Moving Loads Assignment - Part A

Menno Marissen, Gabriele Mylonopoulos, Bart Slingerland, Sulongge Sulongge and Jerin Thomas (Group 2)

5381827, 4807421, 5309913, N/A and 6276423

In [1]:
```
import numpy as np
import sympy as sym
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from scipy.integrate import quad
from matplotlib.ticker import ScalarFormatter
```

Problem 1

First, the wave numbers and their corresponding waves are derived for a beam on elastic foundation. This is to help with interpretation.

In [2]:
```
# define equation parameters
x, t = sym.symbols('x t', real=True)
rho, A, EI, ksi = sym.symbols('rho A EI ksi', positive=True)

# define numerical values (taken from the provided example)
values = {EI:6.42E6, rho:1, A: 268.33, ksi: 8.333E7}

omega, k = sym.symbols('omega k')
w = sym.Function('w')(x, t)

w = sym.exp(-sym.I * (omega*t - k*x))

# define the equation of motion
EQM = sym.Eq(rho*A * sym.diff(w, t, 2) + EI * sym.diff(w, x, 4) + ksi * w, 0)
```

In [3]:
```
# define the dispersion relation
display(EQM.simplify())
disp_eq = EI*k**4 - omega**2 * rho*A + ksi
display(disp_eq.simplify())
```

$$\left(-A\omega^2\rho + EIk^4 + ksi\right) e^{i(kx-\omega t)} = 0$$

$$-A\omega^2\rho + EIk^4 + ksi$$

In [4]:
```
# solve the dispersion relation for k
k_sol = sym.solve(disp_eq, k)
for k in k_sol:
    display(k)

# calculate the cut-off frequency
wc = np.sqrt(values[ksi]/(values[rho]*values[A]))
```

$$-\frac{\sqrt[4]{A\omega^2\rho - ksi}}{\sqrt[4]{EI}}$$

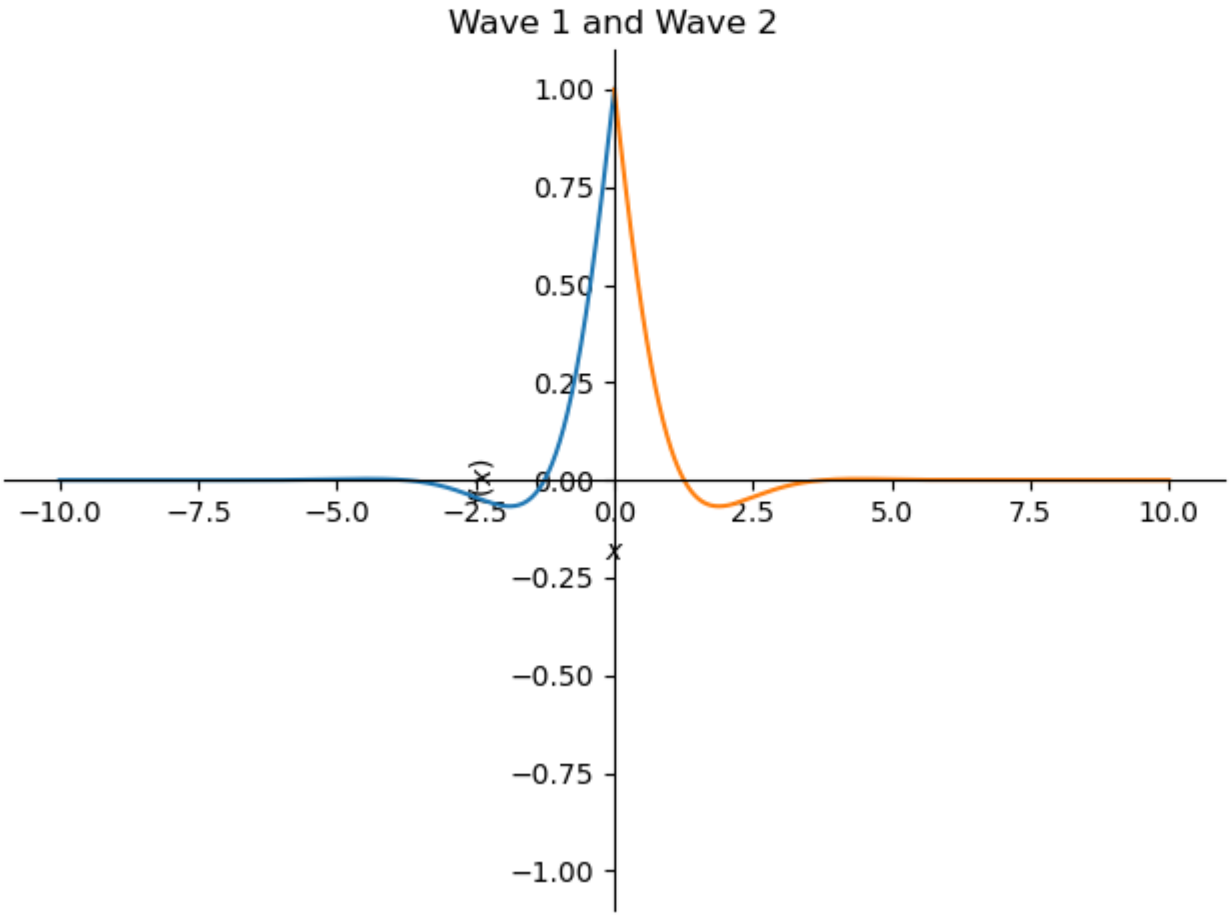
$$\frac{\sqrt[4]{A\omega^2\rho - ksi}}{\sqrt[4]{EI}}$$

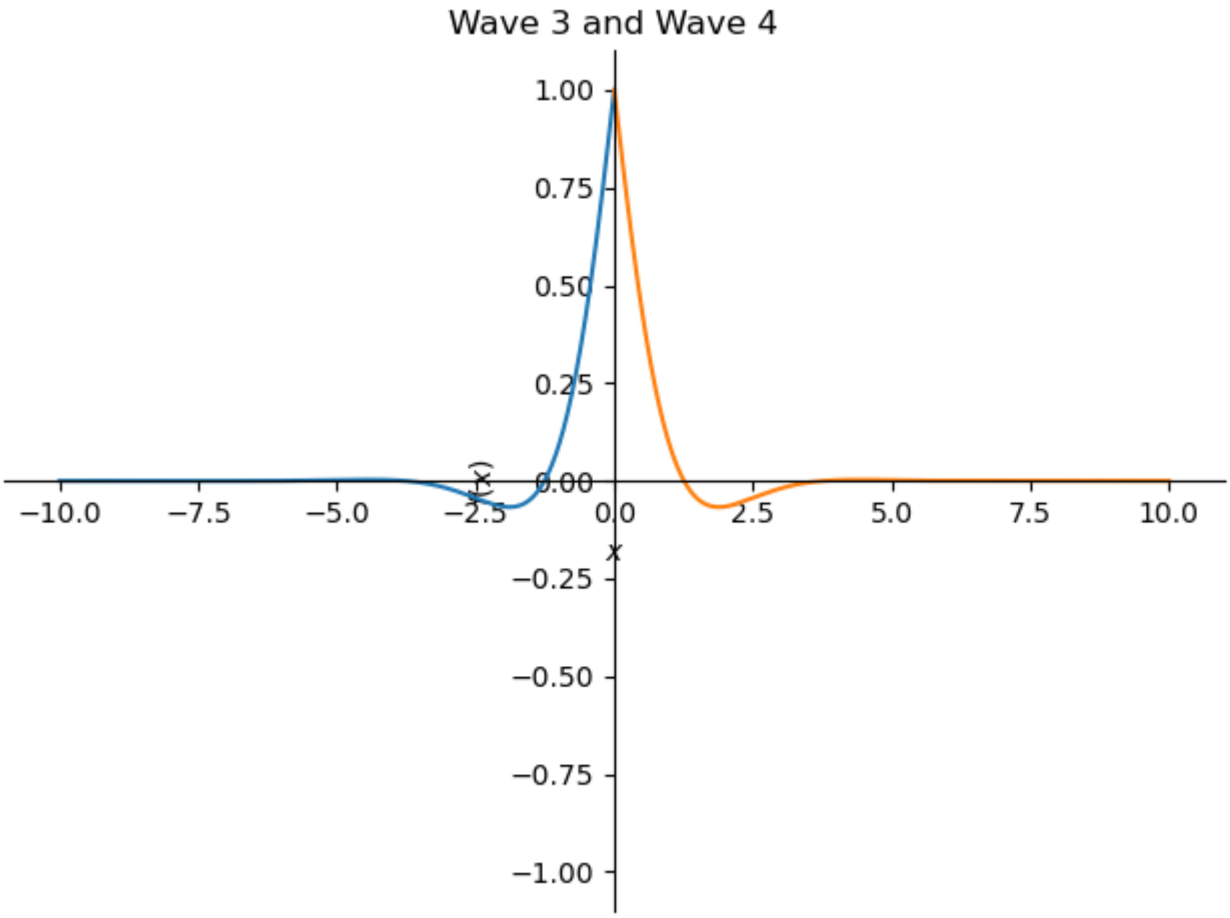
$$-\frac{i\sqrt[4]{A\omega^2\rho - ksi}}{\sqrt[4]{EI}}$$

$$\frac{i\sqrt[4]{A\omega^2\rho - ksi}}{\sqrt[4]{EI}}$$

In [5]:

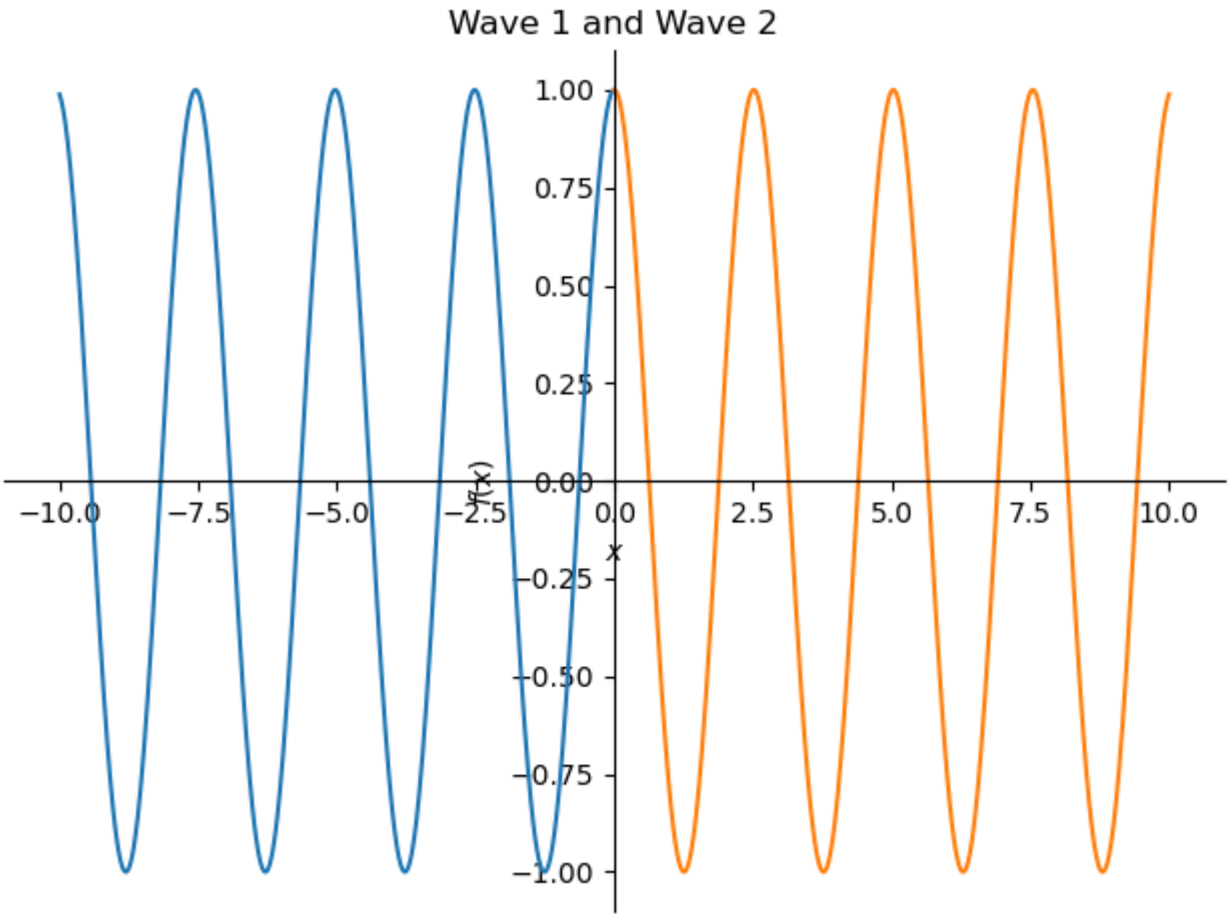
```
# Plot the steady state response for relatively low frequency
omega_val = 0.5*wc  # Set a value for omega
wave1 = sym.exp(sym.I * k_sol[0]*x).subs(values).subs({omega: omega_val})
wave2 = sym.exp(sym.I * k_sol[1]*x).subs(values).subs({omega: omega_val})
wave3 = sym.exp(sym.I * k_sol[2]*x).subs(values).subs({omega: omega_val})
wave4 = sym.exp(sym.I * k_sol[3]*x).subs(values).subs({omega: omega_val})

sym.plot((sym.re(wave1), (x,0,-10)), (sym.re(wave2), (x,10,0)), title='Wave 1 and Wave 2', sho
sym.plot((sym.re(wave3), (x,0,-10)), (sym.re(wave4), (x,10,0)), title='Wave 3 and Wave 4', sho
```

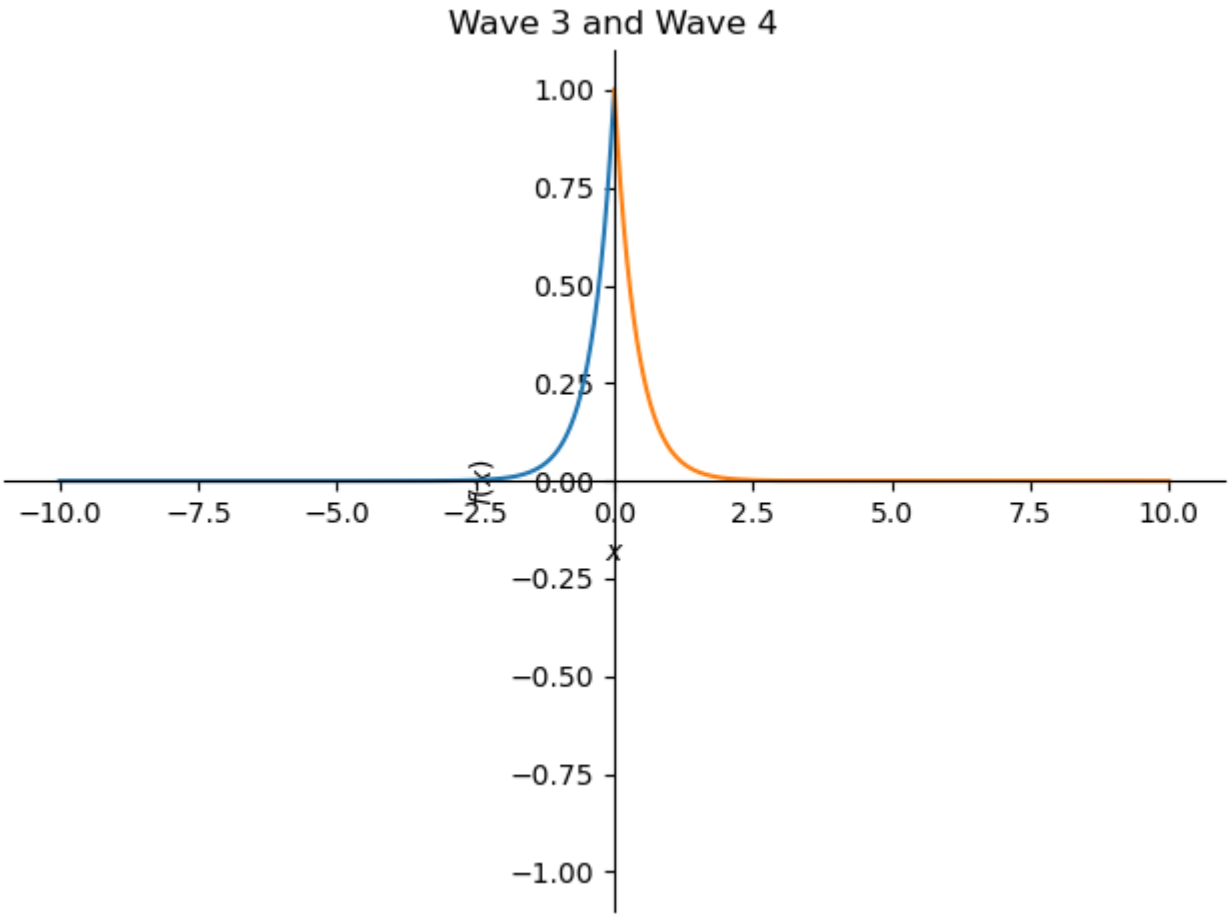




In [6]:

```
# Plot the steady state response for relatively low frequency
omega_val = 2*wc  # Set a value for omega
wave1 = sym.exp(sym.I * k_sol[0]*x).subs(values).subs({omega: omega_val})
wave2 = sym.exp(sym.I * k_sol[1]*x).subs(values).subs({omega: omega_val})
wave3 = sym.exp(sym.I * k_sol[2]*x).subs(values).subs({omega: omega_val})
wave4 = sym.exp(sym.I * k_sol[3]*x).subs(values).subs({omega: omega_val})

sym.plot((sym.re(wave1), (x,0,-10)), (sym.re(wave2), (x,10,0)), title='Wave 1 and Wave 2', sho
sym.plot((sym.re(wave3), (x,0,-10)), (sym.re(wave4), (x,10,0)), title='Wave 3 and Wave 4', sho
```





- For frequencies smaller than the cut-off frequency, the steady state response consists of standing waves.
- For frequencies larger than the cut-off frequency, the steady state response consists of two standing waves, and two propagating waves.

In [7]:
```
W = sym.Function('W')(x)
C1, C2, C3, C4 = sym.symbols('C1 C2 C3 C4')
F0, wc = sym.symbols('F0 wc')

W = C1*sym.exp(sym.I*k_sol[0]*x) + C2*sym.exp(sym.I*k_sol[1]*x) + C3*sym.exp(sym.I*k_sol[2]*x)
display(W)

# Define boundary conditions
# No moment at x=0
eq1 = sym.Eq(W.diff(x,x).subs(x,0),0)
# Force F0 at x=0
eq2 = sym.Eq(W.diff(x,x,x).subs(x,0), F0/EI)

# Radiation conditions at infinity
eq3 = sym.Eq(C1,0)
eq4 = sym.Eq(C3,0)

sol = sym.solve([eq1, eq2, eq3, eq4], (C1, C2, C3, C4))
sol
```

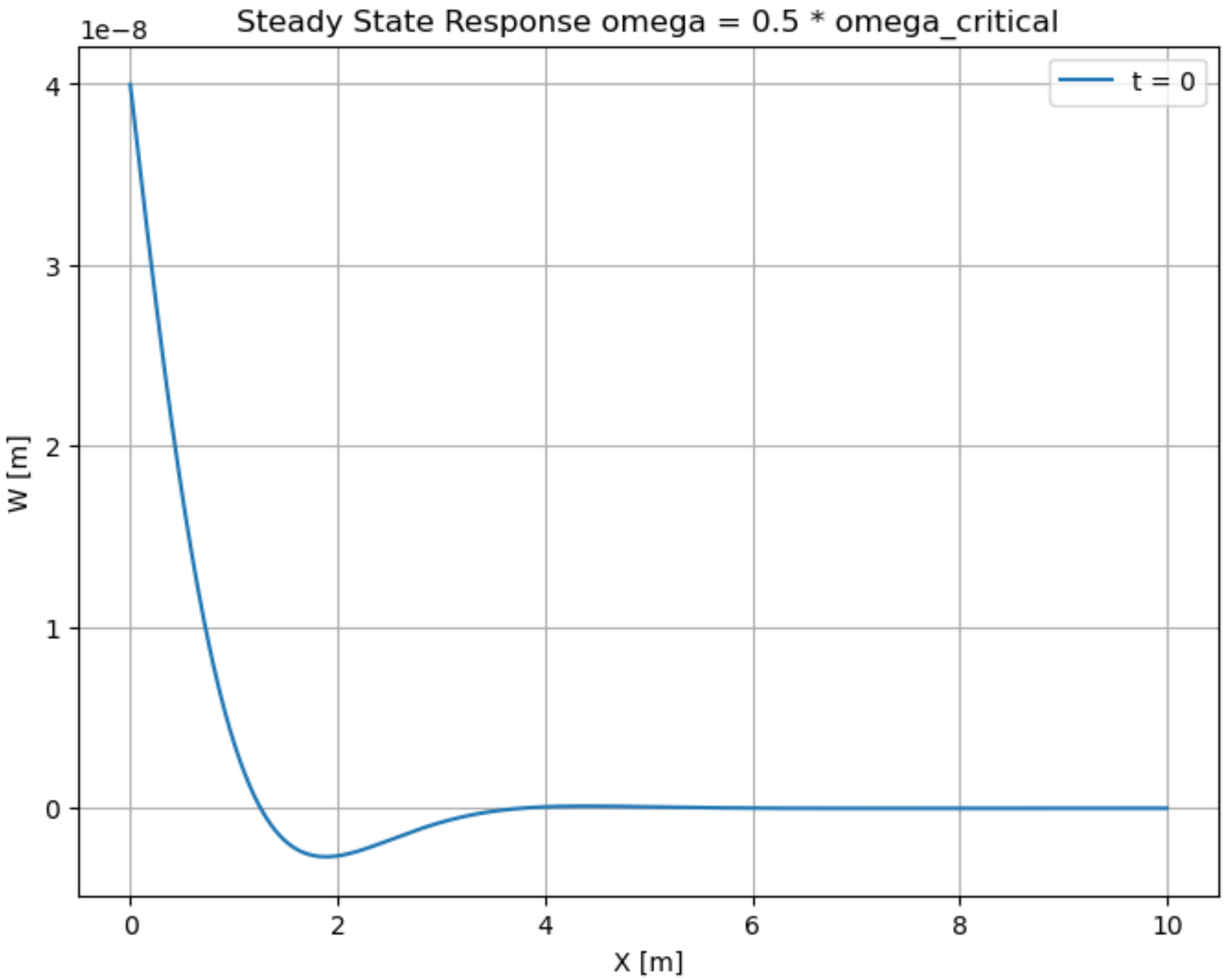
$$C_1 e^{-\frac{ix\sqrt[4]{A\omega^2\rho-ksi}}{\sqrt[4]{EI}}} + C_2 e^{\frac{ix\sqrt[4]{A\omega^2\rho-ksi}}{\sqrt[4]{EI}}} + C_3 e^{\frac{x\sqrt[4]{A\omega^2\rho-ksi}}{\sqrt[4]{EI}}} + C_4 e^{-\frac{x\sqrt[4]{A\omega^2\rho-ksi}}{\sqrt[4]{EI}}}$$

Out[7]:
```
{C1: 0,
 C2: -F0*(1 - I)/(2*EI**(1/4)*(A*omega**2*rho - ksi)**(3/4)),
 C3: 0,
 C4: -F0*(1 - I)/(2*EI**(1/4)*(A*omega**2*rho - ksi)**(3/4))}
```

In [8]:
```
W = W.subs(sol)
values = {EI:6.42E6, rho:1, A: 268.33, ksi: 8.333E7, F0: 1}
wc = np.sqrt(values[ksi]/(values[rho]*values[A]))
```

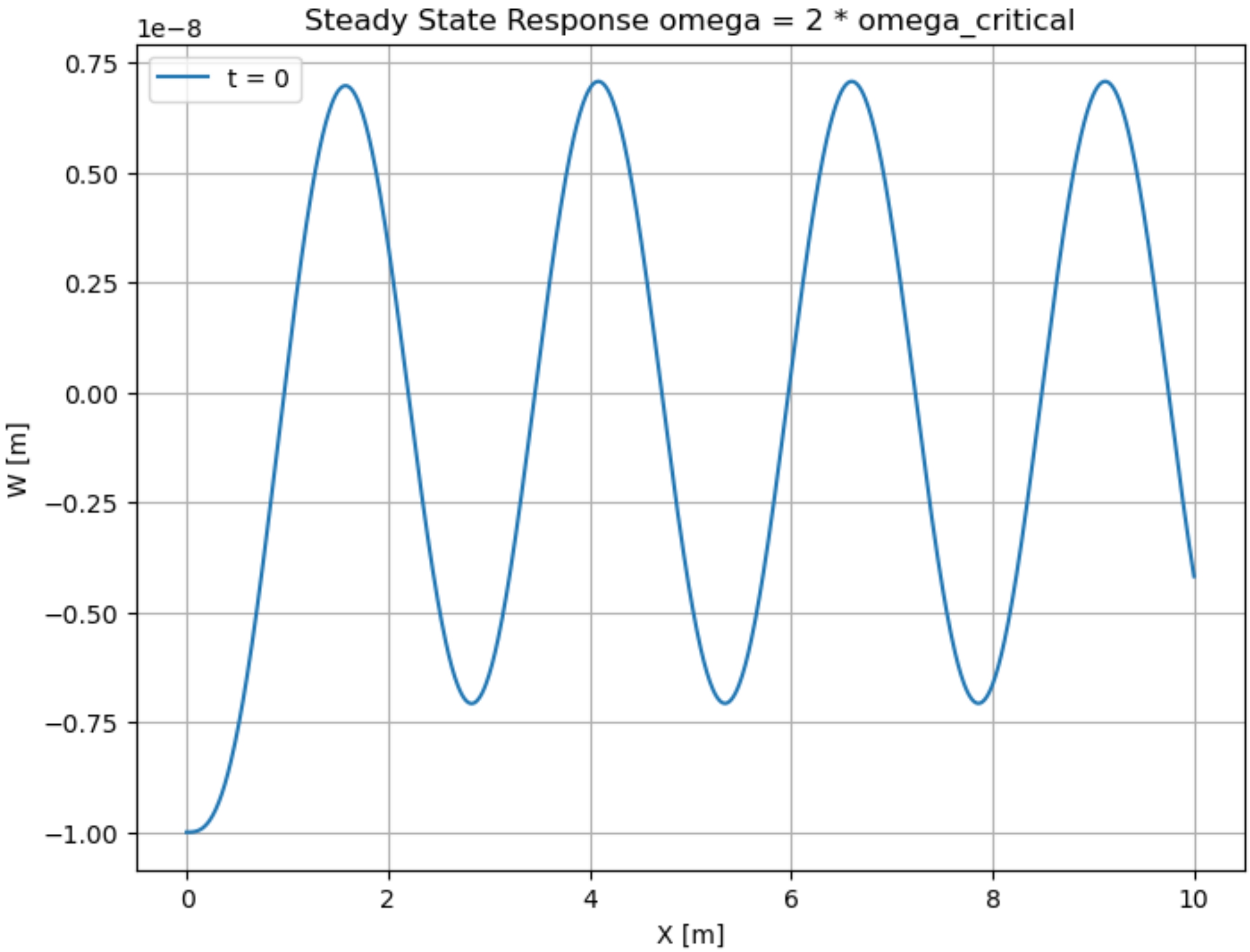
In [9]:
```
omega_val = 0.5*wc
W_time = sym.re(W.subs(values)*sym.exp(sym.I*omega*t)).subs(omega, omega_val)
```

In [10]:
```
W_plot = sym.lambdify([x, t], W_time, "numpy")

# Define a fixed moment in time
fixed_time = 0  # You can change this value to any desired time

# Generate x values for the plot
x_values = np.linspace(0, 10, 1000)

# Compute W_plot values at the fixed time
W_values = W_plot(x_values, fixed_time)

# Plot the results
plt.figure(figsize=(8, 6))
plt.plot(x_values, W_values, label=f't = {fixed_time}')
plt.xlabel('X [m]')
plt.ylabel('W [m]')
plt.title('Steady State Response omega = 0.5 * omega_critical')
plt.legend()
plt.grid(True)
plt.show()
```

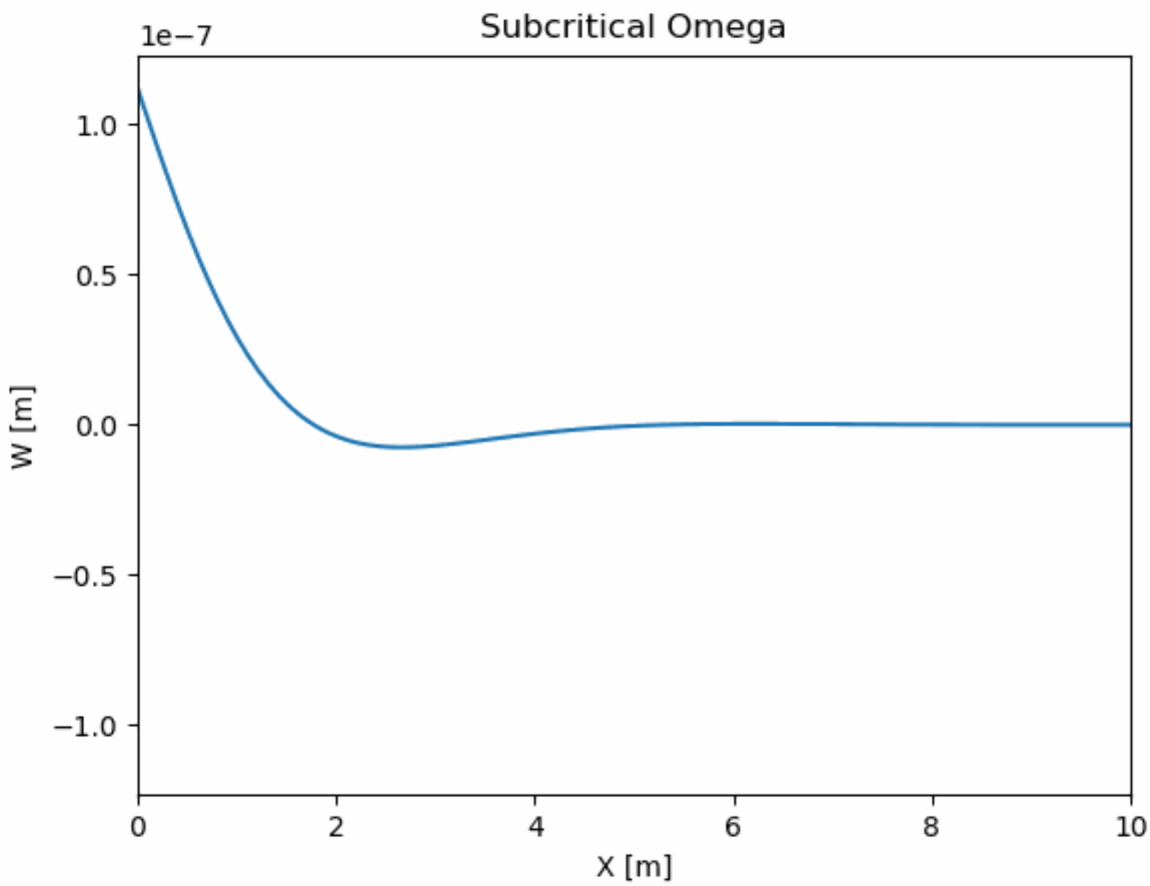


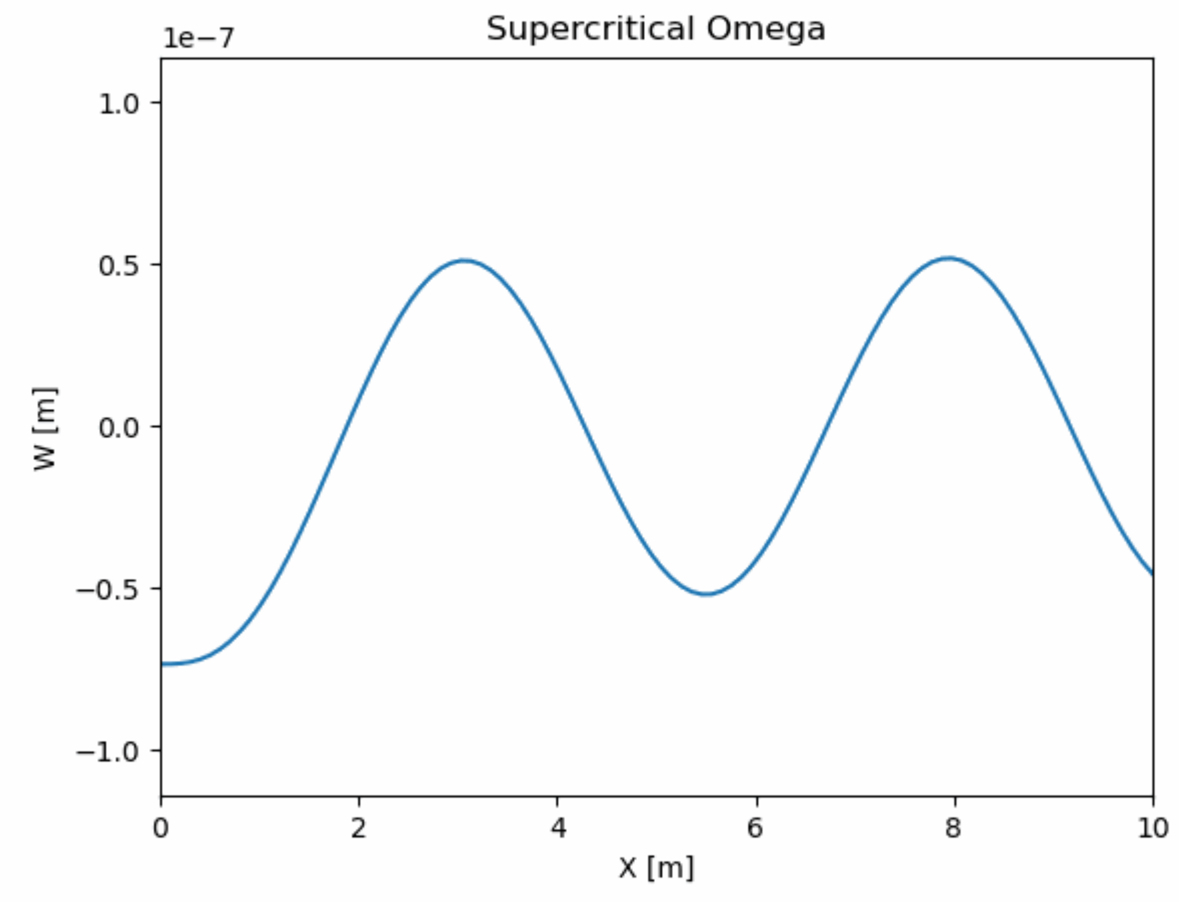
In [11]:
```
omega_val = 2.0*wc
W_time = sym.re(W.subs(values)*sym.exp(sym.I*omega*t)).subs(omega, omega_val)
```

In [12]:
```
W_plot = sym.lambdify([x, t], W_time, "numpy")

# Define a fixed moment in time
fixed_time = 0  # You can change this value to any desired time

# Generate x values for the plot
x_values = np.linspace(0, 10, 1000)

# Compute W_plot values at the fixed time
W_values = W_plot(x_values, fixed_time)

# Plot the results
plt.figure(figsize=(8, 6))
plt.plot(x_values, W_values, label=f't = {fixed_time}')
plt.xlabel('X [m]')
plt.ylabel('W [m]')
plt.title('Steady State Response omega = 2 * omega_critical')
plt.legend()
plt.grid(True)
plt.show()
```



Animations of the steady state response





The steady state response for subcritical omega consists of standing waves only: There is no energy propagting away from the source.

The steady state response for supercritical omega consist of a standing wave and a propagating wave. The standing wave is only of significance close to the source, whilst the propagting wave is visible all the way to infinity.

The standing waves and propagting waves derived in the first part of this problem can be recognized.

In [13]:
```
# code for creating animations
# fig, ax = plt.subplots()

# t = np.linspace(0, 2*np.pi/omega_val, 50)
# x = np.linspace(0, 10, 100)

# W_max = np.max(W_plot(0, t))

# line = ax.plot(x, W_plot(x, t[0]))[0]
# ax.set(xlim=[0, 10], ylim=[-1.1*W_max, 1.1*W_max], xlabel='X [m]', ylabel='W [m]', title='Su

# def update(frame):
#     w = W_plot(x, t[frame])

#     line.set_ydata(w)
#     return (line)


# anim = animation.FuncAnimation(fig=fig, func=update, frames=50, interval=50)
# anim.save(filename="./omega_subcritical.gif", writer="pillow");
# plt.close()
```

Problem 2

The Equation of motion of the system in this problem is:

$$\rho A\frac{\partial^2 w}{\partial t^2}+EI\frac{\partial^4 w}{\partial x^4}+\chi w+\eta\frac{\partial w}{\partial t}=Q_0\delta(x-Vt) \tag{1}$$

In order to obtain the steady-state response of the system, we first apply the Fourier Transform in both space and time to the equation:

$$(-\rho A\omega^2+EIk^4+\chi+i\eta)\tilde{w}(k,\omega)=2\pi Q_0\delta(\omega-kV) \tag{2}$$

$$\tilde{w}(k,\omega)=\frac{2\pi Q_0\delta(\omega-kV)}{-\rho A\omega^2+EIk^4+\chi+i\eta} \tag{3}$$

in which

$$\tilde{w}(k,\omega)=\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}w(x,t)\mathrm{e}^{-\mathrm{i}(\omega t-kx)}dxdt \tag{4}$$

The solution in the space-frequency domain is:

$$\hat{w}(x,\omega)=\frac{1}{2\pi}\int_{-\infty}^{+\infty}2\pi\frac{Q_0}{EI}\frac{\delta(\omega-kV)}{\Delta(\omega,k)}\mathrm{e}^{-\mathrm{i}kx}dk\overset{k=\frac{\omega}{V}}{=}\frac{1}{V}\frac{Q_0}{EI}\frac{\mathrm{e}^{-\mathrm{i}\frac{\omega}{V}x}}{\Delta(\omega,V)} \tag{5}$$

and in space-time domain, with the moving reference frame substituted:

$$w(\xi)=\frac{1}{2\pi}\frac{1}{V}\frac{Q_0}{EI}\int_{-\infty}^{+\infty}\frac{\mathrm{e}^{-\mathrm{i}\frac{\omega}{V}\xi}}{\Delta(\omega,V)}d\omega \tag{6}$$

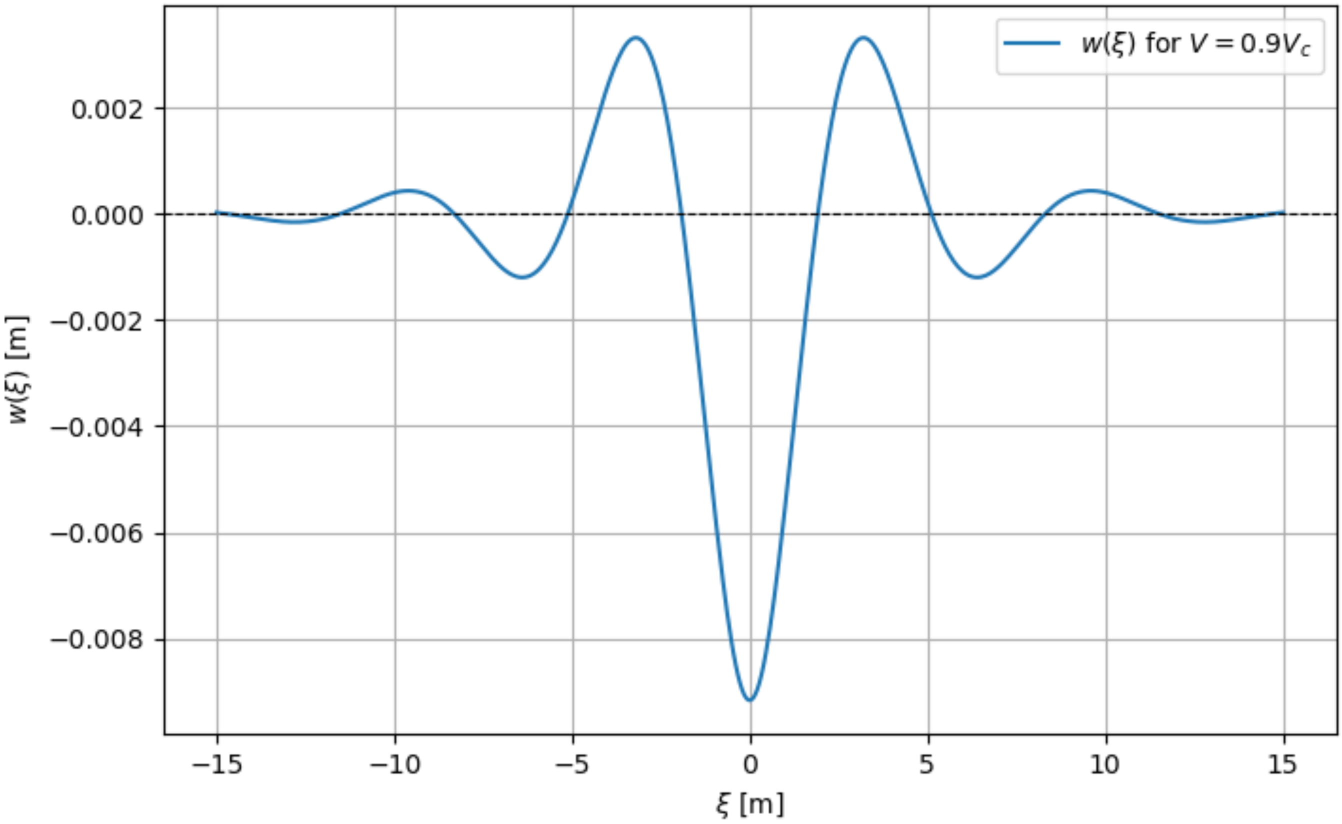
in which:

$$\Delta(\omega,V)=\left(\frac{\omega}{V}\right)^4-\frac{\rho A}{EI}\omega^2+i\frac{\eta}{EI}\omega+\frac{\chi}{EI} \tag{7}$$

The integration in the solution can be computed numerically, and plots are shown below.

In [14]:
```
# Define constants
rhoA = 268.3
EI = 6.42e6
chi = 7.3e6
eta = 1e2
Q0 = 80e3
Vc = (4 * chi * EI / (rhoA**2))**(1/4)
V = 0.9 * Vc

# Integrand
def integrand(omega, xi):
    Delta = (omega / V)**4 - (rhoA / EI) * omega**2 + (chi / EI) + 1j * eta / EI
    return np.exp(-1j * omega * xi / V) / Delta

# Calculate the w_xi function with numerical integration
def w_xi_func(xi):
    result, _ = quad(lambda omega: integrand(omega, xi).real, -1000, 1000)
    return (1 / (2 * np.pi * V * EI)) * Q0 * result

# Plot
xi_vals = np.linspace(-15, 15, 500)
w_xi_vals = np.array([w_xi_func(xi) for xi in xi_vals])
#
plt.figure(figsize=(8, 5))
plt.plot(xi_vals, -w_xi_vals, label=r"$w(\xi)$ for $V = 0.9V_c$")
plt.axhline(0, color='black', linewidth=0.8, linestyle='--')
```

```python
plt.xlabel(r"$\xi$ [m]")
plt.ylabel(r"$w(\xi)$ [m]")
plt.legend()
plt.grid()
plt.show()
```



Problem 3

For the same problem considered in the previous question, derive and compute the equivalent stiffness at the loading/contact point and plot it versus velocity. Consider in the plot only the sub-critical velocity range (up to the 99% of the critical velocity) and explain what you observe.

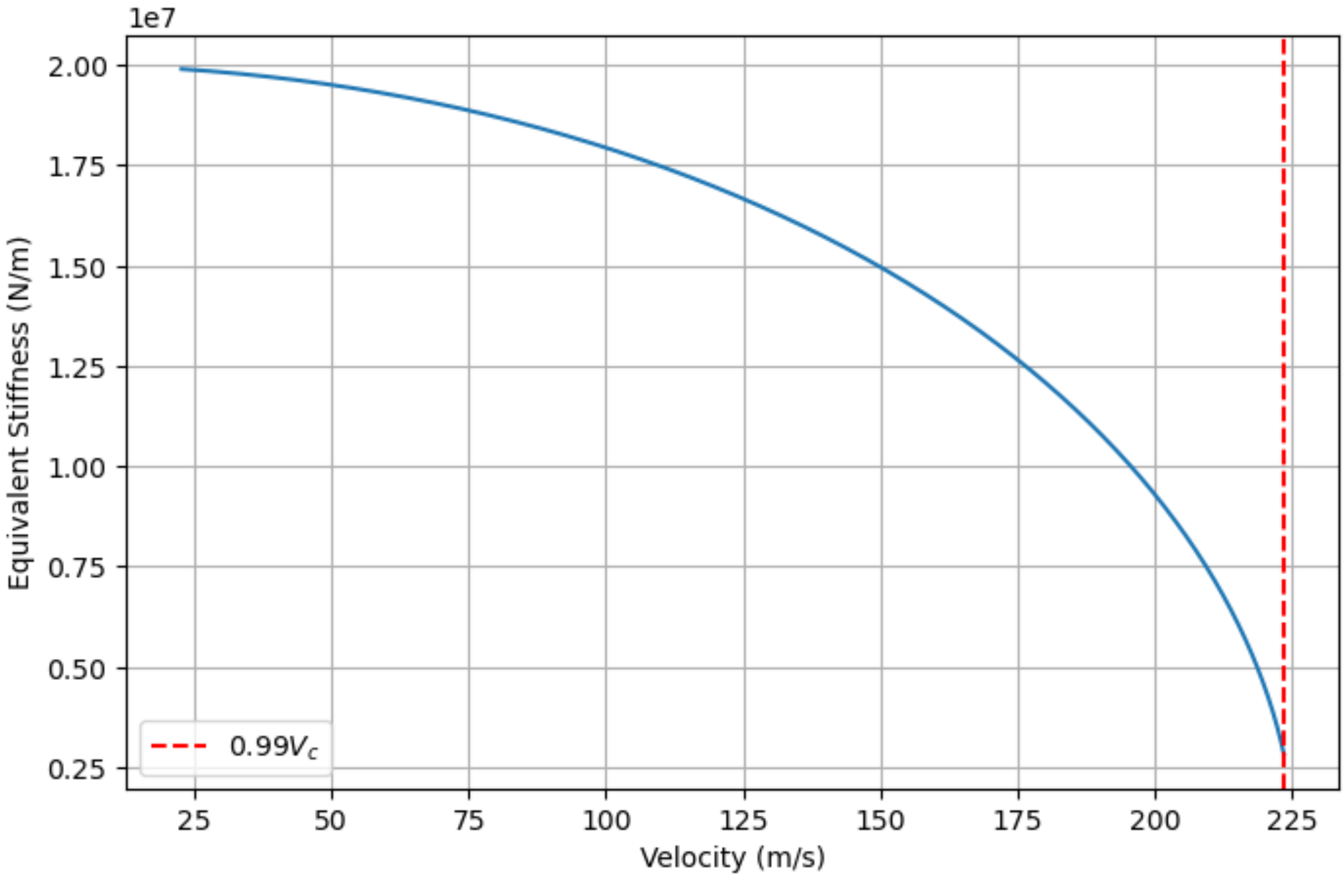
The equivalent stiffness under the moving load is:

$$k_{\mathrm{eq}} = \frac{Q_0}{w(\xi = 0)} \tag{8}$$

The deflection under the moving load depends on the speed of the moving load. The result is plotted below. The general trend is obvious, that when the speed approaches critical speed, the equivalent stiffness drops significantly. It does not go to zero as an viscous damping is included in the model. On the other side, when the speed is very small, the equivalent stiffness approaches the static stiffness.

In [15]:
```python
V_vals = np.linspace(0.1 * Vc, 0.99 * Vc, 1000)

# Compute equivalent stiffness
k_eq_vals = []
for V in V_vals:
    w_xi_0 = w_xi_func(0)
    k_eq = Q0 / w_xi_0
    k_eq_vals.append(k_eq)

# Plot
plt.figure(figsize=(8, 5))
plt.plot(V_vals, k_eq_vals)
plt.axvline(0.99 * Vc, color='red', linestyle='--', label=r"$0.99 V_c$")
```

```
plt.xlabel("Velocity (m/s)")
plt.ylabel("Equivalent Stiffness (N/m)")
plt.legend()
plt.grid()
plt.show()
```



Problem 4

Firstly, the material constants are defined as in the assignment.

In [16]:
```
Q0 = 80E3
EI = 6.42E6
rhoA = 268.3
L = 100
xi = 7.3E6
zeta = 0.05

V_crit = ((4*xi*EI)/(rhoA)**2)**(1/4)
V = 0.75*V_crit
```

In order to solve the response of simply supported beam, the modal expansion method is used, and solution to problem is:

$$w(x,t) = \sum_{m=1}^{\infty} q_m(t)\sin(\beta_m x) \tag{9}$$

in which $\sin(\beta_m x)$ is the mode shapes for m_{th} mode and can be computed easily as follows:

In [17]:
```
def beta(n):
    return np.pi * n / L

def mode_shape(x, n):
    return np.sin(beta(n) * x)
```

$q_m(t)$ is the time-dependant amplitude for each mode. It can be computed using the green's function method, in which q_m is obtained by:

$$q_m = \int_{\tau=0}^{t} f_m(\tau) g_m(t-\tau) d\tau \tag{10}$$

where the modal forcing f_m and modal green's function g_m are expressed as:

$$f_m(t) = \frac{Q_0 B(t) \varphi_m(Vt)}{\bar{m}_m} \tag{11}$$

$$g_m(t) = \frac{1}{\omega_m} \sin(\omega_m t) H(t) \tag{12}$$

Terms mentioned above thus can be defined:

In [18]:
```
def omega(n):
    return np.sqrt(beta(n)**4 * EI/rhoA + xi/rhoA)

def omega_b(n):
    return omega(n) * np.sqrt(1 - zeta**2)


def green_function(t, n):
    return 1/omega_b(n) * np.exp(-zeta*omega(n)*t) * np.sin(omega_b(n)*t) * (t >= 0)

def f_n(t, n):
    return 2*Q0/(rhoA*L) * mode_shape(V*t, n)
```

Thus, the expression for the time-dependant magnitude term q_m can be written, and implemented as follows:

$$q_m = \frac{Q_0}{\bar{m}} \begin{cases} \int_{\tau=0}^{t} \varphi_m(V\tau) g_m(t-\tau) d\tau, & 0 < t < L/V \\ \int_{\tau=0}^{L/V} \varphi_m(V\tau) g_m(t-\tau) d\tau, & t > L/V \end{cases} \tag{13}$$

In [19]:
```
def q_n(t, n):
    n_integration = 1000
    def integrand(tau):
        return f_n(tau, n) * green_function(t-tau, n)
    if t < L/V:
        result = np.trapezoid(integrand(np.linspace(0, t, n_integration)), dx=t/n_integration)
    else:
        result = np.trapezoid(integrand(np.linspace(0, L/V, n_integration)), dx=L/V/n_integrat
    return result
```

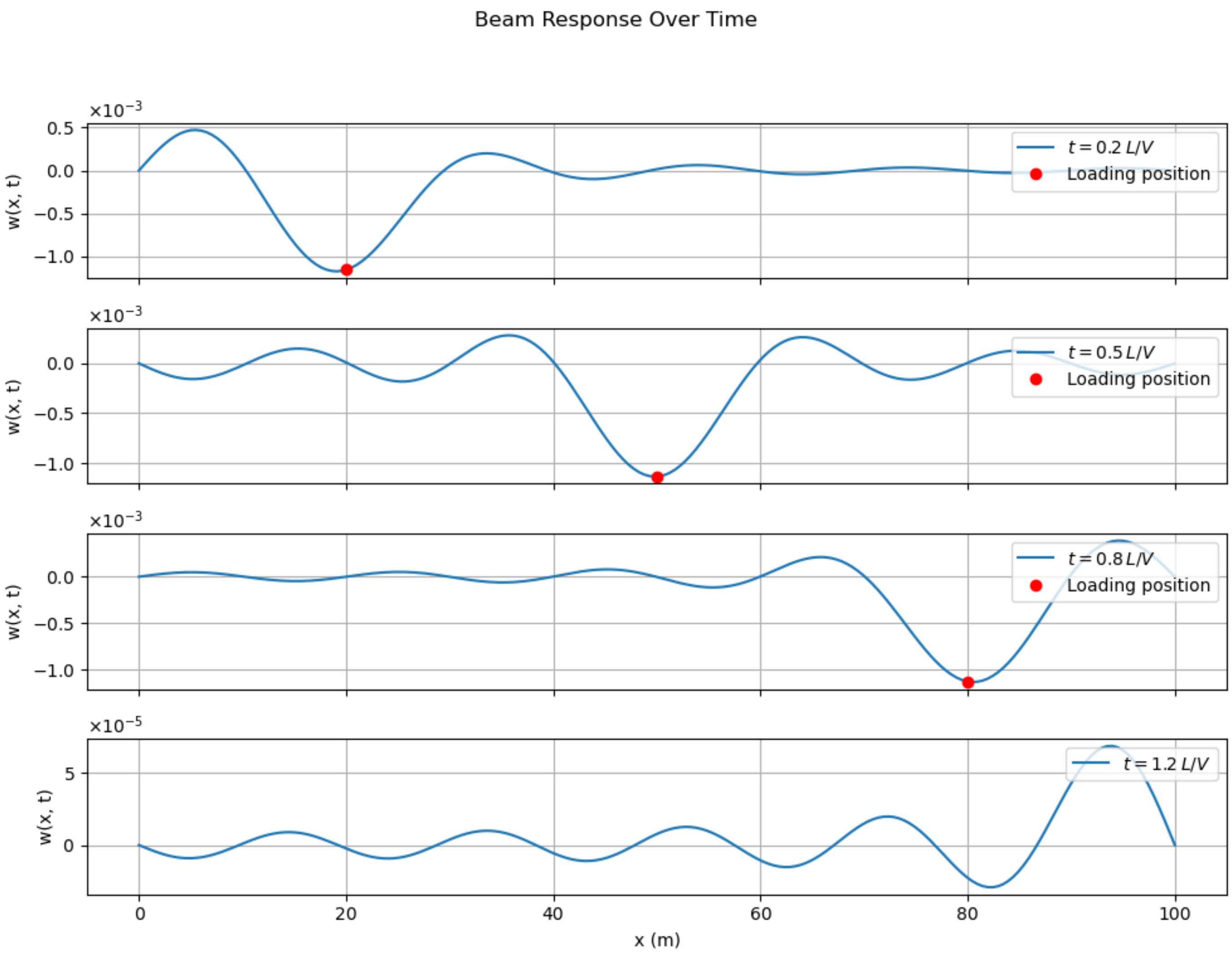
Finally, using modal expansion method, the total solution is equal to the contributions from all n modes being considered, and implemented as follows:

In [20]:
```
def w(x,t, N=10):
    total = 0
    for n in range(1, N+1):
        total += q_n(t, n) * mode_shape(x, n)

    return total
```

The response at various time instance are plotted below. It can be seen that: 1) when the load is on the beam, the deflection under the moving load is largest. 2) When load left the beam, in this case, the response is much smaller.

In [21]:
```
x_vals = np.linspace(0, L, 200)
t_vals = [0.2*L/V, 0.5*L/V, 0.8*L/V, 1.2 *L/V]  # Include t > L/V
```

```python
# Plot results
fig, axes = plt.subplots(4, 1, figsize=(10, 8), sharex=True)
for i, t in enumerate(t_vals):
    w_vals = [-w(x, t) for x in x_vals]
    n = t / (L / V)
    axes[i].plot(x_vals, w_vals, label=fr'$t = {n:.1f}\,L/V$')
    x_load = V * t
    if 0 <= x_load <= L:
        w_load = -w(x_load, t)
        axes[i].plot(x_load, w_load, 'ro', label='Loading position')
    axes[i].set_ylabel('w(x, t)')
    axes[i].legend(loc='upper right')
    axes[i].grid(True)
    axes[i].yaxis.set_major_formatter(ScalarFormatter(useMathText=True))
    axes[i].ticklabel_format(axis='y', style='sci', scilimits=(0,0))
axes[-1].set_xlabel('x (m)')
plt.suptitle('Beam Response Over Time')
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```



Problem 5

question A

Deriving the dispersion equation using sympy

In [22]:

```python
# define the paramters for the ODE
x = sym.symbols('x')
m = sym.symbols('m')
k, omega = sym.symbols('k, omega')
```

```python
xi, K_s = sym.symbols('xi, K_s')

C1, C2 = sym.symbols('C1 C2')
kappa, L, k_s, eta_s, T = sym.symbols('kappa L k_s eta_s T')
W = sym.Function('W')(x)

# Define the ODE for the beam in terms of kappa
ODE = sym.Eq(W.diff(x, 2) + kappa**2*W, 0)
display(ODE)

W = C1 * sym.sin(kappa*x) + C2 * sym.cos(kappa*x)
```

$$\kappa^2 W(x) + \frac{d^2}{dx^2} W(x) = 0$$

In [23]:
```python
# Define W of the m-th beam element
W_m = sym.Function('W_m')(x,m)
W_m = W.subs(x, x-m*L)*sym.exp(-1j*k*m*L)
display(W_m)
```

$$\left(C_1 \sin\left(\kappa\left(-Lm + x\right)\right) + C_2 \cos\left(\kappa\left(-Lm + x\right)\right)\right) e^{-1.0iLkm}$$

In [24]:
```python
# interface conditions
# Deflection continuity
eq1 = sym.Eq(W_m.subs(m, m), W_m.subs(m, m+1))
# Force equilibrium
eq2 = sym.Eq(-W_m.diff(x).subs(m, m) + W_m.diff(x).subs(m, m+1), k_s/T * W_m.subs(m,m))

display(eq1)
display(eq2)

# Convert the equations to matrix form
eqns = [eq1.expand(), eq2.expand()]
M = sym.linear_eq_to_matrix(eqns, [C1, C2])[0]
display(M)
```

$$\left(C_1 \sin\left(\kappa\left(-Lm + x\right)\right) + C_2 \cos\left(\kappa\left(-Lm + x\right)\right)\right) e^{-1.0iLkm} = \left(C_1 \sin\left(\kappa\left(-L\left(m+1\right) + x\right)\right) + C_2 \cos\left(\kappa\left(-L\left(m+1\right)\right.\right.\right.$$

$$-\left(C_1\kappa \cos\left(\kappa\left(-Lm + x\right)\right) - C_2\kappa \sin\left(\kappa\left(-Lm + x\right)\right)\right) e^{-1.0iLkm} + \left(C_1\kappa\cos\left(\kappa\left(-L\left(m+1\right) + x\right)\right) - C_2\kappa\sin\left(\kappa\left(-L\right.\right.\right.$$

$$\left[\begin{matrix} -e^{-1.0iLkm}\sin\left(L\kappa m - \kappa x\right) + e^{-1.0iLk}e^{-1.0iLkm}\sin\left(L\kappa m + L\kappa - \kappa x\right) & e^{-} \\ -\kappa e^{-1.0iLkm}\cos\left(L\kappa m - \kappa x\right) + \kappa e^{-1.0iLk}e^{-1.0iLkm}\cos\left(L\kappa m + L\kappa - \kappa x\right) + \frac{k_s e^{-1.0iLkm}\sin\left(L\kappa m - \kappa x\right)}{T} & -\kappa e^{-1.0iLkm}\sin \end{matrix}\right.$$

In [25]:
```python
# Compute the determinant of the matrix
disp_eq = M.det().expand()

# Substitute xi and K_s to simplify the equation
disp_eq = disp_eq.subs(k_s/T, K_s).subs(sym.exp(-1j*k*L), xi)
disp_eq = disp_eq/kappa
disp_eq = disp_eq/sym.exp(-2j*L*k*m)
```

In [26]:
```python
# Display the derived dispersion equation
display(disp_eq.simplify())
```

$$-\frac{K_s \xi \sin\left(L\kappa\right)}{\kappa} - 2\xi\cos\left(L\kappa\right) + \xi^{2.0} + 1$$

Dispersion equation provided in the problem:

$$\xi^2 - \left(\frac{K_s}{\kappa}\sin(\kappa L) + 2\cos(\kappa L)\right)\xi + 1 = 0, \quad \xi = \exp(-ikL), \quad K_s = \frac{k_s}{T}, \quad \kappa = \frac{\omega}{c} = \frac{\omega}{\sqrt{T/\rho A}}$$

The provided dispersion equation and the derived dispersion equation are equal.

question B

In [27]:
```
# Defining numerical values
values = {L:10, K_s:4E3/15E3}

# Solve the dispersion equation for xi
eq = sym.Eq(disp_eq.subs(values), 0)
sol = sym.solve(eq, xi)
```

In [28]:
```
xi1 = sol[0]
xi2 = sol[1]

k1 = 1j*sym.log(xi1)/(L)
k2 = 1j*sym.log(xi2)/(L)

print("Expressions for the wavenumbers k1 and k2:")
display(k1.simplify())
display(k2.simplify())

xi1_lambda = sym.lambdify(kappa, xi1)
xi2_lambda = sym.lambdify(kappa, xi2)

k1_lambda = sym.lambdify(kappa, k1.subs(values))
k2_lambda = sym.lambdify(kappa, k2.subs(values))
```

```
Expressions for the wavenumbers k1 and k2:
```

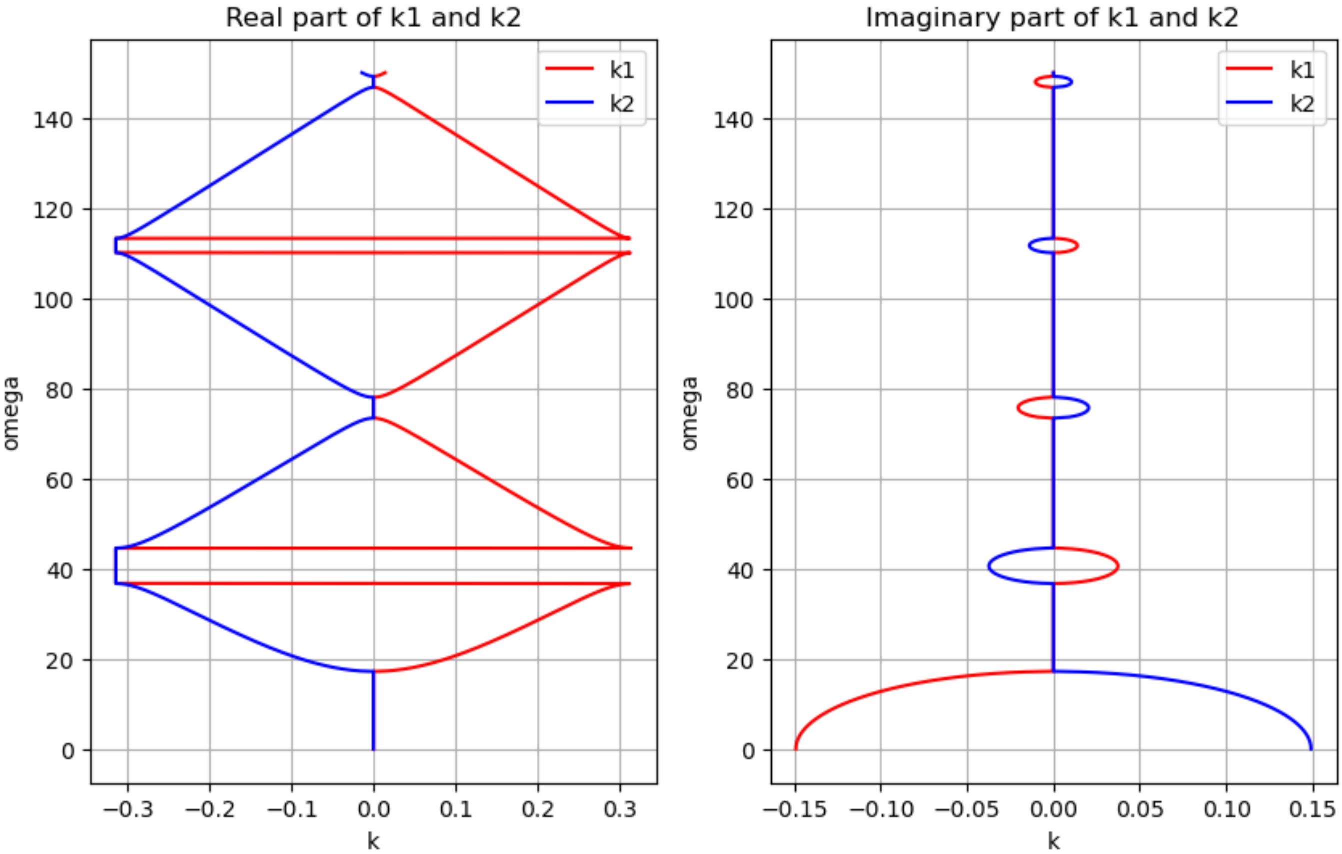
$$\frac{1.0 i \log\left(\frac{1.0\kappa\cos(10.0\kappa) - 1.0\sqrt{-1.0\kappa^2\sin^2(10.0\kappa) + 0.133333333333333\kappa\sin(20.0\kappa) + 0.0177777777777778\sin^2(10.0\kappa)} + 0.133333333333333\sin(10.0\kappa)}{\kappa}\right)}{L}$$

$$\frac{1.0 i \log\left(\frac{1.0\kappa\cos(10.0\kappa) + 1.0\sqrt{-1.0\kappa^2\sin^2(10.0\kappa) + 0.133333333333333\kappa\sin(20.0\kappa) + 0.0177777777777778\sin^2(10.0\kappa)} + 0.133333333333333\sin(10.0\kappa)}{\kappa}\right)}{L}$$

In [29]:
```
# Plot the real parts of the dispersion curves
omega_values = np.linspace(0.001, 150, 10000, dtype=complex)

kappa_values = omega_values/np.sqrt(15E3/1.1)

T_val = 15E3
rhoA = 1.1

fig, ax = plt.subplots(1, 2, figsize=(10, 6))

ax[0].plot(np.real(k1_lambda(kappa_values)), omega_values, 'r', label='k1')
ax[0].plot(np.real(k2_lambda(kappa_values)), omega_values, 'b',  label='k2')

ax[1].plot(np.imag(k1_lambda(kappa_values)), omega_values, 'r', label='k1')
ax[1].plot(np.imag(k2_lambda(kappa_values)), omega_values, 'b', label='k2')

ax[0].set_ylabel('omega')
ax[0].set_xlabel('k')
ax[0].legend()
ax[0].set_title('Real part of k1 and k2')
ax[0].grid();

ax[1].set_ylabel('omega')
ax[1].set_xlabel('k')
ax[1].legend()
ax[1].set_title('Imaginary part of k1 and k2')
```

```
ax[1].grid();
```

```
c:\Users\bart\anaconda3\envs\slender\Lib\site-packages\matplotlib\cbook.py:1762: ComplexWarnin
g: Casting complex values to real discards the imaginary part
  return math.isfinite(val)
c:\Users\bart\anaconda3\envs\slender\Lib\site-packages\matplotlib\cbook.py:1398: ComplexWarnin
g: Casting complex values to real discards the imaginary part
  return np.asarray(x, float)
```



- The real part is nonzero when the imaginary part is zero and the other way around.
- Repeating in k with period of 2pi/L.
- Repeating in omega with 2pi/L * sqrt(T/rhoA)
- Has two stop bands every period in omega.
- The dispersion curve shows a lot of similarities with the periodically supported beam model.

question C

- Equation of Motion in omega, k domain

$$(-\rho A\omega^2 + Tk^2)\tilde{w} = 2\pi Q_0 \delta(\omega - kV) - \hat{R}(\omega)\sum_{n=-\infty}^{\infty} e^{-i\left(\frac{\omega}{V}-k\right)nL}$$

$$(-\rho A\omega^2 + Tk^2)\tilde{w} = 2\pi \frac{Q_0}{V}\,\delta\left(\frac{\omega}{V} - k\right) - \hat{R}(\omega)\frac{2\pi}{L}\sum_{n=-\infty}^{\infty} \delta\left(\frac{\omega}{V} - k - \frac{2\pi n}{L}\right)$$

- Solution for generic R (omega, k domain)

$$\tilde{w}(\omega, k) = \frac{2\pi}{\Delta_b(\omega,k)}\frac{1}{V}\frac{1}{T}\left[Q_0\delta\left(\frac{\omega}{V}-k\right) - \hat{R}\frac{V}{L}\sum_{n=-\infty}^{\infty}\delta\left(\frac{\omega-\omega_n}{V}-k\right)\right]$$

$$\Delta_b(\omega,k) = k^2 - \frac{\rho A}{T}\omega^2$$

$$\omega_n = n\frac{2\pi V}{L}$$

- Solution for generic R (omega, x domain)

$$\hat{w}(x,\omega) = \frac{1}{V}\frac{1}{T}\left[\frac{1}{\Delta_b(\omega,V)}Q_0 e^{-i\frac{\omega}{V}x} - \hat{R}\frac{V}{L}\sum_{n=-\infty}^{\infty}\frac{1}{\Delta_b(\omega,\omega_n,V)}e^{-i\frac{(\omega-\omega_n)}{V}x}\right]$$

$$\Delta_b(\omega,V) = \left(\frac{\omega}{V}\right)^2 - \frac{\rho A}{T}\omega^2$$

$$\Delta_b(\omega,\omega_n,V) = \left(\frac{\omega-\omega_n}{V}\right)^2 - \frac{\rho A}{T}\omega^2$$

- Solution for generic R at origin

$$\hat{w}(0,\omega) = \frac{1}{V}\frac{1}{T}\left[\frac{1}{\Delta_b(\omega,V)}Q_0 - \hat{R}\frac{V}{L}\sum_{n=-\infty}^{\infty}\frac{1}{\Delta_b(\omega,\omega_n,V)}\right]$$

- Use following definition to get rid of infinite sum

$$\sum_{n=-\infty}^{\infty}\frac{1}{\left(\frac{\omega L}{V}-2\pi n\right)^2-(\kappa L)^2} = \frac{1}{2\kappa L}\frac{\sin(\kappa L)}{\cos(\kappa L)-\cos\left(\frac{\omega}{V}L\right)}$$

$$\kappa = \frac{\omega}{\sqrt{T/\rho A}}$$

$$\Delta_b(\omega,\omega_n,V) = \left(\frac{\omega-\frac{2\pi n V}{L}}{V}\right)^2 - \kappa^2$$

$$L^2\Delta_b(\omega,\omega_n,V) = \left(\left(\frac{\omega}{V}-\frac{2\pi n}{L}\right)^2-\kappa^2\right)L^2$$

$$L^2\Delta_b(\omega,\omega_n,V) = \left(\frac{\omega L}{V}-2\pi n\right)^2-(\kappa L)^2$$

$$\frac{1}{\Delta_b(\omega,\omega_n,V)} = \frac{L^2}{\left(\frac{\omega L}{V}-2\pi n\right)^2-(\kappa L)^2}$$

$$\sum_{n=-\infty}^{\infty}\frac{1}{\Delta_b(\omega,\omega_n,V)} = \frac{L^2}{2\kappa L}\frac{\sin(\kappa L)}{\cos(\kappa L)-\cos\left(\frac{\omega}{V}L\right)}$$

- Solution for generic R at origin

$$\hat{w}(0,\omega) = \frac{1}{V}\frac{1}{T}\left[\frac{1}{\Delta_b(\omega,V)}Q_0 - \hat{R}\frac{V}{L}\frac{L}{2\kappa}\frac{\sin(\kappa L)}{\cos(\kappa L)-\cos\left(\frac{\omega}{V}L\right)}\right]$$

- Specific reaction force

$$\hat{R} = (k_s + i\omega\eta_s)\hat{w}(0,\omega)$$

Now we can solve for $\hat{w}(0, \omega)$ with sympy.

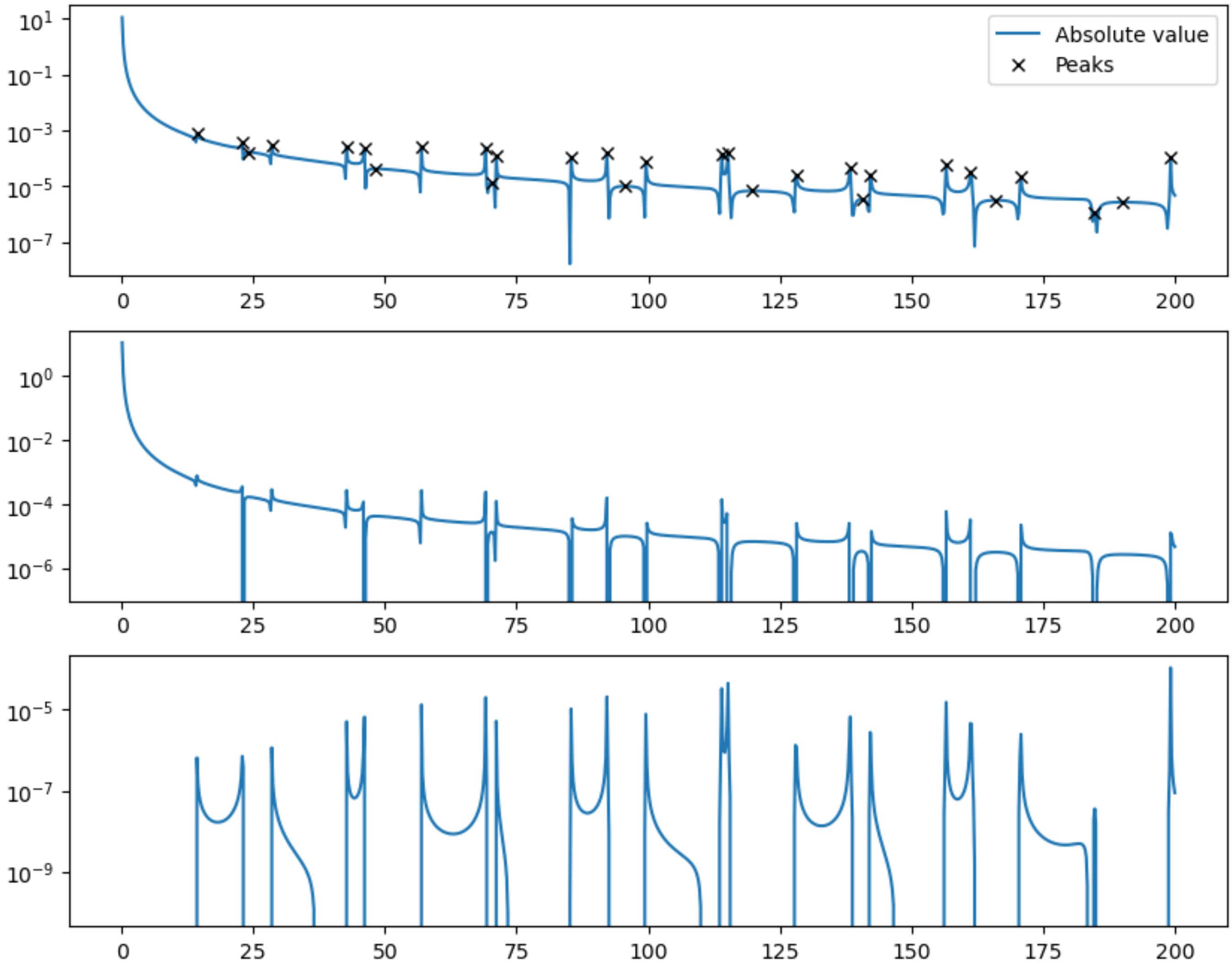
In [30]:
```
rhoA, V, T, Q0, L, omega, k_s, eta_s = sym.symbols('rhoA V T Q0 L omega k_s eta_s')

values = {rhoA:1.1, T:15E3, L:10, k_s:4E3, eta_s:0.5, Q0:55, V:28}

w_hat = sym.Function('w_hat')(omega)

delta_b = (omega/V)**2 - rhoA/T * omega**2
R_hat = (k_s + sym.I*eta_s*omega)*w_hat
kappa = omega/sym.sqrt(T/rhoA)

eq1 = sym.Eq(w_hat, 1/V * 1/T * (1/delta_b * Q0 - R_hat * V/2*kappa * sym.sin(kappa*L)/(sym.cc
```

In [31]:
```
display(eq1.subs(values))

w_hat = sym.solve(eq1.subs(values), w_hat)[0]
```

$$w_{hat}(\omega) = -\frac{2.85449612859225 \cdot 10^{-7}\omega\,(0.5i\omega + 4000.0)\, w_{hat}(\omega)\sin(0.0856348838577675\omega)}{\cos(0.0856348838577675\omega) - \cos\left(\frac{5\omega}{14}\right)} + \frac{0.10892937980986}{\omega^2}$$

In [32]:
```
from scipy.signal import find_peaks

w_hat_lambda = sym.lambdify(omega, w_hat)
omega_values = np.linspace(0.1, 200, 1000)

fig, ax = plt.subplots(3, 1, figsize=(10, 8))

ax[0].semilogy(omega_values, np.abs(w_hat_lambda(omega_values)), label='Absolute value')
ax[1].semilogy(omega_values, np.real(w_hat_lambda(omega_values)), label='Real part')
ax[2].semilogy(omega_values, np.imag(w_hat_lambda(omega_values)), label='Imag part')

# Find peaks in the absolute value of w_hat
abs_vals = np.abs(w_hat_lambda(omega_values))
peaks, _ = find_peaks(abs_vals)

# Plot the peaks on the absolute value plot
ax[0].plot(omega_values[peaks], abs_vals[peaks], "kx", label='Peaks')
ax[0].legend()

print(omega_values[peaks])
```

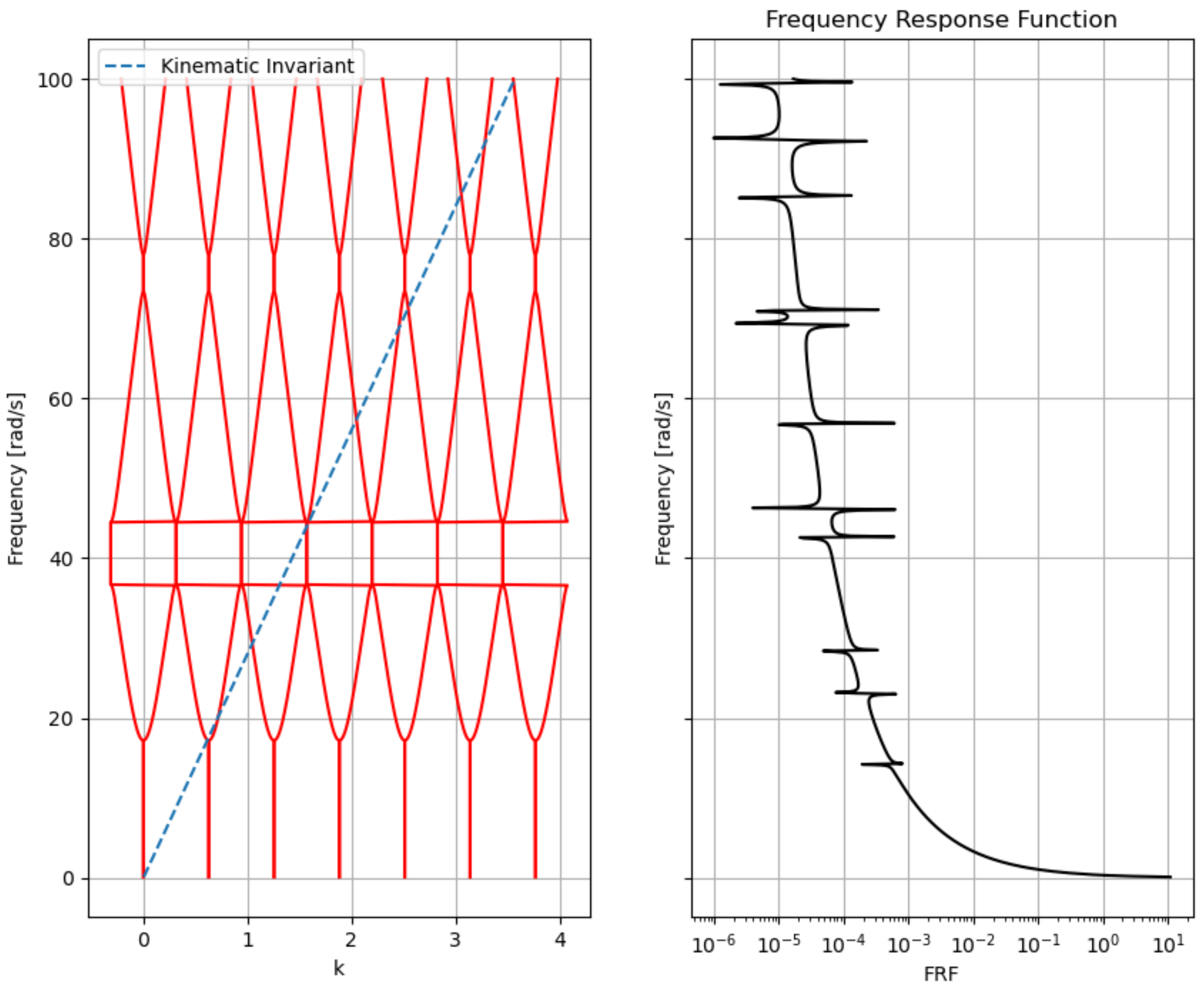
```
[ 14.30710711  22.91141141  24.11201201  28.51421421  42.72132132
  46.12302302  48.32412412  56.92842843  69.13453453  70.33513514
  71.13553554  85.34264264  92.14604605  95.54774775  99.54974975
 113.95695696 115.15755756 119.75985986 128.16406406 138.36916917
 140.57027027 142.17107107 156.57827828 161.18058058 165.98298298
 170.78538539 184.79239239 189.99499499 199.1995996 ]
```



In [41]:

```
omega_values = np.linspace(0.1, 100, 1000, dtype=complex)
kappa_values = omega_values/np.sqrt(15E3/1.1)


fig, ax = plt.subplots(1,2, figsize=(10, 8), sharey=True)

for i in range(7):
    ax[0].plot(np.real(k1_lambda(kappa_values))+2*i*np.pi/10, omega_values, 'r')
    ax[0].plot(np.real(k2_lambda(kappa_values))+2*i*np.pi/10, omega_values, 'r')

ax[0].plot(omega_values/28, omega_values, '--', label='Kinematic Invariant')
ax[0].set_xlabel('k')
ax[0].set_ylabel('Frequency [rad/s]')
ax[0].grid()
ax[0].legend()

ax[1].semilogx(np.abs(w_hat_lambda(omega_values)), omega_values, 'black', label='FRF');
ax[1].set_xlabel('FRF')
ax[1].set_ylabel('Frequency [rad/s]')
ax[1].set_title('Frequency Response Function')
ax[1].grid()
```

```
c:\Users\bart\anaconda3\envs\slender\Lib\site-packages\matplotlib\cbook.py:1762: ComplexWarnin
g: Casting complex values to real discards the imaginary part
  return math.isfinite(val)
c:\Users\bart\anaconda3\envs\slender\Lib\site-packages\matplotlib\cbook.py:1398: ComplexWarnin
g: Casting complex values to real discards the imaginary part
  return np.asarray(x, float)
```



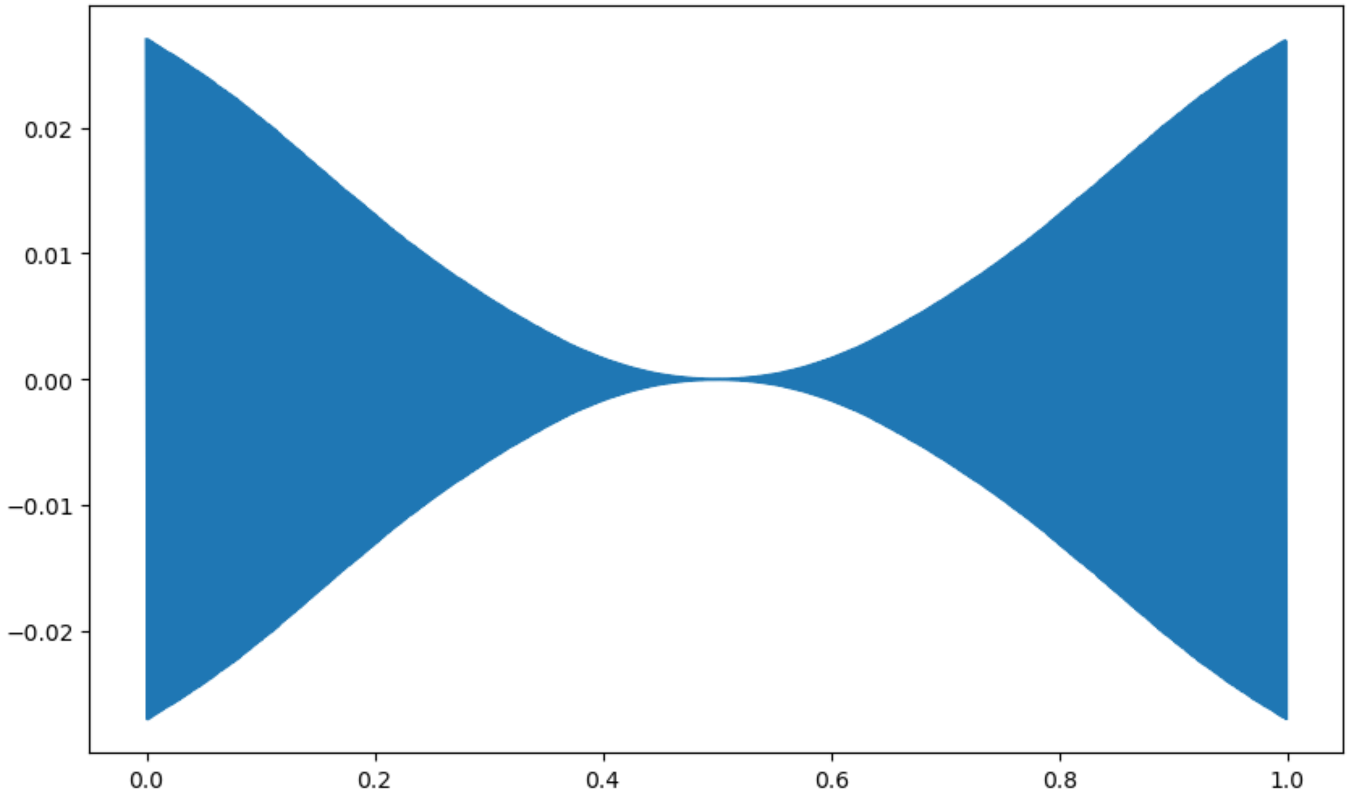
The obtained FRF is plotted next to the dispersion curve and the kinematic invariant. It is expected that the crossing of the kinematic invariant and the dispersion curve correspond to peaks in the FRF. For higher frequencies they match perfectly, however for lower frequencies this is not the case. We were not able to find the reason why this is the case. When increasing the range to even higher frequencies, the peaks keep on matching with the crossing locations.

question D

Unfortunately, we were not able to anwer question d. We tried using an ifft on the FRF, but the obtained results were not logical at all:

In [43]:
```
N = 1000

fs = 1000
df = fs/N
frequency_values = np.linspace(-fs/2, fs/2, N, dtype=complex)
omega_values = 2 * np.pi * frequency_values

dt = 1 / fs

w_hat = w_hat_lambda(omega_values)

w_time = np.fft.fft(w_hat)

time = np.arange(N) * dt

plt.figure(figsize=(10, 6))
# plt.plot(time, w_time.imag)
```

```python
plt.plot(time, w_time.real);
```



In [ ]:

In [ ]:

In [ ]: