Homework 2 Solutions

January 31, 2026

Problem 1 (10 points)

Continue reading the draft course notes (linked from `https://github.com/mitmath/matrixcalc/`). Find another place that you found confusing, in a different chapter from your answer in homework 1, 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)

Do Problem 7.1 from the course notes, which is based on a common task in multivariate statistics.

Solution:

We have $f(A) = \log\det A + \operatorname{tr}(A^{-1}X)$ where $A = A^T$ and $X = X^T$ are real-symmetric positive-definite $m \times m$ matrices, with perturbations dA that are also real-symmetric.

1. To compute ∇f, let's first compute df and then compare it to $\langle \nabla f, dA\rangle = \operatorname{tr}(\nabla f^T, dA)$. The log-determinant derivative is already computed in section 7.2.2 of the course notes, and we can apply our formula for $d(A^{-1})$ to obtain:

$$df = \operatorname{tr}(A^{-1}dA) - \operatorname{tr}(A^{-1}dAA^{-1}X) = \operatorname{tr}\left((I - A^{-1}X)A^{-1}dA\right) .$$

   By inspection, we then obtain

$$\nabla f = \boxed{A^{-1}(I - XA^{-1})}$$

   by transposition. While this is an acceptable answer, the discussion of gradients in spaces with constraints (in chapter 14) tells us that if A is restricted to the subspace of symmetric matrices we should also expect a symmetric ∇f. To exploit the symmetry of dA, we can use the identity that $\operatorname{tr} M = \operatorname{tr}(M + M^T)/2$:

$$\begin{aligned} df &= \operatorname{tr}\left((I - A^{-1}X)A^{-1}dA + dA\, A^{-1}(I - XA^{-1})\right)/2 \\ &= \operatorname{tr}\left(\frac{(I - A^{-1}X)A^{-1} + A^{-1}(I - XA^{-1})}{2}dA\right) , \end{aligned}$$

   which gives

$$\text{symmetric } \nabla f = \boxed{\frac{(I - A^{-1}X)A^{-1} + A^{-1}(I - XA^{-1})}{2}}.$$

2. In either case, $\nabla f = 0$ if and only if $\boxed{A = X}$ (which is, in fact, a minimizer of this function, related to a negative log-likelihood for covariance estimation).

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

Let $f(A)$ be a function that maps $m \times m$ matrices to $m \times m$ matrices. Recall that its derivative $f'(A)$ is a linear operator that maps any change δA in A to the corresponding change $\delta f = f(A + \delta A) - f(A) \approx f'(A)[\delta A]$, to first order in δA.

In this problem, you will study and prove a remarkable identity (Mathias, 1996): if $f(A)$ is sufficiently smooth,[1] then for *any* δA (not necessarily small!) the following formula holds *exactly*:

$$f\left(\underbrace{\begin{bmatrix} A & \delta A \\ & A \end{bmatrix}}_{M}\right) = \begin{bmatrix} f(A) & f'(A)[\delta A] \\ & f(A) \end{bmatrix}.$$

That is, one applies f to a $2m \times 2m$ "block upper-trianguar" matrix M (blank lower-left = zeros), and the desired derivative is in the upper-right $m \times m$ corner of the result $f(M)$. (Note: please do your *own* derivation here, don't just look it up.)

1. Check this identity numerically against a finite-difference approximation $f(A + \delta A) - f(A)$, which should match the exact $f'(A)[\delta A]$ to a few digits, for $f(A) = \exp(A)$ (the **matrix exponential** e^A, computed by `exp(A)` in Julia, or `expm` in Scipy or Matlab—do *not* use the elementwise exponential!), for a random 3×3 matrix `A = randn(3,3)` and a random small perturbation `dA = randn(3,3) * 1e-8`. Note that you can make the block matrix above in Julia by `using LinearAlgebra` followed by `M = [A dA; 0I A]`, and you can extract an upper-right corner by (*e.g.*) `M[1:3,4:6]`.

   It is also worth verifying (by a finite-difference check) that the derivative of the matrix exponential is *not* simply multiplication by e^A (on either the left or right or both): $(e^A)'[dA] \neq e^A\,dA$ or $dA\,e^A$ or $e^{A/2}\,dA\,e^{A/2}$, unlike the scalar case.

2. Prove the identity by explicit computation for the cases: $f(A) = I$, $f(A) = A$, $f(A) = A^2$, and $f(A) = A^3$ (the latter two derivatives were computed in class). (Two of these are trivial! This is "bargain-basement induction": do a few small examples and see the pattern.)

3. Prove the identity for $f(A) = A^n$ for any $n \geq 0$ by induction: assume it works for A^{n-1} and show using the product rule that it therefore must work for A^n. (You already proved the trivial $n = 0$ base case in the previous part.)

   Remark: Once it works for any A^n, it immediately follows that it works for any $f(A)$ described by a Taylor series, such as $\exp(A) = I + A + A^2/2 + A^3/6 + \cdots + A^n/n! + \cdots$, since such a function is just a linear combination of A^n terms.

4. Prove the identity for $f(A) = A^{-1}$: since we know (from class) that $f'(A)[\delta A] = -A^{-1}\,\delta A\,A^{-1}$, plug this into the right-hand side of the formula above and show that it is the inverse of M (multiply by M and show you get I).[2]

Solution:

1. Here is our finite-difference check, showing good agreement (to 5–6 significant digits) for the derivative of the matrix exponential:

```
julia> using LinearAlgebra

julia> A = randn(3,3); dA = randn(3,3) * 1e-8;

julia> exp_deriv(A, dA) = exp([A dA; 0I A])[1:3, 4:6]; # Mathias formula

julia> exp(A + dA) - exp(A) # finite-difference approximation
3×3 Matrix{Float64}:
 -9.43083e-9    2.40296e-8  -1.01382e-8
  8.41809e-10  -3.80726e-9   3.84864e-9
 -1.18475e-8    2.39357e-8  -1.41492e-8

julia> exp_deriv(A, dA) # exp'(A)[dA]
3×3 Matrix{Float64}:
 -9.43083e-9    2.40296e-8  -1.01382e-8
  8.41809e-10  -3.80726e-9   3.84864e-9
 -1.18475e-8    2.39357e-8  -1.41492e-8
```

[1] The result is easiest to show when $f(A)$ has a Taylor series (is "analytic"), and in fact you will do this below, but Higham (2008) shows that it remains true whenever f is $2m - 1$ times differentiable, or even just differentiable if A is "normal" ($AA^T = A^TA$).

[2] This is a special case of the famous "Schur complement" formula for the inverse of a 2×2 block matrix.

2. These three cases are:

   (a) $f(A) = A^0 = I$: in this case $f(M) = I$ ($2m \times 2m$), and so the upper-right block is **zero**. This, of course, is the correct result $d(I) = 0$.

   (b) $f(A) = A$: in this case, $f(M) = M$, and the upper-right block is δA. Again, this is the correct result: $df = f'(A)[dA] = d(A) = dA$, so $f'(A)[\delta A] = \delta A$.

   (c) $f(A) = A^2$. In this case,

   $$f(M) = M^2 = \begin{pmatrix} A & \delta A \\ & A \end{pmatrix} \begin{pmatrix} A & \delta A \\ & A \end{pmatrix} = \begin{pmatrix} A^2 & A\,\delta A + \delta A\,A \\ & A^2 \end{pmatrix}$$

   by the usual "rows-times-columns" rule (which works for matrix *blocks* as well as for scalar elements). But then the upper-right block $A\,\delta A + \delta A\,A$ is precisely $f'(A)[\delta A]$ as derived in class by the product rule.

   (d) $f(A) = A^3$. Building off the previous part, we have

   $$f(M) = M^3 = M^2 M = \begin{pmatrix} A^2 & A^2\,\delta A + \delta A\,A \\ & A \end{pmatrix} \begin{pmatrix} A & \delta A \\ & A \end{pmatrix} = \begin{pmatrix} A^3 & A^2\,\delta A + A\,\delta A\,A + \delta A\,A^2 \\ & A^3 \end{pmatrix}$$

   again by the usual "rows-times-columns" rule. The upper-right block $\delta A + A\,\delta A\,A + \delta A\,A^2$ is again $f'(A)[\delta A]$ as derived in class, corresponding to $d(A^3) = dA + A\,dA\,A + dA\,A^2$

3. For an inductive proof, we assume (for $n > 0$) that the identity holds for $n-1$, i.e. that:

   $$M^{n-1} = \begin{pmatrix} A^{n-1} & (A^{n-1})'[\delta A] \\ & A^{n-1} \end{pmatrix}.$$

   It then follows that

   $$M^n = M^{n-1} M = \begin{pmatrix} A^{n-1} & (A^{n-1})'[\delta A] \\ & A^{n-1} \end{pmatrix} \begin{pmatrix} A & \delta A \\ & A \end{pmatrix} = \begin{pmatrix} A^n & A^{n-1}\,\delta A + (A^{n-1})'[\delta A]\,A \\ & A^n \end{pmatrix},$$

   again by the usual "rows-times-columns" rule. But the upper-right block corresponds exactly to the product rule $d(A^n) = d(A^{n-1}A) = A^{n-1}dA + d(A^{n-1})A$, so it is indeed $(A^n)'[\delta A]$ as desired.

   As noted in the problem, the inductive base case $n = 0$ was already shown in the previous part, so our result must now hold for all $n \geq 0$.

4. For $f(A) = A^{-1}$, we know from class that $f'(A)[\delta A] = -A^{-1}\,\delta A\,A^{-1}$. We now want to show that

   $$M^{-1} = \begin{pmatrix} A^{-1} & -A^{-1}\,\delta A\,A^{-1} \\ & A^{-1} \end{pmatrix},$$

   which we can establish by explicit multiplication with M (on either the left or right):

   $$\begin{pmatrix} A^{-1} & -A^{-1}\,\delta A\,A^{-1} \\ & A^{-1} \end{pmatrix} \begin{pmatrix} A & \delta A \\ & A \end{pmatrix} = \begin{pmatrix} A^{-1}A & A^{-1}\,\delta A - A^{-1}\,\delta A\,A^{-1}A \\ & A^{-1}A \end{pmatrix} = I$$

   as desired: $f(M) = M^{-1}$ indeed must have the correct derivative $-A^{-1}\,\delta A\,A^{-1}$ in the upper-right block and A^{-1} on the diagonal blocks.

Problem 4 (5+5 points)

Implement Newton's method on the characteristic-polynomial function $f(z) = \det(zI - A)$ to find an eigenvalue of the $m \times m$ matrix A (a root λ where $f(\lambda) = 0$), using the formula for the derivative of a determinant from class.

1. Show that each step of Newton's method can be implemented via a single matrix inversion (without knowing the eigenvalues of A).[3]

[3] In practice, there are ways to accelerate this further by preprocessing A, for example by putting A in "upper-Hessenberg" form. See `https://github.com/JuliaLinearAlgebra/GenericLinearAlgebra.jl/issues/167`. But you need not do this here—this is not a numerical linear-algebra class!

2. Implement Newton's method using your algorithm (using some programming language with a linear-algebra library for matrix inversion etcetera) for the 3×3 matrix

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix},$$

starting from an initial guess $z = \operatorname{tr} A = 15$. Compare to your library's built-in eigenvalues function—how many iterations are needed to get one of the eigenvalues to at least 10 significant digits?

Solution:

1. From our formula for the differential of the determinant,

$$\begin{aligned} d(\det(zI - A)) &= \det(zI - A) \operatorname{tr}\left((zI - A)^{-1} d(zI - A)\right) \\ &= \det(zI - A) \operatorname{tr}\left((zI - A)^{-1}\right) dz \\ \implies \frac{d}{dz} \det(zI - A) &= \det(zI - A) \operatorname{tr}\left((zI - A)^{-1}\right) \end{aligned}$$

where dz is a scalar so it is okay to divide by it (technically, dividing by δz and taking the limit $\delta z \to 0$). But when you are finding a root of $f(z) = 0$, the Newton step is $z \to z - f'(z)^{-1} f(z)$, so in this case the Newton step is

$$\boxed{z \to z - \frac{1}{\operatorname{tr}\left((zI - A)^{-1}\right)}},$$

in which the $\det(zI - A)$ factor has *cancelled*. (This is also discussed in section 7.2 of the course notes.)

2. The Julia implementation and results are listed below, in which we see that it takes only **4 Newton steps** to obtain more than 10 digits compared to the biggest eigenvalue $\boxed{\lambda \approx 16.11684396980704\ldots}$ returned by `eigvals(A)`. (Newton's method asymptotically *doubles* the number of digits on every step. If you use `z = BigFloat(15)`, it will perform the Newton steps in arbitrary-precision arithmetic, defaulting to about 77 decimal digits, which allows you to see the rapid convergence even more clearly: it only takes about 7 steps to converge all 75+ digits.)

```
julia> using LinearAlgebra

julia> A = [1 2 3; 4 5 6; 7 8 9]
3×3 Matrix{Int64}:
 1  2  3
 4  5  6
 7  8  9

julia> z = 15
15

julia> for i = 1:5   # 5 Newton steps
         @show z = z - 1/tr(inv(z*I - A))
       end
z = z - 1 / tr(inv(z * I - A)) = 16.304347826086957
z = z - 1 / tr(inv(z * I - A)) = 16.12092749547181
z = z - 1 / tr(inv(z * I - A)) = 16.11684597057029
z = z - 1 / tr(inv(z * I - A)) = 16.116843969807523
z = z - 1 / tr(inv(z * I - A)) = 16.116843969807043

julia> eigvals(A)   # Julia's eigensolver
3-element Vector{Float64}:
 -1.1168439698070434
 -8.582743335036247e-16
 16.11684396980703
```

Problem 5 (10 points)

In class (and in the course notes, chapter 9), we went over the example of backpropagation in a 3-layer neural network (NN). Evaluation of the neural network for inputs v_1 produced "hidden layer" data v_2 and v_3, and finally an output v_4 that was fed into a loss function $L(v_4)$. To compute the gradient with respect to the NN parameters x (e.g. weights and biases), evaluated in reverse mode, we first derived a "backwards recurrence" ("adjoint" recurrence) $w_3 = \nabla_{v_4} L \to w_2 = \frac{\partial F_3}{\partial v_3}^T w_3 \to w_1 = \frac{\partial F_2}{\partial v_2}^T w_2$, from which we obtained the gradient

$$(\nabla_x \text{loss})^T = \sum_{k=1}^{3} w_k^T \frac{\partial F_k}{\partial x} .$$

(And there were further simplifications if the layers had disjoint parameters x_k, were of specific forms, etcetera.)

Suppose that instead our loss function is

$$\text{loss} = L(v_4) + \alpha \|v_3\|^2$$

so that it depends on *both* the final output v_4 and on the "hidden" layer v_3 with some scalar strength $\alpha > 0$. (Loss functions that depend on the hidden layers are common in machine learning, e.g. as regularizations to prevent the hidden values from becoming too big.) Give the modified backpropagation algorithm for ∇_xloss in this case—does the backwards recurrence for w_k change, or the final formula for the gradient in terms of w_k, or both?

Solution:

The gradient of the $L(v_4)$ term is the same as in class and the course notes. Clearly, there will be an additional term in the gradient from the derivative $\alpha\|v_3\|^2 = \alpha\|F_2(x, F_1(x, v_1))\|^2$. (Recall from class that $d(\alpha\|v_3\|^2) = 2\alpha v_3^T d(v_3)$, and we must then apply the chain rule to $d(v_3)$.) Let's combine the old and new gradients together and see what happens:

$$\begin{aligned} d(\text{loss}) &= (\nabla L)^T \left[\frac{\partial F_3}{\partial x} + \frac{\partial F_3}{\partial v_3} \underbrace{\left(\frac{\partial F_2}{\partial x} + \frac{\partial F_2}{\partial v_2}\frac{\partial F_1}{\partial x} \right)}_{d(v_3)} \right] dx \\ &\quad + 2\alpha v_3^T \underbrace{\left(\frac{\partial F_2}{\partial x} + \frac{\partial F_2}{\partial v_2}\frac{\partial F_1}{\partial x} \right)}_{d(v_3)} dx \\ &= (\nabla_x \text{loss})^T dx . \end{aligned}$$

Note that the $(\cdots)$ term, from $d(v_3)$, appears twice, so we can group terms, evaluating left-to-right in reverse mode as in class and the notes:

$$(\nabla_x \text{loss})^T = \underbrace{(\nabla L)^T}_{w_3^T} \frac{\partial F_3}{\partial x} + \underbrace{\left[\underbrace{(\nabla L)^T \frac{\partial F_3}{\partial v_3}}_{w_3^T} + 2\alpha v_3^T \right]}_{\text{changed } w_2^T} \frac{\partial F_2}{\partial x} + \overbrace{\left(w_2^T \frac{\partial F_2}{\partial v_2} \right)}^{w_1^T} \frac{\partial F_1}{\partial x} ,$$

Comparing to what we did previously, the *only* change is to the definition of w_2!

That is, we have the *same* final formula $(\nabla_x \text{loss})^T = \sum_{k=1}^{3} w_k^T \frac{\partial F_k}{\partial x}$ as before, and almost the same backwards recurrence

$$w_3 = \nabla_{v_4} L \to \boxed{w_2 = \frac{\partial F_3}{\partial v_3}^T w_3 + 2\alpha v_3} \to w_1 = \frac{\partial F_2}{\partial v_2}^T w_2 ,$$

with the only change being the w_2 step (boxed).