International Journal of Bifurcation and Chaos, Vol. 18, No. 5 (2008) 1551–1559

EXPLOITING NONLINEAR DYNAMICS TO STORE AND PROCESS INFORMATION

ABRAHAM MILIOTIS*, SUDESHNA SINHA†,‡
and WILLIAM L. DITTO*,‡
*J. Crayton Pruitt Family Department of Biomedical Engineering,
University of Florida, Gainesville, FL 32611-6131, USA
†The Institute of Mathematical Sciences,
Taramani, Chennai 600 113, India
‡ChaoLogix, Inc., 101 SE 2nd Place, Suite 201-B,
Gainesville, FL 32601, USA

Received May 29, 2007; Revised June 25, 2007

By applying nonlinear dynamics to the dense storage of information, we demonstrate how a single nonlinear dynamical element can store M items, where M is variable and can be large. This provides the capability for naturally storing data in different bases or in different alphabets and can be used to implement multilevel logic. Further we show how this method of storing information can serve as a preprocessing tool for (exact or inexact) pattern matching searches. Since our scheme involves just a single procedural step, it is naturally set up for parallel implementation and can be realized with hardware currently employed for chaos-based computing architectures.

Keywords: Chaos computing; data storage; information processing; encoding; search; threshold control.

1. Introduction

Information encoding, storage, and retrieval are fundamental functions of computing devices. Today most commonly used devices for storing and processing information are based on the binary encoding of information, i.e. upon bits. Larger chunks of information are encoded by combining consecutive bits into bytes and words. In this work we develop a different approach for information encoding and storage based on the wide variety of patterns that can be extracted from nonlinear dynamical systems.

Recent research has demonstrated that the richness of the patterns embedded in nonlinear dynamical systems can be utilized to perform computations [Munakata *et al.*, 2002; Sinha & Ditto, 1998, 1999; Prusha & Lindner, 1999; Sinha *et al.*, 2002a, 2002b; Murali *et al.*, 2003a, 2003b]. Here we demonstrate the use of arrays of nonlinear dynamical systems (or *elements*) to stably encode and store information (such as patterns and strings). Furthermore we will demonstrate how this storage method also enables the efficient and rapid search for specified items of information in the data store. The nonlinear dynamics of the array elements provides flexible-capacity storage, as well as a means to preprocess data for exact and inexact pattern matching. In particular, we choose chaotic systems to store and process data through the natural evolution of their dynamics. The abundance of distinct dynamical behaviors (i.e. the fact that a chaotic system incorporates an infinite number of unstable patterns) gives it the ability to represent a large set of items. One can process data stored in such systems by controlling the system to restrict it to a single

one of those fixed points. In the following we give specific details of the scheme and then demonstrate it with examples.

2. Encoding and Storing Information

Consider a list of N data elements (labeled as $j = 1, 2, \ldots, N$), where each element is comprised of one of M distinct items. N can be arbitrarily large and M is determined by the kind of data being stored. For instance when storing English text one can consider the letters of the alphabet to be the naturally distinct items with $M = 26$. For the case of data stored in decimal representation $M = 10$, and for work in bioinformatics (manipulating the symbols A, T, C, and G) one has $M = 4$. One can also consider strings and patterns as the items. For instance for manipulating English text one might use a large set of keywords as the basis, necessitating very large M.

We store this list of N elements by N dynamically evolving chaotic elements. The state of the elements at discrete time n is given by $X_j^m[n]$, where j ($j = 1, 2, \ldots, N$) indexes each element of our list and m ($m = 1, 2, \ldots, M$) indexes an item in our "alphabet" (namely one of the M distinct items). To reliably store information one must confine each dynamical system to a fixed point behavior, i.e. a state that is stable and constant throughout the dynamical evolution of the system over time n.

To flexibly control the dynamical elements onto a large set of period 1 fixed points, we employ a threshold mechanism. This is given by the following simple strategy: whenever the value of a state variable of a dynamical system X exceeds a threshold value T (i.e. when $X > T$), X is reset to T, or clipped down to T. The threshold mechanism allows us to exploit the richness of chaos in a direct and efficient way, by creating new stable points of different periodicities from the chaotic dynamics. It achieves this by changing the shape and structure of the chaotic attractor by implementing a "wall" or a "limiter" in phase space. The position of the limiter depends on the threshold value, and can be varied. Different threshold values allow the chaotic time sequence to be "clipped" to different controlled sequences, with different periodicities. So one can create a wide range of stable regular behaviors from the chaotic dynamics (and not merely stabilize the natural unstable periodic points of the system) [Sinha, 1994; Glass & Zeng, 1994; Sinha, 2001; Wagner & Stoop, 2001].

Typically a large continuous window of threshold values ($T_{\min} < T < T_{\max}$) can be found where the system is confined to period 1 fixed points, namely, the state of the chaotic element under thresholding is stable at T (i.e. $X[n] = T$, for all times n) if T is in $[T_{\min}, T_{\max}]$. Thus the system is capable of yielding a continuous range of fixed points $X[n] = T$, as one varies the threshold over the interval. Note again, that this is quite unlike what can be achieved by stabilizing the (typically) discrete set of unstable period 1 points of a nonlinear system.

Now for encoding we will use the range of threshold values that yield period 1 fixed points. Specifically, we can take a large set of thresholds $\{T^1, T^2, \ldots, T^M\}$ from the fixed point range, setting up a one-to-one correspondence of these M thresholds with the M distinct items of our data. This allows each item m to be uniquely encoded by a specific threshold T^m ($m = 1, 2, \ldots M$). That is, if element j holds item m in the database, the threshold value of element j is set to T^m. So the number of distinct items that can be stored in a single dynamical element is typically large, as the size of M is limited only by the precision and resolution of the threshold setting and the noise characteristics of the physical system being employed.

So, denoting the threshold of element j by T_j^m we have the following: if element j of the system, $X_j^m[n]$, exceeds its prescribed threshold T_j^m (i.e. when $X_j^m[n] > T_j^m$) the variable $X_j^m[n]$ is reset to T_j^m. Since the thresholds lie in the range yielding fixed points, this will enable the element to hold its state at value $X_j^m[n] = T_j^m$ for all times n.

In particular, consider a collection of storage elements that evolve in discrete time n according to the tent map,

$$f(X_j^m[n]) = 2\min(X_j^m[n], 1 - X_j^m[n]) \qquad (1)$$

with each dynamical element storing one element of the given list of items ($j = 1, \ldots N$). Each element can hold any one of the M distinct items indicated by the index m ($m = 1, 2, \ldots, M$). As described above, a threshold will be applied to each dynamical element to confine it to the fixed point corresponding to the item to be stored. For the tent map, thresholds in the range from 0 to 2/3 yield fixed points, namely $X_j^m[n] = T_j^m$, for all time, when threshold $0 < T_j^m < 2/3$.

See Fig. 1 for a schematic of the tent map under the threshold mechanism, which is effectively described by a "be-headed map". It is clear from

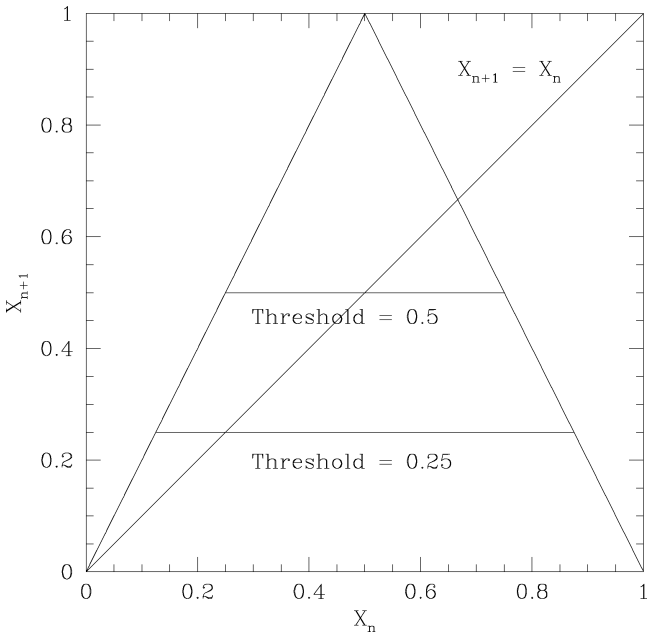


Fig. 1. The Tent Map under the threshold mechanism effectively yields a map with a "plateau" at the threshold value. Here threshold values 0.25 and 0.5 are explicitly shown. The diagonal line is the $X_{n+1} = X_n$ line.

Fig. 1 that in the range 0 to 2/3 the value of $X[n+1]$ lies above $X[n]$ implying that the system with state $X[n]$ at threshold T will be mapped to a state higher than T in the subsequent iterate and thus will be clipped back to T. Another way of graphically rationalizing this is to note that fixed point solutions are obtained where the $X[n+1] = X[n]$ line intersects the "beheaded" tent map. The value of X at the intersection yields the value of the fixed point, and the slope at the intersection naturally gives the stability of the fixed point. It is clear from Fig. 1 that in the range 0 to 2/3 this intersection is on the "plateau", namely the fixed point solution is equal to the threshold value. Further the solution of the fixed point for this map is superstable as the slope is exactly zero on the "plateau". This makes the thresholded state very robust and quite insensitive to noise.

In our encoding, the thresholds are chosen from the interval $(0, 1/2)$, namely a subset of the fixed point window $(0, 2/3)$. For specific illustration, without loss of generality consider each item to be represented by an integer m, in the range $[1, M]$. Defining a resolution r between each integer as:

$$r = \frac{1}{2} \cdot \frac{1}{(M+1)} \tag{2}$$

gives a lookup table mapping the encoded number to the threshold, relating the integers m in the range $[1, M]$ to the thresholds $T^m[j]$ in the range $[r, 1/2 - r]$ by:

$$T_j^m = m \cdot r \tag{3}$$

Therefore we obtain a direct correspondence between the set of integers 1 to M, where each integer represents an item, and a set of M threshold values. This correspondence or *representation* is important for the process of encoding information in an M-level representation and, as we shall see below, it is primarily important for the process of searching the list for certain bits of information. So we can store the N list elements by setting appropriate thresholds [via Eq. (3)] on N dynamical elements. As mentioned before, the thresholded states encoding different items are very robust to noise since they are superstable fixed points.

3. Searching for Information

Once we have a given list stored by setting appropriate thresholds on N dynamical elements, we can query for the existence of a specific item in the list. Here we show how the manner in which the information is encoded helps us preprocess the data such that the effort required in the pattern matching searches is reduced. Specifically we will demonstrate how we can use *one global* operational step to map the state of elements with the matching item to an unique maximal state that can be easily detected. Note that such an operation enables us to detect matches to strings/patterns (of length equivalent to $\log_2 M$ binary bits) in one step. It would take typically $\log_2 M$ steps to do the same for the case of binary encoded data.

When searching for a specific item in the list, one globally shifts the state of all elements of the list up by the amount that represents the queried item. Specifically the state $X_j^m[n]$ of all the elements $(j = 1, \ldots, N)$ is raised to $X_n^m[j] + Q^k$, where Q^k is a search key given by:

$$Q^k = \frac{1}{2} - T^k \tag{4}$$

where k is the number being searched for. This addition shifts the interval that the list elements can span, from $[r,\ 1/2 - r]$ to $[r + Q^k,\ 1/2 - r + Q^k]$, where Q^k is the globally applied shift. See Fig. 2 for a schematic of this process.

Notice that what we are searching for is the representation of the item, not the item itself. For example, we can encode each letter of the alphabet

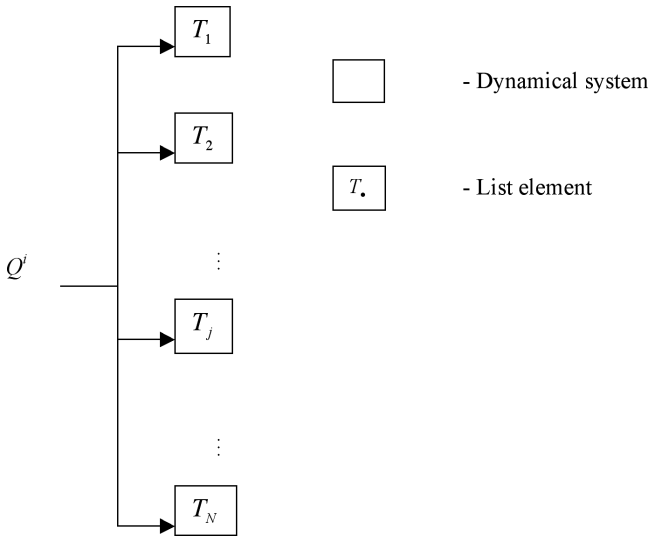


Fig. 2. Schematic of the list held in an array of nonlinear dynamical system elements and the parallelized search operation.

by a number, such that the lowest threshold $T^1[j]$ represents the letter A, the next highest $T^2[j]$ represents B, etc. When we search for A, we are really searching for the state with threshold $T^1[j]$.

Notice that the information or item being searched for is encoded in a manner "complimentary" to the encoding of the items in the list (much like a key that fits a particular lock); i.e. $Q^k + T^k$ adds up to 1/2. This guarantees that *only* the element matching the item being searched for will have its state shifted to 1/2. The value of 1/2 is special in that it is the *only state value* that on the subsequent update will reach the value of 1.0, which is the maximum state value for this system. So only the elements holding an item matching the queried item will reach the extremal value 1.0 on the dynamical update following a search query. Note that the important feature here is the nonlinear dynamics mapping the state 1/2 to 1, while all other states (both higher and lower than 1/2) get mapped to values *lower than* 1. See Fig. 3 for a schematic of this process.

The salient characteristic of the point 1/2 is the fact that it is the unique critical point, and so it acts as "pivot" point for the nonlinear dynamical folding that will occur on the interval $[r+Q^k, 1/2-r+Q^k]$ during the next update. This provides us with a single global monitoring operation to push the state of all the elements matching the queried item to the unique maximal point in parallel.

The crucial ingredient is the use of the existing critical point in the dynamical mapping to implement selection. Chaos is not strictly necessary here. It is evident that for unimodal maps higher nonlinearities allow larger operational ranges for the search operation and also enhance the resolution of the encoding. For the tent map specifically, it can be shown that the minimal nonlinearity necessary for the above search operation to work is operation in the chaotic region. Another specific feature of the tent map is that its piecewise linearity allows the encoding and search operation to be very simple indeed.

Of course to complete the search we must now detect the maximal state located at 1. This can be accomplished in a variety of ways. For example, one can simply employ a level detector to register all elements at the maximal state. This will directly give the total number of matches, if any. So the total search process is rendered simpler as the state with the matching pattern is selected out and mapped to the maximal value, allowing easy detection.

Further, by relaxing the detection level by a prescribed "tolerance", we can check for the existence within our list of numbers or patterns that are *close to* the number or pattern being searched for. In this case "close to" means "having a representation that is close to the representation of the item for which we are searching. Using the earlier example of English letters of the alphabet encoded using the lowest threshold $T^1[j]$ for A, the next higher threshold for B, etc., relaxing the detection threshold a bit allows us to find mistyped words where L or N were substituted for M or where X or Z were substituted for Y. However, if we had chosen our representation such that the ordering put T and U before and after Y (as is the case on a standard QWERTY keyboard), then our relaxed search would find spellings of *bot* or *bou* when *boy* was intended. Thus "nearness" is defined by the choice of the representation and can be chosen advantageously depending on the intended use. Figure 6 gives an illustrative example of detecting such inexact matches.

So nonlinear dynamics works as a powerful "preprocessing" tool, reducing the determination of matching patterns to the detection of maximal states, an operation that can conceivably be accomplished by simple addition and in parallel. For instance, content-addressable memory (CAM) is a special type of computer memory used in certain very high speed searching applications, such as routers. Unlike standard computer memory (random access memory or RAM) in which the user supplies a memory address and the RAM returns

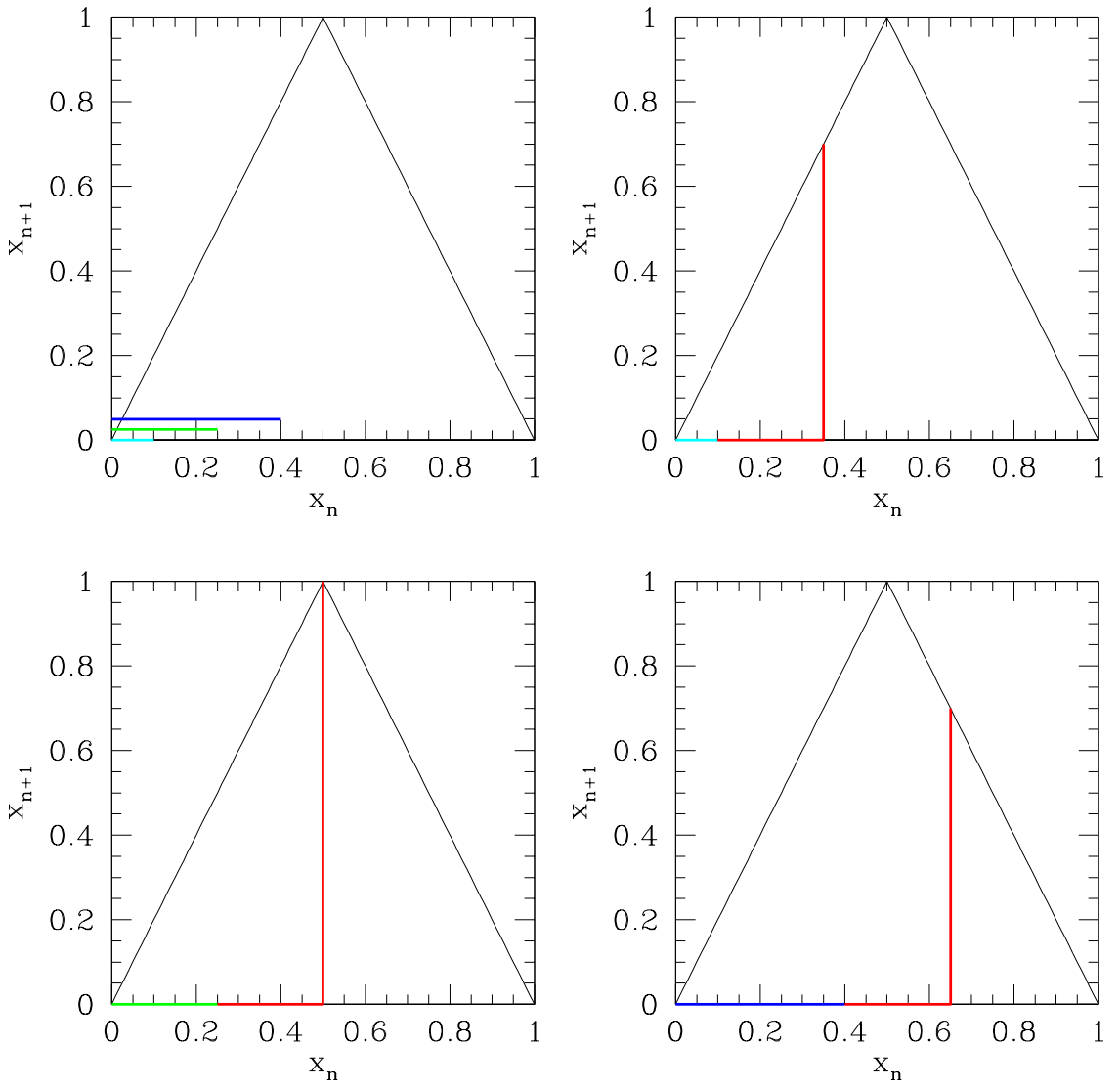
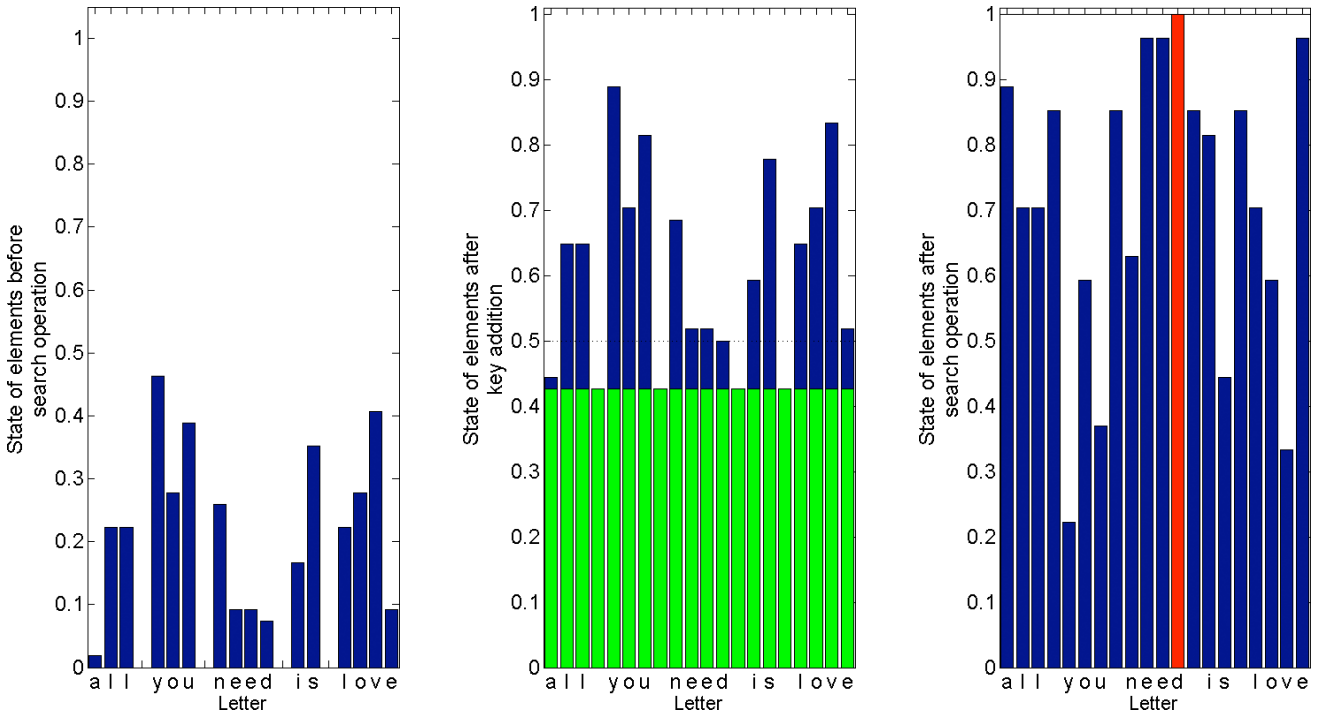


Fig. 3. Schematic representation of the state of an element: (i) matching a queried item, (ii) higher than the queried item, (iii) lower than the queried item. The top left panel shows the state of the system encoding a list element. Three distinct elements are depicted. The state of the first element is held at 0.1 (cyan); the second element is held at 0.25 (green) and the third element is held at 0.4 (blue). These are shown as lines of proportional lengths on the x-axis. The top right, bottom right and bottom left panels show each of these elements with the search key added to their states. Here the queried for item is encoded by 0.25. So $Q^k = 1/2 - 0.25 = 0.25$. This amount is shown in red. After the addition of the search key, the subsequent dynamical update yields the maximal state 1 only for the element holding 0.25 (green). The ones with states higher and lower than the matching state (namely 0.1 and 0.4, shown in cyan and blue) are mapped to lower values.

the data word stored at that address, a CAM is designed such that the user supplies a data word and the CAM searches its entire memory to see if that data word is stored anywhere in it [Krikelis, 1997]. What we attempt to design here is a CAM-like device.

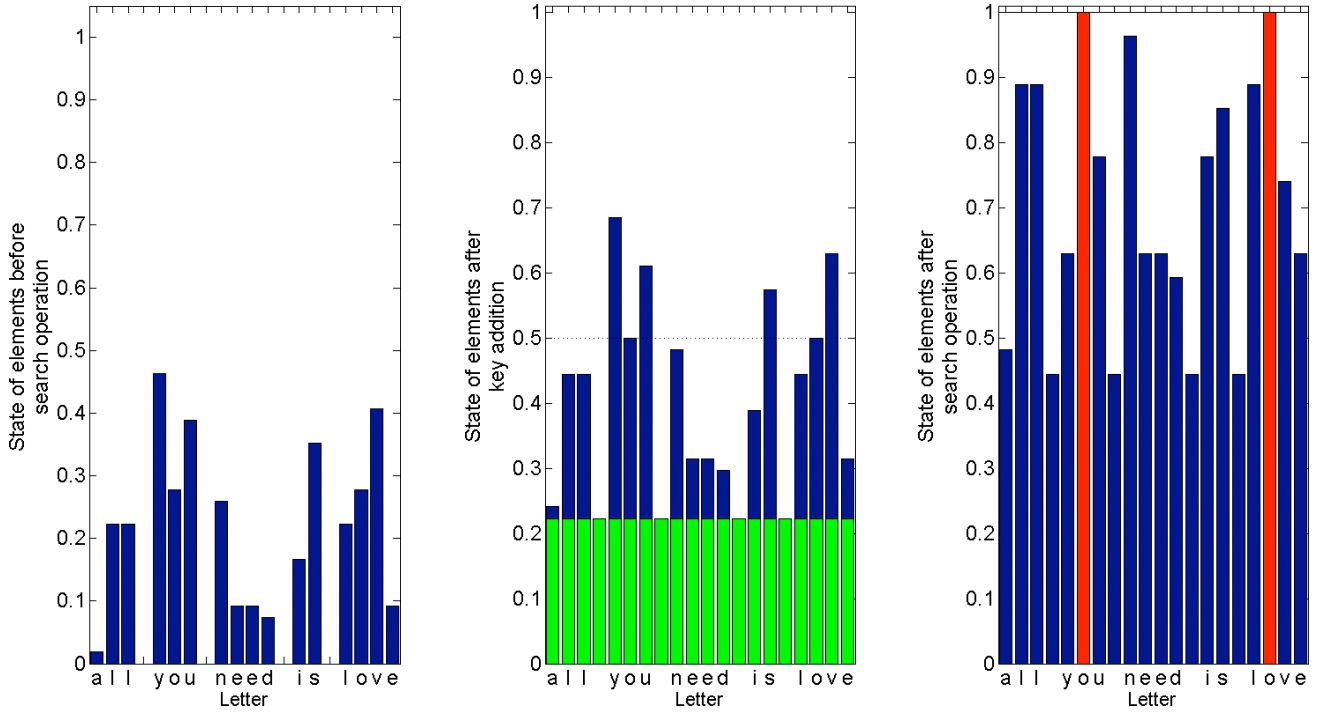
4. Encoding, Storing and Searching: An Example

Consider the case where our data is English language text, encoded as described above by an array of tent maps. In this case the distinct items are the letters of the English alphabet. As a result $M = 26$ and we obtain $r = 0.0185185\ldots$ from Eq. (2), and the appropriate threshold level for each item is obtained from Eq. (3). More specifically, consider as our list the letters in the title of the Beatles song "all you need is love". Each letter in this phrase is an element of the list with a value selected from our 26 possible values and can be encoded using the appropriate threshold, as in Fig. 4(a).



(a) (b) (c)

Fig. 4. (a) Threshold levels encoding the sentence "all you need is love", (b) the search key value for letter "d" is added to all elements, (c) the elements update to the next time step. For clarity, we marked by red any elements that reach the detection level.



(a) (b) (c)

Fig. 5. (a) Threshold levels encoding the sentence "all you need is love", (b) the search key value for letter "o" is added to all elements, (c) the elements update to the next time step. For clarity, we marked by red any elements that reach the detection level.

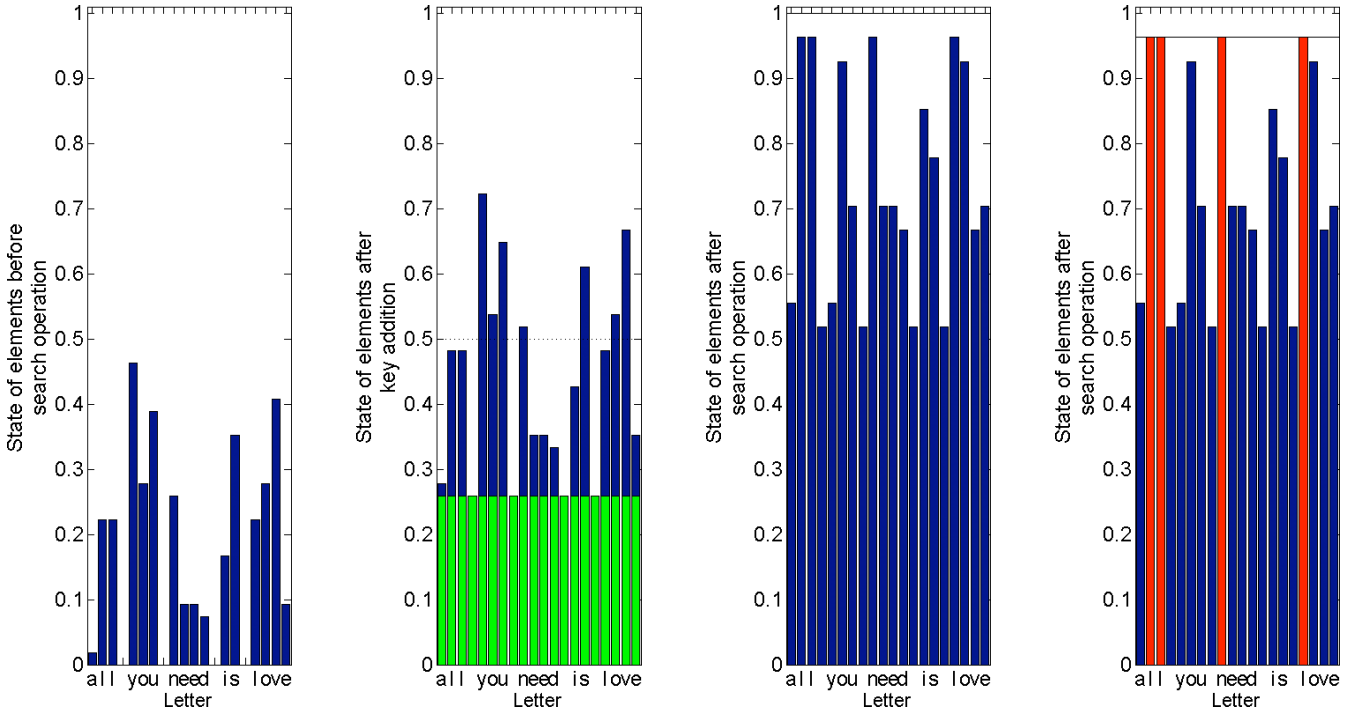


Fig. 6. (a) Threshold levels encoding the sentence "all you need is love", (b) the search key value for letter "m" is added to all elements, (c) the elements update to the next time step. It is clear that no elements reach the detection level at 1. (d) By lowering the detection level we can detect whether items "adjacent" to "m" are present.

Now the list, as encoded above, can be searched for specific items. Figure 4 presents the example of searching for the letter "d". To do so the search key value corresponding to letter "d" [Eq. 4] is added globally to the state of all elements. Then through their natural evolution, at the next time step the state of the element(s) containing the letter "d" is maximized. In Fig. 5 we performed an analogous query for the letter "o", which is present twice in our list, to show that multiple occurrences of the same item can be detected. Finally in Fig. 6 we search for an item that is not part of our given list, the letter "m". As expected Fig. 6(c) shows that none of the elements are maximized. By lowering the detection level to the value $1 - (2 \cdot r)$, we have detected whether adjacent items to the queried one are present. Specifically we have detected that the letters "l" and "n" are contained in the list. This demonstrates that inexact matches can also be found by this scheme.

5. Discussion

A significant feature of this search scheme is that it employs a single simple global shift operation and does not entail accessing each item separately at any stage. It also uses a nonlinear folding to select out the matched item, and this nonlinear operation is the result of the natural dynamical evolution of the elements. So the search effort is considerably simplified because it uses the native responses of the nonlinear dynamical elements. One can then think of this as a natural application, at the machine level, in a computing machine consisting of chaotic modules [Munakata *et al.*, 2002; Sinha & Ditto, 1998, 1999; Prusha & Lindner, 1999; Sinha *et al.*, 2002a, 2002b; Murali *et al.*, 2003a, 2003b; Corron *et al.*, 2000; Hunt, 1991; Myneni *et al.*, 1999; Murali & Sinha, 2003; Chlouverakis & Adams, 2005]. It is also equally potent as a special-applications "search chip", which can be added on to regular circuitry and should prove especially useful in machines, which are repeatedly employed for selection/search operations.

In terms of the processor timescale, the search operation requires one dynamical step, namely one unit of the processor's intrinsic update time. The principal point here is the scope for *parallelism* that exists in our scheme. This is due to the selection process occurring through *one global shift*, which implies that there is no scale-up (in principle) with size N. Additionally conventional search algorithms

work with ordered lists, and the time required for ordering generically scales with N as $O\ (N \log N)$. Here in contrast, there is no need for ordering, and this further reduces the search time.

Regarding information storage capacity, note that we employ an M-state encoding, where M can be very large in principle. This offers much gain in encoding capacity. As in the example we present above, the letters of the alphabet are encoded by one element each; binary coding would require much more hardware to do the same.

Specifically, consider the illustrative example of encoding a list of names, and then searching the list for the existence of a certain name. In the current ASCII encoding technique, each ASCII letter is encoded into two hexadecimal numbers or 8 bits. Assuming a maximum name length of k letters, this implies that one has to use $8 * k$ binary bits per name. So typically the search operation scales as $O\ (8kN)$.

Consider in comparison what our scheme offers: if base 26 ("alphabetical" representation) is used, each letter is encoded into *one* dynamical system (an "alphabit"). As mentioned before, the system is capable of this dense encoding as it can be controlled on to 26 distinct fixed points, each corresponding to a letter. Again assuming a maximum length of k letters per name, one needs to use k "alphabits" per name. So the search effort scales as kN. Namely, the storage is 8 times more efficient and the search can be done roughly 8 times faster as well!

If base S encoding is employed, where S is the set of all possible names (size(S) $\leq N$), then each name is encoded into one dynamical system with S fixed points (a "superbit"). So one needs to use just 1 "superbit" per name, implying that the search effort scales simply as N, i.e. $8k$ times faster than the binary encoded case.

In practice the final step of detecting the maximal values can conceivably be performed in parallel. This would reduce the search effort to two time steps (one to map the matching item to the maximal value and another step to detect the maximal value simultaneously). In that case the search effort would be $8k$N times faster than the binary benchmark.

Alternate ideas to implement the increasingly important problem of search have included the use of quantum computers [Grover, 1997]. However, our nonlinear dynamical scheme has the distinct advantage that the enabling technology for practical implementation need not be very different from conventional silicon devices. Namely, the physical design of a dynamical search chip should be realizable through conventional CMOS circuitry. Implemented at the machine level, this scheme can perform unsorted searches efficiently. Primitive CMOS circuit realizations of chaotic systems, like the tent map, already operate in the region of 1 MHz. Thus a complete search for an item comprising of search key addition, update, threshold detection, and list restoration can be performed at 250 kHz, regardless of the length of the list. Commercial efforts are underway to construct VLSI circuitry in GHz ranges and are showing promising results in terms of power, size and speed.

Finally, regarding the general outreach of the scheme: nonlinear systems are abundant in nature, and so embodiments of this concept can be conceived in many different physical systems ranging from fluids to electronics to optics. Potentially good candidates for physical realization of the scheme include nonlinear electronic circuits and optical devices [Garcia-Ojalvo & Roy, 2001]. Also systems such as single electron tunneling junctions [Yang & Chua, 2000], which are naturally piecewise linear maps, can conceivably be employed to make such search devices. All of this underscores the general scope of the concept. In particular, we have implemented this idea with a model of a resistively shunted Josephson junctions with current bias and RF drive (details follow in a future publication [Ditto & Sinha, 2008]). So the very general idea of using a nonlinear function for selection of a particular (desired) state can find embodiment in many different physical contexts.

In summary we have presented a method using nonlinear dynamical elements to store information efficiently and flexibly. We demonstrate how a single element can store M items, where M can be large and can vary to best suit the nature of the data being stored and the application at hand. Namely, we obtain information storage elements of flexible capacity, capable of naturally storing data in different bases or in different alphabets or with multilevel logic. This cuts down space requirements by $\log_2 M$ vis-à-vis elements storing via binary bits. Further we have shown how this method of storing information can be naturally exploited for searching of information. In particular, we demonstrate a scheme to determine the existence of an item in the unsorted list. The scheme involves a single global shift operation applied simultaneously to all the elements comprising the list and this

operation, after one dynamical step, pushes the element(s) storing the matching item (and only those) to a unique, maximal state. This extremal state can then be detected by a simple level detector, directly giving the number of matches. Nonlinear dynamics is exploited as a powerful "preprocessing" tool, reducing the determination of matching patterns to the detection of maximal states. The scheme can also be extended to identify inexact matches. Since the method involves just one parallel procedural step it is naturally set-up for parallel implementation on existing and future implementations of chaos-based computing hardware ranging from conventional CMOS-based VLSI circuitry to more esoteric chaotic computing platforms such as magneto-based circuitry [Koch, 2005] and high speed chaotic photonic integrated circuits operating in the GHz frequency range [Yousefi *et al.*, 2007].

Acknowledgments

We acknowledge the support of the Office of Naval Research [N000140211019] and of ChaoLogix, Inc.

References

Chlouverakis, K. E. & Adams, M. J. [2005] "Optoelectronic realization of NOR logic gate using chaotic two-section lasers," *Electron. Lett.* **41**, 359–360.

Corron, N. J., Pethel, S. D. & Hopper, B. A. [2000] "Controlling chaos with simple limiters," *Phys. Rev. Lett.* **84**, 3835–3838.

Garcia-Ojalvo, J. & Roy, R. [2001] "Parallel communication with optical spatiotemporal chaos," *IEEE Trans. Circuits Syst.-I* **48**, 1491–1497.

Glass, L. & Zeng, W. [1994] "Bifurcations in flat-topped maps and the control of cardiac chaos," *Int. J. Bifurcation and Chaos* **4**, 1061–1067.

Grover, L. [1997] "Quantum mechanics helps in searching for a needle in a haystack," *Phys. Rev. Lett.* **79**, 325–328.

Hunt, E. [1991] "Stabilizing high-period orbits in a chaotic system — the diode resonator," *Phys. Rev. Lett.* **67**, 1953–1955.

Koch, R. [2005] "Morphware," *Sci. Amer.* **293**, 56–63.

Krikelis, A. & Weems, C. C. (eds.) [1997] *Associative Processing and Processors* (IEEE Computer Science Press).

Munakata, T, Sinha, S. & Ditto, W. L. [2002] "Chaos computing: Implementation of fundamental logical gates by chaotic elements," *IEEE Trans. Circuits Syst.-I* **49**, 1629–1633.

Murali, K. & Sinha, S. [2003] "Experimental realization of chaos control by thresholding," *Phys. Rev. E* **68**, 016210.

Murali, K., Sinha, S. & Ditto, W. L. [2003a] "Implementation of NOR gate by a chaotic Chua's circuit," *Int. J. Bifurcation and Chaos* **13**, 2669–2672.

Murali, K., Sinha, S. & Ditto, W. L. [2003b] "Realization of the fundamental NOR gate using a chaotic circuit," *Phys. Rev. E* **68**, 016205.

Myneni, K., Barr, T. A., Corron, N. J. & Pethel, S. D. [1999] "New method for the control of fast chaotic oscillations," *Phys. Rev. E* **83**, 2175–2178.

Prusha, B. S. & Lindner, J. F. [1999] "Nonlinearity and computation: Implementing logic as a nonlinear dynamical system," *Phys. Lett. A* **263**, 105–111.

Sinha, S. [1994] "Unidirectional adaptive dynamics," *Phys. Rev. E* **49**, 4832–4842.

Sinha, S. & Ditto, W. L. [1998] "Dynamics based computation," *Phys. Rev. Lett.* **81**, 2156–2159.

Sinha, S. & Ditto, W. L. [1999] "Computing with distributed chaos," *Phys. Rev. E* **81**, 2156–2159.

Sinha, S. [2001] "Using thresholding at varying intervals to obtain different temporal patterns," *Phys. Rev. E* **63**, 036212.

Sinha, S., Munakata, T. & Ditto, W. L. [2002a] "Parallel computing with extended dynamical systems," *Phys. Rev. E* **65**, 036214.

Sinha, S., Munakata, T. & Ditto, W. L. [2002b] "Flexible parallel implementation of logic gates using chaotic elements," *Phys. Rev. E* **65**, 036216.

Sinha, S. & Ditto, W. L. [2008] to be published.

Wagner, C. & Stoop, R. [2001] "Optimized chaos control with simple limiters," *Phys. Rev. E* **63**, 036212.

Yang, T. & Chua, L. O. [2000] "Nonlinear dynamics of driven single-electron tunneling junctions," *Int. J. Bifurcation and Chaos* **10**, 1091–1113.

Yousefi, M., Barbarin, Y., Beri, S., Bente, E. A. J. M., Smit, M. K., Notzel, R. & Lenstra, D. [2007] "New role for nonlinear dynamics and chaos in integrated semiconductor laser technology," *Phys. Rev. Lett.* **98**, 044101.