ME 575 - Assignment #4 Computing Derivatives

Overview: We are going to optimize the same 10-bar truss problem you saw in Homework #1. However, this time we will focus on providing derivatives to the optimizer. Our goal is to use various approaches for numerically computing derivatives, understand their advantages and disadvantages, and apply these methods in an optimization problem to better understand the impact of accurate gradients. The truss problem is described in D.2.2 of the book and the code is available in the repository: https://github.com/mdobook/resources/tree/main/exercises/tenbartruss

---

4.1 Derivatives using Complex Step

Calculate the Jacobian for the 4-dimensional Rosenbrock function (see Equation D.3) using the complex step at the point x = [0.5, -0.5, 0.5, 1], with h going from 0.1, 0.001, 0.001, ... 1e-30. This means there are 4 design variables, and therefore the gradient will be 4x1 and the Jacobian would be 1x4. Compare this to the answer with forward finite differencing using the same values of h = 0.1, 0.001, 0.001, ... 1e-30. Compare both the complex and forward step to the exact derivative in a plot similar to Fig 6.9 in the textbook. (Note, you will have to decide how to present the error between the Jacobians).

4.2 Derivatives using AD

Solve for the Jacobian at the point $(x = 2, y = 1.5)$, for the vector function $f = [f_1, f_2]^T$ presented in the following python code. Show your work as needed (by hand or with code).

```
a = x**2 + 3*y**2
b = sin(a)
c = 3*x**2 + y**2
d = sin(c)
e = x**2 + y**2
g = -0.5*exp(-e/2)
f1 = a + c + g
h = x*y
f2 = a - g + h
```

4.3 Truss Derivatives

Before we can optimize the truss we need to provide the following derivatives:

- The derivatives of mass (objective) with respect to cross-sectional areas (design vars), dm/dA_i, for $i = 1,...,n_x$.
- The derivatives of stress (constraints) with respect to cross-sectional areas (design vars), $d\sigma_i/dA_j$, for $i = 1,...,n_g$ and $j = 1,...,n_x$. In other words, this is an $n_g \times n_x$ matrix (and in this case $n_g = n_x$).

You will likely need to make some minor adjustments to accommodate complex step and AD. Compute the derivatives of the objective (mass) and the constraints (stresses) using the following three methods:

- A finite-difference formula of your choice.
- The complex-step derivative method.
- Algorithmic differentiation.

Report the relative errors of the various methods. Results will differ somewhat depending on your evaluation point. Discuss your findings and the relative merits of the approaches in 200 words or less.

4.4 Truss Optimization With the provided gradients you can now optimize the truss. The problem is identical to Homework #1, except this time ***you will supply the derivatives***. Solve this optimization problem using **one** of the exact derivative methods (not finite differencing). Generate a convergence plot and report the number of function calls to the truss function required to converge. Discuss your findings in 200 words or less.

Tips:

- Your overall constraint derivative matrix will be non-square and so you will know immediately if it is in the correct orientation or not. However, it is made up of two square matrices and so it is easy to accidentally put those square matrices in backwards (i.e., the transpose of what it should be).
- Most optimizers have a helpful option to check your user-supplied derivatives (usually against an internal finite differencing). In this case, it's not so much a question of the correctness of your gradients, because you have likely already established that by comparing them using separate methods, but it is more a question of whether or not you are supplying them to the optimizer in the requested format/order.
- Suggested AD tools (this is not to suggest that they are the best for all applications, but they could be useful for this assignment).
  - Matlab: AutoDiff_R2016b (download as "toolbox"). The documentation is ok, but not very clear. Two things that may help: 1) Use `amatinit` to convert your design variables to dual numbers, 2) The function `ajac` will allow you to extract the Jacobian from an output (e.g., `J = ajac(x, 0)`).
  - Python: JAX
  - Julia: ForwardDiff
- As discussed in class, for both complex step and AD small changes are sometimes needed. These are sometimes a bit tricky so here is some potential guidance.
  - Complex step.
    - Matlab: compare transpose with complex conjugate transpose.
    - Python: You'll need to initialize the numpy K and S matrices with complex types (see dtype).
    - Julia: no changes needed
  - AD.
    - Matlab: You'll get an error when adding items to the stiffness matrix (K). The problem is that K was initialized with oats and Ksub with dual number types so when you add them together you get a dual number type but you can't store that back into K since it is expecting oats. To initialize K properly the easiest thing to do is just to multiply it by an element in A when it is initialized. That way if A is a float K will initialize with oats, and if A contains dual numbers then K will initialize with dual numbers. In other words: `K = A(1)*zeros(...)`.
    - Python: You can use jax.numpy as a drop-in replacement for all the numpy operations. However, jax arrays are immutable so any operations that modify an array need to be replaced with the corresponding functions jax supplies with the at property
    - Julia: Similar initialization needed as in Matlab and you can use the same trick. However, a better way is to set the `eltype` in the call to zeros (for K). By default it is a `Float64` and instead it should be `eltype(A)`. In other words, it will be Floats if A contains Floats, or a Dual type if A contains Duals.