Fall 2021: Computational and Variational Methods for Inverse Problems
CSE 397/GEO 391/ME 397/ORI 397
Assignment 2: Optimization methods and automatic differentiation
due October 20, 2021

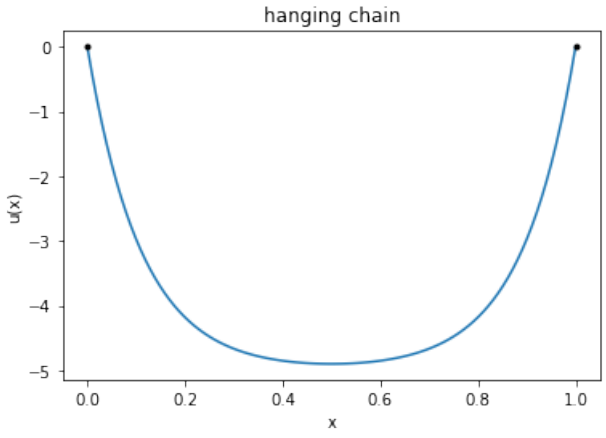
In this assignment we will solve the *catenary problem* to explore the convergence behavior of several optimization methods we studied in class. We will also introduce a tool for automatic differentiation. The catenary problem is a classical problem in variational calculus that seeks to find the shape of a hanging chain that is fixed at both ends:

(credit: https://en.wikipedia.org/wiki/Catenary#/media/File:Kette_Kettenkurve_Catenary_2008_PD.JPG)

We provide a jupyter notebook called *catenary.ipynb* on Canvas that will walk you through problems 2, 3, and 4 using python/numpy/scipy/autograd. We recommend that you complete problems 2, 3, and 4 by filling out this notebook. You may solve this assignment using any language/framework of your choosing, but using another language may make the assignment more difficult.[1]

Problem 1: Derivation of the optimization problem

Let $u : [0, 1] \to \mathbb{R}$ denote the height (positive upward) of an idealized chain as a function of horizontal position, x. Here the left end of the chain is fixed at $x = 0$ and height $u(0) = 0$, and the right end of the chain is fixed at at $x = 1$ and height $u(1) = 0$:



[1] You will have to track down optimization solvers and automatic differentiation tools in that language.

To determine the shape of the hanging chain, we find the $u(x)$ that minimizes the potential energy $E(u)$ of the chain, subject to the constraint that the chain is fixed at both ends and has a given length L_0. This leads to the following *constrained optimization* problem:

$$\begin{aligned} \min_u \quad & E(u) \\ \text{such that} \quad & L(u) = L_0 \\ \text{and} \quad & u(0) = u(1) = 0. \end{aligned}$$

where $L(u)$ is the length of the chain for a given shape $u(x)$.

The solution to this optimization problem may be approximated by solving the following *unconstrained optimization* problem,[2] in which the length constraint is enforced approximately using a quadratic penalty method:

$$\begin{aligned} \min_u \quad & E(u) + \alpha \left(L(u) - L_0\right)^2 \\ \text{such that} \quad & u(0) = u(1) = 0. \end{aligned} \tag{1}$$

Here $\alpha > 0$ is a penalty parameter. As $\alpha \to \infty$, the solution found using the penalty method approaches the solution to the original constrained optimization problem. However the penalty term makes the unconstrained optimization problem (1) increasingly ill-conditioned, and thus difficult to solve, as $\alpha \to \infty$, so we typically use a large enough, but not too large, value of α.

a) Show that the length of the chain, $L(u)$, is given by

$$L(u) = \int_0^1 \sqrt{1 + (u')^2}\, dx,$$

where $u' := \frac{du}{dx}$.

b) Show that the potential energy of the chain, $E(u)$, is given by

$$E(u) = \int_0^1 \rho g u \sqrt{1 + (u')^2}\, dx,$$

where ρ is the linear density (mass per unit length) of the chain, and g is the gravitational acceleration constant.

For convenience, let $\rho g = 1$ for all subsequent problems, so that $E(u) = \int_0^1 u\sqrt{1 + (u')^2}\, dx$.

Problem 2: Discretization

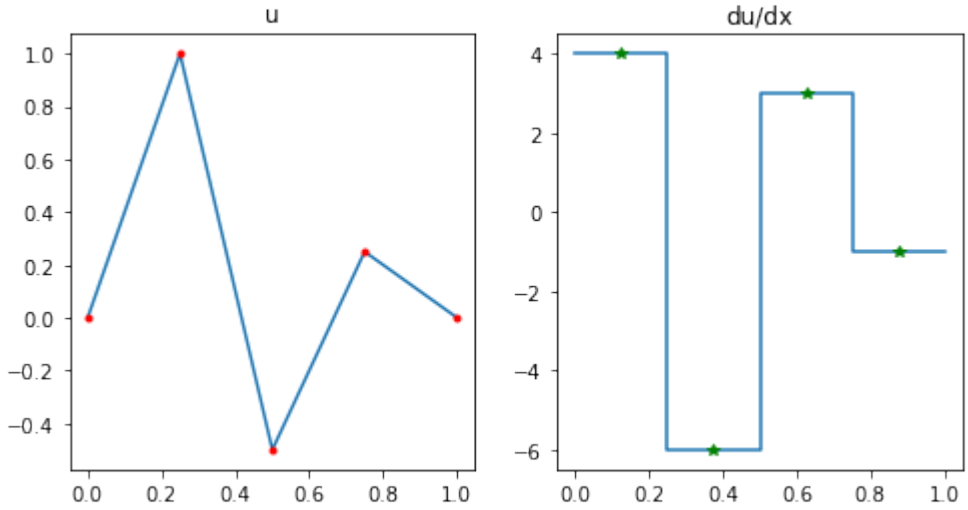
We discretize the optimization problem (1) by approximating $u(x)$ as a continuous piecewise linear function on $[0, 1]$, which is defined by its values at $N + 2$ equally spaced nodes at locations

$$0, h, 2h, 3h, \ldots, Nh, 1.$$

The spacing between nodes is given by $h = \frac{1}{N+1}$. Let $\mathbf{u} \in \mathbb{R}^{N+2}$ be the vector of nodal values at the nodes. Note that here we are including the values at the endpoints in the vector $\mathbf{u}$.

[2] The endpoint constraints $u(0) = u(1) = 0$ will be directly built into our representation of $u(x)$, so it's ok to call this problem unconstrained.

a) Since u is continuous piecewise linear, the derivative u' is piecewise constant. This is illustrated in the image below. Write a function that takes $\mathbf{u} \in \mathbb{R}^{N+2}$ as input, and returns the vector of derivative values, $\mathbf{u}' \in \mathbb{R}^{N+1}$, where the entries of $\mathbf{u}'$ are the values of u' in the intervals (*cells*) between consecutive nodes.



b) Write a function that takes in $\mathbf{u} \in \mathbb{R}^{N+2}$ as input, and returns the arc length, $L(u) \in \mathbb{R}$, of the corresponding continuous piecewise linear function u as output. Hint: the integrand is a piecewise constant function.

c) Write a function that takes in $\mathbf{u} \in \mathbb{R}^{N+2}$ as input, and returns the energy, $E(u) \in \mathbb{R}$ as output. Hint: the integrand is a *discontinuous* piecewise linear function.

Problem 3: Objective, gradient, and Hessian

In this problem we implement the objective function, gradient, and Hessian for the unconstrained optimization problem (1), using the automatic differentiation library *autograd* [3] to compute derivatives. We will use finite difference checks to verify that the derivatives are computed correctly.

Let $J : \mathbb{R}^N \to \mathbb{R}$,

$$J(\mathbf{u}_{\text{int}}) := E(u) + \alpha \left(L(u) - L_0\right)^2$$

denote the discretized objective function for the unconstrained optimization problem (1), where $\mathbf{u}_{\text{int}} \in \mathbb{R}^N$ is the vector consisting of the *interior* nodal values of u (recall $\mathbf{u} \in \mathbb{R}^{N+2}$ includes the values for the endpoints in the first and last entries). In the Jupyter notebook, we provide a function that takes in $\mathbf{u}_{\text{int}} \in \mathbb{R}^N$ as input, and returns $J(\mathbf{u}_{\text{int}})$ as output. If you are using another programming language, you will need to write this function yourself using the functions you already wrote for $E(u)$ and $L(u)$.

a) Use automatic differentiation to construct a function that takes in $\mathbf{u}_{\text{int}} \in \mathbb{R}^N$ as input, and computes the gradient,

$$\boldsymbol{g}(\boldsymbol{u}_{\text{int}}) := \frac{\partial J}{\partial \boldsymbol{u}_{\text{int}}}(\boldsymbol{u}_{\text{int}}) \in \mathbb{R}^N$$

as output.

[3] Automatic differentiation (AD) is a technique to differentiate the output of code with respect to its input; it makes use of knowledge of how to differentiate elementary functions, and chains together derivatives of each line of code library library *autograd* to*autograd* to (i.e., derivatives of output with respect to input) to form the derivatives of the overall code. Autograd is a popular library for AD of python codes.

b) Check that the gradient is computed correctly by using the gradient to compute the directional derivative of J in a random direction $\boldsymbol{h}_{\text{int}} \in \mathbb{R}^N$, and comparing this to to the finite difference approximation,

$$\boldsymbol{g}(\boldsymbol{u}_{\text{int}})^T \boldsymbol{h}_{\text{int}} \approx \frac{1}{s} \left(J(\boldsymbol{u}_{\text{int}} + s\boldsymbol{h}_{\text{int}}) - J(\boldsymbol{u}_{\text{int}}) \right),$$

where $s > 0$ is a small step size. Make a log-log plot of the discrepancy between the directional derivative and the finite difference approximation as a function of step size s ranging from $s = 10^{-13}$ to $s = 10^0$.

c) Use automatic differentiation to construct a function that takes $\boldsymbol{u}_{\text{int}} \in \mathbb{R}^N$ and $\boldsymbol{p}_{\text{int}} \in \mathbb{R}^N$ as input, and computes the Hessian matrix-vector product

$$\boldsymbol{H}(\boldsymbol{u}_{\text{int}})\, \boldsymbol{p}_{\text{int}} \in \mathbb{R}^N$$

as output. Here $\boldsymbol{H}(\boldsymbol{u}_{\text{int}}) = \frac{d^2 J}{d\boldsymbol{u}_{\text{int}}^2}(\boldsymbol{u}_{\text{int}}) \in \mathbb{R}^{N \times N}$ is the Hessian at $\boldsymbol{u}_{\text{int}}$. Compute the Hessian matrix-vector product *matrix-free*. Do not form the Hessian matrix and then multiply it with $\boldsymbol{p}_{\text{int}}$. Hint: the Hessian is the derivative of the gradient, so

$$\boldsymbol{H}(\boldsymbol{u}_{\text{int}})\, \boldsymbol{p}_{\text{int}} = \frac{d}{d\boldsymbol{u}} \left(\frac{dJ}{d\boldsymbol{u}_{\text{int}}}(\boldsymbol{u}_{\text{int}}) \boldsymbol{p}_{\text{int}} \right) = \frac{d}{d\boldsymbol{u}_{\text{int}}} \left(g(\boldsymbol{u}_{\text{int}})^T \boldsymbol{p}_{\text{int}} \right).$$

In other words, $\boldsymbol{H}(\boldsymbol{u}_{\text{int}})\, \boldsymbol{p}_{\text{int}}$ is the gradient of the scalar-valued function

$$q(\boldsymbol{u}_{\text{int}}) := g(\boldsymbol{u}_{\text{int}})^T \boldsymbol{p}_{\text{int}}.$$

d) The Hessian-vector product may be approximated by finite differencing the gradient in the direction of the vector:

$$\boldsymbol{H}(\boldsymbol{u}_{\text{int}})\, \boldsymbol{p}_{\text{int}} = \frac{d}{d\boldsymbol{u}_{\text{int}}} q(\boldsymbol{u}_{\text{int}}) \approx \frac{1}{s} \left(\boldsymbol{g}(\boldsymbol{u}_{\text{int}} + s\boldsymbol{p}_{\text{int}}) - \boldsymbol{g}(\boldsymbol{u}_{\text{int}}) \right).$$

Construct a log-log plot of the discrepancy between the action of the Hessian on a random vector $\boldsymbol{p}_{\text{int}}$ and the corresponding finite difference, as a function of step size s ranging from $s = 10^{-13}$ to $s = 10^0$.

Problem 4: Comparison of optimization methods

In this problem, we solve the unconstrained optimization problem (1) with BFGS, the method of steepest descent, and the Newton-conjugate-gradient method. We estimate the convergence rate for all methods. In the jupyter notebook, we provide code to do these tasks for BFGS, as an example. You will repeat these tasks for steepest descent and Newton-CG.

For this problem, use $N = 32$, $L_0 = 3.0$, and $\alpha = 1e2$. A reasonable initial guess is the parabola

$$u^{(0)}(x) = -C\, x(1 - x)$$

with constant

$$C = 2\,(L_0 - 1).$$

We would like to choose the constant C to make $L\left(u^{(0)}\right)$ equal to L_0. But there's no need to exactly evaluate the integral in $L(u)$, since this is just an initial guess for the optimizer. Instead, we approximate $L\left(u^{(0)}\right)$ using the sum of the lengths of the left, right, and bottom edges of the box that bounds the parabola. This gives you the expression for C above.

a) Implement the method of steepest descent and use it to find the minimizer of $J(\boldsymbol{u}_{\text{int}})$. You may use any step length choice scheme that you want, but we recommend using the Wolfe linesearch conditions, as implemented in the *scipy.optimize.line_search* function. Terminate the iteration after the norm of the gradient has decreased by a factor of 10^6 from the initial norm of the gradient (i.e., $||g(\boldsymbol{u}_{\text{int}}^{(k)})|| \leq 10^{-6}||g(\boldsymbol{u}_{\text{int}}^{(0)})||$). Save the values of $J(\boldsymbol{u}_{\text{int}}^{(k)})$ for all iterations k so that we can study the convergence of the method.

b) Asymptotically, the error should decrease as

$$J(\boldsymbol{u}_{\text{int}}^{(k+1)}) - J(\boldsymbol{u}_{\text{int}}^{*}) \leq c\left(J(\boldsymbol{u}_{\text{int}}^{(k)}) - J(\boldsymbol{u}_{\text{int}}^{*})\right)^q$$

for some constant c and convergence rate q, where $\boldsymbol{u}_{\text{int}}^{*}$ is the solution to the optimization problem (1). Estimate the convergence rate, q, for steepest descent by plotting $J(\boldsymbol{u}_{\text{int}}^{(k+1)}) - J(\boldsymbol{u}_{\text{int}}^{*})$ vs. $J(\boldsymbol{u}_{\text{int}}^{(k)}) - J(\boldsymbol{u}_{\text{int}}^{*})$ on a log-log plot and finding the slope. You may use the final value of J as a proxy for the solution $J(\boldsymbol{u}_{\text{int}}^{*})$. Is the observed convergence rate consistent with the theoretical convergence rate for steepest descent?

c) Minimize J using the Newton-CG method. You may use the Newton-CG implementation in *scipy.optimize.minimize*, or any other existing implementation. (For your own sanity, please don't write Newton-CG from scratch!) Use a stopping tolerance of 10^{-6}. Save the values of $J(\boldsymbol{u}_{\text{int}}^{(k)})$ for all Newton (outer) iterations k so that we can study the convergence of the method.

d) Estimate the convergence rate, q, for Newton-CG by plotting $J(\boldsymbol{u}_{\text{int}}^{(k+1)}) - J(\boldsymbol{u}_{\text{int}}^{*})$ vs. $J(\boldsymbol{u}_{\text{int}}^{(k)}) - J(\boldsymbol{u}_{\text{int}}^{*})$ on a log-log plot and finding the slope. You may use the final value of J as a proxy for the solution $J(\boldsymbol{u}_{\text{int}}^{*})$. What does your observed convergence rate tell you about the choice of the forcing term η_k in scipy's implementation of Newton-CG? Comment on the efficiency of Newton-CG relative to steepest descent and BFGS.

Problem 5: Mesh independence (Extra credit)

Mesh independence (sometimes called dimension independence) is a property often enjoyed by Newton methods for infinite dimensional problems, but not by other methods. This concept refers to the relatively constant number of iterations taken by Newton's method as the mesh is refined.

a) Confirm that steepest descent does not exhibit mesh independence by solving the optimization problem (1) for a sequences of meshes from $N + 1 = 8, 16, 32, 64, 128$.

b) Use Newton-CG to solve (1) for a series of meshes from $N + 1 = 8, 16, 32, 64, 128, 256$. Can you give an explanation of what might be preventing Newton-CG from delivering mesh independence? Hint: plot some of the early iterates $\boldsymbol{u}_{\text{int}}^{(k)}$ and see if you can spot abnormalities. Can you suggest a fix so that we can observe mesh independence. [We have not discussed reasons for lack of mesh independence in class, so this will take some creative thinking.]