MA398 Matrix Analysis and Algorithms: Exercise Sheet 4

1. Prove that the condition number of a scalar function $f : \mathbb{R} \to \mathbb{R}$ is invariant under scaling of the function. That is, show that if $f(x) = ag(x)$ for some constant a and function g, then $\kappa_f(x) = \kappa_g(x)$.

   Hint: Use the fact that the derivative of $f(x)$ with respect to x is $f'(x) = ag'(x)$ and apply the definition of the condition number.

   Answer: Proof:

   Given the function $f(x) = ag(x)$, where a is a constant, we want to show that the condition number $\kappa_f(x) = \kappa_g(x)$.

   Recall the definition of the condition number:

   $$\kappa_f(x) = \left\| \frac{xf'(x)}{f(x)} \right\|$$

   The derivative of $f(x)$ is $f'(x) = ag'(x)$. Therefore, we can express the condition number of f as:

   $$\kappa_f(x) = \left\| \frac{xag'(x)}{ag(x)} \right\|$$

   The constant a can be cancelled from the numerator and the denominator:

   $$\kappa_f(x) = \left\| \frac{xg'(x)}{g(x)} \right\|$$

   But this is just the condition number of g:

   $$\kappa_f(x) = \kappa_g(x)$$

   This completes the proof. It shows that the condition number of a function is invariant under scaling – i.e., multiplying the function by a constant does not change its condition number.

2. Write a Python function that computes the condition number of a non-singular matrix A. Use numpy's built-in functions to compute matrix norms and inverses. Your function should take a 2D numpy array A as input and return the condition number of A as output.

   Then, generate a random matrix A (you can use numpy's np.random.rand(n, n) function to generate an $n \times n$ matrix with entries between 0 and 1), compute its condition number using your function, and discuss the implications of the condition number in the context of solving a system of linear equations $Ax = b$.

   After you run the program, interpret the output by discussing how the condition number affects the sensitivity of the solution x to small changes in the matrix A or the right-hand side b. You might also discuss what it means if the condition number is close to 1, significantly larger than 1, or very large.

3. Analyze the backward stability of the division operation in floating-point arithmetic similar to the subtraction operation shown in Example 1 in lecture 11. Assume that the exact input is $x = (x_1, x_2)^T$, and exact operation is $f(x) = x_1/x_2 = y$. Define and calculate the backward error, discuss the stability of this operation and any potential limitations or special cases that need to be considered.

Answer: The exact operation is $x = (x_1, x_2)^T$ and $f(x) = x_1/x_2 = y$. Suppose we have the computed version as $\theta = fl(x_1) \oslash fl(x_2)$, where $\oslash$ represents the division operation in floating-point arithmetic and $fl(\cdot)$ represents the floating-point approximation of a number.

In floating-point arithmetic, each operation is subject to a small rounding error. Let's denote these rounding errors as $\epsilon^{(1)}$ and $\epsilon^{(2)}$ for $fl(x_1)$ and $fl(x_2)$ respectively, with $|\epsilon^{(i)}| \leq \epsilon_m$, $i = 1, 2$. Then we can write $fl(x_1) = x_1(1 + \epsilon^{(1)})$ and $fl(x_2) = x_2(1 + \epsilon^{(2)})$.

Now, the computed division is $\theta = fl(x_1) \oslash fl(x_2) = \xi_1 \oslash \xi_2 = (\xi_1/\xi_2)(1 + \epsilon^{(3)})$, where $\epsilon^{(3)}$ is another rounding error introduced by the division operation.

This simplifies to:

$$\begin{aligned}\theta &= (x_1(1 + \epsilon^{(1)})/x_2(1 + \epsilon^{(2)}))(1 + \epsilon^{(3)}) \\ &= (x_1/x_2)(1 + \epsilon^{(1)} - \epsilon^{(2)} + \epsilon^{(1)}\epsilon^{(2)})(1 + \epsilon^{(3)}) \\ &= y(1 + \epsilon^{(4)})\end{aligned}$$

where $\epsilon^{(4)} = \epsilon^{(1)} - \epsilon^{(2)} + \epsilon^{(1)}\epsilon^{(2)} + \epsilon^{(3)} + \epsilon^{(1)}\epsilon^{(3)} - \epsilon^{(2)}\epsilon^{(3)} + \epsilon^{(1)}\epsilon^{(2)}\epsilon^{(3)} = O(\epsilon_m)$ as $\epsilon_m \searrow 0$.

Thus, the computed division can be written as an exact division with perturbed input data: $\theta = f(\zeta)$ for $\zeta = (x_1(1 + \epsilon^{(4)}), x_2)^T$. The absolute backward error is $|\zeta - x| = |(\epsilon^{(4)}x_1, 0)^T| \leq \epsilon_m|x_1|$. The relative backward error is $|\zeta - x|/|x| \leq \epsilon_m$.

This means that the division operation is backward stable under the assumption that $x_2 \neq 0$.

4. Consider a simple algorithm for calculating the dot product between two vectors in Python using numpy's built-in function np.dot(). Write a Python function that introduces a small perturbation to each element of the input vectors before computing the dot product. Then, compare the result of the perturbed dot product with the exact result and calculate the relative forward and backward errors. Discuss your findings with respect to the backward stability of the dot product operation.

```
import numpy as np

# Define the function
def perturbed_dot_product(vector1, vector2, eps):
    """
    This function computes the dot product between two vectors after
        ↪ adding a small perturbation to each element of the vectors.

    Parameters:
    vector1, vector2: Input vectors
    eps: The magnitude of the perturbation

    Returns:
    dot_product_perturbed: The dot product after perturbation
    relative_forward_error: The relative forward error
    relative_backward_error: The relative backward error
    """
    # Add your implementation here

# Test the function with example vectors and a small perturbation
vector1 = np.array([1, 2, 3])
vector2 = np.array([4, 5, 6])
```

```
eps = 1e-5

# Call the function
# dot_product_perturbed, relative_forward_error, relative_backward_error =
    ↪ perturbed_dot_product(vector1, vector2, eps)

# Print the results
# print("Dot product after perturbation: ", dot_product_perturbed)
# print("Relative forward error: ", relative_forward_error)
# print("Relative backward error: ", relative_backward_error)
```

5. (Growth factor) Show that

$$A = \begin{pmatrix} 1 & 0 & \cdots & 0 & 1 \\ -1 & \ddots & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ \vdots & & \ddots & \ddots & \vdots \\ -1 & \cdots & \cdots & -1 & 1 \end{pmatrix}$$

has growth factor $g_n(A) = 2^{n-1}$. Hint: Denoting the U matrix in algorithm LU after step K by $U^{(k)}$, show by induction that $U^{(k)}_{n,n} = 2^k$.