Lab 3

Noah Prentice

8 November 2024

Note.

Code for this assignment was done in Python. Only important code snippets and outputs are shown here, but complete files can be found at this GitHub repository.[a] All code for Exercise N is in `ExN.py`.

[a] URL: https://github.com/NoahPrentice/Numerical-Linear-Algebra-MTH551-F24/tree/main/Lab3

Exercise 1. Let $\varepsilon = 10^{-10}$ and consider the overdetermined system $Ax = b$ with

$$\begin{bmatrix} 1 & 1 \\ \varepsilon & 0 \\ 0 & \varepsilon \end{bmatrix} x = \begin{bmatrix} -\varepsilon \\ 1+\varepsilon \\ 1-\varepsilon \end{bmatrix}$$

(b) Try to compute the least squares solution with MATLAB using `mldivide` .

(c) Try to compute the least squares solution with MATLAB using the normal equations and `mldivide` .

(d) Use MATLAB's `chol` command to try to compute the least squares solution by the classical Cholesky method.

(e) Use MATLAB's `qr` command to try to compute the least squares solution by the QR method.

(f) Use MATLAB's `svd` command to try to compute the least squares solution via the SVD.

(g) Comment on the difference between results obtained in *each* of the above.

Solution. **(b)** As Python was used instead of MATLAB, we use the `numpy.linalg.lstsq` function to compute the least squares solution. It seems that this function differs from `mldivide` in MATLAB only for under-determined systems, so the functions should be essentially equivalent for this problem.

```
11 def part_b():
12     return np.linalg.lstsq(a, b)
```

This produces the output $x = \begin{bmatrix} 1.00000131 & -1.00000131 \end{bmatrix}^T$.

(c) Here we use the `numpy.linalg.lstsq` function again, but we solve the normal equations $A^*Ax = A^*b$.

```
15 def part_c():
16     return np.linalg.lstsq(a_star @ a, a_star @ b)
```

This produces the output $x = \begin{bmatrix} -4.48100399 \cdot 10^{-37} & -4.48100399 \cdot 10^{-37} \end{bmatrix}^T$.

(d) Now we use the `numpy.linalg.cholesky` function to compute the Cholesky factorization of A^*A.

```python
def part_d():
    # numpy produces LL^* Cholesky factorization instead of R^*R.
    l, l_star = np.linalg.cholesky(a_star @ a)
    r = np.transpose(l_star)
    r_star = np.transpose(l)

    w = np.linalg.solve(r_star, a_star @ b)
    return np.linalg.solve(r, w)
```

This produces no output; an exception is thrown on the grounds that A^*A is not symmetric positive definite (s.p.d.). Note, however, that

$$A^*A = \begin{bmatrix} 1+\varepsilon^2 & 1 \\ 1 & 1+\varepsilon^2 \end{bmatrix},$$

which *is*, in fact, s.p.d.:

(i) A^*A is clearly symmetric by observation.

(ii) Recall that M is positive definite if and only if it has positive eigenvalues. The eigenvalues of A^*A are roots of its characteristic polynomial, which after a small calculation is

$$p(t) = t^2 + (-2-2\varepsilon^2)t + (\varepsilon^2+\varepsilon^4).$$

The quadratic formula and some calculation yields eigenvalues

$$\lambda_1, \lambda_2 = 1+\varepsilon^2 \pm \sqrt{1+\varepsilon^2}.$$

Since $\varepsilon^2 > 0$, we have that $\lambda_1 = 1+\varepsilon^2+\sqrt{1+\varepsilon^2} > 0$ automatically. For the other root, we know that $1+\varepsilon^2 > 1$, so $\sqrt{1+\varepsilon^2} < 1+\varepsilon^2$ and thus $\lambda_2 = 1+\varepsilon^2-\sqrt{1+\varepsilon^2} > 0$, too. Thus both eigenvalues of A^*A are positive, and hence A^*A is positive definite.

So, the exception produced by the `cholesky` command must be caused by numerical error (e.g. round-off error). This is not incredibly surprising: $\varepsilon^2 = 10^{-20}$, which is smaller than $\varepsilon_{machine} \approx 10^{-16}$ for double-precision floating point computers, so numerical round-off is expected. In particular, $\text{fl}(\varepsilon^2) = 0$, that is, the computer rounds ε^2 to 0, and therefore $\text{fl}(\lambda_2) = 1-1 = 0$. Thus round-off would cause the computer representation of A^*A to be singular and in particular not positive-definite.

(e) For a QR-factorization, we use the `numpy.linalg.qr` command.

```python
def part_e():
    q, r = np.linalg.qr(a)
    q_star = np.transpose(q)
    return np.linalg.solve(r, q_star @ b)
```

This produces an output of $x = \begin{bmatrix} 1.0000007 & -1.0000007 \end{bmatrix}^T$.

(f) For a reduced SVD factorization, we use the `numpy.linalg.svd` command.

```python
def part_f():
    u, s, v_star = np.linalg.svd(a, full_matrices=False)
    u_star = np.transpose(u)
    s = np.diag(s)  # singular values are put in a 1d array by default
    v = np.transpose(v_star)

    w = np.linalg.solve(s, u_star @ b)
    return v @ w
```

This produces the same output as in part (e), $x = \begin{bmatrix} 1.0000007 & -1.0000007 \end{bmatrix}^T$.

(g) The two algorithms to have results significantly worse results were the ones in (c) and (d), which attempt to solve the problem through the normal equations. In (c), applying a least-squares solver to the normal equations yielded a very inaccurate result. Since the true solution is $x = \begin{bmatrix} 1 & -1 \end{bmatrix}^T$, (c) produces a result with ∞-norm error of $\approx 1 - 10^{-37}$, which is very high. Then, (d), as already discussed, does not produce a result at all.

The other algorithms fared far better: the built-in least-squares solver from (b) had ∞-norm error of $\approx 10^{-6}$, and the QR and SVD methods in (e) and (f) had ∞-norm error of $\approx 7 \cdot 10^{-7}$.[1] This makes these algorithms highly preferable to the algorithms in (c) and (d) when the matrix is close to singular. This highlights the difference in the algorithmic stability of the methods: (c) and (d) are sensitive to perturbations in the inputs, while (b), (e), and (f) are not nearly as sensitive. □

[1] These errors are very comparable, so there is no significant difference between the accuracy of (a) and that of (e) or (f). Still, (e) and (f) were slightly more accurate in this case.