Rainflow Cycle Counting

`rs-rainflow` version 20160508.1803

Ralph Schleicher

This is the reference manual for the `rs-rainflow` library version 20160508.1803.

Copyright © 2015 Ralph Schleicher

Permission is granted to make and distribute verbatim copies of this manual, provided the copyright notice and this permission notice are preserved on all copies.

Table of Contents

1 Introduction

Rainflow cycle counting is a widely accepted method for transforming a sequence of signal values into a sequence of cycles. Each cycle is a tuple with three values. The first value is the signal amplitude $\hat{s}$, i.e. half the distance between the trough and peak signal value. The second value is the mean value $\bar{s}$, i.e. the arithmetic mean of the trough and peak signal value. The third value is the cycle count n, i.e. the number of alterations between the trough and peak signal value. The cycle count can be expressed as the number of full cycles or the number of half cycles.

If s_1 is the trough signal value and s_2 is the peak signal value, the signal amplitude $\hat{s}$ and mean value $\bar{s}$ are defined as follows:

$$\hat{s} = \frac{s_2 - s_1}{2}$$
$$\bar{s} = \frac{s_1 + s_2}{2}$$

Likewise, the trough signal value s_1 and peak signal value s_2 can be calculated from the signal amplitude $\hat{s}$ and mean value $\bar{s}$ via the equations

$$s_1 = \bar{s} - \hat{s}$$
$$s_2 = \bar{s} + \hat{s}$$

Cycle counting is mainly used in fatigue analysis. A cumulative damage model, e.g. Miner's rule, is applied on the cycle counting sequence to assess a part's fatigue life with the help of material S-N curves. Beside that, cycle counting is also useful to derive fatigue, duty cycle, or endurance spectra itself.

2 API Reference

The `rs-rainflow` library contains functions to perform rainflow cycle counting. All symbols described in this chapter are defined in the header file `rs-rainflow.h`. The implementation has several features:

- The procedure is re-entrant[1], that means you can call it multiple times in a row until all input data is processed.
- Support for different signal data types. You can switch the signal data type between consecutive invocations.
- Support for alternative memory managers.
- Cycles can be cached or shifted (consumed) according to your needs.
- Cycles can be sorted and merged according to your needs.

N.b.: The `rs-rainflow` library always counts half cycles. You have to divide the cycle count by two if you want to know the number of full cycles.

2.1 Instantiation

You have to create a rainflow cycle counting object before you can start counting cycles.

`rs_rainflow_t` [Data Type]

The data type of a rainflow cycle counting object.

This is an opaque data type. You only deal with pointers to rainflow cycle counting objects.

rs_rainflow_t * `rs_rainflow_new` (*void*) [Function]

Create a rainflow cycle counting object.

Return value is a pointer to a new rainflow cycle counting object. In case of an error, a null pointer is returned and `errno` is set to describe the error.

void `rs_rainflow_delete` (*rs_rainflow_t* ***obj***) [Function]

Delete a rainflow cycle counting object.

- Argument *obj* is a pointer to a rainflow cycle counting object. It is no error if argument *obj* is a null pointer.

Deleting a rainflow cycle counting object means to unconditionally return any allocated memory back to the system including the object itself. After that, all references to the rainflow cycle counting object are void.

The `rs_rainflow_new` function utilizes the standard memory management functions of the C library, i.e. `malloc`, `realloc`, and `free`. There is another rainflow cycle counting object creation function where you can specify alternative memory management functions.

rs_rainflow_t * `rs_rainflow_alloc` (*void* *(***malloc***) (*size_t*), *void* *(***realloc***) (*void* *, *size_t*), *void* (***free***) (*void* *)) [Function]

Create a rainflow cycle counting object using an alternative memory manager.

[1] But not thread safe.

- First argument *malloc* is a function to allocate a block of memory. The semantic of this function is the same as of the `malloc` function. It is guaranteed that the argument to the *malloc* function is greater than zero.
- Second argument *realloc* is a function to resize a block of memory allocated by the *malloc* function. The semantic of this function is the same as of the `realloc` function. It is guaranteed that the first argument to the *realloc* function is not a null pointer and that the second argument is greater than zero.
- Third argument *free* is a function to free a block of memory allocated by the *malloc* function. The semantic of this function is the same as of the `free` function. It is guaranteed that the argument to the *free* function is not a null pointer. If argument *free* is a null pointer, it is assumed that unused memory allocated via the *malloc* function is collected by the memory manager.

Return value is a pointer to a new rainflow cycle counting object. In case of an error, a null pointer is returned and `errno` is set to describe the error. The following error conditions are defined for this function:

`EINVAL` Argument *malloc* or *realloc* is a null pointer.

You usually call `rs_rainflow_delete` when you are done with a rainflow cycle counting object. Then you call `rs_rainflow_new` or `rs_rainflow_alloc` if you need another rainflow cycle counting object. The following convenience function has the same effect except that the reference to the rainflow cycle counting object remains the same.

int `rs_rainflow_reset` (*rs_rainflow_t* ***obj***) [Function]

Reset a rainflow cycle counting object.

- Argument *obj* is a pointer to a rainflow cycle counting object.

The effect of this function is like calling `rs_rainflow_delete` followed by a call to `rs_rainflow_new` except that the rainflow cycle counting object itself and the associated memory management functions remain the same.

Return value is zero on success. In case of an error, the return value is -1 and `errno` is set to describe the error. The following error conditions are defined for this function:

`EINVAL` Argument *obj* is a null pointer.

2.2 Execution

The `rs_rainflow` function is the core procedure for counting cycles. The result of the rainflow cycle counting method is a cycle counting sequence. The state of a rainflow cycle counting object controls the behavior of the `rs_rainflow` function. The default behavior is as follows.

- The signal history is expected to be an array of double precision floating-point numbers whose elements can be accessed sequentially.

  See function [`rs_rainflow_set_signal_type`], page 6 for how to change the array element type. See function [`rs_rainflow_set_read_signals`], page 7 for how to install a user-defined signal history access function.

- The cycle counting sequence is cached by the rainflow cycle counting object.

  See function [`rs_rainflow_set_shift_cycle`], page 7 for how to install a user-defined function which will be called when a cycle can be added to the cycle counting sequence.
- The length of the cycle counting sequence is derived from the number of elements in the signal history.

  See function [`rs_rainflow_set_length`], page 8 for how to change the memory allocation strategy for the cycle counting sequence.
- Similar consecutive cycles are merged by adding the individual cycle counts. This optimization reduces the length of the cycle counting sequence without loosing any information.

  See function [`rs_rainflow_set_merge_cycles`], page 8 for how to change this behavior.

You have to call the `rs_rainflow` function to perform rainflow cycle counting. After that, you can call the `rs_rainflow_cycles` function to determine the current length of the cycle counting sequence. Then you can shift (consume) cycles by calling the `rs_rainflow_shift` function. After all signal values have been feed to the `rs_rainflow` function, call the `rs_rainflow_finish` function to terminate cycle counting.

int `rs_rainflow` (*rs_rainflow_t* ***obj**, *void* ***sig**, *size_t* ***sig_len**, *int* ***finish**) [Function]

Perform rainflow cycle counting.

- First argument *obj* is a pointer to a rainflow cycle counting object.
- Second argument *sig* is a pointer to the signal history.
- Third argument *sig_len* is the number of elements in the signal history. A value of '`(size_t) -1`' means that the length of the signal history is undetermined, i.e. the signal history has infinite length.
- Fourth argument *finish* is a flag whether or not to finish rainflow cycle counting. A value of zero or `RS_RAINFLOW_CONTINUE` means to continue cycle counting. A non-zero value or `RS_RAINFLOW_FINISH` means to terminate cycle counting.

Return value is zero on success. In case of an error, the return value is -1 and `errno` is set to describe the error. The following error conditions are defined for this function:

`EINVAL` One of the following is true.

- Argument *obj* is a null pointer.
- Argument *sig* is a null pointer and argument *sig_len* is greater than zero and no user-defined signal history access function is installed.
- Cycle counting is finished.

Non-system error conditions are indicated via the following non-zero return values:

`RS_RAINFLOW_ERROR_STACK_OVERFLOW`
: The stack size exceeds system limits.

`RS_RAINFLOW_ERROR_CYCLE_OVERFLOW`
: The number of cached cycles exceeds system limits.

size_t `rs_rainflow_cycles` (*rs_rainflow_t* *`obj`) [Function]

Return the number of shiftable cycles.

- Argument *obj* is a pointer to a rainflow cycle counting object.

Return value is the number of shiftable cycles. This can be zero. In case of an error, the return value is '`(size_t) -1`' and `errno` is set to describe the error. The following error conditions are defined for this function:

`EINVAL` Argument *obj* is a null pointer.

int `rs_rainflow_shift` (*rs_rainflow_t* *`obj`, *void* *`buffer`, *size_t* `count`) [Function]

Shift (consume) the oldest cycles.

- First argument *obj* is a pointer to a rainflow cycle counting object.
- Second argument *buffer* is a pointer to a buffer where the cycles shall be stored. If argument *buffer* is a null pointer, cycles are shifted but not stored.
- Third argument *count* is the number of cycles to be shifted.

The caller is responsible for providing a large enough buffer.

Return value is zero on success. In case of an error, the return value is -1 and `errno` is set to describe the error. The following error conditions are defined for this function:

`EINVAL` Argument *obj* is a null pointer.

`EAGAIN` There are not enough cycles available.

Rainflow cycle counting ends when the `rs_rainflow` function is called with a non-zero fourth argument. There is a convenience function which does this explicitly:

int `rs_rainflow_finish` (*rs_rainflow_t* *`obj`) [Function]

Finish rainflow cycle counting.

- Argument *obj* is a pointer to a rainflow cycle counting object.

Return value is zero on success. In case of an error, the return value is -1 and `errno` is set to describe the error. The following error conditions are defined for this function:

`EINVAL` One of the following is true.

- Argument *obj* is a null pointer.
- Cycle counting is already finished.

Non-system error conditions are indicated via the following non-zero return values:

`RS_RAINFLOW_ERROR_CYCLE_OVERFLOW`
The number of cached cycles exceeds system limits.

Calling this function is equal to '`rs_rainflow (obj, NULL, 0, RS_RAINFLOW_FINISH)`'.

When rainflow cycle counting is finished, you can still call the `rs_rainflow_cycles` and `rs_rainflow_shift` functions to consume the remaining cycles. But there is an alternative method to access the cycle counting sequence:

void * **rs_rainflow_capture** (*rs_rainflow_t* ***obj***) [Function]

Return the cycle counting sequence.

- Argument *obj* is a pointer to a rainflow cycle counting object.

Return value is a pointer to the cycle counting sequence, i.e. a block of memory. The memory block is allocated with the configured memory allocation function and the caller is responsible for freeing the memory block with the appropriate procedure. When this function succeeds, any call to the `rs_rainflow_cycles` function will return zero. Thus, you have to determine the number of cycles *before* calling `rs_rainflow_capture`.

In case of an error, the return value is a null pointer and `errno` is set to describe the error. The following error conditions are defined for this function:

`EINVAL` Argument *obj* is a null pointer.

`EBUSY` Cycle counting is not finished.

2.3 Customization

The default values can be restored with the following sequence of statements.

```
rs_rainflow_set_length (obj, 0, 0);
rs_rainflow_set_signal_type (obj, RS_RAINFLOW_TYPE_DOUBLE);
rs_rainflow_set_shift_cycle (obj, NULL, NULL);
rs_rainflow_set_merge_cycles (obj, 1);
```

The individual functions are documented below.

int **rs_rainflow_set_signal_type** (*rs_rainflow_t* ***obj***, *int* ***type***) [Function]

Customize the array element type.

The default is a double precision floating-point number.

- First argument *obj* is a pointer to a rainflow cycle counting object.
- Second argument *type* is the array element type. Value should be one of the predefined array element types (see Section 2.5.1 [Array Element Types], page 9, for a complete list).

When you call this function, it is expected that the signal history is an array with elements of the specified type and that the array elements can be accessed sequentially.

You can specify the predefined array element type `RS_RAINFLOW_TYPE_UNKNOWN` to clear any assumption about how signal values are stored in the signal history. See function [`rs_rainflow_set_read_signals`], page 7 for how to install a user-defined signal history access function.

Return value is zero on success. In case of an error, the return value is -1 and `errno` is set to describe the error. The following error conditions are defined for this function:

`EINVAL` One of the following is true.

- Argument *obj* is a null pointer.
- Argument *type* is not one of the predefined array element types.

int **rs_rainflow_set_read_signals** (*rs_rainflow_t* ***obj**, *size_t* (***fun**) (*void **, *double **, *size_t*), *size_t* **incr**) [Function]

Customize the signal history access function.

- First argument *obj* is a pointer to a rainflow cycle counting object.
- Second argument *fun* is the address of a function with three arguments.
  - First argument is a pointer to the signal history. See function [rs_rainflow], page 4 for more details.
  - Second argument is a pointer to a signal value buffer.
  - Third argument is the number of signal values to be copied.

  Return value is the actual number of signal values copied. A value of zero means that the end of the signal history is reached.

  It is guaranteed that the signal value buffer can store the requested number of signal values.
- Third argument *incr* is the signal history address increment. A value of zero means to not increment the pointer to the signal history after copying signal values. Otherwise, the pointer to the signal history is incremented *incr* times the number of signal values copied.

When you call this function, you are responsible for converting signal values from the signal history to double precision floating-point numbers. This function replaces the built-in signal history access function installed by the `rs_rainflow_set_signal_type` function.

Return value is zero on success. In case of an error, the return value is -1 and `errno` is set to describe the error. The following error conditions are defined for this function:

`EINVAL` One of the following is true.

- Argument *obj* is a null pointer.
- Argument *fun* is a null pointer and argument *incr* is greater than zero.

int **rs_rainflow_set_shift_cycle** (*rs_rainflow_t* ***obj**, *void* (***fun**) (*void **, *double*, *double*, *double*), *void* ***arg**) [Function]

Customize the cycle shift function.

Default is to cache shifted cycles.

- First argument *obj* is a pointer to a rainflow cycle counting object.
- Second argument *fun* is the address of a function with four arguments.
  - First argument is the value of the *arg* argument.
  - Second argument is the signal amplitude of the shifted cycle.
  - Third argument is the signal mean value of the shifted cycle.
  - Fourth argument is the cycle count of the shifted cycle.
- Third argument *arg* is the first argument of the cycle shift function *fun*.

The which will be called when a cycle can be added to the cycle counting sequence

Return value is zero on success. In case of an error, the return value is -1 and `errno` is set to describe the error. The following error conditions are defined for this function:

`EINVAL` One of the following is true.

- Argument *obj* is a null pointer.
- Argument *fun* is a null pointer and argument *arg* is not a null pointer.

int **rs_rainflow_set_length** (*rs_rainflow_t* ***obj**, *size_t* **len**, *size_t* **add**) [Function]

Provide hints for memory allocation.

The default is to infer internal buffer sizes from the number of elements in the signal history.

- First argument *obj* is a pointer to a rainflow cycle counting object.
- Second argument *len* is the initial number of elements. A value of zero means to infer this number from the signal length.
- Third argument *add* is the number of elements to be added iff a buffer has to grow. A value of zero means to infer this number from the value of argument *len*.

Return value is zero on success. In case of an error, the return value is -1 and `errno` is set to describe the error. The following error conditions are defined for this function:

`EINVAL` Argument *obj* is a null pointer.

`EBUSY` Cycle counting has already started.

int **rs_rainflow_set_merge_cycles** (*rs_rainflow_t* ***obj**, *int* **merge**) [Function]

Define whether or not to merge similar consecutive cycles.

Cycles are similar if the signal amplitude and mean value are equal. If cycle merging is enabled, similar consecutive cycles are merged by adding the individual cycle counts. Cycle merging is enabled by default.

- First argument *obj* is a pointer to a rainflow cycle counting object.
- If second argument *merge* is non-zero, enable merging of similar consecutive cycles.

Return value is zero on success. In case of an error, the return value is -1 and `errno` is set to describe the error. The following error conditions are defined for this function:

`EINVAL` Argument *obj* is a null pointer.

`EBUSY` Cycle counting has already started.

2.4 Sorting and Merging

The functions in this section sort and compare *cycles*. Each cycle is an array of three double precision floating-point numbers. The first element of a cycle is the signal amplitude, the second element is the signal mean value, and the third element is the number of half cycles.

2.4.1 High-Level Procedures

int **rs_rainflow_sort** (*rs_rainflow_t* ***obj**, *int* (***compare**) (*void const* *, *void const* *)) [Function]

Sort the cached cycles.

Since sorting destroys the order of the cycle counting sequence, unconditionally merge similar cycles, too. See function [rs_rainflow_set_merge_cycles], page 8 for more details.

- First argument *obj* is a pointer to a rainflow cycle counting object.
- Second argument *compare* is a comparison function. Default is rs_rainflow_compare_descending. See Section 2.4.3 [Comparison Functions], page 9, for more predefined comparison functions.

Return value is zero on success. In case of an error, the return value is -1 and errno is set to describe the error. The following error conditions are defined for this function:

EINVAL Argument *obj* is a null pointer.

2.4.2 Low-Level Procedures

int rs_rainflow_sort_cycles (*void* ***buffer***, *size_t* ***count***, *int* (****compare***) (*void const **, *void const **)) [Function]

int rs_rainflow_merge_cycles (*void* ***buffer***, *size_t* ****count***, *int* (****compare***) (*void const **, *void const **)) [Function]

2.4.3 Comparison Functions

int rs_rainflow_compare_descending (*void const* ****left***, *void const* ****right***) [Function]

Compare two cycles in descending order.

Cycles are first compared by the signal amplitude. If the signal amplitude is equal, then the cycles are compared by the mean value.

- First argument *left* is the address of a cycle.
- Second argument *right* is the address of a cycle.

int rs_rainflow_compare_ascending (*void const* ****left***, *void const* ****right***) [Function]

Compare two cycles in ascending order.

Cycles are first compared by the signal amplitude. If the signal amplitude is equal, then the cycles are compared by the mean value.

- First argument *left* is the address of a cycle.
- Second argument *right* is the address of a cycle.

2.5 Enumerated Constants

2.5.1 Array Element Types

int RS_RAINFLOW_TYPE_UNKNOWN [Constant]
int RS_RAINFLOW_TYPE_DOUBLE [Constant]
int RS_RAINFLOW_TYPE_FLOAT [Constant]
int RS_RAINFLOW_TYPE_CHAR [Constant]
int RS_RAINFLOW_TYPE_UCHAR [Constant]

int `RS_RAINFLOW_TYPE_SHORT` [Constant]
int `RS_RAINFLOW_TYPE_USHORT` [Constant]
int `RS_RAINFLOW_TYPE_INT` [Constant]
int `RS_RAINFLOW_TYPE_UINT` [Constant]
int `RS_RAINFLOW_TYPE_LONG` [Constant]
int `RS_RAINFLOW_TYPE_ULONG` [Constant]
int `RS_RAINFLOW_TYPE_INT8_T` [Constant]
int `RS_RAINFLOW_TYPE_UINT8_T` [Constant]
int `RS_RAINFLOW_TYPE_INT16_T` [Constant]
int `RS_RAINFLOW_TYPE_UINT16_T` [Constant]
int `RS_RAINFLOW_TYPE_INT32_T` [Constant]
int `RS_RAINFLOW_TYPE_UINT32_T` [Constant]
int `RS_RAINFLOW_TYPE_INT64_T` [Constant]
int `RS_RAINFLOW_TYPE_UINT64_T` [Constant]

The predefined array element types. See function [`rs_rainflow_set_signal_type`], page 6. The array element type `RS_RAINFLOW_TYPE_CHAR` specifies a signed character, that is `signed char` in C.

2.5.2 Extended Error Codes

int `RS_RAINFLOW_ERROR_STACK_OVERFLOW` [Constant]

The stack size exceeds system limits.

int `RS_RAINFLOW_ERROR_CYCLE_OVERFLOW` [Constant]

The number of cached cycles exceeds system limits.

3 Examples

The source code archive contains several examples for how to use the `rs-rainflow` library.

3.1 Basic Example

The first example shows a naive usage of the `rs-rainflow` library.

```
int
main (void)
{
  rs_rainflow_t *obj;
  size_t buf_len;
  double *buf;

  /* Maximum number of signal values to be read.  */
  buf_len = 1000;

  /* Signal value buffer.  */
  buf = calloc (buf_len, sizeof (double));
  if (buf == NULL)
    abort ();

  /* Create rainflow cycle counting object.  */
  obj = rs_rainflow_new ();
  if (obj == NULL)
    abort ();

  /* Process signal values.  */
  while (1)
    {
      size_t count;

      count = read_from_stream (stdin, buf, buf_len);
      if (count == 0)
        break;

      if (rs_rainflow (obj, buf, count, RS_RAINFLOW_CONTINUE) != 0)
        abort ();
    }

  if (rs_rainflow_finish (obj) != 0)
    abort ();

  /* Print cycle counting sequence.  */
  print_cycles (stdout, obj);

  /* Destroy object.  */
  rs_rainflow_delete (obj);

  return 0;
}
```

3.2 Using Call-back Functions

This program does the same as the previous example, but much more memory efficient.

```
int
main (void)
{
  rs_rainflow_t *obj;

  /* Create rainflow cycle counting object.  */
  obj = rs_rainflow_new ();
  if (obj == NULL)
    abort ();

  /* Install call-back functions.

     Copy signal values from the signal history (a stream) to the signal
     buffer of the rainflow cycle counting object.  Do not increment the
     signal history pointer.  */
  rs_rainflow_set_read_signals (obj, (void *) read_from_stream, 0);

  /* Print cycle counting sequence to 'stdout'.  */
  rs_rainflow_set_shift_cycle (obj, (void *) print_cycle, stdout);

  /* Perform rainflow cycle counting.

     Read signal values from 'stdin' until end of file.  */
  if (rs_rainflow (obj, stdin, (size_t) -1, RS_RAINFLOW_FINISH) != 0)
    abort ();

  /* Destroy rainflow cycle counting object.  */
  rs_rainflow_delete (obj);

  return 0;
}
```

3.3 Reservoir Cycle Counting

This example shows how to implement reservoir cycle counting on top of rainflow cycle counting.

Reservoir counting creates the same result as rainflow counting iff the signal history starts and ends with the absolute signal maximum. Otherwise, reservoir counting is slightly more conservative than rainflow counting. Another property of reservoir counting is that the resulting cycle counting sequence only contains full cycles. However, you have to know the full signal history in advance so that you can find the global maximum.

```
void
reservoir (rs_rainflow_t *obj, double *sig_buf, size_t sig_len)
{
  size_t j, k;

  /* Locate global maximum of the signal history.  */
  k = 0;

  for (j = 1; j < sig_len; ++j)
    {
      if (sig_buf[j] > sig_buf[k])
        k = j;
    }

  /* Rearrange the cycle counting history so that it starts and ends
     with the global maximum.  */
  if (rs_rainflow (obj, sig_buf + k, sig_len - k, RS_RAINFLOW_CONTINUE) != 0)
    abort ();

  if (rs_rainflow (obj, sig_buf, k + 1, RS_RAINFLOW_FINISH) != 0)
    abort ();
}
```

3.4 Auxiliary Procedures

Here are three auxiliary procedures for the examples. The first function reads a number of floating-point numbers from a stream.

```
size_t
read_from_stream (FILE *stream, double *buffer, size_t count)
{
  size_t n;

  for (n = 0; count > 0; --count, ++buffer, ++n)
    {
      if (fscanf (stream, "%lf", buffer) != 1)
        break;
    }

  return n;
}
```

The next function prints a single cycle to a stream.

```
void
print_cycle (FILE *stream, double const *cycle)
{
  double ampl, mean, count;

  ampl = cycle[0];
  mean = cycle[1];
  count = cycle[2];

  /* Print cycle count as the number of full cycles.  */
  fprintf (stream, "%.5G;%.5G;%.5G\n", ampl, mean, count / 2.0);
}
```

The last function prints the whole cycle counting sequence of a `rs-rainflow` object.

```
void
print_cycles (FILE *stream, rs_rainflow_t *obj)
{
  double cycle[3];
  size_t n;

  for (n = rs_rainflow_cycles (obj); n > 0; --n)
    {
      /* Consume oldest cycle.  */
      rs_rainflow_shift (obj, cycle, 1);
      print_cycle (stream, cycle);
    }
}
```

Symbol Index

Concept Index

References

[1] ASTM E1049-85: *Standard Practices for Cycle Counting in Fatigue Analysis*. ASTM International, `http://www.astm.org`.