CS205L Homework 9

Due: March 12th at 11:59 PM

- **Each question this week has a difficulty level (*,**,***) assigned to it based on how conceptually involved it is.** Each problem is graded on a $0, {}^1/_4, {}^1/_2, {}^3/_4, 1$ point scale. We recommend starting the single-starred questions first if you're unfamiliar with the concepts.
- You may work in groups up to 4 people. You should
  - individually turn in your own unique-as-possible write-up, and
  - list your collaborators' SUIDs in the spaces provided below.
    (A gentle reminder that failure to do so would be a violation of the honor code.)
- Submit your homeworks on Gradescope by the deadline (course code 9DYYZX).
- **You will have 5 late days in total** to use on the written homeworks this quarter, and are allowed to use multiple late days per homework. If you've used up your late days, you may turn in your homework late with a flat 50% penalty.
- You may submit either typed-up solutions or legible scans of handwritten solutions. The LaTeX source is provided separately so you can use it if you choose to.
- Each problem has a TA assigned to it, and the assigned TA is labelled next to each problem. You may want to read the suggest readings before attempting the question.
- **For each homework-related question, attend the office hours of the assigned TA**.
  TAs will not handle homework-specific questions on problems other than their own.
  Additionally, if you post questions on Piazza about the assignments, please take care to not post parts of your solutions on public posts!
- **You may submit solutions to more than 4 questions, but please specify which solutions you'd like to count towards your homework grade below.** If you don't specify which questions you'd like to be graded for points, we will use the first 4 non-blank questions. Please also tag any/all questions on Gradescope that you'd like to received feedback on.

In the box below, place an X on questions you would like to count towards your grade. Do not select more than 4.

Q1	Q2	Q3	Q4	Q5	Q6

Name: _____

SUID: _____

Collaborators: _____

(SUID) _____

_____

Problem 9.1. (*) Quasi-Newton methods: The Good Broyden's Method

In class, we recently discussed quasi-Newton methods for solving non-linear systems and optimization. Of these methods, we introduced **Broyden's Method** specifically as a popular example. Skimming over the Unit 18 notes and Heath Chapter 5.6.3 Secant Updating Methods, you might notice that a step in the main loop of Broyden's Method involves a linear solve:

```
1: function Broyden's Method
2:     ...
3:     for k ← 0, 1, 2, .... do
4:         Solve B_k s_k = −f(x_k) for s_k
5:         ...
```

Note: For consistency, I'll be using Heath's notation from the textbook.
As you might expect, this linear solve takes up the bulk of the work of the algorithm. Of course, as computer scientists, we're always interested in making our algorithms as fast as possible. And so we have to wonder whether we can optimize Broyden's Method to avoid a possibly demanding solve each iteration. The answer is: we can.

1. To optimize Broyden's Method, we need to first add a useful formula from Linear Algebra known as the **Sherman-Morrison** formula into our toolkit. For an invertible matrix $A \in \mathbb{R}^{n \times n}$ and vectors $\vec{u}, \vec{v} \in \mathbb{R}^n$, we have:

$$(A + \vec{u}\vec{v}^T)^{-1} = A^{-1} - \frac{A^{-1}\vec{u}\vec{v}^T A^{-1}}{1 + \vec{v}^T A^{-1}\vec{u}} \tag{1}$$

   Verify this formula.
   Hint: A matrix and its inverse must multiply to I, right? Also, remember that matrix multiplication is not commutative.

Notice the update rule for the Jacobian matrix in the Broyden's Method pseudocode:

$$B_{k+1} = B_k + ((\vec{y_k} - B_k\vec{s_k})\vec{s_k}^T)/(\vec{s_k}^T\vec{s_k}) \tag{2}$$

Defining appropriate vectors $\vec{u}, \vec{v}$, we might be able to express the right-hand side of Equation 2 as $B_k + \vec{u}\vec{v}^T$. This would allow us write an equation for the inverse Jacobian matrix using the Sherman-Morrison formula.

2. Apply the Sherman-Morrison formula in Equation 1 to the Jacobian matrix in Equation 2 to find the following equation for the inverse Jacobian matrix:

$$B_{k+1}^{-1} = B_k^{-1} + \frac{\vec{s_k} - B_k^{-1}\vec{y_k}}{\vec{s_k}^T B_k^{-1}\vec{y_k}}\vec{s_k}^T B_k^{-1} \tag{3}$$

Equation 3 gives us a formula to approximate the inverse Jacobian matrix directly, which in turn gives us a way to avoid the linear solve mentioned above.

3. Use Equation 3 to modify Broyden's Method so that it no longer needs to perform a linear solve each iteration. Rather, the algorithm should just consist of simple addition, subtraction, and multiplication operations involving matrices and vectors. This version of Broyden's Method is often called the **"good Broyden's Method."**

   Hint: With an equation to approximate the inverse Jacobian matrix, do you really need to still *solve* for $\vec{s_k}$?

4. How does the $\mathcal{O}(n)$ complexity of the good Broyden's Method compare to that of the original Broyden's Method? Can you think of a reason why we might prefer matrix times matrix multiplication operations over a linear solve with today's technology? This second part is open-ended – any reasonable answer is fine.

CA:Yilin
Reading: Unit 18 notes and Heath 5.6.3 Secant Updating Methods

Your Solution.

Your Solution.

Problem 9.2. (***) Quasi-Newton Methods for Optimization - BFGS

Background/Overview
To minimize an objective function $F(\vec{c})$, we repeat the following steps at each iteration:

1. Use current Jacobian $\hat{J}(\vec{c}^k)$ and Hessian $\hat{H}(\vec{c}^k)$ to find the search direction via $\hat{H}(\vec{c}^k)\overrightarrow{\Delta c} = -\hat{J}(\vec{c}^k)$ or $\overrightarrow{\Delta c} = -\hat{H}^{-1}(\vec{c}^k)\hat{J}(\vec{c}^k)$.
2. Use your favorite line search method to determine the step size α and obtain the next estimate $\vec{c}^{k+1} = \vec{c}^k + \alpha\overrightarrow{\Delta c}$.
3. Obtain the next Jacobian and Hessian $\hat{J}(\vec{c}^{k+1})$ and $\hat{H}(\vec{c}^{k+1})$.

In the last step, there are many choices for obtaining the Hessian $\hat{H}$:

- Compute the Hessian exactly – *Newton's method*
- Approximate the Hessian via secant-style methods – *Quasi-Newton methods*
- Give up and use the identity matrix – *Gradient/steepest descent method*

In class, we have learned Newton's method and gradient descent method, and briefly discussed quasi-Newton methods such as Broyden's method. In this problem, we will look at another popular quasi-Newton method: **BFGS** (Broyden-Fletcher-Goldfarb-Shanno).

Notation
To simplify notation, let's define the following shorthand:

$\hat{H}_k = \hat{H}(\vec{c}^k), \quad \hat{H}_{k+1} = \hat{H}(\vec{c}^{k+1})$	the Hessians $\hat{H}$
$\vec{x}_k = \vec{c}^{k+1} - \vec{c}^k = \alpha\overrightarrow{\Delta c}$	change in the estimate $\vec{c}$
$\vec{y}_k = \hat{J}(\vec{c}^{k+1}) - \hat{J}(\vec{c}^k)$	change in the Jacobian $\hat{J}$

Intuition
Quasi-Newton methods generally follow a **secant-style approach** to successively update the estimated Hessian based on the current iteration. Intuitively, the idea is to approximate the Hessian by looking only at the change in the Jacobian in the most recent update direction. That is, one tries to approximate the Hessian such that it satisfies

$$\hat{H}_{k+1}\vec{x}_k = \vec{y}_k. \tag{4}$$

Very informally, one can think of this as $\hat{H}_{k+1} \approx \vec{y}_k/\vec{x}_k$. But to be clear, one would never actually write something like $\vec{y}_k/\vec{x}_k$ for vectors $\vec{x}_k, \vec{y}_k$ in practice, because it does not have clear mathematical meaning; here we are using this to give an analogy to the secant method.

BFGS

For BFGS, the update rule is

$$\hat{H}_{k+1} = \hat{H}_k + \frac{\vec{y}_k \vec{y}_k^T}{\vec{y}_k^T \vec{x}_k} - \frac{\hat{H}_k \vec{x}_k \vec{x}_k^T \hat{H}_k^T}{\vec{x}_k^T \hat{H}_k \vec{x}_k}. \tag{5}$$

1. Secant-style update. First, prove that the update rule in Equation 5 satisfies Equation 4.
   Hint: $u^T A u = u \cdot (Au) = (Au) \cdot u = (Au)^T u = u^T A^T u$ for square matrix A.

Now that you have convinced yourself that BFGS indeed follows the secant-style update rule, let's look at some of its properties. The idea is that **quasi-Newton methods usually try to maintain certain "good" properties of the Hessian matrix during the update, such as symmetry and positive definitenss**. Let's prove that for BFGS. Make sure to show your work.

2. Symmetry. Prove that if $\hat{H}_k$ is symmetric, then $\hat{H}_{k+1}$ is also symmetric.
   Hint: First, show that 1st term + 3rd term $\geqslant 0$ (Cauchy-Schwarz), and 2nd term $\geqslant 0$. Then show that if 1st term + 3rd term $= 0$, then 2nd term > 0.

3. Positive definiteness. Assume that $\vec{x}_k$ satisfies $\vec{y}_k^T \vec{x}_k = \vec{y}_k \cdot \vec{x}_k > 0$ (one can check/enforce this in line search). Prove that if $\hat{H}_k$ is symmetric and positive definite, then $\hat{H}_{k+1}$ is also positive definite.

   hint 1: Recall the definition that A is positive definite if $\vec{v}^T A \vec{v} > 0$ for all $\vec{v} \neq 0$.
   hint 2: You may find the Cauchy-Schwarz inequality useful. For vectors $\vec{a}, \vec{b}$ and SPD matrix P, one has $(\vec{a}^T P \vec{b})^2 \leq (\vec{a}^T P \vec{a})(\vec{b}^T P \vec{b})$, and equality holds only if $\vec{b} = \lambda \vec{a}$ with $\lambda \in \mathbb{R}$, i.e., $\vec{b}$ is a multiple of $\vec{a}$.

To more easily compute the search direction, quasi-Newton methods also keep track of the inverse Hessian. This can be done efficiently since Equation 5 defines a **low rank update**, and thus can be inverted using the Sherman-Morrison-Woodbury formula.

4. Inverse. Show that Equation 5 performs two symmetric rank-one updates in the form of $\hat{H}_{k+1} = \hat{H}_k + \gamma \vec{u} \vec{u}^T + \delta \vec{w} \vec{w}^T$. Write down the scalar coefficients γ, δ and the vectors $\vec{u}, \vec{w}$.

For completeness, the inverse update rule for BFGS is given as:

$$\hat{H}_{k+1}^{-1} = (I - \frac{\vec{x}_k \vec{y}_k^T}{\vec{y}_k^T \vec{x}_k}) \hat{H}_k^{-1} (I - \frac{\vec{y}_k \vec{x}_k^T}{\vec{y}_k^T \vec{x}_k}) + \frac{\vec{x}_k \vec{x}_k^T}{\vec{y}_k^T \vec{x}_k}. \tag{6}$$

CA:Yilin

Reading: Unit 18 notes and Heath 5.6.3 Secant Updating Methods

Your Solution.

Your Solution.

Problem 9.3. (**) Evaluating Gradient Descent Using Condition Number
Recall that the condition number of a matrix A is defined as

$$\kappa = \max_{i,j} \left| \frac{\lambda_i}{\lambda_j} \right|.$$

where λ_i, λ_j are eigenvalues of A. This value is powerful because it measures how sensitive a function is to changes or noise in the input, and thus how much error in the output is attributed to the input error. A problem is ill-conditioned (e.g. large κ) if small changes in the input data lead to large changes in the output.

Early on in this class, we saw how condition number can help guide our choice of basis in polynomial interpolation. In this problem, we will see how the condition number of the Hessian matrix in gradient descent can be used to determine a lower bound on the optimal learning rate.

Consider a function with quadratic form $f(x) = \frac{1}{2}x^T Ax - b^T x + c$ where $f : \mathbb{R}^n \to \mathbb{R}$, $x \in \mathbb{R}^n$, and $A \in \mathbb{R}^{n \times n}$. We want to solve for the optimal value x^* that minimizes this function by setting $\nabla f(x) = Ax - b = 0$ or equivalently $Ax = b$. Recall that the solution to this equation is *guaranteed* to be a local minimum only if A is SPD. **Assume A is SPD in this problem.**

It follows that the update rule for gradient descent with learning rate α is

$$x^{k+1} = x^k - \alpha \nabla f(x^k).$$

We want to choose a learning rate $\alpha \geq 0$ that will allow our algorithm to converge efficiently, but without overshooting the minimum x^*. How can we do this?

1. The first step is to understand how the error at each step of gradient descent relates to the eigenvalues of A.

   (a) Recall that if A is symmetric, it has an eigenvalue decomposition

   $$A = Q\Sigma Q^T$$

   where Q is orthogonal and Σ is a diagonal matrix containing the eigenvalues λ_i of A. With this in mind, re-write the gradient descent update rule in terms of this decomposition.
   Hint: Your equation should only contain $x^{k+1}, x^k, \alpha, \Sigma, Q$, and b.

   (b) Since we are only interested in the eigenvalues λ_i and not Q, we can define a latent variable z^k where

   $$z^k = Q^T(x^k - x^*).$$

   Write the update rule for each z_i^{k+1}, e.g. each entry of the vector z^{k+1}. Notice how this new equation is only dependent on α and λ_i!

   (c) Modify your equation in Part 1b to depend not on z_i^k but z_i^0 where z^0 represents the initial guess.

(d) Finally, use Parts 1a-c to derive an expression for the error e^k at the kth iteration where $e^k = x^k - x^*$. You may use the variables z^0, α, λ_i, and q_i, e.g. the ith column of Q.

2. Using the derivations from Part 1, we can obtain an expression for the function error:

$$f(x^k) - f(x^*) = \sum_{i=1}^{n} (1 - \alpha\lambda_i)^{2k} \lambda_i (z_i^0)^2$$

This leads to the conclusion (after further analysis) that in order for gradient descent to converge, each $|1 - \alpha\lambda_i|$ must be strictly less than 1. Using this constraint, state a lower and upper bound for $\alpha\lambda_i$.

3. Since convergence is determined by the slowest error component, the convergence rate is defined as

$$\text{rate}(\alpha) = \max\{|1 - \alpha\lambda_1|, |1 - \alpha\lambda_n|\}$$

where λ_1, λ_n are the smallest and largest eigenvalues, respectively. Therefore, we want to minimize this rate, which occurs when the rates corresponding to λ_1 and λ_n are equal. After some further analysis (again), we can obtain an expression for this optimal rate:

$$\min_{\alpha} \text{rate}(\alpha) = \frac{\lambda_n - \lambda_1}{\lambda_n + \lambda_1} \tag{7}$$

(a) What are the values of λ_1, λ_n that minimize Equation 7? You should be able to answer this by inspection.

(b) Write Equation 7 in terms of the condition number κ. To summarize, we have shown that κ is a direct measure of how fast gradient descent can converge given we choose an optimal value for the learning rate.

CA:Jane
Recommended Readings: Unit 1, 3, and 19 notes

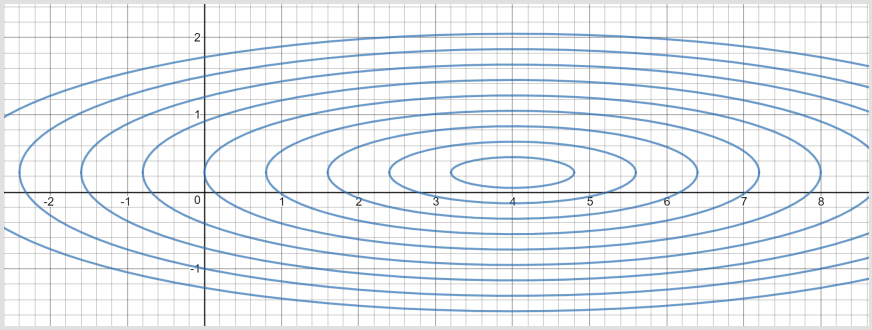
Your Solution.

Your Solution.

Problem 9.4. (*) k-th order Runge Kutta. In class and section this week, we talked about various methods for solving ODEs numerically. In particular, Runge-Kutta methods can give you higher order accuracy without explicitly computing higher order derivatives. Let's consider the function

$$F(\vec{c}_{\in\mathbb{R}^2}) = \frac{1}{8}(c_1 - 4)^2 + 2(c_2 - \frac{1}{4})^2.$$

We've plotted out some isocontours of this function below, i.e. a set of curves $\{C_\gamma\}$ in the 2D plane where $C_\gamma = \{(c_1, c_2) : F((c_1, c_2)^T) = \gamma\}$.



1. Ron mentioned in class that when we want to minimize some function F, we can formulate the gradient descent algorithm for this problem in terms of a gradient flow ODE, with $\frac{d\vec{c}}{dt} = -\nabla_{\vec{c}} F(\vec{c})$. Write out the gradient flow ODE that corresponds to performing gradient descent on $F(\vec{c})$ as defined above.
   (Reminder: $\nabla_{\vec{c}} F = \left[\frac{\partial F}{\partial c_1}, \frac{\partial F}{\partial c_2}, \ldots, \frac{\partial F}{\partial c_n}\right]^T = J_F^T$ for a scalar function F.

Recall from HW1.2 that one straightforward way to numerically evaluate the initial value problem $\frac{dy(t)}{dt} = f(t, y)$ with $y_0 = y(t_0)$ and step size h is via iteratively solving $y_{n+1} = y_n + h \cdot f(t_n, y_n)$– known as Euler's method, also known as the first order Runge Kutta Method (RK1). The second order RK method (RK2) is

$$y_{n+1} = y_n + h(\frac{k_1}{2} + \frac{k_2}{2}) \text{ with}$$
$$k_1 = f(t_n, y_n)\ ,\ k_2 = f(t_k + h, y_n + hk_1),$$

And the 4th order RK method (RK4) is

$$y_{n+1} = y_n + h(\frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}) \text{ with}$$
$$k_1 = f(t_n, y_n)\ ,\ k_2 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_1)\ ,$$
$$k_3 = f(t_n + \frac{h}{2}, y_n + \frac{h}{2}k_2)\ ,\ k_4 = f(t_n + h, y_n + hk_3).$$

2. Evaluate RK1,2,4 for 1 step, on $f(t, \vec{c}(t)) = -\nabla F(\vec{c})$ as defined above. Start at $\vec{c}_0 = (0, 0)$ with step size $h = 0.5$. Show k_i's for RK2,4 and $\vec{c}_1$ for RK1,2,4. If you decide to use a calculator, your solution should be within 5% of the answer evaluated in terms of reduced fractions (so don't truncate prematurely!)

3. In the isocontour plot above, sketch out the vector going from y_0 to y_1 for RK1,2,4. It doesn't have to be exact, but should be in roughly the correct 1/5 by 1/5 subgrid square.

CA:Winnie

Readings: Unit 20 notes and Heath 9.3.5,9.3.6 [Numerical Solutions of ODEs]

Your Solution.

Your Solution.

Problem 9.5. (*) Stability and Error of ODEs
In this question, we look at a high level review of ODE methods and how they compare against each other. Suppose we had some system of ODEs defined as

$$\vec{c'} = e^{-t}A\vec{c}$$

for some matrix A.

1. With the given ODE above, explicitly write out the update rule for Forward Euler, Backward Euler, and Trapezoidal Rule.
2. List one advantage and one disadvantage each for the three numerical schemes you provided in the previous question.

CA:Winnie

Your Solution.

Problem 9.6. (*) A Review on Optimizers

In this question, we will look at the optimization methods: gradient descent, momentum method, and Adagrad.

We are given a least square problem, which is fitting a line $y = w_1 x + w_2$ to data $x, y \in \mathbb{R}^n$. $(x = (x_1, x_2, ..., x_n), y = (y_1, y_2, ..., y_n))$. We want to minimize the error term $L(w) = \sum_i (w_1 x_i + w_2 - y_i)^2$.

1. If we use gradient descent method to minimize the error, we have $\begin{bmatrix} w_{t+1,1} \\ w_{t+1,2} \end{bmatrix} = \begin{bmatrix} w_{t,1} \\ w_{t,2} \end{bmatrix} - \eta \begin{bmatrix} \frac{\partial L}{\partial w_1} \\ \frac{\partial L}{\partial w_2} \end{bmatrix}$. Write out $\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}$ in terms of x_i, y_i and finish the update rule.

2. Recalled from class, momentum method is adding last gradient to the current gradient. Rewrite the update rule of momentum method with $\gamma = 0.9$. (You can write with $\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}$).

3. Adagrad is an adaptive learning rate method, which adjusts the learning rate for each parameter. A reminder: learning rate means step size of the optimization method in the problem.
   We denote $g_{t,i} = \frac{\partial L(w)}{\partial w_i}$ at step t. The update rule for Adagrad is:

$$w_{t+1,i} = w_{t,i} + \frac{\eta}{\sqrt{G_{t,i} + \epsilon}} g_{t,i}, \text{ where } G_{t,i} = \sum_t g_{t,i}^2$$

   The ϵ is a small number to avoid division by 0 if $G_{t,i} = 0$. Adagrad multiplies $\frac{1}{\sqrt{G_{t,i}+\epsilon}}$ to the learning rate, so that the method is more robust with parameters w_i with sparse and small gradients. Give reasoning about how the learning rate is different for parameters with large gradients and sparse/small gradients.

   A Side Note: Adaptive learning rate is very useful when the features and the parameters are in high dimensional space and some features are more deterministic but with sparse gradient.

CA:Yilin

Your Solution.

Your Solution.