
Article:
He, L., Gilbert, M. orcid.org/0000-0003-4633-2839 and Song, X. (2019) A Python script for adaptive layout optimization of trusses. Structural and Multidisciplinary Optimization, 60 (2). pp. 835-847. ISSN 1615-147X

https://doi.org/10.1007/s00158-019-02226-6

Noname manuscript No.
(will be inserted by the editor)

A Python script for adaptive layout optimization of trusses

Linwei He · Matthew Gilbert · Xingyi Song

Received: date / Accepted: date

Abstract Numerical layout optimization employing an adaptive 'member adding' solution scheme provides a computationally efficient means of generating (near-)optimum trusses for problems involving single or multiple load cases. To encourage usage of the method a Python script is presented, allowing medium to large-scale problems to be solved efficiently. As well as handling multiple load cases, the short (98 line) script presented can tackle truss optimization problems involving unequal limiting tensile and compressive stresses, joint-costs, and non-convex polygonal domains, with or without holes. Various numerical examples are used to demonstrate the efficacy of the script presented.

Keywords Truss · Layout optimization · Ground structure method · Python · Education

1 Introduction

Truss layout optimization using the 'ground structure' approach provides a fully automated means of identifying (near-)optimum truss structures. Although first proposed in the 1960s by Dorn et al (1964), it has not been widely used in practice for various reasons. For example, the solutions generated appear impractical, at least when conventional manufacturing techniques are envisaged. However, in recent decades new manufacturing techniques have been developing apace, e.g., additive manufacturing, and means of rationalizing the solutions obtained via layout optimization have also been developed (e.g., He and Gilbert 2015). Also, in 2003 an adaptive 'member adding' scheme was proposed by Gilbert and Tyas (2003), which allowed solutions to be obtained more quickly, and also for extremely large problems (e.g., $>$1,000,000,000 members) to be solved. This means that layout optimization can be used to obtain highly accurate benchmark solutions that can be used to validate analytical solutions (e.g., Sokół 2011b; Sokół and Rozvany 2012), and also to discover new optimum structural forms (e.g., Darwich et al 2010; Tyas et al 2011; Fairclough et al 2018). To increase the practical applicability of solutions, Pritchard et al (2005) extended the adaptive 'member adding' scheme to solve problems involving multiple load-cases whilst Tyas et al (2006) incorporated structural stability considerations. Also, Smith et al (2016) utilized layout optimization to design truss-like metallic components suitable for fabrication via additive manufacturing, with the techniques involved recently extended by He et al (2018). In parallel, the layout optimization technique has been successfully transferred to other applications. For example, Bolbotowski et al (2018) utilized layout optimization to design (near-)optimum grillages and Smith and Gilbert (2007); Gilbert et al (2014) proposed the 'discontinuity layout optimization' (DLO) method to identify critical patterns of discontinuities at collapse for in-plane and out-of-plane plasticity problems.

However, although interest in numerical layout optimization has grown in recent years, the power of the method still appears under-appreciated in the structural optimization research community. One possible reason for this is that accessible educational resources for use by new researchers have been lacking. Although a script was made available by Sokół (2011a), the Mathematica language employed is

L. He
Department of Civil and Structural Engineering, University of Sheffield, Sir Frederick Mappin Building, Mappin Street, Sheffield, S1 3JD, UK. E-mail: linwei.he@sheffield.ac.uk

M. Gilbert
Department of Civil and Structural Engineering, University of Sheffield, Sir Frederick Mappin Building, Mappin Street, Sheffield, S1 3JD, UK. E-mail: m.gilbert@sheffield.ac.uk

X. Song
Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello, Sheffield, S1 4DP, UK. E-mail: x.song@sheffield.ac.uk

not commonly used in the community. Matlab scripts developed by Zegard and Paulino (2014, 2015) helped in this respect, although these contain relatively complex functions that could potentially limit uptake. Also, most significantly, although proposed in 2003, the 'member adding' scheme has not yet been incorporated in any of the aforementioned publicly available scripts, potentially leading researchers to underestimate the potential computational efficiency of the layout optimization method. In contrast, the 99-line Matlab script for SIMP (Sigmund 2001) helped pave the way for worldwide research in the field of continuum topology optimization. As an indication of the dominance of SIMP over numerical layout optimization in terms of published educational source codes, only one of the 22 scripts listed by Wei et al (2018) employ the latter.

In order to provide an accessible yet computationally efficient educational implementation of numerical layout optimization, this paper introduces a simple Python script. With 98 lines, it incorporates the efficient adaptive 'member adding' scheme for 2D problems subject to multiple load cases, with the potential for unequal limiting tensile and compressive stresses, joint-costs, and non-convex polygonal domains (with or without holes). The paper is organized as follows: first, the mathematical layout optimization formulation is introduced; second, important code sections are explained; third, numerical examples are shown to demonstrate the efficacy of the script; finally, conclusions are drawn.

2 Formulation of layout optimization

2.1 Single load case problem

The standard layout optimization process (Dorn et al 1964; Gilbert and Tyas 2003; Pritchard et al 2005) involves a number of steps, as shown in Fig. 1. Firstly the design domain, load and support conditions are specified, Fig. 1(a); secondly, the design domain is discretized using nodes, Fig. 1(b); thirdly, these nodes are interconnected with potential members to create a 'ground structure', Fig. 1(c); finally, an optimum layout is identified by solving the optimization problem below (Fig. 1(d)):

$$\min_{\mathbf{a},\mathbf{q}} V = \mathbf{l}^{\mathrm{T}}\mathbf{a} \tag{1a}$$

s.t.

$$\mathbf{B}\mathbf{q} = \mathbf{f} \tag{1b}$$

$$\mathbf{q} \geq -\sigma^{-}\mathbf{a} \tag{1c}$$

$$\mathbf{q} \leq \sigma^{+}\mathbf{a} \tag{1d}$$

$$\mathbf{a} \geq \mathbf{0}, \tag{1e}$$

where, V is the structural volume, $\mathbf{a} = [a_1, a_2, ..., a_m]^{\mathrm{T}}$ is a vector containing member cross-sectional areas, with m denoting the number of members. $\mathbf{l} = [l_1, l_2, ..., l_m]^{\mathrm{T}}$ is a vector of member lengths. $\mathbf{B}$ is a $2n \times m$ equilibrium matrix comprising direction cosines, with n denoting the number of nodes and $\mathbf{q}$ is vector containing the internal member forces; a simple example of the equilibrium constraint is presented in Fig. 2. $\mathbf{f}$ is a vector containing the external forces; σ^{+} and σ^{-} are limiting tensile and compressive stresses respectively.

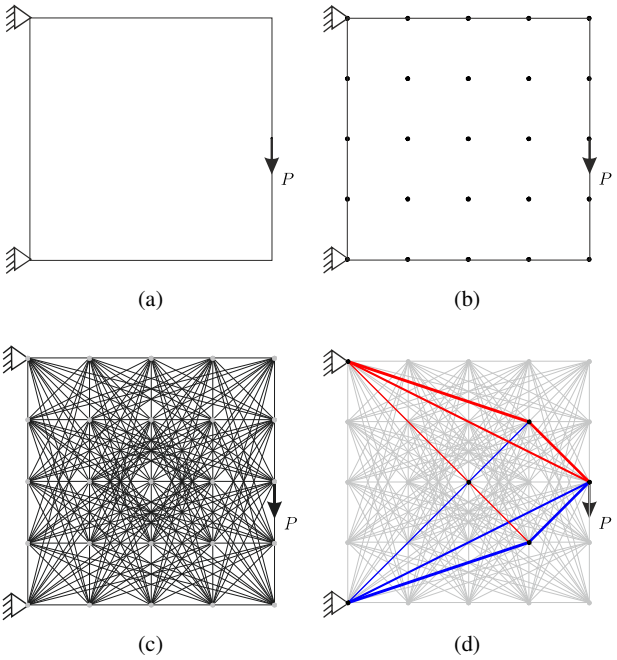


Fig. 1: Steps in layout optimization: (a) specify design domain, loads and supports; (b) discretize domain using nodes; (c) interconnect nodes with potential truss members; (d) use optimization to identify the optimal structural layout

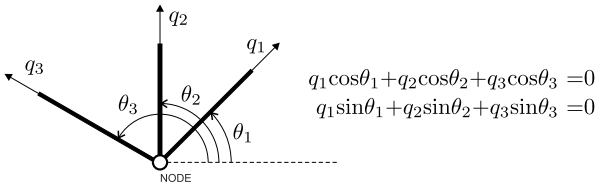


Fig. 2: Equilibrium condition at a typical (unloaded) node

2.2 Multiple load case problem

Problem (1) can easily be extended to obtain solutions for multiple load case plastic design problems. In this case the equilibrium constraint (1b), stress limits (1c) and (1d) must be satisfied in each load case, e.g., the former gives:

$$\mathbf{B}\mathbf{q}^k = \mathbf{f}^k, \text{ for } k = 1, 2, ..., p, \tag{2}$$

where, k is the load case identifier and p represents the total number of load cases.

2.3 Joint costs

It was found in previous studies (e.g., Gilbert and Tyas 2003; Pritchard et al 2005) that layout optimization will often generate structures that are complex in form, containing large numbers of short members. A simple means of addressing this is to include a notional joint cost (Parkes 1978), which has the effect of penalizing short members and hence simplifying the solution in many cases: $\tilde{\mathbf{l}} = [l_1 + s, l_2 + s, ..., l_m + s]^{\mathrm{T}}$ in the objective function (1a), where $\tilde{\mathbf{l}}$ is the augmented member length vector with predefined joint cost (or length) s.

2.4 Full problem

Considering multiple load cases and joint costs, the full problem can now be written as:

$$\min_{\mathbf{a},\mathbf{q}} V = \tilde{\mathbf{l}}^{\mathrm{T}}\mathbf{a} \tag{3a}$$

s.t.

$$\left.\begin{aligned} \mathbf{B}\mathbf{q}^k &= \mathbf{f}^k \\ \mathbf{q}^k &\geq -\sigma^{-}\mathbf{a} \\ \mathbf{q}^k &\leq \sigma^{+}\mathbf{a} \end{aligned}\right\} \text{for } k = 1, 2, ..., p \tag{3b}$$

$$\mathbf{a} \geq \mathbf{0}, \tag{3c}$$

which is, like (1), a linear programming (LP) problem that can be solved efficiently using modern optimization solvers. However, when using the ground structure approach the number of potential numbers grows rapidly with the number of nodes employed, with up to $\frac{n(n-1)}{2}$ members required. This can easily lead to extremely large optimization problems that are computationally expensive to solve. To address this, a 'member adding' scheme, which is based on the 'column generation' approach (Dantzig and Wolfe 1960) was proposed by Gilbert and Tyas (2003). This allows problem (3) to be decomposed into a number of smaller subproblems, with information from the dual problem used to guide the process.

2.5 Adaptive 'member adding' scheme

Using the adaptive 'member adding' scheme (Gilbert and Tyas 2003; Pritchard et al 2005), initially only a reduced set of members are used to solve problem (3), and remaining potential members in the 'ground structure' are gradually added to the set, until the following constraint is satisfied for all potential members (readers who are interested in derivation of (4) can refer to Appendix A):

$$\epsilon_i = \sum_{k=1}^{p} \frac{\max\left\{\sigma^{+}\mathbf{B}_i^{\mathrm{T}}\mathbf{u}_i^k,\ -\sigma^{-}\mathbf{B}_i^{\mathrm{T}}\mathbf{u}_i^k\right\}}{\tilde{l}_i} \leq 1, \quad (\text{for } i = 1, ..., m), \tag{4}$$

where ϵ_i is the summed maximum virtual strain of member i in all load cases; $\mathbf{u}_i^k$ is a vector containing virtual displacement of nodes connected by member i, under load case k. Note that, when a primal-dual interior point method is used to solve problem (3), $\mathbf{u}_i^k$ is obtained automatically with no extra cost. Although only a reduced set of members are used in (3), a full virtual displacement field $\mathbf{u}^k$ of all nodes can still be generated. If any violation of (4) is found in the virtual displacement field, the most violated potential members can be added to the reduced set of members. Then (3) can be solved again, this time generating an improved virtual displacement field in which (4) is no longer violated for the newly added members. This adaptive 'member adding' process continues until no violation is detected; according to duality theory, the solution obtained will then be equivalent to that obtained by solving the full problem from the outset. The entire process is shown in Fig. 3.

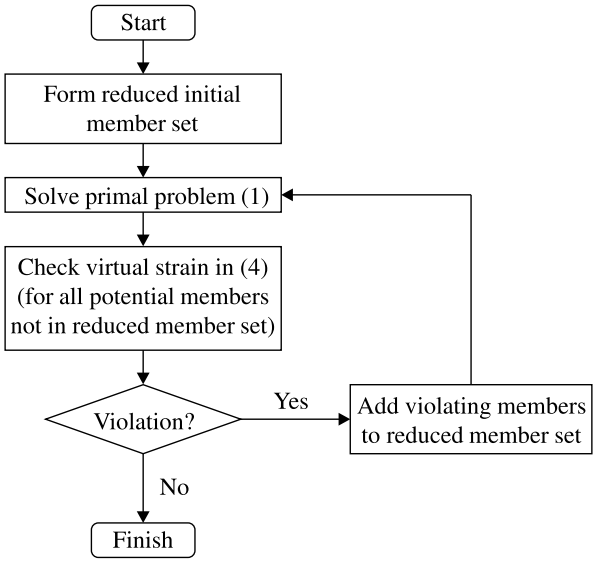


Fig. 3: Flowchart of the adaptive 'member adding' process

3 Python implementation

Python is an open source interpreted high-level programming language that is becoming increasingly popular for solving scientific and engineering problems. Unlike Matlab, where built-in toolboxes are automatically loaded when

started, in Python such tools are imported using the `import` keyword at the beginning of the script. These extra `import` lines allow dependencies on external tools to be clearly seen. In this paper, a number of freely available tools are used, including mathematical tools such as `scipy` and `numpy`, the convex optimization tool `cvxpy` (Diamond and Boyd 2016; Agrawal et al 2018), and the geometry tool `shapely`.

In this section the various functions used to construct and solve the layout optimization problem (3) are first introduced, then the main workflow is outlined. As Python is, like Matlab, an interpreted language, loop structures are generally inefficient. Therefore, to improve execution speed, vectorized operations are utilized in place of loop structures wherever possible.

3.1 Equilibrium matrix $\mathbf{B}$

For a member i, interconnecting nodes I and II, its contribution to equilibrium matrix $\mathbf{B}$ can be written as:

$$\mathbf{B}_i = \begin{bmatrix} -X_i/l_i & -Y_i/l_i & X_i/l_i & Y_i/l_i \end{bmatrix}^{\mathrm{T}}, \tag{5}$$

where, $X_i = x_i^{II} - x_i^I$ and $Y_i = y_i^{II} - y_i^I$ are the projected length of the member of length l_i in the x and y axis directions respectively, and where $[x_i^I, y_i^I]$ and $[x_i^{II}, y_i^{II}]$ are the co-ordinates of nodes I and II are respectively. If a connected node is supported, then the coefficients of the corresponding row are set to zero (i.e. supported degrees of freedom are removed). Thus taking into account boundary conditions, the above formulation can be expressed as:

$$\mathbf{B}_i = \begin{bmatrix} -d_0 X_i/l_i & -d_1 Y_i/l_i & d_2 X_i/l_i & d_3 Y_i/l_i \end{bmatrix}^{\mathrm{T}}, \tag{6}$$

where d_0, d_1, d_2, d_3 are 0-1 coefficients describing the degree of freedom in the x and y directions for the connected nodes

The equilibrium matrix $\mathbf{B}$ is then assembled using the $\mathbf{B}_i$ matrices for all members. $\mathbf{B}$ is a sparse matrix, which can be expressed using a row-column-value format comprising only non-zero entries, constructed by concatenating the coefficients calculated in (6) as well as their corresponding row (degree of freedom at nodes) and column (member) numbers.

3.2 Solving the LP problem

In the script, problem (3) is solved in function `solveLP`.

In Python, the use of object oriented programming (OOP) allows problem (3) to be written in a format that is very similar to its natural expression, improving readability. For example, using the optimization tool `cvxpy`, the stress constraints in (3b) are expressed as `q[k] >= -sc * a` and `q[k] <= st * a`.

3.3 Check dual violation

When the primal problem (3) is solved via the primal-dual interior point method, the virtual displacements in $\mathbf{u}$ for each load case are obtained from the equilibrium constraints defined in `eqn`. Therefore, violation of constraint (4) can be calculated for every member in the full 'ground structure' (except for those already present in the reduced set, where this is unnecessary as there will be no violation). All potential members are then sorted by their constraint violation numbers, and the most violating ones are added to the primal problem, with the number of new members calculated using:

$$\Delta m = \begin{cases} \alpha m_{\mathrm{V}}, & m_{\mathrm{V}} \geq \beta m_{\mathrm{P}}, \\ \alpha\beta m_{\mathrm{P}}, & \beta m_{\mathrm{P}} > m_{\mathrm{V}} > \alpha\beta m_{\mathrm{P}}, \\ m_{\mathrm{V}}, & \alpha\beta m_{\mathrm{P}} \geq m_{\mathrm{V}}, \end{cases} \tag{7}$$

where, Δm is the number of new members to be added, m_{P} and m_{V} are, respectively, the numbers of remaining members in the potential member list (i.e., members not yet included in (3)), and those violating (4). α and β are control parameters, both set to 0.05 in this paper. Using the adaptive 'member adding' scheme, the size of problem (3) gradually increases, with (7) used to control the growth rate. When number of violations is large, only a small proportion are added into (3); on the other hand, when the number of violations is small, most (or all) are added in a given iteration. This balances problem size with the number of iterations required in order to achieve an efficient iterative process. Various alternative heuristics can be used to determine the priority and number of members to be added at each iteration, e.g., the two-stage high pass filter in Gilbert and Tyas (2003) and the active member set method described in Sokół (2014). It is important to note that, although a heuristic algorithm is used here to improve computational efficiency, the final solution, obtained when (4) is satisfied for all members in the ground structure, is equivalent to the solution obtained by solving the full problem, due to the strong duality nature of LP problems (Vanderbei 2001).

3.4 Visualization

The solutions are visualized in function `plotTrusses`. Members with areas greater than a threshold number (e.g., 1×10^{-3} of the maximum member area) are plotted in red or blue, depending on the sign of their internal forces. When the signs of internal forces vary in different load cases, gray is used instead.

3.5 Main workflow

The main `trussopt` function contains several steps to define the design domain, load and support conditions for any

given layout optimization problem, to create the 'ground structure' and to solve LP problem (3) using the adaptive 'member adding' scheme.

- Firstly, a polygonal design domain is specified using the `Polygon` function imported from `shapely`, and then its convexity checked.
- Secondly, a uniformly distributed grid of nodes is created using the `meshgrid` function, with points lying inside the design domain added to node list `Nd`; then support and load conditions are added.
- Thirdly, the 'ground structure' is created by generating all valid potential members, excluding overlapping lines (checked using the greatest common divider, function `gcd`, when joint costs are not specified), and lines crossing the polygonal boundary (using function `contains` from `shapely`).
- Finally, the initial reduced set of members is created by including only short member connections. The adaptive 'member adding' process then starts, in which potential members are added to this reduced set until no violation of (4) is found in any member in the 'ground structure'.

4 Numerical examples

A number of examples serve to demonstrate the efficacy of the script.

Firstly, a simple cantilever problem (Fig. 4) can be solved by running the following command in a terminal:

```
python trussopt.py
```

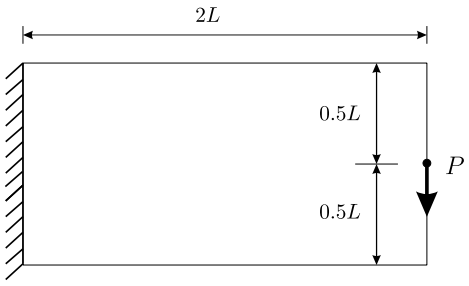


Fig. 4: Cantilever example: problem definition

Alternatively, the script can be imported as a module in Python using:

```
>>>from trussopt import trussopt
```

And then function `trussopt` can be called to perform a layout optimization with the following input arguments:

```
>>>trussopt(20, 10, 1, 1, 0)
```

which solves the simple cantilever problem shown in Fig. 4 with width = 20, height = 10, limiting stress in tension = 1, in compression = 1 and joint cost = 0. In this section, all line numbers refer to the script in Appendix C, and all quoted CPU times were obtained using a laptop PC equipped with an Intel I7-7700HQ CPU and running 64-bit Windows 10.

4.1 Effect of using the adaptive 'member adding' scheme

Using the adaptive 'member adding' scheme, a large-scale layout optimization problem is solved as a series of much smaller LP problems solved in succession; e.g., Fig. 5 shows steps in the solution of the cantilever design problem shown in Fig. 4.

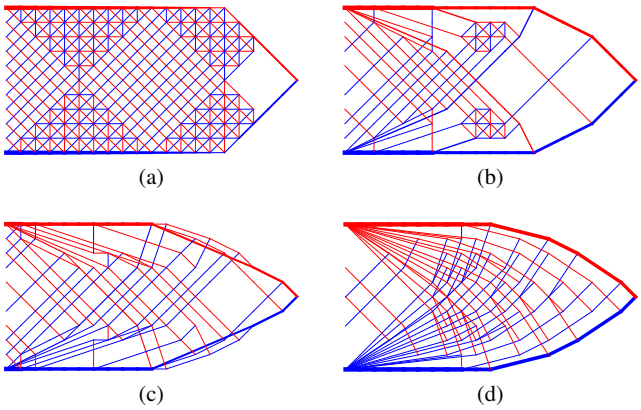


Fig. 5: Cantilever example: steps in the adaptive 'member adding' process following, (a) iteration 1, (b) iteration 2, (c) iteration 3, (d) final iteration

When solving medium or large-scale problems, the adaptive 'member adding' scheme reduces memory consumption and can result in significant speed improvements, as indicated in Table 1. Whilst the resulting structural volumes are identical (to five significant figures), the latter becomes increasingly efficient with increasing problem size, e.g., 7.5 times faster when 60×30 nodal divisions are used.

Note that 'member adding' can be switched off by increasing the maximum initial member connection distance in line 85, for example changing this to:

```
for pm in [p for p in PML if p[2] <= 1000]:
```

for the purposes of this paper.

4.2 Alternative LP solvers

By default, the LP problem (3) is solved using the open-source solver ECOS (Domahidi et al 2013) distributed with `cvxpy`. Alternatively, a range of other, potentially more efficient, LP solvers such as GUROBI and MOSEK can be used with `cvxpy`. To use MOSEK, for example, line 29 is

Table 1: Cantilever example: effect of using 'member adding' scheme (small problems)

Nodal divisions	Number of nodes	Number of members	Without 'member adding'		With 'member adding'				With 'member adding' speed up factor
			Volume ($\frac{PL}{\sigma_0}$)	CPU time (s) LP†	Volume ($\frac{PL}{\sigma_0}$)	CPU time (s) LP†	Dual‡	Total	
20×10	231	16,290	7.0747	1.0	7.0747	0.6	0.2	0.8	1.3
40×20	861	225,848	7.0454	47.7	7.0454	6.2	0.5	6.7	7.1
60×30	1,891	1,086,938	7.0376	579.0	7.0376	68.5	2.5	71.0	8.2

†: cumulative CPU time spent in function `solveLP`
‡: cumulative CPU time spent in function `stopViolation`

replaced with the following (note that this requires MOSEK has already been added to Python environment; if not, guidance can be found in Appendix B):

```
vol = prob.solve(solver = cvx.MOSEK, \
     mosek_params={"MSK_IPAR_INTPNT_BASIS":0})
```

The MOSEK parameter `"MSK_IPAR_INTPNT_BASIS"` is optional; however this disables the unnecessary basis identification step to improve speed. Using MOSEK, the CPU cost required to solve problems is significantly reduced, making the layout optimization process very efficient. Table 2 shows sample solutions for the cantilever problem shown in Fig. 4, indicating that a numerical solution within 0.04% of the analytical solution can be obtained for this problem on a laptop PC.

4.3 Joint cost

To approximately account for the costs associated with joints, a joint cost can be directly specified when calling the main function `trussopt`. For example, a joint cost of $s = 1$ unit length can be added using the following command; the simpler solution then generated is shown on Fig. 6.

```
>>>trussopt(20, 10, 1, 1, 1)
```

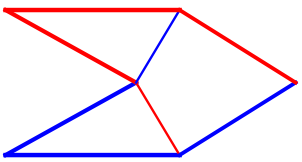


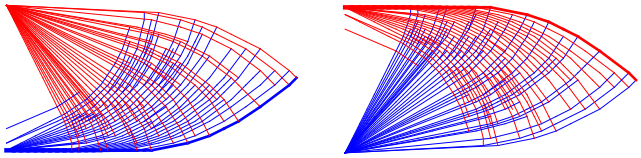
Fig. 6: Cantilever example: design obtained using a joint cost $s = 1$

4.4 Unequal stress limits

Unequal stress limits can also be directly specified. For example, a compressive stress limit can be set to $\sigma^- = 0.1\sigma_0$ with the following input arguments:

```
>>>trussopt(40, 20, 1, 0.1, 0)
```

Alternatively, a different tensile stress limit can be set if desired. The corresponding results are shown in Fig. 7.



(a) (b)

Fig. 7: Cantilever example: effect of specifying unequal stress limits, (a) $\sigma^+ = \sigma_0$, $\sigma^- = 0.1\sigma_0$; (b) $\sigma^+ = 0.1\sigma_0$, $\sigma^- = \sigma_0$

4.5 Change load and support conditions

Load and support conditions can be changed by modifying lines 73 and 74. For example, to generate the 'half wheel' example shown in Fig. 8, the pin and roller supports can be applied by replacing line 73 with:

```
if (nd == [0, 0]).all(): dof[i,:] = [0, 0]
if (nd == [width, 0]).all(): dof[i,:] = [1, 0]
```

which locate the two corner nodes at the base of the domain and then set the appropriate degrees of freedom. Similarly, the point load can be applied by replacing the original line 74 with:

```
f+=[0,-1] if(nd==[width/2,0]).all() else [0,0]
```

4.6 Multiple load cases

Additional load cases can be included by appending load vector `f` after line 74. For the two load case cantilever example problem shown in Fig. 9, load case Q is added using the following lines:

```
for i, nd in enumerate(Nd):
    f+=[0, 1] if (nd==[width,height]).all()\
        else [0, 0]
```

Table 2: Cantilever example: effect of using Mosek LP solver (small to moderately large problems)

Nodal divisions	Number of nodes	Number of members	Volume ($\frac{PL}{\sigma_0}$)	Error (%)*	With 'member adding'			With 'member adding' speed up factor
					LP time† (s)	Dual time‡ (s)	Total time (s)	
20×10	231	16,290	7.0747	0.71	0.9	0.05	0.95	0.6
40×20	861	225,848	7.0454	0.30	3.3	0.6	3.9	2.4
60×30	1,891	1,086,938	7.0376	0.18	13.2	2.9	16.1	4.0
80×40	3,321	3,352,500	7.0337	0.13	37.7	7.8	45.5	5.6
100×50	5,151	8,067,890	7.0312	0.092	96.8	18.5	115.3	11.7
120×60	7,381	16,559,996	7.0300	0.075	270.6	48.1	318.7	⋆
140×70	10,011	30,462,670	7.0291	0.063	602.9	97.4	700.3	⋆
160×80	13,041	51,699,820	7.0284	0.053	1921.9	242.5	2164.4	⋆
180×90	16,471	82,475,558	7.0279	0.046	2496.6	264.6	2761.2	⋆
200×100	20,301	125,265,288	7.0275	0.039	6296.9	515.7	6812.6	⋆

∗: compared with the exact answer, $V = 7.024707829\frac{PL}{\sigma_0}$, given by Graczykowski and Lewiński (2010)
†: cumulative CPU time spent in function `solveLP`
‡: cumulative CPU time spent in function `stopViolation`
⋆: problems with high numbers of nodal divisions could only be solved with 'member adding' due to memory usage

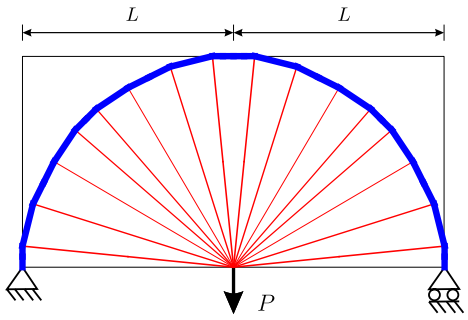


Fig. 8: 'Half wheel' example: problem definition and layout obtained using 20×10 nodal divisions

And the point load P is moved to the bottom-right corner by replacing the original line 74 with:

```
f+=[0,-1] if (nd==[width,0]).all() else [0,0]
```

Initially all load cases are concatenated in vector `f`, which is then reformatted to a $p \times 2n$ array in line 83.

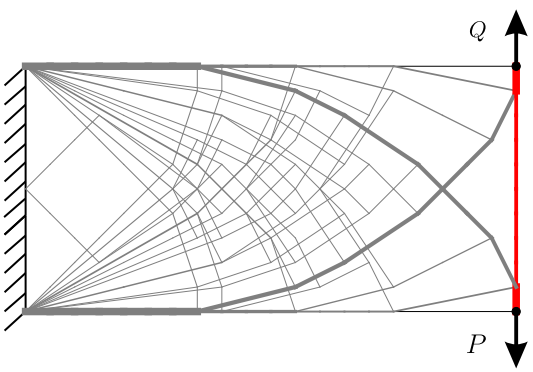


Fig. 9: Two load case cantilever example: problem specification and layout obtained using 20×10 nodal divisions

4.7 Non-convex domain

Various polygonal domains can be specified using `shapely`. For example, to add a hole to the design domain, as shown in Fig. 10, the following lines can be added after line 65:

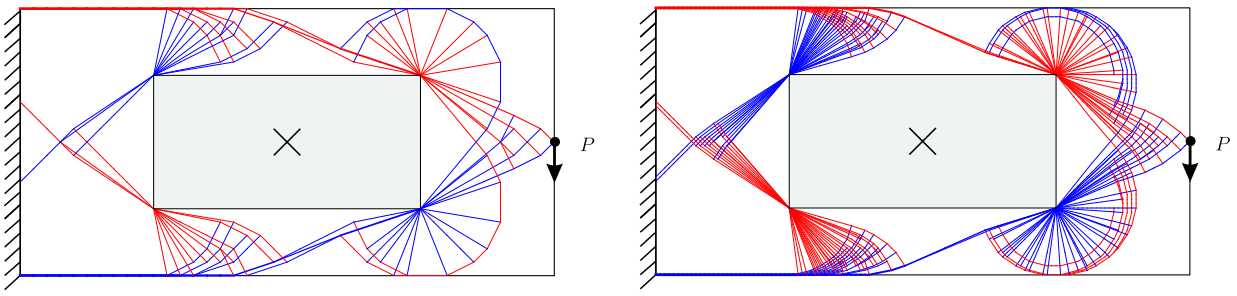
```
poly=poly.difference(Polygon([ \
                 (width/4,   height/4),\
                 (width/4*3, height/4),\
                 (width/4*3, height/4*3),\
                 (width/4,   height/4*3)]))
```

This subtracts a rectangular region from the original design domain in Fig. 4. Note that, when a non-convex domain is used, lines that intersect domain boundary should be excluded from the 'ground structure'. Line 80 checks such intersections using the `contains` function from `shapely`. Since this can become computationally expensive when the number of potential connections is large, it is skipped when a convex domain is used.

5 Conclusions

A simple Python implementation of truss layout optimization using an adaptive 'member adding' scheme has been presented for educational use. Various examples are used to demonstrate the efficacy of the script for problems involving single and multiple load cases, unequal limiting stresses in tension and compression, joint-costs, and non-convex polygonal domains, with or without holes. The solutions obtained are globally optimal for the prescribed nodal discretization and can be used to benchmark other methods or to provide inspiration for structural designers.

The 98-line Python script described is listed for reference in Appendix B and is also provided in the form of downloadable source code; see Supplementary material section.



(a) (b)

Fig. 10: Cantilever example with hole: problem specification and layout obtained using: (a) 40×20 nodal divisions; (b) 120×60 nodal divisions

6 Acknowledgements

The first two authors acknowledge the financial support of EPSRC, under grant reference EP/N023471/1.

7 Supplementary material

Downloadable Python script, `trussopt.py`, and installation files:
https://figshare.com/s/0ec40d238819ae4ec6ec

8 Conflict of interest

On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

Agrawal A, Verschueren R, Diamond S, Boyd S (2018) A rewriting system for convex optimization problems. J Control Decision 5(1):42–60

Bolbotowski K, He L, Gilbert M (2018) Design of optimum grillages using layout optimization. Struct Multidisc Optim 58(3):851–868

Boyd S, Vandenberghe L (2004) Convex optimization. Cambridge University Press

Dantzig GB, Wolfe P (1960) Decomposition principle for linear programs. Oper Res 8(1):101–111

Darwich W, Gilbert M, Tyas A (2010) Optimum structure to carry a uniform load between pinned supports. Struct Multidisc Optim 42(1):33–42

Diamond S, Boyd S (2016) CVXPY: A Python-embedded modeling language for convex optimization. J Mach Learn Res 17(83):1–5

Domahidi A, Chu E, Boyd S (2013) ECOS: An SOCP solver for embedded systems. In: European Control Conference (ECC), pp 3071–3076

Dorn WS, Gomory RE, Greenberg HJ (1964) Automatic design of optimal structures. J Mècanique 3:25–52

Fairclough HE, Gilbert M, Pichugin AV, Tyas A, Firth I (2018) Theoretically optimal forms for very long-span bridges under gravity loading. Proc R Soc A 474(2217):20170,726

Gilbert M, Tyas A (2003) Layout optimization of large-scale pin-jointed frames. Eng Comput 20(8):1044–1064

Gilbert M, He L, Smith C, Le C (2014) Automatic yield-line analysis of slabs using discontinuity layout optimization. Proc R Soc A 470

Graczykowski C, Lewiński T (2010) Michell cantilevers constructed within a half strip. tabulation of selected benchmark results. Struct Multidisc Optim 42(6):869–877

He L, Gilbert M (2015) Rationalization of trusses generated via layout optimization. Struct Multidisc Optim 52(4):677–694

He L, Gilbert M, Johnson T, Pritchard T (2018) Conceptual design of am components using layout and geometry optimization. Computers & Mathematics with Applications (in press)

Parkes E (1978) Joints in optimum frameworks. Int J Solids Struct 11(9):1017–1022

Pritchard T, Gilbert M, Tyas A (2005) Plastic layout optimization of large-scale frameworks subject to multiple load cases, member self-weight and with joint length penalties. 6th World Congresses of Structural and Multidisciplinary Optimization, Rio de Janeiro, Brazil

Sigmund O (2001) A 99 line topology optimization code written in Matlab. Struct Multidisc Optim 21(2):120–127

Smith C, Gilbert M (2007) Application of discontinuity layout optimization to plane plasticity problems. Proc R Soc A 463:2461–2484

Smith CJ, Gilbert M, Todd I, Derguti F (2016) Application of layout optimization to the design of additively manufactured metallic components. Struct Multidisc Optim 54(5):1297–1313

Sokół T (2011a) A 99 line code for discretized Michell truss optimization written in Mathematica. Struct Multidisc Optim 43(2):181–190

Sokół T (2011b) Topology optimization of large-scale trusses using ground structure approach with selective subsets of active bars. In: 19th International Conference on 'Computer Methods in Mechanics', CMM2011, ACM Press, pp 457–458

Sokół T (2014) Multi-load truss topology optimization using the adaptive ground structure approach. In: Lodygowski T, Rakowski J, Litewka P (eds) Recent Advances in Computational Mechanics, CRC Press, pp 9–16

Sokół T, Rozvany GIN (2012) New analytical benchmarks for topology optimization and their implications. Part I: bi-symmetric trusses with two point loads between supports. Struct Multidisc Optim 46(4):477–486

Tyas A, Gilbert M, Pritchard TJ (2006) Practical plastic layout optimization of trusses incorporating stability considerations. Comput Struct 84(3-4):115–126

Tyas A, Pichugin A, Gilbert M (2011) Optimum structure to carry a uniform load between pinned supports: exact analytical solution. Proc R Soc A 467(2128):1101–1120

Vanderbei RJ (2001) Linear programming: foundations and extensions, 2nd edn. Springer Verlag

Wei P, Li Z, Li X, Wang MY (2018) An 88-line MATLAB code for the parameterized level set method based topology optimization using radial basis functions. Struct Multidisc Optim 58(2):831–849

Zegard T, Paulino GH (2014) GRAND - Ground structure based topology optimization for arbitrary 2D domains using MATLAB. Struct Multidisc Optim 50(5):861–882

Zegard T, Paulino GH (2015) GRAND3 - Ground structure based topology optimization for arbitrary 3D domains using MATLAB. Struct Multidisc Optim 52(6):1161–1184

A Dual problem formulation

The adaptive solution scheme requires information from the dual problem, the derivation of which is now described.

Firstly the objective function (3a) is augmented by taking into account constraints (3b) and (3c) using a weighted sum, formulating the so-called Lagrange function, written as:

$$\begin{aligned}\mathcal{L}(\mathbf{a},\mathbf{q},\mathbf{u},\mathbf{z}_1,\mathbf{z}_2,\mathbf{z}_3) = \; & \tilde{\mathbf{l}}^{\mathrm{T}}\mathbf{a} + \sum_{k=1}^{p}\left(\mathbf{f}^k - \mathbf{B}\mathbf{q}^k\right)^{\mathrm{T}}\mathbf{u}^k \\ & + \sum_{k=1}^{p}\left(-\sigma^{-}\mathbf{a} - \mathbf{q}^k\right)^{\mathrm{T}}\mathbf{z}_1^k \\ & + \sum_{k=1}^{p}\left(-\sigma^{+}\mathbf{a} + \mathbf{q}^k\right)^{\mathrm{T}}\mathbf{z}_2^k \\ & - \mathbf{a}^{\mathrm{T}}\mathbf{z}_3, \end{aligned} \tag{8}$$

where, $\mathbf{u}^k, \mathbf{z}_1^k$ and $\mathbf{z}_2^k$ are Lagrange multipliers corresponding to constraints in (3b) in load case k, and $\mathbf{z}_3$ to area constraint in (3c). $\mathbf{q} = [\mathbf{q}^{1\,\mathrm{T}}, ..., \mathbf{q}^{p\,\mathrm{T}}]^{\mathrm{T}}$ is a vector containing internal forces in all load cases, and same for $\mathbf{u}$, $\mathbf{z}_1$ and $\mathbf{z}_2$. The Lagrange dual of (3) can be derived using (readers interested in duality theory can refer e.g. to Boyd and Vandenberghe (2004)):

$$\max_{\mathbf{u},\mathbf{z}_1,\mathbf{z}_2,\mathbf{z}_3} \; \mathcal{L}(\mathbf{a},\mathbf{q},\mathbf{u},\mathbf{z}_1,\mathbf{z}_2,\mathbf{z}_3) \tag{9a}$$

s.t.

$$\nabla_{\mathbf{a}}\mathcal{L}(\mathbf{a},\mathbf{q},\mathbf{u},\mathbf{z}_1,\mathbf{z}_2,\mathbf{z}_3) = \mathbf{0} \tag{9b}$$

$$\nabla_{\mathbf{q}}\mathcal{L}(\mathbf{a},\mathbf{q},\mathbf{u},\mathbf{z}_1,\mathbf{z}_2,\mathbf{z}_3) = \mathbf{0} \tag{9c}$$

$$\mathbf{z}_1,\mathbf{z}_2,\mathbf{z}_3 \geq \mathbf{0}, \tag{9d}$$

which gives:

$$\max_{\mathbf{u},\mathbf{z}_1,\mathbf{z}_2,\mathbf{z}_3} \; W = \sum_{k=1}^{p}\mathbf{f}^{k\,\mathrm{T}}\mathbf{u}^k \tag{10a}$$

s.t.

$$\sum_{k=1}^{p}\left(\sigma^{-}\mathbf{z}_1^k + \sigma^{+}\mathbf{z}_2^k\right) + \mathbf{z}_3 = \tilde{\mathbf{l}} \tag{10b}$$

$$\mathbf{B}^{\mathrm{T}}\mathbf{u}^k + \mathbf{z}_1^k - \mathbf{z}_2^k = \mathbf{0} \text{ for } k = 1, 2, ..., p \tag{10c}$$

$$\mathbf{z}_1,\mathbf{z}_2,\mathbf{z}_3 \geq \mathbf{0}, \tag{10d}$$

where, W is the virtual work done by the applied loads. The Lagrange multiplier $\mathbf{u}^k$ corresponds to the virtual displacement of nodes in load case k. Note that, since the two inequality constraints in (3b) will not be active simultaneously (as a member cannot be in both tension and compression under one load case), either z_1^k or z_2^k must be zero for any member. Therefore, constraints (10b) and (10c) can be reformulated by converting the equality constraint to inequality constraints using non-negative dual variables, as follows:

$$\sum_{k=1}^{p}\mathbf{z}^k \leq \tilde{\mathbf{l}} \tag{11a}$$

$$\left.\begin{aligned} -\frac{\mathbf{z}^k}{\sigma^{-}} \leq \mathbf{B}^{\mathrm{T}}\mathbf{u}^k \leq \frac{\mathbf{z}^k}{\sigma^{+}} \\ \mathbf{z}^k \geq \mathbf{0} \end{aligned}\right\} k = 1, 2, ...p, \tag{11b}$$

where, $\mathbf{z}^k$ is a new variable which combines $\mathbf{z}_1^k$ and $\mathbf{z}_2^k$. For each member, (11) can alternatively be written as a single expression:

$$\epsilon_i = \sum_{k=1}^{p}\frac{\max\left\{\sigma^{+}\mathbf{B}_i^{\mathrm{T}}\mathbf{u}_i^k, \;\; -\sigma^{-}\mathbf{B}_i^{\mathrm{T}}\mathbf{u}_i^k\right\}}{\tilde{l}_i} \leq 1, \qquad (\text{for } i = 1, ..., m). \tag{12}$$

where ϵ_i is the summed maximum virtual strain of member i in all load cases; this expression is reproduced in (4).

B Setting up Python environment

B.1 Basic setup

The script is designed to run with Python 3 (version 3.5 and later; tested with version 3.7). The script can also be used with Python 2 but some modifications are required - see Appendix B.2 for details.

The Anaconda package (64-bit version freely available at https://www.anaconda.com/download) includes a Python distribution and development environment and is recommended for new Python users. The following platform-specific instructions that install the required tools to allow the Python script to run assume that Anaconda is being used:

Windows: An Anaconda Prompt window should be opened (in Anaconda Navigator, open a terminal from base(root) in the Environments tab) and the following batch file (provided with the Python script) executed:

```
install.bat
```

Linux & Mac: Assuming that conda has is referenced in the `PATH` variable (e.g. via the `.bashrc` file), the following shell script (provided with the Python script) should be executed:

```
bash install.sh
```

Manual install: Alternatively the required tools can be installed on any platform one-by-one by starting an Anaconda Prompt window and entering the following commands:

```
conda install numpy=1.15.4
conda install scipy=1.1.0
conda install shapely=1.6.4
conda install -c cvxgrp cvxpy=0.4.*
conda install matplotlib=2.2.3
```

Note that the commands above, and the script and batch file, specify particular versions of the relevant tools, that have been tested to be compatible. Alternatively the version numbers can be omitted to ensure the latest versions of these tools are used, though in this case compatibility is not guaranteed.

B.2 Installation with Python 2

It is possible to run the script with Python 2 (2.6 and later) by making minor modifications to the script, as detailed below.

First, remove the `gcd` method from `math` in the first line:

```
from math import ceil
```

And instead import it from `fractions` instead, by adding the following new line:

```
from fractions import gcd
```

Also, line 47 is replaced with:

```
for i in range(int(num)):
```

to take into account the fact that function `ceil` returns a floating point rather than integer value in Python 2.

B.3 Alternative solvers

To use an alternative solver, as described in Section 4.2, this also needs be installed. For example, MOSEK can be installed using the following command:

```
conda install -c mosek mosek
```

In addition, a licence file is required, which can be freely obtained for academic use from the MOSEK website (https://www.mosek.com).

B.4 Python IDE

For beginners, the Spyder IDE (Integrated Development Environment) bundled with the Anaconda package provides an easy-to-use means of editing and running the script.

C Python script

```
from math import gcd, ceil
import itertools
from scipy import sparse
import numpy as np
import cvxpy as cvx
import matplotlib.pyplot as plt
from shapely.geometry import Point, LineString, Polygon
#Calculate equilibrium matrix B
def calcB(Nd, Cn, dof):
    m, n1, n2 = len(Cn), Cn[:,0].astype(int), Cn[:,1].astype(int)
    l, X, Y = Cn[:,2], Nd[n2,0]-Nd[n1,0], Nd[n2,1]-Nd[n1,1]
    d0, d1, d2, d3 = dof[n1*2], dof[n1*2+1], dof[n2*2], dof[n2*2+1]
    s = np.concatenate((-X/l * d0, -Y/l * d1, X/l * d2, Y/l * d3))
    r = np.concatenate((n1*2, n1*2+1, n2*2, n2*2+1))
    c = np.concatenate((np.arange(m), np.arange(m), np.arange(m), np.arange(m)))
    return sparse.coo_matrix((s, (r, c)), shape = (len(Nd)*2, m))
#Solve linear programming problem
def solveLP(Nd, Cn, f, dof, st, sc, jc):
    l = [col[2] + jc for col in Cn]
    B = calcB(Nd, Cn, dof)
    a = cvx.Variable(len(Cn))
    obj = cvx.Minimize(np.transpose(l) * a)
    q, eqn, cons= [],  [], [a>=0]
    for k, fk in enumerate(f):
        q.append(cvx.Variable(len(Cn)))
        eqn.append(B * q[k] == fk * dof)
        cons.extend([eqn[k], q[k] >= -sc * a, q[k] <= st * a])
    prob = cvx.Problem(obj, cons)
    vol = prob.solve()
    q = [np.array(qi.value).flatten() for qi in q]
    a = np.array(a.value).flatten()
    u = [-np.array(eqnk.dual_value).flatten() for eqnk in eqn]
    return vol, a, q, u
#Check dual violation
def stopViolation(Nd, PML, dof, st, sc, u, jc):
    lst = np.where(PML[:,3]==False)[0]
    Cn = PML[lst]
    l = Cn[:,2] + jc
    B = calcB(Nd, Cn, dof).tocsc()
    y = np.zeros(len(Cn))
    for uk in u:
        yk = np.multiply(B.transpose().dot(uk) / l, np.array([[st], [-sc]]))
        y += np.amax(yk, axis=0)
    vioCn = np.where(y>1.0001)[0]
    vioSort = np.flipud(np.argsort(y[vioCn]))
    num = ceil(min(len(vioSort), 0.05*max( [len(Cn)*0.05, len(vioSort)])))
    for i in range(num):
        PML[lst[vioCn[vioSort[i]]]][3] = True
    return num == 0
#Visualize truss
def plotTruss(Nd, Cn, a, q, threshold, str, update = True):
    plt.ion() if update else plt.ioff()
    plt.clf(); plt.axis('off'); plt.axis('equal');  plt.draw()
    plt.title(str)
    tk = 5 / max(a)
    for i in [i for i in range(len(a)) if a[i] >= threshold]:
        if all([q[lc][i]>=0 for lc in range(len(q))]): c = 'r'
        elif all([q[lc][i]<=0 for lc in range(len(q))]): c = 'b'
        else: c = 'tab:gray'
        pos = Nd[Cn[i, [0, 1]].astype(int), :]
        plt.plot(pos[:, 0], pos[:, 1], c, linewidth = a[i] * tk)
    plt.pause(0.01) if update else plt.show()
#Main function
def trussopt(width, height, st, sc, jc):
    poly = Polygon([(0, 0), (width, 0), (width, height), (0, height)])
    convex = True if poly.convex_hull.area == poly.area else False
    xv, yv = np.meshgrid(range(width+1), range(height+1))
    pts = [Point(xv.flat[i], yv.flat[i]) for i in range(xv.size)]
    Nd = np.array([[pt.x, pt.y] for pt in pts if poly.intersects(pt)])
```

```
    dof, f, PML = np.ones((len(Nd),2)), [], []
    #Load and support conditions
    for i, nd in enumerate(Nd):
        if nd[0] == 0: dof[i,:] = [0, 0]
        f += [0, -1] if (nd == [width, height/2]).all() else [0, 0]
    #Create the 'ground structure'
    for i, j in itertools.combinations(range(len(Nd)), 2):
        dx, dy = abs(Nd[i][0] - Nd[j][0]), abs(Nd[i][1] - Nd[j][1])
        if gcd(int(dx), int(dy)) == 1 or jc != 0:
            seg = [] if convex else LineString([Nd[i], Nd[j]])
            if convex or poly.contains(seg) or poly.boundary.contains(seg):
                PML.append( [i, j, np.sqrt(dx**2 + dy**2), False] )
    PML, dof = np.array(PML), np.array(dof).flatten()
    f = [f[i:i+len(Nd)*2] for i in range(0, len(f), len(Nd)*2)]
    print('Nodes: %d Members: %d' % (len(Nd), len(PML)))
    for pm in [p for p in PML if p[2] <= 1.42]:
        pm[3] = True
    #Start the 'member adding' loop
    for itr in range(1, 100):
        Cn = PML[PML[:,3] == True]
        vol, a, q, u = solveLP(Nd, Cn, f, dof, st, sc, jc)
        print("Itr: %d, vol: %f, mems: %d" % (itr, vol, len(Cn)))
        plotTruss(Nd, Cn, a, q, max(a) * 1e-3, "Itr:" + str(itr))
        if stopViolation(Nd, PML, dof, st, sc, u, jc): break
    print("Volume: %f" % (vol))
    plotTruss(Nd, Cn, a, q, max(a) * 1e-3, "Finished", False)
#Execution function when called directly by Python
if __name__ =='__main__':
    trussopt(width = 20, height = 10, st = 1, sc =1, jc = 0)
##########################################################################
# This Python script was written by L. He, M. Gilbert, X. Song           #
# University of Sheffield, United Kingdom                                #
# Please send comments to: linwei.he@sheffield.ac.uk                     #
# The script is intended for educational purposes - theoretical details  #
# are discussed in the following paper, which should be cited in any     #
# derivative works or technical papers which use the script:             #
#                                                                        #
# "A Python script for adaptive layout optimization of trusses",         #
# L. He, M. Gilbert, X. Song, Struct. Multidisc. Optim., 2018            #
#                                                                        #
# Disclaimer:                                                            #
# The authors reserve all rights but do not guarantee that the script is #
# free from errors. Furthermore, the authors are not liable for any      #
# issues caused by the use of the program.                               #
##########################################################################
```