Feedback Controlled Software Systems

William B. Dunbar Eric Klavins Stephen Waydo

Division of Applied Science and Engineering
California Institute of Technology
Pasadena, CA 91125
{dunbar, klavins, waydo}@caltech.edu

CDS Technical Report 2003-002
March 24, 2003

Abstract: *Software systems generally suffer from a certain fragility in the face of "disturbances" such as bugs, unforeseen user input, unmodeled interactions with other software components, and so on. A single such disturbance can make the machine on which the software is executing hang or crash. We postulate that what is required to address this fragility is a general means of using feedback to stabilize these systems. In this paper we develop a preliminary dynamical systems model of an arbitrary iterative software process along with the conceptual framework for "stabilizing" it in the presence of disturbances. To keep the computational requirements of the controllers low, randomization and approximation are used. We describe our initial attempts to apply the model to a faulty list sorter, using feedback to improve its performance. Methods by which software robustness can be enhanced by distributing a task between nodes each of which are capable of selecting the "best" input to process are also examined, and the particular case of a sorting system consisting of a network of partial sorters, some of which may be buggy or even malicious, is examined.*

1 Introduction

Software systems are very often not robust to disturbances, which may come in the form of bugs, unforeseen input, unexpected interactions with other system components (both hard- and software), and so on. A single disturbance is often sufficient to cause an instability in the system, and the software can cause its host machine hang or crash. The language of dynamical systems is a natural one in which to express the idea of stability, and in this paper we explore how software systems can be modeled within this framework. When a software system can be described in this way, control theory can provide a methodology by which feedback can be applied to (provably) ensure robustness to certain types of disturbances. In this paper we describe the tools needed to apply these concepts to a general iterative software system, then apply the model to a faulty list sorter. Feedback is shown to improve the faulty sorter's performance. We then examine a system wherein multiple sorters that use feedback to monitor their progress are networked together and are thus able to exchange their states.

In Section 2 we explore a preliminary model of the time/space evolution of a generic software system and suggest analogs to the traditional control-theoretical notions of estimators and controllers that may be used to feedback-stabilize and thereby improve the performance of the system. These modeling concepts are applied, in Section 4, to sorting algorithms. In Section 4.1 we define an appropriate metric on the group of permutations of n elements and a *pseudo-energy function* measuring the sortedness of a list for the purposes of control. Section 4.2 provides an analysis of the open-loop dynamics of a faulty list sorter using a Markov

chain model. In Section 4.3 we use a simple feedback controller to stabilize the sorting system from Section 4.2. In Section 5 we describe a distributed array of nodes in which each node consists of a partial sorter and a controller. We investigate the behavior of the system through simulation. Conclusions and future directions are given in Section 6.

2 A Control Theoretic Approach to Modeling Software

A piece of software along with its data (and the hardware upon which it is run) can be modeled as a dynamic system with internal state x, input u, and output y. Incorrect operation may manifest in the form of bugs in the software and unforeseen inputs or interactions with other software components. These types of uncertainty correspond to the standard notions of *model uncertainty* and *external disturbance* in robust control theory [6]. For the purpose of the present discussion, we refer simply to disturbances without making the above distinction.

A particularly interesting class of programs we investigate are iterative processes that do not run to completion but instead provide output after some number of steps n, then use this output as their input for the next set of iterations. If a disturbance, such as occasional incorrect iteration, exists in such a system, there is no guarantee that the process will converge to the correct output, or even that it will converge at all. Because iterative processes already incorporate feedback in the sense that the output of one iteration is the input to the next (as is usually the considered the case in dynamical systems and control), applying control to iterative software processes may be useful step toward correcting aberrant behaviors.

2.1 States, Metrics and Measures

To develop a systems theory perspective of software systems, we consider defining a state $x \in X$ of the system and a metric d on X, which quantifies the "closeness" of two system states. For a software system, x may simply be a snapshot of the memory used by the software. A metric d, describing the "distance" between two states, is a means by which we may determine how well the software system is performing its assigned task. Metrics are typically used for analysis of a system, e.g. to define stability of a system state. However, in software systems X is rarely a metric space (or even a reasonable topological space), and thus we may need to resort to a surrogate for d that, while not strictly a metric, can serve a similar purpose. For example, we use a *Pseudo-Energy Function* $V : X \to \mathbb{N}$ of the system state, with the set $\{x \mid V(x) = 0\}$ defining the goal state. We desire that V does not increase dramatically in normal operation of the software and decreases when the software is bug-free. The pseudo-energy function is a quantity useful for feedback controller design as well as for analysis and can be thought of, roughly, as a *Lyapunov Function* [12] for the system. In the context of sorting algorithms where $X = S_n$, the symmetric group of all permutations of $\{1, ..., n\}$, we define several pseudo-energy functionsand a metric in Section 4.1.

2.2 Stability of Software

Assume we have a model of the software system, for example a discrete-time state transition relation. Once a suitable metric has been defined on the appropriate (state) space, the notions of stability and performance can be formalized. Denote a fixed point of the system model x_{fp}. In the context of sorting, the desired fixed point state is the sorted version of the list, i.e. $x_{fp} = [1, 2, ..., n]$ for $X = S_n$, the set of all permutations of $\{1, ..., n\}$. Suppose that the state of the system at iteration $k \in \mathbb{N}$ is x_k. Given a well-defined metric $d(x_k, x_j) : X \times X \to \mathbb{R}$, a fixed-point x_{fp} is said to be *stable* if and only if

$$\forall \epsilon > 0,\ \exists \delta > 0\ :\ d(x_0, x_{fp}) < \delta \Rightarrow d(x_k, x_{fp}) < \epsilon,\ \forall k > 0.$$

The fixed point is *asymptotically stable* if it is stable and

$$\lim_{k \to \infty} d(x_k, x_{fp}) = 0.$$

As will be seen, a pseudo-energy function does not in general correspond to a unique state. However, the pseudo-energy may be adequate to describe a system from a control perspective. For this reason, we may treat the pseudo-energy as the state of our model and consider the system dynamics with respect to this measure. The state space of this reduced-order model is $\tilde{X} = \{0, 1, ..., V_{max}\}$ and V_{max} is determined by the chosen measure. As such, we can define the metric $d(V_k, V_j) \equiv |V(x_k) - V(x_j)|$. For the pseudo-energies defined in this paper, a sorted list uniquely corresponds to $V(x_{fp}) = 0$.

2.3 Sensors and Estimators

The state x of a software process may be enormously complicated. In fact, the computational cost of determining $V(x)$ may be equivalent to the cost of executing the software process to be controlled to completion. Clearly, some method of generating an approximation of x, and consequently $V(x)$, will be necessary. The main observation of this section is that a computationally inexpensive approximation of $V(x)$ is similar to a noisy sensor. In control, an "ideal sensor" provides continuous information about some portion or all of the state. In practice, sensors provide a discretized (and therefore approximate) version of this information. In the context of sensing software, we already have a truly discrete process, but require approximation, by spatial discretization of the information available, for computational efficiency. In both contexts, the more time/space the sensors are given to collect information, the more reliable the control decision could be, at the price of increased delay. Our postulated role for sensing in feedback-stabilized software processes is detailed in Section 2.5. In this context, a generic control algorithm would incorporate a sensor that sub-samples state information according to some randomized algorithm.

In control theory, an estimator [21] uses a model of the process dynamics along with the system inputs and outputs to generate an estimate of the system state in the presence of noisy and/or incomplete data. This concept can be extended to software systems, with the distinction that the estimator will be here used to compute a pseudo-energy function of the state, rather than the state itself. Generally speaking, the estimator takes the sensed information (state approximation) along with knowledge of the software's inputs and nominal operation (or a model), to approximate the true pseudo-energy of the state. The example investigated in Section 4.3 is intended to help solidify these abstractions.

2.4 Controllers

Consider now designing a controller that takes as input an estimate of x and/or $V(x)$ and decides the next action the software should take. We suppose that at each step the controller allows the system to perform I iterations. In this case, several possibilities for control action are available. First, the controller may judge that after I iterations, the new output is "further" from the goal state than the previous output (i.e. the pseudo-energy has increased). If the system is nondeterministic or probabilistic, the controller might simply instruct the software to perform the previous set of I iterations again (rollback) – in the hope that whatever went wrong the first time through does not happen again. Second, the controller might adjust the number I of iterations the software performs at each step.

Given the probability that any single iteration is correct, and given that some quantity of processing resources are used during a state estimate and pseudo-energy comparison, there will be an optimal number of iterations to perform between such comparisons. If the software is very reliable, all processing resources should be devoted to running it. As reliability decreases, checks should become more frequent to maximize the amount of progress made, or a decision to abort may be appropriate. This reliability figure, however, will not in general be known a priori, and may in fact change in time as more knowledge is gained about the behavior of the system. In this case active control could be used to ensure that the system is sitting near an optimal point.

Another implementation of active control may be in tuning the fidelity of the estimation/sensing loop. While general estimators in control theory are limited in some way by the amount of noise in a system, software estimators can be made arbitrarily accurate as more processing resources are devoted to them. The estimated accuracy of the current pseudo-energy estimate may be used to tune the fidelity of future estimates, perhaps by increasing or decreasing the sub-sampling resolution in the sensor.

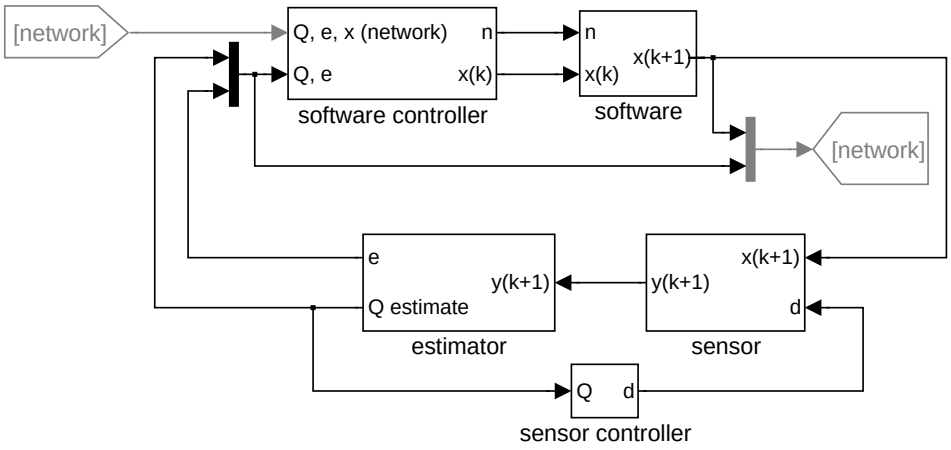


Figure 1: A block diagram of a feedback controlled software system.

2.5 A Feedback Controlled Software Model

Figure 1 is a block diagram depicting a feedback controlled software system. At the top of the diagram is the a block representing the software itself, which we consider to be the dynamic system to be controlled. The software has a state that is some function of its internal variables and that evolves in some manner particular to the task that the software performs. The number of iterations n the software performs before producing output is an input coming from the controller. There is also an input port on the software block that allows the controller to replace the state of the software with another state, for example to roll back a set of incorrect iterations or to incorporate new information coming in from other nodes on a network. After performing n iterations, the software outputs its state x. For the reasons discussed above, it is likely that operating on this full state will be too computationally intensive for active control, so it is passed to a "sensor" which performs a sub-sampling operation to produce a less information rich but more manageable output set y. The sensor takes as a control input d, the sub-sampling density; this density can be actively tuned by the sensor controller to suit the current needs of the system. The reduced information set y forms the input to an "estimator," which evaluates an approximation to the pseudo-energy V of the software state, outputting its estimate $\hat{V}$ and error bound e. This information, possibly along with similar information from agents elsewhere in a network of similar controlled software systems, comprises the input to the controller. This controller can then adjust the number of iterations the software does between checks, n, based on some function of how the software has affected the pseudo-energy of the state. For example, the software controller could output a new value for n as a PID function on $\hat{V}$. In the context of distributed software systems, another function of the controller could be to compare the outputs of some number of other nodes on the network to the output from the node in question and provide the "best" data set to its software process as input for the next set of iterations.

3 Related Work

The research we report on in this paper is is closely related to the theory of self-checking programs as described in, for example, [17, 23, 3]. In this theory, checkers C are constructed that check the accuracy of programs P that (supposedly) compute some function f. The checker is supposed to fundamentally differ from the program (e.g. it might have a different computational complexity). For example, one might check the result of a program P that computes $y = f(x) = \sqrt{x}$ by multiplying y by itself to see if it equals x. The problem of sorting has also been considered in this community [3], albeit in a different manner. In particular, in this paper we suppose that a software process may be monitored *as it is executing* to see if it is converging

to a correct answer or diverging. Thus, we check at each iteration of a sorting program whether the list on which it is operating is getting *more sorted*. In the self-testing literature, one first attempts to sort the list with P (running the program to completion), and then checks that the result is in fact sorted (which can be done in linear time). The present work, therefore, is more concerned with robustness and disturbance rejection of an executing process, while the latter is concerned with what programs can be checked and corrected and the complexity classes involved. Given the similarities, however, we are extremely interested in the relation between self-testing and feedback control theory.

Somewhat related to checking a program as it executes is the design of synchronous observers [11]. In [10] finite state machine abstractions of programs written in the data-flow programming language LUSTRE [9] are used to verify safety properties. The abstractions specify more behaviors than the actual code (i.e. they contain all *safe* behaviors) and are easier to verify than the original programs. Presumably, the abstractions could also be executed in parallel with the actual programs as models of their behavior. If the program ever violates the behavior of the observer (because it has a bug, for instance), a warning flag is raised and the program can be shut down. Similar, if less formal, ideas appear in operating systems [22, 19, 8] where various quantities can be measured to determine of a system is operating normally, or if it has ben "infected" or otherwise compromised. We expect such an approach to be useful in control theory. What has not been investigated, to the best of our knowledge, is the use of feedback to control rather than merely terminate a faulty process. The addition of a control input to a program may in fact require fundamental reworking of the program/observer paradigm. We hope that the present paper suggests a possible avenue for such an effort.

The networked sorters example in Section 5 resembles N-Version Programming [2], where a number of separate implementations P_1, ..., P_k of the same function f are computed on separate processors. A voting mechanism is used to determine which of the outputs to use. The goal is that because the implementations are software engineered separately, they will not have bugs [20] on the same inputs – safety is thereby acheived through redundancy. We are interested in the logical extension of this idea to large networks of software processes (such as networked computers on the internet, or parallel processing systems) wherein each node receives data from adjacent nodes and computes some function of that data. Robustness and control in our networked systems is thus similar to problems found in for example internet congestion control [16] and coupled oscillators [14].

Also similar is the idea of self-stabilizing protocols [5, 18] wherein a network of processors executing a self-stabilizing protocol can be shown to recover from disturbances and arbitrary initial conditions into a set of *legal* states. In fact, the same analogy that we employ here of a *pseudo-energy* function, or Lyapunov function, can be used to show the protocols are robust and stable as is demonstrated in [13]. We believe the idea of self-stabilization exactly corresponds to robust control design and hope to make this and related notions formal in future work.

4 Sorting Lists

4.1 Metrics and pseudo-energy functions for List Sorting

In this subsection we describe the set of lists and the metrics and pseudo-energy functions that can be defined on them. In subsequent subsections, we use some of these definitions, but not all. The purpose of this subsection, then, is to illustrate the kinds of properties we would like to have in a software process.

We make the simplifying assumption that all lists generated by partial sorting are equal when viewed as sets, although many of the results herein do not require it. Therefore, a faulty sorter or disturbance may unsort the list, but the assumption requires that the list may not change as a set. A list $L = [L[1], ..., L[n]]$ drawn from the set $\{1, 2, ..., m\}$ is a sequence of n ordered and distinct elements. We further assume that $m = n$: the set of all lists of length n is then the symmetric group S_n of all permutations of $\{1, ..., n\}$. A list L is sorted if $L[i] < L[j]$, for all $i < j$, and we denote the sorted list as $L^* = [1, 2, ..., n]$. A metric for sortedness quantifies the distance between any two lists in a given group. Pseudo-energy functions, in the context of sorting, refer to functions from $S_n \to \mathbb{N}$ that rank lists by sortedness. For example, a (trivial)

pseudo-energy function might output 0 if the list is sorted and 1 otherwise. Pseudo-energy functions can be used to prove the correctness of a particular sorting algorithm, e.g. Bubblesort [15]. From the control analysis perspective, a metric is likely to prove useful in verifying properties of the closed-loop behavior of a sorter/controller agent. The function of the controller as described above requires the pseudo-energy for any given list. We now give a few example pseudo-energy functions and metrics for list sortedness.

Definition 4.1 (Total Inversion Function). The *total inversion function* V_{TI} of a list L is

$$V_{TI}(L) \triangleq \sum_{i=1}^{n}\sum_{j=i}^{n}\langle L[i]-L[j]\rangle$$

where

$$\langle x\rangle \triangleq \begin{cases} 1, & x>0 \\ 0, & \text{otherwise.} \end{cases}$$

In words, V_{TI} gives the total number of pairs that are out of order, with a maximum of $\binom{n}{2}$. Determining V_{TI} is $\mathrm{O}(n^2)$. A simpler pseudo-energy function is defined next:

Definition 4.2 (Adjacent Inversion Function). The *adjacent inversion function* V_{AI} of a list L is

$$V_{AI}(L) \triangleq \sum_{i=2}^{n}\langle L[i]-L[i-1]\rangle.$$

This function gives the total number of adjacent pairs out of order, with a maximum of $n-1$. Determining V_{AI} is $\mathrm{O}(n)$. Another measure is n minus the length of the longest increasing sub-sequence. Discussion of these pseudo-energy functions and other examples are given in [15] (where pseudo-energy functions are called *measures*).

A given value for a pseudo-energy function does not in general define a unique list, except possibly for the zero value (corresponding to the sorted list) and in most cases the maximum value of the function. In the cases of the total and adjacent inversion functions, the range of the pseudo-energy functions are $\{0,1,...,\binom{n}{2}\}$ and $\{0,1,...,n-1\}$, respectively. Thus, V_{TI} gives a finer resolution over S_n than V_{AI}. For example, consider the lists

$$L_1=[1,3,4,2] \quad \text{and} \quad L_2=[3,4,1,2].$$

The pseudo-energies are $V_{TI}(L_1)=2$, $V_{TI}(L_2)=4$ and $V_{AI}(L_1)=V_{AI}(L_2)=1$, so the two lists are equally sorted in the sense of adjacency while L_2 is less sorted according to the total inversion function. The trade-off between pseudo-energy resolution and computation demands is apparent.

Two metrics, the Kendall distance K and Spearman's footrule distance F are given in [1] when $m=n$ (which we have assumed to be the case). The time complexity of determining K and F are $\mathrm{O}(n^2)$ and $\mathrm{O}(n)$, respectively. We now define two vectors that will be used in the metric we give below.

Definition 4.3 (Total Inversion Vectors). The *total inversion vector* $\boldsymbol{q}: S_n \to \{0,...,n-1\}^n$ has n components $[q_1(L),...,q_n(L)]^T$, where the k^{th} component is defined by

$$q_k(L)=\sum_{j=k}^{n}\langle L[k]-L[j]\rangle.$$

The *ordered total inversion vector* $\boldsymbol{q}^o: S_n \to \{0,...,n-1\}^n$ has n components $[q_1^o(L),...,q_n^o(L)]^T$, where the $L[k]^{th}$ component is defined by

$$q^o_{L[k]}(L)=\sum_{j=k}^{n}\langle k-L[j]\rangle.$$

In words, q_i^o is the number of elements less than i, located in $\{L[1], ..., L[n]\}$, to the right of i. The definition for $\boldsymbol{q}^o$ is based on [4] and references therein, which discuss the construction of a (total) inversion list from a given permutation. The reason for the use of the word "ordered" in defining $\boldsymbol{q}^o$ is that its construction depends upon the location of each $L[k]$ relative to its value; consequently, the definition does not generalize to operating on lists. On the other hand, component k in the $\boldsymbol{q}$ vector corresponds to the $i = k$ summation term in the expression for V_{TI}. As such, $\boldsymbol{q}$ is already well-defined for operating on lists, rather than being restricted to permutations. Note that the n^{th} component in $\boldsymbol{q}$ and the 1^{st} component in $\boldsymbol{q}^o$ are always zero. We now define a metric based on the ordered total inversion vector.

Lemma 4.1. *Given the function $d : S_n \times S_n \to \mathbb{R}$ defined by*

$$d(L_1, L_2) \triangleq \|\boldsymbol{q}^o(L_1) - \boldsymbol{q}^o(L_2)\|,$$

where $\|\cdot\|$ is any norm on $\mathbb{R}^n$, (S_n, d) is a metric space.

Proof. The positivity, symmetry and triangle inequality properties of this metric follow from the properties of the norm on $\mathbb{R}^n$. The only non-obvious condition that should be proved is that $d(L_1, L_2) = 0 \Rightarrow L_1 = L_2$, which is equivalent to proving $\boldsymbol{q}^o(L_1) = \boldsymbol{q}^o(L_2) \Rightarrow L_1 = L_2$. This is proved by showing that for any given $\boldsymbol{q}^o(L)$, a unique L can be constructed.

Component q_1^o is always zero, as there are no elements less than 1. Component $q_2^o \in \{0, 1\}$ and if $q_2^o = 1$, 2 is left of 1 in L, otherwise 2 is right of 1 in L. Component $q_3^o \in \{0, 1, 2\}$ and if $q_3^o = 2$, 3 is left of 2 and 1; if $q_3^o = 1$, 3 is between 2 and 1; otherwise, 3 is right of 2 and 1. Continuing in this way, L can be reconstructed uniquely given $\boldsymbol{q}^o$. ■

We conjecture that d is still a metric when defined using $\boldsymbol{q}$ rather than $\boldsymbol{q}^o$, i.e. the permutation (or list) L that corresponds to a given $\boldsymbol{q}(L)$ is unique. In this case, it follows from the definitions that $V_{TI}(L) = d(L, L^*)$.

Henceforth, in referencing the metric d above we will utilize the $\|\cdot\|_1$ norm, e.g. on $\mathbb{R}^2$ we have $\|(x, y)\|_1 = |x| + |y|$. From the example above,

$$\boldsymbol{q}^o(L_1)^T = [0, 0, 1, 1],\ \boldsymbol{q}^o(L_2)^T = [0, 0, 2, 2] \quad \Rightarrow \quad d(L_1, L_2) = 2.$$

Spearman's footrule distance F is given by $F(L_1, L_2) \triangleq \sum_{i=1}^{n} |L_1[i] - L_2[i]|$, which is itself a $\|\cdot\|_1$-like norm. Over S_n, the metric F clearly has computational advantages over d and should be used when needed, say, for control or analysis. However, there is a subtle reason that, in certain circumstances, one might require d. In particular, assume that the universe dimension m is larger than the list length n and that a given random list is to be sorted. In such a case, the sorted version of the list is not available. However, assuming d is defined with $\boldsymbol{q}$, we have that $\boldsymbol{q}(L^*) = \boldsymbol{0}$ and so the distance *to the sorted list* can be computed with d but not F.

The Kendall distance is defined as the number of inversions between two permutations, i.e. for any $\sigma, \tau \in S_n$, where $\sigma(u)$ denotes the rank of u in σ, K is the number of pairs $[u, v]$ such that $\sigma(u) < \sigma(v)$ and $\tau(u) > \tau(v)$. In fact, K is more indicative of the progress of a sorter in the sense of adjacency than the metric d, as K actually gives the number of adjacent sort operations between lists. To see this by example, consider the permutations

$$\sigma = [2, 4, 1, 3], \quad \tau = [1, 4, 2, 3], \quad \xi = [4, 2, 1, 3].$$

The values for the Kendall distances are $K(\sigma, \tau) = 3$, $K(\sigma, \xi) = 1$ and $K(\xi, \tau) = 2$ while the metric d (define with $\boldsymbol{q}^o$) returns $d(\sigma, \tau) = 1$, $d(\sigma, \xi) = 1$ and $d(\xi, \tau) = 2$. It appears that d gives distance in terms of the number of swaps between permutations, not requiring that the swaps be adjacent. If we define d with $\boldsymbol{q}$, we have $d(\sigma, \tau) = 1$, $d(\sigma, \xi) = 3$ and $d(\xi, \tau) = 4$, which has no obvious bearing on the relation to sort operations. For this reason, K may be preferable to d, although the trade-offs in time and space complexity have yet to be determined.

4.2 Open-loop Behavior

To explore the issues involved in stabilizing and improving the performance of a sorting system, we consider a model of the simplest imaginable (buggy) sorting system. The sorter is a dynamic system whose state at step k is the list $L(k)$. The pseudo-energy at time k is taken to be the value of the total inversion function of the list,

$$V(k) \triangleq V_{TI}(L(k)),$$

which for a list of length n can vary from 0 (no pairs are out of order) to $V_{max} = \binom{n}{2}$ (all pairs are out of order). At each time step, the sorter picks an adjacent pair of list entries. We suppose that this is a "correct" operation (i.e. the chosen pair is out of order) with probability p. The sorter then swaps the pair with probability d. If the list is already completely sorted or unsorted ($V = 0$ *or* V_{max}), the sorter simply swaps some adjacent pair with probability d. $L(k)$ is thus a random variable, and $V(k)$ is a random variable that is a function of $L(k)$. The probability distribution of $V(k+1)$ is dependent only on the distribution of $V(k)$, and so it can be modeled using a Markov chain. Define the state transition matrix T with its $(i,j)^{th}$ element given by

$$T_{i,j} \triangleq P[\, V(k+1) = j \mid V(k) = i \,].$$

Denoting a *pseudo-energy value* by $q = V(k)$, a state transition matrix of dimension $m+1 \times m+1$, where $m = V_{max}$, is obtained. Note that swapping an adjacent pair (with distinct values) will always increment or decrement V_{TI} by 1. The state transition probabilities are:

$$\begin{aligned}
T_{q,q} &= (1-d) \\
T_{q,q-1} &= pd, \;\; 1 \le q < m \\
T_{q,q+1} &= (1-p)d, \;\; 1 \le q < m \\
T_{0,1} &= d \\
T_{m,m-1} &= d \\
T_{q,q\pm\delta} &= 0, \;\; \forall\, \delta > 1.
\end{aligned}$$

The left eigenvector of this matrix corresponding to eigenvalue 1, which we call the *stable left eigenvector*, describes the long-term distribution of the pseudo-energy q. Following the method of [7], we have the following proposition.

Proposition 4.1. *The stable left eigenvector of the state transition matrix T is given by:*

$$v \;=\; \left[1, \frac{1}{p}, \frac{1-p}{p^2}, ..., \frac{(1-p)^{i-1}}{p^i}, ..., \frac{(1-p)^{m-2}}{p^{m-1}}, \frac{(1-p)^{m-1}}{p^{m-1}}\right]$$

or:

$$\begin{aligned}
v_0 &= 1 \\
v_i &= \frac{(1-p)^{i-1}}{p^i}, \;\; 1 \le i < m \\
v_m &= \frac{(1-p)^{m-1}}{p^{m-1}}.
\end{aligned}$$

Proof: *It is sufficient to show that $vT = v$:*

$$
\begin{aligned}
(vT)_i &= \sum_{j=1}^{m} v_j T_{ji} \\
(vT)_0 &= 1(1-d) + \frac{1}{p}pd && = 1 && = v_0 \\
(vT)_1 &= 1d + \frac{1}{p}(1-d) + \frac{1-p}{p^2}pd && = \frac{1}{p} && = v_1 \\
(vT)_i &= \frac{(1-p)^{i-2}}{p^{i-1}}(1-p)d + \frac{(1-p)^{i-1}}{p^i}(1-d) + \frac{(1-p)^i}{p^{i+1}}pd && = \frac{(1-p)^{i-1}}{p^i} && = v_i,\ 2 \le i < m-1 \\
(vT)_{m-1} &= \frac{(1-p)^{m-3}}{p^{m-2}}(1-p)d + \frac{(1-p)^{m-2}}{p^{m-1}}(1-d) + \frac{(1-p)^{m-1}}{p^{m-1}}d && = \frac{(1-p)^{m-2}}{p^{m-1}} && = v_{m-1} \\
(vT)_m &= \frac{(1-p)^{m-2}}{p^{m-1}}(1-p)d + \frac{(1-p)^{m-1}}{p^{m-1}}(1-d) && = \frac{(1-p)^{m-1}}{p^{m-1}} && = v_m
\end{aligned}
$$

so,

$$vT = v$$

and v is a stable left eigenvector of T.

Normalizing v we obtain the long-term probability distribution of V,

$$
\begin{aligned}
\eta &= \sum_{i=0}^{m} v_i, \\
v' &= \frac{v}{\eta}.
\end{aligned}
$$

The weighted sum of the entries of v' is the asymptotic expected value of V, which we call the *fixed point pseudo-energy* V_{fp},

$$V_{fp} \triangleq \lim_{k\to\infty} E[V(k)] = \sum_{i=0}^{m}(i)v'(i) = \frac{p^m - (1-p)^m[1+2m(2p-1)]}{2(2p-1)[p^m-(1-p)^m]}.$$

Figure 2(a) depicts the dependence of V_{fp} on p for a list of length 10. Note that the curve drops sharply around $p = 0.5$, and that relatively favorable values of V_{fp} are reached even for fairly low p, in the range of 0.6 and greater.

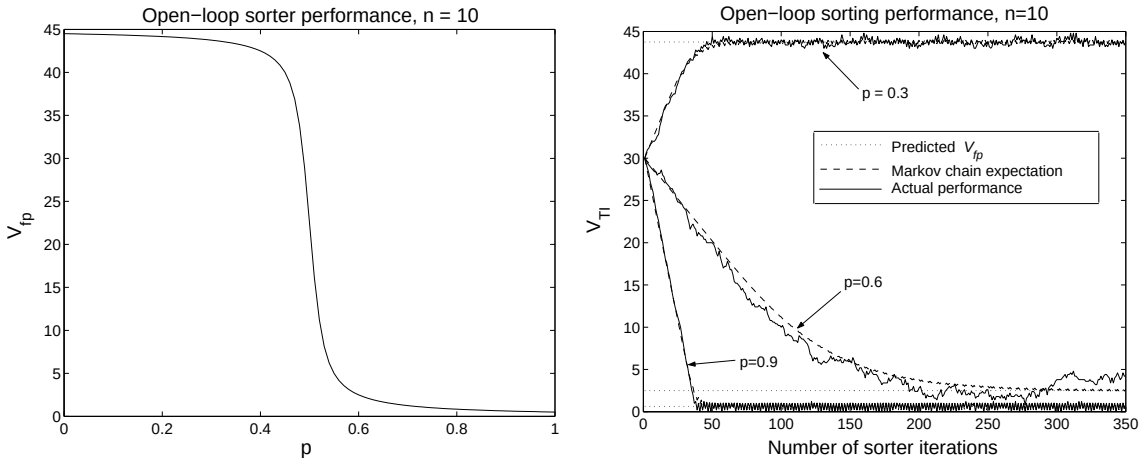
Figure 2(b) is a plot of the Markov chain-predicted time history, the predicted V_{fp}, and a time average of 10 actual sorting runs for a list of length 10. The actual sorter performance closely matches that predicted by the Markov chain analysis.

4.3 Closing the Loop

The above Markov chain model can be extended to show the benefit of including a simple controller. We now model the same sorter along with an approximate checker that computes an approximation $\hat{V}$ of V. After each iteration k, the checker picks l random pairs and calculates $\hat{V}(k)$, the number of sub-sampled pairs that are out of order. The checker then rejects the sorting step if $\hat{V}(k) \ge \hat{V}(k-1)$. The accuracy of the checker is derived as follows: The sample space of the checker consists of $\binom{n}{2}$ pairs. Let a given pair have value 1 if it is out of order, and 0 if it is in order (or if the values are the same). Again use:

$$V(k) \triangleq V_{TI}(L(k)).$$

If $V(k) = b$, the sample space of the checker consists of b out of order pairs and $\binom{n}{2} - b$ in order pairs. In order for $\hat{V}$ to be equal to some value c, the checker must pick c out of order pairs and $l - c$ in order pairs.



(a) Variation of V_{fp} with p.

(b) Comparison of sorter performance to model.

Figure 2: Open-loop sorter – Analysis and simulation results.

The probability that the checker does so is the number of ways it can pick c of the b out of order pairs times the number of ways it can pick $l-c$ of the $\binom{n}{2}-b$ in order pairs divided by the number of ways it can pick l of the $\binom{n}{2}$ total pairs,

$$P[\hat{V} \;=\; c \mid V \;=\; b] \;=\; \frac{\binom{b}{c}\binom{\binom{n}{2}-b}{l-c}}{\binom{\binom{n}{2}}{l}}.$$

Two probabilities are used to characterize the checker – the probability r_1 that it recognizes a good step as decreasing V and the probability r_2 that it recognizes a bad step as increasing V. The probability that the checker recognizes a correct step is the probability that $\hat{V}$ improves given that V improves, and is a function of b and l given by

$$r_1(b,l) \;=\; P[\hat{V}(k) \;<\; \hat{V}(k-1) \mid V(k-1) \;=\; b, V(k) \;=\; b-1].$$

Because $\hat{V}(k)$ and $\hat{V}(k-1)$ are separate measurements, they are independent random variables, and

$$P[\hat{V}(k-1) = c_1,\ \hat{V}(k) = c_2 \mid V(k-1) = b,\ V(k) = b-1]$$

$$= P[\hat{V}(k-1) \;=\; c_1 \mid V(k-1) \;=\; b]\; P[\hat{V}(k) \;=\; c_2 \mid V(k) \;=\; b-1]$$
$$= \frac{\binom{b}{c_1}\binom{m-b}{l-c_1}\binom{b-1}{c_2}\binom{m-(b-1)}{l-c_2}}{\binom{m}{l}^2},$$

where again $m \;=\; \binom{n}{2}$. Summing the above expression over all c_1, c_2 with $c_2 \;<\; c_1$ gives the needed probability,

$$r_1(b,l) \;=\; P[\hat{V}(k) \;<\; \hat{V}(k-1) \mid V(k-1) \;=\; b, V(k) \;=\; b-1]$$
$$= \binom{m}{l}^{-2} \sum_{c_1=1}^{m} \binom{b}{c_1}\binom{m-b}{l-c_1} \sum_{c_2=0}^{c_1-1} \binom{b-1}{c_2}\binom{m-(b-1)}{l-c_2}.$$

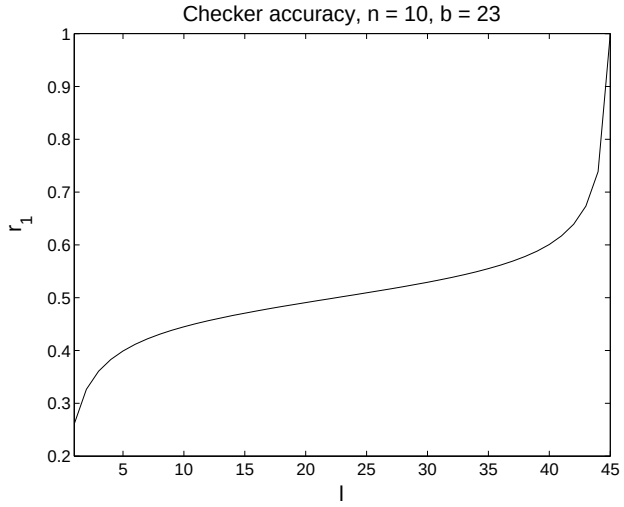


Figure 3: Theoretical checker accuracy

Figure 3 is a plot of r_1 as l ranges from 1 to $\binom{n}{2}$ for a list of length 10 with $V = 23 \approx \binom{n}{2}/2$. Similar reasoning leads to the probability that the checker recognizes a bad step increasing V (or at least failing to decrease V),

$$\begin{aligned} r_2(b,l) &= P[\hat{V}(k) \geq \hat{V}(k-1) \mid V(k-1) = b, V(k) = b+1] \\ &= \binom{m}{l}^{-2} \sum_{c_2=0}^{m} \binom{b+1}{c_2} \binom{m-(b+1)}{l-c_2} \sum_{c_1=0}^{c_2} \binom{b}{c_1} \binom{m-b}{l-c_1}. \end{aligned}$$

The state transition probabilities are very similar to the open-loop case, with the addition that the system may now reject sorter steps (correctly or incorrectly) according to the above probabilities. Using the same definition of the state transition matrix T as above we have:

$$\begin{aligned} T_{q,q} &= (1-d) + pd(1-r_1) + (1-p)dr_2, \;\; 1 \leq q < m \\ T_{q,q-1} &= pdr_1, \;\; 1 \leq q < m \\ T_{q,q+1} &= (1-p)d(1-r_2), \;\; 1 \leq q < m \\ T_{0,0} &= (1-d) + dr_2 \\ T_{0,1} &= d(1-r_2) \\ T_{m,m} &= (1-d) + d(1-r_1) \\ T_{m,m-1} &= dr_1 \\ T_{q,q\pm\delta} &= 0, \;\; \forall\, \delta > 1. \end{aligned}$$

Work is currently in progress to develop a closed-form solution for or approximation to the stable eigenvector of the closed-loop transition matrix following the methods used above. Until such a solution is found, numerical methods can be used to predict v' and V_{fp}.

Note that if the list is fully sorted (i.e. $V = 0$), the subsampled pairs will all be in order and so any sorting step will keep $\hat{V}$ the same or increase it. Thus $r_2 = 1$ and any sorting step will be rejected once the goal state is reached, so the goal state is a sink in the Markov chain. This means that as long as there is some non-zero probability of improvement $p > 0$, closing the loop will result in a system that always fully sorts the list, given enough time. Using the terminology of Section 2.2, applying feedback makes $V = 0$

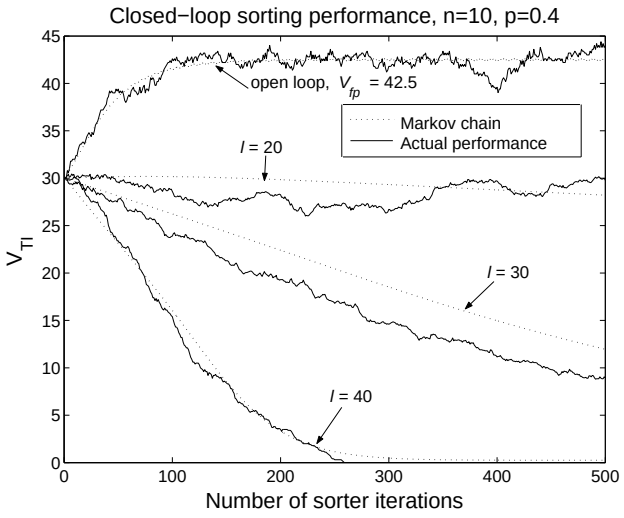


Figure 4: Comparison of closed-loop sorter performance to model.

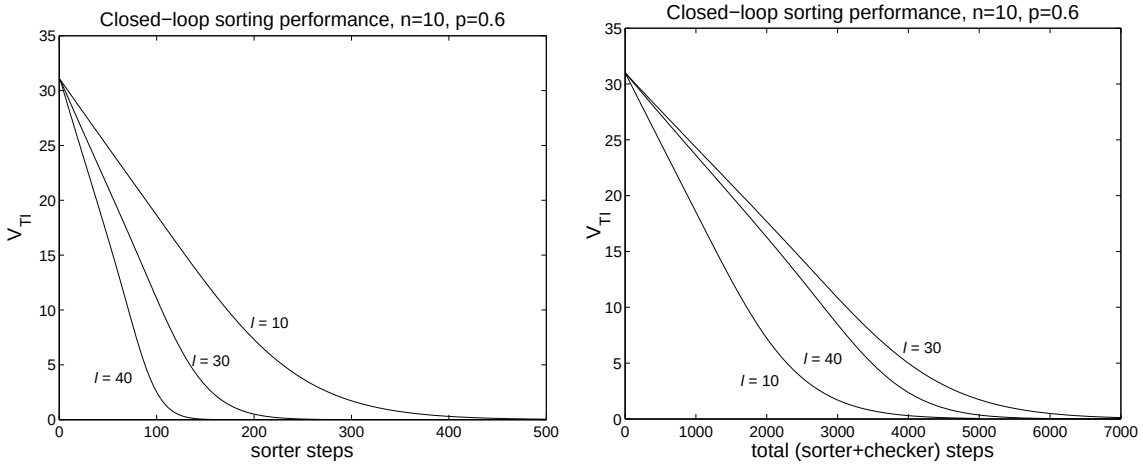
an asymptotically stable fixed point. Adjusting the fidelity of the state estimate alters the rate at which V approaches zero. Higher fidelity checking will result in acceptance of fewer incorrect steps and rejection of fewer correct steps, and so will decrease the number of sorter iterations required.

Figure 4 is a plot of the Markov chain-predicted time history, the predicted V_{fp}, and a time average of 10 actual sorting runs for a list of length 10 and a sorter with $p = 0.4$. The open-loop performance is shown along with that of checkers with l equal to 20, 30, and 40. Note that V_{fp} drops quickly once l becomes larger than $\binom{n}{2}/2 \approx 23$; further increasing l increases the rate at which q approaches V_{fp}. The actual sorter performance again closely matches that predicted by the Markov chain analysis. Note that in all closed-loop cases $V_{fp} = 0$, but lower values for l resulted in much slower convergence rates.

Note that in the above discussion only *sorter* iterations were taken into account when judging convergence rates. In this case, larger values of l will clearly always improve convergence time. When checking steps are taken into account, however, it appears that for a given sorter accuracy there will be an optimal checking density that may not be equal to the maximum or the minimum that will result in the fewest *total* iterations. For the low accuracy sorter depicted in Figure 4, correcting for checker steps did not change the result that larger l improved convergence time, but for a slightly more accurate sorter very different results were obtained. Figure 5 is a plot of the Markov chain-predicted closed-loop time history of the sorting of a list of length 10 by a sorter with $p = 0.6$. In Figure 5(a), only the sorting steps are taken into account, and lower l results in fewer sorting steps as expected. Figure 5(b) depicts the same data, corrected to include checking steps. In this case, $l = 10$ resulted in the fewest total steps to converge and $l = 30$ the most, with $l = 40$ in between. It appears that for sorter accuracies near this value, there will be some optimal l (as a function of p) that may be neither the maximum nor the minimum value that will result in the fastest total convergence time. This is an issue we would like to explore in considerably more detail, especially with regard to how to design a controller that finds a near-optimal checking density without a priori information about the sorting accuracy.

5 A Network of Sorters

Let $G = (U, E)$ be a graph where $U = \{1, ..., n\}$ and $E \subseteq U \times U$. The set of vertices U are intended to refer to the indices in a network of sorters, the edges in E to the connections between them. Let $\mathcal{L}$ denote the set of all lists over a suitable domain. The state of sorter i at time $k \in \mathbb{N}$ is a list $L_i(k) \in \mathcal{L}$. The state of the network is then an n dimensional vector of lists. The operation of each sorter is to attempt to partially sort



(a) sorter steps only

(b) total steps

Figure 5: Effect of including checking steps in convergence time.

one of the lists incoming from its neighbors. In particular, it chooses to partially sort the incoming list it believes is already the best sorted, arriving at this belief by applying an approximate pseudo-energy function to each list and taking the apparently best one.

To each sorter i we associate a sorting function $sort_i$ and a "picking" function $pick_i$ where

$$\begin{aligned} sort_i &: \mathbb{N} \times \mathcal{L} \to \mathcal{L} \\ pick_i &: \mathbb{N} \times \mathcal{P}(\mathcal{L}) \to \mathcal{L}. \end{aligned}$$

Here, $\mathcal{P}(X)$ refers to the power set of the set X. The functions $sort_i$ and $pick_i$ are supposed to make random choices. To model this we suppose each is equipped with a pseudo random number generator that takes as input a natural number k and returns the k^{th} pseudo random number, or a pair of numbers, etc. as required. Thus, each function associated with a sorter takes as its first argument the current time step.

The sorting function operates by randomly choosing a pair of elements from its second argument. If the elements are out of order, it flips them with some probability p_i (which we model with the pseudo random generator, although the intent is really to model unforeseen bugs in the sorter). If the elements are in order, it flips them with probability $1 - p_i$. Thus $p_i = 1$ implies a perfectly good sorter and $p_i = 0$ implies a perfectly bad sorter.

The picking function evaluates a pseudo-energy approximation on each list in its second argument and returns the list with the least pseudo-energy. That is

$$pick_i(k, \mathcal{L}) = L$$

where

$$\hat{V}(k)(L) > \hat{V}(k)(L') \; for \; all \; L' \in \mathcal{L} - \{L\}.$$

The approximate pseudo-energy function $\hat{V}(k)$ when applied to $L \in \mathcal{L}$ chooses κ (a constant) pairs from its argument and returns the number of them that are out of order, thereby approximating $V_{TI}(L)$. As discussed earlier in the paper, approximation is important to make the computations fast enough relative to the partial sorting time scales.

Given the initial states $[L_1(0), ..., L_n(0)]$ the sorting process proceeds according to

$$L_i(k+1) = sort_i(k+1, \; pick_i(k+1, \; \{L_j(k) \mid (i,j) \in E\})). \tag{1}$$

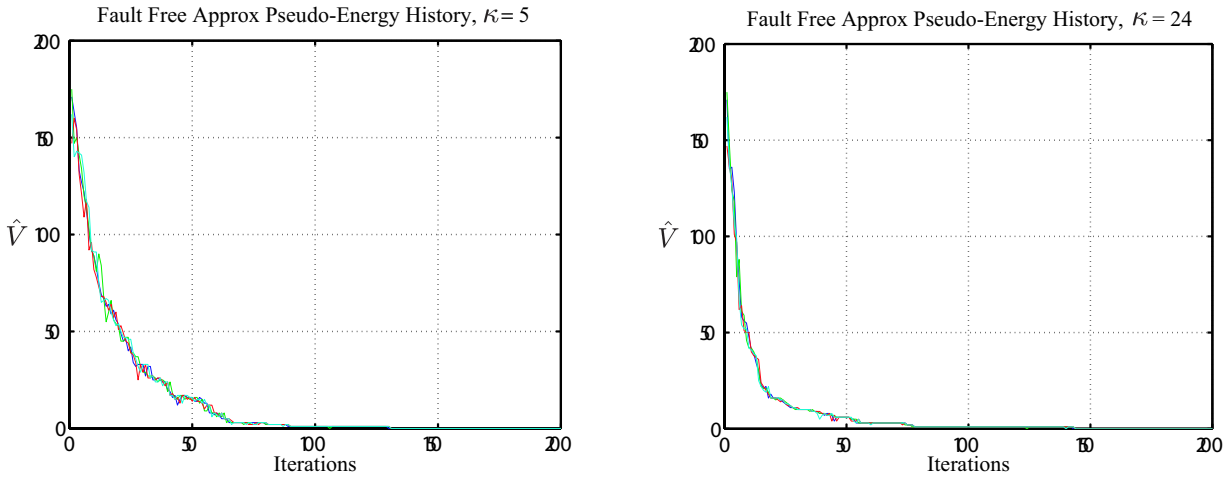


Figure 6: Approximate pseudo-energyas a function of time for four sorters in fully connected network.

We usually view the output of a network of sorters at time k as the state of the sorter with the lowest pseudo-energy at time k. A network is thus said to converge if the pseudo-energy of its output converges. Equivalently, a network is said to converge if

$$x(k) \triangleq \min\{V_{TI}(L_i)(k)) \mid i \in U\}$$

converges. The stability of $x(k)$ and the expected value of $x(k)$ (when the network is considered to be a true random process) are the main indicators of the performance of the network. We are particularly interested in the performance of $x(k)$ in situations where one or more of the sorters in the network is imperfect and κ_i is fairly low (not computationally intensive). Will the outputs of bad sorters propagate through the network or will the pickers be good enough to weed out bad lists? Although we have not yet rendered the class of networks we have described amenable to analysis, we have begun to investigate their performance in simulation studies.

In the following, we consider a fully connected, four node network ($U = \{1, 2, 3, 4\}$ and $E = U \times U$). The parameter j represents the number of sort iterations performed in between picking operations. We hard code $j = 10$ for the results given. The initial lists $L_i(0)$ are identical for all i and equal to a randomly chosen list $L^r(0)$ of length 30, with elements from the set $\{0, ..., 100\}$. Each sort operation randomly picks an ordered pair, with the first element to the left of the second element in the list, and swaps them if they are out of order. The picker computes the (approximate) pseudo-energy for all lists available to it (all of them for a fully connected network) and hands the sorter the list with the lowest pseudo-energy.

In the picker operation, the parameter κ determines some resolution in how close the approximate pseudo-energycomes to the true pseudo-energy. Specifically, given a list of length n, $\kappa \in \{1, ..., n\}$ determines the size of an array window randomly extracted from the original list. Given κ, we can compute $\hat{V}(k)$.

Figure 6 shows the evolution of the approximate pseudo-energyfor all four sorters when $\kappa = 5,\ 24$. The iteration histories shown are actually an average of 50 separate runs, all with a different randomly chosen initial list $L^r(0)$. Notice that the rate of convergence of the pseudo-energy (to zero) is faster for a better approximation of the true pseudo-energy, i.e. for larger κ (when the additional computational steps taken in checking are not taken into account). To explore the affect of a faulty sorter, agent 4 is given a bad sorter that flips a coin and if it is heads, swaps the randomly chosen ordered pair; otherwise, the pair is kept in the original order. The corresponding pseudo-energy histories are shown in Figure 7 for $\kappa = 5,\ 24$, where again we average over 50 runs, all with different $L^r(0)$. The figure shows that for a better approximation of the true pseudo-energy, i.e. for larger κ, the fault-free sorters converge *on average* independently of the faulty sorter. For $\kappa = 5$, convergence suffers as the faulty sorter's list is more likely to be chosen by the good sorters more frequently. We can also make the conjecture from the figures that as the number of iterations

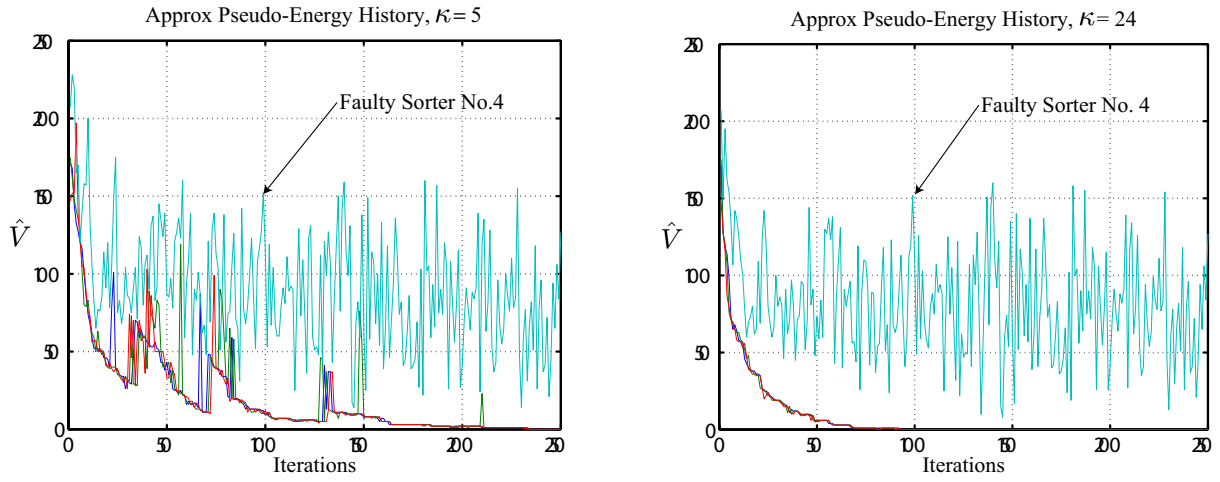


Figure 7: Approximate pseudo-energy history for four sorters in fully connected network; Sorter 4 is faulty.

increases, the chances of the fault-free sorters to converge improves, for the following reason. As the number of iterations increases, the correctly sorted lists from sorters 1-3 becomes substantially more sorted than sorter 4's list, which (on average) maintains its initial unsortedness as the bad and good sort operations are equally likely. As a result, the approximate pseudo-energies of the four lists within pickers 1-3 suggests with increasing frequency that lists 1-3 are more sorted. Once lists 1-3 are fully sorted for "long enough", pickers 1-3 will continue to pick from lists 1-3. By long enough, we mean the picker is instructed to keep a list if its pseudo-energy is zero for more than 10 iterations, as long as its pseudo-energy never goes above zero in the future. As this can not be the case with the faulty sorter, we are guaranteed (eventual) convergence.

Figure 8 displays the role of κ in the number of iterations to reach zero approximate pseudo-energy among the agents. An average pseudo-energy of zero means that the average is below 0.5, i.e. at least half of the agents have zero pseudo-energy and one other can have at most a pseudo-energy of 2 or two have at most a pseudo-energy of 1. The line traces out the average of 70 different cases examined for each κ, and the error bar bounds one standard deviation above and below this average. Each of the $70 \times 30 = 2100$ cases considered used a different random initial list. The trend shows the improved rate of convergence, on average, as κ is increased. We hope that an estimator, that tries to construct a more educated guess for the true pseudo-energy from the approximate pseudo-energy will lower the averages for the smaller κ values, keeping in mind the computational trade-off between smaller κ and more work for the estimator.

6 Conclusions and Future Work

We have attempted to put the problem of making software robust to certain kinds of disturbances into a dynamical systems and control framework. Analogs of control theoretic sensors, estimators, and controllers within software systems, and methods for applying feedback to such systems were discussed. We defined several metrics and pseudo-energy functions for potential use in stabilizing and analyzing software processes that perform sorting. Further, the case of a single sorter operating in open and closed-loop was thoroughly examined, and closing the loop was shown to dramatically improve the accuracy of a faulty sorter. Simulation for a network of sorters was also detailed, where approximation and randomization were important components. We plan to further extend the analysis of the closed loop sorter dynamics as well as those of the networked sorters. The utility and construction of metrics and pseudo-energies as used on lists above for more general software systems will also be explored.

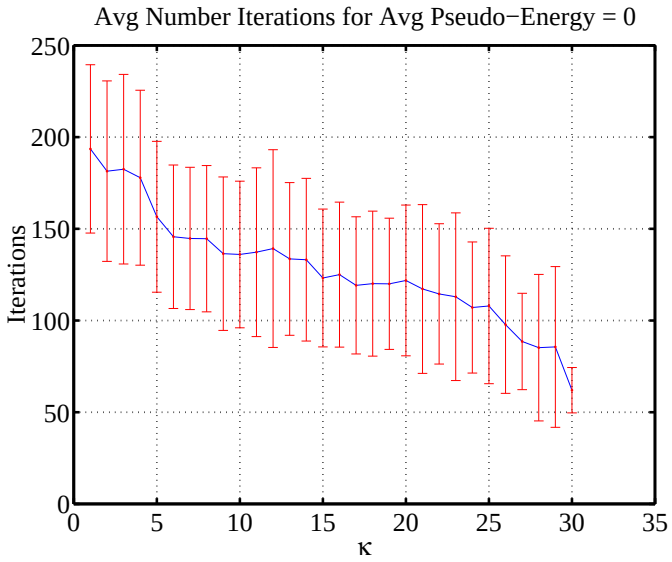


Figure 8: Average Number of iterations to get an average pseudo-energy equal to zero, as κ is varied.

Acknowledgments

The authors with to thank Richard Murray and Jason Hickey for their suggesting several of the problems we consider in this paper and Natarajan Shankar for his advice and suggestions on relating this work to other, similar, fields.

References

[1] Miklos Ajtai, T.S. Jayram, Ravi Kumar, and D. Sivakumar. Approximate counting of inversions in a data stream. In *34th ACM Symposium on Theory of Computing*, Montral, Qubec, Canada, 2002.

[2] A. Avizienis. *The Methodology of N-Version Programming*. John Wiley & Sons, New York, 1995.

[3] M. Blum and S. Kannan. Designing programs that check their work. *Journal of the Association for Computing Machinery*, 42(1):269–291, 1995.

[4] Alexander Bogomolny. Various ways to define a permutation. http://www.cut-the-knot.org/do_you_know/Perm.shtml.

[5] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, November 1974.

[6] G. E. Dullerud and F. Paganini. *A Course in Robust Control Theory: A Convex Approac*. Texts in Applied Mathematics. Springer, New York, 2000.

[7] William Feller. *An Introduction to Probability Theory and Its Applications*. J Wiley and Sons, 1957.

[8] S. Forrest, S. A. Hofmeyr, and A. Somayaji. Computer immunology. *Communications of the ACM*, 40(10):88–96, October 1997.

[9] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[10] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language lustre. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.

[11] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[12] H. A. Khalil. *Nonlinear Systems*. Printice Hall, 2nd edition, 1996.

[13] E. Klavins. A formal language approach to embedded systems modeling and verification. In *International Workshop in Hybrid Systems: Computation and Control*, Prague, Czech Republic, 2003. Submitted for review.

[14] E. Klavins and D. E. Koditschek. Phase regulation of decentralized cyclic robotic systems. *International Journal of Robotics and Automation*, 21(3):257–276, 2002.

[15] Michael L. Littman. CPS130 course notes - sorting(5), Fall 1997. http://www.cs.duke.edu/mlittman.

[16] S. H. Low, F. Paganini, and J. C. Doyle. Internet congestion control. *IEEE Control Systems Magazine*, 22(1):28–43, Feb. 2002.

[17] Ronitt Rubinfeld. *A Mathematical Theory of Self-Checking, Self-Testing and Self-Correcting Programs*. PhD thesis, University of California, Berkeley, 1996.

[18] M. Schneider. Self-stabilzation. *ACM Computing Surveys*, 25(1), March 1993.

[19] M. Seltzer and C. Small. Self-montitoring and self-adapting operating systems. In *Proceedings ot the 1997 Workshop in Hot Topics in Operating Systems*, Chatham, MA, May 1997.

[20] L. Sha. Using simplicity to control complexity. *IEEE Software*, July/August 2001.

[21] Sigurd Skogestad and Iam Postlethwaite. *Multivariable Feedback Control; Analysis and Design*. J Wiley and Sons, 1996.

[22] A. Buntwal Somayaji. *Operating System Stability and Security through Process Homeostasis*. PhD thesis, The University of New Mexico, 2002.

[23] H. Wasserman and M. Blum. Software reliability via run-time result-checking. *Journal of the Association for Computing Machinery*, 44(6):826–849, November 1997.