```python
class BaseCell:
    def __init__(self, E:int,I:float,A:float, L:float,rho:float,n:int,k:float,m:float):
        """
        Creates a BaseCell object

        Creates an object of a metamaterial basecell with input parametres.
        :param E: Young module (GPa)
        :param I: Moment of inertia (mm4)
        :param A: Cross-section (mm3)
        :param L: Lenght (mm)
        :param rho: Density of material (kg/m3)
        :param n: Discretization
        :param k: Stiffness of resonator (N/m)
        :param m: Mass of resonator unit (kg)
        """
        self.E = E*1000
        self.I = I*10**(-12)
        self.L = L*10**(-3)
        self.A = A*10**(-6)
        self.rho = rho
        self.n = n
        self.k = k
        self.m = m

        self.DOF = 4

        #Lenght of finite element
        self.l = self.L/self.n

    def f0(self):
        return (1/(2*np.pi))*(np.sqrt(self.k/self.m))

    def Generate_Rred(self, mu):
        """
        Creates Right reduction matrix

        Creates a right reduction matrix for harmonic wave propagation in direction mu

        :param mu: Wave propagation vector
        """
        self.mu = mu
        return np.block(
            [[np.identity(2)],
             [np.identity(2)*np.e**(1j*self.mu)],
            ])

        """FINAL VARIANT
        return np.block(
            [[np.identity(2),                       np.zeros((2,3))],
             [np.identity(2)*np.e**(1j*self.mu),    np.zeros((2,3))],
             [np.zeros((3,2)),                      np.identity(3)]])"""

    def Generate_Lred(self, mu):
        """
        Creates Left reduction matrix

        Creates a left reduction matrix for harmonic wave propagation in direction mu

        :param mu: Wave propagation vector
        """
        self.mu = mu
        return np.block(
            [[np.identity(2)],
             [np.identity(2)*(np.e**(1j*self.mu))**(-1)],
            ]).T

        """FINAL VARIANT
        return np.block(
            [[np.identity(2),                              np.zeros((2,3))],
             [np.identity(2)*(np.e**(1j*self.mu))**(-1),    np.zeros((2,3))],
             [np.zeros((3,2)),                             np.identity(3)]]).T"""
```

```python
    def Generate_M(self):
        """
        Generates a local Mass matrix

        Generates a local Mass matrix for a Euler-Bernoulli beam finite element using parametres of an object
        """
        #Trenutno programirano za verzijo s štirimi prostostnimi stopnjami brez resonatorja
        len = self.l

        return np.array(
            [[156,          22*len,         54,         -13*len],
             [22*len,       4*len**2,       13*len,     -3*len**2],
             [54,           13*len,         156,        22*len],
             [-13*len,      156,            -22*len,    4*len**2]])*(self.rho*self.A*len)/420

    def Generate_K(self):
        """
        Generates a local Stiffness matrix

        Generates a local Stiffness matrix for a Euler-Bernoulli beam finite element using parametres of an
object
        """
        #Trenutno programirano za verzijo s štirimi prostostnimi stopnjami brez resonatorja
        len = self.l

        return np.array(
            [[12,           6*len,          -12,        6*len],
             [6*len,        4*len**2,       -6*len,     2*len**2],
             [-12,          -6*len,         12,         -6*len],
             [6*len,        2*len**2,       -6*len,     4*len**2]])*(self.E*self.I)/len**3

    def Generate_Mg(self):
        """
        Generates a global Mass matrix

        Generates a global Mass matrix for a Euler-Bernoulli beam finite element using parametres of an object
        """
        #Trenutno programirano za verzijo s štirimi prostostnimi stopnjami brez resonatorja
        M = self.Generate_M()
        dim = self.DOF*self.n - (self.n-1)*2

        #Create empty matrix
        Mg = np.zeros((dim,dim))

        #Adding up local matrices (Nodes have 2 DOF in common)
        for i in range(self.n):
            #Coordinates
            if i != 0:
                x = y = i*(self.DOF-2)
            else:
                x = y = 0
            MgC = Mg.copy()
            Mg[x:x+self.DOF,y:y+self.DOF] = np.add(MgC[x:x+self.DOF,y:y+self.DOF], M)

        return Mg

    def Generate_Kg(self):
        """
        Generates a global Stiffness matrix

        Generates a global Stiffness matrix for a Euler-Bernoulli beam finite element using parametres of an
object
        """
        #Trenutno programirano za verzijo s štirimi prostostnimi stopnjami brez resonatorja
        K = self.Generate_K()
        dim = self.DOF*self.n - (self.n-1)*2

        #Create empty matrix
        Kg = np.zeros((dim,dim))

        #Adding up local matrices (Nodes have 2 DOF in common)
        for i in range(self.n):
```

```
            #Coordinates
            if i != 0:
                x = y = i*(self.DOF-2)
            else:
                x = y = 0
            KgC = Kg.copy()
            Kg[x:x+self.DOF,y:y+self.DOF] = np.add(KgC[x:x+self.DOF,y:y+self.DOF], K)

        return Kg

    def Generate_Lredg(self,mu):
        """
        Generates a global left reduce matrix
        Generates a global left reduce matrix for a Euler-Bernoulli beam finite element using parametres of an
object
        """
        #Trenutno programirano za verzijo s štirimi prostostnimi stopnjami brez resonatorja
        Lred = self.Generate_Lred(mu)
        dim1 = (self.DOF-2)*self.n
        dim2 = self.DOF*self.n - (self.n-1)*2

        #Create empty matrix
        Lredg = np.zeros((dim1,dim2),dtype='complex_')

        #Adding up local matrices (Nodes have 2 DOF in common)
        for i in range(self.n):
            #Coordinates
            if i != 0:
                x = y = i*(self.DOF-2)
            else:
                x = y = 0
            LredgC = Lredg.copy()
            Lredg[x:x+self.DOF-2,y:y+self.DOF] = np.add(LredgC[x:x+self.DOF-2,y:y+self.DOF], Lred)

        return Lredg

    def Generate_Rredg(self,mu):
        """
        Generates a global right reduce matrix
        Generates a global right reduce matrix for a Euler-Bernoulli beam finite element using parametres of an
object
        """
        #Trenutno programirano za verzijo s štirimi prostostnimi stopnjami brez resonatorja
        Rred = self.Generate_Rred(mu)
        dim1 = self.DOF*self.n - (self.n-1)*2
        dim2 = (self.DOF-2)*self.n

        #Create empty matrix
        Rredg = np.zeros((dim1,dim2),dtype='complex_')

        #Adding up local matrices (Nodes have 2 DOF in common)
        for i in range(self.n):
            #Coordinates
            if i != 0:
                x = y = i*(self.DOF-2)
            else:
                x = y = 0
            RredgC = Rredg.copy()
            Rredg[x:x+self.DOF,y:y+self.DOF-2] = np.add(RredgC[x:x+self.DOF,y:y+self.DOF-2], Rred)

        return Rredg

    def Generate_Mgred(self,Mg, mu):
        """
        Generates a reduced global Mass matrix
        Generates a reduced global Mass matrix for a Euler-Bernoulli beam finite element using parametres of an
object
        """
        Mg = self.Generate_Mg()
```

```python
        Lredg = self.Generate_Lredg(mu)
        Rredg = self.Generate_Rredg(mu)

        return np.matmul(np.matmul(Lredg, Mg), Rredg)

    def Generate_Kgred(self,Kg, mu):
        """
        Generates a reduced global Stiffness matrix

        Generates a reduced global Stiffness matrix for a Euler-Bernoulli beam finite element using parametres of
an object
        """
        Kg = self.Generate_Kg()
        Lredg = self.Generate_Lredg(mu)
        Rredg = self.Generate_Rredg(mu)

        return np.matmul(np.matmul(Lredg, Kg), Rredg)

    def Generate_Mginv(self,Mg, mu):
        """
        Generates an inversed global reduced Mass matrix
        """
        return np.linalg.inv(self.Generate_Mgred(Mg, mu))

    def Calc_omega(self, mu:float, Mg, Kg):
        """
        Calculates frequencies for mu and Inversed mass matrix

        :param mu: mu
        :param Mg: Global Mass Matrix
        :param Kg: GLobal Stiffness Matrix
        """
        Kgred = self.Generate_Kgred(Kg, mu)
        Mginv = self.Generate_Mginv(Mg, mu)

        Dg = np.matmul(Mginv,Kgred)

        return lin.eig(Dg)
```