Exercises for Lecture Course on Numerical Optimization (NUMOPT)
Albert-Ludwigs-Universität Freiburg – Winter Term 2022-2023

Exercise 4: Calculation of Derivatives, Equality Constrained Optimization

Prof. Dr. Moritz Diehl, Dimitris Kouzoupis, Andrea Zanelli and Florian Messerer

1. **Simple equality constrained optimization:** Recall the simple equality constrained example discussed in the lecture,

$$\min_{x_1, x_2 \in \mathbb{R}} \quad x_2 \tag{1a}$$

$$\text{s.t.} \quad x_1^2 + x_2^2 - 1 = 0, \tag{1b}$$

which consists of a linear objective and a nonlinear equality constraint.

   (a) Is this problem convex?

   (b) On paper, derive the first order necessary conditions (FONC) of optimility for this problem. Use FONC Variant 4 from the lecture notes (Theorem 11.6).

   (c) These FONC define a root finding problem. Code your own implementation of Newton's method to solve it. To which point (or points) does your method converge?

   (d) Problem (1) has two stationary points (points that fulfill the FONC). For both, use the second order sufficient conditions or second order necessary conditions to determine whether they are local minima (on paper). Can you decide whether one of them is also a global minimum? If yes, how?

   (e) Pick one of the stationary points. Invent an additional equality constraint, such that the linear independence constraint qualification (LICQ) is violated at this point.

2. **LICQ and Newton method:** Consider the following nonlinear equality constrained optimization problem,

$$\min_{x \in \mathbb{R}^n} \quad f(x) \tag{2a}$$

$$\text{s.t.} \quad g(x) = 0 \tag{2b}$$

with $g : \mathbb{R}^n \to \mathbb{R}^m$. The linear system associated with the k-th iteration of the Newton method is

$$\begin{bmatrix} B & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ -\Delta\lambda \end{bmatrix} = - \begin{bmatrix} \nabla f(x_k) - \nabla g(x_k)\lambda \\ g(x_k) \end{bmatrix} \tag{3}$$

with $\lambda \in \mathbb{R}^m$, $B = \nabla^2 f(x_k) - \sum_{i=1}^{p} \lambda_i \nabla^2 g(x_k)$ and $A = \nabla g(x_k)^\top$.

Prove that if A has full row rank and $Z^T B Z \succ 0$, with the columns of $Z \in \mathbb{R}^{n \times (m-n)}$ forming a basis for the null space of A, the iteration matrix in (3) is invertible.

Remark: this provides a sufficient condition under which a search direction can be obtained.

3. **Control of a dynamic system:** Our goal is to drive the state $x_k \in \mathbb{R}$ of a discrete time system to the origin using controls $u_k \in \mathbb{R}$ in N time intervals, where subscript k denotes discrete time. The initial state is $x_0 = \bar{x}_0$, the dynamics of the system are

$$x_{k+1} = \phi(x_k, u_k) \quad \text{with} \quad \phi(x_k, u_k) = x_k + \frac{T}{N}((1 - x_k)x_k + u_k), \tag{4}$$

$k = 0, \ldots, N$, and T is the terminal time (corresponding to discrete time N). We can formulate this as the following optimization problem,

$$\min_{x,\,u} \quad \sum_{k=0}^{N-1} u_k^2 + q x_N^2 \tag{5a}$$

$$\text{s.t.} \quad x_0 = \bar{x}_0, \tag{5b}$$

$$x_{k+1} = \phi(x_k, u_k), \quad k = 0, \ldots, N-1, \tag{5c}$$

with control trajectory $u = (u_0, \ldots, u_{N-1}) \in \mathbb{R}^N$ and state trajectory $x = (x_0, \ldots, x_N) \in \mathbb{R}^{N+1}$. The objective (5a) expresses our aim to bring the terminal state x_N to zero, using the least amount of effort in terms of control actions u_k. Weighting factor $q \in \mathbb{R}$ defines the trade-off between these two aims. The equality constraints (5b) and (5c) uniquely determine the state trajectory $x_0, \ldots x_N$ given controls $u_0, \ldots, u_{N-1}$. Therefore we can write (5) in the equivalent unconstrained form:

$$\min_{u} \quad \sum_{k=0}^{N-1} u_k^2 + \Phi(u) \tag{6a}$$

using the constraints to eliminate all states x_k and to define $\Phi(u)$ as

$$\Phi(u) = \varphi(\phi(\ldots(\phi(\phi(\bar{x}_0, u_0), u_1), \ldots), u_{N-1})), \tag{7}$$

with $\varphi(x_N) = q x_N^2$.

This function is implemented for you in the template (`Phi.py` for Python / `Phi.m` for MATLAB). You can call it as `f = Phi(u, param)` where $u \in \mathbb{R}^N$ is a control trajectory and param a structure with the problem parameters, similar to the previous exercise sheet. You will now implement different methods for obtaining derivatives of Φ and compare their results. We will use a random control trajectory u_{rand} to evaluate the derivatives. This has already been implemented for you in `test_derivatives.py` (Python) / `test_derivatives.m` (MATLAB).

(a) Use your code from last week to differentiate $\Phi(u)$ at u_{rand} with finite differences.

(b) Using the same syntax, write a function `[F,J] = i_trick(fun,x,param)` that calculates the Jacobian of $\Phi(u)$ using the imaginary trick.

(c) Now let's implement both forward and backward modes of Automatic Differentiation. Before you start coding, which of the two you think would perform faster in our example and why?

(d) Write a function `[F,J] = Phi_FAD(u,param)` that returns the function evaluation and the Jacobian of $\Phi(u)$ using the forward mode of AD. You can start by copying the code from the given function `Phi`.

Note: Unlike your implementations of finite difference and the imaginary trick, you will not write a general purpose AD function. Rather, your code will be hand-tailored to the given function $\Phi(\cdot)$.

Hint: It is recommended that you consider function $\phi(\cdot, \cdot)$ as 'elementary operation' of which $\Phi(u)$ is composed, instead of further decomposing $\phi(\cdot, \cdot)$ into the additions and multiplications of which it consists (as a general-purpose AD algorithm would probably do). This will greatly simplify your implementation.

(e) Write a function `[F,J] = Phi_BAD(u,param)` that implements the backward mode of AD.

(f) The 'AD' in 'CasADi' stands for Algorithmic Differentiation, since this is how CasADi computes derivatives. Using CasADi we can compute the Jacobian of our nonlinear function within a few lines only.

Complete the template `Phi_casadi.py` (Python) / `casadi_script.m` (MATLAB) to compute the Jacobian of $\Phi(u)$. Note that this time we are not using the `Opti()` environment, since we are interested in derivatives only.

(g) Once you have everything implemented, run the script `test_derivatives.py` (Python) / `test_derivatives.m` (MATLAB) to check (and demonstrate) that your results are correct. Which order of magnitude should the difference between the results of the various differentiation methods be? If you did not implement all methods, comment out or delete the corresponding lines.

(h) Use `timeit.default_timer()` (Python) / `tic toc` (MATLAB) to measure the total time spent in the derivative calculations for the different functions you have implemented, with $N = 200$. For CasADi make sure you are only measuring the function evaluation time, i.e., without the setup time / time for running the script from (f). How do the timings change if you set $N = 1000$? Give a short reason for this behaviour. Report the time values for all methods and both values of N.

Remark: Depending on the performance of your CPU you may adapt the given values of N for purpose of better demonstration / to decrease the runtime. The cost of calling one of your derivative functions should be in the order of magnitude from approx. 10^{-5} to 1 seconds.

(i) **Extra:** Solve the optimization problem in (5) using the BFGS method with globalization similarly to the previous exercise (you can simply adapt your code from the last exercise sheet). Plot the state and controls as a function of time to confirm that the system behaves as expected. Don't forget to add the derivative of the quadratic term $\sum_{k=0}^{N-1} u_k^2$ to your result for $\nabla\Phi(u)^\top$ when computing the Jacobian of your objective function. Use $N = 50$, $x_0 = 2$, $T = 5$ and $q = 50$.