Stability and Condition Numbers

Math 131: Numerical Analysis

J.C. Meza

2/1/24

Section 1

Introduction: Algorithms

Topics Covered:

- What is an algorithm?
- What is a stable algorithm?
- What types of problems are amenable to accurate solutions?
- What is the condition number of a problem?

What is an algorithm?

- Algorithms have a long history dating back over 4000 years. The name itself is said to be derived from the mathematician al-Khwārizmī (c.780-c.850), who wrote a book on the subject.
- One can think of an algorithm as a step by step procedure for solving some problem. A common analogy is that of a recipe in which one is given explicit instructions on how to make something.
- Can also think of an algorithm as a sytematic calculation or process for achieving some desired result.
- More recently, the notion of *finiteness* has been added to the concept although it is not essential.

For our purposes we will use the following definition:

An ***algorithm*** is a well-defined process that takes an input (or inputs) and produces an output in a finite-number of steps or time.

Desired Properties of Algorithms

- What kind of properties should a good numerical algorithm have?
- While one can come up with a long list of desirable characteristics for an algorithm, we will concentrate on three that were briefly introduced on the first day of class:
  1. Accuracy
  2. Efficiency
  3. Robustness
- Let's take a look at each of these in turn.

Section 2

Accuracy

Accuracy of an algorithm

- Since a numerical algorithm is designed to solve some mathematical problem, one of the key characteristics should be that it provides us with an accurate approximation to the true solution, whatever that might be.
- We've already seen some examples of these and how we can use error analysis to allow us to predict the behavior of an algorithm.
- Here we are free to use any of a number of measures (metrics): absolute error, relative, error, residual, etc.

Section 3

Efficiency

Preliminaries: Big O Notation

- Big O Notation is frequently used both in mathematics and computer science.
- While it can be confusing at first, the thing to remember is that it is mostly a means to characterize and understand how an algorithm or an approximation behaves asymptotically.
- In fact, one of the chief advantages to using this notation is that it hides some of the detailed information that isn't usually useful to the analysis.

Uses of Big O Notation

There are two main uses that you will see for Big O Notation in this class.

1. Understand how an approximation to a given function behaves - for example, when we use Taylor's Theorem to approximate a function. In these cases, we usually use h or x for our notation and implicitly assume that h or x are small or tending to zero.
2. The computational workload of an algorithm as a function of the dimension of the problem, typically denoted by n. In these cases, we usually assume that n is large or tending to infinity.

The second case is more prominent in computer science applications when analyzing the complexity of an algorithm, but we will have use for it as well in numerical analysis.

Motivation for Big O in computational workload

- It is important to understand how the workload will behave as the size of a problem increases. An algorithm that works fine on a problem of dimension $1, 2$, or 3, can be prohibitively expensive when applied to a problem with dimension $1,000,000$.
- Many algorithms can be quite complicated. The power of Big O notation is that it is a high-level description that allows us to quickly say something about an algorithm or an approximation without getting into all the messy details.
- In addition, counting every single arithmetic operation could be tedious (and with modern computers not as important).
- Big O can be used to compare two algorithms to see which one is more efficient.

One definition for Big O

We say that $g(n)$ is $O(f(n))$ if:

$$g(n) \leq C\ f(n), \quad \forall \text{ integers } n \geq n_0$$

for some constants $C > 0, n_0 > 0$.

Example:

$$3n^3 + 2n^2 + 5 = O(n^3)$$

because

$$3n^3 + 2n^2 + 5 \leq 4n^3, \quad n \geq 3$$

- It should also be noted that neither one of the constants C, n_0 are (usually) known.
- For more information see [@knuth1997],

Caution on Big O Notation

- In all cases the statements are understood to be ***in the limit.***
- For specific cases there may be (and usually are) counterexamples.

Caution

An important point to remember is that we are only interested with how the computational workload of an algorithm behaves as the dimension increases, and ***not with the details of the computation***.

Example of Big O

Suppose we wanted to compute the mean of a set of n numbers:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

It is easy to see that the formula entails adding n numbers, which amounts to $n-1$ additions. We then have to divide by n, so there's 1 division. The total is therefore n floating point operations (*flops*).

Everything else being equal, that's also the computational workload for this simple algorithm.

- If $n = 100$, then we have 100 *flops*;
- if $n = 1,000,000$ then we have $1,000,000$ *flops*.

We say that this algorithm is $O(n)$, because the workload increases linearly with the dimension of the problem n.

Example 2 of Big O

Let's consider the problem of evaluating a polynomial given by:

$$p_n(x) = c_0 + c_1 x + c_2 x^2 + \ldots c_n x^n.$$

- A naive algorithm would require $O(n^2)$ operations or ***flops*** (floating point operations) as they are called. To see this all you need to do is think about each of the terms in the polynomial in turn and add up all of the operations.
- For example, the last term requires n multiplications: $n-1$ to compute x^n plus one more to multiply by c_n. Each term of lower order will require one less multiplication.
- In all, there are $n + (n-1) + (n-2) + \ldots + 1 = O(n^2)$ multiplications. Finally there are the n additions required to sum up all of the individual terms.

Nested form

Consider rewriting the formula in what is known as the ***nested form***:

$$p_n(x) = c_0 + x(c_1 + x(c_2 + \cdots + x(c_{n-1} + x(c_n))\cdots)),$$

which only requires $O(n)$ operations. (You should work this out for yourself.)

- This may not make much of a difference for small values of n, but one nonetheless needs to be careful as generally speaking we don't know ahead of time how an algorithm will be used.
- For example, this polynomial evaluation could be at the heart of an algorithm that is called millions of times.

A note on Flops

- Historically, the operation count (flops) has been an important measure of efficiency (or at least workload).
- SuperComputing Top500: https://www.top500.org/ lists top 500 supercomputers in the world
- On today's computers, and especially supercomputers, there are many more things to consider when considering computational efficiency:
  - memory access,
  - communication,
  - vectorization/parallelization,
  - etc.

Section 4

Robustness

Robustness in plain terms

- Finally, we would like an algorithm that we can "trust", in the sense that it either gives us an answer or reports back that something went wrong.
- Ideally, an algorithm should also work under most foreseeable circumstances and different values of inputs.
- Another thing to watch out for is the rate of accumulation of errors within an algorithm.

Tip

You should be on the lookout for any calculations that might cause an IEEE floating point exception: overflow, underflow, infinity, NaN.

Section 5

Condition Numbers and Stability

Condition Numbers

- The first concept is that of the problem sensitivity or as it is more commonly referred, ***condition numbers.***
- By this we mean that a problem is ***well-conditioned*** if small changes in the inputs do not lead to large changes in the outputs.
- Otherwise the problem is said to be ***ill-conditioned.***

Ill-Conditioned Problem

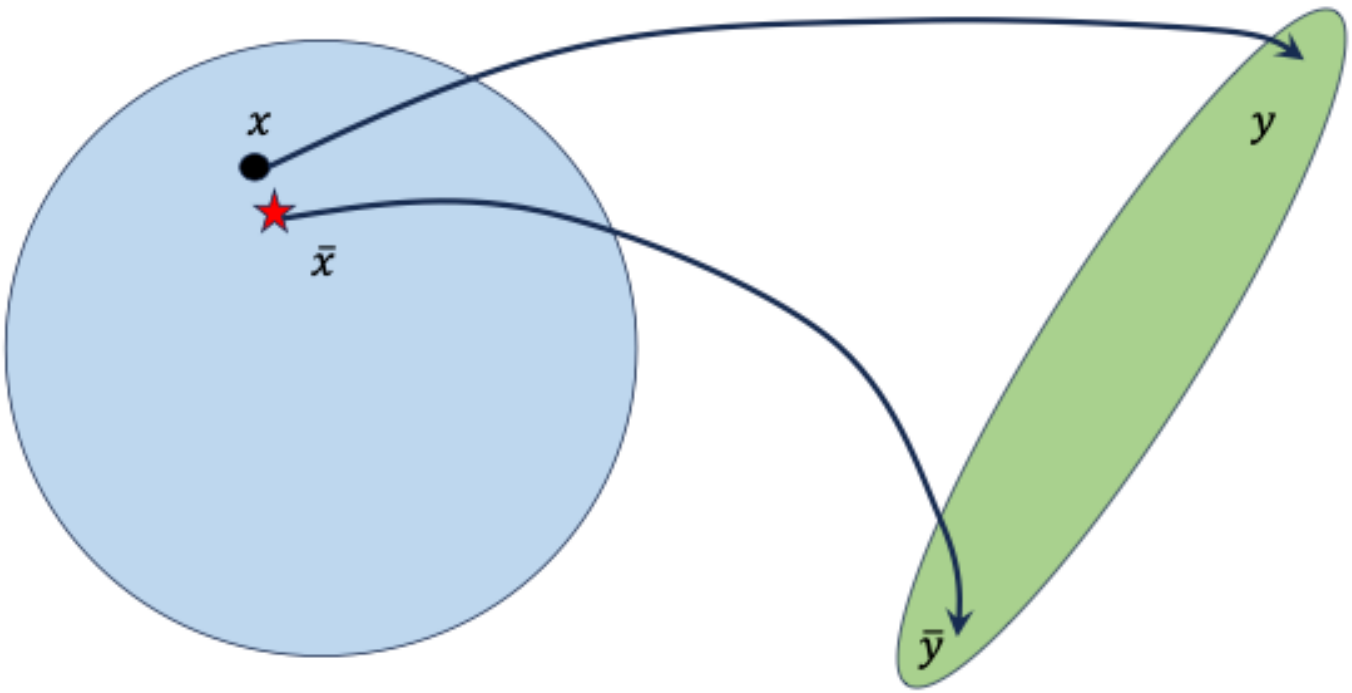

Figure 1: Ill-conditioned problem where small changes in the input can lead to large changes in the output

Example: Well-conditioned problem

Let's take a look at a simple example of a well-conditioned problem first.

Consider the function $f(x) = \sqrt{1+x}$ with $f'(x) = \frac{1}{2\sqrt{1+x}}$.

Let's suppose that we fix $|x| << 1$ and consider $\bar{x} = 0$ as a small perturbation of x, and $y = f(x)$. Then

$$\bar{y} = f(\bar{x}) = \sqrt{1+0} = 1,$$

and the change in the output can be given by

$$y - \bar{y} = \sqrt{1+x} - 1.$$

Example (cont.)

Approximating $\sqrt{1+x}$ by the first 2 terms of a Taylor series about 0,

$$f(x) \approx 1 + \frac{x}{2}.$$

Substituting into the prior equation we can write the change in the output:

$$y - \bar{y} \approx \left(1 + \frac{x}{2}\right) - 1 = \frac{x}{2}.$$

This can then be rewritten in terms of the change in the inputs as:

$$y - \bar{y} \approx \frac{1}{2}(x - \bar{x}).$$

Notice that if $x = 0.001$ then the change in the output is only $y - \bar{y} \approx 0.0005$, one half of what the change in input was.

Condition Number

- We're now ready for a more rigorous definition of condition number.
- First, note that when we introduced the concept of a condition number it was to describe the sensitivity of a problem (function) to change in the input data x.
- As such it seems natural to look at:

$$\frac{\text{Relative change in } f(x)}{\text{Relative change in } x}.$$

Mathematically we can translate this into:

$$\begin{aligned}
\frac{\text{Relative change in } f(x)}{\text{Relative change in } x} &= \frac{\frac{f(x)-f(\bar{x})}{f(x)}}{\frac{x-\bar{x}}{x}} \\
&= \frac{f(x)-f(\bar{x})}{f(x)} \cdot \frac{x}{x-\bar{x}}, \\
&= \frac{f(x)-f(\bar{x})}{x-\bar{x}} \cdot \frac{x}{f(x)}, \\
&\approx \frac{f'(x)\cdot x}{f(x)}.
\end{aligned}$$

This leads us to the following definition:

The quantity

$$\kappa = \left| \frac{f'(x) \cdot x}{f(x)} \right|$$

is called the (relative) condition number of the problem.

- If κ is large, we say that the problem is ***ill-conditioned***.
- More on this important topic later!

Looking ahead

- The condition number of a problem is especially useful in numerical linear algebra, where it can be used to estimate the accuracy of the solutions for systems of linear equations, such as $Ax = b$.
- Also important for many partial differential equations, inverse problems, etc.

Section 6

Stability

Stability of algorithms

- The second important concept is that of stability in an algorithm.
- Here we say that an algorithm is ***stable*** if the output generated is the exact result for a slightly perturbed input.
- Another way of phrasing this is that a stable algorithm solves a ***nearby problem.***

Stability (cont.)

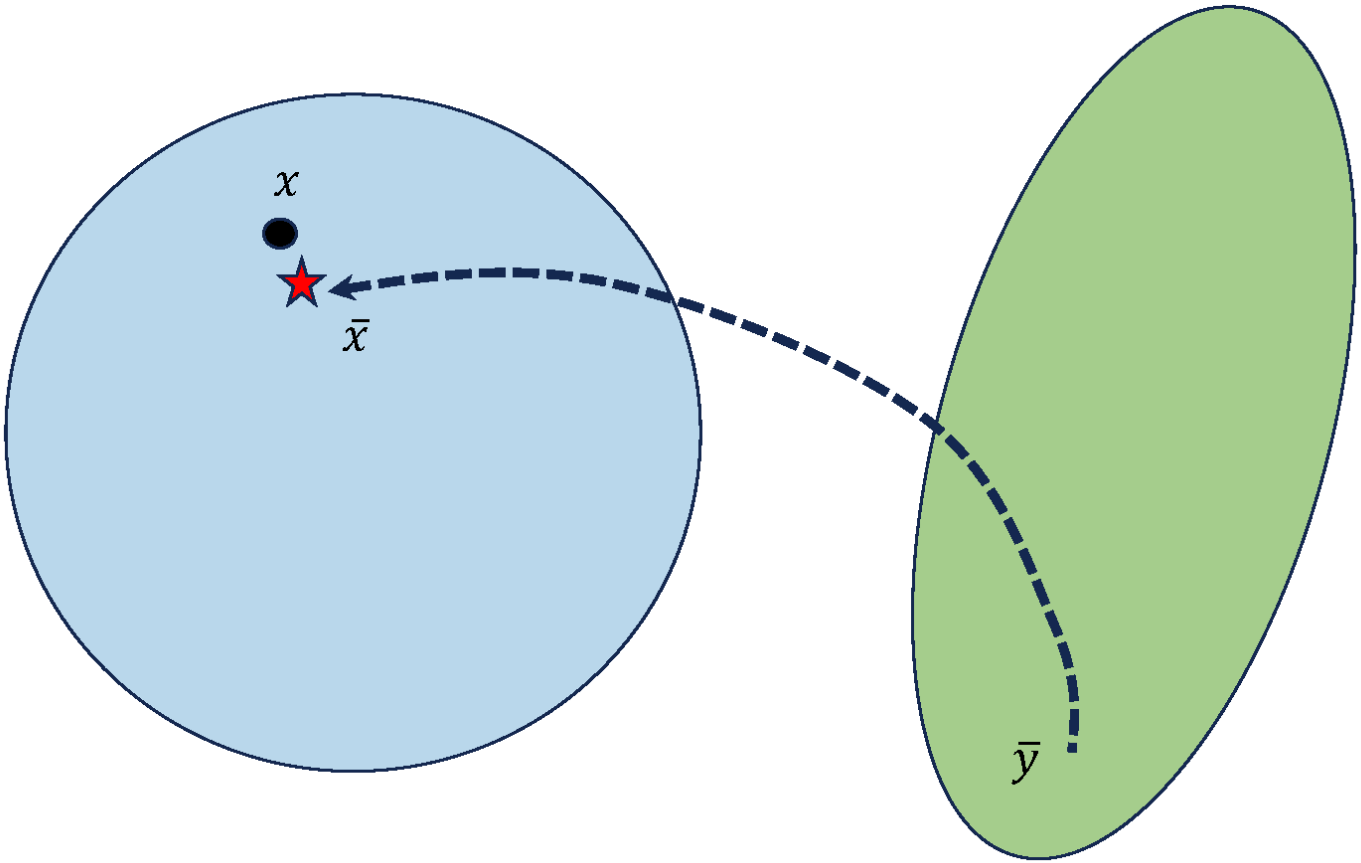

Figure 2: A stable algorithm is one that is the exact solution to a slightly perturbed problem.

Roundoff error and stability

- It is natural to expect that any algorithm will produce some roundoff error. In fact, it is common that accumulation of roundoff error will occur as we progress through a computation.
- If the error grows slowly as the computation proceeds it won't be a problem, but if it grows too fast then we need to be more careful.
- Let E_n be the error at the nth step of some computation. If the error $E_n \approx C \cdot nE_0$, where C is a constant and E_0 is the original error, then the growth of error is said to be ***linear***.

Stability (cont.)

- If the error $E_n \approx C^n E_0$, where C is a constant and E_0 is the original error, then the growth of error is said to be ***exponential***.
- An algorithm that has an exponential error growth rate is said to be ***unstable***.
- We should note that an algorithm could be stable for solving one problem, but unstable for solving another problem.
- The stability of algorithms will become important in our study of differential equations (e.g. IVPs)

Tips on designing stable algorithms

While there are no hard and fast rules, there are several things to watch out for in designing an algorithm. The tips below are paraphrased from the excellent discussion given in Higham [@higham2002].

- Watch out for cancellation errors. Try to avoid subtracting quantities of near equal magnitude, especially if they are contaminated by error.
- Look for different formulations that are mathematically equivalent, but perhaps numerical better.
- It can be useful to write formulas in the form of:

$$x_{new} = x_{old} + \Delta x$$

  where Δx is a small correction to the previous (old) value. You'll see that many numerical methods take this form.

Tips on designing stable algorithms (cont.)

- Minimize the size of intermediate results relative to the final result. It's also a good idea to check the intermediate results when first writing a code.
- Avoid ill-conditioned transformations of the problem.
- Take precautions against underflow and overflow.

Summary

This lesson covered the fundamental concepts of condition numbers of numerical problems and stability of algorithms. Key takeaway messages include:

- The condition number of a problem, which speaks to the sensitivity of the outputs to the inputs is related to the ***relative change in the output of a problem to the relative change in the inputs.***
- All algorithms will generate errors as a consequence of doing computer arithmetic. If the error growth rate is reasonable, for example linear with respect to the number of steps in the calculation, then we can expect good results (if the problem is also well-conditioned).
- If the error growth rate is exponential instead, then the algorithm is said to be ***unstable***.
- An algorithm can be stable for one type of problem but unstable for another type of problem.

References

1. Math 131 Lecture Notes
2. Big O notation, https://en.wikipedia.org/wiki/Big_O_notation
3. The Art of Computer Programming, [@knuth1997]