Homework 1

January 16, 2026

Please submit your HW on Canvas; include a PDF printout of any code and results, clearly labeled, e.g. from a Jupyter notebook. For coding problems, we recommend using Julia, but you can use other languages if you wish. It is due Friday January 23rd by 11:59pm EST.

Problem 1 (10 points)

Start reading the draft course notes (linked from https://github.com/mitmath/matrixcalc/). Find a place that you found confusing, and write a paragraph explaining the source of your confusion and (ideally) suggesting a possible improvement.

(Any other corrections/comments are welcome, too.)

Problem 2 (5+5 points)

Show that *any* linear operator L mapping $m \times n$ matrices X to $m \times n$ matrices $L[X]$ can be written as a linear combination of linear operations of the form of BXA^T . In particular, show that any such $L[X]$ can be written in the form

$$L[X] = \sum_{\substack{i_1,i_2 \in \{1,\ldots,m\} \\ j_1,j_2 \in \{1,\ldots,n\}}} c_{i_1,i_2,j_1,j_2} M^{(i_1,i_2)} X (N^{(j_1,j_2)})^T ,$$

where $c_{i_1,i_2,j_1,j_2} \in \mathbb{R}$ are some coefficients, $M^{(i_1,i_2)}$ is the "one-hot" $m \times m$ matrix with a 1 in entry (i_1, i_2) and zero elsewhere, and $N^{(j_1,j_2)}$ is the "one-hot" $n \times n$ matrix with a 1 in entry (j_1, j_2) and zero elsewhere.

You can do this in two steps:

1. Show that the set of matrices $M^{(i_1,i_2)} \otimes N^{(j_1,j_2)}$ form a *basis* for the vector space of all $mn \times mn$ matrices. (Try computing one or two examples of these basis matrices to start with.)
2. Relate this to operators $L[X]$ via vectorization and the Kronecker identity $\operatorname{vec}(BXA^T) = (A \otimes B) \operatorname{vec} X$ from class.

Problem 3 (5+5 points)

1. Pick a computer language with built-in function for Kronecker products and other linear algebra (e.g. `kron` in Julia's `LinearAlgebra` library or `numpy.kron` in Python. Generate random 50×50 matrices A, B, C. Use a timing tool to measure the time to compute the matrix–matrix–matrix product BCA^T. (The `BenchmarkTools` package in Julia does a good job of this, timing multiple runs and collecting statistics.) Then, form $M = A \otimes B$ and $c = \operatorname{vec} C$ (via `c=vec(C)` in Julia or `c=numpy.ravel(C, order='F')` in Python). Check that Mc approximately equals $\operatorname{vec}(BCA^T)$ up to roundoff errors, and measure the time to compute the matrix–vector product Mc.
2. In the previous part, you should hopefully have found that of the two times is *much* slower than the other. Give a likely explanation for this observation. (Predicting precise performance timings is difficult because computers are complicated, but you can get a rough idea by estimating a count of arithmetic operations.)

Problem 4 (3+3+3+3 points)

Do Exercise 2.1 (Elementwise Products) in the course notes, parts (a–d) (not part e).

Problem 5 (3+3+3+3+3 points)

Find the derivatives f' of the following functions. If f maps column vectors or matrices to scalars, give ∇f (so that $f'(x)[dx] = \langle \nabla f, dx \rangle$ in the usual inner product $x^T y$ for column vectors or $\operatorname{trace}(X^T Y)$ for matrices). If f maps column vectors to column vectors, give the Jacobian matrix. Otherwise, simply write down f' as a linear operation.

1. $f(x) = (xx^T) \sin.(x)$, the dot denoting (in Julia notation) element-wise application of the sin function, for $x \in \mathbb{R}^m$. (The notations of problem 4 might be helpful in expressing the final answer.)
2. $f(x) = \operatorname{trace}(Axx^T)$ where $x \in \mathbb{R}^n$ and A is a constant $n \times n$ matrix.
3. $f(x) = \operatorname{trace}((A \operatorname{diag}(x) A^T)^{-1})$ where $x \in \mathbb{R}^n$, A is a constant $m \times n$ matrix, and diag was defined in problem 4. You can assume that the matrix inverse here exists (which is true if A has full row rank and the entries of x are nonnegative, for example). (Differentiating this function came up on an online forum from a real application!)
4. $f(A) = \operatorname{trace}(Axx^T)$ as above, except that you are differentiating with respect to $n \times n$ matrices A and x is constant.
5. $f(x) = \operatorname{trace}((A \operatorname{diag}(x) A^T)^{-1})$ as above, except that you are differentiating with respect to $m \times n$ matrices A and x is constant.

Problem 6 (5+5+5 points)

Suppose that $A(p)$ takes a vector of parameters $p \in \mathbb{R}^n$ and returns the $n \times n$ matrix

$$A(p) = A_0 + pp^T$$

where A_0 is some constant $n \times n$ vector. matrix, define a scalar-valued function $g(p)$ by

$$g(p) = \|A(p)^{-1} b\|^2$$

for some constant vector $b \in \mathbb{R}^n$ (assuming we choose p and A_0 so that A is invertible). Note that, in practice, $A(p)^{-1}b$ is best *not* computed by explicitly inverting the matrix A or even by Gaussian elimination/factorization, especially if the process is repeated for multiple p values—instead, it can be computed more quickly by exploiting the structure of this matrix (e.g. by computing A_0^{-1} or its equivalent *once* and thereafter using the Sherman–Morrison formula; look it up).

(a) Write down a formula for computing the gradient ∇g with respect to p (in terms of matrix–vector products and matrix inverses).

(b) Outline an algorithm to compute both g and ∇g using only *two* linear solves $x = A^{-1}b$ and an "adjoint" solve $v = A^{-T}$(something), plus only $\Theta(n)$ (i.e., roughly proportional to n) additional arithmetic operations.

(c) Write a program implementing your ∇g algorithm (in Julia, Python, Matlab, or any language you want) from the previous part. (You don't need to use a fancy Sherman–Morrison solver; you can solve A^{-1}(vector) straightforwardly using your favorite matrix library.) Implement a finite-difference test: Choose A_0, b, p at random for $n = 5$, and check that $\nabla g \cdot \delta p \approx g(p + \delta p) - g(p)$ (to a few digits) for a randomly chosen small δp.