Technical University of Munich
Department of Mathematics
Chair of Mathematical Optimization
Johannes Milz, M. Sc.
Fabian Schaipp, M. Sc.

Introduction to Software Tools for Nonlinear Optimization (WS 2020/2021)
Exercise Sheet 1

Preliminaries Download the file `sheet1.zip` from Moodle page and extract the archive.

The programming exercise have been tested on the computers located in the MI building using remote desktop access. The instructions for the second and third exercise sheet are tailored to `Linux` operating systems.

P1. Computation of a polyhedron with 60 vertices (C_{60}-molecule)

Determine 60 vertices $v_i = (v_i^1, v_i^2, v_i^3)^T \in \mathbb{R}^3$ of a polyhedron such that the surface consists of 12 regular pentagons and 20 regular hexagons with side length 1 (C_{60}-molecule). Two neighboring nodes v_i, v_j fulfill

$$\|v_i - v_j\|_2^2 - 1 = 0.$$

Additionally, the nodes $(v_{i_1}, v_{i_2}, v_{i_3}, v_{i_4}, v_{i_5})$ of a pentagon are related by

$$\|v_{i_j} - v_{i_{j+2}}\|_2^2 - 4\sin^2(3\pi/10) = 0, \quad j = 1, \dots, 5,$$

where $i_6 = i_1$, $i_7 = i_2$. Analogously, for every hexagon $(v_{i_1}, v_{i_2}, v_{i_3}, v_{i_4}, v_{i_5}, v_{i_6})$ we obtain

$$\|v_{i_j} - v_{i_{j+2}}\|_2^2 - 4\sin^2(\pi/3) = 0, \quad j = 1, \dots, 6,$$

where $i_7 = i_1$, $i_8 = i_2$. In order to fix the location of the polyhedron, we additionally require $v_1 = 0$, $v_2^1 = v_2^2 = 0$, $v_3^1 = 0$. These $90 + 60 + 120 + 6 = 276$ equations can be written in the form

$$F(x) = 0$$

where $F : \mathbb{R}^{180} \to \mathbb{R}^{276}$ and $x = (v_1, \dots, v_{60})^T$. For the numerical computations we reformulate the system of equations as an optimization problem:

$$\min_{x \in \mathbb{R}^{180}} f(x) := (1/2)\|F(x)\|_2^2.$$

We analyze the performance of optimization algorithms applied to this problem.

A gradient descent and a Gauß–Newton algorithm are implemented in `MATLAB` and `Julia`. The corresponding procedures are called `gradmethod` and `GaussNewton`. Additionally, `xinit1`, `xinit2`, etc., provide initial values.

a) Decide on using `MATLAB` or `Julia`. `Julia` users install `NLPModels` and `PyPlot` via `Julia`'s package manager. Visualize the optimal solution `xopt` of the optimization problem by executing the `visualize_solution`-file.

b) The `main`-file containes calls to the gradient method and Gauß–Newton method. The codes print k, $f(x^k)$, $\|\nabla f(x^k)\|_2$ and the step sizes. If you use `MATLAB`, press `Enter` to execute an additional iteration. After every iteration, the current iterate is visualized in a `MATLAB` figure. If you use `Julia`, a plot of each iterate is saved as a `PDF` in the subdirectory `plots`.

Apply the optimization methods using the six initial values and compare their performances.

c) Increase the maximum number of iterations `itmax` to 100, and rerun the Gauß–Newton method with the initial value `xinit6`. Provide an interpretation of the output.

P2. Gradient descent, Newton's and inverse BFGS method

We compare the performance of three optimization algorithms applied to unconstrained optimization problems.

a) Decide on using `Julia` or `MATLAB`, and switch to the corresponding subdirectory. The gradient descent, and the Newton and the inverse BFGS method and some test functions are already implemented. Open the `main`-file.

b) Test the optimization algorithms on the following functions and initial values:

	function	initial value	name
1)	$(x_1 - 2x_2)^2 + (x_1 - 4)^4$	$(0,0)^T$, $(10,10)^T$	polynomial
2)	$x_1^2 + kx_2^2$, for $k \in \{1, 10, 100\}$	$(10,1)^T$, $(1,100)^T$	parameterized quadratic function
3)	$100(x_2 - x_1^2)^2 + (1 - x_1)^2$	$(0,0)^T$, $(-1.2,1)^T$	Rosenbrock's function
4)	$(x_1 - 10^6)^2 - (x_2 - 2 \cdot 10^{-6})^2$	$(1,1)^T$, $(10,10)^T$	Brown's badly scaled function

- Compare the performance of the algorithms applied to the polynomial.
- How does the empirical performance differ for the parameters $k \in \{1, 10, 100\}$?
- Does superlinear or quadratic convergence occur? Why or why not?
- Check whether the sufficient second order optimality conditions are satisfied.

c) Choose and implement your own test function, which should have a unique optimal solution. Apply the algorithms to your test function and compare their empirical performance.

The following subproblems are optional. In order to solve these you need to modify the algorithms because you require the iterates. You can find hints about useful `MATLAB`/`Julia`-commands in the corresponding `hints`-file.

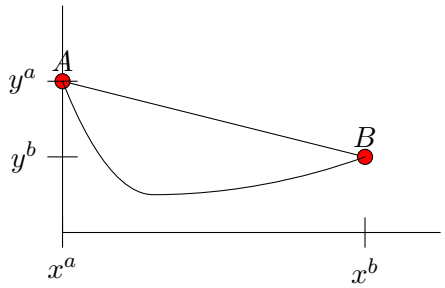
d) Create a contour plot for your test function and visualize the iterates in the same plot.

e) Compute the optimal solution of your test function. Create a convergence plot showing the Euclidean distance of the iterates to the optimal solution. Use a logarithmic scale for the vertical axis. Compare the empirical convergence rates.

f) Report the run times of the algorithms and visualize the distance of the iterates to the optimal value as a function of the run time.

P3. Brachistochrone Problem

Given two points $A = (x^a, y^a)$ and $B = (x^b, y^b)$, find the curve such that a particle, with the initial velocity $v_0 > 0$, slides (without friction) on the curve from A to B in the least time. Here $x^a < x^b$ and $y^a > y^b$.



Let $y : [x^a, x^b] \to \mathbb{R}$ be a continuously differentiable function with $y(x^a) = y^a$ and $y(x^b) = y^b$. The time for the particle to get from A to B on the curve y is given by

$$T(y) := \int_{x^a}^{x^b} \left(\frac{1 + y'(x)^2}{v_0^2 + 2g(y^a - y(x))} \right)^{1/2} \mathrm{d}x,$$

where g is the gravitational acceleration. Mathematically, the Brachistochrone problem can be formulated as the optimization problem

$$\min_{y \in Y} T(y),$$

where

$$Y = \Big\{ y \in C^1[x^a, x^b] \colon \; y(x^a) = y^a, \; y(x^b) = y^b, \; v_0^2 + 2g(y^a - y(x)) > 0 \quad \text{for all} \quad x \in [x^a, x^b] \Big\}.$$

The Brachistochrone problem is an infinite-dimensional optimization problem which we discretize. We divide the interval $[x^a, x^b]$ into $n + 1$ disjoint sub-intervals $[x_k, x_{k+1})$ of length $h = (x^b - x^a)/(n + 1)$, where $x_k = x^a + kh$ and $k = 0, \ldots, n$. Let y be a continuous function that is linear on each subinterval $[x_k, x_{k+1})$. It is uniquely determined by its values $\mathbf{y}_k = y(x_k)$ at the endpoints of the sub-intervals. In order to fulfill the boundary conditions, we set $\mathbf{y}_0 = y^a$ and $\mathbf{y}_{n+1} = y^b$. Applying the mid point rule to T on every sub-interval yields

$$T(y) \approx h \sum_{k=0}^{n} \left(\frac{1 + ((\mathbf{y}_{k+1} - \mathbf{y}_k)/h)^2}{v_0^2 + 2g(y^a - (\mathbf{y}_k + \mathbf{y}_{k+1})/2)} \right)^{1/2} =: T_h(\mathbf{y}).$$

We formulate the finite-dimensional problem

$$\min_{\hat{\mathbf{y}} \in \mathbb{R}^n} T_h(\mathbf{y}), \quad \mathbf{y} = (y^a, \hat{\mathbf{y}}^T, y^b)^T.$$

In the following, we solve the finite dimensional problem for several values of n. Decide on using `MATLAB` or `Julia`.

a) We have implemented the function and gradient computations of T_h in `brachiF.m` and `brachiF.jl`. Apply the gradient method for $n = 10, 20, 40, 80, 500$. To do so, execute either `main.m` or `main.jl` and modify n as required. By how much and why does the run time increase with increasing n?

b) The gradient method may not be a suitable large problem's dimensions n. Therefore, we want to use Newton's method, which requires us to supply the Hessian of T_h. The Hessian is implemented in `brachiF`. Apply the globalized Newton method for $n = 80$, 500, 1000, 5000, 10000. Compare the run times of Newton's method with those of the gradient method, and explain your findings.

c) Visualize the sparsity pattern of the Hessian matrix. You can find hints in the corresponding `hints`-file.

d) Modify the code for the use of sparse Hessians. Exploit the fact that the Hessian matrices are tridiagonal. Compare the run times and the memory usage with the former implementation.

P4. Automatic Differentation

We use Automatic Differentation to compute derivatives. Automatic Differentation can be used to automatically calculate derivatives of mappings that are implemented as computer code.

a) For **`MATLAB` users**: Download `ADiGator` and add it to `MATLAB`'s path. The README-file indicates the location of `ADiGator`'s user's guide.

For **`Julia` users**: Install `ForwardDiff`, `JuMP`, and `Ipopt` using `Julia`'s package manager.

b) Open the file `derivatives` in the subdirectory `polynomial`. Which commands are used to automatically compute derivatives? Are the results correct?

Execute `solve`.

c) Use Automatic Differentation within an optimization method to solve the Brachistochrone problem.

`Julia` users model the problem with `JuMP` and solve it using `Ipopt`. Use the commands `@NLobjective` and `sum` to implement the objective function.

`MATLAB` users use `ADiGator`'s command `adigatorGenFiles4Fminunc` to automatically compute derivatives, and solve the problem using `fminunc`. The objective function is implemented in `objective` located in the subdirectory `brachistochrone`.

Hint: Check out the users' manuals and documentations.