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 derivatvies. 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 is 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. It had an error for central and 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 and central of 10^{-4} which was too much. The time to calculate was around the same with both step sizes, nevertheless I chose the step size of 10^{-8} for forward and central methods.

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	1849	5	0.783	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 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.

Central:

Optimization completed: The relative first-order optimality measure, 7.607382e-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, 1.512774e-09, 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 was only different for the fmincon function w

2 Automatic Differentiation

A Truss Optimization

```

% ------------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------------

options = optimoptions(@fmincon,'display','iter-detailed','Diagnostics','on',...
    'SpecifyObjectiveGradient',true,'SpecifyConstraintGradient',true);
[xopt, fopt, exitflag, output] = fmincon(@obj, x0, A, b, Aeq, beq, lb, ub, @con,
    options);

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) = (sqrt((stress(i))^2)-25000)/10^2; % check stress both pos and neg
        end

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

    % ------------Separate obj/con You may wish to change------------
    function [f, grad] = obj(x)
        [f, c, ~] = objcon(x);
        type = "complex";
        h = 1e-30;
        [gradc,~] = findGrad(x,f,c,h,type);
        type = "forward";
        h = 1e-8;
        [grad,~] = findGrad(x,f,c,h,type);
    end
    function [c, ceq, cgrad, ceqgrad] = con(x)
        [f, c, ceq] = objcon(x);
        type = "forward";
        h = 1e-8;
        [~, cgrad] = findGrad(x,f,c,h,type);
        ceqgrad = ceq;
    end

    function [grad, cgrad] = findGrad(x,fo,co,h,type)
        % Define method of numerical differentiation
        % "forward", "central", or "complex"
        % Define step size
        [~,sizex] = size(x);
        grad = zeros(sizex,1);
        [nc,~] = size(co);
        cgrad = zeros(nc,nc);
        for index=1:sizex
            if (type=="forward" || type=="central")
                xf = x;
                xf(index) = x(index) + h;
                [f_f,c_f,~] = objcon(xf);
                if(type=="forward")
                    grad(index) = (f_f-fo)/h;
                    for j = 1:nc
                        cgrad(index,j) = (c_f(j)-co(j))/h;
                    end
                else
                    xb = x;
                    xb(index) = x(index) - h;
                    [f_b,c_b,~] = objcon(xb);
                    grad(index) = (f_f-f_b)/(2*h);
                    for j = 1:nc
                        cgrad(index,j) = (c_f(j)-c_b(j))/h;
                    end
                end
```

```
            else
                xI = x;
                xI(index) = x(index)+1j*h;
                [f_im,c_im,~] = objcon(xI);
                grad(index) = imag(f_im)/h;
                for  k= 1:nc
                    cgrad(index,k) = imag(c_im(k))/h;
                end
            end
        end
    end
```

B Automatic differentiation

```
% ------------Starting point and bounds------------
%var= d D n hf   %design variables
x0 = [0.07, 0.67 7.6 1.4]; %starting point
[Values, Jacobians] = getValues(x0)

function [Values, Jacobians] = getValues(x)

    %design variables
    d = x(1);  % height (in)
    D = x(2);  % diameter (in)
    n = x(3); % number of coils (treating as continuous for this example)
    hf = x(4); % free height (in)
    d = valder(d,[1,0,0,0]);
    D = valder(D,[0,1,0,0]);
    n = valder(n,[0,0,1,0]);
    hf = valder(hf,[0,0,0,1]);

    % Constants
    G = 12e6;  % psi
    Se = 45000; % psi
    w = 0.18;
    Sf = 1.5;
    Q= 150000; % psi

    % Analysis variables
    h0 = 1.0; % preload height
    delta0 = 0.4;

    % Output variables
    hdef = h0-delta0;
    k = (G*d^4)/(8*D^3*n);
    K = (4*D-d)/(4*(D-d))+0.62*d/D;
    F_h0 = k*(hf-h0); %Fmin
    F_hdef = k*(hf-hdef); %Fmax
    tauh0 = (8*F_h0*D)/(pi*d^3)*K; %taumin
    tauhdef = (8*F_hdef*D)/(pi*d^3)*K; %taumax

    taumean = (tauhdef+tauh0)/2;
    tauavg = (tauhdef-tauh0)/2;

    hs = n*d;
    Fhs = k*(hf-hs);
    tauhs = (8*Fhs*D)/(pi*d^3)*K;

    Sy = 0.44*Q/(d^w);


    %objective function
    f = F_h0;
    Values = [k.val, K.val, F_h0.val, F_hdef.val, tauh0.val, tauhdef.val, taumean.val, tauavg.val, hs.val, Fhs.val, tauhs.val, Sy.val];
    Jacobians = [k.der, K.der, F_h0.der, F_hdef.der, tauh0.der, tauhdef.der, taumean.der, tauavg.der, hs.der, Fhs.der, tauhs.der, Sy.der];
end
```