Dr. Christian Vollmann

Winter Term 2022

[November 4, 2022]

Elements of Mathematics

Sheet 01

Due date: **XXX**

Name:

Matriculation number:

Task:	1	2	3	4	Total	Grade
Points:						
Total:	6	10	5	10	31	–

Matrix Product as Sum of rank-1 Matrices

1. Let $A \in \mathbb{R}^{m\times k}$ and $B \in \mathbb{R}^{k\times n}$. Show that

$$A \cdot B = \sum_{i=1}^{k} a_i b_i^\top = \sum_{i=1}^{k} \begin{pmatrix} a_{1i} \\ \vdots \\ a_{mi} \end{pmatrix} \cdot \begin{pmatrix} b_{i1} & \dots & b_{in} \end{pmatrix},$$

   where $a_i \in \mathbb{R}^{m\times 1}$ denotes the i-th *column* of A and $b_i^\top \in \mathbb{R}^{1\times n}$ denotes the i-th *row* of B.

2. Construct two examples with actual numbers.

Remark: Also see p. 10-11 in Gilbert Strang's "Linear Algebra and Learning from Data".

Solution:

Note that by definition of the matrix product we have that the entry at (μ, ν) of AB is given by

$$(AB)_{\mu\nu} = \sum_{i=1}^{k} a_{\mu i} b_{i\nu}.$$

Again, by definition of the matrix product, for the i-th column $a_i = (a_{1i}, \dots, a_{mi})^\top \in \mathbb{R}^{m\times 1}$ and i-th row $b_i^\top = (b_{i1}, \dots, b_{in}) \in \mathbb{R}^{1\times n}$, we find

$$(a_i b_i^\top)_{\mu\nu} = \sum_{j=1}^{1} (a_i)_{\mu j} (b_i^\top)_{j\nu} = (a_i)_{\mu 1} (b_i^\top)_{1\nu} = a_{\mu i} b_{i\nu}.$$

Thus

$$\left(\sum_{i=1}^{k} a_i b_i^\top \right)_{\mu\nu} = \sum_{i=1}^{k} \left(a_i b_i^\top \right)_{\mu\nu} = \sum_{i=1}^{k} a_{\mu i} b_{i\nu} = (AB)_{\mu\nu}.$$

Inverse Matrix

Please prove the following statements.

1. An invertible matrix $A \in \mathbb{F}^{n\times n}$ has exactly one inverse matrix.
2. The inverse A^{-1} of an invertible matrix $A \in \mathbb{F}^{n\times n}$ is also invertible, with inverse $\left(A^{-1}\right)^{-1} = A$.
3. The product of two invertible matrices, say A and B, is invertible with inverse

$$(AB)^{-1} = B^{-1} A^{-1}.$$

4. A diagonal matrix

$$D = \mathrm{diag}(d_1, \dots, d_n) = \begin{pmatrix} d_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & d_n \end{pmatrix} \in \mathbb{F}^{n\times n}$$

   is invertible if and only if $d_i \neq 0$ for all $i = 1, \dots, n$. What is its inverse?
   Hint: It may be useful to split up the equivalence $\Leftrightarrow$ into $\Rightarrow$ and $\Leftarrow$ and to prove each of them separately.

5. Construct an example matrix and derive its inverse.

Solution:

1. Suppose $BA = I$ and $AC = I$, then
$$B = BI = B(AC) = (BA)C = IC = C.$$
In the next subtasks we verify that the suggested inverse, say $\widetilde{A}$, satisfies the determining requirement $A\widetilde{A} = \widetilde{A}A = I$.

2. Let $B := A^{-1}$ and $\widetilde{B} := A$, then by definition of the inverse for A we find
$$B\widetilde{B} = A^{-1}A = I$$
and
$$\widetilde{B}B = AA^{-1} = I.$$
Thus $B^{-1} = (A^{-1})^{-1} = \widetilde{B} = A$.

3. Let $C := AB$ and $\widetilde{C} := B^{-1}A^{-1}$, then by exploiting the rules for matrix computations we obtain
$$C\widetilde{C} = (AB)B^{-1}A^{-1} = A(BB^{-1})A^{-1} = A \cdot I \cdot A^{-1} = AA^{-1} = I$$
and similarly
$$\widetilde{C}C = B^{-1}A^{-1}(AB) = B^{-1}(A^{-1}A)B = B^{-1} \cdot I \cdot B = B^{-1}B = I.$$
Thus $C^{-1} = (AB)^{-1} = \widetilde{C} = B^{-1}A^{-1}$.

4. We again split the proof for the equivalence ("$\Leftrightarrow$", "if and only if") into two statements ("$\Rightarrow$", "$\Leftarrow$").
"$\Leftarrow$": First, let $d_i \neq 0$ for all i (thus we can divide by d_i) and set
$$\widetilde{D} := \mathrm{diag}(d_1^{-1}, \dots, d_n^{-1}) = \begin{pmatrix} d_1^{-1} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & d_n^{-1} \end{pmatrix} \in \mathbb{F}^{n \times n}.$$
Then by definition of the matrix product we can quickly verify that
$$D\widetilde{D} = I \quad \text{and} \quad \widetilde{D}D = I,$$
implying $D^{-1} = \widetilde{D}$ in this case.
"$\Rightarrow$": Proof by contradiction: Let $d_i = 0$ for at least one i. Then the i-th row (and column) of D solely contains 0 entries. Thus for any $\widetilde{D} \in \mathbb{F}^{n \times n}$ we have that the i-th row of $D\widetilde{D}$ is necessarily a zero row. Thus there cannot be a matrix $\widetilde{D}$ so that $D\widetilde{D} = I$. In particular, there cannot be a matrix $\widetilde{D}$ satisfying the requirements of the inverse matrix for D.

Alternatively:
The invertibility statement also follows from:
$$D \in \mathrm{GL}_n(\mathbb{F}) \quad \Leftrightarrow \quad \det(D) = \prod_{i=1}^{n} d_i \neq 0 \quad \Leftrightarrow \quad \forall 1 \leq i \leq n \colon d_i \neq 0$$
Then with the first part above we can derive the explicit expression for the inverse D^{-1}.

5. Take for example $D = \mathrm{diag}(1, 2, \dots, n) \in \mathbb{R}^{n \times n}$ for $n \in \mathbb{N}$, then $D^{-1} = \mathrm{diag}(1, \frac{1}{2}, \dots, \frac{1}{n})$.

Projections and Least Squares
Let $a, b \in \mathbb{R}^n \setminus \{0\}$ be two nonzero vectors. Consider the 1-dimensional optimization task
$$\min_{c \in \mathbb{R}} \frac{1}{2}\|ca - b\|_2^2 =: f(c),$$
where $\|x\|_2 := \left(\sum_{i=1}^n x_i^2\right)^{\frac{1}{2}}$ denotes the Euclidean norm of a vector $x \in \mathbb{R}^n$.

1. Determine the parameter $c \in \mathbb{R}$ which minimizes f.
Hint: As in high-school, compute the derivative f' of f with respect to c and solve the equation $f'(c) = 0$.

2. Compare your results to the projection of b onto a, i.e., $\mathrm{proj}_a(b) := \frac{a^\top b}{\|a\|_2} \frac{a}{\|a\|_2}$.

Solution:

First we note that

$$f(c) = \frac{1}{2}\|ca - b\|_2^2 = \frac{1}{2}\left(c^2 a^\top a - 2ca^\top b - b^\top b\right)$$

Thus, for the derivative with respect to the scalar c, we find

$$f'(c) = ca^T a - a^T b.$$

Since $a \neq 0$ and therefore $a^\top a \neq 0$, we find

$$f'(\hat{c}) = 0 \quad \Leftrightarrow \quad \hat{c} = \frac{a^T b}{a^T a}.$$

By convexity of f we can conclude that $\hat{c}$ is a minimizer (you will learn this in the course "Numerical Optimization").

Remark: We will later identity the equation $ca^T a - a^T b = 0$ as the **normal equation**. The vector on the line span(a) closest to b in terms of the Euclidean norm is given by

$$\hat{c}a = \frac{a^T b}{a^T a} a = \frac{a^T b}{\|a\|_2} \frac{a}{\|a\|_2} = \text{proj}_a(b).$$

Ex 4 Linear Systems, Python 10

Implementation Matters

In order to solve the problem $Ax = b$, there are plenty of algorithms available. In this exercise we invoke several SciPy routines to solve linear systems numerically. Thereby, we will learn that different algorithms, or even different implementations of the same algorithm, can have a huge effect on the efficiency.

Consider the following test example: The matrix $A \in \mathbb{R}^{n \times n}$ with constant diagonals given by

$$A = \begin{pmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{pmatrix} \in \mathbb{R}^{n \times n}$$

and the right-hand side vector

$$b = (1, 0, \cdots, 0, 1)^\top \in \mathbb{R}^n.$$

In this case we know that the unique[1] solution $x^* = A^{-1}b$ is given by

$$x^* = (1, 1, \cdots, 1)^\top \in \mathbb{R}^n.$$

Implement the following options to solve the problem $Ax = b$ (i.e., given A and b from above, find a numerical solution $\tilde{x}$ such that $A\tilde{x} \approx b$):

1. Dense: Work with A as dense `numpy.ndarray`.
   a) The forbidden way: Find a SciPy function to compute a numerical inverse and apply it to b to obtain $\tilde{x}$.
   b) The default way: Find a general solver for linear systems.
   c) Structure exploiting solver I: Find a way to inform this general solver about the fact that A is *positive definite*.[2]
   d) Structure exploiting solver II: Find another solver which exploits the fact, that A has constant diagonals.
2. Sparse: Exploit the sparsity of the matrix and work with A as `scipy.sparse.csr_matrix`.
   a) The forbidden way: Find a SciPy function to compute the inverse of a sparse matrix and apply it to b to obtain $\tilde{x}$.
   b) The default way: Find a general solver for sparse linear systems.

[1] One can show that the matrix A is invertible.
[2] We'll learn about this property later.

c) Structure exploiting solver: Find a function that implements the *conjugate gradient (cg)* method to solve the system iteratively. Play with the optional parameter <maxiter> to restrict the number of iterations. What do you observe?

Run your code for different dimensions n (especially large $n \geq 10^5$) and:

1. measure the time needed for each of your solving routine,
2. and find a SciPy function to compute the error $\frac{1}{n}\|\tilde{x} - x^*\|_2$ for each solving routine.

Remark: For now, it is not important to understand how the algorithms work. We'll learn more about them throughout this course. Here, you learn to find appropriate SciPy functions which solve your problem and to serve the interfaces of these functions.

Solution:

```
import numpy as np
import scipy.linalg as linalg
import scipy.sparse as sparse
from time import time


n = 5000
a = -np.eye(n,k=-1) + 2*np.eye(n) + -np.eye(n,k=1)
b = np.zeros(n)
b[0], b[-1] = 1, 1


start=time()
a_inv = linalg.inv(a)
x1 = a_inv @ b
print("\ninverse dense\n", time()-start)
print(linalg.norm(x1-np.ones(n))/n)
#print(np.allclose(x1, np.ones(n), atol=1e-12, rtol=1e-12))

start=time()
x = linalg.solve(a, b)
print("\nsolve dense\n", time()-start)
print(linalg.norm(x-np.ones(n))/n)


# symmwetric
start=time()
x = linalg.solve(a, b, assume_a = "sym")
print("\nsolve symmetric\n",time()-start)
print(linalg.norm(x-np.ones(n))/n)

# symmetric and positive definite
start=time()
x = linalg.solve(a, b, assume_a = "pos")
print("\nsolve symmetric + positive definite\n",time()-start)
print(linalg.norm(x-np.ones(n))/n)

# tell the solver about special structures
# constant diagonals
start=time()
x = linalg.solve_toeplitz([2,-1]+[0]*(n-2),b)
print("\nsolve toeplitz dense\n", time()-start)
print(linalg.norm(x-np.ones(n))/n)

start=time()
x = sparse.linalg.cg(a, b)[0]
print("\ncg dense\n", time()-start)
print(linalg.norm(x-np.ones(n))/n)


##############################################
```

```
# sparsity
a = sparse.csr_matrix(a)


# solve
start=time()
x = sparse.linalg.spsolve(a, b)
print("\nsolve sparse\n", time()-start)
print(linalg.norm(x-np.ones(n))/n)

# sparse inv
a = sparse.csr_matrix(a)
start=time()
a_inv = sparse.linalg.inv(a)
x = a_inv @ b
print("\ninverse sparse\n", time()-start)
print(linalg.norm(x-np.ones(n))/n)

# if some errors are tolerable, try to approximate
start=time()
x = sparse.linalg.cg(a, b, maxiter=50)[0]
print("\ncg sparse\n", time()-start)
print(linalg.norm(x-np.ones(n))/n)



#if __name__ == "__main__":
```