1. Objective

The objective of this assignment is to write the code for the simulated annealing algorithm that allows for a one-pass penalty and quadratic penalty. Then, using this simulated annealing code find an appropriate c and epsilon value given the bounds using the bump function. Then, use this value for running the SA algorithm on a 10-bar truss for optimizing the weight of the trusses.

2. Code Structure

Please see the docstrings on the Python files.

3. Bump Function Results

The primary procedure for getting a good value for the cooling schedule parameter, c, is by trial and error after fixing a value for the rest of the parameters. As suggested by the assignment, the following parameter values are used that were fixed, so that only one parameter will be changed at a time:
`t_start` (starting temperature) `= 1000`
`max_iter` (maximum number of iterations) `= 5000`
`epsilon = 2`
The epsilon value was chosen as per the suggestion on the lecture slide which says it is typically between 0.1-0.3 – and I simply chose the middle value which is 0.2. However, since I did not normalize move.py, epsilon had to be multiplied by the range of the bounds i.e. upper bound – lower bound. The upper bound is 10, therefore, 0.2*10 = 2.

The initial design variable value, x0 is [1, 1] (since n=2, per the assignment). Also, per the assignment, lower bound is [0, 0] and upper bound is [10, 10] and are all inputted into `move.py` as floats. The trial and error showed a trend that is easy to see that values closer to 1 are more optimized, lower value than those farther so the more detailed analyses where I graphed these values and looke at convergence were only performed at high c values shown in Figure 1.

The best solution for x and f, outputted as xopt and fopt in my `Assignment3.py` file over multiple times of running the same file and averaging the values are the following:
`xopt [ 1.59415299  0.47298035]`
`fopt -0.362774184804`

The best value for the parameter c is 0.996. This is shown as a low point when fopt is plotted against c. Beyond this for c=0.998 for example (as shown in figure 1), it has difficulty converging in 5000 iterations and is also not the lowest as per figure 2.

A more rigourous approach is to create take a lot of values close to 1 since that's the trend I noticed by trial and error and run each of these c values ran 5 times and these runs are averaged. The graphs for the convergence are shown below, as can be seen at aroung c=0.998 it is not converging with 5000 iterations and 0.996 has the lowest average of `fopt` values. I also added a random seed to reduce the randomness, though this did not have a noticeable effect.

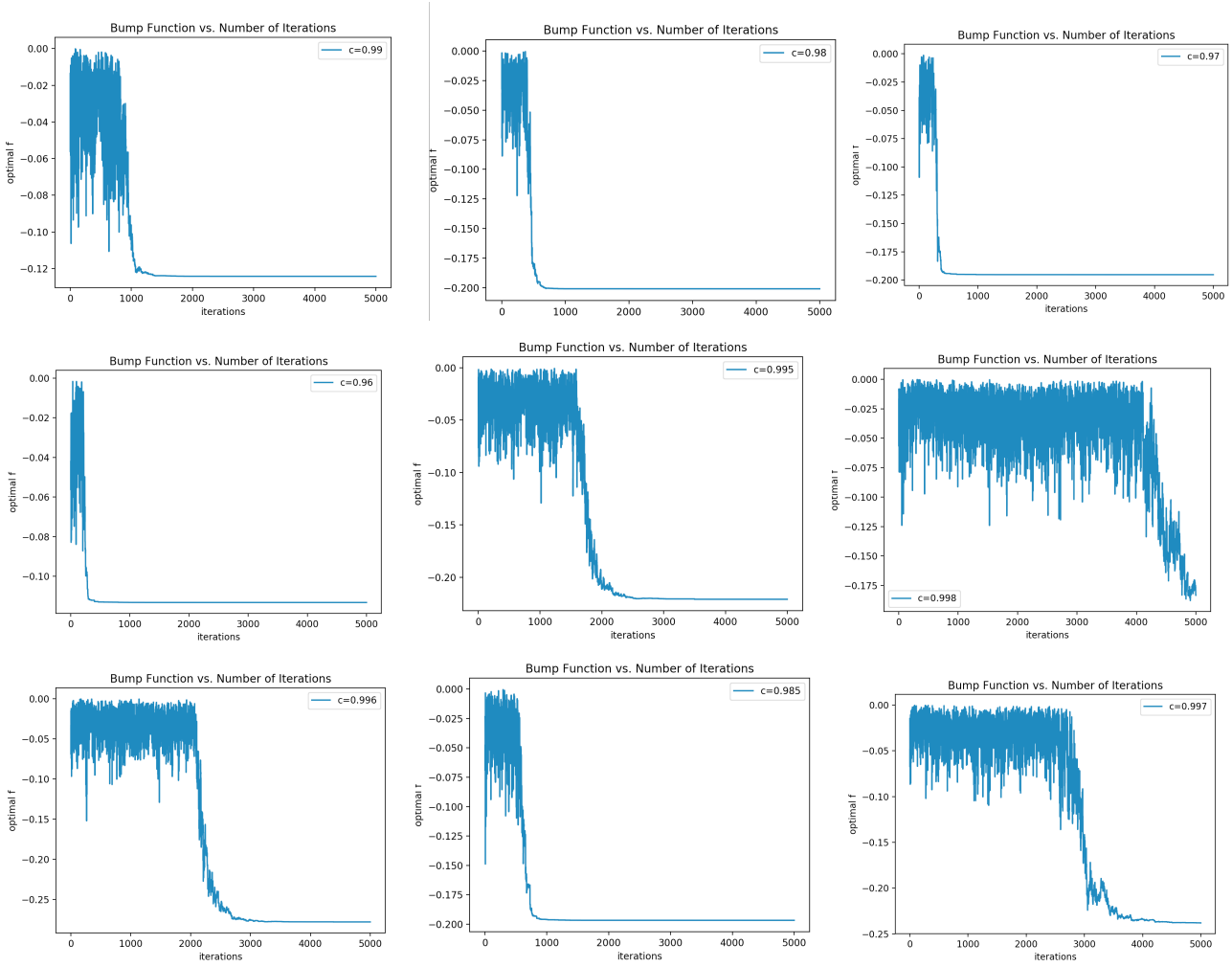

Figure 1: Plots of multiple c parameter values to show convergence and for finding the best c.

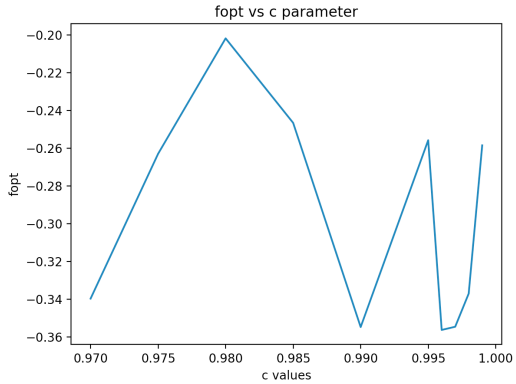

Figure 2: optimized f plotted against multiple c parameter values.

4. 10-bar truss optimality

`fopt` is optimized total weight of the truss, calculated assuming the density and yield strength of a 6061 aluminium alloy. This has a density of 2700kg/m^3 and a yield strength of 270 * 10^6 N/m^2.

`xopt` is the vector of cross-sectional area of each bar with the indices corresponding to the bar elements shown in Assignment 1.

One Pass Penalty

Using one of the runs for **a one-pass penalty** the following output is generated:

```
xopt [  6.59132677e-07   1.00451299e-07   2.66050288e-07   4.85029805e-07
   1.32222080e-07   1.13622423e-07   1.05595013e-06   3.90481588e-08
   4.30654870e-07   8.92079244e-08]
fopt 0.0109921230571
```

Figure 3 shows the average of 5 independent runs of 5000 iterations. f as shown on the y axis is the optimal weight calculated and it is plotted against the number of iterations (time) for the 10-bar truss problem. The number of independent runs is limited to 5 due to time constraints as it took around 30 minutes to run due to the inefficiencies of Assignment 1.

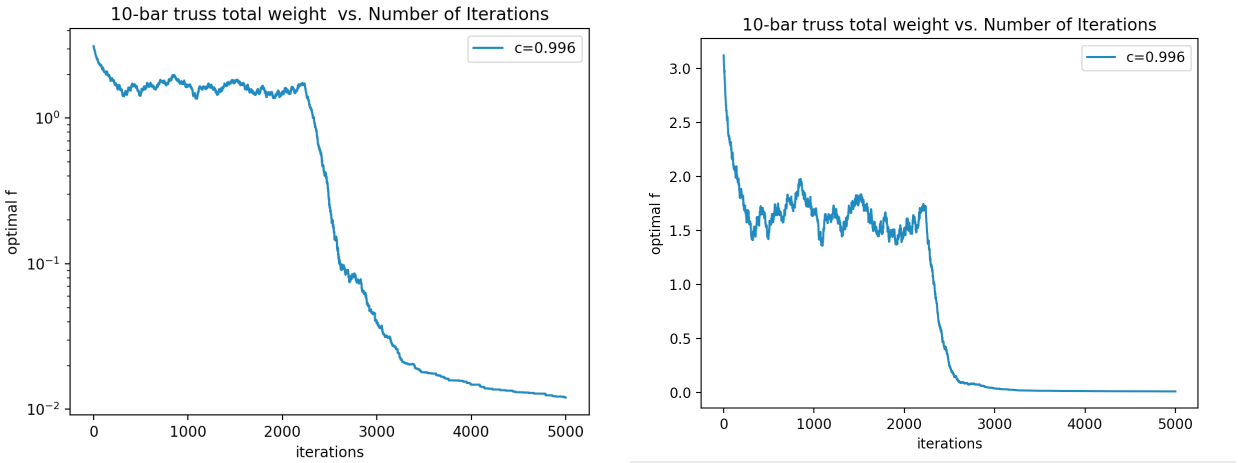

Figure 3: Average of 5 independent runs of the optimal weight for the 10 bar truss. Left side is scaled with semilogy for clarity using a one-pass penalty. The optimal f is the weight of the truss in total in kg.

Convergence Trend of one-pass penalty

Again, the convergence trend is generated by averaging 5 independent runs because of the randomness. The higher number of independent runs is better but due to bad assignment 1 code I am restricted to a relatively small value because 5 independent runs already take around 30 min.

Quadratic Penalty

Using one of the runs with **P=1, 10, 100** using a quadratic penalty the following output is generated. This takes quite a while to run so I only showed one example output per P value, and also serves as to minimize the clutter on the document:

P=1

```
xopt [  1.67962874e-06   2.29641819e-07   1.41070501e-07   3.19882045e-07
   3.31579780e-07   1.16674870e-06   7.25534924e-08   4.93178099e-07
   5.29934413e-07   1.90177136e-07]
fopt 0.0161082282816
```

P=10

```
xopt [  7.65153013e-07   1.24423101e-07   2.66686809e-07   3.80886755e-07
   4.94411950e-07   3.28012683e-07   9.21434498e-07   2.94054194e-09
   4.33066706e-07   1.87165876e-07]
fopt 0.0126322973516
```

P = 100

```
xopt [  1.24762904e-06   1.58891204e-07   4.45165774e-07   1.59348258e-07
   4.43174028e-08   3.28803263e-07   1.24335250e-07   6.87394371e-07
   3.77365586e-07   1.63029885e-07]
fopt 0.0111991012678
```

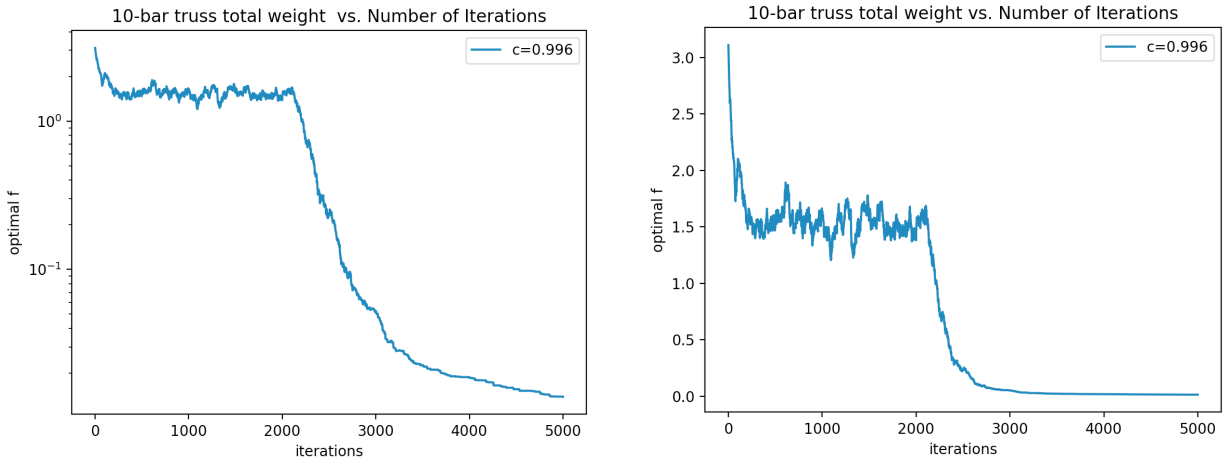

Figure 4: Average of 5 independent runs for the 10-bar truss using Quadratic Penalty P=1. The optimal f is the weight of the truss in total in kg.

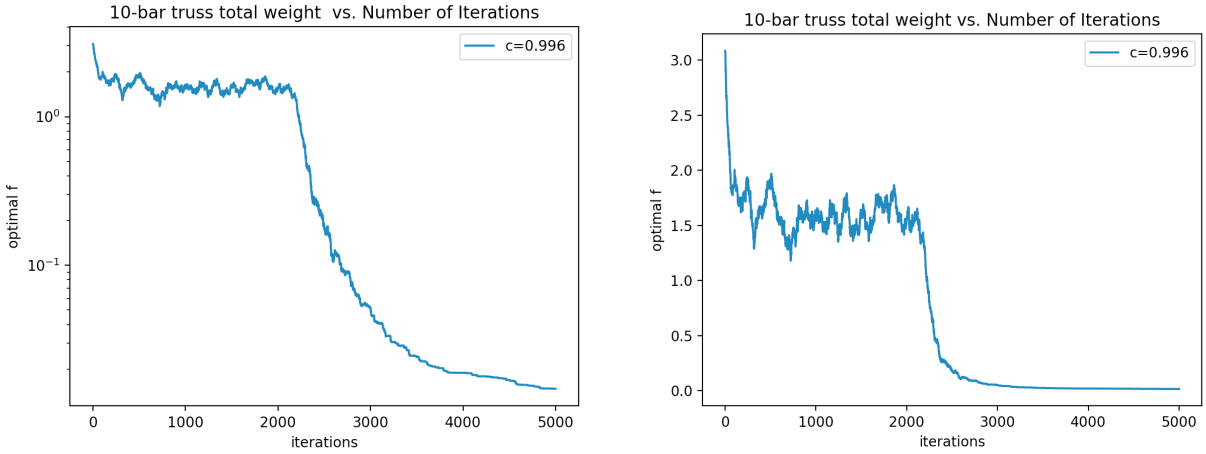

Figure 5: Average of 5 independent runs for the 10-bar truss using Quadratic Penalty P=10. The optimal f is the weight of the truss in total in kg. Left is the same as the right, apart from it is using semilogy

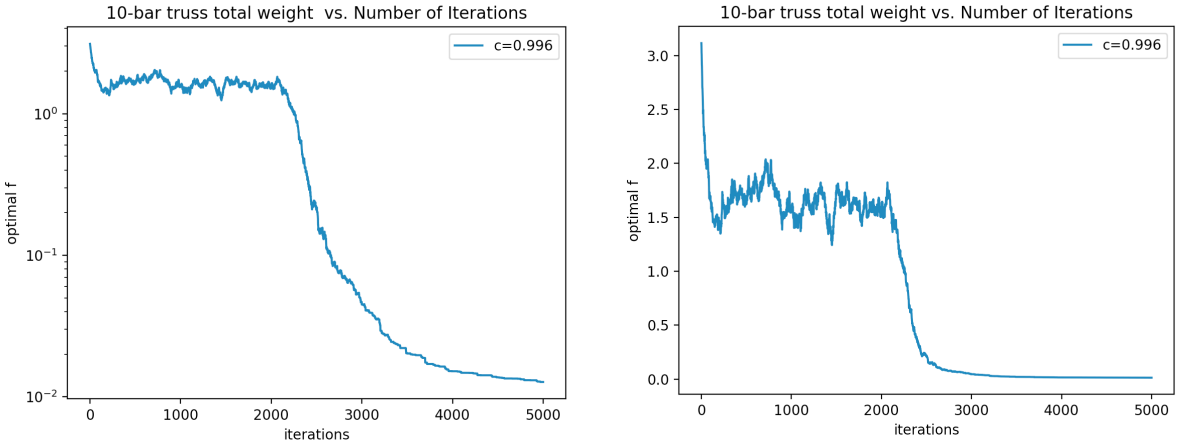

Figure 6: Average of 5 independent runs for the 10-bar truss using Quadratic Penalty P=100. The optimal f is the weight of the truss in total in kg.

For more output, run the Python code and the values will be printed out on console. The code takes quite some time to run though.

Compare 3 penalty parameters and convergence using quadratic penalty

They don't make much of a difference apart from a higher P giving lower weights thus more optimal. It also looks to be ever so slightly begin to converge earlier.

One-Pass Penalty vs Quadratic Penalty

The results from the one-pass and quadratic penalty are similar both around the order of 10g. From the graphs there is not much difference between the one-pass penalty and the quadratic penalty apart from that the one-pass penalty look like it converges more steeply than the quadratic penalty. Again, this difference is minute and could also be a result of randomness and are not significant.
Note: Values outputted for one-pass and quadratic are verified against the constraint after the fact by checking the bounds by hand.

Feasibility of Results

No the designs are not feasible, because there is no buckling it is an incredibly thin and light weight truss structure. It is not feasible because we must account for buckling in truss designs. The end result is essentially a wire of aluminium.

5. Bonus

Bump

The bonus was implemented for the bump function and the following was generated by scipy.optimize.minimize()

```
fun: -0.29297987317495194
     jac: array([ 0.59436553,  0.15002728])
 message: 'Iteration limit exceeded'
    nfev: 1228
     nit: 101
    njev: 100
  status: 9
 success: False
       x: array([ 0.45105023,  1.54909176])
```

The values are close enough to the -0.36 gotten by iterating before and is a lot faster but I had to change with the initial start value because it was getting stuck in local minima. I think this is because scipy optimizes with gradient descent.

10-bar truss

It was incredibly difficult to find just the right initial condition for minimizing the weight of the truss. The initial value of 0.0001 for all members did not work and there are too many combinations for the 10 bars to initialize a guess for the scipy.optimize.minimize() to get a reasonable minimization. As such, I just tested

my code using a previous value gotten for P=1 and it confirms that the values are almost the same as the iterated version using SA.

```
     fun: 0.01610822827274784
     jac: array([ 2700.        ,  2700.        ,  2700.        ,
2700.        ,
        2700.        ,  3818.37661841,  3818.37661841,  2700.        ,
        3818.37661841,  3818.37661841])
 message: 'Positive directional derivative for linesearch'
    nfev: 12
     nit: 5
    njev: 1
  status: 8
 success: False
       x: array([  1.67962874e-06,   2.29641819e-07,   1.41070501e-07,
         3.19882045e-07,   3.31579780e-07,   1.16674870e-06,
         7.25534924e-08,   4.93178099e-07,   5.29934413e-07,
         1.90177136e-07])
```