Rainflow Cycle Counting

`rs-rainflow` version 20180218.2240

Ralph Schleicher

This is the reference manual for the `rs-rainflow` library version 20180218.2240.

Table of Contents

1 Introduction

Rainflow cycle counting is a widely accepted method for transforming a sequence of signal values into an equivalent 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:

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

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 User's Guide

The `rs-rainflow` library contains functions to perform rainflow cycle counting. 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, i.e. the signal history, is processed.
- Support for different signal data types. You can switch the signal data type between consecutive invocations.
- Intermediate values and holds are automatically removed from the signal history.
- Support for alternative memory managers.
- Support for different cycle representations. You can choose between amplitude/mean, range/mean, and from/to cycle representation. You can also enable signed cycles for the amplitude/mean and range/mean cycle representations so that you have no loss of information compared to the from/to cycle representation.
- Support for signal labels so that you can analyse which signal values, i.e. load cases, build the critical cycles, e.g. those with the highest damages.
- 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. Calling the `rs_rainflow_new` function is the standard procedure to do so.

A rainflow cycle counting object has the opaque data type `rs_rainflow_t`. Therefore, you can only work with pointers to rainflow cycle counting objects. For example,

```
rs_rainflow_t *obj = rs_rainflow_new ();
```

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

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 `rs_rainflow_reset` convenience function has the same effect except that the reference to the rainflow cycle counting object remains the same.

2.2 Execution

The `rs_rainflow` function is the core procedure for counting cycles. The result of rainflow cycle counting is a cycle counting sequence. A cycle counting sequence is an array of double precision floating-point numbers where each row represents a cycle. The first two elements

[1] But not thread safe.

of a cycle are used for the cycle representation and the third element contains the cycle count.

You can call the `rs_rainflow_cycles` function to determine the current length of the cycle counting sequence. If cycles are available, then you can shift (consume) cycles by calling the `rs_rainflow_shift` function.

Rainflow cycle counting ends when the `rs_rainflow` function is called with a non-zero fourth argument. You can also call the `rs_rainflow_finish` convenience function to terminate cycle counting.

When rainflow cycle counting is finished, you can still call the `rs_rainflow_cycles` and `rs_rainflow_shift` functions to consume the remaining cycles.

An alternative method to access the cycle counting sequence is to call the `rs_rainflow_capture` function. Calling this function makes you the owner of the cycle counting sequence. Therefore, this is only possible if rainflow cycle counting is finished.

2.3 Customisation

The state of a rainflow cycle counting object controls the behaviour of the `rs_rainflow` function. The default behaviour is as follows.

- The signal history is expected to be an array of double precision floating-point numbers whose elements can be accessed sequentially.
- Signal labels are disabled.
- Cycles are represented by amplitude and mean value.
- Similar consecutive cycles are merged by adding the individual cycle counts.
- The cycle counting sequence is cached by the rainflow cycle counting object.

The rest of this section shows you how to adjust these settings to suite your needs.

2.3.1 Signal History

You can provide the signal history in two ways; as an array of numbers, or via a user-defined call-back function.

2.3.1.1 Array of Numbers

You can pass an array of numbers, i.e. a pointer to the first array element, as the second argument to the `rs_rainflow` function. The data type of the array elements is defined by the `rs_rainflow_set_signal_type` function. The data type conversion from the array element type to the internal format is performed by the `rs_rainflow` function. All numeric C data types are supported that way.

Please note that you can switch the array element type between consecutive invocations of the `rs_rainflow` function.

2.3.1.2 User-defined Call-back Function

If your signal history is not an array of numbers, then you can write a user-defined call-back function and install it with the `rs_rainflow_set_read_signals` function. See Section 4.2 [Using Call-back Functions], page 21, for an example.

2.3.2 Signal Labels

Signal labels can be used to assign an identifier to a signal value. This information is traced and recorded in the cycle counting sequence so that you can analyse which signal values were used to build a cycle. Signals labels are enabled or disabled by calling the `rs_rainflow_set_signal_label` function.

When signal labels are enabled, each cycle is a tuple with five elements. The first three elements have the usual meaning, i.e. cycle representation and cycle count, the fourth element is the signal label of the *from* signal value, and the fifth element is the signal label of the *to* signal value. Although a cycle is defined as an array of double precision floating-point numbers, any value that can be stored in the eight bytes of a `double` can be used as a signal label.

There are two types of signal labels; implicit signal labels and explicit signal labels.

2.3.2.1 Implicit Signal Labels

Implicit signal labels are automatically assigned to a signal value by the `rs_rainflow` function. The signal label is an integer which is incremented after each signal value. Thus, the signal label is like a linear index into the signal history. The start index, i.e. the signal label of the first signal value, can be set via the `rs_rainflow_set_signal_index` function. The default start index is zero.

Implicit signal labels only work with the built-in data types. If you install a user-defined signal history access function, then you have to use explicit signal labels.

2.3.2.2 Explicit Signal Labels

Explicit signal labels have to be assigned in a user-defined signal history access function (see [User-defined Call-back Function], page 3). That means the call-back function has to copy the signal value and the signal label. For example, suppose the elements of a signal history are defined as follows:

```
struct sig
  {
    /* Signal value.  */
    float value;

    /* Signal label.  */
    char *label;
  };
```

With that the user-defined call-back function for reading elements from the signal history could look like this:

```
size_t
read_signals (struct sig const *sig, double *buffer, size_t count)
{
  size_t c;

  for (c = count; c > 0; --c, ++sig)
    {
      /* Copy signal value.  */
      *buffer++ = sig->value;
```

```
        /* Copy signal label.  */
        memcpy (buffer, &sig->label, sizeof (char *));
        ++buffer;
      }

    return count - c;
  }
```

To play save, you should check at the beginning of your program if '`sizeof (char *)`' is not greater than '`sizeof (double)`'. It is no problem if the inverse is true since argument *buffer* is initialised with zeros. See the `rs_rainflow_set_read_signals` function for more details about the calling conventions.

2.3.3 Cycle Representation

A cycle is a tuple with three or five elements. The first and second element is the cycle representation. The third element is the cycle count, i.e. the number of half cycles. If signal labels are enabled, then the fourth and fifth element are the from/to signal labels. In C, a cycle is an array of three or five double precision floating-point numbers.

The `rs-rainflow` library supports three different cycle representations. You can choose between amplitude/mean, range/mean, and from/to cycle representation. The cycle representation can be changed via the `rs_rainflow_set_cycle_style` function but this has to be done before you start counting cycles.

2.3.3.1 Amplitude/Mean Cycle Representation

Amplitude/mean cycle representation is the default cycle representation of the `rs-rainflow` library. Thus, a cycle has the form $\{s_a, s_m, n, t_1, t_2\}$ where s_a is the signal amplitude, s_m is the signal mean, n is the cycle count, and t_1 and t_2 are the optional from/to signal labels.

The signal amplitude is unsigned by default. That means you can calculate the peak and trough signal value but you don't know the direction of the cycle[2]. You can enable signed cycle representation by calling the `rs_rainflow_set_cycle_sign` function. With that a positive signal amplitude means that the *from* signal value is less than the *to* signal value and a negative signal amplitude means that the *from* signal value is greater than the *to* signal value.

2.3.3.2 Range/Mean Cycle Representation

Range/mean cycle representation is like amplitude/mean cycle representation except that the signal amplitude is replaced by the signal range, i.e. two times the signal amplitude. Thus, a cycle has the form $\{s_r, s_m, n, \dots\}$ where s_r is the signal range and s_m is the signal mean.

2.3.3.3 From/To Cycle Representation

With from/to cycle representation a cycle has the form $\{s_1, s_2, n, \dots\}$ where s_1 and s_2 are the extrema values of the cycle in chronological order.

[2] It would be possible with the help of signal labels since signal labels are always saved in chronological order of the signal values

2.3.4 Merging Cycles

Similar consecutive cycles are merged by adding the individual cycle counts. This optimisation reduces the length of the cycle counting sequence without loosing any information. Cycles are similar if the signal values and the optional signal labels are equal and in the correct chronological order.

For example, if a half cycle from s_1 to s_2 is directly followed by a half cycle from s_2 to s_1, then this is equal to a full cycle from s_1 to s_2. If another full cycle from s_1 to s_2 follows, then this is equal to two full cycles from s_1 to s_2.

You can enable or disable this feature via the `rs_rainflow_set_merge_cycles` function. Merging cycles is enabled by default.

2.3.5 Consuming Cycles

By default the cycle counting sequence is cached by the rainflow cycle counting object. The length of the cycle counting sequence is estimated from the number of elements in the signal history. See function `rs_rainflow_set_length` for how to change the memory allocation strategy for the cycle counting sequence.

You can call the `rs_rainflow_cycles` function to determine the current length of the cycle counting sequence. If cycles are available, then you can shift (consume) cycles by calling the `rs_rainflow_shift` function.

This explicit process can be automated by installing a user-defined call-back function via the `rs_rainflow_set_shift_cycle` function. This call-back function is invoked by the `rs_rainflow` function whenever a cycle can be added to the cycle counting sequence. The benefit of this method is that the cached cycle counting sequence does not grow no matter how long the signal history is. See Section 4.2 [Using Call-back Functions], page 21, for an example.

2.4 Sorting Cycles

After rainflow cycle counting has finished, you can sort the cycles of the cached cycle counting sequence with the `rs_rainflow_sort` function. You have to write a user-defined cycle comparison function for sorting cycles. First you need a function to compare two floating-point numbers, like this one:

```
/* Compare the number a against b.  If a is considered greater than b, the return value is
   a positive number.  If a is considered less than b, the return value is a negative number.
   If the two numbers are equal, the return value is zero.   */
int
fcmp (double a, double b)
{
  return (a > b) - (a < b);
}
```

Now we can use the `fcmp` function to compare two cycles.

```
/* Compare the cycle a against b.  */
int
compare_cycles (double const *a, double const *b)
{
  int diff;

  diff = fcmp (a[0], b[0]);
  if (diff != 0)
    return diff;

  return fcmp (a[1], b[1]);
}
```

The `compare_cycles` function can be used to sort cycles in either ascending or descending order. For the later, simply exchange the arguments *a* and *b*. With GNU C, you can also use the expression '`fcmp (a[0], b[0]) ? : fcmp (a[1], b[1])`'. Whether or not signal labels are considered by the cycle comparison function is your choice.

2.5 Rainflow Matrix

A rainflow matrix is also a kind of sorting but the two elements of the cycle representation are always sorted in strictly monotonic increasing order. The cycle count of similar cycles is summed up. Signal labels are always ignored. The name rainflow matrix comes from the fact that the cycle representation can be considered as the row and column indices into a two-dimensional array and the cycle count is the corresponding matrix element.

You can create and destroy a rainflow matrix by calling the `rs_rainflow_matrix_new` function and `rs_rainflow_matrix_delete` respectively.

Cycles are added to the rainflow matrix by calling the `rs_rainflow_matrix_add` or `rs_rainflow_matrix_add3` function. Matrix elements, i.e. the cycle counts, are queried via the `rs_rainflow_matrix_get` or `rs_rainflow_matrix_get2` function.

You can query the range of a rainflow matrix dimension, i.e. the smallest and largest value of the corresponding element of the cycle representation, with the `rs_rainflow_matrix_limits` function.

The `rs_rainflow_matrix_non_zero` function returns the number of non-zero elements in a rainflow matrix. You can apply a function on each non-zero element by calling the `rs_rainflow_matrix_map` function.

Cycles are usually binned before they are added to a rainflow matrix. The `rs_rainflow_round_amplitude_mean`, `rs_rainflow_round_range_mean`, and `rs_rainflow_round_from_to` functions do this conservatively, i.e. the peak signal value is rounded up (toward positive infinity) and the trough signal value is rounded down (toward negative infinity).

Here is a simple cycle shift function (see Section 2.3.5 [Consuming Cycles], page 6) that adds the cycle to a rainflow matrix.

```
void
shift_cycle (rs_rainflow_matrix_t *mat, double const *cycle)
{
  double tem[3];

  memcpy (tem, cycle, 3 * sizeof (double));
  /* Signal values are binned to a multiple or 20 MPa.  Thus, the signal
     amplitude and signal mean are both binned to 10 MPa.  */
  rs_rainflow_round_amplitude_mean (tem, 20.0);
  rs_rainflow_matrix_add (mat, tem);
}
```

3 API Reference

All symbols described in this chapter are defined in the header file `rs-rainflow.h`.

3.1 Data Types

`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_matrix_t` [Data Type]

The data type of a rainflow matrix object.

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

3.2 Functions

*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.

*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.

- 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.

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.

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.

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.

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)`'.

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*, *double *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.

*double ** `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.

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_signal_type` (*rs_rainflow_t* **`obj`*, *int* *`type`*) [Function]

Customise 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 3.3.1 [Array Element Types], page 19, 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` 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]

Customise 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` 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 const **), *void* *`arg`) [Function]

Customise 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 two arguments.
  - First argument is the value of the *arg* argument.
  - Second argument is a pointer to the cycle.
- Third argument *arg* is the first argument of the cycle shift function *fun*.

The *fun* function 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_signal_label** (*rs_rainflow_t *obj*, *int label*) [Function]

Define whether or not to enable signal labels.

Signal labels are either implicit or explicit. If no user-defined signal history access function is defined, implicit signal labels are in effect. Implicit signal labels start with the number defined by the `rs_rainflow_set_signal_index` function and increment by one for each new signal value. If a user-defined signal history access function is defined and signal labels are enabled, the signal labels have to be provided by the user-defined signal history access function.

- First argument *obj* is a pointer to a rainflow cycle counting object.
- If second argument *label* is non-zero, enable signal labels 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.

int **rs_rainflow_set_signal_index** (*rs_rainflow_t *obj*, *double index*) [Function]

To be written.

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.

int **rs_rainflow_set_cycle_style** (*rs_rainflow_t *obj*, *int style*) [Function]

Change the cycle representation.

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 *style* is an invalid cycle representation.

`EBUSY` Cycle counting has already started.

int **rs_rainflow_set_cycle_sign** (*rs_rainflow_t *obj*, *int flag*) [Function]

Enable or disable signed cycle representation.

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.

`EBUSY` Cycle counting has already started.

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` 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 [Comparison Functions], page 15, 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.

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

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

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.

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.

rs_rainflow_matrix_t * **`rs_rainflow_matrix_new`** (*void*) [Function]

Create a rainflow matrix object.

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

void `rs_rainflow_matrix_delete` (*rs_rainflow_matrix_t* **obj*) [Function]

Delete a rainflow matrix object.

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

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

int `rs_rainflow_matrix_add` (*rs_rainflow_matrix_t* **obj*, *double const* **cycle*) [Function]

Add the cycle count of a cycle to the rainflow matrix.

- First argument *obj* is a pointer to a rainflow matrix object.
- Second argument *cycle* is a pointer to a cycle.

The cycle representation of *cycle*, i.e. the first and second element, should be discretised according to the desired bin edges of the rainflow matrix. Third element of *cycle* is the cycle count.

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* or *cycle* is a null pointer.

`EDOM` The first or second element of *cycle* is not-a-number.

`ERANGE` The third element of *cycle* is not-a-number or less than zero.

int `rs_rainflow_matrix_add3` (*rs_rainflow_matrix_t* **obj*, *double* **`first`**, *double* **`second`**, *double* **`count`**) [Function]

Add the cycle count of a cycle to the rainflow matrix.

- First argument *obj* is a pointer to a rainflow matrix object.
- Second argument *first* is the index of the first dimension of the rainflow matrix. This is usually the first element of the cycle representation.
- Third argument *second* is the index of the second dimension of the rainflow matrix. This is usually the second element of the cycle representation.
- Fourth argument *count* is the cycle count.

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.

`EDOM` Argument *first* or *second* is not-a-number.

`ERANGE` Argument *count* is not-a-number or less than zero.

double `rs_rainflow_matrix_get` (*rs_rainflow_matrix_t* **obj*, *double const* **cycle*) [Function]

Get the cycle count of a cycle from the rainflow matrix.

- First argument *obj* is a pointer to a rainflow matrix object.
- Second argument *cycle* is a pointer to a cycle.

The cycle representation of *cycle*, i.e. the first and second element, should be discretised according to the desired bin edges of the rainflow matrix.

Return value is the cycle count. In case of an error, the return value is not-a-number and `errno` is set to describe the error. The following error conditions are defined for this function:

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

`EDOM` The first or second element of *cycle* is not-a-number.

double `rs_rainflow_matrix_get2` (*rs_rainflow_matrix_t* *`obj`, *double* `first`, *double* `second`) [Function]

Get the cycle count of a cycle from the rainflow matrix.

- First argument *obj* is a pointer to a rainflow matrix object.
- Second argument *first* is the index of the first dimension of the rainflow matrix. This is usually the first element of the cycle representation.
- Third argument *second* is the index of the second dimension of the rainflow matrix. This is usually the second element of the cycle representation.

Return value is the cycle count. In case of an error, the return value is not-a-number and `errno` is set to describe the error. The following error conditions are defined for this function:

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

`EDOM` Argument *first* or *second* is not-a-number.

int `rs_rainflow_matrix_limits` (*rs_rainflow_matrix_t* *`obj`, *int* `dim`, *double* *`min`, *double* *`max`) [Function]

Get the lower and upper bound of a rainflow matrix dimension.

- First argument *obj* is a pointer to a rainflow matrix object.
- Second argument *dim* is the rainflow matrix dimension. Value is either zero, i.e. the first dimension, or 1, i.e. the second dimension.
- Third argument *min* is the address where to store the value of the lower bound. No value will be stored if *min* is a null pointer.
- Fourth argument *max* is the address where to store the value of the upper bound. No value will be stored if *max* is a null pointer.

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.

`ERANGE` The rainflow matrix is empty.

size_t `rs_rainflow_matrix_non_zero` (*rs_rainflow_matrix_t* *`obj`) [Function]

Return the number of non-zero elements of a rainflow matrix.

- Argument *obj* is a pointer to a rainflow matrix 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` Argument *obj* is a null pointer.

void `rs_rainflow_matrix_map` (*rs_rainflow_matrix_t* *`obj`, *void* (*`fun`) (*void* *, *double const* *), *void* *`arg`) [Function]

Apply a function on any non-zero element of a rainflow matrix.

- First argument *obj* is a pointer to a rainflow matrix object.
- Second argument *fun* is the address of a function with two arguments.
  - First argument is the value of the *arg* argument.
  - Second argument is a pointer to the cycle.
- Third argument *arg* is the first argument of the function *fun*.

In case of an error, `errno` is set to describe the error. The following error conditions are defined for this function:

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

void `rs_rainflow_round_amplitude_mean` (*double* *`cycle`, *double* `scale`) [Function]
void `rs_rainflow_round_range_mean` (*double* *`cycle`, *double* `scale`) [Function]
void `rs_rainflow_round_from_to` (*double* *`cycle`, *double* `scale`) [Function]

Round the signal values of *cycle* to a multiple of *scale*.

- First argument *cycle* is the address of a cycle.
- Second argument *scale* is the rounding scale factor. Value has to be a positive number.

The cycle representation of *cycle* is modified in-place.

`rs_rainflow_round_amplitude_mean`
Round an amplitude/mean cycle representation.

`rs_rainflow_round_range_mean`
Round a range/mean cycle representation.

`rs_rainflow_round_from_to`
Round a from/to cycle representation.

double `rs_rainflow_round_up` (*double* `number`, *double* `scale`) [Function]
double `rs_rainflow_round_down` (*double* `number`, *double* `scale`) [Function]
double `rs_rainflow_round_zero` (*double* `number`, *double* `scale`) [Function]
double `rs_rainflow_round_inf` (*double* `number`, *double* `scale`) [Function]

Round *number* to a multiple of *scale*.

- First argument *number* is a number.
- Second argument *scale* is the rounding scale factor. Value has to be a positive number.

Return value is the rounded number.

`rs_rainflow_round_up`
Round towards plus infinity.

`rs_rainflow_round_down`
Round towards minus infinity.

`rs_rainflow_round_zero`
: Round towards zero (away from infinity).

`rs_rainflow_round_inf`
: Round away from zero (towards infinity).

3.3 Enumerated Constants

3.3.1 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`. The array element type `RS_RAINFLOW_TYPE_CHAR` specifies a signed character, that is `signed char` in C.

3.3.2 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.

4 Examples

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

4.1 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;
}
```

4.2 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;
}
```

4.3 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 signal 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 ();
}
```

4.4 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)
{
  /* Print cycle count as the number of full cycles.  */
  fprintf (stream, "%.5G;%.5G;%.5G\n", cycle[0], cycle[1], cycle[2] / 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[5];
  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`.