Resampling Event-Based power data to fixed timebase

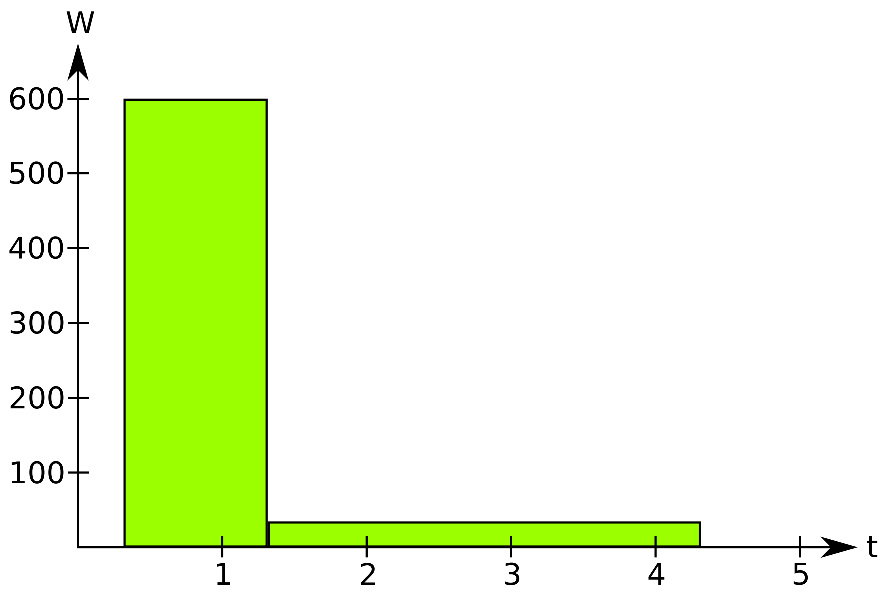
Mark Rages, Quarq / SRAM

October 2012

Crank-based power meter data is traditionally sampled per revolution. This uneven sampling rate produces smoother power data by rejecting the cadence*2 signal from the pulses of power from each leg's stroke, and the cadence*1 signal from L-R stroke imbalances.

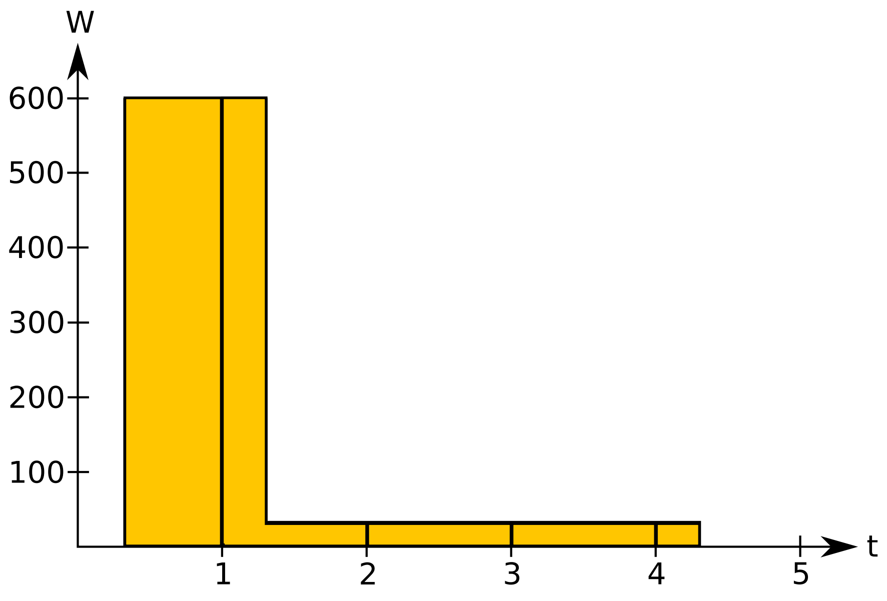
For analysis, the data could be used in the unevenly-sampled domain, but most analysis algorithms prefer to work on evenly-sampled data. So the problem is to convert the unevenly-sampled data into evenly-sampled data.

One important criterium of a resampling algorithm is how well the energy content in the file is preserved. For example, consider the following two crank revolutions. The first is at 60 RPM (one second) and averages 600W, and the second is at 20 RPM (three seconds) and averages 66.7 W.

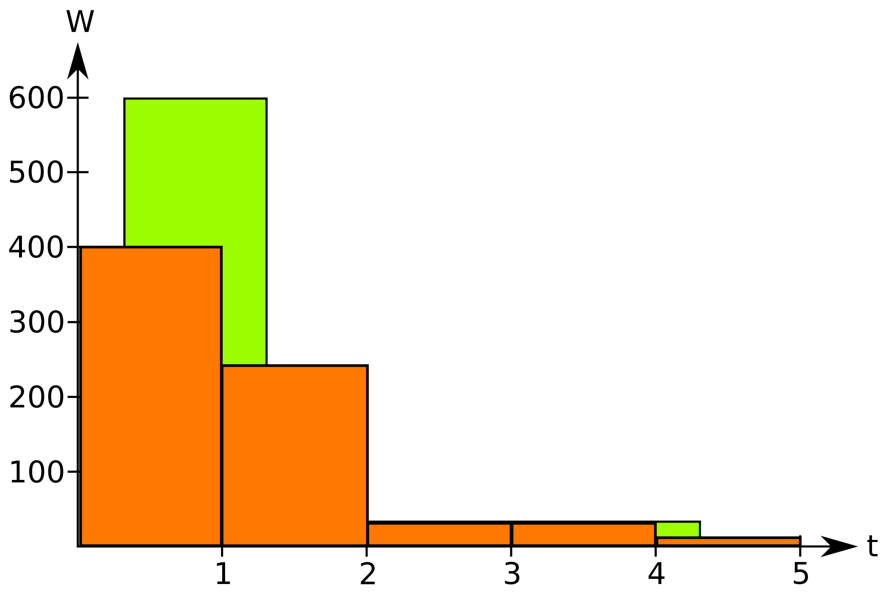


The energy of the first crank revolution is 600 W * 1 s, or 600 J. The energy of the second revolution is 66.7 W * 3 s, or 200 J. The objective for resampling is to have the resampled signal maintain the same energy content.

Let us consider resampling to a one-second timebase. Each second can be considered a bin that we distribute some of the incoming energy to. Graphically, the result will look like this:



Now the energy in the first bin is 400 J, the second bin is 200 J + 44.4 J, the energy in the third and fourth bins is 66.7 J each, and the energy in the last bin is 22.2J. Because we are resampling to a one-second timebase, each energy is numerically equivalent to the average watts for that bin. So the wattage looks like this:



The energy is redistributed in time a little bit, but computing the average will work correctly.

Here is an ANSI-C implementation of the algorithm:

```
typedef struct {
  float duration; // of the power sample in seconds
  float power; // average, in watts
} resample_t;

const float resample_period=1.0; // seconds

float *resample(resample_t input[], int count, int *outcount) {
  int in_i,out_i;
  float duration = 0.0;

  for (in_i=0; in_i<count; in_i++) {
    duration += input[in_i].duration;
  }

  int output_bins = ceilf(duration / resample_period);
  float* ret = (float *)calloc(output_bins,sizeof(float));

  float out_starttime=0.0;
  float in_starttime=0.0;

  for (in_i=0, out_i=0; (in_i<count) && (out_i<output_bins); ) {

    float in_endtime = in_starttime+input[in_i].duration;
    float out_endtime = out_starttime+resample_period;

    float overlap_start = max(in_starttime, out_starttime);
    float overlap_end = min(in_endtime, out_endtime);

    float overlap_duration = overlap_end - overlap_start;

    assert(overlap_duration >= 0);

    ret[out_i] += input[in_i].power * overlap_duration / resample_period;

    // move along appropriate pointers
    if (in_endtime < out_endtime) in_i++, in_starttime = in_endtime;
    else out_i++, out_starttime = out_endtime;
  }

  *outcount=output_bins;
  return ret;
}
```