Homework 1 Solutions

January 25, 2026

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.)

Solution:

Student-dependent, but full marks if clearly written and explained.

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.

Solution:

1. By the definition of the Kronecker product, $M^{(i_1,i_2)} \otimes N^{(j_1,j_2)}$ looks like:

$$M^{(i_1,i_2)} \otimes N^{(j_1,j_2)} = \begin{pmatrix} \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \\ \vdots & \vdots & \mathbf{0} & \vdots & \vdots \\ \mathbf{0} & \cdots & N^{(j_1,j_2)} & \cdots & \mathbf{0} \\ \mathbf{0} & \cdots & \mathbf{0} & \cdots & \mathbf{0} \\ \vdots & \vdots & & \vdots & \vdots \\ \mathbf{0} & \mathbf{0} & \cdots & \mathbf{0} & \mathbf{0} \end{pmatrix},$$

   where $\mathbf{0}$ denotes an $n \times n$ matrix of zeros and $N^{(j_1,j_2)}$ appears only in the (i_1, i_2)-th block. But this is an $mn \times mn$ matrix with a 1 in the $(i_1 m + j_1, i_2 m + j_2)$ entry and zeros elsewhere, which is one of the "Cartesian" basis matrices of the space of $mn \times mn$ matrices. Furthermore, it should be clear that by varying (i_1, i_2, j_1, j_2) we can get a 1 in *any* entry that we want: (i_1, i_2) determines the block and (j_1, j_2) determines the position within the block. Hence, we can express any $mn \times mn$ matrix A in this basis as:

$$A = \sum_{\substack{i_1,i_2 \in \{1,\ldots,m\} \\ j_1,j_2 \in \{1,\ldots,n\}}} A_{i_1 m + j_1, i_2 m + j_2} M^{(i_1,i_2)} \otimes N^{(j_1,j_2)}$$

where A_{ℓ_1,ℓ_2} denotes the (ℓ_1,ℓ_2)-th entry of A. Moreover, the $M^{(i_1,i_2)} \otimes N^{(j_1,j_2)}$ are linearly independent since each such matrix has a 1 in a different entry. Since they are **linearly independent** and **span** the space of $mn \times mn$ matrices, they are basis.

Erratum: For the next part, we actually need the matrices $N^{(j_1,j_2)} \otimes M^{(i_1,i_2)}$ to form our basis. Fortunately, the proof is almost exactly the same, giving us the alternative basis:

$$A = \sum_{\substack{i_1,i_2 \in \{1,\ldots,m\} \\ j_1,j_2 \in \{1,\ldots,n\}}} A_{i_1+j_1 n, i_2+j_2 n} N^{(j_1,j_2)} \otimes M^{(i_1,i_2)}$$

2. As you hopefully learned in linear algebra, any linear operator $L[X]$ on X can be expressed as some $mn \times mn$ a matrix A acting on $\operatorname{vec} X$. That is, $\operatorname{vec}(L[X]) = A \operatorname{vec} X$ — "vectorizing" is simply a choice of basis. But if we express A in the Kronecker-product basis as in the previous part, by the Kronecker identity it follows that

$$\begin{aligned}
\operatorname{vec}(L[X]) &= A \operatorname{vec} X \\
&= \sum_{\substack{i_1,i_2 \in \{1,\ldots,m\} \\ j_1,j_2 \in \{1,\ldots,n\}}} A_{i_1+j_1 n, i_2+j_2 n} \left(N^{(j_1,j_2)} \otimes M^{(i_1,i_2)} \right) \operatorname{vec} X \\
&= \operatorname{vec} \sum_{\substack{i_1,i_2 \in \{1,\ldots,m\} \\ j_1,j_2 \in \{1,\ldots,n\}}} A_{i_1+j_1 n, i_2+j_2 n} M^{(i_1,i_2)} X (N^{(j_1,j_2)})^T
\end{aligned}$$

using the fact that vec is linear (so that we can exchange it with the sum). But then, by inspection, we have the desired expansion for $L[X]$, with coefficients

$$c_{i_1,i_2,j_1,j_2} = A_{i_1+j_1 n, i_2+j_2 n}$$

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.)

Solution:

1. The following code shows the output of this benchmark and test in Julia (on an Apple M4), with the help of the `BenchmarkTools` package for accurate timing. To make the results easier to interpret, we've also disabled multi-threading in the matrix library so that everything is running on a single CPU. The output shows that, while BCA^T and Mc give the same results up to roundoff errors (with the help of Julia's handy `isapprox` function), the former is faster by a factor of $704.375/9.875 \approx 71$ times, i.e. almost **100 times** faster.

```
julia> using LinearAlgebra, BenchmarkTools

julia> LinearAlgebra.BLAS.set_num_threads(1); # disable multithreading

julia> A = rand(50,50); B = rand(50,50); C = rand(50,50);

julia> @btime $B * $C * $A';
  9.875 μs (6 allocations: 40.16 KiB)

julia> M = kron(A, B); c = vec(C);

julia> isapprox(M*c, vec(B*C*A')) # check results match up to roundoff
true
```

```
julia> @btime $M * $c;
  704.375 μs (3 allocations: 20.06 KiB)
```

2. As outlined in section 3.3.4 of the course notes, there is a good reason this. Multiplying two $m \times m$ matrices has $O(m^3)$ cost (by the usual algorithm; there are algorithms that theoretically have even better complexity but are mostly impractical for typical m), while multiplying an $n \times n$ matrix by a vector has $O(n^2)$ cost, but here $n = m^2$ (the size of $M = A \otimes B$) so the $O(n^2) = O(m^4)$ cost of Mc should be *much worse* than that of BCA^T for large enough m.

   Can we be more quantitative? A simplistic model of performance is to count arithmetic operations (floating-point additions and multiplications), called "flops". An $m \times m$ matrix–matrix multiplication by the usual algorithms costs about $2m^3$ flops, whereas $n \times n$ matrix–vector multiplication costs about $2n^2$ flops. Here, $m = 50$ and so BCA^T costs about 4×50^3 flops, while $n = 50^2$ so the matrix–vector product Mc costs about 2×50^4 flops, for a ratio of $50^2/2 = 1250$. So, if the computational cost were completely determined by arithmetic, we woulc expect Mc to be about *1000 times* slower than BCA^T. Instead, we found that it is only about 100 times slower. Essentially, this is because computers are complicated: not only do lots of things affect performance besides arithmetic, but a 50×50 matrix (or a 2500×2500 matrix–vector product) is small enough that overheads like memory allocations start to be a big chunk of the cost.

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

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

Solution:

1. Show $x .* y = y .* x$: The k-th element of $x .* y$ is defined to be $x_k y_k = y_k x_k$ (multiplication of real numbers is commutative), so this is the same as the k-element of $y .* x$.
2. Show $A(x .* y) = A\,\mathrm{diag}(x)y$: This follows from associativity of matrix products, since $A\,\mathrm{diag}(x)y = A\,(\mathrm{diag}(x)y)$ and it was already given that $\mathrm{diag}(x)y = x .* y$.
3. Show $d(x .* y) = (dx) .* y + x .* (dy)$: This can be shown a variety of ways. Elementwise, it is just the ordinary product rule for scalar multiplication. Or We can use $x .* y = \mathrm{diag}(x)y$ and use the product rule for matrix–vector multiplication from class and the fact that $d(\mathrm{diag}(x)) = \mathrm{diag}(dx)$ since diag is linear. Or you could just note that the derivation of the product rule from class applies unmodified, since $dx .* dy$ is higher-order and hence vanishes (in the infinitesimal limit, compared to the linear terms).
4. For $f(x) = A(x .* x)$, show that the Jacobian is $2A\,\mathrm{diag}(x)$: we can just apply the rules from above. In particular, $df = A\,d(x .* x) = A(dx .* x + x .* dx)$ by the product rule, which equals $A(2x .* dx)$ by commutativity, which equals $A\,\mathrm{diag}(2x)dx = 2A\,\mathrm{diag}(x)dx$ (pulling out the factor of 2 by linearity), and hence $f' = 2A\,\mathrm{diag}(x)$ is the Jacobian.

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 $\mathrm{trace}(X^T Y)$ 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. $f(x) = \mathrm{trace}(Axx^T)$ where $x \in \mathbb{R}^n$ and A is a constant $n \times n$ matrix.
3. $f(x) = \mathrm{trace}((A\,\mathrm{diag}(x)A^T)^{-1})$ 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. $f(A) = \mathrm{trace}(Axx^T)$ as above, except that you are differentiating with respect to $n \times n$ matrices A and x is constant.
5. $f(x) = \mathrm{trace}((A\,\mathrm{diag}(x)A^T)^{-1})$ as above, except that you are differentiating with respect to $m \times n$ matrices A and x is constant.

Solution:

1. By the product rule, $df = dx\, x^T \sin.(x) + x\, dx^T \sin.(x) + xx^T d(\sin.(x))$. For the third term, we can see that

$$d(\sin.(x)) = d\begin{pmatrix}\sin(x_1)\\ \sin(x_2)\\ \vdots\end{pmatrix} = \begin{pmatrix}\cos(x_1)dx_1\\ \cos(x_2)dx_2\\ \vdots\end{pmatrix} = \cos.(x) .* dx$$

in the notation of the previous problem. Since $f(x)$ takes a column vector in and returns a column vector, we want to express the whole thing as a Jacobian matrix times dx. The first term is $dx\,(x^T \sin.(x))$, which is a *scalar* multiplying dx, and hence can be written as $(x^T \sin.(x))I_m\, dx$ where I_m is the $m \times m$ identity. For the second term $x\, dx^T \sin.(x)$, we can use the identity $x^T y = y^T x$ to rewrite it as $x \sin.(x)^T dx$. For the third term, we can use the identity $x .* y = \operatorname{diag}(x)y$ from the previous part. Hence, we have

$$df = \underbrace{\boxed{\left((x^T \sin.(x))I_m + x \sin.(x)^T + xx^T \operatorname{diag}(\cos.(x))\right)}}_{\text{Jacobian } f'(x)} dx$$

in terms of an $m \times m$ Jacobian matrix $f'(x)$.

2. Here, by linearity of the trace and the product rule, $df = \operatorname{trace}(A\, dx\, x^T + Ax\, dx^T)$. But since this is a function taking a vector to a scalar, we expect to be able to rewrite it as a dot product with a gradient. Using the cyclic rule for the trace, combined with $\operatorname{trace} M = \operatorname{trace}(M^T)$, combined with trace(scalar) = scalar, we have:

$$\begin{aligned} df &= \operatorname{trace}(A\, dx\, x^T + Ax\, dx^T) \\ &= \operatorname{trace}(x^T A\, dx) + \operatorname{trace}(dx^T Ax) \\ &= \operatorname{trace}(x^T A\, dx) + \operatorname{trace}(x^T A^T dx) \\ &= x^T(A + A^T)\, dx = \langle (A + A^T)x,\, dx\rangle \end{aligned}$$

with the usual Euclidean inner product. Hence the gradient is $\nabla f = \boxed{(A + A^T)x}$.

Alternatively, we could have simply rewritten $f(x) = \operatorname{trace}(Axx^T) = \operatorname{trace}(x^T Ax) = x^T Ax$ *before* taking the derivative, and then quoted the result from class that $\nabla(x^T Ax) = (A + A^T)x$.

3. Applying the rule for the differential of a matrix inverse in class, and defining $B(x) = A \operatorname{diag}(x) A^T$ for brevity, we have

$$\begin{aligned} df &= d \operatorname{trace}(B(x)^{-1}) = \operatorname{trace}\left[-B(x)^{-1} dB B(x)^{-1}\right] \\ &= \operatorname{trace}\left[-B(x)^{-1} A \operatorname{diag}(dx) A^T B(x)^{-1}\right] \\ &= \operatorname{trace}\left[-A^T B(x)^{-2} A \operatorname{diag}(dx)\right] . \end{aligned}$$

Since $f(x)$ takes in a vector and returns a scalar, we expect the result to be a vector-shaped gradient. We have an expression of the form $\operatorname{trace}(Y^T \operatorname{diag}(dx))$ where Y is an $n \times n$ matrix, and we need to rewrite it as an inner product $y^T x$ for some vector y. But, from class, $\operatorname{trace}(Y^T \operatorname{diag}(dx))$ is a Frobenius inner product that is simply the sum of the elementwise products of Y and $\operatorname{diag}(dx)$, which is then the sum of elementwise products of the *vector of diagonal entries* of Y with dx. If we let $\operatorname{diag} Y = \operatorname{diag}(Y) \in \mathbb{R}^n$ denote the column vector of diagonal entries of an $n \times n$ matrix Y ("overloading" the diag function in the style of Matlab or `numpy.diag` depending on whether the argument is a matrix or a vector), we therefore immediately find that:

$$\nabla f = \operatorname{diag}\left[-A^T B(x)^{-2} A\right] = \boxed{\operatorname{diag}\left[-A^T (A \operatorname{diag}(x) A^T)^{-2} A\right]}.$$

4. $df = \operatorname{trace}(dA\, xx^T) = \operatorname{trace}(xx^T dA) = \langle xx^T, dA\rangle_{\mathrm{F}}$ via the Frobenius inner product, so the matrix gradient is $\nabla f = \boxed{xx^T}$. (We looked at very similar functions in class.)

5. Defining $B = A \operatorname{diag}(x) A^T$ as above, and noting that $B = B^T$, we can apply our usual trace identities $\operatorname{trace} XY = \operatorname{trace} YX$ and $\operatorname{trace} M = \operatorname{trace} M^T$ to show:

$$\begin{aligned} df &= d \operatorname{trace}(B^{-1}) = \operatorname{trace}\left[-B^{-1} dB B^{-1}\right] \\ &= \operatorname{trace}\left[-B^{-1}(dA \operatorname{diag}(x) A^T + A \operatorname{diag}(x) dA^T) B^{-1}\right] \\ &= \operatorname{trace}\left[-\operatorname{diag}(x) A^T B^{-2} dA\right] + \operatorname{trace}\left[-dA^T B^{-2} A \operatorname{diag}(x)\right] \\ &= \operatorname{trace}\left[-2 \operatorname{diag}(x) A^T B^{-2} dA\right] = \langle -2B^{-2} A \operatorname{diag}(x), dA\rangle_{\mathrm{F}} \end{aligned}$$

and hence the matrix gradient is $\nabla f = -2B^{-2}A\,\mathrm{diag}(x) = \boxed{-2(A\,\mathrm{diag}(x)A^T)^{-2}A\,\mathrm{diag}(x)}$. (Note that this is a matrix of the same $m \times n$ shape as A!)

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.

Solution:

1. Recall from class that $d\|x\|^2 = d(x^Tx) = 2x^T dx$. Hence, if we define $x = A(p)^{-1}b$, and using the fact that scalars like p^Tx commute with the other multiplications, we have

$$\begin{aligned} dg &= 2x^T d(A^{-1})b = 2x^T(-A^{-1}dA\,A^{-1}b) \\ &= -2x^T A^{-1}d(pp^T)x \\ &= -2x^T A^{-1}(dp\,p^T + p\,dp^T)x \\ &= -2(p^Tx)x^T A^{-1}dp - dp^Tx(2x^T A^{-1}p) \\ &= -2(p^Tx)(A^{-T}x)^T dp - (2x^T A^{-1}p)x^T dp \\ &= -2(p^Tx)(A^{-T}x)^T dp - (2(A^{-T}x)^T p)x^T dp \\ &= \nabla g^T dp\,, \end{aligned}$$

and by inspection obtain

$$\nabla g = \boxed{-2\left((p^Tx)v + (v^Tp)x\right)}.$$

where we have defined $v = A^{-T}x$.

2. From above, it should be apparent that once we have computed $x = A^{-1}b$ and $v = A^{-T}x$ (two linear solves, which in practice can share most of their computations by re-using A^{-1} or the LU factorization of A, or better yet both using something fancier like the Sherman–Morrison formula), the rest of the gradient computation is simply dot products or multiplying scalars by vectors or adding vectors, all of which have $\Theta(n)$ cost.

3. The following code implements a finite-difference check in Julia, implemented in the most straightforward (but inefficient) way, and we can see that $\nabla g^T dp$ matches the finite difference to 6–7 digits, a strong indication that we didn't make any algebra mistakes:

```
julia> using LinearAlgebra

julia> n = 5; A₀ = randn(n,n); b = randn(5); p = randn(n); dp = randn(5)*1e-8;

julia> A(p) = A₀ + p*p';
```

```
julia> g(p) = norm(A(p) \ b)^2;

julia> x = A(p) \ b; v = A(p)' \ x; g = -2( (p'x) * v + (v'p) * x );

julia> g(p + dp) - g(p)  # finite difference
5.71845458985365e-7

julia> g'dp              # analytical gradient
5.718454227678724e-7
```