MA3227 Numerical Analysis II

Lecture 1: Big O Notation

Simon Etter

National University of Singapore

Semester II, AY 2020/2021

Big O Notation

Introduction

We will repeatedly encounter the following situation in this module.

- We have a mathematical function $F(X)$ that we would like to evaluate with the help of a computer.
- Evaluating $F(X)$ exactly is infeasible because the input, output and/or some intermediate values of this function contain an excessive amount of information.
- We therefore devise algorithms $F(X, n)$ which depend on one or more "effort parameters" n such that

  $$\lim_{n \to \infty} F(X, n) = F(X)$$

  but evaluating $F(X, n)$ becomes increasingly time-consuming for growing n.

This lecture will introduce several ideas and definitions to facilitate the discussion of convergence in later lectures.

Big O Notation

Example: floating-point multiplication

Consider the problem of computing the product of two real numbers, i.e. the problem of evaluating the function

$$\text{prod} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}, \quad (x, y) \mapsto xy.$$

Real numbers have an infinite number of nonzero digits in general; hence both the input and output of this function contain an infinite amount of information and cannot be represented exactly on a finite machine.

Computers circumvent this problem by storing only a finite number number of significant digits for each real number.

Devising a $\text{prod}(x, y, n)$ algorithm which takes two "n significant digits"-numbers x, y as inputs and produces a single "n significant digits"-number xy as output is then fairly straightforward. However, this algorithm is only an approximation to the exact $\text{prod}(x, y)$ function.

The number of digits n is thus an effort parameter of the $\text{prod}(x, y, n)$ algorithm; as we increase n, the results become more accurate but the computations become more costly.

I will return to the computer representation of real numbers on slide 30.

Big O Notation

The importance of runtime

An important goal of numerical analysis is to answer the following question:

Given a function $F(X)$, an input X, an algorithm $F(X, n)$ and an error tolerance τ, how long does it take to evaluate $F(X, n)$ with an effort parameter n large enough such that $|F(X) - F(X, n)| \leq \tau$?

This question is important because its answer allows us to answer several related questions.

- Which algorithm should I use?

  Clearly, the best algorithm is the one which produces a sufficiently accurate result in the least amount of time.

- What should be my workflow when running simulations?

  Working with a simulation which takes a couple of seconds is very different from working with one which takes several days.

Big O Notation

The importance of runtime (continued)

An important goal of numerical analysis is to answer the following question:

Given a function $F(X)$, an input X, an algorithm $F(X, n)$ and an error tolerance τ, how long does it take to evaluate $F(X, n)$ with an effort parameter n large enough such that $|F(X) - F(X, n)| \leq \tau$?

This question is important because its answer allows us to answer several related questions.

- Are the computation feasible and sensible?

  You likely would not want to invest precious hardware and electricity in running a simulation which takes longer than the age of the universe. Similarly, there is no point in running a weather simulation for the next day if the simulation itself takes more than one day.

Big O Notation

The importance of runtime (continued)

Unfortunately, it is often not possible to give a definitive answer to the above runtime question. The reasons for this can be grouped into two broad categories.

- Determining the "correct" value of n is difficult.
- Determining the runtime of $F(X, n)$ is difficult even for a fixed n.

In the following, I will first elaborate why estimating the runtime of a given algorithm is difficult and then introduce a mathematical tool (the big O notation) which allows us to nevertheless argue about the runtime of algorithms to some extent.

Later on, we will then see that the big O notation also helps us with picking the appropriate value of the effort parameter n.

Big O Notation

Why estimating runtimes is difficult

Any computer program is ultimately just a composition of a fairly limited set of basic operations.

In numerical analysis, the most important of these basic operations are addition and multiplication. In an ideal world, we would therefore be able to estimate the runtime of a program using the formula

$$\begin{aligned}(\text{overall runtime}) = (\#\ \text{additions}) \times (\text{time per addition}) + \\ (\#\ \text{multiplications}) \times (\text{time per multiplication}).\end{aligned}$$

Unfortunately, this is not the case for a variety of reasons. The following slides will list some of them.

Big O Notation

Why estimating runtimes is difficult (continued)

Implementation matters.

The matrix product $C_{ij} = \sum_k A_{ik} B_{kj}$ is easy to define and can be implemented in just a few lines of code, see `matrix_product(A,B)`.

However, this simple implementation is about 100x slower than an expert implementation, see `matrix_product()`.

Big O Notation

Why estimating runtimes is difficult (continued)

Contiguous memory accesses are faster than spread out ones.

Computer memory can be thought of as a very vector $\texttt{mem} \in \{0,1\}^N$.

Computers must therefore store matrices by reshaping them into a one-dimensional sequence of numbers which is then further translated into a sequence of bits.

$$\begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix} \longrightarrow \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{pmatrix}$$

This storage scheme means that consecutive entries in a column are stored contiguously in memory, but consecutive entries in a row are spread out. CPUs are designed to work faster if the inputs are stored contiguously in memory; hence

$$\sum_j \sum_i A[i,j] \quad \text{runs faster than} \quad \sum_i \sum_j A[i,j],$$

see `matrix_sum()`.

Big O Notation

Why estimating runtimes is difficult (continued)

Pipelining and if-else branches.

Processors have separate read, compute and write units. These units support *pipelining*, which means that the compute unit can work on instruction `k` while the read unit works on instruction `k+1` and the write unit works on instruction `k-1`.

Actually, your CPU has even more units than `read`, `compute` and `write` which can work in parallel, but let us assume these are all for the sake of simplicity.

Time	0	1	2	3	4	5	...
read	0	1	2	3	4	5	...
compute		0	1	2	3	4	...
write			0	1	2	3	...

This pipelining breaks down whenever it encounters an if-else statement, since in this case the read unit does know which instruction to prepare for until the compute unit finishes.

Big O Notation

Why estimating runtimes is difficult (continued)

Pipelining and if-else branches (continued).

Processors try to avoid this issue by guessing which branch of the if-else statement will be picked and letting the read unit work on that branch.

If the guess is correct, then pipelining works just fine. If the guess is wrong, then the pipeline has to be reset which incurs a heavy performance penalty.

This seemingly harmless optimisation has some surprising consequences, see `branch_prediction()` and https://en.wikipedia.org/wiki/Spectre_(security_vulnerability).

Big O Notation

Why estimating runtimes is difficult (conclusion)

The take-away points of the above examples are:

- We cannot accurately predict runtimes solely by counting operations because the runtimes of operations depend on context.
- Nevertheless, operation counts do allow for some insight into runtimes: if we do X times as many operations of the same type in the same context, then the runtime will go up by a factor X.

Examples

- `sum_if()` performs n comparisons and $\frac{n}{2}$ additions in expectation; hence doubling n leads two twice as many operations and takes twice as long.
- `sum_ij()` and `sum_ji()` perform n^2 additions; hence doubling n leads to four times more operations and takes four times as long.
- `my_matrix_product()` performs n^3 additions and multiplication; hence doubling n leads to eight times more operations and takes eight times as long.

Big O Notation

Warning: Runtimes in the preasymptotic regime

The rule of thumb "X times as many operations take X times as long" may break down if the number of operations is very small.

Example: See `preasymptotic()`.

However, such anomalies tend to disappear once we go to large enough problem sizes. See `asymptotic()`.

Big O Notation

More complicated operation counts

In the above examples, counting the number of operations and figuring out how this number changes as n doubles was very easy. Unfortunately this is not always the case.

Example:

```
s = 0.0
for i = 1:n; s += i; end
for i = 1:n, j = 1:i; s += i*j; end
return s
```

The exact number of operations in this piece of code is

$$n + \frac{n\,(n+1)}{2} \text{ additions} \quad \text{and} \quad \frac{n\,(n+1)}{2} \text{ multiplications,}$$

but this answer is unsatisfying for two reasons.

- It requires quite a bit of thinking to verify.
- Even if you know the answer, it does not immediately tell you by what factor the number of operations will change if you double n. However, we have seen that this is the only reasonable piece of information to be extracted from an operation count.

Big O Notation

Conclusion

The discussion on the above slides indicates that it would be convenient to have a notation which allows us to condense a formula like $T(n) = n + \frac{n(n+1)}{2}$ into something which contains just enough information so we can predict $T(2n)$ given $T(n)$. This notation is provided by the following definition.

Def: Big O notation

We say $f(x) = O\big(g(x)\big)$ for $x \to x_0$ if

$$\lim_{x \to x_0} \left| \frac{f(x)}{g(x)} \right| < \infty.$$

The limit x_0 can be either finite or $\pm\infty$.

Big O Notation

Example

$$4x^3 - 2x^2 + 5x = O\big(x^3\big) \qquad \text{for } x \to \infty$$

Proof.

$$\lim_{x\to\infty} \frac{4x^3 - 2x^2 + 5x}{x^3} = 4 < \infty.$$

Discussion.

Note how $O(x^3)$ is a much simpler expression than $4x^3 - 2x^2 + 5x$, but it still contains enough information such that we can predict

$$f(x) = O(x^3) \quad \Longrightarrow \quad f(2x) \approx 2^3\, f(x) \quad \text{for “large enough” } x.$$

Thus, the big O notation indeed achieves its purpose of providing just enough information to answer the $f(2x) = [\text{what factor?}]\, f(x)$ question.

Big O Notation

Implicit limits

The above definition of $f(x) = O\big(g(x)\big)$ requires you to specify a limit $x \to x_0$, but often it is implicitly understood what this limit should be and mentioning it explicitly and/or repeatedly can be distracting and annoying.

Example: Consider the following statement:

Computing the inner product of two vectors
of length n is an $O(n)$ operation.

It is implicitly understood that this statement is meant with reference to the limit $n \to \infty$.

On the other hand, failing to mention the limit can occasionally be confusing, in particular if there are multiple variables involved in the statement.

As a writer, you will therefore sometimes have to make judgement calls whether it is better to mention the limit or not mention the limit.

Conversely, if you encounter a confusing big O statement as a reader, a good first step to resolve the confusion is to ask: "What is the limit considered in this statement?"

Big O Notation

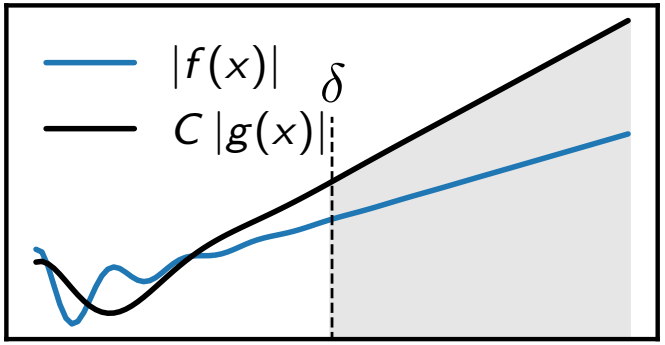
Alternative definition of big O notation

The above definition of $O\big(g(x)\big)$ is convenient to work with, but the following, equivalent definition is perhaps more illuminating:

We say $f(x) = O\big(g(x)\big)$ for $x \to x_0$ if there are $C, \delta \in \mathbb{R}$ such that

$$\left.\begin{aligned} |x - x_0| < \delta \quad & \text{if } |x_0| < \infty \\ x > \delta \quad & \text{if } x_0 = \infty \\ x < \delta \quad & \text{if } x_0 = -\infty \end{aligned}\right\} \quad \Longrightarrow \quad |f(x)| \leq C\,|g(x)|.$$

Illustration for $x_0 = \infty$:



Note how in this plot $|f(x)| > C\,|g(x)|$ for some $x < \delta$. This shows that the big O notation allows us to hide "preasymptotic complications" such as those illustrated on slide 12.

Big O Notation

How we got here

Understanding the big O notation is the single most important learning outcome of this lecture. Now that we have achieved a major part of this goal, let us step back for a moment and look back at how we got here.

- A major theme in numerical analysis is to approximate a given function $F(X)$ using a sequence of approximations $F(X, n)$.
- Given such an approximation, it would be helpful if we could predict how long it takes to evaluate $F(X, n)$ with a value of n large enough such that $F(X, n)$ is within some error tolerance of $F(X)$.
- Doing so is difficult in part because estimating the runtime of $F(X, n)$ is difficult even if n is fixed. In fact, benchmarking the algorithm for a moderately large n and extrapolating from there is usually the best we can do.

  benchmark = measure the runtime
- The big O notation is a mathematical tool which allows us to reduce a complicated operation count into a much simpler expression which contains just enough information so we can do the aforementioned extrapolation.

Big O Notation

Big O for convergence

Now that we know about the big O notation, we can start looking into the other reason why estimating the runtime of numerical algorithms is difficult, namely that we usually do not know what value of n will be required to achieve sufficient accuracy.

In an ideal world, we would be able to derive a simple function $e(n)$ such that $|F(X) - F(X, n)| \leq e(n)$ for all n. The n required to meet an error tolerance τ would then be given by $n = e^{-1}(\tau)$.

Unfortunately, determining such an $e(n)$ is usually not possible, but instead we can derive estimates like $e(n) = O(n^{-1/2})$. This then indicates that improving the accuracy by a factor 2 requires us to increase n by a factor 4.

The big O notation is thus useful not only for talking about runtime but also for characterising the speed of convergence.

Big O Notation

Outlook

The remainder of this lecture will address the following topics.

- I will occasionally use the big O notation in ways which follow from the above definition "in spirit" but which are not strictly speaking covered by this definition. The generalised definition of big O introduced on slide 22 will remedy this circumstance.
- The "=" in $f(x) = O\big(g(x)\big)$ may seem to suggest that big O is an equivalence relation, i.e. that we should have

  $$f = O\big(g\big) \quad \Longrightarrow \quad g = O\big(f\big). \qquad \text{(Not true in general!)}$$

  This is not true in general, but often we implicitly expect that big O is used in this way. I will elaborate further on this on slide 25.
- The $f(x) = O\big(g(x)\big)$ in principle allows you to choose any arbitrary function $g(x)$, but in practice $g(x)$ almost always falls into one of three categories. These categories come with special terminology which I will describe on slide 27.
- It is often helpful to be able to "empirically observe" big O scaling. Slide 29 will provide some tips and tricks for this.

Big O Notation

Big O as a set of functions

It will occasionally be useful to make statements like

$$\exp(x) = 1 + O(x) \quad \text{for } x \to 0$$

to emphasise the point that $\exp(x)$ is "1 plus a small perturbation for small values of x". Similarly, I will occasionally make claims like

$$O(x^2)\, O(x^{-1}) = O(x) \quad \text{for } x \to \infty.$$

These statements may look intuitively plausible, but at this point they have no clear mathematical meaning: the definitions on slides 14 and 17 assign meaning only to expressions of the form $f(x) = O\big(g(x)\big)$, and neither of the above expressions is of this form.

Big O Notation

Big O as a set of functions (continued)

We can assign a precise meaning to the above expressions which is consistent with our intuition using the following three-step program:

- Redefine $O(g(x))$ as the set of functions

$$O\big(g(x)\big) = \left\{ \; f(x) \;\middle|\; \limsup_{x \to x_0} \tfrac{|f(x)|}{|g(x)|} < \infty \; \right\},$$

  or equivalently,

$$O\big(g(x)\big) = \left\{ f(x) \;\middle|\; \exists C, \delta : |x - x_0| < \delta \quad \Longrightarrow \quad |f(x)| \leq C\,|g(x)| \right\}.$$

- Define $F\big(O(g)\big) = \big\{F(f) \mid f \in O(g)\big\}$.
- Interpret $f(x) = O\big(g(x)\big)$ as a somewhat peculiar notation for $f(x) \in O\big(g(x)\big)$, and similarly interpret $O\big(f(x)\big) = O\big(g(x)\big)$ as a special notation for $O\big(f(x)\big) \subseteq O\big(g(x)\big)$.

Big O Notation

Big O as a set of functions (continued)

The definitions on the previous slide clearly extend the earlier definition of big O, i.e. the two definitions agree when both are applicable, but the second set of definitions covers more expressions than the first definition.

When you encounter a big O statement which does not make sense to you, you should thus first try to explain it using the somewhat simpler definition from slide 14, and if that definition does not apply then try the definitions on the previous slide.

Big O Notation

Big O as an order relation

The $\leq$ in the definition of big O on slide 17 shows that $f(x) = O\big(g(x)\big)$ is an order rather than an equivalence relation, i.e.

$$f = O\big(g\big) \;\wedge\; g = O\big(h\big) \quad \Longrightarrow \quad f = O\big(h\big)$$

but

$$f = O\big(g\big) \quad \not\Longrightarrow \quad g = O\big(f\big).$$

This order aspect of big O is sometimes emphasized by writing

$$f(x) \lesssim g(x) \qquad \text{instead of} \qquad f(x) = O\big(g(x)\big),$$

but this notation is not always applicable. For example, there is no fully satisfying way of expressing

$$\exp(x) = 1 + O(x) \quad \text{for } x \to 0$$

using the $\lesssim$ notation.

Big O Notation

Sharpness

The fact that big O is an order relation indicates that we have some nontrivial freedom in how to choose the right-hand side function $g(x)$ in $f(x) = O\big(g(x)\big)$. For example, we have

$$x\,(x+1) = O(x^2) \quad \text{for } x \to \infty,$$

but it is equally correct to write

$$x\,(x+1) = O(x^3) \qquad \text{since} \qquad O(x^2) \subset O(x^3).$$

This freedom is sometimes useful. For example, it allows us to say that $ax + 1 = O(x)$ for all values of a even though $ax + 1 = O(1)$ if $a = 0$.

Big O Notation

Sharpness (continued

On the other hand, we usually expect a statement like $f(x) = O\big(g(x)\big)$ to be *sharp*, i.e. we usually expect that

$$f = O\big(g\big) \quad \Longrightarrow \quad g = O\big(f\big). \qquad \text{(Not true in general!)}$$

Furthermore, if $f(x)$ describes a family of functions (e.g. $f(x) = ax + 1$), then we expect this property to hold at least for some parameter values.

This expectation is evident in situations like the following.

- A claim like

$$f(x) = O(x^2) \quad \Longrightarrow \quad f(2x) \approx 4\, f(x)$$

  implicitly assumes that $f(x) = O(x^2)$ is sharp, i.e. that $f(x) \neq O(x)$.
- Assume I ask you to determine p such that $x\,(x+1) = O(x^p)$.
  It is technically correct to answer $p = 3$, but I would not give you full marks because p is not as small as possible.

Big O Notation

Important big O classes

The vast majority of runtimes and speeds of convergence fall into one of the following three categories:

- Logarithmic scaling $= O\big(\log(x)\big)$.
- Algebraic / polynomial scaling $= O\big(x^p\big)$.
- Exponential scaling $= O\big(a^x\big)$.

In the limit $x \to \infty$, these categories satisfy the following relations:

- $\log(x) = O\big(x^p\big)$ if $p > 0$.
- $x^p = O\big(x^q\big)$ if $p \leq q$.
- $x^p = O\big(a^x\big)$ if p and $a > 1$.
- $a^x = O\big(x^p\big)$ if $|a| < 1$ and p.
- $a^x = O\big(b^x\big)$ if $a \leq b$.

You should be fluent in using these scaling classes and their relationship, i.e. you should be able to use them without much thinking.

Big O Notation

Further scaling terminology

- Algebraic scaling ($O(x^p)$) with $p = 1, 2$ and 3 is called *linear*, *quadratic* and *cubic* scaling, respectively.
- *Super-* means "faster than", and *sup-* means "slower than".

  Example: $\exp(\sqrt{x})$ scales super-algebraically (it grows faster than any polynomial), but sub-exponentially (it is asymptotically dominated by all functions of the form a^x with $a > 1$).
- The power p in the algebraic scaling estimate $O(x^p)$ is called *order*.

  Example: $a_n = n^{-2}$ is said to be second-order convergent.
- The base a in the exponential scaling estimate $O(a^x)$ is called *rate*.

  Example: $a_n = 2^{-n}$ is said to be exponentially conv. with rate $\frac{1}{2}$.

Big O Notation

Empirical demonstration of big O scaling

It is often Consider the following scenario: After some long and tedious calculations, you determine that the runtime of your algorithm should be $O(n)$ but you are not sure whether you made a .

How can you test whether the runtime is indeed $O(n)$?

In the examples at the beginning of this lecture, we could experimentally observe the scaling of the algorithms by doubling n and determining the increase in runtime, but this approach does not always work. For example, we have

$$f(x) = \big(2 + \sin(x)\big)\, x = O(x),$$

but it is hard to recognise $f(2x) \approx 2\, f(x)$ from e.g. the data

```
round.(Int, f.(2.0.^(4:8))) == [ 27, 82, 187, 348, 256 ]
```

A better approach is to plot $f(x)$ on appropriate axis, see `algebraic_scaling()` and `exponential_scaling()`.

Big O Notation

Machine precision and convergence plots

Recall from slide 3 that computers represent real numbers by truncating their binary representation after the first n digits.

Experience tells us that keeping $n = 53$ binary digits is a reasonable compromise between performance and accuracy for most purposes, and most programming environments (C/C++, Python, Matlab, R, Julia, etc.) use this representation by default.

For reasons beyond the scope of this module, this "53 significant bits"-representation actually requires 64 bits of storage and hence this representation is called `Float64` in Julia.

Keeping 53 significant binary digits corresponds to a relative accuracy of roughly 2.2×10^{-16}. This number is called the *machine precision* or *machine epsilon*, and you can obtain this number using the command `eps()` in Julia.

Big O Notation

Machine precision and convergence plots (continued)

The machine precision is often visible in convergence plots as a lower bound beyond which we get no further convergence.

Example. `machine_precision()` computes Euler's number $e = \exp(1)$ using the formula

$$\exp(x) = \lim_{n\to\infty} \sum_{k=0}^{n} \frac{x^k}{k!}.$$

We observe that the error stagnates once it is roughly of size `eps()`.

Students occasionally get worried about this kink in their convergence plot, but there is nothing to worry about here: relative errors stagnating at roughly `eps()` simply mean that you have computed your answer as accurately as you possibly can under the given floating-point representation.

Big O Notation

Summary

- Big O notation:

$$f(x) = O\big(g(x)\big) \text{ for } x \to x_0 \qquad \Longleftrightarrow \qquad \limsup_{x \to x_0} \left| \frac{f(x)}{g(x)} \right| < \infty.$$

- Logarithmic, algebraic and exponential scaling.
- Appropriate axes for a given scaling:
  - Algebraic scaling, $f(x) = O(x^p)$: use `loglog(x,f)`.
  - Exponential scaling, $f(x) = O(a^x)$: use `semilogy(x,f)`.