Math 563 Lecture notes
Introduction: what is numerical analysis?

Spring 2020

The point: The goal here is to introduce the themes of the course and get a sense of computational analysis by way of example. A minimal introduction to computer arithmetic is also provided, omitting most of the messy details that are distracting.

Related reading: For all the un-needed (but interesting) details on computer arithmetic, see Section 2.5 of Quarteroni. Other examples here will be addressed in detail later in the course.

Preface

The field of **numerical analysis**, broadly speaking, is concerned with obtaining approximate solutions to mathematical problems that can be implemented on a computer.[1] The theory of approximation can be surprisingly deep and elegant, given the messiness of the problems it seeks to solve. Under the wide umbrella of the subject is both pure analysis and more practical computational work. Some examples include:

- Theory (the analysis)
  - Convergence (limits of sequences that approach the true solution)
  - Finite-dimensional spaces for approximation
  - Discrete analogues of continuous processes
- Applied (somewhere in between)
  - Derivation of (practical) numerical methods
  - Intuition for interpreting results, measuring error
  - Adapting/generalizing methods to get desired properties
- Implementation (the numerical)
  - Translating methods to actual code
  - Efficient implementation
  - Developing packages for computing (COMSOL, Matlab, R etc.)

In this course, we focus more on the first two aspects and address the last one in less depth. Hopefully, you will be convinced by the end that an understanding of the underlying mathematics is invaluable, even when one is concerned with practical results.

[1] The same was true centuries ago - in the 1700s, advances in astronomy demanded precise calculations, which motivated numerical tools like Napier's tables of logarithms. In that time, a 'computer' was an actual person tasked with doing the computations which is, thankfully, no longer the case

1 A quick note on error

1.1 Relative error

Suppose $\tilde{x} \approx x$ is an approximation to x. There are two basic 'measures' of error in practice:

$$\text{absolute error} = |\tilde{x} - x|,$$

$$\text{relative error} = \frac{|\tilde{x} - x|}{|x|}.$$

Which error is appropriate depends on context - we'll find throughout the course that both are useful, and some judgment is required to pick the right one. Many heuristics and theorems make statements about one or the other, and it is important to know the difference.

Scaling (in)variance: Note that in particular, since relative error is 'relative' to the base value, it is independent of a trivial scaling; i.e. if $\tilde{x}$ and x are replaced by $a\tilde{x}$ and ax then the relative error is the same, but the absolute error gets scaled by a.

Aside (Why not both at once?): A convenient trick is to use 'combined' measure

$$\text{(unnamed) error} = \frac{|\tilde{x} - x|}{1 + |x|} \approx \begin{cases} \text{abs. error} & \text{if } |x| \ll 1 \\ \text{rel. error} & \text{if } |x| \gg 1 \end{cases}.$$

This measure can occasionally be used to write a desired error tolerance in terms of a single quantity instead of two separate conditions for the relative and absolute error.

1.2 Cancellation

There are operations that are dangerous when there is an associated error. This means that in designing algorithms, we will need to keep in mind what formulas are 'good' or 'bad' for computation, even when they are equivalent in theory.

One notable culprit is the dramatically named **catastrophic cancellation**. Suppose, for example, we are working with quantities known to **three** decimal digits and end up computing

$$\frac{1.234\cdots - 1.230\cdots}{1.001\cdots - 1.000\cdots} = \frac{0.04\cdots}{0.01\cdots} = 4.\cdots .$$

The ellipses indicate 'insignificant' digits (the values are only known to be accurate to the first three). The result only has **one** digit of accuracy - a cause for concern. In general, we see that if f is continuous then

$$x \approx y \implies \frac{f(x) - f(y)}{x - y} \text{ leads to loss of relative accuracy.}$$

Unfortunately, such expressions are common - so one has to be careful. For example, consider the seemingly innocent quadratic formula when a is small. Then

$$r = \frac{-1 + \sqrt{1 - 4ac}}{2a} = \frac{-1 + (\approx 1)}{\text{small}}$$

which leads to catastrophic cancellation.

However, a simple manipulation fixes this issue:

$$r = \frac{-1+\sqrt{1-4ac}}{2a}\frac{-1-\sqrt{1-4ac}}{-1-\sqrt{1-4ac}} = \frac{4ac}{-1-\sqrt{1-4ac}}.$$

The two formulas are theoretically equivalent, but computationally different!

The small correction rule: However, it is often true that a low-accuracy number is okay when it is a 'small correction' to the last few digits. An expression like the

$$x_{n+1} = x_n + \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}} \tag{1}$$

(the secant method for root finding) is not problematic. This iteration (ideally) converges to a zero x^* of a function $f(x)$. Assuming that $x_n \to x^*$ and that we are close to the zero (so $x_n \approx x^*$), the difference quotient in (1) has cancellation since $x_n, x_{n-1} \approx x^*$.

But this is not actually a problem because by this point in the iteration, only some digits of x_n need updating - which does not require full significant digits.

For instance, if $x_n \to 1$ and $x_9 = 1.00234$ then it may look like

$$x_{10} = 1.00234 + (-0.00221)$$
$$x_{11} = 1.00013 + (-0.00010)$$

and so on, so even if the 'updates' have less accuracy (three and two for $n = 10$ and $n = 11$ here). In short, 'small corrections' only need to be accurate to the number of digits they are updating.

2 Floating point arithmetic (briefly)

A computer must store a finite amount of data - and as such, all numbers and arithmetic are done with some error. At times, this 'finite precision' issue is minor, and the theory can largely ignore it (accepting there will be error in the practical answer). We will typically develop theory without too much concern for rounding error unless it really matters.

It is important to be able to recognize rounding error and understand how it manifests (and have some intuition for when it is important - e.g. the catastrophic cancellation example above).

Let us define the set of **machine numbers** to be the number system used by a typical computer/language - that is, a 'double precision' number (a `double` in `C/C++`, and the default numeric type in `python/matlab`).[2] Such a number is stored in memory in the 'floating point' form

$$\text{(base 2)} \quad \pm 1.d_1 d_2 \cdots d_N \times 2^e = \left(1 + \sum_{k=1}^{n} d_k 2^{-k}\right) 2^e, \qquad m \le e \le M \tag{2}$$

where the d_i's are binary digits (zero or one) and $N = 52$ and m, M are limits for the exponent.[3]

[2] You could, of course, use a `float` (single precision) or even a 'quad-precision' number, but at this point, a `double` is standard and there is rarely a need for anything else.

[3] For all the nasty details, consult the IEEE standard for double precision numbers, where this is defined.

Further, let us define the 'rounding' operation

$$\mathrm{fl}(x) = \text{'nearest' machine number (2) to } x \in \mathbb{R}.$$

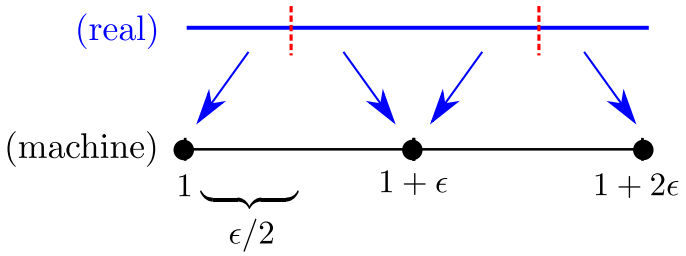
Because there are only N binary digits in the machine number, the numbers are a finite sequence. Starting from 1, the first few values are

$$1, \quad 1.\underbrace{00\cdots0}_{N-1 \text{ zeros}}1 = 1 + 2^{-N}, \quad \cdots$$

The distance from 1 to the next largest number is important and has a special name:

$$\textbf{machine epsilon} = \epsilon_m := 2^{-N} \quad (\approx 2.2 \times 10^{-16} \text{ for a } \texttt{double})$$

The 'rounding error' incurred by representing a real number x by a machine number $\mathrm{fl}(x)$ is bounded above by half this distance, as the sketch below indicates.



Moreover (guaranteed by implementation on processors), $\epsilon_m/2$ represents a bound on the **relative** error in basic arithmetic on the computer. The value $\mathtt{u}_m = \epsilon_m/2$ is called the **rounding unit**.

Rule (rounding error): Let $\mathtt{u}_m = \epsilon_m/2$ be the rounding unit ($\approx 1.1 \times 10^{-16}$ for a `double`).

(i) The (relative) 'rounding error' in representing a number is bounded by $\mathtt{u}_m$:

$$\mathrm{fl}(x) = x(1+\delta), \quad |\delta| \le \mathtt{u}_m.$$

(ii) The error in arithmetic operations $+, -, \times, /$ have a similar bound, e.g. if x, y are machine numbers then it holds that

$$\mathrm{fl}(x+y) = (x+y)(1+\delta), \quad |\delta| \le \mathtt{u}_m.$$

Short version: Rounding/arithmetic produce a 'machine epsilon' sized relative error.

The presence of these rounding errors means that numerical codes will invariably have some error that accumulates as operations are done. For instance, consider computing the sum

$$y = \sum_{n=1}^{1000} \frac{(-1)^n}{2n+1}.$$

There are about 1000 additions (plus the reciprocals), so we expect at least a $1000 \times \epsilon_m \approx 10^{-13}$-sized error (order of magnitude). The calculation

$$y = \frac{\sin(1)}{10^{-5}} + \cos(1)$$

would have an absolute error of about 10^{-11} (a relative error about $\mathtt{u}_m \approx 10^{-16}$ if an accurate method for sin is used). This gives you a rule of thumb that can indicate when a small discrepancy is 'just rounding error' and not anything deeper. Most of the time, such an error can be accepted.

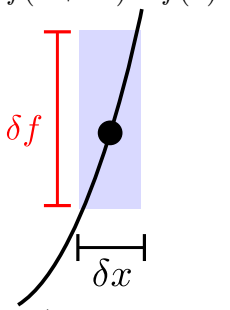
The more interesting part, as we will see, is making sure that these errors are not amplified by the algorithm to the point of spoiling the solution.

2.1 Condition/sensitivity

Suppose we wish to solve a problem with an input x and output $f(x)$. If the value of x is changed by an amount δx of size $|\delta x| \leq \epsilon$, then the output f changes by an amount $\delta f = f(x + \delta x) - f(x)$.

Conditioning: A problem is called **well-conditioned** if small changes in the input lead to small changes in the output (δx small implies δf small, with 'small' in whatever sense is relevant).

If the problem is sensitive to small changes in δx - to the point of computational difficulty - the problem is called **ill-conditioned**.



For each type of problem, there is a measure of condition - the **condition number**). Given δx of this small size, we have that

$$\text{relative sensitivity to } \delta x \;=\; \sup_{|\delta x| \leq \epsilon} \left| \frac{\delta f / f}{\delta x / x} \right|.$$

Taking the limit as $\epsilon \to 0$ gives the desired measure of the system's sensitivity:

$$\text{(relative) condition number} = \lim_{\epsilon \searrow 0} \sup_{|\delta x| \leq \epsilon} \left| \frac{(f(x + \delta x) - f(x))/f(x)}{\delta x / x} \right| \tag{3}$$

The problem is ill-conditioned if this number is large, since then a small error made in the input can lead to a drastic difference in the output.

Key point (ill-conditioned problems): Unfortunately, the poor condition is inherent to the problem, so a correct algorithm would likely inherit the same sensitivity. For this reason, ill-conditioned problems are hard to solve numerically (and best avoided if possible!).

For example, consider the problem of evaluating

$$f(x) = \tan x, \qquad x \approx \pi/2.$$

Suppose, say, we take $x_1 = \pi/2 - 0.001$ and $x_2 = \pi/2 - 0.002$. Then

$$|x_1 - x_2| = 0.001, \quad |f(x_1) - f(x_2)| = 500$$

so the small difference in the x-values leads to large differences in f.

In general, for evaluating a real function $f(x)$, the limit in the condition number (3) reduces to a derivative, resulting in

$$\text{(rel.) cond. number} = \frac{x f'(x)}{f(x)}$$

The absolute version of this is just $f'(x)$. A large $f'(x)$ causes trouble.

3 Numerical instability

The sensitivity of an algorithm may differ from the condition of the problem. An algorithm is said to be **(numerically) stable** if small errors in the inputs and at each step lead to small errors in the solution. Because of the presence of errors, too much amplification of errors by the algorithm will render it useless! An algorithm that amplifies errors (too much) is called **unstable**.

Let's look at a somewhat contrived example. Suppose a problem has the solution

$$a_n = 3^{-n}$$

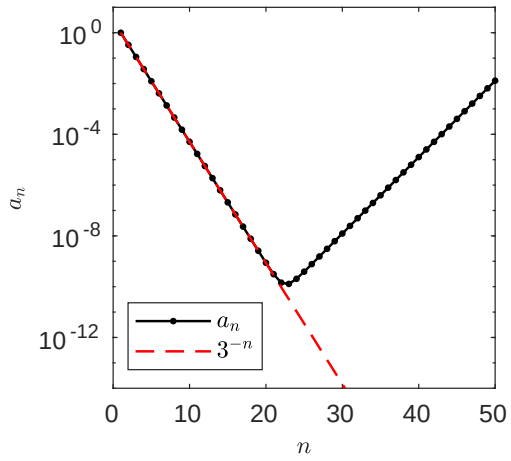
A method (not a good one!) that can be used to compute 3^{-n} it is the recurrence

$$a_{n+1} = \frac{7}{3}a_n - \frac{2}{3}a_{n-1}, \qquad a_0 = 1, \quad a_1 = 1/3. \tag{4}$$

This 'algorithm' generates $a_n = 1/3^n$ in theory. Suppose now there is an initial error:

$$\tilde{a}_0 = 1, \quad \tilde{a}_1 = 1/3 + \delta$$

where $|\delta| < \mathtt{u}_m \approx 1.1 \cdot 10^{-16}$ (on the order of rounding error). Running the code leads to disaster (**??**). Eventually, the computed solution begins to grow exponentially, even though the initial error in writing `a = 1/3` is on the order of 10^{-16}.



The exact solution[4] to the recurrence for $\tilde{a}_n$ is actually

$$a_n = \left(1 - \frac{3\delta}{5}\right) 3^{-n} + \frac{3\delta}{5} 2^n.$$

There is a 'spurious' term that grows (not part of the 3^{-n} we want), and it spoils the sequence since the initial δ-sized error (small) grows exponentially:

$$a_n \sim \frac{3\delta}{5} 2^n \text{ as } n \to \infty.$$

We say this recurrence is a **numerically unstable** method for computing 3^{-n}, because errors become amplified and render the algorithm impractical.[5]

[4] To solve, guess $a_n = r^n$ to find that $r = 1/3$ and $r = 2$ yield solutions; then $a_n = c_1 3^{-n} + c_2 2^n$ and apply ICs.

[5] While this example is contrived on its own, we will see recurrences like this appear in solving ODEs!

4 A motivating example

Note: we'll cover this in more rigorous detail later. A task of obvious importance is computing the derivative of a function $f(x)$ at a point x_0. A basic approach is the **forward difference**

$$D(f,h) := \frac{f(x_0+h)-f(x_0)}{h} \approx f'(x_0).$$

Denote by $E(h)$ the absolute error in the approximation:

$$E(h) = f'(x_0) - D(f,h).$$

Given a function f and point x_0, some key questions are

- How does the accuracy of the approximation relate to h?
- Does the error go to zero as $h \to 0$ or is there a best possible accuracy?

4.1 Two flavors of error

The approximation can be derived by using a Taylor series. Expand $f(x_0+h)$ around x_0:

$$f(x_0+h) = f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + O(h^3)$$

where $O(h^3)$ denotes the h^3 and higher terms in the series. Now solve for $f'(x_0)$ to obtain

$$f'(x_0) = \frac{f(x_0+h)-f(x_0)}{h} - \frac{h}{2}f''(x_0) + O(h^2).$$

We conclude that

$$D(f,h) = f'(x_0) + \frac{f''(x_0)}{2}h + \cdots.$$

The terms to the right of f' are the **truncation error** - the part of the formula that is dropped to get the approximation. From the leading term of the error (the h-term) we see that

$$\text{truncation error} \sim Ch \text{ as } h \to 0.$$

If f were computed exactly, this would also be $E(h)$. But the function f, in practice, is not computed exactly (nor is the subtraction in the numerator) so our analysis must take into account such errors. Let's assume that f is simple and implemented to high accuracy on the computer. Then for any evaluation $\tilde{f}$ of the function,

$$\tilde{f} = f + \delta, \quad |\delta| < \mathtt{u}_m.$$

Then the computed value of D is really

$$\tilde{D}(f,h) = D(f,h) + \frac{\delta_1 - \delta_0}{h}$$

which introduces a 'rounding error' (not part of the truncation error, only present in computation)

$$|\text{'rounding' error}| \leq \frac{\mathtt{u}_m + \mathtt{u}_m}{h} = \frac{2\mathtt{u}_m}{h}.$$

In summary, the total error $E(h)$ has two parts:

$$E(h) = \text{truncation} + \text{`rounding' error} \leq Ch + \frac{2u_m}{h}. \tag{5}$$

The analysis can be continued, but let's instead explore the question numerically, to see how numerical evidence can deduce this result.

Numerical approach: To start, pick a reasonable function/point:

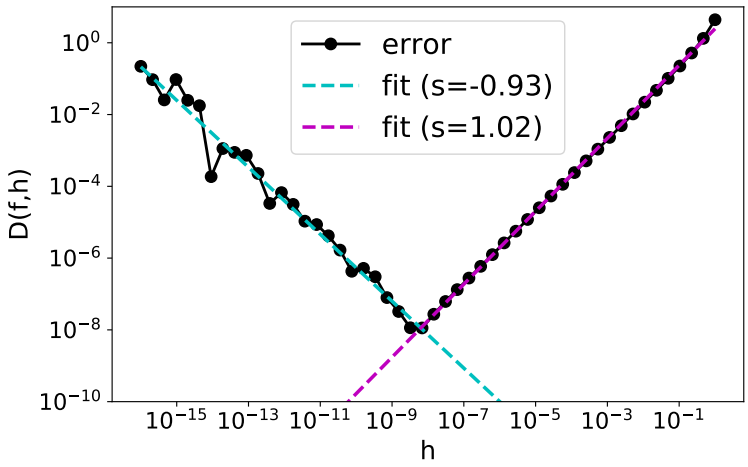
$$f(x) = e^{2x}, \quad x_0 = 0.$$

Since $f'(x_0) = 2$ is known, we can compute the error in the computed $\tilde{D}(f, h)$. A log-log plot (see box below) reveals that the total error

$$E(h) = |f'(x_0) - \tilde{D}(f, h)|$$

has behavior like Ch^p, but there are two distinct cases, separated by $h^* \approx 10^{-8}$:

- For 'not-too-small' values of h (when $h \gg h^*$), the slope is 1 so $E(h) \approx \text{const.} \cdot h$. The error decreases with h, heading towards zero as $h \to 0$.
- However, when h is 'too-small' ($h \ll h^*$), the slope is instead -1 and $E(h) \approx \text{const.}/h$.



To avoid the disastrous (unreliable or error-prone) behavior, we must take $h \gg 10^{-8}$. The minimum error possible is around 10^{-8} also - it cannot be made arbitrarily small. However, when h is above this threshold, the error scales like Ch (which is good to know).

Now this observation only holds for the chosen function, but it gives us some insight into the general rule of thumb. We can then use this idea as a guide when developing the theory. Numerics can often be used this way - to explore the answer to a question before knowing how to answer it mathematically.

Important tool (convergence plots): To see the behavior of $f(h)$ as $h \to 0$, two types of 'convergence plots' can be used:

- If $f(h) = Ch^p$, then $\log f$ vs. $\log h$ should be linear (slope p)

- If $f(h) = Ce^{-ah}$ then $\log f$ vs. h should be linear (slope $-a$)

Use a `loglog` or `semilogy` plot to get the axes right (not a `plot` of $\log f$!). A linear fit will give you the value of the parameters (or pick two points if you are confident the data is really linear).

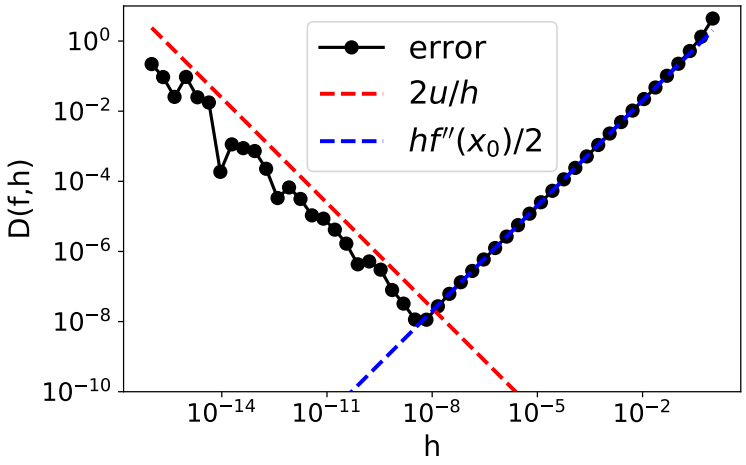
A good table of data can also be used (to be emphasized later!).

Theory, briefly: Plotting the functions

$$\frac{h}{2}f''(x_0), \qquad \frac{2u_m}{h},$$

we see that the theoretical prediction (5) does indeed match the computed results.

The result for the rounding error is just a bound, which is reflected in the plot by the unpredictable behavior (the jagged lines); however, it still follows the trend of the bound in the worst case (scaling like $1/h$) as shown below.



5 Algorithms

A numerical **algorithm** is a sequence of steps that takes an input and returns some output that (approximately) solves some mathematical problem. There are three levels:

$$\text{Mathematical description} \rightarrow \text{Algorithm (pseudocode)} \rightarrow \text{Implementation (code)}$$

The code has complete computational detail (variables, memory, control structure etc.), and on the other end, the method is described in abstract terms - the math is defined but not the code. We will often work with this'high-level' description of the algorithm.

In between is pseudocode. At this level, the computational steps are well defined but the implementation is not. Pseudocode is typically specific enough that any two users who implement it should get code that does the same thing.[6]

Example (algorithms and pseudocode): A trivial algorithm to illustrate the point. The *Fibonacci numbers* are defined by

$$F_0 = F_1 = 1, \qquad F_j = F_{j-1} + F_{j-2}, \quad j \geq 2. \tag{6}$$

This is the 'mathematical description' - it tells us exactly how to generate the numbers. But it does not specify how it should be done. There is not much to say here; however, one decision must be made: to generate all the numbers or just the n-th. Algorithm 1 takes an integer $N > 0$ and generates all the Fibonacci numbers up to F_N (typeset using the `algorithmcx` package).

If we need only the N-th number, storing the array is inefficient. Instead, we **overwrite** in the loop, storing only a few variables instead of a length N array (Algorithm 2).

Algorithm 1 Fibonacci numbers: Version 1

Input: $N \geq 2$, array F of length $N+1$
Output: F stores $F_0, \cdots F_N$
  $F[0] \leftarrow 1$ and $F[1] \leftarrow 1$
  for $i = 2, \cdots n$ **do**
    $F[i] \leftarrow F[i-1] + F[i-2]$
  end for

Algorithm 2 Fibonacci numbers: Version 2

Input: $N \geq 2$
Output: the N-th Fibonacci number F_N
  $y \leftarrow 1$ and $z \leftarrow 1$ ▷ F_{i-2} and F_{i-1}
  $t \leftarrow 0$ ▷ temp. variable
  for $i = 2, \cdots N-1$ **do**
    $t \leftarrow z$
    $z \leftarrow z + y$
    $y \leftarrow t$
  end for
  return y

---

[6] For instance, by design, Matlab code is already pseudocode-like. Use of Matlab libraries or Matlab-specific vectorization would be implementation details.

5.1 Practical goals

The mathematical theory is motivated by a need to design good numerical methods. There is no perfect algorithm for a given (non-trivial) problem, so each property has a cost - gaining one typically means losing another.

Here are some of the main (overlapping) concerns. The meaning depends on context - different problems demand different properties, and the needs of the user (how accurate or fast does it need to be?) matter also.

- **Efficiency** and **accuracy** tradeoffs:
  - Given a tolerance ϵ, can the algorithm find a solution to within ϵ?
  - How much time and memory (space) is required to solve the problem at a given accuracy?
  - How does the time/memory scale with problem complexity?
  - Can the correctness of the solution be verified (reliable error bounds)?
- **Robustness/scope**
  - What is the scope of the algorithm - what problems can it solve? How general is it?
  - Can the algorithm adapt to deal with hard cases or does the user need to step in?
  - Does the user need to see the inner workings is it a 'black box'?
- **Stability:**
  - Does the algorithm keep rounding and other errors under control?
  - We'll have much more to say about this later!

Example: Consider Newton's method

$$x_{n+1} = x_n - \frac{f(x)}{f'(x)}, \quad x_0 = ?$$

which finds a zero x^* of a function $f(x)$ (the sequence x_n, hopefully, converges to x^*).

On efficiency/accuracy, we want to know whether x_n converges to x^* as $n \to \infty$ and describe how fast the error $|x_n - x^*|$ goes to zero. Given a maximum allowed error ϵ, how many iterations are needed to make the error less than ϵ? (answer: the error goes to zero 'quadratically'; the number of significant digits doubles at each step if the function is nice).

On scope, we want to know for which functions $f(x)$ the algorithm works (answer: $f \in C^2$ or sometimes less, but f must be at least differentiable; only fast when x^* is a simple zero)

On stability, we want to know if an initial error in x_0 or errors in evaluating f are amplified as the algorithm progresses. (answer: it works fine).

On robustness, we want to know if the algorithm may fail when given a not-good starting 'guess' x_0. (answer: not very robust; hard to know how close x_0 must be and diverges if too far from the root; requires some human attention).

In short, Newton's method is **extremely accurate/efficient** when x_0 is chosen close to x^* and the function is smooth and the zero is simple. However, it is not very robust unless coupled with a good scheme for choosing x_0 (which is hard to design).