

Optimization Homework #4

Ryan Day

February 23, 2019

1 Truss Optimization

1.1 Scaling

Scaling the constraints is useful. I scaled it by dividing the constraints by 10^2 . This reduced the number of function calls by more than half when I implemented it. I tried to make the constraints the same order of magnitude as the design functions. The design variables are all around the same order of magnitude, so I don't see the use of scaling there.

1.2 Matlab Code implementation

The matlab code is included in appendix A. I used a function that took in x , then perturbed the function with a step depending on a type input (forward, central, or complex), and then calculated the gradient and constraints gradient. It was not too hard to implement this function. I simply added it to the end of obj and con in order to get the derivatives. However, I had to make one change in order to get the complex function to work. This change was changing the inequality constraint from using the abs function to taking the square root of the value squared. This did the same thing as abs but could be used with complex variables.

1.3 Expected Errors of the derivatives

I expected the errors of the derivative to be greatest for the forward, then central, then complex. The merits of the forward method is it only takes one function call per derivative. Central method is slightly more accurate, but takes twice as many function calls. Both central and forward methods have subtractive error which makes it so you can't have the step size be too small. The complex step method does not have this subtractive error which makes it so you can have an extremely small step size, but then you have to make sure your function can handle complex numbers. In addition to this, the computations of the complex step can take longer than the forward step because of the included complex numbers.

I figured out the optimal perturbation of forward and central methods by comparing it to the complex step derivative. I checked compared the estimated derivatives from the first iteration between the complex step with a step size of 10^{-30} . I knew the complex derivative would be pretty accurate because it has no subtractive error, so you can make the step size extremely small. 10^{-8} turned out to be the ideal step size for forward and central. It had an error for forward on the order of 10^{-6} which was fine. When I tried at $1e-9$ there was an error in the gradient of forward of 10^{-4} which was too much. For some reason the central wasn't converging when I scaled it, so that is why it was so much slower than the other functions.

1.4 Table and stopping criteria

	# Function calls	# Iterations	Avg Time execution	Final Objective value
No Derivatives supplied	287	12	0.420	1.5932e+03
Forward method	309	12	0.415	1.5932e+03
Central method	4621	5	1.114	1.5932e+03
Complex method	287	12	0.510	1.5932e+03

The execution time was fastest by just a little bit with the forward method. It barely beat out the fmincon with no supplied derivatives. The forward method and complex method had the same number of function calls as expected since they both call the objective function just once to calculate derivatives. The complex method took longer than the forward method because it had to deal with complex numbers. The central method took about twice as many function calls and so took a lot longer than any other method. However, it had about half the iterations, because it goes in a more accurate direction with .

Stopping Criterion for no derivatives supplied:

Optimization completed: The relative first-order optimality measure, 5.312214e-07, 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.

Stopping Criterion for other methods:

Forward:

Optimization stopped because the relative changes in all elements of x are less than options.StepTolerance = 1.000000e-10, and the relative maximum constraint violation, 0.000000e+00, is less than options.ConstraintTolerance = 1.000000e-06.

Central:

Optimization completed: The relative first-order optimality measure, 7.486241e-07, 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.

Complex:

Optimization completed: The relative first-order optimality measure, 3.928371e-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.

The stopping criterion for forward and without supplying derivatives was that the x inputs had very small changes, signalling that there was an optimum. For the complex and central step, the function output met the optimality measure. Because I only looked at 5 significant digits for the optimum, these measures didn't affect the optimum at all.

2 Automatic Differentiation

2.1 How does this method work?

I define the values to be of the valder class. Any other function that has a valder variable as part of its calculation will then take the valder variable into account and then operator overloading is used to compute derivatives for the original valder functions. In this way you can get derivatives without having to change your code much.

2.2 How is AD different from other numerical methods?

AD is different because it solves the derivatives analytically using lookup functions and operator overloading instead of numerically approximating the value.

2.3 Matlab Code

AD code is found in appendix B

A Truss Optimization

```

1      % -----Starting point and bounds-----
2      %design variables
3      x0 = [5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5]; %starting point (all areas = 5 in^2)
4      lb = [0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1]; %lower bound
5      ub = [20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20]; %upper bound
6      global nfun;
7      nfun = 0;
8

```

```

9
10 % -----Linear constraints-----
11 A = [];
12 b = [];
13 Aeq = [];
14 beq = [];
15
16 % -----Call fmincon-----
17
18 options = optimoptions(@fmincon,'display','iter-detailed','Diagnostics','on',...
19 'SpecifyObjectiveGradient',true,'SpecifyConstraintGradient',true);
20 [xopt, fopt, exitflag, output] = fmincon(@obj, x0, A, b, Aeq, beq, lb, ub, @con,
    options);
21
22 xopt    %design variables at the minimum
23 fopt    %objective function value at the minumum
24 [f, c, ceq] = objcon(xopt);
25 c
26 nfun
27
28 % -----Objective and Non-linear Constraints-----
29 function [f, c, ceq] = objcon(x)
30     global nfun;
31
32     %get data for truss from Data.m file
33     Data;
34
35     % insert areas (design variables) into correct matrix
36     for i=1:nelem
37         Elem(i,3) = x(i);
38     end
39
40     % call Truss to get weight and stresses
41     [weight, stress] = Truss(ndof, nbc, nelem, E, dens, Node, force, bc, Elem);
42
43     %objective function
44     f = weight; %minimize weight
45
46     %inequality constraints (c<=0)
47     c = zeros(10,1); % create column vector
48     for i=1:10
49         c(i) = (sqrt((stress(i))^2)-25000)/10^2; % check stress both pos and neg
50     end
51
52     %equality constraints (ceq=0)
53     ceq = [];
54     nfun = nfun + 1;
55 end
56
57 % -----Separate obj/con You may wish to change-----
58 function [f, grad] = obj(x)
59     [f, c, ~] = objcon(x);
60     [grad,~] = findGrad(x,f,c);
61 end
62 function [c, ceq, cgrad, ceqgrad] = con(x)
63     [f, c, ceq] = objcon(x);
64     [~, cgrad] = findGrad(x,f,c);
65     ceqgrad = ceq;
66 end
67
68 function [grad, cgrad] = findGrad(x,fo,co)
69     % Define method of numerical differentiation
70     type = "central"; % "forward", "central", or "complex"
71     % Define step size
72     h = 1e-8;
73
74     [~,sizeX] = size(x);
75     grad = zeros(sizeX,1);
76     [nc,~] = size(co);
77     cgrad = zeros(nc,nc);

```

```

78     for index=1:size(x)
79         if (type=="forward" || type=="central")
80             xf = x;
81             xf(index) = x(index) + h;
82             [f_f,c_f,~] = objcon(xf);
83             if(type=="forward")
84                 grad(index) = (f_f-fo)/h;
85                 for j = 1:nc
86                     cgrad(index,j) = (c_f(j)-co(j))/h;
87                 end
88             else
89                 xb = x;
90                 xb(index) = x(index) - h;
91                 [f_b,c_b,~] = objcon(xb);
92                 grad(index) = (f_f-f_b)/(2*h);
93                 for j = 1:nc
94                     cgrad(index,j) = (c_f(j)-c_b(j))/h;
95                 end
96             end
97         else
98             xI = x;
99             xI(index) = x(index)+1j*h;
100            [f_im,c_im,~] = objcon(xI);
101            grad(index) = imag(f_im)/h;
102            for k= 1:nc
103                cgrad(index,k) = imag(c_im(k))/h;
104            end
105        end
106    end
107 end

```

B Automatic differentiation

```

1  % -----Starting point and bounds-----
2  %var= d D n hf %design variables
3  x0 = [0.07, 0.67 7.6 1.4]; %starting point
4  [Values, Jacobians] = getValues(x0)
5
6  function [Values, Jacobians] = getValues(x)
7
8      %design variables
9      d = x(1); % height (in)
10     D = x(2); % diameter (in)
11     n = x(3); % number of coils (treating as continuous for this example)
12     hf = x(4); % free height (in)
13     d = valder(d,[1,0,0,0]);
14     D = valder(D,[0,1,0,0]);
15     n = valder(n,[0,0,1,0]);
16     hf = valder(hf,[0,0,0,1]);
17
18     % Constants
19     G = 12e6; % psi
20     Se = 45000; % psi
21     w = 0.18;
22     Sf = 1.5;
23     Q= 150000; % psi
24
25     % Analysis variables
26     h0 = 1.0; % preload height
27     delta0 = 0.4;
28
29     % Output variables
30     hdef = h0-delta0;
31     k = (G*d^4)/(8*D^3*n);
32     K = (4*D-d)/(4*(D-d))+0.62*d/D;
33     F_h0 = k*(hf-h0); %Fmin
34     F_hdef = k*(hf-hdef); %Fmax
35     tauh0 = (8*F_h0*D)/(pi*d^3)*K; %taumin
36     tauhdef = (8*F_hdef*D)/(pi*d^3)*K; %taumax

```

```

37
38     taumean = (tauhdef+tauh0)/2;
39     tauavg = (tauhdef-tauh0)/2;
40
41     hs = n*d;
42     Fhs = k*(hf-hs);
43     tauhs = (8*Fhs*D)/(pi*d^3)*K;
44
45     Sy = 0.44*Q/(d^w);
46
47
48     %objective function
49     f = F_h0;
50     Values = [k.val, K.val, F_h0.val, F_hdef.val, tauh0.val, tauhdef.val, taumean.val,
51              tauavg.val, hs.val, Fhs.val, tauhs.val, Sy.val];
52     Jacobians = [k.der, K.der, F_h0.der, F_hdef.der, tauh0.der, tauhdef.der, taumean.der
53                 , tauavg.der, hs.der, Fhs.der, tauhs.der, Sy.der];
54 end

```