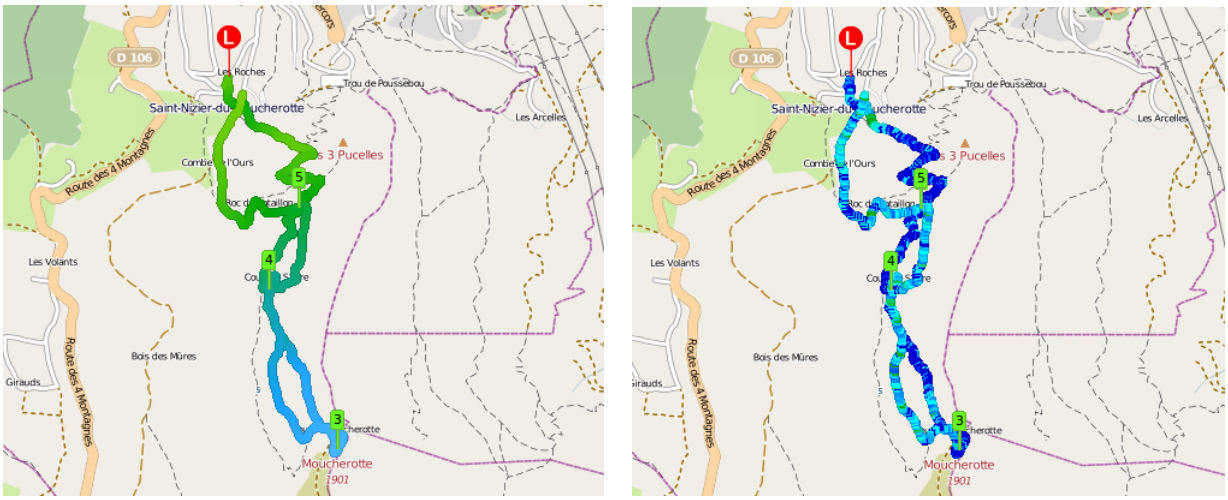
Smoothing out GPS and heart rate monitor data

August 7, 2011

GPS or heart rate data coming from GPS devices and heart rate monitors can be so fluctuating in time that their analysis may unfortunately be useless. For instance, it is sometimes possible to view data on a map as illustrated in Fig. 1 and, in this case, if the altitude data are sufficiently regular, the speed data is so rough that it is even almost impossible to see that going down is faster than going up (by a factor close to 2 in this case). It would then be interesting to get smoother data in order to facilitate their analysis.



(a) Altitude

(b) Speed

Figure 1: Altitude and speed data recorded during a hike and shown on a map.

In this note, a simple filtering procedure is presented and is then applied to GPS and heart rate data.

1 A simple filtering procedure

1.1 Forward filtering

Let us consider a function $u(t)$ that shows fluctuations at high frequencies that one would like to get rid of in order to get smoother data. The objective is thus to determine a function $y_{fw}(t)$ that is free of these high frequency fluctuations. A simple way to achieve this is to look for $y_{fw}(t)$ as a solution of the following differential equation:

$$\frac{dy_{fw}}{dt} = \frac{1}{\tau}\left(u(t) - y_{fw}(t)\right) \tag{1}$$

where τ is a characteristic time scale called the *relaxation time*.

This differential equation is a simple relaxation equation that tends to make y_{fw} get close to u with a characteristic time τ: if y_{fw} is larger than u (i.e., $y_{fw}(t) > u(t)$), then $(dy_{fw}/dt) < 0$, which makes y_{fw} decrease to get closer to u and if y_{fw} is smaller than u (i.e., $y_{fw}(t) < u(t)$), then $(dy_{fw}/dt) > 0$, which makes y_{fw} increase to get closer to u.

If the relaxation time τ is large, y_{fw} will get close to u very slowly and, at the limit where $\tau \to \infty$, $y_{fw}(t)$ will be constant and will not "see" the variations of u. If τ is small, y_{fw} will get close to u very rapidly and, at the limit where $\tau \to 0$, $y_{fw}(t)$ will be equal to $u(t)$ and will thus follow all the variations of u.

The objective is to discretize the differential equation (1). We then assume that the initial function $u(t)$ is discretized by N values noted u^n, $n \in [1; N]$. The differential equation (1) can be discretized as follows:

$$\frac{y_{fw}^{n+1} - y_{fw}^{n}}{t^{n+1} - t^{n}} = \frac{1}{\tau}\left(u^{n+1} - y_{fw}^{n+1}\right) \tag{2}$$

where $\psi^n = \psi(t^n)$ for any function ψ.

The fact that y_{fw}^{n+1} (and therefore u^{n+1}) is considered in the right-hand-side of this equation comes from stability reasons: if one chooses y_{fw}^{n} (and therefore u^n) instead, stability criteria on $(t^{n+1} - t^n)$ must be satisfied, whereas with y_{fw}^{n+1}, the numerical scheme is always stable, whatever the value of $(t^{n+1} - t^n)$. Since, for the application considered, the time discretization is imposed (by the frequency of acquisition of the devices), making this choice ensures that y_{fw}^{n+1} will never diverge.

One can then compute y_{fw}^{n+1} very easily:

$$y_{fw}^{n+1} = \frac{y_{fw}^{n} + a^{n+1/2}\, u^{n+1}}{1 + a^{n+1/2}} \tag{3}$$

where

$$a^{n+1/2} = \frac{t^{n+1} - t^{n}}{\tau}$$

Starting from the initial value

$$y_{fw}^{0} = u_{fw}^{0}$$

for instance, all the future values of y_{fw}^{n+1} can be computed using the relation (3).

The solution obtained for the particular case where $u(t)$ is a Heaviside function is shown in Fig. 2. This example shows that the function $y_{fw}(t)$ "goes" towards $u(t)$ after the discontinuity of this function with a finite relaxation time ($\tau = 2$ in this example).

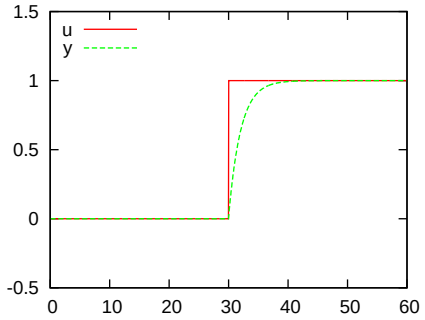


Figure 2: Example of solution of $y_{fw}(t)$ obtained by solving Eq. (3) where $u(t)$ is a Heaviside function; $\tau = 2$ in this example.

This figure also shows an important feature: $y(t)$ is "late" compared to $u(t)$, which makes regularization non-symmetric.

1.2 Backward filtering

Another way of filtering the high frequencies of $u(t)$ is to reverse the time in Eq. (1):

$$\frac{dy_{bw}}{dt} = \frac{y_{bw}(t) - u(t)}{\tau} \tag{4}$$

The discretization of this differential equation reads:

$$\frac{y_{bw}^{n+1} - y_{bw}^{n}}{t^{n+1} - t^{n}} = \frac{y_{bw}^{n} - u^{n}}{\tau}$$

y_{bw}^{n} (and therefor u^{n}) is considered in the right-hand-side of this equation because the time is reversed.

This equation is used to express y_{bw}^{n} as a function of y_{bw}^{n+1} (because time is reversed):

$$y_{bw}^{n} = \frac{y_{bw}^{n+1} + a^{n+1/2}\, u^{n}}{1 + a^{n+1/2}} \tag{5}$$

where

$$a^{n+1/2} = \frac{t^{n+1} - t^{n}}{\tau}$$

"Starting" from (or ending by) the value

$$y_{bw}^{N} = u^{N}$$

for instance, all the past values of y_{bw}^{n} can be computed using the relation (5).

The solution obtained for the particular case where $u(t)$ is a Heaviside function is shown in Fig. 3. This figure shows that the solution is not symmetric and is moved backwards compared to the function $u(t)$.

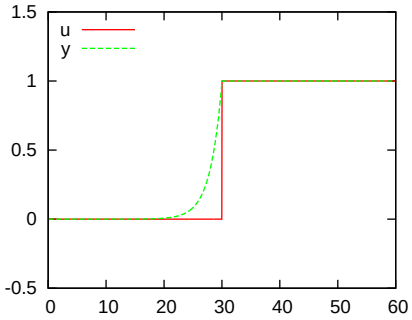


Figure 3: Example of solution of $y_{bw}(t)$ obtained by solving Eq. (5) where $u(t)$ is a Heaviside function; $\tau = 2$ in this example.

1.3 Centered filtering

In order to get a centered regularized function, one can take the average of the forward and backward solutions $y_{fw}(t)$ and $y_{bw}(t)$ respectively:

$$y^{n} = \frac{y_{fw}^{n} + y_{bw}^{n}}{2},\ \forall n \in [1; N] \tag{6}$$

The corresponding graph of the function $y(t)$ shown in Fig. 4 shows that the function $y(t)$ is indeed symmetric.

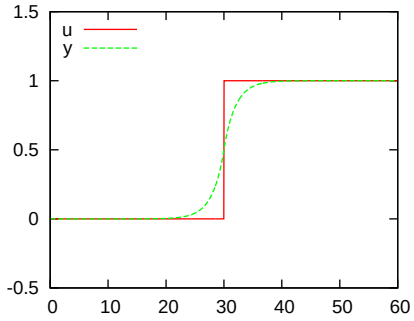


Figure 4: Example of solution of $y(t)$ obtained by the solution (6) where $u(t)$ is a Heaviside function; $\tau = 2$ in this example.

In order to see the influence of the relaxation time τ on the regularized function $y(t)$, this function is plotted for different values of τ in Fig. 5.

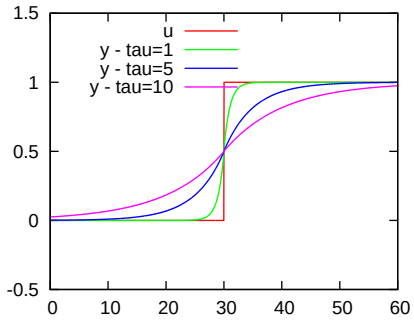


Figure 5: Example of solutions of $y(t)$ obtained by the solution (6) where $u(t)$ is a Heaviside function for different values of the relaxation time τ.

1.4 Keeping the start and end values

As visible in Fig. 5 for $\tau = 10$, the start and end values of the filtered data may not be conserved by the filtering. Sometimes, these values may be important to conserve, for instance to compute the elevation gain in the case of the altitude. A simple way to conserve the start and end values is to apply this simple transformation to the function $y(t)$:

$$y_{se}(t) = y(t) + (u_s - y_s) + \frac{(u_e - y_e) - (u_s - y_s)}{t_e - t_s}(t - t_s)$$

where the subscripts s and e denote the *start* and *end* values respectively.

It must be noticed that the same transformation (and approximation) could be used to **keep constant particular values such as those corresponding to tour markers** (in MyTourbook for instance).

1.5 Mathematical add-ons

1.5.1 Differential equation for $y(t)$

Adding Eqs (1) and (4), one gets

$$\frac{d\left(y_{fw}+y_{bw}\right)}{dt} = -\frac{\left(y_{fw}-y_{bw}\right)}{\tau} \tag{7}$$

Subtracting Eqs (1) and (4), one gets

$$\frac{d\left(y_{fw}-y_{bw}\right)}{dt} = \frac{2\,u-\left(y_{fw}+y_{bw}\right)}{\tau} \tag{8}$$

Differentiating Eq. (7) and using (8), one gets

$$\frac{d^2\left(y_{fw}+y_{bw}\right)}{dt^2} = \frac{\left(y_{fw}+y_{bw}\right)-2\,u}{\tau^2}$$

In other words, the filtered function $y(t)$ is solution of the following second order differential equation:

$$\frac{d^2y}{dt^2} = \frac{y-u}{\tau^2} \tag{9}$$

This equation could be solved directly, in particular to impose the boundary conditions on both ends, but it would require the inversion of a matrix, which would be more complex to code.

1.6 Derivative of the filtered function

It is straightforward to show that the derivative of the filtered function is equal to the filtered derivative. In other words, if the filtered function of u is denoted $\overline{u}$, then one has: $\overline{du/dt} = d\overline{u}/dt$.

1.6.1 Filter

The objective is to determine the filter that corresponds to the filtering procedure described previously.

It can be shown that the solution $y_H(t)$ in the case where $u(t)$ is a Heaviside function $H(t)$ is

$$y_H(t) = \frac{1}{2}\left[e^{t/\tau}\left(1-H(t)\right)-e^{-t/\tau}H(t)\right]+H(t)$$

Differentiating this expression, one gets

$$\frac{dy_H}{dt} = \frac{1}{2\,\tau}\left[e^{t/\tau}\left(1-H(t)\right)+e^{-t/\tau}H(t)\right]$$

and this function is nothing but the filter whose graph is given in Fig. 6.

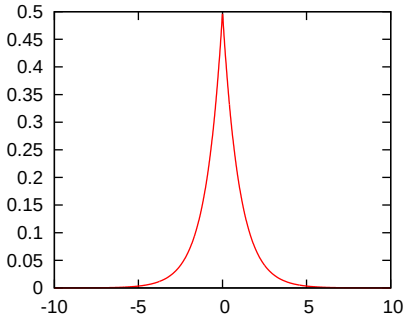


Figure 6: Graph of the filter ($\tau = 1$).

2 Application to GPS and heart rate data

2.1 Illustration of the smoothing effect on the vertical speed data

Data coming from outdoor computers can be very fluctuating in time, which can make their analysis very difficult. A typical case corresponds to the time evolution of vertical speed recorded by a GPS device during a hike for instance. Indeed, the GPS precision for the altitude is much lower than that for the longitude and latitude. This lack of precision is such that altitude data exhibit rather large fluctuations in time. The time variation of the vertical speed, which is nothing but the time variation of the altitude ($v_v(t) = dh/dt$), is therefore a rough function of time as illustrated in Fig. 7.

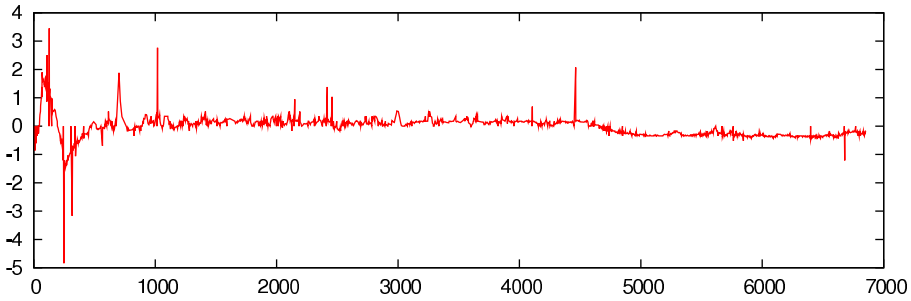


Figure 7: Example of the time variation of the vertical speed obtained from raw GPS data.

In order to regularize this function, we start by regularizing the altitude data using the procedure described in section 1. In the example considered here, this regularization is illustrated in Fig. 8.

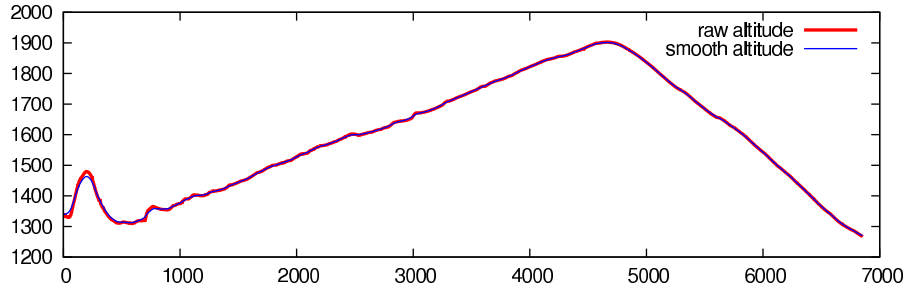


Figure 8: Example of the effect of the smoothing of the time variation of the altitude ($\tau = 30$ in this case).

Using these smoother altitude data, the time variation of the vertical speed is shown in Fig. 9[1].

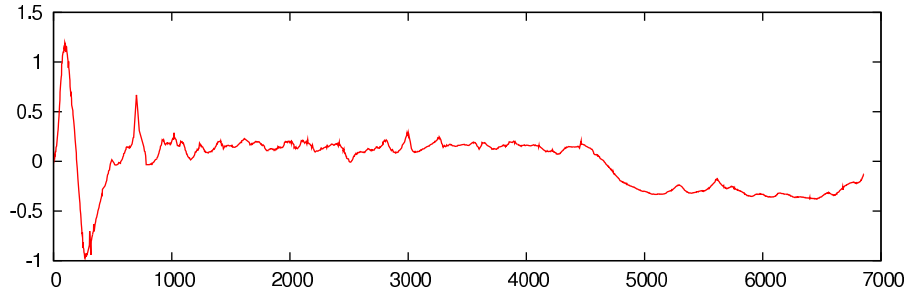


Figure 9: Example of the time variation of the vertical speed obtained after smoothing out GPS altitude data.

[1]It is worth noting that the graph of the vertical speed obtained directly by smoothing out the raw vertical speed is very close to this graph, which corresponds to the application of the property $\overline{dy/dt} = d\bar{y}/dt$.

These data are smoother than the inital ones (see Fig. 7), but they still exhibit high frequency fluctuations. Theoretically, the high frequencies of these fluctuations have the same characteristics as that of the initial function. Therefore, to get rid of these high frequencies, the smoothing should be similar to that used to smooth out the initial data, i.e., using the same value of the relaxation time τ. With this (second) smoothing, one gets the time variation shown in Fig. 10.

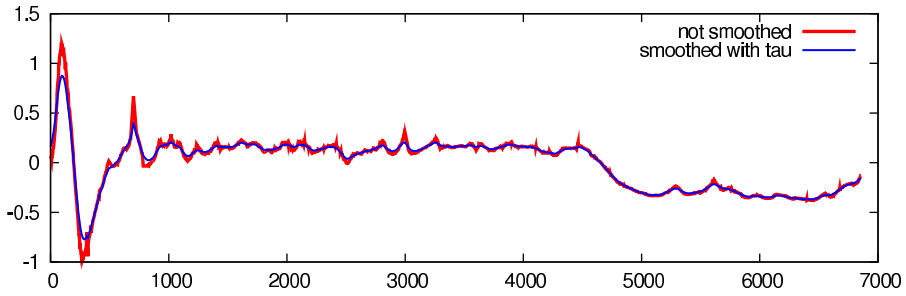


Figure 10: Example of the time variation of the vertical speed obtained after smoothing out altitude GPS data and then smoothing out the vertical speed with the relaxation time τ.

Thanks to these smoothed-out data, new data can be analyzed like the local terrain slope for instance. Fig. 11 shows that the raw data are almost useless because of their large fluctuations whereas the slope obtained after smoothing can be analyzed much more easily.

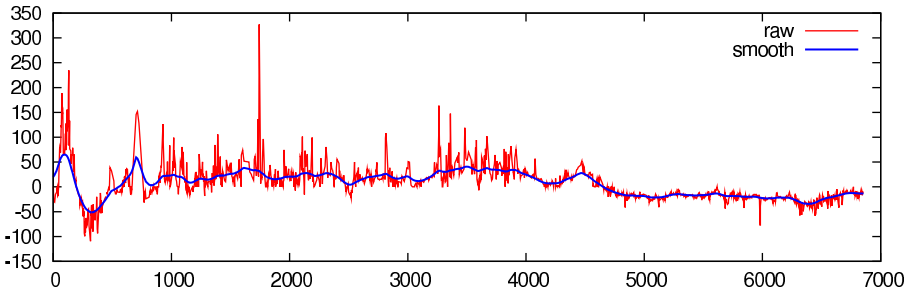


Figure 11: Example of the time variation of the terrain slope.

2.2 Illustration of the smoothing effect on heart rate data

The same smoothing procedure can be applied to heart rate data. The smoothing effect is shown in Fig. 12.

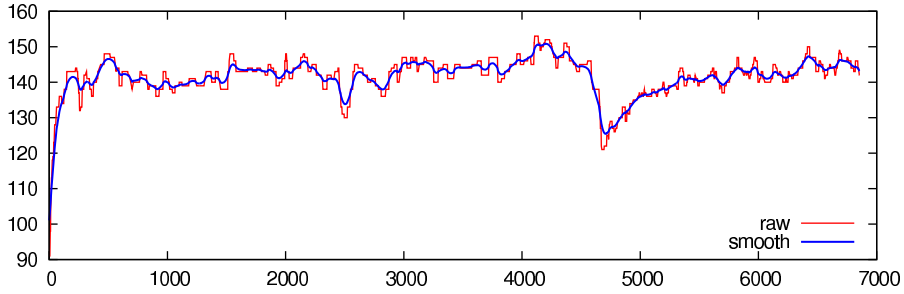
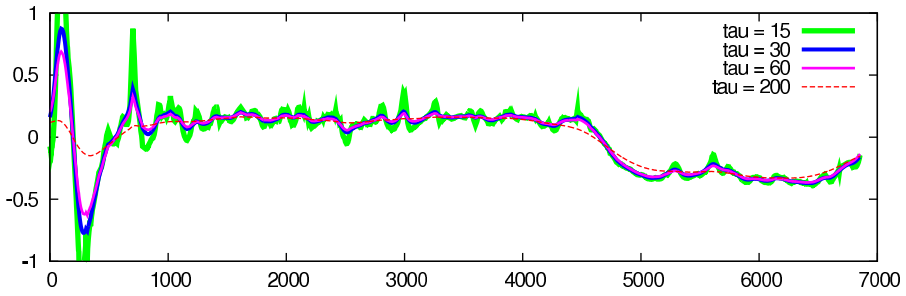


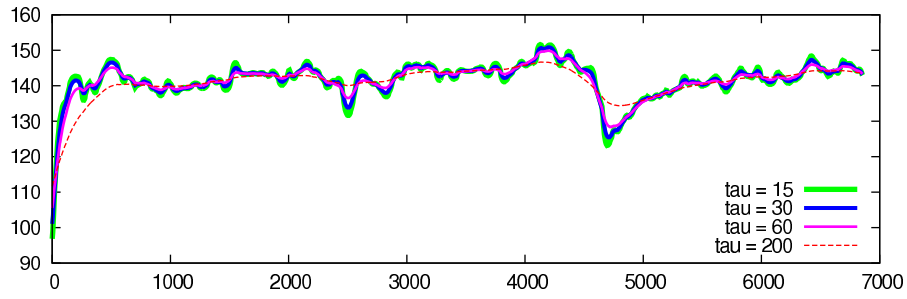
Figure 12: Example of the smoothing effect ($\tau = 30$) on the time variation of the heart rate.

2.3 Effect of the relaxation time on the smoothing

The effect of the value of the relaxation time τ used to smooth out the data is shown in Fig. 13. This figure shows that (i) the larger τ is the smoother the data are but that (ii) the local extrema are captured less accurately, which is not a surprise.



(a) Vertical speed

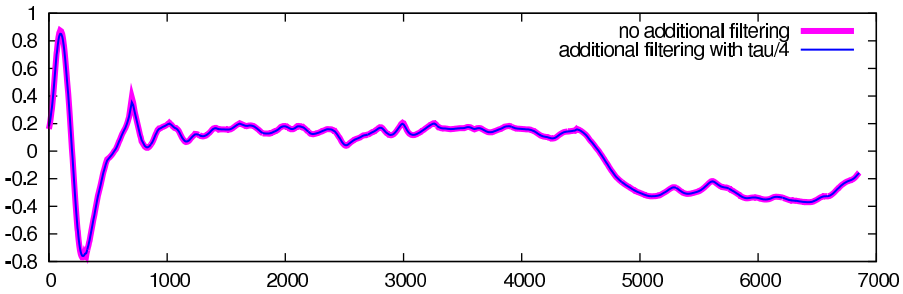


(b) Heart rate

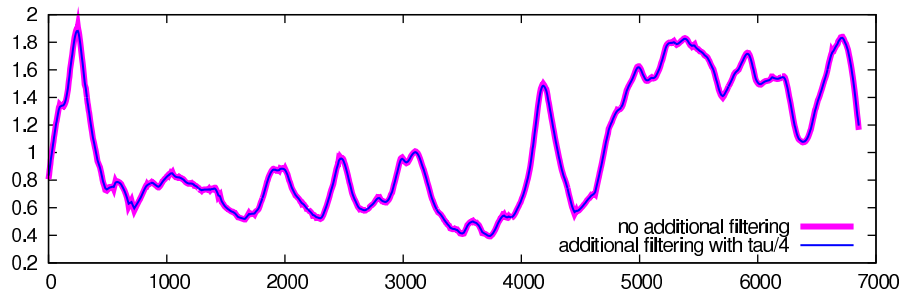
Figure 13: Effect of the value of the relaxation time τ on the smoothing of the time variation of the heart rate and the vertical speed.

2.4 Effect of an additional smoothing

It has already been mentioned that, despite the application of the filter with a relaxation time τ, high frequencies may still remain. To get rid of them, one may apply an additional filtering with a smaller relaxation time. Fig. 14 shows the effect of such an additional filtering on the vertical and horizontal speeds in the case where the relaxation time used for this additional filtering is $\tau/4$. This figure shows that in the case where high frequency fluctuations are barely existing (vertical speed in Fig. 14), this additional filtering has no visible effect whereas it does filter out high frequencies when they exist (horizontal speed in Fig. 14). Therefore, it can be recommended to apply this additional filtering (with $\tau/4$) systematically after applying the first one (with τ).



(a) Vertical speed



(b) Horizontal speed

Figure 14: Effect of an extra-smoothing with $\tau/4$.

3 Code

An coding example in C/C++ of the filtering procedure described previously is presented. The main parts are emphasized in color: magenta for the filtering, blue for the procedure to compute missing initial data and cyan for the computation of the speeds and slope.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define SIZE 1528
#define pi 3.14159265358979323846264338327795


////////////////////////////////////////////////
// Filters the high frequencies of field(time) //
////////////////////////////////////////////////

//=========================
// Main filtering function
//=========================
void smoothing1(double* time, double* field, double* field_sc, double tau)
{
  int i;
  double field_sf[SIZE], field_sb[SIZE];
  double dt;

// Forward smoothing
//------------------
  field_sf[0] = field[0];
  for (i=1; i<SIZE; i++)
    {
      dt = (time[i] - time[i-1]) / tau;
      field_sf[i] = ( field_sf[i-1] + dt * field[i] ) / (1. + dt);
    }
// Backward smoothing
//-------------------
  field_sb[SIZE-1] = field[SIZE-1];
  for (i=2; i<SIZE+1; i++)
    {
      dt = (time[SIZE-i+1] - time[SIZE-i]) / tau;
      field_sb[SIZE-i] = ( field_sb[SIZE-i+1] + dt * field[SIZE-i] ) / (1. + dt);
    }
// Centered smoothing
//-------------------
  for (i=0; i<SIZE; i++)
    {
      field_sc[i] = (field_sf[i] + field_sb[i])/2.;
    }
```

```
}

//================================================
// Filters twice:
//    1. a first time with relaxation time tau
//    2. a second time with relaxation time tau/4
//================================================
void smoothing(double* time, double* field, double* field_sc, double tau, int keep_start_end)
{
  int i;
  double field_sc1[SIZE];
  double Delta_start, Delta_end;

// First smoothing with tau
//-------------------------
  smoothing1(time, field, field_sc1, tau);

// Second smoothing with tau/4
//----------------------------
  smoothing1(time, field_sc1, field_sc, tau/4);

// Keep start and end values
//--------------------------
  if (keep_start_end == 1)
    {
      Delta_start = field[0] - field_sc[0];
      Delta_end = field[SIZE-1] - field_sc[SIZE-1];
      for (i=0; i<SIZE; i++)
        {
          field_sc[i] = field_sc[i] + Delta_start + (Delta_end-Delta_start) / (double)(SIZE-1) * (double)(i);
        }
    }

}


////////////////////////////////////////////////
// Computes the distance between 2 GPS points //
////////////////////////////////////////////////
double distance_gps(double lat1, double lat2, double lon1, double lon2)
{
  double earth_radius = 6366000.; // earth radius in meters
  double lat1_rad = lat1*pi/180.;
  double lat2_rad = lat2*pi/180.;
  double lon1_rad = lon1*pi/180.;
  double lon2_rad = lon2*pi/180.;

  return( earth_radius * 2. * asin( sqrt( (sin((lat1_rad-lat2_rad)/2.)) * (sin((lat1_rad-lat2_rad)/2.)) + cos(lat1_rad) * cos(lat2_rad) * (sin((lon1_rad-lo
}


main()
{
  int i;
  double time[SIZE];
  double latitude[SIZE], longitude[SIZE];
  double altitude[SIZE], altitude_sc[SIZE];
  double heart_rate[SIZE], heart_rate_sc[SIZE];
  double distance[SIZE], distance_sc[SIZE];
  double Vh_ini[SIZE], Vh[SIZE], Vh_sc[SIZE];
  double Vv_ini[SIZE], Vv[SIZE], Vv_sc[SIZE];
  double slope_sc[SIZE];
  double tau=30.;

  FILE *file;

// Initialization
//===============
  for(i=0; i<SIZE; i++)
    {
      time[i] = -1.;
      latitude[i] = -1.;
      longitude[i] = -1.;
      altitude[i] = -1.;
      heart_rate[i] = -1.;
    }

// Reading raw data in a text file
//================================
  file = fopen("example.txt", "r");
  for(i=0; i<SIZE; i++)
    {
      fscanf(file, "%lf\t%lf\t%lf\t%lf\t%lf\n", &time[i], &latitude[i], &longitude[i], &altitude[i], &heart_rate[i]);
    }
  fclose(file);


// Modify data so that no value is invalid (i.e. equal -1)
//========================================================
// We look for the first valid value (i.e. different from -1) and set the initial value to that first valid value
//-----------------------------------------------------------------------------------------------------------------
// Latitude
//.........
  if (latitude[0]==-1)
    {
      i=0;
      do
```

```
        i++;
      while (latitude[i]==-1);
      latitude[0]=latitude[i];
    }
// Longitude
//.........
  if (longitude[0]==-1)
    {
      i=0;
      do
        i++;
      while (longitude[i]==-1);
      longitude[0]=longitude[i];
    }
// Altitude
//.........
  if (altitude[0]==-1)
    {
      i=0;
      do
        i++;
      while (altitude[i]==-1);
      altitude[0]=altitude[i];
    }
// Heart Rate
//...........
  if (heart_rate[0]==-1)
    {
      i=0;
      do
        i++;
      while (heart_rate[i]==-1);
      heart_rate[0]=heart_rate[i];
    }

// If the value is invalid (i.e. equal -1), we set it to the previous value (in time)
//-----------------------------------------------------------------------------
  for (i=1; i<SIZE; i++)
    {
      if (latitude[i]==-1)
        latitude[i]=latitude[i-1];
      if (longitude[i]==-1)
        longitude[i]=longitude[i-1];
      if (altitude[i]==-1)
        altitude[i]=altitude[i-1];
      if (heart_rate[i]==-1)
        heart_rate[i]=heart_rate[i-1];
    }


// Compute the distance from latitude and longitude data
//======================================================
  distance[0] = 0.;
  for (i=1; i<SIZE; i++)
    {
      distance[i] = distance[i-1] + distance_gps(latitude[i], latitude[i-1], longitude[i], longitude[i-1]);
    }


// Compute the horizontal and vertical speeds from the raw distance and altitude data
//====================================================================================
 for (i=0; i<SIZE-1; i++)
    {
      if (time[i+1] == time[i])
        {
          if (i==0)
            {
              Vh_ini[i] = 0.;
              Vv_ini[i] = 0.;
            }
          else
            {
              Vh_ini[i] = Vh_ini[i-1];
              Vv_ini[i] = Vv_ini[i-1];
            }
        }
      else
        {
          Vh_ini[i] = (distance[i+1] - distance[i]) / (time[i+1] - time[i]);
          Vv_ini[i] = (altitude[i+1] - altitude[i]) / (time[i+1] - time[i]);
        }
    }
  Vh_ini[SIZE-1] = Vh_ini[SIZE-2];
  Vv_ini[SIZE-1] = Vv_ini[SIZE-2];


// Smooth out the time variations of the distance, the altitude and the heart rate
//================================================================================
  smoothing(time, distance, distance_sc, tau, 0);
  smoothing(time, altitude, altitude_sc, tau, 0);
  smoothing(time, heart_rate, heart_rate_sc, tau, 0);


// Compute the horizontal and vertical speeds from the smoothed distance and altitude
//====================================================================================
 for (i=0; i<SIZE-1; i++)
    {
      if (time[i+1] == time[i])
```

```
        {
          if (i==0)
            {
              Vh[i] = 0.;
              Vv[i] = 0.;
            }
          else
            {
              Vh[i] = Vh[i-1];
              Vv[i] = Vv[i-1];
            }
        }
      else
        {
          Vh[i] = (distance_sc[i+1] - distance_sc[i]) / (time[i+1] - time[i]);
          Vv[i] = (altitude_sc[i+1] - altitude_sc[i]) / (time[i+1] - time[i]);
        }
    }
  Vh[SIZE-1] = Vh[SIZE-2];
  Vv[SIZE-1] = Vv[SIZE-2];


// Smooth out the time variations of the horizontal and vertical speeds
//=====================================================================
  smoothing(time, Vh, Vh_sc, tau, 0);
  smoothing(time, Vv, Vv_sc, tau, 0);


// Compute the terrain slope
//==========================
  for (i=0; i<SIZE; i++)
    {
      slope_sc[i] = Vv_sc[i] / Vh_sc[i] *100.;
    }


  file = fopen("smooth_simple_ini.txt", "w");
  for(i=0; i<SIZE; i++)
    {
      fprintf(file, "%g\t%g\t%g\t%g\t%g\n", time[i], distance[i], altitude[i], Vh_ini[i], Vv_ini[i], heart_rate[i]);
    }
  fclose(file);

  file = fopen("smooth_simple_sc.txt", "w");
  for(i=0; i<SIZE; i++)
    {
      fprintf(file, "%g\t%g\t%g\t%g\t%g\t%g\n", time[i], distance_sc[i], altitude_sc[i], Vh_sc[i], Vv_sc[i], slope_sc[i], heart_rate_sc[i]);
    }
  fclose(file);
}
```