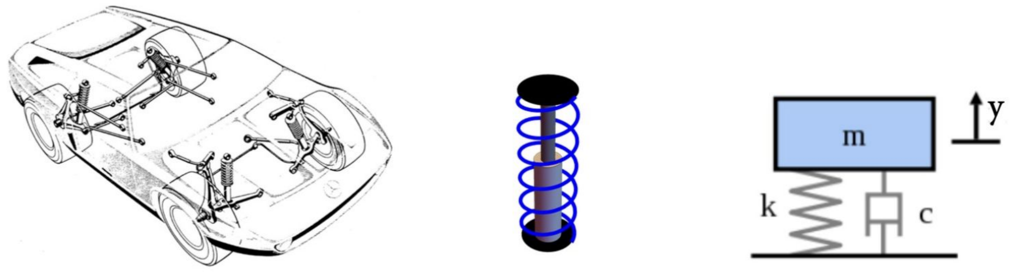


Figure 1. (a) Suspension (b) Spring Assembly (c) Diagram

Problem Setup

$My''(t) + cy'(t) + ky(t) = 0$, $y(0) = y_0$, $y'(0) = 0$

Parameters:

- $M = 450$ kg: Mass on one wheel
- $k = 16{,}000$ N/m: Spring constant (restoring force)
- $c = 1{,}600$ Ns/m: Damping coefficient (shock absorber)
- Initial position: $y_0 = 0.2$ m, Initial velocity: $y'(0) = 0$, $t_0 = 0$

Question A Goals:

- Explore what happens with and without a shock absorber.
- Study how adding mass (like loading the car) impacts the motion.
- Find how long it takes for the car to return to within 0.1 m of equilibrium.
- Use graphs to illustrate results and compare ride smoothness.

Question A Big Question:

Why does a suspension with a shock absorber give a smoother ride?

Intro Analytical Solution

- This entire project is 100% solvable analytically given the specific initial conditions for each problem. The reason for this is that all of these differential equations are linear and not chaotic. Although the setup for part B is coupled, it is still linear.
- The method for solving analytically is by auxiliary equation for part A.
- For solving analytically in part B a system of equations must be setup which can be solved by Laplace Transform or other methods
- Part B is very tedious to solve by hand, so by using a MatLab ODE solver, it is made much more simple
- Even with the solver, the general solution is very complex
- By using MatLab we can easily calculate the eigenvalues of our solutions, the settling time for each case, as well as a displacement vs. time graph

```
% Parameters
m = 450;        % mass
c = 0;%1600;    % damping coefficient
k = 16000;      % stiffness

% External force function F(t)
F = @(t) 0;%1.0 * sin(2 * t);

% Time span
tspan = [0 10];

% Initial conditions: [position, velocity]
x0 = [.2; 0];  % x(0) = 0, x'(0) = 0

% ODE system
odefunc = @(t, x) [x(2); (1/m)*(F(t) - c*x(2) - k*x(1))];

% Solve ODE
[t, X] = ode45(odefunc, tspan, x0);

% Extract position
position = X(:,1);

% Plot position vs time
figure;
plot(t, position, 'b', 'LineWidth', 2);
xlabel('Time (s)');
ylabel('Position (m)');
title('Position vs. Time');
grid on;
```

1 Equation ODE Solver Analytically

```matlab
% Parameters
m = 450;        % mass
c = 0;%1600;     % damping coefficient
k = 16000;       % stiffness

% External force function F(t)
F = @(t) 0;%1.0 * sin(2 * t);

% Time span
tspan = [0 10];

% Initial conditions: [position, velocity]
x0 = [.2; 0];  % x(0) = 0, x'(0) = 0

% ODE system
odefunc = @(t, x) [x(2); (1/m)*(F(t) - c*x(2) - k*x(1))];

% Solve ODE
[t, X] = ode45(odefunc, tspan, x0);

% Extract position
position = X(:,1);

% Plot position vs time
figure;
plot(t, position, 'b', 'LineWidth', 2);
xlabel('Time (s)');
ylabel('Position (m)');
title('Position vs. Time');
grid on;
```

```matlab
  t_cross = zeros(size(idx));

  % Loop and linearly interpolate for each crossing
  for kx = 1:length(idx)
      i = idx(kx);
      if i < length(t)
          t1 = t(i);    x1 = abs(x(i));
          t2 = t(i+1);  x2 = abs(x(i+1));
          if x1 == thr
              t_cross(kx) = t1;
          else
              t_cross(kx) = t1 + (thr - x1)*(t2 - t1)/(x2 - x1);
          end
      else
          t_cross(kx) = t(end);
      end
  end
  t_cross = t_cross(~isnan(t_cross));

  % Compute actual x-values at those times
  x_cross = interp1(t, x, t_cross);

% Settling threshold relative to U(t)
x = X(:,1);     % position

% Threshold band ±δ
thr = 0.1;

% Find indices where abs(x) crosses the threshold
idx = find( (abs(x) - thr) .* ([abs(x(2:end)); NaN] - thr) <= 0 );
```

What happens without a damping coefficient?

$$My'' + ky = 0$$

$$Mr^2 + kr = 0$$

$$r = \frac{\pm\sqrt{-4Mk}}{2M} = \pm i\sqrt{\frac{k}{M}}$$

$$y(t) = C_1 e^{i\sqrt{\frac{k}{M}}} + C_2 e^{-i\sqrt{\frac{k}{M}}}$$

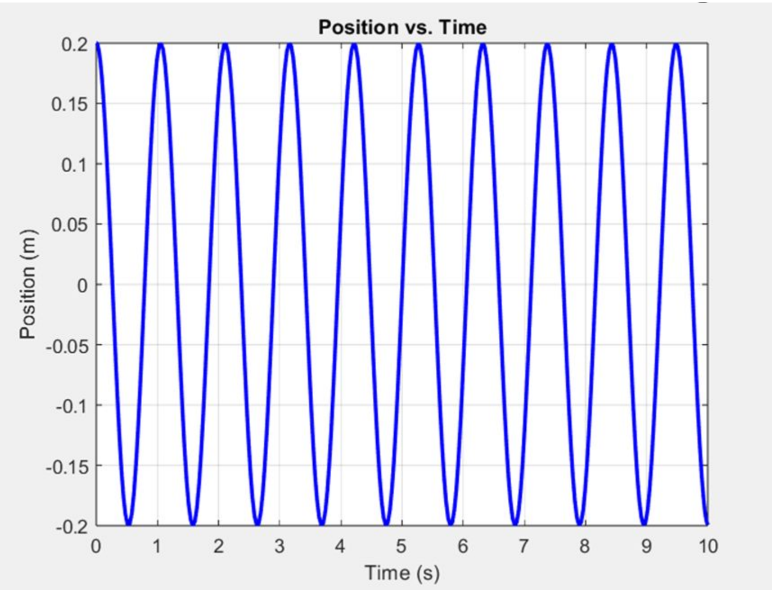
$$y(0) = y_0 = C_1 + C_2$$

$$y'(0) = 0 = i\sqrt{\frac{k}{M}}C_1 - i\sqrt{\frac{k}{M}}C_2$$

$$\therefore C_1 = C_2$$

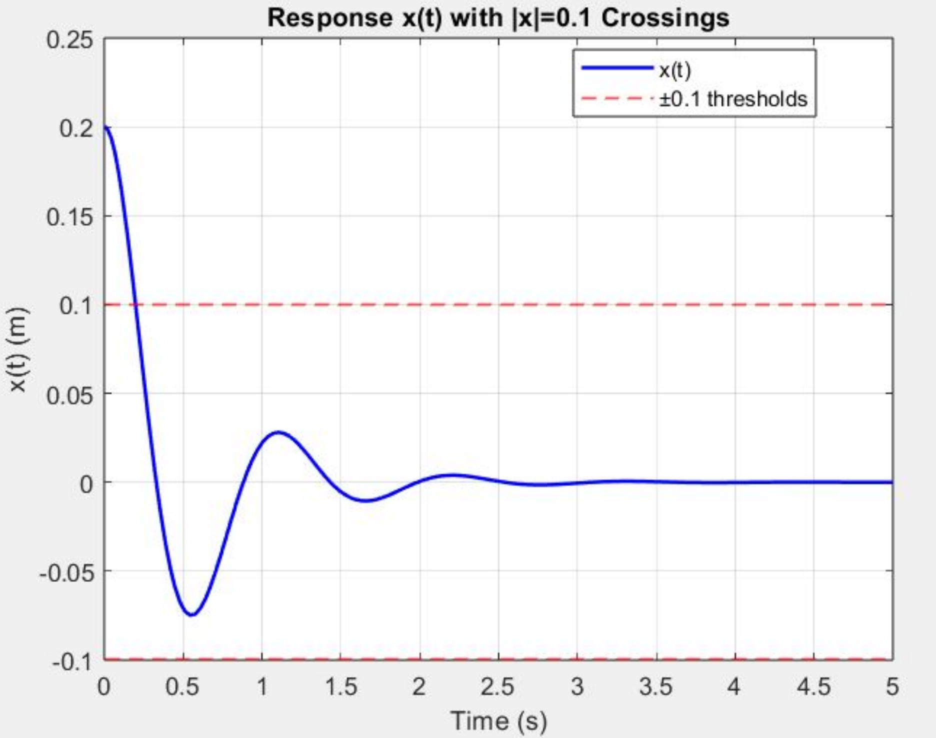
$$C_1 = C_2 = \frac{y_0}{2}$$

$$\therefore y(t) = \frac{y_0}{2}e^{i\sqrt{\frac{k}{M}}} + \frac{y_0}{2}e^{-i\sqrt{\frac{k}{M}}}$$

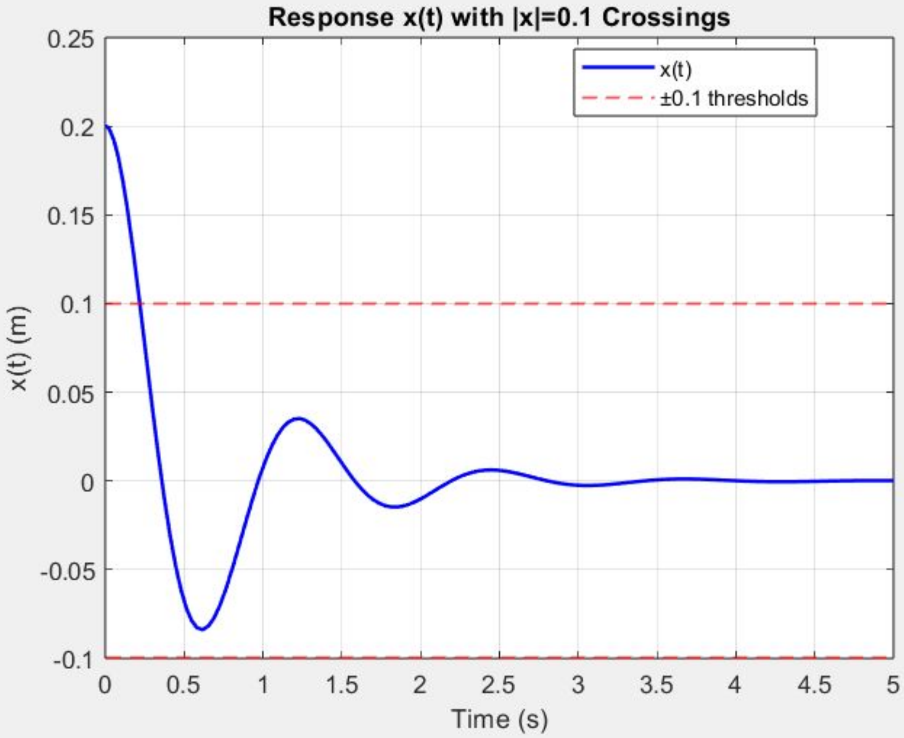


Result: Sinusoidal motion with no decay, so our tire would bounce up and down for eternity

Analytical Solution Part A



Settling Time: 0.198s (450 kgs)

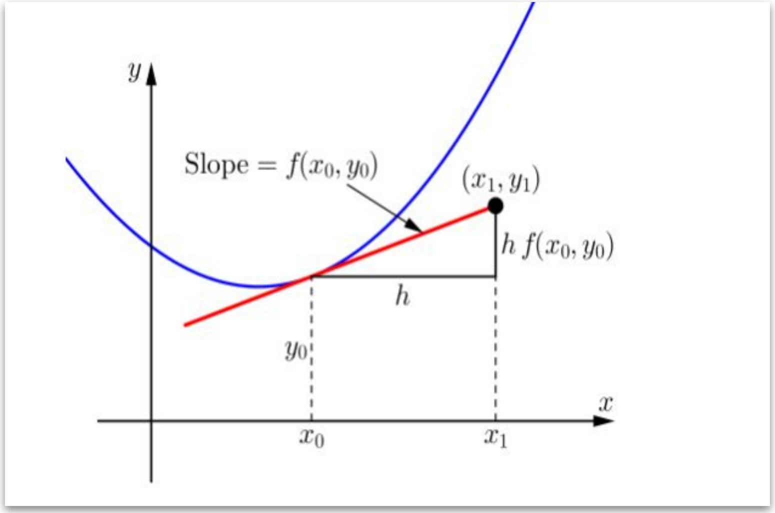


Settling Time: 0.219s (increase by 1000 lbs)

Numerical Methods

Often, the exact solution to the IVP cannot be solved analytically, so we must approximate it.
In Part B, the system becomes nontrivial.

Using the ODE for part A, we can build the approximate solution by stepping forward in time and using derivative information to estimate the function's behavior.



Selected Methods:

- Euler's
- RK-4

Both from class, however with different error bounds.

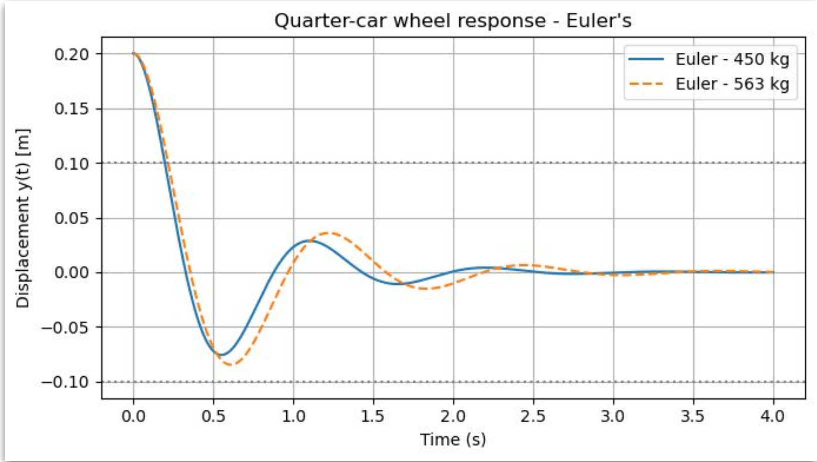
Euler's Method

Using the ODE from Question A, we chose to use Euler's and RK-4 to approximate the solution in preparation for use in Question B.

Euler's Method:

- H = 0.001
- Simple to implement and computationally efficient
- Can lose accuracy with larger step sizes
- Implementation Run Time: 0.4 seconds

$$y_{i+1} = y_i + hf(t_i, y_i).$$




```
Euler (simulated) settling time (±0.1 m):
  450 kg  =  0.199 s
  563 kg  =  0.219 s
```

Euler’s Method – Implementation

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

# Parameters
k, c = 16_000.0, 1_600.0
M_base = 450.0
M_heavy = 450.0 + 113.4
y0, v0 = 0.20, 0.0
t_end, h = 4.0, 0.001
tol = 0.10

def accel(y, v, M):
    return -(c*v + k*y) / M

def euler_step(y, v, h, M):
    y_next = y + h*v
    v_next = v + h*accel(y, v, M)
    return y_next, v_next

def simulate(M):
    n = int(np.ceil(t_end / h))
    t = np.linspace(0, n*h, n+1)
    y = np.empty_like(t); v = np.empty_like(t)
    y[0], v[0] = y0, v0
    for i in range(1, n+1):
        y[i], v[i] = euler_step(y[i-1], v[i-1], h, M)
    return t, y

def settling_time(M, y0=0.20, band=0.10):
    alpha = c / (2 * M)
    return np.log(y0 / band) / alpha
```

```python

#Sims
t_b, y_b = simulate(M_base)
t_h, y_h = simulate(M_heavy)

#settling times
t_env_b = settling_time(M_base, y0, tol)
t_env_h = settling_time(M_heavy, y0, tol)

print(f"Euler settling time (±{tol} m):")
print(f"  450 kg  =  {t_env_b:.3f} s")
print(f"  563 kg  =  {t_env_h:.3f} s")

#Plots
plt.figure(figsize=(7, 4))
plt.plot(t_b, y_b, label="Euler - 450 kg")
plt.plot(t_h, y_h, "--", label="Euler - 563 kg")
plt.axhline( tol, color="gray", linestyle=":")
plt.axhline(-tol, color="gray", linestyle=":")
plt.xlabel("Time (s)")
plt.ylabel("Displacement y(t) [m]")
plt.title("Quarter-car wheel response - Euler's")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

✓ 0.1s

Runge-Kutta 4

Runge-Kutta 4

- H = 0.001
- Improves accuracy by sampling multiple slope estimates per step
- Implementation Run Time: 0.5 seconds

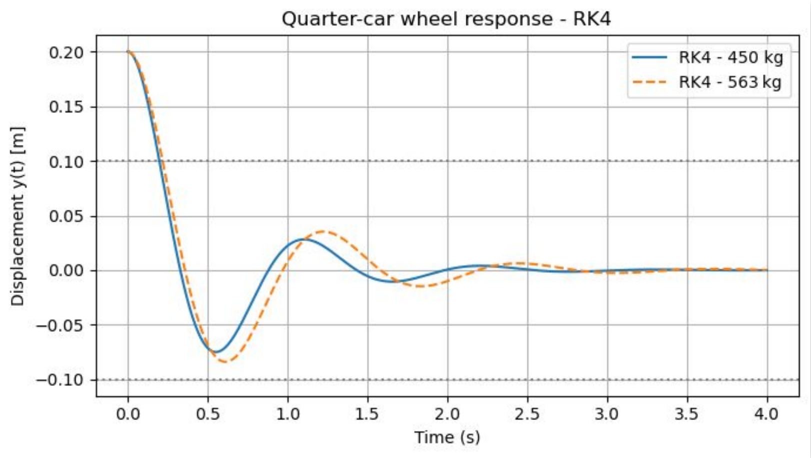
$$y_{i+1} = y_i + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \quad i = 0, 1, \cdots, N-1$$

$$\text{where} \quad k_1 = hf\left(t_i, y_i\right)$$

$$k_2 = hf\left(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_1\right)$$

$$k_3 = hf\left(t_i + \frac{1}{2}h, y_i + \frac{1}{2}k_2\right)$$

$$k_4 = hf\left(t_i + h, y_i + k_3\right).$$




```
RK4 (simulated) settling time (±0.1 m):
  450 kg  =  0.199 s
  563 kg  =  0.219 s
```

RK-4 - Implementation

```python
# Parameters
k, c = 16_000.0, 1_600.0
M_base = 450.0
M_heavy = 450.0 + 113.4
y0, v0 = 0.20, 0.0
t_end, h = 4.0, 0.001
tol = 0.10

def accel(y, v, M):
    return -(c*v + k*y) / M

def rk4_step(y, v, h, M):
    k1y, k1v = v, accel(y, v, M)
    k2y, k2v = v + 0.5*h*k1v, accel(y + 0.5*h*k1y, v + 0.5*h*k1v, M)
    k3y, k3v = v + 0.5*h*k2v, accel(y + 0.5*h*k2y, v + 0.5*h*k2v, M)
    k4y, k4v = v + h*k3v, accel(y + h*k3y,v + h*k3v,M)
    y_next = y + (h/6)*(k1y + 2*k2y + 2*k3y + k4y)
    v_next = v + (h/6)*(k1v + 2*k2v + 2*k3v + k4v)
    return y_next, v_next

def simulate(M):
    n = int(np.ceil(t_end / h))
    t = np.linspace(0, n*h, n+1)
    y = np.empty_like(t); v = np.empty_like(t)
    y[0], v[0] = y0, v0
    for i in range(1, n+1):
        y[i], v[i] = rk4_step(y[i-1], v[i-1], h, M)
    return t, y

def settling_time(M, y0=0.20, band=0.10):
    alpha = c / (2 * M)
    return np.log(y0 / band) / alpha

```

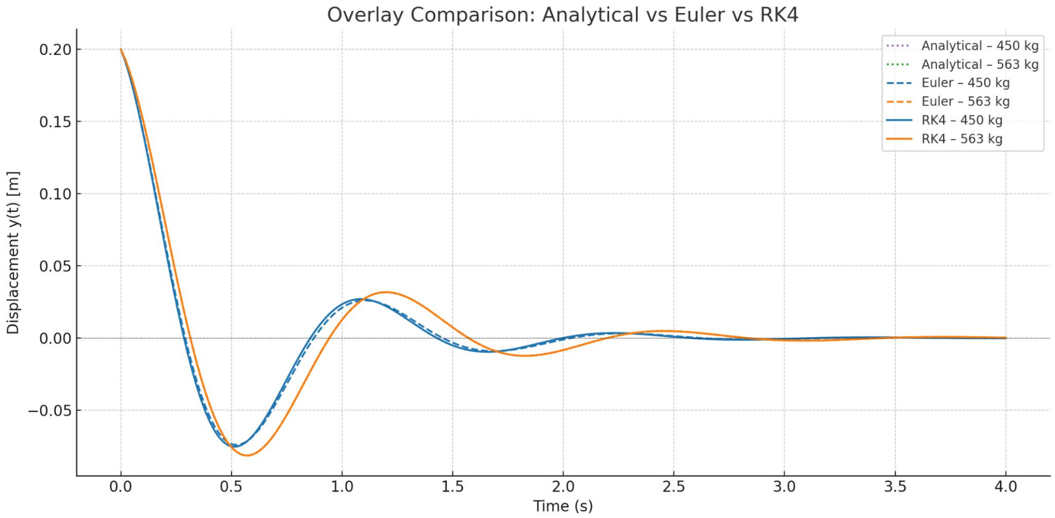
```python
# Sims
t_b, y_b = simulate(M_base)
t_h, y_h = simulate(M_heavy)

# Setttling time
t_env_b = settling_time(M_base, y0, tol)
t_env_h = settling_time(M_heavy, y0, tol)

print(f"RK4 settling time (±{tol} m):")
print(f"  450 kg  =  {t_env_b:.3f} s")
print(f"  563 kg  =  {t_env_h:.3f} s")

# Plots
plt.figure(figsize=(7, 4))
plt.plot(t_b, y_b, label="RK4 - 450 kg")
plt.plot(t_h, y_h, "--", label="RK4 - 563 kg")
plt.axhline( tol, color="gray", linestyle=":")
plt.axhline(-tol, color="gray", linestyle=":")
plt.xlabel("Time (s)")
plt.ylabel("Displacement y(t) [m]")
plt.title("Quarter-car wheel response - RK4")
plt.grid(True)
plt.legend()
plt.tight_layout()
plt.show()
```

✓ 0.2s

Error Analysis

Analytical solution vs. Euler's and RK-4

- Analytical Solution (exact)
- Runge-Kutta 4 - O(h^4) GTE
- Euler's Method - O(h) GTE



All methods begin with similar behavior.

RK4 and Euler track the analytical solution closely.

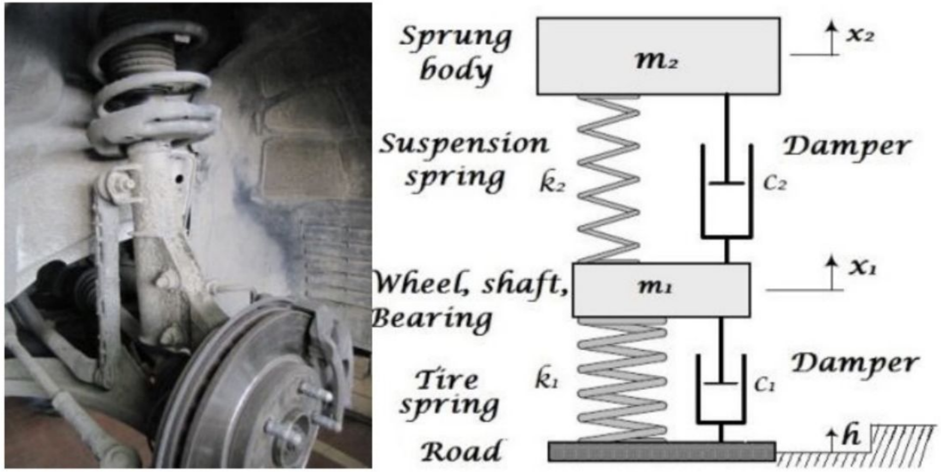
Increased mass (563 kg) shows slower damping and slightly longer oscillation.

Difference in solid orange line (RK-4 563 kg) could be due to the increased mass at 563 kg making the system more oscillatory and sensitive to error.

Small changes in the RK4- slope estimates may show up more clearly in the response curve.

Question B Introduction - Quarter Car Suspension

Now we consider the case when the tire acts as a second spring.



(a) (b)

Question B Introduction cont.

$$m_1 x_1'' = -k_1(x_1 - U) + k_2(x_2 - x_1) + c(x_2' - x_1'),$$
$$m_2 x_2'' = -k_2(x_2 - x_1) - c(x_2' - x_1').$$

System Parameters:

- m_1=18 kg: unsprung mass (wheel and tire)
- m_2=450 kg: sprung mass (car body)
- Various k_1, k_2, c values to model four different cases; U denotes road disturbance
- Initial positions: x_2(0) = 0.3 m, x_1(0) = 0.29 m
- Time interval: 0≤t≤25 s

Objective: Simulate the response of the suspension to different road conditions and compare ride quality across parameter sets.

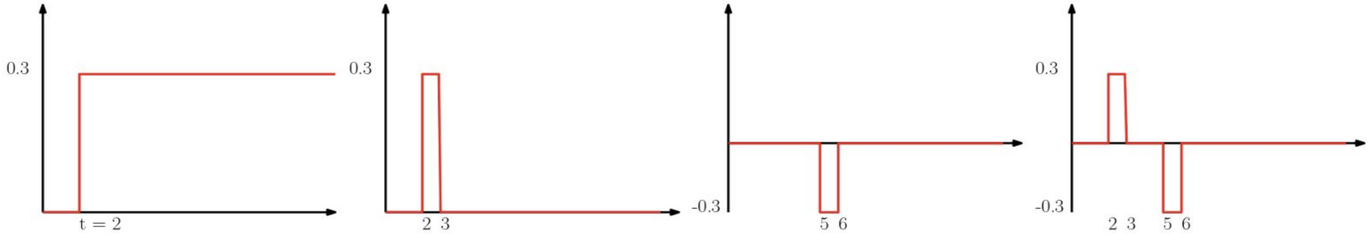


FIGURE 1. $U(t)$ for different road cases.

Code

```python
def rk4_system_two_second_order(f1, f2, y1_0, y1p_0, y2_0, y2p_0, t0, t
    """
    Solves a system of two second-order ODEs using RK4.

    Arguments:
        f1 : function for y1'' = f1(t, y1, y1', y2, y2')
        f2 : function for y2'' = f2(t, y1, y1', y2, y2')
        y1_0 : initial condition for y1
        y1p_0 : initial condition for y1'
        y2_0 : initial condition for y2
        y2p_0 : initial condition for y2'
        t0 : initial time
        tf : final time
        h : step size

    Returns:
        t_vals, y1_vals, y2_vals: time and solutions for y1 and y2
```

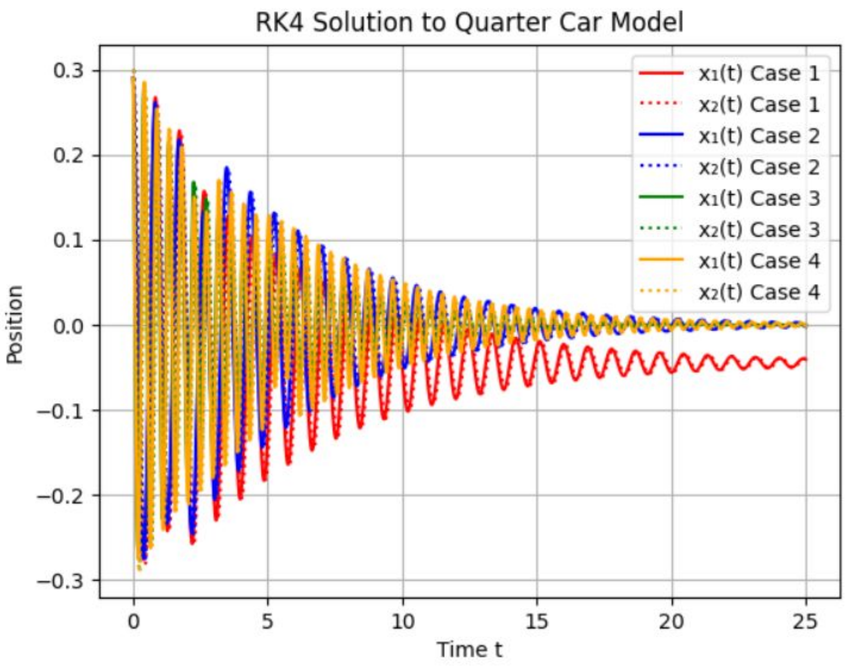
```python
n = int((tf - t0) / h)
t_vals = np.zeros(n+1)
y1_vals = np.zeros(n+1)
y2_vals = np.zeros(n+1)

# Initialize variables
z1, z2 = y1_0, y1p_0  # y1 and y1'
z3, z4 = y2_0, y2p_0  # y2 and y2'

y1_vals[0] = z1
y2_vals[0] = z3
t_vals[0] = t0

for i in range(n):
    t = t_vals[i]

    # Compute all k's for the 4 components (z1 to z4)
    k11 = h * z2
    k12 = h * f1(t, z1, z2, z3, z4)
    k13 = h * z4
    k14 = h * f2(t, z1, z2, z3, z4)

    k21 = h * (z2 + 0.5 * k12)
    k22 = h * f1(t + 0.5 * h, z1 + 0.5 * k11, z2 + 0.5 * k12, z3 + 0.5 * k13, z4 + 0.5 * k14)
    k23 = h * (z4 + 0.5 * k14)
    k24 = h * f2(t + 0.5 * h, z1 + 0.5 * k11, z2 + 0.5 * k12, z3 + 0.5 * k13, z4 + 0.5 * k14)

    k31 = h * (z2 + 0.5 * k22)
    k32 = h * f1(t + 0.5 * h, z1 + 0.5 * k21, z2 + 0.5 * k22, z3 + 0.5 * k23, z4 + 0.5 * k24)
    k33 = h * (z4 + 0.5 * k24)
    k34 = h * f2(t + 0.5 * h, z1 + 0.5 * k21, z2 + 0.5 * k22, z3 + 0.5 * k23, z4 + 0.5 * k24)

    k41 = h * (z2 + k32)
    k42 = h * f1(t + h, z1 + k31, z2 + k32, z3 + k33, z4 + k34)
    k43 = h * (z4 + k34)
    k44 = h * f2(t + h, z1 + k31, z2 + k32, z3 + k33, z4 + k34)

    # Update each component
    z1 += (k11 + 2*k21 + 2*k31 + k41) / 6
    z2 += (k12 + 2*k22 + 2*k32 + k42) / 6
    z3 += (k13 + 2*k23 + 2*k33 + k43) / 6
    z4 += (k14 + 2*k24 + 2*k34 + k44) / 6
```

Code Cont.

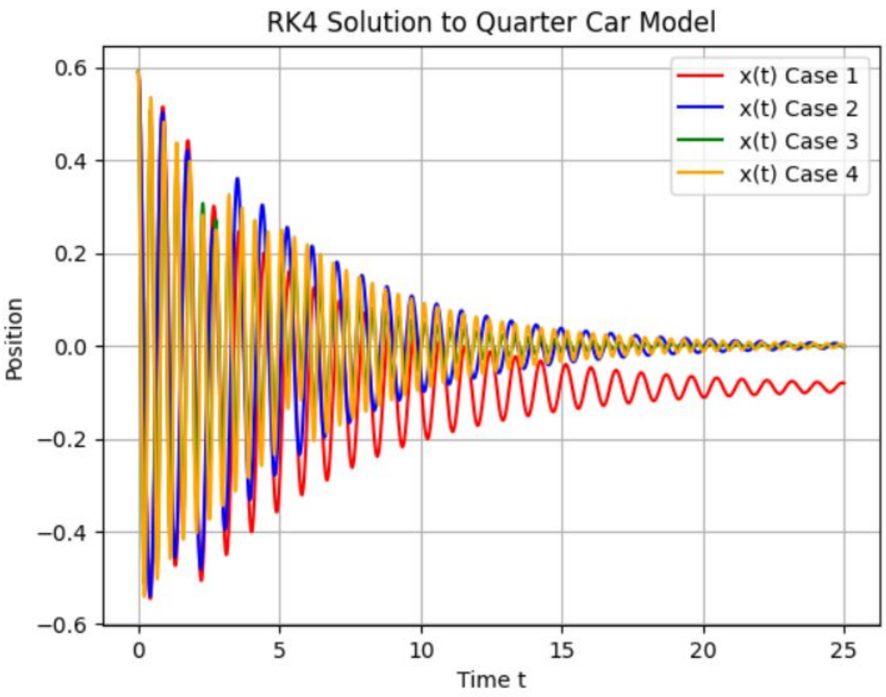
```
# Our equations are:
# m1 x1'' = -c x1' + c x2' - (k1 + k2) x1 + k2 x2 - k1 U
# m2 x2'' = c x1' - c x2' + k2 x1 - k2 x2

# Given values
m1 = 18
m2 = 450

# Initial conditions
x1_0, x1p_0 = 0.29, 0
x2_0, x2p_0 = 0.3, 0
t0, tf, h = 0, 25, 0.001

# Cases
k11, k21, c1 = 3500, 10000, 7300
k12, k22, c2 = 4000, 10000, 7300
k13, k23, c3 = 8000, 40000, 7300
k14, k24, c4 = 8000, 40000, 10000

# U functions
def u1(t):
  if (t < 2):
    return 0
  else:
    return 0.3
def u2(t):
  if (t > 2 and t < 3):
    return 0.3
  else:
    return 0
def u3(t):
```

```
# Define f1 and f2 for each case
def f11(t, x1, x1p, x2, x2p):
    return (-c1 * x1p + c1 * x2p - (k11 + k21) * x1 - k21 * x2 - k11 * u1(t))/m1

def f21(t, x1, x1p, x2, x2p):
    return (c1 * x1p - c1 * x2p + k21 * x1 - k21 * x2)/m2

def f12(t, x1, x1p, x2, x2p):
  return (-c2 * x1p + c2 * x2p - (k12 + k22) * x1 - k22 * x2 - k12 * u2(t))/m1

def f22(t, x1, x1p, x2, x2p):
  return (c2 * x1p - c2 * x2p + k22 * x1 - k22 * x2)/m2

def f13(t, x1, x1p, x2, x2p):
  return (-c3 * x1p + c3 * x2p - (k13 + k23) * x1 - k23 * x2 - k13 * u3(t))/m1

def f23(t, x1, x1p, x2, x2p):
  return (c3 * x1p - c3 * x2p + k23 * x1 - k23 * x2)/m2

def f14(t, x1, x1p, x2, x2p):
  return (-c4 * x1p + c4 * x2p - (k14 + k24) * x1 - k24 * x2 - k14 * u4(t))/m1

def f24(t, x1, x1p, x2, x2p):
  return (c4 * x1p - c4 * x2p + k24 * x1 - k24 * x2)/m2


# Solve
t1, x11, x21 = rk4_system_two_second_order(f11, f21, x1_0, x1p_0, x2_0, x2p_0, t0, tf, h)
t2, x12, x22 = rk4_system_two_second_order(f12, f22, x1_0, x1p_0, x2_0, x2p_0, t0, tf, h)
t3, x13, x23 = rk4_system_two_second_order(f13, f23, x1_0, x1p_0, x2_0, x2p_0, t0, tf, h)
t4, x14, x24 = rk4_system_two_second_order(f14, f24, x1_0, x1p_0, x2_0, x2p_0, t0, tf, h)
```

Results



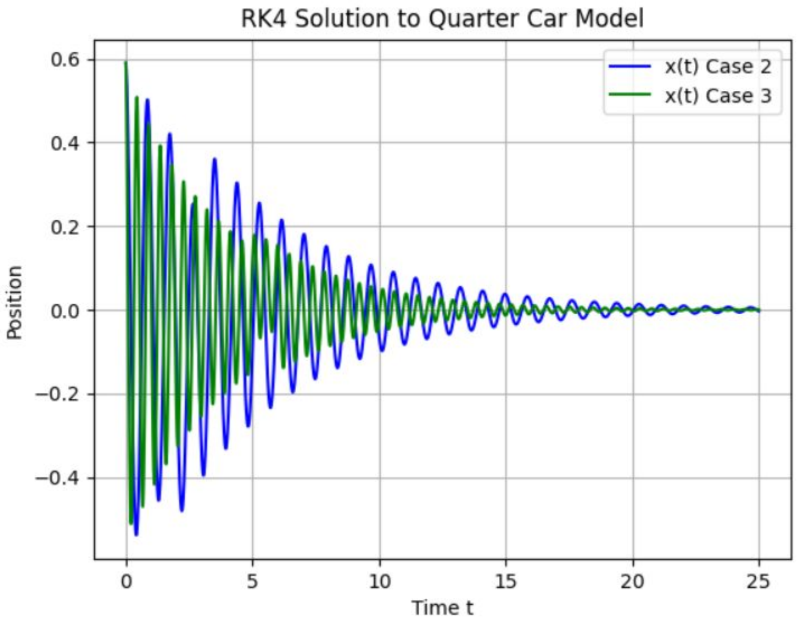
RK4 Solution to Quarter Car Model
$x_1(t)$ Case 1
$x_2(t)$ Case 1
$x_1(t)$ Case 2
$x_2(t)$ Case 2
$x_1(t)$ Case 3
$x_2(t)$ Case 3
$x_1(t)$ Case 4
$x_2(t)$ Case 4
Position
Time t



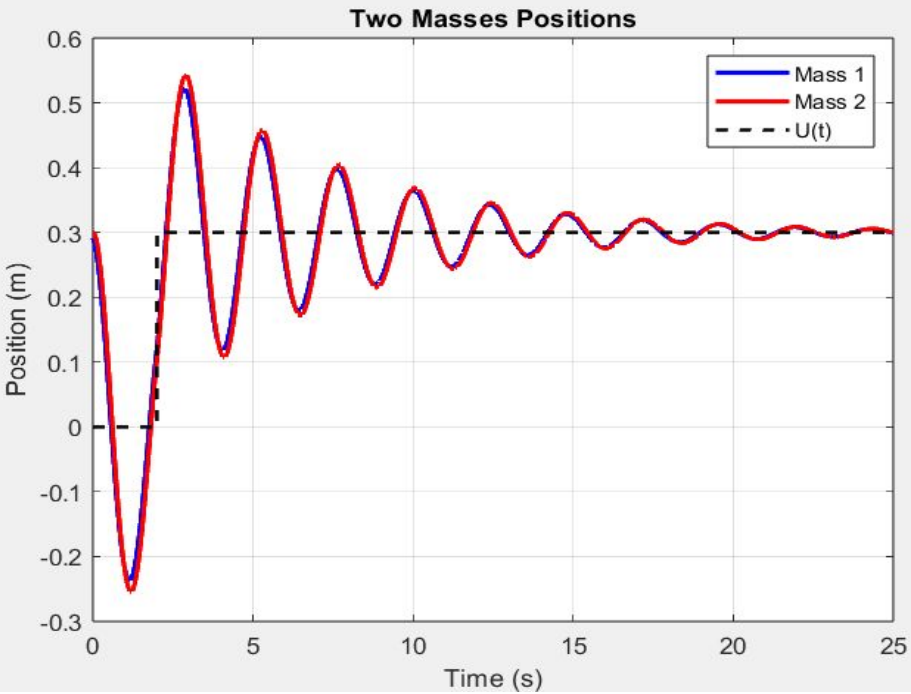
RK4 Solution to Quarter Car Model
x(t) Case 1
x(t) Case 2
x(t) Case 3
x(t) Case 4
Position
Time t

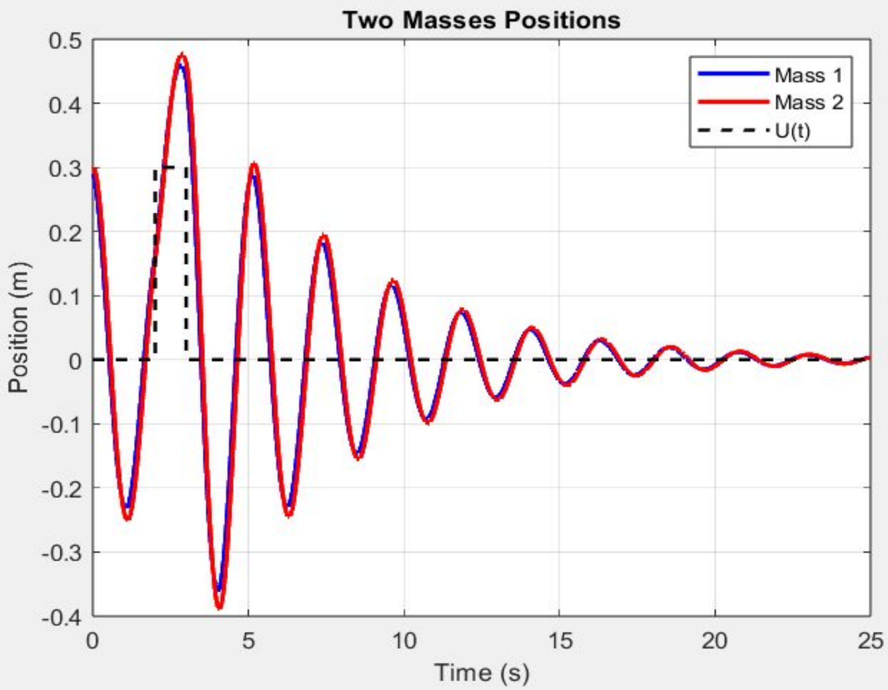
Analysis of Results

- Case 3 corrected the deviation fastest
  - It had the highest values for k1 and k2, and a relatively low damping constant compared to case 4
- Cases 2 and 4 had a roughly similar result
  - Case 2 had a lower value for k constants, but also a smaller value for the c constant
- Case 1 corrected the deviation slowest
  - It had the lowest value of each constant
- In general, cases with faster corrections had more rapid oscillations overall
  - This can be attributed to the larger k constants



Analytical Results

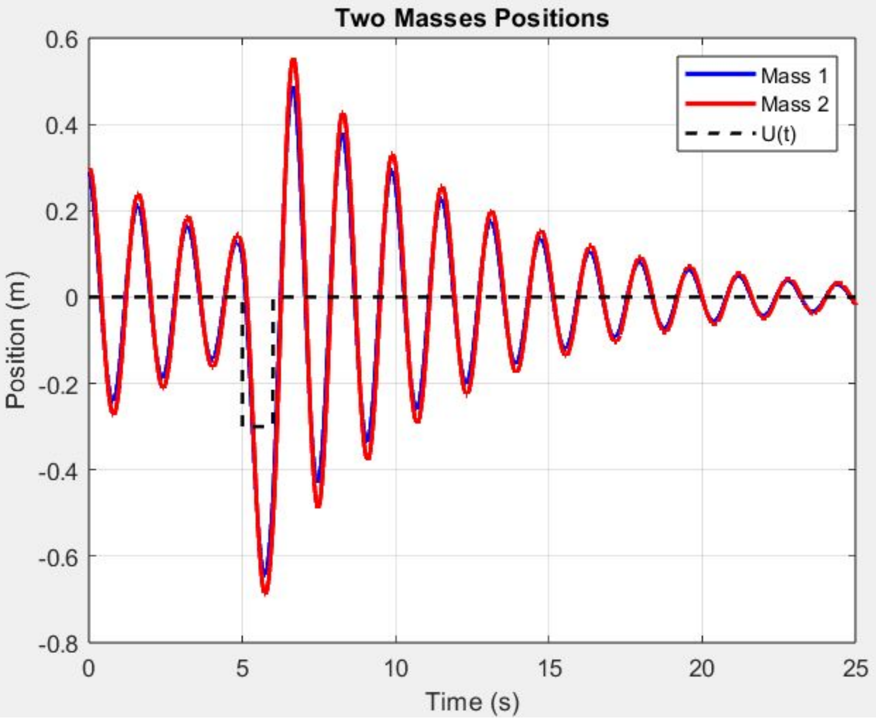




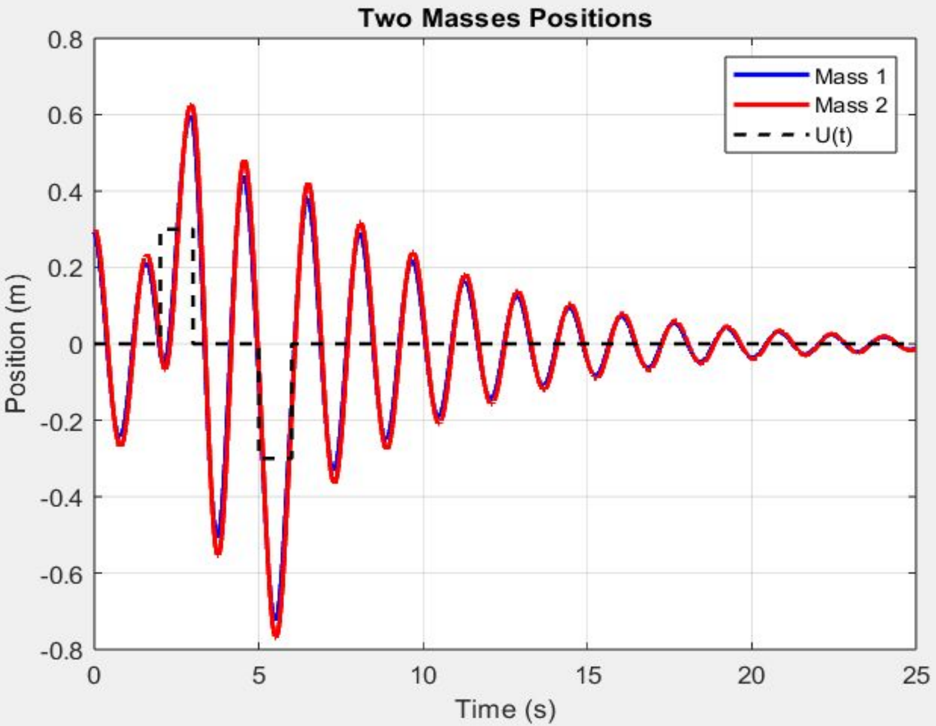
Settling Time: 5.56 s, at x=0.3

Settling Time: 9.78s

Analytical Results



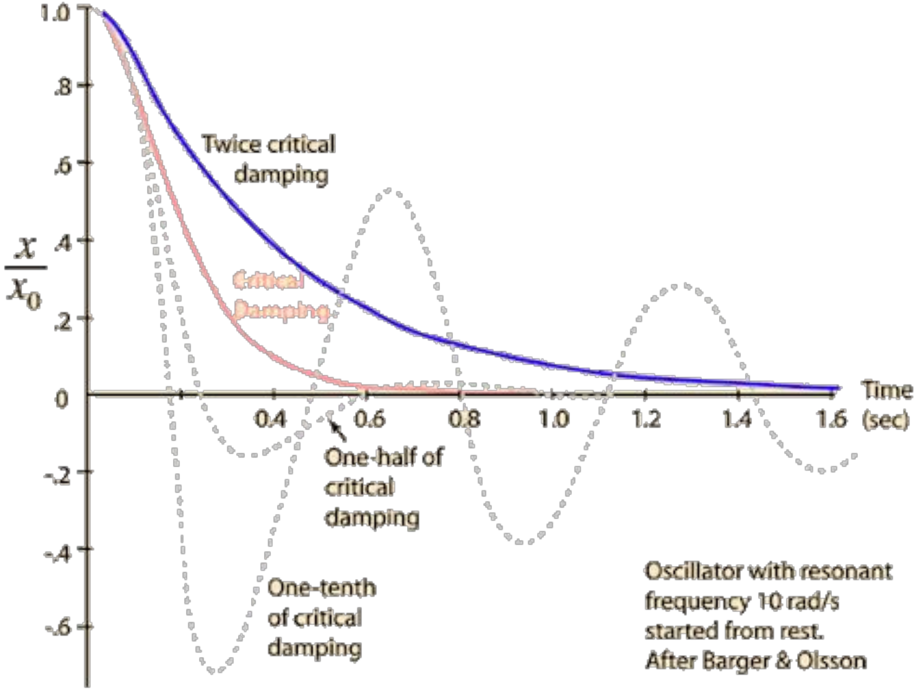
Settling Time: 16.42s



Settling Time: 13.76s

Challenging Problem

- The goal for a suspension system is to achieve critically damped motion, or very slightly underdamped motion
- Thus we should look at the shortest settling time
- The second case has the shortest settling time which shows that our suspension is best suited for that scenario
- Actually finding the optimal damping and spring constants is extremely challenging due to the coupling
- It is not a matter of just finding the eigenvalues and finding when they are equal like it would be for a simpler system
- Instead trial and error is easier by iterating an increase in either each spring constant or the damping constant



Challenging Problem Cont.

Which improves ride comfort more - increasing spring stiffness *k_i* or damping *c*?

Assuming bounce (oscillation) has a bigger impact on ride comfort than how fast the system settles…

Case 1 (top left): shortest settling time, low bounce

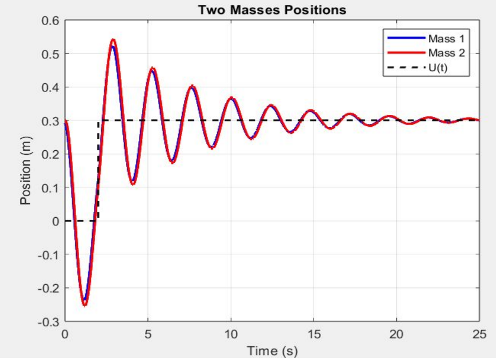
Case 2 (top right): increasing k_1 increases settling time and bounce increase

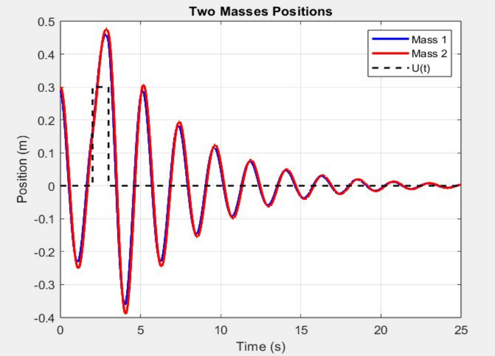
Case 3 (bottom left): increasing k_1 and k_2 leads to lots of bounce, high stiffness causes dramatic oscillation.

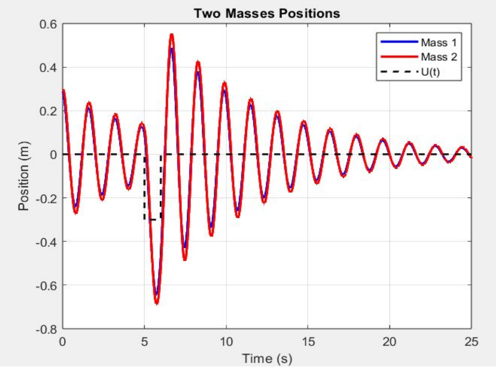
Case 4 (bottom right): same springs as Case 3, but more damping (increasing c) -> visibly smoother

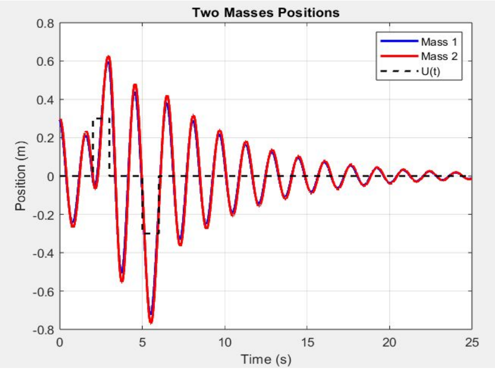
Increasing spring stiffness (k_1) may make the system more reactive, but it leads to uncomfortable oscillations.

Increasing damping (c) is more effective for comfort because it reduces bouncing — the thing passengers actually feel.

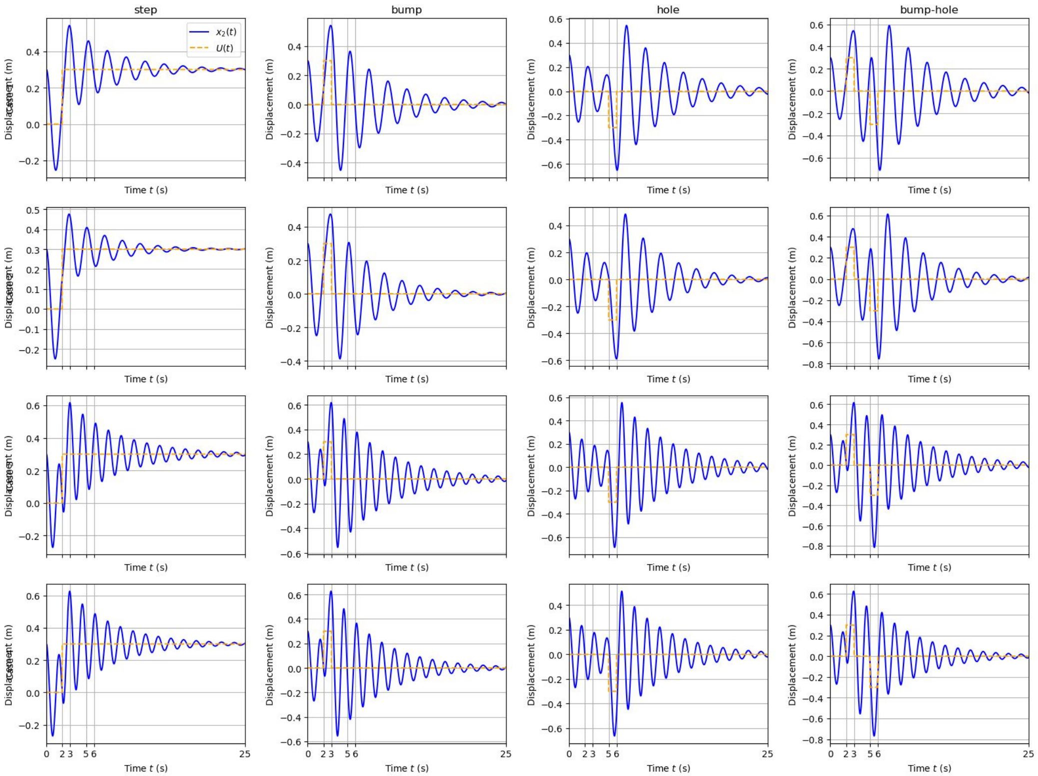








Challenging Problem Cont.




```
Settling Times (tol = 0.1 m):
         step   bump   hole  bump-hole  Average
Case 1   7.78  12.62  16.90      17.16    13.61
Case 2   5.24   9.88  14.03      15.30    11.11
Case 3  10.26  14.37  17.25      16.31    14.54
Case 4   9.42  13.41  15.53      14.54    13.23

Parameter Change Improvement:
                  Change  Before Avg (s)  After Avg (s)  Improvement (%)
0  Case1 → Case2 (k1 ↑)            13.6           11.1             18.4
1   Case3 → Case4 (c ↑)            14.5           13.2              9.1

Best base case by avg settling time: Case 2
```

1-2: Increase in spring stiffness results in slight drop in amplitude and increase in oscillations
2-3: Significant increase in spring stiffness results in dramatically increased frequency of oscillations
3-4: Increase in dampening results in slightly slower frequency of oscillations

Challenging Problem Cont.

```
#Params
m1, m2 = 18.0, 450.0
x1_0, x2_0 = 0.29, 0.30
v1_0, v2_0 = 0.0, 0.0
t_final = 25.0

cases = {
    'Case 1': {'k1':3.5e3, 'k2':1e4, 'c':7.3e3},
    'Case 2': {'k1':4.0e3, 'k2':1e4, 'c':7.3e3},
    'Case 3': {'k1':8.0e3, 'k2':4e4, 'c':7.3e3},
    'Case 4': {'k1':8.0e3, 'k2':4e4, 'c':1e4},
}

#RK4 h = 0.001
h = 0.001
t_eval = np.arange(0, t_final + h, h)

def rk4_step(rhs, y0, t, args):
    Y = np.zeros((len(t), len(y0)))
    Y[0] = y0
    for i in range(len(t)-1):
        ti, yi = t[i], Y[i]
        hi = t[i+1] - ti
        k1 = rhs(ti,          yi,        *args)
        k2 = rhs(ti + hi/2,   yi + hi*k1/2, *args)
        k3 = rhs(ti + hi/2,   yi + hi*k2/2, *args)
        k4 = rhs(ti + hi,     yi + hi*k3,   *args)
        Y[i+1] = yi + hi*(k1 + 2*k2 + 2*k3 + k4)/6
    return Y
```

Challenging Problem Cont.

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

#road functions
def U_step(t, A=0.3, t0=2.0):
    return A * (t >= t0)
def U_bump(t, A=0.3, t0=2.0, duration=1.0):
    return A * ((t >= t0) & (t < t0 + duration))
def U_hole(t, A=0.3, t0=5.0, duration=1.0):
    return -A * ((t >= t0) & (t < t0 + duration))
def U_bump_hole(t, A=0.3, t0=2.0, duration=1.0, t1=5.0):
    bump = ((t >= t0) & (t < t0 + duration)) * A
    hole = ((t >= t1) & (t < t1 + duration)) * (-A)
    return bump + hole

U_funcs = {
    'step':      U_step,
    'bump':      U_bump,
    'hole':      U_hole,
    'bump-hole': U_bump_hole
}

# quarter car model
def quarter_car_rhs(t, y, m1, m2, k1, k2, c, U_func):
    x1, x1d, x2, x2d = y
    U = U_func(t)
    x1dd = (-k1*(x1-U) + k2*(x2-x1) + c*(x2d-x1d))/m1
    x2dd = (-k2*(x2-x1) - c*(x2d-x1d))/m2
    return np.array([x1d, x1dd, x2d, x2dd])
```

Questions?

Thank you!