The Monte Carlo Method and Software Reliability Theory

Brian Korver[1]
briank@cs.pdx.edu

TR 94-1. February 18, 1994

1.0 Abstract

The Monte Carlo method of reliability prediction is useful when system complexity makes the formulation of exact models essentially impossible. The characteristics of the Monte Carlo method make it ideal for estimating the reliability of software systems. Unlike many other mathematical models, system complexity is irrelevant to the method. Not only can the structure of the system be dynamic, but the precise structure of the software system need not even be known. Instead, system components need only be tested for failure during operation, which ensures that components which are used more often contribute proportionally more to the overall reliability estimate. Combined with self-checking algorithms which respond to randomly-generated inputs, the method obviates the need for valid, nontrivial input data and an external oracle.

2.0 The Monte Carlo Method

The Monte Carlo method of estimating integrals is probably the best-known Monte Carlo technique. Instead of attempting to solve integrals analytically, the method estimates an integral by firing random points at the function. The law of large numbers predicts that as more random points are chosen, the ratio of points below to points above the function will approximate the ratio of the area beneath the function to total area of the sample space from which the random points are drawn. For instance, in order to solve the integral for the unit normal distribution,

$$\int_{-2}^{2} e^{-\left(\frac{x^2}{2}\right)} dx \qquad \textbf{(EQ 1)}$$

random points (x,y) are chosen from the range $-2 < x < 2$, $0 \le y \le 1$. As can be seen from Figure 1, darts thrown randomly at that range can be expected to fall below the function

$$f(x) = e^{-\left(\frac{x^2}{2}\right)} \qquad \textbf{(EQ 2)}$$

proportional to the area that lies beneath this function. Such factors as the complexity of the function and the number of variables are essentially irrelevant. Since the Monte Carlo method does not attempt to solve the integral analytically, the function need not be known and, in fact, can be in the form of data since the method only needs to

1. Work supported by NSF grant CCR 9110111. I would like to thank Dick Hamlet for his considerable insight and patience in helping me to put together this tech report.

be informed of whether a given point falls above or below the function.

3.0 Crude Monte Carlo

The most general version of the Monte Carlo method of reliability prediction is based on the "structure function" which says that the state of a system is the product of the states of its components (Kaufmann, Grouchko, & Cruon, 1977, p. 56; Melchers, 1987, p. 91). Specifically, if

$$x_i = \begin{cases} 1 & \text{if component } e_i \text{ is in a good state,} \\ 0 & \text{otherwise} \end{cases}$$

then the state of the system y is

$$y = \prod_{i=1}^{n} x_i \qquad \text{(EQ 3)}$$

for components in series, and

$$y = 1 - \prod_{i=1}^{n} (1 - x_i) \qquad \text{(EQ 4)}$$

for components in parallel. Since system behavior is based on the behavior of system components, the possible error in the resulting reliability estimate is reduced (Kamat & Riley, 1975, p. 73).

While "crude"—or "direct"—Monte Carlo is statistically sound and easy to understand, it can be a computationally expensive technique. Since most systems are highly reliable, crude Monte Carlo simulations require large sample sizes to obtain a sufficient quantity of failures to provide reliable estimates. Moreover, since the standard error of the final result is inversely proportional to the square root of the sample size n, the sample size must be increased k^2-fold to reduce the standard error by a factor of k (Hammersley & Handscomb, 1964, pp. 21-22). Thus, many improvements—called variance reduction techniques—have been developed to reduce the required sample sizes. According to Hammersley and Handsomb,

> The so-called variance-reducing techniques, which lie at the heart of good Monte Carlo work, are techniques which reduce the coefficient of $1/n$ in the sampling variance of the final estimator, where n is the sample size (or, perhaps more generally, the amount of labour expended on the calculation).

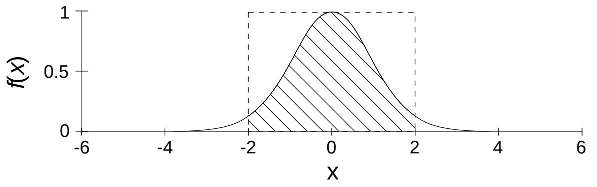
For instance, Kumamoto, Tanaka, Inoue, and Henley describe a variance reduction technique that increased the efficiency of a Monte Carlo simulation by 934-fold (1980, p. 379). However, for the sake of simplicity and without loss of generality, this paper describes the application of crude Monte Carlo to the reliability simulation of software systems.

4.0 Reliability Simulation

Because the reliability of each component is based on probability distributions, the reliability of each component in a system flow chart can be modelled by a set of random numbers. For instance, if the reliability of a component is 0.8, then successful operation of that component can be represented by the numbers from 0.0 through 0.79 and failure by the numbers from 0.8 through 0.99 (Amstadter, 1971, p. 176). By generating random numbers as the system flow chart is traced, it is possible to simulate the state of each component. These component states can then be combined using the structure function to determine the state of the system. Since "each execution of a simulation tells only whether a particular set of conditions did or did not" exist, the Monte Carlo method is an experimental problem-solving technique such that "many simulation runs have to be made to understand the relationships involved in the system" (Gordon, 1978, pp. 42, 43). Each repetition of the simulation results in another independent estimate of the reliability of the system. As the number of simulations increases, the sample mean of these independent estimates approaches the actual characteristics of the system (Amstadter, 1971, p. 176). According to Verma, Fu, and Moses,

> Monte Carlo methods can be used to good advantage since the required probability of failure is formulated as a multi-dimensional integral of the probability density function of the basic variables over the sys-

FIGURE 1. Monte Carlo Integration



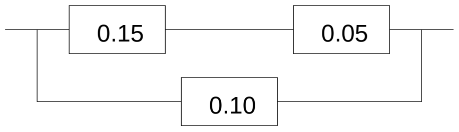
tem failure domain (which may not have a simple boundary). (1989, p. 895)

That is, the Monte Carlo method computes the statistical expectation of the reliability distributions (Zaremba, 1968, p. 304). Thus, because estimates of system reliability are based on probability distributions functions representing the failure rates of components, the Monte Carlo method will accurately simulate system reliability whatever the complexity of those distributions, even in the case that they are entirely empirical.

5.0 System Reliability Estimation

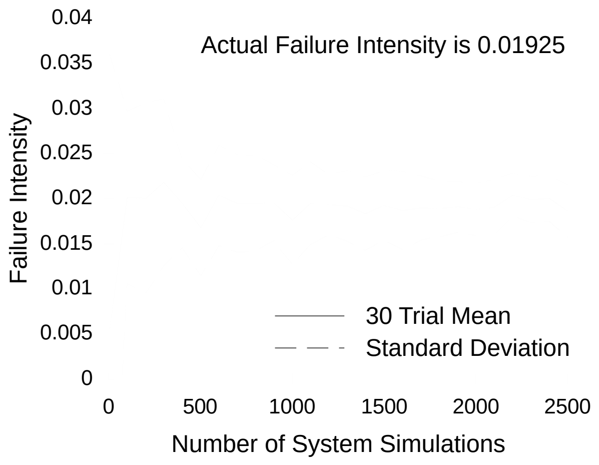
Consider the arbitrary, trivial three-component system with the failure intensities shown in Figure 2. Listing 1 demonstrates the use of Monte Carlo methods to simulate the behavior of this system and thus can be used to estimate its reliability. The failure of one of the two components that form the series subsystem causes the failure of the entire subsystem. A given iteration of the subsystem is simulated by testing the state of the first component of the subsystem by generating a random number in the range [0,1]. If the first component does not fail, then the state of the second component is tested. If the series subsystem fails, then the parallel component is tested. If it too fails, then the entire system fails. After a number of these system simulations have been performed, the failure intensity of the system is calculated by dividing the number of observed failures by the total number of simulations. The actual failure intensity for this system is 0.01925.[1] The results from actual simulation runs of Listing 1 are given in Figure 3. As the number of system simulations increases, the simulation results converage to the actual mean.

FIGURE 2. Three-Component System



1. [0.15 + 0.05 - (0.05)(0.15)](0.10) = 0.01925

FIGURE 3. Results from Listing 1



6.0 MTBF Estimation

Listing 2 demonstrates the use of Monte Carlo methods to estimate the mean time between failures (MTBF) of the system shown in Figure 2 by counting the average amount of time (measured here in iterations) between each failure. Given sample size n with t_1, t_2, ..., t_n being the number of successful iterations between failures,

$$\mathrm{MTBF} = \frac{1}{n}\sum_{i=1}^{n} t_i \qquad \textbf{(EQ 5)}$$

(Shooman, 1983, pp. 570-571). As each run of successes until a failure occurs constitutes a geometric experiment, the actual MTBF of the system in Figure 2 is 51.948.[2] The results from actual simulation runs of Listing 2 are given in Figure 4. Because each sample in a MTBF estimate consists of the number of Bernoulli trials until a failure occurs, the variance of the results is less than that of the single Bernoulli trial simulated failure intensity.

7.0 Self-Checking Algorithms

As mentioned previously, the reliability of each system component is modelled using probability distributions.

2. 1/(0.01925) = 51.948

FIGURE 4. Results from Listing 2

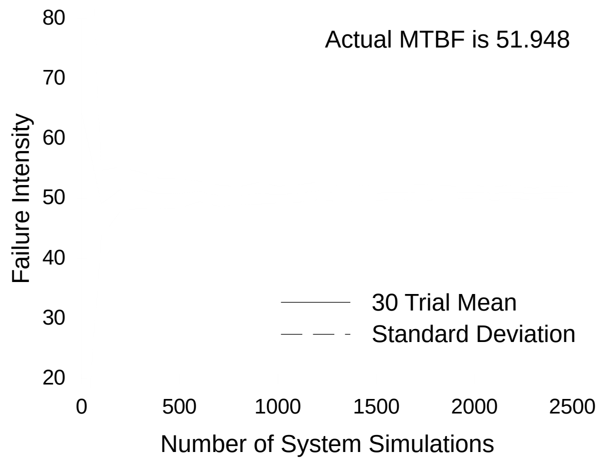


TABLE 1. Listing 3 Components

Component	Description
1	Generate a system of linear equations
2	Check if the system has been generated randomly
3	Reduce the matrix to echelon form using Gaussian elimination
4	Check if the matrix is in echelon form
5	Backsolve to obtain the roots
6	Check if the roots are a solution to the system of linear equations

Although the reliability characteristics of the components in software systems are rarely known, it may be possible to write self-checking algorithms which are able to report the status of their behavior (Blum & Kannan, 1989; Blum, Luby, & Rubinfeld, 1990). According to Blum and Kannan, many of these checkers are simpler than the programs they check (p. 86). Program checkers are concerned with the task of verifying that a given program returns a correct answer on a given input rather than all inputs (p. 87). Therefore, like the Monte Carlo method, self-checking algorithms only report the state of the component for a given set of conditions. Listing 3 demonstrates the use of Monte Carlo methods with self-checking algorithms to estimate the reliability of a program which generates a random matrix and then solves it as system of linear equations. Instead of using random numbers to model the states of components, checking functions are employed to report the actual behavior of each component on a given input.

8.0 Software System Composition

Table 1 descriptions the system components of Listing 3. The structure of Listing 3 is given in Figure 5. While these mirror the primary functions of Listing 3, this is coincidental and irrelevant to the method. According to Parnas,

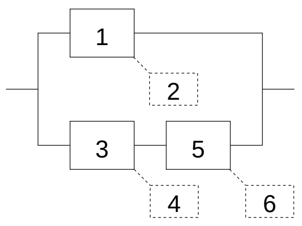
> the way that the program is divided into subprograms can be rather arbitrary. For *any* program, some decompositions into subprograms may reveal a hierarchical structure, while other decompositions may show a graph with loops in it. (1974, p. 336)

In a software system, two components are structurally in parallel if the failure of one is not influenced by the failure of the other. Conversely, two components are structurally in series if failures are propagated from one to the other. That is, two components are structurally in series of one "uses" the other:

> The relation "uses" may be defined by USES(p_i,p_j) = *iff* p_i calls p_j *and* p_i will be considered incorrect if p_j does not function properly.[3] (Parnas, 1974, p. 336)

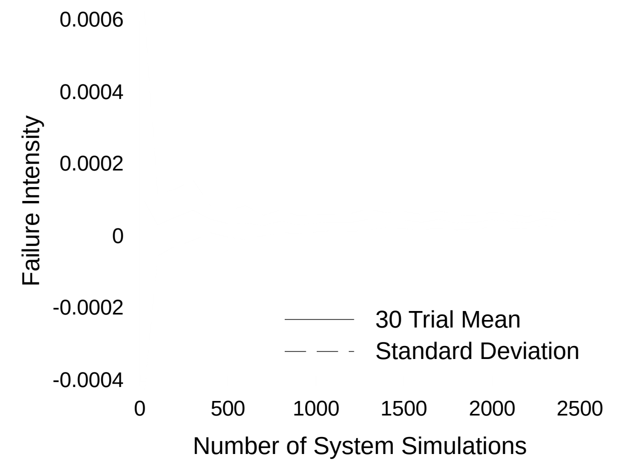
However, because the structure of a software system is implicitly expressed in the code itself, software systems need not be explicitly decomposed into series and parallel subsystems. Instead, self-checking algorithms must be placed at the very least at the junctions of parallel components, when control passes between components that are not related by "uses." In Listing 3, a failure in the Gaussian elimination function will be uncovered when the roots are checked in the original system of linear equations. Thus, for a rough estimate of system reliability, it may not be necessary to verify that the reduced matrix is in echelon

FIGURE 5. Decomposition of Listing 3



3. The requirement that p_i call p_j has been ignored.

FIGURE 6. Results from Listing 3



form. However, to increase accuracy, all component failures should be included in failure rate calculations. For serial components, it is important to note that simple system composition calculations would result in counting some component failures more than once since a failure by component p_j causes component p_i to fail. Thus, the failure intensity of component p_i is computed by factoring out those errors caused by component p_j.[4]

The technique employed in Listing 3 can be used to determine if components not related by "uses" are effecting each other. By conducting a number of one-run simulations, the number of times each component fails independently and the number of times they fail together can be counted. Whether one is causing the other to fail can be determined by the standard test of independence.[5]

Although several arbitrary constraints have been coded into Listing 3—for instance, only generating linear systems of twenty variables or less—these constraints provide an analogue to the requisite operational profile. As Hamlet states, "unless random tests are drawn as the software will be actually used, the tests are not a representative sample and all statistical bets are off" (1993, p. 4).

> Encountering an error (ie executing an error-embedded instruction) does not necessarily cause a failure; it merely provides an opportunity, the degree of which depends on the probability of simultaneously processsing failure-inducing input. (Trachteberg, 1990, p. 93)

4. $\text{Actual}_i = \text{Reported}_i - \text{Reported}_j + (\text{Reported}_i * \text{Reported}_j)$
5. $P(EF) = P(E)P(F)$

That is, if you are trying to empirically determine the probability that a king will come up in a poker deck, a pinochle deck will not suffice. More importantly, these arbitrary limits facilitated the random testing of the individual components which was necessary in order to compare the analytically computed failure intensity with the results of the Monte Carlo simulation. The results of these tests demonstrate that Monte Carlo simulation techniques match the analytically computed failure intensities. The results from actual simulation runs of Listing 3 are given in Figure 6. However, these results could not be compared to the "actual" failure intensity since there is no reliable method of analytically calculating the failure intensity of software systems.

Similarly, "bugs" which were triggered by calls to a random number generator were inserted into each of the components. For 500 system simulations, the results of the analytically-computed failure intensity and the results of the Monte Carlo reliability simulation were relatively close. Example results from these tests are shown in Table 2.

TABLE 2. Monte Carlo Simulation Tests

Component Failure Intensity			System Failure Intensity		
1	**3**	**5**	**Actual**	**Simulated**	**% Error**
0.25	0.10	0.05	0.03625	0.03041	16.11
0.03	0.10	0.05	0.00435	0.00459	5.52
0.03	0.20	0.05	0.00720	0.00837	16.25
0.03	0.20	0.10	0.00840	0.00726	13.57

It is important to note that failures in the checking components will either increase or decrease the reported system failure intensity, depending upon whether it is more likely for the checking routine to incorrectly report, respectively, "failure" or "success." See Table 3 for an example of when a checking routine fails.

TABLE 3. Example Checking Routine Failures

Actual Failure Intensity	Checker Failure Intensity	Checker Incorrectly Reports	Reported Failure Intensity
0.20	0.05	failure	0.24
0.20	0.05	success	0.19
0.20	0.05	50/50 either	0.215

9.0 Conclusion

Although these examples are trivial, the complexity of a dynamic software system is essentially irrelevant to the Monte Carlo method of reliability prediction thus making the method ideal for estimating the reliability of software systems. Most importantly, the results of the Monte Carlo method of reliability prediction match those obtained by analytical methods.

10.0 References and Bibliography

Amstadter, B.L. (1971). *Reliability mathematics: Fundamentals; practices; procedures*. New York: McGraw-Hill Book Company.

Babbit, A., Powell, S.T., & Hamlet, D. (1990). Prototype testing tools. In *Proceedings of the 9th Annual Pacific Northwest Software Quality Conference* (pp. 264-280). Portland, OR : [The Conference].

Barlow, R.E., & Proschan, F. (1965). *Mathematical theory of reliability.* New York: John Wiley & Sons.

Binder, K., & Heermann, D.W. (1988). *Monte Carlo simulation in statistical physics: An introduction.* New York: Springer-Verlag.

Birnbaum, Z.W. (1955). On a use of the Mann-Whitney statistic. *Proceedings of the Third Berkeley Symposium on Mathematical Statistics and Reliability* (pp. 13-17). Berkeley and Los Angeles: University of California Press.

Birolini, A. (1985). *On the use of stochastic processes in modeling reliability problems.* New York: Springer-Verlag.

Blum, M., & Kannan, S. (1989). Designing programs that check their work. *Proceedings of the 21st Annual ACM Symposium on Theory of Computing* (pp. 86-97). New York: Association for Computing Machinery.

Blum, M., Luby, M., & Rubinfeld, R. (1990). Self-testing/correcting with applications to numerical problems. *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing* (pp. 73-83). New York: Association for Computing Machinery.

Bratley, P., Fox, B.L., & Schrage, L.E. (1993). *A guide to simulation.* New York: Springer-Verlag.

Fluendy, M. (1970). Monte Carlo methods. In G.G. Lowry (Ed.), *Markov chains and Monte Carlo calculations in polymer science* (pp. 46-90). New York: Marcel Dekker, Inc.

Gordon, G. (1978). *System simulation* (2nd ed.). Englewood Cliffs, NJ: Prentice-Hall, Inc.

Hammersley, J.M., & Handscomb, D.C. (1964). *Monte Carlo methods.* New York: John Wiley & Sons, Inc.

Hamlet, D. (1992, July). Are we testing for true reliability? *IEEE Software*, pp. 21-27.

Hamlet, D. (1993). *Random testing* (PSU TR 93-10). Portland, OR: Portland State University Computer Science Department.

Hamlet, D., & Voas, J. (1993). Faults on its sleeve: Amplifying software reliability testing. In *Proceedings of the International Symposium on Software Testing and Analysis* (pp. 89-98). Cambridge, MA: [The Conference].

Kamat, S.J., & Riley, M.W. (1975). Determination of reliability using event-based Monte Carlo simulation. *IEEE Transactions on Reliability, R-24*(1), pp. 73-75.

Kaufmann, A., Grouchko, D., & Cruon, R. (1977). *Mathematical models for the study of the reliability of systems.* New York: Academic Press.

Kumamoto, H. Tanaka, K., Inoue, K., & Henley, E. (1980). State-transition Monte Carlo for evaluating large, repairable systems. *IEEE Transactions on Reliability, R-29*(5), pp. 376-380.

Kleijnen, J.P.C. (1974). *Statistical techniques in simulation* (Part 2). New York: Marcel Dekker, Inc.

Levy, L.L, & Moore, A.H. (1967). A Monte Carlo technique for obtaining system reliability confidence limits from component test data. *Transactions on Reliability, R-16*(2), pp. 69-72.

Mann, N.R., Schafer, R.E., & Singpurwalla, N.D. (1974). *Methods for statistical analysis of reliability and life data.* New York: John Wiley & Sons.

Melchers, R.E. (1987). *Structural reliability: Analysis and prediction.* New York: John Wiley and Sons.

Miyakawa, M. (1984). On stochastic coherent systems. In M. Beckman & W. Krelle (Eds.), *Stochastic models in reliability theory* (pp. 1-11). New York: Springer-Verlag.

Moore, A.H., Harter, H.L., & Snead, R.C. (1980). Comparison of Monte Carlo techniques for obtaining system-reliability confidence limits. *IEEE Transactions of Reliability, R-29*(4), pp. 327-331.

Parnas, D. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15*(12), pp. 1053-1058.

Parnas, D. (1974). On a ‘buzzword’: Hierarchical structure. In J.L. Rosenfeld (Ed.), *Proceedings of IFIP Congress 74* (pp. 336-339). New York: North-Holland Publishing Company.

Ross, S. (1988). *A first course in probability* (3rd ed.). New York: Macmillan Publishing Company.

Shannon, R.E. (1975). *Systems simulation: The art and science.* Englewood Cliffs, NJ: Prentice-Hall, Inc.

Shooman, M.L. (1983). *Software engineering: Design, reliability, and management.* New York: McGraw-Hill Book Company.

Sobol, I.M. (1974). *The Monte Carlo method* (R. Messer, J. Stone, & P. Fortini, Trans.). Chicago: University of Chicago Press.

Trachtenberg, M. (1990). A general theory of software-reliability modeling. *IEEE Transactions on Reliability, 39*(1), pp. 92-96.

Verma, D., Fu, G., & Moses, F. (1989). Efficient structural system reliability assessment by Monte-Carlo methods. In A.H-S. Ang, M. Shinozuka, & G.I. Schueller (Eds.), *Structural Safety & Reliability: Proceedings of ICOSSAR ‘89, the 5th International Conference on Structural Safety and Reliability* (pp. 895-901). New York: American Society of Civil Engineers.

Walpole, R.E., & Myers, R.H. (1989). *Probability and statistics for engineers and scientists* (4th ed.). New York: Macmillan Publishing Company.

Zaremba, S.K. (1968). The mathematical basis of Monte Carlo and quasi-Monte Carlo methods. *SIAM Review 10*(3), pp. 303-314.

LISTING 1. Estimate of Reliability

```
enum states { FAILURE = 0, SUCCESS = 1 };


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  FUNCTION:   system_test
 *
 *  AUTHOR:     Brian Korver
 *  PURPOSE:    Perform a Bernoulli trial, computing the state of the three-component system
 *              using the structure function.
 *
 *                   #2                #3
 *                /------\          /------\
 *           -----| 0.15 |----------| 0.05 |-----
 *             |  \------/    #1    \------/  |
 *             |          /------\            |
 *             ------------| 0.10 |------------
 *                        \------/
 *
 *  ARGUMENTS:  -
 *  RETURN:     0 = FAILURE, 1 = SUCCESS
 *  INPUT:      -
 *  OUTPUT:     -
 *  EXIT CODE:  -
 *  CALLS:      drand48()
 */
int
system_test()
{
        if (    (drand48() >= 0.90) &&          /* if component 1 fails and   */
               ((drand48() >= 0.85) ||          /* either component 2 or      */
                (drand48() >= 0.95)) )          /* component 3 fails,         */
                return FAILURE;                 /* then the system fails      */

        return SUCCESS;                         /* otherwise, the system doesn't fail */
}



/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  FUNCTION:   estimate_reliability
 *
 *  AUTHOR:     Brian Korver
 *  PURPOSE:    Conduct a Binomial experiment of a number of independent Bernoulli trials.
 *              Count the number of system failures.  Compute the failure intensity by
 *              dividing the number of failures by the number of Bernoulli trials.
 *  ARGUMENTS:  number_of_trials -- the number of bernoulli trials to conduct
 *  RETURN:     -
 *  INPUT:      -
 *  OUTPUT:     the number of failed Bernoulli trials, the number of trials conducted, and
 *              the computed failure intensity.
 *  EXIT CODE:  -
 *  CALLS:      seed(), system_test() (the system to test), printf()
 */
void
estimate_reliability (int number_of_trials)
{
        int failures = 0;                       /* Bernoulli trials that failed */
        int trials_conducted;                   /* for loop index */
        float failure_intensity;                /* reliability estimate */

        seed();                                 /* seed random number generator */

        for (   trials_conducted = 0;
                trials_conducted < number_of_trials;
                ++trials_conducted
        )       if (system_test() == FAILURE) ++failures;

        failure_intensity = (float) failures / number_of_trials;

        printf("%d\t%d\t%f\n", failures, number_of_trials, failure_intensity );
}



/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  FUNCTION:   main (listing1)
 *
 *  AUTHOR:     Brian Korver
 *  PURPOSE:    estimate the reliability of a simple system
 *  ARGUMENTS:  argv[1] -- the number of system simulations to run
 *  RETURN:     -
 *  INPUT:      -
 *  OUTPUT:     -
 *  EXIT CODE:  -
 *  CALLS:      atoi(), estimate_reliability()
 */
void
main(int argc, char *argv[])
{
        if (argc == 2) estimate_reliability(atoi(argv[1]));
}
```

LISTING 2. Estimate of Mean Time Between Failures

```
enum states { FAILURE = 0, SUCCESS = 1 };


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  FUNCTION:   system_test
 *
 *  AUTHOR:     Brian Korver
 *  PURPOSE:    Perform a Bernoulli trial, computing the state of the three-component system
 *              using the structure function.
 *
 *                   #2                #3
 *                /------\          /------\
 *           -----| 0.15 |----------| 0.05 |-----
 *             |  \------/    #1    \------/  |
 *             |           /------\           |
 *             ------------| 0.10 |------------
 *                         \------/
 *
 *  ARGUMENTS:  -
 *  RETURN:     0 = FAILURE, 1 = SUCCESS
 *  INPUT:      -
 *  OUTPUT:     -
 *  EXIT CODE:  -
 *  CALLS:      drand48()
 */
int
system_test()
{
        if (    (drand48() >= 0.90) &&          /* if component 1 fails and   */
               ((drand48() >= 0.85) ||          /* either component 2 or      */
                (drand48() >= 0.95)) )          /* component 3 fails,         */
                return FAILURE;                 /* then the system fails      */

        return SUCCESS;                         /* otherwise, the system doesn't fail */
}



/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  FUNCTION:   estimate_MTBF
 *
 *  AUTHOR:     Brian Korver
 *  PURPOSE:    Conduct a number of Geometric experiments, which consist of conducting
 *              Bernoulli trials until the first system failure occurs.  Compute the MTBF--
 *             in number of iterations--by computing the mean number of iterations between
 *             failures.
 *  ARGUMENTS:  number_of_experiments -- the number of geometric experiments to conduct
 *  RETURN:     -
 *  INPUT:      -
 *  OUTPUT:     the number of successful Bernoulli trials, the number of experiments
 *             conducted, and the computed MTBF
 *  EXIT CODE:  -
 *  CALLS:      seed(), system_test() (the system to test), printf()
 */
void
estimate_MTBF (int number_of_experiments)
{
        int successful_iterations = 0;          /* Bernoulli trials that succeeded */
        int experiments_conducted;              /* for loop index */
        float MTBF;                             /* mean time between failures estimate */

        seed();                                 /* seed random number generator */

        for (   experiments_conducted = 0;
                experiments_conducted < number_of_experiments;
                ++experiments_conducted
        )       while (system_test() == SUCCESS) ++successful_iterations;

        MTBF = (float) successful_iterations / number_of_experiments;

        printf("%d\t%d\t%f\n", successful_iterations, number_of_experiments, MTBF);
}



/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  FUNCTION:   main (listing2)
 *
 *  AUTHOR:     Brian Korver
 *  PURPOSE:    estimate the MTBF of a system
 *  ARGUMENTS:  argv[1] -- the number of system simulations to run
 *  RETURN:     -
 *  INPUT:      -
 *  OUTPUT:     -
 *  EXIT CODE:  -
 *  CALLS:      atoi(), estimate_MTBF()
 */
void
main(int argc, char *argv[])
{
        if (argc == 2) estimate_MTBF(atoi(argv[1]));
}
```

LISTING 3. Estimate Reliability of Solving a System of Linear Equations

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 * Function prototypes for a matrix ADT, the code is not included in this listing
 *
matrix *initmatrix(int rows, int cols);
void delmatrix(matrix *m);
matrix *dupmatrix(matrix *m);
int ncols(const matrix *m);
int nrows(const matrix *m);
element getel(const matrix *m, int row, int col);
void putel(element el, matrix *m, int row, int col);




/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  FUNCTION:   parallel
 *
 *  AUTHOR:     Brian Korver
 *  PURPOSE:    to compute the failure intensity of two parallel components given the failure
 *              intensity of the two components.  The parallel subsystem fails if both
 *              components fail.
 *  ARGUMENTS:  c1, c2 -- the failure intensity of two components which are in parallel
 *  RETURN:     the parallel failure intensity of the two components
 *  INPUT:      -
 *  OUTPUT:     -
 *  EXIT CODE:  -
 *  CALLS:      -
 */
float
parallel(float c1, float c2)
{
        return (c1 * c2);
}




/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  FUNCTION:   serial
 *
 *  AUTHOR:     Brian Korver
 *  PURPOSE:    to compute the failure intensity of two serial components given the failure
 *              intensity of the two components.  The series subsystem fails when one of the
 *             two components fail.
 *
 *             Note: the second argument *must* be the second component in the series
 *             subsystem.  Since a failure by c1 causes a failure in c2, failures would be
 *             overestimated if all of the c2 failures were counted in addition to the c1
 *             failures.  Thus, the failure intensity of c2 is computed as:
 *
 *                     c2  <-  c2  -  c1  +  (c1)(c2)
 *
 *             while the failure intensity of the series subsystem is is computed as:
 *
 *                     c1  +  c2  -  (c1)(c2)
 *
 *             This function combines both of these computations.
 *  ARGUMENTS:  c1, c2 -- the failure intensity of two components which are in series
 *  RETURN:     the series failure intensity of the two components
 *  INPUT:      -
 *  OUTPUT:     -
 *  EXIT CODE:  -
 *  CALLS:      -
 */
float
serial(float c1, float c2)
{
        return (c1 * c1 * (1 - c2) + c2);
}



/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  FUNCTION:   randomize
 *
 *  PURPOSE:    fill a matrix with random numbers
 *  ARGUMENTS:  m -- a pointer to the matrix
 *  RETURN:     -
 *  INPUT:      -
 *  OUTPUT:     -
 *  EXIT CODE:  -
 *  CALLS:      mrand48()
 */
void
randomize(matrix *m)
{

        register element rnumber;
        register int c, r;
        for (r = m->rows; r >= 1; --r) {
                for (c = 1; c <= m->cols; ++c) {
                        rnumber = (element) (mrand48() >> 16);
                        putel(rnumber,m,r,c);
                }
        }
}




/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  FUNCTION:   report_failure
 *
 *  AUTHOR:     Brian Korver
 *  PURPOSE:    for “reporting” that a failure occured.  Reports to the file pointer OUT
 *              which is defined globally.
 *  ARGUMENTS:  component -- the number of the failured component
 *  RETURN:     -
 *  INPUT:      -
 *  OUTPUT:     prints the component number (in ascii) to the file attached to the global file
 *              pointer OUT
 *  EXIT CODE:  -
 *  CALLS:      fprintf()
 */
void
report_failure(int component)
{
        extern FILE *OUT;
        fprintf(OUT,”%d\n”,component);
}




/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  FUNCTION:   seed
 *
 *  PURPOSE:    seed random number generator with system time times the process id
 *  ARGUMENTS:  -
 *  RETURN:     -
 *  INPUT:      -
 *  OUTPUT:     -
 *  EXIT CODE:  -
 *  CALLS:      lcong48(), seed48(), srand48(), ftime(), getpid()
 */
void
seed()
```

```
{
        extern pid_t getppid();
        extern ftime();
        long s;
        struct timeb tp;

        (void) ftime(&tp);
        s = (long) tp.millitm * getpid();

        lcong48((unsigned short *) &s);
        (void) seed48((unsigned short *) &s);
        srand48((long) s);

}




/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  FUNCTION:   generate_system_of_equations
 *
 *  AUTHOR:     Brian Korver
 *  PURPOSE:    generate a system of linear equations in a random n x n+1 matrix.  The
 *              matrix size is being arbitrarily limited to 0 < n < n_max.
 *  ARGUMENTS:  n_max -- the maximum value for n
 *  RETURN:     a random matrix
 *  INPUT:      -
 *  OUTPUT:     -
 *  EXIT CODE:  -
 *  CALLS:      mrand48(), abs(), initmatrix(), randomize()
 */
matrix *
generate_system_of_equations(int n_max)
{
        extern long int mrand48();
        register int size;
        register matrix *m;

        size = (abs(mrand48()) % n_max) + 1;

        m = initmatrix(size,size + 1);

        randomize(m);

        return(m);
}
```

```
const element VARIANCE_THRESHOLD = 2.0E+8;              /* arbitrary value that the */
                                                        /* variance of the elements must */
                                                        /* be above */

/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  FUNCTION:   matrix_is_random
 *
 *  AUTHOR:     Brian Korver
 *  PURPOSE:    check to see if the matrix being passed qualifies as a random matrix
 *  ARGUMENTS:  m -- a pointer to a matrix
 *  RETURN:     1 if the matrix is random, 0 otherwise
 *  INPUT:      -
 *  OUTPUT:     -
 *  EXIT CODE:  -
 *  CALLS:      nrows(), ncols(), fabs(), getel()
 */
int
matrix_is_random(const matrix *m)
{
        register int r, c;                              /* row, column loop index */
        register long double mean, variance;
        register int size;

        if (!m) return(0);                              /* matrix doesn't exist */

        if ((nrows(m) + 1) != ncols(m)) return(0);      /* invalid size */

        mean = 0;                                       /* compute the mean of the elements */
        size = (nrows(m) * ncols(m));
        for (r = 1; r <= nrows(m); ++r) {
                for (c = 1; c <= ncols(m); ++c) {
                        mean += getel(m,r,c) / size;
                }
        }
        variance = 0;                                   /* compute the variance */
        --size;
        for (r = 1; r <= nrows(m); ++r) {
                for (c = 1; c <= ncols(m); ++c) {
                        variance += (getel(m,r,c) - mean) * ((getel(m,r,c) - mean) / size);
                }
        }

        if (variance <= VARIANCE_THRESHOLD) {           /* not random enough */
                return(0);
        }

        return(1);                                      /* the matrix is random */
```

```
}



/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  FUNCTION:   gaussian_elimination
 *
 *  AUTHOR:     Brian Korver
 *  PURPOSE:    reduce a matrix to echelon form using gaussian elimination (see pp. 330-333
 *              of Burden, R.L., & Faires, J.D. (1993), _Numerical Analysis_ (5th Ed.).
 *              Boston: PWS Publishing Company or for a description of this algorithm )
 *  ARGUMENTS:  orig -- a pointer to a n x n+1 matrix containing a system of linear equations
 *  RETURN:     a pointer to the reduced echelon matrix, or 0 if no unique solution exists
 *  INPUT:      -
 *  OUTPUT:     -
 *  EXIT CODE:  -
 *  CALLS:      dupmatrix(), initmatrix(), ncols(), nrows(), getel(), putel(), fabs(),
 *              delmatrix()
 */
matrix *
gaussian_elimination(const matrix *orig)
{
        register matrix *m, *A;
        register int i, j, p, n, k;
        int NROW[n];
        register element NCOPY;

        if (!orig) return (0);                                 /* matrix doesn't exist */

        n =  nrows(orig);

        if ( (n + 1) != ncols(orig) ) return(0);               /* wrong size matrix */

        A = dupmatrix(orig);
        m = initmatrix(nrows(A),ncols(A));

        for (i = 1; i <= n; ++i) NROW[i] = i;

        for (i = 1; i < n; ++i) {
                p = i;
                for (j = i + 1; j <= n; ++j) {
                        if (fabs(getel(A,NROW[p],i)) < fabs(getel(A,NROW[j],i)))
                                p = j;
                }

                if (getel(A,NROW[p],i) == 0) {  /* no unique solution exists */
                        delmatrix(A);
                        delmatrix(m);
                        return(0);
                }

                if (NROW[i] != NROW[p]) {
                        NCOPY = NROW[i];
                        NROW[i] = NROW[p];
                        NROW[p] = NCOPY;
                }

                for (j = i + 1; j <= n; ++j) {
                        putel( (getel(A,NROW[j],i) / getel(A,NROW[i],i)), m,NROW[j],i);

                        for (k = 1;  k <= n + 1; ++k) {
                                putel( (getel(A,NROW[j],k) -
                                                (getel(m,NROW[j],i) * getel(A,NROW[i],k))),
                                        A,NROW[j],k);
                        }
                }
        }

        if (getel(A,NROW[n],n) == 0) {          /* no unique solution exists */
                        delmatrix(A);
                        delmatrix(m);
                        return(0);
        }

        for (i = 1; i <= n; ++i) {
                for (j = 1; j <= n + 1; ++j) {
                        putel ( getel(A,NROW[i],j), m, i, j);
                }
        }

        delmatrix(A);
        return(m);
}



const element ZERO = 1.0e-9;                    /* arbitrary limit below which a number is 0 */


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  FUNCTION:   matrix_is_reduced
 *
 *  AUTHOR:     Brian Korver
 *  PURPOSE:    check to determine if the passed matrix e is in echelon form
 *  ARGUMENTS:  m -- a pointer to a matrix
 *             e -- a pointer to the matrix in echelon form
 *  RETURN:     0 if the matrix is not in echelon form, 1 otherwise
 *  INPUT:      -
 *  OUTPUT:     -
 *  EXIT CODE:  -
 *  CALLS:      getel(), ncols(), nrows()
 */
int
matrix_is_reduced(const matrix *m, const matrix *e)
{
        register int r, c;                                  /* loop indices */
        register int n;                                     /* column pointer */

        if (!m && !e) return (1);                           /* no matrix */

        if (!m || !e) return (0);                           /* one is missing */

        n = 0;
        for (r = 1; r <= nrows(e); ++r) {                   /* loop through rows & cols */
                c = 1;
                while ((c <= ncols(e)) && (fabs(getel(e,r,c)) < ZERO))
                                                            /* look for 1st non-0 in row */
                        ++c;

                if ((c <= n) && (c <= ncols(e))) return(0);     /* not in echelon form */
                n = c;
        }

        return(1);                                          /* in echelon form */
}



/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  FUNCTION:   roots_from_backsubstitution
 *
 *  AUTHOR:     Brian Korver
 *  PURPOSE:    get root from backsubstituting using an echelon matrix
 *  ARGUMENTS:  A -- a pointer to a matrix in echelon form
 *  RETURN:     a pointer to a n x 1 vector (matrix) containing the roots [x1,x2,...,xN]
 *              or a null pointer if the vector A is not valid
 *  INPUT:      -
 *  OUTPUT:     -
 *  EXIT CODE:  -
 *  CALLS:      nrows(), initmatrix(), getel(), putel()
 */
matrix *
```

```
roots_from_backsubstitution(const matrix *A)
{

        register element ncopy;
        register int i, j, n;
        register matrix *roots;

        if (A == 0) return (0);
        n =  nrows(A);
        roots = initmatrix(nrows(A),1);

        putel((getel(A,n,n+1) / getel(A,n,n)),roots,n,1);

        for (i = n-1; i >= 1; --i) {
                ncopy = 0;
                for (j = i+1; j <= n; ++j)
                        ncopy += getel(A,i,j) * getel(roots,j,1);
                putel( ((getel(A,i,n+1) - ncopy) / getel(A,i,i)),roots,i,1);
        }

        return(roots);
}



const double SOLUTION_THRESHOLD = 1.0E-9;


/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  FUNCTION:   roots_are_solution
 *
 *  AUTHOR:     Brian Korver
 *  PURPOSE:    substitute roots into a system of linear equations to see if they are
 *              actually the solution to the system [given the tolerance SOLUTION_THRESHOLD].
 *  ARGUMENTS:  M -- a pointer to a system of linear equations
 *              R -- a pointer to a matrix containing the roots [x1,x2,...,xN]
 *  RETURN:     0 if any of the equations are off by more than 1.0E-9, otherwise 1 (success)
 *  INPUT:      -
 *  OUTPUT:     -
 *  EXIT CODE:  -
 *  CALLS:      nrows(), ncols(), getel(), fabs()
 */
int
roots_are_solution(const matrix *M, const matrix *R)
{
        register int r, c;
        register element sum;
        register int rows, cols;
        if (M == 0 && R == 0) return(1);                /* no matrix means no roots */
        else if (M == 0 || R == 0) return(0);           /* else, either missing == failure */

        rows = nrows(M);
        cols = ncols(M);

        for (r = 1; r <= rows; ++r) {
                sum = 0;
                for (c = 1; c < cols; ++c) {
                        sum += (getel(M,r,c) * getel(R,c,1));
                }

                if (fabs((element) sum - getel(M,r,cols)) > SOLUTION_THRESHOLD) return(0);
                                                        /* the roots were off by too much */
        }

        return 1;                                       /* the roots are the solution */
}



/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  FUNCTION:   solve_linear_system
 *
 *  AUTHOR:     Brian Korver
 *  PURPOSE:    solve a system of linear equations
 *  ARGUMENTS:  m -- a pointer to a n x n+1 matrix to treat as the system of linear equations
 *  RETURN:     a pointer to a n x 1 vector (matrix) containing the roots [x1,x2,...,xN]
 *              or a null pointer if no solution is possible (for whatever reason)
 *  INPUT:      -
 *  OUTPUT:     -
 *  EXIT CODE:  -
 *  CALLS:      gaussian_elimination(), matrix_is_reduced(), report_failure(),
 *              roots_from_backsubstitution(), roots_are_solution()
 */
matrix *
solve_linear_system(const matrix *m)
{
        matrix *echelon_matrix, *roots;

        echelon_matrix = gaussian_elimination(m);

        if (!matrix_is_reduced(m,echelon_matrix))               /* elimination failed */
                report_failure(3);

        if (echelon_matrix == 0) {                              /* ignore when no unique */
```

```
                                                                /* solution exists */
                return ((matrix *) 0);                          /* successful termination */
        }

        roots = roots_from_backsubstitution(echelon_matrix);    /* compute roots */

        if (!roots_are_solution(m,roots))                       /* backsolving failed */
                report_failure(5);

        return(roots);

}




/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  FUNCTION:   fgetd
 *
 *  AUTHOR:     Brian Korver
 *  PURPOSE:    like return(atoi(fgets(...,iop))) except it breaks on any
 *              non-numeric (provided by isdigit())
 *              See K&R II p. 165 for the inspiration for this function
 *  ARGUMENTS:  iop -- the file pointer to read from
 *  RETURN:     the number if a number was read, otherwise EOF (from stdio.h)
 *  INPUT:      reads from iop
 *  OUTPUT:     -
 *  EXIT CODE:  -
 *  CALLS:      getc(), isdigit(), atoi()
 */
int
fgetd(FILE *iop)
{
        register int c, n = (sizeof(int) * sizeof(int) + 2);
        register char *cs;
        char s[(sizeof(int) * sizeof(int) + 2)];
        int r;

        cs = s;

        while (--n && isdigit(c = getc(iop)))
                *cs++ = (char) c;

        *cs = '\0';

        if (cs == s) {
                r = EOF;
        } else {
                r = atoi(s);
        }

        return r;
}




/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  FUNCTION:   linear_system_software
 *
 *  AUTHOR:     Brian Korver
 *  PURPOSE:    solve a randomly-generated system of linear equations while checking for
 *              software system failures
 *  ARGUMENTS:  error_report -- a file (or pipe) to report where errors occured
 *  RETURN:     -
 *  INPUT:      -
 *  OUTPUT:     -
 *  EXIT CODE:  0 (success) as long as both subsystems didn’t fail, otherwise 1 (failure)
 *              the error
 *  CALLS:      seed(), generate_system_of_equations(), matrix_is_random(),
 *              solve_linear_system()
 */
void
linear_system_software()
{
        matrix *random_matrix, *roots;
        int generation_failure, solution_failure;

        seed();                                                 /* random number generator */

        random_matrix = generate_system_of_equations(20);       /* limit size to 20 x 21 */

        if (!matrix_is_random(random_matrix))                   /* generation failed */
                report_failure(1);

        roots = solve_linear_system(random_matrix);             /* solve the system */

        exit(0);                                                /* successful termination */

}



#define COMPONENTS 6                                    /* number of system components */

FILE *IN, *OUT;                                         /* ends of an interprocess pipe */
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  FUNCTION:   estimate_software_reliability
 *
 *  AUTHOR:     Brian Korver
 *  PURPOSE:    Conduct an experiment of a number of independent trials.  Count the number of
 *              failures attributed to each component.  Compute the system failure intensity by
 *              treating components related by USES() as serial components, and those not
 *              related by USES() as parallel components.
 *  ARGUMENTS:  number_of_trials -- the number of independent trials to conduct
 *  RETURN:     -
 *  INPUT:      -
 *  OUTPUT:     the number of failures attributed to each component, the number of trials
 *              conducted, and the computed system failure intensity.
 *  EXIT CODE:  -
 *  CALLS:      pipe(), fdopen(), fork(), fclose(), linear_system_software(),
 *              wait(), feof(), fgetd(), printf(), parallel(), serial()
 */
void
estimate_software_reliability(int number_of_trials)
{
        register int failure = 0;                     /* Bernoulli trials that failed */
        register int trials_conducted;                /* for loop index */
        register int component_number;                /* failed component_number */
        float system_failure_intensity;               /* reliability estimate */
        int component_failures[COMPONENTS + 1];       /* component failure counts */
        float failure_intensity[COMPONENTS + 1];      /* component failure intensities */
        int statusp;                                  /* status of child process */
        int fd[2];                                    /* pipe file descriptors */

        for (component_number = 0; component_number <= COMPONENTS; ++component_number)
                component_failures[component_number] = 0;
                                                      /* initialize # of failures to 0 */

        for (   trials_conducted = 0;
                trials_conducted < number_of_trials;
                ++trials_conducted      ) {

                pipe(fd);                             /* create a pipe */

                IN = fdopen(fd[0],"r");               /* grab the ends of the pipe */
                OUT = fdopen(fd[1],"w");

                if (fork() == 0) {                    /* fork a child process */
                                                      /* if we are that child process */
                        linear_system_software();     /* conduct a test */
                        exit(9);                      /* (should never reach this point) */
                }
                fclose(OUT);                            /* don’t need this end */

                wait(&statusp);                         /* wait until the child is finished */

                while (!feof(IN) && ((component_number = fgetd(IN)) != EOF))
                        ++component_failures[component_number];
                                                        /* count the failures by each */
                                                        /* component */

                fclose(IN);                             /* close the pipe */

                if (statusp) ++component_failures[0];   /* if exit code not 0 or a signal */
                                                        /* stopped the child */
        }

        for (component_number = 0; component_number <= COMPONENTS; ++component_number) {
                failure_intensity[component_number]
                        = (float) component_failures[component_number]
                        / number_of_trials;             /* compute the failure intensity of */
                                                        /* each component */
                printf(“%f\t”, failure_intensity[component_number]);
        }

        system_failure_intensity =
                parallel(failure_intensity[1],serial(failure_intensity[3],failure_intensity[5]));

        system_failure_intensity = serial(system_failure_intensity,failure_intensity[0]);
                                                        /* include signal failures and */
                                                        /* failures in which exit() was used. */
                                                        /* Actually, these should be trapped */
                                                        /* and traced to their source. */

        printf(“%d\t%f\n”, number_of_trials, system_failure_intensity );
}




/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *
 *  FUNCTION:   main (listing3)
 *
 *  AUTHOR:     Brian Korver
 *  PURPOSE:    estimate the reliability of a software system
 *  ARGUMENTS:  argv[1] -- the number of system simulations to run
 *  RETURN:     -
 *  INPUT:      -
 *  OUTPUT:     -
 *  EXIT CODE:  -
 *  CALLS:      atoi(), estimate_software_reliability()
 */
void
main(int argc, char *argv[])
{
        if (argc == 2) estimate_software_reliability(atoi(argv[1]));
}
```