2013 IEEE Fifth International Conference on Technology for Education

Viewing algorithms as iterative systems and plotting their dynamic behaviour

Venkatesh Choppella[*], K. Viswanath[†] and P. Manjula[‡]
Software Engineering Research Centre
International Institute of Information Technology - Hyderabad, Hyderabad, India
Email:[*]venkatesh.choppella@iiit.ac.in;[†]viswanath.iiithyd@gmail.com; [‡]manjula.p@research.iiit.ac.in

***Abstract*—We revive an old but little explored idea about how to think about algorithms and problem solving. Algorithms are discrete dynamical systems, also called iterative systems. Pursuing this point of view pays rich dividends. Important concepts like state space, next-state function, termination, fixed points, invariants, traces etc., can be mapped from dynamical systems to elements of algorithm design. Many of these concepts can be visualised through plots that trace the dynamic behaviour of the algorithm.**

***Keywords*-Algorithms, Programming, Dynamics, Computer Science Education, Engineering Education**

I. Introduction

Students have a brush with algorithms informally in their first programming course. Then, they learn how to design and analyse them in their data structures and algorithms course. As is evident from the curricula recommendations by ACM and IEEE[1] and the All India Council for Technical Education (AICTE) [2], introductory programming courses focus on the use of algorithms and algorithmic patterns (recursion, iteration, etc.) while courses on algorithm and data structures present problem solving and time and space complexity. The formal notion of an algorithm and its precise connection with problem solving is rarely presented. Even well respected textbooks begin by describing algorithms informally as a "sequence of computational steps that transform input to output"[3], or as procedures that are "precise, unambiguous, mechanical, efficient and correct."[4]. This description is unsatisfactory to the beginning computer science student, because it does not give the student a clear, general structure on how to construct an algorithm to solve a problem and verify that the construction is sound.

This paper starts with the premise that algorithms are iterative systems, also called discrete dynamical systems. The subject of dynamics is age-old in physics. Dynamical systems form the basis of studying various natural and artificial phenomena (atmosphere, biology, economics, finance, etc.). The idea of dynamics is pervasive and it is natural to ask the question of how dynamical systems relate to computing. This question is an old one. As early as 1968, Knuth introduced a formal notion of a "computational method" in his celebrated Art of Computer Programming ([5], Section 2.2). Although he did not use the term discrete dynamical systems, his computational method is essentially identical to it. Surprisingly, the connection was not explored further in his book or elsewhere. Only much later, in 2008, discrete dynamical systems and the Mapcode method were suggested as a unifying approach to study algorithmic problem solving[6]. As far as we know, few professional or academic bodies suggest approaching the subject of algorithms and problem solving through dynamical systems in their curriculum recommendations[1].

II. Iterative systems

An iterative system is a set along with a next state function on it. Many systems in science and engineering are modeled as iterative systems. Iteration is a universal concept in nature. Evolution is an iterative process with the purpose of survival and adaption to environment. Nature's fractal geometry (clouds, coastlines, mountains, ferns) reveals itself best when understood as an iterative system. Continuous dynamical systems in science and engineering can be approximated by iterative systems. Students begin the study of simple dynamical systems in their high school and continue studying them through undergraduate courses in mathematics, physics and mechanics.

Iteration is also pervasive in programming and computer science. Iteration is a core concept in computing and is classified as one of the threshold concepts in Computer Science[7]. Students first see iteration through while loops in their programming class. Yet iteration does not come naturally to them[8]. Wing[9] has underscored that computational thinking — of which iteration is an important pillar — should involve conceptualising, not just coding and learning the syntax of a language. Students of functional programming are taught that iteration is a special form of recursion called tail recursion. Many important algorithms (searching, sorting, numerical) are specified via iteration. In computer engineering, compilers and architectures are optimised to make iteration run as fast as possible. Engineering students write converging iterative loops to simulate and solve differential equations. How can algorithmic problem solving be viewed as constructing iterative systems and visualising their dynamic behaviour? In the rest of this paper we explore this question.

The idea of an iterative system is simple and easily accessible to someone with equipped with high school level

[1]Dynamical systems should not to be confused with dynamic programming, a popular topic in algorithms.

mathematical background in basic set theory and algebra. An iterative system is a pair $\langle X, F\rangle$ that consists of a *state space* X and a *next-state* function F over X. Every element x of X defines a *trace*, which is the infinite sequence $\langle x, F(x), F^2(x), \ldots\rangle$.

A *fixed point* a is an element of the state space for which $F(a) = a$, i.e., it is left unchanged by F. The subset $\mathit{fix}(F)$ of X denotes the set of all fixed points of F. An element x is said to *terminate* (at a) if there is a fixed point a of F such that $a = F^m(x)$ for some iteration m. The subset $\mathit{term}(F)$ of X denotes the set of elements of X that terminate. $F^\infty : \mathit{term}(F) \to \mathit{fix}(F)$ is called the *limit map* of F.

A property P is any predicate over the state space. P is *invariant* for an iterative system $\langle X, F\rangle$ if for each $x \in X$, P is true at x implies P is true at $F(x)$. The invariant plays a key role in the correctness of an algorithm designed for problem solving.

The key concepts of iterative systems have nice pictorial representations that can be easily plotted. Some simple pictures can be found in [6]. Evocative animations can make the concepts even more accessible. In this paper, we limit ourselves to presenting simple plots that, nonetheless, convey important information about the iterative system.

There are several examples one can use to introduce iterative systems to the student before showing how they help in formulating algorithms for problem solving. Simple examples from physics and mechanics can illustrate how iterative systems can *model* physical processes. We consider one such simple example: a ball moving along a horizontal plane at a constant velocity (Figure 1).

A systematic modelling of this simple phenomenon requires the student to first identify the main quantity of interest: the horizontal position of the ball with respect to a fixed origin. The next step would be to model the position as a non-negative real number. Thus the state space X is the set of non-negative real numbers. The next important decision in the model is to assume that the position of the ball is captured in a series of snapshots, rather than continuously. This clearly positions the problem of modelling the phenomenon as an iterative system. Since the ball is assumed to move at a constant velocity, the ball at position x would move one unit to the position $x+1$ at the next snapshot. This gives rise to the next state function as $F(x) = x + 1$. Assuming that the initial position of the ball is $x = 0$, the phenomenon of the rolling ball may be represented by the trace $\langle 0, 1, 2, \ldots\rangle$ that goes off to infinity. An important task in the model is to construct a plot, in this case a displacement plot as shown in Figure 1. The graph plots the position of the ball as a function of the *iteration* (zeroeth, first, second, or third snapshot, etc.). This simple example illustrates the notion of state space, the next state function and iteration.

A different scenario considers the problem of a ball rolling on a billiards table. At the end of the table is a pocket. A ball rolling on such a table eventually reaches the pocket,

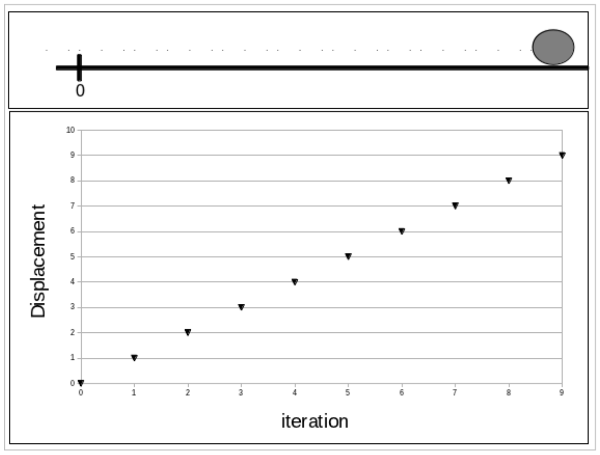

Figure 1. Displacement plot for a ball moving on an infinitely long plane.

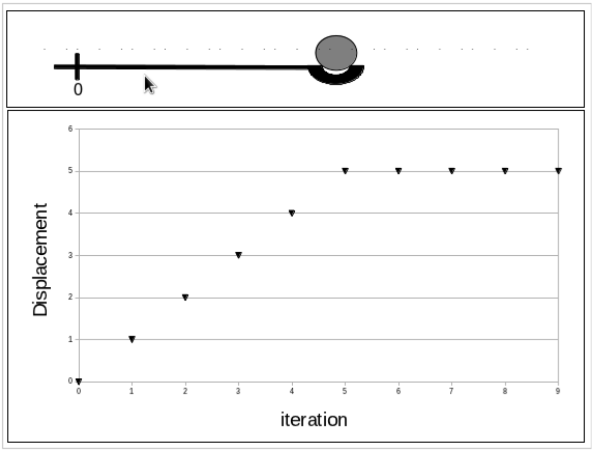

Figure 2. Displacement plot of a moving ball on a billiards table 5 units long.

falls into it and stays there. This example nicely illustrates the notion of a fixed point. Suppose the pocket is five units from the origin of the table. The state space X now is the closed interval $[0, 5]$ and the next state function F is

$$\begin{aligned} F(x) &= 5 \text{ if } 4 \le x \le 5 \\ &= x + 1 \text{ if } 0 \le x < 4 \end{aligned}$$

Figure 2 is a plot that traces the position of the ball at each iteration. When the ball enters the pocket, its position remains fixed, i.e., it reaches a fixed point. It is clear that $x = 5$ is the fixed point. An animation is useful to "see" the phenomenon, but the plot shows the position with each iteration. After the ball reaches the fixed point, the trace from there on is a horizontal line. This is a universal property of the trace fixed points and it is easy for the student to see this. Many of the key ideas of iterative systems used in modelling physical phenomena carry over to algorithm design, which we explore next.

III. Iterative systems and algorithm design

Beginning computer science students are taught algorithms as part of their problem solving course. The iterative systems approach leads to a systematic framework for designing an algorithm to solve a problem. This framework is called the *Mapcode* framework in [6]. The Mapcode approach consists of four parts : the specification of the problem as a function, the specification of the resources on the computational platform, the specification of the algorithm as an iterative system, and finally, verification that the iterative system terminates, and, upon termination, implements the function that represents the problem. We illustrate the iterative systems (Mapcode) approach by considering the problem of designing an algorithm to compute factorial.

Problem specification:

The problem specification consists of identifying an input space, an output space and a function from input to output.

1) Identify the *input space* A: For the factorial problem, the input space is the set N of natural numbers.
2) Identify the *output space* B: For factorial, this is N.
3) Identify the problem as a *specification map* f from A to B: This is simply the factorial function ! from N to N.

A systematic formulation of the problem specification is important not only during constructing algorithms, but also during programming. The input and output space are part of the signature of the program implementing the specification map. The signature of program is an important part of its documentation.

Platform resource constraints:

4) Identify *platform resource constraints* : Algorithms are realised on computational platforms that support only specific primitive data types and operators. We assume that our algorithmic implementation of factorial has recourse to natural numbers and the primitive operations of decrementing a positive natural number and multiplication of two natural numbers.

Algorithm design needs insight into the structure of the problem, ingenuity and creativity. The iterative system specification does not obviate this need, but guides the design exercise by identifying specific milestones in the design. In the factorial example, the insight comes from exploiting the observation that $n!$ can be broken down into two factors $n(n-1)\ldots j+1$ and $j!$. Thus a partial computation of $n!$ can be stored in a pair of natural numbers (j,a). The first, j, starts at n and goes down to 0. The second, a, keeps track of the partial answer so far $n(n-1)\ldots j+1$. The final answer is obtained when j becomes 0, and at that point, a is equal to $n!$. Notice that at each stage the product a times $j!$ is equal to $n!$.

The insight of how factorial may be computed can now be systematically presented as an iterative system.

Iterative system specification:

5) Identify the *state space* X: In this example, the state space is $N \times N$. An element of the state space is the tuple (j,a).
6) Construct the next state function $F : X \to X$: The state (j,a) moves to $(j-1, a*j)$ when $j > 0$. When j is 0, $F(0,a) = (0,a)$.
7) Construct the *input map init* : $A \to X$: The input map injects the input into a state space element. The iterative system starts from $(n,1)$, so *init* is the map $n \mapsto (n,1)$.
8) Construct the *output map answer* : $X \to B$: The output map projects the answer from a state space element. For the factorial example, this is the function $(j,a) \mapsto a$.

The iterative system suggests the following general algorithm:

```
function(a: A) returns B {
   var x:X  := init(a);
   while (x != F(x)) {x := F(x);};
   return answer(x);}
```

To obtain the algorithm for factorial, we substitute $N \times N$ for X, (i,j) for x and use the functions *init*, *answer* and F defined above.

Verification:

9) Verify *termination*: This step verifies that that for every $y \in A$, *init*(y) terminates, i.e., *init*(y) is a member of *term*(F). The proof proceeds by demonstrating a *measure function* on *term*(F) that is positive but decreases with each iteration, i.e., for all x in *term*(F), $M(x) > 0$ and $M(x) > M(F(x))$ unless x is a fixed point.
10) Verify *correctness*: This step shows that the iterative system indeed computes the correct answer upon termination. More precisely, in this step, one is required to show that the problem specification commutes with its algorithmic implementation: $f = answer \circ F^{\infty} \circ init$. In long hand, this means that for each input n, *init*(n) terminates at s such that *answer*(s) is $f(n)$.

In the factorial case, the measure function is simply $M(j,a) = j$. Correctness reduces to showing that for every n the trace starting from $(n,1)$ terminates at $(0,n!)$. Consider the predicate $P_n = \{(j,a) | a * j! = n!\}$ and the trace $\langle (n,1), F(n,1), F^2(n,1), \ldots, P^k(n,1), \ldots \rangle$. It is easy to see that P_n is an invariant. By induction on the iteration variable k, all elements of the trace of $(n,1)$ satisfy P_n. $F^{\infty}(n,1)$, the fixed point of the trace is $(0,a)$, therefore a is equal to $n!$.

IV. Iteration plots and phase plots

The iterative systems approach just described lays down the rules for designing an a correct algorithm to solve a problem. Additionally, much insight could be gained by

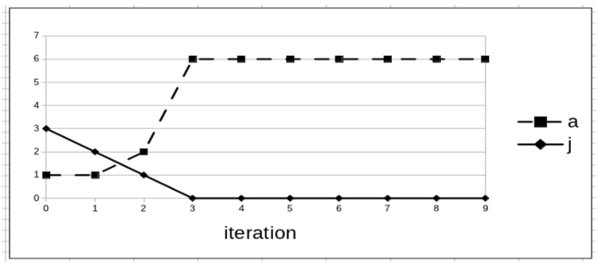

Figure 3. Plot showing the state variables j and a as functions of number of iterations in the iterative system solving factorial.

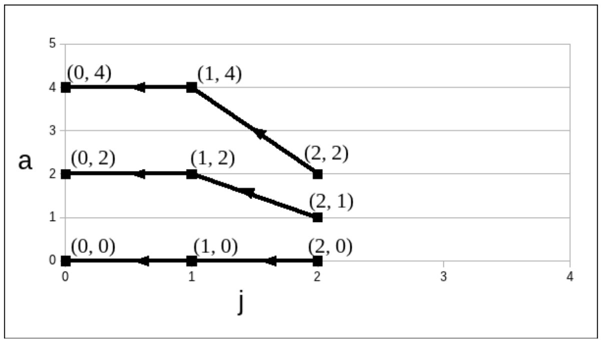

Figure 4. Phase Plot of the iterative system implementing factorial

simple experimentation and observation of how a system's state variables vary across iterations and also with respect to each other.

While computing students are encouraged to trace their programs, and often end up doing it any way as part of debugging code, one rarely sees them *plotting* values. Figure 3 plots the evolution of j and a against the iteration index. The plot of the factorial algorithm opens up many avenues for experimentation, for example, one can now measure and plot the number of iterations needed to reach the fixed point, and this gives an empirical understanding of the efficiency of the algorithm. The plot in Figure 3 also shows how j and a reach a fixed point and stay there, much like the plot of the motion of the moving ball in Figure 2. This illustrates the similarity between physical phenomena and computational phenomena. Such connections are invaluable to the student.

Scientists and engineers employ *phase plots*, which illustrate the trajectories in the state space. Such phase plots are useful for locating which are the fixed points in the state space, and also the sets of points that terminate at a given fixed point. Figure 4 shows the phase plot for the factorial iterative system. The state variable a is plotted against j. Again, such a phase plot is very useful to the student: the fixed points are seen to all align along the y-axis. Every point in the state space can be seen to terminate. Such visual understanding about the dynamics of the computation complements, in an important way, the formal reasoning that is part of algorithm design.

Sometimes it is more natural to express an algorithm using recursion. E.g., $0! = 1$ and $n! = n*(n-1)!$ can be directly translated into a recursive algorithm. The state space approach seems heavy-handed for such a simple example when an elegant recursive solution exists. But the execution of the recursive algorithm depends on a hidden data structure, the stack. A translation of the recursive version to an iterative system exposes the stack as a state variable whose dynamics (growing and shrinking) can then be observed explicitly.

V. Conclusion

Algorithm design and problem solving can profit from the theory of iterative systems. Further, much insight can be gained in the algorithm design process by plotting the state space's variables. In the coming years, with the advent of areas like cyber-physical systems, students of computer science will need to depend more on such systematic approaches because physical systems are invariably described by dynamical systems.

VI. Acknowledgements

We thank Jayashree Prasad and Santosh Adimoolam for their helpful comments about the paper.

References

[1] ACM and IEEE, "Computer Science Curriculum 2008: An interim revision of CS 2001. Report from the interim review task force," http://www.acm.org//education/curricula/ComputerScience2008.pdf, 2008, last visited 28 September 2013.

[2] AICTE, "Model curriculum for undergraduate programme in computer science and engineering," http://www.aicte-india.org/downloads/MODEL_SYLLABI_FOR_UG_Computer_Sci_Engg.pdf, 2000, last visited, 28 September 2013.

[3] C. E. Leiserson, T. H. Cormen, C. Stein, and R. Rivest, *Introduction to Algorithms*. MIT Press, 1990.

[4] Dasgupta, Papadimitrou, and Vazirani, *Algorithms*. McGraw Hill Higher Education, 2008.

[5] D. Knuth, *The Art of Computer Programming*. Addison-Wesley, New York, 1968.

[6] K. Viswanath, *An Introduction to Mathematical Computer Science*. The Universities Press, 2008.

[7] J. Boustedt, A. Eckerdal, R. McCartney, J. E. Mostrom, M. Ratcliffe, K. Sanders, and C. Zander, "Threshold concepts in computer science: Do they exist and are they useful?" *SIGCSE*, 2007.

[8] S. Grover, "Learning to Code Isn't Enough," https://www.edsurge.com/n/2013-05-28-opinion-learning-to-code-isn-t-enough, 2013, last visited 28 September 2013.

[9] J. M. Wing, "Computational thinking," *Communications of the ACM*, vol. 49, no. 3, March 2006.