MA5233 Computational Mathematics

Lecture 2: Big O Notation

Simon Etter

Semester I, AY 2020/2021

Big O Notation

Introduction

We have seen:

- Computer programs are functions mapping inputs to outputs.
- Key characteristics of a program are correctness and runtime.

The last lecture discussed various aspects regarding the correctness of machine arithmetic, but it largely ignored runtime.

However, runtime is important:

- It makes the difference between what is only correct and what is also practical.
- If we know how long a piece of code will take, we can plan ourselves accordingly.

  Example:

$$\begin{aligned} \text{runtime} < 1\text{min} &\implies \text{just wait} \\ \text{runtime} < 15\text{min} &\implies \text{go for coffee break} \\ \text{runtime} > 1\text{h} &\implies \text{run overnight} \end{aligned}$$

Unfortunately, estimating how long a piece of code will take is not as straightforward as you might think.

Big O Notation

Why estimating runtimes is difficult

In an ideal world, the runtime of a program with a additions and p products would be

$$a \times (\text{time of one addition}) + p \times (\text{time of one product}).$$

This is not the case for a variety of reasons.

The following slides list some of them.

Big O Notation

Why estimating runtimes is difficult (continued)

Contiguous memory accesses are faster than spread out ones.

Recall: computer memory is just a very long sequence of bits.

Consequently, computers must store matrices by reshaping them into a one-dimensional sequence of numbers which can then be 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,

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

see `matrix_sum()`.

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 the expert implementation provided by Julia, see `matrix_product()`.

The main improvement in the expert implementation is that it cleverly rearranges the operations to operate on contiguous data as much as possible, see previous slide.

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`.

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()`.

Big O Notation

Conclusion

We cannot accurately predict runtimes solely by counting operations because the runtimes of operations depend on context.

However, 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

Discussion

It is difficult to precisely pin down what constitutes the "context" of an operation. For example, it is often the case that the runtime exhibits unexpected behaviour if the program is run on a very small amount of data.

Example:

```
julia> x = zeros(32); @btime sum($x);
  16.210 ns (0 allocations: 0 bytes)
julia> x = zeros(64); @btime sum($x);
  15.915 ns (0 allocations: 0 bytes)
# Twice the amount of work in the same amount of time?!
```

However, these anomalies tend to disappear once we go to large enough problem sizes.

Example:

```
julia> x = zeros(1024); @btime sum($x);
  76.779 ns (0 allocations: 0 bytes)
julia> x = zeros(2048); @btime sum($x);
  155.262 ns (0 allocations: 0 bytes)
julia> x = zeros(4096); @btime sum($x);
  328.230 ns (0 allocations: 0 bytes)
# Now we are back to something sensible:
# runtime (roughly) doubles every time we double `length(x)`.
```

Big O Notation

Discussion

In the above examples, counting the number of operations and figuring out how this number changes as n doubles was very easy. However, 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 $n + \frac{n(n+1)}{2}$ additions and $\frac{n(n+1)}{2}$ multiplications, but this answer is unsatisfying for two reasons.

- It requires at least a couple of seconds to work out.
- Even if you know the answer, it does not immediately tell you how the runtime will change if you double n.

Conclusion: we need a way 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 is the purpose of the big O notation.

Big O Notation

Def: Big O notation

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

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

Remarks

- The "sup" in "lim sup" is required for cases when ${}^{f(x)}/_{g(x)}$ oscillates or is only defined for certain x. These edge cases are rare in practice, so feel free to ignore this sup if it confuses you.
- The limit x_0 can be either finite or $\pm\infty$.

Big O Notation

Example

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

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

Discussion.

For $g(x) = x^3$, we immediately recognise that $g(2x) = 8\,g(x)$.

Similarly, $f(x) = O\left(x^3\right)$ tells us that $f(2x) \approx 8\,f(x)$, and the $\approx$ will become more and more accurate for larger and larger x.

This statement is not quite accurate. See slide 18 for further discussion.

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

Big O Notation

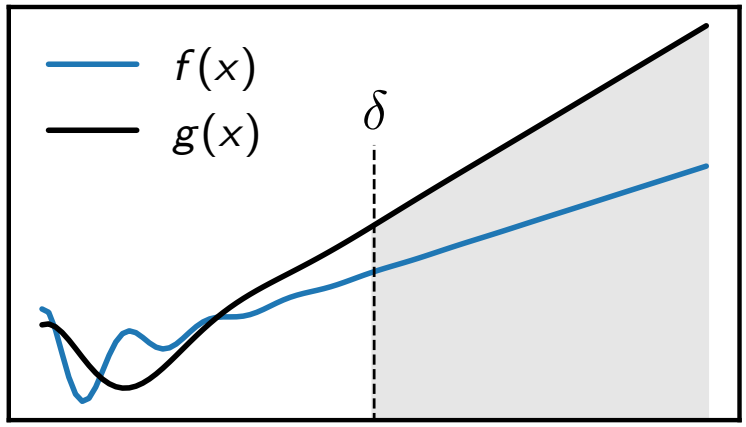
Big O is an order relation

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 $M, \delta \in \mathbb{R}$ such that

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

Illustration for $x_0 = \infty$:



Big O Notation

Big O as an order relation (continued)

The $\leq$ in this definition 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

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

Counterexample:

$x = O(x^2)$ for $x \to \infty$ since $\lim_{x\to\infty} \frac{x}{x^2} = 0$, but $x^2 \neq O(x)$ since $\lim_{x\to\infty} \frac{x^2}{x} = \infty$.

The order aspect of O is sometimes emphasized by writing $f(x) \lesssim g(x)$ instead of $f(x) = O\big(g(x)\big)$.

However, the $\lesssim$ notation is not always applicable. For example, there is no sensible way to express

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

using $\lesssim$.

Big O Notation

Notation related to big O

The next slide lists several notations closely related to the big O notation introduced above.

Unlike big O, these related notations are rarely used. My reasons for listing them are

- to illustrate the big O notation by demonstrating how to adapt it for different purposes, and
- to introduce terminology for discussing some subtleties of the big O notation.

I will not be using any notation other than big O outside of this lecture, and you are not expected to know the other notations by heart.

Big O Notation

Big X notation	Relation notation	Definition
$f(x) = O\big(g(x)\big)$	$f(x) \lesssim g(x)$ "asymptotically (upper) bounded"	$\limsup_{x \to x_0} \left\lvert \frac{f(x)}{g(x)} \right\rvert < \infty$
$f(x) = o\big(g(x)\big)$	$f(x) \ll g(x)$ "asymptotically smaller"	$\lim_{x \to x_0} \left\lvert \frac{f(x)}{g(x)} \right\rvert = 0$
$f(x) = \Omega\big(g(x)\big)$	$f(x) \gtrsim g(x)$ "asymptotically lower bounded"	$\liminf_{x \to x_0} \left\lvert \frac{f(x)}{g(x)} \right\rvert > 0$
$f(x) = \omega\big(g(x)\big)$	$f(x) \gg g(x)$ "asymptotically larger"	$\lim_{x \to x_0} \left\lvert \frac{f(x)}{g(x)} \right\rvert = \infty$
$f(x) = \Theta\big(g(x)\big)$	"asymptotically similar"	$f(x) = O\big(g(x)\big) \wedge$ $f(x) = \Omega\big(g(x)\big)$

See also https://en.wikipedia.org/wiki/Big_O_notation#Family_of_Bachmann-Landau_notations.

Big O Notation

Remark: Implicit limits

The limit x_0 is often not explicitly mentioned. For example, we say that the runtime of a matrix product is $O(n^3)$, and it is implicitly understood that we mean the limit $n \to \infty$.

Remark: Other applications of big O

This lecture introduced the big O notation as a means for describing the "essential" part of a runtime estimate.

However, big O is also used in contexts unrelated to runtimes. For example, in computational mathematics it is often the case that we can get a more accurate result by increasing some parameter n. In this context, we might say something like "error $= O(n^{-2})$".

Big O Notation

Remark: Big O vs big Θ

We have seen on slide 14 that big O is an order relation similar to $\leq$.

Thus, instead of writing $4x^3 - 2x^2 + 5x = O(x^a)$ with $a = 3$ as on slide 12, we could equally well pick any $a \geq 3$.

This $\leq$ aspect of big O is sometimes useful. For example, it allows us to say that an algorithm requires $O(n^2)$ operations even if this algorithm requires only $O(n)$ operations in some special cases.

However, authors frequently write $O\big(g(x)\big)$ in contexts where they actually mean $\Theta\big(g(x)\big)$. I already did so in the example on slide 12

Correspondingly, it is usually assumed that big O estimates are sharp, i.e. $f(x) = O\big(g(x)\big)$ is usually assumed to mean "$f(x) = \Theta\big(g(x)\big)$ at least in some cases".

For example, while it is technically correct to say that the runtime of a matrix product is $O(n^5)$, this statement would immediately be criticised (by me as your examiner, the referees of your paper, etc.) for not being as sharp as possible.

Big O Notation

Remark: Big O of numbers

In an abuse of notation, I will sometimes apply big O to numbers, i.e. I will make statements like $c = O(10)$.

This might mean something like $c \in (1, 100)$, though the precise extent of the interval depends on the context.

Examples:

- The duration of a typical escalator ride is $O(30\text{sec})$.
- The duration of a typical MRT ride is $O(30\text{min})$.
- The duration of a typical flight is $O(3\text{h})$.

I will refer to $f(x) = O\big(g(x)\big)$ as the "function interpretation of big O" and to $c = O(C)$ as the "number interpretation of big O".

Big O Notation

Important big O relations (assuming $x \to \infty$)

- $a^x = O\big(b^x\big)$ if $a \leq b$.
- $p(x) = O\big(a^x\big)$ for any polynomial $p(x)$ and any $a > 1$.
- $p(x) = O\big(x^a\big)$ for any polynomial $p(x)$ with $\text{degree}(p) \leq a$.
- $\log(x) = O\big(x^a\big)$ for any $a > 0$.

Proof. According to L'Hôpital's rule, we have

$$\log(x) = O\big(x^a\big) \quad \Longleftrightarrow \quad \lim_{x\to\infty} \frac{\log(x)}{x^a} = \lim_{x\to\infty} \frac{x^{-1}}{x^{a-1}} = \lim_{x\to\infty} x^{-a} = 0.$$

Similarly, $p(x) = O\big(\exp(ax)\big)$ can be shown by $\text{degree}(p)$-fold application of L'Hôpital's rule.

The other statements are trivial.

Terminology

- Exponential scaling: $f(x) = O\big(a^x\big)$.
- Algebraic scaling: $f(x) = O\big(x^a\big)$.

Big O Notation

Demonstrating big O scaling

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

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

For the examples in the beginning of this lecture, we could experimentally observe the scaling of the algorithms by doubling n and determining the increase in runtime.

This approach does not always work. For example, we have

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

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

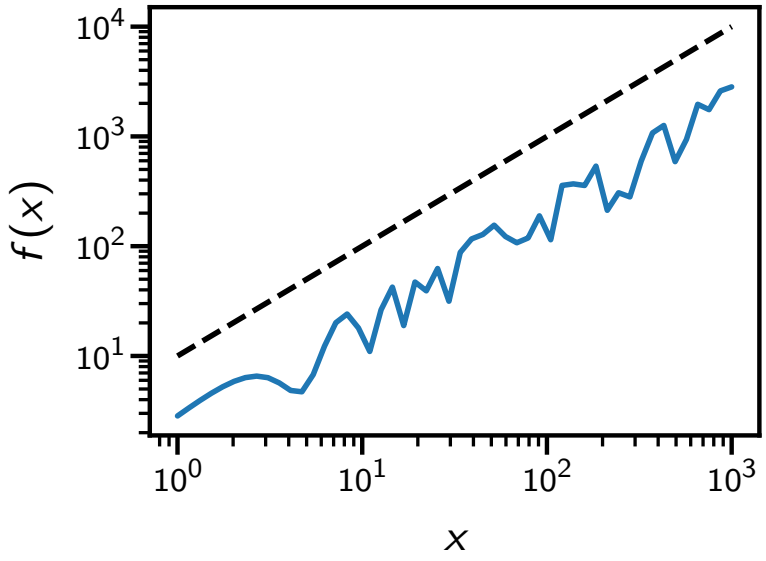
```
round.(Int, f.(2.0.^(4:8))) == [ 27, 82, 187, 348, 256 ]
```

A better approach is to plot $f(x)$ on appropriate axis, see next slide.

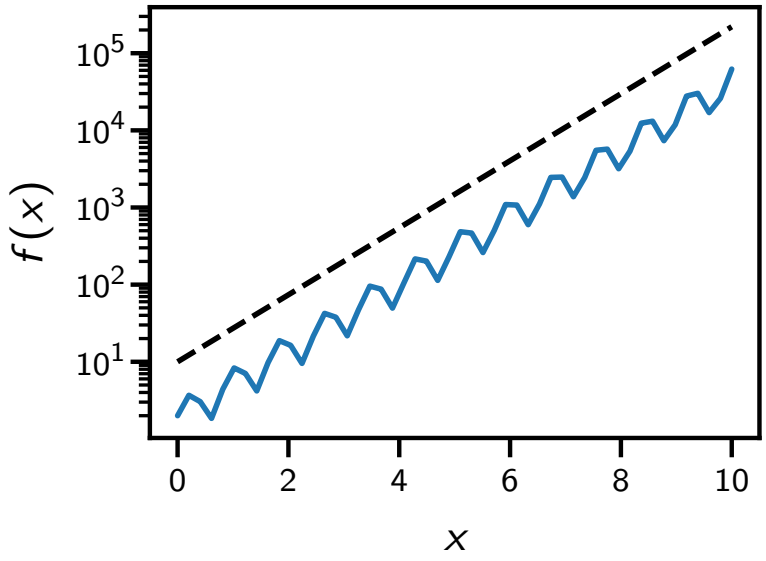
Big O Notation

Demonstrating big O scaling (continued)

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



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



For algebraic scaling, $f(x) = O(x^a)$:

- Use logarithmic x- and y-axis (in PyPlot: `loglog`).

For exponential scaling, $f(x) = O(a^x)$:

- Use linear x-axis and logarithmic y-axis (in PyPlot: `semilogy`).

In both cases, plot x^a / a^x as a reference (dashed black line).

Big O Notation

Summary

- $f(x) = O\big(g(x)\big)$ for $x \to x_0 \quad \Longleftrightarrow \quad \limsup_{x \to x_0} \left| \dfrac{f(x)}{g(x)} \right| < \infty.$
- Axes for a given asymptotic scaling:

$$f(x) = O(x^p) \quad \longrightarrow \quad \begin{cases} \texttt{loglog(x, f)} \\ \texttt{loglog(x, c.*x.\^{}p, "k--")} \end{cases}$$

$$f(x) = O(a^x) \quad \longrightarrow \quad \begin{cases} \texttt{semilogy(x, f)} \\ \texttt{semilogy(x, c.*a.\^{}x, "k--")} \end{cases}$$

Recommended exercises

Verify the following runtimes:

- Vector addition and inner products: $O(n)$
- Matrix-vector product, triangular solves: $O(n^2)$
- Matrix-matrix product, LU factorisation: $O(n^3)$