Computational Complexity of real functions

Amaury Pouly

April 8, 2015

Outline

Examples

"computable = can be solved with a computer"

Examples

"computable = can be solved with a computer"

Example (Sorting)

SORT: given n integers $x_1, \ldots, x_n$, find a permutation σ such that $x_{\sigma(1)} \leqslant x_{\sigma(2)} \leqslant \ldots \leqslant x_{\sigma(n)}$.

Examples

"computable = can be solved with a computer"

Example (Sorting)

SORT: given n integers $x_1, \dots, x_n$, find a permutation σ such that $x_{\sigma(1)} \leqslant x_{\sigma(2)} \leqslant \dots \leqslant x_{\sigma(n)}$.

$\Rightarrow$ "clearly computable"

Examples

"computable = can be solved with a computer"

Example (Sorting)

SORT: given n integers $x_1, \ldots, x_n$, find a permutation σ such that $x_{\sigma(1)} \leqslant x_{\sigma(2)} \leqslant \ldots \leqslant x_{\sigma(n)}$.

$\Rightarrow$ "clearly computable"

Example (Ackermann function)

ACK: given n and m, compute $A_{m,n}$ defined by

$$A_{0,n} = n + 1 \qquad A_{m,0} = A_{m-1,1} \qquad A_{m,n} = A_{m-1,A_{m,n-1}}$$

Examples

"computable = can be solved with a computer"

Example (Sorting)

SORT: given n integers $x_1, \ldots, x_n$, find a permutation σ such that $x_{\sigma(1)} \leqslant x_{\sigma(2)} \leqslant \ldots \leqslant x_{\sigma(n)}$.

$\Rightarrow$ "clearly computable"

Example (Ackermann function)

ACK: given n and m, compute $A_{m,n}$ defined by

$$A_{0,n} = n + 1 \qquad A_{m,0} = A_{m-1,1} \qquad A_{m,n} = A_{m-1,A_{m,n-1}}$$

$\Rightarrow$ "clearly computable"...

Examples

"computable = can be solved with a computer"

Example (Sorting)

SORT: given n integers $x_1, \ldots, x_n$, find a permutation σ such that $x_{\sigma(1)} \leqslant x_{\sigma(2)} \leqslant \ldots \leqslant x_{\sigma(n)}$.

$\Rightarrow$ "clearly computable"

Example (Ackermann function)

ACK: given n and m, compute $A_{m,n}$ defined by

$$A_{0,n} = n + 1 \qquad A_{m,0} = A_{m-1,1} \qquad A_{m,n} = A_{m-1,A_{m,n-1}}$$

$\Rightarrow$ "clearly computable"...but slow ? ($A_{4,2} \approx 10^{20000}$)

Examples (2)

Example (Collatz/Syracuse sequence)

COLLATZ: given n decide if this sequence converges to 1:

$$u_0 = n \qquad u_{k+1} = \begin{cases} \frac{u_k}{2} & \text{if } u_k \text{ is even} \\ 3u_k & \text{otherwise} \end{cases}$$

Examples (2)

Example (Collatz/Syracuse sequence)

COLLATZ: given n decide if this sequence converges to 1:

$$u_0 = n \qquad u_{k+1} = \begin{cases} \frac{u_k}{2} & \text{if } u_k \text{ is even} \\ 3u_k & \text{otherwise} \end{cases}$$

$\Rightarrow$ unclear...

Examples (2)

Example (Collatz/Syracuse sequence)

COLLATZ: given n decide if this sequence converges to 1:

$$u_0 = n \qquad u_{k+1} = \begin{cases} \frac{u_k}{2} & \text{if } u_k \text{ is even} \\ 3u_k & \text{otherwise} \end{cases}$$

$\Rightarrow$ unclear...

Example (Halting problem)

HALT: given a program P and an input x, decide if P halts on x

Examples (2)

Example (Collatz/Syracuse sequence)

COLLATZ: given n decide if this sequence converges to 1:

$$u_0 = n \qquad u_{k+1} = \begin{cases} \frac{u_k}{2} & \text{if } u_k \text{ is even} \\ 3u_k & \text{otherwise} \end{cases}$$

$\Rightarrow$ unclear...

Example (Halting problem)

HALT: given a program P and an input x, decide if P halts on x

$\Rightarrow$ what does a "program" mean ?

Computability theory

Computability theory is about:

Computability theory

Computability theory is about:

- formalise a notion of program and “being computable”

Computability theory

Computability theory is about:

- formalise a notion of program and “being computable”
- study different “levels” of computability

Computability theory

Computability theory is about:

- formalise a notion of program and “being computable”
- study different “levels” of computability
- identify and separate classes of similar problems

Computability theory

Computability theory is about:

- formalise a notion of program and “being computable”
- study different “levels” of computability
- identify and separate classes of similar problems

Back to the examples:

- `SORT`: primitive recursive

Computability theory

Computability theory is about:

- formalise a notion of program and “being computable”
- study different “levels” of computability
- identify and separate classes of similar problems

Back to the examples:

- `SORT`: primitive recursive
- `ACK`: recursive/computable but not primitive recursive

Computability theory

Computability theory is about:

- formalise a notion of program and "being computable"
- study different "levels" of computability
- identify and separate classes of similar problems

Back to the examples:

- `SORT`: primitive recursive
- `ACK`: recursive/computable but not primitive recursive
- `COLLATZ`: open problem

Computability theory

Computability theory is about:

- formalise a notion of program and “being computable”
- study different “levels” of computability
- identify and separate classes of similar problems

Back to the examples:

- `SORT`: primitive recursive
- `ACK`: recursive/computable but not primitive recursive
- `COLLATZ`: open problem
- `HALT`: undecidable/non-recursive/uncomputable

Computability theory

Computability theory is about:

- formalise a notion of program and “being computable”
- study different “levels” of computability
- identify and separate classes of similar problems

Back to the examples:

- `SORT`: primitive recursive
- `ACK`: recursive/computable but not primitive recursive
- `COLLATZ`: open problem
- `HALT`: undecidable/non-recursive/uncomputable

Going further:

Computability theory

Computability theory is about:

- formalise a notion of program and "being computable"
- study different "levels" of computability
- identify and separate classes of similar problems

Back to the examples:

- `SORT`: primitive recursive
- `ACK`: recursive/computable but not primitive recursive
- `COLLATZ`: open problem
- `HALT`: undecidable/non-recursive/uncomputable

Going further:

- Computational complexity: refine the notion of primitive recursive

Computability theory

Computability theory is about:

- formalise a notion of program and “being computable”
- study different “levels” of computability
- identify and separate classes of similar problems

Back to the examples:

- `SORT`: primitive recursive
- `ACK`: recursive/computable but not primitive recursive
- `COLLATZ`: open problem
- `HALT`: undecidable/non-recursive/uncomputable

Going further:

- Computational complexity: refine the notion of primitive recursive
- Turing degrees: refine the notion of uncomputable problems

Computational complexity

Example (Sorting)

$\texttt{SORT}$: given n integers $x_1, \ldots, x_n$, find a permutation σ such that $x_{\sigma(1)} \leqslant x_{\sigma(2)} \leqslant \ldots \leqslant x_{\sigma(n)}$.

Computational complexity

Example (Sorting)

SORT: given n integers $x_1, \dots, x_n$, find a permutation σ such that $x_{\sigma(1)} \leqslant x_{\sigma(2)} \leqslant \dots \leqslant x_{\sigma(n)}$.

$\Rightarrow$ "easy problem" $\rightarrow \mathcal{O}(n \log n)$ comparisons suffices

Computational complexity

Example (Sorting)

SORT: given n integers $x_1, \ldots, x_n$, find a permutation σ such that $x_{\sigma(1)} \leqslant x_{\sigma(2)} \leqslant \ldots \leqslant x_{\sigma(n)}$.

$\Rightarrow$ "easy problem" $\rightarrow \mathcal{O}(n \log n)$ comparisons suffices

Example (Subset sum)

SUBSET-SUM: given n integers $A_1, \ldots, A_n$ and an integer B, find a subset $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} A_i$.

Computational complexity

Example (Sorting)

SORT: given n integers $x_1, \ldots, x_n$, find a permutation σ such that $x_{\sigma(1)} \leqslant x_{\sigma(2)} \leqslant \ldots \leqslant x_{\sigma(n)}$.

$\Rightarrow$ "easy problem" $\rightarrow \mathcal{O}(n \log n)$ comparisons suffices

Example (Subset sum)

SUBSET-SUM: given n integers $A_1, \ldots, A_n$ and an integer B, find a subset $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} A_i$.

$\Rightarrow$ "not so easy problem" $\rightarrow$ we can check all possibilities

Computational complexity

Example (Sorting)

SORT: given n integers $x_1, \ldots, x_n$, find a permutation σ such that $x_{\sigma(1)} \leqslant x_{\sigma(2)} \leqslant \ldots \leqslant x_{\sigma(n)}$.

$\Rightarrow$ "easy problem" $\rightarrow \mathcal{O}(n \log n)$ comparisons suffices

Example (Subset sum)

SUBSET-SUM: given n integers $A_1, \ldots, A_n$ and an integer B, find a subset $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} A_i$.

$\Rightarrow$ "not so easy problem" $\rightarrow$ we can check all possibilities

Computational complexity

Example (Sorting)

$\texttt{SORT}$: given n integers $x_1, \ldots, x_n$, find a permutation σ such that $x_{\sigma(1)} \leqslant x_{\sigma(2)} \leqslant \ldots \leqslant x_{\sigma(n)}$.

$\Rightarrow$ "easy problem" $\rightarrow \mathcal{O}(n \log n)$ comparisons suffices

Example (Subset sum)

$\texttt{SUBSET-SUM}$: given n integers $A_1, \ldots, A_n$ and an integer B, find a subset $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} A_i$.

$\Rightarrow$ "not so easy problem" $\rightarrow$ we can check all possibilities

Example (Set game)

$\texttt{SET-GAME}$: given n finite sets $S_1, \ldots, S_n$, each player takes turn a non-empty set S_i and remove the elements of S_i from all the sets S_j. Decide if the first player has a winning (empty all sets) strategy.

Computational complexity

Example (Sorting)

SORT: given n integers $x_1, \dots, x_n$, find a permutation σ such that $x_{\sigma(1)} \leqslant x_{\sigma(2)} \leqslant \dots \leqslant x_{\sigma(n)}$.

$\Rightarrow$ "easy problem" $\rightarrow \mathcal{O}(n \log n)$ comparisons suffices

Example (Subset sum)

SUBSET-SUM: given n integers $A_1, \dots, A_n$ and an integer B, find a subset $I \subseteq \{1, \dots, n\}$ such that $\sum_{i \in I} A_i$.

$\Rightarrow$ "not so easy problem"$\rightarrow$ we can check all possibilities

Example (Set game)

SET-GAME: given n finite sets $S_1, \dots, S_n$, each player takes turn a non-empty set S_i and remove the elements of S_i from all the sets S_j. Decide if the first player has a winning (empty all sets) strategy.

$\Rightarrow$ "looks quite hard"$\rightarrow$ we can check all strategies !

Computability theory

Computational complexity theory is about:

Computability theory

Computational complexity theory is about:

- formalise a notion of “time”/“steps”, “space”/“memory”, ...

Computability theory

Computational complexity theory is about:

- formalise a notion of “time”/“steps”, “space”/“memory”, ...
- study different “levels” of complexity depending on space, time, ...

Computability theory

Computational complexity theory is about:

- formalise a notion of “time”/“steps”, “space”/“memory”, ...
- study different “levels” of complexity depending on space, time, ...
- identify and separate classes of similar problems

Computability theory

Computational complexity theory is about:

- formalise a notion of “time”/“steps”, “space”/“memory”, ...
- study different “levels” of complexity depending on space, time, ...
- identify and separate classes of similar problems
- give alternative characterisation of these classes

Computability theory

Computational complexity theory is about:

- formalise a notion of “time”/“steps”, “space”/“memory”, ...
- study different “levels” of complexity depending on space, time, ...
- identify and separate classes of similar problems
- give alternative characterisation of these classes

Back to the examples:

- `SORT`: polynomial time (P)

Computability theory

Computational complexity theory is about:

- formalise a notion of “time”/“steps”, “space”/“memory”, ...
- study different “levels” of complexity depending on space, time, ...
- identify and separate classes of similar problems
- give alternative characterisation of these classes

Back to the examples:

- `SORT`: polynomial time (*P*)
- `SUBSET-SUM`: nondeterministic polynomial time (*NP*)

Computability theory

Computational complexity theory is about:

- formalise a notion of “time”/“steps”, “space”/“memory”, ...
- study different “levels” of complexity depending on space, time, ...
- identify and separate classes of similar problems
- give alternative characterisation of these classes

Back to the examples:

- `SORT`: polynomial time (*P*)
- `SUBSET-SUM`: nondeterministic polynomial time (*NP*)
- `SET-GAME`: polynomial space (*PSPACE*)

Example (Halting problem)

HALT: given a program P and an input x, decide if P halts on x

Example (Halting problem)

HALT: given a program P and an input x, decide if P halts on x

$\Rightarrow$ undecidable

Example (Finite index)

FIN: given a program P, define $C_P = \{x \mid P \text{ halts on } x\}$, decide if C_P is finite.

Example (Halting problem)

HALT: given a program P and an input x, decide if P halts on x

$\Rightarrow$ undecidable

Example (Finite index)

FIN: given a program P, define $C_P = \{x \mid P \text{ halts on } x\}$, decide if C_P is finite.

$\Rightarrow$ “more undecidable than HALT”

Example (Halting problem)

HALT: given a program P and an input x, decide if P halts on x

$\Rightarrow$ undecidable

Example (Finite index)

FIN: given a program P, define $C_P = \{x \mid P \text{ halts on } x\}$, decide if C_P is finite.

$\Rightarrow$ "more undecidable than HALT"

$\Rightarrow$ undecidable even if we assume we can "solve" HALT

Turing degrees

Turing degrees complexity are about:

Turing degrees

Turing degrees complexity are about:

- formalising the “degree of unsolvability”

Turing degrees

Turing degrees complexity are about:

- formalising the "degree of unsolvability"
- study sets of integers for those degrees, links with ordinal theory

Turing degrees

Turing degrees complexity are about:

- formalising the “degree of unsolvability”
- study sets of integers for those degrees, links with ordinal theory

Back to the examples:

- SORT: degree $\varnothing$

Turing degrees

Turing degrees complexity are about:

- formalising the “degree of unsolvability”
- study sets of integers for those degrees, links with ordinal theory

Back to the examples:

- $\texttt{SORT}$: degree $\varnothing$
- $\texttt{HALT}$: degree $\varnothing'$

Turing degrees

Turing degrees complexity are about:

- formalising the "degree of unsolvability"
- study sets of integers for those degrees, links with ordinal theory

Back to the examples:

- $\texttt{SORT}$: degree $\varnothing$
- $\texttt{HALT}$: degree $\varnothing'$
- $\texttt{FIN}$: degree $\varnothing''$

Computability is about...

Computability is about...

- the study of models of computation (not necessarily realistic/practical)

Computability is about...

- the study of models of computation (not necessarily realistic/practical)
- the study of complexity measures and classes

Computability is about...

- the study of models of computation (not necessarily realistic/practical)
- the study of complexity measures and classes
- the study of alternative characterisations

Motivating example

Motivating example

Example (Sine function)

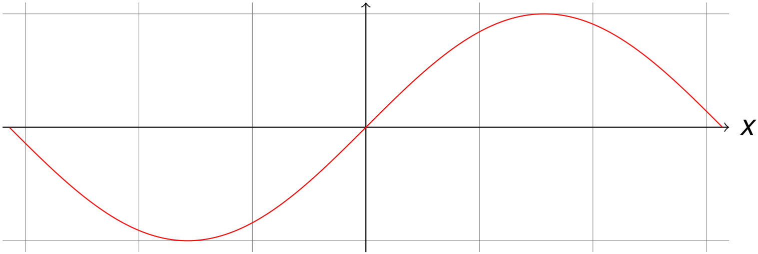
Given $x \in \mathbb{R}$, compute $\sin(x)$.

Motivating example

Example (Sine function)

Given $x \in \mathbb{R}$, compute $\sin(x)$.

$\Rightarrow$ "clearly sin is computable:"

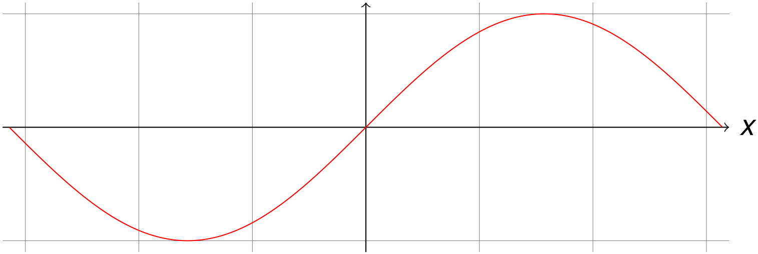


Motivating example

Example (Sine function)

Given $x \in \mathbb{R}$, compute $\sin(x)$.

$\Rightarrow$ "clearly sin is computable:"



But...

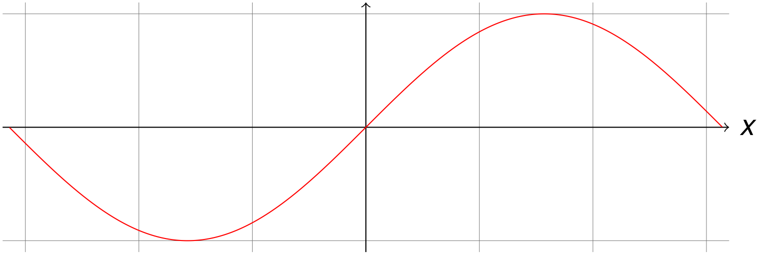
- how do you represent a real number ? (infinite object)

Motivating example

Example (Sine function)

Given $x \in \mathbb{R}$, compute $\sin(x)$.

$\Rightarrow$ "clearly sin is computable:"



But...

- how do you represent a real number ? (infinite object)
- what is a program working on them ?

Two classical approaches

Two classical approaches

Blum–Shub–Smale machine

Two classical approaches

Blum–Shub–Smale machine

- register machine (like your computer)...

Two classical approaches

Blum–Shub–Smale machine

- register machine (like your computer)...
- ...but registers can store real numbers...

Two classical approaches

Blum–Shub–Smale machine

- register machine (like your computer)...
- ...but registers can store real numbers...
- ...and can perform operations on them $(+, -, =, \dots)$

Two classical approaches

Blum–Shub–Smale machine

- register machine (like your computer)...
- ...but registers can store real numbers...
- ...and can perform operations on them $(+, -, =, \dots)$

$\Rightarrow$ Very algebraic, usually much more powerful than a Turing machine

Two classical approaches

Blum–Shub–Smale machine

- register machine (like your computer)...
- ...but registers can store real numbers...
- ...and can perform operations on them $(+, -, =, \dots)$

$\Rightarrow$ Very algebraic, usually much more powerful than a Turing machine

Computable analysis

Two classical approaches

Blum–Shub–Smale machine

- register machine (like your computer)...
- ...but registers can store real numbers...
- ...and can perform operations on them $(+, -, =, \ldots)$

$\Rightarrow$ Very algebraic, usually much more powerful than a Turing machine

Computable analysis

- a real number is a program:

Two classical approaches

Blum–Shub–Smale machine

- register machine (like your computer)...
- ...but registers can store real numbers...
- ...and can perform operations on them $(+, -, =, \ldots)$

$\Rightarrow$ Very algebraic, usually much more powerful than a Turing machine

Computable analysis

- a real number is a program: it computes arbitrary approximations

Two classical approaches

Blum–Shub–Smale machine

- register machine (like your computer)...
- ...but registers can store real numbers...
- ...and can perform operations on them $(+, -, =, \dots)$

$\Rightarrow$ Very algebraic, usually much more powerful than a Turing machine

Computable analysis

- a real number is a program: it computes arbitrary approximations
- a function is a program transformation:

Two classical approaches

Blum–Shub–Smale machine

- register machine (like your computer)...
- ...but registers can store real numbers...
- ...and can perform operations on them $(+, -, =, \ldots)$

$\Rightarrow$ Very algebraic, usually much more powerful than a Turing machine

Computable analysis

- a real number is a program: it computes arbitrary approximations
- a function is a program transformation: it transformes one approximation into another

Two classical approaches

Blum–Shub–Smale machine

- register machine (like your computer)...
- ...but registers can store real numbers...
- ...and can perform operations on them $(+, -, =, \ldots)$

$\Rightarrow$ Very algebraic, usually much more powerful than a Turing machine

Computable analysis

- a real number is a program: it computes arbitrary approximations
- a function is a program transformation: it transformes one approximation into another

$\Rightarrow$ Very analytic, approximation theory

Computable real

Computable real

Definition (Computable Real)

A real $r \in \mathbb{R}$ is computable is one can compute an arbitrary close approximation for a given precision:

Computable real

Definition (Computable Real)

A real $r \in \mathbb{R}$ is computable is one can compute an arbitrary close approximation for a given precision:

$$\text{Given } p \in \mathbb{N}, \text{ compute } r_p \text{ s.t. } |r - r_p| \leqslant 2^{-p}$$

Computable real

Definition (Computable Real)

A real $r \in \mathbb{R}$ is computable is one can compute an arbitrary close approximation for a given precision:

$$\text{Given } p \in \mathbb{N}, \text{ compute } r_p \text{ s.t. } |r - r_p| \leqslant 2^{-p}$$

Example

Rational numbers, π, e, ...

Computable real

Definition (Computable Real)

A real $r \in \mathbb{R}$ is computable is one can compute an arbitrary close approximation for a given precision:

$$\text{Given } p \in \mathbb{N}, \text{ compute } r_p \text{ s.t. } |r - r_p| \leqslant 2^{-p}$$

Example

Rational numbers, π, e, ...

Example (Non-computable real)

$$r = \sum_{n=0}^{\infty} d_n 2^{-n}$$

where

$$d_n = 1 \Leftrightarrow \text{the } n^{th} \text{ Turing Machine halts on input } n$$

Computable function

Definition (Computable function)

$f : [a, b] \to \mathbb{R}$ is computable iff $\exists m, \psi$ computable functions s.t $\forall n \in \mathbb{N}$:

- $\forall x, y, |x - y| \leqslant 2^{-m(n)} \Rightarrow |f(x) - f(y)| \leqslant 2^{-n}$ ▶ effective continuity
- $\forall r \in \mathbb{Q}, |\psi(r, n) - f(r)| \leqslant 2^{-n}$ ▶ approximability

Computable function

Definition (Computable function)

$f : [a, b] \to \mathbb{R}$ is computable iff $\exists m, \psi$ computable functions s.t $\forall n \in \mathbb{N}$:

- $\forall x, y, |x - y| \leqslant 2^{-m(n)} \Rightarrow |f(x) - f(y)| \leqslant 2^{-n}$ ► effective continuity
- $\forall r \in \mathbb{Q}, |\psi(r, n) - f(r)| \leqslant 2^{-n}$ ► approximability

Definition (Equivalent)

$f : [a, b] \to \mathbb{R}$ is computable iff $\exists M$ a Turing Machine s.t. $\forall x \in [a, b]$ and oracle $\mathcal{O}$ computing x, $M^{\mathcal{O}}$ computes $f(x)$.

Computable function

Definition (Computable function)

$f : [a, b] \to \mathbb{R}$ is computable iff $\exists m, \psi$ computable functions s.t $\forall n \in \mathbb{N}$:

- $\forall x, y, |x - y| \leqslant 2^{-m(n)} \Rightarrow |f(x) - f(y)| \leqslant 2^{-n}$ ► effective continuity
- $\forall r \in \mathbb{Q}, |\psi(r, n) - f(r)| \leqslant 2^{-n}$ ► approximability

Definition (Equivalent)

$f : [a, b] \to \mathbb{R}$ is computable iff $\exists M$ a Turing Machine s.t. $\forall x \in [a, b]$ and oracle $\mathcal{O}$ computing x, $M^{\mathcal{O}}$ computes $f(x)$.

Example

Polynomials, trigonometric functions, $e^{\cdot}$, $\sqrt{\cdot}$, ...

Computable function

Definition (Computable function)

$f : [a, b] \to \mathbb{R}$ is computable iff $\exists m, \psi$ computable functions s.t $\forall n \in \mathbb{N}$:

- $\forall x, y, |x - y| \leqslant 2^{-m(n)} \Rightarrow |f(x) - f(y)| \leqslant 2^{-n}$ ▶ effective continuity
- $\forall r \in \mathbb{Q}, |\psi(r, n) - f(r)| \leqslant 2^{-n}$ ▶ approximability

Definition (Equivalent)

$f : [a, b] \to \mathbb{R}$ is computable iff $\exists M$ a Turing Machine s.t. $\forall x \in [a, b]$ and oracle $\mathcal{O}$ computing x, $M^{\mathcal{O}}$ computes $f(x)$.

Example

Polynomials, trigonometric functions, $e^{\cdot}$, $\sqrt{\cdot}$, ...

Example (Counter-Example)

$$f(x) = \lceil x \rceil \qquad \blacktriangleright \text{ not continuous}$$

Thoughts on the model

- reuses existing theory on Turing machines

Thoughts on the model

- reuses existing theory on Turing machines
- gives "natural" complexity classes related to the classical ones

Thoughts on the model

- reuses existing theory on Turing machines
- gives “natural” complexity classes related to the classical ones
- but feels very discrete machine oriented

Thoughts on the model

- reuses existing theory on Turing machines
- gives “natural” complexity classes related to the classical ones
- but feels very discrete machine oriented

Question

Can we give a purely analog model of computation ?

GPAC

General Purpose Analog Computer

- by Claude Shanon (1941)

GPAC

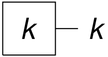
General Purpose Analog Computer

- by Claude Shanon (1941)
- idealization of an analog computer: Differential Analyzer

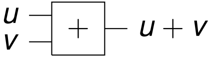
GPAC

General Purpose Analog Computer

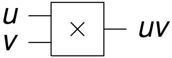
- by Claude Shanon (1941)
- idealization of an analog computer: Differential Analyzer
- circuit built from:



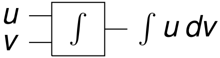
A constant unit



An adder unit



An multiplier unit



An integrator unit

GPAC: beyond the circuit approach

Theorem

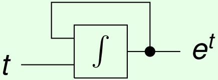
y is generated by a GPAC iff it is a component of the solution $y = (y_1, \ldots, y_d)$ of the ordinary differential equation (ODE):

$$\begin{cases} y' = p(y) \\ y(t_0) = y_0 \end{cases}$$

where p is a vector of polynomials.

GPAC: examples

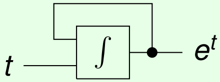
Example (One variable, linear system)



$$\begin{cases} y' = y \\ y(0) = 1 \end{cases}$$

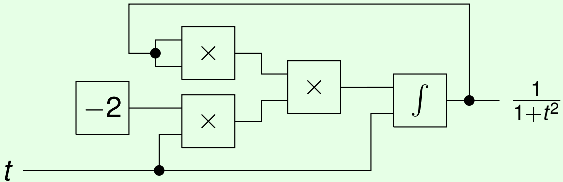
GPAC: examples

Example (One variable, linear system)



$$\begin{cases} y' = y \\ y(0) = 1 \end{cases}$$

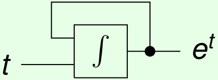
Example (One variable, nonlinear system)



$$\begin{cases} y' = -2ty^2 \\ y(0) = 1 \end{cases}$$

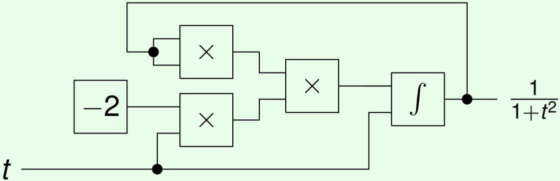
GPAC: examples

Example (One variable, linear system)



$$\begin{cases} y' = y \\ y(0) = 1 \end{cases}$$

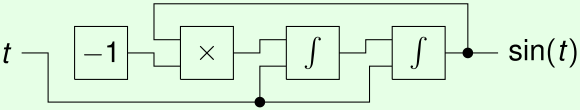
Example (Two variable, nonlinear system)



$$\begin{cases} y' = -2ty^2 \\ y(0) = 1 \\ t' = 1 \\ t(0) = 0 \end{cases}$$

GPAC: examples

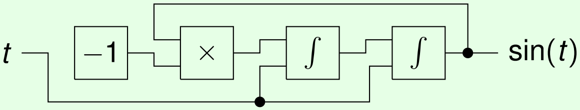
Example (Two variables, linear system)



$$\begin{cases} y' &= z \\ z' &= -y \\ y(0) &= 0 \\ z(0) &= 1 \end{cases}$$

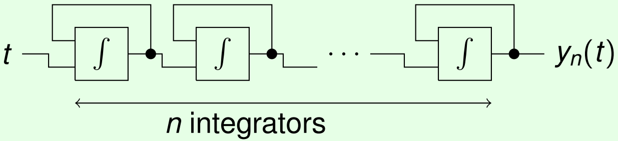
GPAC: examples

Example (Two variables, linear system)



$$\begin{cases} y' &= z \\ z' &= -y \\ y(0) &= 0 \\ z(0) &= 1 \end{cases}$$

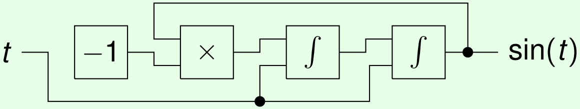
Exercice (Tear your mind apart)



$$\begin{cases} y_1' = y_1 \\ y_2' = y_2 y_1' \\ \vdots \\ y_n' = y_n y_{n-1}' \end{cases}$$

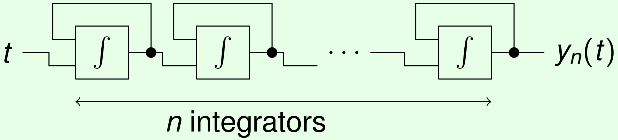
GPAC: examples

Example (Two variables, linear system)



$$\begin{cases} y' = z \\ z' = -y \\ y(0) = 0 \\ z(0) = 1 \end{cases}$$

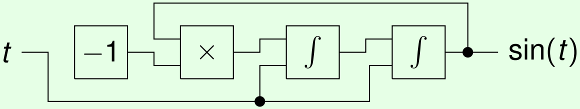
Exercice (Tear your mind apart)



$$\begin{cases} y_1' = y_1 \\ y_2' = y_2 y_1 \\ \vdots \\ y_n' = y_n y_{n-1} \cdots y_2 y_1 \end{cases}$$

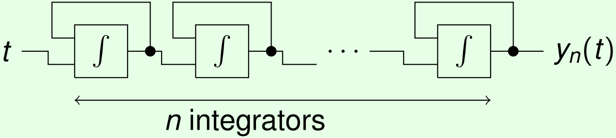
GPAC: examples

Example (Two variables, linear system)



$$\begin{cases} y' &= z \\ z' &= -y \\ y(0) &= 0 \\ z(0) &= 1 \end{cases}$$

Exercice (Tear your mind apart)



$$\begin{cases} y_1(t) = e^t \\ y_2(t) = e^{e^t} \\ \quad \cdots \\ y_n(t) = e^{e^{\cdot^{\cdot^{\cdot^{t}}}}} \end{cases}$$

Slight issue is...

- the GPAC generated functions are analytical

Slight issue is...

- the GPAC generated functions are analytical
- the computable functions from Computable Analysis are continuous

Question

Can we bridge the gap ? Why should we ?

The case of discrete computations

Many models:

- Recursive functions
- Turing machines
- λ-calculus
- circuits
- ...

The case of discrete computations

Many models:

- Recursive functions
- Turing machines
- λ-calculus
- circuits
- ...

And

Church Thesis

All reasonable discrete models of computation are equivalent.

GPAC: back to the basics

Definition

f is **generated** by a GPAC iff it is a component of the solution y of:

$$\begin{cases} y' = p(y) \\ y(t_0) = y_0 \end{cases}$$

GPAC: back to the basics

Definition

f is **generated** by a GPAC iff it is a component of the solution y of:

$$\begin{cases} y' &= p(y) \\ y(t_0) &= y_0 \end{cases}$$

Definition

f is **computable** by a GPAC iff $\exists p, q$ polynomials s.t. $\forall x \in \mathbb{R}$, the solution $y = (y_1, \dots, y_d)$ of:

$$\begin{cases} y' &= p(y) \\ y(t_0) &= q(x) \end{cases}$$

satisfies $f(x) = \lim_{t\to\infty} y_1(t)$.

GPAC: back to the basics

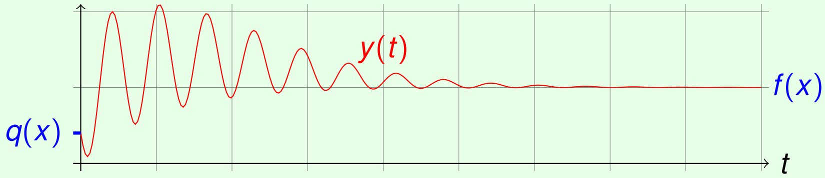
Definition

f is **computable** by a GPAC iff $\exists p, q$ polynomials s.t. $\forall x \in \mathbb{R}$, the solution $y = (y_1, \ldots, y_d)$ of:

$$\begin{cases} y' &= p(y) \\ y(t_0) &= q(x) \end{cases}$$

satisfies $f(x) = \lim_{t\to\infty} y_1(t)$.

Example



Computable Analysis = GPAC ? (again)

Theorem (Bournez, Campagnolo, Graça, Hainry)

f is GPAC-computable functions iff it is computable (in the sense of Computable Analysis).

Time Scaling

System	#1	#2
PIVP	$\begin{cases} y'(t) = p(y(t)) \\ y(1) = q(x) \end{cases}$	$\begin{cases} z'(t) = u(t)p(z(t)) \\ u'(t) = u(t) \\ z(t_0) = q(x) \\ u(1) = 1 \end{cases}$

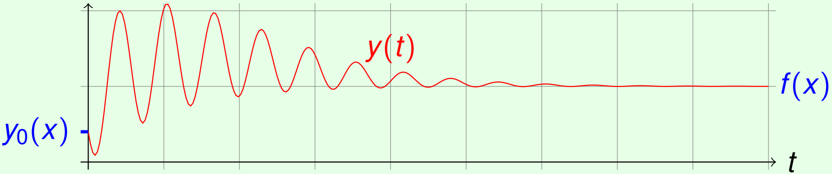
Time Scaling

System	#1	#2
PIVP	$\begin{cases} y'(t) = p(y(t)) \\ y(1) = q(x) \end{cases}$	$\begin{cases} z'(t) = u(t)p(z(t)) \\ u'(t) = u(t) \\ z(t_0) = q(x) \\ u(1) = 1 \end{cases}$

Remark

Same curve, different speed: $u(t) = e^t$ and $z(t) = y(e^t)$

Example



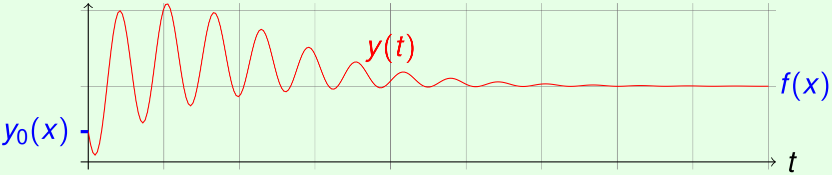
Time Scaling

System	#1	#2
PIVP	$\begin{cases} y'(t) = p(y(t)) \\ y(1) = q(x) \end{cases}$	$\begin{cases} z'(t) = u(t)p(z(t)) \\ u'(t) = u(t) \\ z(t_0) = q(x) \\ u(1) = 1 \end{cases}$
Computed Function	Same	

Remark

Same curve, different speed: $u(t) = e^t$ and $z(t) = y(e^t)$

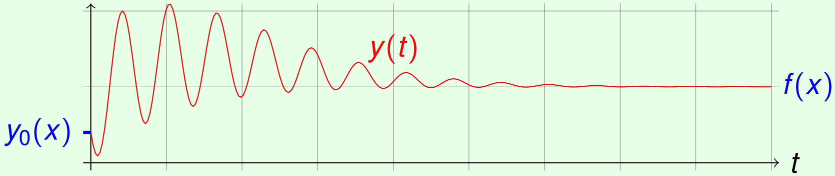
Example



Time Scaling

PIVP	$y' = p(y)$	$z(t) = y(e^t) \to \begin{cases} z' = up(z) \\ u' = u \end{cases}$
Computed Function	Same	
Convergence		Exponentially faster

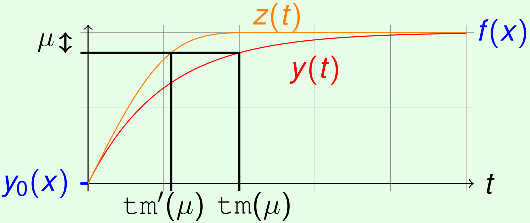
Example



Time Scaling

PIVP	$y' = p(y)$	$z(t) = y(e^t) \rightarrow \begin{cases} z' = up(z) \\ u' = u \end{cases}$
Computed Function	Same	
Time for precision μ	$\mathtt{tm}(\mu)$	$\mathtt{tm}'(\mu) = \log(\mathtt{tm}(\mu))$

Example



$$\|y_1(\mathtt{tm}(\mu)) - f(x)\| \leqslant \mu$$

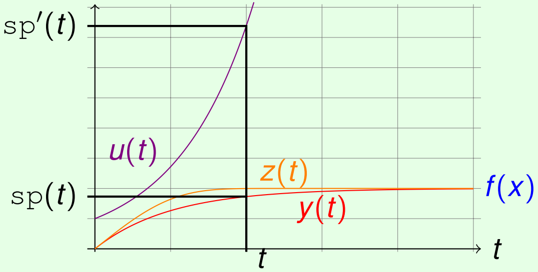
Remark

$\mathtt{tm}$ is not a good measure of complexity.

Time Scaling

PIVP	$y' = p(y)$	$z(t) = y(e^t) \rightarrow \begin{cases} z' = up(z) \\ u' = u \end{cases}$
Computed Function	Same	
Time for precision μ	$\mathtt{tm}(\mu)$	$\mathtt{tm}'(\mu) = \log(\mathtt{tm}(\mu))$
Bounding box for PIVP at time t	$\mathtt{sp}(t)$	$\mathtt{sp}'(t) = \max(\mathtt{sp}(e^t), e^t)$

Example



$$\mathtt{sp}(t) = \sup_{\xi\in[1,t]} \|y(\xi)\|$$

$$\mathtt{sp}'(t) = \sup_{\xi\in[1,t]} \|z(\xi), u(\xi)\|$$

Time Scaling

PIVP	$y' = p(y)$	$z(t) = y(e^t) \to \begin{cases} z' = up(z) \\ u' = u \end{cases}$
Computed Function	Same	
Time for precision μ	$\mathtt{tm}(\mu)$	$\mathtt{tm}'(\mu) = \log(\mathtt{tm}(\mu))$
Bounding box for PIVP at time t	$\mathtt{sp}(t)$	$\mathtt{sp}'(t) = \max(\mathtt{sp}(e^t), e^t)$

Remark

- $\mathtt{tm}(\mu)$ and $\mathtt{sp}(t)$ depend on the convergence rate

Time Scaling

PIVP	$y' = p(y)$	$z(t) = y(e^t) \rightarrow \begin{cases} z' = up(z) \\ u' = u \end{cases}$
Computed Function	Same	
Time for precision μ	$\mathtt{tm}(\mu)$	$\mathtt{tm}'(\mu) = \log(\mathtt{tm}(\mu))$
Bounding box for PIVP at time t	$\mathtt{sp}(t)$	$\mathtt{sp}'(t) = \max(\mathtt{sp}(e^t), e^t)$
Bounding box for PIVP at precision μ	$\mathtt{sp}(\mathtt{tm}(\mu))$	$\max(\mathtt{sp}(\mathtt{tm}(\mu)), \mathtt{tm}(\mu))$

Remark

- $\mathtt{tm}(\mu)$ and $\mathtt{sp}(t)$ depend on the convergence rate
- $\mathtt{sp}(\mathtt{tm}(\mu))$ seems not

Proper Measures

Proper measures of “complexity”:

- time scaling invariant
- property of the curve

Proper Measures

Proper measures of “complexity”:

- time scaling invariant
- property of the curve

Possible choices:

- Bounding Box at precision $\mu \Rightarrow$ Ok but geometric interpretation ?

Proper Measures

Proper measures of “complexity”:

- time scaling invariant
- property of the curve

Possible choices:

- Bounding Box at precision $\mu \Rightarrow$ Ok but geometric interpretation ?
- Length of the curve until precision $\mu \Rightarrow$ Much more intuitive

Questions ?

- Do you have any questions ?