Jesse Wynn
ME EN 575 Optimization
Homework 4 Report
February 23, 2018

Computing Derivatives

Section I. Truss Optimization

For this assignment, objective function derivatives and constraint derivatives were computed and passed to MATLAB's 'fmincon' function in order to optimize the classic Ten Bar Truss problem. Three different derivative computing methods were implemented namely forward differencing, central differencing, and complex step. Findings and observations from this assignment are given in this report.

Part a. Is this a problem that could benefit from scaling?

	Forward Difference		Central Difference		Complex Step	
No Scaling	nfun = 991	eltime = 0.40	nfun = 2773	eltime = 0.61	nfun = 1849	eltime = 0.87
Scaled 1e-3	nfun = 331	eltime = 0.14	nfun = 1513	eltime = 0.43	nfun = 859	eltime = 0.40

As can be seen from the table above, this is a problem that benefits significantly from scaling. As can be seen in the case of forward difference, the number of function calls and execution time is three times less than it is for the unscaled problem. Similarly central difference and the complex step methods have significantly fewer function calls and reduced execution time after scaling is applied. The benefits of scaling were expected due to the large difference in magnitude between the values for weight and the values for stress in the truss problem. By dividing the stresses by 1000 and bringing them down closer to the order of magnitude of the weights, the optimization operates much more efficiently. Note: A step size of 1e-4 was used for derivative calculations in all three methods.

Part b. Listings of MATLAB code

Listed below are the sections of MATLAB code where derivatives were computed for each of the three methods.

Forward Difference:
```
function [f, gradf] = obj(x)
  [f, ~, ~] = objcon(x);
  % compute forward difference objective derivatives
  delta_x = 1e-4;              % set the step size
  gradf = zeros(length(x),1);       % initialize gradient vector
  % perturb each element of x and compute the partial derivatives
  for i = 1:length(x)
    x_p = x;                  % reset x_p
    x_p(i) = x_p(i) + delta_x;    % perturb x_p
    [f_p, ~, ~] = objcon(x_p);     % compute objective at perturbed x
    gradf(i) = (f_p - f)/delta_x;  % compute forward diff derivative
  end
  % myobjgrad = gradf
end
```

```matlab
function [c, ceq, DC, DCeq] = con(x)
    [~, c, ceq] = objcon(x);
    % compute forward difference constraint derivatives
    delta_x = 1e-4;                % set the step size
    DC = zeros(length(x));         % initialize gradient vector
    % perturb each element of x and compute the partial derivatives
    for i = 1:length(x)
        x_p = x;                   % reset x_p
        x_p(i) = x_p(i) + delta_x;     % perturb x_p
        [~, c_p, ~] = objcon(x_p);     % compute constraint at perturbed x
        DC(i,:) = (c_p - c)/delta_x;   % compute forward diff derivative
        DCeq = [];
    end
    % mycongrad = DC
end
```

For the forward difference method, I simply integrated the calculation of derivatives into the obj() and con() functions that we pass to fmincon(). This approach is nice because we only have to compute the unperturbed objective value once and we can use it for both the objective function value that we pass to fmincon, as well as for the computation of the derivative. The only difficulty encountered with this method was that the constraint derivatives initially would not pass the fmincon 'CheckGradients' routine—regardless of how the step size was chosen. However after scaling was applied, and the step size for the constraint derivatives was set to 1e-6, the gradients passed without any issue. The forward difference method computes the derivative by evaluating the function at a small step forward from the current point, and then computing the slope between the current point and the forward point. This method works well enough if the step size is chosen appropriately. If the step size is too large, truncation error will creep in, and if the step size is too small, roundoff error due to subtractive cancellation will cause problems. For this method, the expected truncation error is proportional to the step size—i.e., expected error is on the order of delta_x.

Central Difference:
```matlab
function [f, gradf] = obj(x)
    [f, ~, ~] = objcon(x);
    % compute central difference objective derivatives
    delta_x = 1e-4;                % set the step size
    gradf = zeros(length(x),1);    % initialize gradient vector
    % perturb each element of x and compute the partial derivatives
    for i = 1:length(x)
        x_p = x;                       % reset x_p
        x_m = x;
        x_p(i) = x_p(i) + delta_x;         % perturb x_p
        x_m(i) = x_m(i) - delta_x;
        [f_p, ~, ~] = objcon(x_p);         % compute objective at perturbed x
        [f_m, ~, ~] = objcon(x_m);         % compute objective at perturbed x
        gradf(i) = (f_p - f_m)/(2*delta_x);  % compute central diff derivative
    end
end

function [c, ceq, DC, DCeq] = con(x)
    [~, c, ceq] = objcon(x);
    % compute central difference constraint derivatives
    delta_x = 1e-4;                % set the step size
    DC = zeros(length(x));         % initialize gradient vector
    % perturb each element of x and compute the partial derivatives
```

```matlab
    for i = 1:length(x)
        x_p = x;                      % reset x_p
        x_m = x;
        x_p(i) = x_p(i) + delta_x;         % perturb x_p
        x_m(i) = x_m(i) - delta_x;
        [~, c_p, ~] = objcon(x_p);         % compute constraint at perturbed x
        [~, c_m, ~] = objcon(x_m);          % compute constraint at perturbed x
        DC(i,:) = (c_p - c_m)/(2*delta_x);   % compute forward diff derivative
        DCeq = [];
    end
end
```

Derivatives for the central difference method were incorporated into the code the same way as they were for forward difference. The only major difference here is that the objective function must be called within the obj() and con() functions three times instead of twice to allow for perturbing the function forward a step and backwards a step. The central difference method computes the derivative by evaluating the function at a small step backwards **and** forwards of the current point, and then computing the slope between the backward point and the forward point. No problems were encountered for this method and 'CheckGradients' passed without any problems. ==Central differencing is more accurate than forward differencing and the expected error is on the order of delta_x^2.== The main drawback to this method is that it is more computationally heavy since additional function evaluations are required.

Complex Step:
```matlab
  function [f, gradf] = obj(x)
    [f, ~, ~] = objcon(x);

    % compute the partial objective derivatives via complex step

    delta_x = 1e-4;               % set the step size
    gradf = zeros(length(x),1);       % initialize gradient vector
    for it = 1:length(x)
        x_p = x;
        x_p(it) = x_p(it) + delta_x*i;
        [f_p, ~, ~] = objcon(x_p);
        gradf(it) = imag(f_p)/delta_x;
    end
  end
  function [c, ceq, DC, DCeq] = con(x)
    [~, c, ceq] = objcon(x);

    % compute the partial constraint derivatives via complex step

    delta_x = 1e-4;               % set the step size
    DC = zeros(length(c),length(x));       % initialize gradient vector
    for it = 1:length(x)
        x_p = x;
        x_p(it) = x_p(it) + delta_x*i;
        [~, c_p, ~] = objcon(x_p);
        DC(:, it) = imag(c_p)/delta_x;
    end
    DC = DC';
    DCeq = [];
  end
```

Once again, derivatives were incorporated into the code the same way as the previous two methods. The only challenge encountered was changing how the constraint functions were defined within the objcon() function. Since the absolute value function isn't defined for complex numbers, constraints were split into 20 separate constraints instead of 10. For this method, derivatives are computed by perturbing the function by a complex delta_x as is shown in Eq. 4.18 from the notes. This method is really slick and cleverly avoids the ill effects of subtractive cancellation. Essentially you can choose a delta_x for this method to be as small as you like, and still compute exceptionally accurate derivatives. Similar to the central difference method, expected error is on the order of delta_x^2 but now we can make delta_x to be very very small.

Part c. Error Evaluation

Since weight scales linearly in this problem, accuracy of derivatives isn't dependent on step size and so all three methods produced equally accurate derivatives for weight. The constraint derivatives related to stresses however are sensitive to step size and so accuracy between the three methods could be evaluated. Since the complex step method allows us to pick a very very small step size (delta_x), for this evaluation, a step size of 1e-20 was selected for the complex step, and the constraint derivatives for this method were treated as the 'true' derivatives for comparison. For all three methods, the constraint derivatives from the starting point were collected and then compared to the 'true' derivative generated by the complex step approach. For the forward difference and central difference methods a step size of 1e-4 was chosen. This step size was chosen because it gave sufficiently small error in the derivatives and could be used to optimize the truss problem without issue.

To evaluate the errors, I simply took the norm of the difference between the complex step constraint derivatives and the forward/central difference constraint derivatives. See code snippet below.

error_fd = norm(complexDC.DC - fdDC.DC)

error_cd = norm(complexDC.DC - cdDC.DC)

error_complex = norm(complexDC.DC – complexDC.DC)

Result:

error_fd =

   2.0060e-04

error_cd =

   3.9676e-09

error_complex =

   0

As expected and as can be seen, the forward difference method has the most error and is on the order of delta_x. The central difference method has significantly less error even though the same step size was

used and its error is approximately on the order of delta_x^2. Error in the complex step here is assumed zero. Each of the three methods has its merits. Forward difference is fast (computationally) and calculation of derivatives is the most straightforward. Central differencing takes approximately twice as much computation as forward differencing but for the same step size in this case has error that is five orders of magnitude smaller. That's pretty significant! The complex step method is the most accurate but is not a 'blackbox' method like the other two are and requires that the objective function can handle complex values. Additionally, it costs more computationally to perform operations on complex variables. Depending on the nature of the function, and the constraints on computation, there are trade offs of speed vs accuracy.

Part d. Table

	Forward Difference	Central Difference	Complex Step
Number of Function Calls	397	1639	859
Time of Execution	0.20	0.35	0.37
Stopping Criterion	* below	* below	* below
Final Objective Value	1.5932e+03	1.5932e+03	1.5932e+03

Note: Data in the above table is *after* appropriate scaling was applied and time of execution represents the average of three trial runs.

* The relative first-order optimality measure, 1.295203e-08,
  is less than options.OptimalityTolerance = 1.000000e-06, and the relative maximum constraint
  violation, 0.000000e+00, is less than options.ConstraintTolerance = 1.000000e-06.

Discussion
The biggest difference that is observed in the data above is the number of function calls, and the time of execution. As expected, forward difference requires the fewest function evaluations and as a result, requires the least amount of time to execute. Central difference requires the most function evaluations and the time of execution is increased. Complex step takes more evaluations than forward difference, and less than complex step but takes the most time to execute. This increased execution time results from the extra computational resources that are required to evaluate complex variables.

Section II. Automatic Differentiation for the Spring Problem

Part a. Explanation

The main mathematical mechanism of the Automatic Differentiation method (AD) is the chain rule for differentiation. In this method, function values are computed one component at a time such that each line in the program is composed of the values computed on previous lines. In other words, function derivatives are automatically computed as part of computing the actual function value. Then using object oriented programming, and function overloading, the function values and function derivatives can be computed simultaneously.

Part b. MATLAB code applying valder class to the Spring Optimization Problem

```
clc
clear
close all

x = [0.015, 0.5, 10.0, 1.5];

[function_values, function_jacobians, constraint_values, constraint_jacobians] = objcon(x);

function_values
function_jacobians

constraint_values
constraint_jacobians


% ------------Objective and Non-linear Constraints------------
function [function_vals, function_jacobians, constraint_vals, constraint_jacobians] = objcon(x)

% set objective/constraints here

% make design variables valder objects (things we'll adjust to find optimum)
d = valder(x(1),[1,0,0,0]);  % wire dia (in)
D = valder(x(2),[0,1,0,0]);  % coil dia (in)
n = valder(x(3),[0,0,1,0]);  % num coils
hf = valder(x(4),[0,0,0,1]); % free height (no load) (in)

% other analysis variables (constants that the optimization won't touch)
h0 = 1.0;         % preloaded height (in)
delta0 = 0.4;       % deflection (in)
hdef = h0 - delta0; % deflected spring height (in)
G = 12e6;
Q = 150e3;
w = 0.18;
Se = 45e3;
Sf = 1.5;

% delta_x = 0.4;  % not sure if this is right??
delta_x = (hf - h0);  % maybe this insteadc = zeros(9,1);???

% analysis functions
k = G*d^4/(8*D^3*n);
F = k*delta_x;
```

```matlab
K = ((4*D-d)/(4*(D-d)))+0.62*(d/D);
% Tau = (8*F*D/pi*d^3)*K;
hs = n*d;
F_min = k*(hf - h0);
% F_max = F_min + delta0*k;
F_max = k*(hf - (h0 - delta0));
F_hs = k*(hf - hs);
Tau_min = 8*F_min*D*K/(pi*(d^3));
Tau_max = 8*F_max*D*K/(pi*(d^3));
Tau_m = (Tau_max + Tau_min)/2;
Tau_a = (Tau_max - Tau_min)/2;
Tau_hs = 8*F_hs*D*K/(pi*(d^3));
Sy = 0.44*(Q/d^w);

function_vals = [k.val, F.val, K.val, hs.val, F_min.val, F_max.val, ...
                F_hs.val, Tau_min.val, Tau_max.val, Tau_m.val, Tau_a.val, ...
                Tau_hs.val, Sy.val];
function_jacobians = [k.der; F.der; K.der; hs.der; F_min.der; F_max.der; ...
                 F_hs.der; Tau_min.der; Tau_max.der; Tau_m.der; Tau_a.der; ...
                 Tau_hs.der; Sy.der];

% objective function (what we're trying to optimize)
% f = -F;  % maximize Force

% inequality constraints (c<=0)
c1 = Tau_hs - Sy;
c2 = Tau_a - Se/Sf;
c3 = Tau_a + Tau_m - Sy/Sf;
c4 = (D/d) - 16;
c5 = -(D/d) + 4;
c6 = d - 0.2;
c7 = -d + 0.01;
c8 = D + d - 0.75;
c9 = -hdef + hs + 0.05;

constraint_vals = [c1.val, c2.val, c3.val, c4.val, c5.val,...
                c6.val, c7.val, c8.val, c9.val];

constraint_jacobians = [c1.der; c2.der; c3.der; c4.der; c5.der;...
                  c6.der; c7.der; c8.der; c9.der];

end
```

Result:

```
function_values =

   1.0e+05 *

  0.000000607500000
  0.000000303750000
  0.000010417958763
  0.000001500000000
  0.000000303750000
  0.000000546750000
  0.000000820125000
  0.119381013650947
```

0.214885824571704
0.167133419111326
0.047752405460379
0.322328736857557
1.405553914176020

function_jacobians =

  1.0e+06 *

0.000016200000000	-0.000000364500000	-0.000000006075000	0
0.000008100000000	-0.000000182250000	-0.000000003037500	0.000000060750000
0.000002834218302	-0.000000085026549	0	0
0.000010000000000	0	0.000000015000000	0
0.000008100000000	-0.000000182250000	-0.000000003037500	0.000000060750000
0.000014580000000	-0.000000328050000	-0.000000005467500	0.000000060750000
0.000021262500000	-0.000000492075000	-0.000000009112500	0.000000060750000
0.828351173720028	-0.048726737941790	-0.001193810136509	0.023876202730189
1.491032112696050	-0.087708128295222	-0.002148858245717	0.023876202730189
1.159691643208040	-0.068217433118506	-0.001671334191113	0.023876202730189
0.331340469488011	-0.019490695176716	-0.000477524054604	0
1.997786141742184	-0.131562192442834	-0.003581430409528	0.023876202730189
-1.686664697011224	0	0	0

constraint_values =

  1.0e+05 *

-1.083225177318463
-0.252247594539621
-0.722150118212309
 0.000173333333333
-0.000293333333333
-0.000001850000000
-0.000000050000000
-0.000002350000000
-0.000004000000000

constraint_jacobians =

  1.0e+06 *

3.684450838753409	-0.131562192442834	-0.003581430409528	0.023876202730189
0.331340469488011	-0.019490695176716	-0.000477524054604	0
2.615475244036867	-0.087708128295222	-0.002148858245717	0.023876202730189
-0.002222222222222	0.000066666666667	0	0
0.002222222222222	-0.000066666666667	0	0
0.000001000000000	0	0	0
-0.000001000000000	0	0	0
0.000001000000000	0.000001000000000	0	0
0.000010000000000	0	0.000000015000000	0

Part c. How AD differs from other numerical methods

One way that AD differs from other methods in that it requires function variables to be 'objects' (in the sense of object oriented programming) that can be overloaded to handle the computation of both values and derivatives simultaneously. AD is also different than traditional numerical derivative computation in that there is no approximation error. This arises from the fact that we're not perturbing the function to get approximate derivatives but instead we are composing function values by computing its derivatives along the way. Finally similar to the complex step, AD is not a blackbox method and requires that the analysis software be modified and executed in an environment that supports AD (such as MATLAB using the valder class). Pros of AD include simultaneous values and derivatives, and no approximation error. The primary con is that its not supported in all environments, an it requires more computational resources.

Note: The description of the assignment is subject to change slightly. However, I don't anticipate changes to the requested tasks. Indeed, I may add something on Automatic Differentiation. I am posting this now to provide it for this who wish to get started early.

Overview
For this assignment you will optimize a 10-bar truss problem where you will provide derivatives to the optimizer. Our goals are to learn several approaches for numerically computing derivatives, understand their advantages and disadvantages, and apply these methods to a problem to better understand the need for accurate gradients. You will use software already developed for truss analysis.

Truss Specification
Everyone will optimize the same 10-bar truss. This is a problem taken from the literature that is well known. The description of the problem (along with the solution) is given in the Appendix. The truss analysis software (***Truss.m***) reads in a data file (***Data.m***) that describes the particular problem. These files will be provided to you.

Truss Derivatives
Before we can optimize the truss we need to provide the following derivatives:
1) Derivatives of weight (objective) with respect to cross-sectional areas (design variables), i.e.

$$\frac{\partial w}{\partial a_i} \quad i = 1, 2, \ldots n$$

2) Derivatives of stress (constraints) with respect to cross-sectional areas, i.e.,

$$\frac{\partial \sigma_j}{\partial a_i} \quad i = 1, 2, \ldots n; j = 1, 2, \ldots m$$

There are *m* constraints (*m*=10), and *n* variables (*n*=10) so this will be a 10 by 10 matrix.

The truss analysis function is already provided that returns weight and stress for given cross-sectional areas (***Truss.m***). There is also a driver program to call this routine as a stand-alone program (***StandAloneTruss.m***), a data file which describes the 10-bar truss (***Data.m***), and a file (***OptimizeTruss.m***) that is already setup to optimize the truss. All files are available under Content > MATLAB Examples on Learning Suite. Executing ***OptimizeTruss.m*** will optimize the truss using ***fmincon*** and built-in derivatives. You will modify this file to add your own derivatives to the code.

You should compute the derivatives of the objective (weight) and the constraints (stress) using the following methods:

(a) A forward difference approach
(b) A central difference approach
(c) The complex-step approach

Truss Optimization

After you have developed the gradients, optimize the truss using **fmincon**. Previously we have used the default solver, which is the Interior Point method. The **fmincon** optimizer has three other solvers, and you might want to experiment to find which works best on this problem. Solve this optimization problem, from the same starting point, using all the derivative methods.

Report on the following:
a) Is this a problem that could benefit from some simple scaling? Report on what you learn here.
b) Provide a listing giving your MATLAB code for each of the methods (just the code that shows the method and how implemented) with comments. Provide a brief explanation of how you integrated derivatives into the code and discuss any difficulties encountered. Indicate for each method how it is calculated and what the expected error is.
c) Evaluate the errors of the derivatives of the various methods at the starting point. Are these in line with expectations? How did you decide on a good perturbation? What are the relative merits of the approaches?
d) Provide a table giving,
  - The number of function calls and iterations required by each derivative method to reach the optimum. You should keep track of function calls using the gobal variable **"nfun"**
  - If possible, report the time of execution. (I tried the "tic/toc" function of MATLAB but wasn't sure if this was giving accurate results. If you use this function, run it several times and average.)
  - The stopping criterion given by MATLAB, i.e. why did it terminate, and the final objective value. Discuss any differences you observe.

Automatic Differentiation

Apply Automatic differentiation (AD) to the Spring design problem we did for Homework 1. (Note: you do not need to optimize it—just evaluate the functions once and show how you can get function values and derivatives simultaneously). Follow the example of the Two-bar truss, using the MATLAB AD approach given in the notes. You can copy the necessary AD functions (file **valder.m**) from Content > MATLAB on Learning Suite.

Include the following:
a) Explain briefly how this method works.
b) Include your MATLAB code
c) Discuss how AD differs from other numerical methods.

Notes/Suggestions/Things to Watch Out For
  - I would start with integrating derivatives into the stand-alone program and then move from there into integrating into the optimization.

- I would first provide derivatives for the objective function but let MATLAB do the constraints. Then move on to the constraints.
- Note that "i" cannot be used as an imaginary number and as a loop counter or array index in the same function
- I would spend some time reviewing how MATLAB handles the cases when you supply your own derivatives. In this regard, I have copied some relevant parts of documentation in the attached sheet.
- There is an option in **fmincon** to run a check on your gradients (see **CheckGradients**). This allows you to not only check the accuracy of your derivatives but to make sure they are in the correct order.

  Note that for this I set the finite difference type to central (in the code below) so we have a more accurate derivative as a check on our derivatives. Otherwise we have a relatively inaccurate method (forward difference) as the standard against which our derivatives are checked.

  Options:
  'SpecifyObjectiveGradient',true,'SpecifyConstraintGradient', true,...
  'FiniteDifferenceType','central','CheckGradients',true)

- Absolute value of constraints issue
  I took the absolute value of the stresses as a way to check that both compressive (negative) and tensile (positive) stresses were less than the allowed value, and the optimizer took these just fine (see code in **OptimizeTruss.m**). That is, the constraints have both upper and lower allowable values:

$$-25000 \leq Stress \leq 25000$$

However, the absolute value function is not defined for complex variables, and this caused my derivatives to come back as zero. So you will need to change how these constraints are handled, at least for that method.

I could have changed this in the original function, but chose not to so you could see the issues that come up.

Step 1: Write a file for the objective function and gradient.

```
function [f,gradf] = objfungrad(x)
f = exp(x(1))*(4*x(1)^2+2*x(2)^2+4*x(1)*x(2)+2*x(2)+1);
% Gradient of the objective function:
if nargout  > 1
    gradf = [ f + exp(x(1)) * (8*x(1) + 4*x(2)),
    exp(x(1))*(4*x(1)+4*x(2)+2)];
end
```

Step 2: Write a file for the nonlinear constraints and the gradients of the nonlinear constraints.

```
function [c,ceq,DC,DCeq] = confungrad(x)
c(1) = 1.5 + x(1) * x(2) - x(1) - x(2); % Inequality constraints
c(2) = -x(1) * x(2)-10;
% No nonlinear equality constraints
ceq=[];
% Gradient of the constraints:
if nargout > 2
    DC= [x(2)-1, -x(2);
        x(1)-1, -x(1)];
    DCeq = [];
end
```

`gradf` contains the partial derivatives of the objective function, `f`, returned by `objfungrad(x)`, with respect to each of the elements in `x`:

$$\nabla f = \begin{bmatrix} e^{x_1}\left(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1\right) + e^{x_1}\left(8x_1 + 4x_2\right) \\ e^{x_1}\left(4x_1 + 4x_2 + 2\right) \end{bmatrix}.$$

(6-58)

The columns of `DC` contain the partial derivatives for each respective constraint (i.e., the `ith` column of `DC` is the partial derivative of the `ith` constraint with respect to `x`). So in the above example, `DC` is

$$\begin{bmatrix} \dfrac{\partial c_1}{\partial x_1} & \dfrac{\partial c_2}{\partial x_1} \\ \dfrac{\partial c_1}{\partial x_2} & \dfrac{\partial c_2}{\partial x_2} \end{bmatrix} = \begin{bmatrix} x_2 - 1 & -x_2 \\ x_1 - 1 & -x_1 \end{bmatrix}.$$

(6-59)

Since you are providing the gradient of the objective in `objfungrad.m` and the gradient of the constraints in `confungrad.m`, you *must* tell `fmincon` that these files contain this additional information. Use `optimoptions` to turn the options `SpecifyObjectiveGradient` and `SpecifyConstraintGradient` to `true` in the example's existing options:

```
options = optimoptions(options,'SpecifyObjectiveGradient',true,'SpecifyConstraintGrad
```

If you do not set these options to `'on'`, `fmincon` does not use the analytic gradients.

The arguments `lb` and `ub` place lower and upper bounds on the independent variables in `x`. In this example, there are no bound constraints, so set both to `[]`.

Step 3: Invoke the constrained optimization routine.

```
x0 = [-1,1];             % Starting guess
options = optimoptions(@fmincon,'Algorithm','sqp');
options = optimoptions(options,'SpecifyObjectiveGradient',true,'SpecifyConstraintGrad
lb = [ ]; ub = [ ];   % No upper or lower bounds
[x,fval] = fmincon(@objfungrad,x0,[],[],[],[],lb,ub,...
    @confungrad,options);
```

4

Appendix
Listing of Data.m:

```matlab
%Ten bar truss of Venkaya
E = 1.e7;
ndof = 12;
nbc = 4;
nnode = 6;
nelem = 10;

% format is coordinate, then degree of freedom numbers
Node = [720 360 1 2; 720 0 3 4; 360 360 5 6; 360 0 7 8;...
    0 360 9 10; 0 0 11 12];

% format is for each degree of freedom (horizontal then
vertical)
force = [0; 0; 0; -100000; 0; 0; 0; -100000; 0; 0; 0; 0];

% put boundary conditions in reverse order, highest to lowest
bc = [12, 11, 10, 9];

% format is start node to end node, area
Elem = [5 3 5; 3 1 5; 6 4 5; 4 2 5; 3 4 5; 1 2 5; 5 4 5; ...
    6 3 5; 3 2 5.; 4 1 5];
```

Listing of OptimizeTruss.m:

```matlab
 % ------------Starting point and bounds------------
    %design variables
    x0 = [5, 5, 5, 5, 5, 5, 5, 5, 5, 5]; %starting point (all
areas = 5 in^2)
    lb = [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1];
%lower bound
    ub = [20, 20, 20, 20, 20, 20, 20, 20, 20, 20]; %upper
bound
    global nfun;
    nfun = 0;

    % -----------Linear constraints-----------
    A = [];
    b = [];
    Aeq = [];
    beq = [];

    % -----------Call fmincon-----------
```

```matlab
    tic;
    options = optimoptions(@fmincon,'display','iter-
detailed','Diagnostics','on');
    [xopt, fopt, exitflag, output] = fmincon(@obj, x0, A, b,
Aeq, beq, lb, ub, @con, options);
    eltime = toc;
    eltime
    xopt    %design variables at the minimum
    fopt    %objective function value at the minumum
    [f, c, ceq] = objcon(xopt);
    c
    nfun


    % ------------Objective and Non-linear
Constraints-----------
    function [f, c, ceq] = objcon(x)
        global nfun;

        %get data for truss from Data.m file
        Data

        % insert areas (design variables) into correct matrix
        for i=1:nelem
            Elem(i,3) = x(i);
        end

        % call Truss to get weight and stresses
        [weight,stress] = Truss(ndof, nbc, nelem, E, dens,
Node, force, bc, Elem);

        %objective function
        f = weight; %minimize weight

        %inequality constraints (c<=0)
        c = zeros(10,1);            % create column vector
        for i=1:10
            c(i) = abs(stress(i))-25000; % check stress both
pos and neg
        end

        %equality constraints (ceq=0)
        ceq = [];
        nfun = nfun + 1;

    end
```

```matlab
% ------------Separate obj/con (do not change)------------
function [f] = obj(x)
    [f, ~, ~] = objcon(x);
end
function [c, ceq] = con(x)
    [~, c, ceq] = objcon(x);
end
```

Ten Bar Truss Sizing Optimization

The ten bar truss model was developed by Venkayya for:
Venkayya, V. B. (1971). "Design of Optimum Structures," Computers & Structures, 1(1), 265-309.

The design problems for this page are described in:
Haftka, Raphael T. Elements of Structural Optimization. Springer, 1992. Pages 238, 244.

Example CoFE input files for this page are provided here:
NASTRAN_CoFE/CoFE_examples/o1_tenbarOpt/

L = 360"
P = 100 Kips
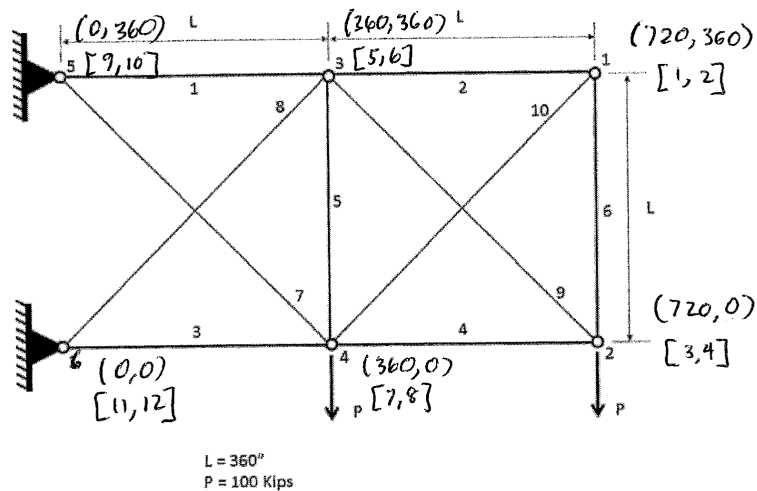
Details

- Young's modulus: 10^7 psi
- Specific mass: 0.1 lbm/in^3
- Minimum area: 0.1 in^2
- Allowable stress: +/- 25000 psi
- The ten design variables are the areas of the ten members.

Design Problem 1

Design for minimum weight with stress constraints.

Design Variable	Design from CoFE + SQP	Design in Haftka
x_1	7.94	7.94
x_2	0.1	0.1
x_3	8.06	8.06
x_4	3.94	3.94
x_5	0.1	0.1
x_6	0.1	0.1
x_7	5.74	5.74
x_8	5.57	5.57
x_9	5.57	5.57
x_{10}	0.1	0.1

Objective Function	Design from CoFE + SQP	Design in Haftka
Weight	1593.2 lb	1593.2 lb