ALGORITHM OF THE RAINFLOW METHOD

Hiroyuki Anzai
Department of Electric, Electronic and Computer Engineering
Kyushu Institute of Technology, Kitakyushu 804, Japan

Abstract: This paper presents a fundamental theory for the digital system-directed rainflow method given in the paper[1]. For a given strain/time history, P/V differences (the absolute difference values between successive peaks and valleys) are detected. For successive three of them, this paper shows a condition of existing a close loop in a stress/strain hysteresis and an operation of giving the three the same effect as cutting the loop off and storing it. Given successive P/V differences are modified by this operation so as to become a set of P/V differences such that each of them reasonably corresponds to a half loop in the stress/strain hysteresis. For the set, rainflow cycle counting or damage computation is done.

Keywords: rainflow, fatigue damage, P/V difference.

1. INTRODUCTION

The rainflow algorithm is implemented by a system of the input process, the rainflow process and the display process[1]. The input process is an interrupt program which samples a strain wave or a strain/time history, computes a P/V difference, and sends it to the input buffer. The rainflow process is a resident program, which takes the first-in P/V difference value from the input buffer, and executes the rainflow method. The display process displays the obtained cycle counts or accumulated total damage.

2. INPUT PROCESS, P/V DIFFERENCES AND HALF CYCLE COUNTS

A strain wave is measured and sampled by a strain gauge and the sampled data is sent to an A/D converter. The converted data is an integer $d$ such that $-2^{m-1} \leq d < 2^{m-1}$, where $m$ (called the system parameter) is the number of output lines of the converter. From the sampled digital data, a peak or a valley is detected and the absolute value of difference between it and the value of the latest valley or peak is computed. The obtained value is called a peak and valley difference or a "P/V differences" for short, which is a positive integer $d$ such that $1 \leq d \leq 2^{m-1}-1$. It is sent to a data buffer called "input buffer".

Fig.1 P/V differences
For a given half loop in the stress/strain hysteresis, we use its strain image as its size, which is equivalent to the P/V difference of the associated part of the strain wave. For a complex stress/strain hysteresis, the rainflow process reasonably analyzes it into a set of half loops by sampling the associated strain wave. In the set, let there exist \( n_1, n_2, \ldots, n_k \) half loops whose sizes (the related P/V differences) are \( d_1, d_2, \ldots, d_k \). Each \( n_i \) is called a "half cycle count" related to \( d_i \). Note that \( K \) is at most \( 2^{n_i} - 1 \) because arbitrary \( d_i \) is an integer such that \( 1 \leq d_i \leq 2^{n_i} - 1 \). Namely, for a given system parameter \( m \), possible P/V differences are \( 1, 2, \ldots, 2^{m-1} \). Therefore, if we want to store all of the half cycle counts \( n_1, n_2, \ldots, n_k \), it is necessary to provide at most \( 2^{m-1} \) memory places.

3. TOTAL DAMAGE

For the upper or the lower half of a loop, let us assume that the effect of the half loop to fatigue damage is a function of the strain-coordinate image of the half loop. This means that two half loops whose projected strain images are equivalent have the same effect to fatigue damage no matter when and where they occur in the stress/strain graph. Therefore, the effect of the half loop to fatigue damage is a function of the P/V difference of the related part in the strain wave.

When assuming the linear damage rule, the total damage can be written as follows:

\[
D = n_1 \cdot f(1) + n_2 \cdot f(2) + \cdots + n_k \cdot f(k) + \cdots + n_{m-1} \cdot f(2^{m-1})
\]

(1)

where \( f(d_i) \) is the damage caused by the half loop whose size is \( d_i \).

If we want to compute the total damage in real time, we have to compute, at every time to obtain a P/V difference \( d \), the damage function \( f(d) \). It may be, however, impossible to compute it on a micro-processor because of its time consuming property. As a solution, we can use a table indexed by integers (P/V differences) \( 1, 2, \ldots, 2^{m-1} \). For each integer \( k \), the \( k \)-th element of the table contains the already computed damage value \( f(k) \).

4. LOOP CONDITION AND LOOP REAPING

In Fig. 2, for the strain wave, you can find that following two statements are equivalent.

- A rain which drops from a peak \( b \) of the strain wave is reachable to the next peak \( e \) of the wave (or which drops from a valley of strain wave is reachable to the next valley).
- For the related successive three P/V differences \( d_1, d_2 \) and \( d_3 \), the following condition holds

\[
d_1 > d_2 \leq d_3
\]

(2)
Furthermore, in Fig. 2, you can find that if the above condition holds, there exists a (closed) loop in the stress/strain hysteresis. We call this condition "loop condition".

For a given stress/strain hysteresis, if there exists a loop satisfying the loop condition (2), the rainflow method picks the loop off the stress/strain hysteresis. In the related strain wave, the rainflow dropping from $b$ to $d$ cuts off the part according to the loop, whose size (P/V difference) is $d_1$. The residual strain wave is just the track of the rainflow $a-b-d-e$, making a wave from a valley to the following peak or from a peak to the following valley. Thus newly obtained monotonously ascending (or descending) part of the residual strain wave corresponds to the residual half loop $\alpha-\beta-\delta$ in the stress/strain hysteresis, and its size is just the P/V difference $d_1 + (d_3 - d_2) = d_3 - d_2 + d_1$. We call this operation "loop reap".

5. RAINFLOW PROCESS AND LOOP-FREE SEQUENCE

Let a given sequence of successive P/V differences be $d_1, d_2, \ldots, d_m, d_{m+1}, \ldots, d_{m+n}$. Then, for any successive three in the sequence, the loop condition (2) does not hold if and only if the following relation is valid.

$$d_1 \leq d_2 \leq \ldots \leq d_m > d_{m+1} > \ldots > d_{m+n}$$

where $m = 1, 2, \ldots$ and $n = -1, 0, 1, \ldots$

and when $m = 1$ and $n = -1$, the sequence is empty.
In this sequence, the rightmost maximum P/V difference $d_m$ is called the “border”. This sequence is separated by this border $d_m$ into the monotonously increasing left part and the strongly decreasing right part. Note that in the left part, there may exist loops in the related stress/strain hysteresis. We have, however, no loop in the stress/strain hysteresis related to the right part.

For a given sequence such that the relation (3) holds and for a newly given P/V difference $d$, the rainfall process has four kinds of operations:

When a P/V difference $d$ is given to the non-empty sequence (3), if it holds that $d_{\text{max}}>d$, $d$ can be placed at the right of $d_{\text{max}}$ without disturbing the loop-free property of the right side of the sequence. After increasing $n$ by one, $d$ is placed at the $(m+n)$-th position and is represented by $d_{\text{max}}$. We call this operation “data-shift”.

On the other hand, if it is valid that $d_{\text{max}} \leq d$, we have two cases of either $n > 0$ or $n = 0$. The former case ($n > 0$) gives the sequence the loop condition $d_{\text{max}} - 1 > d_{\text{max}} \leq d$. So, the “loop-reap” operation is applied. The latter case ($n = 0$) means that the sequence (3) has only the monotonously increasing left part $d_1, d_2, \ldots, d_m$ such that

$$d_1 \leq d_2 \leq \cdots \leq d_m$$

(4)

This situation is called “critical”, where we have two kinds of operations concerned with the border $d_m$. When $d_m > d$, the process applies the data-shift operation as shown above. When $d_m \leq d$, the role of the border is transferred from $d_m$ to $d$, and $m$ is increased by one. We call this operation “border-shift”.

The initial condition is that $m = 1$ and $n = -1$, where there is no element in the above sequence. For the initially given P/V difference $d$, the rainfall process increases $n$ by one to 0 and puts $d$ as the border $d_1$.

For successively given P/V differences, the rainfall process goes on as follows: When a border-shift occurs, $m$ is increased by one. When a data-shift occurs, $n$ is increased by one. And when a loop-reap occurs, $n$ is decreased by two at once and sometimes $n$ is decreased to -1. Thus we have sometimes the state where the condition $n = -1$ and $m \geq 1$ holds. In this state, the sequence (4) has lost the border $d_m$. However, the rainfall process just after then increases $n$ by one to 0 and puts there the residual P/V difference of the loop-reap operation performed just before. This loop-reaping should have been applied to a loop triple $d_m, d_{m+1}$ and $d$ such that $d_{m+1} > d_{\text{max}} \leq d$ where $d$ is a given P/V difference and $n = 1$, and resulted in obtaining the residual P/V difference $d' = d_m + (d - d_{\text{max}}) \geq d_m$ and $n = -1$. Therefore, it is clear that $d'$ is the border in the sequence where the old border $d_m$ is replaced.
by $d'$. We call this operation "border-reset", which replaces the old border $d_n$ by a given P/V difference $d'$. This operation always occurs when and only when $n = -1$. Note that this condition $n = -1$ contains the initial condition where the old border does not exist but can be assumed to be $d_1 = 0$.

From the above, we have the following rainflow program as the basis of the rainflow cycle counting or the damage computation.

```pascal
program rainflow:
  var d, m, n, j: integer;
  a, b, c: array[0 .. 1000] of integer;
procedure loopReap(d: integer);
begin j := j + 1; c[j] := d; end;
procedure loopDetect(var d: integer);
begin
  if n = -1 then begin /* border-reset */
    n := 0; b[0] := d
  end else if b[n] > d then begin /* data-shift */
    n := n + 1; b[n] := d;
  end else if n = 0 then begin /* border-shift */
    a[m] := b[0]; m := m + 1; b[0] := d;
  end else begin /* loop-reap */
    loopReap(b[n]); loopReap(b[n]);
    d := d - b[n] + b[n-1];
    n := n - 2;
    loopDetect(d);
  end;
end;
begin
  m := 1; n := -1; j := 0; d := get();
  while d >= 0 do begin
    loopDetect(d); d := get();
  end;
end.
```

Where, "get()" is a function which, if the input buffer is not empty, takes the first-in P/V difference out of the buffer and returns it. The tail-recursive structure of the procedure loopDetect can be easily transformed into a non-recursive structure. A label "L" is declared and used at the top statement, and then the procedure–call "loopDetect(d)" is replaced by a statement "goto L". In the main program, the statement "loopDetect(d)" must be replaced by "loopDetect" or the body of the procedure itself.
6. RAINFLOW CYCLE COUNTING

The program rainflow inputs successive P/V differences, modifies and classifies them into three sets $S_a$, $S_b$ and $S_c$ according to the three arrays $a$, $b$ and $c$, respectively.

As shown in the above program, $S_a$ receives an element when and only when a *border-shift* occurs, and similarly, $S_b$ receives two elements when and only when a *loop-repeat* occurs. Any one of members in both is never removed or modified. Namely, when a P/V difference is entered into $S_a$ or $S_b$, it becomes *fixed*. This property is very useful to implement the rainflow algorithm either for cycle counting or for damage computation, because any element in $S_a$ and $S_b$ is not necessary to store for further use after once using it for cycle counting or damage computation.

On the other hand, all elements of the ordered set $S_c$ are necessary to store because they are not *fixed*, that is, they may be removed or modified each when a loop-repeat occurs. $S_c$ must contain at the worst case $2^n-1$ members $2^n-1, 2^n-2, \ldots, 2, 1$ in this order. Therefore, for $S_c$, we must provide at least $2^n-1$ memory cells of each contains an integer of at least $m$ bits long.

This section modifies, using the above, the program rainflow to obtain half-cycle counts $n, m_1, \ldots, m^n-1$ from P/V differences successively detected from a given strain wave, as follows:

```pascal
program rainflowCycleCount;

const max = 2^n-1;
var d, n: integer;
    b: array [0..max] of integer;
    m: array [1..max] of integer;

procedure count(d: integer); /* increment the count of */
begin m[d] := m[d] + 1 end;

procedure discount(d: integer); /* the size a half cycles */
begin m[d] := m[d] - 1 end;

begin
    for n := 1 to max do m[n] := 0; /* initialize m */
    n := -1; d := get();
    while d >= 0 do begin
        L: if n = -1 then begin /* border-reset */
            count(d); n := 0; b[n] := d;
        end else if b[n] > d then begin /* data-shift */
            count(d); n := n + 1; b[n] := d
        end else if n = 0 then begin /* border-shift */
```
count(d); b[0] := d;
end else begin /* loop-reap */
count(b[n]); discount(b[n-1]);
d := d - b[n] + b[n-1]; n := n - 2;
goto L;
end;
d := get();
end

For all P/V differences \( d = 1, 2, \ldots, 2^n - 1 \), each half cycle, count of size \( d \), \( n_d \), is stored in the array element \( m[d] \), which is increased by the procedure \( \text{count}(d) \) and is decreased by the procedure \( \text{discount}(d) \). Note that whenever \( d \) is entered into \( S_n \) (array \( b \)), the procedure \( \text{count}(d) \) is called for increasing the half cycle count \( n_d \) in \( m[d] \), which may, however, be overcounted because this \( d \) is not fixed.

The border is always set in \( b[0] \). When the border-reset is done, the old border in \( b[0] \) is vanished and the new border is set into the \( b[0] \). The old border need not be stored because its half cycle counting has been done and it is implicitly sent to \( S_n \), that is, it becomes fixed.

When the loop reaping occurs, the last two elements of \( b \), i.e., \( b[n-1] \) and \( b[n] \), are removed from \( b \) and \( b[n] \) is implicitly entered into \( S_n \) twice. Therefore, their half cycle counts \( n_{b[n-1]} \) in \( m[b[n-1]] \) and \( n_{b[n]} \) in \( m[b[n]] \) have to be corrected as follow:

\[ \text{discount}(b[n]); \text{count}(b[n]); \text{count}(b[n]); \text{discount}(b[n-1]); \]

In these statements, \( \text{discount}(b[n]) \) and \( \text{count}(b[n]) \) are cancelled each other, resulting into the statements as shown in the program.

Example Let a given sequence of P/V differences be \( 2, 5, 3, 7, 7, 5, 6, 6, 3 \), \( 6 \). Then the behavior of the above program is shown as follows:

<table>
<thead>
<tr>
<th>input/ array b</th>
<th>operation</th>
<th>count operation</th>
<th>contents of counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>border-reset</td>
<td>m[9]++</td>
<td>m[9]=1</td>
</tr>
<tr>
<td>9</td>
<td>data-shift</td>
<td>m[8]++</td>
<td>m[8]=1</td>
</tr>
</tbody>
</table>

17
In each relational expression of the column array b, each number except for the underlined is the content of an element of b; the first number is contained in b[0], the second in b[1], and so on. In the column count operation, m[i]++ (or m[i]--) means that the content of m[i] is increased (decreased) by one.

7. USE OF A PUSHDOWN STACK

In Section 6., note that S has the same input/output mechanism as the pushdown stack or the last-in first-out list. That is, when requested a data, S always outputs only the last-in data. For this kind of data container, some commands called push, pop, top and empty are provided. A function “pop()” takes a data as its function value out of the container. The last-in data can be seen by a function “top()” and whether or not the container is empty can be detected by the Boolean function “empty()”. Use of them can make us rewrite the execution part of the program rainflowCycleCount in Section 6., as follows:

begin
  stackInitialize; /* make the stack empty */
  d := get(); /* get a data from the buffer */
  while d >= 0 do begin /* d (<0) is an end marker */
    L: if empty then begin /* border-reset */
      count(d); push(d);
      end else if top() > d then begin /*data-shift */
      count(d); push(d);
      end else begin
          r := pop(); /* r is a register */
          if empty then begin /* border-shift */
            count(d); push(d);
            end else begin /* loop-reap */
            count(r); d := d - r;
            r := pop();
            discount(r); d := d + r;
            goto L;
          end
        end;
    end;
    d := get();
8. RAINFLOW DAMAGE ACCUMULATION

Instead of counting half cycles, we can directly compute the total fatigue damage from successive P/V differences detected from a given strain wave. This is given under the assumption of the linear damage rule by the following program where the count and discount in the program rainflowCycleCount are replaced by the procedures addDamage and subtractDamage, respectively. As stated in Section 3, all the damage values \( f(1), f(2), \cdots, f(2^n-1) \) are previously computed and initially stored in the array \( P \) of the program.

```plaintext
program DamageAccumulator;
    label L;
    const max = 2^n - 1;
    var d, n: integer; D: real;
    b: array [0 .. max] of integer;
    F: array [0 .. max] of real; /* damage table */
    function damageTabInit;
        begin for d := 1 to max do read(F[d]) end;
    procedure addDamage(d: integer);
        begin D := D + F[d] end;
    procedure subtractDamage(d: integer);
        begin D := D - F[d] end;
    begin
        damageTabInit; D := 0; n := 0; d := get();
        while d >= 0 do begin
            L: if n = -1 then begin /* border-reset */
                addDamage(d); n := 0; b[0] := d;
            end else if b[n] > d then begin /* data-shift */
                addDamage(d); n := n + 1; b[n] := d;
            end else if n = 0 then begin /* border-shift */
                addDamage(d); b[0] := d;
            end else begin /* loop-reap */
                addDamage(b[n]); subtractDamage(b[n-1]);
                d := d - b[n] + b[n-1]; n := n - 2;
                goto L;
            end;
            d := get();
        end
    end.
end.
```
9. EVALUATION

Algorithm of the rainflow method was shown as an implementation for the digital system. This system works the fastest when the size of the input buffer is equal to (or larger than) \((2^m-1)/3\), where \(m\) is the system parameter which is set at most the number of the output lines of the A/D converter. The input process itself consumes a few data area and a few computing time.

The rainflow process treats, as its data, only P/V differences i.e., integers 1, 2, \(\cdots\), or \(2^m-1\). The program rainflowCycleCount uses two data area, the arrays \(m\) and \(b\) (or a pushdown stack), of which sizes are each \(2^m-1\). Thus, for \(m = 8, 10, 12, 14\) or 16, the total size of the input buffer and the arrays \(b\) and \(m\) is \((2+1/3)(2^{16}-1) = 598, 2,387, 9,222, 28,227\) or \(152,915\), respectively. Because of using the input buffer, the rainflow process itself can accept theoretically a strain wave whose cycle time is larger than \(3t\), where \(t\) is the execution time of one loop-reap operation.

ACKNOWLEDGEMENTS

Dedicate this paper to the memory of the late T. Endo, the inventor of the rainflow method and my dear collaborator.

References