Scientific Computing, Courant Institute, Fall 2021
http://www.math.nyu.edu/faculty/goodman/teaching/SciComp2021/index.html

Fourth assignment

updates. Exercise 1 reworded and corrected. The function in Exercise 2 corrected.

1. (*Affine invariance* is an important robustness property for multi-variate optimization and root-finding.) An *affine* change of variables in n dimensional space is given by $y = Ax + b$ with A an invertible $n \times n$ matrix and x and b in $\mathbb{R}^n$. A linear transformation is an affine transformation with $b = 0$. In high school algebra, a line $y = ax + b$ is an affine function (in the present terminology), and linear only if $b = 0$. We usually take $b = 0$, but call the transformations "affine" anyway. Affine changes of variable are used in practical optimization to improve the conditioning of the problem. In that context, they may be called *pre-conditioners*.

   Suppose $\overline{x} = \Phi(x, V)$ represents an optimization algorithm that takes a current iterate x_k and a function V and produces a new iterate $x_{k+1} = \Phi(x_k, V)$. An affine preconditioner $y = Ax$ changes the objective function $V(x)$ to $V_A(y)$. We want the function values to be equal, which is $V_A(y) = V(x)$, if $y = Ax$. This may be written either as $V_A(Ax) = V(x)$ or as $V_A(y) = V(A^{-1}y)$. An algorithm is *affine invariant* if it does the "same thing" in the x and y variable. More precisely, if $y = Ax$, then $\overline{y} = A\overline{x}$, or $y_k = Ax_k \implies y_{k+1} = Ax_{k+1}$. If you apply the iterate function Φ at the point $y = Ax$ using the objective function V_A, you should get the "same" iterate as if you applied the iterate function at x using the objective function V. Written more fully, this is

   $$A\Phi(x, V) = \Phi(Ax, V_A) \ .$$

   [The left side is $A\overline{x}$, and the right side is $\overline{y}$.] Proper mathematical terminology would call this affine "covariance", but people in the business say "invariant".

   (a) Show that gradient descent is not affine invariant.

   (b) Give an example where affine preconditioning improves the convergence rate of gradient descent a lot. (This is theory – show that the convergence is faster but don't code it. If your theory is right, your code would do what your theory predicts.)

   (c) Show that simple Newton iteration $x_{k+1} = H(x_k)^{-1}\nabla V(x_k)$ is affine invariant. Conclude that Newton's method does not need or benefit from affine preconditioning.

(d) As explained in class, there is a *modified Cholesky* version of Newton's method that uses $L\,|D|\,L^t$ in place of H in Newton's method. That is $x_{k+1} = x_k - \left(L\,|D|\,L^t\right)^{-1} \nabla V(x_k)$. Show that this method is invariant under diagonal scalings, which means invariant (covariant) in the sense above if A is a diagonal matrix.

2. (Exercise 6.3 from the book) Consider Newton's method for solving the equation $f(x) = 0$ with $f(x) = x/\sqrt{1+x^2}$. Show that, without safeguards, it gives $x_{k+1} = x_k^3$.

   (a) Show that unsafeguarded Newton's method converges to the solution if the initial guess satisfies $|x_0| < 1$ and fails to converge to the solution otherwise.

   (b) Explain the fact that the convergence is cubic rather than quadratic.

3. (nothing to hand in for this one) Do a web search on "newton's method mandelbrot set" and look at the pictures. These show that the basin of attraction of Newton's method can be very complicated.

4. Write an optimizer to minimize $f(x)$ using a safeguarded Newton's method. The optimizer should use functions (defined in other modules, probably) that supply $f(x)$, $\nabla f(x)$, and $H(x)$. The "bare" locally convergent method is

$$x_{k+1} = x_k - H(x_k)^{-1} \nabla f(x_k) \; .$$

   The two safeguards are

   - Descent direction using modified Cholesky. Use the `scipy` routine that computes the LDL^t composition (called `LDLH` because it's the hermitian conjugate of L, not jus the transpose). Replace the diagonals d_j with $|d_j|$ and call the resulting diagonal matrix $|D|$. Replace $H = LDL^t$ with $\widetilde{H} = L|D|L^t$.
   - Line search. The search direction is $p = -\widetilde{H}^{-1}\nabla f$. Do a binary search to find $s > 0$ with $f(x + sp) < f(x)$.

   Try your algorithm on some test functions starting from good and from poor initial guesses (you decide exactly how much of this to do). Try it with and without safeguards to see when the safeguards "rescue" a computation that would have failed otherwise. Make sure your code does not have infinite loops anywhere. Possible test functions include

   - $f(x) = \sqrt{x^2+1}$. This is 1D so you can see everything. With a poor initial guess, and without safeguards, it should fail because the function is nearly linear for large $|x|$.
   - $f(x,y) = e^{ax^2 + b(y - c\sin kx)^2}$. This function has parameters that make it easy or hard. In the hard case, with a small (but positive) and b and k large, the contour looks like a winding valley. Make a contour plot to see.

- $f(x_1, \ldots, x_n) = \displaystyle\sum_{k=1}^{n} x_k + \sum_{k=0}^{n} \left[(x_{k+1} - x_k)^2 + a(x_{k+1} - x_k)^4 \right].$

  The second sum on the right involves variables x_0 and x_{n+1} that are not part of f. Take $x_0 = 0$ and $x_{n+1} = 0$. You can think of $k = 1$ and $k = n$ as "boundaries". Then, $x_0 = 0$ and $x_{n+1} = 0$ are "boundary conditions". You can write the sum as

  $$x_1^2 + ax_1^4 + \sum_{k=1}^{n-1} \left[(x_{k+1} - x_k)^2 + a(x_{k+1} - x_k)^4 \right] + x_n^2 + ax_n^4 \; .$$

  This is typical of problems that arise in solving some differential equations. It's a high dimensional minimization problem, so if n is large the linear algebra will be expensive (don't tell Python it's tridiagonal). Get a feel for the solution by plotting x_k as a function of k.