4.450 Project Report
Exploration of Deviations from FL-Optimized Truss Shapes Under Buckling Constraints

Gavin Ridley

December 2, 2020

NOTE: I need a better title. When I say FL-optimized, I mean optimization of the summed force magnitudes times lengths. Also, another note: I have tested my code for the FL-optimization, but tested optimization with areas of members in compression calculated by Euler buckling. I have most of the formulas added now though, and don't expect this to be hard. I have tested parallelization of my code and found it to work successfully.

1 Abstract

Discrete volume-optimal truss shapes under a given truss topology have been explored by several studies, but these tend to restrict analysis to not consider effects such as local and global buckling in accord with Michell's original analyses. Because volume-optimized truss shapes under consideration of buckling depend on both material properties and member geometry, these nonideal cases have not been explored in the literature. In this work, the truss shape optimization problem on a given topology is solved for both the idealized case and buckling-constrained case using an MPI-parallelized implementation of constrained differential evolution using the open-source Frame3DD code as the physics backend of the optimization. Comparisons are drawn between each geometry. Source code is available at `https://github.com/gridley/Frame3DD-Opt`.

2 Introduction

The shape of optimal truss structures under various simplifying assumptions and simple loading patterns is known for a given topology in both 2D [14, 9] and 3D [7]. These geometries are closely related to Michell's continuous optimum truss solution [10]. These results, though, rely on a few assumptions which tend not to hold in real applications. To name a few, these results neglect global buckling, and penalize tensile stresses in the same way compressive stresses are

penalized. While these optimal structures indeed solve the minimum compliance problem under a given topology, they do not necessarily solve the minimum mass problem once buckling constraints are taken into account–including both local and global buckling. The main contribution of this paper is to depict how discrete optimal truss geometries change depending on material properties when under realistic buckling constraints, with particular attention paid to perturbation of discrete Michell truss geometries under these constraints for the mass minimization problem.

While a multitude of open-source truss and frame analysis tools can be found online, the majority of these focus on topology optimization. I designed a new truss optimization program capable of performing parallel differential evolution on the largest of supercomputers. The code uses the validated Frame3DD frame analysis code [4] to solve the solid mechanics physics. Frame3DD can do truss analysis calculations with linear elasticity, or nonlinear elasticity making consideration of the geometric stiffness matrix. The code calculates whether a structure buckles under loading by outputting a specific error code when the stiffness matrix loses the positive-definite property, which indicates that a buckling instability has been reached given the current load. The code can also model the shear of truss members, but I left that option off for the analyses presented here. Notably, frame joints are treated as rigid rather than pinned and thus resist moments, which is different from the typical academic assumption on trusses of non-moment-resistsing joints. Notably, by taking the approach of checking the stiffness matrix for semidefiniteness, local buckling is automatically considered in addition to global buckling since both effects lead to a loss of semi-definiteness. For more information on elastic stability, the monograph [18] is recommended.

Frames or trusses have a tendency to fail in buckling well before ideal tensile or compressive yield limits are reached–analogous to how a beam under compression tends to fail in buckling well before it fails due to compressive yielding. This can easily be shown to be true using Euler's buckling formula, available from any mechanics of materials text e.g. [5], and comparing this buckling stress to that where yielding occurs. For most geometries relevant to structural members, buckling happens well before yielding.

Multiple prior works have considered optimization of truss structures under both local and global buckling constraints. Early work on the handling of local buckling constraints in truss optimization was explored in the two-part study [21]. It was soon realized by the same authors that consideration of only local buckling is a fundamentally flawed and incorrect approach for truss optimization in the works [20, 15]. The aforementioned work [20] is an especially recommended read due to its brevity and presentation of specific, incisive examples which clearly undermine the credibility of any truss optimization which only considers local buckling.

To address this, many studies have explored both shape and topology optimization of trusses under both global and local buckling constraints. The studies examining buckling-constrained topology optimization, presenting their own specialized methods, include the member sizing optimization of [1] which

could easily be applied to a ground structure, as well as [8] which solved solved several truss topology optimization problems under a global stability constraint.

This study does not employ topology optimization by reason of practical considerations. True, leaving the truss topology severely artificially contracts the design space in which solutions should be found. While this it is indeed restrictive to specify a topology in advance, the motivation of this work is aimed towards simply constructible designs as could be done with lumber from a home improvement store in a backyard. A topology-optimized truss will usually have a multitude of nodes tangled by members of a variety of thicknesses, e.g. the results presented in [2]. In contrast, the geometric complexity of a *shape-optimized* truss remains approximately constant so long as the topology is fixed, even if the connecting joint positions have been moved. As such, these structures remain constructible for small-scale projects.

TODO get graphic of complicated, clearly unconstructible topology optimization result

Multiple other works have optimized trusses using the node movement approach. [19] also used evolutionary optimization to modify the nodal positions of a truss as in this work, but without consideration of buckling. In constrast to the work [19], checks that a local optimum is indeed obtained after the program terminates via evaluation of the Karush-Kuhn-Tucker condition is not made, although this could be explored in the future.

Being an evolutionary algorithm, this approach would straightforwardly fit into the interactive evolutionary interface specification developed in chapter three of [11].

In this regard, the work [12] also particularly focuses on practicalities, employing an optimization of frame node positions under physically realistic constraints including displacement, stress, characteristic frequencies, and buckling. The work [12] is likely the study published thus far most related to this one. While this study does not study constraints on displacement or vibrational modes, it seeks to further the work of [12] by exploring the influence of practical constraints on geometries considered to be optimal in simplified truss physics models.

The analysis method differs from the work [12] in a few ways though. Firstly, as previously mentioned, no constraints are placed on the characteristic frequencies of the truss, nor is any consideration made regarding displacements. Only phenomena which would physically destroy the structure–yielding or buckling–are considered. Morever, while the study [12] used Danish standard methods for accounting for engineering uncertainty via safety factor techniques applied to the elasticity modulus and some other parameters, our analysis simply employs an arbitrary safety factor applied to either the yield stress, local buckling stress, or global buckling load.

Table 1: Available modes in Frame3DD-Opt

Mode	Functionality	Formal Definition
1	No buckling, minimizes $\sum_i \|F_i\|L_i$	Eq. 2
2	Local buckling, minimizes frame mass	Eq. 3
3	Local and global buckling, minimizes frame mass	Eq. 4

3 Methods

An implementation of the differential evolution algorithm as originally presented by [17] has been implemented in Python. As noted in the work [17], differential evolution easily parallelizes, and this has been done via the Message Passing Interface (MPI) [6] bindings for Python [3]. The outer optimization loop, written in an interpreted scripting language, wraps a low-level, optimized frame physics solver called Frame3DD, which is available under the GNU license [4]. Because the truss physics is the computationally intensive part of the algorithm, Frame3DD was recompiled using the `-Ofast` flag in `gcc` [16] which does not guarantee standard floating point behavior, but gives enormous code performance. Solutions obtained with and without this compiler flag were verified to agree. The script has been named `Frame3DD-Opt`, and can be found at `https://github.com/gridley/Frame3DD-Opt`.

This code has been written to operate in three modes–the first being a linear elasticity problem only without consideration of buckling as commonly employed in [11] and the MIT course 4.450, and the second making consideration of local buckling. The third also considers a global buckling constraint. These three modes are summarized by Table 1. In all modes, only the node positions are the variables to optimize on, rather than having member thickness as an additional variable as in [12]. This choice was made in order to reduce the dimensionality of the problem.

For the mode not considering buckling, the frame solution is essentially just a statics solver, so the member thicknesses negligibly affect the solution. Joints do resist moments in Frame3DD though, so this statement is only approximately true. In this mode, the quantity $\sum_i |F_i|L_i$ is the objective variable, which is known to be equivalent to minimizing frame mass under simple assumptions which neglect buckling [11].

In the second mode, local buckling is accounted for via the Euler buckling formula. In this code, since the joints resist moments, I have verified that the buckling effect is well-predicted as the case where each end of the member is clamped against both rotation and translational movement. To review this formula which can be found in [5], the buckling stress is simply:

$$\sigma = \frac{\pi^2 EI}{(KL)^2} \tag{1}$$

E is the modulus of elasticity, I the minimal area moment of inertia perpendicular to the loading axis, and L the member length. Theoretically, given

the previously mentioned conditions, K should be taken to be 0.5. However, for design purposes, this is instead commonly taken to be 0.65 to account for defects in materials and other engineering uncertainties. The same convention is adopted here.

Given a set of truss node coordinates, to evaluate the objective function in one of the latter two nodes, a linear elasticity calculation is first performed using large member thicknesses. After this, the members are sized using a safety factor applied to either the tensile yield stress limit or the same safety factor and the Euler buckling formula for respectively tensioned and compressed members. After sizing the members using a linear elasticity approach, in mode two, the frame mass is calculated and this is returned as the objective function. In mode three, an additional nonlinear elasticity calculation is performed. If the nonlinear stiffness matrix is not positive definite, the design is marked as infeasible since it will buckle either locally or globally. If the nonlinear stiffness matrix is positive definite, the design is feasible and the mass of the frame is calculated and this is returned as the objective function value.

To formally state each of the aforementioned modes' optimization problems, the mode one optimization solves:

$$\begin{aligned} &\min_{p \in B} \sum_{i=1}^{m} L_i(p)|F_i(p)| \\ \text{s.t.} \quad &K(p)u = f \end{aligned} \tag{2}$$

m denotes the number of bars in the frame, and n the number of nodes. Where $B \subset \mathbb{R}^n$, represents the locations where truss nodes may be located. p is a length n vector of the unknown node variables in a Cartesian product of closed intervals of the real line, as the code is currently set up. Of course, more exotic search regions could be implemented to describe B with some effort. $K(p)$ is the stiffness matrix calculated by Frame3DD, and u is the resulting displacement vector from Frame3DD's linear solve with the nodal loading vector f. The forces in each member $F_i(p)$ are obtained from Frame3DD output, with negative values denoting member compressions and positive tensions. The L_i are the lengths of each member. We know from [11] that if the above objective function is minimized, the mass of the structure is minimized as well.

For the second mode, indeed, the mass of the structure should be minimized. However, it is no longer true that minimizing the sum of the absolute values of forces times member lengths minimizes the truss mass. Instead, because local buckling puts differing sizing critera for compressive stresses and tensile stresses,

a different quantity must be minimized:

$$\begin{aligned} & \min_{p \in B} \sum_{i=1}^{m} L_i(p) A_i \\ \text{s.t.} \quad & K(p)u = f \\ & A_i = \begin{cases} \gamma F_i / \sigma_y & F_i \geq 0 \\ \max\left(I^{-1}\left(\frac{|F_i|(0.65 L_i)^2 \gamma}{\pi^2 E} \right), \frac{\gamma |F_i|}{\sigma_y} \right) & F_i < 0 \end{cases} \end{aligned} \tag{3}$$

Where σ_y is the yield stress of the material under consideration, A_i the area of member i, γ is a user-defined safety factor, and $I^{-1} : \mathbb{R} \to \mathbb{R}$ is a function which maps minimal member moments of area in the member-perpendicular plane to member areas. This map is only uniquely defined under a restriction of the geometric properties of a member, e.g. a few aspect ratios. E is the elasticity modulus of the material under consideration.

Lastly, in the third mode, geometric stiffness effects are accounted for in the calculation, and a constraint is placed on the problem that the nonlinear stiffness matrix be positive definite:

$$\begin{aligned} & \min_{p \in B} \sum_{i=1}^{m} L_i(p) A_i \\ \text{s.t.} \quad & \big(K(p) - G(p,u)\big)\, u = f \\ & A_i = \begin{cases} \gamma F_i / \sigma_y & F_i \geq 0 \\ \max\left(I^{-1}\left(\frac{|F_i|(0.65 L_i)^2 \gamma}{\pi^2 E} \right), \frac{\gamma |F_i|}{\sigma_y} \right) & F_i < 0 \end{cases} \\ & A(p) - G(p,u) \succ 0 \end{aligned} \tag{4}$$

Where G is the geometric stiffness matrix, which is updated by Frame3DD's nonlinear solver. If the matrix $K - G$ gains any nonpositive eigenvalue, Frame3DD returns a specific error code, which is processed by my wrapper program to reject the proposed solution as infeasible.

Sizing members to be resistant against local buckling requires knowledge of the moment of area of a given member. However, the area of the member is dependent on the result of the linear elasticity calculation. As such, some invariant geometric factors must additionally be fixed in order to make the problem well-defined. The code currently has formulas built in for either circular or square tubes. For the former, the ratio of inner to outer radius is taken to be fixed and is a user-controlled parameter. For the latter, the ratio of the wall thickness to the width of the member is fixed as well. The formulas for the moments of area as a function of area are then calculated by: TODO I have coded these but not typeset them.

TODO: describe parallelism methodology

TODO: Flowchart for Mode 1, 2, 3 calculations

The population size in differential evolution should scale with the number of variables, if possible. The rule of thumb proposed by [13]

Other things

- Neglect gravity loading
- orientation of square tubes aligned with Cartesian axes
- Buckling model is not linear–true nonlinear buckling

4 Results

4.1 Bridge-like Truss with Vertically Moved Nodes

TODO: This is the HW4 problem. I have tested my solver on it, and found reasonable results, but am yet to check the influence of accounting for buckling.

4.2 Bridge-like Truss With Free Nodes

TODO: Same as the previous section, but let the nodes move horizontally as well. I have done this calculation too, and it can be found in the github repository.

4.3 Discrete 2D Michell Truss

In this section, I'll check that my solver can converge to that first diagram in Mazurek's paper. This problem has nearly 100 dimensions, and I will use the parallelized version of my solver here. I have access to the Sawtooth supercomputer at Idaho National Lab (and am experienced on it), and plan to run the optimization there using a population of around 500 as per a rule of thumb in a paper I will cite. I will show a convergence plot of the calculation over each evolution.

After that, this is where showing the results where Euler buckling has been accounted for will be really interesting. I think the structure will lose its symmetry about the axis dividing its supports. I will then check if the obtained design is subject to any global buckling failure, and if so, add that constraint and re-run my solver. Otherwise, I will not consider global buckling.

5 Conclusions

Code works, good results found, ran big parallel problem

Future work

- Checking of KKT conditions after run
- "Polishing" of solution via gradient-based method after termination of differential evolution

- Multiple loading conditions
- Other differential evolution selection rules

References

[1] Aharon Ben-Tal et al. "Optimal Design of Trusses Under a Nonconvex Global Buckling Constraint". In: *Optimization and Engineering* 1.2 (July 1, 2000), pp. 189–213. ISSN: 1573-2924. DOI: 10.1023/A:1010091831812. URL: https://doi.org/10.1023/A:1010091831812 (visited on 11/23/2020).

[2] Martin P. Bendsøe and Ole Sigmund. "Topology Design of Truss Structures". In: *Topology Optimization: Theory, Methods, and Applications*. Ed. by Martin P. Bendsøe and Ole Sigmund. Berlin, Heidelberg: Springer, 2004, pp. 221–259. ISBN: 978-3-662-05086-6. DOI: 10.1007/978-3-662-05086-6_4. URL: https://doi.org/10.1007/978-3-662-05086-6_4 (visited on 11/29/2020).

[3] Lisandro D. Dalcin et al. "Parallel Distributed Computing Using Python". In: *Advances in Water Resources*. New Computational Methods and Software Tools 34.9 (Sept. 1, 2011), pp. 1124–1139. ISSN: 0309-1708. DOI: 10.1016/j.advwatres.2011.04.013. URL: http://www.sciencedirect.com/science/article/pii/S0309170811000777 (visited on 11/29/2020).

[4] Henri Gavin. *User Manual and Reference for Frame3DD: A Structural Frame Analysis Program*. URL: http://svn.code.sourceforge.net/p/frame3dd/code/trunk/doc/Frame3DD-manual.html (visited on 11/28/2020).

[5] James M. Gere and Stephen P. Timoshenko. *Mechanics of Materials*. 4th Instrs edition. Boston: Pws Pub Co, Nov. 1, 1996. ISBN: 978-0-534-95102-3.

[6] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface*. second edition. Cambridge, Mass: The MIT Press, Nov. 26, 1999. 350 pp. ISBN: 978-0-262-57132-6.

[7] Benjamin (Benjamin Paul Emmanuel) Jacot. "A Strain Tensor Method for Three-Dimensional Optimal Michell Structures". Thesis. Massachusetts Institute of Technology, 2016. URL: https://dspace.mit.edu/handle/1721.1/104125 (visited on 11/01/2020).

[8] M. Kočvara. "On the Modelling and Solving of the Truss Design Problem with Global Stability Constraints". In: *Structural and Multidisciplinary Optimization* 23.3 (Apr. 1, 2002), pp. 189–203. ISSN: 1615-1488. DOI: 10.1007/s00158-002-0177-3. URL: https://doi.org/10.1007/s00158-002-0177-3 (visited on 11/29/2020).

[9] Arkadiusz Mazurek, William F. Baker, and Cenk Tort. "Geometrical Aspects of Optimum Truss like Structures". In: *Structural and Multidisciplinary Optimization* 43.2 (Feb. 1, 2011), pp. 231–242. ISSN: 1615-1488. DOI: 10.1007/s00158-010-0559-x. URL: https://doi.org/10.1007/s00158-010-0559-x (visited on 11/01/2020).

[10] A.G.M. Michell. "The Limits of Economy of Material in Frame-Structures". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 8.47 (Nov. 1, 1904), pp. 589–597. ISSN: 1941-5982. DOI: 10.1080/14786440409463229. URL: https://doi.org/10.1080/14786440409463229 (visited on 11/29/2020).

[11] Caitlin T. Mueller. "Computational Exploration of the Structural Design Space". Thesis. Massachusetts Institute of Technology, 2014. URL: https://dspace.mit.edu/handle/1721.1/91293 (visited on 11/29/2020).

[12] N.L. Pedersen and A.K. Nielsen. "Optimization of Practical Trusses with Constraints on Eigenfrequencies, Displacements, Stresses, and Buckling". In: *Structural and Multidisciplinary Optimization* 25.5 (Dec. 1, 2003), pp. 436–445. ISSN: 1615-1488. DOI: 10.1007/s00158-003-0294-7. URL: https://doi.org/10.1007/s00158-003-0294-7 (visited on 11/29/2020).

[13] Adam P. Piotrowski. "Review of Differential Evolution Population Size". In: *Swarm and Evolutionary Computation* 32 (Feb. 1, 2017), pp. 1–24. ISSN: 2210-6502. DOI: 10.1016/j.swevo.2016.05.003. URL: http://www.sciencedirect.com/science/article/pii/S2210650216300268 (visited on 11/26/2020).

[14] William Prager. "A Note on Discretized Michell Structures". In: *Computer Methods in Applied Mechanics and Engineering* 3.3 (May 1, 1974), pp. 349–355. ISSN: 0045-7825. DOI: 10.1016/0045-7825(74)90019-X. URL: http://www.sciencedirect.com/science/article/pii/004578257490019X (visited on 11/29/2020).

[15] G. I. N. Rozvany. "Difficulties in Truss Topology Optimization with Stress, Local Buckling and System Stability Constraints". In: *Structural optimization* 11.3 (June 1, 1996), pp. 213–217. ISSN: 1615-1488. DOI: 10.1007/BF01197036. URL: https://doi.org/10.1007/BF01197036 (visited on 11/29/2020).

[16] Richard M. Stallman and GCC DeveloperCommunity. *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. Scotts Valley, CA: CreateSpace, 2009. ISBN: 978-1-4414-1276-8.

[17] Rainer Storn and Kenneth Price. "Differential Evolution – A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces". In: *Journal of Global Optimization* 11.4 (Dec. 1, 1997), pp. 341–359. ISSN: 1573-2916. DOI: 10.1023/A:1008202821328. URL: https://doi.org/10.1023/A:1008202821328 (visited on 11/29/2020).

[18] Stephen P. Timoshenko and James M. Gere. *Theory of Elastic Stability*. 2nd ed. edition. Mineola, N.Y: Dover Publications, June 22, 2009. 560 pp. ISBN: 978-0-486-47207-2.

[19] D Wang, W. H Zhang, and J. S Jiang. "Truss Shape Optimization with Multiple Displacement Constraints". In: *Computer Methods in Applied Mechanics and Engineering* 191.33 (June 21, 2002), pp. 3597–3612. ISSN: 0045-7825. DOI: 10.1016/S0045-7825(02)00297-9. URL: http://www.sciencedirect.com/science/article/pii/S0045782502002979 (visited on 11/29/2020).

[20] M. Zhou. "Difficulties in Truss Topology Optimization with Stress and Local Buckling Constraints". In: *Structural optimization* 11.2 (Apr. 1, 1996), pp. 134–136. ISSN: 1615-1488. DOI: 10.1007/BF01376857. URL: https://doi.org/10.1007/BF01376857 (visited on 11/29/2020).

[21] M. Zhou and G. I. N. Rozvany. "DCOC: An Optimality Criteria Method for Large Systems Part I: Theory". In: *Structural optimization* 5.1 (Mar. 1, 1992), pp. 12–25. ISSN: 1615-1488. DOI: 10.1007/BF01744690. URL: https://doi.org/10.1007/BF01744690 (visited on 11/29/2020).

A Frame3DD-Opt

```
#!/usr/bin/env python3
# frame3dd_opt — a frame optimization tool to wrap Frame3DD (frame3dd.sourcefor
# This tool uses the differential evolution algorithm to optimize frame designs
# geometric constraints and stability constraints. Given a certain truss topolog
# variables to optimize, this tool can find an optimized member sizing and node
# pattern for the truss. The tool can work in either 2D or 3D.
from abc import ABC, abstractmethod
from jinja2 import Template
import numpy as np
import random
import subprocess # for running Frame3DD
import os # required for making temporary output directories for each MPI proc t
from mpi4py import MPI
import scipy.optimize

class Frame3DDOutput:
    ''' Responsible for parsing output from Frame3DD. This
    includes member lengths, and member compressions and tensions.
    Assumes you're not turning on shear effects in the frame analysis.
    '''

    # Section separators in Frame3DD output file
    element_data_header = 'F R A M E   E L E M E N T   D A T A'
    node_data_header = 'N O D E   D A T A'
```

```
    element_force_header = 'F R A M E   E L E M E N T   E N D
F O R C E S'
    reactions = 'R E A C T I O N S'

    def __init__(self, outfile_name):
        frame_element_data = []
        node_data = []
        frame_element_reactions = []

        # set some variables used for sizing frames
        self.yield_stress = None
        self.beam_geometry = None

        with open(outfile_name) as outfile:
            self.node_data = [x for x in self.read_node_section(outfile)]
            self.frame_element_data = [x for x in self.read_element_data_section
            self.frame_element_reactions = [x for x in self.read_element_forces(

    @classmethod
    def read_node_section(cls, filehandle):
        '''
        Each time this is called, this yields the next x,y,z coordinate of a nod
        This assumes that node indices have been assigned in strictly increasing
        numerical order. If you do not follow that convention, this method is in
        '''
        on_section = False
        for line in filehandle:
            if on_section:
                if line.startswith(cls.element_data_header):
                    next(filehandle)
                    return
                else:
                    split_line = line.split()
                    yield (float(split_line[1]), float(split_line[2]), float(spl
            elif line.startswith(cls.node_data_header):
                next(filehandle) # skip a line
                on_section = True
        raise Exception('Could not find nodes section in Frame3DD output file.')

    @classmethod
    def read_element_data_section(cls, filehandle):
        '''
        Reads element geometric data. This _must_ be run AFTER
        read_node_section, and will be incorrect otherwise.
        It yields the node indices and member area (in a tuple) on each call
        '''
```

```
        for line in filehandle:
            if line.startswith('  Neglect shear deformations.'):
                return
            split_line = line.split()
            yield (int(split_line[1]), int(split_line[2]), float(split_line[3]))

    @classmethod
    def read_element_forces(cls, filehandle):
        '''
        returns element forces. Negative => compression,
        positive => tension. yields:

        (member ID, force)
        '''
        on_section = False
        for line in filehandle:
            if on_section:
                if line.startswith(cls.reactions):
                    return
                else:
                    split_line = line.split()
                    force = abs(float(split_line[2][:-1]))
                    if split_line[2][-1] == 'c':
                        force *= -1
                    yield (int(split_line[0]), force)
                    next(filehandle)
            elif line.startswith(cls.element_force_header):
                next(filehandle)
                on_section = True

    def calculate_frame_volume(self):
        ''' Calculates total volume of the structure.
        This willl often be a figure of merit for an optimization.
        '''
        total_volume = 0
        for member in self.frame_element_data:
            node1 = member[0] - 1 # convert to one-based indexing
            node2 = member[1] - 1
            member_length = 0
            for i in range(3):
                member_length += (self.node_data[node1][i]-self.node_data[node2]
            member_length = member_length**0.5
            total_volume += member_length * member[2]
        return total_volume

    def member_length(self, i):
```

```
        '''
        Calculates the length of the i_th member
        '''
        member = self.frame_element_data[i]
        node1 = member[0] - 1 # convert to one-based indexing
        node2 = member[1] - 1
        member_length = 0
        for j in range(3):
            member_length += (self.node_data[node1][j]-self.node_data[node2][j])
        member_length = member_length**0.5
        return member_length

    def calculate_f_times_l(self):
        ''' Calculates the common objective sum_i |F_i| L_i, which is directly r
        to the total mass of the truss as shown in MIT 4.450, or simple algebra.
        no buckling, however.
        '''
        sum_fl = 0
        for i, member in enumerate(self.frame_element_data):
            node1 = member[0] - 1 # convert to one-based indexing
            node2 = member[1] - 1
            member_length = 0
            for j in range(3):
                member_length += (self.node_data[node1][j]-self.node_data[node2]
            member_length = member_length**0.5
            sum_fl += member_length * abs(self.frame_element_reactions[i][1])
        return sum_fl

# Here, a two possible element shapes are defined: either round tubes or square
# This could be easily be modified to include other geometries in the future.
# The role of these are to:
# 1) Store a geometric parameter independent of area, for these two, that being
#    relative size of the tube wall compared to its thickness.
# 2) Calculate torsional constant, and moments of area in the axes perpendicular
class MemberGeometricPropertyCalculator(ABC):

    def __init__(self, area):
        self.area = area

    @abstractmethod
    def get_Asy(self):
        ''' Calculates the shear area in first perpendicular axis
        '''
        raise NotImplementedError
    @abstractmethod
    def get_Asz(self):
```

```
        ''' Calculates the shear area in first perpendicular axis
        '''
        raise NotImplementedError
    @abstractmethod
    def get_Jxx(self):
        ''' Calculates the torsional geometric constant of the member
        '''
        raise NotImplementedError
    @abstractmethod
    def get_Iyy(self):
        ''' Calculates the moment of are along the first perpendicular axis
        '''
        raise NotImplementedError
    @abstractmethod
    def get_Izz(self):
        ''' Calculates the moment of are along the second perpendicular axis
        '''
        raise NotImplementedError

class SquareTube(MemberGeometricPropertyCalculator):
    ''' given the ratio of the tube thickness
    to the side length, calculate the geometric properties.
    The thickness ratio can be set to 0.5 to have a solid tube.
    '''
    def __init__(self, area, thickness_to_side_ratio):
        super().__init__(area)
        self.t_over_s = thickness_to_side_ratio
        self.b = (area / (4 * (thickness_to_side_ratio - thickness_to_side_ratio
        self.t = self.t_over_s * self.b
    def get_Asy(self):
        return self.area / (2.39573 - 0.25009*self.t_over_s - 7.89675*self.t_ove
    def get_Asz(self):
        return self.get_Asy()
    def get_Jxx(self):
        return (self.b-self.t)**3 * self.t
    def get_Iyy(self):
        return (1/12) * (self.b**4 - (self.b - 2*self.t)**4)
    def get_Izz(self):
        return self.get_Iyy()

class RoundTube(MemberGeometricPropertyCalculator):
    ''' given the ratio of the tube radii and area, calculates properties
    '''
    def __init__(self, area, ri_over_ro):
        super().__init__(area)
        self.ri_over_ro = ri_over_ro
```

```
        self.ro = np.sqrt(self.area / (np.pi * (1-self.ri_over_ro**2)))
        self.ri = self.ri_over_ro * self.ro
    def get_Asy(self):
        return self.area / (0.54414 + 2.97294 * self.ri_over_ro - 1.51899 * self
    def get_Asz(self):
        return self.get_Asy()
    def get_Jxx(self):
        return 0.5 * np.pi * (self.ro**4 - self.ri**4)
    def get_Iyy(self):
        return self.get_Jxx() / 2
    def get_Izz(self):
        return self.get_Iyy()

class UniformPerturbationSampler:
    '''
    Returns perturbations in the range [-pert_size, +pert_size], uniformly distri
    '''
    def __init__(self, pert_size):
        self.pert_size = pert_size
    def __call__(self):
        return random.uniform(-pert_size, pert_size)

# A few material classes. Units are kips/inches (yikes)
class A36Steel:
    E = 29000 # elastic modulus (ksi)
    G = 11500 # shear modulus (ksi)
    tensile_yield_stress = 36.3 # ksi (typically 0.2% elongation)
    compressive_yield_stress = 22 # ksi (typically 0.2% elongation)
    density = 2.8e-4 # ksi/in
    expansion = 0 # not touching this for now

# A few material classes. Units are kips/inches (yikes)
class DouglasFir:
    E = 12400 # elastic modulus (ksi)
    G = 4600 # shear modulus (ksi)
    tensile_yield_stress = 5 # ksi (typically 0.2% elongation)
    compressive_yield_stress = 3.94 # ksi (typically 0.2% elongation)
    density = 2.2e-5 # ksi/in
    expansion = 0 # not touching this for now

class OptimizationProblem:
    '''
    Structural optimizations problems, in this code, are defined by a few things
    Firstly, a valid, templated, input file to Frame3DD should be provided, with
    templated values filled in using Jinja2 syntax. Secondly, those variables mu
    be listed separately, along with initial guesses and rules for perturbing th
```

```
variables. Lastly, constraints on each variable must also be given. The fina
argument should be a list of 2-tuples of the length of the variable list whi
denote the allowable upper and lower bounds of each variable.

With all that information, this class can run differential evolution on the
problem, as it is defined. A few other solver settings are available.

The Jinja2 template / Frame3DD input you provide MUST include a template var
where the geometric stiffness option is toggled. This template variable shou
be named {{ geom }}, and appears as the second entry after the frame element
specifications.
'''
def __init__(self, template_file_name, connectivity_file, variable_file, saf
        mutation=0.5, recombination=0.7, consider_local_buckling=True, consi

    # Load the Jinja2 template data
    with open(template_file_name, 'r') as template_fh:
        self.template = Template(template_fh.read())

    # Load node-to-node connectivities
    self.connectivities = np.loadtxt(connectivity_file, dtype=np.int32)
    self.n_members = self.connectivities.shape[0]
    self.member_thicknesses = np.zeros(self.n_members)
    if self.connectivities.shape[1] != 2:
        raise Exception('connectivities should be two entries per line')

    # Load in the variables to optimize on
    self.variable_names = []
    self.constrained_boundaries= []
    with open(variable_file, 'r') as variables_fh:
        for line in variables_fh:
            split_line = line.split()
            if len(split_line) != 3:
                raise Exception("there should be three entries per line in t
            self.variable_names.append(split_line[0])
            assert float(split_line[1]) < float(split_line[2])
            self.constrained_boundaries.append((float(split_line[1]), float(
    self.n_variables = len(self.variable_names)

    # Save various settings
    self.rank = MPI.COMM_WORLD.Get_rank()
    self.mpi_size = MPI.COMM_WORLD.Get_size()
    self.safety_factor = safety_factor
    self.population_per_rank = population_per_rank
    self.population_size = self.population_per_rank * self.mpi_size
    self.constrain_global_buckling = consider_global_buckling
```

```
        self.maxiter = maxiter
        self.mutation = 0.5
        self.recombination = 0.7
        self.material = material
        self.consider_local_buckling = consider_local_buckling

        if not consider_local_buckling and not consider_global_buckling:
            self.evaluate_objective = self.evaluate_objective_force_times_length
        elif not consider_global_buckling:
            self.evaluate_objective = self.evaluate_objective_local_buckling
        elif consider_global_buckling:
            raise NotImplementedError('Have not added global buckling optimizati
        else:
            raise Exception('wtf??')

        # Allocate space for both previous iteration active populations
        # and last iteration active populations.
        self.last_iteration_population = np.zeros((self.population_size, self.n_
        self.current_population = np.zeros_like(self.last_iteration_population)

        # Array of cost functions for the current population (just start at larg
        self.cost_function = np.ones(self.population_size) * 1e100

        # Check that either the author or user of this code isn't insane
        if consider_global_buckling:
            raise NotImplementedError
        if not consider_local_buckling and consider_global_buckling:
            raise Exception('Cannot consider global buckling without local buckl

        # Ensure that temporary output directories are available for each proc t
        if not os.path.exists('out%i'%self.rank):
            os.mkdir('out%i'%self.rank)

        # Get path for each proc to use
        self.path = os.getenv('PATH')

    def evaluate_objective_force_times_length(self, variable_values):
        '''
        Does a linear elasticity calculation, using large member thicknesses
        in order to calculate the sums of magnitudes of member forces times
        their lengths, which is proportional to frame mass under some simple
        assumptions.
        '''
        # Do elastic frame calculation to calculate member sizes
        inputname = 'input_%i.3dd'%self.rank
        outputname = 'output_%i.out'%self.rank
```

```
        member_string = '%i\n'%self.n_members
        for member_id in range(self.connectivities.shape[0]):
            member_string += '%i %i %i 1000.0 1.0 1.0 1.0 1.0 2 %f %f 0 %f\n'%(m
                    self.connectivities[member_id, 1], self.material.E, self.ma
        with open(inputname, 'w') as fh:
            fh.write(self.template.render(members=member_string, nmodes=0, **dic

        the_env = {'PATH': self.path, 'FRAME3DD_OUTDIR': 'out%i'%self.rank}
        result = subprocess.run(['frame3dd', '-i', inputname, '-o', outputname,

        # Note: exit code 182 is given for large strains. For the linear elastic
        # so this is expected, and should not cause any difference in the soluti
        if result.returncode and result.returncode!=182:
            print("Frame3DD exited with error code %i on proc %i"%(result.returnc
            MPI.COMM_WORLD.Abort()

        # Read in results
        elastic_result = Frame3DDOutput(outputname)

        # First get an estimate on member sizes based on the yield stress
        total_frame_volume = 0
        for i in range(self.n_members):

            # Calculate area of member based on yield stress approach
            if elastic_result.frame_element_reactions[i][1] < 0:
                member_area = abs(elastic_result.frame_element_reactions[i][1])
            else:
                member_area = abs(elastic_result.frame_element_reactions[i][1])
            self.member_thicknesses[i] = member_area
            total_frame_volume += elastic_result.member_length(i) * member_area

        # Need to clean up the result, since Frame3DD just writes to make output
        removal_result = subprocess.run(['rm', outputname])
        if removal_result.returncode:
            print("Frame3DD screwed up in some undecipherable way on proc %i..."%
            MPI.COMM_WORLD.Abort()

        return total_frame_volume

    def evaluate_objective_local_buckling(self, variable_values):
        '''
        This proceeds in two steps. Firstly, a linear elasticity calculation is
        size of the members such that both local buckling, tensile yielding, and
        avoided. After that, a more accurate calculation is done which includes
        stiffness effects. If local buckling or global buckling are toggled to n
        is a little bit different.
```

```
'''
# Do elastic frame calculation to calculate member sizes
inputname = 'input_%i.3dd'%self.rank
outputname = 'output_%i.out'%self.rank
member_string = '%i\n'%self.n_members
for member_id in range(self.connectivities.shape[0]):
    member_string += '%i %i %i 1000.0 1.0 1.0 1.0 1.0 2 %f %f 0 %f\n'%(m
            self.connectivities[member_id, 1], self.material.E, self.ma
with open(inputname, 'w') as fh:
    fh.write(self.template.render(members=member_string, nmodes=0, **dic

the_env = {'PATH': self.path, 'FRAME3DD_OUTDIR': 'out%i'%self.rank}
result = subprocess.run(['frame3dd', '-i', inputname, '-o', outputname,

# Note: exit code 182 is given for large strains. For the linear elastic
# so this is expected, and should not cause any difference in the soluti
if result.returncode and result.returncode!=182:
    print("Frame3DD exited with error code %i on proc %i"%(result.returnc
    MPI.COMM_WORLD.Abort()

# Read in results
elastic_result = Frame3DDOutput(outputname)

total_frame_volume = 0

# First get an estimate on member sizes based on the yield stress
for i in range(self.n_members):

    # Calculate area of member based on yield stress approach
    if elastic_result.frame_element_reactions[i][1] > 0:
        member_area = elastic_result.frame_element_reactions[i][1] / sel

    # If member is in compression, also do a buckling sizing:
    else:
        # First, size based off compressive yield stress
        member_area = abs(elastic_result.frame_element_reactions[i][1])

        buckling_I = abs(elastic_result.frame_element_reactions[i][1]) *

        # Now, given the minimum acceptable moment of area, the member a
        # under whatever geometric constraints have been placed on it. H
        # of a certain aspect ratio for now.
        alpha = 0.9
        A = np.sqrt(4 * buckling_I * np.pi * (1-alpha**2)**2 / (1-alpha*
        if A > member_area:
            member_area = A
```

```
            self.member_thicknesses[i] = member_area
            total_frame_volume += elastic_result.member_length(i) * member_area

        # Need to clean up the result, since Frame3DD just writes to make output
        removal_result = subprocess.run(['rm', outputname])
        if removal_result.returncode:
            print("Frame3DD screwed up in some undecipherable way on proc %i..."%
            MPI.COMM_WORLD.Abort()

        return total_frame_volume

    def write_input_with_sized_members(self, variable_values):
        '''
        Using the stored member areas from a linear elasticity calculation (self
        this method writes a Frame3DD input file that uses those areas rather th
        area. This input file is then recommended for use to check whether globa
        place by turning on the geometric stiffness option and checking that the
        remains positive definite.
        '''
        # Do elastic frame calculation to calculate member sizes
        inputname = 'input_%i.3dd'%self.rank
        outputname = 'output_%i.out'%self.rank
        member_string = '%i\n'%self.n_members
        for member_id in range(self.connectivities.shape[0]):
            area = self.member_thicknesses[member_id]
            geom = RoundTube(area, 0.9)
            member_string += '%i %i %i %f %f %f %f %f %f %f %f 0 %f\n'%(member_i
                    self.connectivities[member_id, 1],area,
                    geom.get_Asy(),
                    geom.get_Asz(),
                    geom.get_Jxx(),
                    geom.get_Iyy(),
                    geom.get_Izz(),
                    self.material.E, self.material.G, self.material.density)
        with open(inputname, 'w') as fh:
            fh.write(self.template.render(members=member_string, nmodes=0, **dic

    def optimize_scipy(self):
        '''
        Carries out differential evolution using a Scipy backend.
        '''
        result = scipy.optimize.differential_evolution(self.evaluate_objective,
        print(result)

    def generate_feasible_solution(self):
```

```
        # First off, create an initial guess array of feasible solutions
        for row_i in range(self.population_per_rank):

            # Sample a feasible solution. There is a max number of attempts
            state_is_feasible = False
            max_attempts = 1000 # max number of attempts to sample a feasible so
            feasibility_attempt = 0 # iteration counter
            while not state_is_feasible:
                for j_var in range(self.n_variables):
                    self.current_population[self.rank * self.population_per_rank
                state_is_feasible = True # TODO replace with global buckling che
                self.cost_function[self.rank * self.population_per_rank + row_i]
                feasibility_attempt += 1
                if feasibility_attempt > max_attempts:
                    raise Exception("Unable to sample a feasible solution. Eithe

        self.synchronize_population_across_ranks()
        self.last_iteration_population = self.current_population


    def exclusive_sample(self, *args):
        '''
        Samples an integer in [0, population_size) which is guaranteed not equal
        of the provided arguments.
        '''
        while True:
            propose = random.randrange(0, self.population_size)
            if all([propose != a for a in args]):
                return propose

    def optimize_mine(self, print_convergence_history=True):
        '''
        Carries out differential evolution using my backend. Parallelized over M
        a fixed amount of population on each MPI rank.
        '''
        self.generate_feasible_solution()

        evolution_iteration = 0

        trial = np.zeros(self.n_variables) # work space for generating trial mut

        if print_convergence_history and self.rank == 0:
            convergence_file = open('convergence.txt', 'w')
            convergence_file.write('# Iteration  Min objective
Max objective  Mean objective  St. Dev. objective\n')
```

```
while evolution_iteration < self.maxiter:

    # Parallel loop over population (only loop over chunk of population
    for p in range(self.rank * self.population_per_rank, (self.rank+1) *

        # Original Storn and Price method:
        # Get partners
        a = self.exclusive_sample(p)
        b = self.exclusive_sample(p, a)
        c = self.exclusive_sample(p, a, b)

        # Mutate'n'mate
        j = random.randrange(0, self.n_variables)
        for k in range(self.n_variables):
            if random.random() < self.recombination or k == self.n_varia
                trial[j] = self.last_iteration_population[c][j] + self.m
                        self.last_iteration_population[a][j] - self.last_
            else:
                trial[j] = self.last_iteration_population[p][j]
            j = (j+1)%self.n_variables
        # ----------------------------------

        # # Best1Bin
        # # Get partners
        # a = np.argmin(self.cost_function)
        # b = self.exclusive_sample(p, a)
        # c = self.exclusive_sample(p, a, b)

        # # Mutate'n'mate
        # j = random.randrange(0, self.n_variables)
        # for k in range(self.n_variables):
        #     if random.random() < self.recombination or k == self.n_var
        #         trial[j] = self.last_iteration_population[a][j] + self
        #                 self.last_iteration_population[b][j] - self.la
        #     else:
        #         trial[j] = self.last_iteration_population[p][j]
        #     j = (j+1)%self.n_variables
        # ----------------------------------

        score = self.evaluate_objective(trial)
        if score <= self.cost_function[p]:
            self.current_population[p, :] = trial
            self.cost_function[p] = score
        else:
            self.current_population[p, :] = self.last_iteration_populatio
```

```
            self.synchronize_population_across_ranks()
            self.last_iteration_population = self.current_population

            if (self.rank == 0):
                print("Iteration %i: f(x) = %f" % (evolution_iteration, np.min(s
                if print_convergence_history:
                    convergence_file.write('%i %f %f %f %f\n'%(evolution_iteratio
                        np.max(self.cost_function), np.mean(self.cost_function),

            evolution_iteration += 1

        # Make it so that the winning one definitely has output printed out show
        if self.rank == 0:
            best_design = np.argmin(self.cost_function)
            self.evaluate_objective(self.current_population[best_design, :])
            self.write_input_with_sized_members(self.current_population[best_des
            if print_convergence_history:
                convergence_file.close()

    def synchronize_population_across_ranks(self):
        '''
        Synchronizes the modified populations across all ranks
        '''
        MPI.COMM_WORLD.Barrier()
        for rank in range(self.mpi_size):
            indx_start = rank * self.population_per_rank
            MPI.COMM_WORLD.Bcast(self.current_population[indx_start:indx_start+s
            MPI.COMM_WORLD.Bcast(self.cost_function[indx_start:indx_start+self.p
        MPI.COMM_WORLD.Barrier()

if __name__ == '__main__':
    import argparse
    # TODO add check that frame3dd is indeed present on the system.

    # When a new material class is added, put it in this dictionary to make it a
    material_map = {'A36Steel': A36Steel, 'DouglasFir': DouglasFir}

    cmd_parser = argparse.ArgumentParser(description='Frame3DD-Opt: Parallel dif
    cmd_parser.add_argument('frame3dd_template', help="name of the Frame3DD inpu
    cmd_parser.add_argument('connectivity_file', help="a file containing a membe
    cmd_parser.add_argument('variable_file', help="this file contains rows of va
    cmd_parser.add_argument('--no_local_buckling', dest='local_buckling', action
    cmd_parser.add_argument('--no_global_buckling', dest='global_buckling',
action='store_false', help="turn off constraining the optimization to avoid glob
    cmd_parser.add_argument('--safety_factor', type=float, default=5, help="safe
```

```
    cmd_parser.add_argument('--mutation', type=float, default=0.5, help="mutatio
    cmd_parser.add_argument('--recombination', type=float, default=0.7, help="re
    cmd_parser.add_argument('--material', default='A36Steel', help="material to
\n'.join(material_map.keys())), metavar='')
    cmd_parser.add_argument('--population_per_rank', type=int, default=15, help=
    cmd_parser.add_argument('--max_iter', type=int, default=1000, help="max DE i
    cmd_parser.set_defaults(local_buckling=True, global_buckling=True)
    args = cmd_parser.parse_args()

    opt = OptimizationProblem(args.frame3dd_template,
            args.connectivity_file,
            args.variable_file,
            consider_local_buckling=args.local_buckling,
            consider_global_buckling=args.global_buckling,
            safety_factor=args.safety_factor,
            mutation=args.mutation,
            recombination=args.recombination,
            material=material_map[args.material],
            population_per_rank=args.population_per_rank,
            maxiter=args.max_iter)

    # opt.optimize_scipy()
    opt.optimize_mine()
```