Developing a Framework for Programming Physical Computing Systems

Steven Abreu

Bernoulli Institute & CogniGron, University of Groningen, Netherlands

Summary. The goal of physical computing is to harness a physical system's potential for computation. Currently, physical computing researchers draw from a very limited set of programming methods - often, the system is treated as a blackbox dynamical system where only few of its many parameters are optimized. To be competitive, physical computing must use modern methods from computer science, machine learning and dynamical systems. A framework for programming physical systems can clarify important concepts, point to promising research directions, make existing programming methods more widely applicable, and contribute to better collaboration between scientists working on physical, computational, and cognitive levels. This contribution aims to foster discussion and interdisciplinary awareness by clarifying problems in physical computer programming.

Programming is more than just "writing code" - it also encompasses deep learning (*differentiable programming*), probabilistic modeling (*probabilistic programming*), and various optimization procedures (*program synthesis*). Programming can be defined as "*a general activity of changing a computing system's functionality*" [6]. The computing system consists of a physical system Ψ and a matching computational model λ, see Figure 1a. The system's functionality is its input-output behavior. Programming begins with an intention - a mental conception of the computing system's target behavior Ω[1]. The programmer's goal is then to come up with a program which solves the task Ω, is expressible within the computational model λ, and implementable in the physical system Ψ, see Figure 1b.

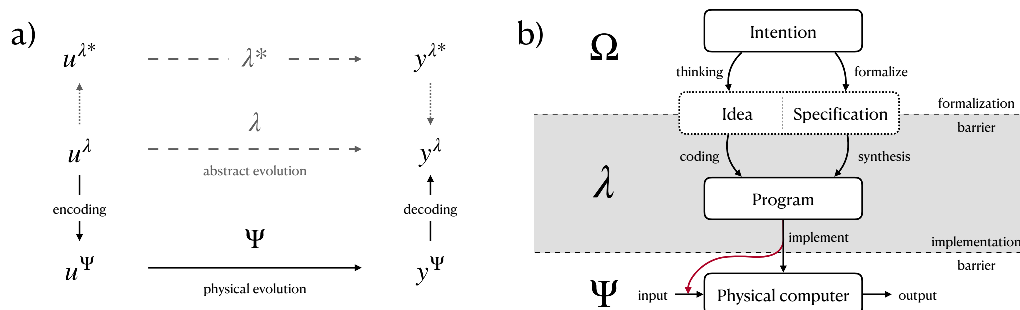

Figure 1: **a)** A computation from input u^λ to output y^λ yields the same results going through the abstract model λ as through the physical system Ψ, adapted from Horsman *et al.* [3]. See text for explanation of λ^*. **b)** Process of conventional computer programming, extended from Grünert [2], see text for explanation.

However, programs are often too complex to be understood by programmers. One then describes the program in a different computational model λ^*, which must match the original model λ as shown in Figure 1a. This λ^* can be a higher level of abstraction (*e.g.* Python instead of assembler), or an entirely different level of description (*e.g.* a dataset and learning algorithm that yields the desired program).

Conventional programming assumes that programs can be represented in some formal language, and that the computer offers an interface through which the program can be transferred.This is not the case for physical computers - which may not be fully captured by the computational model due to device mismatch or unobservability. This breaks the "implementation" arrow in Figure 1b, making conventional programming impossible. One may instead *optimize* or *learn* accessible parameters. In adaptive, goal-directed systems without access to the underlying program (*e.g.* in biology), we resort to methods of *"persuasion"* [7] from cybernetics or reinforcement learning.

We arrive at a picture that looks markedly different from conventional programming. To clarify this picture, the concept of a *programming interface* is introduced. A computer generally offers interfaces for input and output, but typically also offers programming interfaces to modify its behavior. The input and programming interfaces may map to the same physical interface, yet they are semantically different. The input to a computer $u(t)$ is thus a superposition of data $u_{data}(t)$ and a program $u_{prog}(t)$, *e.g.* a sequence of

[1] We assume here that the task Ω is computational and abstract - fabrication is different from computation.

instructions (shown by the red arrow in Figure 1b). This is analogous to how universal Turing machines expect both input data and a program to be executed.

The programming interfaces of a computer determine its programmability, *i.e.* how the computer's program can be changed. Three basic programming interfaces may be supported by any computer, allowing a programmer to: 1) (re)set the state of the computer, 2) change parameters of the program, 3) change the program itself. If supported, each of the three interfaces may be total or partial. In conventional programming, all three interfaces are totally supported and thus give the programmer perfect control. In physical computing, at least one of the interfaces is typically only partially supported or fully unsupported. For example, it is often possible to set values on the computer device, but the device mismatch makes this setting only approximate, thus *partial*.

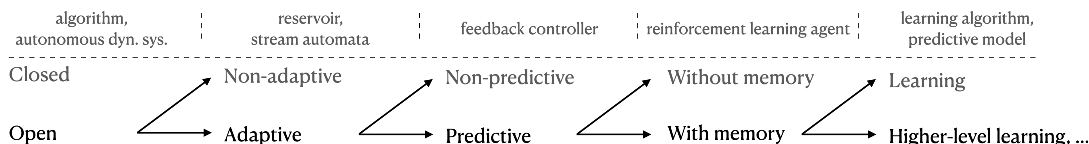

Figure 2: Preliminary computer program taxonomy (adapted from Ref. [5]), see text for explanation.

Programming interfaces closely relate to what *kind of programs* can be implemented. Figure 2 shows a computer program taxonomy, adapted from Rosenblueth *et al.* [5]. Closed systems are initialized by the user and then run autonomously, like Turing machines or autonomous dynamical systems. Programming means to (re)set the state and let the system evolve. Open systems receive input and interact with their environment while they evolve, like reservoir computers or stream automata. Adaptive systems target a setpoint - a well-defined target state that they aim to reach through internal feedback. They can be programmed by changing their target setpoint. Predictive systems try to reach some goal through a policy of how to interact with the environment, which can be changed through reward and punishment (*i.e.* programming through operant conditioning). Learning systems have long-term memory and can thus learn rich predictive or generative models, which may be trained, *e.g.* through input-output examples.

Although programming is conventionally limited to *changing* an existing computer's program, *creating* a computation is also of interest. The goal is to *instill* a computation that solves a task Ω into some physical system. This requires a theory that connects computation with physics. Much can be said about the differences between programming and instilling, but progress on either will contribute to the other.

When programming, the model of computation λ is the lens through which we see and interact with the physical computer. We solve the cognitive task Ω using a computational model λ which is then physically implemented in Ψ. *Might it be possible to reverse this direction, and come up with new models of computation directly from physics?* A challenge is presented by the dominance of symbolic models of computation in any area of computing, even those not well-suited to symbols. The (seemingly) physics-first theory of quantum computing is historically rooted in symbolic computation and still heavily influenced by it. Computing *directly* with physics may require a new general theory of computing [4].

Although a general theory of computing is lacking, there is a rich diversity of computational models which programmers can draw from to program diverse physical computing systems [1, 2]. To improve our ability to program physical systems, we need to expand our repertoire of computational models and widen our conception of programming. New models of computation may be discovered by exploring the *computational universe* on (and between) all levels: the computational, the physical, and the cognitive.

References

[1] J. Banâtre, P. Fradet, J. Giavitto, O. Michel. *Unconventional Programming Paradigms.* Springer, 2005.

[2] G. Grünert. *Unconventional programming: non-programmable systems.* PhD thesis, 2017.

[3] C. Horsman, S. Stepney, R. C. Wagner, V. Kendon. *When does a physical system compute?* Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences, 470(2169):20140182, 2014.

[4] H. Jaeger. *Towards a generalized theory comprising digital, neuromorphic and unconventional computing.* Neuromorphic Computing and Engineering, 2021.

[5] A. Rosenblueth, N. Wiener, J. Bigelow. *Behavior, purpose & teleology.* Philosophy of Science, 10(1):18, 1943.

[6] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming.* Prentice-Hall, 2004.

[7] M. Levin. *Technological approach to mind everywhere: An experimentally-grounded framework for understanding diverse bodies and minds.* Frontiers in Systems Neuroscience, 16, 2022.