Assignment #3: Computing Derivatives — ME 575

due 2/24/2021 before midnight via Learning Suite — **50 possible points**

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 advantageous and disadvantageous, and apply these methods in an optimization problem to better understand the impact of accurate gradients. The truss problem is described in C.2.2 of the book and the code is available in the repository: `https://github.com/mdobook/resources`

3.1 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 x 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, and certainly will need to make modifications for the implicit analytic method. Compute the derivatives of the objective (mass) and the constraints (stresses) using **three** of the following four methods:

(a) A finite-difference formula of your choice.

(b) The complex-step derivative method.

(c) Algorithmic differentiation.

(d) The implicit analytic method.

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.

3.2 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). Report the following:

(a) a convergence plot. Plot the number of function calls on the x-axis and the norm of the Lagrangian on the y-axis (first order optimality condition for a constrained problem). If your solver doesn't provide that info, some other relevant convergence metric is fine.

(b) the number of function calls to the *truss* function required to converge. Side note: one starting point can be misleading so these kinds of results are more useful if you use multiple starting points and average (or other statistics) but that isn't required for this assignment.

Discuss your findings.

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 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 the 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 to suggest they are necessarily the best, just that I've tested them for this problem).
  - Matlab: AutoDiff_R2016b (download as "toolbox"). The documentation is ok, but not that detailed. 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: AlgoPy
  - Julia: ForwardDiff
- As discussed in class, for both complex step and AD small changes are sometimes needed. These are sometimes a bit tricky to resolve without some experience so here is some 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 floats 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 floats. 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 floats, and if A contains dual numbers then K will initialize with dual numbers. In other words: `K = A(1)*zeros(...)`.
    - Python: Python can be a difficult language to perform AD in directly. So unfortunately a lot of changes are required in this case. 1) like complex step you'll need to initialize K for a dual type, see the first warning in the algopy docs, 2) the indexing and deletion operations don't work so we'll have to manually index. Change `K[np.ix_(idx, idx)] += Ksub` to
      ```
      for j in range(4):
        for k in range(4):
          K[idx[j], idx[k]] += Ksub[j, k]
      ```
      3) Change `K = np.delete(K, remove, axis=0); K = np.delete(K, remove, axis=1)` to `K = K[:8, :8]`. 4) Finally, you'll have to use algopy's version of `solve` and `dot`
    - Julia: Similar initialization need as Matlab, you can use the same trick (though that's not necessarily the best way).