Algorithmic Intelligence

Bruno Buchberger

Conference “Symbolic Computation in Software Science”
Tokyo University of Science
August 28, 2024

buchberger.bruno@gmail.com

Overview:

Questions
Automated Programming and Symbolic Computation
Automated Programming and Machine Learning
A Bit of Philosophy (... maybe skip)
Conclusion

Some parts of this talk are contained in: B. Buchberger. Automated programming, symbolic computation, machine learning : my personal view. Annals of Mathematics and Artificial Intelligence (2023) 91 : 569–589.

Questions

Will “AI” outperform human mathematicians, programmers, ... in inventing, proving, applying mathematics, software, ... ?

Will mathematicians, programmers, ... become jobless?

Is math, computer science, ... education obsolete?

Will “machines” ... outperform, control, exclude, get rid of, ... humans ...? ... Will AI program itself? Will the planet finally “enjoy” a “human-free robot society” ...?

Are “machines” intelligent? Do they have consciousness?

Let’s discuss views, find answers, ...by just clarifying notions and their relations ...

Clarity is a firm basis for individual and societal action (in research, engineering, education, politics, ...)

Notions: .... Consciousness, Reflexion, Intelligence, Mathematics, Models, Knowledge, Algorithms, Machines, Computation, Software, Symbolics, Numerics, Programming, Automated Reasoning, Automated Programming, Machine Learning, Artificial Intelligence, Algorithmic Intelligence, ...

Automated Programming and Symbolic Computation

Notation

We consider a (fixed) programming language (like C, Python, ...) and define:

$p[\,d]$:= the result (a data value) of applying program p to input data d.

We also consider a (fixed) logic language (like predicate logic, ...) and define:

$q[\,d]$:= the result (a truth value) of evaluating the formula q on input data d.

(With some generosity / sloppiness, ... most of the usual languages can be considered as both a logic and a programming language ....

and I use the words “program” and “algorithm” interchangeably.)

Programming

Programming is the following task: given a problem, find a program that solves the problem.

Roughly, a problem can be specified by a formula q that explains, for all data d and e, whether or not e is an admissible answer to the problem input (“problem instance”) d.

One often presents problems by the following wording: Given d, find e such that $q[d, e]$.

A program p solves problem q iff

for all d, $q[\mathrm{d}, p[d]]$. (The “correctness statement” for program p w.r.t. problem q.)

(Here, I only consider “explicit” problem specifications ... of the above form.)

Proving (arguing the truth ... of) the correctness statement is an integral part of problem solving.

Programming Needs “Intelligence”

Programming can be easy for some problems. Example: ...

Programming may need some well trained mind. Example: ...

For some problems, finding an algorithm was many years open but, finally an algorithm was found. Example: ...

For some problems, finding an algorithm is still open. Example: ...

For some problems, finding a “good” algorithm is still open. Example: ...

For some problems, finding an algorithm was proved to be impossible. Example: ...

The World of Global Programming

By the work of thousands / millions of “programmers” (algorithm inventors, mathematicians) over the past centuries, decades, years:

- We have thousands of algorithms for fundamental problems (made available in well organized algorithm libraries).
- Millions / billions of complex programs for easy and more and more difficult problems are composed from the fundamental algorithms (made available in “software systems” for thousands of “applications”).
- These programs are executed on billions of computers (that become faster, bigger, smaller, ... at an impressive rate).
- A growing flood of data (in / out) is produced.
- By the internet, these computers, the software systems, and the data are globally interconnected forming kind of a “global computer”.
- The global computer interconnects the users and becomes the digital memory and the digital processing power of a “global digital society”.
- The global society is the global programmer and the global user of the global computer.

Programming Needs “Intelligence” (Let me call it “Algorithmic Intelligence”)

“Algorithmic Intelligence”: More generally, “formal intelligence” (= “proving intelligence” + “algorithmic intelligence”), “reasoning intelligence”, ...

Don’t tell this to outsiders:

“Programming” (solving problems by provably correct algorithms) is the most intelligent human activity.

Why?

- Programming is the abstract form of all problem solving (= composing solutions for problems from available solutions for sub-problems).
- Programming is the essence of mathematics and is essentially mathematics.

(Don’t tell this to certain computer scientists. Don’t tell this to certain mathematicians.)

Programming needs “hot” (creative, intuitive, artistic, ...) and “cool” (impeccable, unaffected, merciless ...) intelligence.

(Please tell this to politicians, artists, humanists, philosophers...)

Can Programming be Automated? Can "Intelligence" be Automated?

This question is not new: Mathematics, in essence, is “reflexive” (= “meta”)!

Reflexiveness of mathematics: After struggling with solving a few (many) problems in a certain class of problems individually (each by an extra method (algorithm)), one “looks over one’s own shoulder” (one acts “self-reflexively”) and tries to find one method (algorithm) that solves all problems in that class:

Instead of

finding a program p_1 for problem q_1 such that, for all d, $q_1[d, p_1[d]]$,

finding a program p_2 for problem q_2 such that, for all d, $q_2[d, p_2[d]]$,

...

finding a program p_m for problem q_m such that, for all d, $q_m[d, p_m[d]]$,

try to find a “general algorithm” G such that, for all q in an infinite class of problems (in which q_1, ..., q_n are contained),

for all d, $q[d, G[q, d]]$.

Note: the general algorithm G has the problem q as an input parameter! (One also says: “G is on the meta-level of q”.)

Reflexiveness: Already at the Beginning of Universal Programming

Kurt Gödel, who introduced "universal programming" in 1930 (under a different name), already proved in 1931 that going to meta-levels, ... has no upper bound!

Alan Turing, who introduced an abstract version of the universal computer in 1936, already proved soon afterwards that there exists a "universal interpreter" U that explains the execution of all other programs, i.e. U has the property that

for all programs p and data p, $U[p, d] = p[d]$.

Edgar F. Codd, in 1968 constructed a "cellular automaton" that was able to reproduce itself.

And, practically speaking, soon after the first technical realizations of universal computers (by H. Aiken, K. Zuse and others) around 1940, people started to program compilers that work on programs (of a "higher level language") and construct programs (of a "lower level language").

An interesting exercise: Think about other examples of going to the meta-level in the early and later days of computing.

Example for Going to the Meta-Level for Fundamental Math Problems

Example: There existed various algorithms for canonical simplifiers and solvers for certain classes of “multivariate polynomial equation systems”, e.g.

Gauß’ algorithm for linear multivariate polynomial systems,

Euclid’s algorithm for non-linear univariate polynomial systems.

Meta-level (generalization): Gröbner bases algorithm (BB 1965) for arbitrary non-linear multivariate polynomial systems.

Example for Going to the Meta-Level for Fundamental Math Problems

Example: There existed various algorithms for certain classes of “indefinite integration”, e.g.

indefinite integration of rational functions by partial fraction decomposition,

indefinite integration of certain “nice” integrands by heuristic application of certain basic “integration rules” (e.g. substitution rule).

Meta-level (generalization): Risch’ algorithm (1968) for all “elementary integrands” based on Liouville’s theory (19th century).

Examples for Going to the Meta-Level for Fundamental Math Problems

Example: There existed various algorithms for certain classes of “symbolic summation”, e.g.

indefinite summation of (roughly) rational functions by Gosper’s algorithm,

indefinite summation of certain “nice” summands by heuristic application of certain basic “summation rules” (see D. Knuth 1968).

Meta-level (generalization): An entire new arsenal of algorithms for the general class of hypergeometric summands by ..., the JKU RISC-group led by P. Paule, now C. Schneider,..., and others, since 1990.

Symbolic Computation: The Meta-Level of Fundamental Math Problems

Going to the meta-level of finding algorithms for classes of fundamental math problems (as in the above examples) is the first big part of what is called "symbolic computation".

There is a second big branch of "symbolic computation": The automation of programming in general.

The automation of programming has two ingredients:

- (meta-) algorithms that construct programs *p* from given problem specifications *q*,
- (meta-) algorithms that prove the correctness of formulae, in particular the correctness statements of programs *p* for problems *q*.

This branch of symbolic computation has various other names, like "automated reasoning", "automated theorem proving", "computer-aided demonstration", "computational logic" etc.

This is the notion of "Symbolic Computation" as introduced in the editorial of the Journal of Symbolic Computation, BB 1985.

Numerical Computation: ... If the “Ideal Case” Is Not Possible

Numerical computation is “the face of mathematics to the outside world”. Enormously important.

If “exact” algorithms for a given problem q are (theoretically or practically) not possible, numerical computation has two approaches:

- formulate an “approximative version” q^* of the problem with some tolerance ϵ as an extra input parameter,
- in case a specification of q “in general terms” is not possible but only finitely many input/output example pairs for q are known: we have a “fitting problem” (in a very general sense). (All of “machine learning” falls into this category, see below!)

The “Brain Power Constancy Hypothesis”

One might think: the higher the meta-level, the more “algorithmic intelligence” is needed to find algorithms on that level.

However, after some more analysis, I tend to believe that the following principle holds:

Brain Power Constancy Hypothesis (“The Reflexion Principle”):

> *The human brain power for problem-solving (including “algorithmic intelligence”) did not change over the past, say, ten thousands years and it will not drastically change over the next ten thousand years.*
>
> *The spectacular increase in the problem-solving capacity and the dramatic acceleration in the increase is a consequence of applying the constant brain power in higher and higher rounds of going from the object level to the meta-level. In one round, the objects of the previous round become the actors on the next round. (I call this transition “reflexion”.)*

I discuss the reflexion principle in more detail and more generality in the first part of my new book: B.B. Science and Meditation: Creating the Future. Amazon.de. 2014. See also below: “A Bit of Philosophy”.

The Misunderstanding about “Artificial Intelligence”

What seems to be “hard” in one round of going through the meta-levels, after invention of an algorithm, becomes easy on the next level.

Therefore, in the early years of computing and programming, the algorithmic solution of the above kind of mathematical problems was also considered to be part of what some people wanted to call “artificial intelligence” or “machine intelligence”.

This terminology, however, was the source of a big misunderstanding: The “symbolic” algorithms on the higher levels of problem solving, like any other algorithms, are the creation of human intelligence. The machines on which they are executed are the same dull, completely unintelligent, machines as ever!

The field is just “human algorithmic intelligence” (= the human intelligence for inventing algorithms = the human intelligence for automating processes) on higher and higher levels!

The Race Between Human and Machine Intelligence

In other words:

- There is no race between human intelligence and machine intelligence.
- There is - and will always be - a race between human intelligence for solving each instance of a problem class extra and human intelligence that tries to establish one algorithm for all instances of this problem class!

In other words: Human intelligence on the higher level aims at making human intelligence on a lower level superfluous!

In other words: It is the goal of mathematics (computer science, ...) to trivialize itself!

This simple insight is not clear even to some (“pure” and “applied”) mathematicians:

- “I can do this and this ... integral faster than ‘the machine’!”
- “I can do integrals ‘the machine’ cannot do!”
- “I can translate a poem ‘the machine’ translated inappropriately”.

The First Approach to Automated Programming: Symbolic Computation

The path from a problem specification *q* to a program *p* that solves *q*, is a transformation of symbolic objects.

A huge research effort has been to automate or semi-automate this process. This is part of symbolic computation.

I do not want to go into any details about this approach in this talk. I have given many talks on this topic, in particular on my own approach by the “Lazy Thinking” method within the Theorema system. Much of the SCSS papers were on this topic.

A recent survey paper on Automated Programming: Armando Solar-Lezama. A Survey of Program Synthesis Techniques. MIT. 2023.

A survey on earlier research on this topic: D. Basin, Y. Deville, P. Flener, A. Hamfelt, J. F. Nilsson. Synthesis of Programs in Computational Logic. In: M. Bruynooghe, K. K. Lau (eds.), Program Development in Computational Logic, Lecture Notes in Computer Science, Vol. 3049, Springer, 2004, pp. 30-65.

A survey on very early research on this topic: B. Buchberger, Computer-Aided Algorithm Design (German). In: Proceedings of the Spring School in Artificial Intelligence. Teisendorf, March 15-24, 1982, W. Bibel, J. H. Siekmann (ed.), Informatik-Fachberichte 59,Springer -Verlag Berlin - Heidelberg - New York, pp. 141-202. 1982.

An Example of a General Automatic Programming Method: “Lazy Thinking” (BB 2002,...)

This method tries to be an “algorithm synthesis algorithm” (at least a “heuristics”) *S*, such that, for all problems q in a quite large class of problems

for all *d*, *q*[*d*, *S*[*q*] [*d*]]   (*)

The method is such that, for all *q*, if it succeeds, the program S[*q*] is guaranteed to satisfy the “correctness theorem” (*).

The method is implemented in the Theorema system (BB, W. Windsteiger et al . 1995 - ...)

For the following,
I assume you know what an induction proof is
and I assume you believe that induction proofs can be automatically generated (for a wide variety of domains and properties).

Algorithm Synthesis by "Lazy Thinking"

"Lazy Thinking" method for algorithm synthesis =
my personal advice to "humans" (or "computers") how to invent an algorithm for a given problem *q*.

Overall Strategy of Lazy Thinking: (Automatically) reduce problem *q* to a couple of (hopefully simpler) problems *Q*, ...

until ...

Three Key Ideas for Automated Lazy Thinking

Given: A problem (specification) *q*.   Find: An algorithm *A* for *q.*

- ♣ (Understand the problem "completely": Specification *q* must be spelled out and "complete" knowledge must be available on the notions that occur in the specification *q*.)

- ♣ Consider known fundamental ideas of how to structure algorithms in terms of sub-algorithms ("algorithm schemes *A*").
  Try one scheme *A* after the other.

- ♣ For the chosen scheme *A*, try to prove (automatically): for all *d*, *q*[*d*, *A*[*d*]]: From the failing proof construct (automatically) specifications for the sub-algorithms *B, ...* occurring in *A* that will turn the failing proof into a successful proof.

Example of an Algorithm Scheme ("Divide and Conquer"):

$$\left(A[d] = \begin{cases} S[d] & \Leftarrow \text{isBasic}[d] \\ M[A[L[d]], A[R[d]]] & \Leftarrow \text{otherwise} \end{cases} \right)$$

A is the unknown algorithm.

S, M, L, R are unknown sub-algorithms.

Example: Synthesis of Merge-Sort [BB et al. 2003]

Problem: Synthesize algorithm "sorted" such that

for all *d*, isSortedVersion[*d*, sorted[*d*]] .

("Correctness Theorem")

Knowledge on the Problem:

```
isSortedVersionOf[list1_, list2_] :=
 isSorted[list2] ∧ isPermutation[list1, list2]
isSorted[list_] := forAll[ {i, length[list] - 1},  list⟦i⟧ ≤ list⟦i + 1⟧ ]
...
```

etc. (approx. 20 formulae, see notebook of proofs in the Appendix.)

Now We Take, for Example, the Algorithm Scheme "Divide and Conquer"

"Divide and Conquer":

$$\left(A[d] = \begin{cases} S[d] & \Leftarrow \text{isBasic}[d] \\ M[A[L[d]], A[R[d]]] & \Leftarrow \text{otherwise} \end{cases} \right)$$

A is the unknown algorithm.

S, M, L, R are unknown sub-algorithms.

We now start an automated induction proof of the correctness theorem

for all d, $q[d, A[\mathrm{d}]]$

This proof will fail because nothing is know about the sub-algorithms S, M, L, R .

Automated Invention of Sufficient Specifications for the Subalgorithms , from a Failing Proof

“Sufficient” specifications: sufficient for turning the failing proof into a successful proof!

A simple (but amazingly powerful) rule (m ... an unknown subalgorithm):

Collect the temporary assumptions T[x0, ... A [...], ...]
and the temporary goals G[x0, ...m [A [...]]]

and produce the specification

$$\underset{X,\ \ldots,\ Y,\ \ldots}{\forall} \left(T[X,\ \ldots Y,\ \ldots] \Rightarrow G[X,\ \ldots m[Y]]\right).$$

Details: see papers [BB 2002, ...] and example (in Appendix 1).

This rule is the essence of my Lazy Thinking methods.

The Result of Applying Lazy Thinking in the Sorting Example

Lazy Thinking, automatically (in approx. 1 minute on a laptop using the *Theorema* system), finds the following specifications for the sub-algorithms that provenly guarantee the correctness of the above algorithm (scheme):

$$\underset{x}{\forall} (\mathtt{isBasic}[x] \Rightarrow S[x] = x)$$

$$\underset{y,z}{\forall} \left(\begin{matrix} \mathtt{isSorted}[y] \\ \mathtt{isSorted}[z] \end{matrix} \Rightarrow \begin{matrix} \mathtt{isSorted}[M[y, z]] \\ M[y, z] \approx (y \asymp z) \end{matrix} \right)$$

$$\underset{x}{\forall} (L[x] \asymp R[x] \approx x)$$

Note: the specifications generated are not only sufficient but natural ! They specify merge algorithms M and pairing algorithms L and R.

Now, what did we get automatically? A problem reduction ! All details of the intermediate steps: Appendix 1.

Now, we either have suitable *M, L, R* in our algorithm library, or we can apply the Lazy Thinking principle again until we arrive at basic algorithms in our library. (Library: Algorithms with there specifications and, maybe, other knowledge on the algorithms.)

Thus, by the Lazy Thinking Method, entire hierarchies of provenly correct algorithms can be generated in arbitrary domains.

How Far Can We Go With the "Lazy Thinking" Method ?

Can we automatically synthesize algorithms for non-trivial problems?

Example of a non-trivial (?) problem: construction of Gröbner bases (for non-linear multivariate polynomial systems).

"Non-trivial" part of the invention: The invention of the notion of S-polynomial and the characterization of Gröbner-bases by finitely many S-polynomial checks.

With the "Lazy Thinking" method, it is possible to invent the essential idea of my (age 23) Gröbner bases algorithm (1965) fully automatically. (Details: [BB 2005 (age 63), Craciun 2008]) and Appendix 2.)

Cum grano salis: I managed to create an avatar of my mathematical self.

Automated Programming and Machine Learning

A Completely Different Approach to Automatic Programming (Algorithm Synthesis): “Machine Learning”

Recall: An algorithm synthesis algorithm $\mathcal{S}$ for a class of problems is an algorithm that, for all problems q the class, satisfies

for all d, $q[d, \mathcal{S}[q][d]]$.

However, there is a huge class of practically important problems q, for which no general specification can be given. Rather, the problem is only “specified by examples”, i.e. we have

- a number of examples of *“input / output” pairs* $(d_1, e_1), \ldots, (d_n, e_n)$ for which $q[d_1, e_1], \ldots, q[d_n, e_n]$ holds.

and an “oracle” (typically the “user” or “a physical system”) that

- for any given d and e, can decide (or just “apodictically” determines) whether or not $q[d, e]$ holds.

For such a problems q, can one still find an algorithm, i.e. a p such that, for all d, $q[d, p[d]]$?

For an entire class of such a problems q, can one still find algorithm synthesis algorithm, i.e. an algorithm $\mathcal{S}$ such that, for all problems q in the class,

for all d, $q[d, \mathcal{S}[q][d]]$.

Can One Find an Algorithm for a Problem Specified by Examples?

Answer: In principle “no”. Of course, one can give an algorithm p (e.g. “table-look/up”) that gives output e_i for input d_i, for all $i = 1, .., n$. However, how can we be sure that $q[d, p[d]]$ will also be satisfied for all other d?

Only under some additional assumptions, one can find a suitable algorithm:

- One may know or “want to believe” that the algorithm should have the simple form of a linear function or a polynomial function or a series or ... (which can be considered as “algorithm schemata”).
- One may be satisfied with the result that the answers of the algorithm give the output not exactly but up to some “tolerance”.
- One may be satisfied with the result that the answers of the algorithm give a tolerable output only with some probability.

In this form, for many classes of problems, many “algorithm schemata” and many notions of “tolerance” finding algorithms (even automatically) is an old and rich area of mathematics with many different variants: interpolation, extrapolation, regression, fitting, polynomial approximation, series expansion (Taylor, Fourier, Walsh, ...) etc.

Example of a Problem “Specified by Examples”

An algorithm *p* should be found that produces the output values 1, 4, 2, 1, 3, 5, 8, 5 for the input values 1, 1.5, 2, 2.5, 3, 4, 5, 6.

Additional requirement: Between the given points the algorithm should be a polynomial of maximum degree 3 and, at the given points, the partial curves should "fit" (and the transition between the curve pieces should be “smooth”).

Algorithms that synthesize an algorithm for all problems of this type, are well known (and are examples of what is called “numerical computation”). Here is an implementation of such a method, called ‘Interpolation’ in Mathematica:

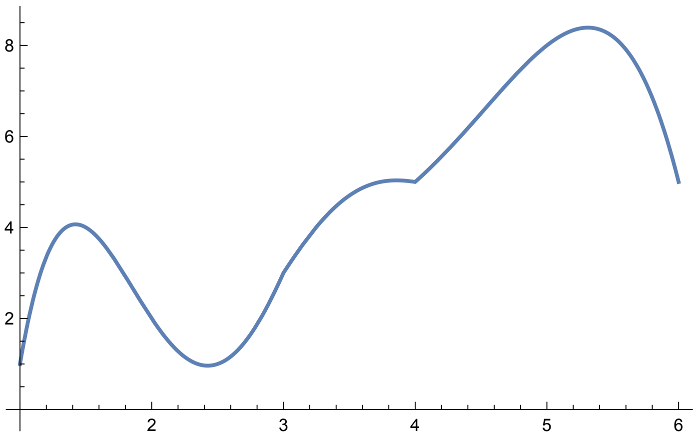
In[ ]:=
```
p = Interpolation[{{1, 1}, {1.5, 4}, {2, 2}, {2.5, 1},
    {3, 3}, {4, 5}, {5, 8}, {6, 5}}, InterpolationOrder → 4];
```

```
InterpolatingFunction[ Domain: {{1, 6}} Output: scalar ]
```

InterpolatingFunction : Input expression InterpolatingFunction [{{1, 6}}, <>] contains insufficient information to interpret the result.

In[ ]:=
```
Plot[p[x], {x, 1, 6}]
```

Out[ ]=



Interpolation: What Happens Inside?

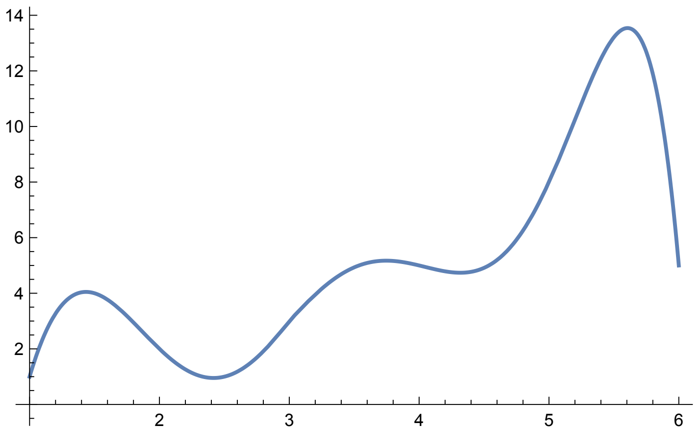
In[∘]:=
```
p = Interpolation[{{1, 1}, {1.5, 4}, {2, 2}, {2.5, 1},
    {3, 3}, {4, 5}, {5, 8}, {6, 5}}, InterpolationOrder → 3];
```

InterpolatingFunction[Domain: {{1, 6}} Output: scalar]

InterpolatingFunction : Input expression InterpolatingFunction [{{1, 6}}, <>] contains insufficient information to interpret the result.

In[∘]:=
```
Plot[p[x], {x, 1, 6}]
```

Out[∘]=



Note that ‘Interpolation’ is an algorithm synthesis algorithm: It takes a problem (specified by finitely many input / output pairs) as an input and produces an algorithm for “solving” the problem.

In more detail (in case InterpolationOrder -> 3): ‘Interpolation’ proceeds by setting on the curves between points , as a (cubic, ...) polynomial $c_{i3}\,x^3 + c_{i2}\,x^2 + c_{i1}\,x + c_{i0}$ (i = 1,...,5) and determining then the “coefficients”
c_{i0}, c_{i1}, c_{i2}, c_{i3} such that the conditions
(evaluation to the given outputs at the given inputs and fitting conditions) are satisfied.

More Challenging Example of a Problem “Specified by Examples”

Problem (a variant of the “relation extraction problem”): Given a phrase (in English), decide whether or not the phrase contains a part that expresses that “somebody cooperates with somebody else” and, if the answer is “yes” provide the sub-phrases that describe the two “somebodies”.

Why is it difficult / impossible to give a complete general specification of this problem?

(For most problems on natural language texts referring to “semantics” it is difficult / impossible to give a complete general specification. Why?)

However, “an oracle” (the community of English native speakers) can give lots of examples of input and corresponding output for this problem and can also decide whether a certain sentence expresses cooperation between certain people:

In: “Ann and Peter work together”. Out: “yes”, “Ann”, “Peter”.
In: “They sat in different rooms”. Out: “no”. (Coudn’t it be “yes”?)
In: “During the operation, the firefighters exchanged helpful information by mobile.” Out: “yes”, “the firefighters”.
In: “He uses to spend a lot of time watching TV.” Out: “no”.
...

How many examples of in / out would a person, who does not know the meaning of “cooperate”, need to get the meaning in the sense that he would be able to produce the correct answer for any of potentially infinitely many input phrases?

We want an algorithm that generates an algorithm for all such problems! Does interpolation work (after some numerical encoding of texts)?

A Different View on Interpolation: The “Learning” Perspective

A different way of looking at processes like interpolation (in fact, only a different wording for this process):

- We have certain “training data”: {{1,1},{1.5,4},{2,2},{2.5,1},{3,3},{4,5},{5,8},{6,5}}.
- In the “training phase”, the “learning method” ‘Interpolation’ takes a particular algorithm scheme (in this case a sequence of cubic polynomials) as an “ansatz” with “unknown coefficients”. The “learning method” “trains” the scheme by determining the coefficients from the training data.
  The scheme with the coefficients determined is called a “trained model” (i.e. an algorithm for the problem given by the input / output pairs).
- In the “application phase”, the “trained model” can be applied to other input data:

```
In[•]:= p[5.67]
        p[3.5602]
        p[8.9]
Out[•]=
        7.08408
Out[•]=
        4.25735
```

InterpolatingFunction : Input value {8.9} lies outside the range of data in the interpolating function. Extrapolation will be used.

```
Out[•]=
        -102.286
```

When is the “Learning” Perspective Appropriate?

```
In[∘]:= p = Interpolation[
          {{1, 1}, {1.5, 4}, {2, 2}, {2.5, 1}, {3, 3}, {4, 5}, {5, 8}, {6, 5}}];

        InterpolatingFunction[ Domain: {{1, 6}}  Output: scalar ]
```

The perspective of the process as a “learning process”, of course, only makes sense if one considers the process in the context of “big data”: Assume, in a particular applications, as the result of some measurements, one has one billion of input / output pairs and wants to “predict” the next 10 billion measurements. Then one could proceed, with the same mathematics, as follows:

- Partition the available input / output pairs in two sets: a “training set” and a “test set”.

- Training phase on the training set as above. This yields a trained model *p*.

- Test phase: Apply the model *p* to all inputs in the given input / output set and compare the outputs generated by the model with the given outputs. If they are “tolerably” similar, go to the application phase. If not, give up or use a much bigger training set.

- Application phase : One can now apply the "model" *p* to any of the (infinitely) many inputs d (in the input domain considered) and obtain the output *p*[*d*]. What “relevance” / “trust” one gives to *p*[*d*], depends on many factors that should be made as explicit as possible in order not to draw conclusions that will cause disappointment.

Is This Type of Learning “Intelligent”?

In[◦]:=
```
p = Interpolation[
   {{1, 1}, {1.5, 4}, {2, 2}, {2.5, 1}, {3, 3}, {4, 5}, {5, 8}, {6, 5}}];
```

```
InterpolatingFunction[ Domain: {{1, 6}} Output: scalar ]
```

The “learning” terminology applied to the above mathematical process seduces people to think that this process is “intelligent” since “learning” is something considered to be possible only for intelligent creatures. Hence, some people call this approach “machine learning” (machines learn!) or even “artificial intelligence”. However, this terminology does not really help. The process is as it is and it does a wonderful job whether one sees the process itself or any of the trained models as an “intelligent machine” or not.

One thing is clear, the process was invented by (intelligent) humans and, hence, is another example of what I like to call "human algorithmic intelligence".

After this warm-up with some well-known mathematical algorithms, we are now prepared for an analysis of current “machine learning” methods.

What is Different in Current “Machine Learning” Methods?

As analyzed above, “machine learning” methods are algorithm synthesis algorithms for the class of problems that are “specified by examples”.

(Here, we consider only “supervised learning”. Other variants of the learning approach can be analyzed and viewed in a similar way.)

There a couple of methods summarized under the catch word “machine learning”: neural networks, support vector machines, Baysian statistics, evolutionary algorithms, Large Language Models, ... Here, we only consider Neural Networks (NN). The other methods can be analyzed and viewed in a similar way.

The Neural Networks approach is different from polynomial interpolation, regression, fitting, etc.just by the type of algorithm schemata used as “ansatz” in the training phase.

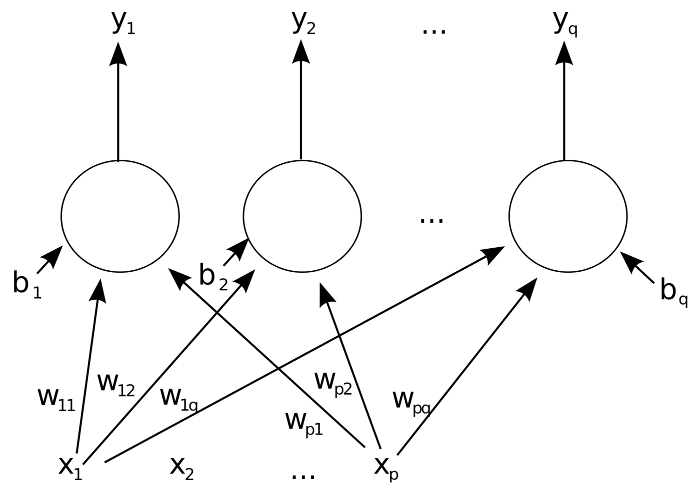
The Ansatz in the Neural Network Approach

Instead, of linear terms or polynomials or ..., in the NN approach, the “ansatz” (algorithm scheme) used in the training phase are nested “if-then-else” terms combined from terms of the form $x_1\, w_{12} + x_2\, w_{22} + \ldots + x_p\, w_{p2} - b_2$ etc. (enclosed by “activation functions” σ for technical reasons, in order to allow back-propagation of errors by differentiation ...).

Note that there is no “intelligence” in these terms! They are just another choice of terms that contain some “unknown” coefficients (called “weights”) w_{ij}, ... These unknown coefficients may be determined in an iterative training cycle in a similar way as described above for the simple interpolation method. When it is decided to stop the training cycle, the nested if-then-else term with concrete weight values (the “trained model”) is again an algorithm that produces output values from input values. It can then be used in the test phase and, if sufficiently good, in the application phase.

The invention of methods for determining the weights iteratively is again a product of human mathematical (algorithmic) intelligence.(E.g. JKU researcher Sepp Hochreiter who invented the “LSTM approach)”.

The particular form of these if-then-else terms with weights is motivated by the functioning of biological neural networks consisting of neurons and axons (in animals and humans)”, for example:



Drawing taken from: https : // en . wikipedia . org/wiki/Artificial_neural _network

The Practice of Neural Networks

Hundreds of “layers”. Hence, millions of unknown weights (“unknown coefficients”). Layers for “coarseness” of learned features (“deep learning”). Some arrows in the layers backward (“recurrent” NN).

Millions of labeled items in the training set. “Labeling” is a big issue!

Rule of thumb: Twice as much “accuracy” may cause 500 times computational effort.

Enormous practical success in recent years: natural language translation, image recognition, dialogues, ... (ushering in the “ChatGPT era”).

However, trained models do not “explain” anything about the structure behind the problem. (Research field: “Explainable AI”.)

Are Neural Networks or the “Models” Produced “Intelligent”?

Since the choice of these terms were motivated by biological neural networks, some people believe / wish / think ... that these networks and the algorithms (“models”) produced by them bear / mimic / grasp / “are” “intelligent”.

However, the algorithmic process based on these terms works (well or badly) independently of whether or not one considers these networks and the models produced as intelligent or not.

Also, the above if-then-else terms describe “real” biological neural networks only on a very superficial way. In fact, what happens in real neural networks and, in particular, the human brain and in the human “system” is much more complicated: Current biological research is far from “understanding the brain”. (However, biological research is much more advanced than, say, thirty years ago and advances at high speed.)

The people who invented the NN approach to “machine learning” and refine it permanently, though, are quite intelligent. Thus, NN-based “machine learning” is another example of “human algorithmic intelligence”. In contrast, the machines (computers) that execute the approach are as dull as ever ...

Going to the Meta-Meta-Level: Automatic Programming by ChatGPT et al.

How can LLM be used for generating programs for problems specified in general terms?

By going to the “meta-meta-level”:

- LLM systems like ChatGPT generate models that take NL texts (e.g. questions) as input and produce NL texts (e.g. answers to the questions) as output.
- The models were trained on large sets of pairs of input texts and output texts.
- Now, problem specifications are texts (mostly in NL) and appropriate programs that solve the problems specified are texts (in programming languages) often together with documentation (texts in NL or some formal language).
- Huge sets of pairs (problem specification, program that solves the problem) are available in program repositories like GitHub etc.
- LLMs have been trained also on these repositories and, therefore, can be used for generating program (program proposals; with documentation!) for problems specified in general terms!

This will change the world of programming dramatically and make many “low level programmers” jobless. See, however, my conclusions!

Automatic Programming by ChatGPT: Example

Prompt to ChatGPT: “Given a set of appointments in a calendar and a new appointment, check whether the new appointment collides with any of the appointments in the calendar. ...“

ChatGPT:

“Here is a solution in Python:

```
class Appointment:
  ...

def check_collision(appointments, new_appointment):
  for appointment in appointments:
    if appointment.start < new_appointment.end and appointment.end > new_appointment.start:
      return True
  return False


...
```

This solution creates an Appointment class to represent an appointment, with start, end, and text as its attributes. It then defines the check_collision function which takes in a list of appointments and a new_appointment and checks whether there is a collision by iterating through the list of appointments and checking if the start or end time of any of the appointments fall within the start and end time of the new appointment. If a collision is detected, the function returns True, otherwise it returns False.”

Automatic Programming by ChatGPT et al.

The answer is not only correct, it is also documented.

Note, however, that the explanation of the collision condition in the documentation is logically wrong whereas the formulation of the condition in the program is correct!

The logical mistake in the documentation is exactly the one which I encountered in experiments with human programmers!

Please compare this with the potential of quantifier elimination (a powerful symbolic computation method), which is able to start from the “common sense” explanation of the notion of collision (which involves a hidden existential quantifier) and, from there, produces the correct condition on the start and end times of the two appointments.

Sorry, no time and space for details! See: Buchberger, B . : Is ChatGPT Smarter Than Master' s Applicants?RISC Report Series 23 - 04, Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Altenberger Straße 69, 4040 Linz, Austria (2023)

The Interaction of Symbolic Computation and Artificial Intelligence

I use the following terminology:

“Artificial Intelligence” = “Symbolic Computation” (in the JSC sense) + “Machine Learning”.

Machine Learning is the only “new” (300, 70, 10 years old) method within “Artificial Intelligence”. Everything else was already done before (in symbolic or numeric computation).

Recall: Symbolic computation and machine learning are just mathematical methods whose invention and improvement needs human “algorithmic intelligence” and whose application is completely “unintelligent”, i.e.

“artificial intelligence” = human “algorithmic intelligence” + “artificial silliness”.

The next step in “algorithmic intelligence” with enormous theoretical and practical potential:

symbolic computation in interaction with machine learning.

Symbolic Computation Plus Machine Learning: The Next Level of Sophistication

Apply Machine Learning to problems specified in general terms (for which no algorithms or only computationally complex algorithms are known):

Example: Learn a “model” for computing Gröbner bases from many examples of input / outputs (see recent work by H. Kera et al.)

Example: Learn a “model” for the complexity of computing Gröbner bases from many examples of input / outputs.

Integrate the basic algorithms of ML/NLP/LLM (Machine Learning, Natural Language Processing, Large Language Models) into the algorithm library of mathematical software systems like Mathematica, Maple, etc.:

Example: Rich new ML/NLP/LLM library in Mathematica’s latest version 14.1.

Call symbolic computation algorithms from within the models generated by LLM:

Example: There is an option requesting the call of Mathematica functions from within ChatGPT.

Take the models generated by ML/NLP/LLM as a guess and then try to verify the correctness of the model by SC methods:

Very little work done in this area. I think this could be the way to go to make “automated programming” practical.

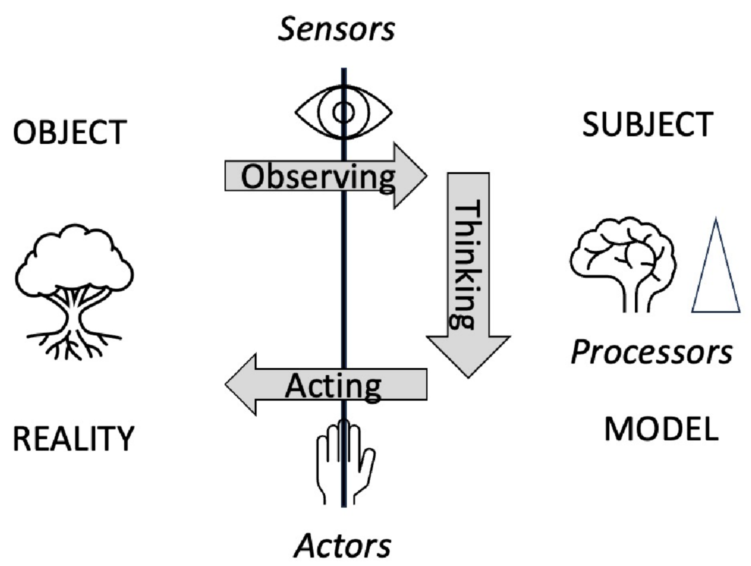
Use SC methods inside AI algorithms and for the generation and verification of AI software: ?

How could all this be combined to come up with a new level of “intelligence” and reliability in “Mathematical Knowledge Management” systems?

A Bit of Philosophy: Intelligence and Consciousness

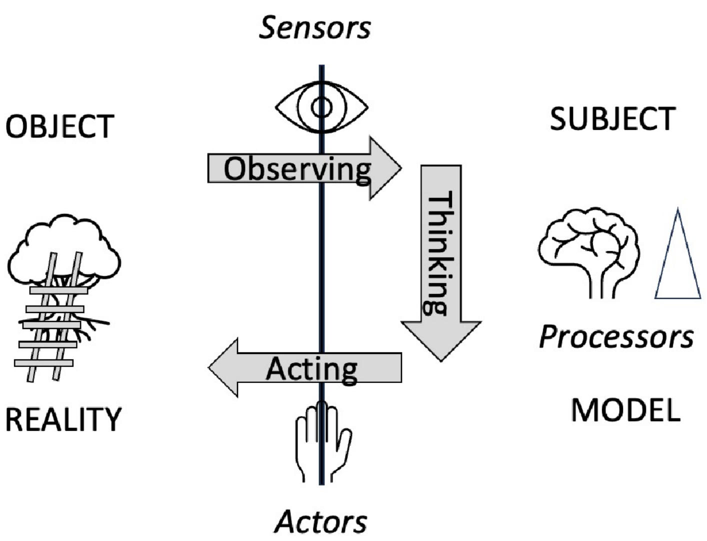
Interaction with Reality

We interact with a “reality” by iterating the triad step “observing - thinking - acting”. In short, let me call this the “intellectual approach” to reality:



Problem Solving by Intelligence

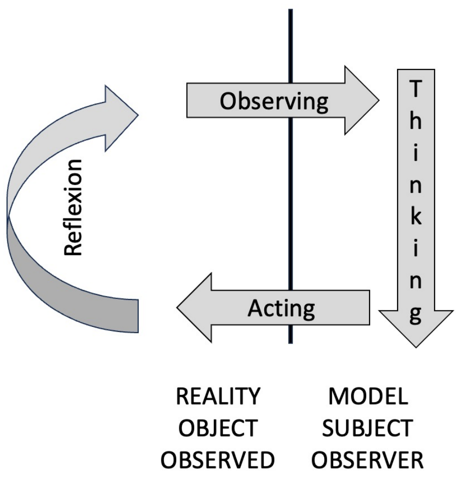
We encounter and solve problems in realities by the “intellect”:



“Reflexion”

By “reflexion” evolution proceeds to higher and higher levels (e.g. in Vedic Philosophy: reactions (Manas) ⟶ “free will” decisions (Buddhi)⟶ “I”-consciousness (Ahamkara)).

The result of “observing - thinking - acting” is also used for improving the methods of observing, the methods of thinking, the methods of acting.



Reflexion Is the Nature of Intelligence

Reflexion leads to the development of "devices".

Humans + devices ⟶ more and more efficient humans.

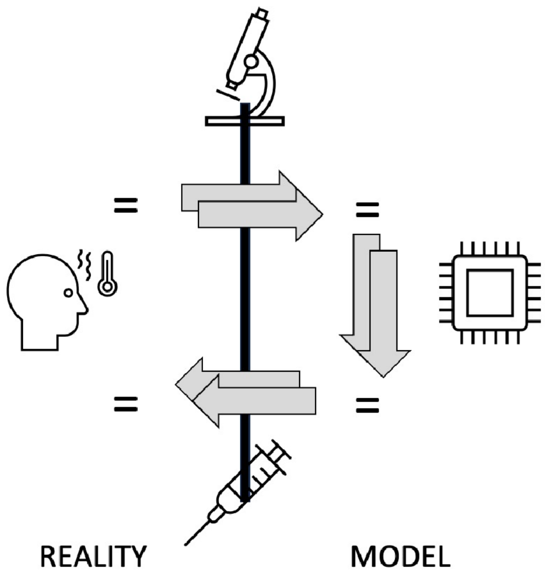
Reflexion is the intelligence of nature.

Reflexion is the nature of intelligence.

There is no upper bound to reflexion driven evolution.

Algorithmic intelligence is the embodiment of intelligence in humans in the past centuries.

AI, ML, LLM, ... just the most recent waves in this evolution. More and more waves will come in ever faster steps.



Consciousness

"Intelligence" and "Consciousness" are used interchangeably and in many different nuances.

In Vedic philosophy (and my view): "Pure Consciousness" is the absence of Observing ⟶ Thinking ⟶ Acting.

What remains is the experience of "Consciousness is conscious of itself" (pure reflexion, state achieved by "meditation").

In this sense, "machines" cannot be "conscious". "Machines" (algorithms) can be arbitrarily powerful ("look intelligent").

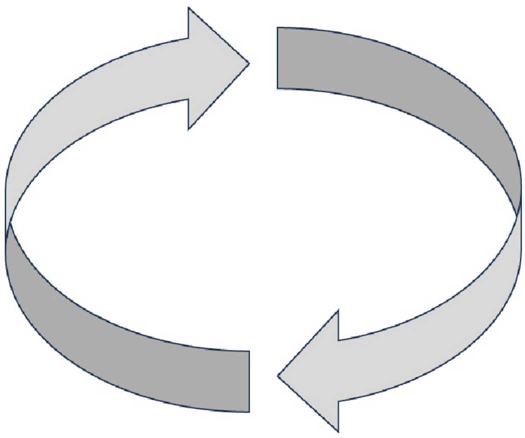


Figure 9

Conclusions

Mathematical (Algorithmic) Intelligence

Mathematics:

- Inventing and proving new knowledge (“theorems”) and methods from available knowledge and methods by reasoning (thinking).
- The “mathematical principle”: Think once deeply about the general case so that you do not have to think any more in infinitely many instances!
- In other words: The goal of mathematics is automation, i.e. to trivialize itself.

The three aspects of mathematics:

- Solving (providing methods for problems).
- Proving (theorems, e.g. correctness theorems for methods).
- Simplification (in particular “computation”; applying methods to problem instances / “input”)

Mathematical Intelligence (including “algorithmic intelligence”):

- Solving and proving needs intelligence.
- Simplification should not need (much, any) intelligence.

Automated Programming, Meta, Reflexion

Algorithmic Mathematics:

- Provide methods (“algorithms”, “programs”) whose application needs zero intelligence, i.e. can be executed by an idiot / “the computer”.
- Programming (= providing algorithms for problems), depending on the problem, may be easy, ..., extremely difficult, ... provably impossible.
- The essence of mathematics: Go to the meta-level (“reflexion”): provide algorithms that, for some class of problems, generate algorithms from problem specifications automatically (“automated programming”).
- The hierarchy of meta-levels has no upper bound. The higher the level, the more human algorithmic intelligence is needed. (? Maybe not, see the “Intelligence constancy principle”).

“Artificial Intelligence”

- Symbolic computation: automated programming for problems that are specified in general terms.
- Machine Learning: automated programming for problems that are specified only by examples.
- Some (many, ...) people like to call both, Symbolic Computation and Machine Learning, “Artificial Intelligence” (“Machine Intelligence”, ...).
- However, this terminology leads / led to a lot of misunderstandings:

  - In both, SC and ML, the machines (computer) are dull as ever.

  - The algorithms provided in SC and ML are the product of human algorithmic (mathematical) intelligence.

  - The dull computers with the programs established by intelligent SC and ML researchers may look “intelligent” from the outside but they are the product of human intelligence.

  - The fascinating algorithms invented throughout the history of algorithmic mathematics, in particular within the last decades, work fantastically independent of whether one calls them “intelligent” or not.

  - There is no race between intelligent machines and intelligent humans but, rather, between humans working on higher and higher levels of human algorithmic intelligence.

Mathematics: Highest Intellectual Aspiration

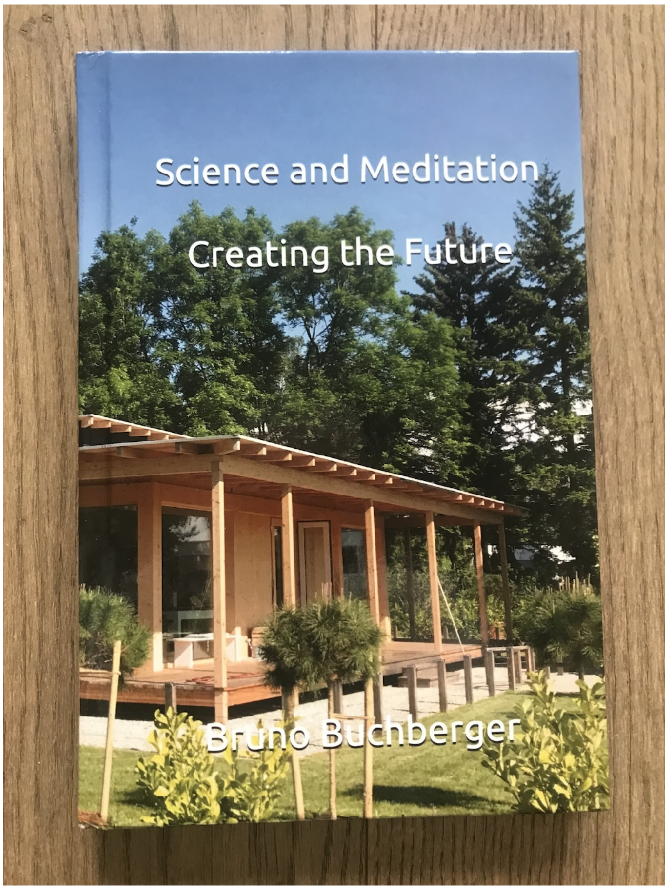
Working in the field of algorithmics (symbolic computation, numerical computation, machine learning, ...) is exciting:

- It needs the best minds (human algorithmic intelligence).
- The ultimate level in technical evolution can / should / must be done by humans.
- It needs highest formal education ("mathematics") to stay on top of technical evolution.
- Technical evolution has high scientific, technological, economical, societal, political, ... relevance.
- Preserving, extending the level of formal (mathematical) education will be indispensable for the survival of human society (and the planet).
- Is "survival" a sufficiently attractive goal ...?

Mathematics Is Not Sufficient for Happiness

Mathematical / formal / intellectual education alone is not sufficient for leading the next steps of human evolution.

- Today, more than ever, the application of the results of algorithmic human intelligence needs high ethical responsibility.
- The ethical aspect must be taken up by the researchers working in algorithmics and must not be left to people who do not understand the basics of algorithmics!
- Each of us must find their answer / path / ... to lead a responsible life in the technological age.
- The goal of evolution is “happiness for all beings”. How can this be achieved?
- My personal answer: “Science” + “Meditation”, see my new book (available at amazon.de, amazon.com):



THANK YOU!

You are welcome for questions and discussion by mail:

buchberger.bruno@gmail.com