10 Conditioning and Stability

Lab Objective: *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 accurately that algorithm computes the value of a function from exact input. Both of these concepts are important for answering the crucial question, "is my computer telling the truth?" In this lab, we examine the conditioning of common linear algebra problems, including computing polynomial roots and matrix eigenvalues. We also present an example to demonstrate how two different algorithms for the same problem may not have the same level of stability.*

Conditioning

The *absolute condition number* of a function $f : \mathbb{R}^m \to \mathbb{R}^n$ at a point $\mathbf{x} \in \mathbb{R}^m$ is defined by

$$\hat{\kappa}(\mathbf{x}) = \lim_{\delta \to 0^+} \sup_{\|\mathbf{h}\| < \delta} \frac{\|f(\mathbf{x}+\mathbf{h}) - f(\mathbf{x})\|}{\|\mathbf{h}\|}. \tag{10.1}$$

In other words, the absolute 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.

$$\kappa(\mathbf{x}) = \lim_{\delta \to 0^+} \sup_{\|\mathbf{h}\| < \delta} \left(\frac{\|f(\mathbf{x}+\mathbf{h}) - f(\mathbf{x})\|}{\|f(\mathbf{x})\|} \Big/ \frac{\|\mathbf{h}\|}{\|\mathbf{x}\|} \right) = \frac{\|\mathbf{x}\|}{\|f(\mathbf{x})\|} \hat{\kappa}(\mathbf{x}). \tag{10.2}$$

A function with a large condition number is called *ill-conditioned*. Small changes to the input of an ill-conditioned function may produce large changes in output. It is important to know if a function is ill-conditioned because floating point representation almost always introduces some input error, and therefore the outputs of ill-conditioned functions cannot be trusted.

The *condition number* of a matrix A, $\kappa(A) = \|A\|\|A^{-1}\|$, is an upper bound on the condition number for many of the common problems associated with the matrix, such as solving the system $A\mathbf{x} = \mathbf{b}$. If A is square but not invertible, then $\kappa(A) = \infty$ by convention. To compute $\kappa(A)$, we often use the matrix 2-norm, which is the largest singular value $\sigma_{\max}$ of A. Recall that if σ is a singular value of A, $\frac{1}{\sigma}$ is a singular value of A^{-1}. Thus, we have that

$$\kappa(A) = \frac{\sigma_{\max}}{\sigma_{\min}}, \tag{10.3}$$

which is also a valid equation for non-square matrices.

Achtung!

Ill-conditioned matrices can wreak havoc in even simple applications. For example, the matrix

$$A = \begin{bmatrix} 1 & 1 \\ 1 & 1.0000000001 \end{bmatrix}$$

is extremely ill-conditioned, with $\kappa(A) \approx 4 \times 10^{10}$. Solving the systems $A\mathbf{x} = \mathbf{b}_1$ and $A\mathbf{x} = \mathbf{b}_2$ can result in wildly different answers, even when $\mathbf{b}_1$ and $\mathbf{b}_2$ are extremely close.

```
>>> import numpy as np
>>> from scipy import linalg as la

>>> A = np.array([[1, 1], [1, 1+1e-10]])
>>> np.linalg.cond(A)
39999991794.058899

# Set up and solve a simple system of equations.
>>> b1 = np.array([2, 2])
>>> x1 = la.solve(A, b1)
>>> print(x1)
[ 2.  0.]

# Solve a system with a very slightly different vector b.
>>> b2 = np.array([2, 2+1e-5])
>>> la.norm(b1 - b2)
>>> x2 = la.solve(A, b2)
>>> print(x2)
[-99997.99172662  99999.99172662]   # This solution is hugely different!
```

If you find yourself working with matrices that have large condition numbers, check your math carefully or try to reformulate the problem entirely.

Note

An *orthonormal matrix* U has orthonormal columns and satisfies $U^\mathsf{T}U = I$ and $\|U\|_2 = 1$. If U is square, then $U^{-1} = U^\mathsf{T}$ and U^T is also orthonormal. Therefore $\kappa(U) = \|U\|_2\|U^{-1}\|_2 = 1$. Even if U is not square, all of its singular values are equal to 1, and again $\kappa(U) = \sigma_{\max}/\sigma_{\min} = 1$.

The condition number of a matrix cannot be less than 1 since $\sigma_{\max} \geq \sigma_{\min}$ by definition. Thus orthonormal matrices are, in a sense, the best kind of matrices for computations. This is one of the main reasons why numerical algorithms based on the QR decomposition or the SVD are so important.

Problem 1. Write a function that accepts a matrix A and computes its condition number using (10.3). Use `scipy.linalg.svd()`, or `scipy.linalg.svdvals()` to compute the singular values of A. Avoid computing A^{-1}. If the smallest singular value is 0, return ∞ (`np.inf`).

Validate your function by comparing it to `np.linalg.cond()`. Check that orthonormal matrices have a condition number of 1 (use `scipy.linalg.qr()` to generate an orthonormal matrix) and that singular matrices have a condition number of ∞ according to your function.

The Wilkinson Polynomial

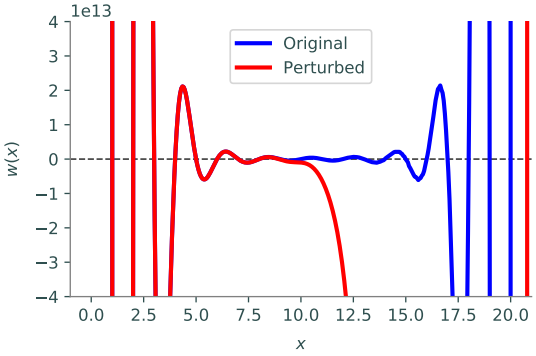
Let $f : \mathbb{C}^{n+1} \to \mathbb{C}^n$ be the function that maps a collection of $n+1$ coefficients $(c_n, c_{n-1}, \dots, c_0)$ to the n roots of the polynomial $c_n x^n + c_{n-1}x^{n-1} + \dots + c_2 x^2 + c_1 x + c_0$. Finding polynomial roots is an extremely ill-conditioned problem in general, so the condition number of f is likely very large. To see this, consider the *Wilkinson polynomial*, made famous by James H. Wilkinson in 1963.

$$w(x) = \prod_{r=1}^{20}(x - r) = x^{20} - 210x^{19} + 20615x^{18} - 1256850x^{17} + \cdots$$

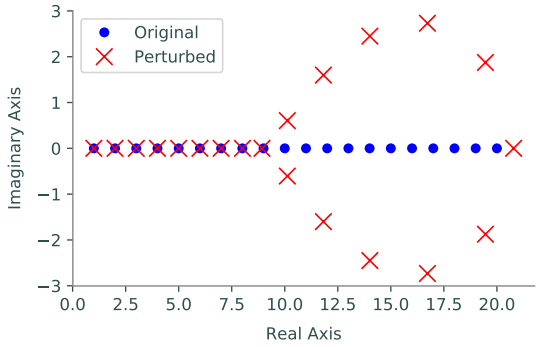
Let $\tilde{w}(x)$ be $w(x)$ where the coefficient on x^{19} is very slightly perturbed from -210 to -210.0000001. The following code computes and compares the roots of $\tilde{w}(x)$ and $w(x)$ using NumPy and SymPy.

```
>>> import sympy as sy
>>> from matplotlib import pyplot as plt

# The roots of w are 1, 2, ..., 20.
>>> w_roots = np.arange(1, 21)

# Get the exact Wilkinson polynomial coefficients using SymPy.
>>> x, i = sy.symbols('x i')
>>> w = sy.poly_from_expr(sy.product(x-i, (i, 1, 20)))[0]
>>> w_coeffs = np.array(w.all_coeffs())
>>> print(w_coeffs[:6])
[1 -210 20615 -1256850 53327946 -1672280820]

# Perturb one of the coefficients very slightly.
>>> h = np.zeros(21)
>>> h[1]=1e-7
>>> new_coeffs = w_coeffs - h
>>> print(new_coeffs[:6])
[1 -210.000000100000 20615 -1256850 53327946 -1672280820]

# Use NumPy to compute the roots of the perturbed polynomial.
>>> new_roots = np.roots(np.poly1d(new_coeffs))
```

Figure 10.1a plots $w(x)$ and $\tilde{w}(x)$ together, and Figure 10.1b and compares their roots in the complex plane.



(a) The original and perturbed Wilkinson polynomials. They match for only about half of the domain.



(b) Roots of the original and pertubed Wilkinson polynomials. About half of the perturbed roots are imaginary.

Figure 10.1

Figure 10.1 clearly indicates that a very small change in just a single coefficient drastically changes the nature of the polynomial and its roots. To quantify the difference, estimate the condition numbers (this example uses the L^∞ norm to compute $\hat{\kappa}$ and κ).

```
# Sort the roots to ensure that they are in the same order.
>>> w_roots = np.sort(w_roots)
>>> new_roots = np.sort(new_roots)

# Estimate the absolute condition number in the infinity norm.
>>> k = la.norm(new_roots - w_roots, np.inf) / la.norm(h, np.inf)
>>> print(k)
28262391.3304

# Estimate the relative condition number in the infinity norm.
>>> k * la.norm(w_coeffs, np.inf) / la.norm(w_roots, np.inf)
1.95063629993970+25                    # This is huge!!
```

There are some caveats to this example.

1. Computing the quotients in (10.1) and (10.2) for a fixed perturbation $\mathbf{h}$ only approximates the condition number. The true condition number is the limit of such quotients. We hope that when $\|\mathbf{h}\|$ is small, a random quotient is at least the same order of magnitude as the limit, but there is no way to be sure.

2. This example assumes that NumPy's root-finding algorithm, `np.roots()`, is *stable*, so that the difference between `w_roots` and `new_roots` is due to the difference in coefficients, and not to problems with `np.roots()`. We will return to this issue in the next section.

Even with these caveats, it is apparent that root finding is a difficult problem to solve correctly. Always check your math carefully when dealing with polynomial roots.

Problem 2. Write a function that carries out the following experiment 100 times.

1. Randomly perturb the true coefficients of the Wilkinson polynomial by replacing each coefficient c_i with $c_i * r_i$, where r_i is drawn from a normal distribution centered at 1 with standard deviation 10^{-10} (use `np.random.normal()`).
2. Plot the perturbed roots as small points in the complex plane. That is, plot the real part of the coefficients on the x-axis and the imaginary part on the y-axis. Plot on the same figure in each experiment.
   (Hint: use a pixel marker, `marker=','`, to avoid overcrowding the figure.)
3. Compute the absolute and relative condition numbers with the L^∞ norm.

Plot the roots of the unperturbed Wilkinson polynomial with the perturbed roots. Your final plot should resemble Figure 10.2. Finally, return the average computed absolute and relative condition numbers.

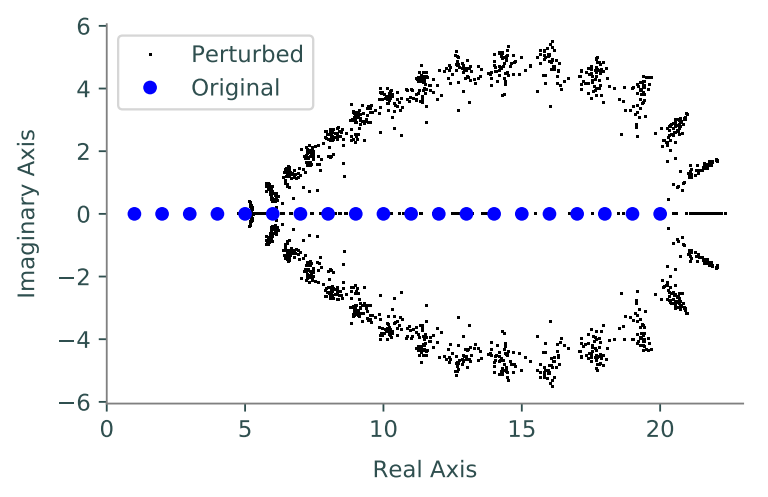


Figure 10.2: This figure replicates Figure 12.1 on p. 93 of *Numerical Linear Algebra* by Lloyd N. Trefethen and David Bau III.

Calculating Eigenvalues

Let $f : M_n(\mathbb{C}) \to \mathbb{C}^n$ be the function that maps an $n \times n$ matrix with complex entries to its n eigenvalues. This problem is well-conditioned for symmetric matrices, but it can be extremely ill-conditioned for non-symmetric matrices. Let A be an $n \times n$ matrix and let $\boldsymbol{\lambda}$ be the vector of the n eigenvalues of A. If $\tilde{A} = A + H$ is a pertubation of A and $\tilde{\boldsymbol{\lambda}}$ are its eigenvalues, then the condition numbers of f can be estimated by

$$\hat{\kappa}(A) = \frac{\|\boldsymbol{\lambda} - \tilde{\boldsymbol{\lambda}}\|}{\|H\|}, \qquad \kappa(A) = \frac{\|A\|}{\|\boldsymbol{\lambda}\|}\hat{\kappa}(A). \tag{10.4}$$

Problem 3. Write a function that accepts a matrix A and estimates the condition number of the eigenvalue problem using (10.4). For the perturbation H, construct a matrix with complex entries where the real and imaginary parts are drawn from normal distributions centered at 0 with standard deviation $\sigma = 10^{-10}$.

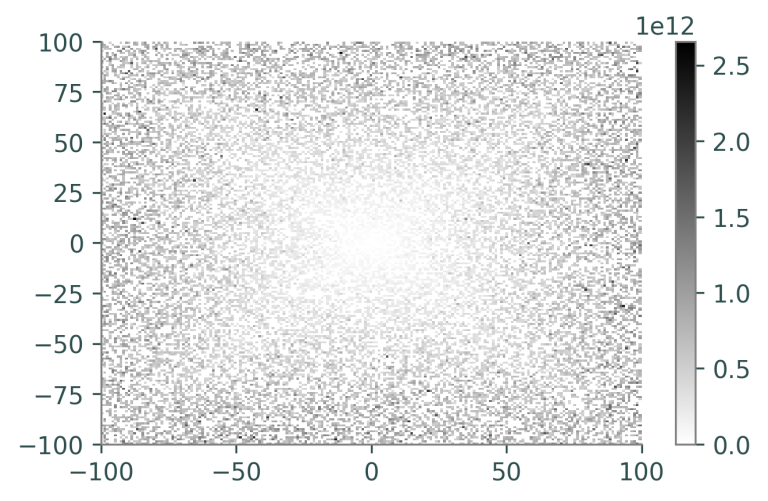
```
reals = np.random.normal(0, 1e-10, A.shape)
imags = np.random.normal(0, 1e-10, A.shape)
H = reals + 1j*imags
```

Use `scipy.linalg.eig()` or `scipy.linalg.eigvals()` to compute the eigenvalues of A and $A + H$, and use the 2-norm for both the vector and matrix norms. Return the absolute and relative condition numbers.

Problem 4. Write a function that accepts bounds $[x_{\min}, x_{\max}, y_{\min}, y_{\max}]$ and an integer `res`. Use your function from Problem 3 to compute the relative condition number of the eigenvalue problem for the 2×2 matrix

$$\begin{bmatrix} 1 & x \\ y & 1 \end{bmatrix}$$

at every point of an evenly spaced `res`$\times$`res` grid over the domain $[x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$. Plot these estimated relative condition numbers using `plt.pcolormesh()` and the colormap `cmap='gray_r'`. With `res=200`, your plot should look similar to the following figure.



Problem 4 shows that the conditioning of the eigenvalue problem depends heavily on the matrix, and that it is difficult to know a priori how bad the problem will be. Luckily, most real-world problems requiring eigenvalues are symmetric. In their book on Numerical Linear Algebra, L. Trefethen and D. Bau III summed up the issue of conditioning and eigenvalues when they stated, "*if the answer is highly sensitive to perturbations, you have probably asked the wrong question.*"

Stability

The *stability* of an algorithm is measured by the error in its output. Let $f : \mathbb{R}^m \to \mathbb{R}^n$ be a problem to be solved, as in the previous section, and let $\tilde{f}$ be an actual algorithm for solving the problem. 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 called *stable* if its relative forward error is small.[1]

As an example, consider again NumPy's root-finding algorithm that we used to investigate the Wilkinson polynomial. The exact roots of $w(x)$ are clearly $1, 2, \ldots, 20$. Had we not known this, we could have tried computing the roots from the coefficients using `np.roots()` (without perturbing the coefficients at all).

```
# w_coeffs holds the coefficients and w_roots holds the true roots.
>>> computed_roots = np.sort(np.roots(np.poly1d(w_coeffs)))
>>> print(computed_roots[:6])       # The computed roots are close to integers.
[ 1.          2.          3.          3.99999999  5.00000076  5.99998749]

# Compute the forward error.
>>> forward_error = la.norm(w_roots - computed_roots)
>>> print(forward_error)
0.020612653126379665

# Compute the relative forward error.
>>> forward_error / la.norm(w_roots)
0.00038476268486104599              # The error is nice and small.
```

This analysis suggests that `np.roots()` is a stable algorithm, so large condition numbers of Problem 2 really are due to the poor conditioning of the problem, not the way in which the problem was solved.

> **Note**
>
> Conditioning is a property of a **problem** to be solved, such as finding the roots of a polynomial or calculating eigenvalues. Stability is a property of an **algorithm** to solve a problem, such as `np.roots()` or `scipy.linalg.eig()`. If a problem is ill-conditioned, any algorithm used to solve that problem may result in suspicious solutions, even if that algorithm is stable.

Least Squares

The *ordinary least squares* (OLS) problem is to find the $\mathbf{x}$ that minimizes $\|A\mathbf{x} - \mathbf{b}\|_2$ for fixed A and $\mathbf{b}$. It can be shown that an equivalent problem is finding the solution of $A^\mathsf{H}A\mathbf{x} = A^\mathsf{H}\mathbf{b}$, called the *normal equations*. A common application of least squares is polynomial approximation. Given a set of m data points $\{(x_k, y_k)\}_{k=1}^m$, the goal is to find the set of coefficients $\{c_i\}_{i=0}^n$ such that

$$y_k \approx c_n x_k^n + c_{n-1} x_k^{n-1} + \cdots + c_2 x_k^2 + c_1 x_k + c_0$$

[1] See the Additional Material section for alternative (and more rigorous) definitions of algorithmic stability.

for all k, with the smallest possible error. These m linear equations yield the following linear system.

$$A\mathbf{x} = \begin{bmatrix} x_1^n & x_1^{n-1} & \cdots & x_1^2 & x_1 & 1 \\ x_2^n & x_2^{n-1} & \cdots & x_2^2 & x_2 & 1 \\ x_3^n & x_3^{n-1} & \cdots & x_3^2 & x_3 & 1 \\ \vdots & \vdots & & \vdots & \vdots & \vdots \\ x_m^n & x_m^{n-1} & \cdots & x_m^2 & x_m & 1 \end{bmatrix} \begin{bmatrix} c_n \\ c_{n-1} \\ \vdots \\ c_2 \\ c_1 \\ c_0 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{bmatrix} = \mathbf{b} \tag{10.5}$$

Problem 5. Write a function that accepts an integer n. Solve for the coefficients of the polynomial of degree n that best fits the data found in `stability_data.npy`. Use two approaches to get the least squares solution:

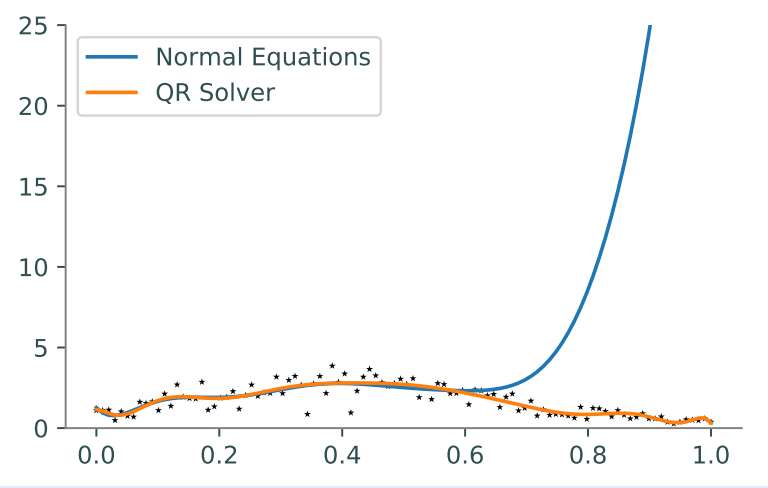
1. Use `la.inv()` to solve the normal equations: $\mathbf{x} = (A^\mathsf{T}A)^{-1}A^\mathsf{T}\mathbf{b}$. Although this approach seems intuitive, it is actually highly unstable and can return an answer with a very large forward error.
2. Use `la.qr()` with `mode='economic'` and `la.solve_triangular()` to solve the system $R\mathbf{x} = Q^\mathsf{T}\mathbf{b}$, which is equivalent to solving the normal equations. This algorithm has the advantage of being stable.

Load the data and set up the system (10.5) with the following code.

```
xk, yk = np.load("stability_data.npy").T
A = np.vander(xk, n+1)
```

Plot the resulting polynomials together with the raw data points. Return the forward error $\|A\mathbf{x} - \mathbf{b}\|_2$ of both approximations.
(Hint: The function `np.polyval()` will be helpful for plotting the resulting polynomials.)

Test your function using various values of n, taking special note of what happens for values of n near 14 (pictured below).



Catastrophic Cancellation

When a computer takes the difference of two very similar numbers, the result is often stored with a small number of significant digits and the tiniest bit of information is lost. However, these small errors can propagate into large errors later down the line. This phenomenon is called *catastrophic cancellation*, and is a common cause for numerical instability.

Catastrophic cancellation is a potential problem whenever floats or large integers that are very close to one another are subtracted. This problem can be avoided by either rewriting the program to not use subtraction, or by increasing the number of significant digits that the computer tracks.

For example, consider the simple problem of computing $\sqrt{a} - \sqrt{b}$. The computation can be done directly with subtraction, or by performing the equivalent division

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

```
>>> from math import sqrt            # np.sqrt() fails for very large numbers.

>>> a = 10**20 + 1
>>> b = 10**20
>>> sqrt(a) - sqrt(b)                # Do the subtraction directly.
0.0                                  # a != b, so information has been lost.

>>> (a - b) / (sqrt(a) + sqrt(b))    # Use the alternative formulation.
5e-11                                # Much better!
```

In this example, a and b are distinct enough that the computer can still tell that $a - b = 1$, but $\sqrt{a}$ and $\sqrt{b}$ are so close to each other that $\sqrt{a} - \sqrt{b}$ is computed as 0.

Problem 6. Let $I(n) = \int_0^1 x^n e^{x-1} dx$. It can be shown that for a positive integer n,

$$I(n) = (-1)^n !n + (-1)^{n+1} \frac{n!}{e}, \tag{10.6}$$

where $!n = n! \sum_{k=0}^{n} \frac{(-1)^k}{k!}$ is the *subfactorial* of n. Write a function to do the following.

1. Use SymPy's `sy.integrate()` to evaluate the integral form of $I(n)$ for $n = 5, 10, \dots, 50$. Convert the symbolic results of each integration to a float. Since this is done symbolically, these values can be accepted as the true values of $I(n)$.
   (Hint: be careful that the values of n in the integrand are actual integers, not floats.)

2. Use (10.6) to compute $I(n)$ for the same values of n. Use `sy.subfactorial()` to compute $!n$ and `sy.factorial()` to compute $n!$.
   (Hint: be careful to only pass actual integers to these functions.)

3. Plot the relative forward error of the results computed in step 2 at each of the given values of n. Use a log scale on the y-axis. Is (10.6) a stable way to compute $I(n)$? Why?

The examples presented in this lab are just a few of the ways that a mathematical problem can turn into a computational train wreck. Always use stable algorithms when possible, and remember to check if problems are well conditioned or not.

Additional Material

Other Notions of Stability

The definition of stability can be made more rigorous in the following way. Let f be a problem to solve and $\tilde{f}$ an algorithm to solve it. If for every $\mathbf{x}$ in the domain there exists a $\tilde{\mathbf{x}}$ such that

$$\frac{\|\tilde{\mathbf{x}} - \mathbf{x}\|}{\|\mathbf{x}\|} \quad \text{and} \quad \frac{\|\tilde{f}(\mathbf{x}) - f(\tilde{\mathbf{x}})\|}{\|f(\tilde{\mathbf{x}})\|}$$

are small (close to $\epsilon_{\text{machine}} \approx 10^{-16}$), then $\tilde{f}$ is called stable. In other words, "A stable algorithm gives nearly the right answer to nearly the right question" (Trefethen, Bao, 104). Note carefully that the quantity on the right is slightly different from the plain forward error introduced earlier.

Stability is desirable, but plain stability isn't the best possible condition. For example, if for every input $\mathbf{x}$ there exists a $\tilde{\mathbf{x}}$ such that $\|\tilde{\mathbf{x}} - \mathbf{x}\|/\|\mathbf{x}\|$ is small and $\tilde{f}(\mathbf{x}) = f(\tilde{\mathbf{x}})$ exactly, then $\tilde{f}$ is called *backward stable*. Thus "A backward stable algorithm gives exactly the right answer to nearly the right question" (Trefethen, Bao, 104). Backward stable algorithms are generally more trustworthy than stable algorithms, but they are also less common.

Stabilty of Linear System Solvers

The algorithms presented so far in this manual have different levels of stability. The LU decomposition (with pivoting) is usually very good, but there are some pathological examples of matrices that can cause it to break down. Even so, `scipy.linalg.solve()` uses the LU decomposition. The QR decomposition (also with pivoting) is generally considered to be a better option than the LU decomposition and is more stable. However, solving a linear system using the SVD is even more stable than using the QR decomposition. For this reason, `scipy.linalg.lstsq()` uses the SVD.