Introduction and basic linear algebra reviews

Computing resources: Hypergator (Caleb's slides) (`https://caleb-huo.github.io/teaching/2021FALL/lectures/Week4_ggplot2/HiperGator.html#(1)`).

Software dissemination: GitHub and Rpackge (Caleb's slides`https://caleb-huo.github.io/teaching/2021FALL/lectures/Week6_RPackage/Rpackage.html#(1)`,`https://caleb-huo.github.io/teaching/2021FALL/lectures/Week7_GitHub/github.html#(1)`). I would personally recommend using Github desktop. It works just like Dropbox and so easy to use.

The latest edition of his course is `https://caleb-huo.github.io/teaching/2023FALL/ProgrammingBasics.html`.
Not today, but eventually I would go over Caleb's slides.

1 Requirements

- Working knowledge of linear algebra.
- Working knowledge of calculus.
- Working knowledge of some programming language like R, Python, C++, etc.

2 Computing stuffs

- For R: Rstudio, Rcpp, Rcpparmadillo (which uses excellent Cpp linear algebra library `http://arma.sourceforge.net/docs.html#linspace`)
- For Python: Jupyter notebook, Posit.

3 Recommended books and notes

John Monahan, Numerical Methods of Statistics, 2nd Edition, Cambridge University Press (2011)
Kenneth Lange, Numerical Analysis for Statisticians, 2nd Edition, Springer (2010)
James Gentle, Computational Statistics, Springer (2009)
https://www.stat.cmu.edu/ ryantibs/convexopt/
An Introduction to Statistical Learning `https://www.statlearning.com///`

Purpose of this course is teaching algorithms, coding tricks etc. as well as letting know about different resources available online. 'Don't end up as just an end user of certain software/models throughout your life, try to learn the algorithms/theory behind them to become innovators and contributors.'//

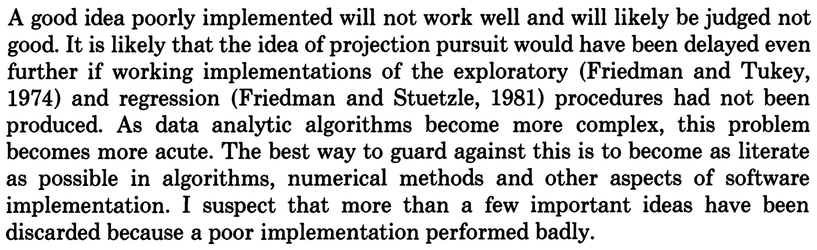
How statistical analysis work?

$$\begin{tikzcd}
\boxed{\text{Conceptualization of the data generation mechanism}} \arrow[d] \\
\boxed{\text{Optimization problem for parameter estimation}}
\end{tikzcd}$$

4 Why learning computational algorithms are important?

Statistics is NOT mathematic NOR coding. It is a combination of the two.

From Jerry Friedman's discussion of Peter Huber's 1985 projection pursuit paper, in Annals of Statistics: Take home message: You need to provide a good implementation of



A good idea poorly implemented will not work well and will likely be judged not good. It is likely that the idea of projection pursuit would have been delayed even further if working implementations of the exploratory (Friedman and Tukey, 1974) and regression (Friedman and Stuetzle, 1981) procedures had not been produced. As data analytic algorithms become more complex, this problem becomes more acute. The best way to guard against this is to become as literate as possible in algorithms, numerical methods and other aspects of software implementation. I suspect that more than a few important ideas have been discarded because a poor implementation performed badly.

your method to ensure its better reach to the broader scientific community.

In addition, Let our statistical problem is defined as $P : \min_{\boldsymbol{\beta} \in D} f(\boldsymbol{\beta})$ [for example P could be a linear regression; then $f(\beta) = \|\mathbf{y} - \mathbf{X}\boldsymbol{\beta}\|_2^2$, where $\|\mathbf{a}\|_2^2 = \sum_{i=1}^{p} a_i^2$ and $\mathbf{a} = (a_1, \ldots, a_p)$; D represents the space of allowed values. In general, for the regression problem $D = \mathbb{R}$. But for non-negative regression $D = \mathbb{R}_+$;] Knowing how different computation algorithms work will tell us:

- Different algorithms can perform better or worse for different problems P (sometimes drastically so).[Like in high dimension, we need to consider penalty.]
- Knowledge of algorithms also helps to understand when and why a package fails to produce output.
- Studying P through an optimization lens can actually give you a deeper understanding of the task/procedure at hand.

- Knowledge of optimization can actually help you create a new problem P that is even more interesting/useful. (e.g. Dantzig selector [1] for variable selection.)

Check out this page on No free lunch theorem for optimization: `https://en.wikipedia.org/wiki/No_free_lunch_in_search_and_optimization` **or this paper** `https://ieeexplore.ieee.org/document/585893`

As an example, cars used to have gears to be changed manually. We now get cars with automatic gear transmission. It indeed uses an algorithm to change the gear optimally. If you know that, you might be able to use this technology more efficiently, thereby improving the cars life-cycle.

In my opinion, learning computational methods is as important as learning large-sample theory results. 1) There are two routes one can take to justify the utility of any statistical method, either by proving some cool theoretical properties or establishing computational efficiency (computational time and/or accuracy) over other methods. 2) One can also verify complicated theoretical results using computational methods.

5 Vectorization of code

High level languages like R, Python are slow in execution of loops. However, these programming languages are designed to help programmers writing short and portable codes. They thus support vectorized calculations. If you have two vectors $\mathbf{a} = (1, 2), \mathbf{b} = (3, 5)$ and you want to sum them element-wise to get $\mathbf{c} = (4, 7)$. In C, you cannot perform operations like $\mathbf{c} = \mathbf{a} + \mathbf{b}$. Instead, you need to run a loop. However, R, Python and even C++ (using some advanced libraries) will allow you to do this calculation directly without writing any loop. But in the background, they indeed use some C or Fortran implementations that sometimes use loop or other complex parallel processing. However, at the user front, you do not need to write any explicit loop or complex parallelization of code. This makes your code concise and more readable. It then looks like how you would write the associated math operations on a paper in the first place.

Throughout the course, we will try to avoid loops whenever possible to speed up the computation. Now, writing loop is easy, but vectorization can be difficult for some cases. We thus need to learn matrix algebra and use some results to accomplish this step. Here are some examples of vectorization:

```
a= rand(1,4);
b= rand(1,4);
c= a + b;
```

```
a= rand(1,4);
b= rand(1,4);
for k= 1:length(a)
    c(k)= a(k) + b(k);
end
```

Even in C:

```
 for (i = 0; i < n; i++){
     a[i] = b[i] + c[i];
 }
```

Will run slower than

```
for (i = 0; i < n; i+=4){
        a[i] = b[i] + c[i];
        a[i+1] = b[i+1] + c[i+1];
        a[i+2] = b[i+2] + c[i+2];
        a[i+3] = b[i+3] + c[i+3];
    }
```

C++ has several automatic vectorization methods, which are implemented or used in several packages and modules in R and Python. For our daily coding, we just need to be mindful to use those efficiently. However, for your own implementation of any algorithm, you might need to build the vectorization manually from ground up. They are in general a bit complicated. Alternative direction will be using linear algebra. This is what separates us from the CS-folks. CS-folks (not theoretical CS, theoretical CS people know more math than stat/biostat people.) do less math and more compiler coding. We are however able to do more math-based simplifications than compiler coding.

'Vectorization of code' is synonymous with 'avoiding loop'.

6 Linear algebra

Book on basic linear algebra tricks for statisticians: `https://link.springer.com/book/10.1007/978-3-642-10473-2`. Let me know if you don't get it.

Also another one in `https://www.cs.cmu.edu/~zkolter/course/linalg/index.html`.

One way to better conceptualize linear algebra methods is to understand a commonality that its topics provide tools to figure out the solution space of equations $\mathbf{Ax} = \mathbf{b}$ and $\mathbf{Ax} = \mathbf{0}$.

We come across the problem of solving $\mathbf{Ax} = \mathbf{b}$ or $\mathbf{Ax} = \mathbf{0}$ all the time in statistics. Linear algebra methods are like 'magic tools' that allow us to solve them smoothly on a case-by-case basis.

This is the most important piece to write efficient codes. The Linear Algebra include matrix operations, determinants, systems of linear equations, eigendecomposition, singular value decomposition (SVD), etc. We will discuss some basic calculations today and solve some in the HW. It is hard to compile all the linear algebra results. (The key thing to remember is that 'Element-wise operations' in high level languages are usually parallelized in the background. So, if we can convert any operation to element-wise operations, our life will become simpler.)

Linear algebra specifically deals with matrices. A matrix $\mathbf{M}$ has two dimension. If a matrix has only one dimension, it is called 1D array or just array. If we go beyond two dimension, we again call those arrays. Like 3D array, or 4D array, etc. So, a Matrix is a special type of array which has two dimensions. Arrays are sometimes called vectors as well. I do not like that, as the terminology 'vector' is expected to be associated with some direction. 'Array' is more general. Like if we use an array to described demographics of a subject like (age, sex, height), this array does not have any notion of direction. However, most of the programming languages consider those as a similar data type. When you have a matrix, the rows and columns have separate alignment. To denote 'by row' alignment, we use row vector and to denote 'by column' alignment, we use column vector.

We use a (small) to denote scalars, $\mathbf{a}$ (bold small) for arrays and $\mathbf{A}$ (bold capital) for matrix. In R, it always stores arrays as column vector. Hence, R follows column-major order like Fortran, MATLAB, GNU Octave, S-Plus, Julia, Scilab, and Rasdaman. ROw-major is used in C/C++, Python's NumPy library. What this means is that if nothing is mentioned specifically, the associated compiler will run its operation by column if it's column major or by row if it's row major. Quick example (in R):

```
A <- matrix(1:6, 3, 2)

A[1, 2]
A[4] #I am trying to acess the 4-th entry although its a matrix.
     #The compiler will then assume A as an array by stacking its entries column-wise.
```

Examples of different data types:

```
a <- 2 #scalar
a <- c(1,3,4) #vector/array and it's always stored as a column vector
A <- matrix(c(1,3,4, 2, -1, 0), 3, 2)
```

Significance of the second line will be discussed later.

Multiplication of two scalars is widely different from multiplication of two arrays, two matrices or matrix-vector. The last three are very similar and follows the same principal. (I will not provide any list of functions. As we start coding, we will learn those automatically.)

6.1 Some basic matrix notations:

$\mathbf{A}^T$ stands for transpose of a matrix $\mathbf{A}$. trace($\mathbf{A}$) is the sum of the diagonal entries of $\mathbf{A}$.

Properties:

- $$\begin{aligned} \operatorname{tr}(\mathbf{A}+\mathbf{B}) &= \operatorname{tr}(\mathbf{A}) + \operatorname{tr}(\mathbf{B}) \\ \operatorname{tr}(c\mathbf{A}) &= c\operatorname{tr}(\mathbf{A}) \end{aligned}.$$
- $\operatorname{tr}(\mathbf{A}) = \operatorname{tr}\left(\mathbf{A}^\mathsf{T}\right)$.
- $\operatorname{tr}\left(\mathbf{A}^\mathsf{T}\mathbf{B}\right) = \operatorname{tr}\left(\mathbf{A}\mathbf{B}^\mathsf{T}\right) = \operatorname{tr}\left(\mathbf{B}^\mathsf{T}\mathbf{A}\right) = \operatorname{tr}\left(\mathbf{B}\mathbf{A}^\mathsf{T}\right) = \sum_{i=1}^m \sum_{j=1}^n a_{ij}b_{ij}$.
- **Most important:** The trace is invariant under cyclic permutations, that is, $\operatorname{tr}(\mathbf{ABCD}) = \operatorname{tr}(\mathbf{BCDA}) = \operatorname{tr}(\mathbf{CDAB}) = \operatorname{tr}(\mathbf{DABC})$. Here the example is for four matrices, however it holds for any number of matrices, even two.

Linear Algebra Operations

$\mathbf{A}^+$	Moore-Penrose pseudoinverse of $\mathbf{A}$
$\mathbf{A} \odot \mathbf{B}$	Element-wise (Hadamard) product of $\mathbf{A}$ and $\mathbf{B}$
$\det(\mathbf{A})$	Determinant of $\mathbf{A}$

6.2 Multiplication

For scalar numbers, there is only type of multiplication.

For matrices, there are several notions of multiplication. By matrix multiplication, we usually mean the following. If A is an $m \times p$ matrix and B is an $p \times n$ matrix,
$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1p} \\ a_{21} & a_{22} & \cdots & a_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mp} \end{pmatrix}$, $\mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{p1} & b_{p2} & \cdots & b_{pn} \end{pmatrix}$ the matrix product $C = AB$ (denoted without multiplication signs or dots) is defined to be the $m \times p$ matrix.

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} \end{pmatrix} \text{ such that}$$

$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{ip}b_{pj} = \sum_{k=1}^{p} a_{ik}b_{kj}$, for $i = 1, \ldots, m$ and $j = 1, \ldots, n$.

There are two other types, namely Kronecker's product and Hadamard product.

Kronecker's product: If A is an $m\times n$ matrix and B is a $p\times q$ matrix, then the Kronecker product $A \otimes B$is the $pm \times qn$ block matrix:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & \cdots & a_{11}b_{1q} & \cdots & \cdots & a_{1n}b_{11} & a_{1n}b_{12} & \cdots & a_{1n}b_{1q} \\ a_{11}b_{21} & a_{11}b_{22} & \cdots & a_{11}b_{2q} & \cdots & \cdots & a_{1n}b_{21} & a_{1n}b_{22} & \cdots & a_{1n}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{11}b_{p1} & a_{11}b_{p2} & \cdots & a_{11}b_{pq} & \cdots & \cdots & a_{1n}b_{p1} & a_{1n}b_{p2} & \cdots & a_{1n}b_{pq} \\ \vdots & \vdots & & \vdots & \ddots & & \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots & & \ddots & \vdots & \vdots & & \vdots \\ a_{m1}b_{11} & a_{m1}b_{12} & \cdots & a_{m1}b_{1q} & \cdots & \cdots & a_{mn}b_{11} & a_{mn}b_{12} & \cdots & a_{mn}b_{1q} \\ a_{m1}b_{21} & a_{m1}b_{22} & \cdots & a_{m1}b_{2q} & \cdots & \cdots & a_{mn}b_{21} & a_{mn}b_{22} & \cdots & a_{mn}b_{2q} \\ \vdots & \vdots & \ddots & \vdots & & & \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{p1} & a_{m1}b_{p2} & \cdots & a_{m1}b_{pq} & \cdots & \cdots & a_{mn}b_{p1} & a_{mn}b_{p2} & \cdots & a_{mn}b_{pq} \end{bmatrix}.$$

Hadamard (Schur) product: For two matrices A and B of the same dimension $m \times n$, the Hadamard product $A \circ B$ (or $A \odot B$) is a matrix of the same dimension as the operands, with elements given by

$(A \circ B)_{ij} = (A \odot B)_{ij} = (A)_{ij}(B)_{ij}$. For matrices of different dimensions ($m \times n$ and $p \times q$, where $m \neq p$ or $n \neq q$), the Hadamard product is undefined.

6.3 A Cool Generalization of Vector Dot-Product

The dot product (also called the scalar product) of two vectors $\mathbf{a}$ and $\mathbf{b}$ in $\mathbb{R}^n$ is defined as:

$$\mathbf{a}^{\mathrm{T}}\mathbf{b} = \sum_{i=1}^{n} a_i b_i$$

where $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ and $\mathbf{b} = (b_1, b_2, \ldots, b_n)$ are vectors in $\mathbb{R}^n$, and a_i and b_i are the components of the vectors $\mathbf{a}$ and $\mathbf{b}$, respectively.

But, above product is easily generalizable to $\mathbf{a}$ and $\mathbf{b}$ with 'non'-scalar entries. Let $\mathbf{A} = \begin{pmatrix} \mathbf{a}_1^T \\ \mathbf{a}_2^T \\ \vdots \\ \mathbf{a}_m^T \end{pmatrix}$, $\quad \mathbf{B} = \begin{pmatrix} \mathbf{b}_1^T \\ \mathbf{b}_2^T \\ \vdots \\ \mathbf{b}_m^T \end{pmatrix}$

Then $\mathbf{A}^{\mathrm{T}}\mathbf{B} = \big(\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_m\big) \begin{pmatrix} \mathbf{b}_1^T \\ \mathbf{b}_2^T \\ \vdots \\ \mathbf{b}_m^T \end{pmatrix} = \sum_{i=1}^m \mathbf{a}_i \mathbf{b}_i^T$

6.4 Checking storage

```
library(pryr)
object_size(diag(rnorm(100)))
object_size(matrix(rnorm(10000), 100, 100))
object_size(rnorm(100))
```

A good resource to learn memory management in R: `http://adv-r.had.co.nz/memory.html`. Similar ideas hold for other software, too. So, it is often better to store a diagonal matrix just by its diagonal entries.

6.5 Data structure

In regards to storage, understanding the underlying data structure is super important. To see this, try the following in Python:

```
x = 2
y = x
x = 3
print(y)

import numpy as np

x = np.array([2])
y = x
x[0] = 3
print(y)
```

The basic Python data structures store data directly, whereas NumPy arrays store data in contiguous memory locations and operate primarily by referencing these addresses for efficient computation. Thus the second case would print the *new* value of ‘x’ while printing ‘y’. Thus, when you assign a new variable to an existing NumPy array, it creates a new reference to the same memory location, so no additional memory is used. However, if you modify the new variable or explicitly create a copy, new memory is allocated.

6.6 Computationally efficient usage of vectorized functions

In software like R and Python, many basic mathematical operations are vectorized, enabling them to be applied efficiently to entire arrays or vectors without explicit loops. However, to maximize efficiency, it is crucial to consider both memory usage and the number of floating-point operations (FLOPs) performed. Memory efficiency can be improved by avoiding unnecessary data copies and using in-place operations whenever possible, thereby minimizing memory overhead. Additionally, selecting appropriate data types, such as using `float32` instead of `float64`, can reduce both memory consumption and computational time. To further minimize FLOPs, it is best to use optimized vectorized functions that combine multiple operations, such as ‘%*%’ or `np.dot()` for matrix multiplication, instead of iterating manually. Moreover, storing intermediate results when they are reused can prevent redundant computations. By carefully managing these aspects, vectorized operations can be leveraged effectively to improve both computational speed and memory efficiency.

Example 1: Element-wise operations run faster than matrix-matrix multiplication. This is a very simple example of calculating square of a diagonal matrix

```
d <- rnorm(1000)
D <- diag(d)
system.time(D2 <- D%*%D)
system.time(D3 <- diag(d^2)) #or system.time(D3 <- diag(d*d))
system.time(D4 <- D*D)
range(D3-D2)
```

We see that the first one takes way more time than the second one. However, the results are the same as it should be. Here, the power operator is actually an element-wise operation. ‘%*%’ is a notation for matrix-matrix multiplication, which constitutes $mn(2p-1)$ floating point operations while multiplying $(m \times p)$ and $(p \times n)$ matrices (which produces a $m \times n$ matrix). In our case, thus, it involves $1000^2(2000-1)$ operations for the first approach, whereas the second approach only requires 1000 operations. For a diagonal matrix, many of those $1000^2(2000-1)$ operations are unnecessary.

Proof of the number of floating point operations result for matrix-matrix multiplication: In our final resulting matrix, there are $n \times m$ elements. Each of these entries is obtained by p multiplications (1 element from the first matrix and 1 from the second), then summing up. To sum p entries, we need to apply the '+' operator $p-1$ times. So the number of operations for one element in the output matrix is p multiplications and $p-1$ additions, meaning $2p-1$ operations in total. Then for all elements, we get $nm(2p-1)$ operations.

The proofs like above are good to know. ChatGPT might also give you proof if you ask for different problems. But you should be able to verify whether they make sense or not.

Example 2: We want to perform this sum $\mathbf{A} = \mathbf{B} + \mathbf{D}$, where $\mathbf{B}$ is a dense matrix and $\mathbf{D}$ is a diagonal matrix. How efficiently we can perform this sum? Do we really need to store the diagonal matrix as a matrix for this operation?

7 Inverse

Inverse is defined via multiplication. Primarily, to define an inverse, we consider the first notion of multiplication. An $n \times n$ square matrix A is called invertible (also nonsingular or nondegenerate), if there exists an $n \times n$ square matrix B such that
$\mathbf{AB} = \mathbf{BA} = \mathbf{I}_n$

Note that for a square matrix $\mathbf{A}$, left and right inverses are both the same!!!

For non-square matrices, the notion of invertibility is a bit complicated. They can be left or right invertible. The matrix A has a left inverse when there exists a B such that, $BA = I$ or a right inverse when there exists a C such that $AC = I$. When we say (just) invertible, we mean that both left and right inverse exist, and they are the same. A well-known result for matrix inverse is If $\mathbf{A}$ is an invertible matrix, then $A^{-1} = \dfrac{1}{\det(A)}(\operatorname{adj}(A))$. But this formula is not used for computing inverse in a computer.

Throughout the course, we will learn several other numerical algebra results.

8 Matrix decomposition

There are many useful matrix decomposition methods such as QR decomposition, LU decomposition, Cholesky decomposition.

QR decomposition is one of the fundamental decomposition. It can be implemented easily using Gram–Schmidt orthogonalization. In this decomposition, the aim is to represent any matrix $\mathbf{A}$ as $\mathbf{A} = \mathbf{QR}$ where $\mathbf{Q}$ is an orthogonal matrix and $\mathbf{R}$ is an upper triangular matrix. The eigenvalues in most of the standard packages use QR algorithm which uses QR decomposition. Thus it is one of the fundamentally important decompositions.

LU decomposition is motivated to represent any matrix as $\mathbf{A} = \mathbf{LU}$ are $\mathbf{L}$ and $\mathbf{U}$ are lower and upper triangular matrix respectively. It can be implemented using Gauss elimination. Cholesky decomposition is much faster than LU, however, only applicable to square PD matrices. Gauss elimination is also applied for this case too. How Gauss elimination works for Cholesky? Let's find out.

We try to represent a symmetric and positive definite matrix $\mathbf{A}$ as $\mathbf{A} = \mathbf{RR}^T$, where $\mathbf{R}$ is a lower triangular matrix. So what do we get?

$$a_{ii} = \sum_{j=1}^{i} r_{i,j}^2, \quad a_{ij} = \sum_{k=1}^{\min(i,j)} r_{j,k} r_{i,k}$$

Can you think of a way to recursively solve the above system of equations to get $r_{i,j}$'s? Note that $r_{i,k}$'s are zero when $k > i$.

One application of Cholesky is to generate multivariate normal random variable. If we want to generate $\mathbf{x} \sim \text{MVN}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$. We first generate standard normal random variable $\mathbf{y} \sim \text{Normal}(0, \mathbf{I})$. Then set $\mathbf{x} = \boldsymbol{\mu} + \mathbf{Ry}$, where $\mathbf{R}$ is the Cholesky decomposition of $\boldsymbol{\Sigma} = \mathbf{RR}^T$. Gauss elimination is also used for inverting any matrix. If a matrix belongs to a special class, we may do better.

9 Matrix calculus

Here I will discuss some frequently used derivative results, specifically for vectors and matrices.

The trace is a linear operator, hence it commutes with the derivative: $d\,\text{tr}(\mathbf{X}) = \text{tr}(d\mathbf{X})$.

Jacobi's formula: In matrix calculus, Jacobi's formula expresses the derivative of the determinant of a matrix A in terms of the adjugate of A and the derivative of A.

$\dfrac{\partial \det(A)}{\partial A_{ij}} = \text{adj}(A)_{ji}$.

Please check `https://en.wikipedia.org/wiki/Matrix_calculus#Identities`

In general, for scalar-matrix and scalar-vector differentiation, the general formula is the following. [Note that: scalar-vector differentiation means differentiating a scalar-valued function with respect to a vector-valued variable.]

1. scalar-matrix: Let $\mathbf{X}$ be matrix-valued variable and $f(\mathbf{X})$ stands for a scalar-valued function of $\mathbf{X}$ (e.g. $f(\mathbf{X}) = \text{trace}(\mathbf{X}^T\mathbf{A})$). Then $\frac{\partial f(\mathbf{x})}{\partial \mathbf{X}}$ would be a matrix such that $\left(\frac{\partial f(\mathbf{X})}{\partial \mathbf{X}}\right)_{i,j} = \frac{\partial f(\mathbf{X})}{\partial x_{i,j}}$
2. scalar-vector: Let $\mathbf{x}$ be vector-valued variable and $f(\mathbf{x})$ stands for a scalar-valued function of $\mathbf{x}$ (e.g. $f(\mathbf{x}) = \mathbf{x}^T\mathbf{a}$). Then $\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$ would be a vector such that $\left(\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}\right)_i = \frac{\partial f(\mathbf{x})}{\partial x_i}$

For vector-vector differentiation, the output is a matrix. Let $g(\mathbf{x})$ be a vector-valued function of $\mathbf{x}$ (e.g. $g(\mathbf{x}) = 2\mathbf{x} \odot \mathbf{x} + 3\mathbf{x}$, where $\odot$ stands for element-wise product) Then $\frac{\partial g(\mathbf{x})}{\partial \mathbf{x}}$ is a matrix such that $\left(\frac{\partial g(\mathbf{x})}{\partial \mathbf{x}}\right)_{i,j} = \frac{\partial (g(\mathbf{x}))_i}{\partial x_j}$.

For all the other varieties like matrix-matrix, vector-matrix, and matrix-vector differentiations, the output will be four, three, and three-dimensional arrays respectively. And if you think about it, you can derive the results easily using the above formulas. Or one can also vectorize the matrices and use vector-vector differentiations for all the cases.

References

[1] Emmanuel Candes and Terence Tao. The dantzig selector: Statistical estimation when p is much larger than n. *The Annals of Statistics*, 35(6):2313–2351, 2007.