Universität Augsburg
Mathematisch-Naturwissenschaftlich-Technische Fakultät

Towards exploiting he potential of computational thinking for math – design of a programming course

Reinhard Oldenburg, Bochum, 28.8.25

Agenda

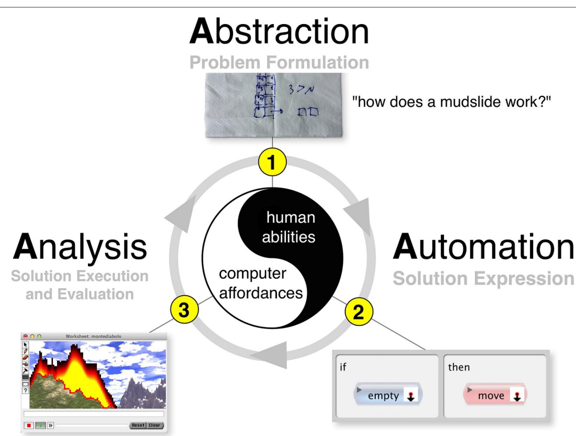
From theory to practice of teaching

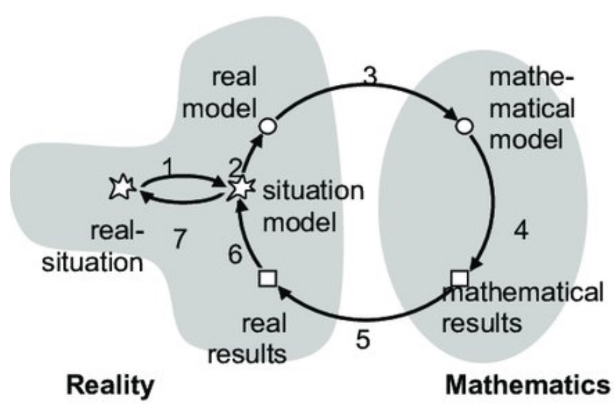
- Computational and mathematical thinking
  - What is the relation and how can CT and programming support MT?
- A “CT aware” course on programming for math teacher students
- Reflections

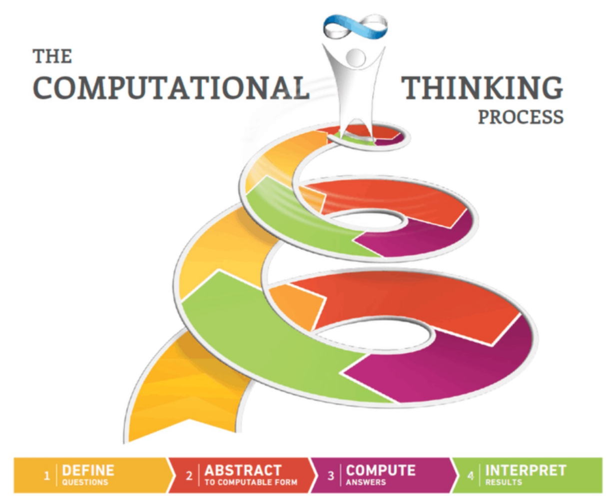
CT and MT

CT an overview

- Shute et al. (2017) summarizes Wing:
  - 1. Problem reformulation
  - 2. Recursion – Construct a system incrementally
  - 3. Problem **decomposition** – Break the problem down into manageable units.
  - 4. **Abstraction** – Model the core aspects of complex problems or systems.
  - 5. Systematic **testing** – Take purposeful actions to derive solutions."
- Selby & Woollard (2013) define CT as cognitive though process:
  - 1. the ability to think in **abstractions**,
  - 2. the ability to think in terms of **decomposition**,
  - 3. the ability to think algorithmically,
  - 4. the ability to think in terms of **evaluations**, and
  - 5. the ability to think in generalizations.
- There are obvious connections e.g. to modelling (Wing, Blum/Leiss, Wolfram)



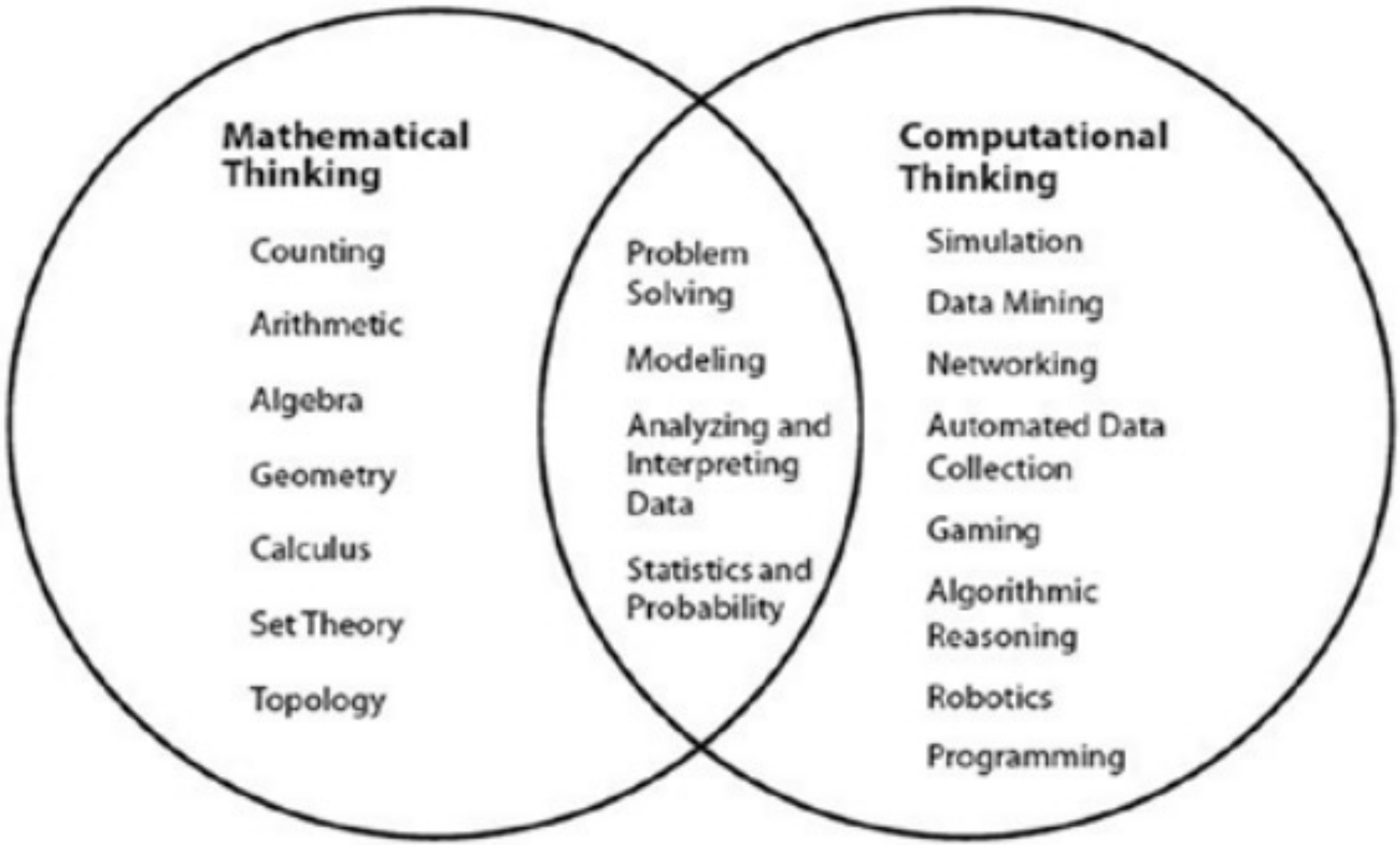




CT and MT

The relation

- Sneider et al. (2014) summarize the relation regarding content as follows:



CT and MT

The relation

- Knuth (1985!) looks at processes

	Formula manipulation	Representation of reality	Behavior of function values	Reduction to simpler problems	Dealing with infinity	Generalization	Abstract reasoning	Information structures	Algorithms
1 (Thomas)	*xx*	*xx*	*xx*						
2 (Lavrent'ev)	*xx*		*x*		*xx*				
3 (Kelley)	*x*					*xx*	*xx*		
4 (Euler)	*xx*		*xx*	*x*		*xx*			*x*
5 (Zariski)	*x*			*x*	*xx*	*x*	*xx*	*xx*	
6 (Kleene)	*x*					*xx*	*xx*		*x*
7 (Knuth)	*xx*	*x*		*x*					
8 (Pólya)	*xx*		*xx*	*xx*	*xx*				
9 (Bishop)	*xx*		*xx*	*xx*		*x*	*xx*	*xx*	*x*

CT and MT

The relation

- Weintorp et al. (2016) structure CT in math and science as follows:

Data Practices	Modeling & Simulation Practices	Computational Problem Solving Practices	Systems Thinking Practices
Collecting Data	Using Computational Models to Understand a Concept	Preparing Problems for Computational Solutions	Investigating a Complex System as a Whole
Creating Data	Using Computational Models to Find and Test Solutions	Programming	Understanding the Relationships within a System
Manipulating Data	Assessing Computational Models	Choosing Effective Computational Tools	Thinking in Levels
Analyzing Data	Designing Computational Models	Assessing Different Approaches/Solutions to a Problem	Communicating Information about a System
Visualizing Data	Constructing Computational Models	Developing Modular Computational Solutions	Defining Systems and Managing Complexity
		Creating Computational Abstractions	
		Troubleshooting and Debugging	

CT and MT

This widely underestimates the mutual relation of CS and Math!

- Programming helps students understand algebra
  - Evaluating expressions: Tall & Thomas (1991)
  - Bootstrap Algebra: Lee (2013), Schanzer et al. (2018)
  - Although: Bridge needs attention (e.g. Sutherland, 1989)
- Mathematical expressions are (small, functional) programs!
- Computers model math (e.g. Bundy 1986)
- Programming language notation can enhance precision formal presentation of math! (Sussman & Wisdom, 2000) cite Spivak (1965)

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial x} + \frac{\partial f}{\partial v}\frac{\partial v}{\partial x}.$$

Note that f means something different on the two sides of the equation!

CT and MT

This widely undere

- Mathematical expre
- Programming helps
  - Evaluating expressi
  - Bootstrap Algebra: l
  - Although: Bridge ne
- Computers model m
- Programming langu math! (Sussman & \

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial u}\frac{\partial u}{\partial x} + \frac{\partial f}{\partial v}\frac{\partial}{\partial}$$

Note that f means sc

The free particle

Consider again the case of a free particle. The Lagrangian is implemented by the procedure `L-free-particle`. Rather than numerically integrating and minimizing the action, as we did in section 1.4, we can check Lagrange's equations for an arbitrary straight-line path $t \mapsto (at + a_0, bt + b_0, ct + c_0)$

```
(define (test-path t)
  (up (+ (* 'a t) 'a0)
      (+ (* 'b t) 'b0)
      (+ (* 'c t) 'c0)))

(print-expression
 (((Lagrange-equations (L-free-particle 'm))
   test-path)
  't))
(down 0 0 0)
```

That the residuals are zero indicates that the test-path satisfies the Lagrange equations.[59]

Instead of checking the equations for an individual path in three-dimensional space, we can also apply the Lagrange-equations procedure to an arbitrary function:[60]

```
(show-expression
 (((Lagrange-equations (L-free-particle 'm))
   (literal-function 'x))
  't))
(* (((expt D 2) x) t) m)
```

CT and MT

Recipes, programs, typed expressions, and proofs

- Consider the following four situations:
  - A recipe defines a sequence of activities. If the ingredients are good, the end result is a certain product (e.g. a cake) of a certain quality.
  - A program is a sequence of commands. If the inputs make sense, it determines a meaningful output.
  - A term is a composition of sub-terms, and if the parts have certain types, then so does the result. For example, if x is a real number, then $\sqrt{1+x^2}$ is of type "nonnegative real number".
  - A proof is a sequence of deductions. If the hypotheses are true, then the proven statement is also true.
- In fact: Recipes ⊂ programs = typed expressions = proofs
- Example: Reasoning about programs and proving:

  If $a: A$ and $f: A \to B$, then conclude that $f(a): B$

  From A and $A \to B$, derive B
- Establishing $A \wedge B$ can be done by doing A and then doing B (sequence of operations)
- Curry-Howard correspondence: Programs in typed functional languages and proofs are the same in different syntax (Curry & Feys, 1958; Mimram, 2020)

CT and MT

Recipes, programs, typed expressions, and proofs

- Curry-Howard correspondence: Programs in typed functional languages and proofs are the same in different syntax
  - Programs/Expressions = Proofs
  - Types of Expressions = Propositions they prove
- Deep theory : Different logics
- Relevant for teaching? Yes: O.2024
- A metaphor
  - Program = collection of instruction steps that manipulate the computer's working memory to produce a result
  - Proof = collection of derivation steps that manipulate the reader's brain (working memory) such that the final state is belief in the correctness of the proposition

CT and MT

Recipes, programs, typed expressions, and proofs

- Still: Is this important down to earth? Now: Joint Work with Michael Fischer
- Weber & Tanswell (2022): Proofs as recipes. An example:

Theorem 3.57 (Bolzano-Weierstrass). Every bounded sequence of real numbers has a convergent subsequence.

[1] Proof. Suppose that (x_n) is a bounded sequence of real numbers.

[2] Let $M = \sup_{n \in \mathbf{N}} x_n$, $m = \inf_{\in \mathbf{N}} x_n$,

[3] and define the closed interval $I_0 = [m, M]$.

[4] Divide $I_0 = L_0 \cup R_0$ in half into two closed intervals, where $L_0 = [m, (m+M)/2]$, $R_0 = [(m+M)/2, M]$.

[5] At least one of the intervals L_0, R_0 contains infinitely many terms of the sequence, meaning that $x_n \in L_0$ or $x_n \in R_0$ for infinitely many $n \in N$ (even if the terms themselves are repeated).

[6] Choose I_1 to be one of the intervals L_0, R_0 that contains infinitely many terms.

and choose $n_1 \in \mathbf{N}$ such that $x_{n1} \in I_1$.

[7] Divide $I_1 = L_1 \cup R_1$ in half into two closed intervals.

[8] One or both of the intervals L_1, R_1 contains infinitely many terms of the sequence.

[9] Choose I_2 to be one of these intervals and choose $n_2 > n_1$ such that

…

Note: The proof is not in executing, but in writing the recipe for execution!

There is computational content in proofs!

CT and MT

Programs=Proofs in different syntax – This raises questions:

- Is MT=CT? I think: No. Difference in informal aspects, mental models
- Can CT help to improve especially proving skills?
  - Maybe: Nurlaelah et al. (2025): Improving mathematical proof based on computational thinking components for prospective teachers in abstract algebra courses

CT	**Proof**
Formalization / Specification	Axiomatization / Statement formulation
Decomposition / Disassembly	Splitting proof into lemmas
Abstraction / Parameterization	Generalization / Introducing variables / Transition from example to general
Precondition / Postcondition	Prerequisite / Conclusion
Evaluation - Debugging	Proof validation - Closing Gaps in Evidence
Pattern recognition - Design patterns	Concept of Evidence - Area-specific proof strategies
Algorithmization a) Sequence `A; B` b) Case split: `if a then A else B` c) Transformation through functional application d) Recursion / Iteration	Proof steps (proof construction) a) Proof of $A \wedge B$ b) Proof of with indicator what has been proven$A \vee B$ c) Proof with modus ponens (implication) d) Proof by induction / succession

UNI

The challenge

How can all this become effective in introductory programming courses?

- … especially for teacher students!
- Honest answer:
- But here are the principles in my course on programming for teacher students
  - Reducing complexity for teaching: Subset of programming language
  - Using few but clearly defined concepts
  - Using variable roles (Ben-Ari & Sajaniemi, 2003) as low-level “design patterns”
  - Showing typical use cases ; emphasize similarities math and programming
  - Emphasizing computability over performance (slow but transparent)
  - Genetic introduction of concepts
  - Emphasizing types (→ type theory) and using functions as first-class objs. (→ lambda)
  - Relevant applications: CT as part of empowerment
  - Intellectual challenges, e.g. limits of computability
  - Outlook: Showing the power of professional tools
- Practical considerations
  - Python (although Julia or Haskell are tempting) – because its popular and friendly
  - JupyterLab-Notebooks: eases interactive experimentation, allows graphing

Outline of the CT programming course

1 Computers compute

- Using Jupyter notebook as advanced pocket calculator. E.g. `4*17, 12/5, 12//5, 3**2, math.sqrt(8)`…
- Variables introduced as “memory”: references to numbers/objects
  - Concept of variable-value table: Valuation in mathematical logic!

Variable	Value
a	5
x	0,7
b2	13

- First awareness of types: $4/2 \dashrightarrow 2.0$, but $4//2 \dashrightarrow 2$, cf. `type(2)` vs. `type(2.0)`
- Wing (2006): “type checking as generalization of dimensional analysis”
- Types determine what operations can be done
  - In Scratch and Snap the form of blocks indicates their types!

Outline of the CT programming course

2 Turtle graphics

- Drawing squares, regular polygons, stars, houses, …..
- Code like

  ```
  import turtle
  turtle.forward(100)
  turtle.left(90)
  turtle.forward(100)
  turtle.left(90)
  turtle.forward(100)
  turtle.left(90)
  turtle.forward(100)
  turtle.left(90)
  ```

  naturally (genetic method) calls for loops: `for i in range(a,b)`:
- Procedural abstraction: It is natural (genetic!) to define e.g. rectangles ones for later use:

  `def rectangle(a,b):` …
- This is the second role of variables: parameters to represent not yet known numbers – cf. generalization in algebra education: Abstraction
- Case split with `if` needs Boolean logic: `and, or, not`

Outline of the CT programming course

2b Turtle graphics and recursion

- Interlude: Define functions for typical tasks, e.g. volume of a cylinder, …
- Interlude: First recursive function: factorial: Recursive and iterative

```
def fakR(n):
    if n<=1: return 1
    return n*fakR(n-1)
def fakI(n):
    f=1
    for i in range(1,n+1):
        f=f*i
    return f
for i in range(1,7):
    print(["Fakultät von",i," ist ",fakR(i),fakI(i)])
```

f: Variable as accummulator

i: Variable as counter

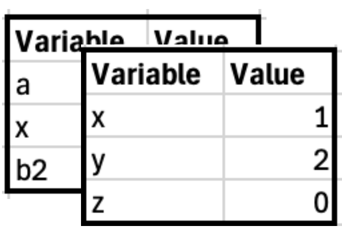
```
['Fakultät von', 1, ' ist ', 1, 1]
['Fakultät von', 2, ' ist ', 2, 2]
['Fakultät von', 3, ' ist ', 6, 6]
['Fakultät von', 4, ' ist ', 24, 24]
['Fakultät von', 5, ' ist ', 120, 120]
['Fakultät von', 6, ' ist ', 720, 720]
```

- Intellectual challenge: Drawing a fractal Koch curve

Outline of the CT programming course

3 A closer look at functions and types

- Can you explain…
  - The same variable x is used globally and locally, which value will be used?
  - How does recursion work?
- Concept: Stack of variable-value tables: A new table is created when a function is called, and removed upon return. Code in def is executed there



- More tricks with functions:
  - Default values `def(x,y=0): …`
  - Type declarations, e.g. Mainly as documentation

```
def fact(n:int) ->int:
    if n<=1: return 1
    return n*fact(n-1)
```

- Using %whos

```
[1]: n=5
     x=1.2
```

```
[2]: %whos
```

```
Variable   Type     Data/Info
-----------------------------
n          int      5
x          float    1.2
```

- Function names are just variables (of type function) referring to an address in computer memory where the definition code is stored

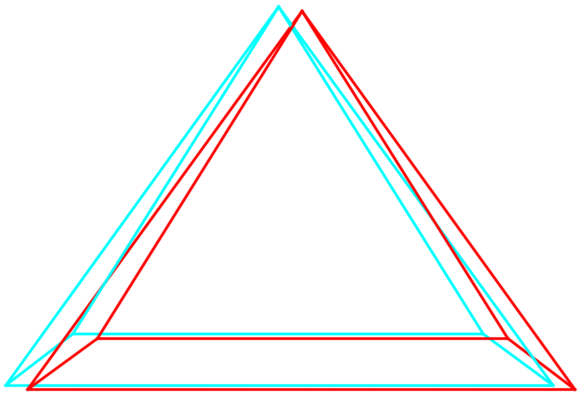
Outline of the CT programming course

4 The many uses of lists

- List literals like `L=[5,2,-9]` suggest what can be done with them (O., 2011)
  - `L[2], L[-1], L[i:j], len(L), sum(L), min(L), max(L), L+M`
- List comprehension `[f(x) for x in M if p(x)]` similar to set comprehension $\{f(x) | x \in M \wedge p(x)\}$
- Mini project using lists to implement rational numbers
  - And hint to `from fractions import Fraction`
- Statistics: Define function for mean, sd, cov, cor, binomial distribution
- Number theory, e.g.

```
def divisors(n): return [d for d in range(1,n+1) if n%d==0]
def prime(n): return len(divisors(n))==2
```

  - gcd, lcm, φ-function, explore numbers with 3,4,5 divisors, perfect numbers, challenge: RSA
- Exception handling, e.g. in vector addition: raise
- Mini project to implement vector operations
  - Application: 3d graphics and stereoscopic projections
  - Outlook: hint to numpy



Outline of the CT programming course

5 All the math down to the integers

- Rationals as lists of integers are easy
- But how are sqrt, exp,sin etc calculated by computers? (w.o. Taylor)

```python
def heron(a,x=1,n=7):
    if n<0: return x
    return heron(a, (x+a/x)/2, n-1)
```

```python
def Power(a,b,eps=1e-8):
    if a<0: raise Exception("a^b requires a>=0")
    if b<0: return 1/Power(a,-b)
    if abs(b)<eps: return 1
    if abs(b-1)<eps: return a
    if b>1: return a*Power(a,b-1)
    if b<1: return Power(heron(a),2*b)
```

```python
def Log(a, b, eps=1e-8, lo=0, hi=None): # Log by bisection
    if a<=0 or b<=0 or b==1:
        raise Exception("log_b(a) requires a>0, b>0, b≠1")
    if hi==None: hi=max(1.0, a)
    mid= (lo+hi)/2
    p= Power(b, mid, eps)
    if abs(p-a)<eps: return mid
    elif p<a: return Log(a, b, eps, mid, hi)
    else: return Log(a, b, eps, lo, mid)
```

```python
def Sin(x,eps=1e-3):
    if abs(x)<eps: return x
    s=Sin(x/3,eps)
    return 3*s-4*s**3
```

```python
def deriv(f,x,dx=1e-7):
    return (f(x+dx)-f(x))/dx

def integral(f,a,b,n=1000):
    dx=(b-a)/n
    s=0
    for i in range(n):
        s+=f(a+i*dx)*dx
    return s

def integralR(f,a,b,n=1000):
    if n==0: return 0
    dx=(b-a)/n
    return f(a)*dx+ integralR(f,a+dx,b,n-1)

def LN(x):
    return integral(lambda t: 1/t, 1,x,1000)
```

Note: Bisection also useful for roots and Bolzano-W.

Outline of the CT programming course

5 All the math down to the integers

- Recall: CT resembles modelling, both….
  - Modelling of real world problems
  - Modelling of mathematical structures (Model theory)
- The computer's model of math
  - In Python, `int`, `Fraction` are almost faithful models of $\mathbb{N}, \mathbb{Q}$
  - `float` is very coarse model of $\mathbb{R}$, in fact `float` $\subset \mathbb{Q}$, |`float`|$< \infty$
  - Demonstration

```python
x=0.2
for i in range(20):
    x=11*x-2
    print(x)
```

Outline of the CT programming course

6 More on functions

- Derivations and integral promote functions as first class objects

```
def deriv(f,x,dx=1e-7):
    return (f(x+dx)-f(x))/dx
```

- Input `f` is a function, not an expression!
- Unfortunately, full type declaration is cumbersome

```
from typing import Callable
def deriv(f: Callable[[float], float], x: float, dx: float = 1e-7) -> float:
    return (f(x+dx)-f(x))/dx
```

- Calling `deriv` with explicitly defined function or with lambda
  `deriv(lambda x: x**2, 1)`
- Defining functions that return functions:

```
def D(f: Callable[[float],float], dx=1e-7) -> Callable[[float],float]:
    def d(x:float)->float:
        return (f(x+dx)-f(x))/dx
    return d
```

```
In [61]: c=D(math.sin)

In [62]: c(0)
Out[62]: 0.9999999999999983
```

- Didactical idea: Pave the way of thinking in type theory and lambda calculus
- PS: Of course, all this should be done after showing show to graph a function with matplotlib

Outlin

7 Optimizati

- Optimi-
  zation
  in one
  variable
  is simple:

```python
def min1(f,x0):
    delta=0.1 # step size
    while True: # repeat
        if f(x0+delta)<f(x0):# move to the right
            x0+=delta; continue # next step
        if f(x0-delta)<f(x0): # move to the left
            x0-=delta; continue
        if delta<0.000001: break # limit step size
        delta=delta/2 # neither left nor right improved, hence smaller step
    return x0
```

```python
min1(lambda x: (x-4)**2,0.1)
```

- And in $\mathbb{R}^n$, too

```python
minN(lambda x: (x[0]-2)**2+(x[1]-3)**2,[5,5])
```

```python
def minN(f,x0):
    delta=0.1
    while True:
        improved=False # success?
        for i in range(len(x0)):
            x1=x0.copy(); x1[i]=x0[i]+delta
            if f(x1)<f(x0):
                x0=x1; improved=True; continue
            x1=x0.copy(); x1[i]=x0[i]-delta
            if f(x1)<f(x0):
                x0=x1; improved=True; continue
        if improved: continue
        if delta<0.000001: break
        delta=delta/2
    return x0
```

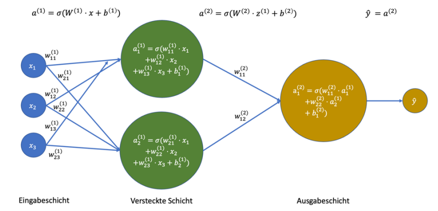
Many uses:

- (lin/nonlin) Regression (up to ANOVA and SEM)
- Physical system (e.g. bending beams)
- Geometric configurations
- …

Outline of the CT programming course

8 Neural networks are just optimization

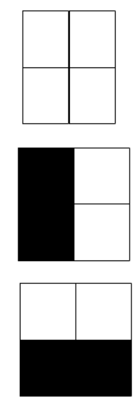
- (Self-)Supervised learning: Training data $(x_i, y_i) \in \mathbb{R}^n \times \mathbb{R}^m$, learning is $\min_\theta \arg \sum_{i=1}^{n} \|y_i - f_\theta(x_i)\|^2$ where f_θ is the net-function
- Deep net = composition of layer functions, e.g. $x \mapsto \sigma(Wx + b)$



- Mini NN to recognize vertical or horizontal structures in 2x2 gray scale; black=1, white=0
- Position mapping: →

1	2
3	4

[1,0,1,0]
[0,0,1,1]



- Invent some training data, e.g. $\big((1{,}0{,}1{,}0), (1{,}0)\big)$, …
- Use a two layer net $f_\theta(x) = \sigma(W_2 \sigma(W_1 x + b_1) + b_2): \mathbb{R}^4 \to \mathbb{R}^2$, with $\theta = (W_1, W_2, b_1, b_2)$, minimize loss function on training data set
- Larger nets need more advanced optimization (iterative via back-propagation), but central concepts can be understood at this level (Schönbrodt & O., 2024)

Outline of the CT programming course

9 Computer algebra: Modelling symbolic part of math

- Expressions follow a recursive context-free grammar. Encoding as lists (binary)
  - $x+5$ as `['+','x',5]`
  - $x^2 - \sin\left(\frac{x}{2}\right)$ as `['-', ['^','x',2], ['sin', ['/','x',2]]]`
- Conversion functions (parser, pretty-printer) given to students as black-box
- They defined helper functions `isSum, add, freeOf` etc.
- One task was: Define a function to calculate derivatives. Possible solution

```
def derivative(term,var):
    if term==var: return 1
    if freeOf(term,var): return 0
    if isSum(term): return add(derivative(term[1],var),derivative(term[2],var))
    if isDifference(term): return subtrahiere(derivative(term[1],var),derivative(term[2],var))
    if isProduct(term):
        u=term[1]; v=term[2]; us=derivative(u,var); vs=derivative(v,var)
        return add(mult(us,v),mult(u,vs))
    if isQuotient(term):
        u=term[1]; v=term[2]; us=derivative(u,var); vs=derivative(v,var)
        return divide(subtrahiere(mult(us,v),mult(u,vs)),  power(v,2))
    if istPotenz(term):
        a=term[1]; b=term[2]
        if frei(b,var): return mult(mult(b,power(a,subtrahiere(b,1))),derivative(a,var))
        return derivative(["exp",["*",b,["ln",a]]],var)
    if isFunc(term) and term[0]=="exp": return mult(term,derivative(term[1],var))
    if isFunc(term) and term[0]=="ln": return divide(derivative(term[1],var),term[1])
    if isFunc(term) and term[0]=="sin": return mult(["cos",term[1]],derivative(term[1],var))
    return ["derivative",term,var]
```

Outline of the CT programming course

10 Intellectual challenges

- It's easy to write a program that analyses the definition of a function and gives a list of all variables used: Nice for debugging
  - Is it also possible to have a function halts that takes the definition of a function and gives out, if the function halts or goes into a loop?
  - Obviously not:

```
def problem():
    if halts(problem):
        while True: pass
    else: return 0
```

- Lambdas can be used to define…
  - natural numbers numbers (church numerals)
  - pairs, lists, sets,…. all of math (Set theory is not the only possible foundation of math)
  - Computable real numbers (challenging the often heard false statement that computers can only deal with rational approximations of irrational numbers such as $\sqrt{n}$)
- …. But I have to admit that this usually doesn't happen because of the semester's end

Looking back

Does it work and how can math benefit from this? – A personal view

- Empowering can be experienced
- Distinction between program time and runtime explains semantics of algebra
- Referential transparency: Variables refer to exactly one object at each instance of time
- Variable-Value bindings are the same as assignments in mathematical logic (model theory)
- Formalization can be experiences as pathway to precision and clarity
- Debugging provides the bridge to STEM: Identify potential causal relations and experiment by systematic variation and controlling (fixing) other parameters
- Urge to give precise definitions (type declarations, meaning…), distinguish functions and expressions
- Intellectual flexibility (e.g. functions as objects) and challenges
- BUT: No empirical evidence collected: What practically possible empirical study should be more convincing than experience?

Looking back

CT in this course

- Recall: Selby & Woollard (2013) define CT as cognitive though process:
  - 1. the ability to think in **abstraction**s,
  - 2. the ability to think in terms of **decomposition**,
  - 3. the ability to think algorithmically,
  - 4. the ability to think in terms of **evaluation**s, and
  - 5. the ability to think in generalizations.
- Claim: All of this can be found in this course!
- E.g. its already contained in finding rectangle with maximal area (Lehmann, 2025)
- Note that the course avoided advanced topics
  - data structures (queues, trees, dictionaries,…)
  - large scale problems (hence no OOP, no modules)
- Nevertheless, some content goes beyond the math relevant for PST:
  - Using functions as parameters and results of functions → functional analysis
  - Optimization → Statistics and data mining
  - Lambda and types → Proof theory
- Fully exploiting all the connections to proving needs more time and simplification

Looking forward

AI? Why still programming, CAS etc?

- AI is very useful for coding, spotting errors
  - Encourage students to use it for technical problems e.g., with matplotlib
- Why program at all?
  - AI can generate and execute programs - but judging if they are correct and solve the right problem remains essential
    - Requires ability to read code!
  - A program has a clearly defined semantics. Formalization gives reliable results (in contrast to AI)
- Programming contrasts sloppiness of LLMs!
  - E.g. CAS algorithms can be debugged, LLMs can't!
- All in all: Understand basics of AI, understand what is can do and what it can't. I.e., math supports humans in staying ahead and mastering AI
  - … instead of being a slave of AI
- Only humans can set normative rules, moral values and epistemic values. Only human can decide what an interesting question is and what a convincing argument is.
  - AI urges us to teach more logic, not less!