Lab 17

Conditioning and Stability

Lab Objective: *Explore the condition of problems and the stability of algorithms.*

The *condition number* of a function measures how sensitive that function is to changes in the input. On the other hand, the *stability* of an algorithm measures how well that algorithm computes the value of a function from exact input.

Condition Number of a Function

The (absolute) condition number of a function $f : \mathbb{R}^m \to \mathbb{R}^n$ is

$$J(\mathbf{x}) = \lim_{\delta \to 0} \sup_{\|\delta \mathbf{x}\| \leq \delta} \frac{\|\delta f\|}{\|\delta \mathbf{x}\|} \tag{17.1}$$

where $\delta f = f(\mathbf{x} + \delta \mathbf{x}) - f(\mathbf{x})$. In other words, the condition number of f is (the limit of) the change in output over the change of input.

Similarly, the *relative condition number* of f is the limit of the relative change in output over the relative change in input, i.e.,

$$\kappa(\mathbf{x}) = \lim_{\delta \to 0} \sup_{\|\delta \mathbf{x}\| \leq \delta} \left(\frac{\|\delta f\|}{\|f(\mathbf{x})\|} \Big/ \frac{\|\delta \mathbf{x}\|}{\|\mathbf{x}\|} \right). \tag{17.2}$$

In fact,

$$\kappa(\mathbf{x}) = \frac{\|\mathbf{x}\|}{\|f(\mathbf{x})\|} J(\mathbf{x})$$

A function is *ill-conditioned* if its condition number is large.

Small changes to the input of an ill-conditioned function produce large changes in output. In applications, it is important to know if a function is ill-conditioned because there is usually some error in the parameters passed to the function.

Example: the Wilkinson Polynomial

Let $f : \mathbb{C}^{n+1} \to \mathbb{C}^n$ be the function that sends $(a_1, \ldots, a_{m+1})$ to the roots of $a_1 x^n + a_2 x^{n-1} + \ldots + a_n x + a_{n+1}$. In other words, this function describes the problem

of finding the roots of a polynomial. Unfortunately, root finding is extremely ill-conditioned.

A classic example is the Wilkinson polynomial

$$w(x) = \prod_{r=1}^{20}(x-r) = x^{20} - 210x^{19} + 20615x^{18} - \dots.$$

We will use NumPy to explore the condition number of f at the coefficients of $w(x)$. These coefficients are contained in the NumPy array `w_coeffs` below. We also create an array `w_roots` containing the roots of $w(x)$.[1]

```
>>> import numpy as np
>>> from scipy import linalg as la
>>> w_coeffs = np.array([1, -210, 20615, -1256850, 53327946, -1672280820,
                  40171771630, -756111184500, 11310276995381,
                  -135585182899530, 1307535010540395,
                  -10142299865511450, 63030812099294896,
                  -311333643161390640, 1206647803780373360,
                  -3599979517947607200, 8037811822645051776,
                  -12870931245150988800, 13803759753640704000,
                  -8752948036761600000, 2432902008176640000])
>>> w_roots = np.arange(1, 21)
```

Next we perturb the polynomial by changing the x^{19}-coefficient from -210 to -210.0000001.

```
>>> perturb = np.zeros(21)
>>> perturb[1]=1e-7
>>> perturbed_coeffs = w_coeffs - perturb
```

Now we find the roots of the perturbed polynomial. The function `np.poly1d()` creates a polynomial object using the array passed to it as coefficients, and `np.roots()` finds the roots of a polynomial.

```
>>> perturbed_roots = np.roots(np.poly1d(perturbed_coeffs))
```

The new roots are plotted with the original roots in Figure 17.1.

Finally we compute an approximation to the condition number. We sort the roots before we compare them to ensure that they are in the same order.

```
>>> w_roots = np.sort(w_roots)
>>> perturbed_roots = np.sort(perturbed_roots)
>>> la.norm(perturbed_roots-w_roots)/la.norm(perturb)
68214100.15878984
```

Thus, the condition number of this problem is something like 10^7.

When we try to estimate the relative condition number, NumPy will not compute the 2-norm of `w_coeffs` because it is so large. NumPy will not compute the square root of a very large number. One easy solution is to use a different norm that does not require the square root.

[1] It is possible to create this array in NumPy by expanding the product $\prod_{r=1}^{20}(x-r)$, for example using `np.poly()`. However, this approach is *unstable* and will give you the wrong coefficients!

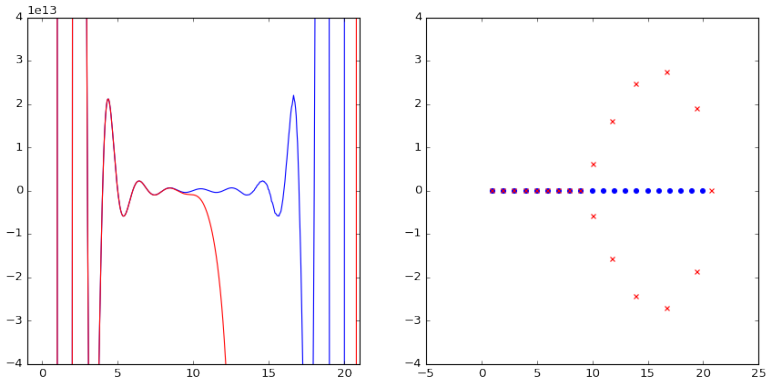

Figure 17.1: In these images, blue is associated with $w(x)$ and red is associated with $w(x)$ perturbed by 1e-7 in the x^{19}-coefficient. On the left is a plot of the two polynomials. On the right is a graphical representation of the roots of the polynomials. The blue dots are the roots of $w(x)$. The red x's are the roots of the perturbed polynomial.

```
>>> # Estimate the absolute condition number in the infinity norm
>>> k = la.norm(perturbed_roots-w_roots, np.inf)/la.norm(perturb, np.inf)
>>> k
28260604.34345689
>>> # Estimate the relative condition number in the infinity norm
>>> k*la.norm(w_coeffs, np.inf)/la.norm(w_roots, np.inf)
1.9505129642488696e+25
```

As you can see, the order of magnitude of the absolute condition number is the same as when we computed it with the 2-norm. The relative condition number for this problem is approximately 10^{24}.

There are some caveats to this example. First, when we compute the quotients in (17.1) and (17.2) for a fixed $\delta\mathbf{x}$, we are only *approximating* the condition number. The actual condition number is a limit of such quotients. We hope that when $||\delta\mathbf{x}||$ is small, a random quotient is at least the same order of magnitude as the limit, but we have no way to be sure.

Second, this example assumes that NumPy's root-finding algorithm is *stable*, so that the difference between the roots of `w_coeffs` and `perturbed_coeffs` is due to the difference in coefficients, and not the difference in roots. We will return to this issue in the next section.

Problem 1. Write a Python function that investigates the condition number of the Wilkinson polynomial by doing the following.

1. Perform this experiment:

Randomly perturb $w(x)$ by replacing each coefficient a_i with $a_i * r_i$, where r_i is drawn from a normal distribution centered at 1 with standard deviation $1e-10$.

Plot the results of 100 such experiments in a single graphic, along with the roots of the unperturbed polynomial $w(x)$. The plot should look something like Figure 17.2. This exercise reproduces Figure 12.1 on p. 93 of *Numerical Linear Algebra* by Lloyd N. Trefethen and David Bau III.

2. Using the final experiment only, estimate the relative and absolute condition number (in any norm you prefer). Print these numbers to the screen.

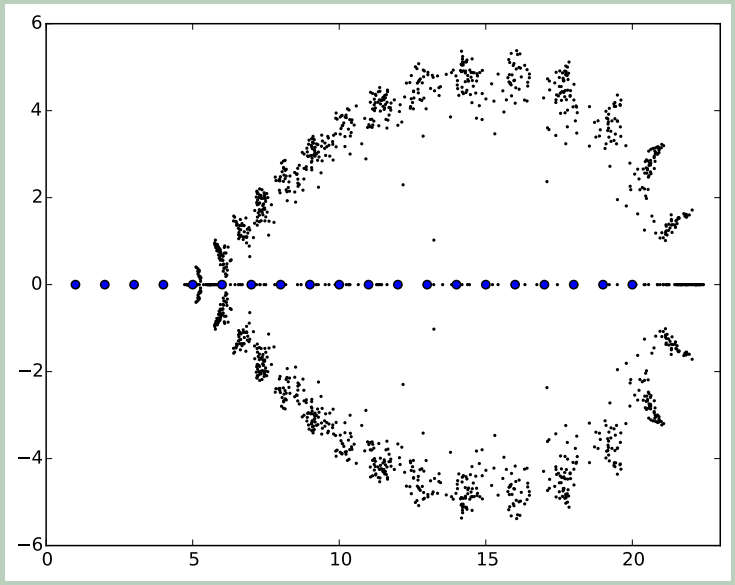

Figure 17.2: Sample result of Problem 1. The blue dots are the roots of $w(x)$ and the black dots are roots of random perturbations. This figure replicates Figure 12.1 on p. 93 of *Numerical Linear Algebra* by Lloyd N. Trefethen and David Bau III.

Example: Calculating Eigenvalues

Let $f : \mathbb{C}^{n^2} \to \mathbb{C}^n$ be the function that sends an $n \times n$ matrix to its n eigenvalues. This problem is well-conditioned for symmetric matrices, but can be extremely ill-conditioned for non-symmetric matrices.

Let us use NumPy to calculate the condition number of the eigenvalue problem at the identity matrix. First we check that the eigenvalue solver is stable here.

```
>>> M = np.array([[1,0],[0,1]])
>>> eigs = la.eig(M)[0]
```

```
>>> eigs
array([ 1.+0.j,  1.+0.j])
```

Now we perturb M by adding a matrix drawn from a random normal distribution over the complex numbers. We calculate the eigenvalues of the perturbed matrix.

```
>>> perturb = np.random.normal(0, 1e-10, M.shape) + np.random.normal(0,1e-10, M.↩
    shape)*1j
>>> eigsp = la.eig(M+perturb)[0]
```

Finally we use this data to approximate the condition number.

```
>>> k = la.norm(eigs-eigsp)/la.norm(perturb)    # Absolute condition number
0.62957336119253127
>>> k*la.norm(M)/la.norm(eigs)  # Relative condition number
0.62957336119253127
```

The absolute and relative condition number are the same because `la.norm(M)` and `la.norm(eigs)` are both $\sqrt{2}$.

Problem 2 (Optional). Let us explore the condition number of the eigenvalue problem.

1. (a) Write the following function.

```
def eig_condit(M):
    '''
    Approximate the condition number of the eigenvalue problem ↩
        at M.

    INPUT:
    M  - A 2-D NumPy array, representing a matrix.

    RETURN:
    A tuple containing approximations to the absolute and ↩
        relative condition numbers of the eigenvalue problem ↩
        at M.
    '''
```

   (b) Find an example of a 2×2 matrix with a very large condition number. (Hint: Look at matrices whose off-diagonal entries are very different in magnitude.)

   (c) What is the order of magnitude of the condition number of a symmetric 2×2 matrix?

2. Write the following function.

```
def plot_eig_condit(x0=-100, x1=100, y0=-100, y1=100, res=10):
    '''
    Plot the condition number of the eigenvalue problem on [x0, x1]x↩
        [y0,y1].
```

```
    Specifically, use plt.pcolormesh to plot the relative condition ↩
        number of the eigenvalue problem at [[1,x],[y,1]] on this ↩
        domain.
    The variable `res' should be the number of sample points taken ↩
        along each axis, for a total of `res'**2 points in the plot↩
        .
    '''
```

3. Call your function for `res`=10, 50, 100, 200 and 400 (output for $res = 200$ is pictured below). Recall that matplotlib scales the colorbar of the output to fit the largest and smallest output values. What can you conclude about the condition number of the eigenvalue problem at a "random" 2×2 matrix?

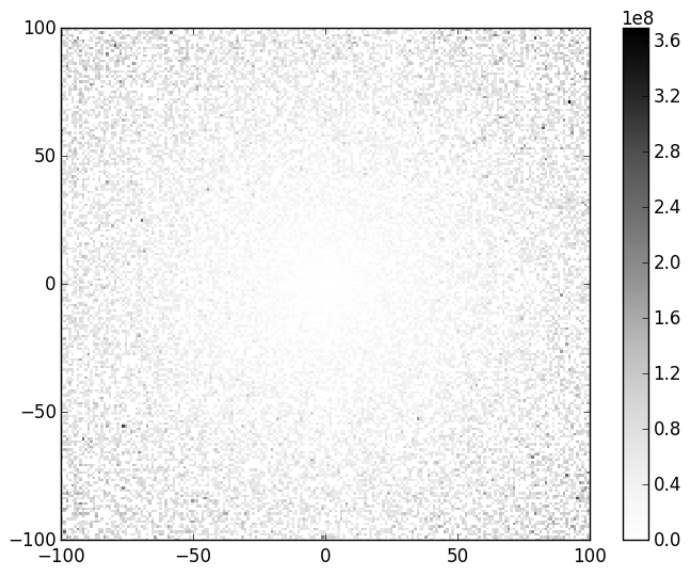

Figure 17.3: Output of `plot_eig_condit(res=200)`.

Stability of an Algorithm

The stability of an algorithm is measured by the error in its output. Suppose we have some algorithm to compute $f : \mathbb{R}^m \to \mathbb{R}^n$. Let $\tilde{f}(\mathbf{x})$ represent the value computed by the algorithm at $\mathbf{x}$. Then the *forward error* of f at $\mathbf{x}$ is $||f(\mathbf{x}) - \tilde{f}(\mathbf{x})||$, and the *relative forward error* of f at $\mathbf{x}$ is

$$\frac{||f(\mathbf{x}) - \tilde{f}(\mathbf{x})||}{||f(\mathbf{x})||}.$$

An algorithm is *stable* if this relative forward error is small.

As an example, let us examine the stability of NumPy's root finding algorithm

that we used to investigate the Wilkinson polynomial. We know the exact roots of $w(x)$, and we can also compute these roots using NumPy's `np.roots()` function.

```
>>> roots = np.arange(1,21)
>>> w_coeffs = np.array([1, -210, 20615, -1256850, 53327946, -1672280820,
                    40171771630, -756111184500, 11310276995381,
                    -135585182899530, 1307535010540395,
                    -10142299865511450, 63030812099294896,
                    -311333643161390640, 1206647803780373360,
                    -3599979517947607200, 8037811822645051776,
                    -12870931245150988800, 13803759753640704000,
                    -8752948036761600000, 2432902008176640000])
>>> computed_roots = np.roots(np.poly1d(w_coeffs))
```

We sort the roots to ensure they are in the same order, then compute the absolute and relative forward error.

```
>>> roots = np.sort(roots)
>>> computed_roots = np.sort(computed_roots)
>>> la.norm(roots-computed_roots)   # Forward error
0.020612653126379665
>>> la.norm(roots-computed_roots)/la.norm(roots)    # Relative forward error
0.00038476268486104599
```

This analysis gives us hope that questions of stability did not interfere too much with our experiments in Problem 1.

Catastrophic Cancellation

Catastrophic Cancellation is a term for when a computer takes the difference of two very similar numbers, and the result is stored with a small number of significant digits. Because of the way computers store and perform arithmetic on numbers, future computations can amplify a catastrophic cancellation into a huge error.

You are at risk for catastrophic cancellation whenever you subtract floats or large integers that are very close to each other. You can avoid the problem either by rewriting your program to not use subtraction, or by increasing the number of significant digits that your computer tracks.

Here is an example of catastrophic cancellation. Suppose we wish to compute $\sqrt{a} - \sqrt{b}$. We can either do this subtraction directly or perform the equivalent division

$$\sqrt{a} - \sqrt{b} = (\sqrt{a} - \sqrt{b})\frac{\sqrt{a} + \sqrt{b}}{\sqrt{a} + \sqrt{b}} = \frac{a - b}{\sqrt{a} + \sqrt{b}}.$$

Let us perform this computation both ways in NumPy with $a = 10^{20} + 1$ and $b = 10^{20}$.

```
>>> np.sqrt(1e20+1)-np.sqrt(1e20)
0.0
>>> 1/(np.sqrt(1e20+1)+np.sqrt(1e20))
5.0000000000000002e-11
```

Since $a \neq b$, clearly $\sqrt{a} - \sqrt{b}$ should be nonzero.

Problem 3. Let $I(n) = \int_0^1 x^n e^{x-1} dx$.

1. Prove that $0 \leq I(n) \leq 1$ for all n.

2. It can be shown that for $n > 1$,

$$I(n) = (-1)^n !n + (-1)^{n+1} \frac{n!}{e}$$

where $!n$ is the *subfactorial* of n. Use this formula to write the following function.

```
def integral(n):
    '''Return I(n).'''
```

Hint: The subfactorial function can be imported from SymPy with the line `from sympy import subfactorial`.

3. The actual values of $I(n)$ for many values of n are listed in the table below. Use your function `integral()` to compute $I(n)$ for these same values of n, and create a table comparing the data. How can you explain what is happening?

n	Actual value of $I(n)$
1	0.367879441171
5	0.145532940573
10	0.0838770701034
15	0.0590175408793
20	0.0455448840758
25	0.0370862144237
30	0.0312796739322
35	0.0270462894091
40	0.023822728669
45	0.0212860390856
50	0.0192377544343