10/3/20/91

2/25/91

SANDIA REPORT

SAND90—2260 • UC—261
Unlimited Release
Printed January 1991

Programmer's Guide for LIFE2's Rainflow Counting Algorithm

L. L. Schluter

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550
for the United States Department of Energy
under Contract DE-AC04-76DP00789

DO NOT MICROFILM COVER

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
Office of Scientific and Technical Information
PO Box 62
Oak Ridge, TN 37831

Prices available from (615) 576-8401, FTS 626-8401

Available to the public from
National Technical Information Service
US Department of Commerce
5285 Port Royal Rd
Springfield, VA 22161

NTIS price codes
Printed copy: A03
Microfiche copy: A01

DO NOT MICROFILM
COVER

SAND--90-2260

DE91 009182

PROGRAMMER'S GUIDE FOR LIFE2'S RAINFLOW COUNTING ALGORITHM*

by

L. L. Schluter

Wind Energy Research Division
Sandia National Laboratories
Albuquerque, NM 87185

ABSTRACT

The LIFE2 computer code is a fatigue/fracture analysis code that is specialized to the analysis of wind turbine components. The numerical formulation of the code uses a series of cycle count matrices to describe the cyclic stress states imposed upon the turbine. In this formulation, each stress cycle is counted or "binsed" according to the magnitude of its mean stress and alternating stress components and by the operating condition of the turbine. A set of numerical algorithms has been incorporated into the LIFE2 code. These algorithms determine the cycle count matrices for a turbine component using stress-time histories of the imposed stress states. This paper describes the design decisions that were made and explains the implementation of these algorithms using Fortran 77.

---

*This work is supported by the U.S. Department of Energy at Sandia National Laboratories under contract DE-AC04-76DP00789.

MASTER

ACKNOWLEDGEMENTS

The author wishes to thank H. J. Sutherland, P. S. Veers, and D. P. Burwinkle for their help in implementing and checking these algorithms.

TABLE OF CONTENTS

	page
ACKNOWLEDGMENTS	4
INTRODUCTION	6
BACKGROUND INFORMATION	7
Pre-Count Algorithms	7
Rainflow Counting Algorithms	7
Post-Count Algorithms	8
DESIGN DECISIONS	9
Nonstandard Fortran 77	9
Rainflow Algorithm Coding	10
Use of Constants	10
Temporary Files	10
CODE ORGANIZATION	11
CODE WALKTHROUGH	13
Main Driving Routines	13
Rainfl	13
New_File	13
Old_File	14
Insert_Data	15
LIFE2 Support Routines	17
Extreme_Values	17
Init_Matrix	17
Input_Cycle_Count	18
Input_Header	18
Input_Speed_Range	18
List_Matrices	18
Make_Note	19
Matrix_Information	19
Move_Stress_Arrays	20
Read_Matrices	21
Read_Note	21
Save_Header_Info	21
Setup_Stress_Arrays	21
Update_Matrix_File	22
Rainflow Counting Routines	23
Rain_Count	23
Peaks	23
Rtrack	24
Srain	27
SUMMARY	28
REFERENCES	29

INTRODUCTION

The LIFE2 computer code is a fatigue/fracture analysis code specifically designed for the analysis of wind turbine components (1,2). It is a PC-compatible Fortran code that is written in a top-down modular format with a "user friendly" interactive interface. In this numerical formulation, an "S-n" fatigue analysis is used to describe the initiation, growth and coalescence of micro-cracks into macro-cracks. A linear, "da/dn" fracture analysis is used to describe the growth of a macro-crack.

In the LIFE2 formulation, the cyclic stresses imposed on the turbine component are characterized by the magnitude of their mean stress and alternating stress components and by the operating condition of the turbine. A set of numerical algorithms (3) that permits the code to analyze stress-time histories of component stress states has been incorporated into the code. This paper describes the design decisions that were made in implementing the algorithms using Fortran 77. Also included is an explanation of the organization of the code and a description of the function of each subroutine.

This paper is intended for programmers who wish to understand how the rainflow algorithms have been incorporated into the LIFE2 code. A detailed explanation of the use of the algorithms is presented in Schluter and Sutherland (4).

BACKGROUND INFORMATION

The prime algorithm used to count the number of cycles in the time series data is a rainflow counting algorithm (5). This algorithm defines a stress cycle to be a closed stress/strain hysteresis loop. The algorithm determines the mean and alternating stress level for each stress cycle in the histogram. Pre-count and post-count algorithms support the rainflow counter. The pre-count algorithms prepare the time series data for the rainflow counting algorithm. The rainflow counting algorithm counts the cycles in the time series and stores the mean and cyclic values of each cycle in a file. The post-count algorithms insert the cycles into a cycle count matrix that is compatible with the LIFE2 code.

The following discussion gives a brief description of the algorithms. Schluter and Sutherland (4) has a more complete discussion of these algorithms.

Pre-Count Algorithms

The initial set of algorithms prepares the full time series data for counting by selecting peaks and valleys and discarding "small" stress cycles. The pre-count algorithms reduce the data record to a sequential list of peaks and valleys. This list is stored in a temporary file for processing by the count algorithm.

Peak-Valley Selection. The first algorithm identifies peaks and valleys in the data record by scanning for changes in the sign of the slope. When a change in slope is detected the algorithm will fit a parabola through the three nearest points to estimate the peak or valley found.

Filter. A "race track" filtering algorithm has been incorporated into the pre-processing algorithms to eliminate "small" stress cycles (6). In the technique used, the operator sets a "threshold" value for the algorithm. When the absolute value of the difference between the maximum and minimum values of a stress cycle is greater than the threshold, the algorithm retains that cycle. When the difference is less than the threshold, the cycle is discarded.

Rainflow Counting Algorithm

The rainflow counting algorithm (5) counts the number of closed stress/strain hysteresis loops in the data. It determines the mean and the peak-to-peak alternating stress level (i.e., the range) for each stress cycle in

the histogram. These stress levels are stored in a temporary file for post processing. To speed operation, the algorithm uses "one-pass" through the data to count the stress cycles; i.e., the peak-valley data are read only once during processing by the count algorithm.

Post-Count Algorithms

The final algorithms map each stress cycle into a cycle count matrix that can be processed by the LIFE2 code. The algorithms sort the stress cycle data into bins that are functions of mean stress and alternating stress levels.

The cycle counts from a data record may be used to create a new cycle count matrix, or they may be added to an existing cycle count matrix, at the discretion of the operator.

DESIGN DECISIONS

Nonstandard Fortran 77

There were two major design decisions in the coding of the rainflow algorithms that deviate from the decisions made when coding the main body of the LIFE2 program. The first decision was to replace common blocks with 'INCLUDE' statements. This makes the code shorter and easier to read and update. The second decision was to increase variable names from a maximum of six characters to a maximum of 31 characters. The intent of this decision was to increase readability and make coding and updating easier.

The 'INCLUDE' statement is as a metacommand in the Microsoft Fortran Optimizing Compiler (7). This statement directs the compiler to proceed as though the 'INCLUDE' statement were replaced by a specified file. The syntax for the command is as follows:

$INCLUDE:'*filename*'

The argument *filename* is the name of a file that contains the common block just as the common block would appear in the program.

One major effect of the INCLUDE statement is to decrease the overall size of the code by replacing a multiline common block with a single line statement. This also increases the readability of the code. The second effect that the statement has is to make code modification and updating easier. If a change or addition in the common block is needed, only the file containing the common block needs to be changed. Without the INCLUDE statement, the common block will need to be changed wherever it appears in the code.

By increasing the variable size from a maximum of six characters to a maximum of 31 characters the coding of algorithms becomes much easier. This change also increases the readability of the code to a great extent, which in turn aids in updating the code at a later date. The maximum of 31 characters is based on the Microsoft Fortran Optimizing Compiler. In version 4.1 this feature must be enabled by using the NOTRUNCATE metacommand. In version 5.0, and later, this becomes the default replacing the standard six-character maximum.

These decisions were not made in the original LIFE2 code because both of these coding features are not part of the standard Fortran 77 programming language. However, most modern Fortran compilers include them. If a compiler is to be used that does not support these features, the code must be changed. The INCLUDE statements must be replaced by the actual common blocks. The variable names in each subroutine are listed in the

header of the subroutine so that they can be found and changed easily. While the maximum of 31 characters is specified in the Microsoft Compiler, the actual maximum length used is 20 characters so a compiler that supports a smaller maximum may be used.

Rainflow Algorithm Coding

The original coding of the rainflow algorithms used arrays to store the time series data as the cycles were counted. This method proved to be a limiting factor when working with large time series files. Not only was the amount of data fixed by the size of the array, but large arrays also used large amounts of memory. To overcome this deficiency, the current code reads the time series data in from disc only as it is needed. When data points need to be stored an output file is created to store them. A disadvantage of this approach is that speed of the program is slowed by reading and writing to the disc. Also, the operator must ensure that there is ample room on the disc for these temporary output files.

Use of Constants

Throughout the program, constants are used instead of numbers (i.e., instead of using the number 5 for the screen output specifier, the constant *screen* is set equal to 5 and used). This aids in the readability of the program. The value of the constants can be found in the files stdio.dat and files.dat.

Temporary Files

Temporary files are used frequently to reduce the amount of memory that the program requires. For example, the variable *note* is used in several places. The header information uses *note* to store the title of the data set in. Each wind speed matrix uses *note* to store its description (i.e., number of records in the matrix and the wind speed range). There are also notes that may be added to the end of each data file. Instead of having several note variables, each of which would be 72 bytes, one variable is used. When the operator inputs information into the variable note, it is written to a temporary file for storage, freeing up the variable to be used again.

CODE ORGANIZATION

In the remainder of this document subroutine names are written in **bold letters** and variable names are written in *italics*.

Figure 1 shows a hierarchy diagram of the rainflow subroutines. The main driving routine is called **Rainfl**. This routine links the rainflow subroutines with LIFE2. Since LIFE2 uses the six character limit for the size of its variables, this name is restricted to six characters.

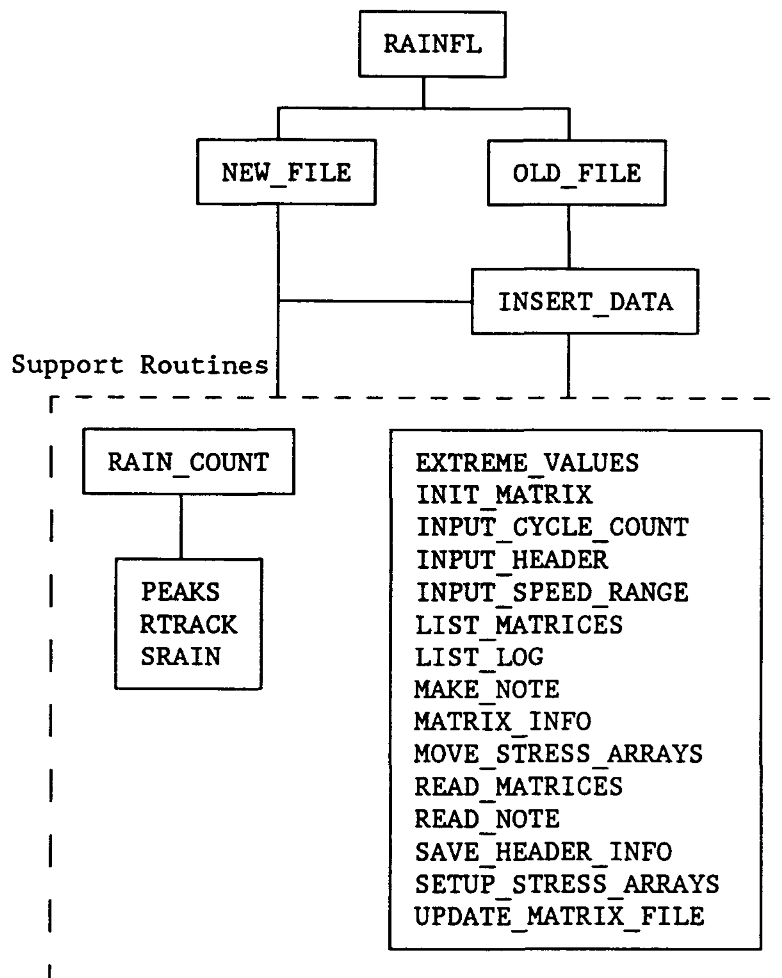


Figure 1 **Hierarchy of Rainflow Subroutines**

Rainfl will then call either **New_File** or **Old_File** depending on whether the time series is inserted into a new data file or an existing data file. **New_File** will call the support routines directly to create the matrix for the first time series. Once created, the file is treated as an existing data file and **Insert Data** is used to insert additional time series. **Old_File** is called by **Rainfl** if the data file initially exists. **Old_File** will call **Insert_Data** to insert the time series.

There are two groups of support routines. One group implements the pre-count and rainflow counting algorithms. These routines are needed to accomplish the rainflow counting. When given a time series data file of stresses, these routines would produce an output file listing the mean and range values for each cycle found in the time series. **Rain_Count** is a driver routine that calls **Peaks** (to find the maxima in the time series), **Rtrack** (to do the racetrack filtering), and **Srain** (implements the single pass rainflow counting algorithm). The other group of subroutines is used to implement the algorithms to create and manipulate the data file that is compatible with LIFE2.

CODE WALKTHROUGH

The following section describes the function of each subroutine used to implement the algorithms. The main driving routines are discussed first, then the support routines that create and manipulate the data file, followed by the pre-count and rainflow counting routines. The source code for all of the routines may be found in the file rain.for on the distribution discs.

Main Driving Routines

Rainfl. This subroutine displays a menu that gives the operator the option of putting the time series into an existing data file or creating a new data file. Subroutine **New_File** is used if a new file is being created, and subroutine **Old_File** is used if an existing file is being used.

New_File. This subroutine creates a new data file in which the cycle count will be placed. First the operator must input the header information for the data file. The program prompts for this information in subroutine **Input Header_Info**. Next, if the stresses are operational or buffeting, the program calls **Input_Speed_Range** to get the lower and upper speed bounds for the time series data. If the stresses are start/stop, then there is no wind speed associated with the matrix. Since this is a new data file, this will be a new matrix. As such, the number of records in the matrix will be 1, so the variable *num_records* is set accordingly, and the number of miscellaneous notes placed at the end of the data file (variable *nnotes*) is initialized to 0.

A note is then created to identify this wind matrix. The note contains the number of records contained in the matrix, the lower wind speed and the upper wind speed. Subroutine **Make_Note** creates the note.

The rainflow counting algorithms are then used by calling subroutine **Rain Count**. On return from this subroutine the variable *opstim* will contain the length of the time series data in seconds if operating or buffeting stresses are being used. If the time series data contained start/stop stresses, then the value of *opstim* will be 1 since this will be the first start/stop record in the matrix.

The matrix can now be set up and filled with the cycle counts found in the time series data. Subroutine **Setup_Stress_Arrays** is used to initialize the mean stress array (*opsm*) and the cyclic stress array (*opsc*). Then **Init Matrix** is used to set all of the count values to zero. Finally, **Input_Cycle**

Count will put the cycle counts found in the time series data into the matrix in their proper locations. In memory, the matrix is stored in the two-dimensional variable *opscc*.

At this point all of the variables contain the required information. The next step is to store this information in the calculational file. This file will be OPS.CAL, BUF.CAL, or STS.CAL depending on the type of data contained in the time series data. The calculational file is opened along with the file containing the header information. The header information is copied to the calculational file using **Copyf**. The file containing the header information is then deleted since it is no longer needed. The matrix information is now written to the calculational file using **Writmx**.

The operator may now add additional time series to this data file. If the operator chooses to do so, the calculational file is rewound and then treated as if it were and old file and **Insert_Data** is called. The variable *modified* is used by **Old_File** and is not applicable in **New_File**. If the operator does not want to put additional time series into the file, the program will ask if any notes are to be added to the data file. Once the notes are entered, the operator has the option of storing the data file in the library for retrieval at a later date. The routine then returns control to **Rainfl**.

Old_File. Subroutine **Old_File** will add time series data to an existing data file. The operator is asked if the current calculational file is the desired data file in which a time series is to be inserted. If not, the data files that are available are listed and the one desired becomes the current calculational file. **Insert_Data** is used to perform the rainflow counting on the time series and to insert the counted cycles into the desired data file. Once all desired time series have been added to the data file, control is returned to **Old_File**.

It is possible to go into an existing data file and examine the matrices without modifying the data file. In this situation, the variable *modified* will be set equal to false when **Insert_Data** returns. This then indicates that the notes should not be modified, so the routine does not prompt the operator for notes.

If there are currently no notes in the data file, the operator is asked if notes are desired. These notes are added to the data file. If notes already exist the operator has three options: 1) leave the existing notes as they are; 2) add to the existing notes; or 3) delete the existing notes and create new notes. If notes already exist then they will be contained in a file called mnote.tmp. With option 1 this file is simply copied to the end of the data file. With option 2 the variable *nnote* (it contains the number of notes in the data file) is increased by the specified number. Then the new notes are added to the file mnote.tmp before this file is copied to the data file. With option 3 the new number of notes is put into *nnote* and then the file

mnote.tmp is rewound. When the new notes are written to this file, the old notes are simply written over. The note file is then copied to the data file.

The code then asks the operator if the data file is to be stored in a library before returning control to **Rainfl**.

Insert_Data. Subroutine **Insert_Data** will insert counted cycles into an existing data file. The counted cycles can be inserted into a matrix that currently exists, or a new matrix may be created to insert the data.

The variable *modified* is used to indicate that the data file has been modified. This information is used by **Old_File** to decide if it should prompt for changing the notes. The assumption is that if the file has not been changed then the notes do not need to be changed. Variable *modified* is set to false initially and changed to true when the file is modified.

Insert_Data must save the data file's header information. This is done with subroutine **Save_Header_Info**. The number of matrices that currently exist in the data file is stored in the variable *nops*. This value is read in from the data file.

Two temporary files are now opened. The first is called matrix.tmp and is used to hold the notes that describe each matrix. This is done so that the notes can be displayed allowing the operator to see what matrices currently exist. The second is called windsp.tmp and it is used to hold the upper wind speed value associated with each matrix. This creates a numerical list of the current matrices within the data file. When the operator creates a new matrix, the program scans this list to find the proper place to insert the new matrix. This way the matrices can be kept in ascending wind speed order. **Readmx** is used to read a matrix. Then the note and the upper wind speed are written to their respective temporary file. This is done for each wind speed matrix within the data file. The existing notes are then read and put into a temporary file called mnote.tmp.

List_Matrices is used to display the matrices that currently exist within the data file. When **List_Matrices** returns, variable *matrix_num* will contain the number of the matrices to add to or create, and the variable *create_new* will tell whether a new matrix must be created. If a new matrix is to be created, then *wind_lower* and *wind_upper* will contain the lower and upper wind speed associated with the new matrix. These values are used in creating a note for the matrix.

If *matrix_num* is 0, then the operator wishes to return to the main rainflow menu (subroutine **Rainfl**). If the routine has not gotten past this point previously, then *modified* will be false to indicate the data file has not been modified. Once past this point in the routine *modified* becomes true and will remain in this state until **Insert_Data** is used again.

The next step is to read through the data file to where the data are to be inserted. As this is done the information that is read must be saved in a temporary file. The temporary file is opened and is called temp1.cal. The matrices are then read by rewinding the data file, using **Save_Header_Info** to read over the header information, reading the number of matrices (*nops*), and then using **Read_Matrices** to read past the matrices that currently exist to the point where the data are to be inserted. Subroutine **Read_Matrices** reads a matrix and then writes the matrix to the temporary file.

If the *create_new* is true then a new matrix must be created. Since it is a new matrix the variable *num_records* is initialized to 1. Then **Make_Note** is used to create the note for the new matrix. **Rain_Count** is used to count the cycles in the time series data. The length of the time series in seconds is returned in the variable *time*. Since this is a new matrix *opstim* is set to this value if the time series contained operational or buffeting stresses. If the time series contained start/stop stresses *opstim* is initialized to 1.

The matrix can now be set up and filled with the cycle counts found in the time series data. **Setup_Stress_Arrays** is used to initialize the mean stress array (*opsm*) and the cyclic stress array (*opsc*). Then **Init_Matrix** is used to set all of the count values of the matrix to zero. Finally, **Input_Cycle_Count** will put the cycle counts found in the time series data into the matrix in their proper locations. In memory the matrix is stored in the two-dimensional variable *opscc*.

Variable *wndupp* is contained in the main common block. It is used in the lifetime calculations and also in the reading and writing routines **Readmx** and **Writmx**. From the point where the operator inputs the upper wind speed for the current matrix to the point where this value is written to the data file, **Readmx** may be used several times to read past existing matrices (this is done in **Read_Matrix**). This means *wndupp* is changed several times and will not contain the correct value. Variable *wind_upper* is used to store the value input by the operator so that it is not lost. *Wndupp* is set to this value before it is written it to the data file.

The matrix is now written to the temporary file using **Writmx**. To keep the temporary files matrix.tmp and windsp.tmp current **Update_Matrix_File** is used. This routine places the new note in matrix.tmp and the upper wind speed associated with the matrix into windsp.tmp. Then **Read_Matrices** is used to read the remaining matrices in the calculational file and write them to the temporary file. The temporary file now contains the updated version of the data file.

If the operator chooses to add the time series data to an existing file, then a different set of sequences occurs. The first time **Read_Matrices** is used it reads the matrices up to the desired one. Therefore, **Readmx** is used to read in the matrix in which the data are to be inserted. Then **Read_Note** is called to get the numerical values of the variables *num_records*, *wind_lower*, and *wind_upper* from the matrix's note. **Make_Note** then takes these values

and creates a new note with the number of records increased by one. Subroutine **Rain_Count** is used to count the cycles in the time series data. The length of the time series contained in the variable *time* is then added to the old time value stored in *opstim* if operational or buffeting data are used. This procedure keeps track of the total time the matrix represents. If the time series contained start/ stop data, *opstim* is incremented by one to reflect the number of records the matrix contains.

If a cycle is found that has a mean or cyclic stress greater than the maximum found in the current stress arrays, the respective stress array and cycle count matrix must be adjusted. This is done with **Move_Stress_Array**. Once the stress arrays have been adjusted, the counted cycles are inserted into the matrix with **Input_Cycle_Count**.

The program continues as with the new matrix option. It assigns *wndupp* the upper wind speed, writes the matrix to the temporary file, updates the temporary files matrix.tmp and windsp.tmp, and finally writes the rest of the matrices in the calculational file to the temporary file.

The temporary file now contains the updated matrices. This information must be copied back into the data file. This is accomplished by first copying the header information to the data file, then copying the temporary file to the data file. Upon being completed, the program asks the operator if another time series is to be added to this data file. For a positive reply, the current matrices are listed and the process repeats. If negative, control is returned to the calling routine.

LIFE2 Support Routines

Extreme_Values. This subroutine searches through the counted cycles and finds the minimum and maximum of the mean and range stresses. These values are required when initializing the stress arrays and also to ensure that current stress arrays cover the maximums found in the counted cycles.

The rainflow algorithms will store the counted cycles in a file called rain.dat. The mean and cyclic data are read in from this file and compared to the existing minimums and maximums. If the new value represents a new minimum or maximum, the old values are replaced. This continues until there are no more data in the file.

Init_Matrix. This subroutine is used when a new matrix is created. It simply zeros all of the values within the two-dimensional variable *opscc*.

Input_Cycle_Count. This subroutine will input the cycle count found by the rainflow algorithms into a matrix. The rainflow algorithms will store the counted cycles in a file called rain.dat. Each entry will have a mean value and a range value. A do loop is executed until the first value is found in the mean stress array (arranged in ascending order) that is greater than the mean value of the cycle. When this occurs the do loop index (mean_value) will contain the mean stress index in the matrix. The same is done for the cyclic stress. The do loop will always find a value in the stress arrays that is greater since, the arrays are adjusted to cover the maximum values found in the counted cycles.

Once the proper indices have been found, the corresponding entry in the matrix opscc is incremented by one to represent that another cycle has been found. This process repeats until the end of file is reached in rain.dat (i.e., no more cycles).

Input_Header. This subroutine is used when a new data file is created. It simply prompts the operator for all of the necessary information required in the header of a data file. It stores this information in the temporary file header.tmp so that it may be copied into the calculational file.

Input_Speed_Range. This subroutine simply prompts the operator for the wind speed range that is represented by the time series.

List_Matrices. This subroutine will list the matrices that currently exist in a data file. They are listed on the screen 10 at a time with the operator being able to page forward and backward through the matrices. The operator may add time series data to an existing matrix or create a new matrix.

The variables *ifirst* and *ilast* are used to keep track of the first and last matrices to be displayed on the screen. These are initialized to 1 and 10 respectively. The temporary file matrix.tmp contains the description of each matrix in the data file. This is the description that is displayed. Temporary file windsp.tmp contains the upper wind speed for each matrix. This information is used to find the proper place to insert a new matrix. After these files are opened, a loop is entered. The first statements within the loop are not needed the first time but are required for successive passes through the loop. The first statement rewinds the matrix file. The next statements are used to ensure that the variables *ifirst* and *ilast* are between 1 and the number of matrix in the file (variable *nops*).

A do loop is used to read over the matrix descriptions up to the first description to be displayed. Then another do loop is used to display the

desired descriptions. This is followed by displaying the operator options. If the operator chooses to page backward, then *ifirst* is decreased by 10. If the operator chooses to page forward, *ifirst* is increased by 10. The loop is then repeated. In both cases the *ifirst* and *ilast* are held between 1 and nops by the statements at the beginning of the loop.

If the time series is to be input into an existing matrix, then the code prompts the operator for the corresponding matrix number. Variable *create_new* is set to false to indicate the option chosen, and the program exits the loop to return to the calling routine.

If a new matrix is to be created, **List_Matrices** calls **Input_Speed_Range** to prompt the operator for the lower and upper wind speeds of the time series. Then the file windsp.tmp is searched to find the correct place for the new matrix to be inserted. The matrices are kept in the data file in ascending upper wind speed. Once the insertion point is found, the *create_new* is set to true and the program exits the loop to return control to the calling routine.

Make_Note. This subroutine will create a note describing a matrix. If the data are start/stop data then there is no wind speed associated with the matrix. In this case the operator is prompted to input the note. If the data are operational or buffeting stresses then the subroutine will take the number of records and the lower and upper wind speeds and create the note. For example, if the data are operational stresses with lower wind speed of 10 and upper wind speed of 15 and contain 2 records, the subroutine will create the following note:

Operational Stresses; # Records = 2; Range 10 to 15

To insert the number of records and wind speeds into the character string, the note is first written to file note.tmp with the proper format statement. Then the information is read in from the file as a character string into the variable note.

Matrix_Information. This subroutine collects and displays information on a matrix. It first prints a message informing the operator that it is retrieving the desired matrix. If the matrix is one of the first in the data file this message may scroll off the screen before the operator can read it. Therefore, **WAIT** is used to pause for 2 seconds to ensure the operator has time to read the message. The routine then reads over the header information with **Save_Header_Info** and reads in the number of matrices. It then reads matrices until the desired matrix is read into memory.

Next, information that is to be displayed is tabulated. The routine displays a message on the screen to inform the operator of its status. **Read_Note** is then used to get the lower and upper wind speed values. Then the number of cycles in the first bin for the mean and cyclic stresses is tabulated together with the number of cycles in the rest of the matrix.

The information is now displayed, and a pause statement is used to suspend program execution until the operator presses <ENTER>

Move_Stress_Arrays. This subroutine will increase the maximum value in the cyclic and/or mean stress arrays if the time series maximums is not contained in the arrays. If there are not 50 intervals (the maximum number of intervals) in the stress array, intervals are added until the time series maximum is covered. If there are 50 intervals in the stress array, the two smallest intervals are combined with another interval being added until the maximum is covered.

Subroutine **Extreme_Values** will find the maximum values of the mean and cyclic stresses of the counted cycles. The routine will also find the minimum values, but these are not used in **Move_Stress_Arrays.**

An 'if' statement is used to display the message that the mean stress array needs adjusting. Since this message may scroll off the screen before the operator has time to read it, **Wait** is used to delay for two seconds.

An 'if' statement is used inside a loop to determine the maximum in the mean stress array is larger than the maximum value found in the counted cycles. The resolution is found by calculating the difference between the first two intervals.

If the number of intervals in the mean stress array is less than 50, another interval is added. This is accomplished by increasing the number of intervals (*nopsm*) by one, assigning the new interval the next greater resolution step, and putting all zeros in the new interval. The routine then loops back to determine if the time series maximum is contained within the array.

If 50 intervals currently exist, then the first and second intervals are combined. Then all of the intervals are moved back by one (third interval to second, forth interval to third, etc.). The last interval is reinitialized to zero. The mean stress array is moved back by one, with the last value being increased by the resolution step size. The routine then loops back to determine if the time series maximum is contained within the mean stress array.

Once the time series mean stress maximum is contained within the mean stress array the same procedure is used for the cyclic stress array.

Read_Matrices. This subroutine will read cycle count matrices and write them to a temporary file. It is used to read existing matrices within a data file to the point where new data are to be inserted. It is assumed that the temporary file is already open when this routine is called.

Read_Note. This subroutine will read the note for a matrix and extract the number of records and the lower and upper wind speeds from the character string. This information is required when a matrix has a record added to it. The number of records for the matrix must be updated. To accomplish this task, the string value must be converted to a numerical value. This is accomplished by writing only the numerical fields within the note to a temporary file as characters. Then the program reads in the information as numerical values. The wind speeds must also be done so that a new note can be constructed using *make_note*.

Save_Header_Info. This routine reads the header information in a data file and stores it in a temporary file for later retrieval. The temporary file is called header.tmp.

Setup_Stress_Arrays. This routine sets up the stress arrays opsm and opsc. The program first calls **Extreme_Values** to find the minimum and maximum of both the mean and cyclic stresses. It then displays these values and prompts the operator for the desired resolution of the arrays.

The starting mean stress value is found by starting at zero. If the minimum time series stress is less than zero, the starting value is decreased by the desired resolution until the starting value is less than the minimum time series value. If the minimum time series value is greater than zero, the starting value is increased by the resolution until another increase would result in a starting value greater than the minimum time series value.

The variable nopsm (contains the number of elements in the mean stress array) is initialized to zero and a loop is entered. If nopsm is less than 50, it is incremented by 1 and a stress value is assigned to the appropriate element in the array *opsm*. A check is used to insure that the final element in *opsm* is larger than the time series maximum.

The maximum number of elements in the array is 50. If 50 elements are used before the time series maximum is contained within the stress array, the array must be shifted. This is done by assigning each element in the array the stress value of the previous element (i.e., first = second, second = third, etc.). The last array element is then assigned the next greater stress

value. This is repeated until the time series maximum is contained within the mean stress array.

This procedure is then repeated for the cyclic stress. The only difference in the procedure is in finding the starting stress value. Since the cyclic stress will always be greater than zero, a check does not have to be done to see if the time series minimum is less than zero.

Update_Matrix_File. The temporary files matrix.tmp and windsp.tmp contain information about the matrices contained in the current data file. File matrix.tmp contains the notes for each matrix in the file, and windsp.tmp contains the upper wind speed associated with each matrix. When a time series is added to a data file these temporary files need to be updated. Subroutine *update_matrix_file* performs this task.

File matrix.tmp must be updated when a time series is added. If the time series is added to an existing matrix the old note must be discarded and replaced with an updated note. If a new matrix has been created, then the new note must be inserted into the temporary file.

Since the new note is stored in the variable *note*, it must be temporarily stored so the program can use *note*. This is done by writing the note to the temporary file note.tmp. Then a do loop is entered that reads the current notes to the point the new note is to be inserted. As the current notes are read, they are written to a temporary file that will contain the new list of notes. The new note is then read in from note.tmp and written to the temporary file. The program then reads the next note from matrix.tmp. When a time series is added to an existing matrix, this next note is being replaced by the new note so it is discarded. When a new matrix is created, this next note must be written to the temporary file. The remainder of the notes in matrix.tmp are then written to the temporary file. The temporary file now contains the updated list of notes and is copied back into matrix.tmp.

File windsp.tmp contains the upper wind speed associated with each matrix. Therefore it is necessary to update this file when a new matrix is created. The procedure is the same as when updating matrix.tmp. The subroutine writes existing wind speeds to a temporary file up to the point where the new wind speed is to be inserted. The new wind speed is written to the temporary file, then the remainder of the wind speeds in windsp.tmp are written to the temporary file. The temporary file is then copied back into windsp.tmp.

Rainflow Counting Routines

Rain_Count. Subroutine **Rain_Count** calls the subroutines that implement the rainflow counting algorithms. The rainflow counting algorithms are implemented by subroutines **Peaks, Rtrack,** and **Srain.**

Subroutine **Rain_Count** first prompts the operator for the name of the file that contains the time series data and the length of the time series in seconds. It then tries to open the file displaying an error if it cannot open the file. Three data files are then opened to store data from the rainflow counting routines. File extrema.dat stores the peaks and valleys found in the data by **Peaks.** File filtered.dat stores the data after they have been filtered by **Rtrack.** File rain.dat is used to store the counted cycles found by **Srain.**

The three subroutines are then called to implement the rainflow counting algorithms. Subroutine **Peaks** is called first to find the peaks and valleys in the time series. It will read data from the time series file and write the results to the file extrema.dat. Subroutine **Rtrack** is called next to racetrack filter the data. A threshold value, input by the operator, is passed to the routine. It will read the data from file extrema.dat and write the results to filtered.dat. Subroutine **Srain** is called last to do the actual cycle counting. This routine will read the data in from the file filtered.dat and write the mean and cyclic stress of each cycle to the file rain.dat.

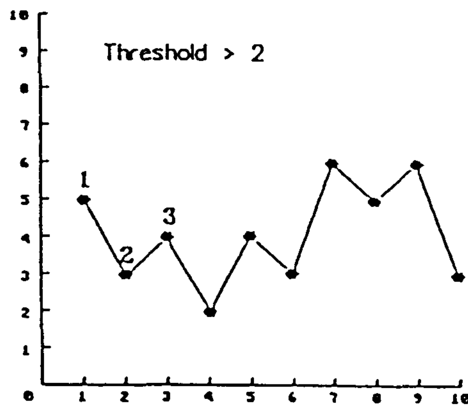
Peaks. Subroutine **Peaks** will find the extrema (peaks and valleys) in the time series data. Since the time series is collected at uniform time intervals, a maxima in the data may have been truncated. The routine compensates for this by doing a parabolic interpolation with the three nearest points to extremas.

The first point is considered an extrema by default so it is read in from the time series, stored in variable $x1$, and written to the output file. Then the second data point is read and stored in variable $x2$. The slope of these data points is calculated and stored in $dx1$. The first data point is no longer needed so the second data point is stored in $x1$. A new data point is read into $x2$. A new slope is calculated and stored in $dx2$. If there is a change in the sign of the two slopes then the routine knows that an extrema has been located. A parabolic interpolation is done to find the extrema, and it is then written to the output file. The slope $dx2$ is then written to $dx1$, and the routine loops back to read in another data point. If there is no change in the signs of the two slopes, then an extrema does not exist between the data points. The slope in $dx2$ is written to $dx1$, and the process loops back to read in another data point. This process continues until there are no more data in the time series file. The last point in the time series is considered an extrema by default so it is written to the output file also.

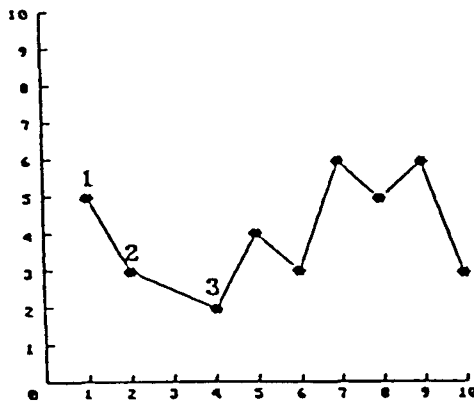
Rtrack. This subroutine implements a racetrack filtering technique. It eliminates all cycles found in the data that are less than an operator-specified threshold value. There are two main processes contained in the routine; the initialization process and the filter/ store process. The initialization will search through the data until a cycle is found that is greater than the threshold value. The purpose of the initialization process is to keep the cycle with the greatest cyclic value until a value is found that is greater than the threshold. When this is found the routine will begin the filter/store process in which all cycles greater than the threshold are written to the output file.

The routine reads in the first three data points and calculates the differences between each point. The absolute difference between the first and second data points is referred to as *diff12*. The absolute difference between the first and third data points is referred to as *diff13*, and the absolute difference between the second and third data points is referred to as *diff23*.

To illustrate the filtering algorithm, first consider the case shown in Fig. 2. In this case, if *diff12* is greater than the threshold value, the routine jumps to the filter/ store process. If *diff12* is less than the threshold but is greater than *diff23* and *diff13*, then the data may look like that shown in Fig. 2a. In this situation the third data point is discarded (Fig. 2b), and the next data point is read.



a) *diff12* has largest value

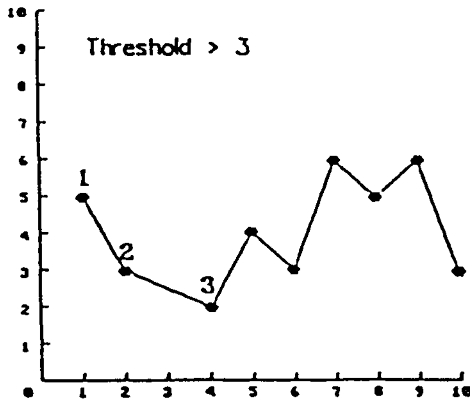


b) Third data point discarded

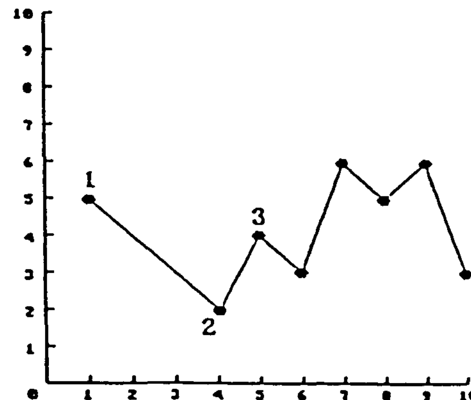
Figure 2

Next consider the case where *diff13* has the largest value. This is shown in Fig. 3a. Note that Fig. 2 shows how this situation may arise. In this case the

routine will discard the second point as shown in Fig. 3b. If *diff13* is greater than the threshold value, then the routine begins the filter/store process. If not, the routine will read in a new value for the third data point and continue with the initialization process.



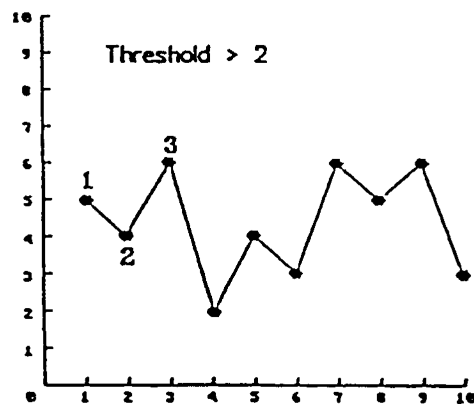
a) *diff13* has largest value



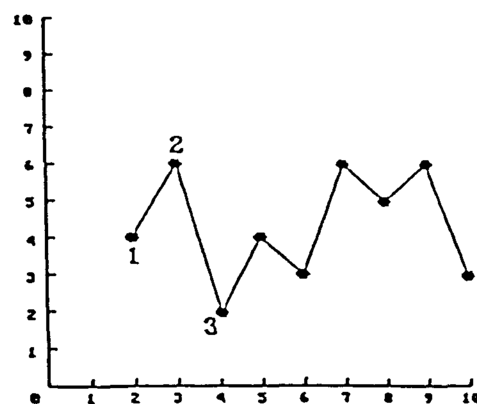
b) Second data point discarded

Figure 3

If *diff23* has the largest value, then the data may look like those shown in Fig. 4a. When this occurs the first data point is discarded and the points are shifted, as shown in Fig. 4b. If *diff23* is greater than the threshold, then the routine jumps to the filter/store process. If not, then the routine will read in a new value for the third data point and continue with the initialization process.



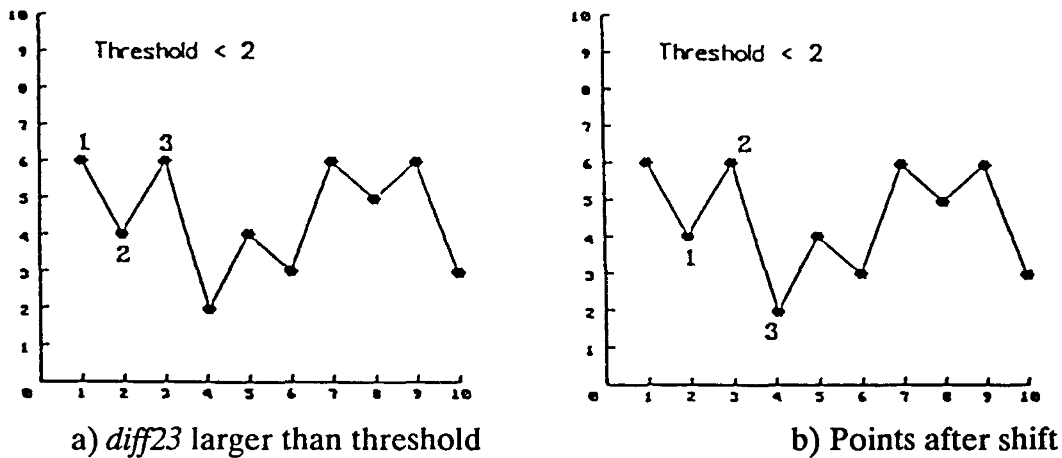
a) *diff23* has largest value



b) First data point discarded

Figure 4

When a cycle is found that is larger than the threshold the program begins the filter/store process. In this process the difference between the first two data points will always be greater than the threshold value. If the difference between the second and third data points is also greater than the threshold value, the data may appear as shown in Fig. 5a. In this case the first data point is written to the output file and then the points are shifted, as shown in Fig. 5b. The routine loops back to read in a new third data point.

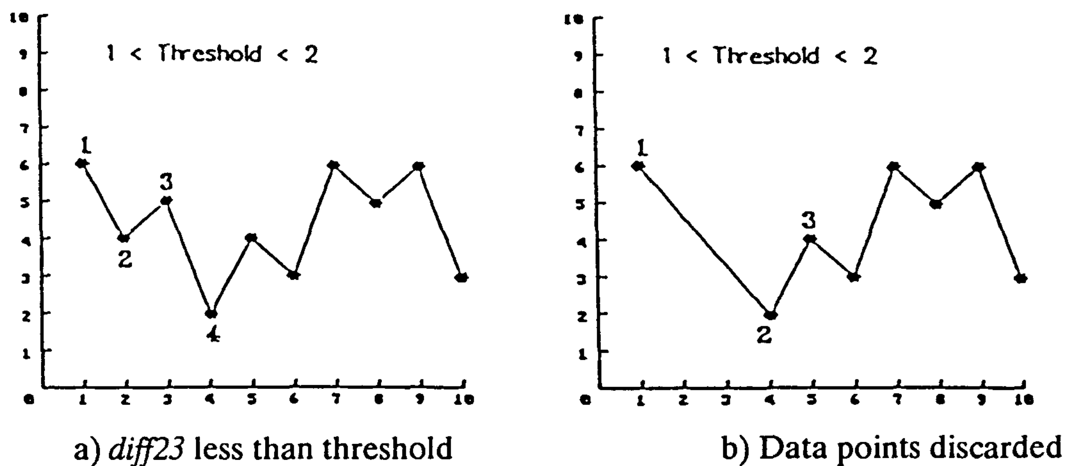


a) *diff23* larger than threshold

b) Points after shift

Figure 5

If the difference between the second and third data points is less than the threshold value, then the data may be as shown in Fig. 6a. In this case the next data point is read as a fourth data point. It can be seen in Fig. 6a that



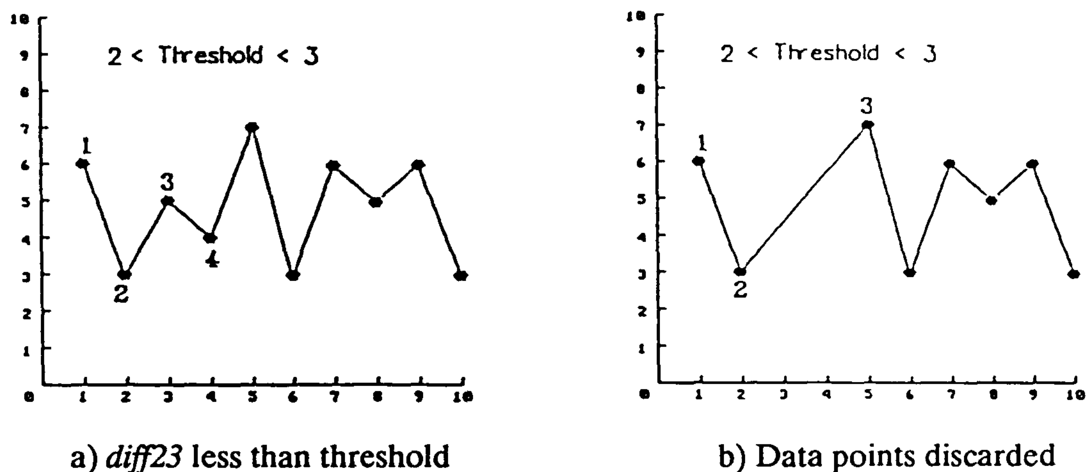
a) *diff23* less than threshold

b) Data points discarded

Figure 6

the difference between the first and and fourth data points is larger than the difference between the first and second data points. In this situation the second and third data points are discarded (see Fig. 6b).

Another situation may have the data as shown in Fig. 7a. Here *diff23* is less than the threshold and *diff12* is greater than *diff14*. In this case the third and fourth data points are discarded (see Fig. 7b).



a) *diff23* less than threshold

b) Data points discarded

Figure 7

This process continues until there are no more data in the input file. When this occurs, there is one more cycle to write to the output file. As mentioned earlier, the difference between the first and second data points will always be greater than the threshold value. This being the case, when the end of file is reached these data points must be written to the output file.

Srain. This subroutine implements the actual rainflow counting of cycles. It reads in the filtered data and writes the mean and range stress of each cycle to the rain.dat output file. A detailed explanation of this algorithm is given in Downing and Socie (5). The paper discusses two types of algorithms. The one incorporated into this routine is the single-pass algorithm. The only difference is that the algorithm in the paper uses arrays to store the data, where as the algorithm incorporated here uses files.

SUMMARY

A set of algorithms that permits the analysis of time-series stress data has been incorporated into the LIFE2 fatigue/fracture analysis code. The algorithms are built around a rainflow counting algorithm. Support algorithms that structure the data in a format compatible with the LIFE2 code have also been implemented. This report describes the major design decisions that were made in implementing the algorithms and gives a detailed description of the subroutines used to implement the algorithms. For a more detailed description of the algorithms and an explanation of how to use them please see Schluter and Sutherland (4).

REFERENCES

1. Schluter, L. L. and Sutherland, H. J., Reference Manual for the LIFE2 Computer Code, SAND89-1396, Sandia National Laboratories, Albuquerque, NM, September 1989.

2. Sutherland, H. J., Analytical Framework for the LIFE2 Computer Code, SAND89-1397, Sandia National Laboratories, Albuquerque, NM, September 1989.

3. Schluter, L. L. and Sutherland, H. J., "Rainflow Counting Algorithm for the LIFE2 Fatigue Analysis Code," Ninth ASME Wind Energy Symposium, Vol. 9, 1990, pp.121-123.

4. Schluter, L. L. and Sutherland, H. J., User's Guide for LIFE2's Rainflow Counting Algorithm, SAN90-2259, Sandia National Laboratories, Albuquerque, NM, August 1990

5. Downing, S. D., and Socie, D. F., "Simple Rainflow Counting Algorithms," *International Journal of Fatigue*, Vol. 4, N. 1, 1982, pp. 31-40.

6. Veers, P.S., Winterstein, S. R., Nelson, D. V. and Cornell, C. A., "Variable Amplitude Load Models for Fatigue Damage and Crack Growth," *Development of Fatigue Loading Spectra*, ASTM STP 1006, J. M. Potter and R. T. Watanabe, eds., 1989, pp. 172-197.

7. Microsoft FORTRAN Optimizing Complier, Version 5.0, Microsoft Corp. (1989).

DISTRIBUTION:

Dr. R. E. Akins
Washington & Lee University
P.O. Box 735
Lexington, VA 24450

The American Wind Energy Association
777 N. Capitol Street, NE
Suite 805
Washington, DC 20002

Dr. R. N. Clark
USDA
Agricultural Research Service
Southwest Great Plains Research Center
Bushland, TX 79012

P. R. Goldman
Wind/Hydro/Ocean Division
U.S. Department of Energy
1000 Independence Avenue
Washington, DC 20585

R. W. Thresher
Solar Energy Research Institute
1617 Cole Boulevard
Golden, CO 80401

W. A. Vachon
W. A. Vachon & Associates
P.O. Box 149
Manchester, MA 01944

1520	L. W. Davison
1522	R. C. Reuter, Jr.
1522	D. W. Lobitz
1522	E. D. Reedy
1523	J. H. Biffle
1524	C. R. Dohrmann
1524	D. R. Martinez
3141	S. A. Landenberger (5)
3151	G. L. Esch (3)
3154-1	C. L. Ward (8)
3161	P. S. Wilson
6000	V. L. Dugan, Acting
6200	B. W. Marshall, Acting
6220	D. G. Schueler
6225	H. M. Dodd (50)
6225	T. D. Ashwill
6225	D. E. Berg
6225	M. A. Rumsey
6225	L. L. Schluter
6225	W. A. Stephenson
6225	H. J. Sutherland
6225	P. S. Veers
7543	R. Rodeman
7543	T. G. Carne
7543	J. Lauffer
8524	J. R. Wackerly

DO NOT MICROFILM
THIS PAGE