

A Sequence Retainable Iterative Algorithm for Rainflow Cycle Counting

Dongfang Jiang

Northwestern Polytechnical university

ABSTRACT

To get a sequence retainable rainflow cycle counting algorithm for fatigue analysis, an alternate equivalent explanation to rainflow cycle counting is introduced, based on which an iterative rainflow counting algorithm is proposed. The algorithm decomposes any given load-time history with more than one crest into three sub-histories by two troughs; each sub-history with more than one crest is iteratively decomposed into three shorter sub-histories, till each sub-history obtained contains only one single or no crest. Every sub-history that contains a single crest corresponds to a local closed (full) cycle. The mean load and alternate load component of the local cycle are calculated in parallel with the iterative procedure. After a local cycle is counted and its constituent crest and trough are removed from the analyzed load-time history, history reconstruction from remnants of the original load-time history is not needed before successive iterative counting, which is a preferable property different from other rainflow cycle counting algorithms. The algorithm can count local cycles in the analyzed load-time history in their chronological order. The first extracted cycle can also be easily managed to be the major cycle. The disadvantages of the algorithm are that it is not suitable for rigorous real-time applications and it requires larger stack capacity than non-iterative algorithms when applied.

CITATION: Jiang, D., "A Sequence Retainable Iterative Algorithm for Rainflow Cycle Counting," *SAE Int. J. Mater. Manf.* 7(1):2014, doi:10.4271/2013-01-9091.

1. INTRODUCTION

Counting local cycles is a basic requirement in fatigue estimation to material under spectrum loads. Since the propagation of fatigue crack depends not only on alternate and mean load component of load cycles, but also on the order the load cycles applied [1, 2, 3, 4, 5], it is helpful to have a counting algorithm that can not only count local cycles efficiently, but also count them according to their chronological appearance order in the load-history concurrently.

Most of the rainflow counting algorithms [6, 7, 8] popularly employed cannot retain the order information of local cycles in their load-time history while counting. Some algorithms [9, 10] do keep the load sequence information of local cycles but are complicate in realization.

In this paper, an iterative rainflow counting algorithm that can count cycles in a given load-time history sequentially is to be introduced. In cases where sequentially counting a load-time history is not critical, the first extracted cycle by the algorithm can be managed to be the major cycle, which is

also a preferred, even critical, property for some applications [11].

An alternate equivalent explanation to rainflow counting is introduced first. Followed is a description of the iterative algorithm. An illustration example with explanation is provided in section 4. Merits and defects of the algorithm are discussed in section 5 before conclusion drawn in section 6.

2. AN EQUIVALENT EXPLANATION TO RAINFLOW COUNTING

In an off-line two stage rainflow counting procedure, any original load-time history can be broken into two segments from the deepest trough of the load-time history. Swapping the position of these two segments and then connecting them, a new equivalent load-time history with equal load at both its start and end reverse points is formed, and the load at these two points are the smallest in the newly formed load-time history. In the following sections of this paper, load-time histories to be analyzed are restricted to these rearranged histories.

Assume [fig. 1](#) is a diagram of a rearranged load-time history obtained by the method described above. In this figure, letters are employed to mark load crests and troughs actually recorded in the load-time history; Integer numbers, which will help in describing the iterative algorithm later, are borrowed to mark auxiliary points in the load-time history.

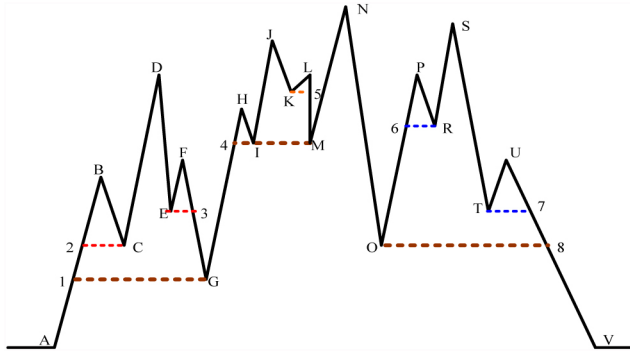


Fig. 1. A hypothetical load-time history

It is known that any load-time history with N closed (full) load cycles can be decomposed by rainflow counting algorithm into $N - 1$ minor closed cycles and one major closed cycle. The major cycle is the load cycle with maximum alternate load component among all cycles contained in the analyzed load-time history.

Now assume that crest N is the highest crest in the load-time history depicted with [fig. 1](#), and trough A and V are respectively start and end trough of the load-time history. It is obvious that trough A and V correspond to the smallest load trough located to the left and right side of crest N in the analyzed load-time history respectively. According to definition of rainflow counting, the major load cycle in this load-time history is the one outlined by route $A1G4IMNO8V$. The load variation range of this cycle equals to the load difference between the load at the highest crest N and the lowest trough A . The mean load of this cycle equals to the algebraic mean of load at crest N and trough A . The formation of this major cycle outline can be viewed as rain drops flowing along the history diagram from the highest crest N toward both sides of itself respectively. When the flowing rain drops encounter a slope of the load-time history diagram, assume that they flow downward along the slope. When they arrive at a local trough and if the trough is not the deepest one on the corresponding side, assume that they flow horizontally outwards to next slope in their direction ahead. The rain drops stop flowing when they arrive at the lowest load trough, which is exactly the start or end trough of the load-time history being analyzed. The route of the flowing rain drops outlined a closed load cycle. This is our equivalent explanation of rainflow counting for the major load cycle. Other local load cycles can be extracted similarly from the load-time history remnants generated after removing the major cycle. It can be proved, to any given load-time history, that both the number of load cycles counted and their parameters (mean load and alternate load of a cycle)

calculated by this method are equivalent to those counted and calculated by a four-points counting method [8].

Now let us count manually the cycles in the example load-time history depicted with [fig. 1](#) according to our equivalent rainflow explanation. Assume there is a rain drop flowing from the highest load crest N to its left side; it will reach point M first. As point M is not the lowest trough to the left of crest N in the history, (in fact, the lowest trough to the left of point N in this load-time history is the start trough A), the rain drop is assumed to flow horizontally to trough I , to point 4, and then to the final trough A after passing trough G and point 1 sequentially. We can denote this left rainflow route $NMI4G1A$. Similarly, if a rain drop flows from the highest load crest N to its right side, a right rainflow route $NO8V$ is formed. Crest N connects left route $NMI4G1A$ and right route $NO8V$ to form an outline $A1G4IMNO8V$, which marks exactly the major cycle in the analyzed load-time history. The minimum load of this cycle equals the load at start trough A or at end trough V . The maximum load of this cycle equals the load at the highest crest N .

Now we erase the major cycle $A1G4IMNO8V$ from [fig. 1](#). Three load-time sub-histories is left behind. They are marked in [fig. 1](#) with outlines $12BCDEF3G$, $4HIJKL5M$, and $6PRSTU78$ respectively. These three load-time sub-histories can be processed in the same way as that is used in extracting the major cycle from its parent load-time history respectively. Taking load-time sub-history $12BCDEF3G$ as an example, its highest crest is crest D , the lowest load points to the left and right side of crest D in the load-time sub-history is point 1 and trough G , a minor cycle outlined by $12CDE3G$ can be detected according to the method described in the forward part of this section. For this minor cycle, the maximum load equals the load corresponding to crest D , and the minimum load equals the load corresponding to trough G . Further iterative processing with the left segment of $12BCDEF3G$ after removing the minor cycle outlined by $12CDE3G$ will generate minor cycles $2BC$ and $EF3$. Similar process to sub-history $4HIJKL5M$ will generate cycles outlined by $4IJK5M$, $4HI$, and $KL5$. Load-time sub-history $6PRSTU78$ will be decomposed to cycles outlined by $6RST78$, $6PR$, and $TU7$. There are 10 crests in total in the complete load-time history depicted with [fig. 1](#). It implies that the history contains 10 full (closed) load cycles. The equivalent explanation of rainflow counting method counts 10 closed cycles from the load-time history. Comparing the output of this newly explained rainflow counting method with other rainflow counting methods [7, 8, 12, 13, 14, 15, 16], the number of cycles counted is equal; each counted local closed cycle is one-to-one mapped, and in a mapped cycle pair, the mean and alternate load of the cycle is equal respectively.

Now denoting the load at trough A , crest N and trough V in the example load-time history as σ_A , σ_N and σ_V respectively, and denoting the load at other points of the load-time history similarly, by taking

$$\sigma_{\max} = \sigma_N, \quad \sigma_{\min} = \sigma_A = \sigma_V,$$

The mean load σ_{ave} and alternate load σ_{osc} of the major cycle outlined by A1G4IMNO8V can be calculated with

$$\sigma_{ave} = (\sigma_{max} + \sigma_{min})/2, \quad \sigma_{osc} = (\sigma_{max} - \sigma_{min})/2 \quad (1)$$

Similar computation can be carried out to other minor cycles. Taking minor cycle outlined by 12CDE3G as an instance, its mean load $\bar{\sigma}_{ave}$ and alternate load $\bar{\sigma}_{osc}$ can be computed with

$$\bar{\sigma}_{ave} = (\sigma_D + \sigma_G)/2, \quad \bar{\sigma}_{osc} = (\sigma_D - \sigma_G)/2$$

It is not difficult to prove that the parameters calculated in this way are identical to those of the corresponding cycles extracted by other popularly used rainflow counting methods [6, 7, 8].

Having established the equivalent alternate explanation of rainflow counting, an algorithm to implement the iterative cycle counting method can be constructed.

3. ITERATIVE ALGORITHMS

The base of iterative algorithms is to factorize a complicated problem into simpler analogue problems. This process is employed iteratively till the simplest form of the problem appears. The answer to the simplest form of the iterative problem is normally easy and direct forward. By back substitution, the original complicated problem can be solved when solution to the simplest form of the problem is given.

3.1. An Example of Iterative Algorithm

Factorial $f(n) = n!$ is one of the best examples used to explain iterative computational algorithms.

As $f(n) = n! = n \cdot (n-1)! = n \cdot f(n-1)$, the iterative algorithm for computation of factorial can be described with the following function in pseudo C-language syntax:

int Factorial (int n){

1. **if**($n < 0$) **then** {
 Report Error;
 return -1 ; }
2. **if**($n == 0$ or $n == 1$) **then** { **return** 1 ; }
3. { **return** $n \cdot \text{Factorial}(n-1)$; }

The algorithm implementation of function Factorial(n) contains only three sentences. It can compute factorial of any positive integer n without a loop as it is needed in a non-iterative factorial computation algorithm. In the iterative algorithm above, the sentence numbered with 3 carries out the functions of a loop in a non-iterative algorithm by iteratively calling function Factorial(n) with a parameter $n-1$, and the sentence numbered with 2, i.e. assigning 1 to Factorial(0) or Factorial(1), provides the stop condition for the iterative process.

Taking advantages of iterative computation, an iterative algorithm for rainflow cycle counting is constructed.

3.2. Iterative Algorithm for Rainflow Cycle Counting

To make description of the iterative rainflow cycle counting algorithm convenient, assume that the load-time history to be analyzed is stored in an array named $data[n]$ with $N_{start} \leq n \leq N_{end}$. That is to say that the recorded load of the start reverse point of the given load-time history is stored in, $data[N_{start}]$, the load of the end reverse point of the analyzed load-time history is stored in $data[N_{end}]$, and the load of other load reverse points of the analyzed load-time history is stored in $data[n]$ with $N_{start} < n < N_{end}$. Obviously, the indices n of $data[n]$ indicate the positions of crests and troughs in the recorded load-time history. The smaller the index n is, the early the related crest or trough appears in the analyzed load-time history. As troughs and crests appear alternately in any load-time history, if $data[N_{start}]$ stores the load of a trough, then $data[N_{start}+1]$ stores definitely the load of a crest, $data[N_{start}+2]$ stores the load of an adjacent trough, $data[N_{start}+3]$ stores the load of a crest again, and so on. Thus, load crests and troughs are also stored in array $data[n]$ alternately. A load-time history containing only single full cycle has at least three recorded load reverse points. A load-time history without half cycles should have an odd number of recorded reverse points.

Without losing generality, assume that the first and the last load reverse point in the analyzed load-time history are both troughs, and assume that $data[N_{start}] = data[N_{end}]$ is the smallest load in the analyzed load-time history, then the minimum load of the major cycle in the analyzed load-time history is $\sigma_{min} = data[N_{start}]$. We use N_{bottom} to denote the index of one of, $data[n]$, $N_{start} \leq n \leq N_{end}$ which satisfies $data[N_{bottom}] = \sigma_{min}$. Therefore, we have $N_{bottom} = N_{start}$ initially. The iterative rainflow cycle counting algorithm can now be described with the following function:

void Rainflow (int N_{start} , int N_{end} , int N_{bottom}){

Comment: The function used to implement the iterative cycle counting algorithm contains three parameters, namely the index corresponding to the start reverse point, end reverse point, and the smallest load reverse point in the analyzed load-time sub-history respectively. It is worth to note that any load-time history being analyzed here should be viewed as a sub-history since the function is normally called iteratively in an iterative algorithm.

Step 1: **If** ($N_{end} - N_{start} < 2$), **then** {**return**; }

Comment: This step corresponds to the situation that the load-time history being analyzed contains no crest, as the shortest load-time sub-history that contains at least one closed load cycle should have at least two troughs at both end and a crest in-between. If $N_{end} - N_{start} < 2$, further iterative should be blocked; No cycle is counted and hence no mean and