Assignment 6

Correction added Nov 6. See the **Values out of range** discussion in Exercise 5. The optimizer is unlikely to work if you don't do something about this.

1. If $u(x)$, with $x \in \mathbb{R}^d$, is of class[1] C^2, then, for any x,

$$u(x+y) = u(x) + \nabla u(x)^T y + \tfrac{1}{2} y^T H(x) y + O(\|y\|^3) \ .$$

   (a) Let H be a symmetric $d \times d$ matrix. We say that H is *positive definite* if $x^T H x > 0$ for any vector $x \neq 0$. Show that H is positive definite if and only if all the eigenvalues of H are positive. The matrix is positive *semi-definite* $x^T H x \geq 0$ for all x. Show that H is positive semi-definite if none of its eigenvalues is negative ("are" negative?).

   (b) Show that if $H(x_*)$ is non-singular and $\nabla u(x_*) = 0$ then x_* is a local minimizer of u if and only if H is positive definite. *Terminology.* An x_* with $\nabla u(x_*) = 0$ may be called a *stationary point* or a *critical point*. A stationary point is *non-degenerate* if $H(x_*)$ is non-singular.

   (c) Give an example of a stationary point where u has a positive semi-definite Hessian that is a local minimizer and a different example where it is not even a local minimizer. *Hint.* It may be easiest to do this in one dimension, so that the Hessian "matrix" is just the second derivative of u.

2. A function u, defined for all $x \in \mathbb{R}^d$, is *convex* if

$$u(\lambda x + (1-\lambda) y) \leq \lambda u(x) + (1-\lambda) u(y) \ \ , \quad \text{if } \ 0 \leq \lambda \leq 1 \ . \tag{1}$$

   A function is *strictly* convex if the inequality in (4) is strict whenever $0 < \lambda < 1$. Suppose a and b are two points in $\mathbb{R}^d$. The *line segment* defined by a and b is the set of points

$$x(t) = a + t(b-a) \ \ , \quad 0 \leq t \leq 1 \ .$$

   You can also think about the function u restricted to this segment

$$f_{[a,b]}(t) = u(x(t)) \ .$$

   This has $f_{[a,b]}(0) = u(a)$ and $f_{[a,b]}(1) = u(b)$.

[1] The C is for *continuous* and the 2 means up to second order. C^2 means that all partial derivatives up to second order exist and are continuous functions of x.

(a) Show that u is a convex function of x if and only if all the functions $f_{[a,b]}(t)$ is a convex function of t (for $0 \le t \le 1$ for every pair of endpoints a and b.

(b) Explain how to derive the formulas

$$f'_{[a,b]}(t) = \nabla u(x(t))^T (b-a)$$
$$f''_{[a,b]}(t) = (b-a)^T H(x(t))(b-a) \; .$$

(c) Show that u is convex if $H(x)$ is positive semi-definite for every x. Show that u is strictly convex if $H(x)$ is positive definite for every x. You may use the fact from calculus (draw pictures to verify) that a function $f(t)$ is convex if $f'' \ge 0$ for all t and strictly convex if $f''(t) > 0$ for all t.

(d) Show that u is not convex if there is an x where $H(x)$ has a negative eigenvalue. You may use the corresponding one dimensional fact that f is not convex if $f'' < 0$ somewhere.

3. This describes a gradient descent code you will need for Exercise 5. The gradient descent optimizer code should be in its own module `someName.py` that can be imported into the data fitting code of exercise 5. The gradient descent function should take as arguments, in some order,

   - An object `u` that has attributes `u.u` and `u.g`, which evaluate the loss function $u(x)$ and the gradient of the loss function $g(x) = \nabla u(x)$. The *Classes* section of the notes *Python for Smarties* explains how to do this.
   - The initial guess x_0.
   - The number of gradient descent steps n_s.

   It should return a tuple consisting of

   - A two index numpy `ndarray` that holds the iterates x_k for $k = 0, \cdots, n_s$. Of course, x_0 is the given initial guess.
   - A one index numpy `ndarray` that contains the $n_s + 1$ values of the loss function $u(x_k)$.

   A gradient descent step has the form

$$x_{k+1} = x_k - s_k \nabla u(x_k) \; . \tag{2}$$

   Here, s_k is the step size/learning rate. The *predicted decrease* (the amount u goes down) is

$$\Delta u_p = s_k \left\| \nabla u(x_k) \right\|_2^2 \tag{3}$$

   The *computed decrease* (the amount u actually decreases) is

$$\Delta u_c = u(x_k) - u(x_k - s_k \nabla u(x_k)) \; . \tag{4}$$

It might happen that the computed "decrease" is negative, which means that the loss function went up instead of down. Your code should have the following features:

- At each iteration, the initial guess for s_k (the step size for iteration k), should be the value used for iteration $k-1$.
- If the computed decrease is too small (or possibly negative), s_k is replaced by $\frac{1}{2}s_k$. "Too small" means $\Delta u_c < \alpha \Delta u_p$.
- If the predicted decrease is too close to the actual decrease, we take a more aggressive step, replacing s_k with $2s_k$. "Too close to the actual decrease" means $\Delta u_c > \beta \Delta u_p$.
- The step size reduction and step size expansion features cannot lead to an infinite reduction/expansion loop.
- In view of Exercise 4, it makes sense to take $\alpha < \frac{1}{2}$ and $\beta > \frac{1}{2}$. In order not to do too many evaluations of u at any one gradient descent step, it's probably best not to take α or β too close to $\frac{1}{2}$.

Test your gradient descent optimizer on the function

$$u(x_1, x_2) = x_1^2 + a x_2^2 \ .$$

Make a plot of $u(x_k)$ as a function of k. Try a moderate value of a (not very far from $a = 1$) and a more challenging value of a (either very small and positive or very large). Compare the convergence graphs to see that ill conditioned optimization problems are more challenging for gradient descent. It is important (for this last point) that the initial guess not be on one of the coordinate axes (why), so you might try initial guess $x_0 = (1, 1)$.

4. In the terminology of Exercise 3, show that if $u(x)$ is quadratic, then $\frac{1}{2}$ is the optimal relation between predicted and computed decrease. That means, if $u(x) = \frac{1}{2}x^T H x$ (and H is positive definite) the optimal step size

$$s_* = \arg\min_s u(x - s\nabla u(x))$$

has the feature that

$$\Delta u_c = \tfrac{1}{2}\Delta u_p \ .$$

5. This computational exercise uses gradient descent to do maximum likelihood fitting of a time series to a sum of simple oscillations. A simple oscillation has the form

$$x(t) = a\cos(\omega t) + b\sin(\omega t) \ .$$

We are concerned with a signal that is a finite sum of simple oscillations

$$x(t) = \sum_{k=1}^{n_f} a_k \cos(\omega_k t) + b_k \sin(\omega_k t) \ . \tag{5}$$

We work with measurements at times t_j that are corrupted by observation noise

$$X_j = x(t_j) + \epsilon_j \ . \tag{6}$$

We make the statistical hypothesis that the observation errors ϵ_j are independent Gaussians with mean zero and standard deviation σ. There are $3n_f + 1$ unknown parameters, the $3n_f$ numbers a_k, b_k, ω_k and the one extra parameter σ. The loss function is the negative of the log likelihood function. The likelihood function says that the ϵ_j are independent Gaussians with the same standard deviation

$$L = \prod_j \frac{1}{\sqrt{2\pi}\,\sigma} e^{-\frac{\epsilon^2}{2\sigma^2}} \ . \tag{7}$$

Thus, we seek to minimize[2]

$$u(\sigma, a_1, \cdots, a_{n_f}, b_1, \cdots, b_{n_f}, \omega_1, \cdots, \omega_{n_f}) = -\log(L) \ . \tag{8}$$

The loss function depends on the parameters because the signal $x(t)$ depends on the a_k, b_k, and ω_k (through (5)), the data X_j and the model $x(t_j)$ determine the observation residuals ϵ_j (through (6)), and σ enters into the likelihood function (7).

Download the module `modelFit.py` and extend it to find the maximum likelihood estimates of the parameters using the gradient descent module of Exercise 3. This module should

- Read data from a run whose name is given on the command line (see the writeup in *Assignment6Codes.pdf* for details).
- Define and instantiate a class that stores the data and uses it to evaluate the log likelihood function and its gradient.
- Creates and stores to a file a plot that contains the data and the function $\widehat{x}(t)$ that is given by (5) with the best fit values $\widehat{a}_k$, $\widehat{b}_k$, and $\widehat{\omega_k}$.
- If everything is automated, it should be easy to experiment with the number of iterations to see whether the fit stops improving. (Of course, it never stops improving completely, so iterate until the curve $\widehat{x}(t)$ stops changing).

Values out of range. It can happen (and usually does in real problems) that the loss/objective function $u(x)$ is not defined for all x. Even if the definition formula defines a function for all x, it often happens that the

[2]Coding tip: Evaluate $\ell = \log(L)$ as a sum rather than L as a product. If you evaluate the product (7), it will *overflow* (be larger than the largest floating point number) or *underflow* (be closer to zero than the smallest positive floating point number). Try it and see. Of course, maximizing ℓ is equivalent, in exact arithmetic, to maximizing L. In floating point, ℓ works and L does not.

result is not what the application calls for. Mathematically, we say that the optimization of u is to happen not over all x but only over some *feasible set*, $\Omega \subset \mathbb{R}^d$. An *infeasible* point $x \notin \Omega$ might be infeasible because u is not defined there, or because we want the "best" x only from the feasible set rather than for all x.

The log likelihood function of this exercise requires σ to be positive. The likelihood formula (7) makes sense (as a formula) if $\sigma < 0$ but this makes $L < 0$ so the log likelihood function (8) is not defined. The feasible set is

$$\Omega = \left\{ \sigma, a_1, \cdots, \omega_{n_f} \mid \sigma > 0 \right\} .$$

An optimization algorithm such as gradient descent may *propose* an infeasible point x_{k+1} even if x_k is feasible. The optimization algorithm must *reject* such a proposal if it is to find the desired optimal value $x_* \in \Omega$. The line search strategy of Exercise 3 gives a simple way to do this: The function `u.u` should return `inf` (the IEEE floating point standard "number" that is larger than any true floating point number) if x is not feasible. There are (at least) two ways to make `inf` in Python. They are equal because there is only one bit string in the IEEE floating point standard that represents `inf`.

```
coreInf =  float('inf')   #  in core Python
npInf   =  np.inf         #  "inf" is a name in the numpy
coreInf == npInf          #  returns True (for me, anyway)
x       >  coreInf        #  returns False if x is not infinite
x       <  coreInf        #  returns True if x is not infinite
(coreInf + x) == coreInf  #  returns True, inf + normal = inf
```

Your code for `u.u` will reject infeasible proposals if it includes something like

```
 if ( sigma <= 0. ):   # compare to floating point zero.
                       # sigma = 0. is bad too,
                       # only sigma > 0 is OK
    return np.inf
```

The line search part of the gradient descent algorithm will reject infeasible points because the computed "decrease" will be negative infinity and therefore guaranteed to satisfy the "is too small" inequality $\Delta u_c < \alpha \Delta u_p$. This triggers replacing s_k with $\frac{1}{2} s_k$, which will continue until s_k is small enough that $x_k - s_k \nabla u(x_k)$ is feasible.

This simple trick will not work if the optimum is on the boundary of the feasible set. The field of "constrained optimization" dedicated to this possibility. An optimal point not on the boundary is an *interior* optimizer. The loss/objective function of this Exercise has only interior (local) optimizers because $u \to \infty$ as x approaches the boundary of Ω. Here, the

boundary of Ω is $\sigma = 0$ and you are close to the boundary if σ is small. When $\sigma \to 0$, the exponent $-\frac{\epsilon^2}{2\sigma^2}$ in (7) goes to $-\infty$ so $\log(L) \to \infty$.

The behavior $u \to \infty$ as $\sigma \to 0$ is natural from a statistical fitting prospective. The residual ϵ is supposed represent observation error and is modeled as a Gaussian with mean zero and variance σ^2. The ratio ϵ/σ is the number of standard deviations that the observation is away from its mean. In the Gaussian distribution, it is "exponentially" unlikely that a sample is many standard deviations from the mean. Thus, a parameter combination with σ very small is a poor fit to the data.