MAE 5350: HW #4

Multidisciplinary Design Optimization

Hanfeng Zhai*

Sibley School of Mechanical and Aerospace Engineering,

Cornell University, Ithaca, NY

November 16, 2021

Q1. Scaling

Consider the following optimization problem:

$$\begin{aligned} \text{minimize } & x_1^2 - 0.001x_2^2 + 1000x_3^2 \\ \text{subject to } \quad & x_1^2 + x_2^2 + x_3^2 = 1 \end{aligned} \tag{1}$$

(a) Find the optimal solution to the problem.

Solution: As the semester is approaching to an end, and as I'm learning a lot from this course by coding with MATLAB®, I decided to try to solve problem with `python` this time for the homework to apply what we've learnt to a broader applications.

In this problem, we pick the `'SLSQP'` optimization method (Sequential Least Squares Programming), which can handle nonlinear constraints pretty well with set tolerance [Ref.]. The corresponding tolerance was given at a value of 10^{-6}. A randomly guess initial point $x_0 = (0.1, 0.3, 0.5)$ was given to initiate the optimization. We thence generate the following code for optimize the given problem:

Step 1. Define the objective function, constraints, and initial point:

[1]:
```
import scipy.optimize
fun = lambda x: x[0] ** 2 - 0.001 * x[1] ** 2 + 1000  * x[2] ** 2
cons = ({'type': 'eq', 'fun': lambda x:  x[0] ** 2 +  x[1] ** 2 +  x[2] ** 2 - 1})
x0 = [0.1, 0.3, 0.5]
```

*Email: hz253@cornell.edu

Step 2. Optimize the problem by recall the `minimize` function:

```
[2]: res = scipy.optimize.minimize(objective_function, x0, method='SLSQP', tol=1e-6,␣
     ↪constraints=cons)
```

Step 3. Print out the solution:

```
[3]: print(res)
```

```
     fun: -0.0010000000002877533
     jac: array([ 3.47680325e-05, -2.00000317e-03,  1.08021792e-04])
 message: 'Optimization terminated successfully'
    nfev: 44
     nit: 9
    njev: 9
  status: 0
 success: True
       x: array([-1.11711296e-07,  1.00000000e+00, -2.87906700e-09])
```

Therefore the optimized point is $(-8.45463244 \times 10^{-6}, 1, 3.42745663 \times 10^{-7})$ - which can be approximately considered as the point $(0, 1, 0)$.

(b) Find a non-singular transformation $x = Ly$ such that the condition number of the Hessian matrix of f is close to unity.

Solution: We first need to rescale the objective; assuming the new form of the objective agrees:

$$\begin{gathered} J(\mathbf{x}) = x_1^2 - \tilde{x}_2^2 + \tilde{x}_3^2 \\ \text{where } x_1 = L_1\tilde{x}_1,\ x_2 = L_2\tilde{x}_2,\ x_3 = L_3\tilde{x}_3 \end{gathered} \tag{2}$$

Therefore we can solve:

$$L_2 = \sqrt{1000} = 31.6228 \approx 10,\ L_3 = \sqrt{0.001} = 0.0316 \approx 0.01$$

In this way we can deduce $L_1 = 10$.

With the new $\tilde{x}_1$ and $\tilde{x}_2$, the optimization problem can be written in the new form:

$$\begin{aligned} \text{minimize } & 1001x_1^2 - 0.1\tilde{x}_2^2 + 0.1\tilde{x}_3^2 \\ \text{subject to } \quad & 100x_1^2 + 100\tilde{x}_2^2 + 0.0001\tilde{x}_3^2 = 1 \end{aligned} \tag{3}$$

Now we can compute the new Hessian of the objective, written as

$$\tilde{\mathbf{H}} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & -0.2 & 0 \\ 0 & 0 & 0.2 \end{bmatrix}$$

which agrees with the condition $\mathcal{O}(\tilde{\mathbf{H}}) = -0.0800 \approx -0.1, \longrightarrow$ the problem is rescaled.

(c) Quantify the effect of rescaling the problem, either on the number of iterations or function evaluations required to find the solution (convergence), or the quality of the optimal solution itself, or a combination of the two.

Solution: Based on the rescaled problem in subquestion (b), we can re-optimize the problem with the following `python` code:

```
[4]: import scipy.optimize


     fun = lambda x: 100 * x[0] ** 2 - 0.1 * x[1] ** 2 +  0.1 * x[2] ** 2
     cons = ({'type': 'eq', 'fun': lambda x:  100 * x[0] ** 2 + 100 * x[1] ** 2 + 1e-4␣
      ↪* x[2] ** 2 - 1})


     x0 = [0.1, 0.3, 0.5]
     print(fun)
```

```
<function <lambda> at 0x7f8f08b444c0>
```

```
[5]: res = scipy.optimize.minimize(fun, x0, method='SLSQP', tol=1e-6, constraints=cons)
```

```
[6]: print(res)
```

```
     fun: -0.0009999951013620627
     jac: array([ 1.31615215e-03, -2.00000045e-02,  1.87065307e-05])
 message: 'Optimization terminated successfully'
    nfev: 31
     nit: 7
    njev: 7
  status: 0
 success: True
```

```
x: array([6.57331016e-06, 1.00000015e-01, 9.35251772e-05])
```

From the output we know that the optimized point is $(x_1, \tilde{x}_2, \tilde{x}_3) = (6.57331016 \times 10^{-6}, 1.00000015 \times 10^{-1}, 9.35251772 \times 10^{-5})$. We already know that $x_2 = \tilde{x}_2 L_2$, $x_3 = \tilde{x}_3 L_3$; the scaled back optimal solution is $(6.57331016 \times 10^{-51}, 1.00000013, 9.35251772 \times 10^{-7})$ - the x_1 and x_3 values truns out to be verry small, which can be considered as 0: to verify this point, we use `fmincon` of MATLAB of `'sqp'` to recompute the optimization problem to verify, and generate the following code:

*The **main program** for running the optimization:*

```
% Set nondefault solver options
options2 = optimoptions('fmincon','PlotFcn',{'optimplotx','optimplotfval'},'Display','iter','
    Algorithm','sqp');

% Solve
[solution,objectiveValue] = fmincon(@obj,x0,[],[],[],[],[],[],@MDOcons,...
    options2);

% Clear variables
clearvars options2
```

*The **objective function**:*

```
function f = obj(x)
    f = 100*x(1)^2 - 0.1*x(2)^2 + 0.1*x(3)^2;
end
```

*The **constraints**:*

```
function [c,ceq] = MDOcons(x0)
    x = x0;
    c = [];
    ceq = 100*x(1)^2 + 100*x(2)^2 + 1e-4*x(3)^2 - 1;
end
```

And generate the following output and figure:

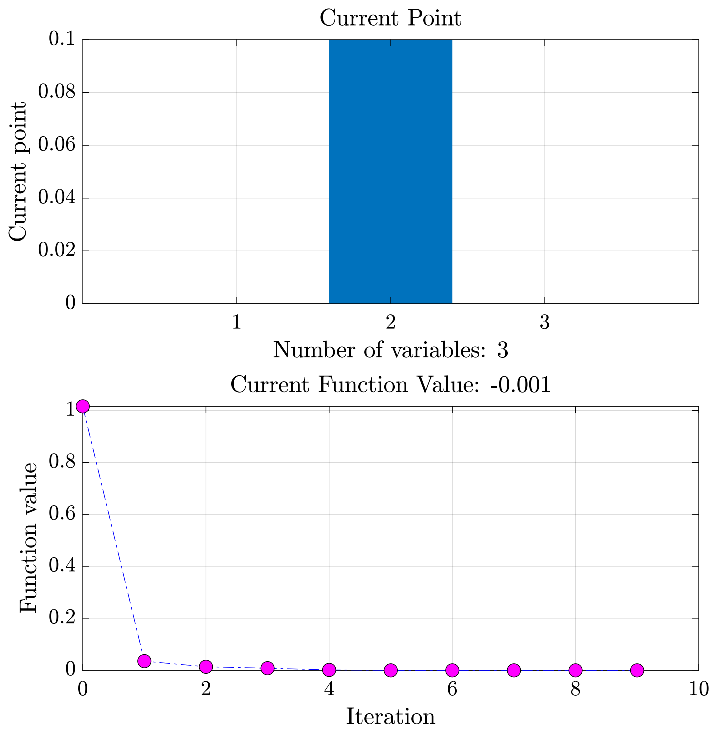
```
Iter  Func-count            Fval   Feasibility   Step Length       Norm of    First-order
                                                                      step     optimality
   0           4    1.016000e+00     9.000e+00     1.000e+00     0.000e+00      2.000e+01
   1          23    3.449389e-02     9.770e+00     4.748e-03     9.017e-02      2.848e+00
   2          27    1.276882e-02     2.222e+00     1.000e+00     1.793e-01      7.996e-02
   3          31    7.254477e-03     3.834e-01     1.000e+00     1.274e-01      3.544e-01
   4          35    6.652805e-04     2.658e-02     1.000e+00     1.641e-01      2.220e-01
   5          39   -9.984504e-04     2.785e-04     1.000e+00     1.216e-01      1.423e-02
   6          43   -9.999895e-04     3.941e-07     1.000e+00     3.726e-03      2.007e-03
   7          47   -9.999993e-04     5.815e-09     1.000e+00     5.035e-05      4.821e-04
   8          51   -1.000000e-03     5.854e-10     1.000e+00     3.955e-05      3.599e-06
```

```
    9          55    -1.000000e-03      2.220e-16      1.000e+00      2.927e-11      3.599e-06

Local minimum possible. Constraints satisfied.

fmincon stopped because the size of the current step is less than
the value of the step size tolerance and constraints are
satisfied to within the value of the constraint tolerance.

<stopping criteria details>
```



We can deduce that the optimization point for the rescaled optimization is $(0, 0.1, 0)$; Therefore it is safe to say that the optimization point is $(0, 1, 0)$, agrees with our previous results before rescale the optimization problem - and verify our hypothesis from the `python` code.

Based on the output, we can deduce that the iterations of the optimization reduces from 9 to 7, which means that rescaling problem makes it easier to solve (converge).

Q2. Isoperformance

The Jacobian matrix at a point x_0 is given as:

$$\nabla J(x_0) = \begin{bmatrix} \frac{\partial J_1}{\partial x_1} & \frac{\partial J_2}{\partial x_1} & \frac{\partial J_3}{\partial x_1} & \frac{\partial J_4}{\partial x_1} \\ \frac{\partial J_1}{\partial x_2} & \frac{\partial J_2}{\partial x_2} & \dots & \frac{\partial J_4}{\partial x_2} \\ & \dots & & \\ & & & \\ & & \dots & \\ \frac{\partial J_1}{\partial x_6} & \dots & \dots & \frac{\partial J_4}{\partial x_6} \end{bmatrix}_{x_0} = \begin{bmatrix} 1 & 4 & 5 & 6 \\ 0.2 & 0.7 & 0.9 & 0.1 \\ 4 & 6 & 1 & 0 \\ 7 & 8 & 8 & 7 \\ 2.4 & 1.7 & 2.9 & -1.1 \\ 12 & 8 & 7 & 1 \end{bmatrix}$$

(a) What are the performance invariant directions that we can step to from x_0?

Solution: First, we apply a singular value decomposition (SVD) to the Jacobian matrix (based on the lecture slides):

$$\Delta J^\mathsf{T} = \mathbf{U\Sigma V}^\mathsf{T}$$

We use MATLAB to conduct such a process:

```
>> A = [1 4 5 6; .2 .7 .9 .1; 4 6 1 0; 7 8 8 7; 2.4 1.7 2.9 -1.1; 12 8 7 1];
>> [U,S,V] = svd(A')

    U =

       -0.5922   -0.5492    0.2306    0.5428
       -0.5645   -0.0401   -0.7525   -0.3367
       -0.5010    0.2446    0.6132   -0.5596
       -0.2823    0.7981   -0.0671    0.5281


    S =

       23.5418         0         0         0         0         0
             0    8.1328         0         0         0         0
             0         0    3.3332         0         0         0
             0         0         0    1.7753         0         0


    V =

       -0.2994    0.6519   -0.0348   -0.2444   -0.6031   -0.2463
       -0.0422    0.0199    0.0194   -0.3256   -0.2033    0.9220
       -0.2658   -0.2696   -0.8939   -0.2304    0.0125   -0.0662
       -0.6221    0.4154    0.0090    0.1830    0.6167    0.1630
       -0.1497   -0.1912    0.3379   -0.8301    0.2828   -0.2406
       -0.6546   -0.5411    0.2918    0.2421   -0.3668   -0.0197
```

Therefore we know matrix $\mathbf{V}$ is written in the form:

$$\mathbf{V} = \begin{bmatrix} -0.2994 & 0.6519 & -0.0348 & -0.2444 & -0.6031 & -0.2463 \\ -0.0422 & 0.0199 & 0.0194 & -0.3256 & -0.2033 & 0.9220 \\ -0.2658 & -0.2696 & -0.8939 & -0.2304 & 0.0125 & -0.0662 \\ -0.6221 & 0.4154 & 0.0090 & 0.1830 & 0.6167 & 0.1630 \\ -0.1497 & -0.1912 & 0.3379 & -0.8301 & 0.2828 & -0.2406 \\ -0.6546 & -0.5411 & 0.2918 & 0.2421 & -0.3668 & -0.0197 \end{bmatrix}$$

where the red part indicate the null space.

Therefore we can compute the performance invariant direction:

$$\begin{aligned} \Delta x &= \alpha \cdot (\beta_1 v_{z+1} + \ldots + \beta_{n-z} v_n) \\ &= \alpha \cdot \left(\beta_1 \begin{bmatrix} -0.6031 \\ -0.2033 \\ 0.0125 \\ 0.6167 \\ 0.2828 \\ -0.3668 \end{bmatrix} + \beta_2 \begin{bmatrix} -0.2463 \\ 0.9220 \\ -0.0662 \\ 0.1630 \\ -0.2406 \\ -0.0197 \end{bmatrix} \right) \end{aligned} \tag{4}$$

(b) Show an example of a step direction (vector), Δx, such that $J(x_0+\Delta x)-J(x_0) \approx 10^{-4}$. **Solution:** Assuming the Hessian at the point x_0 for every objective function J_i is the identity matrix. We also assume the Taylor approximation of J_i writes:

$$J_i(x_0 + \Delta x) = J_i(x_0) + \Delta J_i^{\mathsf{T}}(x_0)\Delta x + \frac{1}{2}\Delta x^{\mathsf{T}} H(x_0)\Delta x + \ldots$$

which further reduces to

$$J_i(x_0 + \Delta x) = J_i(x_0) + \Delta J_i^{\mathsf{T}}(x_0)\Delta x + \frac{1}{2}\Delta x^{\mathsf{T}}\Delta x$$

Based on the instructions, we can establish:

$$\Delta J_i^{\mathsf{T}}(x_0)\Delta x + \frac{1}{2}\Delta x^{\mathsf{T}}\Delta x \approx 10^{-4} \tag{5}$$

By definition, we know that $\Delta J_i^{\mathsf{T}} x = 0$; therefore we need to solve

$$\frac{1}{2}\Delta x^{\mathsf{T}}\Delta x \approx 10^{-4} \tag{6}$$

To solve this equation, substitute Equation (4) back into Equation (6); we generate the following MATLAB® code:

```
A = [1 4 5 6; .2 .7 .9 .1; 4 6 1 0; 7 8 8 7; 2.4 1.7 2.9 -1.1; 12 8 7 1];
[U,S,V] = svd(A);
vec1 = U(:,5);vec2 = U(:,6);
syms Alpha B1 B2; deltaXdirections = vpa(Alpha*(B1*vec1 + B2*vec2),4);
deltaX = @(consts) consts(1)*(consts(2)*vec1 + consts(3)*vec2);
eqn = @(consts) A'*deltaX(consts) + 1/2*deltaX(consts)'*H*deltaX(consts) - 1e-4;
H = eye(6);
guess = [1;1;1];
[consts, fval, exitflag] = fsolve(eqn,guess);
```

We therefore generate the output variable `consts` of the function:

```
consts =


    0.0153
    0.6551
    0.6551
```

We therefore know the constants: $\alpha = 0.0481$, $\beta_1 = 0.6569$, $\beta_2 = 0.6569$, as the example of the step direction Δx:

$$\Delta x = \begin{bmatrix} -0.0085 \\ 0.0072 \\ -0.0005 \\ 0.0078 \\ 0.0004 \\ -0.0039 \end{bmatrix} \tag{7}$$

Verify it we can compute

$$\frac{1}{2}\Delta x^{\mathsf{T}}\Delta x = 1.0046 \times 10^{-4} \tag{8}$$

References

[1] Olivier L. de Weck & Marshall B. Jones. Isoperformance: Analysis and Design of Complex Systems with Known or Desired Outcomes. Fourteenth Annual International Symposium of the International Council on Systems Engineering (INCOSE) Toulouse, France, 21 June – 24 June 2004.