```python
"""
    @Author: Margarita Kuvaldina
    @https://github.com/margkuval
    @date: May 2018
"""

"""SOLVER FOR ANY TRUSS IN GENETIC ALGORITHM - deflections, stress, weight"""

import numpy as np


def deflection(xcoord, ycoord, mem_begin, mem_end, numelem, E, A, F, dof):

    "Link x and y coordinates with member's beginning and end"""
    xi = xcoord[np.ix_(mem_begin)]
    xj = xcoord[np.ix_(mem_end)]  # take mem_end numbers and replace them with corresponding xcoord
    yi = ycoord[np.ix_(mem_begin)]
    yj = ycoord[np.ix_(mem_end)]

    "Connectivity matrix"
    ij = np.vstack([[2 * mem_begin, 2 * mem_begin + 1], [2 * mem_end, 2 * mem_end + 1]]).transpose()

    "Other information"
    numnode = xcoord.shape[0]  # number of nodes, xcoord because all nodes are used
    dof_tot = 2 * numnode  # total degrees of freedom

    """Global Stiffness Matrix"""
    glob_stif = np.zeros((dof_tot, dof_tot))  # empty Global Stiffness Matrix
    length = np.sqrt(pow((xj - xi), 2) + pow((yj - yi), 2))  # defines length of members
    c = (xj - xi) / length  # cos
    s = (yj - yi) / length  # sin

    for p in range(numelem):
        # takes p from the range of numelem 1 by 1 and creates multiple k1 (local) matrices
        # at the end maps k1 matrices on right places in glob_stiff matrix
        n = ij[p]
        cc = c[p] * c[p]
        cs = c[p] * s[p]
        ss = s[p] * s[p]
        k1 = E[p] * A[p] / length[p] * np.array([[cc, cs, -cc, -cs],
                                                  [cs, ss, -cs, -ss],
                                                  [-cc, -cs, cc, cs],
                                                  [-cs, -ss, cs, ss]])
        glob_stif[np.ix_(n, n)] += k1

    """Forces and deflections"""

    "Fixed and active DOFs"
    dof_a = np.array(np.where(dof == 0))  # node where dof = 0 is an active node
    dof_active = dof_a[0]

    "Solve deflections"
    u = np.zeros((dof_tot, 1))  # empty deflections matrix; 1 = number of columns
    u1 = np.linalg.solve(glob_stif[np.ix_(dof_active, dof_active)],
                         F[np.ix_(dof_active)])  # solve equation glob_stif*u = F
    u[np.ix_(dof_active)] = u1  # map back to the empty deflection matrix
    deflection = np.round(u, 3)

    return deflection


def stress(xcoord, ycoord, mem_begin, mem_end, E, A, F, dof, deflection):

    "Link x and y coordinates with member's beginning and end"""
    xi = xcoord[np.ix_(mem_begin)]
    xj = xcoord[np.ix_(mem_end)]
    yi = ycoord[np.ix_(mem_begin)]
    yj = ycoord[np.ix_(mem_end)]

    "Other information"
    numnode = xcoord.shape[0]

    "Reshaped outside forces and DOFs"
    F_x2 = F.reshape(numnode, 2)
    dof_x2 = dof.reshape(numnode, 2)  # reshape for plotting

    """Stress calculation"""

    "Deflections in x, y directions"
    # using mem_end and mem_begin to calculate new nodes location
    length = np.sqrt(pow((xj - xi), 2) + pow((yj - yi), 2))  # members length
    k = E * A / length

    u = deflection
    uxi = u[np.ix_(2 * mem_begin)].transpose()
    uxj = u[np.ix_(2 * mem_end)].transpose()
    uyi = u[np.ix_(2 * mem_begin + 1)].transpose()
    uyj = u[np.ix_(2 * mem_end + 1)].transpose()

    "Inner forces"
```

```
    c = (xj - xi) / length  # cos
    s = (yj - yi) / length  # sin

    Flocal = k * ((uxj - uxi) * c + (uyj - uyi) * s)

    "Stress (kPa)"
    stress = Flocal[0] / A
    stress_normed = [i / sum(abs(stress)) for i in abs(stress)]

    xinew = xi + uxi[0]  # [[ in u array, now solved by taking "list 0" from the MAT
    xjnew = xj + uxj[0]
    yinew = yi + uyi[0]
    yjnew = yj + uyj[0]

    return stress, stress_normed, xi, xj, yi, yj, xinew, xjnew, yinew, yjnew, F_x2, numnode, dof_x2, length


def weight(A, length, ro):

    "Density of each element (1000 kg/m3)"""
    # reinforced concrete = 2500 kg/m3, steel = 7700 kg/m3
    # defined in GA

    "Weigth calculation"
    weight = length * A * ro

    return weight
```

```python
"""
    @Author: Margarita Kuvaldina
    @https://github.com/margkuval
    @date: May 2018
"""

import numpy as np
import random as rnd
import matplotlib.pyplot as plt
import solver_univ as slv
from matplotlib.gridspec import GridSpec
import datetime

"""GENETIC ALGORITHM 1x4 TRUSS"""

# CH = change if implementing on a new structure


class Individual:
    def __init__(self):
        "Structural dimentions (m)"""
        # CH = change if implementing on a new structure
        a = 2  # CH
        h = a  # triangle height  # CH

        "Original coordinates"
        xcoord = np.array([0, a, 2.5 * a, 4 * a, 5 * a, a, 2.5 * a, 4 * a])  # CH
        ycoord = np.array([0, 0, 0, 0, 0, h, h, h])  # CH

        "Choose a random number in range +- (m) from the original coordinate"
        x2GA = rnd.randrange(np.round((xcoord[2] - 0.7) * 100), np.round((xcoord[2] + 0.7) * 100)) / 100  # CH
        x1GA = rnd.randrange(np.round((xcoord[1] - 0.7) * 100), np.round((xcoord[1] + 0.7) * 100)) / 100
        x3GA = rnd.randrange(np.round((xcoord[3] - 0.7) * 100), np.round((xcoord[3] + 0.7) * 100)) / 100

        "New coordinates"
        xcoord = np.array([0, x1GA, x2GA, x3GA, 5 * a, a, 2.5 * a, 4 * a])  # CH
        ycoord = np.array([0, 0, 0, 0, 0, h, h, h])  # can use np.ix_?      # CH

        "Cross-section area (m)"
        self.A = np.random.uniform(low=0.0144, high=0.0529, size=(13,))    # area between 12x12 and 23x23cm       # CH
        self.A[0] = rnd.randrange(0.0004 * 10000, 0.0064 * 10000) / 10000  # special condition for steel elements # CH
        self.A[1] = rnd.randrange(0.0004 * 10000, 0.0064 * 10000) / 10000
        self.A[2] = rnd.randrange(0.0004 * 10000, 0.0064 * 10000) / 10000
        self.A[3] = rnd.randrange(0.0004 * 10000, 0.0064 * 10000) / 10000
        self.A[11] = rnd.randrange(0.0004 * 10000, 0.0064 * 10000) / 10000

        "Material characteristic E=(MPa), ro=kg/m3)"  # CH
        # modulus of elasticity for each member, E_reinforced_concrete C30/37 = 33 000 MPa, E_steel S235 = 210 000 MPa
        self.E = np.array([33000, 33000, 33000, 33000, 33000, 33000, 33000, 33000, 33000, 33000, 33000, 33000, 33000])
        self.E[0] = 210000
        self.E[1] = 210000
        self.E[2] = 210000
        self.E[3] = 210000
        self.E[11] = 210000
        self.E[12] = 210000

        # density for each member, ro_reinforced_concrete C30/37 = 2 600 kg/m3, ro_steel S235= 7850 kg/m3
        self.ro = np.array([2600, 2600, 2600, 2600, 2600, 2600, 2600, 2600, 2600, 2600, 2600, 2600, 2600])/1000
        self.ro[0] = 7850
        self.ro[1] = 7850
        self.ro[2] = 7850
        self.ro[3] = 7850
        self.ro[11] = 7850
        self.ro[12] = 7850

        self._plot_dict = None
        self._nodes = np.array([xcoord, ycoord])
        self._deflection = 0
        self._stress = 0
        self._weight = 0
        self._fitness = 0
        self._probability = 0

    @property
    def deflection(self):
        return self._deflection

    @deflection.setter
    def deflection(self, new):
        self._deflection = new

    @property
    def stress(self):
        return self._stress

    @stress.setter
    def stress(self, new):
        self._stress = new

    @property
```

```
    def weight(self):
        return self._weight

    @weight.setter
    def weight(self, new):
        self._weight = new

    @property
    def fitness(self):
        return self._fitness

    @fitness.setter
    def fitness(self, new):
        self._fitness = new

    @property
    def probability(self):
        return self._probability

    @probability.setter
    def probability(self, new):
        self._probability = new


class GA:
    def __init__(self, pop):
        self.mem_begin = np.array([0, 1, 2, 3, 4, 7, 6, 5, 1, 5, 6, 2, 7])  # beginning of an edge   # CH
        self.mem_end   = np.array([1, 2, 3, 4, 7, 6, 5, 0, 5, 2, 2, 7, 3])  # end of an edge         # CH

        self._pool = list()
        self._popsize = pop

    def initial(self):
        for i in range(self._popsize):
            self._pool.append(Individual())
            print("node_1:{} node_2:{} node_3:{}".format(
                                                     np.round([self._pool[i]._nodes[0, 1], self._pool[i]._nodes[1, 1]], 3),
                                                     np.round([self._pool[i]._nodes[0, 2], self._pool[i]._nodes[1, 2]], 3),
                                                     np.round([self._pool[i]._nodes[0, 3], self._pool[i]._nodes[1, 3]], 3))
)
        print("......................")

    def calculation(self):
        numelem = len(self.mem_begin)  # count number of beginnings

        """Structural characteristics"""

        "Fixed Degrees of Freedom (DOF)"  # CH
        dof = np.zeros((2 * len(np.unique(self.mem_begin)), 1))  # dof vector    # counts unique values in mem_begin
        dof[0] = 1  # 1 = fixed
        dof[1] = 1
        dof[9] = 1

        "Outside Forces [kN]"  # CH
        F = np.zeros((2 * len(np.unique(self.mem_begin)), 1))     # forces vector # counts unique values in mem_begin
        F[10] = 10
        F[11] = -15
        F[13] = -5
        F[14] = 10
        F[15] = -15

        print("calculation ")

        """Access solver"""
        for i in range(self._popsize):

            "DEFLECTION"
            pool = self._pool[i]
            res = slv.deflection(pool._nodes[0], pool._nodes[1], self.mem_begin, self.mem_end, numelem,
                                 pool.E, pool.A, F, dof)
            deflection = res
            pool._deflection = deflection
            pool._probability = 0

            "STRESS"
            # globbing, to "res" save everything that slv.stress returns (tuple)
            res = slv.stress(pool._nodes[0], pool._nodes[1],
                             self.mem_begin, self.mem_end,
                             pool.E, pool.A, F, dof, deflection)
            stress, stress_normed, xi, xj, yi, yj, xinew, xjnew, yinew, yjnew, F_x2, numnode, dof_x2, length = res
            pool._stress        = stress
            pool._stress_normed = stress_normed
            pool._stress_max    = np.round(np.max(pool._stress), 3)

            plot_dict = {"xi": xi, "xj": xj, "yi": yi, "yj": yj, "xinew": xinew, "xjnew": xjnew, "yinew": yinew,
                         "yjnew": yjnew, "F_x2": F_x2, "dof_x2": dof_x2, "stress_normed": stress_normed,
                         "numnode": numnode, "numelem": numelem, "A": pool.A}
            pool._plot_dict = plot_dict

            "WEIGHT"
```

```python
            pool._weight = slv.weight(pool.A, length, pool.ro)

            print("node_1:{} node_2:{} node_3:{} |def| sum:{} |stress| sum:{} |weight| sum:{}".format(
                np.round([pool._nodes[0, 1], pool._nodes[1, 1]], 3),
                np.round([pool._nodes[0, 2], pool._nodes[1, 2]], 3),
                np.round([pool._nodes[0, 3], pool._nodes[1, 3]], 3),
                np.round(abs(pool._deflection).sum(), 3),
                np.round(abs(pool._stress).sum()),
                np.round(pool._weight.sum())))
            print(".....................")


    def fitness(self):
        fitnesses = []

        "Condition to disadvantage members with stress > E"
        # if the inner force is higher than member's strength, make its fitness worse
        for x in self._pool:
            for i in range(len(self.mem_begin)):
                for strength in self._pool[i].E:
                    if strength < abs(x._stress[i]):
                        x._stress[i] = x._stress[i] * 200

        """Rate / give fitness to each member"""
        print("fitness")  # higher fitness = better fitness

        "Conditions to find the best candidate"
        deflections = [(abs(x._deflection)).sum() for x in self._pool]  # sum of absolute deflections
        stresses = [sum(abs(x._stress)).sum() for x in self._pool]      # sum of absolute stresses
        weights = [sum(x._weight).sum() for x in self._pool]            # sum of weights

        "Importance coefficients for stated conditions"
        deflection_coef = 0.40
        stress_coef = 0.50
        weight_coef = 0.10

        "Fill the cell based on conditions and importance coefficients"
        for deflection, stress, weight in zip(deflections, stresses, weights):
            if weight.sum() < 0:
                print("Weight is negative!")
            else:
                fitnesses.append(1/(deflection_coef * deflection +
                                 stress_coef * stress * (min(deflections)/min(stresses)) +
                                 weight_coef * weight * (min(deflections)/min(weights))))
        sum_fit = sum(fitnesses)

        "Fitness of each candidate"
        len_sf = len(self._pool)
        for i in range(len_sf):
            self._pool[i]._fitness = fitnesses[i]

            "Probability record"
            # member with higher fit (= better) has a higher probability to be chosen for the crossover
            self._pool[i]._probability = fitnesses[i] / sum_fit

        "Sort members based on probability"
        # sorting in Python is ascending (if "-x._fitness", is descending)
        self._pool.sort(key=lambda x: -x._fitness)

        "Print results"
        for i in range(self._popsize):
            pool = self._pool[i]
            print("node_1:{} node_2:{} node_3:{} fit:{}  prob:{} "
                  "|def(mm)| sum:{} |stress(kPa)| sum:{} |weight(t)| sum:{}".format(
                np.round([pool._nodes[0, 1], pool._nodes[1, 1]], 3),
                np.round([pool._nodes[0, 2], pool._nodes[1, 2]], 3),
                np.round([pool._nodes[0, 3], pool._nodes[1, 3]], 3),
                np.round(pool._fitness, 3),
                np.round(pool._probability, 3),
                np.round(abs(pool._deflection).sum(), 3),
                np.round(abs(pool._stress).sum()),
                np.round(pool._weight.sum())))

        print("______________________")

    def get_best_fit(self):
        "Best fitness"
        best_obj= max(self._pool, key=lambda x: x._fitness)
        return np.round(best_obj.fitness, 3)

    def get_best_weight(self):
        "Best fitness - weight"
        best_obj= max(self._pool, key=lambda x: x._fitness)
        return np.round(sum(best_obj.weight), 3)

    def get_best_stress(self):
        "Best fitness - sum stress"
        best_obj= max(self._pool, key=lambda x: x._fitness)
        return np.round(sum(abs(best_obj.stress)), 3)

```

```
    def get_best_stress_negative(self):
        "Best fitness - negative stress"
        best_obj = max(self._pool, key=lambda x: x._fitness)
        best_stress_negative_sum = 0
        for num in best_obj.stress:
            if num < 0:
                best_stress_negative_sum += num
        return np.round(best_stress_negative_sum, 3)

    def get_best_stress_positive(self):
        "Best fitness - positive stress"
        best_obj = max(self._pool, key=lambda x: x._fitness)
        best_stress_positive_sum = 0
        for num in best_obj.stress:
            if num > 0:
                best_stress_positive_sum += num
        return np.round(best_stress_positive_sum, 3)

    def get_best_defl(self):
        "Best fitness - deflections"
        best_obj = max(self._pool, key=lambda x: x._fitness)
        return best_obj.deflection

    """def get_avg_fit(self):
        "Best fitness"
        best_obj= np.average(self._pool, key=lambda x: x._fitness)
        return np.round(best_obj.fitness, 3)

    def get_best_weight(self):
        "Best fitness - weight"
        best_obj= max(self._pool, key=lambda x: x._fitness)
        return np.round(sum(best_obj.weight), 3)

    def get_best_stress(self):
        "Best fitness - stress"
        best_obj= max(self._pool, key=lambda x: x._fitness)
        return np.round(sum(abs(best_obj.stress)), 3)

    def get_best_defl(self):
        "Best fitness - deflections"
        best_obj= max(self._pool, key=lambda x: x._fitness)
        return best_obj.deflection"""

    def _switch1(self, individual_pair, axis=0):
        # set switch values between 2 individuals (node 1)
        first = individual_pair[0]
        second = individual_pair[1]
        tmp = first._nodes[axis, 1]  # = temporary
        first._nodes[axis, 1] = second._nodes[axis, 1]
        second._nodes[axis, 1] = tmp

    def _switch2(self, individual_pair, axis=0):
        # set switch values between 2 individuals (node 2)
        first = individual_pair[0]
        second = individual_pair[1]
        tmp = first._nodes[axis, 2]  # temporary
        first._nodes[axis, 2] = second._nodes[axis, 2]
        second._nodes[axis, 2] = tmp

    def _switch3(self, individual_pair, axis=0):
        # set switch values between 2 individuals (node 3)
        first = individual_pair[0]
        second = individual_pair[1]
        tmp = first._nodes[axis, 3]  # temporary
        first._nodes[axis, 3] = second._nodes[axis, 3]
        second._nodes[axis, 3] = tmp

    def _switch_A(self, individual_pair, axis=0):
        # set switch values between 2 individuals (node 3)
        first = individual_pair[0]
        second = individual_pair[1]
        tmp = first._nodes[axis, 0]  # temporary
        first._nodes[axis, 0] = second._nodes[axis, 0]
        second._nodes[axis, 0] = tmp

    def crossover(self):
        probs = ([x._probability for x in self._pool])

        "Select the best and worst member"
        best = max(self._pool, key=lambda x: x._fitness)
        worst = min(self._pool, key=lambda x: x._probability)

        "Nodes crossover"
        # choose 2 individuals that will crossover
        switch_x0 = np.random.choice(self._pool, 2, replace=False, p=probs)
        switch_x1 = np.random.choice(self._pool, 2, replace=False, p=probs)
        switch_x2 = np.random.choice(self._pool, 2, replace=False, p=probs)
        #switch_y = np.random.choice(self._pool, 2, replace=False, p=probs)

        "Areas Crossover"
```

```python
        switch_a = np.random.choice(self._pool, 2, replace=False, p=probs)

        self._switch1(switch_x0, 0)
        self._switch2(switch_x1, 0)
        self._switch3(switch_x2, 0)
        self._switch_A(switch_a, 0)
        #self._switch2(switch_y, 1)

        self._pool.remove(worst)
        self._pool.append(best)

    def mutation(self, mutation_type):
        probs = [x._probability for x in self._pool]

        fits  = np.array([x._fitness for x in self._pool])
        sum_f = sum(x._fitness for x in self._pool)
        if sum(probs) !=1:
            probs = abs(np.array(fits))/sum_f

        "Pick a mutation candidate"
        mutation_candidate = np.random.choice(self._pool, 1, p=probs)[0]

        possible_coefficients = [0.9, 0.9, 0.9, 1.1, 1.1, 1.1, 0.8, 0.85, 0.75, 1.3, 1.2, 1.2]
        coef = np.random.choice(possible_coefficients, 1)  # choose one from the list above

        "Best member included in population"
        best = max(self._pool, key=lambda x: x._fitness)
        worst = min(self._pool, key=lambda x: x._fitness)

        "Mutate"
        for i in range(rnd.randrange(1, 2, 3)):  # CH - 3 nodes for 1x4
            if mutation_type == "x":
                mutation_candidate._nodes[0, i] = mutation_candidate._nodes[0, i] * coef
            if mutation_type == "y":
                mutation_candidate._nodes[1, i] = mutation_candidate._nodes[1, i] * coef
            break

        if mutation_type == "a":
            for i in range(rnd.randrange(len(self.mem_begin))):
                cur_candidate = self._pool[i]
                if cur_candidate == best:
                    self._pool[-1] = best
                    self._pool[-1].probability = 1 - sum(x._probability for x in self._pool[:(len(self._pool))])
                se = np.argmin(self._pool[i]._stress)
                if cur_candidate.A[se] > 0.0001:
                    continue
                cur_candidate.A[se] = cur_candidate.A[se] * coef
                break

        self._pool.remove(worst)
        self._pool.append(best)

    def plot_stress(self):
        num_to_plot = 4

        gs = GridSpec(1, 4)
        gs.update(left=0.05, right=0.95, wspace=0.2)
        fig = plt.figure(figsize=(18, 5))
        fig.suptitle("Best members in generation - stress")

        for index in range(num_to_plot):
            # take num_to_plot best candidates, load data from saved dict
            pool = self._pool[index]
            plot_dict = pool._plot_dict
            stress = pool._stress
            xi = plot_dict['xi']
            xj = plot_dict['xj']
            yi = plot_dict['yi']
            yj = plot_dict['yj']
            xinew = plot_dict['xinew']
            xjnew = plot_dict['xjnew']
            yinew = plot_dict['yinew']
            yjnew = plot_dict['yjnew']
            stress_normed = plot_dict['stress_normed']
            F_x2 = plot_dict['F_x2']
            dof_x2 = plot_dict['dof_x2']
            numnode = plot_dict['numnode']
            numelem = plot_dict['numelem']

            ax = fig.add_subplot(gs[0, index], aspect="equal")

            ax.grid(True)
            ax.set_xlim(-2, 11)   # CH
            ax.set_ylim(-2.5, 5)  # CH
            ax.set_title("Candidate {}".format(index + 1))

            for r in range(numelem):
                x = (xi[r], xj[r])
                y = (yi[r], yj[r])
                line = ax.plot(x, y)
```

```python
                plt.setp(line, ls='-', c='black', lw='1', label='orig')
                xnew = (xinew[r], xjnew[r])
                ynew = (yinew[r], yjnew[r])
                linenew = ax.plot(xnew, ynew)

                plt.setp(linenew, ls='-',
                         c='c' if stress[r] > 0.000001 else ('red' if stress[r] < -0.000001 else 'black'),
                         lw=(1 + 20 * stress_normed[r]), label='strain' if stress[r] > 0 else 'stress')
                ax.plot()

            "Annotate outside forces"
            for r in range(numnode):
                plt.annotate(F_x2[r],
                             xy=(xi[r], yi[r]), xycoords='data', xytext=np.sign(F_x2[r]) * -35,
                             textcoords='offset pixels',
                             arrowprops=dict(facecolor='black', shrink=0, width=1.3, headwidth=5),
                             horizontalalignment='right', verticalalignment='bottom')

            "Annotate fixed DOFs"
            for r in range(numnode):
                if np.array_equal(dof_x2[r], np.array([0, 1])):
                    plt.plot([xi[r]], [yi[r] - 0.2], 'o', c='k', markersize=8)
                if np.array_equal(dof_x2[r], np.array([1, 0])):
                    plt.plot([xi[r] - 0.2], [yi[r]], 'o', c='k', markersize=8)
                if np.array_equal(dof_x2[r], np.array([1, 1])):
                    plt.plot([xi[r]], [yi[r] - 0.2], '^', c='k', markersize=8)

        plt.savefig(datetime.datetime.now().
            strftime('stress_1x4_%Y%m%d_%H%M%S_pop300_cyc200_mx50_myA_45') + ".pdf")

        #plt.show()

    def plot_A(self):
        num_to_plot = 4
        gs = GridSpec(1, 4)  # 1 column, 4 in row
        gs.update(left=0.05, right=0.95, wspace=0.1)

        fig = plt.figure(figsize=(18, 5))
        fig.suptitle("Best members in generation - cross section")

        for index in range(num_to_plot):
            # take num_to_plot best candidates, load data from saved dictionary
            pool = self._pool[index]
            plot_dict = pool._plot_dict

            xi = plot_dict['xi']
            xj = plot_dict['xj']
            yi = plot_dict['yi']
            yj = plot_dict['yj']
            xinew = plot_dict['xinew']
            xjnew = plot_dict['xjnew']
            yinew = plot_dict['yinew']
            yjnew = plot_dict['yjnew']
            stress_normed = plot_dict['stress_normed']
            F_x2 = plot_dict['F_x2']
            dof_x2 = plot_dict['dof_x2']
            numnode = plot_dict['numnode']
            numelem = plot_dict['numelem']

            ax = fig.add_subplot(gs[0, index], aspect="equal")
            ax.grid(True)
            ax.set_xlim(-2, 11)   # CH
            ax.set_ylim(-2.5, 5)  # CH
            ax.set_title("Candidate {}".format(index + 1))

            for r in range(numelem):
                x = (xi[r], xj[r])
                y = (yi[r], yj[r])
                line = ax.plot(x, y)
                plt.setp(line, ls='-', c='black', lw='1', label='orig')

                xnew = (xinew[r], xjnew[r])
                ynew = (yinew[r], yjnew[r])
                linenewA = ax.plot(xnew, ynew)

                plt.setp(linenewA, ls='-', c='green', lw=(1 + 70 * pool.A[r]))
            ax.plot()

            "Annotate outside forces"
            for r in range(numnode):
                plt.annotate(F_x2[r],
                             xy=(xi[r], yi[r]), xycoords='data', xytext=np.sign(F_x2[r]) * -35,
                             textcoords='offset pixels',
                             arrowprops=dict(facecolor='black', shrink=0, width=1.5, headwidth=8),
                             horizontalalignment='right', verticalalignment='bottom')

            "Annotate fixed DOFs"
            for r in range(numnode):
                if np.array_equal(dof_x2[r], np.array([0, 1])):
                    plt.plot([xi[r]], [yi[r] - 0.2], 'o', c='k', markersize=8)
```

```
            if np.array_equal(dof_x2[r], np.array([1, 0])):
                plt.plot([xi[r] - 0.2], [yi[r]], 'o', c='k', markersize=8)
            if np.array_equal(dof_x2[r], np.array([1, 1])):
                plt.plot([xi[r]], [yi[r] - 0.2], '^', c='k', markersize=8)

    plt.savefig(datetime.datetime.now().
                strftime('cross section_1x4_%Y%m%d_%H%M%S_pop300_cyc200_mx50_myA_45') + ".pdf")

    #plt.show()
```

```python
"""
    @Author: Margarita Kuvaldina
    @https://github.com/margkuval
    @date: May 2018
"""

import fc_1x4_GA as GA
import plots_univ as plt_uni
import numpy as np

"""BRAIN FOR GENETIC ALGORITHM 1x4 TRUSS"""

"Initial values - population, number of iterations, mutation 1, mutation 2, iteration of plotting"
inp_task_1 = np.array([40, 50, 10, 8, 25])

population_1 = inp_task_1[0]  # population size
num_cycles_1 = inp_task_1[1]  # number of computation cycles
mut_x_1 = inp_task_1[2]
mut_yA_1 = inp_task_1[3]
plt_s_A_1 = inp_task_1[4]

inp_task_2 = inp_task_1  #np.array([20, 20, 10, 9, 25])

population_2 = inp_task_2[0]  # population size
num_cycles_2 = inp_task_2[1]  # number of computation cycles
mut_x_2 = inp_task_2[2]
mut_yA_2 = inp_task_2[3]
plt_s_A_2 = inp_task_2[4]

inp_task_3 = inp_task_1  #np.array([20, 20, 10, 9, 25])

population_3 = inp_task_3[0]  # population size
num_cycles_3 = inp_task_3[1]  # number of computation cycles
mut_x_3 = inp_task_3[2]
mut_yA_3 = inp_task_3[3]
plt_s_A_3 = inp_task_3[4]

"Task number 1"
task = GA.GA(population_1)

list_iter = []
list_fit = []
list_weight = []
list_stress = []
list_stress_positive = []
list_stress_negative = []
list_defl = []

task.initial()
print("New task 1")
for i in range(num_cycles_1):
    task.calculation()
    task.fitness()
    if i % plt_s_A_1 == 0:
        task.plot_stress()
        task.plot_A()
    task.crossover()
    if i % mut_x_1 == 0:
        task.mutation(mutation_type="x")
    if i % mut_yA_1== 0:
        task.mutation(mutation_type="y")
        task.mutation(mutation_type="a")

    list_iter.append(i)

    task.get_best_fit()
    task.get_best_weight()
    task.get_best_stress()
    task.get_best_stress_positive()
    task.get_best_stress_negative()
    task.get_best_defl()

    list_fit.append(task.get_best_fit())
    list_weight.append(task.get_best_weight())
    list_stress.append(task.get_best_stress())
    list_stress_positive.append(task.get_best_stress_positive())
    list_stress_negative.append(task.get_best_stress_negative())
    list_defl.append(task.get_best_defl())

plt_best_1 = plt_uni.plot_best_1(list_iter, list_fit,list_stress_positive, list_stress_negative, list_weight, list_defl)

"Task number 2"
task_2 = GA.GA(population_2)  # population size

list_iter_2 = []
list_fit_2 = []
list_weight_2 = []
list_stress_2 = []
list_stress_positive_2 = []
list_stress_negative_2 = []
```

```python
list_defl_2 = []

task_2.initial()
print("New task 2")
for r in range(num_cycles_2):  # number of computation cycles
    task_2.calculation()
    task_2.fitness()
    if r % plt_s_A_2 == 0:
        task_2.plot_stress()
        task_2.plot_A()
    task_2.crossover()
    if r % mut_x_2 == 0:
        task_2.mutation(mutation_type="x")
    if r % mut_yA_2 == 0:
        task_2.mutation(mutation_type="y")
        task_2.mutation(mutation_type="a")

    list_iter_2.append(r)

    task_2.get_best_fit()
    task_2.get_best_weight()
    task_2.get_best_stress()
    task_2.get_best_stress_positive()
    task_2.get_best_stress_negative()
    task_2.get_best_defl()

    list_fit_2.append(task_2.get_best_fit())
    list_weight_2.append(task_2.get_best_weight())
    list_stress_2.append(task_2.get_best_stress())
    list_stress_positive_2.append(task_2.get_best_stress_positive())
    list_stress_negative_2.append(task_2.get_best_stress_negative())
    list_defl_2.append(task_2.get_best_defl())

plt_uni.plot_best_2(list_iter_2, list_fit_2, list_stress_positive_2, list_stress_negative_2, list_weight_2, list_defl_2)

"Task number 3"
task_3 = GA.GA(population_3)  # population size

list_iter_3 = []
list_fit_3 = []
list_weight_3 = []
list_stress_3 = []
list_stress_positive_3 = []
list_stress_negative_3 = []
list_defl_3 = []

task_3.initial()
print("New task 3")
for k in range(num_cycles_3):  # number of computation cycles
    task_3.calculation()
    task_3.fitness()
    if k % plt_s_A_3 == 0:
        task_3.plot_stress()
        task_3.plot_A()
    task_3.crossover()
    if k % mut_x_3 == 0:
        task_3.mutation(mutation_type="x")
    if k % mut_yA_3 == 0:
        task_3.mutation(mutation_type="y")
        task_3.mutation(mutation_type="a")

    list_iter_3.append(k)

    task_3.get_best_fit()
    task_3.get_best_weight()
    task_3.get_best_stress()
    task_3.get_best_stress_positive()
    task_3.get_best_stress_negative()
    task_3.get_best_defl()

    list_fit_3.append(task_3.get_best_fit())
    list_weight_3.append(task_3.get_best_weight())
    list_stress_3.append(task_3.get_best_stress())
    list_stress_positive_3.append(task_3.get_best_stress_positive())
    list_stress_negative_3.append(task_3.get_best_stress_negative())
    list_defl_3.append(task_3.get_best_defl())

plt_uni.plot_best_3(list_iter_3, list_fit_3, list_stress_positive_3, list_stress_negative_3, list_weight_3, list_defl_3)

plt_fits_3 = plt_uni.plot_fits_3(list_iter, list_iter_2, list_iter_3,
                list_fit, list_fit_2, list_fit_3,
                population_1, population_2, population_3,
                mut_x_1, mut_x_2, mut_x_3, mut_yA_1, mut_yA_2, mut_yA_3)
```

```python
"""
    @Author: Margarita Kuvaldina
    @https://github.com/margkuval
    @date: May 2018
"""

"""PLOT FOR ANY TRUSS IN BRAIN"""

import matplotlib.pyplot as plt
import numpy as np
import datetime


def plot_best_1(list_iter, list_fit, list_stress_positive, list_stress_negative, list_weight, list_defl):
    fig = plt.figure(figsize=(10, 8))
    fig.suptitle('Individual 1: the fittest individual development in time',
                 horizontalalignment='center', verticalalignment='center')

    "Fitness plot"
    list_fit = np.array(list_fit).transpose()
    x_fit = list_iter
    y_fit = list_fit

    ax1 = fig.add_subplot(2, 2, 1)
    ax1.plot(x_fit, y_fit, c='r')
    ax1.set_title('Fitness evolution')
    ax1.set_xlabel('Iterations')
    ax1.set_ylabel('Fitness')
    plt.grid(b=True, which='both', axis='both')

    "Stress plot"
    list_stress_positive = np.array(list_stress_positive)
    x_stress_positive = list_iter
    y_stress_positive = list_stress_positive
    stress_positive = (x_stress_positive, y_stress_positive)

    list_stress_negative = np.array(list_stress_negative)
    x_stress_negative = list_iter
    y_stress_negative = list_stress_negative

    stress_negative = (x_stress_negative, y_stress_negative)

    ax2 = fig.add_subplot(2, 2, 2)
    ax2.set_title('Stress evolution')
    ax2.set_xlabel('Iterations')
    ax2.plot(x_stress_negative, y_stress_negative, c='coral')
    ax2.set_ylabel('Negative stress sum (MPa)')
    ax2.yaxis.label.set_color('coral')

    ax2_t = ax2.twinx()
    ax2_t.plot(x_stress_positive, y_stress_positive, c='darkslateblue')
    ax2_t.set_ylabel('Positive stress sum (MPa)')
    plt.grid(b=True, which='both', axis='both')
    ax2_t.yaxis.label.set_color('darkslateblue')

    "Weight plot"
    list_weight = np.array(list_weight).transpose()
    x_weight = list_iter
    y_weight = list_weight

    ax3 = fig.add_subplot(2, 2, 3)
    ax3.plot(x_weight, y_weight, c='firebrick')
    ax3.set_title('Weight evolution')
    ax3.set_xlabel('Iterations')
    ax3.set_ylabel('Construction weight (1000kg)')
    plt.grid(b=True, which='both', axis='both')

    "Deflection plot"
    list_defl = np.array(list_defl).transpose()
    x_defl = list_iter
    y_defl = np.round(sum(abs(list_defl[0])), 3)

    ax4 = fig.add_subplot(2, 2, 4)
    ax4.plot(x_defl, y_defl, c='lightcoral')
    ax4.set_title('Deflection evolution')
    ax4.set_xlabel('Iterations')
    ax4.set_ylabel('Abs deflection sum (m)')
    plt.grid(b=True, which='both', axis='both')

   # plt.legend(bbox_to_anchor=(0., 1.007, 1., .101), loc=3, ncol=1, mode="expand", borderaxespad=0.)

    plt.subplots_adjust(wspace=0.5, hspace=0.5)  # keep top
    plt.savefig(datetime.datetime.now().strftime('F_snp_w_d_I1_%Y%m%d_%H%M%S_') + ".pdf")


def plot_best_2(list_iter_2, list_fit_2, list_stress_positive_2, list_stress_negative_2, list_weight_2, list_defl_2):
    fig = plt.figure(figsize=(10, 8))
    fig.suptitle('Individual 2: the fittest individual development in time',
                 horizontalalignment='center', verticalalignment='center')

```

```python
    "Fitness plot"
    list_fit_2 = np.array(list_fit_2).transpose()
    x_fit = list_iter_2
    y_fit = list_fit_2

    ax1 = fig.add_subplot(2, 2, 1)
    ax1.plot(x_fit, y_fit, c='navy')
    ax1.set_title('Fitness evolution')
    ax1.set_xlabel('Iterations')
    ax1.set_ylabel('Fitness')
    plt.grid(b=True, which='both', axis='both')

    "Stress plot"
    list_stress_positive_2 = np.array(list_stress_positive_2)
    x_stress_positive_2 = list_iter_2
    y_stress_positive_2 = list_stress_positive_2
    stress_positive = (x_stress_positive_2, y_stress_positive_2)

    x_stress_negative_2 = list_iter_2
    y_stress_negative_2 = np.array(list_stress_negative_2)
    stress_negative = (x_stress_negative_2, y_stress_negative_2)

    ax2 = fig.add_subplot(2, 2, 2)
    ax2.set_title('Stress evolution')
    ax2.set_xlabel('Iterations')
    ax2.plot(x_stress_negative_2, y_stress_negative_2, c='coral')
    ax2.set_ylabel('Negative stress sum (MPa)')
    ax2.yaxis.label.set_color('coral')

    ax2_t = ax2.twinx()
    ax2_t.plot(x_stress_positive_2, y_stress_positive_2, c='darkslateblue')
    ax2_t.set_ylabel('Positive stress sum (MPa)')
    plt.grid(b=True, which='both', axis='both')
    ax2_t.yaxis.label.set_color('darkslateblue')

    "Weight plot"
    list_weight = np.array(list_weight_2).transpose()
    x_weight = list_iter_2
    y_weight = list_weight

    ax3 = fig.add_subplot(2, 2, 3)
    ax3.plot(x_weight, y_weight, c='cornflowerblue')
    ax3.set_title('Weight evolution')
    ax3.set_xlabel('Iterations')
    ax3.set_ylabel('Construction weight (1000kg)')
    plt.grid(b=True, which='both', axis='both')

    "Deflection plot"
    list_defl = np.array(list_defl_2).transpose()
    x_defl = list_iter_2
    y_defl = np.round(sum(abs(list_defl[0])), 3)

    ax4 = fig.add_subplot(2, 2, 4)
    ax4.plot(x_defl, y_defl, c='mediumblue')
    ax4.set_title('Deflection evolution')
    ax4.set_xlabel('Iterations')
    ax4.set_ylabel('Abs deflection sum (m)')
    plt.grid(b=True, which='both', axis='both')

   # plt.legend(bbox_to_anchor=(0., 1.007, 1., .101), loc=3, ncol=1, mode="expand", borderaxespad=0.)

    plt.subplots_adjust(wspace=0.5, hspace=0.5)  # keep top
    plt.savefig(datetime.datetime.now().strftime('F_snp_w_d_I2_%Y%m%d_%H%M%S_') + ".pdf")


def plot_best_3(list_iter_3, list_fit_3, list_stress_positive_3, list_stress_negative_3, list_weight_3, list_defl_3):
    fig = plt.figure(figsize=(10, 8))
    fig.suptitle('Individual 3: the fittest individual development in time',
                 horizontalalignment='center', verticalalignment='center')

    "Fitness plot"
    list_fit_3 = np.array(list_fit_3).transpose()
    x_fit = list_iter_3
    y_fit = list_fit_3

    ax1 = fig.add_subplot(2, 2, 1)
    ax1.plot(x_fit, y_fit, c='darkgreen')
    ax1.set_title('Fitness evolution')
    ax1.set_xlabel('Iterations')
    ax1.set_ylabel('Fitness')
    plt.grid(b=True, which='both', axis='both')

    "Stress plot"
    list_stress_positive = np.array(list_stress_positive_3)
    x_stress_positive = list_iter_3
    y_stress_positive = list_stress_positive
    stress_positive = (x_stress_positive, y_stress_positive)

    list_stress_negative = np.array(list_stress_negative_3)
    x_stress_negative = list_iter_3
```

```python
    y_stress_negative = list_stress_negative
    stress_negative = (x_stress_negative, y_stress_negative)

    ax2 = fig.add_subplot(2, 2, 2)
    ax2.set_title('Stress evolution')
    ax2.set_xlabel('Iterations')
    ax2.plot(x_stress_negative, y_stress_negative, c='coral')
    ax2.set_ylabel('Negative stress sum (MPa)')
    ax2.yaxis.label.set_color('coral')

    ax2_t = ax2.twinx()
    ax2_t.plot(x_stress_positive, y_stress_positive, c='darkslateblue')
    ax2_t.set_ylabel('Positive stress sum (MPa)')
    plt.grid(b=True, which='both', axis='both')
    ax2_t.yaxis.label.set_color('darkslateblue')

    "Weight plot"
    list_weight = np.array(list_weight_3).transpose()
    x_weight = list_iter_3
    y_weight = list_weight

    ax3 = fig.add_subplot(2, 2, 3)
    ax3.plot(x_weight, y_weight, c='olivedrab')
    ax3.set_title('Weight evolution')
    ax3.set_xlabel('Iterations')
    ax3.set_ylabel('Construction weight (1000kg)')
    plt.grid(b=True, which='both', axis='both')

    "Deflection plot"
    list_defl = np.array(list_defl_3).transpose()
    x_defl = list_iter_3
    y_defl = np.round(sum(abs(list_defl[0])), 3)

    ax4 = fig.add_subplot(2, 2, 4)
    ax4.plot(x_defl, y_defl, c='mediumseagreen')
    ax4.set_title('Deflection evolution')
    ax4.set_xlabel('Iterations')
    ax4.set_ylabel('Abs deflection sum (m)')
    plt.grid(b=True, which='both', axis='both')

   # plt.legend(bbox_to_anchor=(0., 1.007, 1., .101), loc=3, ncol=1, mode="expand", borderaxespad=0.)

    plt.subplots_adjust(wspace=0.5, hspace=0.5)  # keep top
    plt.savefig(datetime.datetime.now().strftime('F_snp_w_d_I3_%Y%m%d_%H%M%S_') + ".pdf")

def plot_fits_3(list_iter, list_iter_2, list_iter_3,
                list_fit, list_fit_2, list_fit_3,
                population_1, population_2, population_3,
                mut_x_1, mut_x_2, mut_x_3, mut_yA_1, mut_yA_2, mut_yA_3):

    list_fit = np.array(list_fit).transpose()
    list_fit_2 = np.array(list_fit_2).transpose()
    list_fit_3 = np.array(list_fit_3).transpose()

    fig = plt.figure(figsize=(10, 8))

    fig.suptitle('Fitness evolution', horizontalalignment='center', verticalalignment='center')
    fig.subplots_adjust(wspace=0.5, hspace=0.6)

    "Fitness plot"
    x_fit_1 = list_iter
    y_fit_1 = list_fit
    x_fit_2 = list_iter_2
    y_fit_2 = list_fit_2
    x_fit_3 = list_iter_3
    y_fit_3 = list_fit_3

    ax1 = fig.add_subplot(1, 1, 1)

    ax1.plot(x_fit_1, y_fit_1, 'r', label=('Population size: %s' % population_1,
                                           'Mutation x: 2 ind every %s iter' % mut_x_1, 'Mutation y, A: every %s' % mut_yA_1))
    ax1.plot(x_fit_2, y_fit_2, 'navy',label=('Population size: %s' % population_2,
                                             'Mutation x: 1 ind every %s iter' % mut_x_2, 'Mutation y, A: every %s' % mut_yA_2))
    ax1.plot(x_fit_3, y_fit_3, 'darkgreen', label=('Population size: %s' % population_3,
                                              'Mutation x: 1 ind every %s iter' % mut_x_3, 'Mutation y, A: every %s' % mut_yA_3))

    #ax1.plot(x_fit_1, y_fit_1, 'r', label='Mutation %s' % )
    #ax1.plot(x_fit_2, y_fit_2, 'navy',label='Pop %s' % population_2)
    #ax1.plot(x_fit_3, y_fit_3, 'gold', label='Pop %s' % population_3)
    ax1.set_xlabel('Iterations')
    ax1.set_ylabel('Fitness')
    plt.grid(b=True, which='both', axis='both')
    plt.legend(bbox_to_anchor=(0., 1.007, 1., .101), loc=3,
               ncol=1, mode="expand", borderaxespad=0.)

    plt.savefig(datetime.datetime.now().strftime('Fit3_%Y%m%d_%H%M%S_') + ".pdf")

```