Are Software Failures Chaotic?

Scott Dick, Cindy Bethel, Abraham Kandel
University of South Florida
Department of Computer Science & Engineering
Tampa, FL, USA
{dick, kandel}@csee.usf.edu

Abstract

We report on an experimental investigation of software reliability data. Our hypothesis in this investigation is that software failures are the result of a fundamentally deterministic process, rather than being realizations of a stochastic process as is commonly assumed. Using the techniques of nonlinear time series analysis, we examine three software reliability datasets for the signatures of deterministic, and possibly chaotic, behavior. In these datasets, we have found firm evidence of deterministic behavior, and hints of chaotic behavior. However, the latter are too limited to permit a definitive conclusion about the presence or absence of chaotic behavior.

1. Introduction

Modern society is totally dependent on computers and the software they run. Software runs our cars, flies our airplanes, and operates our military's weapons. Software systems today are called on to perform a dizzying array of tasks, and have thus become incredibly complex. As of 1995, a large software system might have 10^{20} distinct states [1]; in fact, software systems are the most complex objects know to man, far surpassing the complexity of our own brains (which only have about 10^{10} neurons).

Unfortunately, software systems are also notorious for their unreliability. Just in the last few years, NASA has lost two space probes to software failures, at a cost of hundreds of millions of dollars. The Ariane-5 crash, the Therac-25 radiation therapy machine, the US Navy "smart ship" USS Yorktown... the list of public, embarrassing software failures goes on and on. With life, limb and property now routinely depending on the correct operation of software, software reliability is perhaps the most critical technological challenge of the 21st century. Quite simply, software is at once the *most complex* and *least reliable* technology mankind has yet developed.

The discipline of software reliability engineering has responded to this challenge by extending the traditional techniques of reliability engineering to encompass software systems. This is a difficult undertaking, as software systems have some unique characteristics that differentiate them from all other technological systems. Software reliability models such as [2], [3], [9]-[14], have been developed and tested in various software development projects. Probably the most widely-used models are the Jelinski-Moranda de-eutrophication model [2] and Musa's basic execution time model [3]. Common to all software reliability models is the assumption that the occurrence of software failures is a fundamentally random process, which can be described as either a probability distribution or as a realization of a stochastic process [4].

A software failure is ultimately the result of a mistake in the source code, which we refer to as a *fault*. When an input to the software triggers a fault *and* causes a departure of the program's internal state from its correct value, a software *error* has occurred. If that error propagates all the way to the system output, we then have a software failure, the *observed* departure of a system from its correct behavior [5].

If software failures are ultimately the result of human mistakes, then why use a stochastic model to describe them? After all, no software developer rolls a set of dice and says "Aha! Time to make a mistake!" Human mistakes are irregular in nature, not random. Irregularity is a different form of uncertainty, and should not be modeled by probability theory. Instead, chaos theory and fractal sets are the appropriate techniques. These are now accepted as elements of the soft computing paradigm [6].

We hypothesize that the *fault set* of a program (the set of all inputs that can trigger a software fault) forms a fractal subset of the program's input space. This is an alternative explanation of the unpredictable nature of software failures; rather than being realizations of a stochastic process, software failures are unpredictable because of the peculiar geometry of fractal sets. If this hypothesis is true, then a time series formed from the observed failure times of a software system should

0-7803-7461-4/02/$17.00 © 2002 IEEE

exhibit the signatures of deterministic, and possibly chaotic, behavior. These time series are just software reliability data, and we can use the techniques of nonlinear time series analysis to search for these signatures within them. We have conducted this investigation in three software reliability datasets, and found that all three exhibit deterministic behavior. In addition, there are some indications of chaotic behavior. However, these latter signatures are not strong enough to permit a definitive statement on whether the datasets are chaotic in nature.

The remainder of this paper is organized as follows. In Section 2, we provide a brief overview of reliability theory. In Section 3, we discuss how reliability theory can be applied to software, and how the unique characteristics of software complicate the analysis. In Section 4, we use nonlinear techniques to reconstruct the system state space for each of our three datasets, and to determine if the datasets are stationary. In Section 5, we use statistical hypothesis testing to explore whether the datasets are random or deterministic in nature, and we search for the signatures of chaotic behavior. We offer a conclusion and discussion of future work in Section 6.

2. Reliability Theory

Intuitively, reliability is a characteristic of a technological system that describes how much confidence we have in that system. A system that can be depended on to correctly perform its function all the time is said to be very reliable, while a system that cannot be counted on is said to be unreliable. Formally, reliability is defined as *the probability that a given system, operating in a specified environment, will function correctly during a specified period of time.* The probability density function for this definition is $f(t) = P\{t < T \le t + \Delta t\}$ where the random variable T denotes time, and f(t) is the probability that a failure occurs in the interval [t,t+Δt]. The corresponding CDF is $F(t) = P\{T \le t\}$, or the probability that a failure occurs before time t. Since we are interested in the probability that a failure does *not* occur before time t, we take the negation of the CDF, giving us the reliability function $R(t) = 1 - F(t)$.

In addition to the PDF, CDF and reliability functions, two other quantities are often used to represent the reliability of a system. The *failure rate* is the conditional probability that the system will fail during the time interval [t,t+Δt], given that the system has already survived to time t. This quantity may be expressed as $\lambda(t) = f(t)/R(t)$. The failure rate is also known as the *hazard rate* or *mortality rate*. In general, over the lifetime of a system, the failure rate will follow a bathtub-shaped curve, as in Figure 1. There will be an initial period of frequent *infant mortality* failures, followed by a steady, low rate of failure for the bulk of the system's lifetime, and finally an increasing rate of failures as the system wears out.

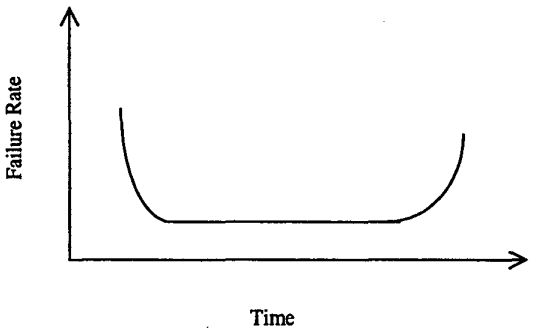

Figure 1: System failures over time

The Mean Time to Failure (MTTF) is the expected time to the next failure, i.e. the expected value of f(t). The MTTF may be determined as

$$MTTF = \int_0^\infty t \cdot f(t) \cdot dt \text{ or } MTTF = \int_0^\infty R(t) \cdot dt .$$

Plainly, by identifying the form and parameters for the distribution of f(t), we also determine F(t), R(t), λ(t) and the MTTF. A number of probability distributions – including the Exponential, Gamma and Weibull distributions – have been successfully used as the distributional form of f(t). Once a distributional form is chosen, the distribution is then fitted to the available data from the system, usually through maximum likelihood estimation. A goodness-of-fit test is then used to check that the estimated distribution does indeed reflect the properties of the system data. Possible choices for this test include the Chi-Square test and the Kolomogorov-Smirnoff test, with the latter considered a more powerful test. Finally, once a satisfactory estimate of f(t) is obtained, estimates of F(t), R(t), λ(t) and the MTTF can be obtained.

Data concerning a system's reliability is obtained through testing. There are actually two forms of reliability testing: reliability growth testing and life testing. In reliability growth testing, we are interested in improving the reliability of a system by revealing and eliminating its failure modes. One copy of a system is run until failure, and the running time until failure is recorded. The cause of the failure is determined and eliminated, and the process is repeated. Over time, this process should remove most

of a system's failure modes, resulting in improved reliability. The data collected in this phase – the operating times until failure for each iteration – are know as reliability growth data.

Life testing is a procedure for estimating the expected lifetime of a system. Multiple copies of a system are run in a homogeneous environment until they fail. The operating life of each copy is recorded, and together the operating life of all the copies is taken to be a random sample from the population of all operating lifetimes for all copies of the system. The expected value of these lifetimes is then estimated from the sample values [7].

3. Software Reliability

Software systems are technological systems, and the reliability of software systems is closely related to general system reliability. However, software has some unique characteristics that complicate any reliability analysis. In this section, we will discuss these difference, and their implications for reliability analysis. We will also provide an overview of software reliability models, which in general are more complex than the models used in general system reliability engineering.

3.1 Software Characteristics

Unlike other technological systems, software is an entirely logical entity, rather than a physical one. Brooks referred to software as "pure thought-stuff" in [8], and this ephemeral nature makes software qualitatively different from other technological systems. Firstly, there is no such thing as aging in software; bits of information do not get older. Thus, there is no wear-out effect. Second, since software is just digital information, is can be perfectly copied. Life testing of software is therefore pointless; each copy of the software is identical to every other copy, and thus the population of all copies of a piece of software has zero variance.

Since software is purely logical, it also much easier to alter. A few keystrokes are all that is needed; no welding torches, no saws, no cutting dies are involved. An unfortunate result (from a reliability engineer's point of view) is the great complexity of software systems. It is not possible to test even a meaningful fraction of the 10^{20} states a software system may have. This means that there can easily be undiscovered failure mechanisms (i.e. bugs) at the end of reliability growth testing. In fact, most software engineers would tell you that having bug-free software is an impossible goal. The sheer complexity of modern software also means that the models used in general systems reliability engineering are wholly inadequate; as we will see in the next section, techniques such as the non-homogeneous Poisson process and Bayesian inference are employed instead of Weibull and Gamma distributions. Despite the flexibility of the latter, they have been found to be too simple to capture the behavior of software undergoing reliability growth testing.

3.2 Software Reliability Models

A wide variety of software reliability models have been proposed in the last 30 years. One of the earliest models to gain widespread use was the Jelinski-Moranda model [2]. In this model, the time between failures is assumed to follow an exponential distribution, parameterized by the number of faults remaining in the software. Beginning in 1975, the Non-Homogeneous Poisson Process (NHPP) was used as the basic framework in a number of models. These include the first NHPP software reliability model, proposed by Goel and Okumoto in [9], and Musa's basic execution time model [3], which is perhaps the most widely applied of all software reliability models. In the NHPP models, the mean value function is parameterized by the number of faults remaining in the software.

Another major trend in software reliability models is the use of Bayesian inference to construct the models. Examples of this latter trend include [10]-[12], among many others. Bayesian inference is the statistician's answer to the irregular nature of software failures, and is a very powerful tool. However, the need to specify some tractable form for the prior distributions of parameters leads to the incorporation of a great many simplifying assumptions. Work in software reliability modeling is ongoing; one recent paper uses a 'quasi-renewal' process as a software reliability model [13]. Another recent paper fits a chaotic function as a reliability model [14]. This latter is the only use of chaos theory in software reliability to date that we are aware of.

4. State-Space Reconstruction

The material in the next two sections closely follows the recommended practices in [15], and uses the implementations of this material in [16] and [17]. We examine three software reliability growth datasets. The first is the dataset "System 5" reported by Musa in [3] and archived at [18]. This dataset is very clean, consisting of 831 failures recorded to the nearest second. The remaining two datasets were generated in the course of IBM's Orthogonal Defect Classification project, and may be found in [5]. ODC1

consists of 1207 failures, recorded to the nearest day, and ODC4 is 2008 failures recorded to the nearest day. We have preprocessed these two datasets by assuming that failures arrive at random, uniformly distributed intervals during a day.

A deterministic system, by definition, follows a continuous trajectory in its native state space. When a time series of observations is taken from a time series, the result is a complex projection of the system state space into a scalar time series. The techniques of nonlinear time series analysis reconstruct a state space that is equivalent to the original by means of a delay embedding (equivalent in this case means the reconstructed space and the original are related by a smooth, invertible mapping). A delay vector $\beta_n = (x_n - (m-1)v, x_n - (m-2)v, \ldots, x_n)$ is formed from m successive elements x_i of the time series. The time lag v means that we might take consecutive elements, every second element, every third element, etc. A delay reconstruction consists of selecting the parameters m and v. There are no unique algorithms for selecting the time lag v; the usual practice is to form two-dimensional delay vectors and graph the resulting phase portrait. We normally choose the value of v that gives the largest possible structures in the phase portrait, since this emphasizes the deterministic patterns in the time series. The embedding dimension m can be found by searching for false nearest neighbors [15].

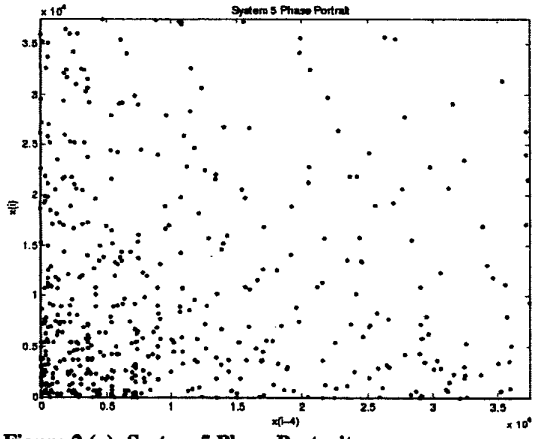

Figure 2 (a): System 5 Phase Portrait

The parameters we have identified for our datasets are as follows: System 5 was reconstructed in 9 embedding dimensions, with a time lag of 4, ODC1 was reconstructed in 9 embedding dimensions with a time lag of 3, and ODC4 was reconstructed in 15 embedding dimensions with a time lag of 5. The two-dimensional phase portraits for each of the datasets are shown in Figure 2(a)-(c), respectively. Note that there are clear trajectories and voids in the phase portrait for System 5, while the presence of structure in ODC1 is very limited. The phase portrait for ODC4 is striking; notice the double-helix structure along the y-axis. Taken together, these phase portraits support our hypothesis of deterministic behavior in software reliability data.

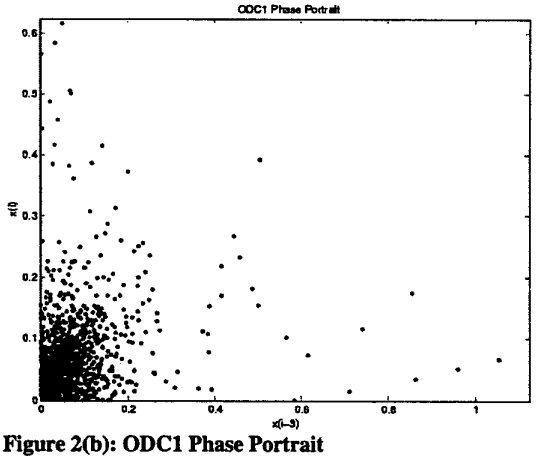

Figure 2(b): ODC1 Phase Portrait

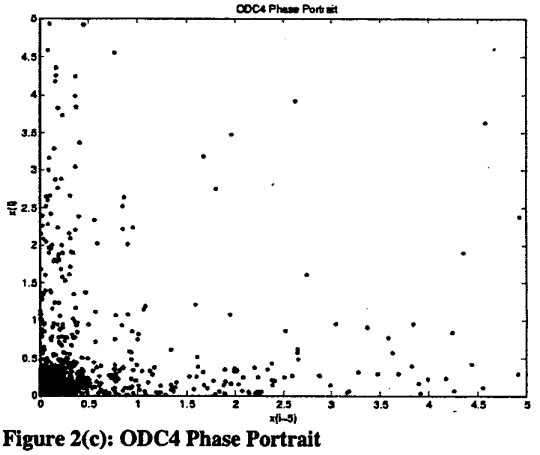

Figure 2(c): ODC4 Phase Portrait

The techniques of nonlinear time series analysis are all based on the assumption that a time series is taken from a stationary process. In a nonlinear setting, linear techniques like first-order autocorrelations are not appropriate tests for stationarity. Instead, we make use of the space-time separation plot. The space-time separation plot allows us to detect nonstationarity by visualizing the internal time scales of a system. If these time scales are of the same order of magnitude

as the time series

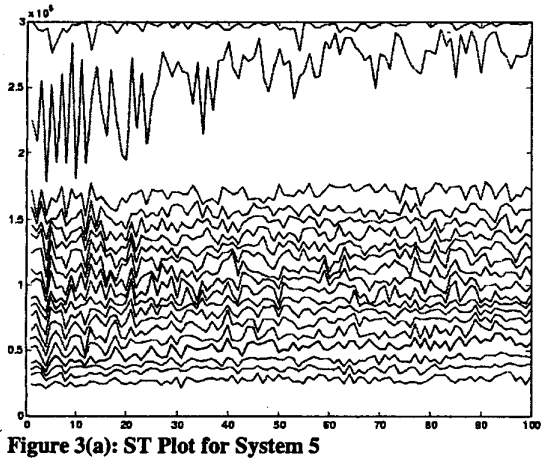

Figure 3(a): ST Plot for System 5

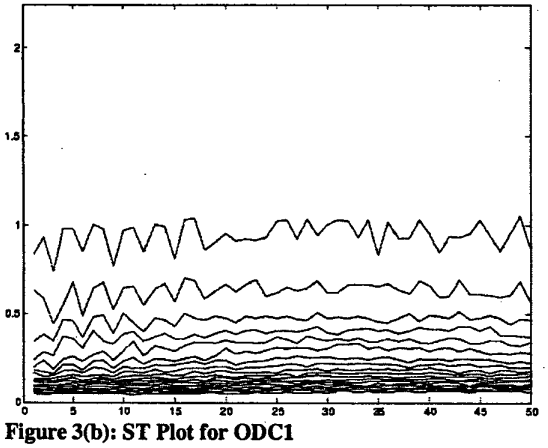

Figure 3(b): ST Plot for ODC1

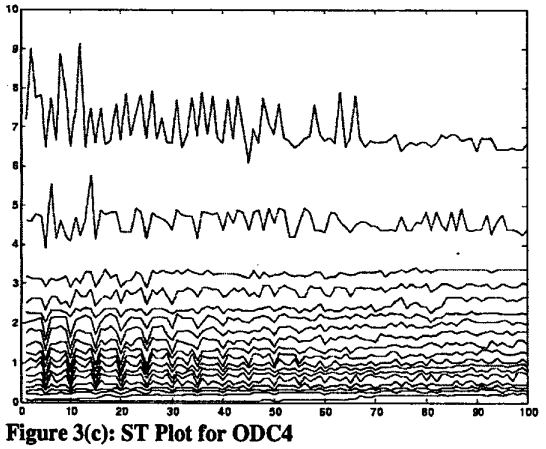

Figure 3(c): ST Plot for ODC4

itself, then the time series cannot be considered stationary. Space time separation plots for System 5, ODC1, and ODC4 are shown in Figure 3(a)-(c), respectively. Since all the contours saturate at an approximate plateau after a relatively small number of time steps in each plot, we can assert that all three datasets appear to be stationary in nature [15]. This may seem a counterintuitive result; after all, the debugging process is constantly altering the software. However, regression testing (the verification that debugging has not introduced new faults) was not included in these datasets. Thus, the test cases that were changed by debugging are not actually revisited in these three datasets, and thus they are effectively stationary.

5. Determinism and Chaos

Testing for the presence is based on the technique of surrogate data. This is a statistical hypothesis test in which the null hypothesis is that the data come from a random process, the alternative is a deterministic process, and the test statistic must quantify some aspect of determinism. Elaborate shuffles of the original dataset are carried out, resulting in datasets that have the same mean and variance, but match the null hypothesis. The test statistic is then applied to the surrogates as well as the original dataset. If the value obtained for the original dataset is greater or less than the values for all surrogates, the null hypothesis is rejected. The number of surrogates involved in the test determines the significance of the test [15].

We have tested all three of our datasets, using two test statistics: the rms error from a nonlinear prediction algorithm (the locally constant algorithm of [15]) and a time-reversal asymmetry statistic. Our null hypothesis is that the datasets arise from a linear Gaussian process, distorted by an invertible, nonlinear observation function. This is the most general hypothesis available in the literature. We have checked if any other probability distribution would be more appropriate, but did not find any distribution that fit any of our datasets. When we performed the surrogate data test, we found that we could reject the null hypothesis at a significance of 95% for all three of our datasets. This result confirms the qualitative evidence of deterministic behavior obtained from inspecting the system phase portraits. This is the main result of this paper.

We have also examined the datasets for the signatures of chaotic behavior. We have attempted to estimate the correlation dimension of each of the three datasets. We found some suggestive behavior in the plot of local slopes for System 5. We are looking for all the curves to plateau and collapse for at least a small range of length scales. As Figure 4 shows, there is a common plateau in a very small scaling region; however, the curves do not collapse together. Neither

ODC1 nor ODC4 showed this kind of behavior; the discretization noise present in those datasets appears to be too large for the correlation dimension algorithm. The evidence from System 5, while suggestive, is not sufficient to conclude that chaotic dynamics are at work in that dataset. We must therefore state that we have some slight indications of chaotic behavior, but no conclusive evidence.

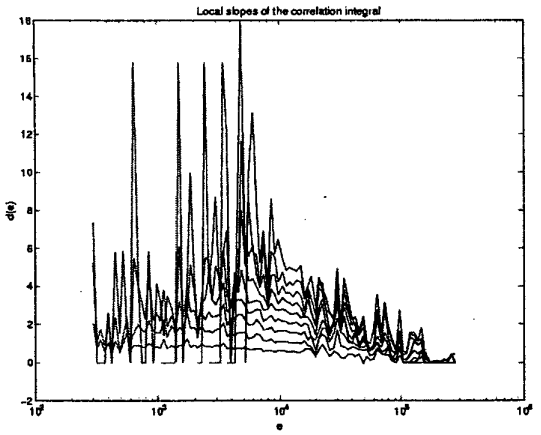

Figure 4: Correlation Integral for System 5

6. Conclusions

In this paper, we have applied the techniques of nonlinear time series analysis to three software reliability growth datasets. Our investigations show strong evidence of deterministic behavior; the main result is a statistical hypothesis test which strongly supports our contention that software failures arise from a deterministic, rather than stochastic, process. Future work in this area will include using deterministic modeling techniques to predict software reliability, and an attempt to predict the location of undiscovered faults in a program based only on the distribution of discovered faults.

Acknowledgement

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada under grant no. PGSB222631-1999 and in part by the National Institute for Systems Test and Productivity at USF under the USA Space and Naval Warfare Systems Command grant no. N00039-01-1-2248.

References

[1]] Friedman, M.A.; Voas, J.M., *Software Assessment: Reliability, Safety, Testability*, New York: John Wiley & Sons, Inc., 1995

[2] Z. Jelinski, P.B. Moranda, "Software reliability research," in *Proceedings of the Statistical Methods for the Evaluation of Computer System Performance*, Academic Press, 1972, pp. 465-484.

[3] J.D. Musa, "Validity of execution-time theory of software reliability," *IEEE Transactions on Reliability*, vol. 28 no. 3, August 1979, pp. 181-191.

[4] A. L. Goel, "Software reliability models: assumptions, limitations, and applicability," *IEEE Transactions on Software Engineering*, vol. 11 no. 12, Dec. 1985, pp. 1411-1423.

[5] M.R. Lyu, Ed., *Handbook of Software Reliability engineering*, New York: McGraw-Hill, 1996.

[6] S.K. Pal, S. Mitra, *Neuro-Fuzzy Pattern Recognition Methods in Soft Computing*, New York: John Wiley & Sons, Inc., 1999.

[7] E.E. Lewis, *Introduction to Reliability Engineering, 2nd Ed.*, New York: John Wiley & Sons, Inc., 1996.

[8] F.P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering Anniversary Edition*, Reading, MA: Addison-Wesley Pub. Co., 1995

[9] A.L. Goel, K. Okumoto, "Time-dependent error-detection rate model for software and other performance measures," *IEEE Transactions on Reliability*, vol. 28 no. 3, Aug. 1979, pp. 206-211.

[10] M. Horigome, N.D. Singpurwalla, R. Soyer, "A Bayes empirical Bayes approach for (software) reliability growth," in *Computer Science and Statistics: Proc. 16th Symp. On the Interface*, Atlanta, GA, USA, March 1984, pp. 47-55.

[11] J. Kyparisis, N.D. Singpurwalla, "Bayesian inference for the Weibull process with applications to assessing software reliability growth and predicting software failures," in *Computer Science and Statistics: Proc. 16th Symp. On the Interface*, Atlanta, GA, USA, March 1984, pp. 57-64.

[12] S.K. Bar-Lev, I. Lavi, B. Reiser, "Bayesian inference for the power law process," *Annals of the Institute of Statistical Mathematics*, vol. 44 no. 4, 1992, pp. 623-629.

[13] H. Pham, H. Wang, "A quasi-renewal process for software reliability and testing costs," *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, vol. 31 no. 6, Nov. 2001, pp. 623-631.

[14] F.-Z. Zou, C.-X. Li, "A chaotic model for software reliability," *Chinese Journal of Computers*, vol. 24 no. 3, 2001, pp. 281-291.

[15] H. Kantz, T. Schreiber, *Nonlinear Time Series Analysis*, New York: Cambridge University Press, 1997.

[16] R. Hegger, H. Kantz, T. Schreiber, "Practical implementation of nonlinear time series methods: the TISEAN package," *CHAOS*, vol. 9 no. 2, 1999, pp. 413-435

[17] T. Schreiber, A. Schmitz, "Surrogate time series," *Physica D*, vol. 142 no. 3-4, 2000, pp. 346-382.

[18] DACS Staff, "DACS Software Reliability Dataset," http://www.dacs.dtic.mil/databases/sled/swrel.shtml, The Data & Analysis Center for Software.